./ - ktap

- Makefile
- _vimrc_local.vim
- genmobi.sh
- include/
- ktapvm.mod.c
- runtime/
- test/
- userspace/
- vim/

Makefile - ktap

```
1
2  #
3  # Define NO_LIBELF if you do not want libelf dependency (e.g. cross-builds)
4  # (this will also disable resolve resolving symbols in DSO functionality)
5  #
6  # Define FFI if you want to compile ktap with FFI support. By default This
7  # toggle is off.
8  #
9  # Define amalg to enable amalgamation build, This compiles the ktapvm as
10  # one huge C file and allows GCC to generate faster and shorter code. Alas,
11  # this requires lots of memory during the build.
12  # Recommend to use amalgmation build as default.
13  amalg = 1
14
15  # Do not instrument the tracer itself:
16  ifdef CONFIG_FUNCTION_TRACER
17  ORIG_CFLAGS := $(KBUILD_CFLAGS)
18  KBUILD_CFLAGS = $(subst -pg,,$(ORIG_CFLAGS))
19  endif
20
21  all: mod ktap
22
23  INC = include
24  RUNTIME = runtime
25
26  FFIDIR = $(RUNTIME)/ffi
27  KTAP_LIBS = -lpthread
28
29  LIB_OBJS += $(RUNTIME)/lib_base.o $(RUNTIME)/lib_kdebug.o \
30          $(RUNTIME)/lib_timer.o $(RUNTIME)/lib_ansi.o \
31          $(RUNTIME)/lib_table.o $(RUNTIME)/lib_net.o
32
33  ifndef amalg
34  ifdef FFI
35  FFI_OBJS += $(FFIDIR)/ffi_call.o $(FFIDIR)/ffi_type.o $(FFIDIR)/ffi_symbol.o \
36      $(FFIDIR)/cdata.o $(FFIDIR)/ffi_util.o
37  RUNTIME_OBJS += $(FFI_OBJS)
38  LIB_OBJS += $(RUNTIME)/lib_ffi.o
39  endif
40  RUNTIME_OBJS += $(RUNTIME)/ktap.o $(RUNTIME)/kp_bcread.o $(RUNTIME)/kp_obj.o \
41          $(RUNTIME)/kp_str.o $(RUNTIME)/kp_mempool.o \
42          $(RUNTIME)/kp_tab.o $(RUNTIME)/kp_vm.o \
43          $(RUNTIME)/kp_transport.o $(RUNTIME)/kp_events.o $(LIB_OBJS)
44  else
45  RUNTIME_OBJS += $(RUNTIME)/amalg.o
46  endif
47
48  ifdef FFI
49  ifeq ($(KBUILD_MODULES), 1)
50  ifdef CONFIG_X86_64
51  # call_x86_64.o is compiled from call_x86_64.S
52  RUNTIME_OBJS += $(FFIDIR)/call_x86_64.o
53  else
54  $(error ktap FFI only supports x86_64 for now!)
55  endif
56  endif
57
58
59  ccflags-y    += -DCONFIG_KTAP_FFI
60  endif
61
62  obj-m        += ktapvm.o
63  ktapvm-y     := $(RUNTIME_OBJS)
64
65  KVERSION ?= $(shell uname -r)
66  KERNEL_SRC ?= /lib/modules/$(KVERSION)/build
67  PWD := $(shell pwd)
68  mod:
69      $(MAKE) -C $(KERNEL_SRC) M=$(PWD) modules
70
71  modules_install:
```

```
    $(MAKE) -C $(KERNEL_SRC) M=$(PWD) modules_install

KTAPC_CFLAGS = -Wall -O2


# try-cc
# Usage: option = $(call try-cc, source-to-build, cc-options, msg)
ifneq ($(V),1)
TRY_CC_OUTPUT= > /dev/null 2>&1
endif
TRY_CC_MSG=echo "    CHK $(3)" 1>&2;

try-cc = $(shell sh -c                              \
         'TMP="/tmp/.$$$$";                          \
          $(TRY_CC_MSG)                              \
          echo "$(1)" |                              \
          $(CC) -x c - $(2) -o "$$TMP" $(TRY_CC_OUTPUT) && echo y;    \
          rm -f "$$TMP"')


define SOURCE_LIBELF
#include <libelf.h>

int main(void)
{
        Elf *elf = elf_begin(0, ELF_C_READ, 0);
        return (long)elf;
}
 endef

 FLAGS_LIBELF = -lelf

 ifdef NO_LIBELF
     KTAPC_CFLAGS += -DNO_LIBELF
 else
 ifneq ($(call try-cc,$(SOURCE_LIBELF),$(FLAGS_LIBELF),libelf),y)
     $(warning No libelf found, disables symbol resolving, please install elfutils-libelf-devel/libelf-dev);
     NO_LIBELF := 1
     KTAPC_CFLAGS += -DNO_LIBELF
 else
     KTAP_LIBS += -lelf
 endif
 endif

 UDIR = userspace

 $(UDIR)/kp_main.o: $(UDIR)/kp_main.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/kp_lex.o: $(UDIR)/kp_lex.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/kp_parse.o: $(UDIR)/kp_parse.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/kp_bcwrite.o: $(UDIR)/kp_bcwrite.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/kp_reader.o: $(UDIR)/kp_reader.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/kp_util.o: $(UDIR)/kp_util.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/kp_parse_events.o: $(UDIR)/kp_parse_events.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 ifndef NO_LIBELF
 $(UDIR)/kp_symbol.o: $(UDIR)/kp_symbol.c KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 endif
 ifdef FFI
 KTAPC_CFLAGS += -DCONFIG_KTAP_FFI
 $(UDIR)/ffi_type.o: $(RUNTIME)/ffi/ffi_type.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/ffi/cparser.o: $(UDIR)/ffi/cparser.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 $(UDIR)/ffi/ctype.o: $(UDIR)/ffi/ctype.c $(INC)/* KTAP-CFLAGS
     $(QUIET_CC)$(CC) $(DEBUGINFO_FLAG) $(KTAPC_CFLAGS) -o $@ -c $<
 endif


 KTAPOBJS =
```

```
KTAPOBJS += $(UDIR)/kp_main.o
KTAPOBJS += $(UDIR)/kp_lex.o
KTAPOBJS += $(UDIR)/kp_parse.o
KTAPOBJS += $(UDIR)/kp_bcwrite.o
KTAPOBJS += $(UDIR)/kp_reader.o
KTAPOBJS += $(UDIR)/kp_util.o
KTAPOBJS += $(UDIR)/kp_parse_events.o
ifndef NO_LIBELF
KTAPOBJS += $(UDIR)/kp_symbol.o
endif
ifdef FFI
KTAPOBJS += $(UDIR)/ffi_type.o
KTAPOBJS += $(UDIR)/ffi/cparser.o
KTAPOBJS += $(UDIR)/ffi/ctype.o
endif

ktap: $(KTAPOBJS) KTAP-CFLAGS
    $(QUIET_LINK)$(CC) $(KTAPC_CFLAGS) -o $@ $(KTAPOBJS) $(KTAP_LIBS)

KMISC := /lib/modules/$(KVERSION)/ktapvm/

install: mod ktap
    make modules_install ktapvm.ko
    install -c ktap /usr/bin/
    mkdir -p ~/.vim/ftdetect
    mkdir -p ~/.vim/syntax
    cp vim/ftdetect/ktap.vim ~/.vim/ftdetect/
    cp vim/syntax/ktap.vim ~/.vim/syntax/

load:
    insmod ktapvm.ko

unload:
    rmmod ktapvm

reload:
    make unload; make load

test: FORCE
    #start testing
    prove -j4 -r test/

clean:
    $(MAKE) -C $(KERNEL_SRC) M=$(PWD) clean
    $(RM) ktap KTAP-CFLAGS


PHONY += FORCE
FORCE:

TRACK_FLAGS = KTAP
ifdef amalg
TRACK_FLAGS += AMALG
endif
ifdef FFI
TRACK_FLAGS += FFI
endif
ifdef NO_LIBELF
TRACK_FLAGS += NO_LIBELF
endif

KTAP-CFLAGS: FORCE
    @FLAGS='$(TRACK_FLAGS)'; \
    if test x"$$FLAGS" != x"`cat KTAP-CFLAGS 2>/dev/null`" ; then \
        echo "$$FLAGS" >KTAP-CFLAGS; \
    fi

#generate tags/etags/cscope index for editor.
define all_sources
        (find . -name '*.[ch]' -print)
endef

.PHONY: tags
tags:
    $(all_sources) | xargs ctags

```

```
224 .PHONY: etags
225 etags:
226     $(all_sources) | xargs etags
227
228 .PHONY: cscope
229 cscope:
230     $(all_sources) > cscope.files
231     cscope -k -b
```

One Level Up Top Level

_vimrc_local.vim - ktap

```
1 set noexpandtab
2 "filetype indent off
3 "filetype plugin off
4 "set nowrap autoindent nosmartindent nocindent indentexpr=
```

genmobi.sh - ktap

```
#!/bin/bash

name=`pwd|perl -e '$d=<>;$d=~s{.*/}{}g;$d=~s/2$//g;print $d'`
src2html='src2html.pl -l -x -c -n'

echo "Generating HTMLs for $name..."
$src2html -t 4 --exclude='*/.*' --exclude='.*' ./ $name
#exit

 echo "Generating .pdf file for $name..."
 rm -rf *.pdf
 ebook-convert html_out/index.html $name.pdf \
     --override-profile-size \
     --paper-size a4 \
     --pdf-default-font-size 12 \
     --pdf-mono-font-size 12 \
     --margin-left 10 --margin-right 10 \
     --margin-top 10 --margin-bottom 10 \
     --page-breaks-before='/'
 #exit

 echo "Generating .epub file for $name..."
 rm -rf *.epub
 ebook-convert html_out/index.html $name.epub \
     --no-default-epub-cover \
     --publisher agentzh \
     --output-profile ipad3 \
     --title $name \
     --language en --authors 'Jovi Zhangwei'

 echo "Generating .mobi file for $name..."
 rm -rf *.mobi
 ebook-convert html_out/index.html $name.mobi \
     --output-profile kindle_dx --no-inline-toc \
     --publisher agentzh \
     --title $name \
     --language en --authors 'Jovi Zhangwei'

 #cp -uv *.mobi ~/mobi/

```

include/ - ktap

- ktap_arch.h
- ktap_bc.h
- ktap_err.h
- ktap_errmsg.h
- ktap_ffi.h
- ktap_types.h

include/ktap_arch.h - ktap

Macros defined

- KP_BE
- KP_BE
- KP_BE
- KP_BE
- KP_ENDIAN_SELECT
- KP_ENDIAN_SELECT
- KP_ENDIAN_SELECT
- KP_ENDIAN_SELECT
- KP_LE
- KP_LE
- KP_LE
- KP_LE
- __KTAP_ARCH__

Source code

```
1  #ifndef __KTAP_ARCH__
2  #define __KTAP_ARCH__
3
4  #ifdef __KERNEL__
5  #include <linux/types.h>
6  #include <asm/byteorder.h>
7
8  #if defined(__LITTLE_ENDIAN)
9  #define KP_LE               1
10  #define KP_BE               0
11  #define KP_ENDIAN_SELECT(le, be)        le
12  #elif defined(__BIG_ENDIAN)
13  #define KP_LE               0
14  #define KP_BE               1
15  #define KP_ENDIAN_SELECT(le, be)        be
16  #endif
17
18  #else /* __KERNEL__ */
19
20  #if __BYTE_ORDER == __LITTLE_ENDIAN
21  #define KP_LE               1
22  #define KP_BE               0
23  #define KP_ENDIAN_SELECT(le, be)        le
24  #elif __BYTE_ORDER == __BIG_ENDIAN
25  #define KP_LE               0
26  #define KP_BE               1
27  #define KP_ENDIAN_SELECT(le, be)        be
28  #else
29  #error "could not determine byte order"
30  #endif
31
32  #endif
33  #endif
```

include/ktap_bc.h - ktap

Data types defined

- BCIns
- BCLine
- BCMode
- BCOp
- BCPos
- BCReg
- MMS

Functions defined

- bc_isret

Macros defined

- BCBIAS_J
- BCDEF
- BCDUMP_F_BE
- BCDUMP_F_FFI
- BCDUMP_F_KNOWN
- BCDUMP_F_STRIP
- BCDUMP_HEAD1
- BCDUMP_HEAD2
- BCDUMP_HEAD3
- BCDUMP_VERSION
- BCENUM
- BCENUM
- BCINS_ABC
- BCINS_AD
- BCINS_AJ
- BCMAX_A
- BCMAX_B
- BCMAX_C
- BCMAX_D

- BCMODE
- BCMODE_FF
- BCM___
- FF_next_N
- KP_STATIC_ASSERT
- NO_JMP
- NO_REG
- __KTAP_BC_H__
- bc_a
- bc_b
- bc_c
- bc_d
- bc_j
- bc_op
- bcmode_a
- bcmode_b
- bcmode_c
- bcmode_d
- bcmode_hasd
- bcmode_mm
- kp_assert
- setbc_a
- setbc_b
- setbc_byte
- setbc_c
- setbc_d
- setbc_j
- setbc_op

Source code

```
/*
 * Bytecode instruction format.
 *
 * Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 * Copyright (C) 2005-2014 Mike Pall.
 */

#ifndef __KTAP_BC_H__
#define __KTAP_BC_H__
```

```
10
11 #include "../include/ktap_arch.h"
12
13 /*TODO*/
14 #define KP_STATIC_ASSERT(cond)
15 #define kp_assert(cond)
16
17 /* Types for handling bytecodes. */
18 typedef uint32_t BCIns;  /* Bytecode instruction. */
19 typedef uint32_t BCPos;  /* Bytecode position. */
20 typedef uint32_t BCReg;  /* Bytecode register. */
21 typedef int32_t BCLine;  /* Bytecode line number. */
22
23 /*
24 * Bytecode instruction format, 32 bit wide, fields of 8 or 16 bit:
25 *
26 * +----+----+----+----+
27 * | B  | C  | A  | OP | Format ABC
28 * +----+----+----+----+
29 * |    D    | A  | OP | Format AD
30 * +--------------------
31 * MSB               LSB
32 *
33 * In-memory instructions are always stored in host byte order.
34 */
35
36 /* Operand ranges and related constants. */
37 #define BCMAX_A        0xff
38 #define BCMAX_B        0xff
39 #define BCMAX_C        0xff
40 #define BCMAX_D        0xffff
41 #define BCBIAS_J    0x8000
42 #define NO_REG        BCMAX_A
43 #define NO_JMP        (~(BCPos)0)
44
45 /* Macros to get instruction fields. */
46 #define bc_op(i)    ((BCOp)((i)&0xff))
47 #define bc_a(i)        ((BCReg)(((i)>>8)&0xff))
48 #define bc_b(i)        ((BCReg)((i)>>24))
49 #define bc_c(i)        ((BCReg)(((i)>>16)&0xff))
50 #define bc_d(i)        ((BCReg)((i)>>16))
51 #define bc_j(i)        ((ptrdiff_t)bc_d(i)-BCBIAS_J)
52
53 /* Macros to set instruction fields. */
54 #define setbc_byte(p, x, ofs) \
55     ((uint8_t *)(p))[KP_ENDIAN_SELECT(ofs, 3 - ofs)] = (uint8_t)(x)
56 #define setbc_op(p, x)    setbc_byte(p, (x), 0)
57 #define setbc_a(p, x)    setbc_byte(p, (x), 1)
58 #define setbc_b(p, x)    setbc_byte(p, (x), 3)
59 #define setbc_c(p, x)    setbc_byte(p, (x), 2)
60 #define setbc_d(p, x) \
61     ((uint16_t *)(p))[KP_ENDIAN_SELECT(1, 0)] = (uint16_t)(x)
62 #define setbc_j(p, x)    setbc_d(p, (BCPos)((int32_t)(x)+BCBIAS_J))
63
64 /* Macros to compose instructions. */
65 #define BCINS_ABC(o, a, b, c) \
66     (((BCIns)(o))|((BCIns)(a)<<8)|((BCIns)(b)<<24)|((BCIns)(c)<<16))
67 #define BCINS_AD(o, a, d) \
68     (((BCIns)(o))|((BCIns)(a)<<8)|((BCIns)(d)<<16))
69 #define BCINS_AJ(o, a, j)    BCINS_AD(o, a, (BCPos)((int32_t)(j)+BCBIAS_J))
70
71 /*
72 * Bytecode instruction definition. Order matters, see below.
73 *
74 * (name, filler, Amode, Bmode, Cmode or Dmode, metamethod)
75 *
76 * The opcode name suffixes specify the type for RB/RC or RD:
77 * V = variable slot
78 * S = string const
79 * N = number const
80 * P = primitive type (~itype)
81 * B = unsigned byte literal
82 * M = multiple args/results
83 */
84 #define BCDEF(_) \
85     /* Comparison ops. ORDER OPR. */ \
```

```
  _(ISLT,    var,    ___,    var,    lt) \
  _(ISGE,    var,    ___,    var,    lt) \
  _(ISLE,    var,    ___,    var,    le) \
  _(ISGT,    var,    ___,    var,    le) \
  \
  _(ISEQV,    var,    ___,    var,    eq) \
  _(ISNEV,    var,    ___,    var,    eq) \
  _(ISEQS,    var,    ___,    str,    eq) \
  _(ISNES,    var,    ___,    str,    eq) \
  _(ISEQN,    var,    ___,    num,    eq) \
  _(ISNEN,    var,    ___,    num,    eq) \
  _(ISEQP,    var,    ___,    pri,    eq) \
  _(ISNEP,    var,    ___,    pri,    eq) \
  \
   /* Unary test and copy ops. */ \
   _(ISTC,    dst,    ___,    var,    ___) \
   _(ISFC,    dst,    ___,    var,    ___) \
   _(IST,    ___,    ___,    var,    ___) \
   _(ISF,    ___,    ___,    var,    ___) \
   _(ISTYPE,    var,    ___,    lit,    ___) \
   _(ISNUM,    var,    ___,    lit,    ___) \
   \
   /* Unary ops. */ \
   _(MOV,    dst,    ___,    var,    ___) \
   _(NOT,    dst,    ___,    var,    ___) \
   _(UNM,    dst,    ___,    var,    unm) \
   \
   /* Binary ops. ORDER OPR. VV last, POW must be next. */ \
   _(ADDVN,    dst,    var,    num,    add) \
   _(SUBVN,    dst,    var,    num,    sub) \
   _(MULVN,    dst,    var,    num,    mul) \
   _(DIVVN,    dst,    var,    num,    div) \
   _(MODVN,    dst,    var,    num,    mod) \
   \
   _(ADDNV,    dst,    var,    num,    add) \
   _(SUBNV,    dst,    var,    num,    sub) \
   _(MULNV,    dst,    var,    num,    mul) \
   _(DIVNV,    dst,    var,    num,    div) \
   _(MODNV,    dst,    var,    num,    mod) \
   \
   _(ADDVV,    dst,    var,    var,    add) \
   _(SUBVV,    dst,    var,    var,    sub) \
   _(MULVV,    dst,    var,    var,    mul) \
   _(DIVVV,    dst,    var,    var,    div) \
   _(MODVV,    dst,    var,    var,    mod) \
   \
   _(POW,    dst,    var,    var,    pow) \
   _(CAT,    dst,    rbase,    rbase,    concat) \
   \
   /* Constant ops. */ \
   _(KSTR,    dst,    ___,    str,    ___) \
   _(KCDATA,    dst,    ___,    cdata,    ___) \
   _(KSHORT,    dst,    ___,    lits,    ___) \
   _(KNUM,    dst,    ___,    num,    ___) \
   _(KPRI,    dst,    ___,    pri,    ___) \
   _(KNIL,    base,    ___,    base,    ___) \
   \
   /* Upvalue and function ops. */ \
   _(UGET,    dst,    ___,    uv,    ___) \
   _(USETV,    uv,    ___,    var,    ___) \
   _(UINCV,    uv,    ___,    var,    ___) \
   _(USETS,    uv,    ___,    str,    ___) \
   _(USETN,    uv,    ___,    num,    ___) \
   _(UINCN,    uv,    ___,    num,    ___) \
   _(USETP,    uv,    ___,    pri,    ___) \
   _(UCLO,    rbase,    ___,    jump,    ___) \
   _(FNEW,    dst,    ___,    func,    gc) \
   \
   /* Table ops. */ \
   _(TNEW,    dst,    ___,    lit,    gc) \
   _(TDUP,    dst,    ___,    tab,    gc) \
   _(GGET,    dst,    ___,    str,    index) \
   _(GSET,    var,    ___,    str,    newindex) \
   _(GINC,    var,    ___,    str,    newindex) \
   _(TGETV,    dst,    var,    var,    index) \
   _(TGETS,    dst,    var,    str,    index) \
```

```
    _(TGETB,    dst,    var,    lit,    index) \
    _(TGETR,    dst,    var,    var,    index) \
    _(TSETV,    var,    var,    var,    newindex) \
    _(TINCV,    var,    var,    var,    newindex) \
    _(TSETS,    var,    var,    str,    newindex) \
    _(TINCS,    var,    var,    str,    newindex) \
    _(TSETB,    var,    var,    lit,    newindex) \
    _(TINCB,    var,    var,    lit,    newindex) \
    _(TSETM,    base,    ___,    num,    newindex) \
    _(TSETR,    var,    var,    var,    newindex) \
    \
    /* Calls and vararg handling. T = tail call. */ \
    _(CALLM,    base,    lit,    lit,    call) \
    _(CALL,    base,    lit,    lit,    call) \
    _(CALLMT,    base,    ___,    lit,    call) \
    _(CALLT,    base,    ___,    lit,    call) \
    _(ITERC,    base,    lit,    lit,    call) \
    _(ITERN,    base,    lit,    lit,    call) \
    _(VARG,    base,    lit,    lit,    ___) \
    _(ISNEXT,    base,    ___,    jump,    ___) \
    \
    /* Returns. */ \
    _(RETM,    base,    ___,    lit,    ___) \
    _(RET,    rbase,    ___,    lit,    ___) \
    _(RET0,    rbase,    ___,    lit,    ___) \
    _(RET1,    rbase,    ___,    lit,    ___) \
    \
    /* Loops and branches. I/J = interp/JIT, I/C/L = init/call/loop. */ \
    _(FORI,    base,    ___,    jump,    ___) \
    _(JFORI,    base,    ___,    jump,    ___) \
    \
    _(FORL,    base,    ___,    jump,    ___) \
    _(IFORL,    base,    ___,    jump,    ___) \
    _(JFORL,    base,    ___,    lit,    ___) \
    \
    _(ITERL,    base,    ___,    jump,    ___) \
    _(IITERL,    base,    ___,    jump,    ___) \
    _(JITERL,    base,    ___,    lit,    ___) \
    \
    _(LOOP,    rbase,    ___,    jump,    ___) \
    _(ILOOP,    rbase,    ___,    jump,    ___) \
    _(JLOOP,    rbase,    ___,    lit,    ___) \
    \
    _(JMP,    rbase,    ___,    jump,    ___) \
    \
    /*Function headers. I/J = interp/JIT, F/V/C = fixarg/vararg/C func.*/ \
    _(FUNCF,    rbase,    ___,    ___,    ___) \
    _(IFUNCF,    rbase,    ___,    ___,    ___) \
    _(JFUNCF,    rbase,    ___,    lit,    ___) \
    _(FUNCV,    rbase,    ___,    ___,    ___) \
    _(IFUNCV,    rbase,    ___,    ___,    ___) \
    _(JFUNCV,    rbase,    ___,    lit,    ___) \
    _(FUNCC,    rbase,    ___,    ___,    ___) \
    _(FUNCCW,    rbase,    ___,    ___,    ___) \
    \
    /* specific purpose bc. */    \
    _(VARGN,    dst, ___,    lit,    ___) \
    _(VARGSTR,    dst, ___,    lit,    ___) \
    _(VPROBENAME,    dst, ___,    lit,    ___) \
    _(VPID,        dst, ___,    lit,    ___) \
    _(VTID,        dst, ___,    lit,    ___) \
    _(VUID,        dst, ___,    lit,    ___) \
    _(VCPU,        dst, ___,    lit,    ___) \
    _(VEXECNAME,    dst, ___,    lit,    ___) \
    \
    _(GFUNC,    dst, ___,    ___,    ___) /*load global C function*/

/* Bytecode opcode numbers. */
typedef enum {
#define BCENUM(name, ma, mb, mc, mt)    BC_##name,
    BCDEF(BCENUM)
#undef BCENUM
    BC__MAX
} BCOp;

KP_STATIC_ASSERT((int)BC_ISEQV+1 == (int)BC_ISNEV);
```

```
KP_STATIC_ASSERT(((int)BC_ISEQV^1) == (int)BC_ISNEV);
KP_STATIC_ASSERT(((int)BC_ISEQS^1) == (int)BC_ISNES);
KP_STATIC_ASSERT(((int)BC_ISEQN^1) == (int)BC_ISNEN);
KP_STATIC_ASSERT(((int)BC_ISEQP^1) == (int)BC_ISNEP);
KP_STATIC_ASSERT(((int)BC_ISLT^1) == (int)BC_ISGE);
KP_STATIC_ASSERT(((int)BC_ISLE^1) == (int)BC_ISGT);
KP_STATIC_ASSERT(((int)BC_ISLT^3) == (int)BC_ISGT);
KP_STATIC_ASSERT((int)BC_IST-(int)BC_ISTC == (int)BC_ISF-(int)BC_ISFC);
KP_STATIC_ASSERT((int)BC_CALLT-(int)BC_CALL == (int)BC_CALLMT-(int)BC_CALLM);
KP_STATIC_ASSERT((int)BC_CALLMT + 1 == (int)BC_CALLT);
KP_STATIC_ASSERT((int)BC_RETM + 1 == (int)BC_RET);
KP_STATIC_ASSERT((int)BC_FORL + 1 == (int)BC_IFORL);
KP_STATIC_ASSERT((int)BC_FORL + 2 == (int)BC_JFORL);
KP_STATIC_ASSERT((int)BC_ITERL + 1 == (int)BC_IITERL);
KP_STATIC_ASSERT((int)BC_ITERL + 2 == (int)BC_JITERL);
KP_STATIC_ASSERT((int)BC_LOOP + 1 == (int)BC_ILOOP);
KP_STATIC_ASSERT((int)BC_LOOP + 2 == (int)BC_JLOOP);
KP_STATIC_ASSERT((int)BC_FUNCF + 1 == (int)BC_IFUNCF);
KP_STATIC_ASSERT((int)BC_FUNCF + 2 == (int)BC_JFUNCF);
KP_STATIC_ASSERT((int)BC_FUNCV + 1 == (int)BC_IFUNCV);
KP_STATIC_ASSERT((int)BC_FUNCV + 2 == (int)BC_JFUNCV);

/* This solves a circular dependency problem, change as needed. */
#define FF_next_N    4

/* Stack slots used by FORI/FORL, relative to operand A. */
enum {
    FORL_IDX, FORL_STOP, FORL_STEP, FORL_EXT
};

/* Bytecode operand modes. ORDER BCMode */
typedef enum {
    /* Mode A must be <= 7 */
    BCMnone, BCMdst, BCMbase, BCMvar, BCMrbase, BCMuv,
    BCMlit, BCMlits, BCMpri, BCMnum, BCMstr, BCMtab, BCMfunc,
    BCMjump, BCMcdata,
    BCM_max
} BCMode;

#define BCM___    BCMnone

#define bcmode_a(op)    ((BCMode)(bc_mode[op] & 7))
#define bcmode_b(op)    ((BCMode)((bc_mode[op] >> 3) & 15))
#define bcmode_c(op)    ((BCMode)((bc_mode[op] >> 7) & 15))
#define bcmode_d(op)    bcmode_c(op)
#define bcmode_hasd(op)    ((bc_mode[op] & (15 << 3)) == (BCMnone << 3))
#define bcmode_mm(op)    ((MMS)(bc_mode[op] >> 11))

#define BCMODE(name, ma, mb, mc, mm) \
    (BCM##ma | (BCM##mb << 3) | (BCM##mc << 7)|(MM_##mm << 11)),
#define BCMODE_FF    0

static inline int bc_isret(BCOp op)
{
    return (op == BC_RETM || op == BC_RET || op == BC_RET0 ||
        op == BC_RET1);
}

/*
* Metamethod definition
* Note ktap don't use any lua methmethod currently.
*/
typedef enum {
    MM_lt,
    MM_le,
    MM_eq,
    MM_unm,
    MM_add,
    MM_sub,
    MM_mul,
    MM_div,
    MM_mod,
    MM_pow,
    MM_concat,
    MM_gc,
    MM_index,
```

```
    MM_newindex,
    MM_call,
    MM__MAX,
    MM____ = MM__MAX
} MMS;


/* -- Bytecode dump format ------------------------------------------------ */

/*
** dump   = header proto+ 0U
** header = ESC 'L' 'J' versionB flagsU [namelenU nameB*]
** proto  = lengthU pdata
** pdata  = phead bcinsW* uvdataH* kgc* knum* [debugB*]
** phead  = flagsB numparamsB framesizeB numuvB numkgcU numknU numbcU
**          [debuglenU [firstlineU numlineU]]
** kgc    = kgctypeU { ktab | (loU hiU) | (rloU rhiU iloU ihiU) | strB* }
** knum   = intU0 | (loU1 hiU)
** ktab   = narrayU nhashU karray* khash*
** karray = ktabk
** khash  = ktabk ktabk
** ktabk  = ktabtypeU { intU | (loU hiU) | strB* }
**
** B = 8 bit, H = 16 bit, W = 32 bit, U = ULEB128 of W, U0/U1 = ULEB128 of W+1
*/

/* Bytecode dump header. */
#define BCDUMP_HEAD1        0x15
#define BCDUMP_HEAD2        0x22
#define BCDUMP_HEAD3        0x06

/* If you perform *any* kind of private modifications to the bytecode itself
** or to the dump format, you *must* set BCDUMP_VERSION to 0x80 or higher.
*/
#define BCDUMP_VERSION        1

/* Compatibility flags. */
#define BCDUMP_F_BE         0x01
#define BCDUMP_F_STRIP        0x02
#define BCDUMP_F_FFI        0x04

#define BCDUMP_F_KNOWN        (BCDUMP_F_FFI*2-1)

/* Type codes for the GC constants of a prototype. Plus length for strings. */
enum {
    BCDUMP_KGC_CHILD, BCDUMP_KGC_TAB, BCDUMP_KGC_I64, BCDUMP_KGC_U64,
    BCDUMP_KGC_COMPLEX, BCDUMP_KGC_STR
};

/* Type codes for the keys/values of a constant table. */
enum {
    BCDUMP_KTAB_NIL, BCDUMP_KTAB_FALSE, BCDUMP_KTAB_TRUE,
    BCDUMP_KTAB_INT, BCDUMP_KTAB_NUM, BCDUMP_KTAB_STR
};

#endif /* __KTAP_BC_H__ */
```

One Level Up Top Level

include/ktap_types.h - ktap

Data types defined

- StkId
- cdata_number
- csymbol
- csymbol_id
- ffi_state
- ffi_state
- ktap_cdata
- ktap_cdata_t
- ktap_cfunction
- ktap_eventdesc
- ktap_eventdesc_t
- ktap_func
- ktap_func_t
- ktap_global_state
- ktap_global_state_t
- ktap_instr_t
- ktap_node_t
- ktap_node_t
- ktap_number
- ktap_obj
- ktap_obj_t
- ktap_option
- ktap_option_t
- ktap_proto
- ktap_proto_t
- ktap_rawobj
- ktap_rawobj
- ktap_state
- ktap_state_t
- ktap_stats

- ktap_stats_t
- ktap_str
- ktap_str_t
- ktap_tab
- ktap_tab_t
- ktap_upval
- ktap_upval_t
- ktap_val
- ktap_val_t
- ptrdiff_t
- u8

Functions defined

- set_cfunc
- set_eventstr
- set_func
- set_ip
- set_kstack
- set_number
- set_proto
- set_string
- set_table

Macros defined

- G
- GCHeader
- KP_MAX_ABITS
- KP_MAX_ASIZE
- KP_MAX_BCINS
- KP_MAX_CACHED_CFUNCTION
- KP_MAX_COLOSIZE
- KP_MAX_HBITS
- KP_MAX_LINE
- KP_MAX_LOCVAR

- KP_MAX_MEMPOOL_SIZE
- KP_MAX_SLOTS
- KP_MAX_STACK_DEPTH
- KP_MAX_STR
- KP_MAX_STRNUM
- KP_MAX_STRTAB
- KP_MAX_UPVAL
- KP_MAX_XLEVEL
- KTAP_AUTHOR
- KTAP_CMD_IOC_EXIT
- KTAP_CMD_IOC_RUN
- KTAP_CMD_IOC_VERSION
- KTAP_COPYRIGHT
- KTAP_ERROR
- KTAP_EXIT
- KTAP_QL
- KTAP_QS
- KTAP_RIDX_GLOBALS
- KTAP_RIDX_LAST
- KTAP_RUNNING
- KTAP_STATS
- KTAP_TCDATA
- KTAP_TCFUNC
- KTAP_TEVENTSTR
- KTAP_TFALSE
- KTAP_TFUNC
- KTAP_TKIP
- KTAP_TKSTACK
- KTAP_TLIGHTUD
- KTAP_TNIL
- KTAP_TNUM
- KTAP_TNUMX
- KTAP_TPROTO

- KTAP_TRACE_END
- KTAP_TSTR
- KTAP_TTAB
- KTAP_TTHREAD
- KTAP_TTRUE
- KTAP_TUDATA
- KTAP_TUIP
- KTAP_TUPVAL
- KTAP_VERSION
- KTAP_VERSION_MAJOR
- KTAP_VERSION_MINOR
- MYINT
- PROTO_CHILD
- PROTO_CLCOUNT
- PROTO_CLC_BITS
- PROTO_CLC_POLY
- PROTO_FFI
- PROTO_FIXUP_RETURN
- PROTO_HAS_RETURN
- PROTO_ILOOP
- PROTO_NOJIT
- PROTO_UV_IMMUTABLE
- PROTO_UV_LOCAL
- PROTO_VARARG
- VERSION
- __KTAP_TYPES_H__
- boolvalue
- cdvalue
- clvalue
- fvalue
- gch
- gco2ts
- gco2uv

- gcvalue
- getstr
- hvalue
- incr_top
- is_bool
- is_btrace
- is_cdata
- is_cfunc
- is_eventstr
- is_false
- is_function
- is_kip
- is_nil
- is_number
- is_proto
- is_string
- is_table
- is_true
- itype
- nvalue
- obj2gco
- phvalue
- proto_bc
- proto_bcpos
- proto_chunkname
- proto_kgc
- proto_lineinfo
- proto_uv
- proto_uvinfo
- proto_varinfo
- ptvalue
- pvalue
- rawtsvalue

- set_bool
- set_cdata
- set_nil
- set_obj
- setitype
- svalue
- val_

Source code

```
/*
* ktap types definition.
*
* Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
* Copyright (C) 2005-2014 Mike Pall.
* Copyright (C) 1994-2008 Lua.org, PUC-Rio.
*/

#ifndef __KTAP_TYPES_H__
 #define __KTAP_TYPES_H__

 #ifdef __KERNEL__
 #include <linux/perf_event.h>
 #else
 typedef char u8;
 #include <stdlib.h>
 #include <stdio.h>
 #include <string.h>
 #include <stdint.h>
 typedef int ptrdiff_t;
 #endif

 #include "../include/ktap_bc.h"

 /* Various VM limits. */
 #define KP_MAX_MEMPOOL_SIZE    10000    /* Max. mempool size(Kbytes). */
 #define KP_MAX_STR    512        /* Max. string length. */
 #define KP_MAX_STRNUM    9999        /* Max. string number. */

 #define KP_MAX_STRTAB    (1<<26)        /* Max. string table size. */
 #define KP_MAX_HBITS    26        /* Max. hash bits. */
 #define KP_MAX_ABITS    28        /* Max. bits of array key. */
 #define KP_MAX_ASIZE    ((1<<(KP_MAX_ABITS-1))+1)  /* Max. array part size. */
 #define KP_MAX_COLOSIZE    16        /* Max. elems for colocated array. */

 #define KP_MAX_LINE    1000        /* Max. source code line number. */
 #define KP_MAX_XLEVEL    200        /* Max. syntactic nesting level. */
 #define KP_MAX_BCINS    (1<<26)        /* Max. # of bytecode instructions. */
 #define KP_MAX_SLOTS    250        /* Max. # of slots in a ktap func. */
 #define KP_MAX_LOCVAR    200        /* Max. # of local variables. */
 #define KP_MAX_UPVAL    60        /* Max. # of upvalues. */

 #define KP_MAX_CACHED_CFUNCTION    128    /* Max. cached global cfunction */

 #define KP_MAX_STACK_DEPTH    50    /* Max. stack depth */

 /*
 * The first argument type of kdebug.trace_by_id()
 * The value is a userspace memory pointer.
 * Maybe embed it into the trunk file in future.
 */
 typedef struct ktap_eventdesc {
     int nr;  /* the number of events id */
     int *id_arr; /* events id array */
     char *filter;
 } ktap_eventdesc_t;
```

```
57
58
59  /* ktap option for each script */
60  typedef struct ktap_option {
61      char *trunk; /* __user */
62      int trunk_len;
63      int argc;
64      char **argv; /* __user */
65      int verbose;
66      int trace_pid;
67      int workload;
68      int trace_cpu;
69      int print_timestamp;
70      int quiet;
71      int dry_run;
72  } ktap_option_t;
73
74  /*
75  * Ioctls that can be done on a ktap fd:
76  * todo: use _IO macro in include/uapi/asm-generic/ioctl.h
77  */
78  #define KTAP_CMD_IOC_VERSION        ('$' + 0)
79  #define KTAP_CMD_IOC_RUN        ('$' + 1)
80  #define KTAP_CMD_IOC_EXIT        ('$' + 3)
81
82  #define KTAP_VERSION_MAJOR       "0"
83  #define KTAP_VERSION_MINOR       "4"
84
85  #define KTAP_VERSION    "ktap " KTAP_VERSION_MAJOR "." KTAP_VERSION_MINOR
86  #define KTAP_AUTHOR    "Jovi Zhangwei <jovi.zhangwei@gmail.com>"
87  #define KTAP_COPYRIGHT  KTAP_VERSION "  Copyright (C) 2012-2014, " KTAP_AUTHOR
88
89  #define MYINT(s)        (s[0] - '0')
90  #define VERSION         (MYINT(KTAP_VERSION_MAJOR) * 16 + MYINT(KTAP_VERSION_MINOR))
91
92  typedef long ktap_number;
93  typedef int ktap_instr_t;
94  typedef union ktap_obj ktap_obj_t;
95
96  struct ktap_state;
97  typedef int (*ktap_cfunction) (struct ktap_state *ks);
98
99  /* ktap_val_t is basic value type in ktap stack, for reference all objects */
100  typedef struct ktap_val {
101      union {
102          ktap_obj_t *gc;        /* collectable objects, str/tab/... */
103          void *p;        /* light userdata */
104          ktap_cfunction f;    /* light C functions */
105          ktap_number n;        /* numbers */
106          struct {
107              uint16_t depth;    /* stack depth */
108              uint16_t skip;    /* skip stack entries */
109          } stack;
110      } val;
111      union {
112          int type;        /* type for above val */
113          const unsigned int *pcr;/* Overlaps PC for ktap frames.*/
114      };
115  } ktap_val_t;
116
117  typedef ktap_val_t *StkId;
118
119  #define GCHeader ktap_obj_t *nextgc; u8 gct;
120
121  typedef struct ktap_str {
122      GCHeader;
123      u8 reserved;  /* Used by lexer for fast lookup of reserved words. */
124      u8 extra;
125      unsigned int hash;
126      int len;  /* number of characters in string */
127  } ktap_str_t;
128
129  typedef struct ktap_upval {
130      GCHeader;
131      uint8_t closed;    /* Set if closed (i.e. uv->v == &uv->u.value). */
132      uint8_t immutable;    /* Immutable value. */
```

```
    union {
        ktap_val_t tv; /* If closed: the value itself. */
        struct { /* If open: double linked list, anchored at thread. */
            struct ktap_upval *prev;
            struct ktap_upval *next;
        };
    };
    ktap_val_t *v;  /* Points to stack slot (open) or above (closed). */
} ktap_upval_t;


typedef struct ktap_func {
    GCHeader;
    u8 nupvalues;
    BCIns *pc;
    struct ktap_proto *p;
    struct ktap_upval *upvals[1];  /* list of upvalues */
} ktap_func_t;

typedef struct ktap_proto {
    GCHeader;
    uint8_t numparams;    /* Number of parameters. */
    uint8_t framesize;    /* Fixed frame size. */
    int sizebc;        /* Number of bytecode instructions. */
    ktap_obj_t *gclist;
    void *k;    /* Split constant array (points to the middle). */
    void *uv;    /* Upvalue list. local slot|0x8000 or parent uv idx. */
    int sizekgc;    /* Number of collectable constants. */
    int sizekn;    /* Number of lua_Number constants. */
    int sizept;    /* Total size including colocated arrays. */
    uint8_t sizeuv;    /* Number of upvalues. */
    uint8_t flags;    /* Miscellaneous flags (see below). */

    /* --- The following fields are for debugging/tracebacks only --- */
    ktap_str_t *chunkname;    /* Chunk name this function was defined in. */
    BCLine firstline;    /* First line of the function definition. */
    BCLine numline;    /* Number of lines for the function definition. */
    void *lineinfo;    /* Compressed map from bytecode ins. to source line. */
    void *uvinfo;    /* Upvalue names. */
    void *varinfo;    /* Names and compressed extents of local variables. */
} ktap_proto_t;

/* Flags for prototype. */
#define PROTO_CHILD        0x01    /* Has child prototypes. */
#define PROTO_VARARG        0x02    /* Vararg function. */
#define PROTO_FFI        0x04    /* Uses BC_KCDATA for FFI datatypes. */
#define PROTO_NOJIT        0x08    /* JIT disabled for this function. */
#define PROTO_ILOOP        0x10    /* Patched bytecode with ILOOP etc. */
/* Only used during parsing. */
#define PROTO_HAS_RETURN    0x20    /* Already emitted a return. */
#define PROTO_FIXUP_RETURN    0x40    /* Need to fixup emitted returns. */
/* Top bits used for counting created closures. */
#define PROTO_CLCOUNT        0x20    /* Base of saturating 3 bit counter. */
#define PROTO_CLC_BITS        3
#define PROTO_CLC_POLY        (3*PROTO_CLCOUNT)  /* Polymorphic threshold. */

#define PROTO_UV_LOCAL        0x8000    /* Upvalue for local slot. */
#define PROTO_UV_IMMUTABLE    0x4000    /* Immutable upvalue. */

#define proto_kgc(pt, idx)    (((ktap_obj_t *)(pt)->k)[idx])
#define proto_bc(pt)        ((BCIns *)((char *)(pt) + sizeof(ktap_proto_t)))
#define proto_bcpos(pt, pc)    ((BCPos)((pc) - proto_bc(pt)))
#define proto_uv(pt)        ((uint16_t *)(pt)->uv)

#define proto_chunkname(pt)    ((pt)->chunkname)
#define proto_lineinfo(pt)    ((const void *)(pt)->lineinfo)
#define proto_uvinfo(pt)    ((const uint8_t *)(pt)->uvinfo)
#define proto_varinfo(pt)    ((const uint8_t *)(pt)->varinfo)


typedef struct ktap_node_t {
    ktap_val_t val; /* Value object. Must be first field. */
    ktap_val_t key; /* Key object. */
    struct ktap_node_t *next;  /* hash chain */
} ktap_node_t;

```

```
/* ktap_tab */
typedef struct ktap_tab {
    GCHeader;
#ifdef __KERNEL__
    arch_spinlock_t lock;
#endif
    ktap_val_t *array;    /* Array part. */
    ktap_node_t *node;    /* Hash part. */
    ktap_node_t *freetop; /* any free position is before this position */

    uint32_t asize;        /* Size of array part (keys [0, asize-1]). */
    uint32_t hmask;        /* log2 of size of `node' array */

    uint32_t hnum;        /* number of all nodes */

    ktap_obj_t *gclist;
} ktap_tab_t;

#ifdef CONFIG_KTAP_FFI
typedef int csymbol_id;
typedef uint64_t cdata_number;
typedef struct csymbol csymbol;

/* global ffi state maintained in each ktap vm instance */
typedef struct ffi_state {
    ktap_tab_t *ctable;
    int csym_nr;
    csymbol *csym_arr;
} ffi_state;

/* instance of csymbol */
typedef struct ktap_cdata {
    GCHeader;
    csymbol_id id;
    union {
        cdata_number i;
        struct {
            void *addr;
            int nmemb;    /* number of memory block */
        } p;            /* pointer data */
        void *rec;        /* struct member or union data */
    } u;
} ktap_cdata_t;
#endif

typedef struct ktap_stats {
    int mem_allocated;
    int nr_mem_allocate;
    int events_hits;
    int events_missed;
} ktap_stats_t;

#define KTAP_STATS(ks)    this_cpu_ptr(G(ks)->stats)


#define KTAP_RUNNING    0 /* normal running state */
#define KTAP_TRACE_END    1 /* running in trace_end function */
#define KTAP_EXIT    2 /* normal exit, set when call exit() */
#define KTAP_ERROR    3 /* error state, called by kp_error */

typedef struct ktap_global_state {
    void *mempool;        /* string memory pool */
    void *mp_freepos;    /* free position in memory pool */
    int mp_size;        /* memory pool size */
#ifdef __KERNEL__
    arch_spinlock_t mp_lock;/* mempool lock */
#endif

    ktap_str_t **strhash;    /* String hash table (hash chain anchors). */
    int strmask;        /* String hash mask (size of hash table-1). */
    int strnum;        /* Number of strings in hash table. */
#ifdef __KERNEL__
    arch_spinlock_t str_lock; /* string operation lock */
#endif

    ktap_val_t registry;
```

```
285     ktap_tab_t *gtab;    /* global table contains cfunction and args */
286     ktap_obj_t *allgc; /* list of all collectable objects */
287     ktap_upval_t uvhead; /* head of list of all open upvalues */
288
289     struct ktap_state *mainthread; /*main state */
290     int state; /* status of ktapvm, KTAP_RUNNING, KTAP_TRACE_END, etc */
291 #ifdef __KERNEL__
292     /* reserved global percpu data */
293     void __percpu *percpu_state[PERF_NR_CONTEXTS];
294     void __percpu *percpu_print_buffer[PERF_NR_CONTEXTS];
295     void __percpu *percpu_temp_buffer[PERF_NR_CONTEXTS];
296
297     /* for recursion tracing check */
298     int __percpu *recursion_context[PERF_NR_CONTEXTS];
299
300     ktap_option_t *parm; /* ktap options */
301     pid_t trace_pid;
302     struct task_struct *trace_task;
303     cpumask_var_t cpumask;
304     struct ring_buffer *buffer;
305     struct dentry *trace_pipe_dentry;
306     struct task_struct *task;
307     int trace_enabled;
308     int wait_user; /* flag to indicat waiting user consume content */
309
310     struct list_head timers; /* timer list */
311     struct ktap_stats __percpu *stats; /* memory allocation stats */
312     struct list_head events_head; /* probe event list */
313
314     ktap_func_t *trace_end_closure; /* trace_end closure */
315 #ifdef CONFIG_KTAP_FFI
316     ffi_state  ffis;
317 #endif
318
319     /* C function table for fast call */
320     int nr_builtin_cfunction;
321     ktap_cfunction gfunc_tbl[KP_MAX_CACHED_CFUNCTION];
322 #endif
323 } ktap_global_state_t;
324
325
326 typedef struct ktap_state {
327     ktap_global_state_t *g;    /* global state */
328     int stop;        /* don't enter tracing handler if stop is 1 */
329     StkId top;        /* stack top */
330     StkId func;        /* execute light C function */
331     StkId stack_last;    /* last stack pointer */
332     StkId stack;        /* ktap stack, percpu pre-reserved */
333     ktap_upval_t *openupval;/* opened upvals list */
334
335 #ifdef __KERNEL__
336     /* current fired event which allocated on stack */
337     struct ktap_event_data *current_event;
338 #endif
339 } ktap_state_t;
340
341 #define G(ks)   (ks->g)
342
343 typedef struct ktap_rawobj {
344     GCHeader;
345     void *v;
346 } ktap_rawobj;
347
348 /*
349 * Union of all collectable objects
350 */
351 union ktap_obj {
352     struct { GCHeader } gch;
353     struct ktap_str ts;
354     struct ktap_func fn;
355     struct ktap_tab h;
356     struct ktap_proto pt;
357     struct ktap_upval uv;
358     struct ktap_state th;  /* thread */
359     struct ktap_rawobj rawobj;
360 #ifdef CONFIG_KTAP_FFI
```

```
    struct ktap_cdata cd;
#endif
};

#define gch(o)            (&(o)->gch)

/* macros to convert a ktap_obj_t into a specific value */
#define gco2ts(o)        (&((o)->ts))
#define gco2uv(o)        (&((o)->uv))
#define obj2gco(v)        ((ktap_obj_t *)(v))

/* predefined values in the registry */
#define KTAP_RIDX_GLOBALS    1
#define KTAP_RIDX_LAST        KTAP_RIDX_GLOBALS

/* ktap object types */
#define KTAP_TNIL        (~0u)
#define KTAP_TFALSE        (~1u)
#define KTAP_TTRUE        (~2u)
#define KTAP_TNUM        (~3u)
#define KTAP_TLIGHTUD        (~4u)
#define KTAP_TSTR        (~5u)
#define KTAP_TUPVAL        (~6u)
#define KTAP_TTHREAD        (~7u)
#define KTAP_TPROTO        (~8u)
#define KTAP_TFUNC        (~9u)
#define KTAP_TCFUNC        (~10u)
#define KTAP_TCDATA        (~11u)
#define KTAP_TTAB        (~12u)
#define KTAP_TUDATA        (~13u)

/* Specfic types */
#define KTAP_TEVENTSTR        (~14u) /* argstr */
#define KTAP_TKSTACK        (~15u) /* stack(), not intern to string yet */
#define KTAP_TKIP        (~16u) /* kernel function ip addres */
#define KTAP_TUIP        (~17u) /* userspace function ip addres */

/* This is just the canonical number type used in some places. */
#define KTAP_TNUMX        (~18u)


#define itype(o)        ((o)->type)
#define setitype(o, t)        ((o)->type = (t))

#define val_(o)            ((o)->val)
#define gcvalue(o)        (val_(o).gc)

#define nvalue(o)        (val_(o).n)
#define boolvalue(o)        (KTAP_TFALSE - (o)->type)
#define hvalue(o)        (&val_(o).gc->h)
#define phvalue(o)        (&val_(o).gc->ph)
#define clvalue(o)        (&val_(o).gc->fn)
#define ptvalue(o)        (&val_(o).gc->pt)

#define getstr(ts)        (const char *)((ts) + 1)
#define rawtsvalue(o)        (&val_(o).gc->ts)
#define svalue(o)        getstr(rawtsvalue(o))

#define pvalue(o)        (&val_(o).p)
#define fvalue(o)        (val_(o).f)
#ifdef CONFIG_KTAP_FFI
#define cdvalue(o)        (&val_(o).gc->cd)
#endif

#define is_nil(o)        (itype(o) == KTAP_TNIL)
#define is_false(o)        (itype(o) == KTAP_TFALSE)
#define is_true(o)        (itype(o) == KTAP_TTRUE)
#define is_bool(o)        (is_false(o) || is_true(o))
#define is_string(o)        (itype(o) == KTAP_TSTR)
#define is_number(o)        (itype(o) == KTAP_TNUM)
#define is_table(o)        (itype(o) == KTAP_TTAB)
#define is_proto(o)        (itype(o) == KTAP_TPROTO)
#define is_function(o)        (itype(o) == KTAP_TFUNC)
#define is_cfunc(o)        (itype(o) == KTAP_TCFUNC)
#define is_eventstr(o)        (itype(o) == KTAP_TEVENTSTR)
#define is_kip(o)        (itype(o) == KTAP_TKIP)
```

```
#define is_btrace(o)        (itype(o) == KTAP_TBTRACE)
#ifdef CONFIG_KTAP_FFI
#define is_cdata(o)        (itype(o) == KTAP_TCDATA)
#endif


#define set_nil(o)        ((o)->type = KTAP_TNIL)
#define set_bool(o, x)        ((o)->type = KTAP_TFALSE-(uint32_t)(x))

static inline void set_number(ktap_val_t *o, ktap_number n)
{
    setitype(o, KTAP_TNUM);
    o->val.n = n;
}

static inline void set_string(ktap_val_t *o, const ktap_str_t *str)
{
    setitype(o, KTAP_TSTR);
    o->val.gc = (ktap_obj_t *)str;
}

static inline void set_table(ktap_val_t *o, ktap_tab_t *tab)
{
    setitype(o, KTAP_TTAB);
    o->val.gc = (ktap_obj_t *)tab;
}

static inline void set_proto(ktap_val_t *o, ktap_proto_t *pt)
{
    setitype(o, KTAP_TPROTO);
    o->val.gc = (ktap_obj_t *)pt;
}

static inline void set_kstack(ktap_val_t *o, uint16_t depth, uint16_t skip)
{
    setitype(o, KTAP_TKSTACK);
    o->val.stack.depth = depth;
    o->val.stack.skip = skip;
}

static inline void set_func(ktap_val_t *o, ktap_func_t *fn)
{
    setitype(o, KTAP_TFUNC);
    o->val.gc = (ktap_obj_t *)fn;
}

static inline void set_cfunc(ktap_val_t *o, ktap_cfunction fn)
{
    setitype(o, KTAP_TCFUNC);
    o->val.f = fn;
}

static inline void set_eventstr(ktap_val_t *o)
{
    setitype(o, KTAP_TEVENTSTR);
}

static inline void set_ip(ktap_val_t *o, unsigned long addr)
{
    setitype(o, KTAP_TKIP);
    o->val.n = addr;
}


#ifdef CONFIG_KTAP_FFI
#define set_cdata(o, x)        { setitype(o, KTAP_TCDATA); (o)->val.gc = x; }
#endif

#define set_obj(o1, o2)        { *(o1) = *(o2); }

#define incr_top(ks)        {ks->top++;}

/*
* KTAP_QL describes how error messages quote program elements.
* CHANGE it if you want a different appearance.
*/
```

```
513 #define KTAP_QL(x)      "'" x "'"
514 #define KTAP_QS         KTAP_QL("%s")
515
516 #endif /* __KTAP_TYPES_H__ */
517
```

One Level Up Top Level

userspace/kp_main.c - ktap

Global variables defined

- dry_run
- dump_bytecode
- forks
- ktap_trunk_mem_size
- oneline_src
- output_filename
- print_timestamp
- quiet
- script_args_end
- script_args_start
- script_file
- trace_cpu
- trace_pid
- uparm
- verbose
- workload_argv

Data types defined

- binary_base

Functions defined

- compile
- fork_workload
- kp_writer
- main
- parse
- parse_option
- print_symbol
- run_ktapvm
- usage

Macros defined

- KTAPVM_PATH
- SIMPLE_ONE_LINER_FMT
- handle_error

Source code

```
/*
* main.c - ktap compiler and loader entry
*
* This file is part of ktap by Jovi Zhangwei.
*
* Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
*
* ktap is free software; you can redistribute it and/or modify it
* under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <string.h>
#include <signal.h>
#include <stdarg.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <math.h>
#include <linux/errno.h>

#include "../include/ktap_types.h"
#include "kp_lex.h"
#include "kp_parse.h"
#include "kp_symbol.h"
#include "cparser.h"

static void usage(const char *msg_fmt, ...)
{
    va_list ap;

    va_start(ap, msg_fmt);
    vfprintf(stderr, msg_fmt, ap);
    va_end(ap);

    fprintf(stderr,
"Usage: ktap [options] file [script args] -- cmd [args]\n"
"   or: ktap [options] -e one-liner  -- cmd [args]\n"
"\n"
"Options and arguments:\n"
"  -o file        : send script output to file, instead of stderr\n"
"  -p pid         : specific tracing pid\n"
"  -C cpu         : cpu to monitor in system-wide\n"
"  -T             : show timestamp for event\n"
"  -V             : show version\n"
"  -v             : enable verbose mode\n"
"  -q             : suppress start tracing message\n"
"  -d             : dry run mode(register NULL callback to perf events)\n"
"  -s             : simple event tracing\n"
"  -b             : list byte codes\n"
```

```
"  -le [glob]     : list pre-defined events in system\n"
#ifndef NO_LIBELF
"  -lf DSO        : list available functions from DSO\n"
"  -lm DSO        : list available sdt notes from DSO\n"
#endif
"  file           : program read from script file\n"
"  -- cmd [args]  : workload to tracing\n");

    exit(EXIT_FAILURE);
}

#define handle_error(str) do { perror(str); exit(-1); } while(0)

ktap_option_t uparm;
static int ktap_trunk_mem_size = 1024;

static int kp_writer(const void* p, size_t sz, void* ud)
{
    if (uparm.trunk_len + sz > ktap_trunk_mem_size) {
        int new_size = (uparm.trunk_len + sz) * 2;
        uparm.trunk = realloc(uparm.trunk, new_size);
        ktap_trunk_mem_size = new_size;
    }

    memcpy(uparm.trunk + uparm.trunk_len, p, sz);
    uparm.trunk_len += sz;

    return 0;
}


static int forks;
static char **workload_argv;

 static int fork_workload(int ktap_fd)
 {
     int pid;

     pid = fork();
     if (pid < 0)
         handle_error("failed to fork");

     if (pid > 0)
         return pid;

     signal(SIGTERM, SIG_DFL);

     execvp("", workload_argv);

     /*
      * waiting ktapvm prepare all tracing event
      * make it more robust in future.
      */
     pause();

     execvp(workload_argv[0], workload_argv);

     perror(workload_argv[0]);
     exit(-1);

     return -1;
 }

 #define KTAPVM_PATH "/sys/kernel/debug/ktap/ktapvm"

 static char *output_filename;

 static int run_ktapvm()
 {
         int ktapvm_fd, ktap_fd;
     int ret;

     ktapvm_fd = open(KTAPVM_PATH, O_RDONLY);
     if (ktapvm_fd < 0)
         handle_error("open " KTAPVM_PATH " failed");

```

```
    ktap_fd = ioctl(ktapvm_fd, 0, NULL);
    if (ktap_fd < 0)
        handle_error("ioctl ktapvm failed");

    kp_create_reader(output_filename);

    if (forks) {
        uparm.trace_pid = fork_workload(ktap_fd);
        uparm.workload = 1;
    }

    ret = ioctl(ktap_fd, KTAP_CMD_IOC_RUN, &uparm);
    switch (ret) {
    case -EPERM:
    case -EACCES:
        fprintf(stderr, "You may not have permission to run ktap\n");
        break;
    }

    close(ktap_fd);
    close(ktapvm_fd);

    return ret;
}

int verbose;
static int quiet;
static int dry_run;
static int dump_bytecode;
static char oneline_src[1024];
static int trace_pid = -1;
static int trace_cpu = -1;
static int print_timestamp;

#define SIMPLE_ONE_LINER_FMT    \
    "trace %s { print(cpu(), tid(), execname(), argstr) }"

static const char *script_file;
static int script_args_start;
static int script_args_end;

#ifndef NO_LIBELF
struct binary_base
{
    int type;
    const char *binary;
};
static int print_symbol(const char *name, vaddr_t addr, void *arg)
{
    struct binary_base *base = (struct binary_base *)arg;
    const char *type = base->type == FIND_SYMBOL ?
        "probe" : "sdt";

    printf("%s:%s:%s\n", type, base->binary, name);
    return 0;
}
#endif

static void parse_option(int argc, char **argv)
{
    char pid[32] = {0};
    char cpu_str[32] = {0};
    char *next_arg;
    int i, j;

    for (i = 1; i < argc; i++) {
        if (argv[i][0] != '-') {
            script_file = argv[i];
            if (!script_file)
                usage("");

            script_args_start = i + 1;
            script_args_end = argc;

            for (j = i + 1; j < argc; j++) {
                if (argv[j][0] == '-' && argv[j][1] == '-')
```

```
                    goto found_cmd;
            }

            return;
        }

        if (argv[i][0] == '-' && argv[i][1] == '-') {
            j = i;
            goto found_cmd;
        }

        next_arg = argv[i + 1];

        switch (argv[i][1]) {
        case 'o':
            output_filename = malloc(strlen(next_arg) + 1);
            if (!output_filename)
                return;

            strncpy(output_filename, next_arg, strlen(next_arg));
            i++;
            break;
        case 'e':
            strncpy(oneline_src, next_arg, strlen(next_arg));
            i++;
            break;
        case 'p':
            strncpy(pid, next_arg, strlen(next_arg));
            trace_pid = atoi(pid);
            i++;
            break;
        case 'C':
            strncpy(cpu_str, next_arg, strlen(next_arg));
            trace_cpu = atoi(cpu_str);
            i++;
            break;
        case 'T':
            print_timestamp = 1;
            break;
        case 'v':
            verbose = 1;
            break;
        case 'q':
            quiet = 1;
            break;
        case 'd':
            dry_run = 1;
            break;
        case 's':
            sprintf(oneline_src, SIMPLE_ONE_LINER_FMT, next_arg);
            i++;
            break;
        case 'b':
            dump_bytecode = 1;
            break;
        case 'l': /* list available events */
            switch (argv[i][2]) {
            case 'e': /* tracepoints */
                list_available_events(next_arg);
                exit(EXIT_SUCCESS);
#ifndef NO_LIBELF
            case 'f': /* functions in DSO */
            case 'm': /* static marks in DSO */ {
                const char *binary = next_arg;
                int type = argv[i][2] == 'f' ?
                        FIND_SYMBOL : FIND_STAPSDT_NOTE;
                struct binary_base base = {
                    .type = type,
                    .binary = binary,
                };
                int ret;

                ret = parse_dso_symbols(binary, type,
                            print_symbol,
                            (void *)&base);
                if (ret <= 0) {
```

```
                    fprintf(stderr,
                    "error: no symbols in binary %s\n",
                        binary);
                    exit(EXIT_FAILURE);
                }
                exit(EXIT_SUCCESS);
            }
#endif
            default:
                exit(EXIT_FAILURE);
            }
            break;
        case 'V':
#ifdef CONFIG_KTAP_FFI
            usage("%s (with FFI)\n\n", KTAP_VERSION);
#else
            usage("%s\n\n", KTAP_VERSION);
#endif
            break;
        case '?':
        case 'h':
            usage("");
            break;
        default:
            usage("wrong argument\n");
            break;
        }
    }

    return;

found_cmd:
    script_args_end = j;
    forks = 1;
    workload_argv = &argv[j + 1];
}

static ktap_proto_t *parse(const char *chunkname, const char *src)
{
    LexState ls;

    ls.chunkarg = chunkname ? chunkname : "?";
    kp_lex_init();
    kp_buf_init(&ls.sb);
    kp_lex_setup(&ls, src);
    return kp_parse(&ls);
}

static void compile(const char *input)
{
    ktap_proto_t *pt;
    char *buff;
    struct stat sb;
    int fdin;

    kp_str_resize();

    if (oneline_src[0] != '\0') {
        ffi_cparser_init();
        pt = parse(input, oneline_src);
        goto dump;
    }

    fdin = open(input, O_RDONLY);
    if (fdin < 0) {
        fprintf(stderr, "open file %s failed\n", input);
        exit(-1);
    }

    if (fstat(fdin, &sb) == -1)
        handle_error("fstat failed");

    buff = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fdin, 0);
    if (buff == MAP_FAILED)
        handle_error("mmap failed");

```

```
    ffi_cparser_init();
    pt = parse(input, buff);

    munmap(buff, sb.st_size);
    close(fdin);

dump:
    if (dump_bytecode) {
#ifdef CONFIG_KTAP_FFI
        kp_dump_csymbols();
#endif
        kp_dump_proto(pt);
        exit(0);
    }

    /* bcwrite */
    uparm.trunk = malloc(ktap_trunk_mem_size);
    if (!uparm.trunk)
        handle_error("malloc failed");

    kp_bcwrite(pt, kp_writer, NULL, 0);
    ffi_cparser_free();
}

int main(int argc, char **argv)
{
    char **ktapvm_argv;
    int new_index, i;
    int ret;

    if (argc == 1)
        usage("");

    parse_option(argc, argv);

    if (oneline_src[0] != '\0')
        script_file = "(command line)";

    compile(script_file);

    ktapvm_argv = (char **)malloc(sizeof(char *)*(script_args_end -
                    script_args_start + 1));
    if (!ktapvm_argv) {
        fprintf(stderr, "canno allocate ktapvm_argv\n");
        return -1;
    }

    ktapvm_argv[0] = malloc(strlen(script_file) + 1);
    if (!ktapvm_argv[0]) {
        fprintf(stderr, "canno allocate memory\n");
        return -1;
    }
    strcpy(ktapvm_argv[0], script_file);
    ktapvm_argv[0][strlen(script_file)] = '\0';

    /* pass rest argv into ktapvm */
    new_index = 1;
    for (i = script_args_start; i < script_args_end; i++) {
        ktapvm_argv[new_index] = malloc(strlen(argv[i]) + 1);
        if (!ktapvm_argv[new_index]) {
            fprintf(stderr, "canno allocate memory\n");
            free(ktapvm_argv);
            return -1;
        }
        strcpy(ktapvm_argv[new_index], argv[i]);
        ktapvm_argv[new_index][strlen(argv[i])] = '\0';
        new_index++;
    }

    uparm.argv = ktapvm_argv;
    uparm.argc = new_index;
    uparm.verbose = verbose;
    uparm.trace_pid = trace_pid;
    uparm.trace_cpu = trace_cpu;
    uparm.print_timestamp = print_timestamp;
    uparm.quiet = quiet;
```

```
446     uparm.dry_run = dry_run;
447
448     /* start running into kernel ktapvm */
449     ret = run_ktapvm();
450
451     cleanup_event_resources();
452     return ret;
453 }
454
455
```

One Level Up Top Level

One Level Up Top Level

userspace/ - ktap

- cparser.h
- ffi/
- kp_bcwrite.c
- kp_lex.c
- kp_lex.h
- kp_main.c
- kp_parse.c
- kp_parse.h
- kp_parse_events.c
- kp_reader.c
- kp_symbol.c
- kp_symbol.h
- kp_util.c
- kp_util.h

One Level Up Top Level

userspace/cparser.h - ktap

Data types defined

- cp_ctype
- parser

Functions defined

- cp_csymf_arg
- cp_csymf_ret
- ffi_cparser_free
- ffi_cparser_init

Macros defined

- ALIGNED_DEFAULT
- ALIGNOF
- DEFAULT_ALIGN_MASK
- FUNCTION_ALIGN_MASK
- IS_CHAR_UNSIGNED
- MAX_TYPE_NAME_LEN
- POINTER_BITS
- POINTER_MAX
- PTR_ALIGN_MASK
- __KTAP_CPARSER_H__
- ct_ffi_cs

Source code

```
#ifndef __KTAP_CPARSER_H__
#define __KTAP_CPARSER_H__

/*
 * Copyright (c) 2011 James R. McKaskill
 *
 * This software is licensed under the stock MIT license:
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

```
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 * DEALINGS IN THE SOFTWARE.
 *
 * ------------------------------------------------------------------------
 */

/*
 * Adapted from luaffi commit: abc638c9341025580099dcf77795c4b320ba0e63
 *
 * Copyright (c) 2013 Yicheng Qin, Qingping Hou
 */

#ifdef CONFIG_KTAP_FFI

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>

#include "../include/ktap_ffi.h"

#define PTR_ALIGN_MASK (sizeof(void*) - 1)
#define FUNCTION_ALIGN_MASK (sizeof(void (*)()) - 1)
#define DEFAULT_ALIGN_MASK 7

struct parser {
    int line;
    const char *next;
    const char *prev;
    unsigned align_mask;
};

enum {
    C_CALL,
    STD_CALL,
    FAST_CALL,
};


#define MAX_TYPE_NAME_LEN CSYM_NAME_MAX_LEN

enum {
    /* 0 - 4 */
    INVALID_TYPE,
    VOID_TYPE,
    BOOL_TYPE,
    INT8_TYPE,
    INT16_TYPE,
    /* 5 - 9 */
    INT32_TYPE,
    INT64_TYPE,
    INTPTR_TYPE,
    ENUM_TYPE,
    UNION_TYPE,
    /* 10 - 12 */
    STRUCT_TYPE,
    FUNCTION_TYPE,
    FUNCTION_PTR_TYPE,
};


#define IS_CHAR_UNSIGNED (((char) -1) > 0)

#define POINTER_BITS 2
#define POINTER_MAX ((1 << POINTER_BITS) - 1)

#define ALIGNOF(S) ((int) ((char*) &S.v - (char*) &S - 1))


/* Note: if adding a new member that is associated with a struct/union
```

```
 * definition then it needs to be copied over in ctype.c:set_defined for when
 * we create types based off of the declaration alone.
 *
 * Since this is used as a header for every ctype and cdata, and we create a
 * ton of them on the stack, we try and minimise its size.
 */
struct cp_ctype {
    size_t base_size; /* size of the base type in bytes */
    int ffi_base_cs_id;
    int ffi_cs_id; /* index for csymbol from ktap vm */
    union {
        /* valid if is_bitfield */
        struct {
            /* size of bitfield in bits */
            unsigned bit_size : 7;
            /* offset within the current byte between 0-63 */
            unsigned bit_offset : 6;
        };
        /* Valid if is_array */
        size_t array_size;
        /* Valid for is_variable_struct or is_variable_array. If
         * variable_size_known (only used for is_variable_struct)
         * then this is the total increment otherwise this is the
         * per element increment.
         */
        size_t variable_increment;
    };
    size_t offset;
    /* as (align bytes - 1) eg 7 gives 8 byte alignment */
    unsigned align_mask : 4;
    /* number of dereferences to get to the base type
     * including +1 for arrays */
    unsigned pointers : POINTER_BITS;
    /* const pointer mask, LSB is current pointer, +1 for the whether
     * the base type is const */
    unsigned const_mask : POINTER_MAX + 1;
    unsigned type : 5; /* value given by type enum above */
    unsigned is_reference : 1;
    unsigned is_array : 1;
    unsigned is_defined : 1;
    unsigned is_null : 1;
    unsigned has_member_name : 1;
    unsigned calling_convention : 2;
    unsigned has_var_arg : 1;
    /* set for variable array types where we don't know
     * the variable size yet */
    unsigned is_variable_array : 1;
    unsigned is_variable_struct : 1;
    /* used for variable structs after we know the variable size */
    unsigned variable_size_known : 1;
    unsigned is_bitfield : 1;
    unsigned has_bitfield : 1;
    unsigned is_jitted : 1;
    unsigned is_packed : 1;
    unsigned is_unsigned : 1;
};

#define ALIGNED_DEFAULT (__alignof__(void* __attribute__((aligned))) - 1)

csymbol *cp_id_to_csym(int id);
#define ct_ffi_cs(ct) (cp_id_to_csym((ct)->ffi_cs_id))

size_t ctype_size(const struct cp_ctype* ct);
int ctype_stack_top();
int cp_ctype_init();
int cp_ctype_free();
struct cp_ctype *ctype_lookup_type(char *name);
void cp_ctype_dump_stack();
void cp_error(const char *err_msg_fmt, ...);
struct cp_ctype *cp_ctype_reg_type(char *name, struct cp_ctype *ct);

void cp_update_csym_in_ctype(struct cp_ctype *ct);
void cp_push_ctype_with_name(struct cp_ctype *ct, const char *name, int nlen);
void cp_push_ctype(struct cp_ctype *ct);
void cp_set_defined(struct cp_ctype *ct);

```

```
172 int cp_symbol_build_func(struct cp_ctype *type, const char *fname, int fn_size);
173 int cp_symbol_build_record(const char *stname, int type, int start_top);
174 int cp_symbol_build_fake_record(const char *stname, int type);
175 int cp_symbol_build_pointer(struct cp_ctype *ct);
176
177 int ffi_parse_cdef(const char *s);
178 void ffi_parse_new(const char *s, struct cp_ctype *ct);
179 int ffi_lookup_csymbol_id_by_name(const char *s);
180
181 void ffi_cparser_init(void);
182 void ffi_cparser_free(void);
183
184
185 static inline csymbol *cp_csymf_ret(csymbol_func *csf)
186 {
187     return cp_id_to_csym(csf->ret_id);
188 }
189
190 static inline csymbol *cp_csymf_arg(csymbol_func *csf, int idx)
191 {
192     return cp_id_to_csym(csf->arg_ids[idx]);
193 }
194
195
196 #else
197 static inline void ffi_cparser_init(void)
198 {
199     return;
200 }
201 static inline void ffi_cparser_free(void)
202 {
203     return;
204 }
205 #endif /* CONFIG_KTAP_FFI */
206
207
208 #endif /* __KTAP_CPARSER_H__ */
```

One Level Up Top Level

include/ktap_ffi.h - ktap

Data types defined

- csymbol
- csymbol
- csymbol_func
- csymbol_func
- csymbol_struct
- csymbol_struct
- ffi_mode
- ffi_type
- struct_member
- struct_member

Macros defined

- CSYM_NAME_MAX_LEN
- NUM_FFI_TYPE
- __KTAP_FFI_H__
- cd_csym
- cd_csym_id
- cd_int
- cd_ptr
- cd_ptr_nmemb
- cd_record
- cd_set_csym_id
- cd_struct
- cd_type
- cd_union
- csym_func
- csym_func_addr
- csym_func_arg
- csym_func_arg_ids
- csym_name

- csym_ptr_deref
- csym_ptr_deref_id
- csym_set_ptr_deref_id
- csym_struct
- csym_struct_mb
- csym_struct_mb_nr
- csym_type
- csymf_addr
- csymf_arg
- csymf_arg_id
- csymf_arg_ids
- csymf_arg_nr
- csymf_ret
- csymf_ret_id
- csymst_mb
- csymst_mb_csym
- csymst_mb_id
- csymst_mb_len
- csymst_mb_name
- csymst_mb_nr
- ffi_type_align
- ffi_type_name
- ffi_type_size
- id_to_csym
- max_csym_id

Source code

```
#ifndef __KTAP_FFI_H__
#define __KTAP_FFI_H__

#ifdef CONFIG_KTAP_FFI

#include "../include/ktap_types.h"

/*
 * Types design in FFI module
 *
 * ktap_cdata_t is an instance of csymbol, so it's a combination of csymbol
 * and it's actual data in memory.
 *
 * csymbol structs are globally unique and readonly type that represent C
 * types.  For non scalar C types like struct and function, helper structs are
 * used to store detailed information. See csymbol_func and csymbol_struct for
```

```
 * more information.
 */

typedef enum {
    /* 0 - 4 */
    FFI_VOID,
    FFI_UINT8,
    FFI_INT8,
    FFI_UINT16,
    FFI_INT16,
    /* 5 - 9 */
    FFI_UINT32,
    FFI_INT32,
    FFI_UINT64,
    FFI_INT64,
    FFI_PTR,
    /* 10 - 12 */
    FFI_FUNC,
    FFI_STRUCT,
    FFI_UNION,
    FFI_UNKNOWN,
} ffi_type;
#define NUM_FFI_TYPE ((int)FFI_UNKNOWN)


/* following struct and macros are added for C typedef
 * size and alignment calculation */
typedef struct {
    size_t size;
    size_t align;
    const char *name;
} ffi_mode;
extern const ffi_mode const ffi_type_modes[];

#define ffi_type_size(t) (ffi_type_modes[t].size)
#define ffi_type_align(t) (ffi_type_modes[t].align)
#define ffi_type_name(t) (ffi_type_modes[t].name)


/* start of csymbol definition */
#define CSYM_NAME_MAX_LEN 64

typedef struct csymbol_func {
    void *addr;        /* function address */
    csymbol_id ret_id;    /* function return type */
    int arg_nr;        /* number of arguments */
    csymbol_id *arg_ids;    /* function argument types */
    unsigned has_var_arg;    /* is this a var arg function? */
} csymbol_func;

typedef struct struct_member {
    char name[CSYM_NAME_MAX_LEN];    /* name for this struct member */
    csymbol_id id;            /* type for this struct member */
    int len;                /* length of the array, -1 for non-array */
} struct_member;

typedef struct csymbol_struct {
    int memb_nr;            /* number of members */
    struct_member *members;        /* array for each member definition */
    size_t size;            /* bytes used to store struct */
    /* alignment of the struct, 0 indicates uninitialization */
    size_t align;
} csymbol_struct;


/* wrapper struct for all C symbols */
typedef struct csymbol {
    char name[CSYM_NAME_MAX_LEN];    /* name for this symbol */
    ffi_type type;            /* type for this symbol  */
    /* following members are used only for non scalar C types */
    union {
        csymbol_id p;        /* pointer type */
        csymbol_func f;        /* C function type */
        csymbol_struct st;    /* struct/union type */
        csymbol_id td;        /* typedef type */
    } u;
```

```
} csymbol;


/* helper macors for c symbols */
#define max_csym_id(ks) (G(ks)->ffis.csym_nr)

/* lookup csymbol address by its id */
 inline csymbol *ffi_get_csym_by_id(ktap_state_t *ks, csymbol_id id);
 #define id_to_csym(ks, id) (ffi_get_csym_by_id(ks, id))

 /* helper macros for struct csymbol */
 #define csym_type(cs) ((cs)->type)
 #define csym_name(cs) ((cs)->name)

 /*
  * helper macros for pointer symbol
  */
 #define csym_ptr_deref_id(cs) ((cs)->u.p)
 #define csym_set_ptr_deref_id(cs, id) ((cs)->u.p = (id))
 /* following macro gets csymbol address */
 #define csym_ptr_deref(ks, cs) (id_to_csym(ks, csym_ptr_deref_id(cs)))

 /*
  * helper macros for function symbol
  * csym_* accepts csymbol type
  * csymf_* accepts csymbol_func type
  */
 #define csymf_addr(csf) ((csf)->addr)
 #define csymf_ret_id(csf) ((csf)->ret_id)
 #define csymf_arg_nr(csf) ((csf)->arg_nr)
 #define csymf_arg_ids(csf) ((csf)->arg_ids)
 /* get csymbol id for the nth argument */
 #define csymf_arg_id(csf, n) ((csf)->arg_ids[n])
 #define csym_func(cs) (&((cs)->u.f))
 #define csym_func_addr(cs) (csymf_addr(csym_func(cs)))
 #define csym_func_arg_ids(cs) (csymf_arg_ids(csym_func(cs)))
 /* following macros get csymbol address */
 #define csymf_ret(ks, csf) (id_to_csym(ks, csymf_ret_id(csf)))
 /* get csymbol address for the nth argument */
 #define csymf_arg(ks, csf, n) (id_to_csym(ks, csymf_arg_id(csf, n)))
 #define csym_func_arg(ks, cs, n) (csymf_arg(ks, csym_func(cs), n))

 /*
  * helper macors for struct symbol
  * csym_* accepts csymbol type
  * csymst_* accepts csymbol_struct type
  */
 #define csymst_mb_nr(csst) ((csst)->memb_nr)
 #define csym_struct(cs) (&(cs)->u.st)
 #define csym_struct_mb(cs, n) (csymst_mb(csym_struct(cs), n))
 #define csym_struct_mb_nr(cs) (csymst_mb_nr(csym_struct(cs)))
 /* following macro gets csymbol address for the nth struct member */
 #define csymst_mb(csst, n) (&(csst)->members[n])
 #define csymst_mb_name(csst, n) ((csst)->members[n].name)
 #define csymst_mb_id(csst, n) ((csst)->members[n].id)
 #define csymst_mb_len(csst, n) ((csst)->members[n].len)
 #define csymst_mb_csym(ks, csst, n) (id_to_csym(ks, (csst)->members[n].id))


 /*
  * helper macros for ktap_cdata_t type
  */
 #define cd_csym_id(cd) ((cd)->id)
 #define cd_set_csym_id(cd, id) (cd_csym_id(cd) = (id))
 #define cd_csym(ks, cd) (id_to_csym(ks, cd_csym_id(cd)))
 #define cd_type(ks, cd) (csym_type(cd_csym(ks, cd)))

 #define cd_int(cd) ((cd)->u.i)
 #define cd_ptr(cd) ((cd)->u.p.addr)
 #define cd_ptr_nmemb(cd) ((cd)->u.p.nmemb)
 #define cd_record(cd) ((cd)->u.rec)
 #define cd_struct(cd) cd_record(cd)
 #define cd_union(cd) cd_record(cd)

 #ifdef __KERNEL__
 size_t csym_size(ktap_state_t *ks, csymbol *sym);
```

```
size_t csym_align(ktap_state_t *ks, csymbol *sym);
size_t csym_record_mb_offset_by_name(ktap_state_t *ks,
        csymbol *cs, const char *name);
struct member *csymst_mb_by_name(ktap_state_t *ks,
        csymbol_struct *csst, const char *name);

void ffi_free_symbols(ktap_state_t *ks);
csymbol_id ffi_get_csym_id(ktap_state_t *ks, char *name);

ktap_cdata_t *kp_cdata_new(ktap_state_t *ks, csymbol_id id);
ktap_cdata_t *kp_cdata_new_ptr(ktap_state_t *ks, void *addr,
        int nmemb, csymbol_id id, int to_allocate);
ktap_cdata_t *kp_cdata_new_record(ktap_state_t *ks, void *val, csymbol_id id);
void kp_cdata_dump(ktap_state_t *ks, ktap_cdata_t *cd);
int kp_cdata_type_match(ktap_state_t *ks, csymbol *cs, ktap_val_t *val);
void kp_cdata_unpack(ktap_state_t *ks, char *dst, csymbol *cs, ktap_val_t *val);
void kp_cdata_ptr_set(ktap_state_t *ks, ktap_cdata_t *cd,
        ktap_val_t *key, ktap_val_t *val);
void kp_cdata_ptr_get(ktap_state_t *ks, ktap_cdata_t *cd,
        ktap_val_t *key, ktap_val_t *val);
void kp_cdata_record_set(ktap_state_t *ks, ktap_cdata_t *cd,
         ktap_val_t *key, ktap_val_t *val);
void kp_cdata_record_get(ktap_state_t *ks, ktap_cdata_t *cd,
         ktap_val_t *key, ktap_val_t *val);

int ffi_call(ktap_state_t *ks, csymbol_func *cf);

#endif /* for CONFIG_FFI_KTAP */
#endif /* for __KERNEL__ */
#endif /* __KTAP_FFI_H__ */
```

One Level Up Top Level

runtime/ffi/ffi_type.c - ktap

Global variables defined

- ffi_type_modes

Macros defined

- CTYPE_MODE
- CTYPE_MODE_HELPER
- CTYPE_MODE_NAME

Source code

```
1  #include "../../include/ktap_ffi.h"
2  #ifdef __KERNEL__
3  #include <linux/types.h>
4  #else
5  #include <stdint.h>
6  #include <stddef.h>
7  #endif
8
9  #define CTYPE_MODE_HELPER(name, type)    \
10  struct _##name##_align {        \
11      type t1;            \
12      char c;                \
13      type t2;            \
14  };
15
16  #define CTYPE_MODE(name)                \
17  {                           \
18      offsetof(struct _##name##_align, c),        \
19      offsetof(struct _##name##_align, t2) -        \
20          offsetof(struct _##name##_align, c),    \
21      #name                   \
22  }
23
24  #define CTYPE_MODE_NAME(name) _##name##_mode
25
26  /* ffi_ctype_mode should be corresponded to ffi_ctype */
27  CTYPE_MODE_HELPER(uint8, uint8_t);
28  CTYPE_MODE_HELPER(int8, int8_t);
29  CTYPE_MODE_HELPER(uint16, uint16_t);
30  CTYPE_MODE_HELPER(int16, int16_t);
31  CTYPE_MODE_HELPER(uint32, uint32_t);
32  CTYPE_MODE_HELPER(int32, int32_t);
33  CTYPE_MODE_HELPER(uint64, uint64_t);
34  CTYPE_MODE_HELPER(int64, int64_t);
35  CTYPE_MODE_HELPER(pointer, void*);
36
37  const ffi_mode ffi_type_modes[NUM_FFI_TYPE+1] = {
38      {0, 1, "void"},
39      CTYPE_MODE(uint8),
40      CTYPE_MODE(int8),
41      CTYPE_MODE(uint16),
42      CTYPE_MODE(int16),
43      CTYPE_MODE(uint32),
44      CTYPE_MODE(int32),
45      CTYPE_MODE(uint64),
46      CTYPE_MODE(int64),
47      CTYPE_MODE(pointer),
48      {0, 1, "function"},
49      {0, 1, "struct"},
50      {0, 1, "union"},
51      {0, 1, "unknown"},
52  };
```

runtime/ffi/ - ktap

- call_x86_64.S
- cdata.c
- ffi_call.c
- ffi_symbol.c
- ffi_type.c
- ffi_util.c

runtime/ - ktap

- amalg.c
- ffi/
- kp_bcread.c
- kp_bcread.h
- kp_events.c
- kp_events.h
- kp_mempool.c
- kp_mempool.h
- kp_obj.c
- kp_obj.h
- kp_str.c
- kp_str.h
- kp_tab.c
- kp_tab.h
- kp_transport.c
- kp_transport.h
- kp_vm.c
- kp_vm.h
- ktap.c
- ktap.h
- lib_ansi.c
- lib_base.c
- lib_ffi.c
- lib_kdebug.c
- lib_net.c
- lib_table.c
- lib_timer.c

runtime/amalg.c - ktap

```c
/*
 * kp_amalg.c - ktapvm kernel module amalgamation.
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "ktap.c"
#include "kp_obj.c"
#include "kp_bcread.c"
#include "kp_str.c"
#include "kp_mempool.c"
#include "kp_tab.c"
#include "kp_transport.c"
#include "kp_vm.c"
#include "kp_events.c"
#include "lib_base.c"
#include "lib_ansi.c"
#include "lib_kdebug.c"
#include "lib_timer.c"
#include "lib_table.c"
#include "lib_net.c"

#ifdef CONFIG_KTAP_FFI
#include "ffi/ffi_call.c"
#include "ffi/ffi_type.c"
#include "ffi/ffi_symbol.c"
#include "ffi/cdata.c"
#include "ffi/ffi_util.c"
#include "lib_ffi.c"
#endif
```

runtime/kp_bcread.c - ktap

Data types defined

- BCReadCtx
- BCReadCtx

Functions defined

- bcread_block
- bcread_byte
- bcread_bytecode
- bcread_dbg
- bcread_error
- bcread_header
- bcread_kgc
- bcread_knum
- bcread_ktab
- bcread_ktabk
- bcread_mem
- bcread_proto
- bcread_uint32
- bcread_uv
- bcread_varinfo
- bswap
- kp_bcread

Macros defined

- bcread_flags
- bcread_oldtop
- bcread_savetop
- bcread_swap

Source code

```
1  /*
2  * Bytecode reader.
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
```

```
 * Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * Adapted from luajit and lua interpreter.
 * Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "../include/ktap_types.h"
#include "../include/ktap_bc.h"
#include "../include/ktap_err.h"
#include "ktap.h"
#include "kp_obj.h"
#include "kp_str.h"
#include "kp_tab.h"


/* Context for bytecode reader. */
typedef struct BCReadCtx {
    ktap_state_t *ks;
    int flags;
    char *start;
    char *p;
    char *pe;
    ktap_str_t *chunkname;
    ktap_val_t *savetop;
} BCReadCtx;


#define bcread_flags(ctx)    (ctx)->flags
#define bcread_swap(ctx) \
    ((bcread_flags(ctx) & BCDUMP_F_BE) != KP_BE*BCDUMP_F_BE)
#define bcread_oldtop(ctx)        (ctx)->savetop
#define bcread_savetop(ctx)    (ctx)->savetop = (ctx)->ks->top;

static inline uint32_t bswap(uint32_t x)
{
    return (uint32_t)__builtin_bswap32((int32_t)x);
}

/* -- Input buffer handling ---------------------------------------------- */

/* Throw reader error. */
static void bcread_error(BCReadCtx *ctx, ErrMsg em)
{
    kp_error(ctx->ks, "%s\n", err2msg(em));
}

/* Return memory block from buffer. */
static inline uint8_t *bcread_mem(BCReadCtx *ctx, int len)
{
    uint8_t *p = (uint8_t *)ctx->p;
    ctx->p += len;
    kp_assert(ctx->p <= ctx->pe);
    return p;
}

/* Copy memory block from buffer. */
static void bcread_block(BCReadCtx *ctx, void *q, int len)
{
    memcpy(q, bcread_mem(ctx, len), len);
}

/* Read byte from buffer. */
```

```
static inline uint32_t bcread_byte(BCReadCtx *ctx)
{
    kp_assert(ctx->p < ctx->pe);
    return (uint32_t)(uint8_t)*ctx->p++;
}

/* Read ULEB128 value from buffer. */
static inline uint32_t bcread_uint32(BCReadCtx *ctx)
{
    uint32_t v;
    bcread_block(ctx, &v, sizeof(uint32_t));
    kp_assert(ctx->p <= ctx->pe);
    return v;
}

/* -- Bytecode reader ------------------------------------------------------ */

/* Read debug info of a prototype. */
static void bcread_dbg(BCReadCtx *ctx, ktap_proto_t *pt, int sizedbg)
{
    void *lineinfo = (void *)proto_lineinfo(pt);

    bcread_block(ctx, lineinfo, sizedbg);
    /* Swap lineinfo if the endianess differs. */
    if (bcread_swap(ctx) && pt->numline >= 256) {
        int i, n = pt->sizebc-1;
        if (pt->numline < 65536) {
            uint16_t *p = (uint16_t *)lineinfo;
            for (i = 0; i < n; i++)
                p[i] = (uint16_t)((p[i] >> 8)|(p[i] << 8));
        } else {
            uint32_t *p = (uint32_t *)lineinfo;
            for (i = 0; i < n; i++)
                p[i] = bswap(p[i]);
        }
    }
}

/* Find pointer to varinfo. */
static const void *bcread_varinfo(ktap_proto_t *pt)
{
    const uint8_t *p = proto_uvinfo(pt);
    int n = pt->sizeuv;
    if (n)
        while (*p++ || --n) ;
    return p;
}

/* Read a single constant key/value of a template table. */
static int bcread_ktabk(BCReadCtx *ctx, ktap_val_t *o)
{
    int tp = bcread_uint32(ctx);
    if (tp >= BCDUMP_KTAB_STR) {
        int len = tp - BCDUMP_KTAB_STR;
        const char *p = (const char *)bcread_mem(ctx, len);
        ktap_str_t *ts = kp_str_new(ctx->ks, p, len);
        if (unlikely(!ts))
            return -ENOMEM;

        set_string(o, ts);
    } else if (tp == BCDUMP_KTAB_NUM) {
        set_number(o, *(ktap_number *)bcread_mem(ctx,
                    sizeof(ktap_number)));
    } else {
         kp_assert(tp <= BCDUMP_KTAB_TRUE);
        setitype(o, ~tp);
    }
    return 0;
}

/* Read a template table. */
static ktap_tab_t *bcread_ktab(BCReadCtx *ctx)
{
    int narray = bcread_uint32(ctx);
    int nhash = bcread_uint32(ctx);

```

```
    ktap_tab_t *t = kp_tab_new(ctx->ks, narray, hsize2hbits(nhash));
    if (!t)
        return NULL;

    if (narray) {  /* Read array entries. */
        int i;
        ktap_val_t *o = t->array;
        for (i = 0; i < narray; i++, o++)
            if (bcread_ktabk(ctx, o))
                return NULL;
    }
    if (nhash) {  /* Read hash entries. */
        int i;
        for (i = 0; i < nhash; i++) {
            ktap_val_t key;
            ktap_val_t val;
            if (bcread_ktabk(ctx, &key))
                return NULL;
            kp_assert(!is_nil(&key));
            if (bcread_ktabk(ctx, &val))
                return NULL;
            kp_tab_set(ctx->ks, t, &key, &val);
        }
    }
    return t;
}

/* Read GC constants(string, table, child proto) of a prototype. */
static int bcread_kgc(BCReadCtx *ctx, ktap_proto_t *pt, int sizekgc)
{
    ktap_obj_t **kr = (ktap_obj_t **)pt->k - (ptrdiff_t)sizekgc;
    int i;

    for (i = 0; i < sizekgc; i++, kr++) {
        int tp = bcread_uint32(ctx);
        if (tp >= BCDUMP_KGC_STR) {
            int len = tp - BCDUMP_KGC_STR;
            const char *p = (const char *)bcread_mem(ctx, len);
            *kr =(ktap_obj_t *)kp_str_new(ctx->ks, p, len);
            if (unlikely(!*kr))
                return -1;
        } else if (tp == BCDUMP_KGC_TAB) {
            *kr = (ktap_obj_t *)bcread_ktab(ctx);
            if (unlikely(!*kr))
                return -1;
        } else if (tp == BCDUMP_KGC_CHILD){
            ktap_state_t *ks = ctx->ks;
            if (ks->top <= bcread_oldtop(ctx)) {
                bcread_error(ctx, KP_ERR_BCBAD);
                return -1;
            }
            ks->top--;
            *kr = (ktap_obj_t *)ptvalue(ks->top);
        } else {
            bcread_error(ctx, KP_ERR_BCBAD);
            return -1;
        }
    }

    return 0;
}

/* Read number constants of a prototype. */
static void bcread_knum(BCReadCtx *ctx, ktap_proto_t *pt, int sizekn)
{
    int i;
    ktap_val_t *o = pt->k;

    for (i = 0; i < sizekn; i++, o++) {
        set_number(o, *(ktap_number *)bcread_mem(ctx,
                    sizeof(ktap_number)));
    }
}

/* Read bytecode instructions. */
static void bcread_bytecode(BCReadCtx *ctx, ktap_proto_t *pt, int sizebc)
```

```
{
    BCIns *bc = proto_bc(pt);
    bc[0] = BCINS_AD((pt->flags & PROTO_VARARG) ? BC_FUNCV : BC_FUNCF,
              pt->framesize, 0);
    bcread_block(ctx, bc+1, (sizebc-1)*(int)sizeof(BCIns));
    /* Swap bytecode instructions if the endianess differs. */
    if (bcread_swap(ctx)) {
        int i;
        for (i = 1; i < sizebc; i++) bc[i] = bswap(bc[i]);
    }
}

/* Read upvalue refs. */
static void bcread_uv(BCReadCtx *ctx, ktap_proto_t *pt, int sizeuv)
{
    if (sizeuv) {
        uint16_t *uv = proto_uv(pt);
        bcread_block(ctx, uv, sizeuv*2);
        /* Swap upvalue refs if the endianess differs. */
        if (bcread_swap(ctx)) {
            int i;
            for (i = 0; i < sizeuv; i++)
                uv[i] = (uint16_t)((uv[i] >> 8)|(uv[i] << 8));
        }
    }
}

/* Read a prototype. */
static ktap_proto_t *bcread_proto(BCReadCtx *ctx)
{
    ktap_proto_t *pt;
    int framesize, numparams, flags;
    int sizeuv, sizekgc, sizekn, sizebc, sizept;
    int ofsk, ofsuv, ofsdbg;
    int sizedbg = 0;
    BCLine firstline = 0, numline = 0;

    /* Read prototype header. */
    flags = bcread_byte(ctx);
    numparams = bcread_byte(ctx);
    framesize = bcread_byte(ctx);
    sizeuv = bcread_byte(ctx);
    sizekgc = bcread_uint32(ctx);
    sizekn = bcread_uint32(ctx);
    sizebc = bcread_uint32(ctx) + 1;
    if (!(bcread_flags(ctx) & BCDUMP_F_STRIP)) {
        sizedbg = bcread_uint32(ctx);
        if (sizedbg) {
            firstline = bcread_uint32(ctx);
            numline = bcread_uint32(ctx);
        }
    }

    /* Calculate total size of prototype including all colocated arrays. */
    sizept = (int)sizeof(ktap_proto_t) + sizebc * (int)sizeof(BCIns) +
             sizekgc * (int)sizeof(ktap_obj_t *);
    sizept = (sizept + (int)sizeof(ktap_val_t)-1) &
             ~((int)sizeof(ktap_val_t)-1);
    ofsk = sizept; sizept += sizekn*(int)sizeof(ktap_val_t);
    ofsuv = sizept; sizept += ((sizeuv+1)&~1)*2;
    ofsdbg = sizept; sizept += sizedbg;

    /* Allocate prototype object and initialize its fields. */
    pt = (ktap_proto_t *)kp_obj_new(ctx->ks, (int)sizept);
    pt->gct = ~KTAP_TPROTO;
    pt->numparams = (uint8_t)numparams;
    pt->framesize = (uint8_t)framesize;
    pt->sizebc = sizebc;
    pt->k = (char *)pt + ofsk;
    pt->uv = (char *)pt + ofsuv;
    pt->sizekgc = 0;  /* Set to zero until fully initialized. */
    pt->sizekn = sizekn;
    pt->sizept = sizept;
    pt->sizeuv = (uint8_t)sizeuv;
    pt->flags = (uint8_t)flags;
    pt->chunkname = ctx->chunkname;
```

```
310
311     /* Close potentially uninitialized gap between bc and kgc. */
312     *(uint32_t *)((char *)pt + ofsk - sizeof(ktap_obj_t *)*(sizekgc+1))
313                                     = 0;
314
315     /* Read bytecode instructions and upvalue refs. */
316     bcread_bytecode(ctx, pt, sizebc);
317     bcread_uv(ctx, pt, sizeuv);
318
319     /* Read constants. */
320     if (bcread_kgc(ctx, pt, sizekgc))
321         return NULL;
322     pt->sizekgc = sizekgc;
323     bcread_knum(ctx, pt, sizekn);
324
325     /* Read and initialize debug info. */
326     pt->firstline = firstline;
327     pt->numline = numline;
328     if (sizedbg) {
329         int sizeli = (sizebc-1) << (numline < 256 ? 0 :
330                     numline < 65536 ? 1 : 2);
331         pt->lineinfo = (char *)pt + ofsdbg;
332         pt->uvinfo = (char *)pt + ofsdbg + sizeli;
333         bcread_dbg(ctx, pt, sizedbg);
334         pt->varinfo = (void *)bcread_varinfo(pt);
335     } else {
336         pt->lineinfo = NULL;
337         pt->uvinfo = NULL;
338         pt->varinfo = NULL;
339     }
340     return pt;
341 }
342
343 /* Read and check header of bytecode dump. */
344 static int bcread_header(BCReadCtx *ctx)
345 {
346     uint32_t flags;
347
348     if (bcread_byte(ctx) != BCDUMP_HEAD1 ||
349         bcread_byte(ctx) != BCDUMP_HEAD2 ||
350         bcread_byte(ctx) != BCDUMP_HEAD3 ||
351         bcread_byte(ctx) != BCDUMP_VERSION)
352         return -1;
353
354     bcread_flags(ctx) = flags = bcread_byte(ctx);
355
356     if ((flags & ~(BCDUMP_F_KNOWN)) != 0)
357         return -1;
358
359     if ((flags & BCDUMP_F_FFI)) {
360         return -1;
361     }
362
363     if ((flags & BCDUMP_F_STRIP)) {
364         ctx->chunkname = kp_str_newz(ctx->ks, "striped");
365     } else {
366         int len = bcread_uint32(ctx);
367         ctx->chunkname = kp_str_new(ctx->ks,
368                 (const char *)bcread_mem(ctx, len), len);
369     }
370
371     if (unlikely(!ctx->chunkname))
372         return -1;
373
374     return 0;
375 }
376
377 /* Read a bytecode dump. */
378 ktap_proto_t *kp_bcread(ktap_state_t *ks, unsigned char *buff, int len)
379 {
380     BCReadCtx ctx;
381
382     ctx.ks = ks;
383     ctx.p = buff;
384     ctx.pe = buff + len;
385
```

```
386     ctx.start = buff;
387
388     bcread_savetop(&ctx);
389     /* Check for a valid bytecode dump header. */
390     if (bcread_header(&ctx)) {
391         bcread_error(&ctx, KP_ERR_BCFMT);
392         return NULL;
393     }
394
395     for (;;) {  /* Process all prototypes in the bytecode dump. */
396         ktap_proto_t *pt;
397         int len;
398         const char *startp;
399         /* Read length. */
400         if (ctx.p < ctx.pe && ctx.p[0] == 0) {  /* Shortcut EOF. */
401             ctx.p++;
402             break;
403         }
404         len = bcread_uint32(&ctx);
405         if (!len)
406             break;  /* EOF */
407         startp = ctx.p;
408         pt = bcread_proto(&ctx);
409         if (!pt)
410             return NULL;
411         if (ctx.p != startp + len) {
412             bcread_error(&ctx, KP_ERR_BCBAD);
413             return NULL;
414         }
415         set_proto(ks->top, pt);
416         incr_top(ks);
417     }
418     if ((int32_t)(2*(uint32_t)(ctx.pe - ctx.p)) > 0 ||
419             ks->top-1 != bcread_oldtop(&ctx)) {
420         bcread_error(&ctx, KP_ERR_BCBAD);
421         return NULL;
422     }
423
424     /* Pop off last prototype. */
425     ks->top--;
426     return ptvalue(ks->top);
427 }
428
```

One Level Up Top Level

include/ktap_err.h - ktap

Data types defined

- ErrMsg

Macros defined

- ERRDEF
- __KTAP_ERR_H__

Source code

```
1  #ifndef __KTAP_ERR_H__
2  #define __KTAP_ERR_H__
3
4  typedef enum {
5  #define ERRDEF(name, msg) \
6      KP_ERR_##name, KP_ERR_##name##_ = KP_ERR_##name + sizeof(msg)-1,
7  #include "ktap_errmsg.h"
8      KP_ERR__MAX
9  } ErrMsg;
10
11  #endif
```

runtime/ktap.h - ktap

Data types defined

- ktap_libfunc
- ktap_libfunc_t

Functions defined

- get_recursion_context
- kp_this_cpu_print_buffer
- kp_this_cpu_state
- kp_this_cpu_temp_buffer
- put_recursion_context
- trace_get_context_bit

Macros defined

- SPRINT_SYMBOL
- SPRINT_SYMBOL
- __KTAP_H__
- err2msg
- kp_arg
- kp_arg_check
- kp_arg_checkfunction
- kp_arg_checknumber
- kp_arg_checkoptnumber
- kp_arg_checkstring
- kp_arg_nr
- kp_error
- kp_puts
- kp_verbose_printf
- raw_cpu_ptr

Source code

```
1  #ifndef __KTAP_H__
2  #define __KTAP_H__
3
4  #include <linux/version.h>
5  #include <linux/hardirq.h>
```

```c
#include <linux/trace_seq.h>

#ifndef raw_cpu_ptr
#define raw_cpu_ptr __this_cpu_ptr
 #endif

 /* for built-in library C function register */
 typedef struct ktap_libfunc {
         const char *name; /* function name */
         ktap_cfunction func; /* function pointer */
 } ktap_libfunc_t;

 long gettimeofday_ns(void); /* common helper function */
 int kp_lib_init_base(ktap_state_t *ks);
 int kp_lib_init_kdebug(ktap_state_t *ks);
 int kp_lib_init_timer(ktap_state_t *ks);
 int kp_lib_init_table(ktap_state_t *ks);
 int kp_lib_init_ansi(ktap_state_t *ks);
 int kp_lib_init_net(ktap_state_t *ks);
 #ifdef CONFIG_KTAP_FFI
 int kp_lib_init_ffi(ktap_state_t *ks);
 #endif

 void kp_exit_timers(ktap_state_t *ks);
 void kp_freeupval(ktap_state_t *ks, ktap_upval_t *uv);

 extern int (*kp_ftrace_profile_set_filter)(struct perf_event *event,
                       int event_id,
                       const char *filter_str);

 extern struct syscall_metadata **syscalls_metadata;

 /* get from kernel/trace/trace.h */
 static __always_inline int trace_get_context_bit(void)
 {
     int bit;

     if (in_interrupt()) {
         if (in_nmi())
             bit = 0;
         else if (in_irq())
             bit = 1;
         else
             bit = 2;
     } else
         bit = 3;

     return bit;
 }

 static __always_inline int get_recursion_context(ktap_state_t *ks)
 {
     int rctx = trace_get_context_bit();
     int *val = raw_cpu_ptr(G(ks)->recursion_context[rctx]);

     if (*val)
         return -1;

     *val = true;
     return rctx;
 }

 static inline void put_recursion_context(ktap_state_t *ks, int rctx)
 {
     int *val = raw_cpu_ptr(G(ks)->recursion_context[rctx]);
     *val = false;
 }

 static inline void *kp_this_cpu_state(ktap_state_t *ks, int rctx)
 {
     return this_cpu_ptr(G(ks)->percpu_state[rctx]);
 }

 static inline void *kp_this_cpu_print_buffer(ktap_state_t *ks)
 {
     return this_cpu_ptr(G(ks)->percpu_print_buffer[trace_get_context_bit()]);
```

```
}

static inline void *kp_this_cpu_temp_buffer(ktap_state_t *ks)
{
    return this_cpu_ptr(G(ks)->percpu_temp_buffer[trace_get_context_bit()]);
}

#define kp_verbose_printf(ks, ...) \
    if (G(ks)->parm->verbose)    \
        kp_printf(ks, "[verbose] "__VA_ARGS__);

/* argument operation macro */
#define kp_arg(ks, idx)    ((ks)->func + (idx))
#define kp_arg_nr(ks)    ((int)(ks->top - (ks->func + 1)))

#define kp_arg_check(ks, idx, type)               \
    do {                              \
        if (unlikely(itype(kp_arg(ks, idx)) != type)) {    \
            kp_error(ks, "wrong type of argument %d\n", idx);\
            return -1;                \
        }                         \
    } while (0)

#define kp_arg_checkstring(ks, idx)               \
    ({                                \
        ktap_val_t *o = kp_arg(ks, idx);        \
        if (unlikely(!is_string(o))) {           \
            kp_error(ks, "wrong type of argument %d\n", idx); \
            return -1;                \
        }                         \
        svalue(o);                    \
    })

#define kp_arg_checkfunction(ks, idx)               \
    ({                                \
        ktap_val_t *o = kp_arg(ks, idx);        \
        if (unlikely(!is_function(o))) {            \
            kp_error(ks, "wrong type of argument %d\n", idx); \
            return -1;                \
        }                         \
        clvalue(o);                    \
    })

#define kp_arg_checknumber(ks, idx)               \
    ({                                \
        ktap_val_t *o = kp_arg(ks, idx);        \
        if (unlikely(!is_number(o))) {            \
            kp_error(ks, "wrong type of argument %d\n", idx); \
            return -1;                \
        }                         \
        nvalue(o);                    \
    })

#define kp_arg_checkoptnumber(ks, idx, def)           \
    ({                                \
        ktap_number n;                    \
        if (idx > kp_arg_nr(ks)) {                \
            n = def;                \
        } else {                    \
            ktap_val_t *o = kp_arg(ks, idx);    \
            if (unlikely(!is_number(o))) {        \
                kp_error(ks, "wrong type of argument %d\n", \
                         idx);        \
                return -1;            \
            }                     \
            n = nvalue(o);                \
        }                             \
        n;                             \
    })

#define kp_error(ks, args...)           \
    do {                    \
        kp_printf(ks, "error: "args);    \
        kp_vm_try_to_exit(ks);        \
        G(ks)->state = KTAP_ERROR;    \
    } while(0)
```

```
158
159
160 /* TODO: this code need to cleanup */
161 #if LINUX_VERSION_CODE > KERNEL_VERSION(3, 5, 0)
162 #define SPRINT_SYMBOL    sprint_symbol_no_offset
163 #else
164 #define SPRINT_SYMBOL    sprint_symbol
165 #endif
166
167 extern int kp_max_loop_count;
168
169 void kp_printf(ktap_state_t *ks, const char *fmt, ...);
170 void __kp_puts(ktap_state_t *ks, const char *str);
171 void __kp_bputs(ktap_state_t *ks, const char *str);
172
173 #define kp_puts(ks, str) ({                        \
174     static const char *trace_printk_fmt               \
175         __attribute__((section("__trace_printk_fmt"))) =    \
176         __builtin_constant_p(str) ? str : NULL;            \
177                                   \
178     if (__builtin_constant_p(str))                    \
179         __kp_bputs(ks, trace_printk_fmt);        \
180     else                                  \
181         __kp_puts(ks, str);        \
182 })
183
184 #define err2msg(em)     (kp_err_allmsg+(int)(em))
185 extern const char *kp_err_allmsg;
186
187 #endif /* __KTAP_H__ */
188
```

One Level Up Top Level

runtime/ktap.c - ktap

Global variables defined

- exit_ktap
- init_ktap
- kp_dir_dentry
- kp_ftrace_profile_set_filter
- kp_max_loop_count
- ktap_fops
- ktapvm_fops
- syscalls_metadata

Functions defined

- exit_ktap
- gettimeofday_ns
- init_dummy_kernel_functions
- init_ktap
- ktap_ioctl
- ktap_main
- ktapvm_ioctl
- load_trunk
- print_version

Source code

```
1  /*
2  * ktap.c - ktapvm kernel module main entry
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10  * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22 /*
23 * this file is the first file to be compile, add CONFIG_ checking in here.
```

```
 * See Requirements in doc/tutorial.md
 */

#include <linux/version.h>
#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 1, 0)
#error "Currently ktap don't support kernel older than 3.1"
#endif

#if !CONFIG_EVENT_TRACING
#error "Please enable CONFIG_EVENT_TRACING before compile ktap"
#endif

#if !CONFIG_PERF_EVENTS
#error "Please enable CONFIG_PERF_EVENTS before compile ktap"
#endif

#include <linux/module.h>
#include <linux/errno.h>
#include <linux/file.h>
#include <linux/slab.h>
#include <linux/fcntl.h>
#include <linux/sched.h>
#include <linux/poll.h>
#include <linux/anon_inodes.h>
#include <linux/debugfs.h>
#include <linux/vmalloc.h>
#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_bcread.h"
#include "kp_vm.h"

/* common helper function */
long gettimeofday_ns(void)
{
    struct timespec now;

    getnstimeofday(&now);
    return now.tv_sec * NSEC_PER_SEC + now.tv_nsec;
}

static int load_trunk(ktap_option_t *parm, unsigned long **buff)
{
    int ret;
    unsigned long *vmstart;

    vmstart = vmalloc(parm->trunk_len);
    if (!vmstart)
        return -ENOMEM;

    ret = copy_from_user(vmstart, (void __user *)parm->trunk,
                parm->trunk_len);
    if (ret < 0) {
        vfree(vmstart);
        return -EFAULT;
    }

    *buff = vmstart;
    return 0;
}

static struct dentry *kp_dir_dentry;

/* Ktap Main Entry */
static int ktap_main(struct file *file, ktap_option_t *parm)
{
    unsigned long *buff = NULL;
    ktap_state_t *ks;
    ktap_proto_t *pt;
    long start_time, delta_time;
    int ret;

    start_time = gettimeofday_ns();

    ks = kp_vm_new_state(parm, kp_dir_dentry);
    if (unlikely(!ks))
        return -ENOEXEC;
```

```
    file->private_data = ks;

    ret = load_trunk(parm, &buff);
    if (ret) {
        kp_error(ks, "cannot load file\n");
        goto out;
    }

    pt = kp_bcread(ks, (unsigned char *)buff, parm->trunk_len);

    vfree(buff);

    if (pt) {
        /* validate byte code */
        if (kp_vm_validate_code(ks, pt, ks->stack))
            goto out;

        delta_time = (gettimeofday_ns() - start_time) / NSEC_PER_USEC;
        kp_verbose_printf(ks, "booting time: %d (us)\n", delta_time);

        /* enter vm */
        kp_vm_call_proto(ks, pt);
    }

out:
    kp_vm_exit(ks);
    return ret;
}


static void print_version(void)
{
}

static long ktap_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    ktap_option_t parm;
    int ret;

    switch (cmd) {
    case KTAP_CMD_IOC_VERSION:
        print_version();
        return 0;
    case KTAP_CMD_IOC_RUN:
        /*
         * must be root to run ktap script (at least for now)
         *
         * TODO: check perf_paranoid sysctl and allow non-root user
         * to use ktap for tracing process(like uprobe) ?
         */
        if (!capable(CAP_SYS_ADMIN))
            return -EACCES;

        ret = copy_from_user(&parm, (void __user *)arg,
                     sizeof(ktap_option_t));
        if (ret < 0)
            return -EFAULT;

        return ktap_main(file, &parm);
    default:
        return -EINVAL;
    };

        return 0;
}

static const struct file_operations ktap_fops = {
    .llseek                 = no_llseek,
    .unlocked_ioctl         = ktap_ioctl,
};

static long ktapvm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    int new_fd, err;
    struct file *new_file;
```

```
176
177     new_fd = get_unused_fd();
178     if (new_fd < 0)
179         return new_fd;
180
181     new_file = anon_inode_getfile("[ktap]", &ktap_fops, NULL, O_RDWR);
182     if (IS_ERR(new_file)) {
183         err = PTR_ERR(new_file);
184         put_unused_fd(new_fd);
185         return err;
186     }
187
188     file->private_data = NULL;
189     fd_install(new_fd, new_file);
190     return new_fd;
191 }
192
193 static const struct file_operations ktapvm_fops = {
194     .owner  = THIS_MODULE,
195     .unlocked_ioctl         = ktapvm_ioctl,
196 };
197
198 int (*kp_ftrace_profile_set_filter)(struct perf_event *event, int event_id,
199                     const char *filter_str);
200
201 struct syscall_metadata **syscalls_metadata;
202
203 /*TODO: kill this function in future */
204 static int __init init_dummy_kernel_functions(void)
205 {
206     unsigned long *addr;
207
208     /*
209      * ktap need symbol ftrace_profile_set_filter to set event filter,
210      * export it in future.
211      */
212 #ifdef CONFIG_PPC64
213     kp_ftrace_profile_set_filter =
214         (void *)kallsyms_lookup_name(".ftrace_profile_set_filter");
215 #else
216     kp_ftrace_profile_set_filter =
217         (void *)kallsyms_lookup_name("ftrace_profile_set_filter");
218 #endif
219     if (!kp_ftrace_profile_set_filter) {
220         pr_err("ktap: cannot lookup ftrace_profile_set_filter "
221             "in kallsyms\n");
222         return -1;
223     }
224
225     /* use syscalls_metadata for syscall event handling */
226     addr = (void *)kallsyms_lookup_name("syscalls_metadata");
227     if (!addr) {
228         pr_err("ktap: cannot lookup syscalls_metadata in kallsyms\n");
229         return -1;
230     }
231
232     syscalls_metadata = (struct syscall_metadata **)*addr;
233     return 0;
234 }
235
236 static int __init init_ktap(void)
237 {
238     struct dentry *ktapvm_dentry;
239
240     if (init_dummy_kernel_functions())
241         return -1;
242
243     kp_dir_dentry = debugfs_create_dir("ktap", NULL);
244     if (!kp_dir_dentry) {
245         pr_err("ktap: debugfs_create_dir failed\n");
246         return -1;
247     }
248
249     ktapvm_dentry = debugfs_create_file("ktapvm", 0444, kp_dir_dentry, NULL,
250                         &ktapvm_fops);
251
```

```
252     if (!ktapvm_dentry) {
253         pr_err("ktapvm: cannot create ktapvm file\n");
254         debugfs_remove_recursive(kp_dir_dentry);
255         return -1;
256     }
257 
258     return 0;
259 }
260 
261 static void __exit exit_ktap(void)
262 {
263     debugfs_remove_recursive(kp_dir_dentry);
264 }
265 
266 module_init(init_ktap);
267 module_exit(exit_ktap);
268 
269 MODULE_AUTHOR("Jovi Zhangwei <jovi.zhangwei@gmail.com>");
270 MODULE_DESCRIPTION("ktap");
271 MODULE_LICENSE("GPL");
272 
273 int kp_max_loop_count = 100000;
274 module_param_named(max_loop_count, kp_max_loop_count, int, S_IRUGO | S_IWUSR);
275 MODULE_PARM_DESC(max_loop_count, "max loop execution count");
276 
```

One Level Up Top Level

test/benchmark/sembench.c - ktap

Global variables defined

- all_done
- allops
- futex_sem_ops
- ipc_sem_ops
- max_burns
- min_burns
- nanosleep_sem_ops
- num_threads
- semid_lookup
- thread_count
- timeout_test
- total_burns
- workers_started
- worklist
- worklist_mutex

Data types defined

- lockinfo
- sem_operations
- sem_wakeup_info

Functions defined

- cleanup_futex_sems
- cleanup_ipc_sems
- cleanup_nanosleep_sems
- futex
- futex_wake_some
- ipc_wake_some
- main
- nanosleep_wake_some
- print_usage

- setup_futex_sems
- setup_ipc_sems
- setup_nanosleep_sems
- smp_mb
- wait_futex_sem
- wait_ipc_sem
- wait_nanosleep_sem
- worker
- worklist_add
- worklist_rm

Macros defined

- FUTEX_CMP_REQUEUE
- FUTEX_FD
- FUTEX_REQUEUE
- FUTEX_WAIT
- FUTEX_WAKE
- FUTEX_WAKE_OP
- NUM_OPERATIONS
- SEMS_PERID
- VERSION
- _GNU_SOURCE
- _POSIX_C_SOURCE

Source code

```c
/*
 * copyright Oracle 2007.  Licensed under GPLv2
 * To compile: gcc -Wall -o sembench sembench.c -lpthread
 *
 * usage: sembench -t thread count -w wakenum -r runtime -o op
 * op can be: 0 (ipc sem) 1 (nanosleep) 2 (futexes)
 *
 * example:
 *    sembench -t 1024 -w 512 -r 60 -o 2
 * runs 1024 threads, waking up 512 at a time, running for 60 seconds using
 * futex locking.
 *
 */
#define  _GNU_SOURCE
#define _POSIX_C_SOURCE 199309
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/ipc.h>
```

```c
#include <sys/types.h>
#include <sys/mman.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <sys/syscall.h>
#include <errno.h>

#define VERSION "0.2"

/* futexes have been around since 2.5.something, but it still seems I
* need to make my own syscall.  Sigh.
*/
#define FUTEX_WAIT              0
#define FUTEX_WAKE              1
#define FUTEX_FD                2
#define FUTEX_REQUEUE           3
#define FUTEX_CMP_REQUEUE       4
#define FUTEX_WAKE_OP           5
static inline int futex (int *uaddr, int op, int val,
             const struct timespec *timeout,
             int *uaddr2, int val3)
{
    return syscall(__NR_futex, uaddr, op, val, timeout, uaddr2, val3);
}

static void smp_mb(void)
{
    __sync_synchronize();
}

static int all_done = 0;
static int timeout_test = 0;

#define SEMS_PERID 250

struct sem_operations;

struct lockinfo {
    unsigned long id;
    unsigned long index;
    int data;
    pthread_t tid;
    struct lockinfo *next;
    struct sem_operations *ops;
    unsigned long ready;
};

struct sem_wakeup_info {
    int wakeup_count;
    struct sembuf sb[SEMS_PERID];
};

struct sem_operations {
    void (*wait)(struct lockinfo *l);
    int (*wake)(struct sem_wakeup_info *wi, int num_semids, int num);
    void (*setup)(struct sem_wakeup_info **wi, int num_semids);
    void (*cleanup)(int num_semids);
    char *name;
};

int *semid_lookup = NULL;

pthread_mutex_t worklist_mutex = PTHREAD_MUTEX_INITIALIZER;
static unsigned long total_burns = 0;
static unsigned long min_burns = ~0UL;
static unsigned long max_burns = 0;

/* currently running threads */
static int thread_count = 0;

struct lockinfo *worklist = NULL;
static int workers_started = 0;

```

```c
/* total threads started */
static int num_threads = 2048;

static void worklist_add(struct lockinfo *l)
{
    smp_mb();
    l->ready = 1;
}

static struct lockinfo *worklist_rm(void)
{
    static int last_index = 0;
    int i;
    struct lockinfo *l;

    for (i = 0; i < num_threads; i++) {
        int test = (last_index + i) % num_threads;

        l = worklist + test;
        smp_mb();
        if (l->ready) {
            l->ready = 0;
            last_index = test;
            return l;
        }
    }
    return NULL;
}

/* ipc semaphore post& wait */
void wait_ipc_sem(struct lockinfo *l)
{
    struct sembuf sb;
    int ret;
    struct timespec *tvp = NULL;
    struct timespec tv = { 0, 1 };

    sb.sem_num = l->index;
    sb.sem_flg = 0;

    sb.sem_op = -1;
    l->data = 1;

    if (timeout_test && (l->id % 5) == 0)
        tvp = &tv;

    worklist_add(l);
    ret = semtimedop(semid_lookup[l->id], &sb, 1, tvp);

    while(l->data != 0 && tvp) {
        struct timespec tv2 = { 0, 500 };
        nanosleep(&tv2, NULL);
    }

    if (l->data != 0) {
        if (tvp)
            return;
        fprintf(stderr, "wakeup without data update\n");
        exit(1);
    }
    if (ret) {
        if (errno == EAGAIN && tvp)
            return;
        perror("semtimed op");
        exit(1);
    }
}

int ipc_wake_some(struct sem_wakeup_info *wi, int num_semids, int num)
{
    int i;
    int ret;
    struct lockinfo *l;
    int found = 0;

    for (i = 0; i < num_semids; i++) {
```

```
        wi[i].wakeup_count = 0;
    }
    while(num > 0) {
        struct sembuf *sb;
        l = worklist_rm();
        if (!l)
            break;
        if (l->data != 1)
            fprintf(stderr, "warning, lockinfo data was %d\n",
                l->data);
        l->data = 0;
        sb = wi[l->id].sb + wi[l->id].wakeup_count;
        sb->sem_num = l->index;
        sb->sem_op = 1;
        sb->sem_flg = IPC_NOWAIT;
        wi[l->id].wakeup_count++;
        found++;
        num--;
    }
    if (!found)
        return 0;
    for (i = 0; i < num_semids; i++) {
        int wakeup_total;
        int cur;
        int offset = 0;
        if (!wi[i].wakeup_count)
            continue;
        wakeup_total = wi[i].wakeup_count;
        while(wakeup_total > 0) {
            cur = wakeup_total > 64 ? 64 : wakeup_total;
            ret = semtimedop(semid_lookup[i], wi[i].sb + offset,
                    cur, NULL);
            if (ret) {
                perror("semtimedop");
                exit(1);
            }
            offset += cur;
            wakeup_total -= cur;
        }
    }
    return found;
}

void setup_ipc_sems(struct sem_wakeup_info **wi, int num_semids)
{
    int i;
    *wi = malloc(sizeof(**wi) * num_semids);
    semid_lookup = malloc(num_semids * sizeof(int));
    for(i = 0; i < num_semids; i++) {
        semid_lookup[i] = semget(IPC_PRIVATE, SEMS_PERID,
                    IPC_CREAT | 0777);
        if (semid_lookup[i] < 0) {
            perror("semget");
            exit(1);
        }
    }
    sleep(10);
}

void cleanup_ipc_sems(int num)
{
    int i;
    for (i = 0; i < num; i++) {
        semctl(semid_lookup[i], 0, IPC_RMID);
    }
}

struct sem_operations ipc_sem_ops = {
    .wait = wait_ipc_sem,
    .wake = ipc_wake_some,
    .setup = setup_ipc_sems,
    .cleanup = cleanup_ipc_sems,
    .name = "ipc sem operations",
};

/* futex post & wait */
```

```c
void wait_futex_sem(struct lockinfo *l)
{
    int ret;
    l->data = 1;
    worklist_add(l);
    while(l->data == 1) {
        ret = futex(&l->data, FUTEX_WAIT, 1, NULL, NULL, 0);
        /*
        if (ret && ret != EWOULDBLOCK) {
            perror("futex wait");
            exit(1);
        }*/
    }
}

int futex_wake_some(struct sem_wakeup_info *wi, int num_semids, int num)
{
    int i;
    int ret;
    struct lockinfo *l;
    int found = 0;

    for (i = 0; i < num; i++) {
        l = worklist_rm();
        if (!l)
            break;
        if (l->data != 1)
            fprintf(stderr, "warning, lockinfo data was %d\n",
                l->data);
        l->data = 0;
        ret = futex(&l->data, FUTEX_WAKE, 1, NULL, NULL, 0);
        if (ret < 0) {
            perror("futex wake");
            exit(1);
        }
        found++;
    }
    return found;
}

void setup_futex_sems(struct sem_wakeup_info **wi, int num_semids)
{
    return;
}

void cleanup_futex_sems(int num)
{
    return;
}

struct sem_operations futex_sem_ops = {
    .wait = wait_futex_sem,
    .wake = futex_wake_some,
    .setup = setup_futex_sems,
    .cleanup = cleanup_futex_sems,
    .name = "futex sem operations",
};

/* nanosleep sems here */
void wait_nanosleep_sem(struct lockinfo *l)
{
    int ret;
    struct timespec tv = { 0, 1000000 };
    int count = 0;

    l->data = 1;
    worklist_add(l);
    while(l->data) {
        ret = nanosleep(&tv, NULL);
        if (ret) {
            perror("nanosleep");
            exit(1);
        }
        count++;
    }
}
```

```
326
327 int nanosleep_wake_some(struct sem_wakeup_info *wi, int num_semids, int num)
328 {
329     int i;
330     struct lockinfo *l;
331
332     for (i = 0; i < num; i++) {
333         l = worklist_rm();
334         if (!l)
335             break;
336         if (l->data != 1)
337             fprintf(stderr, "warning, lockinfo data was %d\n",
338                 l->data);
339         l->data = 0;
340     }
341     return i;
342 }
343
344 void setup_nanosleep_sems(struct sem_wakeup_info **wi, int num_semids)
345 {
346     return;
347 }
348
349 void cleanup_nanosleep_sems(int num)
350 {
351     return;
352 }
353
354 struct sem_operations nanosleep_sem_ops = {
355     .wait = wait_nanosleep_sem,
356     .wake = nanosleep_wake_some,
357     .setup = setup_nanosleep_sems,
358     .cleanup = cleanup_nanosleep_sems,
359     .name = "nano sleep sem operations",
360 };
361
362 void *worker(void *arg)
363 {
364     struct lockinfo *l = (struct lockinfo *)arg;
365     int burn_count = 0;
366     pthread_t tid = pthread_self();
367     size_t pagesize = getpagesize();
368     char *buf = malloc(pagesize);
369
370     if (!buf) {
371         perror("malloc");
372         exit(1);
373     }
374
375     l->tid = tid;
376     workers_started = 1;
377     smp_mb();
378
379     while(!all_done) {
380         l->ops->wait(l);
381         if (all_done)
382             break;
383         burn_count++;
384     }
385     pthread_mutex_lock(&worklist_mutex);
386     total_burns += burn_count;
387     if (burn_count < min_burns)
388         min_burns = burn_count;
389     if (burn_count > max_burns)
390         max_burns = burn_count;
391     thread_count--;
392     pthread_mutex_unlock(&worklist_mutex);
393     return (void *)0;
394 }
395
396 void print_usage(void)
397 {
398     printf("usage: sembench [-t threads] [-w wake incr] [-r runtime]");
399     printf("                [-o num] (0=ipc, 1=nanosleep, 2=futex)\n");
400     exit(1);
401 }
```

```c

#define NUM_OPERATIONS 3
struct sem_operations *allops[NUM_OPERATIONS] = { &ipc_sem_ops,
                        &nanosleep_sem_ops,
                        &futex_sem_ops};

int main(int ac, char **av) {
    int ret;
    int i;
    int semid = 0;
    int sem_num = 0;
    int burn_count = 0;
    struct sem_wakeup_info *wi = NULL;
    struct timeval start;
    struct timeval now;
    int num_semids = 0;
    int wake_num = 256;
    int run_secs = 30;
    int pagesize = getpagesize();
    char *buf = malloc(pagesize);
    struct sem_operations *ops = allops[0];
    cpu_set_t cpu_mask;
    cpu_set_t target_mask;
    int target_cpu = 0;
    int max_cpu = -1;

    if (!buf) {
        perror("malloc");
        exit(1);
    }
    for (i = 1; i < ac; i++) {
        if (strcmp(av[i], "-t") == 0) {
            if (i == ac -1)
                print_usage();
            num_threads = atoi(av[i+1]);
            i++;
        } else if (strcmp(av[i], "-w") == 0) {
            if (i == ac -1)
                print_usage();
            wake_num = atoi(av[i+1]);
            i++;
        } else if (strcmp(av[i], "-r") == 0) {
            if (i == ac -1)
                print_usage();
            run_secs = atoi(av[i+1]);
            i++;
        } else if (strcmp(av[i], "-o") == 0) {
            int index;
            if (i == ac -1)
                print_usage();
            index = atoi(av[i+1]);
            if (index >= NUM_OPERATIONS) {
                fprintf(stderr, "invalid operations %d\n",
                    index);
                exit(1);
            }
            ops = allops[index];
            i++;
        } else if (strcmp(av[i], "-T") == 0) {
            timeout_test = 1;
        } else if (strcmp(av[i], "-h") == 0) {
            print_usage();
        }
    }
    num_semids = (num_threads + SEMS_PERID - 1) / SEMS_PERID;
    ops->setup(&wi, num_semids);

    ret = sched_getaffinity(0, sizeof(cpu_set_t), &cpu_mask);
    if (ret) {
        perror("sched_getaffinity");
        exit(1);
    }
    for (i = 0; i < CPU_SETSIZE; i++)
        if (CPU_ISSET(i, &cpu_mask))
            max_cpu = i;
    if (max_cpu == -1) {
```

```
        fprintf(stderr, "sched_getaffinity returned empty mask\n");
        exit(1);
    }

    CPU_ZERO(&target_mask);

    worklist = malloc(sizeof(*worklist) * num_threads);
    memset(worklist, 0, sizeof(*worklist) * num_threads);

    for (i = 0; i < num_threads; i++) {
        struct lockinfo *l;
        pthread_t tid;
        thread_count++;
        l = worklist + i;
        if (!l) {
            perror("malloc");
            exit(1);
        }
        l->id = semid;
        l->index = sem_num++;
        l->ops = ops;
        if (sem_num >= SEMS_PERID) {
            semid++;
            sem_num = 0;
        }
        ret = pthread_create(&tid, NULL, worker, (void *)l);
        if (ret) {
            perror("pthread_create");
            exit(1);
        }

        while (!CPU_ISSET(target_cpu, &cpu_mask)) {
            target_cpu++;
            if (target_cpu > max_cpu)
                target_cpu = 0;
        }
        CPU_SET(target_cpu, &target_mask);
        ret = pthread_setaffinity_np(tid, sizeof(cpu_set_t),
                        &target_mask);
        CPU_CLR(target_cpu, &target_mask);
        target_cpu++;

        ret = pthread_detach(tid);
        if (ret) {
            perror("pthread_detach");
            exit(1);
        }
    }
    while(!workers_started) {
        smp_mb();
        usleep(200);
    }
    gettimeofday(&start, NULL);
    //fprintf(stderr, "main loop going\n");
    while(1) {
        ops->wake(wi, num_semids, wake_num);
        burn_count++;
        gettimeofday(&now, NULL);
        if (now.tv_sec - start.tv_sec >= run_secs)
            break;
    }
    //fprintf(stderr, "all done\n");
    all_done = 1;
    while(thread_count > 0) {
        ops->wake(wi, num_semids, wake_num);
        usleep(200);
    }
    //printf("%d threads, waking %d at a time\n", num_threads, wake_num);
    //printf("using %s\n", ops->name);
    //printf("main thread burns: %d\n", burn_count);
    //printf("worker burn count total %lu min %lu max %lu avg %lu\n",
    //       total_burns, min_burns, max_burns, total_burns / num_threads);
    printf("%d seconds: %lu worker burns per second\n",
        (int)(now.tv_sec - start.tv_sec),
        total_burns / (now.tv_sec - start.tv_sec));
    ops->cleanup(num_semids);
```

```
554     return 0;
555 }
556
```

One Level Up Top Level

test/benchmark/ - ktap

- cmp_neq.sh
- cmp_profile.sh
- cmp_table.sh
- sembench.c
- test.sh

test/ - ktap

- _vimrc_local.vim
- benchmark/
- ffi/
- ffi_test/
- lib/
- util/

test/_vimrc_local.vim - ktap

```
1 set expandtab
2 set tw=10000
3 "filetype indent off
4 "filetype plugin off
5 "set nowrap autoindent nosmartindent nocindent indentexpr=
```

test/ffi/ - ktap

- ktap_ffi_test.mod.c

One Level Up Top Level

test/ffi/ktap_ffi_test.mod.c - ktap

Global variables defined

- __this_module
- __used

Source code

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);

struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
    .name = KBUILD_MODNAME,
     .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
    .exit = cleanup_module,
#endif
    .arch = MODULE_ARCH_INIT,
};

static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";

```

One Level Up Top Level

test/ffi_test/ - ktap

- Makefile
- cparser_test.c
- ktap_ffi_test.c

test/ffi_test/Makefile - ktap

```
obj-m += ktap_ffi_test.o

all: funct_mod cparser_test

funct_mod:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

INC=../../include
U_DIR=../../userspace
 RUNTIME=../../runtime
 U_FFI_DIR=$(U_DIR)/ffi
 CPARSER_FILES=cparser.o ctype.o ffi_type.o
 KTAPC_CFLAGS = -Wall -O2

 cparser.o: $(U_FFI_DIR)/cparser.c $(INC)/*
     $(QUIET_CC)$(CC) -DCONFIG_KTAP_FFI -o $@ -c $<

 ctype.o: $(U_FFI_DIR)/ctype.c $(INC)/*
     $(QUIET_CC)$(CC) -DCONFIG_KTAP_FFI -o $@ -c $<

 ffi_type.o: $(RUNTIME)/ffi/ffi_type.c $(INC)/*
     $(QUIET_CC)$(CC) -DCONFIG_KTAP_FFI -o $@ -c $<

 cparser_test: cparser_test.c $(CPARSER_FILES) $(INC)/*
     $(QUIET_CC)$(CC) -DCONFIG_KTAP_FFI -I$(INC) -I$(U_DIR) $(KTAPC_CFLAGS) \
         -o $@ $< $(CPARSER_FILES)

 load:
     insmod ktap_ffi_test.ko

 unload:
     rmmod ktap_ffi_test

 test: all
     @echo "testing cparser:"
     ./cparser_test
     @echo "testing ffi module:"
     rmmod ktap_ffi_test > /dev/null 2>&1 || true
     insmod ktap_ffi_test.ko
     ../../ktap ffi_test.kp
     rmmod ktap_ffi_test.ko
     @echo "[*] all ffi tests passed."

 clean:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
     rm -rf cparser_test
```

test/ffi_test/cparser_test.c - ktap

Global variables defined

- csym_state

Functions defined

- find_kernel_symbol
- lookup_csymbol_id_by_name
- main
- test_func_funct_module
- test_func_sched_clock
- test_func_time_to_tm
- test_pointer_symbols
- test_struct_timespec
- test_var_arg_function

Macros defined

- DO_TEST
- assert_csym_arr_type
- assert_farg_type
- assert_fret_type
- cs_arr
- cs_arr_size
- cs_nr

Source code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5
6  #include "ktap_types.h"
7  #include "ktap_opcodes.h"
8  #include "../../userspace/ktapc.h"
9  #include "cparser.h"
10
11  void ffi_cparser_init(void);
12  void ffi_cparser_free(void);
13  int ffi_parse_cdef(const char *s);
14
15  static cp_csymbol_state *csym_state;
16
17  #define cs_nr (csym_state->cs_nr)
18  #define cs_arr_size (csym_state->cs_arr_size)
19  #define cs_arr (csym_state->cs_arr)
```

```
20
21
22  #define DO_TEST(name) do {                  \
23      ffi_cparser_init();                     \
24      int ret;                                \
25      printf("[*] start "#name" test...  ");  \
26      ret = test_##name();                    \
27      if (ret)                                \
28          fprintf(stderr, "\n[!] "#name" test failed.\n");\
29      else                                    \
30          printf(" passed.\n");               \
31      ffi_cparser_free();                     \
32  } while (0)
33
34  #define assert_csym_arr_type(cs_arr, n, t) do {         \
35      csymbol *ncs;                           \
36      ncs = &cs_arr[n];                       \
37      assert(ncs->type == t);                 \
38  } while (0)
39
40  #define assert_fret_type(fcs, t) do {               \
41      csymbol *ncs;                           \
42      ncs = &cs_arr[fcs->ret_id];             \
43      assert(ncs->type == t);                 \
44  } while (0)
45
46  #define assert_farg_type(fcs, n, t) do {            \
47      csymbol *ncs;                           \
48      ncs = &cs_arr[fcs->arg_ids[n]];             \
49      assert(ncs->type == t);                 \
50  } while (0)
51
52
53
54
55  /* mock find_kernel_symbol */
56  unsigned long find_kernel_symbol(const char *symbol)
57  {
58      return 0xdeadbeef;
59  }
60
61  int lookup_csymbol_id_by_name(char *name)
62  {
63      int i;
64
65      for (i = 0; i < cs_nr; i++) {
66          if (!strcmp(name, cs_arr[i].name)) {
67              return i;
68          }
69      }
70
71      return -1;
72  }
73
74  int test_func_sched_clock()
75  {
76      int idx;
77      csymbol *cs;
78      csymbol_func *fcs;
79
80      ffi_parse_cdef("unsigned long long sched_clock();");
81
82      csym_state = ctype_get_csym_state();
83      assert(cs_arr);
84
85      idx = lookup_csymbol_id_by_name("sched_clock");
86      assert(idx >= 0);
87      cs = &cs_arr[idx];
88      assert(cs->type == FFI_FUNC);
89
90      fcs = csym_func(cs);
91
92      /* check return type */
93      assert_fret_type(fcs, FFI_UINT64);
94
95      /* check arguments */
```

```
    assert(fcs->arg_nr == 0);

    return 0;
}

int test_func_funct_module()
{
    int idx;
    csymbol *cs;
    csymbol_func *fcs;

    ffi_parse_cdef("void funct_void();");
    ffi_parse_cdef("int funct_int1(unsigned char a, char b, unsigned short c, "
            "short d);");
    ffi_parse_cdef("long long funct_int2(unsigned int a, int b, "
            "unsigned long c, long d, unsigned long long e, "
            "long long f, long long g);");
    ffi_parse_cdef("void *funct_pointer1(char *a);");

    csym_state = ctype_get_csym_state();
    assert(cs_arr);

    /* check funct_void function */
    idx = lookup_csymbol_id_by_name("funct_void");
    assert(idx >= 0);
    cs = &cs_arr[idx];
    assert(cs->type == FFI_FUNC);
    fcs = csym_func(cs);

    /* check return type */
    assert_fret_type(fcs, FFI_VOID);

    /* check arguments */
    assert(fcs->arg_nr == 0);



    /* check funct_int1 function */
    idx = lookup_csymbol_id_by_name("funct_int1");
    assert(idx >= 0);
    cs = &cs_arr[idx];
    assert(cs);
    assert(cs->type == FFI_FUNC);
    fcs = csym_func(cs);

    /* check return type */
    assert_fret_type(fcs, FFI_INT32);

    /* check arguments */
    assert(fcs->arg_nr == 4);
    assert_farg_type(fcs, 0, FFI_UINT8);
    assert_farg_type(fcs, 1, FFI_INT8);
    assert_farg_type(fcs, 2, FFI_UINT16);
    assert_farg_type(fcs, 3, FFI_INT16);



    /* check funct_int2 function */
    idx = lookup_csymbol_id_by_name("funct_int2");
    assert(idx >= 0);
    cs = &cs_arr[idx];
    assert(cs);
    assert(cs->type == FFI_FUNC);
    fcs = csym_func(cs);

    /* check return type */
    assert_fret_type(fcs, FFI_INT64);

    /* check arguments */
    assert(fcs->arg_nr == 7);
    assert_farg_type(fcs, 0, FFI_UINT32);
    assert_farg_type(fcs, 1, FFI_INT32);
    assert_farg_type(fcs, 2, FFI_UINT64);
    assert_farg_type(fcs, 3, FFI_INT64);
    assert_farg_type(fcs, 4, FFI_UINT64);
    assert_farg_type(fcs, 5, FFI_INT64);
```

```
    assert_farg_type(fcs, 6, FFI_INT64);



    /* check funct_pointer1 function */
    idx = lookup_csymbol_id_by_name("funct_pointer1");
    assert(idx >= 0);
    cs = &cs_arr[idx];
    assert(cs);
    assert(cs->type == FFI_FUNC);
    fcs = csym_func(cs);

    /* check return type */
    assert_fret_type(fcs, FFI_PTR);

    /* check arguments */
    assert(fcs->arg_nr == 1);
    assert_farg_type(fcs, 0, FFI_PTR);
    /*@TODO check pointer dereference type  18.11 2013 (houqp)*/

    return 0;
}

int test_struct_timespec()
{
    int idx;
    csymbol *cs;
    csymbol_struct *stcs;

    ffi_parse_cdef("struct timespec { long ts_sec; long ts_nsec; };");

    csym_state = ctype_get_csym_state();
    assert(cs_arr);

    idx = lookup_csymbol_id_by_name("struct timespec");
    assert(idx >= 0);
    cs = &cs_arr[idx];
    assert(cs);
    assert(cs->type == FFI_STRUCT);

    stcs = csym_struct(cs);
    assert(stcs->memb_nr == 2);

    return 0;
}

int test_func_time_to_tm()
{
    int idx;
    csymbol *cs, *arg_cs;
    csymbol_struct *stcs;
    csymbol_func *fcs;

    ffi_parse_cdef("typedef long time_t;");
    ffi_parse_cdef("struct tm { "
            "int tm_sec;"
            "int tm_min;"
            "int tm_hour;"
            "int tm_mday;"
            "int tm_mon;"
            "long tm_year;"
            "int tm_wday;"
            "int tm_yday;"
        "};");
    ffi_parse_cdef("void time_to_tm(time_t totalsecs, int offset, struct tm *result);");

    csym_state = ctype_get_csym_state();
    assert(cs_arr);

    idx = lookup_csymbol_id_by_name("struct tm");
    assert(idx >= 0);
    cs = cp_id_to_csym(idx);
    assert(cs);
    assert(cs->type == FFI_STRUCT);

    stcs = csym_struct(cs);
```

```
248     assert(stcs->memb_nr == 8);
249
250
251     idx = lookup_csymbol_id_by_name("time_to_tm");
252     assert(idx >= 0);
253     cs = cp_id_to_csym(idx);
254     assert(cs);
255     assert(cs->type == FFI_FUNC);
256
257     fcs = csym_func(cs);
258     assert(csymf_arg_nr(fcs) == 3);
259     /* check first argument */
260     assert_farg_type(fcs, 0, FFI_INT64);
261
262     /* check second argument */
263     assert_farg_type(fcs, 1, FFI_INT32);
264     /* check third argument */
265     assert_farg_type(fcs, 2, FFI_PTR);
266     arg_cs = cp_csymf_arg(fcs, 2);
267     assert(!strcmp(csym_name(arg_cs), "struct tm *"));
268     assert(csym_ptr_deref_id(arg_cs) ==
269             lookup_csymbol_id_by_name("struct tm"));
270
271     return 0;
272 }
273
274 int test_pointer_symbols()
275 {
276     csymbol_func *fcs_foo, *fcs_bar;
277
278     /* int pointer symbol should be resolved to the same id */
279     ffi_parse_cdef("void foo(int *a);");
280     ffi_parse_cdef("int *bar(void);");
281
282     csym_state = ctype_get_csym_state();
283     assert(cs_arr);
284
285     fcs_foo = csym_func(cp_id_to_csym(lookup_csymbol_id_by_name("foo")));
286     fcs_bar = csym_func(cp_id_to_csym(lookup_csymbol_id_by_name("bar")));
287
288     assert(csymf_arg_ids(fcs_foo)[0] == csymf_ret_id(fcs_bar));
289     assert(cp_csymf_arg(fcs_foo, 0) == cp_csymf_ret(fcs_bar));
290
291     return 0;
292 }
293
294 int test_var_arg_function()
295 {
296     csymbol_func *fcs;
297
298     ffi_parse_cdef("int printk(char *fmt, ...);");
299
300     fcs = csym_func(cp_id_to_csym(lookup_csymbol_id_by_name("printk")));
301
302     /* var arg function needs void * type argument type checking */
303     assert(lookup_csymbol_id_by_name("void *") >= 0);
304
305     assert_fret_type(fcs, FFI_INT32);
306     assert_farg_type(fcs, 0, FFI_PTR);
307     assert(fcs->has_var_arg);
308
309     return 0;
310 }
311
312 int main (int argc, char *argv[])
313 {
314     DO_TEST(func_sched_clock);
315     DO_TEST(func_funct_module);
316     DO_TEST(struct_timespec);
317     DO_TEST(func_time_to_tm);
318     DO_TEST(pointer_symbols);
319     DO_TEST(var_arg_function);
320
321     return 0;
322 }
```

userspace/ffi/cparser.c - ktap

Global variables defined

- tok1
- tok2
- tok3

Data types defined

- etoken
- test
- token

Functions defined

- calculate_constant
- calculate_constant1
- calculate_constant10
- calculate_constant11
- calculate_constant12
- calculate_constant13
- calculate_constant2
- calculate_constant3
- calculate_constant4
- calculate_constant5
- calculate_constant6
- calculate_constant7
- calculate_constant8
- calculate_constant9
- check_token
- cp_error
- cp_init_parser
- ffi_cparser_free
- ffi_cparser_init
- ffi_lookup_csymbol_id_by_name
- ffi_parse_cdef

- ffi_parse_new
- increase_ptr_deref_level
- instantiate_typedef
- max_bitfield_size
- next_token
- parse_argument
- parse_argument2
- parse_attribute
- parse_enum
- parse_function
- parse_function_arguments
- parse_record
- parse_root
- parse_struct
- parse_type
- parse_type_name
- parse_typedef
- put_back
- require_token
- set_type_name

Macros defined

- END
- INT_TYPE
- INT_TYPE
- IS_CONST
- IS_LITERAL
- IS_RESTRICT
- IS_VOLATILE
- LONG_TYPE
- LONG_TYPE
- PRAGMA_POP
- SHORT_TYPE
- SHORT_TYPE

- max
- min

Source code

```
#include <stdarg.h>
#include "../cparser.h"

#define IS_CONST(tok) (IS_LITERAL(tok, "const") || IS_LITERAL(tok, "__const") \
            || IS_LITERAL(tok, "__const__"))
#define IS_VOLATILE(tok) (IS_LITERAL(tok, "volatile") || \
                IS_LITERAL(tok, "__volatile") || \
                IS_LITERAL(tok, "__volatile__"))
#define IS_RESTRICT(tok) (IS_LITERAL(tok, "restrict") || \
                 IS_LITERAL(tok, "__restrict") || \
                 IS_LITERAL(tok, "__restrict__"))

 #define max(a,b) ((a) < (b) ? (b) : (a))
 #define min(a,b) ((a) < (b) ? (a) : (b))


 enum etoken {
     /* 0 - 3 */
     TOK_NIL,
     TOK_NUMBER,
     TOK_STRING,
     TOK_TOKEN,

     /* the order of these values must match the token strings in lex.c */

     /* 4 - 5 */
     TOK_3_BEGIN,
     TOK_VA_ARG,

     /* 6 - 14 */
     TOK_2_BEGIN,
     TOK_LEFT_SHIFT, TOK_RIGHT_SHIFT, TOK_LOGICAL_AND, TOK_LOGICAL_OR,
     TOK_LESS_EQUAL, TOK_GREATER_EQUAL, TOK_EQUAL, TOK_NOT_EQUAL,

     /* 15 - 20 */
     TOK_1_BEGIN,
     TOK_OPEN_CURLY, TOK_CLOSE_CURLY, TOK_SEMICOLON, TOK_COMMA, TOK_COLON,
     /* 21 - 30 */
     TOK_ASSIGN, TOK_OPEN_PAREN, TOK_CLOSE_PAREN, TOK_OPEN_SQUARE, TOK_CLOSE_SQUARE,
     TOK_DOT, TOK_AMPERSAND, TOK_LOGICAL_NOT, TOK_BITWISE_NOT, TOK_MINUS,
     /* 31 - 40 */
     TOK_PLUS, TOK_STAR, TOK_DIVIDE, TOK_MODULUS, TOK_LESS,
     TOK_GREATER, TOK_BITWISE_XOR, TOK_BITWISE_OR, TOK_QUESTION, TOK_POUND,

     /* 41 - 43 */
     TOK_REFERENCE = TOK_AMPERSAND,
     TOK_MULTIPLY = TOK_STAR,
     TOK_BITWISE_AND = TOK_AMPERSAND,
 };

 struct token {
     enum etoken type;
     int64_t integer;
     const char *str;
     size_t size;
 };

 #define IS_LITERAL(TOK, STR) \
     (((TOK).size == sizeof(STR) - 1) && \
         0 == memcmp((TOK).str, STR, sizeof(STR) - 1))


 static int parse_type_name(struct parser *P, char *type_name);
 static void parse_argument(struct parser *P, struct cp_ctype *ct,
         struct token *pname, struct parser *asmname);
 static int parse_attribute(struct parser *P, struct token *tok,
         struct cp_ctype *ct, struct parser *asmname);
 static int parse_record(struct parser *P, struct cp_ctype *ct);
```

```c
static void instantiate_typedef(struct parser *P, struct cp_ctype *tt,
        const struct cp_ctype *ft);


/* the order of tokens _must_ match the order of the enum etoken enum */

static char tok3[][4] = {
    "...", /* unused ">>=", "<<=", */
};

static char tok2[][3] = {
    "<<", ">>", "&&", "||", "<=",
    ">=", "==", "!=",
    /* unused "+=", "-=", "*=", "/=", "%=", "&=", "^=",
     * "|=", "++", "--", "->", "::", */
};

static char tok1[] = {
    '{', '}', ';', ',', ':',
    '=', '(', ')', '[', ']',
    '.', '&', '!', '~', '-',
    '+', '*', '/', '%', '<',
    '>', '^', '|', '?', '#'
};


/* this function never returns, but it's an idiom to use it in C functions
 * as return cp_error */
void cp_error(const char *err_msg_fmt, ...)
{
    va_list ap;

    fprintf(stderr, "cparser error:\n");

    va_start(ap, err_msg_fmt);
    vfprintf(stderr, err_msg_fmt, ap);
    va_end(ap);

    exit(EXIT_FAILURE);
}

static int set_type_name(char *dst, unsigned type, const char *src, int len)
{
    int prefix_len;
    char *prefix = NULL;

    if (type == STRUCT_TYPE)
        prefix = "struct ";
    else if (type == UNION_TYPE)
        prefix = "union ";
    else
        cp_error("Only set type name for struct or union\n");
    prefix_len = sizeof(prefix);

    if (len + prefix_len > MAX_TYPE_NAME_LEN)
        return -1;

    memset(dst, 0, MAX_TYPE_NAME_LEN);
    strcpy(dst, prefix);
    strncat(dst, src, len);

    return 0;
}

static void increase_ptr_deref_level(struct parser *P, struct cp_ctype *ct)
{
    if (ct->pointers == POINTER_MAX) {
        cp_error("maximum number of pointer derefs reached - use a "
            "struct to break up the pointers on line %d", P->line);
    } else {
        ct->pointers++;
        ct->const_mask <<= 1;
    }
}

static int next_token(struct parser *P, struct token *tok)
```

```
{
    size_t i;
    const char *s = P->next;

    /* UTF8 BOM */
    if (s[0] == '\xEF' && s[1] == '\xBB' && s[2] == '\xBF') {
        s += 3;
    }

    /* consume whitespace and comments */
    for (;;) {
        /* consume whitespace */
        while (*s == '\t' || *s == '\n' || *s == ' '
                || *s == '\v' || *s == '\r') {
            if (*s == '\n') {
                P->line++;
            }
            s++;
        }

        /* consume comments */
        if (*s == '/' && *(s+1) == '/') {

            s = strchr(s, '\n');
            if (!s) {
                cp_error("non-terminated comment");
            }

        } else if (*s == '/' && *(s+1) == '*') {
            s += 2;

            for (;;) {
                if (s[0] == '\0') {
                    cp_error("non-terminated comment");
                } else if (s[0] == '*' && s[1] == '/') {
                    s += 2;
                    break;
                } else if (s[0] == '\n') {
                    P->line++;
                }
                s++;
            }

        } else if (*s == '\0') {
            tok->type = TOK_NIL;
            return 0;

        } else {
            break;
        }
    }

    P->prev = s;

    for (i = 0; i < sizeof(tok3) / sizeof(tok3[0]); i++) {
        if (s[0] == tok3[i][0] && s[1] == tok3[i][1] && s[2] == tok3[i][2]) {
            tok->type = (enum etoken) (TOK_3_BEGIN + 1 + i);
            P->next = s + 3;
            goto end;
        }
    }

    for (i = 0; i < sizeof(tok2) / sizeof(tok2[0]); i++) {
        if (s[0] == tok2[i][0] && s[1] == tok2[i][1]) {
            tok->type = (enum etoken) (TOK_2_BEGIN + 1 + i);
            P->next = s + 2;
            goto end;
        }
    }

    for (i = 0; i < sizeof(tok1) / sizeof(tok1[0]); i++) {
        if (s[0] == tok1[i]) {
            tok->type = (enum etoken) (TOK_1_BEGIN + 1 + i);
            P->next = s + 1;
            goto end;
        }
```

```
    }

    if (*s == '.' || *s == '-' || ('0' <= *s && *s <= '9')) {
        /* number */
        tok->type = TOK_NUMBER;

        /* split out the negative case so we get the full range of
         * bits for unsigned (eg to support 0xFFFFFFFF where
         * sizeof(long) == 4 */
        if (*s == '-') {
            tok->integer = strtol(s, (char**) &s, 0);
        } else {
            tok->integer = strtoul(s, (char**) &s, 0);
        }

        while (*s == 'u' || *s == 'U' || *s == 'l' || *s == 'L') {
            s++;
        }

        P->next = s;
        goto end;

    } else if (*s == '\'' || *s == '\"') {
        /* "..." or '...' */
        char quote = *s;
        s++; /* jump over " */

        tok->type = TOK_STRING;
        tok->str = s;

        while (*s != quote) {
            if (*s == '\0' || (*s == '\\' && *(s+1) == '\0')) {
                cp_error("string not finished\n");
            }
            if (*s == '\\') {
                s++;
            }
            s++;
        }

        tok->size = s - tok->str;
        s++; /* jump over " */
        P->next = s;
        goto end;

    } else if (('a' <= *s && *s <= 'z') || ('A' <= *s && *s <= 'Z')
            || *s == '_') {
        /* tokens */
        tok->type = TOK_TOKEN;
        tok->str = s;

        while (('a' <= *s && *s <= 'z') || ('A' <= *s && *s <= 'Z')
                || *s == '_' || ('0' <= *s && *s <= '9')) {
            s++;
        }

        tok->size = s - tok->str;
        P->next = s;
        goto end;
    } else {
        cp_error("invalid character %d", P->line);
    }

end:
    return 1;
}

static void require_token(struct parser *P, struct token *tok)
{
    if (!next_token(P, tok)) {
        cp_error("unexpected end");
    }
}

static void check_token(struct parser *P, int type, const char *str,
                const char *err, ...)
```

```
{
    va_list ap;
    struct token tok;
    if (!next_token(P, &tok) || tok.type != type
            || (tok.type == TOK_TOKEN && (tok.size != strlen(str)
                || memcmp(tok.str, str, tok.size) != 0))) {

        va_start(ap, err);
        vfprintf(stderr, err, ap);
        va_end(ap);

        exit(EXIT_FAILURE);
    }
}

static void put_back(struct parser *P) {
    P->next = P->prev;
}

int64_t calculate_constant(struct parser *P);

/* parses out the base type of a type expression in a function declaration,
* struct definition, typedef etc
*
* leaves the usr value of the type on the stack
*/
int parse_type(struct parser *P, struct cp_ctype *ct)
{
    struct token tok;

    memset(ct, 0, sizeof(*ct));

    require_token(P, &tok);

    /* get function attributes before the return type */
    while (parse_attribute(P, &tok, ct, NULL)) {
        require_token(P, &tok);
    }

    /* get const/volatile before the base type */
    for (;;) {
        if (tok.type != TOK_TOKEN) {
            cp_error("unexpected value before type name on line %d",
                    P->line);
            return 0;
        } else if (IS_CONST(tok)) {
            ct->const_mask = 1;
            require_token(P, &tok);
        } else if (IS_VOLATILE(tok) || IS_RESTRICT(tok)) {
            /* ignored for now */
            require_token(P, &tok);
        } else {
            break;
        }
    }

    /* get base type */
    if (tok.type != TOK_TOKEN) {
        cp_error("unexpected value before type name on line %d", P->line);
        return 0;
    } else if (IS_LITERAL(tok, "struct")) {
        ct->type = STRUCT_TYPE;
        parse_record(P, ct);
    } else if (IS_LITERAL(tok, "union")) {
        ct->type = UNION_TYPE;
        parse_record(P, ct);
    } else if (IS_LITERAL(tok, "enum")) {
        ct->type = ENUM_TYPE;
        parse_record(P, ct);
    } else {
        put_back(P);

        /* lookup type */
        struct cp_ctype *lct;
        char cur_type_name[MAX_TYPE_NAME_LEN];

```

```
        memset(cur_type_name, 0, MAX_TYPE_NAME_LEN);
        parse_type_name(P, cur_type_name);
        lct = ctype_lookup_type(cur_type_name);
        if (!lct)
            cp_error("unknow type: \"%s\"\n", cur_type_name);

        instantiate_typedef(P, ct, lct);
    }

    while (next_token(P, &tok)) {
        if (tok.type != TOK_TOKEN) {
            put_back(P);
            break;
        } else if (IS_CONST(tok) || IS_VOLATILE(tok)) {
            /* ignore for now */
        } else {
            put_back(P);
            break;
        }
    }

    return 0;
}

enum test {TEST};

/* Parses an enum definition from after the open curly through to the close
* curly. Expects the user table to be on the top of the stack
*/
static int parse_enum(struct parser *P, struct cp_ctype *type)
{
    struct token tok;
    int value = -1;

    /*@TODO clean up this function when enum support is added*/
    cp_error("TODO: enum not supported!\n");

    for (;;) {
        require_token(P, &tok);

        if (tok.type == TOK_CLOSE_CURLY) {
            break;
        } else if (tok.type != TOK_TOKEN) {
            cp_error("unexpected token in enum at line %d", P->line);
            return 0;
        }
        require_token(P, &tok);

        if (tok.type == TOK_COMMA || tok.type == TOK_CLOSE_CURLY) {
            /* we have an auto calculated enum value */
            value++;
        } else if (tok.type == TOK_ASSIGN) {
            /* we have an explicit enum value */
            value = (int) calculate_constant(P);
            require_token(P, &tok);
        } else {
            cp_error("unexpected token in enum at line %d", P->line);
            return 0;
        }

        if (tok.type == TOK_CLOSE_CURLY) {
            break;
        } else if (tok.type != TOK_COMMA) {
            cp_error("unexpected token in enum at line %d", P->line);
            return 0;
        }
    }

    type->base_size = sizeof(enum test);
    type->align_mask = sizeof(enum test) - 1;

    return 0;
}

/* Parses a struct from after the open curly through to the close curly. */
static int parse_struct(struct parser *P, const struct cp_ctype *ct)
```

```
{
    struct token tok;

    /* parse members */
    for (;;) {
        struct cp_ctype mbase;

        /* see if we're at the end of the struct */
        require_token(P, &tok);
        if (tok.type == TOK_CLOSE_CURLY) {
            break;
        } else if (ct->is_variable_struct) {
            cp_error("can't have members after a variable sized "
                    "member on line %d", P->line);
            return -1;
        } else {
            put_back(P);
        }

        /* members are of the form
         * <base type> <arg>, <arg>, <arg>;
         * eg struct foo bar, *bar2[2];
         * mbase is 'struct foo'
         * mtype is '' then '*[2]'
         * mname is 'bar' then 'bar2'
         */

        parse_type(P, &mbase);

        for (;;) {
            struct token mname;
            struct cp_ctype mt = mbase;

            memset(&mname, 0, sizeof(mname));

            if (ct->is_variable_struct) {
                cp_error("can't have members after a variable "
                    "sized member on line %d", P->line);
                return -1;
            }

            parse_argument(P, &mt, &mname, NULL);

            if (!mt.is_defined && (mt.pointers - mt.is_array) == 0) {
                cp_error("member type is undefined on line %d",
                        P->line);
                return -1;
            }

            if (mt.type == VOID_TYPE
                    && (mt.pointers - mt.is_array) == 0) {
                cp_error("member type can not be void on line %d",
                        P->line);
                return -1;
            }

            mt.has_member_name = (mname.size > 0);
            if (mt.has_member_name) {
                cp_push_ctype_with_name(&mt,
                        mname.str, mname.size);
            } else {
                cp_push_ctype(&mt);
            }

            require_token(P, &tok);
            if (tok.type == TOK_SEMICOLON) {
                break;
            } else if (tok.type != TOK_COMMA) {
                cp_error("unexpected token in struct "
                    "definition on line %d", P->line);
            }
        }
    }

    return 0;
}
```

```
525
526 /* copy over attributes that could be specified before the typedef eg
527 * __attribute__(packed) const type_t */
528 static void instantiate_typedef(struct parser *P, struct cp_ctype *tt,
529         const struct cp_ctype *ft)
530 {
531     struct cp_ctype pt = *tt;
532     *tt = *ft;
533
534     tt->const_mask |= pt.const_mask;
535     tt->is_packed = pt.is_packed;
536
537     if (tt->is_packed) {
538         tt->align_mask = 0;
539     } else {
540         /* Instantiate the typedef in the current packing. This may be
541          * further updated if a pointer is added or another alignment
542          * attribute is applied. If pt.align_mask is already non-zero
543          * than an increased alignment via __declspec(aligned(#)) has
544          * been set. */
545         tt->align_mask = max(min(P->align_mask, tt->align_mask),
546                     pt.align_mask);
547     }
548 }
549
550 /* this parses a struct or union starting with the optional
551 * name before the opening brace
552 * leaves the type usr value on the stack */
553 static int parse_record(struct parser *P, struct cp_ctype *ct)
554 {
555     struct token tok;
556     char cur_type_name[MAX_TYPE_NAME_LEN];
557     int has_name;
558
559     require_token(P, &tok);
560
561     /* name is optional */
562     memset(cur_type_name, 0, MAX_TYPE_NAME_LEN);
563     if (tok.type == TOK_TOKEN) {
564         /* declaration */
565         struct cp_ctype *lct;
566
567         set_type_name(cur_type_name, ct->type, tok.str, tok.size);
568
569         /* lookup the name to see if we've seen this type before */
570         lct = ctype_lookup_type(cur_type_name);
571
572         if (!lct) {
573             /* new type, delay type registration to the end
574              * of this function */
575             ct->ffi_base_cs_id = ct->ffi_cs_id = -1;
576         } else {
577             /* get the exsting declared type */
578             if (lct->type != ct->type) {
579                 cp_error("type '%s' previously declared as '%s'",
580                     cur_type_name,
581                     csym_name(ct_ffi_cs(lct)));
582             }
583
584             instantiate_typedef(P, ct, lct);
585         }
586
587         /* if a name is given then we may be at the end of the string
588          * eg for ffi.new('struct foo') */
589         if (!next_token(P, &tok)) {
590             return 0;
591         }
592         has_name = 1;
593     } else {
594         char anon_name[MAX_TYPE_NAME_LEN];
595         /* create a new unnamed record */
596         sprintf(anon_name, "%d line", P->line);
597         set_type_name(cur_type_name, ct->type,
598                 anon_name, strlen(anon_name));
599         ct->ffi_base_cs_id = ct->ffi_cs_id = -1;
600         has_name = 0;
```

```
601     }
602
603     if (tok.type != TOK_OPEN_CURLY) {
604         /* this may just be a declaration or use of the type as an
605          * argument or member */
606         put_back(P);
607         if (!has_name)
608             cp_error("noname record type declaration\n");
609
610         /* build symbol for vm */
611         ct->ffi_base_cs_id =
612             cp_symbol_build_fake_record(cur_type_name, ct->type);
613         ct->ffi_cs_id = ct->ffi_base_cs_id;
614         return 0;
615     }
616
617     if (ct->is_defined) {
618         cp_error("redefinition in line %d", P->line);
619         return 0;
620     }
621
622     if (ct->type == ENUM_TYPE) {
623         parse_enum(P, ct);
624         cp_set_defined(ct);
625     } else {
626         int start_top = ctype_stack_top();
627         /* we do a two stage parse, where we parse the content first
628          * and build up the temp user table. We then iterate over that
629          * to calculate the offsets and fill out ct_usr. This is so we
630          * can handle out of order members (eg vtable) and attributes
631          * specified at the end of the struct. */
632         parse_struct(P, ct);
633         cp_set_defined(ct);
634         /* build symbol for vm */
635         ct->ffi_base_cs_id = cp_symbol_build_record(
636                 cur_type_name, ct->type, start_top);
637         ct->ffi_cs_id = ct->ffi_base_cs_id;
638         /* save cp_ctype for parser */
639         cp_ctype_reg_type(cur_type_name, ct);
640     }
641
642     return 0;
643 }
644
645 /* parses single or multi work built in types, and pushes it onto the stack */
646 static int parse_type_name(struct parser *P, char *type_name)
647 {
648     struct token tok;
649     int flags = 0;
650
651     enum {
652         UNSIGNED = 0x01,
653         SIGNED = 0x02,
654         LONG = 0x04,
655         SHORT = 0x08,
656         INT = 0x10,
657         CHAR = 0x20,
658         LONG_LONG = 0x40,
659         INT8 = 0x80,
660         INT16 = 0x100,
661         INT32 = 0x200,
662         INT64 = 0x400,
663     };
664
665     require_token(P, &tok);
666
667     /* we have to manually decode the builtin types since they can take up
668      * more then one token */
669     for (;;) {
670         if (tok.type != TOK_TOKEN) {
671             break;
672         } else if (IS_LITERAL(tok, "unsigned")) {
673             flags |= UNSIGNED;
674         } else if (IS_LITERAL(tok, "signed")) {
675             flags |= SIGNED;
676         } else if (IS_LITERAL(tok, "short")) {
```

```
            flags |= SHORT;
        } else if (IS_LITERAL(tok, "char")) {
            flags |= CHAR;
        } else if (IS_LITERAL(tok, "long")) {
            flags |= (flags & LONG) ? LONG_LONG : LONG;
        } else if (IS_LITERAL(tok, "int")) {
            flags |= INT;
        } else if (IS_LITERAL(tok, "__int8")) {
            flags |= INT8;
        } else if (IS_LITERAL(tok, "__int16")) {
            flags |= INT16;
        } else if (IS_LITERAL(tok, "__int32")) {
            flags |= INT32;
        } else if (IS_LITERAL(tok, "__int64")) {
            flags |= INT64;
        } else if (IS_LITERAL(tok, "register")) {
            /* ignore */
        } else {
            break;
        }

        if (!next_token(P, &tok)) {
            break;
        }
    }

    if (flags) {
        put_back(P);
    }

    if (flags & CHAR) {
        if (flags & SIGNED) {
            strcpy(type_name, "int8_t");
        } else if (flags & UNSIGNED) {
            strcpy(type_name, "uint8_t");
        } else {
            if (((char) -1) > 0) {
                strcpy(type_name, "uint8_t");
            } else {
                strcpy(type_name, "int8_t");
            }
        }
    } else if (flags & INT8) {
        strcpy(type_name, (flags & UNSIGNED) ? "uint8_t" : "int8_t");
    } else if (flags & INT16) {
        strcpy(type_name, (flags & UNSIGNED) ? "uint16_t" : "int16_t");
    } else if (flags & INT32) {
        strcpy(type_name, (flags & UNSIGNED) ? "uint32_t" : "int32_t");
    } else if (flags & INT64) {
        strcpy(type_name, (flags & UNSIGNED) ? "uint64_t" : "int64_t");
    } else if (flags & LONG_LONG) {
        strcpy(type_name, (flags & UNSIGNED) ? "uint64_t" : "int64_t");
    } else if (flags & SHORT) {
#define SHORT_TYPE(u) (sizeof(short) == sizeof(int64_t) ? \
        u "int64_t" : sizeof(short) == sizeof(int32_t) ? \
        u "int32_t" : u "int16_t")
        if (flags & UNSIGNED) {
            strcpy(type_name, SHORT_TYPE("u"));
        } else {
            strcpy(type_name, SHORT_TYPE(""));
        }
#undef SHORT_TYPE
    } else if (flags & LONG) {
#define LONG_TYPE(u) (sizeof(long) == sizeof(int64_t) ? \
        u "int64_t" : u "int32_t")
        if (flags & UNSIGNED) {
            strcpy(type_name, LONG_TYPE("u"));
        } else {
            strcpy(type_name, LONG_TYPE(""));
        }
#undef LONG_TYPE
    } else if (flags) {
#define INT_TYPE(u) (sizeof(int) == sizeof(int64_t) ? \
        u "int64_t" : sizeof(int) == sizeof(int32_t) ? \
        u "int32_t" : u "int16_t")
        if (flags & UNSIGNED) {
```

```
            strcpy(type_name, INT_TYPE("u"));
        } else {
            strcpy(type_name, INT_TYPE(""));
        }
#undef INT_TYPE
    } else {
        strncpy(type_name, tok.str, tok.size);
    }

    return 0;
}

/* parse_attribute parses a token to see if it is an attribute. It may then
* parse some following tokens to decode the attribute setting the appropriate
* fields in ct. It will return 1 if the token was used (and possibly some
* more following it) or 0 if not. If the token was used, the next token must
* be retrieved using next_token/require_token. */
static int parse_attribute(struct parser *P, struct token *tok,
        struct cp_ctype *ct, struct parser *asmname)
{
    if (tok->type != TOK_TOKEN) {
        return 0;
    } else if (asmname && (IS_LITERAL(*tok, "__asm__")
                || IS_LITERAL(*tok, "__asm"))) {
        check_token(P, TOK_OPEN_PAREN, NULL,
                "unexpected token after __asm__ on line %d",
                P->line);
        *asmname = *P;

        require_token(P, tok);
        while (tok->type == TOK_STRING) {
            require_token(P, tok);
        }

        if (tok->type != TOK_CLOSE_PAREN) {
            cp_error("unexpected token after __asm__ on line %d",
                    P->line);
        }
        return 1;

    } else if (IS_LITERAL(*tok, "__attribute__")
            || IS_LITERAL(*tok, "__declspec")) {
        int parens = 1;
        check_token(P, TOK_OPEN_PAREN, NULL,
                "expected parenthesis after __attribute__ or "
                "__declspec on line %d", P->line);

        for (;;) {
            require_token(P, tok);
            if (tok->type == TOK_OPEN_PAREN) {
                parens++;
            } else if (tok->type == TOK_CLOSE_PAREN) {
                if (--parens == 0) {
                    break;
                }

            } else if (tok->type != TOK_TOKEN) {
                /* ignore unknown symbols within parentheses */

            } else if (IS_LITERAL(*tok, "align") ||
                    IS_LITERAL(*tok, "aligned") ||
                    IS_LITERAL(*tok, "__aligned__")) {
                unsigned align = 0;
                require_token(P, tok);

                switch (tok->type) {
                case TOK_CLOSE_PAREN:
                    align = ALIGNED_DEFAULT;
                    put_back(P);
                    break;

                case TOK_OPEN_PAREN:
                    require_token(P, tok);

                    if (tok->type != TOK_NUMBER) {
                        cp_error("expected align(#) "
```

```
                        "on line %d", P->line);
                }

                switch (tok->integer) {
                case 1: align = 0; break;
                case 2: align = 1; break;
                case 4: align = 3; break;
                case 8: align = 7; break;
                case 16: align = 15; break;
                default:
                    cp_error("unsupported align "
                        "size on line %d",
                        P->line);
                }

                check_token(P, TOK_CLOSE_PAREN, NULL,
                    "expected align(#) on line %d",
                    P->line);
                break;

            default:
                cp_error("expected align(#) on line %d",
                        P->line);
            }

            /* __attribute__(aligned(#)) is only supposed
             * to increase alignment */
            ct->align_mask = max(align, ct->align_mask);

        } else if (IS_LITERAL(*tok, "packed")
                || IS_LITERAL(*tok, "__packed__")) {
            ct->align_mask = 0;
            ct->is_packed = 1;

        } else if (IS_LITERAL(*tok, "mode")
                || IS_LITERAL(*tok, "__mode__")) {

            check_token(P, TOK_OPEN_PAREN, NULL,
                "expected mode(MODE) on line %d",
                P->line);

            require_token(P, tok);
            if (tok->type != TOK_TOKEN) {
                cp_error("expected mode(MODE) on line %d",
                        P->line);
            }


            struct {char ch; uint16_t v;} a16;
            struct {char ch; uint32_t v;} a32;
            struct {char ch; uint64_t v;} a64;

            if (IS_LITERAL(*tok, "QI")
                    || IS_LITERAL(*tok, "__QI__")
                    || IS_LITERAL(*tok, "byte")
                    || IS_LITERAL(*tok, "__byte__")
               ) {
                ct->type = INT8_TYPE;
                ct->base_size = sizeof(uint8_t);
                ct->align_mask = 0;

            } else if (IS_LITERAL(*tok, "HI")
                    || IS_LITERAL(*tok, "__HI__")) {
                ct->type = INT16_TYPE;
                ct->base_size = sizeof(uint16_t);
                ct->align_mask = ALIGNOF(a16);

            } else if (IS_LITERAL(*tok, "SI")
                    || IS_LITERAL(*tok, "__SI__")
#if defined ARCH_X86 || defined ARCH_ARM
                    || IS_LITERAL(*tok, "word")
                    || IS_LITERAL(*tok, "__word__")
                    || IS_LITERAL(*tok, "pointer")
                    || IS_LITERAL(*tok, "__pointer__")
#endif
                   ) {
```

```
                    ct->type = INT32_TYPE;
                    ct->base_size = sizeof(uint32_t);
                    ct->align_mask = ALIGNOF(a32);

                } else if (IS_LITERAL(*tok, "DI")
                        || IS_LITERAL(*tok, "__DI__")
#if defined ARCH_X64
                        || IS_LITERAL(*tok, "word")
                        || IS_LITERAL(*tok, "__word__")
                        || IS_LITERAL(*tok, "pointer")
                        || IS_LITERAL(*tok, "__pointer__")
#endif
                          ) {
                    ct->type = INT64_TYPE;
                    ct->base_size = sizeof(uint64_t);
                    ct->align_mask = ALIGNOF(a64);

                } else {
                    cp_error("unexpected mode on line %d",
                            P->line);
                }

                check_token(P, TOK_CLOSE_PAREN, NULL,
                    "expected mode(MODE) on line %d", P->line);

            } else if (IS_LITERAL(*tok, "cdecl")
                    || IS_LITERAL(*tok, "__cdecl__")) {
                ct->calling_convention = C_CALL;

            } else if (IS_LITERAL(*tok, "fastcall")
                    || IS_LITERAL(*tok, "__fastcall__")) {
                ct->calling_convention = FAST_CALL;

            } else if (IS_LITERAL(*tok, "stdcall")
                    || IS_LITERAL(*tok, "__stdcall__")) {
                ct->calling_convention = STD_CALL;
            }
            /* ignore unknown tokens within parentheses */
        }
        return 1;

    } else if (IS_LITERAL(*tok, "__cdecl")) {
        ct->calling_convention = C_CALL;
        return 1;

    } else if (IS_LITERAL(*tok, "__fastcall")) {
        ct->calling_convention = FAST_CALL;
        return 1;

    } else if (IS_LITERAL(*tok, "__stdcall")) {
        ct->calling_convention = STD_CALL;
        return 1;

    } else if (IS_LITERAL(*tok, "__extension__")
             || IS_LITERAL(*tok, "extern")) {
        /* ignore */
        return 1;
    } else {
        return 0;
    }
}

/* parses from after the opening paranthesis to after the closing parenthesis */
static void parse_function_arguments(struct parser* P, struct cp_ctype* ct)
{
    struct token tok;
    int args = 0;

    for (;;) {
        require_token(P, &tok);

        if (tok.type == TOK_CLOSE_PAREN)
            break;

        if (args) {
            if (tok.type != TOK_COMMA) {
```

```
                cp_error("unexpected token in function "
                        "argument %d on line %d",
                        args, P->line);
            }
            require_token(P, &tok);
        }

        if (tok.type == TOK_VA_ARG) {
            ct->has_var_arg = true;
            check_token(P, TOK_CLOSE_PAREN, "",
                    "unexpected token after ... in "
                    "function on line %d",
                    P->line);
            break;
        } else if (tok.type == TOK_TOKEN) {
            struct cp_ctype at;

            put_back(P);
            parse_type(P, &at);
             parse_argument(P, &at, NULL, NULL);

             /* array arguments are just treated as their
              * base pointer type */
             at.is_array = 0;

             /* check for the c style int func(void) and error
              * on other uses of arguments of type void */
             if (at.type == VOID_TYPE && at.pointers == 0) {
                 if (args) {
                     cp_error("can't have argument of type "
                             "void on line %d",
                             P->line);
                 }

                 check_token(P, TOK_CLOSE_PAREN, "",
                     "unexpected void in function on line %d",
                     P->line);
                 break;
             }
             cp_push_ctype(&at);
             args++;
         } else {
             cp_error("unexpected token in function argument %d "
                     "on line %d", args+1, P->line);
         }
    }
}

static int max_bitfield_size(int type)
{
    switch (type) {
    case BOOL_TYPE:
        return 1;
    case INT8_TYPE:
        return 8;
    case INT16_TYPE:
        return 16;
    case INT32_TYPE:
    case ENUM_TYPE:
        return 32;
    case INT64_TYPE:
        return 64;
    default:
        return -1;
    }
}

static struct cp_ctype *parse_argument2(struct parser *P, struct cp_ctype *ct,
        struct token *name, struct parser *asmname);

/* parses from after the first ( in a function declaration or function pointer
* can be one of:
* void foo(...) before ...
* void (foo)(...) before foo
* void (* <>)(...) before <> which is the inner type */
static struct cp_ctype *parse_function(struct parser *P, struct cp_ctype *ct,
```

```
1057                 struct token *name, struct parser *asmname)
1058 {
1059     /* We have a function pointer or a function. The usr table will
1060      * get replaced by the canonical one (if there is one) in
1061      * find_canonical_usr after all the arguments and returns have
1062      * been parsed. */
1063     struct token tok;
1064     struct cp_ctype *ret = ct;
1065 
1066     cp_push_ctype(ct);
1067 
1068     memset(ct, 0, sizeof(*ct));
1069     ct->base_size = sizeof(void (*)());
1070     ct->align_mask = min(FUNCTION_ALIGN_MASK, P->align_mask);
1071     ct->type = FUNCTION_TYPE;
1072     ct->is_defined = 1;
1073 
1074     if (name->type == TOK_NIL) {
1075         for (;;) {
1076             require_token(P, &tok);
1077 
1078             if (tok.type == TOK_STAR) {
1079                 if (ct->type == FUNCTION_TYPE) {
1080                     ct->type = FUNCTION_PTR_TYPE;
1081                 } else {
1082                     increase_ptr_deref_level(P, ct);
1083                 }
1084             } else if (parse_attribute(P, &tok, ct, asmname)) {
1085                 /* parse_attribute sets the appropriate fields */
1086             } else {
1087                 /* call parse_argument to handle the inner
1088                  * contents e.g. the <> in "void (* <>)
1089                  * (...)". Note that the inner contents can
1090                  * itself be a function, a function ptr,
1091                  * array, etc (e.g. "void (*signal(int sig,
1092                  * void (*func)(int)))(int)" ). */
1093                 cp_error("TODO: inner function not supported for now.");
1094                 put_back(P);
1095                 ct = parse_argument2(P, ct, name, asmname);
1096                 break;
1097             }
1098         }
1099 
1100         check_token(P, TOK_CLOSE_PAREN, NULL,
1101             "unexpected token in function on line %d", P->line);
1102         check_token(P, TOK_OPEN_PAREN, NULL,
1103             "unexpected token in function on line %d", P->line);
1104     }
1105 
1106     parse_function_arguments(P, ct);
1107 
1108     /*@TODO support for inner function  24.11 2013 (houqp)*/
1109     /* if we have an inner function then set the outer function ptr as its
1110      * return type and return the inner function
1111      * e.g. for void (* <signal(int, void (*)(int))> )(int) inner is
1112      * surrounded by <>, return type is void (*)(int) */
1113 
1114     return ret;
1115 }
1116 
1117 static struct cp_ctype *parse_argument2(struct parser *P, struct cp_ctype *ct,
1118                 struct token *name, struct parser *asmname)
1119 {
1120     struct token tok;
1121 
1122     for (;;) {
1123         if (!next_token(P, &tok)) {
1124             /* we've reached the end of the string */
1125             break;
1126         } else if (tok.type == TOK_STAR) {
1127             increase_ptr_deref_level(P, ct);
1128 
1129             /* __declspec(align(#)) may come before the type in a
1130              * member */
1131             if (!ct->is_packed) {
1132                 ct->align_mask = max(min(PTR_ALIGN_MASK, P->align_mask),
```

```
                        ct->align_mask);
            }
        } else if (tok.type == TOK_REFERENCE) {
            cp_error("NYI: c++ reference types");
            return 0;
        } else if (parse_attribute(P, &tok, ct, asmname)) {
            /* parse attribute has filled out appropriate fields in type */

        } else if (tok.type == TOK_OPEN_PAREN) {
            ct = parse_function(P, ct, name, asmname);
        } else if (tok.type == TOK_OPEN_SQUARE) {
            /* array */
            if (ct->pointers == POINTER_MAX) {
                cp_error("maximum number of pointer derefs "
                    "reached - use a struct to break up "
                    "the pointers");
            }
            ct->is_array = 1;
            ct->pointers++;
            ct->const_mask <<= 1;
            require_token(P, &tok);

            if (ct->pointers == 1 && !ct->is_defined) {
                cp_error("array of undefined type on line %d",
                        P->line);
            }

            if (ct->is_variable_struct || ct->is_variable_array) {
                cp_error("can't have an array of a variably "
                    "sized type on line %d", P->line);
            }

            if (tok.type == TOK_QUESTION) {
                ct->is_variable_array = 1;
                ct->variable_increment = (ct->pointers > 1) ?
                        sizeof(void*) : ct->base_size;
                check_token(P, TOK_CLOSE_SQUARE, "",
                    "invalid character in array on line %d",
                    P->line);

            } else if (tok.type == TOK_CLOSE_SQUARE) {
                ct->array_size = 0;

            } else if (tok.type == TOK_TOKEN && IS_RESTRICT(tok)) {
                /* odd gcc extension foo[__restrict] for arguments */
                ct->array_size = 0;
                check_token(P, TOK_CLOSE_SQUARE, "",
                    "invalid character in array on line %d",
                    P->line);
            } else {
                int64_t asize;
                put_back(P);
                asize = calculate_constant(P);
                if (asize < 0) {
                    cp_error("array size can not be "
                        "negative on line %d", P->line);
                    return 0;
                }
                ct->array_size = (size_t) asize;
                check_token(P, TOK_CLOSE_SQUARE, "",
                    "invalid character in array on line %d",
                    P->line);
            }

        } else if (tok.type == TOK_COLON) {
            int64_t bsize = calculate_constant(P);

            if (ct->pointers || bsize < 0
                    || bsize > max_bitfield_size(ct->type)) {
                cp_error("invalid bitfield on line %d", P->line);
            }

            ct->is_bitfield = 1;
            ct->bit_size = (unsigned) bsize;

        } else if (tok.type != TOK_TOKEN) {
```

```
1209            /* we've reached the end of the declaration */
1210            put_back(P);
1211            break;
1212
1213        } else if (IS_CONST(tok)) {
1214            ct->const_mask |= 1;
1215
1216        } else if (IS_VOLATILE(tok) || IS_RESTRICT(tok)) {
1217            /* ignored for now */
1218
1219        } else {
1220            *name = tok;
1221        }
1222    }
1223
1224    return ct;
1225 }
1226
1227
1228
1229 /* parses after the main base type of a typedef, function argument or
1230 * struct/union member
1231 * eg for const void* bar[3] the base type is void with the subtype so far of
1232 * const, this parses the "* bar[3]" and updates the type argument
1233 *
1234 * type must be as filled out by parse_type
1235 *
1236 * pushes the updated user value on the top of the stack
1237 */
1238 void parse_argument(struct parser *P, struct cp_ctype *ct, struct token *pname,
1239             struct parser *asmname)
1240 {
1241     struct token tok, name;
1242
1243     memset(&name, 0, sizeof(name));
1244     parse_argument2(P, ct, &name, asmname);
1245
1246     for (;;) {
1247         if (!next_token(P, &tok)) {
1248             break;
1249         } else if (parse_attribute(P, &tok, ct, asmname)) {
1250             /* parse_attribute sets the appropriate fields */
1251         } else {
1252             put_back(P);
1253             break;
1254         }
1255     }
1256
1257     if (pname) {
1258         *pname = name;
1259     }
1260 }
1261
1262 static void parse_typedef(struct parser *P)
1263 {
1264     struct token tok;
1265     struct cp_ctype base_type;
1266     char typedef_name[MAX_TYPE_NAME_LEN];
1267
1268     parse_type(P, &base_type);
1269
1270     for (;;) {
1271         struct cp_ctype arg_type = base_type;
1272         struct token name;
1273
1274         memset(&name, 0, sizeof(name));
1275
1276         parse_argument(P, &arg_type, &name, NULL);
1277
1278         if (!name.size) {
1279             cp_error("Can't have a typedef without a name on line %d",
1280                     P->line);
1281         } else if (arg_type.is_variable_array) {
1282             cp_error("Can't typedef a variable length array on line %d",
1283                     P->line);
1284         }
```

```
1285
1286            memset(typedef_name, 0, sizeof(typedef_name));
1287            strncpy(typedef_name, name.str, name.size);
1288            /* link typedef name with ctype for parser */
1289            cp_ctype_reg_type(typedef_name, &arg_type);
1290
1291            require_token(P, &tok);
1292
1293            if (tok.type == TOK_SEMICOLON) {
1294                break;
1295            } else if (tok.type != TOK_COMMA) {
1296                cp_error("Unexpected character in typedef on line %d",
1297                        P->line);
1298            }
1299        }
1300    }
1301
1302    #define END 0
1303    #define PRAGMA_POP 1
1304
1305    static int parse_root(struct parser *P)
1306    {
1307        struct token tok;
1308
1309        while (next_token(P, &tok)) {
1310            /* we can have:
1311             * struct definition
1312             * enum definition
1313             * union definition
1314             * struct/enum/union declaration
1315             * typedef
1316             * function declaration
1317             * pragma pack
1318             */
1319
1320            if (tok.type == TOK_SEMICOLON) {
1321                /* empty semicolon in root continue on */
1322
1323            } else if (tok.type == TOK_POUND) {
1324
1325                check_token(P, TOK_TOKEN, "pragma",
1326                    "unexpected pre processor directive on line %d",
1327                    P->line);
1328                check_token(P, TOK_TOKEN, "pack",
1329                    "unexpected pre processor directive on line %d",
1330                    P->line);
1331                check_token(P, TOK_OPEN_PAREN, "",
1332                    "invalid pack directive on line %d",
1333                    P->line);
1334                require_token(P, &tok);
1335
1336                if (tok.type == TOK_NUMBER) {
1337                    if (tok.integer != 1 && tok.integer != 2
1338                            && tok.integer != 4
1339                            && tok.integer != 8
1340                            && tok.integer != 16) {
1341                        cp_error("pack directive with invalid "
1342                                "pack size on line %d",
1343                                P->line);
1344                        return 0;
1345                    }
1346
1347                    P->align_mask = (unsigned) (tok.integer - 1);
1348                    check_token(P, TOK_CLOSE_PAREN, "",
1349                        "invalid pack directive on line %d",
1350                        P->line);
1351                } else if (tok.type == TOK_TOKEN && IS_LITERAL(tok, "push")) {
1352                    /*int line = P->line;*/
1353                    unsigned previous_alignment = P->align_mask;
1354
1355                    check_token(P, TOK_CLOSE_PAREN, "",
1356                        "invalid pack directive on line %d",
1357                        P->line);
1358
1359                    if (parse_root(P) != PRAGMA_POP) {
1360                        cp_error("reached end of string "
```

```
                        "without a pragma pop to "
                        "match the push on line %d",
                        P->line);
                    return 0;
                }

                P->align_mask = previous_alignment;

            } else if (tok.type == TOK_TOKEN && IS_LITERAL(tok, "pop")) {
                check_token(P, TOK_CLOSE_PAREN, "",
                    "invalid pack directive on line %d",
                        P->line);
                return PRAGMA_POP;
            } else {
                cp_error("invalid pack directive on line %d",
                        P->line);
                return 0;
            }
        } else if (tok.type != TOK_TOKEN) {
            cp_error("unexpected character on line %d", P->line);
            return 0;
        } else if (IS_LITERAL(tok, "__extension__")) {
            /* ignore */
            continue;
        } else if (IS_LITERAL(tok, "extern")) {
            /* ignore extern as data and functions can only be
             * extern */
            continue;
        } else if (IS_LITERAL(tok, "typedef")) {
            parse_typedef(P);
        } else if (IS_LITERAL(tok, "static")) {
            /*@TODO we haven't tested static so far */
            cp_error("TODO: support static keyword.\n");
        } else {
            /* type declaration, type definition, or function
             * declaration */
            struct cp_ctype type;
            struct token name;
            struct parser asmname;

            memset(&name, 0, sizeof(name));
            memset(&asmname, 0, sizeof(asmname));

            put_back(P);
            parse_type(P, &type);

            for (;;) {
                parse_argument(P, &type, &name, &asmname);

                if (name.size) {
                    /* global/function declaration */
                    cp_symbol_build_func(&type, name.str, name.size);
                    /* @TODO asmname is not used for now
                     * since we are not supporting __asm__
                     * as this point.
                     * might need to bind it with function
                     * name later. */
                } else {
                    /* type declaration/definition -
                     * already been processed */
                }
                require_token(P, &tok);

                if (tok.type == TOK_SEMICOLON) {
                    break;
                } else if (tok.type != TOK_COMMA) {
                    cp_error("missing semicolon on line %d",
                            P->line);
                }
            }
        }
    }

    return END;
}
```

```
1437 static int64_t calculate_constant2(struct parser *P, struct token *tok);
1438
1439 /* () */
1440 static int64_t calculate_constant1(struct parser *P, struct token *tok)
1441 {
1442     int64_t ret;
1443
1444     if (tok->type == TOK_NUMBER) {
1445         ret = tok->integer;
1446         next_token(P, tok);
1447         return ret;
1448
1449     } else if (tok->type == TOK_TOKEN) {
1450         /* look up name in constants table */
1451         cp_error("TODO: support name lookup in constant table\n");
1452         next_token(P, tok);
1453         return ret;
1454
1455     } else if (tok->type == TOK_OPEN_PAREN) {
1456         struct parser before_cast = *P;
1457         cp_error("TODO: handle open parent token in constant1\n");
1458         *P = before_cast;
1459         ret = calculate_constant(P);
1460
1461         require_token(P, tok);
1462         if (tok->type != TOK_CLOSE_PAREN) {
1463             cp_error("error whilst parsing constant at line %d",
1464                     P->line);
1465         }
1466
1467         next_token(P, tok);
1468         return ret;
1469     } else {
1470         cp_error("unexpected token whilst parsing constant at line %d",
1471                 P->line);
1472         return 0;
1473     }
1474 }
1475
1476 /* ! and ~, unary + and -, and sizeof */
1477 static int64_t calculate_constant2(struct parser *P, struct token *tok)
1478 {
1479     if (tok->type == TOK_LOGICAL_NOT) {
1480         require_token(P, tok);
1481         return !calculate_constant2(P, tok);
1482
1483     } else if (tok->type == TOK_BITWISE_NOT) {
1484         require_token(P, tok);
1485         return ~calculate_constant2(P, tok);
1486
1487     } else if (tok->type == TOK_PLUS) {
1488         require_token(P, tok);
1489         return calculate_constant2(P, tok);
1490
1491     } else if (tok->type == TOK_MINUS) {
1492         require_token(P, tok);
1493         return -calculate_constant2(P, tok);
1494
1495     } else if (tok->type == TOK_TOKEN &&
1496             (IS_LITERAL(*tok, "sizeof")
1497              || IS_LITERAL(*tok, "alignof")
1498              || IS_LITERAL(*tok, "__alignof__")
1499              || IS_LITERAL(*tok, "__alignof"))) {
1500         cp_error("TODO: support sizeof\n");
1501         bool issize = IS_LITERAL(*tok, "sizeof");
1502         struct cp_ctype type;
1503
1504         require_token(P, tok);
1505         if (tok->type != TOK_OPEN_PAREN) {
1506             cp_error("invalid sizeof at line %d", P->line);
1507         }
1508
1509         parse_type(P, &type);
1510         parse_argument(P, &type, NULL, NULL);
1511
1512         require_token(P, tok);
```

```
        if (tok->type != TOK_CLOSE_PAREN) {
            cp_error("invalid sizeof at line %d", P->line);
        }

        next_token(P, tok);

        return issize ? ctype_size(&type) : type.align_mask + 1;

    } else {
        return calculate_constant1(P, tok);
    }
}

/* binary * / and % (left associative) */
static int64_t calculate_constant3(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant2(P, tok);

    for (;;) {
        if (tok->type == TOK_MULTIPLY) {
            require_token(P, tok);
            left *= calculate_constant2(P, tok);

        } else if (tok->type == TOK_DIVIDE) {
            require_token(P, tok);
            left /= calculate_constant2(P, tok);

        } else if (tok->type == TOK_MODULUS) {
            require_token(P, tok);
            left %= calculate_constant2(P, tok);

        } else {
            return left;
        }
    }
}

/* binary + and - (left associative) */
static int64_t calculate_constant4(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant3(P, tok);

    for (;;) {
        if (tok->type == TOK_PLUS) {
            require_token(P, tok);
            left += calculate_constant3(P, tok);

        } else if (tok->type == TOK_MINUS) {
            require_token(P, tok);
            left -= calculate_constant3(P, tok);

        } else {
            return left;
        }
    }
}

/* binary << and >> (left associative) */
static int64_t calculate_constant5(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant4(P, tok);

    for (;;) {
        if (tok->type == TOK_LEFT_SHIFT) {
            require_token(P, tok);
            left <<= calculate_constant4(P, tok);

        } else if (tok->type == TOK_RIGHT_SHIFT) {
            require_token(P, tok);
            left >>= calculate_constant4(P, tok);

        } else {
            return left;
        }
    }
}
```

```c

/* binary <, <=, >, and >= (left associative) */
static int64_t calculate_constant6(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant5(P, tok);

    for (;;) {
        if (tok->type == TOK_LESS) {
            require_token(P, tok);
            left = (left < calculate_constant5(P, tok));

        } else if (tok->type == TOK_LESS_EQUAL) {
            require_token(P, tok);
            left = (left <= calculate_constant5(P, tok));

        } else if (tok->type == TOK_GREATER) {
            require_token(P, tok);
            left = (left > calculate_constant5(P, tok));

        } else if (tok->type == TOK_GREATER_EQUAL) {
            require_token(P, tok);
            left = (left >= calculate_constant5(P, tok));

        } else {
            return left;
        }
    }
}

/* binary ==, != (left associative) */
static int64_t calculate_constant7(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant6(P, tok);

    for (;;) {
        if (tok->type == TOK_EQUAL) {
            require_token(P, tok);
            left = (left == calculate_constant6(P, tok));

        } else if (tok->type == TOK_NOT_EQUAL) {
            require_token(P, tok);
            left = (left != calculate_constant6(P, tok));

        } else {
            return left;
        }
    }
}

/* binary & (left associative) */
static int64_t calculate_constant8(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant7(P, tok);

    for (;;) {
        if (tok->type == TOK_BITWISE_AND) {
            require_token(P, tok);
            left = (left & calculate_constant7(P, tok));

        } else {
            return left;
        }
    }
}

/* binary ^ (left associative) */
static int64_t calculate_constant9(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant8(P, tok);

    for (;;) {
        if (tok->type == TOK_BITWISE_XOR) {
            require_token(P, tok);
            left = (left ^ calculate_constant8(P, tok));

        } else {
```

```
            return left;
        }
    }
}

/* binary | (left associative) */
static int64_t calculate_constant10(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant9(P, tok);

    for (;;) {
        if (tok->type == TOK_BITWISE_OR) {
            require_token(P, tok);
            left = (left | calculate_constant9(P, tok));

        } else {
            return left;
        }
    }
}

/* binary && (left associative) */
static int64_t calculate_constant11(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant10(P, tok);

    for (;;) {
        if (tok->type == TOK_LOGICAL_AND) {
            require_token(P, tok);
            left = (left && calculate_constant10(P, tok));

        } else {
            return left;
        }
    }
}

/* binary || (left associative) */
static int64_t calculate_constant12(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant11(P, tok);

    for (;;) {
        if (tok->type == TOK_LOGICAL_OR) {
            require_token(P, tok);
            left = (left || calculate_constant11(P, tok));

        } else {
            return left;
        }
    }
}

/* ternary ?: (right associative) */
static int64_t calculate_constant13(struct parser *P, struct token *tok)
{
    int64_t left = calculate_constant12(P, tok);

    if (tok->type == TOK_QUESTION) {
        int64_t middle, right;
        require_token(P, tok);
        middle = calculate_constant13(P, tok);
        if (tok->type != TOK_COLON) {
            cp_error("invalid ternery (? :) in constant on line %d",
                    P->line);
        }
        require_token(P, tok);
        right = calculate_constant13(P, tok);
        return left ? middle : right;

    } else {
        return left;
    }
}

int64_t calculate_constant(struct parser* P)
```

```
{
    struct token tok;
    int64_t ret;
    require_token(P, &tok);
    ret = calculate_constant13(P, &tok);

    if (tok.type != TOK_NIL) {
        put_back(P);
    }

    return ret;
}

void cp_init_parser(struct parser *P, const char *s)
{
    memset(P, 0, sizeof(struct parser));
    P->line = 1;
    P->prev = P->next = s;
    P->align_mask = DEFAULT_ALIGN_MASK;
}

/* used for ffi.cdef */
int ffi_parse_cdef(const char *s)
{
    struct parser P;

    cp_init_parser(&P, s);
    if (parse_root(&P) == PRAGMA_POP) {
        cp_error("pragma pop without an associated push on line %d",
                P.line);
    }

    return 0;
}

/* used for ffi.new */
void ffi_parse_new(const char *s, struct cp_ctype *ct)
{
    struct parser P;

    cp_init_parser(&P, s);
    parse_type(&P, ct);
    parse_argument(&P, ct, NULL, NULL);
    cp_update_csym_in_ctype(ct);
}

int ffi_lookup_csymbol_id_by_name(const char *s)
{
    struct parser P;
    struct cp_ctype ct;

    cp_init_parser(&P, s);
    parse_type(&P, &ct);
    cp_update_csym_in_ctype(&ct);
    return ct.ffi_cs_id;
}

void ffi_cparser_init(void)
{
    cp_ctype_init();
}

void ffi_cparser_free(void)
{
    cp_ctype_free();
}
```

One Level Up Top Level

userspace/ffi/ - ktap

- cparser.c
- ctype.c

userspace/ffi/ctype.c - ktap

Global variables defined

- csym_state
- cte_arr
- cte_arr_size
- cte_nr
- cts

Data types defined

- cp_ctype_entry
- cp_ctype_entry
- cp_ctype_stack
- ctype_stack

Functions defined

- __cp_symbol_dump_func
- __cp_symbol_dump_struct
- cp_ctype_dump_stack
- cp_ctype_free
- cp_ctype_init
- cp_ctype_reg_csymbol
- cp_ctype_reg_type
- cp_id_to_csym
- cp_push_ctype
- cp_push_ctype_with_name
- cp_set_defined
- cp_symbol_build_fake_record
- cp_symbol_build_func
- cp_symbol_build_pointer
- cp_symbol_build_record
- cp_symbol_dump_func
- cp_symbol_dump_struct
- cp_update_csym_in_ctype

- ct_set_type
- ctype_get_csym_state
- ctype_lookup_builtin_type
- ctype_lookup_csymbol_id
- ctype_lookup_type
- ctype_reg_table_grow
- ctype_size
- ctype_stack_free_space
- ctype_stack_grow
- ctype_stack_reset
- ctype_stack_top
- init_builtin_type

Macros defined

- DEFAULT_CTYPE_ARR_SIZE
- DEFAULT_STACK_SIZE
- DEFAULT_SYM_ARR_SIZE
- MAX_STACK_SIZE
- cs_arr
- cs_arr_size
- cs_nr
- ct_stack
- ct_stack_ct

Source code

```
#include "../../include/ktap_types.h"
#include "../../include/ktap_opcodes.h"
#include "../ktapc.h"
#include "../cparser.h"


/* for ktap vm */
cp_csymbol_state csym_state;

 #define cs_nr (csym_state.cs_nr)
 #define cs_arr_size (csym_state.cs_arr_size)
 #define cs_arr (csym_state.cs_arr)

 csymbol *cp_id_to_csym(int id)
 {
     return &cs_arr[id];
 }


 typedef struct cp_ctype_entry {
     char name[MAX_TYPE_NAME_LEN];
```

```
    struct cp_ctype ct;
} cp_ctype_entry;

#define DEFAULT_CTYPE_ARR_SIZE 100
static int cte_nr;
static int cte_arr_size;
static cp_ctype_entry *cte_arr;


/* stack to help maintain state during parsing */
typedef struct cp_ctype_stack {
    int size;
    int top;
    cp_ctype_entry *stack;
} ctype_stack;


static ctype_stack cts;

#define ct_stack(id) (&(cts.stack[id]))
#define ct_stack_ct(id) (&(cts.stack[id].ct))



csymbol_id cp_ctype_reg_csymbol(csymbol *cs);
csymbol_id ctype_lookup_csymbol_id(const char *name);


size_t ctype_size(const struct cp_ctype *ct)
{
    if (ct->pointers - ct->is_array) {
        return sizeof(void*) * (ct->is_array ? ct->array_size : 1);

    } else if (!ct->is_defined || ct->type == VOID_TYPE) {
        cp_error("can't calculate size of an undefined type");
        return 0;
    } else if (ct->variable_size_known) {
        assert(ct->is_variable_struct && !ct->is_array);
        return ct->base_size + ct->variable_increment;
    } else if (ct->is_variable_array || ct->is_variable_struct) {
        cp_error("internal error: calc size of variable type with "
                "unknown size");
        return 0;
    } else {
        return ct->base_size * (ct->is_array ? ct->array_size : 1);
    }
}

#define MAX_STACK_SIZE 100
int ctype_stack_grow(int size)
{
    struct cp_ctype_entry *new_st;

    assert(cts.size + size < MAX_STACK_SIZE);

    new_st = realloc(cts.stack, (cts.size+size)*sizeof(cp_ctype_entry));
    if (new_st)
        cts.stack = new_st;
    else
        return -1;

    cts.size += size;

    return size;
}

int ctype_stack_free_space()
{
    return cts.size - cts.top;
}

int ctype_stack_top()
{
    return cts.top;
}

```

```
98  void ctype_stack_reset(int top)
99  {
100     cts.top = top;
101 }
102
103 /* This function should be called before you would fetch
104  * ffi_cs_id from ctype */
105 void cp_update_csym_in_ctype(struct cp_ctype *ct)
106 {
107     int i;
108     struct cp_ctype *nct;
109
110     assert(ct->ffi_cs_id >= 0);
111     /* we have to check pointer here because cparser does type lookup by name
112      * before parsing '*', and for pointers, ct will always be the
113      * original type */
114     if (ct->pointers) {
115         for (i = 0; i < cte_nr; i++) {
116             nct = &(cte_arr[i].ct);
117             if (nct->type == ct->type &&
118                     ct->ffi_base_cs_id == nct->ffi_base_cs_id &&
119                     nct->pointers == ct->pointers) {
120                 break;
121             }
122         }
123
124         if (i == cte_nr) {
125             /* pointer type not found
126              * create a new pointer symbol for this type */
127             /* associate ctype with new csymbol */
128             ct->ffi_cs_id = cp_symbol_build_pointer(ct);
129             /* register wit new pointer name */
130             cp_ctype_reg_type(csym_name(ct_ffi_cs(ct)), ct);
131         } else {
132             /* pointer type already registered, reinstantiate ct */
133             ct->ffi_cs_id = cte_arr[i].ct.ffi_cs_id;
134         }
135     }
136 }
137
138 /* push ctype to stack, create new csymbol if needed */
139 void cp_push_ctype_with_name(struct cp_ctype *ct, const char *name, int nlen)
140 {
141     if (ctype_stack_free_space() < 1)
142         ctype_stack_grow(4);
143
144     cp_update_csym_in_ctype(ct);
145     memset(ct_stack(cts.top), 0, sizeof(cp_ctype_entry));
146     ct_stack(cts.top)->ct = *ct;
147     if (name)
148         strncpy(ct_stack(cts.top)->name, name, nlen);
149     cts.top++;
150 }
151
152 void cp_push_ctype(struct cp_ctype *ct)
153 {
154     cp_push_ctype_with_name(ct, NULL, 0);
155 }
156
157 void cp_set_defined(struct cp_ctype *ct)
158 {
159     ct->is_defined = 1;
160
161     /* @TODO: update ctypes and cdatas that were created before the
162      * definition came in */
163 }
164
165 void cp_ctype_dump_stack()
166 {
167     int i;
168     struct cp_ctype *ct;
169
170     printf("---------------------------\n");
171     printf("start of ctype stack (%d) dump: \n", cts.top);
172     for (i = 0; i < cts.top; i++) {
173         ct = ct_stack_ct(i);
```

```
        printf("[%d] -> cp_ctype: %d, sym_type: %d, pointer: %d "
            "symbol_id: %d, name: %s\n",
            i, ct->type,
            csym_type(ct_ffi_cs(ct)), ct->pointers, ct->ffi_cs_id,
            ct_stack(i)->name);
    }
}

int ctype_reg_table_grow()
{
    cp_ctype_entry *new_arr;

    new_arr = realloc(cte_arr, sizeof(cp_ctype_entry)*cte_arr_size*2);
    if (!new_arr)
        cp_error("failed to allocate memory for ctype array\n");

    cte_arr_size = cte_arr_size * 2;
    return 0;
}

/* return index in csymbol array */
csymbol_id cp_ctype_reg_csymbol(csymbol *cs)
{
    if (cs_nr >= cs_arr_size) {
        cs_arr_size *= 2;
        cs_arr = realloc(cs_arr, cs_arr_size*sizeof(csymbol));
        if (!cs_arr)
            cp_error("failed to extend csymbol array!\n");
    }

    cs_arr[cs_nr] = *cs;
    cs_nr++;

    return cs_nr-1;
}

/* start csymbol reg table */
csymbol_id ctype_lookup_csymbol_id(const char *name)
{
    int i;
    csymbol *ct;

    for (i = 0; i < cs_nr; i++) {
        ct = cp_id_to_csym(i);
        if (!strcmp(name, ct->name))
            return i;
    }

    return -1;
}

void __cp_symbol_dump_struct(csymbol *cs)
{
    int i;
    csymbol *ncs;
    csymbol_struct *stcs = csym_struct(cs);

    printf("=== [%s] definition ==================\n", csym_name(cs));
    for (i = 0; i < stcs->memb_nr; i++) {
        printf("\t(%d) ", i);
        printf("csym_id: %d, ", stcs->members[i].id);
        ncs = cp_id_to_csym(stcs->members[i].id);
        printf("name: %s, ffi_ctype: %d, %s\n",
            stcs->members[i].name, ncs->type, csym_name(ncs));
    }
}

void cp_symbol_dump_struct(int id)
{
    __cp_symbol_dump_struct(cp_id_to_csym(id));
}

int cp_symbol_build_record(const char *stname, int type, int start_top)
{
    int i, id, memb_size;
    cp_ctype_entry *cte;
```

```
    csymbol nst;
    struct_member *st_membs;
    csymbol_struct *stcs;
    struct cp_ctype *ct;

    if (cts.top <= start_top || !stname ||
            (type != STRUCT_TYPE && type != UNION_TYPE)) {
        cp_error("invalid struct/union definition.\n");
    }

    id = ctype_lookup_csymbol_id(stname);
    if (id >= 0) {
        assert(cp_id_to_csym(id)->type == FFI_STRUCT ||
                cp_id_to_csym(id)->type == FFI_UNION);
        assert(csym_struct(cp_id_to_csym(id))->memb_nr == -1);
    }

    memb_size = cts.top - start_top;
    st_membs = malloc(memb_size*sizeof(struct_member));
    if (!st_membs)
        cp_error("failed to allocate memory for struct members.\n");
    memset(st_membs, 0, memb_size*sizeof(struct_member));

    if (type == STRUCT_TYPE)
        nst.type = FFI_STRUCT;
    else
        nst.type = FFI_UNION;
    strcpy(nst.name, stname);

    stcs = csym_struct(&nst);
    stcs->align = 0;
    stcs->memb_nr = memb_size;
    stcs->members = st_membs;

    for (i = 0; i < memb_size; i++) {
        cte = ct_stack(i + start_top);
        if (cte->name)
            strcpy(st_membs[i].name, cte->name);
        ct = ct_stack_ct(i + start_top);
        st_membs[i].id = ct->ffi_cs_id;
        if (!ct->is_array)
            st_membs[i].len = -1;
        else
            st_membs[i].len = ct->array_size;
    }

    if (id < 0)
        id = cp_ctype_reg_csymbol(&nst);
    else
        cs_arr[id] = nst;

    ctype_stack_reset(start_top);

    return id;
}

int cp_symbol_build_fake_record(const char *stname, int type)
{
    int id;
    csymbol nst;
    csymbol_struct *stcs;

    if (!stname || (type != STRUCT_TYPE && type != UNION_TYPE)) {
        cp_error("invalid fake struct/union definition.\n");
    }

    id = ctype_lookup_csymbol_id(stname);
    if (id >= 0)
        return id;

    if (type == STRUCT_TYPE)
        nst.type = FFI_STRUCT;
    else
        nst.type = FFI_UNION;
    strcpy(nst.name, stname);

```

```
    stcs = csym_struct(&nst);
    stcs->align = 0;
    stcs->memb_nr = -1;
    stcs->members = NULL;

    id = cp_ctype_reg_csymbol(&nst);

    return id;
}


/* build pointer symbol from given csymbol */
int cp_symbol_build_pointer(struct cp_ctype *ct)
{
    int id, ret;
    csymbol ncspt;
    csymbol *ref_cs = ct_ffi_cs(ct);

    /* TODO: Check correctness of multi-level pointer 24.11.2013(unihorn) */
    memset(&ncspt, 0, sizeof(csymbol));
    ncspt.type = FFI_PTR;
    ret = sprintf(ncspt.name, "%s *", csym_name(ref_cs));
    assert(ret < MAX_TYPE_NAME_LEN);

    csym_set_ptr_deref_id(&ncspt, ct->ffi_cs_id);
    id = cp_ctype_reg_csymbol(&ncspt);

    return id;
}

void __cp_symbol_dump_func(csymbol *cs)
{
    int i;
    csymbol *ncs;
    csymbol_func *fcs = csym_func(cs);

    printf("=== [%s] function definition =============\n", csym_name(cs));
    ncs = cp_csymf_ret(fcs);
    printf("address: %p\n", fcs->addr);
    printf("return type: \n");
    printf("\tcsym_id: %d, ffi_ctype: %d, %s\n",
            fcs->ret_id, ncs->type, csym_name(ncs));
    printf("args type (%d): \n", fcs->arg_nr);
    for (i = 0; i < csymf_arg_nr(fcs); i++) {
        printf("\t (%d) ", i);
        printf("csym_id: %d, ", fcs->arg_ids[i]);
        ncs = cp_csymf_arg(fcs, i);
        printf("ffi_ctype: %d, %s\n", ncs->type, csym_name(ncs));
    }
}

void cp_symbol_dump_func(int id)
{
    __cp_symbol_dump_func(cp_id_to_csym(id));
}

int cp_symbol_build_func(struct cp_ctype *type, const char *fname, int fn_size)
{
    int i = 1, arg_nr, id;
    int *argsym_id_arr;
    csymbol nfcs;
    csymbol_func *fcs;

    if (cts.top == 0 || fn_size < 0 || !fname) {
        cp_error("invalid function definition.\n");
    }

    argsym_id_arr = NULL;
    memset(&nfcs, 0, sizeof(csymbol));
    csym_type(&nfcs) = FFI_FUNC;

    strncpy(csym_name(&nfcs), fname, fn_size);

    fcs = csym_func(&nfcs);
    fcs->has_var_arg = type->has_var_arg;
    /* Type needed for handling variable args handle */
```

```
    if (fcs->has_var_arg && !ctype_lookup_type("void *"))
        cp_symbol_build_pointer(ctype_lookup_type("void"));

    /* Fetch start address of function  */
    fcs->addr = (void *)find_kernel_symbol(csym_name(&nfcs));
    if (!fcs->addr)
        cp_error("wrong function address for %s\n", csym_name(&nfcs));

    /* bottom of the stack is return type */
    fcs->ret_id = ct_stack_ct(0)->ffi_cs_id;

    /* the rest is argument type */
    if (cts.top == 1) {
        /* function takes no argument */
        arg_nr = 0;
    } else {
        arg_nr = cts.top - 1;
        argsym_id_arr = malloc(arg_nr * sizeof(int));
        if (!argsym_id_arr)
            cp_error("failed to allocate memory for function args.\n");
        for (i = 0; i < arg_nr; i++) {
            argsym_id_arr[i] = ct_stack_ct(i+1)->ffi_cs_id;
        }
    }
    fcs->arg_nr = arg_nr;
    fcs->arg_ids = argsym_id_arr;

    id = cp_ctype_reg_csymbol(&nfcs);

    /* clear stack since we have consumed all the ctypes */
    ctype_stack_reset(0);

    return id;
}

struct cp_ctype *cp_ctype_reg_type(char *name, struct cp_ctype *ct)
{
    if (cte_nr >= cte_arr_size)
        ctype_reg_table_grow();

    memset(cte_arr[cte_nr].name, 0, MAX_TYPE_NAME_LEN);
    strcpy(cte_arr[cte_nr].name, name);

    cte_arr[cte_nr].ct = *ct;
    cte_nr++;

    return &(cte_arr[cte_nr-1].ct);
}

#if 0
/* TODO: used for size calculation */
static ffi_type ffi_int_type(ktap_state *ks, int size, bool sign)
{
    switch(size) {
    case 1:
        if (!sign)
            return FFI_UINT8;
        else
            return FFI_INT8;
    case 2:
        if (!sign)
            return FFI_UINT16;
        else
            return FFI_INT16;
    case 4:
        if (!sign)
            return FFI_UINT32;
        else
            return FFI_INT32;
    case 8:
        if (!sign)
            return FFI_UINT64;
        else
            return FFI_INT64;
    default:
        kp_error(ks, "Error: Have not support int type of size %d\n", size);
```

```
        return FFI_UNKNOWN;
    }

    /* NEVER reach here, silence compiler */
    return -1;
}
#endif


static inline void ct_set_type(struct cp_ctype *ct, int type, int is_unsigned)
{
    ct->type = type;
    ct->is_unsigned = is_unsigned;
}

static void init_builtin_type(struct cp_ctype *ct, ffi_type ftype)
{
    csymbol cs;
    int cs_id;

    csym_type(&cs) = ftype;
    strncpy(csym_name(&cs), ffi_type_name(ftype), CSYM_NAME_MAX_LEN);
    cs_id = cp_ctype_reg_csymbol(&cs);

    memset(ct, 0, sizeof(*ct));
    ct->ffi_base_cs_id = ct->ffi_cs_id = cs_id;
    switch (ftype) {
    case FFI_VOID:        ct_set_type(ct, VOID_TYPE, 0); break;
    case FFI_UINT8:        ct_set_type(ct, INT8_TYPE, 1); break;
    case FFI_INT8:        ct_set_type(ct, INT8_TYPE, 0); break;
    case FFI_UINT16:    ct_set_type(ct, INT16_TYPE, 1); break;
    case FFI_INT16:        ct_set_type(ct, INT16_TYPE, 0); break;
    case FFI_UINT32:    ct_set_type(ct, INT32_TYPE, 1); break;
    case FFI_INT32:        ct_set_type(ct, INT32_TYPE, 0); break;
    case FFI_UINT64:    ct_set_type(ct, INT64_TYPE, 1); break;
    case FFI_INT64:        ct_set_type(ct, INT64_TYPE, 0); break;
    default:        break;
    }
    ct->base_size = ffi_type_size(ftype);
    ct->align_mask = ffi_type_align(ftype) - 1;
    ct->is_defined = 1;
}

/*
* lookup and register builtin C type on demand
* You should ensure that the type with name doesn't appear in
* csymbol table before calling.
*/
struct cp_ctype *ctype_lookup_builtin_type(char *name)
{
    struct cp_ctype ct;

    if (!strncmp(name, "void", sizeof("void"))) {
        init_builtin_type(&ct, FFI_VOID);
        return cp_ctype_reg_type("void", &ct);
    } else if (!strncmp(name, "int8_t", sizeof("int8_t"))) {
        init_builtin_type(&ct, FFI_INT8);
        return cp_ctype_reg_type("int8_t", &ct);
    } else if (!strncmp(name, "uint8_t", sizeof("uint8_t"))) {
        init_builtin_type(&ct, FFI_UINT8);
        return cp_ctype_reg_type("uint8_t", &ct);
    } else if (!strncmp(name, "int16_t", sizeof("int16_t"))) {
        init_builtin_type(&ct, FFI_INT16);
        return cp_ctype_reg_type("int16_t", &ct);
    } else if (!strncmp(name, "uint16_t", sizeof("uint16_t"))) {
        init_builtin_type(&ct, FFI_UINT16);
        return cp_ctype_reg_type("uint16_t", &ct);
    } else if (!strncmp(name, "int32_t", sizeof("int32_t"))) {
        init_builtin_type(&ct, FFI_INT32);
        return cp_ctype_reg_type("int32_t", &ct);
    } else if (!strncmp(name, "uint32_t", sizeof("uint32_t"))) {
        init_builtin_type(&ct, FFI_UINT32);
        return cp_ctype_reg_type("uint32_t", &ct);
    } else if (!strncmp(name, "int64_t", sizeof("int64_t"))) {
        init_builtin_type(&ct, FFI_INT64);
        return cp_ctype_reg_type("int64_t", &ct);
```

```
554     } else if (!strncmp(name, "uint64_t", sizeof("uint64_t"))) {
555         init_builtin_type(&ct, FFI_UINT64);
556         return cp_ctype_reg_type("uint64_t", &ct);
557     } else {
558         /* no builtin type matched */
559         return NULL;
560     }
561 }
562 
563 /* start ctype reg table */
564 struct cp_ctype *ctype_lookup_type(char *name)
565 {
566     int i;
567     struct cp_ctype *ct;
568 
569     for (i = 0; i < cte_nr; i++) {
570         ct = &cte_arr[i].ct;
571         if (!strcmp(name, cte_arr[i].name))
572             return ct;
573     }
574 
575     /* see if it's a builtin C type
576      * return NULL if still no match */
577     return ctype_lookup_builtin_type(name);
578 }
579 
580 cp_csymbol_state *ctype_get_csym_state(void)
581 {
582     return &csym_state;
583 }
584 
585 #define DEFAULT_STACK_SIZE 20
586 #define DEFAULT_SYM_ARR_SIZE 20
587 int cp_ctype_init()
588 {
589     cts.size = DEFAULT_STACK_SIZE;
590     cts.top = 0;
591     cts.stack = malloc(sizeof(cp_ctype_entry)*DEFAULT_STACK_SIZE);
592 
593     cs_nr = 0;
594     cs_arr_size = DEFAULT_SYM_ARR_SIZE;
595     cs_arr = malloc(sizeof(csymbol)*DEFAULT_SYM_ARR_SIZE);
596     memset(cs_arr, 0, sizeof(csymbol)*DEFAULT_SYM_ARR_SIZE);
597 
598     cte_nr = 0;
599     cte_arr_size = DEFAULT_CTYPE_ARR_SIZE;
600     cte_arr = malloc(sizeof(cp_ctype_entry)*DEFAULT_CTYPE_ARR_SIZE);
601 
602     return 0;
603 }
604 
605 int cp_ctype_free()
606 {
607     int i;
608     csymbol *cs;
609 
610     if (cts.stack)
611         free(cts.stack);
612 
613     if (cs_arr) {
614         for (i = 0; i < cs_nr; i++) {
615             cs = cp_id_to_csym(i);
616             if (csym_type(cs) == FFI_FUNC) {
617                 if (csym_func(cs)->arg_ids)
618                     free(csym_func(cs)->arg_ids);
619             } else if (csym_type(cs) == FFI_STRUCT) {
620                 if (csym_struct(cs)->members)
621                     free(csym_struct(cs)->members);
622             }
623         }
624         free(cs_arr);
625     }
626 
627     if (cte_arr) {
628         free(cte_arr);
629     }
```

```
630
631     return 0;
632 }
```

One Level Up Top Level

userspace/kp_util.h - ktap

Data types defined

- SBuf
- SBuf
- bool
- cp_csymbol_state
- cp_csymbol_state
- ktap_writer

Macros defined

- KP_CHAR_ALNUM
- KP_CHAR_ALPHA
- KP_CHAR_CNTRL
- KP_CHAR_DIGIT
- KP_CHAR_GRAPH
- KP_CHAR_IDENT
- KP_CHAR_LOWER
- KP_CHAR_PUNCT
- KP_CHAR_SPACE
- KP_CHAR_UPPER
- KP_CHAR_XDIGIT
- __KTAP_UTIL_H__
- err2msg
- false
- kp_char_isa
- kp_char_isalnum
- kp_char_isalpha
- kp_char_iscntrl
- kp_char_isdigit
- kp_char_isgraph
- kp_char_isident
- kp_char_islower

- kp_char_ispunct
- kp_char_isspace
- kp_char_isupper
- kp_char_isxdigit
- kp_char_tolower
- kp_char_toupper
- sbufB
- sbufE
- sbufP
- sbufleft
- sbuflen
- sbufsz
- setsbufP
- true
- verbose_printf

Source code

```
#ifndef __KTAP_UTIL_H__
#define __KTAP_UTIL_H__

#include <stdarg.h>

#include "../include/ktap_bc.h"
#include "../include/ktap_err.h"

typedef int bool;
 #define false 0
 #define true 1

 /* Resizable string buffer. */
 typedef struct SBuf {
     char *p; /* String buffer pointer. */
     char *e; /* String buffer end pointer. */
     char *b; /* String buffer base. */
 } SBuf;

 /* Resizable string buffers. Struct definition in kp_obj.h. */
 #define sbufB(sb)    ((char *)(sb)->b)
 #define sbufP(sb)    ((char *)(sb)->p)
 #define sbufE(sb)    ((char *)(sb)->e)
 #define sbufsz(sb)    ((int)(sbufE((sb)) - sbufB((sb))))
 #define sbuflen(sb)    ((int)(sbufP((sb)) - sbufB((sb))))
 #define sbufleft(sb)    ((int)(sbufE((sb)) - sbufP((sb))))
 #define setsbufP(sb, q) ((sb)->p = (q))

 void kp_buf_init(SBuf *sb);
 void kp_buf_reset(SBuf *sb);
 void kp_buf_free(SBuf *sb);
 char *kp_buf_more(SBuf *sb, int sz);
 char *kp_buf_need(SBuf *sb, int sz);
 char *kp_buf_wmem(char *p, const void *q, int len);
 void kp_buf_putb(SBuf *sb, int c);
 ktap_str_t *kp_buf_str(SBuf *sb);


```

```
#define KP_CHAR_CNTRL    0x01
#define KP_CHAR_SPACE    0x02
#define KP_CHAR_PUNCT    0x04
#define KP_CHAR_DIGIT    0x08
#define KP_CHAR_XDIGIT    0x10
#define KP_CHAR_UPPER    0x20
#define KP_CHAR_LOWER    0x40
#define KP_CHAR_IDENT    0x80
#define KP_CHAR_ALPHA    (KP_CHAR_LOWER|KP_CHAR_UPPER)
#define KP_CHAR_ALNUM    (KP_CHAR_ALPHA|KP_CHAR_DIGIT)
#define KP_CHAR_GRAPH    (KP_CHAR_ALNUM|KP_CHAR_PUNCT)

/* Only pass -1 or 0..255 to these macros. Never pass a signed char! */
#define kp_char_isa(c, t)    ((kp_char_bits+1)[(c)] & t)
#define kp_char_iscntrl(c)    kp_char_isa((c), KP_CHAR_CNTRL)
#define kp_char_isspace(c)    kp_char_isa((c), KP_CHAR_SPACE)
#define kp_char_ispunct(c)    kp_char_isa((c), KP_CHAR_PUNCT)
#define kp_char_isdigit(c)    kp_char_isa((c), KP_CHAR_DIGIT)
#define kp_char_isxdigit(c)    kp_char_isa((c), KP_CHAR_XDIGIT)
#define kp_char_isupper(c)    kp_char_isa((c), KP_CHAR_UPPER)
#define kp_char_islower(c)    kp_char_isa((c), KP_CHAR_LOWER)
#define kp_char_isident(c)    kp_char_isa((c), KP_CHAR_IDENT)
#define kp_char_isalpha(c)    kp_char_isa((c), KP_CHAR_ALPHA)
#define kp_char_isalnum(c)    kp_char_isa((c), KP_CHAR_ALNUM)
#define kp_char_isgraph(c)    kp_char_isa((c), KP_CHAR_GRAPH)

#define kp_char_toupper(c)    ((c) - (kp_char_islower(c) >> 1))
#define kp_char_tolower(c)    ((c) + kp_char_isupper(c))

extern const char *kp_err_allmsg;
#define err2msg(em)     (kp_err_allmsg+(int)(em))

extern const uint8_t kp_char_bits[257];


char *strfmt_wuleb128(char *p, uint32_t v);
void kp_err_lex(ktap_str_t *src, const char *tok, BCLine line,
       ErrMsg em, va_list argp);
char *kp_sprintf(const char *fmt, ...);
const char *kp_sprintfv(const char *fmt, va_list argp);

void *kp_reallocv(void *block, size_t osize, size_t nsize);

void kp_str_resize(void);
ktap_str_t *kp_str_newz(const char *str);
ktap_str_t *kp_str_new(const char *str, size_t l);

ktap_tab_t *kp_tab_new();
const ktap_val_t *kp_tab_get(ktap_tab_t *t, const ktap_val_t *key);
const ktap_val_t *kp_tab_getstr(ktap_tab_t *t, const ktap_str_t *ts);
void kp_tab_setvalue(ktap_tab_t *t, const ktap_val_t *key, ktap_val_t *val);
ktap_val_t *kp_tab_set(ktap_tab_t *t, const ktap_val_t *key);

int kp_obj_equal(const ktap_val_t *t1, const ktap_val_t *t2);

bool strglobmatch(const char *str, const char *pat);
int kallsyms_parse(void *arg,
           int(*process_symbol)(void *arg, const char *name,
           char type, unsigned long start));

unsigned long find_kernel_symbol(const char *symbol);
 void list_available_events(const char *match);
 void process_available_tracepoints(const char *sys, const char *event,
                    int (*process)(const char *sys,
                           const char *event));
 int kallsyms_parse(void *arg,
                    int(*process_symbol)(void *arg, const char *name,
                    char type, unsigned long start));


 #ifdef CONFIG_KTAP_FFI
 #include "../include/ktap_ffi.h"

 typedef struct cp_csymbol_state {
     int cs_nr; /* number of c symbols */
     int cs_arr_size; /* size of current symbol arrays */
```

```
115     csymbol *cs_arr;
116 } cp_csymbol_state;
117 
118 cp_csymbol_state *ctype_get_csym_state(void);
119 void kp_dump_csymbols();
120 #endif
121 
122 ktap_eventdesc_t *kp_parse_events(const char *eventdef);
123 void cleanup_event_resources(void);
124 
125 extern int verbose;
126 #define verbose_printf(...) \
127     if (verbose)    \
128         printf("[verbose] " __VA_ARGS__);
129 
130 
131 void kp_dump_proto(ktap_proto_t *pt);
132 typedef int (*ktap_writer)(const void* p, size_t sz, void* ud);
133 int kp_bcwrite(ktap_proto_t *pt, ktap_writer writer, void *data, int strip);
134 
135 int kp_create_reader(const char *output);
136 #endif
```

One Level Up Top Level

userspace/kp_util.c - ktap

Global variables defined

- kp_char_bits
- kp_err_allmsg
- kp_niltv
- strtab
- strtab_nr
- strtab_size

Data types defined

- ksym_addr_t

Functions defined

- __match_charclass
- __match_glob
- createstrobj
- find_kernel_symbol
- kallsyms_parse
- kp_buf_free
- kp_buf_init
- kp_buf_more
- kp_buf_need
- kp_buf_putb
- kp_buf_reset
- kp_buf_str
- kp_buf_wmem
- kp_err_lex
- kp_obj_equal
- kp_reallocv
- kp_sprintf
- kp_sprintfv
- kp_str_new
- kp_str_newz

- kp_str_resize
- kp_tab_get
- kp_tab_getstr
- kp_tab_new
- kp_tab_set
- kp_tab_setvalue
- list_available_events
- process_available_tracepoints
- strfmt_wuleb128
- strglobmatch
- stringtable_insert
- stringtable_search
- symbol_cmp

Macros defined

- AVAILABLE_EVENTS_PATH
- ERRDEF
- KALLSYMS_PATH
- gkey
- gnode
- gval
- handle_error
- niltv

Source code

```
/*
 * util.c
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * Adapted from luajit and lua interpreter.
 * Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
```

```
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include "../include/ktap_types.h"
#include "../include/ktap_bc.h"
#include "kp_util.h"

/* Error message strings. */
const char *kp_err_allmsg =
#define ERRDEF(name, msg)       msg "\0"
#include "../include/ktap_errmsg.h"
;

const uint8_t kp_char_bits[257] = {
    0,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  3,  3,  3,  3,  3,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
    2,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
  152,152,152,152,152,152,152,152,152,152,  4,  4,  4,  4,  4,  4,
    4,176,176,176,176,176,176,160,160,160,160,160,160,160,160,160,
  160,160,160,160,160,160,160,160,160,160,160,  4,  4,  4,  4,132,
    4,208,208,208,208,208,208,192,192,192,192,192,192,192,192,192,
  192,192,192,192,192,192,192,192,192,192,192,  4,  4,  4,  4,  1,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128
};

void kp_buf_init(SBuf *sb)
{
    sb->b = (char *)malloc(200);
    sb->p = NULL;
    sb->e = sb->b + 200;
}

void kp_buf_reset(SBuf *sb)
{
    sb->p = sb->b;
}

void kp_buf_free(SBuf *sb)
{
    free(sbufB(sb));
}

char *kp_buf_more(SBuf *sb, int sz)
{
    char *b;
    int old_len = sbuflen(sb);

    if (sz > sbufleft(sb)) {
        b = realloc(sbufB(sb), sbuflen(sb) * 2);
        sb->b = b;
        sb->p = b + old_len;
        sb->e = b + old_len * 2;
    }

    return sbufP(sb);
}

char *kp_buf_need(SBuf *sb, int sz)
{
    char *b;
    int old_len = sbuflen(sb);
```

```
98
99      if (sz > sbufsz(sb)) {
100         b = realloc(sbufB(sb), sz);
101         sb->b = b;
102         sb->p = b + old_len;
103         sb->e = b + sz;
104     }
105
106     return sbufB(sb);
107 }
108
109 char *kp_buf_wmem(char *p, const void *q, int len)
110 {
111     return (char *)memcpy(p, q, len) + len;
112 }
113
114 void kp_buf_putb(SBuf *sb, int c)
115 {
116     char *p = kp_buf_more(sb, 1);
117     *p++ = (char)c;
118     setsbufP(sb, p);
119 }
120
121 ktap_str_t *kp_buf_str(SBuf *sb)
122 {
123     return kp_str_new(sbufB(sb), sbuflen(sb));
124 }
125
126 /* Write ULEB128 to buffer. */
127 char *strfmt_wuleb128(char *p, uint32_t v)
128 {
129   for (; v >= 0x80; v >>= 7)
130     *p++ = (char)((v & 0x7f) | 0x80);
131   *p++ = (char)v;
132   return p;
133 }
134
135 void kp_err_lex(ktap_str_t *src, const char *tok, BCLine line,
136         ErrMsg em, va_list argp)
137 {
138     const char *msg;
139
140     msg = kp_sprintfv(err2msg(em), argp);
141     msg = kp_sprintf("%s:%d: %s", getstr(src), line, msg);
142     if (tok)
143         msg = kp_sprintf(err2msg(KP_ERR_XNEAR), msg, tok);
144     fprintf(stderr, "%s: %s\n", err2msg(KP_ERR_XSYNTAX), msg);
145     exit(-1);
146 }
147
148 void *kp_reallocv(void *block, size_t osize, size_t nsize)
149 {
150     return realloc(block, nsize);
151 }
152
153 static const ktap_val_t kp_niltv = { {NULL}, {KTAP_TNIL} } ;
154 #define niltv  (&kp_niltv)
155
156 #define gnode(t,i)    (&(t)->node[i])
157 #define gkey(n)        (&(n)->key)
158 #define gval(n)        (&(n)->val)
159
160 const ktap_val_t *kp_tab_get(ktap_tab_t *t, const ktap_val_t *key)
161 {
162     int i;
163
164     switch (itype(key)) {
165     case KTAP_TNIL:
166         return niltv;
167     case KTAP_TNUM:
168         for (i = 0; i <= t->hmask; i++) {
169             ktap_val_t *v = gkey(gnode(t, i));
170             if (is_number(v) && nvalue(key) == nvalue(v))
171                 return gval(gnode(t, i));
172         }
173         break;
```

```
    case KTAP_TSTR:
        for (i = 0; i <= t->hmask; i++) {
            ktap_val_t *v = gkey(gnode(t, i));
            if (is_string(v) && (rawtsvalue(key) == rawtsvalue(v)))
                return gval(gnode(t, i));
        }
        break;
    default:
        for (i = 0; i <= t->hmask; i++) {
            if (kp_obj_equal(key, gkey(gnode(t, i))))
                return gval(gnode(t, i));
        }
        break;
    }

    return niltv;
}

const ktap_val_t *kp_tab_getstr(ktap_tab_t *t, const ktap_str_t *ts)
{
    int i;

    for (i = 0; i <= t->hmask; i++) {
        ktap_val_t *v = gkey(gnode(t, i));
        if (is_string(v) && (ts == rawtsvalue(v)))
            return gval(gnode(t, i));
    }

    return niltv;
}

void kp_tab_setvalue(ktap_tab_t *t, const ktap_val_t *key, ktap_val_t *val)
{
    const ktap_val_t *v = kp_tab_get(t, key);

    if (v != niltv) {
        set_obj((ktap_val_t *)v, val);
    } else {
        if (t->freetop == t->node) {
            int size = (t->hmask + 1) * sizeof(ktap_node_t);
            t->node = realloc(t->node, size * 2);
            memset(t->node + t->hmask + 1, 0, size);
            t->freetop = t->node + (t->hmask + 1) * 2;
            t->hmask = (t->hmask + 1) * 2 - 1;
        }

        ktap_node_t *n = --t->freetop;
        set_obj(gkey(n), key);
        set_obj(gval(n), val);
    }
}

ktap_val_t *kp_tab_set(ktap_tab_t *t, const ktap_val_t *key)
{
    const ktap_val_t *v = kp_tab_get(t, key);

    if (v != niltv) {
        return (ktap_val_t *)v;
    } else {
        if (t->freetop == t->node) {
            int size = (t->hmask + 1) * sizeof(ktap_node_t);
            t->node = realloc(t->node, size * 2);
            memset(t->node + t->hmask + 1, 0, size);
            t->freetop = t->node + (t->hmask + 1) * 2;
            t->hmask = (t->hmask + 1) * 2 - 1;
        }

        ktap_node_t *n = --t->freetop;
        set_obj(gkey(n), key);
        set_nil(gval(n));
        return gval(n);
    }
}


ktap_tab_t *kp_tab_new(void)
```

```
{
    int hsize, i;

    ktap_tab_t *t = malloc(sizeof(ktap_tab_t));
    t->gct = ~KTAP_TTAB;
    hsize = 1024;
    t->hmask = hsize - 1;
    t->node = (ktap_node_t *)malloc(hsize * sizeof(ktap_node_t));
    t->freetop = &t->node[hsize];
    t->asize = 0;

    for (i = 0; i <= t->hmask; i++) {
        set_nil(&t->node[i].val);
        set_nil(&t->node[i].key);
    }
    return t;
}

/* simple interned string array, use hash table in future  */
static ktap_str_t **strtab;
static int strtab_size = 1000; /* initial size */
static int strtab_nr;

void kp_str_resize(void)
{
    int size = strtab_size * sizeof(ktap_str_t *);

    strtab = malloc(size);
    if (!strtab) {
        fprintf(stderr, "cannot allocate stringtable\n");
        exit(-1);
    }

    memset(strtab, 0, size);
    strtab_nr = 0;
}

static ktap_str_t *stringtable_search(const char *str, int len)
{
    int i;

    for (i = 0; i < strtab_nr; i++) {
        ktap_str_t *s = strtab[i];
        if ((len == s->len) && !memcmp(str, getstr(s), len))
            return s;
    }

    return NULL;
}

static void stringtable_insert(ktap_str_t *ts)
{
    strtab[strtab_nr++] = ts;

    if (strtab_nr == strtab_size) {
        int size = strtab_size * sizeof(ktap_str_t *);
        strtab = realloc(strtab, size * 2);
        memset(strtab + strtab_size, 0, size);
        strtab_size *= 2;
    }
}

static ktap_str_t *createstrobj(const char *str, size_t l)
{
    ktap_str_t *ts;
    size_t totalsize;  /* total size of TString object */

    totalsize = sizeof(ktap_str_t) + ((l + 1) * sizeof(char));
    ts = (ktap_str_t *)malloc(totalsize);
    ts->gct = ~KTAP_TSTR;
    ts->len = l;
    ts->reserved = 0;
    ts->extra = 0;
    memcpy(ts + 1, str, l * sizeof(char));
    ((char *)(ts + 1))[l] = '\0';  /* ending 0 */
    return ts;
```

```
}

ktap_str_t *kp_str_new(const char *str, size_t l)
{
    ktap_str_t *ts = stringtable_search(str, l);

    if (ts)
        return ts;

    ts = createstrobj(str, l);
    stringtable_insert(ts);
    return ts;
}

ktap_str_t *kp_str_newz(const char *str)
{
    return kp_str_new(str, strlen(str));
}

/*
* todo: memory leak here
*/
char *kp_sprintf(const char *fmt, ...)
{
    char *msg = malloc(128);

    va_list argp;
    va_start(argp, fmt);
    vsprintf(msg, fmt, argp);
    va_end(argp);
    return msg;
}

const char *kp_sprintfv(const char *fmt, va_list argp)
{
    char *msg = malloc(128);

    vsprintf(msg, fmt, argp);
    return msg;
}

int kp_obj_equal(const ktap_val_t *t1, const ktap_val_t *t2)
{
    switch (itype(t1)) {
    case KTAP_TNIL:
        return 1;
    case KTAP_TNUM:
        return nvalue(t1) == nvalue(t2);
    case KTAP_TTRUE:
    case KTAP_TFALSE:
        return itype(t1) == itype(t2);
    case KTAP_TLIGHTUD:
        return pvalue(t1) == pvalue(t2);
    case KTAP_TFUNC:
        return fvalue(t1) == fvalue(t2);
    case KTAP_TSTR:
        return rawtsvalue(t1) == rawtsvalue(t2);
    default:
        return gcvalue(t1) == gcvalue(t2);
    }

    return 0;
}

/*
* strglobmatch is copyed from perf(linux/tools/perf/util/string.c)
*/

/* Character class matching */
static bool __match_charclass(const char *pat, char c, const char **npat)
{
    bool complement = false, ret = true;

    if (*pat == '!') {
        complement = true;
        pat++;
```

```
    }
    if (*pat++ == c)    /* First character is special */
        goto end;

    while (*pat && *pat != ']') {    /* Matching */
        if (*pat == '-' && *(pat + 1) != ']') {    /* Range */
            if (*(pat - 1) <= c && c <= *(pat + 1))
                goto end;
            if (*(pat - 1) > *(pat + 1))
                goto error;
            pat += 2;
        } else if (*pat++ == c)
            goto end;
    }
    if (!*pat)
        goto error;
    ret = false;

end:
    while (*pat && *pat != ']')    /* Searching closing */
        pat++;
    if (!*pat)
        goto error;
    *npat = pat + 1;
    return complement ? !ret : ret;

error:
    return false;
}

/* Glob/lazy pattern matching */
static bool __match_glob(const char *str, const char *pat, bool ignore_space)
{
    while (*str && *pat && *pat != '*') {
        if (ignore_space) {
            /* Ignore spaces for lazy matching */
            if (isspace(*str)) {
                str++;
                continue;
            }
            if (isspace(*pat)) {
                pat++;
                continue;
            }
        }
        if (*pat == '?') {    /* Matches any single character */
            str++;
            pat++;
            continue;
        } else if (*pat == '[')    /* Character classes/Ranges */
            if (__match_charclass(pat + 1, *str, &pat)) {
                str++;
                continue;
            } else
                return false;
        else if (*pat == '\\') /* Escaped char match as normal char */
            pat++;
        if (*str++ != *pat++)
            return false;
    }
    /* Check wild card */
    if (*pat == '*') {
        while (*pat == '*')
            pat++;
        if (!*pat)    /* Tail wild card matches all */
            return true;
        while (*str)
            if (__match_glob(str++, pat, ignore_space))
                return true;
    }
    return !*str && !*pat;
}

/**
 * strglobmatch - glob expression pattern matching
 * @str: the target string to match
```

```
 * @pat: the pattern string to match
 *
 * This returns true if the @str matches @pat. @pat can includes wildcards
 * ('*','?') and character classes ([CHARS], complementation and ranges are
 * also supported). Also, this supports escape character ('\') to use special
 * characters as normal character.
 *
 * Note: if @pat syntax is broken, this always returns false.
 */
bool strglobmatch(const char *str, const char *pat)
{
    return __match_glob(str, pat, false);
}

#define handle_error(str) do { perror(str); exit(-1); } while(0)

#define KALLSYMS_PATH "/proc/kallsyms"
/*
* read kernel symbol from /proc/kallsyms
*/
int kallsyms_parse(void *arg,
           int(*process_symbol)(void *arg, const char *name,
           char type, unsigned long start))
{
    FILE *file;
    char *line = NULL;
    int ret = 0;
    int found = 0;

    file = fopen(KALLSYMS_PATH, "r");
    if (file == NULL)
        handle_error("open " KALLSYMS_PATH " failed");

    while (!feof(file)) {
        char *symbol_addr, *symbol_name;
        char symbol_type;
        unsigned long start;
        int line_len;
        size_t n;

        line_len = getline(&line, &n, file);
        if (line_len < 0 || !line)
            break;

        line[--line_len] = '\0'; /* \n */

        symbol_addr = strtok(line, " \t");
        start = strtoul(symbol_addr, NULL, 16);

        symbol_type = *strtok(NULL, " \t");
        symbol_name = strtok(NULL, " \t");

        ret = process_symbol(arg, symbol_name, symbol_type, start);
        if (!ret)
            found = 1;
    }

    free(line);
    fclose(file);

    return found;
}

struct ksym_addr_t {
    const char *name;
    unsigned long addr;
};

static int symbol_cmp(void *arg, const char *name, char type,
              unsigned long start)
{
    struct ksym_addr_t *base = arg;

    if (strcmp(base->name, name) == 0) {
        base->addr = start;
        return 1;
```

```
554     }
555
556     return 0;
557 }
558
559 unsigned long find_kernel_symbol(const char *symbol)
560 {
561     int ret;
562     struct ksym_addr_t arg = {
563         .name = symbol,
564         .addr = 0
565     };
566
567     ret = kallsyms_parse(&arg, symbol_cmp);
568     if (ret < 0 || arg.addr == 0) {
569         fprintf(stderr, "cannot read kernel symbol \"%s\" in %s\n",
570             symbol, KALLSYMS_PATH);
571         exit(EXIT_FAILURE);
572     }
573
574     return arg.addr;
575 }
576
577
578 #define AVAILABLE_EVENTS_PATH "/sys/kernel/debug/tracing/available_events"
579
580 void list_available_events(const char *match)
581 {
582     FILE *file;
583     char *line = NULL;
584
585     file = fopen(AVAILABLE_EVENTS_PATH, "r");
586     if (file == NULL)
587         handle_error("open " AVAILABLE_EVENTS_PATH " failed");
588
589     while (!feof(file)) {
590         int line_len;
591         size_t n;
592
593         line_len = getline(&line, &n, file);
594         if (line_len < 0 || !line)
595             break;
596
597         if (!match || strglobmatch(line, match))
598             printf("%s", line);
599     }
600
601     free(line);
602     fclose(file);
603 }
604
605 void process_available_tracepoints(const char *sys, const char *event,
606                    int (*process)(const char *sys,
607                           const char *event))
608 {
609     char *line = NULL;
610     FILE *file;
611     char str[128] = {0};
612
613     /* add '\n' into tail */
614     snprintf(str, 64, "%s:%s\n", sys, event);
615
616     file = fopen(AVAILABLE_EVENTS_PATH, "r");
617     if (file == NULL)
618         handle_error("open " AVAILABLE_EVENTS_PATH " failed");
619
620     while (!feof(file)) {
621         int line_len;
622         size_t n;
623
624         line_len = getline(&line, &n, file);
625         if (line_len < 0 || !line)
626             break;
627
628         if (strglobmatch(line, str)) {
629             char match_sys[64] = {0};
```

```
630             char match_event[64] = {0};
631             char *sep;
632 
633             sep = strchr(line, ':');
634             memcpy(match_sys, line, sep - line);
635             memcpy(match_event, sep + 1,
636                         line_len - (sep - line) - 2);
637 
638             if (process(match_sys, match_event))
639                 break;
640         }
641     }
642 
643     free(line);
644     fclose(file);
645 }
646 
```

One Level Up Top Level

runtime/kp_str.c - ktap

Functions defined

Macros defined

Source code

```
/*
 * kp_str.c - ktap string data struction manipulation
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * Adapted from luajit and lua interpreter.
 * Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
```

```
 */

#include "../include/ktap_types.h"
#include "kp_obj.h"
#include "kp_str.h"
#include "kp_mempool.h"

#include <linux/ctype.h>
#include <linux/module.h>
#include <linux/kallsyms.h>
#include "ktap.h"
#include "kp_transport.h"
#include "kp_vm.h"
#include "kp_events.h"

int kp_str_cmp(const ktap_str_t *ls, const ktap_str_t *rs)
{
    const char *l = getstr(ls);
    size_t ll = ls->len;
    const char *r = getstr(rs);
    size_t lr = rs->len;

    for (;;) {
        int temp = strcmp(l, r);
        if (temp != 0)
            return temp;
        else {
            /* strings are equal up to a `\0' */

            /* index of first `\0' in both strings */
            size_t len = strlen(l);

            /* r is finished? */
            if (len == lr)
                return (len == ll) ? 0 : 1;
            else if (len == ll)  /* l is finished? */
                return -1;

            /*
             * both strings longer than `len';
             * go on comparing (after the `\0')
             */
            len++;
            l += len; ll -= len; r += len; lr -= len;
        }
    }
}

/* Fast string data comparison. Caveat: unaligned access to 1st string! */
static __always_inline int str_fastcmp(const char *a, const char *b, int len)
{
    int i = 0;

    kp_assert(len > 0);
    kp_assert((((uintptr_t)a + len - 1)&(PAGE_SIZE - 1)) <= PAGE_SIZE - 4);

    do {  /* Note: innocuous access up to end of string + 3. */
        uint32_t v = *(uint32_t *)(a + i) ^ *(const uint32_t *)(b + i);
        if (v) {
            i -= len;
#if KP_LE
            return (int32_t)i >= -3 ? (v << (32 + (i << 3))) : 1;
#else
            return (int32_t)i >= -3 ? (v >> (32 + (i << 3))) : 1;
#endif
        }
        i += 4;
    } while (i < len);
    return 0;
}


//TODO: change hash algo

#define STRING_HASHLIMIT    5
static __always_inline unsigned int kp_str_hash(const char *str, size_t len)
```

```
{
    unsigned int h = 201236 ^ len;
    size_t step = (len >> STRING_HASHLIMIT) + 1;
    size_t l1;

    for (l1 = len; l1 >= step; l1 -= step)
        h = h ^ ((h<<5) + (h>>2) + (u8)(str[l1 - 1]));

    return h;
}


/*
* resizes the string table
*/
int kp_str_resize(ktap_state_t *ks, int newmask)
{
    ktap_global_state_t *g = G(ks);
    ktap_str_t **newhash;

    newhash = kp_zalloc(ks, (newmask + 1) * sizeof(ktap_str_t *));
    if (!newhash)
        return -ENOMEM;

    g->strmask = newmask;
    g->strhash = newhash;
    return 0;
}

/*
* Intern a string and return string object.
*/
ktap_str_t * kp_str_new(ktap_state_t *ks, const char *str, size_t len)
{
    ktap_global_state_t *g = G(ks);
    ktap_str_t *s;
    ktap_obj_t *o;
    unsigned int h = kp_str_hash(str, len);
    unsigned long flags;

    if (len >= KP_MAX_STR)
        return NULL;

    local_irq_save(flags);
    arch_spin_lock(&g->str_lock);

    o = (ktap_obj_t *)g->strhash[h & g->strmask];
    if (likely((((uintptr_t)str+len-1) & (PAGE_SIZE-1)) <= PAGE_SIZE-4)) {
        while (o != NULL) {
            ktap_str_t *sx = (ktap_str_t *)o;
            if (sx->len == len &&
                !str_fastcmp(str, getstr(sx), len)) {
                arch_spin_unlock(&g->str_lock);
                local_irq_restore(flags);
                return sx; /* Return existing string. */
            }
            o = gch(o)->nextgc;
        }
    } else { /* Slow path: end of string is too close to a page boundary */
        while (o != NULL) {
            ktap_str_t *sx = (ktap_str_t *)o;
            if (sx->len == len &&
                !memcmp(str, getstr(sx), len)) {
                arch_spin_unlock(&g->str_lock);
                local_irq_restore(flags);
                return sx; /* Return existing string. */
            }
            o = gch(o)->nextgc;
        }
    }

    /* create a new string, allocate it from mempool, not use kmalloc. */
    s = kp_mempool_alloc(ks, sizeof(ktap_str_t) + len + 1);
    if (unlikely(!s))
        goto out;
    s->gct = ~KTAP_TSTR;
```

```
    s->len = len;
    s->hash = h;
    s->reserved = 0;
    memcpy(s + 1, str, len);
    ((char *)(s + 1))[len] = '\0';  /* ending 0 */

    /* Add it to string hash table */
    h &= g->strmask;
    s->nextgc = (ktap_obj_t *)g->strhash[h];
    g->strhash[h] = s;
    if (g->strnum++ > KP_MAX_STRNUM) {
        kp_error(ks, "exceed max string number %d\n", KP_MAX_STRNUM);
        s = NULL;
    }

out:
    arch_spin_unlock(&g->str_lock);
    local_irq_restore(flags);
    return s; /* Return newly interned string. */
}

void kp_str_freeall(ktap_state_t *ks)
{
    /* don't need to free string in here, it will handled by mempool */
    kp_free(ks, G(ks)->strhash);
}

/* kp_str_fmt - printf implementation */

/* macro to `unsign' a character */
#define uchar(c)    ((unsigned char)(c))

#define L_ESC        '%'

/* valid flags in a format specification */
#define FLAGS    "-+ #0"

#define INTFRMLEN    "ll"
#define INTFRM_T    long long

/*
* maximum size of each format specification (such as '%-099.99d')
* (+10 accounts for %99.99x plus margin of error)
*/
#define MAX_FORMAT    (sizeof(FLAGS) + sizeof(INTFRMLEN) + 10)

static const char *scanformat(ktap_state_t *ks, const char *strfrmt, char *form)
{
    const char *p = strfrmt;
    while (*p != '\0' && strchr(FLAGS, *p) != NULL)
        p++;  /* skip flags */

    if ((size_t)(p - strfrmt) >= sizeof(FLAGS)/sizeof(char)) {
        kp_error(ks, "invalid format (repeated flags)\n");
        return NULL;
    }

    if (isdigit(uchar(*p)))
        p++;  /* skip width */

    if (isdigit(uchar(*p)))
        p++;  /* (2 digits at most) */

    if (*p == '.') {
        p++;
        if (isdigit(uchar(*p)))
            p++;  /* skip precision */
        if (isdigit(uchar(*p)))
            p++;  /* (2 digits at most) */
    }

    if (isdigit(uchar(*p))) {
        kp_error(ks, "invalid format (width or precision too long)\n");
        return NULL;
    }

```

```
252     *(form++) = '%';
253     memcpy(form, strfrmt, (p - strfrmt + 1) * sizeof(char));
254     form += p - strfrmt + 1;
255     *form = '\0';
256     return p;
257 }
258
259
260 /*
261 * add length modifier into formats
262 */
263 static void addlenmod(char *form, const char *lenmod)
264 {
265     size_t l = strlen(form);
266     size_t lm = strlen(lenmod);
267     char spec = form[l - 1];
268
269     strcpy(form + l - 1, lenmod);
270     form[l + lm - 1] = spec;
271     form[l + lm] = '\0';
272 }
273
274
275 static void arg_error(ktap_state_t *ks, int narg, const char *extramsg)
276 {
277     kp_error(ks, "bad argument #%d: (%s)\n", narg, extramsg);
278 }
279
280 int kp_str_fmt(ktap_state_t *ks, struct trace_seq *seq)
281 {
282     int arg = 1;
283     size_t sfl;
284     ktap_val_t *arg_fmt = kp_arg(ks, 1);
285     int argnum = kp_arg_nr(ks);
286     const char *strfrmt, *strfrmt_end;
287
288     strfrmt = svalue(arg_fmt);
289     sfl = rawtsvalue(arg_fmt)->len;
290     strfrmt_end = strfrmt + sfl;
291
292     while (strfrmt < strfrmt_end) {
293         if (*strfrmt != L_ESC)
294             trace_seq_putc(seq, *strfrmt++);
295         else if (*++strfrmt == L_ESC)
296             trace_seq_putc(seq, *strfrmt++);
297         else { /* format item */
298             char form[MAX_FORMAT];
299
300             if (++arg > argnum) {
301                 arg_error(ks, arg, "no value");
302                 return -1;
303             }
304
305             strfrmt = scanformat(ks, strfrmt, form);
306             switch (*strfrmt++) {
307             case 'c':
308                 kp_arg_checknumber(ks, arg);
309
310                 trace_seq_printf(seq, form,
311                         nvalue(kp_arg(ks, arg)));
312                 break;
313             case 'd':  case 'i': {
314                 ktap_number n;
315                 INTFRM_T ni;
316
317                 kp_arg_checknumber(ks, arg);
318
319                 n = nvalue(kp_arg(ks, arg));
320                 ni = (INTFRM_T)n;
321                 addlenmod(form, INTFRMLEN);
322                 trace_seq_printf(seq, form, ni);
323                 break;
324             }
325             case 'p': {
326                 char str[KSYM_SYMBOL_LEN];
327
```

```
                kp_arg_checknumber(ks, arg);

                SPRINT_SYMBOL(str, nvalue(kp_arg(ks, arg)));
                _trace_seq_puts(seq, str);
                break;
            }
            case 'o':  case 'u':  case 'x':  case 'X': {
                ktap_number n;
                unsigned INTFRM_T ni;

                kp_arg_checknumber(ks, arg);

                n = nvalue(kp_arg(ks, arg));
                ni = (unsigned INTFRM_T)n;
                addlenmod(form, INTFRMLEN);
                trace_seq_printf(seq, form, ni);
                break;
            }
            case 's': {
                ktap_val_t *v = kp_arg(ks, arg);
                const char *s;
                size_t l;

                if (is_nil(v)) {
                    _trace_seq_puts(seq, "nil");
                    return 0;
                }

                if (is_eventstr(v)) {
                    const char *str = kp_event_tostr(ks);
                    if (!str)
                        return  -1;
                    _trace_seq_puts(seq,
                        kp_event_tostr(ks));
                    return 0;
                }

                kp_arg_checkstring(ks, arg);

                s = svalue(v);
                l = rawtsvalue(v)->len;
                if (!strchr(form, '.') && l >= 100) {
                    /*
                     * no precision and string is too long
                     * to be formatted;
                     * keep original string
                     */
                    _trace_seq_puts(seq, s);
                    break;
                } else {
                    trace_seq_printf(seq, form, s);
                    break;
                }
            }
            default: /* also treat cases `pnLlh' */
                kp_error(ks, "invalid option " KTAP_QL("%%%c")
                         " to " KTAP_QL("format"),
                         *(strfrmt - 1));
                return -1;
            }
        }
    }

    return 0;
}

```

One Level Up Top Level

runtime/kp_obj.c - ktap

Global variables defined

- kp_err_allmsg

Functions defined

- kp_free
- kp_malloc
- kp_obj_dump
- kp_obj_free_gclist
- kp_obj_freeall
- kp_obj_kstack2str
- kp_obj_len
- kp_obj_new
- kp_obj_rawequal
- kp_obj_show
- kp_zalloc

Macros defined

- ERRDEF
- KTAP_ALLOC_FLAGS

Source code

```
1  /*
2  * kp_obj.c - ktap object generic operation
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * Adapted from luajit and lua interpreter.
9  * Copyright (C) 2005-2014 Mike Pall.
10 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
11 *
12 * ktap is free software; you can redistribute it and/or modify it
13 * under the terms and conditions of the GNU General Public License,
14 * version 2, as published by the Free Software Foundation.
15 *
16 * ktap is distributed in the hope it will be useful, but WITHOUT
17 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
18 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
19 * more details.
20 *
21 * You should have received a copy of the GNU General Public License along with
22 * this program; if not, write to the Free Software Foundation, Inc.,
23 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
24 */
25
26 #include <linux/stacktrace.h>
```

```c
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/slab.h>
#include "../include/ktap_types.h"
#include "../include/ktap_ffi.h"
#include "kp_obj.h"
#include "kp_str.h"
#include "kp_tab.h"
#include "ktap.h"
#include "kp_vm.h"
#include "kp_transport.h"

/* Error message strings. */
const char *kp_err_allmsg =
#define ERRDEF(name, msg)       msg "\0"
#include "../include/ktap_errmsg.h"
;

/* memory allocation flag */
#define KTAP_ALLOC_FLAGS ((GFP_KERNEL | __GFP_NORETRY | __GFP_NOWARN) \
            & ~__GFP_WAIT)

/*
* TODO: It's not safe to call into facilities in the kernel at-large,
* so we may need to use ktap own memory pool, not kmalloc.
*/


void *kp_malloc(ktap_state_t *ks, int size)
{
    void *addr;

    addr = kmalloc(size, KTAP_ALLOC_FLAGS);
    if (unlikely(!addr)) {
        kp_error(ks, "kmalloc failed\n");
    }
    return addr;
}

void *kp_zalloc(ktap_state_t *ks, int size)
{
    void *addr;

    addr = kzalloc(size, KTAP_ALLOC_FLAGS);
    if (unlikely(!addr))
        kp_error(ks, "kzalloc failed\n");
    return addr;
}

void kp_free(ktap_state_t *ks, void *addr)
{
    kfree(addr);
}


void kp_obj_dump(ktap_state_t *ks, const ktap_val_t *v)
{
    switch (itype(v)) {
    case KTAP_TNIL:
        kp_puts(ks, "NIL");
        break;
    case KTAP_TTRUE:
        kp_printf(ks, "true");
        break;
    case KTAP_TFALSE:
        kp_printf(ks, "false");
        break;
    case KTAP_TNUM:
        kp_printf(ks, "NUM %ld", nvalue(v));
        break;
    case KTAP_TLIGHTUD:
        kp_printf(ks, "LIGHTUD 0x%lx", (unsigned long)pvalue(v));
        break;
     case KTAP_TFUNC:
         kp_printf(ks, "FUNCTION 0x%lx", (unsigned long)fvalue(v));
         break;
```

```
    case KTAP_TSTR:
        kp_printf(ks, "STR #%s", svalue(v));
        break;
    case KTAP_TTAB:
        kp_printf(ks, "TABLE 0x%lx", (unsigned long)hvalue(v));
        break;
        default:
        kp_printf(ks, "GCVALUE 0x%lx", (unsigned long)gcvalue(v));
        break;
    }
}

void kp_obj_show(ktap_state_t *ks, const ktap_val_t *v)
{
    switch (itype(v)) {
    case KTAP_TNIL:
        kp_puts(ks, "nil");
        break;
    case KTAP_TTRUE:
        kp_puts(ks, "true");
        break;
    case KTAP_TFALSE:
        kp_puts(ks, "false");
        break;
    case KTAP_TNUM:
        kp_printf(ks, "%ld", nvalue(v));
        break;
    case KTAP_TLIGHTUD:
        kp_printf(ks, "lightud 0x%lx", (unsigned long)pvalue(v));
        break;
    case KTAP_TCFUNC:
        kp_printf(ks, "cfunction 0x%lx", (unsigned long)fvalue(v));
        break;
    case KTAP_TFUNC:
        kp_printf(ks, "function 0x%lx", (unsigned long)gcvalue(v));
        break;
    case KTAP_TSTR:
        kp_puts(ks, svalue(v));
        break;
    case KTAP_TTAB:
        kp_printf(ks, "table 0x%lx", (unsigned long)hvalue(v));
        break;
#ifdef CONFIG_KTAP_FFI
    case KTAP_TCDATA:
        kp_cdata_dump(ks, cdvalue(v));
        break;
#endif
    case KTAP_TEVENTSTR:
        /* check event context */
        if (!ks->current_event) {
            kp_error(ks,
            "cannot stringify event str in invalid context\n");
            return;
        }

        kp_transport_event_write(ks, ks->current_event);
        break;
    case KTAP_TKSTACK:
        kp_transport_print_kstack(ks, v->val.stack.depth,
                          v->val.stack.skip);
        break;
        default:
        kp_error(ks, "print unknown value type: %d\n", itype(v));
        break;
    }
}


/*
* equality of ktap values.
*/
int kp_obj_rawequal(const ktap_val_t *t1, const ktap_val_t *t2)
{
    switch (itype(t1)) {
    case KTAP_TNIL:
    case KTAP_TTRUE:
```

```
    case KTAP_TFALSE:
        return 1;
    case KTAP_TNUM:
        return nvalue(t1) == nvalue(t2);
    case KTAP_TLIGHTUD:
        return pvalue(t1) == pvalue(t2);
    case KTAP_TFUNC:
        return fvalue(t1) == fvalue(t2);
    case KTAP_TSTR:
        return rawtsvalue(t1) == rawtsvalue(t2);
    case KTAP_TTAB:
        return hvalue(t1) == hvalue(t2);
    default:
        return gcvalue(t1) == gcvalue(t2);
    }

    return 0;
}

/*
* ktap will not use lua's length operator for table,
* also # is not for length operator any more in ktap.
*/
int kp_obj_len(ktap_state_t *ks, const ktap_val_t *v)
{
    switch(itype(v)) {
    case KTAP_TTAB:
        return kp_tab_len(ks, hvalue(v));
    case KTAP_TSTR:
        return rawtsvalue(v)->len;
    default:
        kp_printf(ks, "cannot get length of type %d\n", v->type);
        return -1;
    }
    return 0;
}

/* need to protect allgc field? */
ktap_obj_t *kp_obj_new(ktap_state_t *ks, size_t size)
{
    ktap_obj_t *o, **list;

    if (ks != G(ks)->mainthread) {
        kp_error(ks, "kp_obj_new only can be called in mainthread\n");
        return NULL;
    }

    o = kp_malloc(ks, size);
    if (unlikely(!o))
        return NULL;

    list = &G(ks)->allgc;
    gch(o)->nextgc = *list;
    *list = o;

    return o;
}


/* this function may be time consuming, move out from table set/get? */
ktap_str_t *kp_obj_kstack2str(ktap_state_t *ks, uint16_t depth, uint16_t skip)
{
    struct stack_trace trace;
    unsigned long *bt;
    char *btstr, *p;
    int i;

    bt = kp_this_cpu_print_buffer(ks); /* use print percpu buffer */
    trace.nr_entries = 0;
    trace.skip = skip;
    trace.max_entries = depth;
    trace.entries = (unsigned long *)(bt + 1);
    save_stack_trace(&trace);

    /* convert backtrace to string */
    p = btstr = kp_this_cpu_temp_buffer(ks);
```

```
255     for (i = 0; i < trace.nr_entries; i++) {
256         unsigned long addr = trace.entries[i];
257 
258         if (addr == ULONG_MAX)
259             break;
260 
261         p += sprint_symbol(p, addr);
262         *p++ = '\n';
263         }
264 
265     return kp_str_new(ks, btstr, p - btstr);
266 }
267 
268 void kp_obj_free_gclist(ktap_state_t *ks, ktap_obj_t *o)
269 {
270     while (o) {
271         ktap_obj_t *next;
272 
273         next = gch(o)->nextgc;
274         switch (gch(o)->gct) {
275         case ~KTAP_TTAB:
276             kp_tab_free(ks, (ktap_tab_t *)o);
277             break;
278         case ~KTAP_TUPVAL:
279             kp_freeupval(ks, (ktap_upval_t *)o);
280             break;
281         default:
282             kp_free(ks, o);
283         }
284         o = next;
285     }
286 }
287 
288 void kp_obj_freeall(ktap_state_t *ks)
289 {
290     kp_obj_free_gclist(ks, G(ks)->allgc);
291     G(ks)->allgc = NULL;
292 }
293 
```

One Level Up Top Level

runtime/kp_transport.c - ktap

Global variables defined

- ftrace_find_event
- tracing_pipe_fops

Data types defined

- ktap_ftrace_entry
- ktap_trace_iterator
- ktap_trace_type

Functions defined

- __find_next_entry
- __kp_bputs
- __kp_puts
- _trace_seq_puts
- _trace_seq_to_user
- kp_printf
- kp_transport_event_write
- kp_transport_exit
- kp_transport_init
- kp_transport_print_kstack
- kp_transport_write
- ns2usecs
- peek_next_entry
- poll_wait_pipe
- print_trace_bputs
- print_trace_fmt
- print_trace_fn
- print_trace_line
- print_trace_stack
- trace_consume
- trace_empty
- trace_find_next_entry_inc

- trace_print_timestamp
- tracing_open_pipe
- tracing_read_pipe
- tracing_release_pipe
- tracing_wait_pipe

Macros defined

- KTAP_TRACE_ITER
- TRACE_BUF_SIZE_DEFAULT

Source code

```
/*
 * kp_transport.c - ktap transport functionality
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <linux/debugfs.h>
#include <linux/ftrace_event.h>
#include <linux/stacktrace.h>
#include <linux/clocksource.h>
#include <asm/uaccess.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/kallsyms.h>
#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_events.h"
#include "kp_transport.h"

struct ktap_trace_iterator {
    struct ring_buffer    *buffer;
    int            print_timestamp;
    void            *private;

    struct trace_iterator    iter;
};

enum ktap_trace_type {
    __TRACE_FIRST_TYPE = 0,

    TRACE_FN = 1, /* must be same as ftrace definition in kernel */
    TRACE_PRINT,
    TRACE_BPUTS,
    TRACE_STACK,
    TRACE_USER_STACK,

    __TRACE_LAST_TYPE,
};
```

```
54
55 #define KTAP_TRACE_ITER(iter)    \
56     container_of(iter, struct ktap_trace_iterator, iter)
57
58 static
59 ssize_t _trace_seq_to_user(struct trace_seq *s, char __user *ubuf, size_t cnt)
60 {
61     int len;
62     int ret;
63
64     if (!cnt)
65         return 0;
66
67     if (s->len <= s->readpos)
68         return -EBUSY;
69
70     len = s->len - s->readpos;
71     if (cnt > len)
72         cnt = len;
73     ret = copy_to_user(ubuf, s->buffer + s->readpos, cnt);
74     if (ret == cnt)
75         return -EFAULT;
76
77     cnt -= ret;
78
79     s->readpos += cnt;
80     return cnt;
81 }
82
83 int _trace_seq_puts(struct trace_seq *s, const char *str)
84 {
85     int len = strlen(str);
86
87     if (s->full)
88         return 0;
89
90     if (len > ((PAGE_SIZE - 1) - s->len)) {
91         s->full = 1;
92         return 0;
93     }
94
95     memcpy(s->buffer + s->len, str, len);
96     s->len += len;
97
98     return len;
99 }
100
101  static int trace_empty(struct trace_iterator *iter)
102  {
103      struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
104      int cpu;
105
106      for_each_online_cpu(cpu) {
107          if (!ring_buffer_empty_cpu(ktap_iter->buffer, cpu))
108              return 0;
109      }
110
111      return 1;
112  }
113
114  static void trace_consume(struct trace_iterator *iter)
115  {
116      struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
117
118      ring_buffer_consume(ktap_iter->buffer, iter->cpu, &iter->ts,
119                  &iter->lost_events);
120  }
121
122  unsigned long long ns2usecs(cycle_t nsec)
123  {
124      nsec += 500;
125      do_div(nsec, 1000);
126      return nsec;
127  }
128
129  static int trace_print_timestamp(struct trace_iterator *iter)
```

```
{
    struct trace_seq *s = &iter->seq;
    unsigned long long t;
    unsigned long secs, usec_rem;

    t = ns2usecs(iter->ts);
    usec_rem = do_div(t, USEC_PER_SEC);
    secs = (unsigned long)t;

    return trace_seq_printf(s, "%5lu.%06lu: ", secs, usec_rem);
}

/* todo: export kernel function ftrace_find_event in future, and make faster */
static struct trace_event *(*ftrace_find_event)(int type);

static enum print_line_t print_trace_fmt(struct trace_iterator *iter)
{
    struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
    struct trace_entry *entry = iter->ent;
    struct trace_event *ev;

    ev = ftrace_find_event(entry->type);

    if (ktap_iter->print_timestamp && !trace_print_timestamp(iter))
        return TRACE_TYPE_PARTIAL_LINE;

    if (ev) {
        int ret = ev->funcs->trace(iter, 0, ev);

        /* overwrite '\n' at the ending */
        iter->seq.buffer[iter->seq.len - 1] = '\0';
        iter->seq.len--;
        return ret;
    }

    return TRACE_TYPE_PARTIAL_LINE;
}

static enum print_line_t print_trace_stack(struct trace_iterator *iter)
{
    struct trace_entry *entry = iter->ent;
    struct stack_trace trace;
    char str[KSYM_SYMBOL_LEN];
    int i;

    trace.entries = (unsigned long *)(entry + 1);
    trace.nr_entries = (iter->ent_size - sizeof(*entry)) /
               sizeof(unsigned long);

    if (!_trace_seq_puts(&iter->seq, "<stack trace>\n"))
        return TRACE_TYPE_PARTIAL_LINE;

    for (i = 0; i < trace.nr_entries; i++) {
        unsigned long p = trace.entries[i];

        if (p == ULONG_MAX)
            break;

        sprint_symbol(str, p);
        if (!trace_seq_printf(&iter->seq, " => %s\n", str))
            return TRACE_TYPE_PARTIAL_LINE;
    }

    return TRACE_TYPE_HANDLED;
}

struct ktap_ftrace_entry {
    struct trace_entry entry;
    unsigned long ip;
    unsigned long parent_ip;
};

static enum print_line_t print_trace_fn(struct trace_iterator *iter)
{
    struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
    struct ktap_ftrace_entry *field = (struct ktap_ftrace_entry *)iter->ent;
```

```
206     char str[KSYM_SYMBOL_LEN];
207
208     if (ktap_iter->print_timestamp && !trace_print_timestamp(iter))
209         return TRACE_TYPE_PARTIAL_LINE;
210
211     sprint_symbol(str, field->ip);
212     if (!_trace_seq_puts(&iter->seq, str))
213         return TRACE_TYPE_PARTIAL_LINE;
214
215     if (!_trace_seq_puts(&iter->seq, " <- "))
216         return TRACE_TYPE_PARTIAL_LINE;
217
218     sprint_symbol(str, field->parent_ip);
219     if (!_trace_seq_puts(&iter->seq, str))
220         return TRACE_TYPE_PARTIAL_LINE;
221
222     return TRACE_TYPE_HANDLED;
223 }
224
225 static enum print_line_t print_trace_bputs(struct trace_iterator *iter)
226 {
227     if (!_trace_seq_puts(&iter->seq,
228                 (const char *)(*(unsigned long *)(iter->ent + 1))))
229         return TRACE_TYPE_PARTIAL_LINE;
230
231     return TRACE_TYPE_HANDLED;
232 }
233
234 static enum print_line_t print_trace_line(struct trace_iterator *iter)
235 {
236     struct trace_entry *entry = iter->ent;
237     char *str = (char *)(entry + 1);
238
239     if (entry->type == TRACE_PRINT) {
240         if (!trace_seq_printf(&iter->seq, "%s", str))
241             return TRACE_TYPE_PARTIAL_LINE;
242
243         return TRACE_TYPE_HANDLED;
244     }
245
246     if (entry->type == TRACE_BPUTS)
247         return print_trace_bputs(iter);
248
249     if (entry->type == TRACE_STACK)
250         return print_trace_stack(iter);
251
252     if (entry->type == TRACE_FN)
253         return print_trace_fn(iter);
254
255     return print_trace_fmt(iter);
256 }
257
258 static struct trace_entry *
259 peek_next_entry(struct trace_iterator *iter, int cpu, u64 *ts,
260         unsigned long *lost_events)
261 {
262     struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
263     struct ring_buffer_event *event;
264
265     event = ring_buffer_peek(ktap_iter->buffer, cpu, ts, lost_events);
266     if (event) {
267         iter->ent_size = ring_buffer_event_length(event);
268         return ring_buffer_event_data(event);
269     }
270
271     return NULL;
272 }
273
274 static struct trace_entry *
275 __find_next_entry(struct trace_iterator *iter, int *ent_cpu,
276           unsigned long *missing_events, u64 *ent_ts)
277 {
278     struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
279     struct ring_buffer *buffer = ktap_iter->buffer;
280     struct trace_entry *ent, *next = NULL;
281     unsigned long lost_events = 0, next_lost = 0;
```

```
282     u64 next_ts = 0, ts;
283     int next_cpu = -1;
284     int next_size = 0;
285     int cpu;
286
287     for_each_online_cpu(cpu) {
288         if (ring_buffer_empty_cpu(buffer, cpu))
289             continue;
290
291         ent = peek_next_entry(iter, cpu, &ts, &lost_events);
292         /*
293          * Pick the entry with the smallest timestamp:
294          */
295         if (ent && (!next || ts < next_ts)) {
296             next = ent;
297             next_cpu = cpu;
298             next_ts = ts;
299             next_lost = lost_events;
300             next_size = iter->ent_size;
301         }
302     }
303
304     iter->ent_size = next_size;
305
306     if (ent_cpu)
307         *ent_cpu = next_cpu;
308
309     if (ent_ts)
310         *ent_ts = next_ts;
311
312     if (missing_events)
313         *missing_events = next_lost;
314
315     return next;
316 }
317
318 /* Find the next real entry, and increment the iterator to the next entry */
319 static void *trace_find_next_entry_inc(struct trace_iterator *iter)
320 {
321     iter->ent = __find_next_entry(iter, &iter->cpu,
322                     &iter->lost_events, &iter->ts);
323     if (iter->ent)
324         iter->idx++;
325
326     return iter->ent ? iter : NULL;
327 }
328
329 static void poll_wait_pipe(void)
330 {
331     set_current_state(TASK_INTERRUPTIBLE);
332     /* sleep for 100 msecs, and try again. */
333     schedule_timeout(HZ / 10);
334 }
335
336 static int tracing_wait_pipe(struct file *filp)
337 {
338     struct trace_iterator *iter = filp->private_data;
339     struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
340     ktap_state_t *ks = ktap_iter->private;
341
342     while (trace_empty(iter)) {
343
344         if ((filp->f_flags & O_NONBLOCK)) {
345             return -EAGAIN;
346         }
347
348         mutex_unlock(&iter->mutex);
349
350         poll_wait_pipe();
351
352         mutex_lock(&iter->mutex);
353
354         if (G(ks)->wait_user && trace_empty(iter))
355             return -EINTR;
356     }
357
```

```
358     return 1;
359 }
360 
361 static ssize_t
362 tracing_read_pipe(struct file *filp, char __user *ubuf, size_t cnt,
363           loff_t *ppos)
364 {
365     struct trace_iterator *iter = filp->private_data;
366     ssize_t sret;
367 
368     /* return any leftover data */
369     sret = trace_seq_to_user(&iter->seq, ubuf, cnt);
370     if (sret != -EBUSY)
371         return sret;
372     /*
373      * Avoid more than one consumer on a single file descriptor
374      * This is just a matter of traces coherency, the ring buffer itself
375      * is protected.
376      */
377     mutex_lock(&iter->mutex);
378 
379 waitagain:
380     sret = tracing_wait_pipe(filp);
381     if (sret <= 0)
382         goto out;
383 
384     /* stop when tracing is finished */
385     if (trace_empty(iter)) {
386         sret = 0;
387         goto out;
388     }
389 
390     if (cnt >= PAGE_SIZE)
391         cnt = PAGE_SIZE - 1;
392 
393     /* reset all but tr, trace, and overruns */
394     memset(&iter->seq, 0,
395            sizeof(struct trace_iterator) -
396            offsetof(struct trace_iterator, seq));
397     iter->pos = -1;
398 
399     while (trace_find_next_entry_inc(iter) != NULL) {
400         enum print_line_t ret;
401         int len = iter->seq.len;
402 
403         ret = print_trace_line(iter);
404         if (ret == TRACE_TYPE_PARTIAL_LINE) {
405             /* don't print partial lines */
406             iter->seq.len = len;
407             break;
408         }
409         if (ret != TRACE_TYPE_NO_CONSUME)
410             trace_consume(iter);
411 
412         if (iter->seq.len >= cnt)
413             break;
414 
415         /*
416          * Setting the full flag means we reached the trace_seq buffer
417          * size and we should leave by partial output condition above.
418          * One of the trace_seq_* functions is not used properly.
419          */
420         WARN_ONCE(iter->seq.full, "full flag set for trace type %d",
421               iter->ent->type);
422     }
423 
424     /* Now copy what we have to the user */
425     sret = trace_seq_to_user(&iter->seq, ubuf, cnt);
426     if (iter->seq.readpos >= iter->seq.len)
427         trace_seq_init(&iter->seq);
428 
429     /*
430      * If there was nothing to send to user, in spite of consuming trace
431      * entries, go back to wait for more entries.
432      */
433     if (sret == -EBUSY)
```

```
434         goto waitagain;
435
436 out:
437     mutex_unlock(&iter->mutex);
438
439     return sret;
440 }
441
442 static int tracing_open_pipe(struct inode *inode, struct file *filp)
443 {
444     struct ktap_trace_iterator *ktap_iter;
445     ktap_state_t *ks = inode->i_private;
446
447     /* create a buffer to store the information to pass to userspace */
448     ktap_iter = kzalloc(sizeof(*ktap_iter), GFP_KERNEL);
449     if (!ktap_iter)
450         return -ENOMEM;
451
452     ktap_iter->private = ks;
453     ktap_iter->buffer = G(ks)->buffer;
454     ktap_iter->print_timestamp = G(ks)->parm->print_timestamp;
455     mutex_init(&ktap_iter->iter.mutex);
456     filp->private_data = &ktap_iter->iter;
457
458     nonseekable_open(inode, filp);
459
460     return 0;
461 }
462
463 static int tracing_release_pipe(struct inode *inode, struct file *file)
464 {
465     struct trace_iterator *iter = file->private_data;
466     struct ktap_trace_iterator *ktap_iter = KTAP_TRACE_ITER(iter);
467
468     mutex_destroy(&iter->mutex);
469     kfree(ktap_iter);
470     return 0;
471 }
472
473 static const struct file_operations tracing_pipe_fops = {
474     .open        = tracing_open_pipe,
475     .read        = tracing_read_pipe,
476     .splice_read    = NULL,
477     .release    = tracing_release_pipe,
478     .llseek        = no_llseek,
479 };
480
481 /*
482 * preempt disabled in ring_buffer_lock_reserve
483 *
484 * The implementation is similar with funtion __ftrace_trace_stack.
485 */
486 void kp_transport_print_kstack(ktap_state_t *ks, uint16_t depth, uint16_t skip)
487 {
488     struct ring_buffer *buffer = G(ks)->buffer;
489     struct ring_buffer_event *event;
490     struct trace_entry *entry;
491     int size;
492
493     size = depth * sizeof(unsigned long);
494     event = ring_buffer_lock_reserve(buffer, sizeof(*entry) + size);
495     if (!event) {
496         KTAP_STATS(ks)->events_missed += 1;
497         return;
498     } else {
499         struct stack_trace trace;
500
501         entry = ring_buffer_event_data(event);
502         tracing_generic_entry_update(entry, 0, 0);
503         entry->type = TRACE_STACK;
504
505         trace.nr_entries = 0;
506         trace.skip = skip;
507         trace.max_entries = depth;
508         trace.entries = (unsigned long *)(entry + 1);
509         save_stack_trace(&trace);
```

```
510
511         ring_buffer_unlock_commit(buffer, event);
512     }
513 }
514
515 void kp_transport_event_write(ktap_state_t *ks, struct ktap_event_data *e)
516 {
517     struct ring_buffer *buffer = G(ks)->buffer;
518     struct ring_buffer_event *event;
519     struct trace_entry *ev_entry = e->data->raw->data;
520     struct trace_entry *entry;
521     int entry_size = e->data->raw->size;
522
523     event = ring_buffer_lock_reserve(buffer, entry_size +
524                     sizeof(struct ftrace_event_call *));
525     if (!event) {
526         KTAP_STATS(ks)->events_missed += 1;
527         return;
528     } else {
529         entry = ring_buffer_event_data(event);
530
531         memcpy(entry, ev_entry, entry_size);
532
533         ring_buffer_unlock_commit(buffer, event);
534     }
535 }
536
537 void kp_transport_write(ktap_state_t *ks, const void *data, size_t length)
538 {
539     struct ring_buffer *buffer = G(ks)->buffer;
540     struct ring_buffer_event *event;
541     struct trace_entry *entry;
542     int size;
543
544     size = sizeof(struct trace_entry) + length;
545
546     event = ring_buffer_lock_reserve(buffer, size);
547     if (!event) {
548         KTAP_STATS(ks)->events_missed += 1;
549         return;
550     } else {
551         entry = ring_buffer_event_data(event);
552
553         tracing_generic_entry_update(entry, 0, 0);
554         entry->type = TRACE_PRINT;
555         memcpy(entry + 1, data, length);
556
557         ring_buffer_unlock_commit(buffer, event);
558     }
559 }
560
561 /* general print function */
562 void kp_printf(ktap_state_t *ks, const char *fmt, ...)
563 {
564     char buff[1024];
565     va_list args;
566     int len;
567
568     va_start(args, fmt);
569     len = vscnprintf(buff, 1024, fmt, args);
570     va_end(args);
571
572     buff[len] = '\0';
573     kp_transport_write(ks, buff, len + 1);
574 }
575
576 void __kp_puts(ktap_state_t *ks, const char *str)
577 {
578     kp_transport_write(ks, str, strlen(str) + 1);
579 }
580
581 void __kp_bputs(ktap_state_t *ks, const char *str)
582 {
583     struct ring_buffer *buffer = G(ks)->buffer;
584     struct ring_buffer_event *event;
585     struct trace_entry *entry;
```

```
    int size;

    size = sizeof(struct trace_entry) + sizeof(unsigned long *);

    event = ring_buffer_lock_reserve(buffer, size);
    if (!event) {
        KTAP_STATS(ks)->events_missed += 1;
        return;
    } else {
        entry = ring_buffer_event_data(event);

        tracing_generic_entry_update(entry, 0, 0);
        entry->type = TRACE_BPUTS;
        *(unsigned long *)(entry + 1) = (unsigned long)str;

        ring_buffer_unlock_commit(buffer, event);
    }
}

void kp_transport_exit(ktap_state_t *ks)
{
    if (G(ks)->buffer)
        ring_buffer_free(G(ks)->buffer);
    debugfs_remove(G(ks)->trace_pipe_dentry);
}

#define TRACE_BUF_SIZE_DEFAULT    1441792UL /* 16384 * 88 (sizeof(entry)) */

int kp_transport_init(ktap_state_t *ks, struct dentry *dir)
{
    struct ring_buffer *buffer;
    struct dentry *dentry;
    char filename[32] = {0};

#ifdef CONFIG_PPC64
    ftrace_find_event = (void *)kallsyms_lookup_name(".ftrace_find_event");
#else
    ftrace_find_event = (void *)kallsyms_lookup_name("ftrace_find_event");
#endif
    if (!ftrace_find_event) {
        printk("ktap: cannot lookup ftrace_find_event in kallsyms\n");
        return -EINVAL;
    }

    buffer = ring_buffer_alloc(TRACE_BUF_SIZE_DEFAULT, RB_FL_OVERWRITE);
    if (!buffer)
        return -ENOMEM;

    sprintf(filename, "trace_pipe_%d", (int)task_tgid_vnr(current));

    dentry = debugfs_create_file(filename, 0444, dir,
                    ks, &tracing_pipe_fops);
    if (!dentry) {
        pr_err("ktapvm: cannot create trace_pipe file in debugfs\n");
        ring_buffer_free(buffer);
        return -1;
    }

    G(ks)->buffer = buffer;
    G(ks)->trace_pipe_dentry = dentry;

    return 0;
}
```

One Level Up Top Level

runtime/kp_events.h - ktap

Data types defined

- KTAP_EVENT_FIELD_TYPE
- KTAP_EVENT_TYPE
- ktap_event
- ktap_event_data
- ktap_event_field

Macros defined

- __KTAP_EVENTS_H__

Source code

```
1  #ifndef __KTAP_EVENTS_H__
2  #define __KTAP_EVENTS_H__
3
4  #include <linux/ftrace_event.h>
5  #include <trace/syscall.h>
6  #include <trace/events/syscalls.h>
7  #include <linux/syscalls.h>
8  #include <linux/kprobes.h>
9
10 enum KTAP_EVENT_FIELD_TYPE {
11     KTAP_EVENT_FIELD_TYPE_INVALID = 0, /* arg type not support yet */
12
13     KTAP_EVENT_FIELD_TYPE_INT,
14     KTAP_EVENT_FIELD_TYPE_LONG,
15     KTAP_EVENT_FIELD_TYPE_STRING,
16
17     KTAP_EVENT_FIELD_TYPE_REGESTER,
18     KTAP_EVENT_FIELD_TYPE_CONST,
19     KTAP_EVENT_FIELD_TYPE_NIL /* arg not exist */
20 };
21
22 struct ktap_event_field {
23     enum KTAP_EVENT_FIELD_TYPE type;
24     int offset;
25 };
26
27 enum KTAP_EVENT_TYPE {
28     KTAP_EVENT_TYPE_PERF,
29     KTAP_EVENT_TYPE_TRACEPOINT,
30     KTAP_EVENT_TYPE_SYSCALL_ENTER,
31     KTAP_EVENT_TYPE_SYSCALL_EXIT,
32     KTAP_EVENT_TYPE_KPROBE,
33 };
34
35 struct ktap_event {
36     struct list_head list;
37     int type;
38     ktap_state_t *ks;
39     ktap_func_t *fn;
40     struct perf_event *perf;
41     int syscall_nr; /* for syscall event */
42     struct ktap_event_field fields[9]; /* arg1..arg9 */
43     ktap_str_t *name; /* intern probename string */
44
45     struct kprobe kp; /* kprobe event */
46 };
47
48 /* this structure allocate on stack */
```

```
49 struct ktap_event_data {
50     struct ktap_event *event;
51     struct perf_sample_data *data;
52     struct pt_regs *regs;
53     ktap_str_t *argstr; /* for cache argstr intern string */
54 };
55
56 int kp_events_init(ktap_state_t *ks);
57 void kp_events_exit(ktap_state_t *ks);
58
59 int kp_event_create(ktap_state_t *ks, struct perf_event_attr *attr,
60             struct task_struct *task, const char *filter,
61             ktap_func_t *fn);
62 int kp_event_create_tracepoint(ktap_state_t *ks, const char *event_name,
63                   ktap_func_t *fn);
64
65 int kp_event_create_kprobe(ktap_state_t *ks, const char *event_name,
66                ktap_func_t *fn);
67 void kp_event_getarg(ktap_state_t *ks, ktap_val_t *ra, int idx);
68 const char *kp_event_tostr(ktap_state_t *ks);
69 const ktap_str_t *kp_event_stringify(ktap_state_t *ks);
70
71 #endif /* __KTAP_EVENTS_H__ */
```

One Level Up Top Level

runtime/kp_events.c - ktap

Data types defined

- ftrace_event_field

Functions defined

- call_probe_closure
- events_destroy
- get_fields
- init_event_fields
- kp_event_create
- kp_event_create_kprobe
- kp_event_getarg
- kp_event_stringify
- kp_event_tostr
- kp_events_exit
- kp_events_init
- perf_callback
- pre_handler_kprobe

Source code

```
/*
 * kp_events.c - ktap events management (registry, destroy, event callback)
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <linux/module.h>
#include <linux/ctype.h>
#include <linux/slab.h>
#include <linux/version.h>
#include <asm/syscall.h>
#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_obj.h"
```

```c
#include "kp_str.h"
#include "kp_transport.h"
#include "kp_vm.h"
#include "kp_events.h"

const char *kp_event_tostr(ktap_state_t *ks)
{
    struct ktap_event_data *e = ks->current_event;
    struct ftrace_event_call *call;
    struct trace_iterator *iter;
    struct trace_event *ev;
    enum print_line_t ret = TRACE_TYPE_NO_CONSUME;
    static const char *dummy_msg = "argstr_not_available";

    /* need to check current context is vaild tracing context */
    if (!ks->current_event) {
        kp_error(ks, "cannot stringify event str in invalid context\n");
        return NULL;
    }

    /*check if stringified before */
    if (ks->current_event->argstr)
        return getstr(ks->current_event->argstr);

    /* timer event and raw tracepoint don't have associated argstr */
    if (e->event->type == KTAP_EVENT_TYPE_PERF && e->event->perf->tp_event)
        call = e->event->perf->tp_event;
    else
        return dummy_msg;

    /* Simulate the iterator */

    /*
     * use temp percpu buffer as trace_iterator
     * we cannot use same print_buffer because we may called from printf.
     */
    iter = kp_this_cpu_temp_buffer(ks);

    trace_seq_init(&iter->seq);
    iter->ent = e->data->raw->data;

    ev = &(call->event);
    if (ev)
        ret = ev->funcs->trace(iter, 0, ev);

    if (ret != TRACE_TYPE_NO_CONSUME) {
        struct trace_seq *s = &iter->seq;
        int len = s->len >= PAGE_SIZE ? PAGE_SIZE - 1 : s->len;

        s->buffer[len] = '\0';
        return &s->buffer[0];
    }

    return dummy_msg;
}

/* return string repr of 'argstr' */
const ktap_str_t *kp_event_stringify(ktap_state_t *ks)
{
    const char *str;
    ktap_str_t *ts;

    /*check if stringified before */
    if (ks->current_event->argstr)
        return ks->current_event->argstr;

    str = kp_event_tostr(ks);
    if (!str)
        return NULL;

     ts = kp_str_newz(ks, str);
     ks->current_event->argstr = ts;
     return ts;
 }

 /*
```

```
106 * This definition should keep update with kernel/trace/trace.h
107 * TODO: export this struct in kernel
108 */
109 struct ftrace_event_field {
110     struct list_head        link;
111     const char              *name;
112     const char              *type;
113     int                     filter_type;
114     int                     offset;
115     int                     size;
116     int                     is_signed;
117 };
118
119 static struct list_head *get_fields(struct ftrace_event_call *event_call)
120 {
121     if (!event_call->class->get_fields)
122         return &event_call->class->fields;
123     return event_call->class->get_fields(event_call);
124 }
125
126 void kp_event_getarg(ktap_state_t *ks, ktap_val_t *ra, int idx)
127 {
128     struct ktap_event_data *e = ks->current_event;
129     struct ktap_event *event = e->event;
130     struct ktap_event_field *event_fields = &event->fields[idx];
131
132     switch (event_fields->type)  {
133     case KTAP_EVENT_FIELD_TYPE_INT: {
134         struct trace_entry *entry = e->data->raw->data;
135         void *value = (unsigned char *)entry + event_fields->offset;
136         int n = *(int *)value;
137         set_number(ra, n);
138         return;
139         }
140     case KTAP_EVENT_FIELD_TYPE_LONG: {
141         struct trace_entry *entry = e->data->raw->data;
142         void *value = (unsigned char *)entry + event_fields->offset;
143         long n = *(long *)value;
144         set_number(ra, n);
145         return;
146         }
147     case KTAP_EVENT_FIELD_TYPE_STRING: {
148         struct trace_entry *entry = e->data->raw->data;
149         ktap_str_t *ts;
150         void *value = (unsigned char *)entry + event_fields->offset;
151         ts = kp_str_newz(ks, (char *)value);
152         if (ts)
153             set_string(ra, ts);
154         else
155             set_nil(ra);
156         return;
157         }
158     case KTAP_EVENT_FIELD_TYPE_CONST: {
159         set_number(ra, (ktap_number)event_fields->offset);
160         return;
161         }
162     case KTAP_EVENT_FIELD_TYPE_REGESTER: {
163         unsigned long *reg = (unsigned long *)((u8 *)e->regs +
164                     event_fields->offset);
165         set_number(ra, *reg);
166         return;
167         }
168     case KTAP_EVENT_FIELD_TYPE_NIL:
169         set_nil(ra);
170         return;
171     case KTAP_EVENT_FIELD_TYPE_INVALID:
172         kp_error(ks, "the field type is not supported yet\n");
173         set_nil(ra);
174         return;
175     }
176 }
177
178 /* init all fields of event, for quick arg1..arg9 access */
179 static int init_event_fields(ktap_state_t *ks, struct ktap_event *event)
180 {
181     struct ftrace_event_call *event_call = event->perf->tp_event;
```

```
    struct ktap_event_field *event_fields = &event->fields[0];
    struct ftrace_event_field *field;
    struct list_head *head;
    int idx = 0, n = 0;

    /* only init fields for tracepoint, not timer event */
    if (!event_call)
        return 0;

    /* intern probename */
    event->name = kp_str_newz(ks, event_call->name);
    if (unlikely(!event->name))
        return -ENOMEM;

    head = get_fields(event_call);
    list_for_each_entry_reverse(field, head, link) {
        if (n++ == 9) {
            /*
             * For some events have fields more than 9, just ignore
             * those rest fields at present.
             *
             * TODO: support access all fields in tracepoint event
             *
             * Examples: mce:mce_record, ext4:ext4_writepages, ...
             */
            return 0;
        }

        event_fields[idx].offset = field->offset;

        if (field->size == 4) {
            event_fields[idx].type = KTAP_EVENT_FIELD_TYPE_INT;
            idx++;
            continue;
        } else if (field->size == 8) {
            event_fields[idx].type = KTAP_EVENT_FIELD_TYPE_LONG;
            idx++;
            continue;
        }
        if (!strncmp(field->type, "char", 4)) {
            event_fields[idx].type = KTAP_EVENT_FIELD_TYPE_STRING;
            idx++;
            continue;
        }

        /* TODO: add more type check */
        event_fields[idx++].type = KTAP_EVENT_FIELD_TYPE_INVALID;
    }

    /* init all rest fields as NIL */
    while (idx < 9)
        event_fields[idx++].type = KTAP_EVENT_FIELD_TYPE_NIL;

    return 0;
}

static inline void call_probe_closure(ktap_state_t *mainthread,
                      ktap_func_t *fn,
                      struct ktap_event_data *e, int rctx)
{
    ktap_state_t *ks;
    ktap_val_t *func;

    ks = kp_vm_new_thread(mainthread, rctx);
    set_func(ks->top, fn);
    func = ks->top;
    incr_top(ks);

    ks->current_event = e;

    kp_vm_call(ks, func, 0);

    ks->current_event = NULL;
    kp_vm_exit_thread(ks);
}
```

```
/*
 * Callback tracing function for perf event subsystem.
 *
 * make ktap reentrant, don't disable irq in callback function,
 * same as perf and ftrace. to make reentrant, we need some
 * percpu data to be context isolation(irq/sirq/nmi/process)
 *
 * The recursion checking in here is mainly purpose for avoiding
 * corrupt ktap_state_t with timer closure callback. For tracepoint
 * recusion, perf core already handle it.
 *
 * Note tracepoint handler is calling with rcu_read_lock.
 */
static void perf_callback(struct perf_event *perf_event,
               struct perf_sample_data *data,
               struct pt_regs *regs)
{
    struct ktap_event *event;
    struct ktap_event_data e;
    ktap_state_t *ks;
    int rctx;

    event = perf_event->overflow_handler_context;
    ks = event->ks;

    if (unlikely(ks->stop))
        return;

    rctx = get_recursion_context(ks);
    if (unlikely(rctx < 0))
        return;

    e.event = event;
    e.data = data;
    e.regs = regs;
    e.argstr = NULL;

    call_probe_closure(ks, event->fn, &e, rctx);

    put_recursion_context(ks, rctx);
}

/*
 * Generic ktap event creation function (based on perf callback)
 * purpose for tracepoints/kprobe/uprobe/profile-timer/hw_breakpoint/pmu.
 */
int kp_event_create(ktap_state_t *ks, struct perf_event_attr *attr,
            struct task_struct *task, const char *filter,
            ktap_func_t *fn)
{
    struct ktap_event *event;
    struct perf_event *perf_event;
    void *callback = perf_callback;
    int cpu, ret;

    if (G(ks)->parm->dry_run)
        callback = NULL;

    /*
     * don't tracing until ktap_wait, the reason is:
     * 1). some event may hit before apply filter
     * 2). more simple to manage tracing thread
     * 3). avoid race with mainthread.
     *
     * Another way to do this is make attr.disabled as 1, then use
     * perf_event_enable after filter apply, however, perf_event_enable
     * was not exported in kernel older than 3.3, so we drop this method.
     */
    ks->stop = 1;

    for_each_cpu(cpu, G(ks)->cpumask) {
        event = kzalloc(sizeof(struct ktap_event), GFP_KERNEL);
        if (!event)
            return -ENOMEM;

        event->type = KTAP_EVENT_TYPE_PERF;
```

```
        event->ks = ks;
        event->fn = fn;
        perf_event = perf_event_create_kernel_counter(attr, cpu, task,
                                  callback, event);
        if (IS_ERR(perf_event)) {
            int err = PTR_ERR(perf_event);
            kp_error(ks, "unable register perf event: "
                     "[cpu: %d; id: %d; err: %d]\n",
                     cpu, attr->config, err);
            kfree(event);
            return err;
        }

        if (attr->type == PERF_TYPE_TRACEPOINT) {
            const char *name = perf_event->tp_event->name;
            kp_verbose_printf(ks, "enable perf event: "
                          "[cpu: %d; id: %d; name: %s; "
                          "filter: %s; pid: %d]\n",
                          cpu, attr->config, name, filter,
                          task ? task_tgid_vnr(task) : -1);
        } else if (attr->type == PERF_TYPE_SOFTWARE &&
             attr->config == PERF_COUNT_SW_CPU_CLOCK) {
            kp_verbose_printf(ks, "enable profile event: "
                          "[cpu: %d; sample_period: %d]\n",
                          cpu, attr->sample_period);
        } else {
            kp_verbose_printf(ks, "unknown perf event type\n");
        }

        event->perf = perf_event;
        INIT_LIST_HEAD(&event->list);
        list_add_tail(&event->list, &G(ks)->events_head);

        if (init_event_fields(ks, event)) {
            kp_error(ks, "unable init event fields id %d\n",
                    attr->config);
            perf_event_release_kernel(event->perf);
            list_del(&event->list);
            kfree(event);
            return ret;
        }

        if (!filter)
            continue;

        ret = kp_ftrace_profile_set_filter(perf_event, attr->config,
                           filter);
        if (ret) {
            kp_error(ks, "unable set event filter: "
                     "[id: %d; filter: %s; ret: %d]\n",
                     attr->config, filter, ret);
            perf_event_release_kernel(event->perf);
            list_del(&event->list);
            kfree(event);
            return ret;
        }
    }

    return 0;
}

/*
* tracepoint_probe_register functions changed prototype by introduce
* 'struct tracepoint', this cause hard to refer tracepoint by name.
* And these ktap raw tracepoint interface is not courage to use, so disable
* it now.
*/
#if 0
/*
* Ignore function proto in here, just use first argument.
*/
static void probe_callback(void *__data)
{
    struct ktap_event *event = __data;
    ktap_state_t *ks = event->ks;
    struct ktap_event_data e;
```

```
410     struct pt_regs regs; /* pt_regs maybe is large for stack */
411     int rctx;
412
413     if (unlikely(ks->stop))
414         return;
415
416     rctx = get_recursion_context(ks);
417     if (unlikely(rctx < 0))
418         return;
419
420     perf_fetch_caller_regs(&regs);
421
422     e.event = event;
423     e.regs = &regs;
424     e.argstr = NULL;
425
426     call_probe_closure(ks, event->fn, &e, rctx);
427
428     put_recursion_context(ks, rctx);
429 }
430
431 /*
432 * syscall events handling
433 */
434
435 static DEFINE_MUTEX(syscall_trace_lock);
436 static DECLARE_BITMAP(enabled_enter_syscalls, NR_syscalls);
437 static DECLARE_BITMAP(enabled_exit_syscalls, NR_syscalls);
438 static int sys_refcount_enter;
439 static int sys_refcount_exit;
440
441 static int get_syscall_num(const char *name)
442 {
443     int i;
444
445     for (i = 0; i < NR_syscalls; i++) {
446         if (syscalls_metadata[i] &&
447             !strcmp(name, syscalls_metadata[i]->name + 4))
448             return i;
449     }
450     return -1;
451 }
452
453 static void trace_syscall_enter(void *data, struct pt_regs *regs, long id)
454 {
455     struct ktap_event *event = data;
456     ktap_state_t *ks = event->ks;
457     struct ktap_event_data e;
458     int syscall_nr;
459     int rctx;
460
461     if (unlikely(ks->stop))
462         return;
463
464     syscall_nr = syscall_get_nr(current, regs);
465     if (unlikely(syscall_nr < 0))
466         return;
467     if (!test_bit(syscall_nr, enabled_enter_syscalls))
468         return;
469
470     rctx = get_recursion_context(ks);
471     if (unlikely(rctx < 0))
472         return;
473
474     e.event = event;
475     e.regs = regs;
476     e.argstr = NULL;
477
478     call_probe_closure(ks, event->fn, &e, rctx);
479
480     put_recursion_context(ks, rctx);
481 }
482
483 static void trace_syscall_exit(void *data, struct pt_regs *regs, long id)
484 {
485     struct ktap_event *event = data;
```

```
    ktap_state_t *ks = event->ks;
    struct ktap_event_data e;
    int syscall_nr;
    int rctx;

    syscall_nr = syscall_get_nr(current, regs);
    if (unlikely(syscall_nr < 0))
        return;
    if (!test_bit(syscall_nr, enabled_exit_syscalls))
        return;

    if (unlikely(ks->stop))
        return;

    rctx = get_recursion_context(ks);
    if (unlikely(rctx < 0))
        return;

    e.event = event;
    e.regs = regs;
    e.argstr = NULL;

    call_probe_closure(ks, event->fn, &e, rctx);

    put_recursion_context(ks, rctx);
}

/* called in dry-run mode, purpose for compare overhead with normal vm call */
static void dry_run_callback(void *data, struct pt_regs *regs, long id)
{

}

static void init_syscall_event_fields(struct ktap_event *event, int is_enter)
{
    struct ftrace_event_call *event_call;
    struct ktap_event_field *event_fields = &event->fields[0];
    struct syscall_metadata *meta = syscalls_metadata[event->syscall_nr];
    int idx = 0;

    event_call = is_enter ? meta->enter_event : meta->exit_event;

    event_fields[0].type = KTAP_EVENT_FIELD_TYPE_CONST;
    event_fields[0].offset = event->syscall_nr;

    if (!is_enter) {
#ifdef CONFIG_X86_64
        event_fields[1].type = KTAP_EVENT_FIELD_TYPE_REGESTER;
        event_fields[1].offset = offsetof(struct pt_regs, ax);
#endif
        return;
    }

    while (idx++ < meta->nb_args) {
        event_fields[idx].type = KTAP_EVENT_FIELD_TYPE_REGESTER;
#ifdef CONFIG_X86_64
        switch (idx) {
        case 1:
            event_fields[idx].offset = offsetof(struct pt_regs, di);
            break;
        case 2:
            event_fields[idx].offset = offsetof(struct pt_regs, si);
            break;
        case 3:
            event_fields[idx].offset = offsetof(struct pt_regs, dx);
            break;
        case 4:
            event_fields[idx].offset =
                        offsetof(struct pt_regs, r10);
            break;
        case 5:
            event_fields[idx].offset = offsetof(struct pt_regs, r8);
            break;
        case 6:
            event_fields[idx].offset = offsetof(struct pt_regs, r9);
            break;
```

```
562         }
563 #else
564 #warning "don't support syscall tracepoint event register access in this arch, use 'trace syscalls:* {}'
instead"
565         break;
566 #endif
567     }
568
569     /* init all rest fields as NIL */
570     while (idx < 9)
571         event_fields[idx++].type = KTAP_EVENT_FIELD_TYPE_NIL;
572 }
573
574 static int syscall_event_register(ktap_state_t *ks, const char *event_name,
575                   struct ktap_event *event)
576 {
577     int syscall_nr = 0, is_enter = 0;
578     void *callback = NULL;
579     int ret = 0;
580
581     if (!strncmp(event_name, "sys_enter_", 10)) {
582         is_enter = 1;
583         event->type = KTAP_EVENT_TYPE_SYSCALL_ENTER;
584         syscall_nr = get_syscall_num(event_name + 10);
585         callback = trace_syscall_enter;
586     } else if (!strncmp(event_name, "sys_exit_", 9)) {
587         is_enter = 0;
588         event->type = KTAP_EVENT_TYPE_SYSCALL_EXIT;
589         syscall_nr = get_syscall_num(event_name + 9);
590         callback = trace_syscall_exit;
591     }
592
593     if (G(ks)->parm->dry_run)
594         callback = dry_run_callback;
595
596     if (syscall_nr < 0)
597         return -1;
598
599     event->syscall_nr = syscall_nr;
600
601     init_syscall_event_fields(event, is_enter);
602
603     mutex_lock(&syscall_trace_lock);
604     if (is_enter) {
605         if (!sys_refcount_enter)
606             ret = register_trace_sys_enter(callback, event);
607         if (!ret) {
608             set_bit(syscall_nr, enabled_enter_syscalls);
609             sys_refcount_enter++;
610         }
611     } else {
612         if (!sys_refcount_exit)
613             ret = register_trace_sys_exit(callback, event);
614         if (!ret) {
615             set_bit(syscall_nr, enabled_exit_syscalls);
616             sys_refcount_exit++;
617         }
618     }
619     mutex_unlock(&syscall_trace_lock);
620
621     return ret;
622 }
623
624 static int syscall_event_unregister(ktap_state_t *ks, struct ktap_event *event)
625 {
626     int ret = 0;
627     void *callback;
628
629     if (event->type == KTAP_EVENT_TYPE_SYSCALL_ENTER)
630         callback = trace_syscall_enter;
631     else
632         callback = trace_syscall_exit;
633
634     if (G(ks)->parm->dry_run)
635         callback = dry_run_callback;
636
```

```
637     mutex_lock(&syscall_trace_lock);
638     if (event->type == KTAP_EVENT_TYPE_SYSCALL_ENTER) {
639         sys_refcount_enter--;
640             clear_bit(event->syscall_nr, enabled_enter_syscalls);
641             if (!sys_refcount_enter)
642                     unregister_trace_sys_enter(callback, event);
643     } else {
644         sys_refcount_exit--;
645             clear_bit(event->syscall_nr, enabled_exit_syscalls);
646             if (!sys_refcount_exit)
647                     unregister_trace_sys_exit(callback, event);
648     }
649     mutex_unlock(&syscall_trace_lock);
650
651     return ret;
652 }
653
654 /*
655 * Register tracepoint event directly, not based on perf callback
656 *
657 * This tracing method would be more faster than perf callback,
658 * because it won't need to write trace data into any temp buffer,
659 * and code path is much shorter than perf callback.
660 */
661 int kp_event_create_tracepoint(ktap_state_t *ks, const char *event_name,
662                     ktap_func_t *fn)
663 {
664     struct ktap_event *event;
665     void *callback = probe_callback;
666     int is_syscall = 0;
667     int ret;
668
669     if (G(ks)->parm->dry_run)
670         callback = NULL;
671
672     if (!strncmp(event_name, "sys_enter_", 10) ||
673         !strncmp(event_name, "sys_exit_", 9))
674         is_syscall = 1;
675
676     event = kzalloc(sizeof(struct ktap_event), GFP_KERNEL);
677     if (!event)
678         return -ENOMEM;
679
680     event->ks = ks;
681     event->fn = fn;
682     event->name = kp_str_newz(ks, event_name);
683     if (unlikely(!event->name)) {
684         kfree(event);
685         return -ENOMEM;
686     }
687
688     INIT_LIST_HEAD(&event->list);
689     list_add_tail(&event->list, &G(ks)->events_head);
690
691     if (is_syscall) {
692         ret = syscall_event_register(ks, event_name, event);
693     } else {
694         event->type = KTAP_EVENT_TYPE_TRACEPOINT;
695         ret = tracepoint_probe_register(event_name, callback, event);
696     }
697
698     if (ret) {
699         kp_error(ks, "register tracepoint %s failed, ret: %d\n",
700                 event_name, ret);
701         list_del(&event->list);
702         kfree(event);
703         return ret;
704     }
705     return 0;
706 }
707
708 #endif
709
710 /* kprobe handler */
711 static int __kprobes pre_handler_kprobe(struct kprobe *p, struct pt_regs *regs)
712 {
```

```
    struct ktap_event *event = container_of(p, struct ktap_event, kp);
    ktap_state_t *ks = event->ks;
    struct ktap_event_data e;
    int rctx;

    if (unlikely(ks->stop))
        return 0;

    rctx = get_recursion_context(ks);
    if (unlikely(rctx < 0))
        return 0;

    e.event = event;
    e.regs = regs;
    e.argstr = NULL;

    call_probe_closure(ks, event->fn, &e, rctx);

    put_recursion_context(ks, rctx);
    return 0;
}

/*
* Register kprobe event directly, not based on perf callback
*
* This tracing method would be more faster than perf callback,
* because it won't need to write trace data into any temp buffer,
* and code path is much shorter than perf callback.
*/
int kp_event_create_kprobe(ktap_state_t *ks, const char *event_name,
               ktap_func_t *fn)
{
    struct ktap_event *event;
    void *callback = pre_handler_kprobe;
    int ret;

    if (G(ks)->parm->dry_run)
        callback = NULL;

    event = kzalloc(sizeof(struct ktap_event), GFP_KERNEL);
    if (!event)
        return -ENOMEM;

    event->ks = ks;
    event->fn = fn;
    event->name = kp_str_newz(ks, event_name);
    if (unlikely(!event->name)) {
        kfree(event);
        return -ENOMEM;
    }

    INIT_LIST_HEAD(&event->list);
    list_add_tail(&event->list, &G(ks)->events_head);

    event->type = KTAP_EVENT_TYPE_KPROBE;

    event->kp.symbol_name = event_name;
    event->kp.pre_handler = callback;
    ret = register_kprobe(&event->kp);
    if (ret) {
        kp_error(ks, "register kprobe event %s failed, ret: %d\n",
                event_name, ret);
        list_del(&event->list);
        kfree(event);
        return ret;
    }
    return 0;
}


static void events_destroy(ktap_state_t *ks)
{
    struct ktap_event *event;
    struct list_head *tmp, *pos;
    struct list_head *head = &G(ks)->events_head;

```

```
789     list_for_each(pos, head) {
790         event = container_of(pos, struct ktap_event,
791                              list);
792         if (event->type == KTAP_EVENT_TYPE_PERF)
793             perf_event_release_kernel(event->perf);
794 #if 0
795         else if (event->type == KTAP_EVENT_TYPE_TRACEPOINT)
796             tracepoint_probe_unregister(getstr(event->name),
797                                 probe_callback, event);
798         else if (event->type == KTAP_EVENT_TYPE_SYSCALL_ENTER ||
799              event->type == KTAP_EVENT_TYPE_SYSCALL_EXIT )
800             syscall_event_unregister(ks, event);
801 #endif
802         else if (event->type == KTAP_EVENT_TYPE_KPROBE)
803             unregister_kprobe(&event->kp);
804         }
805            /*
806      * Ensure our callback won't be called anymore. The buffers
807      * will be freed after that.
808      */
809     tracepoint_synchronize_unregister();
810 
811     list_for_each_safe(pos, tmp, head) {
812         event = container_of(pos, struct ktap_event,
813                              list);
814         list_del(&event->list);
815         kfree(event);
816     }
817 }
818 
819 void kp_events_exit(ktap_state_t *ks)
820 {
821     if (!G(ks)->trace_enabled)
822         return;
823 
824     events_destroy(ks);
825 
826     /* call trace_end_closure after all event unregistered */
827     if ((G(ks)->state != KTAP_ERROR) && G(ks)->trace_end_closure) {
828         G(ks)->state = KTAP_TRACE_END;
829         set_func(ks->top, G(ks)->trace_end_closure);
830         incr_top(ks);
831         kp_vm_call(ks, ks->top - 1, 0);
832         G(ks)->trace_end_closure = NULL;
833     }
834 
835     G(ks)->trace_enabled = 0;
836 }
837 
838 int kp_events_init(ktap_state_t *ks)
839 {
840     G(ks)->trace_enabled = 1;
841     return 0;
842 }
843 
```

One Level Up Top Level

runtime/kp_str.h - ktap

Macros defined

- __KTAP_STR_H__
- kp_str_newz

Source code

```
1  #ifndef __KTAP_STR_H__
2  #define __KTAP_STR_H__
3
4  int kp_str_cmp(const ktap_str_t *ls, const ktap_str_t *rs);
5  int kp_str_resize(ktap_state_t *ks, int newmask);
6  void kp_str_freeall(ktap_state_t *ks);
7  ktap_str_t * kp_str_new(ktap_state_t *ks, const char *str, size_t len);
8
9  #define kp_str_newz(ks, s)    (kp_str_new(ks, s, strlen(s)))
10
11  #include <linux/trace_seq.h>
12  int kp_str_fmt(ktap_state_t *ks, struct trace_seq *seq);
13
14  #endif /* __KTAP_STR_H__ */
```

runtime/kp_vm.h - ktap

Functions defined

- kp_vm_exit_thread
- kp_vm_new_thread
- kp_vm_try_to_exit

Macros defined

- __KTAP_VM_H__

Source code

```
#ifndef __KTAP_VM_H__
#define __KTAP_VM_H__

#include "kp_obj.h"

void kp_vm_call_proto(ktap_state_t *ks, ktap_proto_t *pt);
void kp_vm_call(ktap_state_t *ks, StkId func, int nresults);
int kp_vm_validate_code(ktap_state_t *ks, ktap_proto_t *pt, ktap_val_t *base);
void kp_vm_exit(ktap_state_t *ks);
 ktap_state_t *kp_vm_new_state(ktap_option_t *parm, struct dentry *dir);
 void kp_optimize_code(ktap_state_t *ks, int level, ktap_proto_t *f);
 int kp_vm_register_lib(ktap_state_t *ks, const char *libname,
                const ktap_libfunc_t *funcs);


 static __always_inline
 ktap_state_t *kp_vm_new_thread(ktap_state_t *mainthread, int rctx)
 {
     ktap_state_t *ks;

     ks = kp_this_cpu_state(mainthread, rctx);
     ks->top = ks->stack;
     return ks;
 }

 static __always_inline
 void kp_vm_exit_thread(ktap_state_t *ks)
 {
 }

 /*
 * This function only tell ktapvm this thread want to exit,
 * let mainthread handle real exit work later.
 */
 static __always_inline
 void kp_vm_try_to_exit(ktap_state_t *ks)
 {
     G(ks)->mainthread->stop = 1;
     G(ks)->state = KTAP_EXIT;
 }


 #endif /* __KTAP_VM_H__ */
```

runtime/kp_vm.c - ktap

Global variables defined

- bc_names

Functions defined

- adjust_varargs
- check_hot_loop
- checkstack
- findupval
- free_preserved_data
- func_closeuv
- func_new
- func_new_empty
- gfunc_add
- gfunc_get
- gfunc_getidx
- init_arguments
- init_preserved_data
- init_registry
- kp_freeupval
- kp_vm_call
- kp_vm_call_proto
- kp_vm_exit
- kp_vm_new_state
- kp_vm_register_lib
- kp_vm_validate_code
- poscall
- sl_wait_task_exit_actor
- sl_wait_task_pause_actor
- sleep_loop
- str_concat
- unlinkupval

- wait_user_completion
- wait_user_interrupt

Macros defined

- ALLOC_PERCPU
- BCNAME
- BCNAME
- BCNAME
- BCNAME
- DISPATCH
- KTAP_MIN_RESERVED_STACK_SIZE
- KTAP_PERCPU_BUFFER_SIZE
- KTAP_STACK_SIZE
- KTAP_STACK_SIZE_BYTES
- NUMADD
- NUMDIV
- NUMEQ
- NUMLE
- NUMLT
- NUMMOD
- NUMMUL
- NUMSUB
- NUMUNM
- RA
- RA
- RB
- RB
- RC
- RC
- RD
- RD
- RKD
- SIZE_KTAP_FUNC
- arith_NV

- arith_VN
- arith_VV
- dojump
- donextjump

Source code

```
/*
 * kp_vm.c - ktap script virtual machine in Linux kernel
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * Adapted from luajit and lua interpreter.
 * Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <linux/slab.h>
#include <linux/ftrace_event.h>
#include <linux/signal.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include "../include/ktap_types.h"
#include "../include/ktap_bc.h"
#include "../include/ktap_ffi.h"
#include "ktap.h"
#include "kp_obj.h"
#include "kp_str.h"
#include "kp_mempool.h"
#include "kp_tab.h"
#include "kp_transport.h"
#include "kp_vm.h"
#include "kp_events.h"

#define KTAP_MIN_RESERVED_STACK_SIZE 20
#define KTAP_STACK_SIZE        120 /* enlarge this value for big stack */
#define KTAP_STACK_SIZE_BYTES    (KTAP_STACK_SIZE * sizeof(ktap_val_t))

#define KTAP_PERCPU_BUFFER_SIZE    (3 * PAGE_SIZE)

static ktap_cfunction gfunc_get(ktap_state_t *ks, int idx);
static int gfunc_getidx(ktap_global_state_t *g, ktap_cfunction cfunc);

static ktap_str_t *str_concat(ktap_state_t *ks, StkId top, int start, int end)
{
    int i, len = 0;
    ktap_str_t *ts;
    char *ptr, *buffer;

    for (i = start; i <= end; i++) {
        if (!is_string(top + i)) {
            kp_error(ks, "cannot concat non-string\n");
            return NULL;
        }

```

```
64          len += rawtsvalue(top + i)->len;
65      }
66
67      if (len >= KTAP_PERCPU_BUFFER_SIZE) {
68          kp_error(ks, "Error: too long string concatenation\n");
69          return NULL;
70      }
71
72      preempt_disable_notrace();
73
74      buffer = kp_this_cpu_print_buffer(ks);
75      ptr = buffer;
76
77      for (i = start; i <= end; i++) {
78          int len = rawtsvalue(top + i)->len;
79          strncpy(ptr, svalue(top + i), len);
80          ptr += len;
81      }
82      ts = kp_str_new(ks, buffer, len);
83
84      preempt_enable_notrace();
85
86      return ts;
87  }
88
89  static ktap_upval_t *findupval(ktap_state_t *ks, StkId slot)
90  {
91      ktap_global_state_t *g = G(ks);
92      ktap_upval_t **pp = &ks->openupval;
93      ktap_upval_t *p;
94      ktap_upval_t *uv;
95
96      while (*pp != NULL && (p = *pp)->v >= slot) {
97          if (p->v == slot) {  /* found a corresponding upvalue? */
98              return p;
99          }
100          pp = (ktap_upval_t **)&p->nextgc;
101      }
102
103      /* not found: create a new one */
104      uv = (ktap_upval_t *)kp_malloc(ks, sizeof(ktap_upval_t));
105      if (!uv)
106          return NULL;
107      uv->gct = ~KTAP_TUPVAL;
108      uv->closed = 0; /* still open */
109      uv->v = slot;  /* current value lives in the stack */
110      /* Insert into sorted list of open upvalues. */
111      uv->nextgc = (ktap_obj_t *)*pp;
112      *pp = uv;
113      uv->prev = &g->uvhead;  /* double link it in `uvhead' list */
114      uv->next = g->uvhead.next;
115      uv->next->prev = uv;
116      g->uvhead.next = uv;
117      return uv;
118  }
119
120  static void unlinkupval(ktap_upval_t *uv)
121  {
122      uv->next->prev = uv->prev;  /* remove from `uvhead' list */
123      uv->prev->next = uv->next;
124  }
125
126  void kp_freeupval(ktap_state_t *ks, ktap_upval_t *uv)
127  {
128      if (!uv->closed)  /* is it open? */
129          unlinkupval(uv);  /* remove from open list */
130      kp_free(ks, uv);  /* free upvalue */
131  }
132
133  /* close upvals */
134  static void func_closeuv(ktap_state_t *ks, StkId level)
135  {
136      ktap_upval_t *uv;
137      ktap_global_state_t *g = G(ks);
138      while (ks->openupval != NULL &&
139          (uv = ks->openupval)->v >= level) {
```

```
        ktap_obj_t *o = obj2gco(uv);
        /* remove from `open' list */
        ks->openupval = (ktap_upval_t *)uv->nextgc;
        unlinkupval(uv);  /* remove upvalue from 'uvhead' list */
        set_obj(&uv->tv, uv->v);  /* move value to upvalue slot */
        uv->v = &uv->tv;  /* now current value lives here */
        uv->closed = 1;
        gch(o)->nextgc = g->allgc; /* link upvalue into 'allgc' list */
        g->allgc = o;
    }
}

#define SIZE_KTAP_FUNC(n) (sizeof(ktap_func_t) - sizeof(ktap_obj_t *) + \
                sizeof(ktap_obj_t *) * (n))
static ktap_func_t *func_new_empty(ktap_state_t *ks, ktap_proto_t *pt)
{
    ktap_func_t *fn;

    /* only mainthread can create new function */
    if (ks != G(ks)->mainthread) {
        kp_error(ks, "only mainthread can create function\n");
        return NULL;
    }

    fn = (ktap_func_t *)kp_obj_new(ks, SIZE_KTAP_FUNC(pt->sizeuv));
    if (!fn)
        return NULL;
    fn->gct = ~KTAP_TFUNC;
    fn->nupvalues = 0; /* Set to zero until upvalues are initialized. */
    fn->pc = proto_bc(pt);
    fn->p = pt;

    return fn;
}

static ktap_func_t *func_new(ktap_state_t *ks, ktap_proto_t *pt,
                 ktap_func_t *parent, ktap_val_t *base)
{
    ktap_func_t *fn;
    int nuv = pt->sizeuv, i;

    fn = func_new_empty(ks, pt);
    if (!fn)
        return NULL;

    fn->nupvalues = nuv;
    for (i = 0; i < nuv; i++) {
        uint32_t v = proto_uv(pt)[i];
        ktap_upval_t *uv;

        if (v & PROTO_UV_LOCAL) {
            uv = findupval(ks, base + (v & 0xff));
            if (!uv)
                return NULL;
            uv->immutable = ((v /PROTO_UV_IMMUTABLE) & 1);
        } else {
            uv = parent->upvals[v];
        }
        fn->upvals[i] = uv;
    }
    return fn;
}

static inline int checkstack(ktap_state_t *ks, int n)
{
    if (unlikely(ks->stack_last - ks->top <= n)) {
        kp_error(ks, "stack overflow, please enlarge stack size\n");
        return -1;
    }
    return 0;
}

static StkId adjust_varargs(ktap_state_t *ks, ktap_proto_t *p, int actual)
{
    int i;
    int nfixargs = p->numparams;
```

```
    StkId base, fixed;

    /* move fixed parameters to final position */
    fixed = ks->top - actual;  /* first fixed argument */
    base = ks->top;  /* final position of first argument */

    for (i=0; i < nfixargs; i++) {
        set_obj(ks->top++, fixed + i);
        set_nil(fixed + i);
    }

    return base;
}

static void poscall(ktap_state_t *ks, StkId func, StkId first_result,
           int wanted)
{
    int i;

    for (i = wanted; i != 0 && first_result < ks->top; i--)
        set_obj(func++, first_result++);

    while(i-- > 0)
        set_nil(func++);
}

void kp_vm_call_proto(ktap_state_t *ks, ktap_proto_t *pt)
{
    ktap_func_t *fn;

    fn = func_new_empty(ks, pt);
    if (!fn)
        return;
    set_func(ks->top++, fn);
    kp_vm_call(ks, ks->top - 1, 0);
}

/*
* Hot loop detaction
*
* Check hot loop detaction in three cases:
* 1. jmp -x: this happens in 'while (expr) { ... }'
* 2. FORPREP-FORLOOP
* 3. TFORCALL-TFORLOOP
*/
static __always_inline int check_hot_loop(ktap_state_t *ks, int loop_count)
{
    if (unlikely(loop_count == kp_max_loop_count)) {
        kp_error(ks, "loop execute count exceed max limit(%d)\n",
                 kp_max_loop_count);
        return -1;
    }

    return 0;
}

#define dojump(i, e) { pc += (int)bc_d(i) - BCBIAS_J + e; }
#define donextjump  { instr = *pc; dojump(instr, 1); }

#define NUMADD(a, b)    ((a) + (b))
#define NUMSUB(a, b)    ((a) - (b))
#define NUMMUL(a, b)    ((a) * (b))
#define NUMDIV(a, b)    ((a) / (b))
#define NUMUNM(a)       (-(a))
#define NUMEQ(a, b)     ((a) == (b))
#define NUMLT(a, b)     ((a) < (b))
#define NUMLE(a, b)     ((a) <= (b))
#define NUMMOD(a, b)    ((a) % (b))

#define arith_VV(ks, op) { \
    ktap_val_t *rb = RB; \
    ktap_val_t *rc = RC; \
    if (is_number(rb) && is_number(rc)) { \
        ktap_number nb = nvalue(rb), nc = nvalue(rc); \
        set_number(RA, op(nb, nc)); \
    } else {    \
```

```
292         kp_puts(ks, "Error: Cannot make arith operation\n");    \
293         return;    \
294     } }
295
296 #define arith_VN(ks, op) { \
297     ktap_val_t *rb = RB; \
298     if (is_number(rb)) { \
299         ktap_number nb = nvalue(rb);\
300         ktap_number nc = nvalue((ktap_val_t *)kbase + bc_c(instr));\
301         set_number(RA, op(nb, nc)); \
302     } else {    \
303         kp_puts(ks, "Error: Cannot make arith operation\n");    \
304         return;    \
305     } }
306
307 #define arith_NV(ks, op) { \
308     ktap_val_t *rb = RB; \
309     if (is_number(rb)) { \
310         ktap_number nb = nvalue(rb);\
311         ktap_number nc = nvalue((ktap_val_t *)kbase + bc_c(instr));\
312         set_number(RA, op(nc, nb)); \
313     } else {    \
314         kp_puts(ks, "Error: Cannot make arith operation\n");    \
315         return;    \
316     } }
317
318
319 static const char * const bc_names[] = {
320 #define BCNAME(name, ma, mb, mc, mt)       #name,
321     BCDEF(BCNAME)
322 #undef BCNAME
323     NULL
324 };
325
326
327 /*
328 * ktap bytecode interpreter routine
329 *
330 *
331 * kp_vm_call only can be used for:
332 * 1). call ktap function, not light C function
333 * 2). accept fixed argument function
334 */
335 void kp_vm_call(ktap_state_t *ks, StkId func, int nresults)
336 {
337     int loop_count = 0;
338     ktap_func_t *fn;
339     ktap_proto_t *pt;
340     ktap_obj_t **kbase;
341     unsigned int instr, op;
342     const unsigned int *pc;
343     StkId base; /* stack pointer */
344     int multres = 0; /* temp varible */
345     ktap_tab_t *gtab = G(ks)->gtab;
346
347     /* use computed goto for opcode dispatch */
348
349     static void *dispatch_table[] = {
350 #define BCNAME(name, ma, mb, mc, mt)       &&DO_BC_##name,
351         BCDEF(BCNAME)
352 #undef BCNAME
353     };
354
355 #define DISPATCH()                \
356     do {                    \
357         instr = *(pc++);        \
358         op = bc_op(instr);        \
359         goto *dispatch_table[op];    \
360     } while (0)
361
362 #define RA    (base + bc_a(instr))
363 #define RB    (base + bc_b(instr))
364 #define RC    (base + bc_c(instr))
365 #define RD    (base + bc_d(instr))
366 #define RKD    ((ktap_val_t *)kbase + bc_d(instr))
367
```

```
368     /*TODO: fix argument number mismatch, example: sort cmp closure */
369
370     fn = clvalue(func);
371     pt = fn->p;
372     kbase = fn->p->k;
373     base = func + 1;
374     pc = proto_bc(pt) + 1;
375     ks->top = base + pt->framesize;
376     func->pcr = 0; /* no previous frame */
377
378     /* main loop of interpreter */
379     DISPATCH();
380
381     while (1) {
382     DO_BC_ISLT: /* Jump if A < D */
383         if (!is_number(RA) || !is_number(RD)) {
384             kp_error(ks, "compare with non-number\n");
385             return;
386         }
387
388         if (nvalue(RA) >= nvalue(RD))
389             pc++;
390         else
391             donextjump;
392         DISPATCH();
393     DO_BC_ISGE: /* Jump if A >= D */
394         if (!is_number(RA) || !is_number(RD)) {
395             kp_error(ks, "compare with non-number\n");
396             return;
397         }
398
399         if (nvalue(RA) < nvalue(RD))
400             pc++;
401         else
402             donextjump;
403         DISPATCH();
404     DO_BC_ISLE: /* Jump if A <= D */
405         if (!is_number(RA) || !is_number(RD)) {
406             kp_error(ks, "compare with non-number\n");
407             return;
408         }
409
410         if (nvalue(RA) > nvalue(RD))
411             pc++;
412         else
413             donextjump;
414         DISPATCH();
415     DO_BC_ISGT: /* Jump if A > D */
416         if (!is_number(RA) || !is_number(RD)) {
417             kp_error(ks, "compare with non-number\n");
418             return;
419         }
420
421         if (nvalue(RA) <= nvalue(RD))
422             pc++;
423         else
424             donextjump;
425         DISPATCH();
426     DO_BC_ISEQV: /* Jump if A = D */
427         if (!kp_obj_equal(RA, RD))
428             pc++;
429         else
430             donextjump;
431         DISPATCH();
432     DO_BC_ISNEV: /* Jump if A != D */
433         if (kp_obj_equal(RA, RD))
434             pc++;
435         else
436             donextjump;
437         DISPATCH();
438     DO_BC_ISEQS: { /* Jump if A = D */
439         int idx = ~bc_d(instr);
440
441         if (!is_string(RA) ||
442                 rawtsvalue(RA) != (ktap_str_t *)kbase[idx])
443             pc++;
```

```
444         else
445             donextjump;
446         DISPATCH();
447         }
448     DO_BC_ISNES: { /* Jump if A != D */
449         int idx = ~bc_d(instr);
450
451         if (is_string(RA) &&
452             rawtsvalue(RA) == (ktap_str_t *)kbase[idx])
453             pc++;
454         else
455             donextjump;
456         DISPATCH();
457         }
458     DO_BC_ISEQN: /* Jump if A = D */
459         if (!is_number(RA) || nvalue(RA) !=  nvalue(RKD))
460             pc++;
461         else
462             donextjump;
463         DISPATCH();
464     DO_BC_ISNEN: /* Jump if A != D */
465         if (is_number(RA) && nvalue(RA) ==  nvalue(RKD))
466             pc++;
467         else
468             donextjump;
469         DISPATCH();
470     DO_BC_ISEQP: /* Jump if A = D */
471         if (itype(RA) != ~bc_d(instr))
472             pc++;
473         else
474             donextjump;
475         DISPATCH();
476     DO_BC_ISNEP: /* Jump if A != D */
477         if (itype(RA) == ~bc_d(instr))
478             pc++;
479         else
480             donextjump;
481         DISPATCH();
482     DO_BC_ISTC: /* Copy D to A and jump, if D is true */
483         if (itype(RD) == KTAP_TNIL || itype(RD) == KTAP_TFALSE)
484             pc++;
485         else {
486             set_obj(RA, RD);
487             donextjump;
488         }
489         DISPATCH();
490     DO_BC_ISFC: /* Copy D to A and jump, if D is false */
491         if (itype(RD) != KTAP_TNIL && itype(RD) != KTAP_TFALSE)
492             pc++;
493         else {
494             set_obj(RA, RD);
495             donextjump;
496         }
497         DISPATCH();
498     DO_BC_IST: /* Jump if D is true */
499         if (itype(RD) == KTAP_TNIL || itype(RD) == KTAP_TFALSE)
500             pc++;
501         else
502             donextjump;
503         DISPATCH();
504     DO_BC_ISF: /* Jump if D is false */
505         /* only nil and false are considered false,
506          * all other values are true */
507         if (itype(RD) != KTAP_TNIL && itype(RD) != KTAP_TFALSE)
508             pc++;
509         else
510             donextjump;
511         DISPATCH();
512     DO_BC_ISTYPE: /* generated by genlibbc, not compiler; not used now */
513     DO_BC_ISNUM:
514         return;
515     DO_BC_MOV: /* Copy D to A */
516         set_obj(RA, RD);
517         DISPATCH();
518     DO_BC_NOT: /* Set A to boolean not of D */
519         if (itype(RD) == KTAP_TNIL || itype(RD) == KTAP_TFALSE)
```

```
            setitype(RA, KTAP_TTRUE);
        else
            setitype(RA, KTAP_TFALSE);

        DISPATCH();
    DO_BC_UNM: /* Set A to -D (unary minus) */
        if (!is_number(RD)) {
            kp_error(ks, "use '-' operator on non-number\n");
            return;
        }

        set_number(RA, -nvalue(RD));
        DISPATCH();
    DO_BC_ADDVN: /* A = B + C */
        arith_VN(ks, NUMADD);
        DISPATCH();
    DO_BC_SUBVN: /* A = B - C */
        arith_VN(ks, NUMSUB);
        DISPATCH();
    DO_BC_MULVN: /* A = B * C */
        arith_VN(ks, NUMMUL);
        DISPATCH();
    DO_BC_DIVVN: /* A = B / C */
        /* divide 0 checking */
        if (!nvalue((ktap_val_t *)kbase + bc_c(instr))) {
            kp_error(ks, "divide 0 arith operation\n");
            return;
        }
        arith_VN(ks, NUMDIV);
        DISPATCH();
    DO_BC_MODVN: /* A = B % C */
        /* divide 0 checking */
        if (!nvalue((ktap_val_t *)kbase + bc_c(instr))) {
            kp_error(ks, "mod 0 arith operation\n");
            return;
        }
        arith_VN(ks, NUMMOD);
        DISPATCH();
    DO_BC_ADDNV: /* A = C + B */
        arith_NV(ks, NUMADD);
        DISPATCH();
    DO_BC_SUBNV: /* A = C - B */
        arith_NV(ks, NUMSUB);
        DISPATCH();
    DO_BC_MULNV: /* A = C * B */
        arith_NV(ks, NUMMUL);
        DISPATCH();
    DO_BC_DIVNV: /* A = C / B */
        /* divide 0 checking */
        if (!nvalue(RB)){
            kp_error(ks, "divide 0 arith operation\n");
            return;
        }
        arith_NV(ks, NUMDIV);
        DISPATCH();
    DO_BC_MODNV: /* A = C % B */
        /* divide 0 checking */
        if (!nvalue(RB)){
            kp_error(ks, "mod 0 arith operation\n");
            return;
        }
        arith_NV(ks, NUMMOD);
        DISPATCH();
    DO_BC_ADDVV: /* A = B + C */
        arith_VV(ks, NUMADD);
        DISPATCH();
    DO_BC_SUBVV: /* A = B - C */
        arith_VV(ks, NUMSUB);
        DISPATCH();
    DO_BC_MULVV: /* A = B * C */
        arith_VV(ks, NUMMUL);
        DISPATCH();
    DO_BC_DIVVV: /* A = B / C */
        arith_VV(ks, NUMDIV);
        DISPATCH();
    DO_BC_MODVV: /* A = B % C */
```

```
        arith_VV(ks, NUMMOD);
        DISPATCH();
    DO_BC_POW: /* A = B ^ C, rejected */
        return;
    DO_BC_CAT: { /* A = B .. ~ .. C */
        /* The CAT instruction concatenates all values in
         * variable slots B to C inclusive. */
        ktap_str_t *ts = str_concat(ks, base, bc_b(instr),
                        bc_c(instr));
        if (!ts)
            return;

        set_string(RA, ts);
        DISPATCH();
        }
    DO_BC_KSTR: { /* Set A to string constant D */
        int idx = ~bc_d(instr);
        set_string(RA, (ktap_str_t *)kbase[idx]);
        DISPATCH();
        }
    DO_BC_KCDATA: /* not used now */
        DISPATCH();
    DO_BC_KSHORT: /* Set A to 16 bit signed integer D */
        set_number(RA, bc_d(instr));
        DISPATCH();
    DO_BC_KNUM: /* Set A to number constant D */
        set_number(RA, nvalue(RKD));
        DISPATCH();
    DO_BC_KPRI: /* Set A to primitive D */
        setitype(RA, ~bc_d(instr));
        DISPATCH();
    DO_BC_KNIL: { /* Set slots A to D to nil */
        int i;
        for (i = 0; i <= bc_d(instr) - bc_a(instr); i++) {
            set_nil(RA + i);
        }
        DISPATCH();
        }
    DO_BC_UGET: /* Set A to upvalue D */
        set_obj(RA, fn->upvals[bc_d(instr)]->v);
        DISPATCH();
    DO_BC_USETV: /* Set upvalue A to D */
        set_obj(fn->upvals[bc_a(instr)]->v, RD);
        DISPATCH();
    DO_BC_UINCV: { /* upvalus[A] += D */
        ktap_val_t *v = fn->upvals[bc_a(instr)]->v;
        if (unlikely(!is_number(RD) || !is_number(v))) {
            kp_error(ks, "use '+=' on non-number\n");
            return;
        }
        set_number(v, nvalue(v) + nvalue(RD));
        DISPATCH();
        }
    DO_BC_USETS: { /* Set upvalue A to string constant D */
        int idx = ~bc_d(instr);
        set_string(fn->upvals[bc_a(instr)]->v,
                (ktap_str_t *)kbase[idx]);
        DISPATCH();
        }
    DO_BC_USETN: /* Set upvalue A to number constant D */
        set_number(fn->upvals[bc_a(instr)]->v, nvalue(RKD));
        DISPATCH();
    DO_BC_UINCN: { /* upvalus[A] += D */
        ktap_val_t *v = fn->upvals[bc_a(instr)]->v;
        if (unlikely(!is_number(v))) {
            kp_error(ks, "use '+=' on non-number\n");
            return;
        }
        set_number(v, nvalue(v) + nvalue(RKD));
        DISPATCH();
        }
    DO_BC_USETP: /* Set upvalue A to primitive D */
        setitype(fn->upvals[bc_a(instr)]->v, ~bc_d(instr));
        DISPATCH();
    DO_BC_UCLO: /* Close upvalues for slots . rbase and jump to target D */
        if (ks->openupval != NULL)
```

```
672             func_closeuv(ks, RA);
673         dojump(instr, 0);
674         DISPATCH();
675     DO_BC_FNEW: {
676         /* Create new closure from prototype D and store it in A */
677         int idx = ~bc_d(instr);
678         ktap_func_t *subfn = func_new(ks, (ktap_proto_t *)kbase[idx],
679                           fn, base);
680         if (unlikely(!subfn))
681             return;
682         set_func(RA, subfn);
683         DISPATCH();
684         }
685     DO_BC_TNEW: { /* Set A to new table with size D */
686         /*
687          * preallocate default narr and nrec,
688          * op_b and op_c is not used
689          * This would allocate more memory for some static table.
690          */
691         ktap_tab_t *t = kp_tab_new_ah(ks, 0, 0);
692         if (unlikely(!t))
693             return;
694         set_table(RA, t);
695         DISPATCH();
696         }
697     DO_BC_TDUP: { /* Set A to duplicated template table D */
698         int idx = ~bc_d(instr);
699         ktap_tab_t *t = kp_tab_dup(ks, (ktap_tab_t *)kbase[idx]);
700         if (!t)
701             return;
702         set_table(RA, t);
703         DISPATCH();
704         }
705     DO_BC_GGET: { /* A = _G[D] */
706         int idx = ~bc_d(instr);
707         kp_tab_getstr(gtab, (ktap_str_t *)kbase[idx], RA);
708         DISPATCH();
709         }
710     DO_BC_GSET: /* _G[D] = A, rejected. */
711     DO_BC_GINC: /* _G[D] += A, rejected. */
712         return;
713     DO_BC_TGETV: /* A = B[C] */
714         if (unlikely(!is_table(RB))) {
715             kp_error(ks, "get key from non-table\n");
716             return;
717         }
718
719         kp_tab_get(ks, hvalue(RB), RC, RA);
720         DISPATCH();
721     DO_BC_TGETS: { /* A = B[C] */
722         int idx = ~bc_c(instr);
723
724         if (unlikely(!is_table(RB))) {
725             kp_error(ks, "get key from non-table\n");
726             return;
727         }
728         kp_tab_getstr(hvalue(RB), (ktap_str_t *)kbase[idx], RA);
729         DISPATCH();
730         }
731     DO_BC_TGETB: { /* A = B[C] */
732         /* 8 bit literal C operand as an unsigned integer
733          * index (0..255)) */
734         uint8_t idx = bc_c(instr);
735
736         if (unlikely(!is_table(RB))) {
737             kp_error(ks, "set key to non-table\n");
738             return;
739         }
740         kp_tab_getint(hvalue(RB), idx, RA);
741         DISPATCH();
742         }
743     DO_BC_TGETR: /* generated by genlibbc, not compiler, not used */
744         return;
745     DO_BC_TSETV: /* B[C] = A */
746         if (unlikely(!is_table(RB))) {
747             kp_error(ks, "set key to non-table\n");
```

```
            return;
        }
        kp_tab_set(ks, hvalue(RB), RC, RA);
        DISPATCH();
    DO_BC_TINCV: /* B[C] += A */
        if (unlikely(!is_table(RB))) {
            kp_error(ks, "set key to non-table\n");
            return;
        }
        if (unlikely(!is_number(RA))) {
            kp_error(ks, "use '+=' on non-number\n");
            return;
        }
        kp_tab_incr(ks, hvalue(RB), RC, nvalue(RA));
        DISPATCH();
    DO_BC_TSETS: { /* B[C] = A */
        int idx = ~bc_c(instr);

        if (unlikely(!is_table(RB))) {
            kp_error(ks, "set key to non-table\n");
            return;
        }
        kp_tab_setstr(ks, hvalue(RB), (ktap_str_t *)kbase[idx], RA);
        DISPATCH();
        }
    DO_BC_TINCS: { /* B[C] += A */
        int idx = ~bc_c(instr);

        if (unlikely(!is_table(RB))) {
            kp_error(ks, "set key to non-table\n");
            return;
        }
        if (unlikely(!is_number(RA))) {
            kp_error(ks, "use '+=' on non-number\n");
            return;
        }
        kp_tab_incrstr(ks, hvalue(RB), (ktap_str_t *)kbase[idx],
                nvalue(RA));
        DISPATCH();
        }
    DO_BC_TSETB: { /* B[C] = A */
        /* 8 bit literal C operand as an unsigned integer
         * index (0..255)) */
        uint8_t idx = bc_c(instr);

        if (unlikely(!is_table(RB))) {
            kp_error(ks, "set key to non-table\n");
            return;
        }
        kp_tab_setint(ks, hvalue(RB), idx, RA);
        DISPATCH();
        }
    DO_BC_TINCB: { /* B[C] = A */
        uint8_t idx = bc_c(instr);

        if (unlikely(!is_table(RB))) {
            kp_error(ks, "set key to non-table\n");
            return;
        }
        if (unlikely(!is_number(RA))) {
            kp_error(ks, "use '+=' on non-number\n");
            return;
        }
        kp_tab_incrint(ks, hvalue(RB), idx, nvalue(RA));
        DISPATCH();
        }
    DO_BC_TSETM: /* don't support */
        return;
    DO_BC_TSETR: /* generated by genlibbc, not compiler, not used */
        return;
    DO_BC_CALLM:
    DO_BC_CALL: { /* b: return_number + 1; c: argument + 1 */
        int c = bc_c(instr);
        int nresults = bc_b(instr) - 1;
        StkId oldtop = ks->top;
        StkId newfunc = RA;
```

```
824
825         if (op == BC_CALL && c != 0)
826             ks->top = RA + c;
827         else if (op == BC_CALLM)
828             ks->top = RA + c + multres;
829
830         if (itype(newfunc) == KTAP_TCFUNC) { /* light C function */
831             ktap_cfunction f = fvalue(newfunc);
832             int n;
833
834             if (unlikely(checkstack(ks,
835                     KTAP_MIN_RESERVED_STACK_SIZE)))
836                 return;
837
838             ks->func = newfunc;
839             n = (*f)(ks);
840             if (unlikely(n < 0)) /* error occured */
841                 return;
842             poscall(ks, newfunc, ks->top - n, nresults);
843
844             ks->top = oldtop;
845             multres = n + 1; /* set to multres */
846             DISPATCH();
847         } else if (itype(newfunc) == KTAP_TFUNC) { /* ktap function */
848             int n;
849
850             func = newfunc;
851             pt = clvalue(func)->p;
852
853             if (unlikely(checkstack(ks, pt->framesize)))
854                 return;
855
856             /* get number of real arguments */
857             n = (int)(ks->top - func) - 1;
858
859             /* complete missing arguments */
860             for (; n < pt->numparams; n++)
861                 set_nil(ks->top++);
862
863             base = (!(pt->flags & PROTO_VARARG)) ? func + 1 :
864                         adjust_varargs(ks, pt, n);
865
866             fn = clvalue(func);
867             pt = fn->p;
868             kbase = pt->k;
869             func->pcr = pc - 1; /* save pc */
870             ks->top = base + pt->framesize;
871             pc = proto_bc(pt) + 1; /* starting point */
872             DISPATCH();
873         } else {
874             kp_error(ks, "attempt to call nil function\n");
875             return;
876         }
877         }
878     DO_BC_CALLMT: /* don't support */
879         return;
880     DO_BC_CALLT: { /* Tailcall: return A(A+1, ..., A+D-1) */
881         StkId nfunc = RA;
882
883         if (itype(nfunc) == KTAP_TCFUNC) { /* light C function */
884             kp_error(ks, "don't support callt for C function");
885             return;
886         } else if (itype(nfunc) == KTAP_TFUNC) { /* ktap function */
887             int aux;
888
889             /*
890              * tail call: put called frame (n) in place of
891              * caller one (o)
892              */
893             StkId ofunc = func; /* caller function */
894             /* last stack slot filled by 'precall' */
895             StkId lim = nfunc + 1 + clvalue(nfunc)->p->numparams;
896
897             fn = clvalue(nfunc);
898             ofunc->val = nfunc->val;
899
```

```
            /* move new frame into old one */
            for (aux = 1; nfunc + aux < lim; aux++)
                set_obj(ofunc + aux, nfunc + aux);

            pt = fn->p;
            kbase = pt->k;
            ks->top = base + pt->framesize;
            pc = proto_bc(pt) + 1; /* starting point */
            DISPATCH();
        } else {
            kp_error(ks, "attempt to call nil function\n");
            return;
        }
        }
    DO_BC_ITERC: /* don't support it now */
        return;
    DO_BC_ITERN: /* Specialized ITERC, if iterator function A-3 is next()*/
        /* detect hot loop */
        if (unlikely(check_hot_loop(ks, loop_count++) < 0))
            return;

        if (kp_tab_next(ks, hvalue(RA - 2), RA)) {
            donextjump; /* Get jump target from ITERL */
        } else {
            pc++; /* jump to ITERL + 1 */
        }
        DISPATCH();
    DO_BC_VARG: /* don't support */
        return;
    DO_BC_ISNEXT: /* Verify ITERN specialization and jump */
        if (!is_cfunc(RA - 3) || !is_table(RA - 2) || !is_nil(RA - 1)
            || fvalue(RA - 3) != (ktap_cfunction)kp_tab_next) {
            /* Despecialize bytecode if any of the checks fail. */
            setbc_op(pc - 1, BC_JMP);
            dojump(instr, 0);
            setbc_op(pc, BC_ITERC);
        } else {
            dojump(instr, 0);
            set_nil(RA); /* init control variable */
        }
        DISPATCH();
    DO_BC_RETM: /* don't support return multiple values */
    DO_BC_RET:
        return;
    DO_BC_RET0:
        /* if it's called from external invocation, just return */
        if (!func->pcr)
            return;

        pc = func->pcr; /* restore PC */

        multres = bc_d(instr);
        set_nil(func);

        base = func - bc_a(*pc);
        func = base - 1;
        fn = clvalue(func);
        kbase = fn->p->k;
        ks->top = base + pt->framesize;
        pc++;

        DISPATCH();
    DO_BC_RET1:
        /* if it's called from external invocation, just return */
        if (!func->pcr)
            return;

        pc = func->pcr; /* restore PC */

        multres = bc_d(instr);
        set_obj(base - 1, RA); /* move result */

        base = func - bc_a(*pc);
        func = base - 1;
        fn = clvalue(func);
        kbase = fn->p->k;
```

```
976         ks->top = base + pt->framesize;
977         pc++;
978 
979         DISPATCH();
980       DO_BC_FORI: { /* Numeric 'for' loop init */
981         ktap_number idx;
982         ktap_number limit;
983         ktap_number step;
984 
985         if (unlikely(!is_number(RA) || !is_number(RA + 1) ||
986                 !is_number(RA + 2))) {
987             kp_error(ks, KTAP_QL("for")
988                  " init/limit/step value must be a number\n");
989             return;
990         }
991 
992         idx = nvalue(RA);
993         limit = nvalue(RA + 1);
994         step = nvalue(RA + 2);
995 
996         if (NUMLT(0, step) ? NUMLE(idx, limit) : NUMLE(limit, idx)) {
997             set_number(RA + 3, nvalue(RA));
998         } else {
999             dojump(instr, 0);
1000         }
1001         DISPATCH();
1002         }
1003       DO_BC_JFORI: /* not used */
1004         return;
1005       DO_BC_FORL: { /* Numeric 'for' loop */
1006         ktap_number step = nvalue(RA + 2);
1007         /* increment index */
1008         ktap_number idx = NUMADD(nvalue(RA), step);
1009         ktap_number limit = nvalue(RA + 1);
1010         if (NUMLT(0, step) ? NUMLE(idx, limit) : NUMLE(limit, idx)) {
1011             dojump(instr, 0); /* jump back */
1012             set_number(RA, idx);  /* update internal index... */
1013             set_number(RA + 3, idx);  /* ...and external index */
1014         }
1015 
1016         if (unlikely(check_hot_loop(ks, loop_count++) < 0))
1017             return;
1018 
1019         DISPATCH();
1020         }
1021       DO_BC_IFORL: /* not used */
1022       DO_BC_JFORL:
1023       DO_BC_ITERL:
1024       DO_BC_IITERL:
1025       DO_BC_JITERL:
1026         return;
1027       DO_BC_LOOP: /* Generic loop */
1028         /* ktap use this bc to detect hot loop */
1029         if (unlikely(check_hot_loop(ks, loop_count++) < 0))
1030             return;
1031         DISPATCH();
1032       DO_BC_ILOOP: /* not used */
1033       DO_BC_JLOOP:
1034         return;
1035       DO_BC_JMP: /* Jump */
1036         dojump(instr, 0);
1037         DISPATCH();
1038       DO_BC_FUNCF: /* function header, not used */
1039       DO_BC_IFUNCF:
1040       DO_BC_JFUNCF:
1041       DO_BC_FUNCV:
1042       DO_BC_IFUNCV:
1043       DO_BC_JFUNCV:
1044       DO_BC_FUNCC:
1045       DO_BC_FUNCCW:
1046         return;
1047       DO_BC_VARGN: /* arg0 .. arg9*/
1048         if (unlikely(!ks->current_event)) {
1049             kp_error(ks, "invalid event context\n");
1050             return;
1051         }
```

```
1052
1053            kp_event_getarg(ks, RA, bc_d(instr));
1054            DISPATCH();
1055        DO_BC_VARGSTR: { /* argstr */
1056            /*
1057             * If you pass argstr to print/printf function directly,
1058             * then no extra string generated, so don't worry string
1059             * poll size for below case:
1060             *     print(argstr)
1061             *
1062             * If you use argstr as table key like below, then it may
1063             * overflow your string pool size, so be care of on it.
1064             *     table[argstr] = V
1065             *
1066             * If you assign argstr to upval or table value like below,
1067             * it don't really write string, just write type KTAP_TEVENTSTR,
1068             * the value will be interpreted when value print out in valid
1069             * event context, if context mismatch, error will report.
1070             *     table[V] = argstr
1071             *     upval = argstr
1072             *
1073             * If you want to save real string of argstr, then use it like
1074             * below, again, be care of string pool size in this case.
1075             *     table[V] = stringof(argstr)
1076             *     upval = stringof(argstr)
1077             */
1078            struct ktap_event_data *e = ks->current_event;
1079
1080            if (unlikely(!e)) {
1081                kp_error(ks, "invalid event context\n");
1082                return;
1083            }
1084
1085            if (e->argstr) /* argstr been stringified */
1086                set_string(RA, e->argstr);
1087            else
1088                set_eventstr(RA);
1089            DISPATCH();
1090            }
1091        DO_BC_VPROBENAME: { /* probename */
1092            struct ktap_event_data *e = ks->current_event;
1093
1094            if (unlikely(!e)) {
1095                kp_error(ks, "invalid event context\n");
1096                return;
1097            }
1098            set_string(RA, e->event->name);
1099            DISPATCH();
1100            }
1101        DO_BC_VPID: /* pid */
1102            set_number(RA, (int)current->pid);
1103            DISPATCH();
1104        DO_BC_VTID: /* tid */
1105            set_number(RA, (int)task_pid_vnr(current));
1106            DISPATCH();
1107        DO_BC_VUID: { /* uid */
1108    #if LINUX_VERSION_CODE >= KERNEL_VERSION(3, 5, 0)
1109            uid_t uid = from_kuid_munged(current_user_ns(), current_uid());
1110    #else
1111            uid_t uid = current_uid();
1112    #endif
1113            set_number(RA, (int)uid);
1114            DISPATCH();
1115            }
1116        DO_BC_VCPU: /* cpu */
1117            set_number(RA, smp_processor_id());
1118            DISPATCH();
1119        DO_BC_VEXECNAME: { /* execname */
1120            ktap_str_t *ts = kp_str_newz(ks, current->comm);
1121            if (unlikely(!ts))
1122                return;
1123            set_string(RA, ts);
1124            DISPATCH();
1125            }
1126        DO_BC_GFUNC: { /* Call built-in C function, patched by BC_GGET */
1127            ktap_cfunction cfunc = gfunc_get(ks, bc_d(instr));
```

```
        set_cfunc(RA, cfunc);
        DISPATCH();
        }
    }
}

/*
* Validate byte code and static analysis.
*
* TODO: more type checking before real running.
*/
int kp_vm_validate_code(ktap_state_t *ks, ktap_proto_t *pt, ktap_val_t *base)
{
    const unsigned int *pc = proto_bc(pt) + 1;
    unsigned int instr, op;
    ktap_obj_t **kbase = pt->k;
    ktap_tab_t *gtab = G(ks)->gtab;
    int i;

#define RA    (base + bc_a(instr))
#define RB    (base + bc_b(instr))
#define RC    (base + bc_c(instr))
#define RD    (base + bc_d(instr))

    if (pt->framesize > KP_MAX_SLOTS) {
        kp_error(ks, "exceed max frame size %d\n", pt->framesize);
        return -1;
    }

    if (base + pt->framesize > ks->stack_last) {
        kp_error(ks, "stack overflow\n");
        return -1;
    }

    for (i = 0; i < pt->sizebc - 1; i++) {
        instr = *pc++;
        op = bc_op(instr);


        if (op >= BC__MAX) {
            kp_error(ks, "unknown byte code %d\n", op);
            return -1;
        }

        switch (op) {
        case BC_FNEW: {
            int idx = ~bc_d(instr);
            ktap_proto_t *newpt = (ktap_proto_t *)kbase[idx];
            if (kp_vm_validate_code(ks, newpt, RA + 1))
                return -1;

            break;
            }
        case BC_RETM: case BC_RET:
            kp_error(ks, "don't support return multiple values\n");
            return -1;
        case BC_GSET: case BC_GINC: { /* _G[D] = A, _G[D] += A */
            int idx = ~bc_d(instr);
            ktap_str_t *ts = (ktap_str_t *)kbase[idx];
            kp_error(ks, "cannot set global variable '%s'\n",
                    getstr(ts));
            return -1;
            }
        case BC_GGET: {
            int idx = ~bc_d(instr);
            ktap_str_t *ts = (ktap_str_t *)kbase[idx];
            ktap_val_t val;
            kp_tab_getstr(gtab, ts, &val);
            if (is_nil(&val)) {
                kp_error(ks, "undefined global variable"
                        " '%s'\n", getstr(ts));
                return -1;
            } else if (is_cfunc(&val)) {
                int idx = gfunc_getidx(G(ks), fvalue(&val));
                if (idx >= 0) {
                    /* patch BC_GGET bytecode to BC_GFUNC */
```

```
                    setbc_op(pc - 1, BC_GFUNC);
                    setbc_d(pc - 1, idx);
                }
            }
            break;
            }
        case BC_ITERC:
            kp_error(ks, "ktap only support pairs iteraor\n");
            return -1;
        case BC_POW:
            kp_error(ks, "ktap don't support pow arith\n");
            return -1;
        }
    }

    return 0;
}

/* return cfunction by idx */
static ktap_cfunction gfunc_get(ktap_state_t *ks, int idx)
{
    return G(ks)->gfunc_tbl[idx];
}

/* get cfunction index, the index is for fast get cfunction in runtime */
static int gfunc_getidx(ktap_global_state_t *g, ktap_cfunction cfunc)
{
    int nr = g->nr_builtin_cfunction;
    ktap_cfunction *gfunc_tbl = g->gfunc_tbl;
    int i;

    for (i = 0; i < nr; i++) {
        if (gfunc_tbl[i] == cfunc)
            return i;
    }

    return -1;
}

static void gfunc_add(ktap_state_t *ks, ktap_cfunction cfunc)
{
    int nr = G(ks)->nr_builtin_cfunction;

    if (nr == KP_MAX_CACHED_CFUNCTION) {
        kp_error(ks, "please enlarge KP_MAX_CACHED_CFUNCTION %d\n",
                KP_MAX_CACHED_CFUNCTION);
        return;
    }
    G(ks)->gfunc_tbl[nr] = cfunc;
    G(ks)->nr_builtin_cfunction++;
}

/* function for register library */
int kp_vm_register_lib(ktap_state_t *ks, const char *libname,
               const ktap_libfunc_t *funcs)
{
    ktap_tab_t *gtab = G(ks)->gtab;
    ktap_tab_t *target_tbl;
    int i;

    /* lib is null when register baselib function */
    if (libname == NULL)
        target_tbl = gtab;
    else {
        ktap_val_t key, val;
        ktap_str_t *ts = kp_str_newz(ks, libname);
        if (!ts)
            return -ENOMEM;

        /* calculate the function number contained by this library */
        for (i = 0; funcs[i].name != NULL; i++) {
        }

        target_tbl = kp_tab_new_ah(ks, 0, i + 1);
        if (!target_tbl)
            return -ENOMEM;
```

```
1280
1281        set_string(&key, ts);
1282        set_table(&val, target_tbl);
1283        kp_tab_set(ks, gtab, &key, &val);
1284    }
1285
1286    /* TODO: be care of same function name issue, foo() and tbl.foo() */
1287    for (i = 0; funcs[i].name != NULL; i++) {
1288        ktap_str_t *func_name = kp_str_newz(ks, funcs[i].name);
1289        ktap_val_t fn;
1290
1291        if (unlikely(!func_name))
1292            return -ENOMEM;
1293
1294        set_cfunc(&fn, funcs[i].func);
1295        kp_tab_setstr(ks, target_tbl, func_name, &fn);
1296
1297        gfunc_add(ks, funcs[i].func);
1298    }
1299
1300    return 0;
1301 }
1302
1303 static int init_registry(ktap_state_t *ks)
1304 {
1305    ktap_tab_t *registry = kp_tab_new_ah(ks, 2, 0);
1306    ktap_val_t gtbl;
1307    ktap_tab_t *t;
1308
1309    if (!registry)
1310        return -1;
1311
1312    set_table(&G(ks)->registry, registry);
1313
1314    /* assume there will have max 1024 global variables */
1315    t = kp_tab_new_ah(ks, 0, 1024);
1316    if (!t)
1317        return -1;
1318
1319    set_table(&gtbl, t);
1320    kp_tab_setint(ks, registry, KTAP_RIDX_GLOBALS, &gtbl);
1321    G(ks)->gtab = t;
1322
1323    return 0;
1324 }
1325
1326 static int init_arguments(ktap_state_t *ks, int argc, char __user **user_argv)
1327 {
1328    ktap_tab_t *gtbl = G(ks)->gtab;
1329    ktap_tab_t *arg_tbl = kp_tab_new_ah(ks, argc, 1);
1330    ktap_val_t arg_tblval;
1331    ktap_val_t arg_tsval;
1332    ktap_str_t *argts = kp_str_newz(ks, "arg");
1333    char **argv;
1334    int i, ret;
1335
1336    if (!arg_tbl)
1337        return -1;
1338
1339    if (unlikely(!argts))
1340        return -ENOMEM;
1341
1342    set_string(&arg_tsval, argts);
1343    set_table(&arg_tblval, arg_tbl);
1344    kp_tab_set(ks, gtbl, &arg_tsval, &arg_tblval);
1345
1346    if (!argc)
1347        return 0;
1348
1349    if (argc > 1024)
1350        return -EINVAL;
1351
1352    argv = kzalloc(argc * sizeof(char *), GFP_KERNEL);
1353    if (!argv)
1354        return -ENOMEM;
1355
```

```
1356      ret = copy_from_user(argv, user_argv, argc * sizeof(char *));
1357      if (ret < 0) {
1358          kfree(argv);
1359          return -EFAULT;
1360      }
1361
1362      ret = 0;
1363      for (i = 0; i < argc; i++) {
1364          ktap_val_t val;
1365          char __user *ustr = argv[i];
1366          char *kstr;
1367          int len;
1368          int res;
1369
1370          len = strlen_user(ustr);
1371          if (len > 0x1000) {
1372              ret = -EINVAL;
1373              break;
1374          }
1375
1376          kstr = kmalloc(len + 1, GFP_KERNEL);
1377          if (!kstr) {
1378              ret = -ENOMEM;
1379              break;
1380          }
1381
1382          if (strncpy_from_user(kstr, ustr, len) < 0) {
1383              kfree(kstr);
1384              ret = -EFAULT;
1385              break;
1386          }
1387
1388          kstr[len] = '\0';
1389
1390          if (!kstrtoint(kstr, 10, &res)) {
1391              set_number(&val, res);
1392          } else {
1393              ktap_str_t *ts = kp_str_newz(ks, kstr);
1394              if (unlikely(!ts)) {
1395                  kfree(kstr);
1396                  ret = -ENOMEM;
1397                  break;
1398              }
1399
1400              set_string(&val, ts);
1401          }
1402
1403          kp_tab_setint(ks, arg_tbl, i, &val);
1404
1405          kfree(kstr);
1406      }
1407
1408      kfree(argv);
1409      return ret;
1410  }
1411
1412  static void free_preserved_data(ktap_state_t *ks)
1413  {
1414      int cpu, i, j;
1415
1416      /* free stack for each allocated ktap_state */
1417      for_each_possible_cpu(cpu) {
1418          for (j = 0; j < PERF_NR_CONTEXTS; j++) {
1419              void *percpu_state = G(ks)->percpu_state[j];
1420              ktap_state_t *pks;
1421
1422              if (!percpu_state)
1423                  break;
1424              pks = per_cpu_ptr(percpu_state, cpu);
1425              if (!ks)
1426                  break;
1427              kfree(pks->stack);
1428          }
1429      }
1430
1431      /* free percpu ktap_state */
```

```
1432      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1433          if (G(ks)->percpu_state[i])
1434              free_percpu(G(ks)->percpu_state[i]);
1435      }
1436
1437      /* free percpu ktap print buffer */
1438      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1439          if (G(ks)->percpu_print_buffer[i])
1440              free_percpu(G(ks)->percpu_print_buffer[i]);
1441      }
1442
1443      /* free percpu ktap temp buffer */
1444      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1445          if (G(ks)->percpu_temp_buffer[i])
1446              free_percpu(G(ks)->percpu_temp_buffer[i]);
1447      }
1448
1449      /* free percpu ktap recursion context flag */
1450      for (i = 0; i < PERF_NR_CONTEXTS; i++)
1451          if (G(ks)->recursion_context[i])
1452              free_percpu(G(ks)->recursion_context[i]);
1453  }
1454
1455  #define ALLOC_PERCPU(size)  __alloc_percpu(size, __alignof__(char))
1456  static int init_preserved_data(ktap_state_t *ks)
1457  {
1458      void __percpu *data;
1459      int cpu, i, j;
1460
1461      /* init percpu ktap_state */
1462      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1463          data = ALLOC_PERCPU(sizeof(ktap_state_t));
1464          if (!data)
1465              goto fail;
1466          G(ks)->percpu_state[i] = data;
1467      }
1468
1469      /* init stack for each allocated ktap_state */
1470      for_each_possible_cpu(cpu) {
1471          for (j = 0; j < PERF_NR_CONTEXTS; j++) {
1472              void *percpu_state = G(ks)->percpu_state[j];
1473              ktap_state_t *pks;
1474
1475              if (!percpu_state)
1476                  break;
1477              pks = per_cpu_ptr(percpu_state, cpu);
1478              if (!ks)
1479                  break;
1480              pks->stack = kzalloc(KTAP_STACK_SIZE_BYTES, GFP_KERNEL);
1481              if (!pks->stack)
1482                  goto fail;
1483
1484              pks->stack_last = pks->stack + KTAP_STACK_SIZE;
1485              G(pks) = G(ks);
1486          }
1487      }
1488
1489      /* init percpu ktap print buffer */
1490      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1491          data = ALLOC_PERCPU(KTAP_PERCPU_BUFFER_SIZE);
1492          if (!data)
1493              goto fail;
1494          G(ks)->percpu_print_buffer[i] = data;
1495      }
1496
1497      /* init percpu ktap temp buffer */
1498      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1499          data = ALLOC_PERCPU(KTAP_PERCPU_BUFFER_SIZE);
1500          if (!data)
1501              goto fail;
1502          G(ks)->percpu_temp_buffer[i] = data;
1503      }
1504
1505      /* init percpu ktap recursion context flag */
1506      for (i = 0; i < PERF_NR_CONTEXTS; i++) {
1507          data = alloc_percpu(int);
```

```
        if (!data)
            goto fail;
        G(ks)->recursion_context[i] = data;
    }

    return 0;

fail:
    free_preserved_data(ks);
    return -ENOMEM;
}

/*
* wait ktapio thread read all content in ring buffer.
*
* Here we use stupid approach to sync with ktapio thread,
* note that we cannot use semaphore/completion/other sync method,
* because ktapio thread could be killed by SIG_KILL in anytime, there
* have no safe way to up semaphore or wake waitqueue before thread exit.
*
* we also cannot use waitqueue of current->signal->wait_chldexit to sync
* exit, becasue mainthread and ktapio thread are in same thread group.
*
* Also ktap mainthread must wait ktapio thread exit, otherwise ktapio
* thread will oops when access ktap structure.
*/
static void wait_user_completion(ktap_state_t *ks)
{
    struct task_struct *tsk = G(ks)->task;
    G(ks)->wait_user = 1;

    while (1) {
        set_current_state(TASK_INTERRUPTIBLE);
        /* sleep for 100 msecs, and try again. */
        schedule_timeout(HZ / 10);

        if (get_nr_threads(tsk) == 1)
            break;
    }
}

static void sleep_loop(ktap_state_t *ks,
            int (*actor)(ktap_state_t *ks, void *arg), void *arg)
{
    while (!ks->stop) {
        set_current_state(TASK_INTERRUPTIBLE);
        /* sleep for 100 msecs, and try again. */
        schedule_timeout(HZ / 10);

        if (actor(ks, arg))
            return;
    }
}

static int sl_wait_task_pause_actor(ktap_state_t *ks, void *arg)
{
    struct task_struct *task = (struct task_struct *)arg;

    if (task->state)
        return 1;
    else
        return 0;
}

static int sl_wait_task_exit_actor(ktap_state_t *ks, void *arg)
{
    struct task_struct *task = (struct task_struct *)arg;

    if (signal_pending(current)) {
        flush_signals(current);

        /* newline for handle CTRL+C display as ^C */
        kp_puts(ks, "\n");
        return 1;
    }

```

```
1584      /* stop waiting if target pid is exited */
1585      if (task && task->state == TASK_DEAD)
1586              return 1;
1587
1588      return 0;
1589  }
1590
1591  /* wait user interrupt, signal killed */
1592  static void wait_user_interrupt(ktap_state_t *ks)
1593  {
1594      struct task_struct *task = G(ks)->trace_task;
1595
1596      if (G(ks)->state == KTAP_EXIT || G(ks)->state == KTAP_ERROR)
1597          return;
1598
1599      /* let tracing goes now. */
1600      ks->stop = 0;
1601
1602      if (G(ks)->parm->workload) {
1603          /* make sure workload is in pause state
1604           * so it won't miss the signal */
1605          sleep_loop(ks, sl_wait_task_pause_actor, task);
1606          /* tell workload process to start executing */
1607          send_sig(SIGINT, G(ks)->trace_task, 0);
1608      }
1609
1610      if (!G(ks)->parm->quiet)
1611          kp_printf(ks, "Tracing... Hit Ctrl-C to end.\n");
1612
1613      sleep_loop(ks, sl_wait_task_exit_actor, task);
1614  }
1615
1616  /*
1617  * ktap exit, free all resources.
1618  */
1619  void kp_vm_exit(ktap_state_t *ks)
1620  {
1621      if (!list_empty(&G(ks)->events_head) ||
1622          !list_empty(&G(ks)->timers))
1623          wait_user_interrupt(ks);
1624
1625      kp_exit_timers(ks);
1626      kp_events_exit(ks);
1627
1628      /* free all resources got by ktap */
1629  #ifdef CONFIG_KTAP_FFI
1630      ffi_free_symbols(ks);
1631  #endif
1632      kp_str_freeall(ks);
1633      kp_mempool_destroy(ks);
1634
1635      func_closeuv(ks, 0); /* close all open upvals, let below call free it */
1636      kp_obj_freeall(ks);
1637
1638      kp_vm_exit_thread(ks);
1639      kp_free(ks, ks->stack);
1640
1641      free_preserved_data(ks);
1642      free_cpumask_var(G(ks)->cpumask);
1643
1644      wait_user_completion(ks);
1645
1646      /* should invoke after wait_user_completion */
1647      if (G(ks)->trace_task)
1648          put_task_struct(G(ks)->trace_task);
1649
1650      kp_transport_exit(ks);
1651      kp_free(ks, ks); /* free self */
1652  }
1653
1654  /*
1655  * ktap mainthread initization
1656  */
1657  ktap_state_t *kp_vm_new_state(ktap_option_t *parm, struct dentry *dir)
1658  {
1659      ktap_state_t *ks;
```

```
1660      ktap_global_state_t *g;
1661      pid_t pid;
1662      int cpu;
1663
1664      ks = kzalloc(sizeof(ktap_state_t) + sizeof(ktap_global_state_t),
1665               GFP_KERNEL);
1666      if (!ks)
1667          return NULL;
1668
1669      G(ks) = (ktap_global_state_t *)(ks + 1);
1670      g = G(ks);
1671      g->mainthread = ks;
1672      g->task = current;
1673      g->parm = parm;
1674      g->str_lock = (arch_spinlock_t)__ARCH_SPIN_LOCK_UNLOCKED;
1675      g->strmask = ~(int)0;
1676      g->uvhead.prev = &g->uvhead;
1677      g->uvhead.next = &g->uvhead;
1678      g->state = KTAP_RUNNING;
1679      INIT_LIST_HEAD(&(g->timers));
1680      INIT_LIST_HEAD(&(g->events_head));
1681
1682      if (kp_transport_init(ks, dir))
1683          goto out;
1684
1685      ks->stack = kp_malloc(ks, KTAP_STACK_SIZE_BYTES);
1686      if (!ks->stack)
1687          goto out;
1688
1689      ks->stack_last = ks->stack + KTAP_STACK_SIZE;
1690      ks->top = ks->stack;
1691
1692      pid = (pid_t)parm->trace_pid;
1693      if (pid != -1) {
1694          struct task_struct *task;
1695
1696          rcu_read_lock();
1697          task = pid_task(find_vpid(pid), PIDTYPE_PID);
1698          if (!task) {
1699              kp_error(ks, "cannot find pid %d\n", pid);
1700              rcu_read_unlock();
1701              goto out;
1702          }
1703          g->trace_task = task;
1704          get_task_struct(task);
1705          rcu_read_unlock();
1706      }
1707
1708      if( !alloc_cpumask_var(&g->cpumask, GFP_KERNEL))
1709          goto out;
1710
1711      cpumask_copy(g->cpumask, cpu_online_mask);
1712
1713      cpu = parm->trace_cpu;
1714      if (cpu != -1) {
1715          if (!cpu_online(cpu)) {
1716              kp_error(ks, "ktap: cpu %d is not online\n", cpu);
1717              goto out;
1718          }
1719
1720          cpumask_clear(g->cpumask);
1721          cpumask_set_cpu(cpu, g->cpumask);
1722      }
1723
1724      if (kp_mempool_init(ks, KP_MAX_MEMPOOL_SIZE))
1725          goto out;
1726
1727      if (kp_str_resize(ks, 1024 - 1)) /* set string hashtable size */
1728          goto out;
1729
1730      if (init_registry(ks))
1731          goto out;
1732      if (init_arguments(ks, parm->argc, parm->argv))
1733          goto out;
1734
1735      /* init librarys */
```

```
1736      if (kp_lib_init_base(ks))
1737          goto out;
1738      if (kp_lib_init_kdebug(ks))
1739          goto out;
1740      if (kp_lib_init_timer(ks))
1741          goto out;
1742      if (kp_lib_init_ansi(ks))
1743          goto out;
1744  #ifdef CONFIG_KTAP_FFI
1745      if (kp_lib_init_ffi(ks))
1746          goto out;
1747  #endif
1748      if (kp_lib_init_table(ks))
1749          goto out;
1750
1751      if (kp_lib_init_net(ks))
1752          goto out;
1753
1754      if (init_preserved_data(ks))
1755          goto out;
1756
1757      if (kp_events_init(ks))
1758          goto out;
1759
1760      return ks;
1761
1762  out:
1763      g->state = KTAP_ERROR;
1764      kp_vm_exit(ks);
1765      return NULL;
1766  }
1767
```

One Level Up Top Level

userspace/kp_parse.c - ktap

Global variables defined

- priority

Data types defined

- BinOpr
- BinOpr
- ExpDesc
- ExpDesc
- ExpKind
- FuncScope
- FuncScope
- FuncState
- FuncState
- LHSVarList
- LHSVarList
- VarIndex

Functions defined

- assign_adjust
- assign_hazard
- bcemit_INS
- bcemit_arith
- bcemit_binop
- bcemit_binop_left
- bcemit_branch
- bcemit_branch_f
- bcemit_branch_t
- bcemit_comp
- bcemit_jmp
- bcemit_method
- bcemit_nil
- bcemit_store

- bcemit_store_incr
- bcemit_unop
- bcopisret
- bcreg_bump
- bcreg_free
- bcreg_reserve
- const_gc
- const_num
- const_str
- err_limit
- err_syntax
- err_token
- expr
- expr_binop
- expr_bracket
- expr_cond
- expr_discharge
- expr_field
- expr_free
- expr_index
- expr_init
- expr_kvalue
- expr_list
- expr_next
- expr_numiszero
- expr_primary
- expr_simple
- expr_str
- expr_table
- expr_toanyreg
- expr_tonextreg
- expr_toreg
- expr_toreg_nobranch

- expr_toval
- expr_unop
- foldarith
- fs_finish
- fs_fixup_bc
- fs_fixup_k
- fs_fixup_line
- fs_fixup_ret
- fs_fixup_uv1
- fs_fixup_uv2
- fs_fixup_var
- fs_init
- fs_prep_line
- fs_prep_var
- fscope_begin
- fscope_end
- fscope_uvmark
- gola_close
- gola_findlabel
- gola_fixup
- gola_new
- gola_patch
- gola_resolve
- invertcond
- jmp_append
- jmp_dropval
- jmp_next
- jmp_novalue
- jmp_patch
- jmp_patchins
- jmp_patchtestreg
- jmp_patchval
- jmp_tohere

- kp_parse
- kp_parse_keepstr
- lex_check
- lex_match
- lex_opt
- lex_str
- number_foldarith
- parse_args
- parse_assignment
- parse_block
- parse_body
- parse_body_no_args
- parse_break
- parse_call_assign
- parse_chunk
- parse_for
- parse_for_iter
- parse_for_num
- parse_func
- parse_if
- parse_isend
- parse_label
- parse_local
- parse_params
- parse_repeat
- parse_return
- parse_stmt
- parse_then
- parse_timer
- parse_trace
- parse_while
- predict_next
- synlevel_begin

- token2binop
- var_add
- var_lookup_
- var_lookup_local
- var_lookup_uv
- var_new
- var_remove

Macros defined

- FLS
- FSCOPE_BREAK
- FSCOPE_GOLA
- FSCOPE_LOOP
- FSCOPE_NOCLOSE
- FSCOPE_UPVAL
- KP_MAX_VSTACK
- NAME_BREAK
- UNARY_PRIORITY
- VARNAMEDEF
- VARNAMEENUM
- VARNAMEENUM
- VSTACK_GOTO
- VSTACK_LABEL
- VSTACK_VAR_RW
- bcemit_ABC
- bcemit_AD
- bcemit_AJ
- bcptr
- checkcond
- checklimit
- checklimitgt
- checku8
- const_pri
- expr_hasjump

- expr_isk
- expr_isk_nojump
- expr_isnumk
- expr_isnumk_nojump
- expr_isstrk
- expr_numberV
- expr_numtv
- fs_fixup_line
- fs_fixup_var
- fs_prep_line
- fs_prep_var
- gola_isgoto
- gola_isgotolabel
- gola_islabel
- hsize2hbits
- synlevel_end
- tvhaskslot
- tvkslot
- var_get
- var_lookup
- var_new_fixed
- var_new_lit

Source code

```
/*
* ktap parser (source code -> bytecode).
*
* This file is part of ktap by Jovi Zhangwei.
*
* Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
*
* Adapted from luajit and lua interpreter.
* Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
```

```
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "../include/ktap_types.h"
#include "../include/ktap_err.h"
#include "kp_util.h"
#include "kp_lex.h"

/* Fixed internal variable names. */
#define VARNAMEDEF(_) \
    _(FOR_IDX, "(for index)") \
    _(FOR_STOP, "(for limit)") \
    _(FOR_STEP, "(for step)") \
    _(FOR_GEN, "(for generator)") \
    _(FOR_STATE, "(for state)") \
    _(FOR_CTL, "(for control)")

enum {
    VARNAME_END,
#define VARNAMEENUM(name, str)  VARNAME_##name,
    VARNAMEDEF(VARNAMEENUM)
#undef VARNAMEENUM
    VARNAME__MAX
};

/* -- Parser structures and definitions ---------------------------------- */

/* Expression kinds. */
typedef enum {
    /* Constant expressions must be first and in this order: */
    VKNIL,
    VKFALSE,
    VKTRUE,
    VKSTR,    /* sval = string value */
    VKNUM,    /* nval = number value */
    VKLAST = VKNUM,
    VKCDATA, /* nval = cdata value, not treated as a constant expression */
    /* Non-constant expressions follow: */
    VLOCAL,    /* info = local register, aux = vstack index */
    VUPVAL,    /* info = upvalue index, aux = vstack index */
    VGLOBAL,/* sval = string value */
    VINDEXED,/* info = table register, aux = index reg/byte/string const */
    VJMP,    /* info = instruction PC */
    VRELOCABLE, /* info = instruction PC */
    VNONRELOC, /* info = result register */
    VCALL,    /* info = instruction PC, aux = base */
    VVOID,

    VARGN,
    VARGSTR,
    VARGNAME,
    VPID,
    VTID,
    VUID,
    VCPU,
    VEXECNAME,
    VMAX
} ExpKind;

/* Expression descriptor. */
typedef struct ExpDesc {
    union {
        struct {
            uint32_t info;    /* Primary info. */
            uint32_t aux;    /* Secondary info. */
        } s;
        ktap_val_t nval;    /* Number value. */
        ktap_str_t *sval;    /* String value. */
    } u;
    ExpKind k;
    BCPos t;    /* True condition jump list. */
    BCPos f;    /* False condition jump list. */
} ExpDesc;

/* Macros for expressions. */
#define expr_hasjump(e)        ((e)->t != (e)->f)
```

```
99
100 #define expr_isk(e)        ((e)->k <= VKLAST)
101 #define expr_isk_nojump(e)    (expr_isk(e) && !expr_hasjump(e))
102 #define expr_isnumk(e)        ((e)->k == VKNUM)
103 #define expr_isnumk_nojump(e)    (expr_isnumk(e) && !expr_hasjump(e))
104 #define expr_isstrk(e)        ((e)->k == VKSTR)
105
106 #define expr_numtv(e)        (&(e)->u.nval)
107 #define expr_numberV(e)        nvalue(expr_numtv((e)))
108
109 /* Initialize expression. */
110 static inline void expr_init(ExpDesc *e, ExpKind k, uint32_t info)
111 {
112     e->k = k;
113     e->u.s.info = info;
114     e->f = e->t = NO_JMP;
115 }
116
117 /* Check number constant for +-0. */
118 static int expr_numiszero(ExpDesc *e)
119 {
120     ktap_val_t *o = expr_numtv(e);
121     return (nvalue(o) == 0);
122 }
123
124 /* Per-function linked list of scope blocks. */
125 typedef struct FuncScope {
126     struct FuncScope *prev;    /* Link to outer scope. */
127     int vstart;        /* Start of block-local variables. */
128     uint8_t nactvar;    /* Number of active vars outside the scope. */
129     uint8_t flags;        /* Scope flags. */
130 } FuncScope;
131
132 #define FSCOPE_LOOP        0x01    /* Scope is a (breakable) loop. */
133 #define FSCOPE_BREAK        0x02    /* Break used in scope. */
134 #define FSCOPE_GOLA        0x04    /* Goto or label used in scope. */
135 #define FSCOPE_UPVAL        0x08    /* Upvalue in scope. */
136 #define FSCOPE_NOCLOSE        0x10    /* Do not close upvalues. */
137
138 #define NAME_BREAK        ((ktap_str_t *)(uintptr_t)1)
139
140 /* Index into variable stack. */
141 typedef uint16_t VarIndex;
142 #define KP_MAX_VSTACK        (65536 - KP_MAX_UPVAL)
143
144 /* Variable/goto/label info. */
145 #define VSTACK_VAR_RW        0x01    /* R/W variable. */
146 #define VSTACK_GOTO        0x02    /* Pending goto. */
147 #define VSTACK_LABEL        0x04    /* Label. */
148
149 /* Per-function state. */
150 typedef struct FuncState {
151     ktap_tab_t *kt;        /* Hash table for constants. */
152     LexState *ls;        /* Lexer state. */
153     FuncScope *bl;        /* Current scope. */
154     struct FuncState *prev;    /* Enclosing function. */
155     BCPos pc;        /* Next bytecode position. */
156     BCPos lasttarget;    /* Bytecode position of last jump target. */
157     BCPos jpc;        /* Pending jump list to next bytecode. */
158     BCReg freereg;        /* First free register. */
159     BCReg nactvar;        /* Number of active local variables. */
160     BCReg nkn, nkgc;        /* Number of ktap_number/ktap_obj_t constants*/
161     BCLine linedefined;    /* First line of the function definition. */
162     BCInsLine *bcbase;    /* Base of bytecode stack. */
163     BCPos bclim;        /* Limit of bytecode stack. */
164     int vbase;        /* Base of variable stack for this function. */
165     uint8_t flags;        /* Prototype flags. */
166     uint8_t numparams;    /* Number of parameters. */
167     uint8_t framesize;    /* Fixed frame size. */
168     uint8_t nuv;        /* Number of upvalues */
169     VarIndex varmap[KP_MAX_LOCVAR];/* Map from register to variable idx. */
170     VarIndex uvmap[KP_MAX_UPVAL];    /* Map from upvalue to variable idx. */
171     VarIndex uvtmp[KP_MAX_UPVAL];    /* Temporary upvalue map. */
172 } FuncState;
173
174 /* Binary and unary operators. ORDER OPR */
```

```
typedef enum BinOpr {
    OPR_ADD, OPR_SUB, OPR_MUL, OPR_DIV, OPR_MOD, OPR_POW, /* ORDER ARITH */
    OPR_CONCAT,
    OPR_NE, OPR_EQ,
    OPR_LT, OPR_GE, OPR_LE, OPR_GT,
    OPR_AND, OPR_OR,
    OPR_NOBINOPR
} BinOpr;

KP_STATIC_ASSERT((int)BC_ISGE-(int)BC_ISLT == (int)OPR_GE-(int)OPR_LT);
KP_STATIC_ASSERT((int)BC_ISLE-(int)BC_ISLT == (int)OPR_LE-(int)OPR_LT);
KP_STATIC_ASSERT((int)BC_ISGT-(int)BC_ISLT == (int)OPR_GT-(int)OPR_LT);
KP_STATIC_ASSERT((int)BC_SUBVV-(int)BC_ADDVV == (int)OPR_SUB-(int)OPR_ADD);
KP_STATIC_ASSERT((int)BC_MULVV-(int)BC_ADDVV == (int)OPR_MUL-(int)OPR_ADD);
KP_STATIC_ASSERT((int)BC_DIVVV-(int)BC_ADDVV == (int)OPR_DIV-(int)OPR_ADD);
KP_STATIC_ASSERT((int)BC_MODVV-(int)BC_ADDVV == (int)OPR_MOD-(int)OPR_ADD);

/* -- Error handling ---------------------------------------------------- */

static void err_syntax(LexState *ls, ErrMsg em)
{
    kp_lex_error(ls, ls->tok, em);
}

static void err_token(LexState *ls, LexToken tok)
{
    kp_lex_error(ls, ls->tok, KP_ERR_XTOKEN, kp_lex_token2str(ls, tok));
}

static void err_limit(FuncState *fs, uint32_t limit, const char *what)
{
    if (fs->linedefined == 0)
        kp_lex_error(fs->ls, 0, KP_ERR_XLIMM, limit, what);
    else
        kp_lex_error(fs->ls, 0, KP_ERR_XLIMF, fs->linedefined,
                limit, what);
}

#define checklimit(fs, v, l, m)        if ((v) >= (l)) err_limit(fs, l, m)
#define checklimitgt(fs, v, l, m)    if ((v) > (l)) err_limit(fs, l, m)
#define checkcond(ls, c, em)        { if (!(c)) err_syntax(ls, em); }

/* -- Management of constants ------------------------------------------- */

/* Return bytecode encoding for primitive constant. */
#define const_pri(e)    ((e)->k)

#define tvhaskslot(o)    (is_number(o))
#define tvkslot(o)    (nvalue(o))

/* Add a number constant. */
static BCReg const_num(FuncState *fs, ExpDesc *e)
{
    ktap_val_t *o;

    kp_assert(expr_isnumk(e));
    o = kp_tab_set(fs->kt, &e->u.nval);
    if (tvhaskslot(o))
        return tvkslot(o);
    set_number(o, fs->nkn);
    return fs->nkn++;
}

/* Add a GC object constant. */
static BCReg const_gc(FuncState *fs, ktap_obj_t *gc, uint32_t itype)
{
    ktap_val_t key, *o;

    setitype(&key, itype);
    key.val.gc = gc;
    o = kp_tab_set(fs->kt, &key);
    if (tvhaskslot(o))
        return tvkslot(o);
    set_number(o, fs->nkgc);
    return fs->nkgc++;
}
```

```
251
252 /* Add a string constant. */
253 static BCReg const_str(FuncState *fs, ExpDesc *e)
254 {
255     kp_assert(expr_isstrk(e) || e->k == VGLOBAL);
256     return const_gc(fs, obj2gco(e->u.sval), KTAP_TSTR);
257 }
258
259 /* Anchor string constant. */
260 ktap_str_t *kp_parse_keepstr(LexState *ls, const char *str, size_t len)
261 {
262     ktap_val_t v;
263     ktap_str_t *s = kp_str_new(str, len);
264
265     set_string(&v, s);
266     ktap_val_t *tv = kp_tab_set(ls->fs->kt, &v);
267     if (is_nil(tv))
268         set_bool(tv, 1);
269     return s;
270 }
271
272 /* -- Jump list handling ---------------------------------------------- */
273
274 /* Get next element in jump list. */
275 static BCPos jmp_next(FuncState *fs, BCPos pc)
276 {
277     ptrdiff_t delta = bc_j(fs->bcbase[pc].ins);
278     if ((BCPos)delta == NO_JMP)
279         return NO_JMP;
280     else
281         return (BCPos)(((ptrdiff_t)pc+1)+delta);
282 }
283
284 /* Check if any of the instructions on the jump list produce no value. */
285 static int jmp_novalue(FuncState *fs, BCPos list)
286 {
287     for (; list != NO_JMP; list = jmp_next(fs, list)) {
288         BCIns p = fs->bcbase[list >= 1 ? list-1 : list].ins;
289         if (!(bc_op(p) == BC_ISTC || bc_op(p) == BC_ISFC ||
290             bc_a(p) == NO_REG))
291         return 1;
292     }
293     return 0;
294 }
295
296 /* Patch register of test instructions. */
297 static int jmp_patchtestreg(FuncState *fs, BCPos pc, BCReg reg)
298 {
299     BCInsLine *ilp = &fs->bcbase[pc >= 1 ? pc-1 : pc];
300     BCOp op = bc_op(ilp->ins);
301
302     if (op == BC_ISTC || op == BC_ISFC) {
303         if (reg != NO_REG && reg != bc_d(ilp->ins)) {
304             setbc_a(&ilp->ins, reg);
305         } else {/* Nothing to store or already in the right register */
306             setbc_op(&ilp->ins, op+(BC_IST-BC_ISTC));
307             setbc_a(&ilp->ins, 0);
308         }
309     } else if (bc_a(ilp->ins) == NO_REG) {
310         if (reg == NO_REG) {
311             ilp->ins =
312                 BCINS_AJ(BC_JMP, bc_a(fs->bcbase[pc].ins), 0);
313         } else {
314             setbc_a(&ilp->ins, reg);
315             if (reg >= bc_a(ilp[1].ins))
316                 setbc_a(&ilp[1].ins, reg+1);
317         }
318     } else {
319         return 0;  /* Cannot patch other instructions. */
320     }
321     return 1;
322 }
323
324 /* Drop values for all instructions on jump list. */
325 static void jmp_dropval(FuncState *fs, BCPos list)
326 {
```

```
    for (; list != NO_JMP; list = jmp_next(fs, list))
        jmp_patchtestreg(fs, list, NO_REG);
}

/* Patch jump instruction to target. */
static void jmp_patchins(FuncState *fs, BCPos pc, BCPos dest)
{
    BCIns *jmp = &fs->bcbase[pc].ins;
    BCPos offset = dest-(pc+1)+BCBIAS_J;

    kp_assert(dest != NO_JMP);
    if (offset > BCMAX_D)
        err_syntax(fs->ls, KP_ERR_XJUMP);
    setbc_d(jmp, offset);
}

/* Append to jump list. */
static void jmp_append(FuncState *fs, BCPos *l1, BCPos l2)
{
    if (l2 == NO_JMP) {
        return;
    } else if (*l1 == NO_JMP) {
        *l1 = l2;
    } else {
        BCPos list = *l1;
        BCPos next;
        /* Find last element. */
        while ((next = jmp_next(fs, list)) != NO_JMP)
            list = next;
        jmp_patchins(fs, list, l2);
    }
}

/* Patch jump list and preserve produced values. */
static void jmp_patchval(FuncState *fs, BCPos list, BCPos vtarget,
             BCReg reg, BCPos dtarget)
{
    while (list != NO_JMP) {
        BCPos next = jmp_next(fs, list);
        if (jmp_patchtestreg(fs, list, reg)) {
            /* Jump to target with value. */
            jmp_patchins(fs, list, vtarget);
        } else {
            /* Jump to default target. */
            jmp_patchins(fs, list, dtarget);
        }
        list = next;
    }
}

/* Jump to following instruction. Append to list of pending jumps. */
static void jmp_tohere(FuncState *fs, BCPos list)
{
    fs->lasttarget = fs->pc;
    jmp_append(fs, &fs->jpc, list);
}

/* Patch jump list to target. */
static void jmp_patch(FuncState *fs, BCPos list, BCPos target)
{
    if (target == fs->pc) {
        jmp_tohere(fs, list);
    } else {
        kp_assert(target < fs->pc);
        jmp_patchval(fs, list, target, NO_REG, target);
    }
}

/* -- Bytecode register allocator ---------------------------------------- */

/* Bump frame size. */
static void bcreg_bump(FuncState *fs, BCReg n)
{
    BCReg sz = fs->freereg + n;

    if (sz > fs->framesize) {
```

```
        if (sz >= KP_MAX_SLOTS)
            err_syntax(fs->ls, KP_ERR_XSLOTS);
        fs->framesize = (uint8_t)sz;
    }
}

/* Reserve registers. */
static void bcreg_reserve(FuncState *fs, BCReg n)
{
    bcreg_bump(fs, n);
    fs->freereg += n;
}

/* Free register. */
static void bcreg_free(FuncState *fs, BCReg reg)
{
    if (reg >= fs->nactvar) {
        fs->freereg--;
        kp_assert(reg == fs->freereg);
    }
}

/* Free register for expression. */
static void expr_free(FuncState *fs, ExpDesc *e)
{
    if (e->k == VNONRELOC)
        bcreg_free(fs, e->u.s.info);
}

/* -- Bytecode emitter ----------------------------------------------- */

/* Emit bytecode instruction. */
static BCPos bcemit_INS(FuncState *fs, BCIns ins)
{
    BCPos pc = fs->pc;
    LexState *ls = fs->ls;

    jmp_patchval(fs, fs->jpc, pc, NO_REG, pc);
    fs->jpc = NO_JMP;
    if (pc >= fs->bclim) {
        ptrdiff_t base = fs->bcbase - ls->bcstack;
        checklimit(fs, ls->sizebcstack, KP_MAX_BCINS,
                "bytecode instructions");
        if (!ls->bcstack) {
            ls->bcstack = malloc(sizeof(BCInsLine) * 20);
            ls->sizebcstack = 20;
        } else {
            ls->bcstack = realloc(ls->bcstack,
                ls->sizebcstack * sizeof(BCInsLine) * 2);
            ls->sizebcstack = ls->sizebcstack * 2;
        }
        fs->bclim = (BCPos)(ls->sizebcstack - base);
        fs->bcbase = ls->bcstack + base;
    }
    fs->bcbase[pc].ins = ins;
    fs->bcbase[pc].line = ls->lastline;
    fs->pc = pc+1;
    return pc;
}

#define bcemit_ABC(fs, o, a, b, c)    bcemit_INS(fs, BCINS_ABC(o, a, b, c))
#define bcemit_AD(fs, o, a, d)        bcemit_INS(fs, BCINS_AD(o, a, d))
#define bcemit_AJ(fs, o, a, j)        bcemit_INS(fs, BCINS_AJ(o, a, j))

#define bcptr(fs, e)            (&(fs)->bcbase[(e)->u.s.info].ins)

/* -- Bytecode emitter for expressions ------------------------------- */

/* Discharge non-constant expression to any register. */
static void expr_discharge(FuncState *fs, ExpDesc *e)
{
    BCIns ins;

    if (e->k == VUPVAL) {
        ins = BCINS_AD(BC_UGET, 0, e->u.s.info);
    } else if (e->k == VGLOBAL) {
```

```
        ins = BCINS_AD(BC_GGET, 0, const_str(fs, e));
    } else if (e->k == VINDEXED) {
        BCReg rc = e->u.s.aux;
        if ((int32_t)rc < 0) {
            ins = BCINS_ABC(BC_TGETS, 0, e->u.s.info, ~rc);
        } else if (rc > BCMAX_C) {
            ins = BCINS_ABC(BC_TGETB, 0, e->u.s.info,
                    rc-(BCMAX_C+1));
        } else {
            bcreg_free(fs, rc);
            ins = BCINS_ABC(BC_TGETV, 0, e->u.s.info, rc);
        }
        bcreg_free(fs, e->u.s.info);
    } else if (e->k == VCALL) {
        e->u.s.info = e->u.s.aux;
        e->k = VNONRELOC;
        return;
    } else if (e->k == VLOCAL) {
        e->k = VNONRELOC;
        return;
    } else {
        return;
    }

    e->u.s.info = bcemit_INS(fs, ins);
    e->k = VRELOCABLE;
}

/* Emit bytecode to set a range of registers to nil. */
static void bcemit_nil(FuncState *fs, BCReg from, BCReg n)
{
    if (fs->pc > fs->lasttarget) {  /* No jumps to current position? */
        BCIns *ip = &fs->bcbase[fs->pc-1].ins;
        BCReg pto, pfrom = bc_a(*ip);
        /* Try to merge with the previous instruction. */
        switch (bc_op(*ip)) {
        case BC_KPRI:
            if (bc_d(*ip) != ~KTAP_TNIL) break;
            if (from == pfrom) {
                if (n == 1)
                    return;
            } else if (from == pfrom+1) {
                from = pfrom;
                n++;
            } else {
                break;
            }
            /* Replace KPRI. */
            *ip = BCINS_AD(BC_KNIL, from, from+n-1);
            return;
        case BC_KNIL:
            pto = bc_d(*ip);
            /* Can we connect both ranges? */
            if (pfrom <= from && from <= pto+1) {
                if (from+n-1 > pto) {
                    /* Patch previous instruction range. */
                    setbc_d(ip, from+n-1);
                }
                return;
            }
            break;
        default:
            break;
        }
    }

    /* Emit new instruction or replace old instruction. */
    bcemit_INS(fs, n == 1 ? BCINS_AD(BC_KPRI, from, VKNIL) :
                BCINS_AD(BC_KNIL, from, from+n-1));
}

/* Discharge an expression to a specific register. Ignore branches. */
static void expr_toreg_nobranch(FuncState *fs, ExpDesc *e, BCReg reg)
{
    BCIns ins;

```

```
    expr_discharge(fs, e);
    if (e->k == VKSTR) {
        ins = BCINS_AD(BC_KSTR, reg, const_str(fs, e));
    } else if (e->k == VKNUM) {
        ktap_number n = expr_numberV(e);
        if (n >= 0 && n <= 0xffff) {
            ins = BCINS_AD(BC_KSHORT, reg, (BCReg)(uint16_t)n);
        } else
            ins = BCINS_AD(BC_KNUM, reg, const_num(fs, e));
    } else if (e->k == VRELOCABLE) {
        setbc_a(bcptr(fs, e), reg);
        goto noins;
    } else if (e->k == VNONRELOC) {
        if (reg == e->u.s.info)
            goto noins;
        ins = BCINS_AD(BC_MOV, reg, e->u.s.info);
    } else if (e->k == VKNIL) {
        bcemit_nil(fs, reg, 1);
        goto noins;
    } else if (e->k <= VKTRUE) {
        ins = BCINS_AD(BC_KPRI, reg, const_pri(e));
    } else if (e->k == VARGN) {
        ins = BCINS_AD(BC_VARGN, reg, e->u.s.info);
    } else if (e->k > VARGN && e->k < VMAX) {
        ins = BCINS_AD(e->k - VARGN + BC_VARGN, reg, 0);
    } else {
        kp_assert(e->k == VVOID || e->k == VJMP);
        return;
    }
    bcemit_INS(fs, ins);
noins:
    e->u.s.info = reg;
    e->k = VNONRELOC;
}

/* Forward declaration. */
static BCPos bcemit_jmp(FuncState *fs);

/* Discharge an expression to a specific register. */
static void expr_toreg(FuncState *fs, ExpDesc *e, BCReg reg)
{
    expr_toreg_nobranch(fs, e, reg);
    if (e->k == VJMP) {
        /* Add it to the true jump list. */
        jmp_append(fs, &e->t, e->u.s.info);
    }
    if (expr_hasjump(e)) {  /* Discharge expression with branches. */
        BCPos jend, jfalse = NO_JMP, jtrue = NO_JMP;
        if (jmp_novalue(fs, e->t) || jmp_novalue(fs, e->f)) {
            BCPos jval = (e->k == VJMP) ? NO_JMP : bcemit_jmp(fs);
            jfalse = bcemit_AD(fs, BC_KPRI, reg, VKFALSE);
            bcemit_AJ(fs, BC_JMP, fs->freereg, 1);
            jtrue = bcemit_AD(fs, BC_KPRI, reg, VKTRUE);
            jmp_tohere(fs, jval);
        }
        jend = fs->pc;
        fs->lasttarget = jend;
        jmp_patchval(fs, e->f, jend, reg, jfalse);
        jmp_patchval(fs, e->t, jend, reg, jtrue);
    }
    e->f = e->t = NO_JMP;
    e->u.s.info = reg;
    e->k = VNONRELOC;
}

/* Discharge an expression to the next free register. */
static void expr_tonextreg(FuncState *fs, ExpDesc *e)
{
    expr_discharge(fs, e);
    expr_free(fs, e);
    bcreg_reserve(fs, 1);
    expr_toreg(fs, e, fs->freereg - 1);
}

/* Discharge an expression to any register. */
static BCReg expr_toanyreg(FuncState *fs, ExpDesc *e)
```

```
{
    expr_discharge(fs, e);
    if (e->k == VNONRELOC) {
        if (!expr_hasjump(e))
            return e->u.s.info;  /* Already in a register. */
        if (e->u.s.info >= fs->nactvar) {
            /* Discharge to temp. register. */
            expr_toreg(fs, e, e->u.s.info);
            return e->u.s.info;
        }
    }
    expr_tonextreg(fs, e);  /* Discharge to next register. */
    return e->u.s.info;
}

/* Partially discharge expression to a value. */
static void expr_toval(FuncState *fs, ExpDesc *e)
{
    if (expr_hasjump(e))
        expr_toanyreg(fs, e);
    else
        expr_discharge(fs, e);
}

/* Emit store for LHS expression. */
static void bcemit_store(FuncState *fs, ExpDesc *var, ExpDesc *e)
{
    BCIns ins;

    if (var->k == VLOCAL) {
        fs->ls->vstack[var->u.s.aux].info |= VSTACK_VAR_RW;
        expr_free(fs, e);
        expr_toreg(fs, e, var->u.s.info);
        return;
    } else if (var->k == VUPVAL) {
        fs->ls->vstack[var->u.s.aux].info |= VSTACK_VAR_RW;
        expr_toval(fs, e);
        if (e->k <= VKTRUE)
            ins = BCINS_AD(BC_USETP, var->u.s.info, const_pri(e));
        else if (e->k == VKSTR)
            ins = BCINS_AD(BC_USETS, var->u.s.info,
                    const_str(fs, e));
        else if (e->k == VKNUM)
            ins = BCINS_AD(BC_USETN, var->u.s.info,
                    const_num(fs, e));
        else
            ins = BCINS_AD(BC_USETV, var->u.s.info,
                    expr_toanyreg(fs, e));
    } else if (var->k == VGLOBAL) {
        BCReg ra = expr_toanyreg(fs, e);
        ins = BCINS_AD(BC_GSET, ra, const_str(fs, var));
    } else {
        BCReg ra, rc;
        kp_assert(var->k == VINDEXED);
        ra = expr_toanyreg(fs, e);
        rc = var->u.s.aux;
        if ((int32_t)rc < 0) {
            ins = BCINS_ABC(BC_TSETS, ra, var->u.s.info, ~rc);
        } else if (rc > BCMAX_C) {
            ins = BCINS_ABC(BC_TSETB, ra, var->u.s.info,
                rc-(BCMAX_C+1));
        } else {
            /*
             * Free late alloced key reg to avoid assert on
             * free of value reg. This can only happen when
             * called from expr_table().
             */
            kp_assert(e->k != VNONRELOC || ra < fs->nactvar ||
                    rc < ra || (bcreg_free(fs, rc),1));
            ins = BCINS_ABC(BC_TSETV, ra, var->u.s.info, rc);
        }
    }
    bcemit_INS(fs, ins);
    expr_free(fs, e);
}

```

```c
/* Emit store for '+=' expression. */
static void bcemit_store_incr(FuncState *fs, ExpDesc *var, ExpDesc *e)
{
    BCIns ins;

    if (var->k == VLOCAL) {
        /* don't need to do like "var a=0; a+=1", just use 'a=a+1' */
        err_syntax(fs->ls, KP_ERR_XSYMBOL);
        return;
    } else if (var->k == VUPVAL) {
        fs->ls->vstack[var->u.s.aux].info |= VSTACK_VAR_RW;
        expr_toval(fs, e);
        if (e->k == VKNUM) {
            ins = BCINS_AD(BC_UINCN, var->u.s.info,
                    const_num(fs, e));
        } else if (e->k <= VKTRUE || e->k == VKSTR) {
            err_syntax(fs->ls, KP_ERR_XSYMBOL);
            return;
        } else
            ins = BCINS_AD(BC_UINCV, var->u.s.info,
                    expr_toanyreg(fs, e));
    } else if (var->k == VGLOBAL) {
        BCReg ra = expr_toanyreg(fs, e);
        ins = BCINS_AD(BC_GINC, ra, const_str(fs, var));
    } else {
        BCReg ra, rc;
        kp_assert(var->k == VINDEXED);
        ra = expr_toanyreg(fs, e);
        rc = var->u.s.aux;
        if ((int32_t)rc < 0) {
            ins = BCINS_ABC(BC_TINCS, ra, var->u.s.info, ~rc);
        } else if (rc > BCMAX_C) {
            ins = BCINS_ABC(BC_TINCB, ra, var->u.s.info,
                rc-(BCMAX_C+1));
        } else {
            /*
             * Free late alloced key reg to avoid assert on
             * free of value reg. This can only happen when
             * called from expr_table().
             */
            kp_assert(e->k != VNONRELOC || ra < fs->nactvar ||
                    rc < ra || (bcreg_free(fs, rc),1));
            ins = BCINS_ABC(BC_TINCV, ra, var->u.s.info, rc);
        }
    }
    bcemit_INS(fs, ins);
    expr_free(fs, e);
}


/* Emit method lookup expression. */
static void bcemit_method(FuncState *fs, ExpDesc *e, ExpDesc *key)
{
    BCReg idx, func, obj = expr_toanyreg(fs, e);

    expr_free(fs, e);
    func = fs->freereg;
    bcemit_AD(fs, BC_MOV, func+1, obj);/* Copy object to first argument. */
    kp_assert(expr_isstrk(key));
    idx = const_str(fs, key);
    if (idx <= BCMAX_C) {
        bcreg_reserve(fs, 2);
        bcemit_ABC(fs, BC_TGETS, func, obj, idx);
    } else {
        bcreg_reserve(fs, 3);
        bcemit_AD(fs, BC_KSTR, func+2, idx);
        bcemit_ABC(fs, BC_TGETV, func, obj, func+2);
        fs->freereg--;
    }
    e->u.s.info = func;
    e->k = VNONRELOC;
}

/* -- Bytecode emitter for branches ---------------------------------------- */

/* Emit unconditional branch. */
```

```
783 static BCPos bcemit_jmp(FuncState *fs)
784 {
785     BCPos jpc = fs->jpc;
786     BCPos j = fs->pc - 1;
787     BCIns *ip = &fs->bcbase[j].ins;
788 
789     fs->jpc = NO_JMP;
790     if ((int32_t)j >= (int32_t)fs->lasttarget && bc_op(*ip) == BC_UCLO)
791         setbc_j(ip, NO_JMP);
792     else
793         j = bcemit_AJ(fs, BC_JMP, fs->freereg, NO_JMP);
794     jmp_append(fs, &j, jpc);
795     return j;
796 }
797 
798 /* Invert branch condition of bytecode instruction. */
799 static void invertcond(FuncState *fs, ExpDesc *e)
800 {
801     BCIns *ip = &fs->bcbase[e->u.s.info - 1].ins;
802     setbc_op(ip, bc_op(*ip)^1);
803 }
804 
805 /* Emit conditional branch. */
806 static BCPos bcemit_branch(FuncState *fs, ExpDesc *e, int cond)
807 {
808     BCPos pc;
809 
810     if (e->k == VRELOCABLE) {
811         BCIns *ip = bcptr(fs, e);
812         if (bc_op(*ip) == BC_NOT) {
813             *ip = BCINS_AD(cond ? BC_ISF : BC_IST, 0, bc_d(*ip));
814             return bcemit_jmp(fs);
815         }
816     }
817     if (e->k != VNONRELOC) {
818         bcreg_reserve(fs, 1);
819         expr_toreg_nobranch(fs, e, fs->freereg-1);
820     }
821     bcemit_AD(fs, cond ? BC_ISTC : BC_ISFC, NO_REG, e->u.s.info);
822     pc = bcemit_jmp(fs);
823     expr_free(fs, e);
824     return pc;
825 }
826 
827 /* Emit branch on true condition. */
828 static void bcemit_branch_t(FuncState *fs, ExpDesc *e)
829 {
830     BCPos pc;
831 
832     expr_discharge(fs, e);
833     if (e->k == VKSTR || e->k == VKNUM || e->k == VKTRUE)
834         pc = NO_JMP;  /* Never jump. */
835     else if (e->k == VJMP)
836         invertcond(fs, e), pc = e->u.s.info;
837     else if (e->k == VKFALSE || e->k == VKNIL)
838         expr_toreg_nobranch(fs, e, NO_REG), pc = bcemit_jmp(fs);
839     else
840         pc = bcemit_branch(fs, e, 0);
841     jmp_append(fs, &e->f, pc);
842     jmp_tohere(fs, e->t);
843     e->t = NO_JMP;
844 }
845 
846 /* Emit branch on false condition. */
847 static void bcemit_branch_f(FuncState *fs, ExpDesc *e)
848 {
849     BCPos pc;
850 
851     expr_discharge(fs, e);
852     if (e->k == VKNIL || e->k == VKFALSE)
853         pc = NO_JMP;  /* Never jump. */
854     else if (e->k == VJMP)
855         pc = e->u.s.info;
856     else if (e->k == VKSTR || e->k == VKNUM || e->k == VKTRUE)
857         expr_toreg_nobranch(fs, e, NO_REG), pc = bcemit_jmp(fs);
858     else
```

```
        pc = bcemit_branch(fs, e, 1);
    jmp_append(fs, &e->t, pc);
    jmp_tohere(fs, e->f);
    e->f = NO_JMP;
}

/* -- Bytecode emitter for operators ---------------------------------- */

static ktap_number number_foldarith(ktap_number x, ktap_number y, int op)
{
    switch (op) {
    case OPR_ADD - OPR_ADD: return x + y;
    case OPR_SUB - OPR_ADD: return x - y;
    case OPR_MUL - OPR_ADD: return x * y;
    case OPR_DIV - OPR_ADD: return x / y;
    default: return x;
    }
}

/* Try constant-folding of arithmetic operators. */
static int foldarith(BinOpr opr, ExpDesc *e1, ExpDesc *e2)
{
    ktap_val_t o;
    ktap_number n;

    if (!expr_isnumk_nojump(e1) || !expr_isnumk_nojump(e2))
        return 0;

    if (opr == OPR_DIV && expr_numberV(e2) == 0)
        return 0; /* do not attempt to divide by 0 */

    if (opr == OPR_MOD)
        return 0; /* ktap current do not suppor pow arith */

    n = number_foldarith(expr_numberV(e1), expr_numberV(e2),
                (int)opr-OPR_ADD);
    set_number(&o, n);
    set_number(&e1->u.nval, n);
    return 1;
}

/* Emit arithmetic operator. */
static void bcemit_arith(FuncState *fs, BinOpr opr, ExpDesc *e1, ExpDesc *e2)
{
    BCReg rb, rc, t;
    uint32_t op;

    if (foldarith(opr, e1, e2))
        return;
    if (opr == OPR_POW) {
        op = BC_POW;
        rc = expr_toanyreg(fs, e2);
        rb = expr_toanyreg(fs, e1);
    } else {
        op = opr-OPR_ADD+BC_ADDVV;
        /*
         * Must discharge 2nd operand first since VINDEXED
         * might free regs.
         */
        expr_toval(fs, e2);
        if (expr_isnumk(e2) && (rc = const_num(fs, e2)) <= BCMAX_C)
            op -= BC_ADDVV-BC_ADDVN;
        else
            rc = expr_toanyreg(fs, e2);
        /* 1st operand discharged by bcemit_binop_left,
         * but need KNUM/KSHORT. */
        kp_assert(expr_isnumk(e1) || e1->k == VNONRELOC);
        expr_toval(fs, e1);
        /* Avoid two consts to satisfy bytecode constraints. */
        if (expr_isnumk(e1) && !expr_isnumk(e2) &&
            (t = const_num(fs, e1)) <= BCMAX_B) {
            rb = rc; rc = t; op -= BC_ADDVV-BC_ADDNV;
        } else {
            rb = expr_toanyreg(fs, e1);
        }
    }
```

```
    /* Using expr_free might cause asserts if the order is wrong. */
    if (e1->k == VNONRELOC && e1->u.s.info >= fs->nactvar)
        fs->freereg--;
    if (e2->k == VNONRELOC && e2->u.s.info >= fs->nactvar)
        fs->freereg--;
    e1->u.s.info = bcemit_ABC(fs, op, 0, rb, rc);
    e1->k = VRELOCABLE;
}

/* Emit comparison operator. */
static void bcemit_comp(FuncState *fs, BinOpr opr, ExpDesc *e1, ExpDesc *e2)
{
    ExpDesc *eret = e1;
    BCIns ins;

    expr_toval(fs, e1);
    if (opr == OPR_EQ || opr == OPR_NE) {
        BCOp op = opr == OPR_EQ ? BC_ISEQV : BC_ISNEV;
        BCReg ra;

        if (expr_isk(e1)) { /* Need constant in 2nd arg. */
            e1 = e2;
            e2 = eret;
        }
        ra = expr_toanyreg(fs, e1);  /* First arg must be in a reg. */
        expr_toval(fs, e2);
        switch (e2->k) {
        case VKNIL: case VKFALSE: case VKTRUE:
            ins = BCINS_AD(op+(BC_ISEQP-BC_ISEQV), ra,
                    const_pri(e2));
            break;
        case VKSTR:
            ins = BCINS_AD(op+(BC_ISEQS-BC_ISEQV), ra,
                    const_str(fs, e2));
            break;
        case VKNUM:
            ins = BCINS_AD(op+(BC_ISEQN-BC_ISEQV), ra,
                    const_num(fs, e2));
            break;
        default:
            ins = BCINS_AD(op, ra, expr_toanyreg(fs, e2));
            break;
        }
    } else {
        uint32_t op = opr-OPR_LT+BC_ISLT;
        BCReg ra, rd;
        if ((op-BC_ISLT) & 1) {  /* GT -> LT, GE -> LE */
            e1 = e2; e2 = eret;  /* Swap operands. */
            op = ((op-BC_ISLT)^3)+BC_ISLT;
            expr_toval(fs, e1);
        }
        rd = expr_toanyreg(fs, e2);
        ra = expr_toanyreg(fs, e1);
        ins = BCINS_AD(op, ra, rd);
    }
    /* Using expr_free might cause asserts if the order is wrong. */
    if (e1->k == VNONRELOC && e1->u.s.info >= fs->nactvar)
        fs->freereg--;
    if (e2->k == VNONRELOC && e2->u.s.info >= fs->nactvar)
        fs->freereg--;
    bcemit_INS(fs, ins);
    eret->u.s.info = bcemit_jmp(fs);
    eret->k = VJMP;
}

 /* Fixup left side of binary operator. */
 static void bcemit_binop_left(FuncState *fs, BinOpr op, ExpDesc *e)
 {
     if (op == OPR_AND) {
         bcemit_branch_t(fs, e);
     } else if (op == OPR_OR) {
         bcemit_branch_f(fs, e);
     } else if (op == OPR_CONCAT) {
         expr_tonextreg(fs, e);
     } else if (op == OPR_EQ || op == OPR_NE) {
         if (!expr_isk_nojump(e))
```

```
            expr_toanyreg(fs, e);
    } else {
        if (!expr_isnumk_nojump(e))
            expr_toanyreg(fs, e);
    }
}

/* Emit binary operator. */
static void bcemit_binop(FuncState *fs, BinOpr op, ExpDesc *e1, ExpDesc *e2)
{
    if (op <= OPR_POW) {
        bcemit_arith(fs, op, e1, e2);
    } else if (op == OPR_AND) {
        kp_assert(e1->t == NO_JMP);  /* List must be closed. */
        expr_discharge(fs, e2);
        jmp_append(fs, &e2->f, e1->f);
        *e1 = *e2;
    } else if (op == OPR_OR) {
        kp_assert(e1->f == NO_JMP);  /* List must be closed. */
        expr_discharge(fs, e2);
        jmp_append(fs, &e2->t, e1->t);
        *e1 = *e2;
    } else if (op == OPR_CONCAT) {
        expr_toval(fs, e2);
        if (e2->k == VRELOCABLE && bc_op(*bcptr(fs, e2)) == BC_CAT) {
            kp_assert(e1->u.s.info == bc_b(*bcptr(fs, e2))-1);
            expr_free(fs, e1);
            setbc_b(bcptr(fs, e2), e1->u.s.info);
            e1->u.s.info = e2->u.s.info;
        } else {
            expr_tonextreg(fs, e2);
            expr_free(fs, e2);
            expr_free(fs, e1);
            e1->u.s.info = bcemit_ABC(fs, BC_CAT, 0, e1->u.s.info,
                                      e2->u.s.info);
        }
        e1->k = VRELOCABLE;
    } else {
        kp_assert(op == OPR_NE || op == OPR_EQ || op == OPR_LT ||
              op == OPR_GE || op == OPR_LE || op == OPR_GT);
        bcemit_comp(fs, op, e1, e2);
    }
}

/* Emit unary operator. */
static void bcemit_unop(FuncState *fs, BCOp op, ExpDesc *e)
{
    if (op == BC_NOT) {
        /* Swap true and false lists. */
        { BCPos temp = e->f; e->f = e->t; e->t = temp; }
        jmp_dropval(fs, e->f);
        jmp_dropval(fs, e->t);
        expr_discharge(fs, e);
        if (e->k == VKNIL || e->k == VKFALSE) {
            e->k = VKTRUE;
            return;
        } else if (expr_isk(e)) {
            e->k = VKFALSE;
            return;
        } else if (e->k == VJMP) {
            invertcond(fs, e);
            return;
        } else if (e->k == VRELOCABLE) {
            bcreg_reserve(fs, 1);
            setbc_a(bcptr(fs, e), fs->freereg-1);
            e->u.s.info = fs->freereg-1;
            e->k = VNONRELOC;
        } else {
            kp_assert(e->k == VNONRELOC);
        }
    } else {
        kp_assert(op == BC_UNM || op == BC_LEN);
        /* Constant-fold negations. */
        if (op == BC_UNM && !expr_hasjump(e)) {
            /* Avoid folding to -0. */
            if (expr_isnumk(e) && !expr_numiszero(e)) {
```

```
                ktap_val_t *o = expr_numtv(e);
                if (is_number(o))
                    set_number(o, -nvalue(o));
                return;
            }
        }
        expr_toanyreg(fs, e);
    }
    expr_free(fs, e);
    e->u.s.info = bcemit_AD(fs, op, 0, e->u.s.info);
    e->k = VRELOCABLE;
}

/* -- Lexer support ---------------------------------------------------- */

/* Check and consume optional token. */
static int lex_opt(LexState *ls, LexToken tok)
{
    if (ls->tok == tok) {
        kp_lex_next(ls);
        return 1;
    }
    return 0;
}

/* Check and consume token. */
static void lex_check(LexState *ls, LexToken tok)
{
    if (ls->tok != tok)
        err_token(ls, tok);
    kp_lex_next(ls);
}

/* Check for matching token. */
static void lex_match(LexState *ls, LexToken what, LexToken who, BCLine line)
{
    if (!lex_opt(ls, what)) {
        if (line == ls->linenumber) {
            err_token(ls, what);
        } else {
            const char *swhat = kp_lex_token2str(ls, what);
            const char *swho = kp_lex_token2str(ls, who);
            kp_lex_error(ls, ls->tok, KP_ERR_XMATCH, swhat, swho,
                                 line);
        }
    }
}

/* Check for string token. */
static ktap_str_t *lex_str(LexState *ls)
{
    ktap_str_t *s;

    if (ls->tok != TK_name)
        err_token(ls, TK_name);
    s = rawtsvalue(&ls->tokval);
    kp_lex_next(ls);
    return s;
}

/* -- Variable handling ------------------------------------------------ */

#define var_get(ls, fs, i)    ((ls)->vstack[(fs)->varmap[(i)]])

/* Define a new local variable. */
static void var_new(LexState *ls, BCReg n, ktap_str_t *name)
{
    FuncState *fs = ls->fs;
    int vtop = ls->vtop;

    checklimit(fs, fs->nactvar+n, KP_MAX_LOCVAR, "local variables");
    if (vtop >= ls->sizevstack) {
        if (ls->sizevstack >= KP_MAX_VSTACK)
            kp_lex_error(ls, 0, KP_ERR_XLIMC, KP_MAX_VSTACK);
        if (!ls->vstack) {
            ls->vstack = malloc(sizeof(VarInfo) * 20);
```

```
            ls->sizevstack = 20;
        } else {
            ls->vstack = realloc(ls->vstack,
                ls->sizevstack * sizeof(VarInfo) * 2);
            ls->sizevstack = ls->sizevstack * 2;
        }
    }
    kp_assert((uintptr_t)name < VARNAME__MAX ||
            kp_tab_getstr(fs->kt, name) != NULL);
    ls->vstack[vtop].name = name;
    fs->varmap[fs->nactvar+n] = (uint16_t)vtop;
    ls->vtop = vtop+1;
}

#define var_new_lit(ls, n, v) \
    var_new(ls, (n), kp_parse_keepstr(ls, "" v, sizeof(v)-1))

#define var_new_fixed(ls, n, vn) \
    var_new(ls, (n), (ktap_str_t *)(uintptr_t)(vn))

/* Add local variables. */
static void var_add(LexState *ls, BCReg nvars)
{
    FuncState *fs = ls->fs;
    BCReg nactvar = fs->nactvar;

    while (nvars--) {
        VarInfo *v = &var_get(ls, fs, nactvar);
        v->startpc = fs->pc;
        v->slot = nactvar++;
        v->info = 0;
    }
    fs->nactvar = nactvar;
}

/* Remove local variables. */
static void var_remove(LexState *ls, BCReg tolevel)
{
    FuncState *fs = ls->fs;
    while (fs->nactvar > tolevel)
        var_get(ls, fs, --fs->nactvar).endpc = fs->pc;
}

/* Lookup local variable name. */
static BCReg var_lookup_local(FuncState *fs, ktap_str_t *n)
{
    int i;

    for (i = fs->nactvar-1; i >= 0; i--) {
        if (n == var_get(fs->ls, fs, i).name)
            return (BCReg)i;
    }
    return (BCReg)-1;  /* Not found. */
}

/* Lookup or add upvalue index. */
static int var_lookup_uv(FuncState *fs, int vidx, ExpDesc *e)
{
    int i, n = fs->nuv;

    for (i = 0; i < n; i++)
        if (fs->uvmap[i] == vidx)
            return i;  /* Already exists. */

    /* Otherwise create a new one. */
    checklimit(fs, fs->nuv, KP_MAX_UPVAL, "upvalues");
    kp_assert(e->k == VLOCAL || e->k == VUPVAL);
    fs->uvmap[n] = (uint16_t)vidx;
    fs->uvtmp[n] = (uint16_t)(e->k == VLOCAL ? vidx :
            KP_MAX_VSTACK+e->u.s.info);
    fs->nuv = n+1;
    return n;
}

/* Forward declaration. */
static void fscope_uvmark(FuncState *fs, BCReg level);
```

```
1239
1240 /* Recursively lookup variables in enclosing functions. */
1241 static int var_lookup_(FuncState *fs, ktap_str_t *name, ExpDesc *e,
1242              int first)
1243 {
1244     if (fs) {
1245         BCReg reg = var_lookup_local(fs, name);
1246         if ((int32_t)reg >= 0) {  /* Local in this function? */
1247             expr_init(e, VLOCAL, reg);
1248             if (!first) {
1249                 /* Scope now has an upvalue. */
1250                 fscope_uvmark(fs, reg);
1251             }
1252             return (int)(e->u.s.aux = (uint32_t)fs->varmap[reg]);
1253         } else {
1254             /* Var in outer func? */
1255             int vidx = var_lookup_(fs->prev, name, e, 0);
1256             if ((int32_t)vidx >= 0) {
1257                 /* Yes, make it an upvalue here. */
1258                 e->u.s.info =
1259                     (uint8_t)var_lookup_uv(fs, vidx, e);
1260                 e->k = VUPVAL;
1261                 return vidx;
1262             }
1263         }
1264     } else {  /* Not found in any function, must be a global. */
1265         expr_init(e, VGLOBAL, 0);
1266         e->u.sval = name;
1267     }
1268     return (int)-1;  /* Global. */
1269 }
1270
1271 /* Lookup variable name. */
1272 #define var_lookup(ls, e) \
1273     var_lookup_((ls)->fs, lex_str(ls), (e), 1)
1274
1275 /* -- Goto an label handling ---------------------------------------------- */
1276
1277 /* Add a new goto or label. */
1278 static int gola_new(LexState *ls, ktap_str_t *name, uint8_t info, BCPos pc)
1279 {
1280     FuncState *fs = ls->fs;
1281     int vtop = ls->vtop;
1282
1283     if (vtop >= ls->sizevstack) {
1284         if (ls->sizevstack >= KP_MAX_VSTACK)
1285             kp_lex_error(ls, 0, KP_ERR_XLIMC, KP_MAX_VSTACK);
1286         if (!ls->vstack) {
1287             ls->vstack = malloc(sizeof(VarInfo) * 20);
1288             ls->sizevstack = 20;
1289         } else {
1290             ls->vstack = realloc(ls->vstack,
1291                     ls->sizevstack * sizeof(VarInfo) * 2);
1292             ls->sizevstack = ls->sizevstack * 2;
1293         }
1294     }
1295     kp_assert(name == NAME_BREAK ||
1296           kp_tab_getstr(fs->kt, name) != NULL);
1297     ls->vstack[vtop].name = name;
1298     ls->vstack[vtop].startpc = pc;
1299     ls->vstack[vtop].slot = (uint8_t)fs->nactvar;
1300     ls->vstack[vtop].info = info;
1301     ls->vtop = vtop+1;
1302     return vtop;
1303 }
1304
1305 #define gola_isgoto(v)        ((v)->info & VSTACK_GOTO)
1306 #define gola_islabel(v)        ((v)->info & VSTACK_LABEL)
1307 #define gola_isgotolabel(v)    ((v)->info & (VSTACK_GOTO|VSTACK_LABEL))
1308
1309 /* Patch goto to jump to label. */
1310 static void gola_patch(LexState *ls, VarInfo *vg, VarInfo *vl)
1311 {
1312     FuncState *fs = ls->fs;
1313     BCPos pc = vg->startpc;
1314
```

```
1315      vg->name = NULL; /* Invalidate pending goto. */
1316      setbc_a(&fs->bcbase[pc].ins, vl->slot);
1317      jmp_patch(fs, pc, vl->startpc);
1318  }
1319
1320  /* Patch goto to close upvalues. */
1321  static void gola_close(LexState *ls, VarInfo *vg)
1322  {
1323      FuncState *fs = ls->fs;
1324      BCPos pc = vg->startpc;
1325      BCIns *ip = &fs->bcbase[pc].ins;
1326      kp_assert(gola_isgoto(vg));
1327      kp_assert(bc_op(*ip) == BC_JMP || bc_op(*ip) == BC_UCLO);
1328      setbc_a(ip, vg->slot);
1329      if (bc_op(*ip) == BC_JMP) {
1330          BCPos next = jmp_next(fs, pc);
1331          if (next != NO_JMP)
1332              jmp_patch(fs, next, pc);  /* Jump to UCLO. */
1333          setbc_op(ip, BC_UCLO);  /* Turn into UCLO. */
1334          setbc_j(ip, NO_JMP);
1335      }
1336  }
1337
1338  /* Resolve pending forward gotos for label. */
1339  static void gola_resolve(LexState *ls, FuncScope *bl, int idx)
1340  {
1341      VarInfo *vg = ls->vstack + bl->vstart;
1342      VarInfo *vl = ls->vstack + idx;
1343      for (; vg < vl; vg++)
1344          if (vg->name == vl->name && gola_isgoto(vg)) {
1345              if (vg->slot < vl->slot) {
1346                  ktap_str_t *name =
1347                      var_get(ls, ls->fs, vg->slot).name;
1348                  kp_assert((uintptr_t)name >= VARNAME__MAX);
1349                  ls->linenumber =
1350                      ls->fs->bcbase[vg->startpc].line;
1351                  kp_assert(vg->name != NAME_BREAK);
1352                  kp_lex_error(ls, 0, KP_ERR_XGSCOPE,
1353                  getstr(vg->name), getstr(name));
1354              }
1355              gola_patch(ls, vg, vl);
1356          }
1357  }
1358
1359  /* Fixup remaining gotos and labels for scope. */
1360  static void gola_fixup(LexState *ls, FuncScope *bl)
1361  {
1362      VarInfo *v = ls->vstack + bl->vstart;
1363      VarInfo *ve = ls->vstack + ls->vtop;
1364
1365      for (; v < ve; v++) {
1366          ktap_str_t *name = v->name;
1367          /* Only consider remaining valid gotos/labels. */
1368          if (name != NULL) {
1369              if (gola_islabel(v)) {
1370                  VarInfo *vg;
1371                  /* Invalidate label that goes out of scope. */
1372                  v->name = NULL;
1373                  /* Resolve pending backward gotos. */
1374                  for (vg = v+1; vg < ve; vg++)
1375                      if (vg->name == name &&
1376                          gola_isgoto(vg)) {
1377                          if ((bl->flags&FSCOPE_UPVAL) &&
1378                               vg->slot > v->slot)
1379                              gola_close(ls, vg);
1380                          gola_patch(ls, vg, v);
1381                      }
1382              } else if (gola_isgoto(v)) {
1383                  /* Propagate goto or break to outer scope. */
1384                  if (bl->prev) {
1385                      bl->prev->flags |= name == NAME_BREAK ?                         FSCOPE_BREAK :
FSCOPE_GOLA;
1386                      v->slot = bl->nactvar;
1387                      if ((bl->flags & FSCOPE_UPVAL))
1388                          gola_close(ls, v);
1389                  } else {
```

```
                    ls->linenumber =
                    ls->fs->bcbase[v->startpc].line;
                    if (name == NAME_BREAK)
                        kp_lex_error(ls, 0, KP_ERR_XBREAK);
                    else
                        kp_lex_error(ls, 0, KP_ERR_XLUNDEF, getstr(name));
                }
            }
        }
    }
}

/* Find existing label. */
static VarInfo *gola_findlabel(LexState *ls, ktap_str_t *name)
{
    VarInfo *v = ls->vstack + ls->fs->bl->vstart;
    VarInfo *ve = ls->vstack + ls->vtop;

    for (; v < ve; v++)
        if (v->name == name && gola_islabel(v))
            return v;
    return NULL;
}

/* -- Scope handling ---------------------------------------------------- */

/* Begin a scope. */
static void fscope_begin(FuncState *fs, FuncScope *bl, int flags)
{
    bl->nactvar = (uint8_t)fs->nactvar;
    bl->flags = flags;
    bl->vstart = fs->ls->vtop;
    bl->prev = fs->bl;
    fs->bl = bl;
    kp_assert(fs->freereg == fs->nactvar);
}

/* End a scope. */
static void fscope_end(FuncState *fs)
{
    FuncScope *bl = fs->bl;
    LexState *ls = fs->ls;

    fs->bl = bl->prev;
    var_remove(ls, bl->nactvar);
    fs->freereg = fs->nactvar;
    kp_assert(bl->nactvar == fs->nactvar);
    if ((bl->flags & (FSCOPE_UPVAL|FSCOPE_NOCLOSE)) == FSCOPE_UPVAL)
        bcemit_AJ(fs, BC_UCLO, bl->nactvar, 0);
    if ((bl->flags & FSCOPE_BREAK)) {
        if ((bl->flags & FSCOPE_LOOP)) {
            int idx = gola_new(ls, NAME_BREAK, VSTACK_LABEL,
                               fs->pc);
            ls->vtop = idx;  /* Drop break label immediately. */
            gola_resolve(ls, bl, idx);
            return;
        }  /* else: need the fixup step to propagate the breaks. */
    } else if (!(bl->flags & FSCOPE_GOLA)) {
        return;
    }
    gola_fixup(ls, bl);
}

/* Mark scope as having an upvalue. */
static void fscope_uvmark(FuncState *fs, BCReg level)
{
    FuncScope *bl;

    for (bl = fs->bl; bl && bl->nactvar > level; bl = bl->prev);
    if (bl)
        bl->flags |= FSCOPE_UPVAL;
}

/* -- Function state management ----------------------------------------- */

/* Fixup bytecode for prototype. */
```

```
1466 static void fs_fixup_bc(FuncState *fs, ktap_proto_t *pt, BCIns *bc, int n)
1467 {
1468     BCInsLine *base = fs->bcbase;
1469     int i;
1470
1471     pt->sizebc = n;
1472     bc[0] = BCINS_AD((fs->flags & PROTO_VARARG) ? BC_FUNCV : BC_FUNCF,
1473              fs->framesize, 0);
1474     for (i = 1; i < n; i++)
1475         bc[i] = base[i].ins;
1476 }
1477
1478 /* Fixup upvalues for child prototype, step #2. */
1479 static void fs_fixup_uv2(FuncState *fs, ktap_proto_t *pt)
1480 {
1481     VarInfo *vstack = fs->ls->vstack;
1482     uint16_t *uv = pt->uv;
1483     int i, n = pt->sizeuv;
1484
1485     for (i = 0; i < n; i++) {
1486         VarIndex vidx = uv[i];
1487         if (vidx >= KP_MAX_VSTACK)
1488             uv[i] = vidx - KP_MAX_VSTACK;
1489         else if ((vstack[vidx].info & VSTACK_VAR_RW))
1490             uv[i] = vstack[vidx].slot | PROTO_UV_LOCAL;
1491         else
1492             uv[i] = vstack[vidx].slot | PROTO_UV_LOCAL |
1493                     PROTO_UV_IMMUTABLE;
1494     }
1495 }
1496
1497 /* Fixup constants for prototype. */
1498 static void fs_fixup_k(FuncState *fs, ktap_proto_t *pt, void *kptr)
1499 {
1500     ktap_tab_t *kt;
1501     ktap_node_t *node;
1502     int i, hmask;
1503
1504     checklimitgt(fs, fs->nkn, BCMAX_D+1, "constants");
1505     checklimitgt(fs, fs->nkgc, BCMAX_D+1, "constants");
1506
1507     pt->k = kptr;
1508     pt->sizekn = fs->nkn;
1509     pt->sizekgc = fs->nkgc;
1510     kt = fs->kt;
1511     node = kt->node;
1512     hmask = kt->hmask;
1513     for (i = 0; i <= hmask; i++) {
1514         ktap_node_t *n = &node[i];
1515
1516         if (tvhaskslot(&n->val)) {
1517             ptrdiff_t kidx = (ptrdiff_t)tvkslot(&n->val);
1518             kp_assert(!is_number(&n->key));
1519             if (is_number(&n->key)) {
1520                 ktap_val_t *tv = &((ktap_val_t *)kptr)[kidx];
1521                 *tv = n->key;
1522             } else {
1523                 ktap_obj_t *o = n->key.val.gc;
1524                 ktap_obj_t **v = (ktap_obj_t **)kptr;
1525                 v[~kidx] = o;
1526                 if (is_proto(&n->key))
1527                     fs_fixup_uv2(fs, (ktap_proto_t *)o);
1528             }
1529         }
1530     }
1531 }
1532
1533 /* Fixup upvalues for prototype, step #1. */
1534 static void fs_fixup_uv1(FuncState *fs, ktap_proto_t *pt, uint16_t *uv)
1535 {
1536     pt->uv = uv;
1537     pt->sizeuv = fs->nuv;
1538     memcpy(uv, fs->uvtmp, fs->nuv*sizeof(VarIndex));
1539 }
1540
1541 #ifndef KTAP_DISABLE_LINEINFO
```

```c
/* Prepare lineinfo for prototype. */
static size_t fs_prep_line(FuncState *fs, BCLine numline)
{
    return (fs->pc-1) << (numline < 256 ? 0 : numline < 65536 ? 1 : 2);
}

/* Fixup lineinfo for prototype. */
static void fs_fixup_line(FuncState *fs, ktap_proto_t *pt,
              void *lineinfo, BCLine numline)
{
    BCInsLine *base = fs->bcbase + 1;
    BCLine first = fs->linedefined;
    int i = 0, n = fs->pc-1;

    pt->firstline = fs->linedefined;
    pt->numline = numline;
    pt->lineinfo = lineinfo;
    if (numline < 256) {
        uint8_t *li = (uint8_t *)lineinfo;
        do {
            BCLine delta = base[i].line - first;
            kp_assert(delta >= 0 && delta < 256);
            li[i] = (uint8_t)delta;
        } while (++i < n);
    } else if (numline < 65536) {
        uint16_t *li = (uint16_t *)lineinfo;
        do {
            BCLine delta = base[i].line - first;
            kp_assert(delta >= 0 && delta < 65536);
            li[i] = (uint16_t)delta;
        } while (++i < n);
    } else {
        uint32_t *li = (uint32_t *)lineinfo;
        do {
            BCLine delta = base[i].line - first;
            kp_assert(delta >= 0);
            li[i] = (uint32_t)delta;
        } while (++i < n);
    }
}

/* Prepare variable info for prototype. */
static size_t fs_prep_var(LexState *ls, FuncState *fs, size_t *ofsvar)
{
    VarInfo *vs =ls->vstack, *ve;
    int i, n;
    BCPos lastpc;

    kp_buf_reset(&ls->sb);  /* Copy to temp. string buffer. */
    /* Store upvalue names. */
    for (i = 0, n = fs->nuv; i < n; i++) {
        ktap_str_t *s = vs[fs->uvmap[i]].name;
        int len = s->len+1;
        char *p = kp_buf_more(&ls->sb, len);
        p = kp_buf_wmem(p, getstr(s), len);
        setsbufP(&ls->sb, p);
    }

    *ofsvar = sbuflen(&ls->sb);
    lastpc = 0;
    /* Store local variable names and compressed ranges. */
    for (ve = vs + ls->vtop, vs += fs->vbase; vs < ve; vs++) {
        if (!gola_isgotolabel(vs)) {
            ktap_str_t *s = vs->name;
            BCPos startpc;
            char *p;
            if ((uintptr_t)s < VARNAME__MAX) {
                p = kp_buf_more(&ls->sb, 1 + 2*5);
                *p++ = (char)(uintptr_t)s;
            } else {
                int len = s->len+1;
                p = kp_buf_more(&ls->sb, len + 2*5);
                p = kp_buf_wmem(p, getstr(s), len);
            }
            startpc = vs->startpc;
            p = strfmt_wuleb128(p, startpc-lastpc);
```

```
            p = strfmt_wuleb128(p, vs->endpc-startpc);
            setsbufP(&ls->sb, p);
            lastpc = startpc;
        }
    }

    kp_buf_putb(&ls->sb, '\0');  /* Terminator for varinfo. */
    return sbuflen(&ls->sb);
}

/* Fixup variable info for prototype. */
static void fs_fixup_var(LexState *ls, ktap_proto_t *pt, uint8_t *p,
             size_t ofsvar)
{
    pt->uvinfo = p;
    pt->varinfo = (char *)p + ofsvar;
    /* Copy from temp. buffer. */
    memcpy(p, sbufB(&ls->sb), sbuflen(&ls->sb));
}
#else

/* Initialize with empty debug info, if disabled. */
#define fs_prep_line(fs, numline)        (UNUSED(numline), 0)
#define fs_fixup_line(fs, pt, li, numline) \
  pt->firstline = pt->numline = 0, (pt)->lineinfo = NULL
#define fs_prep_var(ls, fs, ofsvar)        (UNUSED(ofsvar), 0)
#define fs_fixup_var(ls, pt, p, ofsvar) \
  (pt)->uvinfo = NULL, (pt)->varinfo = NULL

#endif

/* Check if bytecode op returns. */
static int bcopisret(BCOp op)
{
    switch (op) {
    case BC_CALLMT: case BC_CALLT:
    case BC_RETM: case BC_RET: case BC_RET0: case BC_RET1:
        return 1;
    default:
        return 0;
    }
}

/* Fixup return instruction for prototype. */
static void fs_fixup_ret(FuncState *fs)
{
    BCPos lastpc = fs->pc;

    if (lastpc <= fs->lasttarget ||
        !bcopisret(bc_op(fs->bcbase[lastpc-1].ins))) {
        if ((fs->bl->flags & FSCOPE_UPVAL))
            bcemit_AJ(fs, BC_UCLO, 0, 0);
        bcemit_AD(fs, BC_RET0, 0, 1);  /* Need final return. */
    }
    fs->bl->flags |= FSCOPE_NOCLOSE;  /* Handled above. */
    fscope_end(fs);
    kp_assert(fs->bl == NULL);
    /* May need to fixup returns encoded before first function
     * was created. */
    if (fs->flags & PROTO_FIXUP_RETURN) {
        BCPos pc;
        for (pc = 1; pc < lastpc; pc++) {
            BCIns ins = fs->bcbase[pc].ins;
            BCPos offset;
            switch (bc_op(ins)) {
            case BC_CALLMT: case BC_CALLT:
            case BC_RETM: case BC_RET: case BC_RET0: case BC_RET1:
                /* Copy original instruction. */
                offset = bcemit_INS(fs, ins);
                fs->bcbase[offset].line = fs->bcbase[pc].line;
                offset = offset-(pc+1)+BCBIAS_J;
                if (offset > BCMAX_D)
                    err_syntax(fs->ls, KP_ERR_XFIXUP);
                /* Replace with UCLO plus branch. */
                fs->bcbase[pc].ins = BCINS_AD(BC_UCLO, 0,
                                offset);
```

```
                break;
            case BC_UCLO:
                return;  /* We're done. */
            default:
                break;
            }
        }
    }
}

/* Finish a FuncState and return the new prototype. */
static ktap_proto_t *fs_finish(LexState *ls, BCLine line)
{
    FuncState *fs = ls->fs;
    BCLine numline = line - fs->linedefined;
    size_t sizept, ofsk, ofsuv, ofsli, ofsdbg, ofsvar;
    ktap_proto_t *pt;

    /* Apply final fixups. */
    fs_fixup_ret(fs);

    /* Calculate total size of prototype including all colocated arrays. */
    sizept = sizeof(ktap_proto_t) + fs->pc*sizeof(BCIns) +
             fs->nkgc*sizeof(ktap_obj_t *);
    sizept = (sizept + sizeof(ktap_val_t)-1) & ~(sizeof(ktap_val_t)-1);
    ofsk = sizept; sizept += fs->nkn*sizeof(ktap_val_t);
    ofsuv = sizept; sizept += ((fs->nuv+1)&~1)*2;
    ofsli = sizept; sizept += fs_prep_line(fs, numline);
    ofsdbg = sizept; sizept += fs_prep_var(ls, fs, &ofsvar);

    /* Allocate prototype and initialize its fields. */
    pt = (ktap_proto_t *)malloc((int)sizept);
    pt->gct = ~KTAP_TPROTO;
    pt->sizept = (int)sizept;
    pt->flags =
        (uint8_t)(fs->flags & ~(PROTO_HAS_RETURN|PROTO_FIXUP_RETURN));
    pt->numparams = fs->numparams;
    pt->framesize = fs->framesize;
    pt->chunkname = ls->chunkname;

    /* Close potentially uninitialized gap between bc and kgc. */
    *(uint32_t *)((char *)pt + ofsk - sizeof(ktap_obj_t *)*(fs->nkgc+1)) = 0;
    fs_fixup_bc(fs, pt, (BCIns *)((char *)pt + sizeof(ktap_proto_t)), fs->pc);
    fs_fixup_k(fs, pt, (void *)((char *)pt + ofsk));
    fs_fixup_uv1(fs, pt, (uint16_t *)((char *)pt + ofsuv));
    fs_fixup_line(fs, pt, (void *)((char *)pt + ofsli), numline);
    fs_fixup_var(ls, pt, (uint8_t *)((char *)pt + ofsdbg), ofsvar);

    ls->vtop = fs->vbase;  /* Reset variable stack. */
    ls->fs = fs->prev;
    kp_assert(ls->fs != NULL || ls->tok == TK_eof);
    return pt;
}

/* Initialize a new FuncState. */
static void fs_init(LexState *ls, FuncState *fs)
{
    fs->prev = ls->fs; ls->fs = fs;  /* Append to list. */
    fs->ls = ls;
    fs->vbase = ls->vtop;
    fs->pc = 0;
    fs->lasttarget = 0;
    fs->jpc = NO_JMP;
    fs->freereg = 0;
    fs->nkgc = 0;
    fs->nkn = 0;
    fs->nactvar = 0;
    fs->nuv = 0;
    fs->bl = NULL;
    fs->flags = 0;
    fs->framesize = 1;  /* Minimum frame size. */
    fs->kt = kp_tab_new();
}

/* -- Expressions ------------------------------------------------------- */

```

```
1770 /* Forward declaration. */
1771 static void expr(LexState *ls, ExpDesc *v);
1772
1773 /* Return string expression. */
1774 static void expr_str(LexState *ls, ExpDesc *e)
1775 {
1776     expr_init(e, VKSTR, 0);
1777     e->u.sval = lex_str(ls);
1778 }
1779
1780 #define checku8(x)     ((x) == (int32_t)(uint8_t)(x))
1781
1782 /* Return index expression. */
1783 static void expr_index(FuncState *fs, ExpDesc *t, ExpDesc *e)
1784 {
1785     /* Already called: expr_toval(fs, e). */
1786     t->k = VINDEXED;
1787     if (expr_isnumk(e)) {
1788         ktap_number n = expr_numberV(e);
1789         int32_t k = (int)n;
1790         if (checku8(k) && n == (ktap_number)k) {
1791             /* 256..511: const byte key */
1792             t->u.s.aux = BCMAX_C+1+(uint32_t)k;
1793             return;
1794         }
1795     } else if (expr_isstrk(e)) {
1796         BCReg idx = const_str(fs, e);
1797         if (idx <= BCMAX_C) {
1798             /* -256..-1: const string key */
1799             t->u.s.aux = ~idx;
1800             return;
1801         }
1802     }
1803     t->u.s.aux = expr_toanyreg(fs, e);  /* 0..255: register */
1804 }
1805
1806 /* Parse index expression with named field. */
1807 static void expr_field(LexState *ls, ExpDesc *v)
1808 {
1809     FuncState *fs = ls->fs;
1810     ExpDesc key;
1811
1812     expr_toanyreg(fs, v);
1813     kp_lex_next(ls);  /* Skip dot or colon. */
1814     expr_str(ls, &key);
1815     expr_index(fs, v, &key);
1816 }
1817
1818 /* Parse index expression with brackets. */
1819 static void expr_bracket(LexState *ls, ExpDesc *v)
1820 {
1821     kp_lex_next(ls);  /* Skip '['. */
1822     expr(ls, v);
1823     expr_toval(ls->fs, v);
1824     lex_check(ls, ']');
1825 }
1826
1827 /* Get value of constant expression. */
1828 static void expr_kvalue(ktap_val_t *v, ExpDesc *e)
1829 {
1830     if (e->k <= VKTRUE) {
1831         setitype(v, ~(uint32_t)e->k);
1832     } else if (e->k == VKSTR) {
1833         set_string(v, e->u.sval);
1834     } else {
1835         kp_assert(tvisnumber(expr_numtv(e)));
1836         *v = *expr_numtv(e);
1837     }
1838 }
1839
1840 #define FLS(x)       ((uint32_t)(__builtin_clz(x)^31))
1841 #define hsize2hbits(s) ((s) ? ((s)==1 ? 1 : 1+FLS((uint32_t)((s)-1))) : 0)
1842
1843
1844 /* Parse table constructor expression. */
1845 static void expr_table(LexState *ls, ExpDesc *e)
```

```
{
    FuncState *fs = ls->fs;
    BCLine line = ls->linenumber;
    ktap_tab_t *t = NULL;
    int vcall = 0, needarr = 0, fixt = 0;
    uint32_t narr = 1;  /* First array index. */
    uint32_t nhash = 0;  /* Number of hash entries. */
    BCReg freg = fs->freereg;
    BCPos pc = bcemit_AD(fs, BC_TNEW, freg, 0);

    expr_init(e, VNONRELOC, freg);
    bcreg_reserve(fs, 1);
    freg++;
    lex_check(ls, '{');
    while (ls->tok != '}') {
        ExpDesc key, val;
        vcall = 0;
        if (ls->tok == '[') {
            expr_bracket(ls, &key);/* Already calls expr_toval. */
            if (!expr_isk(&key))
                expr_index(fs, e, &key);
            if (expr_isnumk(&key) && expr_numiszero(&key))
                needarr = 1;
            else
                nhash++;
            lex_check(ls, '=');
        } else if ((ls->tok == TK_name) &&
                kp_lex_lookahead(ls) == '=') {
            expr_str(ls, &key);
            lex_check(ls, '=');
            nhash++;
        } else {
            expr_init(&key, VKNUM, 0);
            set_number(&key.u.nval, (int)narr);
            narr++;
            needarr = vcall = 1;
        }
        expr(ls, &val);
        if (expr_isk(&key) && key.k != VKNIL &&
            (key.k == VKSTR || expr_isk_nojump(&val))) {
            ktap_val_t k, *v;
            if (!t) {  /* Create template table on demand. */
                BCReg kidx;
                t = kp_tab_new();
                kidx = const_gc(fs, obj2gco(t), KTAP_TTAB);
                fs->bcbase[pc].ins = BCINS_AD(BC_TDUP, freg-1,
                                    kidx);
            }
            vcall = 0;
            expr_kvalue(&k, &key);
            v = kp_tab_set(t, &k);
            /* Add const key/value to template table. */
            if (expr_isk_nojump(&val)) {
                expr_kvalue(v, &val);
            } else {
                /* Otherwise create dummy string key (avoids kp_tab_newkey). */
                set_table(v, t);  /* Preserve key with table itself as value. */
                fixt = 1;/* Fix this later, after all resizes. */
                goto nonconst;
            }
        } else {
nonconst:
            if (val.k != VCALL) {
                expr_toanyreg(fs, &val);
                vcall = 0;
            }
            if (expr_isk(&key))
                expr_index(fs, e, &key);
            bcemit_store(fs, e, &val);
        }
        fs->freereg = freg;
        if (!lex_opt(ls, ',') && !lex_opt(ls, ';'))
            break;
    }
    lex_match(ls, '}', '{', line);
    if (vcall) {
```

```
        BCInsLine *ilp = &fs->bcbase[fs->pc-1];
        ExpDesc en;
        kp_assert(bc_a(ilp->ins) == freg &&
            bc_op(ilp->ins) == (narr > 256 ? BC_TSETV : BC_TSETB));
        expr_init(&en, VKNUM, 0);
        set_number(&en.u.nval, narr - 1);
        if (narr > 256) { fs->pc--; ilp--; }
        ilp->ins = BCINS_AD(BC_TSETM, freg, const_num(fs, &en));
        setbc_b(&ilp[-1].ins, 0);
    }
    if (pc == fs->pc-1) {  /* Make expr relocable if possible. */
        e->u.s.info = pc;
        fs->freereg--;
        e->k = VRELOCABLE;
    } else {
        e->k = VNONRELOC;  /* May have been changed by expr_index. */
    }
    if (!t) {  /* Construct TNEW RD: hhhhhaaaaaaaaaaa. */
        BCIns *ip = &fs->bcbase[pc].ins;
        if (!needarr) narr = 0;
        else if (narr < 3) narr = 3;
        else if (narr > 0x7ff) narr = 0x7ff;
        setbc_d(ip, narr|(hsize2hbits(nhash)<<11));
    } else {
        if (fixt) {  /* Fix value for dummy keys in template table. */
            ktap_node_t *node = t->node;
            uint32_t i, hmask = t->hmask;
            for (i = 0; i <= hmask; i++) {
                ktap_node_t *n = &node[i];
                if (is_table(&n->val)) {
                    kp_assert(tabV(&n->val) == t);
                    /* Turn value into nil. */
                    set_nil(&n->val);
                }
            }
        }
    }
}

/* Parse function parameters. */
static BCReg parse_params(LexState *ls, int needself)
{
    FuncState *fs = ls->fs;
    BCReg nparams = 0;
    lex_check(ls, '(');
    if (needself)
        var_new_lit(ls, nparams++, "self");
    if (ls->tok != ')') {
        do {
            if (ls->tok == TK_name) {
                var_new(ls, nparams++, lex_str(ls));
            } else if (ls->tok == TK_dots) {
                kp_lex_next(ls);
                fs->flags |= PROTO_VARARG;
                break;
            } else {
                err_syntax(ls, KP_ERR_XPARAM);
            }
        } while (lex_opt(ls, ','));
    }
    var_add(ls, nparams);
    kp_assert(fs->nactvar == nparams);
    bcreg_reserve(fs, nparams);
    lex_check(ls, ')');
    return nparams;
}

/* Forward declaration. */
static void parse_chunk(LexState *ls);

/* Parse body of a function. */
static void parse_body(LexState *ls, ExpDesc *e, int needself, BCLine line)
{
    FuncState fs, *pfs = ls->fs;
    FuncScope bl;
    ktap_proto_t *pt;
```

```
    ptrdiff_t oldbase = pfs->bcbase - ls->bcstack;

    fs_init(ls, &fs);
    fscope_begin(&fs, &bl, 0);
    fs.linedefined = line;
    fs.numparams = (uint8_t)parse_params(ls, needself);
    fs.bcbase = pfs->bcbase + pfs->pc;
    fs.bclim = pfs->bclim - pfs->pc;
    bcemit_AD(&fs, BC_FUNCF, 0, 0);  /* Placeholder. */
    lex_check(ls, '{');
    parse_chunk(ls);
    lex_check(ls, '}');
    pt = fs_finish(ls, (ls->lastline = ls->linenumber));
    pfs->bcbase = ls->bcstack + oldbase;  /* May have been reallocated. */
    pfs->bclim = (BCPos)(ls->sizebcstack - oldbase);
    /* Store new prototype in the constant array of the parent. */
    expr_init(e, VRELOCABLE,
        bcemit_AD(pfs, BC_FNEW, 0,
              const_gc(pfs, (ktap_obj_t *)pt, KTAP_TPROTO)));
    if (!(pfs->flags & PROTO_CHILD)) {
        if (pfs->flags & PROTO_HAS_RETURN)
            pfs->flags |= PROTO_FIXUP_RETURN;
        pfs->flags |= PROTO_CHILD;
    }
    //kp_lex_next(ls);
}

/* Parse body of a function, for 'trace/trace_end/profile/tick' closure */
static void parse_body_no_args(LexState *ls, ExpDesc *e, int needself,
                BCLine line)
{
    FuncState fs, *pfs = ls->fs;
    FuncScope bl;
    ktap_proto_t *pt;
    ptrdiff_t oldbase = pfs->bcbase - ls->bcstack;

    fs_init(ls, &fs);
    fscope_begin(&fs, &bl, 0);
    fs.linedefined = line;
    fs.numparams = 0;
    fs.bcbase = pfs->bcbase + pfs->pc;
    fs.bclim = pfs->bclim - pfs->pc;
    bcemit_AD(&fs, BC_FUNCF, 0, 0);  /* Placeholder. */
    lex_check(ls, '{');
    parse_chunk(ls);
    lex_check(ls, '}');
    pt = fs_finish(ls, (ls->lastline = ls->linenumber));
    pfs->bcbase = ls->bcstack + oldbase;  /* May have been reallocated. */
    pfs->bclim = (BCPos)(ls->sizebcstack - oldbase);
    /* Store new prototype in the constant array of the parent. */
    expr_init(e, VRELOCABLE,
        bcemit_AD(pfs, BC_FNEW, 0,
              const_gc(pfs, (ktap_obj_t *)pt, KTAP_TPROTO)));
    if (!(pfs->flags & PROTO_CHILD)) {
        if (pfs->flags & PROTO_HAS_RETURN)
            pfs->flags |= PROTO_FIXUP_RETURN;
        pfs->flags |= PROTO_CHILD;
    }
    //kp_lex_next(ls);
}


/* Parse expression list. Last expression is left open. */
static BCReg expr_list(LexState *ls, ExpDesc *v)
{
    BCReg n = 1;

    expr(ls, v);
    while (lex_opt(ls, ',')) {
        expr_tonextreg(ls->fs, v);
        expr(ls, v);
        n++;
    }
    return n;
}

```

```c
/* Parse function argument list. */
static void parse_args(LexState *ls, ExpDesc *e)
{
    FuncState *fs = ls->fs;
    ExpDesc args;
    BCIns ins;
    BCReg base;
    BCLine line = ls->linenumber;

    if (ls->tok == '(') {
        if (line != ls->lastline)
            err_syntax(ls, KP_ERR_XAMBIG);
        kp_lex_next(ls);
        if (ls->tok == ')') {  /* f(). */
            args.k = VVOID;
        } else {
            expr_list(ls, &args);
            /* f(a, b, g()) or f(a, b, ...). */
            if (args.k == VCALL) {
                /* Pass on multiple results. */
                setbc_b(bcptr(fs, &args), 0);
            }
        }
        lex_match(ls, ')', '(', line);
    } else if (ls->tok == '{') {
        expr_table(ls, &args);
    } else if (ls->tok == TK_string) {
        expr_init(&args, VKSTR, 0);
        args.u.sval = rawtsvalue(&ls->tokval);
        kp_lex_next(ls);
    } else {
        err_syntax(ls, KP_ERR_XFUNARG);
        return;  /* Silence compiler. */
    }

    kp_assert(e->k == VNONRELOC);
    base = e->u.s.info;  /* Base register for call. */
    if (args.k == VCALL) {
        ins = BCINS_ABC(BC_CALLM, base, 2, args.u.s.aux - base - 1);
    } else {
        if (args.k != VVOID)
            expr_tonextreg(fs, &args);
        ins = BCINS_ABC(BC_CALL, base, 2, fs->freereg - base);
    }
    expr_init(e, VCALL, bcemit_INS(fs, ins));
    e->u.s.aux = base;
    fs->bcbase[fs->pc - 1].line = line;
    fs->freereg = base+1;  /* Leave one result by default. */
}

/* Parse primary expression. */
static void expr_primary(LexState *ls, ExpDesc *v)
{
    FuncState *fs = ls->fs;

    /* Parse prefix expression. */
    if (ls->tok == '(') {
        BCLine line = ls->linenumber;
        kp_lex_next(ls);
        expr(ls, v);
        lex_match(ls, ')', '(', line);
        expr_discharge(ls->fs, v);
    } else if (ls->tok == TK_name) {
        var_lookup(ls, v);
    } else {
        err_syntax(ls, KP_ERR_XSYMBOL);
    }

    for (;;) {  /* Parse multiple expression suffixes. */
        if (ls->tok == '.') {
            expr_field(ls, v);
        } else if (ls->tok == '[') {
            ExpDesc key;
            expr_toanyreg(fs, v);
            expr_bracket(ls, &key);
            expr_index(fs, v, &key);
```

```
        } else if (ls->tok == ':') {
            ExpDesc key;
            kp_lex_next(ls);
            expr_str(ls, &key);
            bcemit_method(fs, v, &key);
            parse_args(ls, v);
        } else if (ls->tok == '(' || ls->tok == TK_string ||
                ls->tok == '{') {
            expr_tonextreg(fs, v);
            parse_args(ls, v);
        } else {
            break;
        }
    }
}

/* Parse simple expression. */
static void expr_simple(LexState *ls, ExpDesc *v)
{
    switch (ls->tok) {
    case TK_number:
        expr_init(v, VKNUM, 0);
        set_obj(&v->u.nval, &ls->tokval);
        break;
    case TK_string:
        expr_init(v, VKSTR, 0);
        v->u.sval = rawtsvalue(&ls->tokval);
        break;
    case TK_nil:
        expr_init(v, VKNIL, 0);
        break;
    case TK_true:
        expr_init(v, VKTRUE, 0);
        break;
    case TK_false:
        expr_init(v, VKFALSE, 0);
        break;
    case TK_dots: {  /* Vararg. */
        FuncState *fs = ls->fs;
        BCReg base;
        checkcond(ls, fs->flags & PROTO_VARARG, KP_ERR_XDOTS);
        bcreg_reserve(fs, 1);
        base = fs->freereg-1;
        expr_init(v, VCALL, bcemit_ABC(fs, BC_VARG, base, 2,
        fs->numparams));
        v->u.s.aux = base;
        break;
    }
    case '{':  /* Table constructor. */
        expr_table(ls, v);
        return;
    case TK_function:
        kp_lex_next(ls);
        parse_body(ls, v, 0, ls->linenumber);
        return;
    case TK_argstr:
        expr_init(v, VARGSTR, 0);
        break;
    case TK_probename:
        expr_init(v, VARGNAME, 0);
        break;
    case TK_arg0: case TK_arg1: case TK_arg2: case TK_arg3: case TK_arg4:
    case TK_arg5: case TK_arg6: case TK_arg7: case TK_arg8: case TK_arg9:
        expr_init(v, VARGN, ls->tok - TK_arg0);
        break;
    case TK_pid:
        expr_init(v, VPID, 0);
        break;
    case TK_tid:
        expr_init(v, VTID, 0);
        break;
    case TK_uid:
        expr_init(v, VUID, 0);
        break;
    case TK_cpu:
        expr_init(v, VCPU, 0);
```

```
2226        break;
2227    case TK_execname:
2228        expr_init(v, VEXECNAME, 0);
2229        break;
2230    default:
2231        expr_primary(ls, v);
2232        return;
2233    }
2234    kp_lex_next(ls);
2235 }
2236
2237 /* Manage syntactic levels to avoid blowing up the stack. */
2238 static void synlevel_begin(LexState *ls)
2239 {
2240    if (++ls->level >= KP_MAX_XLEVEL)
2241        kp_lex_error(ls, 0, KP_ERR_XLEVELS);
2242 }
2243
2244 #define synlevel_end(ls)    ((ls)->level--)
2245
2246 /* Convert token to binary operator. */
2247 static BinOpr token2binop(LexToken tok)
2248 {
2249    switch (tok) {
2250    case '+':    return OPR_ADD;
2251    case '-':    return OPR_SUB;
2252    case '*':    return OPR_MUL;
2253    case '/':    return OPR_DIV;
2254    case '%':    return OPR_MOD;
2255    case '^':    return OPR_POW;
2256    case TK_concat: return OPR_CONCAT;
2257    case TK_ne:    return OPR_NE;
2258    case TK_eq:    return OPR_EQ;
2259    case '<':    return OPR_LT;
2260    case TK_le:    return OPR_LE;
2261    case '>':    return OPR_GT;
2262    case TK_ge:    return OPR_GE;
2263    case TK_and:    return OPR_AND;
2264    case TK_or:    return OPR_OR;
2265    default:    return OPR_NOBINOPR;
2266    }
2267 }
2268
2269 /* Priorities for each binary operator. ORDER OPR. */
2270 static const struct {
2271    uint8_t left;    /* Left priority. */
2272    uint8_t right;    /* Right priority. */
2273 } priority[] = {
2274    {6,6}, {6,6}, {7,7}, {7,7}, {7,7},    /* ADD SUB MUL DIV MOD */
2275    {10,9}, {5,4},            /* POW CONCAT (right associative) */
2276    {3,3}, {3,3},                /* EQ NE */
2277    {3,3}, {3,3}, {3,3}, {3,3},        /* LT GE GT LE */
2278    {2,2}, {1,1}                /* AND OR */
2279 };
2280
2281 #define UNARY_PRIORITY        8  /* Priority for unary operators. */
2282
2283 /* Forward declaration. */
2284 static BinOpr expr_binop(LexState *ls, ExpDesc *v, uint32_t limit);
2285
2286 /* Parse unary expression. */
2287 static void expr_unop(LexState *ls, ExpDesc *v)
2288 {
2289    BCOp op;
2290    if (ls->tok == TK_not) {
2291        op = BC_NOT;
2292    } else if (ls->tok == '-') {
2293        op = BC_UNM;
2294 #if 0 /* ktap don't support lua length operator '#' */
2295    } else if (ls->tok == '#') {
2296        op = BC_LEN;
2297 #endif
2298    } else {
2299        expr_simple(ls, v);
2300        return;
2301    }
```

```
2302      kp_lex_next(ls);
2303      expr_binop(ls, v, UNARY_PRIORITY);
2304      bcemit_unop(ls->fs, op, v);
2305  }
2306
2307  /* Parse binary expressions with priority higher than the limit. */
2308  static BinOpr expr_binop(LexState *ls, ExpDesc *v, uint32_t limit)
2309  {
2310      BinOpr op;
2311
2312      synlevel_begin(ls);
2313      expr_unop(ls, v);
2314      op = token2binop(ls->tok);
2315      while (op != OPR_NOBINOPR && priority[op].left > limit) {
2316          ExpDesc v2;
2317          BinOpr nextop;
2318          kp_lex_next(ls);
2319          bcemit_binop_left(ls->fs, op, v);
2320          /* Parse binary expression with higher priority. */
2321          nextop = expr_binop(ls, &v2, priority[op].right);
2322          bcemit_binop(ls->fs, op, v, &v2);
2323          op = nextop;
2324      }
2325      synlevel_end(ls);
2326      return op;  /* Return unconsumed binary operator (if any). */
2327  }
2328
2329  /* Parse expression. */
2330  static void expr(LexState *ls, ExpDesc *v)
2331  {
2332      expr_binop(ls, v, 0);  /* Priority 0: parse whole expression. */
2333  }
2334
2335  /* Assign expression to the next register. */
2336  static void expr_next(LexState *ls)
2337  {
2338      ExpDesc e;
2339      expr(ls, &e);
2340      expr_tonextreg(ls->fs, &e);
2341  }
2342
2343  /* Parse conditional expression. */
2344  static BCPos expr_cond(LexState *ls)
2345  {
2346      ExpDesc v;
2347
2348      lex_check(ls, '(');
2349      expr(ls, &v);
2350      if (v.k == VKNIL)
2351          v.k = VKFALSE;
2352      bcemit_branch_t(ls->fs, &v);
2353      lex_check(ls, ')');
2354      return v.f;
2355  }
2356
2357  /* -- Assignments ----------------------------------------------------------- */
2358
2359  /* List of LHS variables. */
2360  typedef struct LHSVarList {
2361      ExpDesc v;            /* LHS variable. */
2362      struct LHSVarList *prev;    /* Link to previous LHS variable. */
2363  } LHSVarList;
2364
2365  /* Eliminate write-after-read hazards for local variable assignment. */
2366  static void assign_hazard(LexState *ls, LHSVarList *lh, const ExpDesc *v)
2367  {
2368      FuncState *fs = ls->fs;
2369      BCReg reg = v->u.s.info; /* Check against this variable. */
2370      BCReg tmp = fs->freereg; /* Rename to this temp. register(if needed) */
2371      int hazard = 0;
2372
2373      for (; lh; lh = lh->prev) {
2374          if (lh->v.k == VINDEXED) {
2375              if (lh->v.u.s.info == reg) {  /* t[i], t = 1, 2 */
2376                  hazard = 1;
2377                  lh->v.u.s.info = tmp;
```

```
            }
            if (lh->v.u.s.aux == reg) {  /* t[i], i = 1, 2 */
                hazard = 1;
                lh->v.u.s.aux = tmp;
            }
        }
    }
    if (hazard) {
        /* Rename conflicting variable. */
        bcemit_AD(fs, BC_MOV, tmp, reg);
        bcreg_reserve(fs, 1);
    }
}

/* Adjust LHS/RHS of an assignment. */
static void assign_adjust(LexState *ls, BCReg nvars, BCReg nexps, ExpDesc *e)
{
    FuncState *fs = ls->fs;
    int32_t extra = (int32_t)nvars - (int32_t)nexps;

    if (e->k == VCALL) {
        extra++;  /* Compensate for the VCALL itself. */
        if (extra < 0)
            extra = 0;
        setbc_b(bcptr(fs, e), extra+1);  /* Fixup call results. */
        if (extra > 1)
            bcreg_reserve(fs, (BCReg)extra-1);
    } else {
        if (e->k != VVOID)
            expr_tonextreg(fs, e);  /* Close last expression. */
        if (extra > 0) {  /* Leftover LHS are set to nil. */
            BCReg reg = fs->freereg;
            bcreg_reserve(fs, (BCReg)extra);
            bcemit_nil(fs, reg, (BCReg)extra);
        }
    }
}

/* Recursively parse assignment statement. */
static void parse_assignment(LexState *ls, LHSVarList *lh, BCReg nvars)
{
    ExpDesc e;

    checkcond(ls, VLOCAL <= lh->v.k && lh->v.k <= VINDEXED,
            KP_ERR_XSYNTAX);
    if (lex_opt(ls, ',')) {  /* Collect LHS list and recurse upwards. */
        LHSVarList vl;
        vl.prev = lh;
        expr_primary(ls, &vl.v);
        if (vl.v.k == VLOCAL)
            assign_hazard(ls, lh, &vl.v);
        checklimit(ls->fs, ls->level + nvars, KP_MAX_XLEVEL,
                "variable names");
        parse_assignment(ls, &vl, nvars+1);
    } else {  /* Parse RHS. */
        BCReg nexps;
        int assign_incr = 1;

        if (lex_opt(ls, '='))
            assign_incr = 0;
        else if (lex_opt(ls, TK_incr))
            assign_incr = 1;
        else
            err_syntax(ls, KP_ERR_XSYMBOL);

        nexps = expr_list(ls, &e);
        if (nexps == nvars) {
            if (e.k == VCALL) {
                /* Vararg assignment. */
                if (bc_op(*bcptr(ls->fs, &e)) == BC_VARG) {
                    ls->fs->freereg--;
                    e.k = VRELOCABLE;
                } else {  /* Multiple call results. */
                    /* Base of call is not relocatable. */
                    e.u.s.info = e.u.s.aux;
                    e.k = VNONRELOC;
```

```
2454                }
2455            }
2456            if (assign_incr == 0)
2457                bcemit_store(ls->fs, &lh->v, &e);
2458            else
2459                bcemit_store_incr(ls->fs, &lh->v, &e);
2460            return;
2461        }
2462        assign_adjust(ls, nvars, nexps, &e);
2463        if (nexps > nvars) {
2464            /* Drop leftover regs. */
2465            ls->fs->freereg -= nexps - nvars;
2466        }
2467    }
2468    /* Assign RHS to LHS and recurse downwards. */
2469    expr_init(&e, VNONRELOC, ls->fs->freereg-1);
2470    bcemit_store(ls->fs, &lh->v, &e);
2471 }
2472
2473 /* Parse call statement or assignment. */
2474 static void parse_call_assign(LexState *ls)
2475 {
2476    FuncState *fs = ls->fs;
2477    LHSVarList vl;
2478
2479    expr_primary(ls, &vl.v);
2480    if (vl.v.k == VCALL) {  /* Function call statement. */
2481        setbc_b(bcptr(fs, &vl.v), 1);  /* No results. */
2482    } else {  /* Start of an assignment. */
2483        vl.prev = NULL;
2484        parse_assignment(ls, &vl, 1);
2485    }
2486 }
2487
2488 /* Parse 'var'(local in lua) statement. */
2489 static void parse_local(LexState *ls)
2490 {
2491    if (lex_opt(ls, TK_function)) {  /* Local function declaration. */
2492        ExpDesc v, b;
2493        FuncState *fs = ls->fs;
2494        var_new(ls, 0, lex_str(ls));
2495        expr_init(&v, VLOCAL, fs->freereg);
2496        v.u.s.aux = fs->varmap[fs->freereg];
2497        bcreg_reserve(fs, 1);
2498        var_add(ls, 1);
2499        parse_body(ls, &b, 0, ls->linenumber);
2500        /* bcemit_store(fs, &v, &b) without setting VSTACK_VAR_RW. */
2501        expr_free(fs, &b);
2502        expr_toreg(fs, &b, v.u.s.info);
2503        /* The upvalue is in scope, but the local is only valid
2504         * after the store. */
2505        var_get(ls, fs, fs->nactvar - 1).startpc = fs->pc;
2506    } else {  /* Local variable declaration. */
2507        ExpDesc e;
2508        BCReg nexps, nvars = 0;
2509        do {  /* Collect LHS. */
2510            var_new(ls, nvars++, lex_str(ls));
2511        } while (lex_opt(ls, ','));
2512        if (lex_opt(ls, '=')) {  /* Optional RHS. */
2513            nexps = expr_list(ls, &e);
2514        } else {  /* Or implicitly set to nil. */
2515            e.k = VVOID;
2516            nexps = 0;
2517        }
2518        assign_adjust(ls, nvars, nexps, &e);
2519        var_add(ls, nvars);
2520    }
2521 }
2522
2523 /* Parse 'function' statement. */
2524 static void parse_func(LexState *ls, BCLine line)
2525 {
2526    FuncState *fs = ls->fs;
2527    ExpDesc v, b;
2528
2529    kp_lex_next(ls);  /* Skip 'function'. */
```

```
2530
2531      /* function is declared as local */
2532  #if 1
2533      var_new(ls, 0, lex_str(ls));
2534      expr_init(&v, VLOCAL, fs->freereg);
2535      v.u.s.aux = fs->varmap[fs->freereg];
2536      bcreg_reserve(fs, 1);
2537      var_add(ls, 1);
2538      parse_body(ls, &b, 0, ls->linenumber);
2539      /* bcemit_store(fs, &v, &b) without setting VSTACK_VAR_RW. */
2540      expr_free(fs, &b);
2541      expr_toreg(fs, &b, v.u.s.info);
2542      /* The upvalue is in scope, but the local is only valid
2543       * after the store. */
2544      var_get(ls, fs, fs->nactvar - 1).startpc = fs->pc;
2545
2546  #else
2547      int needself = 0;
2548
2549      /* Parse function name. */
2550      var_lookup(ls, &v);
2551      while (ls->tok == '.')  /* Multiple dot-separated fields. */
2552          expr_field(ls, &v);
2553      if (ls->tok == ':') {  /* Optional colon to signify method call. */
2554          needself = 1;
2555          expr_field(ls, &v);
2556      }
2557      parse_body(ls, &b, needself, line);
2558      fs = ls->fs;
2559      bcemit_store(fs, &v, &b);
2560      fs->bcbase[fs->pc - 1].line = line;  /* Set line for the store. */
2561  #endif
2562  }
2563
2564  /* -- Control transfer statements ---------------------------------------- */
2565
2566  /* Check for end of block. */
2567  static int parse_isend(LexToken tok)
2568  {
2569      switch (tok) {
2570      case TK_else: case TK_elseif: case TK_end: case TK_until: case TK_eof:
2571      case '}':
2572          return 1;
2573      default:
2574          return 0;
2575      }
2576  }
2577
2578  /* Parse 'return' statement. */
2579  static void parse_return(LexState *ls)
2580  {
2581      BCIns ins;
2582      FuncState *fs = ls->fs;
2583
2584      kp_lex_next(ls);  /* Skip 'return'. */
2585      fs->flags |= PROTO_HAS_RETURN;
2586      if (parse_isend(ls->tok) || ls->tok == ';') {  /* Bare return. */
2587          ins = BCINS_AD(BC_RET0, 0, 1);
2588      } else {  /* Return with one or more values. */
2589          ExpDesc e;  /* Receives the _last_ expression in the list. */
2590          BCReg nret = expr_list(ls, &e);
2591          if (nret == 1) {  /* Return one result. */
2592              if (e.k == VCALL) {  /* Check for tail call. */
2593                  BCIns *ip = bcptr(fs, &e);
2594                  /* It doesn't pay off to add BC_VARGT just
2595                   * for 'return ...'. */
2596                  if (bc_op(*ip) == BC_VARG)
2597                      goto notailcall;
2598                  fs->pc--;
2599                  ins = BCINS_AD(bc_op(*ip)-BC_CALL+BC_CALLT,
2600                                 bc_a(*ip), bc_c(*ip));
2601              } else { /* Can return the result from any register. */
2602                  ins = BCINS_AD(BC_RET1,
2603                      expr_toanyreg(fs, &e), 2);
2604              }
2605          } else {
```

```
            if (e.k == VCALL) {/* Append all results from a call */
notailcall:
                setbc_b(bcptr(fs, &e), 0);
                ins = BCINS_AD(BC_RETM, fs->nactvar,
                       e.u.s.aux - fs->nactvar);
            } else {
                /* Force contiguous registers. */
                expr_tonextreg(fs, &e);
                ins = BCINS_AD(BC_RET, fs->nactvar, nret+1);
            }
        }
    }
    if (fs->flags & PROTO_CHILD) {
        /* May need to close upvalues first. */
        bcemit_AJ(fs, BC_UCLO, 0, 0);
    }
    bcemit_INS(fs, ins);
}

/* Parse 'break' statement. */
static void parse_break(LexState *ls)
{
    ls->fs->bl->flags |= FSCOPE_BREAK;
    gola_new(ls, NAME_BREAK, VSTACK_GOTO, bcemit_jmp(ls->fs));
}

/* Parse label. */
static void parse_label(LexState *ls)
{
    FuncState *fs = ls->fs;
    ktap_str_t *name;
    int idx;

    fs->lasttarget = fs->pc;
    fs->bl->flags |= FSCOPE_GOLA;
    kp_lex_next(ls);  /* Skip '::'. */
    name = lex_str(ls);
    if (gola_findlabel(ls, name))
        kp_lex_error(ls, 0, KP_ERR_XLDUP, getstr(name));
    idx = gola_new(ls, name, VSTACK_LABEL, fs->pc);
    lex_check(ls, TK_label);
    /* Recursively parse trailing statements: labels and ';'. */
    for (;;) {
        if (ls->tok == TK_label) {
            synlevel_begin(ls);
            parse_label(ls);
            synlevel_end(ls);
        } else if (ls->tok == ';') {
            kp_lex_next(ls);
        } else {
            break;
        }
    }
    /* Trailing label is considered to be outside of scope. */
    if (parse_isend(ls->tok) && ls->tok != TK_until)
        ls->vstack[idx].slot = fs->bl->nactvar;
    gola_resolve(ls, fs->bl, idx);
}

/* -- Blocks, loops and conditional statements ---------------------------- */

/* Parse a block. */
static void parse_block(LexState *ls)
{
    FuncState *fs = ls->fs;
    FuncScope bl;

    fscope_begin(fs, &bl, 0);
    parse_chunk(ls);
    fscope_end(fs);
}

/* Parse 'while' statement. */
static void parse_while(LexState *ls, BCLine line)
{
    FuncState *fs = ls->fs;
```

```
2682      BCPos start, loop, condexit;
2683      FuncScope bl;
2684
2685      kp_lex_next(ls);  /* Skip 'while'. */
2686      start = fs->lasttarget = fs->pc;
2687      condexit = expr_cond(ls);
2688      fscope_begin(fs, &bl, FSCOPE_LOOP);
2689      //lex_check(ls, TK_do);
2690      lex_check(ls, '{');
2691      loop = bcemit_AD(fs, BC_LOOP, fs->nactvar, 0);
2692      parse_block(ls);
2693      jmp_patch(fs, bcemit_jmp(fs), start);
2694      //lex_match(ls, TK_end, TK_while, line);
2695      lex_check(ls, '}');
2696      fscope_end(fs);
2697      jmp_tohere(fs, condexit);
2698      jmp_patchins(fs, loop, fs->pc);
2699  }
2700
2701  /* Parse 'repeat' statement. */
2702  static void parse_repeat(LexState *ls, BCLine line)
2703  {
2704      FuncState *fs = ls->fs;
2705      BCPos loop = fs->lasttarget = fs->pc;
2706      BCPos condexit;
2707      FuncScope bl1, bl2;
2708
2709      fscope_begin(fs, &bl1, FSCOPE_LOOP);  /* Breakable loop scope. */
2710      fscope_begin(fs, &bl2, 0);  /* Inner scope. */
2711      kp_lex_next(ls);  /* Skip 'repeat'. */
2712      bcemit_AD(fs, BC_LOOP, fs->nactvar, 0);
2713      parse_chunk(ls);
2714      lex_match(ls, TK_until, TK_repeat, line);
2715      /* Parse condition (still inside inner scope). */
2716      condexit = expr_cond(ls);
2717      /* No upvalues? Just end inner scope. */
2718      if (!(bl2.flags & FSCOPE_UPVAL)) {
2719          fscope_end(fs);
2720      } else {
2721          /* Otherwise generate: cond: UCLO+JMP out,
2722           * !cond: UCLO+JMP loop. */
2723          parse_break(ls);  /* Break from loop and close upvalues. */
2724          jmp_tohere(fs, condexit);
2725          fscope_end(fs);  /* End inner scope and close upvalues. */
2726          condexit = bcemit_jmp(fs);
2727      }
2728      jmp_patch(fs, condexit, loop);  /* Jump backwards if !cond. */
2729      jmp_patchins(fs, loop, fs->pc);
2730      fscope_end(fs);  /* End loop scope. */
2731  }
2732
2733  /* Parse numeric 'for'. */
2734  static void parse_for_num(LexState *ls, ktap_str_t *varname, BCLine line)
2735  {
2736      FuncState *fs = ls->fs;
2737      BCReg base = fs->freereg;
2738      FuncScope bl;
2739      BCPos loop, loopend;
2740
2741      /* Hidden control variables. */
2742      var_new_fixed(ls, FORL_IDX, VARNAME_FOR_IDX);
2743      var_new_fixed(ls, FORL_STOP, VARNAME_FOR_STOP);
2744      var_new_fixed(ls, FORL_STEP, VARNAME_FOR_STEP);
2745      /* Visible copy of index variable. */
2746      var_new(ls, FORL_EXT, varname);
2747      lex_check(ls, '=');
2748      expr_next(ls);
2749      lex_check(ls, ',');
2750      expr_next(ls);
2751      if (lex_opt(ls, ',')) {
2752          expr_next(ls);
2753      } else {
2754          /* Default step is 1. */
2755          bcemit_AD(fs, BC_KSHORT, fs->freereg, 1);
2756          bcreg_reserve(fs, 1);
2757      }
```

```
2758      var_add(ls, 3);  /* Hidden control variables. */
2759      //lex_check(ls, TK_do);
2760      lex_check(ls, ')');
2761      lex_check(ls, '{');
2762      loop = bcemit_AJ(fs, BC_FORI, base, NO_JMP);
2763      fscope_begin(fs, &bl, 0);  /* Scope for visible variables. */
2764      var_add(ls, 1);
2765      bcreg_reserve(fs, 1);
2766      parse_block(ls);
2767      fscope_end(fs);
2768      /* Perform loop inversion. Loop control instructions are at the end. */
2769      loopend = bcemit_AJ(fs, BC_FORL, base, NO_JMP);
2770      fs->bcbase[loopend].line = line;  /* Fix line for control ins. */
2771      jmp_patchins(fs, loopend, loop+1);
2772      jmp_patchins(fs, loop, fs->pc);
2773  }
2774
2775  /*
2776  * Try to predict whether the iterator is next() and specialize the bytecode.
2777  * Detecting next() and pairs() by name is simplistic, but quite effective.
2778  * The interpreter backs off if the check for the closure fails at runtime.
2779  */
2780  static int predict_next(LexState *ls, FuncState *fs, BCPos pc)
2781  {
2782      BCIns ins = fs->bcbase[pc].ins;
2783      ktap_str_t *name;
2784      const ktap_val_t *o;
2785
2786      switch (bc_op(ins)) {
2787      case BC_MOV:
2788          name = var_get(ls, fs, bc_d(ins)).name;
2789          break;
2790      case BC_UGET:
2791          name = ls->vstack[fs->uvmap[bc_d(ins)]].name;
2792          break;
2793      case BC_GGET:
2794          /* There's no inverse index (yet), so lookup the strings. */
2795          o = kp_tab_getstr(fs->kt, kp_str_newz("pairs"));
2796          if (o && tvhaskslot(o) && tvkslot(o) == bc_d(ins))
2797              return 1;
2798          o = kp_tab_getstr(fs->kt, kp_str_newz("next"));
2799          if (o && tvhaskslot(o) && tvkslot(o) == bc_d(ins))
2800              return 1;
2801          return 0;
2802      default:
2803          return 0;
2804      }
2805
2806      return (name->len == 5 && !strcmp(getstr(name), "pairs")) ||
2807          (name->len == 4 && !strcmp(getstr(name), "next"));
2808  }
2809
2810  /* Parse 'for' iterator. */
2811  static void parse_for_iter(LexState *ls, ktap_str_t *indexname)
2812  {
2813      FuncState *fs = ls->fs;
2814      ExpDesc e;
2815      BCReg nvars = 0;
2816      BCLine line;
2817      BCReg base = fs->freereg + 3;
2818      BCPos loop, loopend, exprpc = fs->pc;
2819      FuncScope bl;
2820      int isnext;
2821
2822      /* Hidden control variables. */
2823      var_new_fixed(ls, nvars++, VARNAME_FOR_GEN);
2824      var_new_fixed(ls, nvars++, VARNAME_FOR_STATE);
2825      var_new_fixed(ls, nvars++, VARNAME_FOR_CTL);
2826
2827      /* Visible variables returned from iterator. */
2828      var_new(ls, nvars++, indexname);
2829      while (lex_opt(ls, ','))
2830          var_new(ls, nvars++, lex_str(ls));
2831      lex_check(ls, TK_in);
2832      line = ls->linenumber;
2833      assign_adjust(ls, 3, expr_list(ls, &e), &e);
```

```
    /* The iterator needs another 3 slots (func + 2 args). */
    bcreg_bump(fs, 3);
    isnext = (nvars <= 5 && predict_next(ls, fs, exprpc));
    var_add(ls, 3);  /* Hidden control variables. */
    //lex_check(ls, TK_do);
    lex_check(ls, ')');
    lex_check(ls, '{');
    loop = bcemit_AJ(fs, isnext ? BC_ISNEXT : BC_JMP, base, NO_JMP);
    fscope_begin(fs, &bl, 0);  /* Scope for visible variables. */
    var_add(ls, nvars-3);
    bcreg_reserve(fs, nvars-3);
    parse_block(ls);
    fscope_end(fs);
    /* Perform loop inversion. Loop control instructions are at the end. */
    jmp_patchins(fs, loop, fs->pc);
    bcemit_ABC(fs, isnext ? BC_ITERN : BC_ITERC, base, nvars-3+1, 2+1);
    loopend = bcemit_AJ(fs, BC_ITERL, base, NO_JMP);
    fs->bcbase[loopend-1].line = line;  /* Fix line for control ins. */
    fs->bcbase[loopend].line = line;
    jmp_patchins(fs, loopend, loop+1);
}

/* Parse 'for' statement. */
static void parse_for(LexState *ls, BCLine line)
{
    FuncState *fs = ls->fs;
    ktap_str_t *varname;
    FuncScope bl;

    fscope_begin(fs, &bl, FSCOPE_LOOP);
    kp_lex_next(ls);  /* Skip 'for'. */
    lex_check(ls, '(');
    varname = lex_str(ls);  /* Get first variable name. */
    if (ls->tok == '=')
        parse_for_num(ls, varname, line);
    else if (ls->tok == ',' || ls->tok == TK_in)
        parse_for_iter(ls, varname);
    else
        err_syntax(ls, KP_ERR_XFOR);
    //lex_check(ls, '}');
    //lex_match(ls, TK_end, TK_for, line);
    lex_match(ls, '}', TK_for, line);
    fscope_end(fs);  /* Resolve break list. */
}

/* Parse condition and 'then' block. */
static BCPos parse_then(LexState *ls)
{
    BCPos condexit;
    kp_lex_next(ls);  /* Skip 'if' or 'elseif'. */
    condexit = expr_cond(ls);
    lex_check(ls, '{');
    parse_block(ls);
    lex_check(ls, '}');
    return condexit;
}

/* Parse 'if' statement. */
static void parse_if(LexState *ls, BCLine line)
{
    FuncState *fs = ls->fs;
    BCPos flist;
    BCPos escapelist = NO_JMP;
    flist = parse_then(ls);
    while (ls->tok == TK_elseif) {  /* Parse multiple 'elseif' blocks. */
        jmp_append(fs, &escapelist, bcemit_jmp(fs));
        jmp_tohere(fs, flist);
        flist = parse_then(ls);
    }
    if (ls->tok == TK_else) {  /* Parse optional 'else' block. */
        jmp_append(fs, &escapelist, bcemit_jmp(fs));
        jmp_tohere(fs, flist);
        kp_lex_next(ls);  /* Skip 'else'. */
        lex_check(ls, '{');
        parse_block(ls);
        lex_check(ls, '}');
```

```
2910      } else {
2911          jmp_append(fs, &escapelist, flist);
2912      }
2913      jmp_tohere(fs, escapelist);
2914      //lex_match(ls, TK_end, TK_if, line);
2915  }
2916
2917  /* Parse 'trace' and 'trace_end' statement. */
2918  static void parse_trace(LexState *ls)
2919  {
2920      ExpDesc v, key, args;
2921      ktap_str_t *kdebug_str = kp_str_newz("kdebug");
2922      ktap_str_t *probe_str = kp_str_newz("trace_by_id");
2923      ktap_str_t *probe_end_str = kp_str_newz("trace_end");
2924      FuncState *fs = ls->fs;
2925      int token = ls->tok;
2926      BCIns ins;
2927      BCReg base;
2928      BCLine line = ls->linenumber;
2929
2930      if (token == TK_trace)
2931          kp_lex_read_string_until(ls, '{');
2932      else
2933          kp_lex_next(ls);  /* skip "trace_end" keyword */
2934
2935      /* kdebug */
2936      expr_init(&v, VGLOBAL, 0);
2937      v.u.sval = kdebug_str;
2938      expr_toanyreg(fs, &v);
2939
2940      /* fieldsel: kdebug.probe */
2941      expr_init(&key, VKSTR, 0);
2942      key.u.sval = token == TK_trace ? probe_str : probe_end_str;
2943      expr_index(fs, &v, &key);
2944
2945      /* funcargs*/
2946      expr_tonextreg(fs, &v);
2947
2948      if (token == TK_trace) {
2949          ktap_eventdesc_t *evdef_info;
2950          const char *str;
2951
2952          /* argument: EVENTDEF string */
2953          lex_check(ls, TK_string);
2954          str = svalue(&ls->tokval);
2955          evdef_info = kp_parse_events(str);
2956          if (!evdef_info)
2957              kp_lex_error(ls, 0, KP_ERR_XEVENTDEF, str);
2958
2959
2960          /* pass a userspace pointer to kernel */
2961          expr_init(&args, VKNUM, 0);
2962          set_number(&args.u.nval, (ktap_number)evdef_info);
2963
2964          expr_tonextreg(fs, &args);
2965      }
2966
2967      /* argument: callback function */
2968      parse_body_no_args(ls, &args, 0, ls->linenumber);
2969
2970      expr_tonextreg(fs, &args);
2971
2972      base = v.u.s.info;  /* base register for call */
2973      ins = BCINS_ABC(BC_CALL, base, 2, fs->freereg - base);
2974
2975      expr_init(&v, VCALL, bcemit_INS(fs, ins));
2976      v.u.s.aux = base;
2977      fs->bcbase[fs->pc - 1].line = line;
2978      fs->freereg = base+1;  /* Leave one result by default. */
2979
2980      setbc_b(bcptr(fs, &v), 1);  /* No results. */
2981  }
2982
2983
2984  /* Parse 'profile' and 'tick' statement. */
2985  static void parse_timer(LexState *ls)
```

```
{
    FuncState *fs = ls->fs;
    ExpDesc v, key, args;
    ktap_str_t *token_str = rawtsvalue(&ls->tokval);
    ktap_str_t *interval_str;
    BCLine line = ls->linenumber;
    BCIns ins;
    BCReg base;

    kp_lex_next(ls);  /* skip '-' */

    kp_lex_read_string_until(ls, '{');
    interval_str = rawtsvalue(&ls->tokval);
    lex_check(ls, TK_string);

    /* timer */
    expr_init(&v, VGLOBAL, 0);
    v.u.sval = kp_str_newz("timer");
    expr_toanyreg(fs, &v);

    /* fieldsel: timer.profile, timer.tick */
    expr_init(&key, VKSTR, 0);
    key.u.sval = token_str;
    expr_index(fs, &v, &key);

    /* funcargs*/
    expr_tonextreg(fs, &v);

    /* argument: interval string */
    expr_init(&args, VKSTR, 0);
    args.u.sval = interval_str;

    expr_tonextreg(fs, &args);

    /* argument: callback function */
    parse_body_no_args(ls, &args, 0, ls->linenumber);

    expr_tonextreg(fs, &args);

    base = v.u.s.info;  /* base register for call */
    ins = BCINS_ABC(BC_CALL, base, 2, fs->freereg - base);

    expr_init(&v, VCALL, bcemit_INS(fs, ins));
    v.u.s.aux = base;
    fs->bcbase[fs->pc - 1].line = line;
    fs->freereg = base+1;  /* Leave one result by default. */

    setbc_b(bcptr(fs, &v), 1);  /* No results. */
}

/* -- Parse statements ---------------------------------------------- */

/* Parse a statement. Returns 1 if it must be the last one in a chunk. */
static int parse_stmt(LexState *ls)
{
    BCLine line = ls->linenumber;
    switch (ls->tok) {
    case TK_if:
        parse_if(ls, line);
        break;
    case TK_while:
        parse_while(ls, line);
        break;
    case TK_do:
        kp_lex_next(ls);
        parse_block(ls);
        lex_match(ls, TK_end, TK_do, line);
        break;
    case TK_for:
        parse_for(ls, line);
        break;
    case TK_repeat:
        parse_repeat(ls, line);
        break;
    case TK_function:
        parse_func(ls, line);
```

```
3062        break;
3063    case TK_local:
3064        kp_lex_next(ls);
3065        parse_local(ls);
3066        break;
3067    case TK_return:
3068        parse_return(ls);
3069        return 1;  /* Must be last. */
3070    case TK_break:
3071        kp_lex_next(ls);
3072        parse_break(ls);
3073        return 0;  /* Must be last. */
3074    case ';':
3075        kp_lex_next(ls);
3076        break;
3077    case TK_label:
3078        parse_label(ls);
3079        break;
3080    case TK_trace:
3081    case TK_trace_end:
3082        parse_trace(ls);
3083        break;
3084    case TK_profile:
3085    case TK_tick:
3086        parse_timer(ls);
3087        break;
3088    default:
3089        parse_call_assign(ls);
3090        break;
3091    }
3092    return 0;
3093 }
3094
3095 /* A chunk is a list of statements optionally separated by semicolons. */
3096 static void parse_chunk(LexState *ls)
3097 {
3098    int islast = 0;
3099
3100    synlevel_begin(ls);
3101    while (!islast && !parse_isend(ls->tok)) {
3102        islast = parse_stmt(ls);
3103        lex_opt(ls, ';');
3104        kp_assert(ls->fs->framesize >= ls->fs->freereg &&
3105            ls->fs->freereg >= ls->fs->nactvar);
3106        /* Free registers after each stmt. */
3107        ls->fs->freereg = ls->fs->nactvar;
3108    }
3109    synlevel_end(ls);
3110 }
3111
3112 /* Entry point of bytecode parser. */
3113 ktap_proto_t *kp_parse(LexState *ls)
3114 {
3115    FuncState fs;
3116    FuncScope bl;
3117    ktap_proto_t *pt;
3118
3119    ls->chunkname = kp_str_newz(ls->chunkarg);
3120    ls->level = 0;
3121    fs_init(ls, &fs);
3122    fs.linedefined = 0;
3123    fs.numparams = 0;
3124    fs.bcbase = NULL;
3125    fs.bclim = 0;
3126    fs.flags |= PROTO_VARARG;  /* Main chunk is always a vararg func. */
3127    fscope_begin(&fs, &bl, 0);
3128    bcemit_AD(&fs, BC_FUNCV, 0, 0);  /* Placeholder. */
3129    kp_lex_next(ls);  /* Read-ahead first token. */
3130    parse_chunk(ls);
3131    if (ls->tok != TK_eof)
3132        err_token(ls, TK_eof);
3133    pt = fs_finish(ls, ls->linenumber);
3134    kp_assert(fs.prev == NULL);
3135    kp_assert(ls->fs == NULL);
3136    kp_assert(pt->sizeuv == 0);
3137    return pt;
```

```
3138 }
3139
```

One Level Up Top Level

userspace/kp_lex.h - ktap

Data types defined

- BCInsLine
- BCInsLine
- LexChar
- LexState
- LexState
- LexToken
- VarInfo
- VarInfo

Macros defined

- TKDEF
- TKENUM1
- TKENUM1
- TKENUM2
- TKENUM2
- _KTAP_LEX_H

Source code

```
1  /*
2  * Lexical analyzer.
3  *
4  * Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
5  *
6  * Adapted from luajit and lua interpreter.
7  * Copyright (C) 2005-2014 Mike Pall.
8  * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
9  */
10
11 #ifndef _KTAP_LEX_H
12 #define _KTAP_LEX_H
13
14 #include <stdarg.h>
15 #include "../include/err.h"
16 #include "../include/ktap_bc.h"
17 #include "kp_util.h"
18
19 /* ktap lexer tokens. */
20 #define TKDEF(_, __) \
21     _(trace) _(trace_end) _(argstr) _(probename) _(ffi) \
22     _(arg0)_(arg1) _(arg2) _(arg3) _(arg4) _(arg5) _(arg6) _(arg7) \
23     _(arg8) _(arg9) _(profile) _(tick) \
24     _(pid) _(tid) _(uid) _(cpu) _(execname) __(incr, +=) \
25     __(and, &&) _(break) _(do) _(else) _(elseif) _(end) _(false) \
26     _(for) _(function) _(goto) _(if) _(in) __(local, var) _(nil) \
27     __(not, !) __(or, ||) \
28     _(repeat) _(return) _(then) _(true) _(until) _(while) \
29     __(concat, ..) __(dots, ...) __(eq, ==) __(ge, >=) __(le, <=) \
```

```
    __(ne, !=) __(label, ::) __(number, <number>) __(name, <name>) \
    __(string, <string>) __(eof, <eof>)

enum {
    TK_OFS = 256,
#define TKENUM1(name)        TK_##name,
#define TKENUM2(name, sym)    TK_##name,
    TKDEF(TKENUM1, TKENUM2)
#undef TKENUM1
#undef TKENUM2
    TK_RESERVED = TK_while - TK_OFS
};

typedef int LexChar;    /* Lexical character. Unsigned ext. from char. */
typedef int LexToken;    /* Lexical token. */

/* Combined bytecode ins/line. Only used during bytecode generation. */
typedef struct BCInsLine {
    BCIns ins;        /* Bytecode instruction. */
    BCLine line;        /* Line number for this bytecode. */
} BCInsLine;

/* Info for local variables. Only used during bytecode generation. */
typedef struct VarInfo {
    ktap_str_t *name;    /* Local variable name or goto/label name. */
    BCPos startpc;    /* First point where the local variable is active. */
    BCPos endpc;    /* First point where the local variable is dead. */
    uint8_t slot;    /* Variable slot. */
    uint8_t info;    /* Variable/goto/label info. */
} VarInfo;

/* lexer state. */
typedef struct LexState {
    struct FuncState *fs;    /* Current FuncState. Defined in kp_parse.c. */
    ktap_val_t tokval;    /* Current token value. */
    ktap_val_t lookaheadval;/* Lookahead token value. */
    const char *p;    /* Current position in input buffer. */
    const char *pe;    /* End of input buffer. */
    LexChar c;        /* Current character. */
    LexToken tok;        /* Current token. */
    LexToken lookahead;    /* Lookahead token. */
    SBuf sb;        /* String buffer for tokens. */
    BCLine linenumber;    /* Input line counter. */
    BCLine lastline;    /* Line of last token. */
    ktap_str_t *chunkname;/* Current chunk name (interned string). */
    const char *chunkarg;    /* Chunk name argument. */
    const char *mode;/* Allow loading bytecode (b) and/or source text (t) */
    VarInfo *vstack;/* Stack for names and extents of local variables. */
    int sizevstack;    /* Size of variable stack. */
    int vtop;    /* Top of variable stack. */
    BCInsLine *bcstack;/* Stack for bytecode instructions/line numbers. */
    int sizebcstack;/* Size of bytecode stack. */
    uint32_t level;    /* Syntactical nesting level. */
} LexState;

int kp_lex_setup(LexState *ls, const char *str);
void kp_lex_cleanup(LexState *ls);
void kp_lex_next(LexState *ls);
void kp_lex_read_string_until(LexState *ls, int c);
LexToken kp_lex_lookahead(LexState *ls);
const char *kp_lex_token2str(LexState *ls, LexToken tok);
void kp_lex_error(LexState *ls, LexToken tok, ErrMsg em, ...);
void kp_lex_init(void);

#endif
```

One Level Up Top Level

userspace/kp_lex.c - ktap

Global variables defined

- tokennames

Functions defined

- kp_lex_cleanup
- kp_lex_error
- kp_lex_init
- kp_lex_lookahead
- kp_lex_next
- kp_lex_read_string_until
- kp_lex_setup
- kp_lex_token2str
- kp_str2d
- lex_longstring
- lex_newline
- lex_next
- lex_number
- lex_save
- lex_savenext
- lex_scan
- lex_skipeq
- lex_string

Macros defined

- LEX_EOF
- TKSTR1
- TKSTR1
- TKSTR2
- TKSTR2
- lex_iseol

Source code

```
1 /*
```

```
 * Lexical analyzer.
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * Adapted from luajit and lua interpreter.
 * Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "../include/ktap_types.h"
#include "../include/ktap_err.h"
#include "kp_util.h"
#include "kp_lex.h"
#include "kp_parse.h"

/* lexer token names. */
static const char *const tokennames[] = {
#define TKSTR1(name)        #name,
#define TKSTR2(name, sym)    #sym,
TKDEF(TKSTR1, TKSTR2)
#undef TKSTR1
#undef TKSTR2
  NULL
};

/* -- Buffer handling ----------------------------------------------------- */

#define LEX_EOF            (-1)
#define lex_iseol(ls)        (ls->c == '\n' || ls->c == '\r')

/* Get next character. */
static inline LexChar lex_next(LexState *ls)
{
    return (ls->c = ls->p < ls->pe ? (LexChar)(uint8_t)*ls->p++ : LEX_EOF);
}

/* Save character. */
static inline void lex_save(LexState *ls, LexChar c)
{
    kp_buf_putb(&ls->sb, c);
}

/* Save previous character and get next character. */
static inline LexChar lex_savenext(LexState *ls)
{
    lex_save(ls, ls->c);
    return lex_next(ls);
}

/* Skip line break. Handles "\n", "\r", "\r\n" or "\n\r". */
static void lex_newline(LexState *ls)
{
    LexChar old = ls->c;

    kp_assert(lex_iseol(ls));
    lex_next(ls);  /* Skip "\n" or "\r". */
    if (lex_iseol(ls) && ls->c != old)
        lex_next(ls);  /* Skip "\n\r" or "\r\n". */
    if (++ls->linenumber >= KP_MAX_LINE)
        kp_lex_error(ls, ls->tok, KP_ERR_XLINES);
}
```

```
78
79  /* -- Scanner for terminals ---------------------------------------------- */
80
81  static int kp_str2d(const char *s, size_t len, ktap_number *result)
82  {
83      char *endptr;
84
85      if (strpbrk(s, "nN"))  /* reject 'inf' and 'nan' */
86          return 0;
87      else
88          *result = (long)strtoul(s, &endptr, 0);
89
90      if (endptr == s)
91          return 0;  /* nothing recognized */
92      while (kp_char_isspace((unsigned char)(*endptr)))
93          endptr++;
94      return (endptr == s + len);  /* OK if no trailing characters */
95  }
96
97
98  /* Parse a number literal. */
99  static void lex_number(LexState *ls, ktap_val_t *tv)
100 {
101     LexChar c, xp = 'e';
102     ktap_number n = 0;
103
104     kp_assert(kp_char_isdigit(ls->c));
105     if ((c = ls->c) == '0' && (lex_savenext(ls) | 0x20) == 'x')
106         xp = 'p';
107     while (kp_char_isident(ls->c) || ls->c == '.' ||
108         ((ls->c == '-' || ls->c == '+') && (c | 0x20) == xp)) {
109         c = ls->c;
110         lex_savenext(ls);
111     }
112     lex_save(ls, '\0');
113     if (!kp_str2d(sbufB(&ls->sb), sbuflen(&ls->sb) - 1, &n))
114             kp_lex_error(ls, ls->tok, KP_ERR_XNUMBER);
115     set_number(tv, n);
116 }
117
118 /* Skip equal signs for "[=...=[" and "]=...=]" and return their count. */
119 static int lex_skipeq(LexState *ls)
120 {
121     int count = 0;
122     LexChar s = ls->c;
123
124     kp_assert(s == '[' || s == ']');
125     while (lex_savenext(ls) == '=')
126         count++;
127     return (ls->c == s) ? count : (-count) - 1;
128 }
129
130 /* Parse a long string or long comment (tv set to NULL). */
131 static void lex_longstring(LexState *ls, ktap_val_t *tv, int sep)
132 {
133     lex_savenext(ls);  /* Skip second '['. */
134     if (lex_iseol(ls))  /* Skip initial newline. */
135         lex_newline(ls);
136     for (;;) {
137         switch (ls->c) {
138         case LEX_EOF:
139             kp_lex_error(ls, TK_eof,
140                     tv ? KP_ERR_XLSTR : KP_ERR_XLCOM);
141             break;
142         case ']':
143             if (lex_skipeq(ls) == sep) {
144                 lex_savenext(ls);  /* Skip second ']'. */
145                 goto endloop;
146             }
147             break;
148         case '\n':
149         case '\r':
150             lex_save(ls, '\n');
151             lex_newline(ls);
152             if (!tv) /* Don't waste space for comments. */
153                 kp_buf_reset(&ls->sb);
```

```
            break;
        default:
            lex_savenext(ls);
            break;
        }
    }
endloop:
    if (tv) {
        ktap_str_t *str = kp_parse_keepstr(ls,
                    sbufB(&ls->sb) + (2 + (int)sep),
                    sbuflen(&ls->sb) - 2*(2 + (int)sep));
        set_string(tv, str);
    }
}

/* Parse a string. */
static void lex_string(LexState *ls, ktap_val_t *tv)
{
    LexChar delim = ls->c;  /* Delimiter is '\'' or '"'. */

    lex_savenext(ls);
    while (ls->c != delim) {
        switch (ls->c) {
        case LEX_EOF:
            kp_lex_error(ls, TK_eof, KP_ERR_XSTR);
            continue;
        case '\n':
        case '\r':
            kp_lex_error(ls, TK_string, KP_ERR_XSTR);
            continue;
        case '\\': {
            LexChar c = lex_next(ls);  /* Skip the '\\'. */
            switch (c) {
            case 'a': c = '\a'; break;
            case 'b': c = '\b'; break;
            case 'f': c = '\f'; break;
            case 'n': c = '\n'; break;
            case 'r': c = '\r'; break;
            case 't': c = '\t'; break;
            case 'v': c = '\v'; break;
            case 'x':  /* Hexadecimal escape '\xXX'. */
                c = (lex_next(ls) & 15u) << 4;
                if (!kp_char_isdigit(ls->c)) {
                    if (!kp_char_isxdigit(ls->c))
                        goto err_xesc;
                    c += 9 << 4;
                }
                c += (lex_next(ls) & 15u);
                if (!kp_char_isdigit(ls->c)) {
                    if (!kp_char_isxdigit(ls->c))
                        goto err_xesc;
                    c += 9;
                }
                break;
            case 'z':  /* Skip whitespace. */
                lex_next(ls);
                while (kp_char_isspace(ls->c))
                    if (lex_iseol(ls))
                        lex_newline(ls);
                    else
                        lex_next(ls);
                continue;
            case '\n': case '\r':
                lex_save(ls, '\n');
                lex_newline(ls);
                continue;
            case '\\': case '\"': case '\'':
                break;
            case LEX_EOF:
                continue;
            default:
                if (!kp_char_isdigit(c))
                    goto err_xesc;
                c -= '0';  /* Decimal escape '\ddd'. */
                if (kp_char_isdigit(lex_next(ls))) {
                    c = c*10 + (ls->c - '0');
```

```
                    if (kp_char_isdigit(lex_next(ls))) {
                        c = c*10 + (ls->c - '0');
                        if (c > 255) {
err_xesc:
                            kp_lex_error(ls,
                                TK_string,
                                KP_ERR_XESC);
                        }
                        lex_next(ls);
                    }
                }
                lex_save(ls, c);
                continue;
            }
            lex_save(ls, c);
            lex_next(ls);
            continue;
        }
        default:
            lex_savenext(ls);
            break;
        }
    }
    lex_savenext(ls);  /* Skip trailing delimiter. */
    set_string(tv,
        kp_parse_keepstr(ls, sbufB(&ls->sb)+1, sbuflen(&ls->sb)-2));
}

/* lex helper for parse_trace and parse_timer */
void kp_lex_read_string_until(LexState *ls, int c)
{
    ktap_str_t *ts;

    kp_buf_reset(&ls->sb);

    while (ls->c == ' ')
        lex_next(ls);

    do {
        lex_savenext(ls);
    } while (ls->c != c && ls->c != LEX_EOF);

    if (ls->c != c)
        kp_lex_error(ls, ls->tok, KP_ERR_XTOKEN, c);

    ts = kp_parse_keepstr(ls, sbufB(&ls->sb), sbuflen(&ls->sb));
    ls->tok = TK_string;
    set_string(&ls->tokval, ts);
}


/* -- Main lexical scanner ---------------------------------------------- */

/* Get next lexical token. */
static LexToken lex_scan(LexState *ls, ktap_val_t *tv)
{
    kp_buf_reset(&ls->sb);
    for (;;) {
        if (kp_char_isident(ls->c)) {
            ktap_str_t *s;
            if (kp_char_isdigit(ls->c)) {  /* Numeric literal. */
                lex_number(ls, tv);
                return TK_number;
            }
            /* Identifier or reserved word. */
            do {
                lex_savenext(ls);
            } while (kp_char_isident(ls->c));
            s = kp_parse_keepstr(ls, sbufB(&ls->sb),
                        sbuflen(&ls->sb));
            set_string(tv, s);
            if (s->reserved > 0)  /* Reserved word? */
                return TK_OFS + s->reserved;
            return TK_name;
        }

```

```
        switch (ls->c) {
        case '\n':
        case '\r':
            lex_newline(ls);
            continue;
        case ' ':
        case '\t':
        case '\v':
        case '\f':
            lex_next(ls);
            continue;

        case '#':
            while (!lex_iseol(ls) && ls->c != LEX_EOF)
                lex_next(ls);
            break;
        case '-':
            lex_next(ls);
            if (ls->c != '-')
                return '-';
            lex_next(ls);
            if (ls->c == '[') { /* Long comment "--[=*[...]=*]". */
                int sep = lex_skipeq(ls);
                /* `lex_skipeq' may dirty the buffer */
                kp_buf_reset(&ls->sb);
                if (sep >= 0) {
                    lex_longstring(ls, NULL, sep);
                    kp_buf_reset(&ls->sb);
                    continue;
                }
            }
            /* Short comment "--.*\n". */
            while (!lex_iseol(ls) && ls->c != LEX_EOF)
                lex_next(ls);
            continue;
        case '[': {
            int sep = lex_skipeq(ls);
            if (sep >= 0) {
                lex_longstring(ls, tv, sep);
                return TK_string;
            } else if (sep == -1) {
                return '[';
            } else {
                kp_lex_error(ls, TK_string, KP_ERR_XLDELIM);
                continue;
            }
        }
        case '+': {
            lex_next(ls);
            if (ls->c != '=')
                return '+';
            else {
                lex_next(ls);
                return TK_incr;
            }
        }
        case '=':
            lex_next(ls);
            if (ls->c != '=')
                return '=';
            else {
                lex_next(ls);
                return TK_eq;
            }
        case '<':
            lex_next(ls);
            if (ls->c != '=')
                return '<';
            else {
                lex_next(ls);
                return TK_le;
            }
        case '>':
            lex_next(ls);
            if (ls->c != '=')
                return '>';
```

```
            else {
                lex_next(ls);
                return TK_ge;
            }
        case '!':
                  lex_next(ls);
            if (ls->c != '=')
                return TK_not;
            else {
                lex_next(ls);
                return TK_ne;
            }
        case ':':
            lex_next(ls);
            if (ls->c != ':')
                return ':';
            else {
                lex_next(ls);
                return TK_label;
            }
        case '"':
        case '\'':
            lex_string(ls, tv);
            return TK_string;
        case '.':
            if (lex_savenext(ls) == '.') {
                lex_next(ls);
                if (ls->c == '.') {
                    lex_next(ls);
                    return TK_dots;   /* ... */
                }
                return TK_concat;   /* .. */
            } else if (!kp_char_isdigit(ls->c)) {
                return '.';
            } else {
                lex_number(ls, tv);
                return TK_number;
            }
        case LEX_EOF:
            return TK_eof;
        case '&':
            lex_next(ls);
            if (ls->c != '&')
                return '&';
            else {
                lex_next(ls);
                return TK_and;
            }
        case '|':
            lex_next(ls);
            if (ls->c != '|')
                return '|';
            else {
                lex_next(ls);
                return TK_or;
            }
        default: {
            LexChar c = ls->c;
            lex_next(ls);
            return c;  /* Single-char tokens (+ - / ...). */
        }
        }
    }
}

/* -- Lexer API ---------------------------------------------------------- */

/* Setup lexer state. */
int kp_lex_setup(LexState *ls, const char *str)
{
    ls->fs = NULL;
    ls->pe = ls->p = NULL;
    ls->p = str;
    ls->pe = str + strlen(str);
    ls->vstack = NULL;
    ls->sizevstack = 0;
```

```c
    ls->vtop = 0;
    ls->bcstack = NULL;
    ls->sizebcstack = 0;
    ls->lookahead = TK_eof;  /* No look-ahead token. */
    ls->linenumber = 1;
    ls->lastline = 1;
    lex_next(ls);  /* Read-ahead first char. */
    if (ls->c == 0xef && ls->p + 2 <= ls->pe &&
        (uint8_t)ls->p[0] == 0xbb &&
        (uint8_t)ls->p[1] == 0xbf) {/* Skip UTF-8 BOM (if buffered). */
        ls->p += 2;
        lex_next(ls);
    }
    if (ls->c == '#') {  /* Skip POSIX #! header line. */
        do {
            lex_next(ls);
            if (ls->c == LEX_EOF)
                return 0;
        } while (!lex_iseol(ls));
        lex_newline(ls);
    }
    return 0;
}

/* Cleanup lexer state. */
void kp_lex_cleanup(LexState *ls)
{
    free(ls->bcstack);
    free(ls->vstack);
    kp_buf_free(&ls->sb);
}

/* Return next lexical token. */
void kp_lex_next(LexState *ls)
{
    ls->lastline = ls->linenumber;
    if (ls->lookahead == TK_eof) {  /* No lookahead token? */
        ls->tok = lex_scan(ls, &ls->tokval);  /* Get next token. */
    } else {  /* Otherwise return lookahead token. */
        ls->tok = ls->lookahead;
        ls->lookahead = TK_eof;
        ls->tokval = ls->lookaheadval;
    }
}

/* Look ahead for the next token. */
LexToken kp_lex_lookahead(LexState *ls)
{
    kp_assert(ls->lookahead == TK_eof);
    ls->lookahead = lex_scan(ls, &ls->lookaheadval);
    return ls->lookahead;
}

/* Convert token to string. */
const char *kp_lex_token2str(LexState *ls, LexToken tok)
{
    if (tok > TK_OFS)
        return tokennames[tok-TK_OFS-1];
    else if (!kp_char_iscntrl(tok))
        return kp_sprintf("%c", tok);
    else
        return kp_sprintf("char(%d)", tok);
}

/* Lexer error. */
void kp_lex_error(LexState *ls, LexToken tok, ErrMsg em, ...)
{
    const char *tokstr;
    va_list argp;

    if (tok == 0) {
        tokstr = NULL;
    } else if (tok == TK_name || tok == TK_string || tok == TK_number) {
        lex_save(ls, '\0');
        tokstr = sbufB(&ls->sb);
    } else {
```

```
534         tokstr = kp_lex_token2str(ls, tok);
535     }
536 
537     va_start(argp, em);
538     kp_err_lex(ls->chunkname, tokstr, ls->linenumber, em, argp);
539     va_end(argp);
540 }
541 
542 /* Initialize strings for reserved words. */
543 void kp_lex_init()
544 {
545     uint32_t i;
546 
547     for (i = 0; i < TK_RESERVED; i++) {
548         ktap_str_t *s = kp_str_newz(tokennames[i]);
549         s->reserved = (uint8_t)(i+1);
550     }
551 }
552 
```

One Level Up Top Level

runtime/kp_tab.c - ktap

Global variables defined

- kp_niltv

Data types defined

- ktap_node2
- ktap_node2_t

Functions defined

- clearapart
- clearhpart
- hashkey
- hashmask
- hist_record_cmp
- keyindex
- kp_tab_clear
- kp_tab_dup
- kp_tab_free
- kp_tab_get
- kp_tab_getint
- kp_tab_getstr
- kp_tab_incr
- kp_tab_incrint
- kp_tab_incrstr
- kp_tab_len
- kp_tab_new
- kp_tab_new_ah
- kp_tab_newkey
- kp_tab_next
- kp_tab_print_hist
- kp_tab_set
- kp_tab_setint
- kp_tab_setstr

- newhpart
- newtab
- string_convert
- tab_get
- tab_getint
- tab_getinth
- tab_getstr
- tab_histdump
- tab_set
- tab_setint
- tab_setinth
- tab_setstr

Macros defined

- DISTRIBUTION_STR
- TABLE_NARR_ENTRIES
- TABLE_NREC_ENTRIES
- hashgcref
- hashlohi
- hashnum
- hashstr
- niltv
- tab_lock
- tab_lock_init
- tab_unlock

Source code

```
/*
 * kp_tab.c - Table handling.
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * Adapted from luajit and lua interpreter.
 * Copyright (C) 2005-2014 Mike Pall.
 * Copyright (C) 1994-2008 Lua.org, PUC-Rio.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
```

```
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <linux/spinlock.h>
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/slab.h>
#include <linux/sort.h>
#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_vm.h"
#include "kp_obj.h"
#include "kp_str.h"
#include "kp_events.h"
#include "kp_tab.h"

#define tab_lock_init(t)                        \
    do {                                \
        (t)->lock = (arch_spinlock_t)__ARCH_SPIN_LOCK_UNLOCKED;    \
    } while (0)
#define tab_lock(t)                         \
    do {                                \
        local_irq_save(flags);                  \
        arch_spin_lock(&(t)->lock);                \
    } while (0)
#define tab_unlock(t)                           \
    do {                                \
        arch_spin_unlock(&(t)->lock);               \
        local_irq_restore(flags);               \
    } while (0)


const ktap_val_t kp_niltv = { {NULL}, {KTAP_TNIL} } ;
#define niltv  (&kp_niltv)

/* -- Object hashing ---------------------------------------------------- */

/* Hash values are masked with the table hash mask and used as an index. */
static __always_inline
ktap_node_t *hashmask(const ktap_tab_t *t, uint32_t hash)
{
    ktap_node_t *n = t->node;
    return &n[hash & t->hmask];
}

/* String hashes are precomputed when they are interned. */
#define hashstr(t, s)        hashmask(t, (s)->hash)

#define hashlohi(t, lo, hi)    hashmask((t), hashrot((lo), (hi)))
#define hashnum(t, o)        hashlohi((t), (o)->val.n & 0xffffffff, 0)
#define hashgcref(t, o)        hashlohi((t),    \
                ((unsigned long)(o)->val.gc & 0xffffffff), \
                ((unsigned long)(o)->val.gc & 0xffffffff) + HASH_BIAS)


/* Hash an arbitrary key and return its anchor position in the hash table. */
static ktap_node_t *hashkey(const ktap_tab_t *t, const ktap_val_t *key)
{
    kp_assert(!tvisint(key));
    if (is_string(key))
        return hashstr(t, rawtsvalue(key));
    else if (is_number(key))
        return hashnum(t, key);
    else if (is_bool(key))
        return hashmask(t, boolvalue(key));
    else
        return hashgcref(t, key);
}

/* -- Table creation and destruction ------------------------------------ */

```

```c
/* Create new hash part for table. */
static __always_inline
int newhpart(ktap_state_t *ks, ktap_tab_t *t, uint32_t hbits)
{
    uint32_t hsize;
    ktap_node_t *node;
     kp_assert(hbits != 0);

     if (hbits > KP_MAX_HBITS) {
         //kp_error(ks, LJ_ERR_TABOV);
         kp_error(ks, "table overflow\n");
         return -1;
     }
     hsize = 1u << hbits;
     node = vmalloc(hsize * sizeof(ktap_node_t));
     if (!node)
         return -ENOMEM;
     t->freetop = &node[hsize];
     t->node = node;
     t->hmask = hsize-1;

     return 0;
}

/*
* Q: Why all of these copies of t->hmask, t->node etc. to local variables?
* A: Because alias analysis for C is _really_ tough.
*    Even state-of-the-art C compilers won't produce good code without this.
*/

/* Clear hash part of table. */
static __always_inline void clearhpart(ktap_tab_t *t)
{
    uint32_t i, hmask = t->hmask;
    ktap_node_t *node = t->node;
    kp_assert(t->hmask != 0);

    for (i = 0; i <= hmask; i++) {
        ktap_node_t *n = &node[i];
        n->next = NULL;
        set_nil(&n->key);
        set_nil(&n->val);
    }

    t->hnum = 0;
}

/* Clear array part of table. */
static __always_inline void clearapart(ktap_tab_t *t)
{
    uint32_t i, asize = t->asize;
    ktap_val_t *array = t->array;
    for (i = 0; i < asize; i++)
        set_nil(&array[i]);
}

/* Create a new table. Note: the slots are not initialized (yet). */
static ktap_tab_t *newtab(ktap_state_t *ks, uint32_t asize, uint32_t hbits)
{
    ktap_tab_t *t;

    t = (ktap_tab_t *)kp_obj_new(ks, sizeof(ktap_tab_t));
    t->gct = ~KTAP_TTAB;
    t->array = NULL;
    t->asize = 0;  /* In case the array allocation fails. */
    t->hmask = 0;

    tab_lock_init(t);

    if (asize > 0) {
        if (asize > KP_MAX_ASIZE) {
            kp_error(ks, "table overflow\n");
            return NULL;
        }

        t->array = vmalloc(asize * sizeof(ktap_val_t));
```

```
        if (!t->array)
            return NULL;
        t->asize = asize;
    }
    if (hbits)
        if (newhpart(ks, t, hbits)) {
            vfree(t->array);
            return NULL;
        }
    return t;
}

/* Create a new table.
*
* The array size is non-inclusive. E.g. asize=128 creates array slots
* for 0..127, but not for 128. If you need slots 1..128, pass asize=129
* (slot 0 is wasted in this case).
*
* The hash size is given in hash bits. hbits=0 means no hash part.
* hbits=1 creates 2 hash slots, hbits=2 creates 4 hash slots and so on.
*/
ktap_tab_t *kp_tab_new(ktap_state_t *ks, uint32_t asize, uint32_t hbits)
{
    ktap_tab_t *t = newtab(ks, asize, hbits);
    if (!t)
        return NULL;

    clearapart(t);
    if (t->hmask > 0)
        clearhpart(t);
    return t;
}

#define TABLE_NARR_ENTRIES    255 /* PAGE_SIZE / sizeof(ktap_value) - 1 */
#define TABLE_NREC_ENTRIES    2048 /* (PAGE_SIZE * 20) / sizeof(ktap_tnode)*/

ktap_tab_t *kp_tab_new_ah(ktap_state_t *ks, int32_t a, int32_t h)
{
    if (a == 0 && h == 0) {
        a = TABLE_NARR_ENTRIES;
        h = TABLE_NREC_ENTRIES;
    }

    return kp_tab_new(ks, (uint32_t)(a > 0 ? a+1 : 0), hsize2hbits(h));
}

/* Duplicate a table. */
ktap_tab_t *kp_tab_dup(ktap_state_t *ks, const ktap_tab_t *kt)
{
    ktap_tab_t *t;
    uint32_t asize, hmask;
    int i;

    /* allocate default table size */
    t = kp_tab_new_ah(ks, 0, 0);
    if (!t)
        return NULL;

    asize = kt->asize;
    if (asize > 0) {
        ktap_val_t *array = t->array;
        ktap_val_t *karray = kt->array;
        if (asize < 64) {
            /* An inlined loop beats memcpy for < 512 bytes. */
            uint32_t i;
            for (i = 0; i < asize; i++)
                set_obj(&array[i], &karray[i]);
        } else {
            memcpy(array, karray, asize*sizeof(ktap_val_t));
        }
    }

    hmask = kt->hmask;
    for (i = 0; i <= hmask; i++) {
        ktap_node_t *knode = &kt->node[i];
        if (is_nil(&knode->key))
```

```
            continue;
        kp_tab_set(ks, t, &knode->key, &knode->val);
    }
    return t;
}

/* Clear a table. */
void kp_tab_clear(ktap_tab_t *t)
{
    clearapart(t);
    if (t->hmask > 0) {
        ktap_node_t *node = t->node;
        t->freetop = &node[t->hmask+1];
        clearhpart(t);
    }
}

/* Free a table. */
void kp_tab_free(ktap_state_t *ks, ktap_tab_t *t)
{
    if (t->hmask > 0)
        vfree(t->node);
    if (t->asize > 0)
        vfree(t->array);
    kp_free(ks, t);
}

/* -- Table getters ------------------------------------------------------- */

static const ktap_val_t *tab_getinth(ktap_tab_t *t, uint32_t key)
{
    ktap_val_t k;
    ktap_node_t *n;

    set_number(&k, (ktap_number)key);
    n = hashnum(t, &k);
    do {
        if (is_number(&n->key) && nvalue(&n->key) == key) {
            return &n->val;
        }
    } while ((n = n->next));
    return niltv;
}

static __always_inline
const ktap_val_t *tab_getint(ktap_tab_t *t, uint32_t key)
{
    return ((key < t->asize) ? arrayslot(t, key) :
                    tab_getinth(t, key));
}

void kp_tab_getint(ktap_tab_t *t, uint32_t key, ktap_val_t *val)
{
    unsigned long flags;

    tab_lock(t);
    set_obj(val, tab_getint(t, key));
    tab_unlock(t);
}

static const ktap_val_t *tab_getstr(ktap_tab_t *t, ktap_str_t *key)
{
    ktap_node_t *n = hashstr(t, key);
    do {
        if (is_string(&n->key) && rawtsvalue(&n->key) == key)
            return &n->val;
    } while ((n = n->next));
    return niltv;
}

void kp_tab_getstr(ktap_tab_t *t, ktap_str_t *key, ktap_val_t *val)
{
    unsigned long flags;

    tab_lock(t);
    set_obj(val,  tab_getstr(t, key));
```

```
322     tab_unlock(t);
323 }
324 
325 static const ktap_val_t *tab_get(ktap_state_t *ks, ktap_tab_t *t,
326                  const ktap_val_t *key)
327 {
328     if (is_string(key)) {
329         return tab_getstr(t, rawtsvalue(key));
330     } else if (is_number(key)) {
331         ktap_number nk = nvalue(key);
332         uint32_t k = (uint32_t)nk;
333         if (nk == (ktap_number)k) {
334             return tab_getint(t, k);
335         } else {
336             goto genlookup;    /* Else use the generic lookup. */
337         }
338     } else if (is_eventstr(key)) {
339         const ktap_str_t *ts;
340 
341         if (!ks->current_event) {
342             kp_error(ks,
343             "cannot stringify event str in invalid context\n");
344             return niltv;
345         }
346 
347         ts = kp_event_stringify(ks);
348         if (!ts)
349             return niltv;
350 
351         return tab_getstr(t, rawtsvalue(key));
352     } else if (!is_nil(key)) {
353         ktap_node_t *n;
354 genlookup:
355         n = hashkey(t, key);
356         do {
357             if (kp_obj_equal(&n->key, key))
358                 return &n->val;
359         } while ((n = n->next));
360     }
361     return niltv;
362 }
363 
364 void kp_tab_get(ktap_state_t *ks, ktap_tab_t *t, const ktap_val_t *key,
365         ktap_val_t *val)
366 {
367     unsigned long flags;
368 
369     tab_lock(t);
370     set_obj(val, tab_get(ks, t, key));
371     tab_unlock(t);
372 }
373 
374 /* -- Table setters ------------------------------------------------------ */
375 
376 /* Insert new key. Use Brent's variation to optimize the chain length. */
377 static ktap_val_t *kp_tab_newkey(ktap_state_t *ks, ktap_tab_t *t,
378                  const ktap_val_t *key)
379 {
380     ktap_node_t *n = hashkey(t, key);
381 
382     if (!is_nil(&n->val) || t->hmask == 0) {
383         ktap_node_t *nodebase = t->node;
384         ktap_node_t *collide, *freenode = t->freetop;
385 
386         kp_assert(freenode >= nodebase &&
387               freenode <= nodebase+t->hmask+1);
388         do {
389             if (freenode == nodebase) {  /* No free node found? */
390                 //kp_error(ks, LJ_ERR_TABOV);
391                 kp_error(ks, "table overflow\n");
392                 return NULL;
393             }
394         } while (!is_nil(&(--freenode)->key));
395 
396         t->freetop = freenode;
397         collide = hashkey(t, &n->key);
```

```
        if (collide != n) {  /* Colliding node not the main node? */
            while (collide->next != n)
                /* Find predecessor. */
                collide = collide->next;
            collide->next = freenode;  /* Relink chain. */
             /* Copy colliding node into free node and
              * free main node. */
            freenode->val = n->val;
            freenode->key = n->key;
            freenode->next = n->next;
            n->next = NULL;
            set_nil(&n->val);
            /* Rechain pseudo-resurrected string keys with
             * colliding hashes. */
            while (freenode->next) {
                ktap_node_t *nn = freenode->next;
                if (is_string(&nn->key) && !is_nil(&nn->val) &&
                    hashstr(t, rawtsvalue(&nn->key)) == n) {
                    freenode->next = nn->next;
                    nn->next = n->next;
                    n->next = nn;
                } else {
                    freenode = nn;
                }
            }
        } else {  /* Otherwise use free node. */
            freenode->next = n->next;  /* Insert into chain. */
            n->next = freenode;
            n = freenode;
        }
    }
    set_obj(&n->key, key);
    t->hnum++;
    return &n->val;
}

static ktap_val_t *tab_setinth(ktap_state_t *ks, ktap_tab_t *t, uint32_t key)
{
    ktap_val_t k;
    ktap_node_t *n;

    set_number(&k, (ktap_number)key);
    n = hashnum(t, &k);
    do {
        if (is_number(&n->key) && nvalue(&n->key) == key)
            return &n->val;
    } while ((n = n->next));
    return kp_tab_newkey(ks, t, &k);
}

static __always_inline
ktap_val_t *tab_setint(ktap_state_t *ks, ktap_tab_t *t, uint32_t key)
{
    return ((key < t->asize) ? arrayslot(t, key) :
                    tab_setinth(ks, t, key));
}

void kp_tab_setint(ktap_state_t *ks, ktap_tab_t *t,
            uint32_t key, const ktap_val_t *val)
{
    ktap_val_t *v;
    unsigned long flags;

    tab_lock(t);
    v = tab_setint(ks, t, key);
    if (likely(v))
        set_obj(v, val);
    tab_unlock(t);
}

void kp_tab_incrint(ktap_state_t *ks, ktap_tab_t *t, uint32_t key,
             ktap_number n)
{
    ktap_val_t *v;
    unsigned long flags;

```

```
    tab_lock(t);
    v = tab_setint(ks, t, key);
    if (unlikely(!v))
        goto out;

    if (likely(is_number(v)))
        set_number(v, nvalue(v) + n);
    else if (is_nil(v))
        set_number(v, n);
    else
        kp_error(ks, "use '+=' operator on non-number value\n");

out:
    tab_unlock(t);
}

static ktap_val_t *tab_setstr(ktap_state_t *ks, ktap_tab_t *t,
                  const ktap_str_t *key)
{
    ktap_val_t k;
    ktap_node_t *n = hashstr(t, key);
    do {
        if (is_string(&n->key) && rawtsvalue(&n->key) == key)
            return &n->val;
    } while ((n = n->next));
    set_string(&k, key);
    return kp_tab_newkey(ks, t, &k);
}

void kp_tab_setstr(ktap_state_t *ks, ktap_tab_t *t, const ktap_str_t *key,
           const ktap_val_t *val)
{
    ktap_val_t *v;
    unsigned long flags;

    tab_lock(t);
    v = tab_setstr(ks, t, key);
    if (likely(v))
        set_obj(v, val);
    tab_unlock(t);
}

void kp_tab_incrstr(ktap_state_t *ks, ktap_tab_t *t, const ktap_str_t *key,
            ktap_number n)
{
    ktap_val_t *v;
    unsigned long flags;

    tab_lock(t);
    v = tab_setstr(ks, t, key);
    if (unlikely(!v))
        goto out;

    if (likely(is_number(v)))
        set_number(v, nvalue(v) + n);
    else if (is_nil(v))
        set_number(v, n);
    else
        kp_error(ks, "use '+=' operator on non-number value\n");
out:
    tab_unlock(t);
}

static ktap_val_t *tab_set(ktap_state_t *ks, ktap_tab_t *t,
               const ktap_val_t *key)
{
    ktap_node_t *n;

    if (is_string(key)) {
        return tab_setstr(ks, t, rawtsvalue(key));
    } else if (is_number(key)) {
        ktap_number nk = nvalue(key);
        uint32_t k = (ktap_number)nk;
        if (nk == (ktap_number)k)
            return tab_setint(ks, t, k);
    } else if (itype(key) == KTAP_TKSTACK) {
```

```
        /* change stack into string */
        ktap_str_t *bt = kp_obj_kstack2str(ks, key->val.stack.depth,
                                key->val.stack.skip);
        if (!bt)
            return NULL;
        return tab_setstr(ks, t, bt);
    } else if (is_eventstr(key)) {
        const ktap_str_t *ts;

        if (!ks->current_event) {
            kp_error(ks,
            "cannot stringify event str in invalid context\n");
            return NULL;
        }

        ts = kp_event_stringify(ks);
        if (!ts)
            return NULL;

        return tab_setstr(ks, t, ts);
        /* Else use the generic lookup. */
    } else if (is_nil(key)) {
        //kp_error(ks, LJ_ERR_NILIDX);
        kp_error(ks, "table nil index\n");
        return NULL;
    }
    n = hashkey(t, key);
    do {
        if (kp_obj_equal(&n->key, key))
            return &n->val;
    } while ((n = n->next));
    return kp_tab_newkey(ks, t, key);
}

void kp_tab_set(ktap_state_t *ks, ktap_tab_t *t,
        const ktap_val_t *key, const ktap_val_t *val)
{
    ktap_val_t *v;
    unsigned long flags;

    tab_lock(t);
    v = tab_set(ks, t, key);
    if (likely(v))
        set_obj(v, val);
    tab_unlock(t);
}

void kp_tab_incr(ktap_state_t *ks, ktap_tab_t *t, ktap_val_t *key,
         ktap_number n)
{
    ktap_val_t *v;
    unsigned long flags;

    tab_lock(t);
    v = tab_set(ks, t, key);
    if (unlikely(!v))
        goto out;

    if (likely(is_number(v)))
        set_number(v, nvalue(v) + n);
    else if (is_nil(v))
        set_number(v, n);
    else
        kp_error(ks, "use '+=' operator on non-number value\n");
out:
    tab_unlock(t);
}


/* -- Table traversal ----------------------------------------------------- */

/* Get the traversal index of a key. */
static uint32_t keyindex(ktap_state_t *ks, ktap_tab_t *t,
             const ktap_val_t *key)
{
    if (is_number(key)) {
```

```
        ktap_number nk = nvalue(key);
        uint32_t k = (uint32_t)nk;
        /* Array key indexes: [0..t->asize-1] */
        if ((uint32_t)k < t->asize && nk == (ktap_number)k)
            return (uint32_t)k;
    }

    if (!is_nil(key)) {
        ktap_node_t *n = hashkey(t, key);
        do {
            if (kp_obj_equal(&n->key, key))
                return t->asize + (uint32_t)(n - (t->node));
            /* Hash key indexes: [t->asize..t->asize+t->nmask] */
        } while ((n = n->next));
        //kp_err_msg(ks, LJ_ERR_NEXTIDX);
        kp_error(ks, "table next index\n");
        return 0;  /* unreachable */
    }
    return ~0u;  /* A nil key starts the traversal. */
}

/* Advance to the next step in a table traversal. */
int kp_tab_next(ktap_state_t *ks, ktap_tab_t *t, ktap_val_t *key)
{
    unsigned long flags;
    uint32_t i;

    tab_lock(t);
    i = keyindex(ks, t, key);  /* Find predecessor key index. */

    /* First traverse the array keys. */
    for (i++; i < t->asize; i++)
         if (!is_nil(arrayslot(t, i))) {
            set_number(key, i);
            set_obj(key + 1, arrayslot(t, i));
            tab_unlock(t);
            return 1;
        }
    /* Then traverse the hash keys. */
    for (i -= t->asize; i <= t->hmask; i++) {
        ktap_node_t *n = &t->node[i];
        if (!is_nil(&n->val)) {
            set_obj(key, &n->key);
            set_obj(key + 1, &n->val);
            tab_unlock(t);
            return 1;
        }
    }
    tab_unlock(t);
    return 0;  /* End of traversal. */
}

/* -- Table length calculation ---------------------------------------- */

int kp_tab_len(ktap_state_t *ks, ktap_tab_t *t)
{
    unsigned long flags;
    int i, len = 0;

    tab_lock(t);
    for (i = 0; i < t->asize; i++) {
        ktap_val_t *v = &t->array[i];

        if (is_nil(v))
            continue;
        len++;
    }

    for (i = 0; i <= t->hmask; i++) {
        ktap_node_t *n = &t->node[i];

        if (is_nil(&n->key))
            continue;

        len++;
    }
```

```
    tab_unlock(t);
    return len;
}

static void string_convert(char *output, const char *input)
{
    if (strlen(input) > 32) {
        strncpy(output, input, 32-4);
        memset(output + 32-4, '.', 3);
    } else
        memcpy(output, input, strlen(input));
}

typedef struct ktap_node2 {
    ktap_val_t key;
    ktap_val_t val;
} ktap_node2_t;

static int hist_record_cmp(const void *i, const void *j)
{
    ktap_number n1 = nvalue(&((const ktap_node2_t *)i)->val);
    ktap_number n2 = nvalue(&((const ktap_node2_t *)j)->val);

    if (n1 == n2)
        return 0;
    else if (n1 < n2)
        return 1;
    else
        return -1;
}

/* todo: make histdump to be faster, just need to sort n entries, not all */

/* print_hist: key should be number/string/ip, value must be number */
static void tab_histdump(ktap_state_t *ks, ktap_tab_t *t, int shownums)
{
    long start_time, delta_time;
    uint32_t i, asize = t->asize;
    ktap_val_t *array = t->array;
    uint32_t hmask = t->hmask;
    ktap_node_t *node = t->node;
    ktap_node2_t *sort_mem;
    char dist_str[39];
    int total = 0, sum = 0;

    start_time = gettimeofday_ns();

    sort_mem = kmalloc((t->asize + t->hnum) * sizeof(ktap_node2_t),
                GFP_KERNEL);
    if (!sort_mem)
        return;

    /* copy all values in table into sort_mem. */
    for (i = 0; i < asize; i++) {
        ktap_val_t *val = &array[i];
        if (is_nil(val))
            continue;

        if (!is_number(val)) {
            kp_error(ks, "print_hist only can print number\n");
            goto out;
        }

        set_number(&sort_mem[total].key, i);
        set_obj(&sort_mem[total].val, val);
        sum += nvalue(val);
        total++;
    }

    for (i = 0; i <= hmask; i++) {
        ktap_node_t *n = &node[i];
        ktap_val_t *val = &n->val;

        if (is_nil(val))
            continue;

```

```
778         if (!is_number(val)) {
779             kp_error(ks, "print_hist only can print number\n");
780             goto out;
781         }
782 
783         set_obj(&sort_mem[total].key, &n->key);
784         set_obj(&sort_mem[total].val, val);
785         sum += nvalue(val);
786         total++;
787     }
788 
789     /* sort */
790     sort(sort_mem, total, sizeof(ktap_node2_t), hist_record_cmp, NULL);
791 
792     dist_str[sizeof(dist_str) - 1] = '\0';
793 
794     for (i = 0; i < total; i++) {
795         ktap_val_t *key = &sort_mem[i].key;
796         ktap_number num = nvalue(&sort_mem[i].val);
797         int ratio;
798 
799         if (!--shownums)
800             break;
801 
802         memset(dist_str, ' ', sizeof(dist_str) - 1);
803         ratio = (num * (sizeof(dist_str) - 1)) / sum;
804         memset(dist_str, '@', ratio);
805 
806         if (is_string(key)) {
807             //char buf[32] = {0};
808 
809             //string_convert(buf, svalue(key));
810             if (rawtsvalue(key)->len > 32) {
811                 kp_puts(ks, svalue(key));
812                 kp_printf(ks, "%s\n%d\n\n", dist_str, num);
813             } else {
814                 kp_printf(ks, "%31s |%s%-7d\n", svalue(key),
815                                 dist_str, num);
816             }
817         } else if (is_number(key)) {
818             kp_printf(ks, "%31d |%s%-7d\n", nvalue(key),
819                             dist_str, num);
820         } else if (is_kip(key)) {
821             char str[KSYM_SYMBOL_LEN];
822             char buf[32] = {0};
823 
824             SPRINT_SYMBOL(str, nvalue(key));
825             string_convert(buf, str);
826             kp_printf(ks, "%31s |%s%-7d\n", buf, dist_str, num);
827         }
828     }
829 
830     if (!shownums && total)
831         kp_printf(ks, "%31s |\n", "...");
832 
833 out:
834     kfree(sort_mem);
835 
836     delta_time = (gettimeofday_ns() - start_time) / NSEC_PER_USEC;
837     kp_verbose_printf(ks, "tab_histdump time: %d (us)\n", delta_time);
838 }
839 
840 #define DISTRIBUTION_STR "------------- Distribution -------------"
841 void kp_tab_print_hist(ktap_state_t *ks, ktap_tab_t *t, int n)
842 {
843     kp_printf(ks, "%31s%s%s\n", "value ", DISTRIBUTION_STR, " count");
844     tab_histdump(ks, t, n);
845 }
```

One Level Up Top Level

runtime/kp_tab.h - ktap

Functions defined

- hashrot

Macros defined

- FLS
- HASH_BIAS
- HASH_ROT1
- HASH_ROT2
- HASH_ROT3
- __KTAP_TAB_H__
- arrayslot
- hsize2hbits
- kp_rol
- kp_ror

Source code

```
#ifndef __KTAP_TAB_H__
#define __KTAP_TAB_H__

/* Hash constants. Tuned using a brute force search. */
#define HASH_BIAS       (-0x04c11db7)
#define HASH_ROT1       14
#define HASH_ROT2       5
#define HASH_ROT3       13

 /* Every half-decent C compiler transforms this into a rotate instruction. */
 #define kp_rol(x, n)    (((x)<<(n)) | ((x)>>(-(int)(n)&(8*sizeof(x)-1))))
 #define kp_ror(x, n)    (((x)<<(-(int)(n)&(8*sizeof(x)-1))) | ((x)>>(n)))

 /* Scramble the bits of numbers and pointers. */
 static __always_inline uint32_t hashrot(uint32_t lo, uint32_t hi)
 {
     /* Prefer variant that compiles well for a 2-operand CPU. */
     lo ^= hi; hi = kp_rol(hi, HASH_ROT1);
     lo -= hi; hi = kp_rol(hi, HASH_ROT2);
     hi ^= lo; hi -= kp_rol(lo, HASH_ROT3);
     return hi;
 }


 #define FLS(x)       ((uint32_t)(__builtin_clz(x)^31))
 #define hsize2hbits(s)  ((s) ? ((s)==1 ? 1 : 1+FLS((uint32_t)((s)-1))) : 0)

 #define arrayslot(t, i)         (&(t)->array[(i)])

 void kp_tab_set(ktap_state_t *ks, ktap_tab_t *t, const ktap_val_t *key,
         const ktap_val_t *val);
 void kp_tab_setstr(ktap_state_t *ks, ktap_tab_t *t,
            const ktap_str_t *key, const ktap_val_t *val);
 void kp_tab_incrstr(ktap_state_t *ks, ktap_tab_t *t, const ktap_str_t *key,
             ktap_number n);
 void kp_tab_get(ktap_state_t *ks, ktap_tab_t *t, const ktap_val_t *key,
```

```
        ktap_val_t *val);
void kp_tab_getstr(ktap_tab_t *t, ktap_str_t *key, ktap_val_t *val);

void kp_tab_getint(ktap_tab_t *t, uint32_t key, ktap_val_t *val);
void kp_tab_setint(ktap_state_t *ks, ktap_tab_t *t,
           uint32_t key, const ktap_val_t *val);
void kp_tab_incrint(ktap_state_t *ks, ktap_tab_t *t, uint32_t key,
            ktap_number n);
ktap_tab_t *kp_tab_new(ktap_state_t *ks, uint32_t asize, uint32_t hbits);
ktap_tab_t *kp_tab_new_ah(ktap_state_t *ks, int32_t a, int32_t h);
ktap_tab_t *kp_tab_dup(ktap_state_t *ks, const ktap_tab_t *kt);

void kp_tab_free(ktap_state_t *ks, ktap_tab_t *t);
int kp_tab_len(ktap_state_t *ks, ktap_tab_t *t);
void kp_tab_dump(ktap_state_t *ks, ktap_tab_t *t);
void kp_tab_clear(ktap_tab_t *t);
void kp_tab_print_hist(ktap_state_t *ks, ktap_tab_t *t, int n);
int kp_tab_next(ktap_state_t *ks, ktap_tab_t *t, StkId key);
int kp_tab_sort_next(ktap_state_t *ks, ktap_tab_t *t, StkId key);
void kp_tab_sort(ktap_state_t *ks, ktap_tab_t *t, ktap_func_t *cmp_func);
void kp_tab_incr(ktap_state_t *ks, ktap_tab_t *t, ktap_val_t *key,
        ktap_number n);
#endif /* __KTAP_TAB_H__ */
```

One Level Up Top Level

One Level Up Top Level

runtime/kp_obj.h - ktap

Macros defined

- __KTAP_OBJ_H__
- kp_obj_equal

Source code

```
1  #ifndef __KTAP_OBJ_H__
2  #define __KTAP_OBJ_H__
3
4  void *kp_malloc(ktap_state_t *ks, int size);
5  void *kp_zalloc(ktap_state_t *ks, int size);
6  void kp_free(ktap_state_t *ks, void *addr);
7
8  void kp_obj_dump(ktap_state_t *ks, const ktap_val_t *v);
9  void kp_obj_show(ktap_state_t *ks, const ktap_val_t *v);
10 int kp_obj_len(ktap_state_t *ks, const ktap_val_t *rb);
11 ktap_obj_t *kp_obj_new(ktap_state_t *ks, size_t size);
12 int kp_obj_rawequal(const ktap_val_t *t1, const ktap_val_t *t2);
13 ktap_str_t *kp_obj_kstack2str(ktap_state_t *ks, uint16_t depth, uint16_t skip);
14 void kp_obj_free_gclist(ktap_state_t *ks, ktap_obj_t *o);
15 void kp_obj_freeall(ktap_state_t *ks);
16
17 #define kp_obj_equal(o1, o2) \
18     (((o1)->type == (o2)->type) && kp_obj_rawequal(o1, o2))
19
20 #endif /* __KTAP_OBJ_H__ */
```

One Level Up Top Level

userspace/kp_parse_events.c - ktap

Global variables defined

- id_nr
- idmap
- idmap_size
- kprobes_text_end
- kprobes_text_start
- probe_list_head

Data types defined

- probe_cb_base
- probe_list

Functions defined

- add_event
- add_tracepoint
- check_kprobe_addr_prohibited
- cleanup_event_resources
- format_symbol_name
- get_id_array
- get_next_eventdef
- get_sys_event_filter_str
- idmap_clear
- idmap_free
- idmap_get_max_id
- idmap_init
- idmap_is_set
- idmap_set
- init_kprobe_prohibited_area
- kp_parse_events
- kprobe_symbol_actor
- parse_events_add_kprobe
- parse_events_add_probe

- parse_events_add_sdt
- parse_events_add_tracepoint
- parse_events_add_uprobe
- parse_events_resolve_symbol
- parse_events_resolve_symbol
- strim
- uprobe_symbol_actor
- write_kprobe_event
- write_uprobe_event

Macros defined

- KPROBE_EVENTS_PATH
- TRACING_EVENTS_DIR
- UPROBE_EVENTS_PATH

Source code

```c
/*
 * parse_events.c - ktap events parser
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <fcntl.h>
#include <ctype.h>

#include "../include/ktap_types.h"
#include "../include/ktap_bc.h"
#include "kp_symbol.h"
#include "kp_util.h"

#define TRACING_EVENTS_DIR "/sys/kernel/debug/tracing/events"

static u8 *idmap;
static int idmap_size = 1024; /* set init size */
static int id_nr;

static int idmap_init(void)
```

```
{
    idmap = malloc(idmap_size);
    if (!idmap)
        return -1;

    memset(idmap, 0, idmap_size);
    return 0;
}

static void idmap_free(void)
{
    id_nr = 0;
    free(idmap);
}

static inline int idmap_is_set(int id)
{
    return idmap[id / 8] & (1 << (id % 8));
}

static void idmap_set(int id)
{
    if (id >= idmap_size * 8) {
        int newsize = id + 100; /* allocate extra 800 id */
        idmap = realloc(idmap, newsize);
        memset(idmap + idmap_size, 0, newsize - idmap_size);
        idmap_size = newsize;
    }

    if (!idmap_is_set(id))
        id_nr++;

    idmap[id / 8] = idmap[id / 8] | (1 << (id % 8));
}

static void idmap_clear(int id)
{
    if (!idmap_is_set(id))
        return;

    id_nr--;
    idmap[id / 8] = idmap[id / 8] & ~ (1 << (id % 8));
}

static int idmap_get_max_id(void)
{
    return idmap_size * 8;
}

static int *get_id_array()
{
    int *id_array;
    int i, j = 0;

    id_array = malloc(sizeof(int) * id_nr);
    if (!id_array)
        return NULL;

     for (i = 0; i < idmap_get_max_id(); i++) {
         if (idmap_is_set(i))
             id_array[j++] = i;
     }

     return id_array;
 }

 static int add_event(char *evtid_path)
 {
     char id_buf[24];
     int id, fd;

     fd = open(evtid_path, O_RDONLY);
     if (fd < 0) {
         /*
          * some tracepoint doesn't have id file, like ftrace,
          * return success in here, and don't print error.
```

```
         */
        verbose_printf("warning: cannot open file %s\n", evtid_path);
        return 0;
    }

    if (read(fd, id_buf, sizeof(id_buf)) < 0) {
        fprintf(stderr, "read file error %s\n", evtid_path);
        close(fd);
        return -1;
    }

    id = atoll(id_buf);

    idmap_set(id);

    close(fd);
    return 0;
}

static int add_tracepoint(const char *sys_name, const char *evt_name)
{
    char evtid_path[PATH_MAX] = {0};

    snprintf(evtid_path, PATH_MAX, "%s/%s/%s/id", TRACING_EVENTS_DIR,
                    sys_name, evt_name);
    return add_event(evtid_path);
}

static int parse_events_add_tracepoint(char *sys, char *event)
{
    process_available_tracepoints(sys, event, add_tracepoint);
    return 0;
}

enum {
    KPROBE_EVENT,
    UPROBE_EVENT,
};

struct probe_list {
    struct probe_list *next;
    int type;
    char event[64];
};

static struct probe_list *probe_list_head; /* for cleanup resources */

/*
* Some symbol format cannot write to uprobe_events in debugfs, like:
* symbol "check_one_fd.part.0" in glibc.
* For those symbols, we change the format to:
* "check_one_fd.part.0" -> "check_one_fd_part_0"
*/
static char *format_symbol_name(const char *old_symbol)
{
    char *new_name = strdup(old_symbol);
    char *name = new_name;
    int changed = 0;

        if (!isalpha(*name) && *name != '_') {
        *name = '_';
        changed = 1;
    }

        while (*++name != '\0') {
                if (!isalpha(*name) && !isdigit(*name) && *name != '_') {
            *name = '_';
            changed = 1;
            continue;
        }
        }

    if (changed)
        fprintf(stderr,
            "Warning: symbol \"%s\" transformed to event \"%s\"\n",
            old_symbol, new_name);
```

```
194
195     /* this is a good name */
196         return new_name;
197 }
198
199
200 #define KPROBE_EVENTS_PATH "/sys/kernel/debug/tracing/kprobe_events"
201
202 /**
203 * @return 0 on success, otherwise -1
204 */
205 static int
206 write_kprobe_event(int fd, int ret_probe, const char *symbol,
207            unsigned long start, char *fetch_args)
208 {
209     char probe_event[128] = {0};
210     char event[64] = {0};
211     struct probe_list *pl;
212     char event_id_path[128] = {0};
213     char *symbol_name;
214     int id_fd, ret;
215
216     /* In case some symbols cannot write to uprobe_events debugfs file */
217     symbol_name = format_symbol_name(symbol);
218
219     if (!fetch_args)
220         fetch_args = " ";
221
222     if (ret_probe) {
223         snprintf(event, 64, "ktap_kprobes_%d/ret_%s",
224              getpid(), symbol_name);
225         /* Return probe point must be a symbol */
226         snprintf(probe_event, 128, "r:%s %s %s",
227              event, symbol, fetch_args);
228     } else {
229         snprintf(event, 64, "ktap_kprobes_%d/%s",
230              getpid(), symbol_name);
231         snprintf(probe_event, 128, "p:%s 0x%lx %s",
232              event, start, fetch_args);
233     }
234
235     sprintf(event_id_path, "/sys/kernel/debug/tracing/events/%s/id", event);
236     /* if event id already exist, then don't write to kprobes_event again */
237     id_fd = open(event_id_path, O_RDONLY);
238     if (id_fd > 0) {
239         close(id_fd);
240
241         /* remember add event id to ids_array */
242         ret = add_event(event_id_path);
243         if (ret)
244             goto error;
245
246         goto out;
247     }
248
249     verbose_printf("write kprobe event %s\n", probe_event);
250
251     if (write(fd, probe_event, strlen(probe_event)) <= 0) {
252         fprintf(stderr, "Cannot write %s to %s\n", probe_event,
253                 KPROBE_EVENTS_PATH);
254         goto error;
255     }
256
257     /* add to cleanup list */
258     pl = malloc(sizeof(struct probe_list));
259     if (!pl)
260         goto error;
261
262     pl->type = KPROBE_EVENT;
263     pl->next = probe_list_head;
264     memcpy(pl->event, event, 64);
265     probe_list_head = pl;
266
267     ret = add_event(event_id_path);
268     if (ret < 0)
269         goto error;
```

```
270
271 out:
272     free(symbol_name);
273     return 0;
274
275 error:
276     free(symbol_name);
277     return -1;
278 }
279
280 static unsigned long kprobes_text_start;
281 static unsigned long kprobes_text_end;
282
283 static void init_kprobe_prohibited_area(void)
284 {
285     static int once = 0;
286
287     if (once > 0)
288         return;
289
290     once = 1;
291     kprobes_text_start     = find_kernel_symbol("__kprobes_text_start");
292     kprobes_text_end       = find_kernel_symbol("__kprobes_text_end");
293 }
294
295 static int check_kprobe_addr_prohibited(unsigned long addr)
296 {
297     if (addr >= kprobes_text_start && addr <= kprobes_text_end)
298         return -1;
299
300     return 0;
301 }
302
303 struct probe_cb_base {
304     int fd;
305     int ret_probe;
306     const char *event;
307     char *binary;
308     char *symbol;
309     char *fetch_args;
310 };
311
312 static int kprobe_symbol_actor(void *arg, const char *name, char type,
313                    unsigned long start)
314 {
315     struct probe_cb_base *base = (struct probe_cb_base *)arg;
316
317     /* only can probe text function */
318     if (type != 't' && type != 'T')
319         return -1;
320
321     if (!strglobmatch(name, base->symbol))
322         return -1;
323
324     if (check_kprobe_addr_prohibited(start))
325         return -1;
326
327     /* ignore reture code of write debugfs */
328     write_kprobe_event(base->fd, base->ret_probe, name, start,
329                base->fetch_args);
330
331     return 0; /* success */
332 }
333
334 static int parse_events_add_kprobe(char *event)
335 {
336     char *symbol, *end;
337     struct probe_cb_base base;
338     int fd, ret;
339
340     fd = open(KPROBE_EVENTS_PATH, O_WRONLY);
341     if (fd < 0) {
342         fprintf(stderr, "Cannot open %s\n", KPROBE_EVENTS_PATH);
343         return -1;
344     }
345
```

```
    end = strpbrk(event, "% ");
    if (end)
        symbol = strndup(event, end - event);
    else
        symbol = strdup(event);

    base.fd = fd;
    base.ret_probe = !!strstr(event, "%return");
    base.symbol = symbol;
    base.fetch_args = strchr(event, ' ');

    init_kprobe_prohibited_area();

    ret = kallsyms_parse(&base, kprobe_symbol_actor);
    if (ret <= 0) {
        fprintf(stderr, "cannot parse symbol \"%s\"\n", symbol);
        ret = -1;
    } else {
        ret = 0;
    }

    free(symbol);
    close(fd);

    return ret;
}

#define UPROBE_EVENTS_PATH "/sys/kernel/debug/tracing/uprobe_events"

/**
* @return 0 on success, otherwise -1
*/
static int
write_uprobe_event(int fd, int ret_probe, const char *binary,
           const char *symbol, unsigned long addr,
           char *fetch_args)
{
    char probe_event[128] = {0};
    char event[64] = {0};
    struct probe_list *pl;
    char event_id_path[128] = {0};
    char *symbol_name;
    int id_fd, ret;

    /* In case some symbols cannot write to uprobe_events debugfs file */
    symbol_name = format_symbol_name(symbol);

    if (!fetch_args)
        fetch_args = " ";

    if (ret_probe) {
        snprintf(event, 64, "ktap_uprobes_%d/ret_%s",
             getpid(), symbol_name);
        snprintf(probe_event, 128, "r:%s %s:0x%lx %s",
             event, binary, addr, fetch_args);
    } else {
        snprintf(event, 64, "ktap_uprobes_%d/%s",
             getpid(), symbol_name);
        snprintf(probe_event, 128, "p:%s %s:0x%lx %s",
             event, binary, addr, fetch_args);
    }

    sprintf(event_id_path, "/sys/kernel/debug/tracing/events/%s/id", event);
    /* if event id already exist, then don't write to uprobes_event again */
    id_fd = open(event_id_path, O_RDONLY);
    if (id_fd > 0) {
        close(id_fd);

        /* remember add event id to ids_array */
        ret = add_event(event_id_path);
        if (ret)
            goto error;

        goto out;
    }

```

```
422     verbose_printf("write uprobe event %s\n", probe_event);
423
424     if (write(fd, probe_event, strlen(probe_event)) <= 0) {
425         fprintf(stderr, "Cannot write %s to %s\n", probe_event,
426                 UPROBE_EVENTS_PATH);
427         goto error;
428     }
429
430     /* add to cleanup list */
431     pl = malloc(sizeof(struct probe_list));
432     if (!pl)
433         goto error;
434
435     pl->type = UPROBE_EVENT;
436     pl->next = probe_list_head;
437     memcpy(pl->event, event, 64);
438     probe_list_head = pl;
439
440     ret = add_event(event_id_path);
441     if (ret < 0)
442         goto error;
443
444 out:
445     free(symbol_name);
446     return 0;
447
448 error:
449     free(symbol_name);
450     return -1;
451 }
452
453 /**
454 * TODO: avoid copy-paste stuff
455 *
456 * @return 1 on success, otherwise 0
457 */
458 #ifdef NO_LIBELF
459 static int parse_events_resolve_symbol(int fd, char *event, int type)
460 {
461     char *colon, *binary, *fetch_args;
462     unsigned long symbol_address;
463
464     colon = strchr(event, ':');
465     if (!colon)
466         return -1;
467
468     symbol_address = strtol(colon + 1 /* skip ":" */, NULL, 0);
469
470     fetch_args = strchr(event, ' ');
471
472     /**
473      * We already have address, no need in resolving.
474      */
475     if (symbol_address) {
476         int ret;
477
478         binary = strndup(event, colon - event);
479         ret = write_uprobe_event(fd, !!strstr(event, "%return"), binary,
480                     "NULL", symbol_address, fetch_args);
481         free(binary);
482         return ret;
483     }
484
485     fprintf(stderr, "error: cannot resolve event \"%s\" without libelf, "
486             "please recompile ktap with NO_LIBELF disabled\n",
487             event);
488     exit(EXIT_FAILURE);
489     return -1;
490 }
491
492 #else
493 static int uprobe_symbol_actor(const char *name, vaddr_t addr, void *arg)
494 {
495     struct probe_cb_base *base = (struct probe_cb_base *)arg;
496     int ret;
497
```

```
    if (!strglobmatch(name, base->symbol))
        return 0;

    verbose_printf("uprobe: binary: \"%s\" symbol \"%s\" "
            "resolved to 0x%lx\n",
            base->binary, base->symbol, (unsigned long)addr);

    ret = write_uprobe_event(base->fd, base->ret_probe, base->binary,
                  name, addr, base->fetch_args);
    if (ret)
        return ret;

    return 0;
}

static int parse_events_resolve_symbol(int fd, char *event, int type)
{
    char *colon, *end;
    vaddr_t symbol_address;
    int ret;
    struct probe_cb_base base = {
        .fd = fd,
        .event = event
    };

    colon = strchr(event, ':');
    if (!colon)
        return 0;

    base.ret_probe = !!strstr(event, "%return");
    symbol_address = strtol(colon + 1 /* skip ":" */, NULL, 0);
    base.binary = strndup(event, colon - event);

    base.fetch_args = strchr(event, ' ');

    /*
     * We already have address, no need in resolving.
     */
    if (symbol_address) {
        int ret;
        ret = write_uprobe_event(fd, base.ret_probe, base.binary,
                      "NULL", symbol_address,
                      base.fetch_args);
        free(base.binary);
        return ret;
    }

    end = strpbrk(event, "% ");
    if (end)
        base.symbol = strndup(colon + 1, end - 1 - colon);
    else
        base.symbol = strdup(colon + 1);

    ret = parse_dso_symbols(base.binary, type, uprobe_symbol_actor,
                (void *)&base);
    if (!ret) {
        fprintf(stderr, "error: cannot find symbol %s in binary %s\n",
            base.symbol, base.binary);
        ret = -1;
    } else if(ret > 0) {
        /* no error found when parse symbols */
        ret = 0;
    }

    free(base.binary);
    free(base.symbol);

    return ret;
}
#endif

static int parse_events_add_uprobe(char *old_event, int type)
{
    int ret;
    int fd;

```

```
    fd = open(UPROBE_EVENTS_PATH, O_WRONLY);
    if (fd < 0) {
        fprintf(stderr, "Cannot open %s\n", UPROBE_EVENTS_PATH);
        return -1;
    }

    ret = parse_events_resolve_symbol(fd, old_event, type);

    close(fd);
    return ret;
}

static int parse_events_add_probe(char *old_event)
{
    char *separator;

    separator = strchr(old_event, ':');
    if (!separator || (separator == old_event))
        return parse_events_add_kprobe(old_event);
    else
        return parse_events_add_uprobe(old_event, FIND_SYMBOL);
}

static int parse_events_add_sdt(char *old_event)
{
    return parse_events_add_uprobe(old_event, FIND_STAPSDT_NOTE);
}

static void strim(char *s)
{
    size_t size;
    char *end;

    size = strlen(s);
    if (!size)
        return;

    end = s + size -1;
    while (end >= s && isspace(*end))
        end--;

    *(end + 1) = '\0';
}

static int get_sys_event_filter_str(char *start,
                    char **sys, char **event, char **filter)
{
    char *separator, *separator2, *ptr, *end;

    while (*start == ' ')
        start++;

    /* find sys */
    separator = strchr(start, ':');
    if (!separator || (separator == start)) {
        return -1;
    }

    ptr = malloc(separator - start + 1);
    if (!ptr)
        return -1;

    strncpy(ptr, start, separator - start);
    ptr[separator - start] = '\0';

    strim(ptr);
    *sys = ptr;

    if (!strcmp(*sys, "probe") && (*(separator + 1) == '/')) {
        /* it's uprobe event */
        separator2 = strchr(separator + 1, ':');
        if (!separator2)
            return -1;
    } else
        separator2 = separator;

```

```
650     /* find filter */
651     end = start + strlen(start);
652     while (*--end == ' ') {
653     }
654
655     if (*end == '/') {
656         char *filter_start;
657
658         filter_start = strchr(separator2, '/');
659         if (filter_start == end)
660             return -1;
661
662         ptr = malloc(end - filter_start);
663         if (!ptr)
664             return -1;
665
666         memcpy(ptr, filter_start + 1, end - filter_start - 1);
667         ptr[end - filter_start - 1] = '\0';
668
669         *filter = ptr;
670
671         end = filter_start;
672     } else {
673         *filter = NULL;
674         end++;
675     }
676
677     /* find event */
678     ptr = malloc(end - separator);
679     if (!ptr)
680         return -1;
681
682     memcpy(ptr, separator + 1, end - separator - 1);
683     ptr[end - separator - 1] = '\0';
684
685     strim(ptr);
686     *event = ptr;
687
688     return 0;
689 }
690
691 static char *get_next_eventdef(char *str)
692 {
693     char *separator;
694
695     separator = strchr(str, ',');
696     if (!separator)
697         return str + strlen(str);
698
699     *separator = '\0';
700     return separator + 1;
701 }
702
703 ktap_eventdesc_t *kp_parse_events(const char *eventdef)
704 {
705     char *str = strdup(eventdef);
706     char *sys, *event, *filter, *next;
707     ktap_eventdesc_t *evdef_info;
708     int ret;
709
710     idmap_init();
711
712 parse_next_eventdef:
713     next = get_next_eventdef(str);
714
715     if (get_sys_event_filter_str(str, &sys, &event, &filter))
716         goto error;
717
718     verbose_printf("parse_eventdef: sys[%s], event[%s], filter[%s]\n",
719                sys, event, filter);
720
721     if (!strcmp(sys, "probe"))
722         ret = parse_events_add_probe(event);
723     else if (!strcmp(sys, "sdt"))
724         ret = parse_events_add_sdt(event);
725     else
```

```
726         ret = parse_events_add_tracepoint(sys, event);
727
728     if (ret)
729         goto error;
730
731     /* don't trace ftrace:function when all tracepoints enabled */
732     if (!strcmp(sys, "*"))
733         idmap_clear(1);
734
735
736     if (filter && *next != '\0') {
737         fprintf(stderr, "Error: eventdef only can append one filter\n");
738         goto error;
739     }
740
741     str = next;
742     if (*next != '\0')
743         goto parse_next_eventdef;
744
745     evdef_info = malloc(sizeof(*evdef_info));
746     if (!evdef_info)
747         goto error;
748
749     evdef_info->nr = id_nr;
750     evdef_info->id_arr = get_id_array();
751     evdef_info->filter = filter;
752
753     idmap_free();
754     return evdef_info;
755 error:
756     idmap_free();
757     cleanup_event_resources();
758     return NULL;
759 }
760
761 void cleanup_event_resources(void)
762 {
763     struct probe_list *pl;
764     const char *path;
765     char probe_event[128] = {0};
766     int fd, ret;
767
768     for (pl = probe_list_head; pl; pl = pl->next) {
769         if (pl->type == KPROBE_EVENT)
770             path = KPROBE_EVENTS_PATH;
771         else if (pl->type == UPROBE_EVENT)
772             path = UPROBE_EVENTS_PATH;
773         else {
774             fprintf(stderr, "Cannot cleanup event type %d\n",
775                     pl->type);
776             continue;
777         }
778
779         snprintf(probe_event, 128, "-:%s", pl->event);
780
781         fd = open(path, O_WRONLY);
782         if (fd < 0) {
783             fprintf(stderr, "Cannot open %s\n", UPROBE_EVENTS_PATH);
784             continue;
785         }
786
787         ret = write(fd, probe_event, strlen(probe_event));
788         if (ret <= 0) {
789             fprintf(stderr, "Cannot write %s to %s\n", probe_event,
790                     path);
791             close(fd);
792             continue;
793         }
794
795         close(fd);
796     }
797 }
798
```

One Level Up Top Level

userspace/kp_reader.c - ktap

Functions defined

- block_sigint
- kp_create_reader
- reader_thread
- sigfunc

Macros defined

- MAX_BUFLEN
- PATH_MAX
- handle_error

Source code

```c
/*
 * reader.c - ring buffer reader in userspace
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/poll.h>
#include <sys/signal.h>
#include <fcntl.h>
#include <pthread.h>

#define MAX_BUFLEN  131072
#define PATH_MAX 128

#define handle_error(str) do { perror(str); exit(-1); } while(0)

void sigfunc(int signo)
{
    /* should not not reach here */
}

static void block_sigint()
{
    sigset_t mask;
```

```
46
47      sigemptyset(&mask);
48      sigaddset(&mask, SIGINT);
49
50      pthread_sigmask(SIG_BLOCK, &mask, NULL);
51  }
52
53  static void *reader_thread(void *data)
54  {
55      char buf[MAX_BUFLEN];
56      char filename[PATH_MAX];
57      const char *output = data;
58      int failed = 0, fd, out_fd, len;
59
60      block_sigint();
61
62      if (output) {
63          out_fd = open(output, O_CREAT | O_WRONLY | O_TRUNC,
64                        S_IRUSR|S_IWUSR);
65          if (out_fd < 0) {
66              fprintf(stderr, "Cannot open output file %s\n", output);
67              return NULL;
68          }
69      } else
70          out_fd = 1;
71
72      sprintf(filename, "/sys/kernel/debug/ktap/trace_pipe_%d", getpid());
73
74  open_again:
75      fd = open(filename, O_RDONLY);
76      if (fd < 0) {
77          usleep(10000);
78
79          if (failed++ == 10) {
80              fprintf(stderr, "Cannot open file %s\n", filename);
81              return NULL;
82          }
83          goto open_again;
84      }
85
86      while ((len = read(fd, buf, sizeof(buf))) > 0)
87          write(out_fd, buf, len);
88
89      close(fd);
90      close(out_fd);
91
92      return NULL;
93  }
94
95  int kp_create_reader(const char *output)
96  {
97      pthread_t reader;
98
99      signal(SIGINT, sigfunc);
100
101      if (pthread_create(&reader, NULL, reader_thread, (void *)output) < 0)
102          handle_error("pthread_create reader_thread failed\n");
103
104      return 0;
105  }
106
```

One Level Up Top Level

userspace/kp_symbol.h - ktap

Data types defined

- symbol_actor
- vaddr_t

Macros defined

- FIND_STAPSDT_NOTE
- FIND_SYMBOL

Source code

```
1  /*
2  * symbol.h - extract symbols from DSO.
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2013 Azat Khuzhin <a3at.mail@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22
23 #define FIND_SYMBOL 1
24 #define FIND_STAPSDT_NOTE 2
25
26 #ifndef NO_LIBELF
27
28 #include <gelf.h>
29 #include <sys/queue.h>
30
31 typedef GElf_Addr vaddr_t;
32 typedef int (*symbol_actor)(const char *name, vaddr_t addr, void *arg);
33
34 /**
35 * Parse all DSO symbols/sdt notes and all for every of them
36 * an actor.
37 *
38 * @exec - path to DSO
39 * @type - see FIND_*
40 * @symbol_actor - actor to call (callback)
41 * @arg - argument for @actor
42 *
43 * @return
44 * If there have errors, return negative value;
45 * No symbols found, return 0;
46 * Otherwise return number of dso symbols found
47 */
48 int
49 parse_dso_symbols(const char *exec, int type, symbol_actor actor, void *arg);
50 #endif
```

userspace/kp_symbol.c - ktap

Global variables defined

- dbg_bin_dir
- dbg_link_name

Functions defined

- dso_follow_debuglink
- dso_sdt_notes
- dso_symbols
- elf_section_by_name
- elf_symbols
- find_load_address
- parse_dso_symbols
- sdt_note_addr
- sdt_note_data
- sdt_note_name

Macros defined

- SDT_NOTE_COUNT
- SDT_NOTE_NAME
- SDT_NOTE_SCN
- SDT_NOTE_TYPE

Source code

```
1  /*
2   * symbol.c
3   *
4   * This file is part of ktap by Jovi Zhangwei.
5   *
6   * Copyright (C) 2013 Azat Khuzhin <a3at.mail@gmail.com>.
7   *
8   * ktap is free software; you can redistribute it and/or modify it
9   * under the terms and conditions of the GNU General Public License,
10   * version 2, as published by the Free Software Foundation.
11   *
12   * ktap is distributed in the hope it will be useful, but WITHOUT
13   * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14   * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15   * more details.
16   *
17   * You should have received a copy of the GNU General Public License along with
18   * this program; if not, write to the Free Software Foundation, Inc.,
19   * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20   */
21
```

```c
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <linux/limits.h>

#include <libelf.h>

#include "../include/ktap_types.h"
#include "kp_symbol.h"

const char *dbg_link_name = ".gnu_debuglink";
const char *dbg_bin_dir = "/usr/lib/debug";

static Elf_Scn *elf_section_by_name(Elf *elf, GElf_Ehdr *ep,
                    GElf_Shdr *shp, const char *name)
{
    Elf_Scn *scn = NULL;

    /* Elf is corrupted/truncated, avoid calling elf_strptr. */
    if (!elf_rawdata(elf_getscn(elf, ep->e_shstrndx), NULL))
        return NULL;

    while ((scn = elf_nextscn(elf, scn)) != NULL) {
        char *str;

        gelf_getshdr(scn, shp);
        str = elf_strptr(elf, ep->e_shstrndx, shp->sh_name);
        if (!strcmp(name, str))
            break;
    }

    return scn;
}

/**
* @return v_addr of "LOAD" program header, that have zero offset.
*/
static int find_load_address(Elf *elf, vaddr_t *load_address)
{
    GElf_Phdr phdr;
    size_t i, phdrnum;

    if (elf_getphdrnum(elf, &phdrnum))
        return -1;

    for (i = 0; i < phdrnum; i++) {
        if (gelf_getphdr(elf, i, &phdr) == NULL)
            return -1;

        if (phdr.p_type != PT_LOAD || phdr.p_offset != 0)
            continue;

        *load_address = phdr.p_vaddr;
        return 0;
    }

    /* cannot found load address */
    return -1;
}

static size_t elf_symbols(GElf_Shdr shdr)
{
    return shdr.sh_size / shdr.sh_entsize;
}

static int dso_symbols(Elf *elf, symbol_actor actor, void *arg)
{
    Elf_Data *elf_data = NULL;
    Elf_Scn *scn = NULL;
    GElf_Sym sym;
    GElf_Shdr shdr;
```

```
    int symbols_count = 0;
    vaddr_t load_address;

    if (find_load_address(elf, &load_address))
        return -1;

    while ((scn = elf_nextscn(elf, scn))) {
        int i;

        gelf_getshdr(scn, &shdr);

        if (shdr.sh_type != SHT_SYMTAB)
            continue;

        elf_data = elf_getdata(scn, elf_data);

        for (i = 0; i < elf_symbols(shdr); i++) {
            char *name;
            vaddr_t addr;
            int ret;

            gelf_getsym(elf_data, i, &sym);

            if (GELF_ST_TYPE(sym.st_info) != STT_FUNC)
                continue;

            name = elf_strptr(elf, shdr.sh_link, sym.st_name);
            addr = sym.st_value - load_address;

            ret = actor(name, addr, arg);
            if (ret)
                return ret;

            ++symbols_count;
        }
    }

    return symbols_count;
}

#define SDT_NOTE_TYPE 3
#define SDT_NOTE_COUNT 3
#define SDT_NOTE_SCN ".note.stapsdt"
#define SDT_NOTE_NAME "stapsdt"

static vaddr_t sdt_note_addr(Elf *elf, const char *data, size_t len, int type)
{
    vaddr_t vaddr;

    /*
     * Three addresses need to be obtained :
     * Marker location, address of base section and semaphore location
     */
    union {
        Elf64_Addr a64[3];
        Elf32_Addr a32[3];
    } buf;

    /*
     * dst and src are required for translation from file to memory
     * representation
     */
    Elf_Data dst = {
        .d_buf = &buf, .d_type = ELF_T_ADDR, .d_version = EV_CURRENT,
        .d_size = gelf_fsize(elf, ELF_T_ADDR, SDT_NOTE_COUNT, EV_CURRENT),
        .d_off = 0, .d_align = 0
    };

    Elf_Data src = {
        .d_buf = (void *) data, .d_type = ELF_T_ADDR,
        .d_version = EV_CURRENT, .d_size = dst.d_size, .d_off = 0,
        .d_align = 0
    };

    /* Check the type of each of the notes */
    if (type != SDT_NOTE_TYPE)
```

```
174         return 0;
175
176     if (len < dst.d_size + SDT_NOTE_COUNT)
177         return 0;
178
179     /* Translation from file representation to memory representation */
180     if (gelf_xlatetom(elf, &dst, &src,
181               elf_getident(elf, NULL)[EI_DATA]) == NULL)
182         return 0; /* TODO */
183
184     memcpy(&vaddr, &buf, sizeof(vaddr));
185
186     return vaddr;
187 }
188
189 static const char *sdt_note_name(Elf *elf, GElf_Nhdr *nhdr, const char *data)
190 {
191     const char *provider = data + gelf_fsize(elf,
192         ELF_T_ADDR, SDT_NOTE_COUNT, EV_CURRENT);
193     const char *name = (const char *)memchr(provider, '\0',
194         data + nhdr->n_descsz - provider);
195
196     if (name++ == NULL)
197         return NULL;
198
199     return name;
200 }
201
202 static const char *sdt_note_data(const Elf_Data *data, size_t off)
203 {
204     return ((data->d_buf) + off);
205 }
206
207 static int dso_sdt_notes(Elf *elf, symbol_actor actor, void *arg)
208 {
209     GElf_Ehdr ehdr;
210     Elf_Scn *scn = NULL;
211     Elf_Data *data;
212     GElf_Shdr shdr;
213     size_t shstrndx;
214     size_t next;
215     GElf_Nhdr nhdr;
216     size_t name_off, desc_off, offset;
217     vaddr_t vaddr = 0;
218     int symbols_count = 0;
219
220     if (gelf_getehdr(elf, &ehdr) == NULL)
221         return 0;
222     if (elf_getshdrstrndx(elf, &shstrndx) != 0)
223         return 0;
224
225     /*
226      * Look for section type = SHT_NOTE, flags = no SHF_ALLOC
227      * and name = .note.stapsdt
228      */
229     scn = elf_section_by_name(elf, &ehdr, &shdr, SDT_NOTE_SCN);
230     if (!scn)
231         return 0;
232     if (!(shdr.sh_type == SHT_NOTE) || (shdr.sh_flags & SHF_ALLOC))
233         return 0;
234
235     data = elf_getdata(scn, NULL);
236
237     for (offset = 0;
238         (next = gelf_getnote(data, offset, &nhdr, &name_off, &desc_off)) > 0;
239         offset = next) {
240         const char *name;
241         int ret;
242
243         if (nhdr.n_namesz != sizeof(SDT_NOTE_NAME) ||
244             memcmp(data->d_buf + name_off, SDT_NOTE_NAME,
245                 sizeof(SDT_NOTE_NAME)))
246             continue;
247
248         name = sdt_note_name(elf, &nhdr, sdt_note_data(data, desc_off));
249         if (!name)
```

```
            continue;

        vaddr = sdt_note_addr(elf, sdt_note_data(data, desc_off),
                    nhdr.n_descsz, nhdr.n_type);
        if (!vaddr)
            continue;

        ret = actor(name, vaddr, arg);
        if (ret)
            return ret;

        ++symbols_count;
    }

    return symbols_count;
}

int dso_follow_debuglink(Elf *elf,
             const char *orig_exec,
             int type,
             symbol_actor actor,
             void *arg)
{
    GElf_Ehdr ehdr;
    size_t shstrndx, orig_exec_dir_len;
    GElf_Shdr shdr;
    Elf_Scn *dbg_link_scn;
    Elf_Data *dbg_link_scn_data;
    char *dbg_link, *dbg_bin, *last_slash;
    int symbols_count;

    /* First try to find the .gnu_debuglink section in the binary. */
    if (gelf_getehdr(elf, &ehdr) == NULL)
        return 0;
    if (elf_getshdrstrndx(elf, &shstrndx) != 0)
        return 0;

    dbg_link_scn = elf_section_by_name(elf, &ehdr, &shdr, dbg_link_name);
    if (dbg_link_scn == NULL)
        return 0;

    /* Debug link section found, read of the content (only get the first
       string, no checksum checking atm). This is debug binary file name. */
    dbg_link_scn_data = elf_getdata(dbg_link_scn, NULL);
    if (dbg_link_scn_data == NULL ||
        dbg_link_scn_data->d_size <= 0 ||
        dbg_link_scn_data->d_buf == NULL)
        return 0;

    /* Now compose debug executable name */
    dbg_link = (char *)(dbg_link_scn_data->d_buf);
    dbg_bin = malloc(strlen(dbg_bin_dir) + 1 +
             strlen(orig_exec) + 1 +
             strlen(dbg_link) + 1);
    if (!dbg_bin)
        return 0;

    orig_exec_dir_len = PATH_MAX;
    last_slash = strrchr(orig_exec, '/');
    if (last_slash != NULL)
        orig_exec_dir_len = last_slash - orig_exec;

    sprintf(dbg_bin, "%s/%.*s/%s",
        dbg_bin_dir, (int)orig_exec_dir_len, orig_exec, dbg_link);

    /* Retry symbol seach with the debug binary */
    symbols_count = parse_dso_symbols(dbg_bin, type, actor, arg);

    free(dbg_bin);

    return symbols_count;
}

int parse_dso_symbols(const char *exec, int type, symbol_actor actor, void *arg)
{
    int symbols_count = 0;
```

```
326     Elf *elf;
327     int fd;
328
329     if (elf_version(EV_CURRENT) == EV_NONE)
330         return -1;
331
332     fd = open(exec, O_RDONLY);
333     if (fd < 0)
334         return -1;
335
336     elf = elf_begin(fd, ELF_C_READ, NULL);
337     if (elf) {
338         switch (type) {
339         case FIND_SYMBOL:
340             symbols_count = dso_symbols(elf, actor, arg);
341             if (symbols_count != 0)
342                 break;
343             /* If no symbols found, try in the debuglink binary. */
344             symbols_count = dso_follow_debuglink(elf,
345                                  exec,
346                                  type,
347                                  actor,
348                                  arg);
349             break;
350         case FIND_STAPSDT_NOTE:
351             symbols_count = dso_sdt_notes(elf, actor, arg);
352             break;
353         }
354
355         elf_end(elf);
356     }
357
358     close(fd);
359     return symbols_count;
360 }
```

One Level Up Top Level

runtime/lib_timer.c - ktap

Global variables defined

- timer_lib_funcs

Data types defined

- ktap_hrtimer

Functions defined

- do_tick_profile
- hrtimer_ktap_fn
- kp_exit_timers
- kp_lib_init_timer
- kplib_timer_profile
- kplib_timer_tick
- set_profile_timer
- set_tick_timer

Source code

```
/*
 * lib_timer.c - timer library support for ktap
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <linux/ctype.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/sched.h>
#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_obj.h"
#include "kp_vm.h"
#include "kp_events.h"

struct ktap_hrtimer {
    struct hrtimer timer;
    ktap_state_t *ks;
    ktap_func_t *fn;
```

```
    u64 ns;
    struct list_head list;
};

/*
* Currently ktap disallow tracing event in timer callback closure,
* that will corrupt ktap_state_t and ktap stack, because timer closure
* and event closure use same irq percpu ktap_state_t and stack.
* We can use a different percpu ktap_state_t and stack for timer purpuse,
* but that's don't bring any big value with cost on memory consuming.
*
* So just simply disable tracing in timer closure,
* get_recursion_context()/put_recursion_context() is used for this purpose.
*/
static enum hrtimer_restart hrtimer_ktap_fn(struct hrtimer *timer)
{
    struct ktap_hrtimer *t;
    ktap_state_t *ks;
    int rctx;

    rcu_read_lock_sched_notrace();

    t = container_of(timer, struct ktap_hrtimer, timer);
    rctx = get_recursion_context(t->ks);

    ks = kp_vm_new_thread(t->ks, rctx);
    set_func(ks->top, t->fn);
    incr_top(ks);
    kp_vm_call(ks, ks->top - 1, 0);
    kp_vm_exit_thread(ks);

    hrtimer_add_expires_ns(timer, t->ns);

    put_recursion_context(ks, rctx);
    rcu_read_unlock_sched_notrace();

    return HRTIMER_RESTART;
}

static int set_tick_timer(ktap_state_t *ks, u64 period, ktap_func_t *fn)
{
    struct ktap_hrtimer *t;

    t = kp_malloc(ks, sizeof(*t));
    if (unlikely(!t))
        return -ENOMEM;
    t->ks = ks;
    t->fn = fn;
    t->ns = period;

    INIT_LIST_HEAD(&t->list);
    list_add(&t->list, &(G(ks)->timers));

    hrtimer_init(&t->timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    t->timer.function = hrtimer_ktap_fn;
    hrtimer_start(&t->timer, ns_to_ktime(period), HRTIMER_MODE_REL);

    return 0;
}

static int set_profile_timer(ktap_state_t *ks, u64 period, ktap_func_t *fn)
{
    struct perf_event_attr attr;

     memset(&attr, 0, sizeof(attr));
     attr.type = PERF_TYPE_SOFTWARE;
     attr.config = PERF_COUNT_SW_CPU_CLOCK;
     attr.sample_type = PERF_SAMPLE_RAW | PERF_SAMPLE_TIME |
                PERF_SAMPLE_CPU | PERF_SAMPLE_PERIOD;
     attr.sample_period = period;
     attr.size = sizeof(attr);
     attr.disabled = 0;

     return kp_event_create(ks, &attr, NULL, NULL, fn);
 }

```

```c
static int do_tick_profile(ktap_state_t *ks, int is_tick)
{
    const char *str = kp_arg_checkstring(ks, 1);
    ktap_func_t *fn = kp_arg_checkfunction(ks, 2);
    const char *tmp;
    char interval_str[32] = {0};
    char suffix[10] = {0};
    int i = 0, ret, n;
    int factor;

    tmp = str;
    while (isdigit(*tmp))
        tmp++;

    strncpy(interval_str, str, tmp - str);
    if (kstrtoint(interval_str, 10, &n))
        goto error;

    strncpy(suffix, tmp, 9);
    while (suffix[i] != ' ' && suffix[i] != '\0')
        i++;

    suffix[i] = '\0';

    if (!strcmp(suffix, "s") || !strcmp(suffix, "sec"))
        factor = NSEC_PER_SEC;
    else if (!strcmp(suffix, "ms") || !strcmp(suffix, "msec"))
        factor = NSEC_PER_MSEC;
    else if (!strcmp(suffix, "us") || !strcmp(suffix, "usec"))
        factor = NSEC_PER_USEC;
    else
        goto error;

    if (is_tick)
        ret = set_tick_timer(ks, (u64)factor * n, fn);
    else
        ret = set_profile_timer(ks, (u64)factor * n, fn);

    return ret;

error:
    kp_error(ks, "cannot parse timer interval: %s\n", str);
    return -1;
}

/*
* tick-n probes fire on only one CPU per interval.
* valid time suffixes: sec/s, msec/ms, usec/us
*/
static int kplib_timer_tick(ktap_state_t *ks)
{
    /* timer.tick cannot be called in trace_end state */
    if (G(ks)->state != KTAP_RUNNING) {
        kp_error(ks,
             "timer.tick only can be called in RUNNING state\n");
        return -1;
    }

    return do_tick_profile(ks, 1);
}

/*
* A profile-n probe fires every fixed interval on every CPU
* valid time suffixes: sec/s, msec/ms, usec/us
*/
static int kplib_timer_profile(ktap_state_t *ks)
{
    /* timer.profile cannot be called in trace_end state */
    if (G(ks)->state != KTAP_RUNNING) {
        kp_error(ks,
             "timer.profile only can be called in RUNNING state\n");
        return -1;
    }

    return do_tick_profile(ks, 0);
}
```

```
188
189 void kp_exit_timers(ktap_state_t *ks)
190 {
191     struct ktap_hrtimer *t, *tmp;
192     struct list_head *timers_list = &(G(ks)->timers);
193
194     list_for_each_entry_safe(t, tmp, timers_list, list) {
195         hrtimer_cancel(&t->timer);
196         kp_free(ks, t);
197     }
198 }
199
200 static const ktap_libfunc_t timer_lib_funcs[] = {
201     {"profile",     kplib_timer_profile},
202     {"tick",    kplib_timer_tick},
203     {NULL}
204 };
205
206 int kp_lib_init_timer(ktap_state_t *ks)
207 {
208     return kp_vm_register_lib(ks, "timer", timer_lib_funcs);
209 }
210
```

One Level Up Top Level

runtime/ffi/ffi_symbol.c - ktap

Functions defined

Source code

```
1  /*
2  * ffi_symbol.c - ktapvm kernel module ffi symbol submodule
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22
23 #include "../../include/ktap_types.h"
24 #include "../../include/ktap_ffi.h"
25 #include "../ktap.h"
26 #include "../kp_vm.h"
27 #include "../kp_obj.h"
28 #include "../kp_str.h"
29 #include "../kp_tab.h"
30
31 static inline csymbol *get_csym_arr(ktap_state_t *ks)
32 {
33     return G(ks)->ffis.csym_arr;
34 }
35
36 static inline int get_csym_nr(ktap_state_t *ks)
37 {
38     return G(ks)->ffis.csym_nr;
```

```
}

static inline void set_csym_arr(ktap_state_t *ks, csymbol *csym)
{
    G(ks)->ffis.csym_arr = csym;
}

static inline void set_csym_nr(ktap_state_t *ks, int nr)
{
    G(ks)->ffis.csym_nr = nr;
}


static inline ktap_tab_t *get_ffi_ctable(ktap_state_t *ks)
{
    return G(ks)->ffis.ctable;
}

static int setup_ffi_ctable(ktap_state_t *ks)
{
    ktap_val_t ffi_name, ffi_lib_name, ffi_mt;
    const ktap_val_t *gt, *ffit;
    ktap_tab_t *registry;

    gt = kp_tab_getint(hvalue(&G(ks)->registry), KTAP_RIDX_GLOBALS);

    G(ks)->ffis.ctable = kp_tab_new(ks, 0, 512);
    if (!G(ks)->ffis.ctable)
        return -1;

    /* get global["ffi"] */
    set_string(&ffi_name, kp_str_new(ks, "ffi"));
    registry = hvalue(gt);
    ffit = kp_tab_get(ks, registry, &ffi_name);
    /* insert ffi C table to ffi table */
    set_table(&ffi_mt, get_ffi_ctable(ks));
    set_string(&ffi_lib_name, kp_str_new(ks, "C"));
    registry = hvalue(ffit);
    kp_tab_setvalue(ks, registry, &ffi_lib_name, &ffi_mt);

    return 0;
}

inline csymbol *ffi_get_csym_by_id(ktap_state_t *ks, int id)
{
    return &(get_csym_arr(ks)[id]);
}

csymbol_id ffi_get_csym_id(ktap_state_t *ks, char *name)
{
    int i;

    for (i = 0; i < get_csym_nr(ks); i++) {
        if (!strcmp(name, csym_name(ffi_get_csym_by_id(ks, i)))) {
            return i;
        }
    }

    kp_error(ks, "Cannot find csymbol with name %s\n", name);
    return 0;
}

 static void add_ffi_func_to_ctable(ktap_state_t *ks, csymbol_id id)
 {
     ktap_val_t func_name, fv;
     ktap_cdata_t *cd;
     csymbol *cs;

     /* push cdata to ctable */
     set_cdata(&fv, kp_obj_newobject(ks, KTAP_TYPE_CDATA, sizeof(ktap_cdata_t),
                     NULL));
     cd = cdvalue(&fv);
     cd_set_csym_id(cd, id);

     cs = id_to_csym(ks, id);
     set_string(&func_name, kp_str_new(ks, csym_name(cs)));
```

```
115     kp_tab_setvalue(ks, get_ffi_ctable(ks), &func_name, &fv);
116 }
117
118 static int setup_ffi_symbol_table(ktap_state_t *ks)
119 {
120     int i;
121     csymbol *cs;
122
123     if (setup_ffi_ctable(ks))
124         return -1;
125
126     /* push all functions to ctable */
127     for (i = 0; i < get_csym_nr(ks); i++) {
128         cs = &get_csym_arr(ks)[i];
129         switch (cs->type) {
130         case FFI_FUNC:
131             kp_verbose_printf(ks, "[%d] loading C function %s\n",
132                     i, csym_name(cs));
133             add_ffi_func_to_ctable(ks, i);
134             kp_verbose_printf(ks, "%s loaded\n", csym_name(cs));
135             break;
136         case FFI_STRUCT:
137         case FFI_UNION:
138             break;
139         default:
140             break;
141         }
142     }
143
144     return 0;
145 }
146
147 void ffi_free_symbols(ktap_state_t *ks)
148 {
149     int i;
150     csymbol_id *arg_ids;
151     csymbol *cs;
152
153     if (!get_csym_arr(ks))
154         return;
155
156     for (i = 0; i < get_csym_nr(ks); i++) {
157         cs = &get_csym_arr(ks)[i];
158         switch (csym_type(cs)) {
159         case FFI_FUNC:
160             arg_ids = csym_func_arg_ids(cs);
161             if (arg_ids)
162                 kp_free(ks, arg_ids);
163             break;
164         case FFI_STRUCT:
165         case FFI_UNION:
166             /*@TODO finish this  20.11 2013 (houqp)*/
167             break;
168         default:
169             break;
170         }
171     }
172
173     kp_free(ks, get_csym_arr(ks));
174 }
175
176 int ffi_set_csym_arr(ktap_state_t *ks, int cs_nr, csymbol *new_arr)
177 {
178     set_csym_nr(ks, cs_nr);
179     set_csym_arr(ks, new_arr);
180     return setup_ffi_symbol_table(ks);
181 }
182
```

One Level Up Top Level

runtime/kp_mempool.c - ktap

Functions defined

- kp_mempool_alloc
- kp_mempool_destroy
- kp_mempool_init

Source code

```
1  /*
2  * kp_mempool.c - ktap memory pool, service for string allocation
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22 #include "../include/ktap_types.h"
23 #include "kp_obj.h"
24 #include "kp_str.h"
25
26 #include <linux/ctype.h>
27 #include <linux/module.h>
28 #include "ktap.h"
29
30
31 /*
32 * allocate memory from mempool, the allocated memory will be free
33 * util ktap exit.
34 * TODO: lock-free allocation
35 */
36 void *kp_mempool_alloc(ktap_state_t *ks, int size)
37 {
38     ktap_global_state_t *g = G(ks);
39     void *mempool = g->mempool;
40     void *freepos = g->mp_freepos;
41     void *addr;
42     unsigned long flags;
43
44     local_irq_save(flags);
45     arch_spin_lock(&g->mp_lock);
46
47     if (unlikely((unsigned long)((char *)freepos + size)) >
48               (unsigned long)((char *)mempool + g->mp_size)) {
49         addr = NULL;
50         goto out;
51     }
52
53     addr = freepos;
54     g->mp_freepos = (char *)freepos + size;
55 out:
56
57     arch_spin_unlock(&g->mp_lock);
58     local_irq_restore(flags);
```

```
59      return addr;
60  }
61
62  /*
63  * destroy mempool.
64  */
65  void kp_mempool_destroy(ktap_state_t *ks)
66  {
67      ktap_global_state_t *g = G(ks);
68
69      if (!g->mempool)
70          return;
71
72      vfree(g->mempool);
73      g->mempool = NULL;
74      g->mp_freepos = NULL;
75      g->mp_size = 0;
76  }
77
78  /*
79  * pre-allocate size Kbytes memory pool.
80  */
81  int kp_mempool_init(ktap_state_t *ks, int size)
82  {
83      ktap_global_state_t *g = G(ks);
84
85      g->mempool = vmalloc(size * 1024);
86      if (!g->mempool)
87          return -ENOMEM;
88
89      g->mp_freepos = g->mempool;
90      g->mp_size = size * 1024;
91      g->mp_lock = (arch_spinlock_t)__ARCH_SPIN_LOCK_UNLOCKED;
92      return 0;
93  }
94
```

One Level Up Top Level

runtime/lib_base.c - ktap

Global variables defined

- base_lib_funcs

Functions defined

- kp_lib_init_base
- kplib_arch
- kplib_curr_taskinfo
- kplib_delete
- kplib_exit
- kplib_gettimeofday_ms
- kplib_gettimeofday_ns
- kplib_gettimeofday_s
- kplib_gettimeofday_us
- kplib_in_interrupt
- kplib_in_iowait
- kplib_ipof
- kplib_kernel_string
- kplib_kernel_v
- kplib_len
- kplib_num_cpus
- kplib_pairs
- kplib_print
- kplib_print_hist
- kplib_print_trace_clock
- kplib_printf
- kplib_stack
- kplib_stack
- kplib_stringof
- kplib_user_string

Macros defined

- HISTOGRAM_DEFAULT_TOP_NUM

- RET_VALUE
- RET_VALUE

Source code

```
/*
 * lib_base.c - base library
 *
 * Caveat: all kernel funtion called by ktap library have to be lock free,
 * otherwise system will deadlock.
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <linux/version.h>
#include <linux/hardirq.h>
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/utsname.h>
#include <linux/time.h>
#include <linux/clocksource.h>
#include <linux/ring_buffer.h>
#include <linux/stacktrace.h>
#include <linux/cred.h>
#if LINUX_VERSION_CODE >= KERNEL_VERSION(3, 5, 0)
#include <linux/uidgid.h>
#endif
#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_obj.h"
#include "kp_str.h"
#include "kp_tab.h"
#include "kp_transport.h"
#include "kp_events.h"
#include "kp_vm.h"

static int kplib_print(ktap_state_t *ks)
{
    int i;
    int n = kp_arg_nr(ks);

    for (i = 1; i <= n; i++) {
        ktap_val_t *arg = kp_arg(ks, i);
        if (i > 1)
            kp_puts(ks, "\t");
        kp_obj_show(ks, arg);
    }

    kp_puts(ks, "\n");
    return 0;
}

/* don't engage with intern string in printf, use buffer directly */
static int kplib_printf(ktap_state_t *ks)
{
    struct trace_seq *seq;
```

```
69
70      preempt_disable_notrace();
71
72      seq = kp_this_cpu_print_buffer(ks);
73      trace_seq_init(seq);
74
75      if (kp_str_fmt(ks, seq))
76          goto out;
77
78      seq->buffer[seq->len] = '\0';
79      kp_transport_write(ks, seq->buffer, seq->len + 1);
80
81  out:
82      preempt_enable_notrace();
83      return 0;
84  }
85
86  #define HISTOGRAM_DEFAULT_TOP_NUM    20
87
88  static int kplib_print_hist(ktap_state_t *ks)
89  {
90      int n ;
91
92      kp_arg_check(ks, 1, KTAP_TTAB);
93      n = kp_arg_checkoptnumber(ks, 2, HISTOGRAM_DEFAULT_TOP_NUM);
94
95      n = min(n, 1000);
96      n = max(n, HISTOGRAM_DEFAULT_TOP_NUM);
97
98      kp_tab_print_hist(ks, hvalue(kp_arg(ks, 1)), n);
99
100      return 0;
101  }
102
103  static int kplib_pairs(ktap_state_t *ks)
104  {
105      kp_arg_check(ks, 1, KTAP_TTAB);
106
107      set_cfunc(ks->top++, (ktap_cfunction)kp_tab_next);
108      set_table(ks->top++, hvalue(kp_arg(ks, 1)));
109      set_nil(ks->top++);
110      return 3;
111  }
112
113  static int kplib_len(ktap_state_t *ks)
114  {
115      int len = kp_obj_len(ks, kp_arg(ks, 1));
116
117      if (len < 0)
118          return -1;
119
120      set_number(ks->top, len);
121      incr_top(ks);
122      return 1;
123  }
124
125  static int kplib_delete(ktap_state_t *ks)
126  {
127      kp_arg_check(ks, 1, KTAP_TTAB);
128      kp_tab_clear(hvalue(kp_arg(ks, 1)));
129      return 0;
130  }
131
132  #ifdef CONFIG_STACKTRACE
133  static int kplib_stack(ktap_state_t *ks)
134  {
135      uint16_t skip, depth = 10;
136
137      depth = kp_arg_checkoptnumber(ks, 1, 10); /* default as 10 */
138      depth = min_t(uint16_t, depth, KP_MAX_STACK_DEPTH);
139      skip = kp_arg_checkoptnumber(ks, 2, 10); /* default as 10 */
140
141      set_kstack(ks->top, depth, skip);
142      incr_top(ks);
143      return 1;
144  }
```

```
#else
static int kplib_stack(ktap_state_t *ks)
{
    kp_error(ks, "Please enable CONFIG_STACKTRACE before call stack()\n");
    return -1;
}
#endif


extern unsigned long long ns2usecs(cycle_t nsec);
static int kplib_print_trace_clock(ktap_state_t *ks)
{
    unsigned long long t;
    unsigned long secs, usec_rem;
    u64 timestamp;

    /* use ring buffer's timestamp */
    timestamp = ring_buffer_time_stamp(G(ks)->buffer, smp_processor_id());

    t = ns2usecs(timestamp);
    usec_rem = do_div(t, USEC_PER_SEC);
    secs = (unsigned long)t;

    kp_printf(ks, "%5lu.%06lu\n", secs, usec_rem);
    return 0;
}

static int kplib_num_cpus(ktap_state_t *ks)
{
    set_number(ks->top, num_online_cpus());
    incr_top(ks);
    return 1;
}

/* TODO: intern string firstly */
static int kplib_arch(ktap_state_t *ks)
{
    ktap_str_t *ts = kp_str_newz(ks, utsname()->machine);
    if (unlikely(!ts))
        return -1;

    set_string(ks->top, ts);
    incr_top(ks);
    return 1;
}

/* TODO: intern string firstly */
static int kplib_kernel_v(ktap_state_t *ks)
{
    ktap_str_t *ts = kp_str_newz(ks, utsname()->release);
    if (unlikely(!ts))
        return -1;

    set_string(ks->top, ts);
    incr_top(ks);
    return 1;
}

static int kplib_kernel_string(ktap_state_t *ks)
{
    unsigned long addr = kp_arg_checknumber(ks, 1);
    char str[256] = {0};
    ktap_str_t *ts;
    char *ret;

    ret = strncpy((void *)str, (const void *)addr, 256);
    (void) &ret;  /* Silence compiler warning. */
    str[255] = '\0';

    ts = kp_str_newz(ks, str);
    if (unlikely(!ts))
        return -1;

    set_string(ks->top, ts);
    incr_top(ks);
    return 1;
```

```
}

static int kplib_user_string(ktap_state_t *ks)
{
    unsigned long addr = kp_arg_checknumber(ks, 1);
    char str[256] = {0};
    ktap_str_t *ts;
    int ret;

    pagefault_disable();
    ret = __copy_from_user_inatomic((void *)str, (const void *)addr, 256);
    (void) &ret;  /* Silence compiler warning. */
    pagefault_enable();
    str[255] = '\0';

    ts = kp_str_newz(ks, str);
    if (unlikely(!ts))
        return -1;

    set_string(ks->top, ts);
    incr_top(ks);
    return 1;
}

static int kplib_stringof(ktap_state_t *ks)
{
    ktap_val_t *v = kp_arg(ks, 1);
    const ktap_str_t *ts = NULL;

    if (itype(v) == KTAP_TEVENTSTR) {
        ts = kp_event_stringify(ks);
    } else if (itype(v) == KTAP_TKIP) {
        char str[KSYM_SYMBOL_LEN];

        SPRINT_SYMBOL(str, nvalue(v));
        ts = kp_str_newz(ks, str);
    }

    if (unlikely(!ts))
        return -1;

    set_string(ks->top++, ts);
    return 1;
}

static int kplib_ipof(ktap_state_t *ks)
{
    unsigned long addr = kp_arg_checknumber(ks, 1);

    set_ip(ks->top++, addr);
    return 1;
}

static int kplib_gettimeofday_ns(ktap_state_t *ks)
{
    set_number(ks->top, gettimeofday_ns());
    incr_top(ks);

    return 1;
}

static int kplib_gettimeofday_us(ktap_state_t *ks)
{
    set_number(ks->top, gettimeofday_ns() / NSEC_PER_USEC);
    incr_top(ks);

    return 1;
}

static int kplib_gettimeofday_ms(ktap_state_t *ks)
{
    set_number(ks->top, gettimeofday_ns() / NSEC_PER_MSEC);
    incr_top(ks);

    return 1;
}
```

```
297
298 static int kplib_gettimeofday_s(ktap_state_t *ks)
299 {
300     set_number(ks->top, gettimeofday_ns() / NSEC_PER_SEC);
301     incr_top(ks);
302
303     return 1;
304 }
305
306 /*
307 * use gdb to get field offset of struct task_struct, for example:
308 *
309 * gdb vmlinux
310 * (gdb)p &(((struct task_struct *)0).prio)
311 */
312 static int kplib_curr_taskinfo(ktap_state_t *ks)
313 {
314     int offset = kp_arg_checknumber(ks, 1);
315     int fetch_bytes  = kp_arg_checkoptnumber(ks, 2, 4); /* fetch 4 bytes */
316
317     if (offset >= sizeof(struct task_struct)) {
318         set_nil(ks->top++);
319         kp_error(ks, "access out of bound value of task_struct\n");
320         return 1;
321     }
322
323 #define RET_VALUE ((unsigned long)current + offset)
324
325     switch (fetch_bytes) {
326     case 4:
327         set_number(ks->top, *(unsigned int *)RET_VALUE);
328         break;
329     case 8:
330         set_number(ks->top, *(unsigned long *)RET_VALUE);
331         break;
332     default:
333         kp_error(ks, "unsupported fetch bytes in curr_task_info\n");
334         set_nil(ks->top);
335         break;
336     }
337
338 #undef RET_VALUE
339
340     incr_top(ks);
341     return 1;
342 }
343
344 /*
345 * This built-in function mainly purpose scripts/schedule/schedtimes.kp
346 */
347 static int kplib_in_iowait(ktap_state_t *ks)
348 {
349     set_number(ks->top, current->in_iowait);
350     incr_top(ks);
351
352     return 1;
353 }
354
355 static int kplib_in_interrupt(ktap_state_t *ks)
356 {
357     int ret = in_interrupt();
358
359     set_number(ks->top, ret);
360     incr_top(ks);
361     return 1;
362 }
363
364 static int kplib_exit(ktap_state_t *ks)
365 {
366     kp_vm_try_to_exit(ks);
367
368     /* do not execute bytecode any more in this thread */
369     return -1;
370 }
371
372 static const ktap_libfunc_t base_lib_funcs[] = {
```

```
373     {"print", kplib_print},
374     {"printf", kplib_printf},
375     {"print_hist", kplib_print_hist},
376
377     {"pairs", kplib_pairs},
378     {"len", kplib_len},
379     {"delete", kplib_delete},
380
381     {"stack", kplib_stack},
382     {"print_trace_clock", kplib_print_trace_clock},
383
384     {"num_cpus", kplib_num_cpus},
385     {"arch", kplib_arch},
386     {"kernel_v", kplib_kernel_v},
387     {"kernel_string", kplib_kernel_string},
388     {"user_string", kplib_user_string},
389     {"stringof", kplib_stringof},
390     {"ipof", kplib_ipof},
391
392     {"gettimeofday_ns", kplib_gettimeofday_ns},
393     {"gettimeofday_us", kplib_gettimeofday_us},
394     {"gettimeofday_ms", kplib_gettimeofday_ms},
395     {"gettimeofday_s", kplib_gettimeofday_s},
396
397     {"curr_taskinfo", kplib_curr_taskinfo},
398
399     {"in_iowait", kplib_in_iowait},
400     {"in_interrupt", kplib_in_interrupt},
401
402     {"exit", kplib_exit},
403     {NULL}
404 };
405
406 int kp_lib_init_base(ktap_state_t *ks)
407 {
408     return kp_vm_register_lib(ks, NULL, base_lib_funcs);
409 }
```

One Level Up Top Level

runtime/lib_kdebug.c - ktap

Global variables defined

- kdebug_lib_funcs

Functions defined

- kp_lib_init_kdebug
- kplib_kdebug_kprobe
- kplib_kdebug_trace_by_id
- kplib_kdebug_trace_end

Source code

```
1  /*
2  * lib_kdebug.c - kdebug library support for ktap
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10  * version 2, as published by the Free Software Foundation.
11  *
12  * ktap is distributed in the hope it will be useful, but WITHOUT
13  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15  * more details.
16  *
17  * You should have received a copy of the GNU General Public License along with
18  * this program; if not, write to the Free Software Foundation, Inc.,
19  * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20  */
21
22 #include <linux/module.h>
23 #include <linux/ctype.h>
24 #include <linux/slab.h>
25 #include <linux/version.h>
26 #include <linux/ftrace_event.h>
27 #include "../include/ktap_types.h"
28 #include "ktap.h"
29 #include "kp_obj.h"
30 #include "kp_str.h"
31 #include "kp_transport.h"
32 #include "kp_vm.h"
33 #include "kp_events.h"
34
35 /**
36  * function kdebug.trace_by_id
37  *
38  * @uaddr: userspace address refer to ktap_eventdesc_t
39  * @closure
40  */
41 static int kplib_kdebug_trace_by_id(ktap_state_t *ks)
42 {
43     unsigned long uaddr = kp_arg_checknumber(ks, 1);
44     ktap_func_t *fn = kp_arg_checkfunction(ks, 2);
45     struct task_struct *task = G(ks)->trace_task;
46     ktap_eventdesc_t eventsdesc;
47     char *filter = NULL;
48     int *id_arr;
49     int ret, i;
50
```

```
    if (G(ks)->mainthread != ks) {
        kp_error(ks,
            "kdebug.trace_by_id only can be called in mainthread\n");
        return -1;
    }

    /* kdebug.trace_by_id cannot be called in trace_end state */
    if (G(ks)->state != KTAP_RUNNING) {
        kp_error(ks,
            "kdebug.trace_by_id only can be called in RUNNING state\n");
        return -1;
    }

    /* copy ktap_eventdesc_t from userspace */
    ret = copy_from_user(&eventsdesc, (void *)uaddr,
                sizeof(ktap_eventdesc_t));
    if (ret < 0)
        return -1;

    if (eventsdesc.filter) {
        int len;

        len = strlen_user(eventsdesc.filter);
        if (len > 0x1000)
            return -1;

        filter = kmalloc(len + 1, GFP_KERNEL);
        if (!filter)
            return -1;

        /* copy filter string from userspace */
        if (strncpy_from_user(filter, eventsdesc.filter, len) < 0) {
            kfree(filter);
            return -1;
        }
    }

    id_arr = kmalloc(eventsdesc.nr * sizeof(int), GFP_KERNEL);
    if (!id_arr) {
        kfree(filter);
        return -1;
    }

    /* copy all event id from userspace */
    ret = copy_from_user(id_arr, eventsdesc.id_arr,
                eventsdesc.nr * sizeof(int));
    if (ret < 0) {
        kfree(filter);
        kfree(id_arr);
         return -1;
     }

     fn = clvalue(kp_arg(ks, 2));

     for (i = 0; i < eventsdesc.nr; i++) {
         struct perf_event_attr attr;

         cond_resched();

         if (signal_pending(current)) {
             flush_signals(current);
             kfree(filter);
             kfree(id_arr);
             return -1;
         }

         memset(&attr, 0, sizeof(attr));
         attr.type = PERF_TYPE_TRACEPOINT;
         attr.config = id_arr[i];
         attr.sample_type = PERF_SAMPLE_RAW | PERF_SAMPLE_TIME |
                    PERF_SAMPLE_CPU | PERF_SAMPLE_PERIOD;
         attr.sample_period = 1;
         attr.size = sizeof(attr);
         attr.disabled = 0;

         /* register event one by one */
```

```
        ret = kp_event_create(ks, &attr, task, filter, fn);
        if (ret < 0)
            break;
    }

    kfree(filter);
    kfree(id_arr);
    return 0;
}

static int kplib_kdebug_trace_end(ktap_state_t *ks)
{
    /* trace_end_closure will be called when ktap main thread exit */
    G(ks)->trace_end_closure = kp_arg_checkfunction(ks, 1);
    return 0;
}

#if 0
static int kplib_kdebug_tracepoint(ktap_state_t *ks)
{
    const char *event_name = kp_arg_checkstring(ks, 1);
    ktap_func_t *fn = kp_arg_checkfunction(ks, 2);

    if (G(ks)->mainthread != ks) {
        kp_error(ks,
            "kdebug.tracepoint only can be called in mainthread\n");
        return -1;
    }

    /* kdebug.tracepoint cannot be called in trace_end state */
    if (G(ks)->state != KTAP_RUNNING) {
        kp_error(ks,
            "kdebug.tracepoint only can be called in RUNNING state\n");
        return -1;
    }

    return kp_event_create_tracepoint(ks, event_name, fn);
}
#endif

static int kplib_kdebug_kprobe(ktap_state_t *ks)
{
    const char *event_name = kp_arg_checkstring(ks, 1);
    ktap_func_t *fn = kp_arg_checkfunction(ks, 2);

    if (G(ks)->mainthread != ks) {
        kp_error(ks,
            "kdebug.kprobe only can be called in mainthread\n");
        return -1;
    }

    /* kdebug.kprobe cannot be called in trace_end state */
    if (G(ks)->state != KTAP_RUNNING) {
        kp_error(ks,
            "kdebug.kprobe only can be called in RUNNING state\n");
        return -1;
    }

    return kp_event_create_kprobe(ks, event_name, fn);
}
static const ktap_libfunc_t kdebug_lib_funcs[] = {
    {"trace_by_id", kplib_kdebug_trace_by_id},
    {"trace_end", kplib_kdebug_trace_end},

#if 0
    {"tracepoint", kplib_kdebug_tracepoint},
#endif
    {"kprobe", kplib_kdebug_kprobe},
    {NULL}
};

int kp_lib_init_kdebug(ktap_state_t *ks)
{
    return kp_vm_register_lib(ks, "kdebug", kdebug_lib_funcs);
}

```

runtime/lib_ansi.c - ktap

Global variables defined

- ansi_lib_funcs

Functions defined

- kp_lib_init_ansi
- kplib_ansi_clear_screen
- kplib_ansi_new_line
- kplib_ansi_reset_color
- kplib_ansi_set_color
- kplib_ansi_set_color2
- kplib_ansi_set_color3

Source code

```
/*
 * lib_ansi.c - ANSI escape sequences library
 *
 * http://en.wikipedia.org/wiki/ANSI_escape_code
 *
 * This file is part of ktap by Jovi Zhangwei.
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "../include/ktap_types.h"
#include "ktap.h"
#include "kp_vm.h"

/**
 * function ansi.clear_screen - Move cursor to top left and clear screen.
 *
 * Description: Sends ansi code for moving cursor to top left and then the
 * ansi code for clearing the screen from the cursor position to the end.
 */

static int kplib_ansi_clear_screen(ktap_state_t *ks)
{
    kp_printf(ks, "\033[1;1H\033[J");
    return 0;
}

/**
 * function ansi.set_color - Set the ansi Select Graphic Rendition mode.
 * @fg: Foreground color to set.
```

```
 *
 * Description: Sends ansi code for Select Graphic Rendition mode for the
 * given forground color. Black (30), Blue (34), Green (32), Cyan (36),
 * Red (31), Purple (35), Brown (33), Light Gray (37).
 */

static int kplib_ansi_set_color(ktap_state_t *ks)
{
    int fg = kp_arg_checknumber(ks, 1);

    kp_printf(ks, "\033[%dm", fg);
    return 0;
}

/**
 * function ansi.set_color2 - Set the ansi Select Graphic Rendition mode.
 * @fg: Foreground color to set.
 * @bg: Background color to set.
 *
 * Description: Sends ansi code for Select Graphic Rendition mode for the
 * given forground color, Black (30), Blue (34), Green (32), Cyan (36),
 * Red (31), Purple (35), Brown (33), Light Gray (37) and the given
 * background color, Black (40), Red (41), Green (42), Yellow (43),
 * Blue (44), Magenta (45), Cyan (46), White (47).
 */
static int kplib_ansi_set_color2(ktap_state_t *ks)
{
    int fg = kp_arg_checknumber(ks, 1);
    int bg = kp_arg_checknumber(ks, 2);

    kp_printf(ks, "\033[%d;%dm", fg, bg);
    return 0;
}

/**
 * function ansi.set_color3 - Set the ansi Select Graphic Rendition mode.
 * @fg: Foreground color to set.
 * @bg: Background color to set.
 * @attr: Color attribute to set.
 *
 * Description: Sends ansi code for Select Graphic Rendition mode for the
 * given forground color, Black (30), Blue (34), Green (32), Cyan (36),
 * Red (31), Purple (35), Brown (33), Light Gray (37), the given
 * background color, Black (40), Red (41), Green (42), Yellow (43),
 * Blue (44), Magenta (45), Cyan (46), White (47) and the color attribute
 * All attributes off (0), Intensity Bold (1), Underline Single (4),
 * Blink Slow (5), Blink Rapid (6), Image Negative (7).
 */
static int kplib_ansi_set_color3(ktap_state_t *ks)
{
    int fg = kp_arg_checknumber(ks, 1);
    int bg = kp_arg_checknumber(ks, 2);
    int attr = kp_arg_checknumber(ks, 3);

    if (attr)
        kp_printf(ks, "\033[%d;%d;%dm", fg, bg, attr);
     else
         kp_printf(ks, "\033[%d;%dm", fg, bg);

     return 0;
 }

 /**
  * function ansi.reset_color - Resets Select Graphic Rendition mode.
  *
  * Description: Sends ansi code to reset foreground, background and color
  * attribute to default values.
  */
 static int kplib_ansi_reset_color(ktap_state_t *ks)
 {
     kp_printf(ks, "\033[0;0m");
     return 0;
 }

 /**
  * function ansi.new_line - Move cursor to new line.
```

```
 *
 * Description: Sends ansi code new line.
 */
static int kplib_ansi_new_line (ktap_state_t *ks)
{
    kp_printf(ks, "\12");
    return 0;
}

static const ktap_libfunc_t ansi_lib_funcs[] = {
    {"clear_screen", kplib_ansi_clear_screen},
    {"set_color", kplib_ansi_set_color},
    {"set_color2", kplib_ansi_set_color2},
    {"set_color3", kplib_ansi_set_color3},
    {"reset_color", kplib_ansi_reset_color},
    {"new_line", kplib_ansi_new_line},
    {NULL}
};

int kp_lib_init_ansi(ktap_state_t *ks)
{
    return kp_vm_register_lib(ks, "ansi", ansi_lib_funcs);
}
```

One Level Up Top Level

runtime/lib_ffi.c - ktap

Global variables defined

- ffi_lib_funcs

Functions defined

- kp_lib_init_ffi
- kplib_ffi_cast
- kplib_ffi_new
- kplib_ffi_sizeof

Source code

```
1  /*
2  * lib_ffi.c - FFI library
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22 #include "../include/ktap_types.h"
23 #include "../include/ktap_ffi.h"
24 #include "ktap.h"
25 #include "kp_vm.h"
26
27
28 static int kplib_ffi_new(ktap_state_t *ks)
29 {
30     int n = kp_arg_nr(ks);
31     csymbol_id cs_id = kp_arg_checknumber(ks, 1);
32     int array_size = kp_arg_checknumber(ks, 2);
33     int is_array = kp_arg_checknumber(ks, 3);
34     ktap_cdata_t *cd;
35
36     if (unlikely(n != 3)) {
37         /* this is not likely to happen since ffi.new arguments are
38          * generated by compiler */
39         set_nil(ks->top++);
40         kp_error(ks, "wrong number of arguments\n");
41         return 1;
42     }
43
44     if (unlikely(cs_id > max_csym_id(ks)))
45         kp_error(ks, "invalid csymbol id\n");
46
47     kp_verbose_printf(ks, "ffi.new symbol %s with length %d\n",
48             id_to_csym(ks, cs_id)->name, array_size);
49
50     if (is_array)
```

```
51          cd = kp_cdata_new_ptr(ks, NULL, array_size, cs_id, 1);
52      else
53          cd = kp_cdata_new_by_id(ks, NULL, cs_id);
54      set_cdata(ks->top, cd);
55      incr_top(ks);
56
57      return 1;
58  }
59
60  static int kplib_ffi_cast(ktap_state_t *ks)
61  {
62      int n = kp_arg_nr(ks);
63      csymbol_id cs_id = kp_arg_checknumber(ks, 1);
64      unsigned long addr = kp_arg_checknumber(ks, 2);
65      ktap_cdata_t *cd;
66
67      if (unlikely(n != 2)) {
68          /* this is not likely to happen since ffi.cast arguments are
69           * generated by compiler */
70          set_nil(ks->top++);
71          kp_error(ks, "wrong number of arguments\n");
72          return 1;
73      }
74
75      if (unlikely(cs_id > max_csym_id(ks)))
76          kp_error(ks, "invalid csymbol id\n");
77
78      cd = kp_cdata_new_record(ks, (void *)addr, cs_id);
79      set_cdata(ks->top, cd);
80      incr_top(ks);
81      return 1;
82  }
83
84  static int kplib_ffi_sizeof(ktap_state_t *ks)
85  {
86      /*@TODO finish this  08.11 2013 (houqp)*/
87      return 0;
88  }
89
90  static const ktap_libfunc_t ffi_lib_funcs[] = {
91      {"sizeof", kplib_ffi_sizeof},
92      {"new", kplib_ffi_new},
93      {"cast", kplib_ffi_cast},
94      {NULL}
95  };
96
97  int kp_lib_init_ffi(ktap_state_t *ks)
98  {
99      return kp_vm_register_lib(ks, "ffi", ffi_lib_funcs);
100 }
```

One Level Up Top Level

runtime/ffi/cdata.c - ktap

Functions defined

Source code

```
/*
 * cdata.c - support functions for ktap_cdata_t
 *
 * This file is part of ktap by Jovi Zhangwei
 *
 * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
 *
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */


#include "../../include/ktap_types.h"
#include "../../include/ktap_ffi.h"
#include "../kp_obj.h"

ktap_cdata_t *kp_cdata_new(ktap_state_t *ks, csymbol_id id)
{
    ktap_cdata_t *cd;

    cd = &kp_obj_newobject(ks, KTAP_TYPE_CDATA, sizeof(ktap_cdata_t), NULL)->cd;
```

```
    cd_set_csym_id(cd, id);

    return cd;
}

ktap_cdata_t *kp_cdata_new_number(ktap_state_t *ks, void *val, csymbol_id id)
{
    ktap_cdata_t *cd;

    cd = kp_cdata_new(ks, id);
    cd_int(cd) = (cdata_number)val;

    return cd;
}

/* argument nmemb here indicates the length of array that is pointed to,
* -1 for unknown */
ktap_cdata_t *kp_cdata_new_ptr(ktap_state_t *ks, void *addr,
                int nmemb, csymbol_id id, int to_allocate)
{
    ktap_cdata_t *cd;
    size_t memb_size;
    csymbol_id deref_id;

    cd = kp_cdata_new(ks, id);

    if (to_allocate) {
        /* allocate new empty space */
        deref_id = csym_ptr_deref_id(id_to_csym(ks, id));
        memb_size = csym_size(ks, id_to_csym(ks, deref_id));
        cd_ptr(cd) = kp_rawobj_alloc(ks, memb_size * nmemb);
    } else {
        cd_ptr(cd) = addr;
    }

    cd_ptr_nmemb(cd) = nmemb;

    return cd;
}

ktap_cdata_t *kp_cdata_new_record(ktap_state_t *ks, void *val, csymbol_id id)
{
    ktap_cdata_t *cd;
    size_t size;

    cd = kp_cdata_new(ks, id);

    /* if val == NULL, allocate new empty space */
    if (val == NULL) {
        size = csym_size(ks, id_to_csym(ks, id));
        cd_record(cd) = kp_rawobj_alloc(ks, size);
    } else
        cd_record(cd) = val;

    return cd;
}

ktap_cdata_t *kp_cdata_new_by_id(ktap_state_t *ks, void *val, csymbol_id id)
{
    csymbol *cs = id_to_csym(ks, id);

    switch (csym_type(cs)) {
    case FFI_VOID:
        kp_error(ks, "Error: Cannot new a void type\n");
        return NULL;
    case FFI_UINT8:
    case FFI_INT8:
    case FFI_UINT16:
     case FFI_INT16:
     case FFI_UINT32:
     case FFI_INT32:
     case FFI_UINT64:
     case FFI_INT64:
         return kp_cdata_new_number(ks, val, id);
     case FFI_PTR:
         return kp_cdata_new_ptr(ks, val, 0, id, 0);
```

```
108     case FFI_STRUCT:
109     case FFI_UNION:
110         return kp_cdata_new_record(ks, val, id);
111     case FFI_FUNC:
112         kp_error(ks, "Error: Cannot new a function type\n");
113         return NULL;
114     case FFI_UNKNOWN:
115     default:
116         kp_error(ks, "Error: unknown csymbol type %s\n", csym_name(cs));
117         return NULL;
118     }
119 }
120
121 void kp_cdata_dump(ktap_state_t *ks, ktap_cdata_t *cd)
122 {
123     switch (cd_type(ks, cd)) {
124     case FFI_UINT8:    case FFI_INT8:
125         kp_printf(ks, "c int(0x%01x)", cd_int(cd));
126         break;
127     case FFI_UINT16:    case FFI_INT16:
128         kp_printf(ks, "c int(0x%02x)", cd_int(cd));
129         break;
130     case FFI_UINT32:    case FFI_INT32:
131         kp_printf(ks, "c int(0x%04x)", cd_int(cd));
132         break;
133     case FFI_UINT64:    case FFI_INT64:
134         kp_printf(ks, "c int(0x%08x)", cd_int(cd));
135         break;
136     case FFI_PTR:
137         kp_printf(ks, "c pointer(0x%p)", cd_ptr(cd));
138         break;
139     case FFI_STRUCT:
140         kp_printf(ks, "c struct(0x%p)", cd_struct(cd));
141         break;
142     case FFI_UNION:
143         kp_printf(ks, "c union(0x%p)", cd_union(cd));
144         break;
145     default:
146         kp_printf(ks, "unsupported cdata type %d!\n", cd_type(ks, cd));
147     }
148 }
149
150 /* Notice: Even if the types are matched, there may exist the lost of
151  * data in the unpack process due to precision */
152 int kp_cdata_type_match(ktap_state_t *ks, csymbol *cs, ktap_val_t *val)
153 {
154     ffi_type type;
155
156     type = csym_type(cs);
157     if (type == FFI_FUNC)
158         goto error;
159
160     switch (ttypenv(val)) {
161     case KTAP_TYPE_LIGHTUSERDATA:
162         if (type != FFI_PTR) goto error;
163         break;
164     case KTAP_TYPE_BOOLEAN:
165     case KTAP_TYPE_NUMBER:
166         if (type != FFI_UINT8 && type != FFI_INT8
167         && type != FFI_UINT16 && type != FFI_INT16
168         && type != FFI_UINT32 && type != FFI_INT32
169         && type != FFI_UINT64 && type != FFI_INT64)
170             goto error;
171         break;
172     case KTAP_TYPE_STRING:
173         if (type != FFI_PTR && type != FFI_UINT8 && type != FFI_INT8)
174             goto error;
175         break;
176     case KTAP_TYPE_CDATA:
177         if (cs != cd_csym(ks, cdvalue(val)))
178             goto error;
179         break;
180     default:
181         goto error;
182     }
183     return 0;
```

```
184
185 error:
186     return -1;
187 }
188
189 static void kp_cdata_value(ktap_state_t *ks, ktap_val_t *val, void **out_addr,
190                size_t *out_size, void **temp)
191 {
192     ktap_cdata_t *cd;
193     csymbol *cs;
194     ffi_type type;
195
196     switch (ttypenv(val)) {
197     case KTAP_TYPE_BOOLEAN:
198         *out_addr = &bvalue(val);
199         *out_size = sizeof(int);
200         return;
201     case KTAP_TYPE_LIGHTUSERDATA:
202         *out_addr = pvalue(val);
203         *out_size = sizeof(void *);
204         return;
205     case KTAP_TYPE_NUMBER:
206         *out_addr = &nvalue(val);
207         *out_size = sizeof(ktap_number);
208         return;
209     case KTAP_TYPE_STRING:
210         *temp = (void *)svalue(val);
211         *out_addr = temp;
212         *out_size = sizeof(void *);
213         return;
214     }
215
216     cd = cdvalue(val);
217     cs = cd_csym(ks, cd);
218     type = csym_type(cs);
219     *out_size = csym_size(ks, cs);
220     switch (type) {
221     case FFI_VOID:
222         kp_error(ks, "Error: Cannot copy data from void type\n");
223         return;
224     case FFI_UINT8:
225     case FFI_INT8:
226     case FFI_UINT16:
227     case FFI_INT16:
228     case FFI_UINT32:
229     case FFI_INT32:
230     case FFI_UINT64:
231     case FFI_INT64:
232         *out_addr = &cd_int(cd);
233         return;
234     case FFI_PTR:
235         *out_addr = &cd_ptr(cd);
236         return;
237     case FFI_STRUCT:
238     case FFI_UNION:
239         *out_addr = cd_record(cd);
240         return;
241     case FFI_FUNC:
242     case FFI_UNKNOWN:
243         kp_error(ks, "Error: internal error for csymbol %s\n",
244                 csym_name(cs));
245         return;
246     }
247 }
248
249 /* Check whether or not type is matched before unpacking */
250 void kp_cdata_unpack(ktap_state_t *ks, char *dst, csymbol *cs, ktap_val_t *val)
251 {
252     size_t size = csym_size(ks, cs), val_size;
253     void *val_addr, *temp;
254
255     kp_cdata_value(ks, val, &val_addr, &val_size, &temp);
256     if (val_size > size)
257         val_size = size;
258     memmove(dst, val_addr, val_size);
259     memset(dst + val_size, 0, size - val_size);
```

```
}

/* Check whether or not type is matched before packing */
void kp_cdata_pack(ktap_state_t *ks, ktap_val_t *val, char *src, csymbol *cs)
{
    size_t size = csym_size(ks, cs), val_size;
    void *val_addr, *temp;

    kp_cdata_value(ks, val, &val_addr, &val_size, &temp);
    if (size > val_size)
        size = val_size;
    memmove(val_addr, src, size);
    memset(val_addr + size, 0, val_size - size);
}

/* Init its cdata type, but not its actual value */
static void kp_cdata_init(ktap_state_t *ks, ktap_val_t *val, void *addr, int len,
              csymbol_id id)
{
    ffi_type type = csym_type(id_to_csym(ks, id));

    switch (type) {
    case FFI_PTR:
        set_cdata(val, kp_cdata_new_ptr(ks, addr, len, id, 0));
        break;
    case FFI_STRUCT:
    case FFI_UNION:
        set_cdata(val, kp_cdata_new_record(ks, addr, id));
        break;
    case FFI_UINT8:
    case FFI_INT8:
    case FFI_UINT16:
    case FFI_INT16:
    case FFI_UINT32:
    case FFI_INT32:
    case FFI_UINT64:
    case FFI_INT64:
        /* set all these value into ktap_number(long) */
        set_number(val, 0);
        break;
    default:
        set_cdata(val, kp_cdata_new(ks, id));
        break;
    }
}

void kp_cdata_ptr_set(ktap_state_t *ks, ktap_cdata_t *cd,
              ktap_val_t *key, ktap_val_t *val)
{
    ktap_number idx;
    csymbol *cs;
    size_t size;
    char *addr;

    if (!is_number(key)) {
        kp_error(ks, "array index should be number\n");
        return;
    }
    idx = nvalue(key);
    if (unlikely(idx < 0 || (cd_ptr_nmemb(cd) >= 0
                    && idx >= cd_ptr_nmemb(cd)))) {
        kp_error(ks, "array index out of bound\n");
        return;
    }

    cs = csym_ptr_deref(ks, cd_csym(ks, cd));
    if (kp_cdata_type_match(ks, cs, val)) {
        kp_error(ks, "array member should be %s type\n", csym_name(cs));
        return;
    }
    size = csym_size(ks, cs);
    addr = cd_ptr(cd);
    addr += size * idx;
    kp_cdata_unpack(ks, addr, cs, val);
}

```

```
void kp_cdata_ptr_get(ktap_state_t *ks, ktap_cdata_t *cd,
              ktap_val_t *key, ktap_val_t *val)
{
    ktap_number idx;
    csymbol *cs;
    size_t size;
    char *addr;
    csymbol_id cs_id;

    if (!is_number(key)) {
        kp_error(ks, "array index should be number\n");
        return;
    }
    idx = nvalue(key);
    if (unlikely(idx < 0 || (cd_ptr_nmemb(cd) >= 0
                    && idx >= cd_ptr_nmemb(cd)))) {
        kp_error(ks, "array index out of bound\n");
        return;
    }

    cs_id = csym_ptr_deref_id(cd_csym(ks, cd));
    cs = id_to_csym(ks, cs_id);
    size = csym_size(ks, cs);
    addr = cd_ptr(cd);
    addr += size * idx;

    kp_cdata_init(ks, val, addr, -1, cs_id);
    kp_cdata_pack(ks, val, addr, cs);
}

void kp_cdata_record_set(ktap_state_t *ks, ktap_cdata_t *cd,
             ktap_val_t *key, ktap_val_t *val)
{
    const char *mb_name;
    csymbol *cs, *mb_cs;
    csymbol_struct *csst;
    struct_member *mb;
    char *addr;

    if (!is_shrstring(key)) {
        kp_error(ks, "struct member name should be string\n");
        return;
    }
    mb_name = svalue(key);
    cs = cd_csym(ks, cd);
    csst = csym_struct(cs);
    mb = csymst_mb_by_name(ks, csst, mb_name);
    if (mb == NULL) {
        kp_error(ks, "struct member %s doesn't exist\n", mb_name);
        return;
    }

    mb_cs = id_to_csym(ks, mb->id);
    if (kp_cdata_type_match(ks, mb_cs, val)) {
        kp_error(ks, "struct member should be %s type\n",
                 csym_name(mb_cs));
        return;
    }

    addr = cd_record(cd);
    addr += csym_record_mb_offset_by_name(ks, cs, mb_name);
    kp_cdata_unpack(ks, addr, mb_cs, val);
}

void kp_cdata_record_get(ktap_state_t *ks, ktap_cdata_t *cd,
             ktap_val_t *key, ktap_val_t *val)
{
    const char *mb_name;
    csymbol *cs, *mb_cs;
    csymbol_struct *csst;
    struct_member *mb;
    char *addr;
    csymbol_id mb_cs_id;

    if (!is_shrstring(key)) {
        kp_error(ks, "struct member name should be string\n");
```

```
412         return;
413     }
414 
415     mb_name = svalue(key);
416     cs = cd_csym(ks, cd);
417     csst = csym_struct(cs);
418     mb = csymst_mb_by_name(ks, csst, mb_name);
419     if (mb == NULL) {
420         kp_error(ks, "struct member %s doesn't exist\n", mb_name);
421         return;
422     }
423 
424     mb_cs_id = mb->id;
425     mb_cs = id_to_csym(ks, mb_cs_id);
426     addr = cd_record(cd);
427     addr += csym_record_mb_offset_by_name(ks, cs, mb_name);
428 
429     kp_cdata_init(ks, val, addr, mb->len, mb_cs_id);
430     if (mb->len < 0)
431         kp_cdata_pack(ks, val, addr, mb_cs);
432 }
433 
```

One Level Up Top Level

runtime/ffi/ffi_util.c - ktap

Functions defined

- csym_align
- csym_record_mb_offset_by_name
- csym_size
- csymst_mb_by_name
- init_csym_struct
- init_csym_union

Source code

```
1  /*
2  * ffi_util.c - utility function for ffi module
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22
23 #include "../../include/ktap_types.h"
24 #include "../../include/ktap_ffi.h"
25 #include "../ktap.h"
26
27 static void init_csym_struct(ktap_state_t *ks, csymbol_struct *csst)
28 {
29     int nr = csymst_mb_nr(csst);
30     size_t size = 0;
31     size_t align = 1;
32     int i, var_len;
33     size_t var_size, var_align;
34
35     if (csymst_mb_nr(csst) < 0) {
36         kp_error(ks, "Find size of undefined struct");
37         return;
38     }
39
40     for (i = 0; i < nr; i++) {
41         csymbol *var_cs = csymst_mb_csym(ks, csst, i);
42         var_len = csymst_mb_len(csst, i);
43         if (var_len < 0) {
44             var_len = 1;
45         } else {
46             var_cs = csym_ptr_deref(ks, var_cs);
47         }
48         var_size = csym_size(ks, var_cs) * var_len;
49         var_align = csym_align(ks, var_cs);
50         size = ALIGN(size, var_align);
51         size += var_size;
```

```
52          align = align > var_align ? align : var_align;
53      }
54      size = ALIGN(size, align);
55      csst->size = size;
56      csst->align = align;
57  }
58
59  static void init_csym_union(ktap_state_t *ks, csymbol_struct *csst)
60  {
61      int nr = csymst_mb_nr(csst);
62      size_t size = 0;
63      size_t align = 1;
64      int i, var_len;
65      size_t var_size, var_align;
66
67      if (csymst_mb_nr(csst) < 0) {
68          kp_error(ks, "Find size of undefined struct");
69          return;
70      }
71
72      for (i = 0; i < nr; i++) {
73          csymbol *var_cs = csymst_mb_csym(ks, csst, i);
74          var_len = csymst_mb_len(csst, i);
75          if (var_len < 0) {
76              var_len = 1;
77          } else {
78              var_cs = csym_ptr_deref(ks, var_cs);
79          }
80          var_size = csym_size(ks, var_cs) * var_len;
81          var_align = csym_align(ks, var_cs);
82          size = size > var_size ? size : var_size;
83          align = align > var_align ? align : var_align;
84      }
85      csst->size = size;
86      csst->align = align;
87  }
88
89
90  size_t csym_size(ktap_state_t *ks, csymbol *cs)
91  {
92      ffi_type type = csym_type(cs);
93      switch(type) {
94      case FFI_STRUCT:
95          if (csym_struct(cs)->align == 0)
96              init_csym_struct(ks, csym_struct(cs));
97          return csym_struct(cs)->size;
98      case FFI_UNION:
99          if (csym_struct(cs)->align == 0)
100             init_csym_union(ks, csym_struct(cs));
101         return csym_struct(cs)->size;
102     default:
103         return ffi_type_size(type);
104     }
105 }
106
107 size_t csym_align(ktap_state_t *ks, csymbol *cs)
108 {
109     ffi_type type = csym_type(cs);
110     switch(type) {
111     case FFI_STRUCT:
112         if (csym_struct(cs)->align == 0)
113             init_csym_struct(ks, csym_struct(cs));
114         return csym_struct(cs)->align;
115     case FFI_UNION:
116         if (csym_struct(cs)->align == 0)
117             init_csym_union(ks, csym_struct(cs));
118         return csym_struct(cs)->align;
119     default:
120         return ffi_type_align(type);
121     }
122 }
123
124 size_t csym_record_mb_offset_by_name(ktap_state_t *ks,
125         csymbol *cs, const char *name)
126 {
127     csymbol_struct *csst = csym_struct(cs);
```

```
128     int nr = csymst_mb_nr(csst);
129     size_t off = 0, sub_off;
130     size_t align = 1;
131     int i, var_len;
132     size_t var_size, var_align;
133 
134     if (nr < 0) {
135         kp_error(ks, "Find size of undefined struct");
136         return 0;
137     }
138 
139     for (i = 0; i < nr; i++) {
140         csymbol *var_cs = csymst_mb_csym(ks, csst, i);
141         var_len = csymst_mb_len(csst, i);
142         if (var_len < 0) {
143             var_len = 1;
144         } else {
145             var_cs = csym_ptr_deref(ks, var_cs);
146         }
147         var_size = csym_size(ks, var_cs) * var_len;
148         var_align = csym_align(ks, var_cs);
149         off = ALIGN(off, var_align);
150         if (!strcmp(name, csymst_mb_name(csst, i)))
151             return off;
152         if (!strcmp("", csymst_mb_name(csst, i))) {
153             if (csym_type(var_cs) != FFI_STRUCT &&
154                     csym_type(var_cs) != FFI_UNION) {
155                 kp_error(ks, "Parse error: non-record type without name");
156                 return -1;
157             }
158             sub_off = csym_record_mb_offset_by_name(ks,
159                     var_cs, name);
160             if (sub_off >= 0)
161                 return off + sub_off;
162         }
163         if (csym_type(cs) == FFI_STRUCT)
164             off += var_size;
165         else
166             off = 0;
167         align = align > var_align ? align : var_align;
168     }
169     return -1;
170 }
171 
172 struct_member *csymst_mb_by_name(ktap_state_t *ks,
173         csymbol_struct *csst, const char *name)
174 {
175     int nr = csymst_mb_nr(csst);
176     int i;
177     struct_member *memb;
178     csymbol *cs = NULL;
179 
180     if (nr < 0) {
181         kp_error(ks, "Find size of undefined struct");
182         return NULL;
183     }
184 
185     for (i = 0; i < nr; i++) {
186         if (!strcmp(name, csymst_mb_name(csst, i)))
187             return csymst_mb(csst, i);
188         if (!strcmp("", csymst_mb_name(csst, i))) {
189             cs = csymst_mb_csym(ks, csst, i);
190             if (csym_type(cs) != FFI_STRUCT && csym_type(cs) != FFI_UNION) {
191                 kp_error(ks, "Parse error: non-record type without name");
192                 return NULL;
193             }
194             memb = csymst_mb_by_name(ks, csym_struct(cs), name);
195             if (memb != NULL)
196                 return memb;
197         }
198     }
199     return NULL;
200 }
```

One Level Up Top Level

runtime/lib_table.c - ktap

Global variables defined

- table_lib_funcs

Functions defined

- kp_lib_init_table
- kplib_table_new

Source code

```
1  /*
2  * lib_table.c - Table library
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22 #include <linux/ctype.h>
23 #include <linux/slab.h>
24 #include <linux/delay.h>
25 #include <linux/sched.h>
26 #include "../include/ktap_types.h"
27 #include "ktap.h"
28 #include "kp_obj.h"
29 #include "kp_vm.h"
30 #include "kp_tab.h"
31
32 static int kplib_table_new(ktap_state_t *ks)
33 {
34     int narr = kp_arg_checkoptnumber(ks, 1, 0);
35     int nrec = kp_arg_checkoptnumber(ks, 2, 0);
36     ktap_tab_t *h;
37
38     h = kp_tab_new_ah(ks, narr, nrec);
39     if (!h) {
40         set_nil(ks->top);
41     } else {
42         set_table(ks->top, h);
43     }
44
45     incr_top(ks);
46     return 1;
47 }
48
49 static const ktap_libfunc_t table_lib_funcs[] = {
50     {"new",    kplib_table_new},
51     {NULL}
52 };
53
54 int kp_lib_init_table(ktap_state_t *ks)
```

```
55 {
56     return kp_vm_register_lib(ks, "table", table_lib_funcs);
57 }
58
```

One Level Up Top Level

runtime/lib_net.c - ktap

Global variables defined

- net_lib_funcs

Functions defined

- kp_lib_init_net
- kplib_net_format_ip_addr
- kplib_net_ip_sock_daddr
- kplib_net_ip_sock_saddr

Source code

```
1  /*
2  * lib_base.c - base library
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10 * version 2, as published by the Free Software Foundation.
11 *
12 * ktap is distributed in the hope it will be useful, but WITHOUT
13 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15 * more details.
16 *
17 * You should have received a copy of the GNU General Public License along with
18 * this program; if not, write to the Free Software Foundation, Inc.,
19 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20 */
21
22 #include <net/inet_sock.h>
23 #include "../include/ktap_types.h"
24 #include "ktap.h"
25 #include "kp_obj.h"
26 #include "kp_str.h"
27 #include "kp_vm.h"
28
29 /**
30 * Return the source IP address for a given sock
31 */
32 static int kplib_net_ip_sock_saddr(ktap_state_t *ks)
33 {
34     struct inet_sock *isk;
35     int family;
36
37     /* TODO: need to validate the address firstly */
38
39     isk = (struct inet_sock *)kp_arg_checknumber(ks, 1);
40     family = isk->sk.__sk_common.skc_family;
41
42     if (family == AF_INET) {
43         set_number(ks->top, isk->inet_rcv_saddr);
44     } else {
45         kp_error(ks, "ip_sock_saddr only support ipv4 now\n");
46         set_nil(ks->top);
47     }
48
49     incr_top(ks);
50     return 1;
```

```
51  }
52
53  /**
54  * Return the destination IP address for a given sock
55  */
56  static int kplib_net_ip_sock_daddr(ktap_state_t *ks)
57  {
58      struct inet_sock *isk;
59      int family;
60
61      /* TODO: need to validate the address firstly */
62
63      isk = (struct inet_sock *)kp_arg_checknumber(ks, 1);
64      family = isk->sk.__sk_common.skc_family;
65
66      if (family == AF_INET) {
67          set_number(ks->top, isk->inet_daddr);
68      } else {
69          kp_error(ks, "ip_sock_daddr only support ipv4 now\n");
70          set_nil(ks->top);
71      }
72
73      incr_top(ks);
74      return 1;
75
76  }
77
78  /**
79  * Returns a string representation for an IP address
80  */
81  static int kplib_net_format_ip_addr(ktap_state_t *ks)
82  {
83      __be32 ip = (__be32)kp_arg_checknumber(ks, 1);
84      ktap_str_t *ts;
85      char ipstr[32];
86
87      snprintf(ipstr, 32, "%pI4", &ip);
88      ts = kp_str_newz(ks, ipstr);
89      if (ts) {
90          set_string(ks->top, kp_str_newz(ks, ipstr));
91          incr_top(ks);
92          return 1;
93      } else
94          return -1;
95  }
96
97  static const ktap_libfunc_t net_lib_funcs[] = {
98      {"ip_sock_saddr", kplib_net_ip_sock_saddr},
99      {"ip_sock_daddr", kplib_net_ip_sock_daddr},
100      {"format_ip_addr", kplib_net_format_ip_addr},
101      {NULL}
102  };
103
104  int kp_lib_init_net(ktap_state_t *ks)
105  {
106      return kp_vm_register_lib(ks, "net", net_lib_funcs);
107  }
```

One Level Up Top Level

userspace/kp_bcwrite.c - ktap

Global variables defined

- bc_mode
- bc_names
- function_nr

Data types defined

- BCWriteCtx
- BCWriteCtx

Functions defined

- bcwrite_bytecode
- bcwrite_footer
- bcwrite_header
- bcwrite_kgc
- bcwrite_knum
- bcwrite_ktab
- bcwrite_ktabk
- bcwrite_proto
- bcwrite_uint32
- dump_bytecode
- kp_bcwrite
- kp_dump_proto

Macros defined

- BCNAME
- BCNAME

Source code

```
1  /*
2   * Bytecode writer
3   *
4   * This file is part of ktap by Jovi Zhangwei.
5   *
6   * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7   *
8   * Copyright (C) 1994-2013 Lua.org, PUC-Rio.
9   *  - The part of code in this file is copied from lua initially.
10  *  - lua's MIT license is compatible with GPL.
11  *
```

```
 * ktap is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * ktap is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/ktap_types.h"
#include "cparser.h"
#include "kp_util.h"

/* Context for bytecode writer. */
typedef struct BCWriteCtx {
    SBuf sb;        /* Output buffer. */
    ktap_proto_t *pt;    /* Root prototype. */
    ktap_writer wfunc;    /* Writer callback. */
    void *wdata;        /* Writer callback data. */
    int strip;        /* Strip debug info. */
    int status;        /* Status from writer callback. */
} BCWriteCtx;


static char *bcwrite_uint32(char *p, uint32_t v)
{
    memcpy(p, &v, sizeof(uint32_t));
    p += sizeof(uint32_t);
    return p;
}

/* -- Bytecode writer ----------------------------------------------------- */

/* Write a single constant key/value of a template table. */
static void bcwrite_ktabk(BCWriteCtx *ctx, const ktap_val_t *o, int narrow)
{
    char *p = kp_buf_more(&ctx->sb, 1+10);
    if (is_string(o)) {
        const ktap_str_t *str = rawtsvalue(o);
        int len = str->len;
        p = kp_buf_more(&ctx->sb, 5+len);
        p = bcwrite_uint32(p, BCDUMP_KTAB_STR+len);
        p = kp_buf_wmem(p, getstr(str), len);
    } else if (is_number(o)) {
        p = bcwrite_uint32(p, BCDUMP_KTAB_NUM);
        p = kp_buf_wmem(p, &nvalue(o), sizeof(ktap_number));
    } else {
        kp_assert(tvispri(o));
        p = bcwrite_uint32(p, BCDUMP_KTAB_NIL+~itype(o));
    }
    setsbufP(&ctx->sb, p);
}

/* Write a template table. */
static void bcwrite_ktab(BCWriteCtx *ctx, char *p, const ktap_tab_t *t)
{
    int narray = 0, nhash = 0;
    if (t->asize > 0) {  /* Determine max. length of array part. */
        ptrdiff_t i;
        ktap_val_t *array = t->array;
        for (i = (ptrdiff_t)t->asize-1; i >= 0; i--)
            if (!is_nil(&array[i]))
                break;
            narray = (int)(i+1);
    }
    if (t->hmask > 0) {  /* Count number of used hash slots. */
        int i, hmask = t->hmask;
```

```
        ktap_node_t *node = t->node;
        for (i = 0; i <= hmask; i++)
            nhash += !is_nil(&node[i].val);
    }
    /* Write number of array slots and hash slots. */
    p = bcwrite_uint32(p, narray);
    p = bcwrite_uint32(p, nhash);
    setsbufP(&ctx->sb, p);
    if (narray) {  /* Write array entries (may contain nil). */
        int i;
        ktap_val_t *o = t->array;
        for (i = 0; i < narray; i++, o++)
            bcwrite_ktabk(ctx, o, 1);
    }
    if (nhash) {  /* Write hash entries. */
        int i = nhash;
        ktap_node_t *node = t->node + t->hmask;
        for (;; node--)
            if (!is_nil(&node->val)) {
                bcwrite_ktabk(ctx, &node->key, 0);
                bcwrite_ktabk(ctx, &node->val, 1);
                if (--i == 0)
                    break;
            }
    }
}

/* Write GC constants of a prototype. */
static void bcwrite_kgc(BCWriteCtx *ctx, ktap_proto_t *pt)
{
    int i, sizekgc = pt->sizekgc;
    ktap_obj_t **kr = (ktap_obj_t **)pt->k - (ptrdiff_t)sizekgc;

    for (i = 0; i < sizekgc; i++, kr++) {
        ktap_obj_t *o = *kr;
        int tp, need = 1;
        char *p;

        /* Determine constant type and needed size. */
        if (o->gch.gct == ~KTAP_TSTR) {
            tp = BCDUMP_KGC_STR + ((ktap_str_t *)o)->len;
            need = 5 + ((ktap_str_t *)o)->len;
        } else if (o->gch.gct == ~KTAP_TPROTO) {
            kp_assert((pt->flags & PROTO_CHILD));
            tp = BCDUMP_KGC_CHILD;
        } else {
            kp_assert(o->gch.gct == ~KTAP_TTAB);
            tp = BCDUMP_KGC_TAB;
            need = 1+2*5;
        }

        /* Write constant type. */
        p = kp_buf_more(&ctx->sb, need);
        p = bcwrite_uint32(p, tp);
        /* Write constant data (if any). */
        if (tp >= BCDUMP_KGC_STR) {
            p = kp_buf_wmem(p, getstr((ktap_str_t *)o),
                    ((ktap_str_t *)o)->len);
        } else if (tp == BCDUMP_KGC_TAB) {
            bcwrite_ktab(ctx, p, (ktap_tab_t *)o);
            continue;
        }
        setsbufP(&ctx->sb, p);
    }
}

/* Write number constants of a prototype. */
static void bcwrite_knum(BCWriteCtx *ctx, ktap_proto_t *pt)
{
    int i, sizekn = pt->sizekn;
    const ktap_val_t *o = (ktap_val_t *)pt->k;
    char *p = kp_buf_more(&ctx->sb, 10*sizekn);

    for (i = 0; i < sizekn; i++, o++) {
        if (is_number(o))
            p = kp_buf_wmem(p, &nvalue(o), sizeof(ktap_number));
```

```
    }
    setsbufP(&ctx->sb, p);
}

/* Write bytecode instructions. */
static char *bcwrite_bytecode(BCWriteCtx *ctx, char *p, ktap_proto_t *pt)
{
    int nbc = pt->sizebc-1;  /* Omit the [JI]FUNC* header. */

    p = kp_buf_wmem(p, proto_bc(pt)+1, nbc*(int)sizeof(BCIns));
    return p;
}

/* Write prototype. */
static void bcwrite_proto(BCWriteCtx *ctx, ktap_proto_t *pt)
{
    int sizedbg = 0;
    char *p;

    /* Recursively write children of prototype. */
    if (pt->flags & PROTO_CHILD) {
        ptrdiff_t i, n = pt->sizekgc;
        ktap_obj_t **kr = (ktap_obj_t **)pt->k - 1;
        for (i = 0; i < n; i++, kr--) {
            ktap_obj_t *o = *kr;
            if (o->gch.gct == ~KTAP_TPROTO)
                bcwrite_proto(ctx, (ktap_proto_t *)o);
        }
    }

    /* Start writing the prototype info to a buffer. */
    p = kp_buf_need(&ctx->sb,
        5+4+6*5+(pt->sizebc-1)*(int)sizeof(BCIns)+pt->sizeuv*2);
    p += 4;  /* Leave room for final size. */

    /* Write prototype header. */
    *p++ = (pt->flags & (PROTO_CHILD|PROTO_VARARG|PROTO_FFI));
    *p++ = pt->numparams;
    *p++ = pt->framesize;
    *p++ = pt->sizeuv;
    p = bcwrite_uint32(p, pt->sizekgc);
    p = bcwrite_uint32(p, pt->sizekn);
    p = bcwrite_uint32(p, pt->sizebc-1);
    if (!ctx->strip) {
        if (proto_lineinfo(pt))
            sizedbg = pt->sizept -
                (int)((char *)proto_lineinfo(pt) - (char *)pt);
        p = bcwrite_uint32(p, sizedbg);
        if (sizedbg) {
            p = bcwrite_uint32(p, pt->firstline);
            p = bcwrite_uint32(p, pt->numline);
        }
    }

    /* Write bytecode instructions and upvalue refs. */
    p = bcwrite_bytecode(ctx, p, pt);
    p = kp_buf_wmem(p, proto_uv(pt), pt->sizeuv*2);
    setsbufP(&ctx->sb, p);

    /* Write constants. */
    bcwrite_kgc(ctx, pt);
    bcwrite_knum(ctx, pt);

    /* Write debug info, if not stripped. */
    if (sizedbg) {
        p = kp_buf_more(&ctx->sb, sizedbg);
        p = kp_buf_wmem(p, proto_lineinfo(pt), sizedbg);
        setsbufP(&ctx->sb, p);
    }

    /* Pass buffer to writer function. */
    if (ctx->status == 0) {
        int n = sbuflen(&ctx->sb) - 4;
        char *q = sbufB(&ctx->sb);
        p = bcwrite_uint32(q, n);  /* Fill in final size. */
        kp_assert(p == sbufB(&ctx->sb) + 4);
```

```
240         ctx->status = ctx->wfunc(q, n + 4, ctx->wdata);
241     }
242 }
243 
244 /* Write header of bytecode dump. */
245 static void bcwrite_header(BCWriteCtx *ctx)
246 {
247     ktap_str_t *chunkname = proto_chunkname(ctx->pt);
248     const char *name = getstr(chunkname);
249     int len = chunkname->len;
250     char *p = kp_buf_need(&ctx->sb, 5+5+len);
251     *p++ = BCDUMP_HEAD1;
252     *p++ = BCDUMP_HEAD2;
253     *p++ = BCDUMP_HEAD3;
254     *p++ = BCDUMP_VERSION;
255     *p++ = (ctx->strip ? BCDUMP_F_STRIP : 0) + (KP_BE ? BCDUMP_F_BE : 0);
256 
257     if (!ctx->strip) {
258         p = bcwrite_uint32(p, len);
259         p = kp_buf_wmem(p, name, len);
260     }
261     ctx->status = ctx->wfunc(sbufB(&ctx->sb),
262         (int)(p - sbufB(&ctx->sb)), ctx->wdata);
263 }
264 
265 /* Write footer of bytecode dump. */
266 static void bcwrite_footer(BCWriteCtx *ctx)
267 {
268     if (ctx->status == 0) {
269         uint8_t zero = 0;
270         ctx->status = ctx->wfunc(&zero, 1, ctx->wdata);
271     }
272 }
273 
274 /* Write bytecode for a prototype. */
275 int kp_bcwrite(ktap_proto_t *pt, ktap_writer writer, void *data, int strip)
276 {
277     BCWriteCtx ctx;
278 
279     ctx.pt = pt;
280     ctx.wfunc = writer;
281     ctx.wdata = data;
282     ctx.strip = strip;
283     ctx.status = 0;
284 
285     kp_buf_init(&ctx.sb);
286     kp_buf_need(&ctx.sb, 1024);  /* Avoids resize for most prototypes. */
287     bcwrite_header(&ctx);
288     bcwrite_proto(&ctx, ctx.pt);
289     bcwrite_footer(&ctx);
290 
291     kp_buf_free(&ctx.sb);
292     return ctx.status;
293 }
294 
295 /* -- Bytecode dump ------------------------------------------------------ */
296 
297 static const char * const bc_names[] = {
298 #define BCNAME(name, ma, mb, mc, mt)       #name,
299     BCDEF(BCNAME)
300 #undef BCNAME
301   NULL
302 };
303 
304 static const uint16_t bc_mode[] = {
305     BCDEF(BCMODE)
306 };
307 
308 static void dump_bytecode(ktap_proto_t *pt)
309 {
310     int nbc = pt->sizebc - 1; /* Omit the FUNC* header. */
311     BCIns *ins = proto_bc(pt) + 1;
312     ktap_obj_t **kbase = pt->k;
313     int i;
314 
315     printf("-- BYTECODE -- %s:%d-%d\n", getstr(pt->chunkname),
```

```
316         pt->firstline, pt->firstline + pt->numline);
317
318     for (i = 0; i < nbc; i++, ins++) {
319         int op = bc_op(*ins);
320
321         printf("%04d\t%s", i + 1, bc_names[op]);
322
323         printf("\t%d", bc_a(*ins));
324         if (bcmode_b(op) != BCMnone)
325             printf("\t%d", bc_b(*ins));
326
327         if (bcmode_hasd(op))
328             printf("\t%d", bc_d(*ins));
329         else
330             printf("\t%d", bc_c(*ins));
331
332         if (bcmode_b(op) == BCMstr || bcmode_c(op) == BCMstr) {
333             printf("\t  ; ");
334             if (bcmode_d(op) == BCMstr) {
335                 int idx = ~bc_d(*ins);
336                 printf("\"%s\"", getstr((ktap_str_t *)kbase[idx]));
337             }
338         }
339         printf("\n");
340     }
341 }
342
343 static int function_nr = 0;
344
345 void kp_dump_proto(ktap_proto_t *pt)
346 {
347     printf("\n----------------------------------------------------\n");
348     printf("function proto %d:\n", function_nr++);
349     printf("numparams: %d\n", pt->numparams);
350     printf("framesize: %d\n", pt->framesize);
351     printf("sizebc: %d\n", pt->sizebc);
352     printf("sizekgc: %d\n", pt->sizekgc);
353     printf("sizekn: %d\n", pt->sizekn);
354     printf("sizept: %d\n", pt->sizept);
355     printf("sizeuv: %d\n", pt->sizeuv);
356     printf("firstline: %d\n", pt->firstline);
357     printf("numline: %d\n", pt->numline);
358
359     printf("has child proto: %d\n", pt->flags & PROTO_CHILD);
360     printf("has vararg: %d\n", pt->flags & PROTO_VARARG);
361     printf("has ILOOP: %d\n", pt->flags & PROTO_ILOOP);
362
363     dump_bytecode(pt);
364
365     /* Recursively dump children of prototype. */
366     if (pt->flags & PROTO_CHILD) {
367         ptrdiff_t i, n = pt->sizekgc;
368         ktap_obj_t **kr = (ktap_obj_t **)pt->k - 1;
369         for (i = 0; i < n; i++, kr--) {
370             ktap_obj_t *o = *kr;
371             if (o->gch.gct == ~KTAP_TPROTO)
372                 kp_dump_proto((ktap_proto_t *)o);
373         }
374     }
375 }
376
```

One Level Up Top Level

test/ffi_test/ktap_ffi_test.c - ktap

Global variables defined

- ffi_test_array
- ffi_test_exit
- ffi_test_init
- ffi_test_int1
- ffi_test_int2
- ffi_test_pointer1
- ffi_test_sched_clock
- ffi_test_struct
- ffi_test_struct_array
- ffi_test_struct_loop
- ffi_test_struct_noname
- ffi_test_union
- ffi_test_var_arg
- ffi_test_void

Data types defined

- ffi_struct
- ffi_struct_array
- ffi_struct_loop
- ffi_struct_noname
- ffi_union

Functions defined

- ffi_test_array
- ffi_test_exit
- ffi_test_init
- ffi_test_int1
- ffi_test_int2
- ffi_test_pointer1
- ffi_test_sched_clock
- ffi_test_struct

- ffi_test_struct_array
- ffi_test_struct_loop
- ffi_test_struct_noname
- ffi_test_union
- ffi_test_var_arg
- ffi_test_void

Source code

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>

void ffi_test_void(void)
{
}
EXPORT_SYMBOL(ffi_test_void);

int ffi_test_int1(unsigned char a, char b, unsigned short c, short d)
{
    return a + b + c + d;
}
EXPORT_SYMBOL(ffi_test_int1);

long long ffi_test_int2(unsigned int a, int b, unsigned long c, long d,
        unsigned long long e, long long f, long long g)
{
    return a + b + c + d + e + f + g;
}
EXPORT_SYMBOL(ffi_test_int2);

void *ffi_test_pointer1(char *a) {
    return a;
}
EXPORT_SYMBOL(ffi_test_pointer1);

long long ffi_test_var_arg(int n, ...) {
    va_list ap;
    int i;
    long long sum = 0;
    va_start(ap, n);
    for (i = 0; i < n; i++) {
        sum += va_arg(ap, long long);
    }
    va_end(ap);
    return sum;
}
EXPORT_SYMBOL(ffi_test_var_arg);

unsigned long long ffi_test_sched_clock(void)
{
    return sched_clock();
}
EXPORT_SYMBOL(ffi_test_sched_clock);

int ffi_test_array(int *arr, int idx)
{
    return arr[idx];
}
EXPORT_SYMBOL(ffi_test_array);

struct ffi_struct {
    int val;
};

int ffi_test_struct(struct ffi_struct *s)
{
```

```
60      return s->val;
61  }
62  EXPORT_SYMBOL(ffi_test_struct);
63
64  struct ffi_struct_loop {
65      int val;
66      struct ffi_struct_loop *next;
67  };
68
69  int ffi_test_struct_loop(struct ffi_struct_loop *s)
70  {
71      if (s == s->next)
72          return s->val;
73      return 0;
74  }
75  EXPORT_SYMBOL(ffi_test_struct_loop);
76
77  union ffi_union {
78      int val;
79      int val2;
80      int val3;
81  };
82
83  int ffi_test_union(union ffi_union *s)
84  {
85      return s->val;
86  }
87  EXPORT_SYMBOL(ffi_test_union);
88
89  struct ffi_struct_array {
90      int val[20];
91  };
92
93  int ffi_test_struct_array(struct ffi_struct_array *s)
94  {
95      int sum = 0, i;
96      for (i = 0; i < 20; i++)
97          sum += s->val[i];
98      return sum;
99  }
100  EXPORT_SYMBOL(ffi_test_struct_array);
101
102  struct ffi_struct_noname {
103      struct {
104          int pad;
105          int val;
106      };
107  };
108
109  int ffi_test_struct_noname(struct ffi_struct_noname *s)
110  {
111      return s->val;
112  }
113  EXPORT_SYMBOL(ffi_test_struct_noname);
114
115
116  static int __init ffi_test_init(void)
117  {
118      return 0;
119  }
120
121  static void __exit ffi_test_exit(void)
122  {
123  }
124
125
126  MODULE_DESCRIPTION("ktap ffi test module");
127  MODULE_LICENSE("GPL");
128
129  module_init(ffi_test_init);
130  module_exit(ffi_test_exit);
```

One Level Up Top Level

test/lib/ - ktap

- Test/

test/lib/Test/ - ktap

- ktap/
- ktap.pm

test/lib/Test/ktap/ - ktap

- features.pm

test/lib/Test/ktap/features.pm - ktap

Data types defined

- has_ffi

Source code

```
1  package Test::ktap::features;
2
3  use strict;
4  use warnings;
5
6  use Exporter 'import';
7  our @EXPORT_OK = qw( &has_ffi );
8
9  sub has_ffi {
10      return `./ktap -V 2>&1 | grep -c FFI` == 1;
11 }
12
13 1;
```

test/lib/Test/ktap.pm - ktap

Data types defined

Source code

```
# Copyright (C) Yichun Zhang (agentzh)

package Test::ktap;

use Test::Base -Base;
use POSIX ();
use IPC::Run ();

our @EXPORT = qw( run_tests );

 sub run_tests () {
     for my $block (Test::Base::blocks()) {
         run_test($block);
     }
 }

 sub bail_out (@) {
     Test::More::BAIL_OUT(@_);
 }

 sub parse_cmd ($) {
     my $cmd = shift;
     my @cmd;
     while (1) {
         if ($cmd =~ /\G\s*"(.*?)"/gmsc) {
             push @cmd, $1;

         } elsif ($cmd =~ /\G\s*'(.*?)'/gmsc) {
             push @cmd, $1;

         } elsif ($cmd =~ /\G\s*(\S+)/gmsc) {
             push @cmd, $1;

         } else {
             last;
         }
     }
     return @cmd;
 }

 sub run_test ($) {
     my $block = shift;
     my $name = $block->name;

     my $timeout = $block->timeout() || 10;
     my $opts = $block->opts;
     my $args = $block->args;

     my $cmd = "./ktap";

     if (defined $opts) {
         $cmd .= " $opts";
     }

     my $kpfile;
     if (defined $block->src) {
```

```
        $kpfile = POSIX::tmpnam() . ".kp";
        open my $out, ">$kpfile" or
            bail_out("cannot open $kpfile for writing: $!");
        print $out ($block->src);
        close $out;
        $cmd .= " $kpfile"
    }

    if (defined $args) {
        $cmd .= " $args";
    }

    #warn "CMD: $cmd\n";

    my @cmd = parse_cmd($cmd);

    my ($out, $err);

    eval {
        IPC::Run::run(\@cmd, \undef, \$out, \$err,
                      IPC::Run::timeout($timeout));
    };
    if ($@) {
        # timed out
        if ($@ =~ /timeout/) {
            if (!defined $block->expect_timeout) {
                fail("$name: ktap process timed out");
            }
    } else {
            fail("$name: failed to run command [$cmd]: $@");
        }
    }

    my $ret = ($? >> 8);

    if (defined $kpfile) {
        unlink $kpfile;
    }

    if (defined $block->out) {
        is $out, $block->out, "$name - stdout eq okay";
    }

     my $regex = $block->out_like;
     if (defined $regex) {
         if (!ref $regex) {
             $regex = qr/$regex/ms;
         }
         like $out, $regex, "$name - stdout like okay";
     }

     if (defined $block->err) {
         is $err, $block->err, "$name - stderr eq okay";
     }

     $regex = $block->err_like;
     if (defined $regex) {
         if (!ref $regex) {
             $regex = qr/$regex/ms;
         }
         like $err, $regex, "$name - stderr like okay";
     }

     my $exp_ret = $block->ret;
     if (!defined $exp_ret) {
         $exp_ret = 0;
     }
     is $ret, $exp_ret, "$name - exit code okay";
}

1;
# vi: et
```

One Level Up Top Level

test/util/ - ktap

- reindex

test/util/reindex - ktap

Data types defined

- reindex

Source code

```
#!/usr/bin/env perl

# reindex
# reindex .t files for Test::Base based test files
# Copyright (C) Yichun Zhang (agentzh)

use strict;
use warnings;

 use Getopt::Std;

 my %opts;
 getopts('hb:', \%opts);
 if ($opts{h} or ! @ARGV) {
     die "Usage: reindex [-b 0] t/*.t\n";
 }

 my $init = $opts{b};
 $init = 1 if not defined $init;

 my @files = map glob, @ARGV;
 for my $file (@files) {
     next if -d $file or $file !~ /\.t_?$/;
     reindex($file);
 }

 sub reindex {
     my $file = $_[0];
     open my $in, $file or
         die "Can't open $file for reading: $!";
     my @lines;
     my $counter = $init;
     my $changed;
     while (<$in>) {
         s/\r$//;
         my $num;
         s/ ^ === \s+ TEST \s+ (\d+)/$num=$1; "=== TEST " . $counter++/xie;
         next if !defined $num;
         if ($num != $counter-1) {
             $changed++;
         }
     } continue {
         push @lines, $_;
     }
     close $in;
     my $text = join '', @lines;
     $text =~ s/(?x) \n+ === \s+ TEST/\n\n\n\n=== TEST/ixsg;
     $text =~ s/__(DATA|END)__\n+=== TEST/__${1}__\n\n=== TEST/;
     #$text =~ s/\n+$/\n\n/s;
     if (! $changed and $text eq join '', @lines) {
         warn "reindex: $file:\tskipped.\n";
         return;
     }
     open my $out, "> $file" or
         die "Can't open $file for writing: $!";
     binmode $out;
     print $out $text;
     close $out;

     warn "reindex: $file:\tdone.\n";
 }
```

test/benchmark/cmp_neq.sh - ktap

```
#!/bin/sh

# This script compare number equality performance between ktap and stap.
# It also compare different ktap tracing interfaces.
#
# 1. ktap -e 'trace syscalls:sys_enter_futex {}'
# 2. ktap -e 'kdebug.tracepoint("sys_enter_futex", function () {})'
# 3. ktap -e 'trace probe:SyS_futex uaddr=%di {}'
# 4. ktap -e 'kdebug.kprobe("SyS_futex", function () {})'
 # 5. stap -e 'probe syscall.futex {}'
 # 6. ktap -d -e 'trace syscalls:sys_enter_futex {}'
 # 7. ktap -d -e 'kdebug.tracepoint("sys_enter_futex", function () {})'
 # 8. ktap -e 'trace syscalls:sys_enter_futex /kernel_buildin_filter/ {}'

 #Result:
 #ktap number computation and comparsion overhead is bigger than stap,
 #nearly 10+% (4 vs. 5 in above)), ktap is not very slow.
 #
 #Perf backend tracing overhead is big, because it need copy temp buffer, and
 #code path is very long than direct callback(1 vs. 4 in above).

 gcc -o sembench sembench.c -O2 -lpthread

 COMMAND="./sembench -t 200 -w 20 -r 30 -o 2"

 #-------------------------------------------------------------#

 echo -e "without tracing:"
 #$COMMAND; $COMMAND; $COMMAND

 #-------------------------------------------------------------#

 ../../ktap -q -e 'trace syscalls:sys_enter_futex {
     var uaddr = arg2
         if (uaddr == 0x100 || uaddr == 0x200 || uaddr == 0x300 ||
         uaddr == 0x400 || uaddr == 0x500 || uaddr == 0x600 ||
         uaddr == 0x700 || uaddr == 0x800 || uaddr == 0x900 ||
         uaddr == 0x1000) {
                 printf("%x %x\n", arg1, arg2)
         }}' &

 echo -e "\nktap tracing: trace syscalls:sys_enter_futex { if (arg2 == 0x100 || arg2 == 0x200 ... }"
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'kdebug.tracepoint("sys_enter_futex", function () {
     var arg = arg2
         if (arg == 0x100 || arg == 0x200 || arg == 0x300 || arg == 0x400 ||
             arg == 0x500 || arg == 0x600 || arg == 0x700 || arg == 0x800 ||
             arg == 0x900 || arg == 0x1000) {
                 printf("%x %x\n", arg1, arg2)
         }})' &

 echo -e '\nktap tracing: kdebug.tracepoint("sys_enter_futex", function (xxx) {})'
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'trace probe:SyS_futex uaddr=%di {
     var arg = arg1
         if (arg == 0x100 || arg == 0x200 || arg == 0x300 || arg == 0x400 ||
             arg == 0x500 || arg == 0x600 || arg == 0x700 || arg == 0x800 ||
             arg == 0x900 || arg == 0x1000) {
                 printf("%x\n", arg1)
         }}' &
 echo -e '\nktap tracing: trace probe:SyS_futex uaddr=%di {...}'
```

```
$COMMAND; $COMMAND; $COMMAND
pid=`pidof ktap`
disown $pid; kill -9 $pid; sleep 1


#-------------------------------------------------------------#
../../ktap -q -e 'kdebug.kprobe("SyS_futex", function () {
    var uaddr = 1
        if (uaddr == 0x100 || uaddr == 0x200 || uaddr == 0x300 ||
        uaddr == 0x400 || uaddr == 0x500 || uaddr == 0x600 ||
        uaddr == 0x700 || uaddr == 0x800 || uaddr == 0x900 ||
        uaddr == 0x1000) {
                printf("%x\n", uaddr)
    }})' &
echo -e '\nktap tracing: kdebug.kprobe("SyS_futex", function () {})'
$COMMAND; $COMMAND; $COMMAND
pid=`pidof ktap`
disown $pid; kill -9 $pid; sleep 1

#-------------------------------------------------------------#

stap -e 'probe syscall.futex {
    uaddr = $uaddr
        if (uaddr == 0x100 || uaddr == 0x200 || uaddr == 0x300 ||
        uaddr == 0x400 || uaddr == 0x500 || uaddr == 0x600 ||
        uaddr == 0x700 || uaddr == 0x800 || uaddr == 0x900 ||
        uaddr == 0x1000) {
                printf("%x\n", uaddr)
         }}' &

echo -e "\nstap tracing: probe syscall.futex { if (uaddr == 0x100 || addr == 0x200 ... }"
$COMMAND; $COMMAND; $COMMAND
pid=`pidof stap`
disown $pid; kill -9 $pid; sleep 1

#-------------------------------------------------------------#


../../ktap -d -q -e 'trace syscalls:sys_enter_futex {
    var uaddr = arg2
        if (uaddr == 0x100 || uaddr == 0x200 || uaddr == 0x300 ||
        uaddr == 0x400 || uaddr == 0x500 || uaddr == 0x600 ||
        uaddr == 0x700 || uaddr == 0x800 || uaddr == 0x900 ||
        uaddr == 0x1000) {
                printf("%x %x\n", arg1, arg2)
        }}' &

echo -e "\nktap tracing dry-run: trace syscalls:sys_enter_futex { if (arg2 == 0x100 || arg2 == 0x200 ... }"
$COMMAND; $COMMAND; $COMMAND
pid=`pidof ktap`
disown $pid; kill -9 $pid; sleep 1


#-------------------------------------------------------------#

../../ktap -d -q -e 'kdebug.tracepoint("sys_enter_futex", function () {
    var arg = arg2
        if (arg == 0x100 || arg == 0x200 || arg == 0x300 || arg == 0x400 ||
            arg == 0x500 || arg == 0x600 || arg == 0x700 || arg == 0x800 ||
            arg == 0x900 || arg == 0x1000) {
                printf("%x %x\n", arg1, arg2)
        }})' &

echo -e '\nktap tracing dry-run: kdebug.tracepoint("sys_enter_futex", function (xxx) {})'
$COMMAND; $COMMAND; $COMMAND
pid=`pidof ktap`
disown $pid; kill -9 $pid; sleep 1


#-------------------------------------------------------------#

../../ktap -q -e 'trace syscalls:sys_enter_futex /
    uaddr == 0x100 || uaddr == 0x200 || uaddr == 0x300 || uaddr == 0x400 ||
    uaddr == 0x500 || uaddr == 0x600 || uaddr == 0x700 || uaddr == 0x800 ||
    uaddr == 0x900 || uaddr == 0x1000/ {
        printf("%x %x\n", arg1, arg2)
```

```
    }' &

echo -e "\nktap tracing: trace syscalls:sys_enter_futex /uaddr == 0x100 || uaddr == 0x200 .../ {}"
$COMMAND; $COMMAND; $COMMAND
pid=`pidof ktap`
disown $pid; kill -9 $pid; sleep 1

#------------------------------------------------------------#

rm -rf ./sembench

```

One Level Up Top Level

test/benchmark/cmp_profile.sh - ktap

```
#!/bin/sh

# This script compare stack profiling performance between ktap and stap.
#
# 1. ktap -e 'profile-1000us { s[stack(-1, 12)] += 1 }'
# 2. stap -e 'probe timer.profile { s[backtrace()] += 1 }'
# 3. stap -e 'probe timer.profile { s[backtrace()] <<< 1 }'

#Result:
 #Currently the stack profiling overhead is nearly same between ktap and stap.
 #
 #ktap reslove kernel stack to string in runtime, which is very time consuming,
 #optimize it in future.


 gcc -o sembench sembench.c -O2 -lpthread

 COMMAND="./sembench -t 200 -w 20 -r 30 -o 2"

 #------------------------------------------------------------#

 echo -e "without tracing:"
 $COMMAND; $COMMAND; $COMMAND

 #------------------------------------------------------------#

 ../../ktap -q -e 'var s = table.new(0, 20000) profile-1000us { s[stack(-1, 12)] += 1 }' &

 echo -e "\nktap tracing: profile-1000us { s[stack(-1, 12)] += 1 }"
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #------------------------------------------------------------#

 stap -o /dev/null -e 'global s[20000]; probe timer.profile { s[backtrace()] += 1 }' &

 echo -e "\nstap tracing: probe timer.profile { s[backtrace()] += 1 }"
 $COMMAND; $COMMAND; $COMMAND
 pkill stap

 #------------------------------------------------------------#

 stap -o /dev/null -e 'global s[20000]; probe timer.profile { s[backtrace()] <<< 1 }' &

 echo -e "\nstap tracing: probe timer.profile { s[backtrace()] <<< 1 }"
 $COMMAND; $COMMAND; $COMMAND
 pkill stap

 #------------------------------------------------------------#


 rm -rf ./sembench

```

test/benchmark/cmp_table.sh - ktap

```
#!/bin/sh

# This script compare table performance between ktap and stap.
#
# 1. ktap -e 'trace syscalls:sys_enter_futex { s[execname] += 1 }'
# 2. ktap -e 'kdebug.tracepoint("sys_enter_futex", function () { s[execname] += 1 })'
# 3. ktap -e 'kdebug.kprobe("SyS_futex", function () { s[execname] += 1 })'
# 4. stap -e 'probe syscall.futex { s[execname()] += 1 }'
# 5. ktap -e 'kdebug.kprobe("SyS_futex", function () { s[probename] += 1 })'
 # 6. stap -e 'probe syscall.futex { s[name] += 1 }'
 # 7. ktap -e 'kdebug.kprobe("SyS_futex", function () { s["constant_string_key"] += 1 })'
 # 8. stap -e 'probe syscall.futex { s["constant_string_key"] += 1 }'

 #Result:
 #Currently ktap table operation overhead is smaller than stap.


 gcc -o sembench sembench.c -O2 -lpthread

 COMMAND="./sembench -t 200 -w 20 -r 30 -o 2"

 #-------------------------------------------------------------#

 echo -e "without tracing:"
 $COMMAND; $COMMAND; $COMMAND

 #-------------------------------------------------------------#

 ../../ktap -q -e 'var s = {} trace syscalls:sys_enter_futex { s[execname] += 1 }' &

 echo -e "\nktap tracing: trace syscalls:sys_enter_futex { s[execname] += 1 }"
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'var s = {} kdebug.tracepoint("sys_enter_futex", function () {
     s[execname] += 1 })' &

 echo -e '\nktap tracing: kdebug.tracepoint("sys_enter_futex", function () { s[execname] += 1})'
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'var s = {} kdebug.kprobe("SyS_futex", function () {
     s[execname] += 1 })' &

 echo -e '\nktap tracing: kdebug.kprobe("SyS_futex", function () { s[execname] += 1 })'
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 stap -e 'global s; probe syscall.futex { s[execname()] += 1 }' &

 echo -e "\nstap tracing: probe syscall.futex { s[execname()] += 1 }"
 $COMMAND; $COMMAND; $COMMAND
 pkill stap

 #-------------------------------------------------------------#

 ../../ktap -q -e 'var s = {} kdebug.kprobe("SyS_futex", function () {
     s[probename] += 1 })' &

 echo -e '\nktap tracing: kdebug.kprobe("SyS_futex", function () { s[probename] += 1 })'
 $COMMAND; $COMMAND; $COMMAND
 pid=`pidof ktap`
```

```
disown $pid; kill -9 $pid; sleep 1

#-------------------------------------------------------------#

stap -e 'global s; probe syscall.futex { s[name] += 1 }' &

echo -e "\nstap tracing: probe syscall.futex { s[name] += 1 }"
$COMMAND; $COMMAND; $COMMAND
pkill stap

#-------------------------------------------------------------#

../../ktap -q -e 'var s = {} s["const_string_key"] = 0 kdebug.kprobe("SyS_futex", function () {
    s["const_string_key"] += 1 })' &

echo -e '\nktap tracing: kdebug.kprobe("SyS_futex", function () { s["const_string_key"] += 1 })'
$COMMAND; $COMMAND; $COMMAND
pid=`pidof ktap`
disown $pid; kill -9 $pid; sleep 1

#-------------------------------------------------------------#

stap -e 'global s; probe syscall.futex { s["const_string_key"] += 1 }' &

echo -e "\nstap tracing: probe syscall.futex { s["const_string_key"] += 1 }"
$COMMAND; $COMMAND; $COMMAND
pkill stap

#-------------------------------------------------------------#

stap -o /dev/null -e 'global s; probe syscall.futex { s["const_string_key"] <<< 1 }' &

echo -e "\nstap tracing: probe syscall.futex { s["const_string_key"] <<< 1 }"
$COMMAND; $COMMAND; $COMMAND
pkill stap

#-------------------------------------------------------------#


rm -rf ./sembench

```

One Level Up Top Level

test/benchmark/test.sh - ktap

```
#!/bin/sh

gcc -o sembench sembench.c -O2 -lpthread

COMMAND="./sembench -t 200 -w 20 -r 30 -o 2"

#-------------------------------------------------------------#

echo -e "without tracing:"
 #$COMMAND
 #$COMMAND
 #$COMMAND

 #-------------------------------------------------------------#

 ../../ktap -d -q -e 'trace syscalls:sys_enter_futex {}' &
 echo -e "\nktap tracing in dry-run mode: trace syscalls:sys_enter_futex {}"
 #$COMMAND
 #$COMMAND
 #$COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'trace syscalls:sys_enter_futex {}' &
 echo -e "\nktap tracing: trace syscalls:sys_enter_futex {}"
 #$COMMAND
 #$COMMAND
 #$COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'kdebug.tracepoint("sys_enter_futex", function () {})' &
 echo -e '\nktap tracing: kdebug.tracepoint("sys_enter_futex", function () {})'
 $COMMAND
 $COMMAND
 $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'kdebug.tracepoint("sys_enter_futex", function () {
     var arg = 1
         if (arg == 0x100 || arg == 0x200 || arg == 0x300 || arg == 0x400 ||
             arg == 0x500 || arg == 0x600 || arg == 0x700 || arg == 0x800 ||
             arg == 0x900 || arg == 0x1000) {
                 printf("%x %x\n", arg1, arg2)
         }}' &

 echo -e '\nktap tracing: kdebug.tracepoint("sys_enter_futex", function (xxx) {})'
 $COMMAND
 $COMMAND
 $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1


 #-------------------------------------------------------------#

 perf record -e syscalls:sys_enter_futex -a &
 echo -e "\nperf tracing: perf record -e syscalls:sys_enter_futex -a"
 $COMMAND
 $COMMAND
 $COMMAND
 pid=`pidof perf`
 disown $pid; kill -9 $pid; sleep 1; rm -rf perf.data

```

```
#-------------------------------------------------------------#

#lttng cannot compiled successful in my box.

#lttng create
#lttng enable-event -k syscalls:sys_enter_futex &
#echo -e "\nlttng tracing: lttng enable-event -k syscalls:sys_enter_futex"
#$COMMAND
#$COMMAND
#$COMMAND
#pid=`pidof perf`
#disown $pid; kill -9 $pid; sleep 1;
#lttng destroy



#-------------------------------------------------------------#

../../ktap -q -e 'trace syscalls:sys_enter_futex {
    if (arg2 == 0x100 || arg2 == 0x200 || arg2 == 0x300 || arg2 == 0x400 ||
        arg2 == 0x500 || arg2 == 0x600 || arg2 == 0x700 || arg2 == 0x800 ||
        arg2 == 0x900 || arg2 == 0x1000) {
        printf("%x %x\n", arg1, arg2)
    }}' &

echo -e "\nktap tracing: trace syscalls:sys_enter_futex { if (arg2 == 0x100 || arg2 == 0x200 ... }"
$COMMAND
$COMMAND
 $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#

 ../../ktap -q -e 'trace syscalls:sys_enter_futex /
     uaddr == 0x100 || uaddr == 0x200 || uaddr == 0x300 || uaddr == 0x400 ||
     uaddr == 0x500 || uaddr == 0x600 || uaddr == 0x700 || uaddr == 0x800 ||
     uaddr == 0x900 || uaddr == 0x1000/ {
         printf("%x %x\n", arg1, arg2)
     }' &

 echo -e "\nktap tracing: trace syscalls:sys_enter_futex /uaddr == 0x100 || uaddr == 0x200 .../ {}"
 $COMMAND
 $COMMAND
 $COMMAND
 pid=`pidof ktap`
 disown $pid; kill -9 $pid; sleep 1

 #-------------------------------------------------------------#
 #lttng don't support kerenl event filter now
 #-------------------------------------------------------------#

 #-------------------------------------------------------------#
 #systemtap compiled failure in my box
 #-------------------------------------------------------------#

 rm -rf ./sembench
```

One Level Up Top Level

runtime/ffi/ffi_call.c - ktap

Data types defined

- arg_status

Functions defined

- ffi_call
- ffi_call_unsupported
- ffi_call_x86_64
- ffi_get_arg_csym
- ffi_set_return
- ffi_type_check
- ffi_unpack

Macros defined

- ALIGN_STACK
- GPR_SIZE
- MAX_GPR
- MAX_GPR_SIZE
- NEWSTACK_SIZE
- REDZONE_SIZE
- STACK_ALIGNMENT
- ffi_call
- ffi_call_arch

Source code

```
1  /*
2  * ffi_call.c - foreign function calling library support for ktap
3  *
4  * This file is part of ktap by Jovi Zhangwei.
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10  * version 2, as published by the Free Software Foundation.
11  *
12  * ktap is distributed in the hope it will be useful, but WITHOUT
13  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15  * more details.
16  *
17  * You should have received a copy of the GNU General Public License along with
18  * this program; if not, write to the Free Software Foundation, Inc.,
19  * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
```

```
*/

#include <linux/ctype.h>
#include <linux/slab.h>
#include "../../include/ktap_types.h"
#include "../../include/ktap_ffi.h"
#include "../ktap.h"
#include "../kp_vm.h"
#include "../kp_obj.h"

static int ffi_type_check(ktap_state_t *ks, csymbol_func *csf, int idx)
{
    StkId arg;
    csymbol *cs;

    if (idx >= csymf_arg_nr(csf))
        return 0;
    arg = kp_arg(ks, idx + 1);
    cs = csymf_arg(ks, csf, idx);

    if (!kp_cdata_type_match(ks, cs, arg))
        return 0;
    else {
        kp_error(ks, "Cannot convert to csymbol %s for arg %d\n",
                csym_name(cs), idx);
        return -1;
    }
}

static csymbol *ffi_get_arg_csym(ktap_state_t *ks, csymbol_func *csf, int idx)
{
    StkId arg;
    csymbol *cs;

    if (idx < csymf_arg_nr(csf))
        return csymf_arg(ks, csf, idx);

    arg = kp_arg(ks, idx + 1);
    cs = id_to_csym(ks, ffi_get_csym_id(ks, "void *"));
    switch (ttypenv(arg)) {
    case KTAP_TYPE_LIGHTUSERDATA:
    case KTAP_TYPE_BOOLEAN:
    case KTAP_TYPE_NUMBER:
    case KTAP_TYPE_STRING:
        return cs;
    case KTAP_TYPE_CDATA:
        return cd_csym(ks, cdvalue(arg));
    default:
        kp_error(ks, "Error: Cannot get type for arg %d\n", idx);
        return cs;
    }
}

static void ffi_unpack(ktap_state_t *ks, char *dst, csymbol_func *csf,
        int idx, int align)
{
    StkId arg = kp_arg(ks, idx + 1);
    csymbol *cs = ffi_get_arg_csym(ks, csf, idx);
    size_t size = csym_size(ks, cs);

    /* initialize the destination section */
    memset(dst, 0, ALIGN(size, align));

    kp_cdata_unpack(ks, dst, cs, arg);
}

#ifdef __x86_64

enum arg_status {
    IN_REGISTER,
    IN_MEMORY,
    IN_STACK,
};

#define ALIGN_STACK(v, a) ((void *)(ALIGN(((uint64_t)v), a)))
#define STACK_ALIGNMENT 8
```

```
#define REDZONE_SIZE 128
#define GPR_SIZE (sizeof(void *))
#define MAX_GPR 6
#define MAX_GPR_SIZE (MAX_GPR * GPR_SIZE)
 #define NEWSTACK_SIZE 512

 #define ffi_call_arch(ks, cf, rvalue) ffi_call_x86_64(ks, cf, rvalue)

 extern void ffi_call_assem_x86_64(void *stack, void *temp_stack,
                    void *func_addr, void *rvalue, ffi_type rtype);

 static void ffi_call_x86_64(ktap_state_t *ks, csymbol_func *csf, void *rvalue)
 {
     int i;
     int gpr_nr;
     int arg_bytes; /* total bytes needed for exceeded args in stack */
     int mem_bytes; /* total bytes needed for memory storage */
     char *stack, *stack_p, *gpr_p, *arg_p, *mem_p, *tmp_p;
     int arg_nr;
     csymbol *rsym;
     ffi_type rtype;
     size_t rsize;
     bool ret_in_memory;
     /* New stack to call C function */
     char space[NEWSTACK_SIZE];

     arg_nr = kp_arg_nr(ks);
     rsym = csymf_ret(ks, csf);
     rtype = csym_type(rsym);
     rsize = csym_size(ks, rsym);
     ret_in_memory = false;
     if (rtype == FFI_STRUCT || rtype == FFI_UNION) {
         if (rsize > 16) {
             rvalue = kp_malloc(ks, rsize);
             rtype = FFI_VOID;
             ret_in_memory = true;
         } else {
             /* much easier to always copy 16 bytes from registers */
             rvalue = kp_malloc(ks, 16);
         }
     }

     gpr_nr = 0;
     arg_bytes = mem_bytes = 0;
     if (ret_in_memory)
         gpr_nr++;
     /* calculate bytes needed for stack */
     for (i = 0; i < arg_nr; i++) {
         csymbol *cs = ffi_get_arg_csym(ks, csf, i);
         size_t size = csym_size(ks, cs);
         size_t align = csym_align(ks, cs);
         enum arg_status st = IN_REGISTER;
         int n_gpr_nr = 0;
         if (size > 32) {
             st = IN_MEMORY;
             n_gpr_nr = 1;
         } else if (size > 16)
             st = IN_STACK;
         else
             n_gpr_nr = ALIGN(size, GPR_SIZE) / GPR_SIZE;

         if (gpr_nr + n_gpr_nr > MAX_GPR) {
             if (st == IN_MEMORY)
                 arg_bytes += GPR_SIZE;
             else
                 st = IN_STACK;
         } else
             gpr_nr += n_gpr_nr;
         if (st == IN_STACK) {
             arg_bytes = ALIGN(arg_bytes, align);
             arg_bytes += size;
             arg_bytes = ALIGN(arg_bytes, STACK_ALIGNMENT);
         }
         if (st == IN_MEMORY) {
             mem_bytes = ALIGN(mem_bytes, align);
             mem_bytes += size;
```

```
            mem_bytes = ALIGN(mem_bytes, STACK_ALIGNMENT);
        }
    }

    /* apply space to fake stack for C function call */
    if (16 + REDZONE_SIZE + MAX_GPR_SIZE + arg_bytes +
            mem_bytes + 6 * 8 >= NEWSTACK_SIZE) {
        kp_error(ks, "Unable to handle that many arguments by now\n");
        return;
    }
    stack = space;
    /* 128 bytes below %rsp is red zone */
    /* stack should be 16-bytes aligned */
    stack_p = ALIGN_STACK(stack + REDZONE_SIZE, 16);
    /* save general purpose registers here */
    gpr_p = stack_p;
    memset(gpr_p, 0, MAX_GPR_SIZE);
    /* save arguments in stack here */
    arg_p = gpr_p + MAX_GPR_SIZE;
    /* save arguments in memory here */
    mem_p = arg_p + arg_bytes;
    /* set additional space as temporary space */
    tmp_p = mem_p + mem_bytes;

    /* copy arguments here */
    gpr_nr = 0;
    if (ret_in_memory) {
        memcpy(gpr_p, &rvalue, GPR_SIZE);
        gpr_p += GPR_SIZE;
        gpr_nr++;
    }
    for (i = 0; i < arg_nr; i++) {
        csymbol *cs = ffi_get_arg_csym(ks, csf, i);
        size_t size = csym_size(ks, cs);
        size_t align = csym_align(ks, cs);
        enum arg_status st = IN_REGISTER;
        int n_gpr_nr = 0;
        if (size > 32) {
            st = IN_MEMORY;
            n_gpr_nr = 1;
        } else if (size > 16)
            st = IN_STACK;
        else
            n_gpr_nr = ALIGN(size, GPR_SIZE) / GPR_SIZE;

        if (st == IN_MEMORY)
            mem_p = ALIGN_STACK(mem_p, align);
        /* Tricky way about storing it above mem_p. It won't overflow
         * because temp region can be temporarily used if necesseary. */
        ffi_unpack(ks, mem_p, csf, i, GPR_SIZE);
        if (gpr_nr + n_gpr_nr > MAX_GPR) {
            if (st == IN_MEMORY) {
                memcpy(arg_p, &mem_p, GPR_SIZE);
                arg_p += GPR_SIZE;
            } else
                st = IN_STACK;
        } else {
            memcpy(gpr_p, mem_p, n_gpr_nr * GPR_SIZE);
            gpr_p += n_gpr_nr * GPR_SIZE;
            gpr_nr += n_gpr_nr;
        }
        if (st == IN_STACK) {
            arg_p = ALIGN_STACK(arg_p, align);
            memcpy(arg_p, mem_p, size);
            arg_p += size;
            arg_p = ALIGN_STACK(arg_p, STACK_ALIGNMENT);
        }
        if (st == IN_MEMORY) {
            mem_p += size;
            mem_p = ALIGN_STACK(mem_p, STACK_ALIGNMENT);
        }
    }

    kp_verbose_printf(ks, "Stack location: %p -redzone- %p -general purpose "
            "register used- %p -zero- %p -stack for argument- %p"
            " -memory for argument- %p -temp stack-\n",
```

```
248             stack, stack_p, gpr_p, stack_p + MAX_GPR_SIZE,
249             arg_p, mem_p);
250     kp_verbose_printf(ks, "GPR number: %d; arg in stack: %d; "
251             "arg in mem: %d\n",
252             gpr_nr, arg_bytes, mem_bytes);
253     kp_verbose_printf(ks, "Return: address %p type %d\n", rvalue, rtype);
254     kp_verbose_printf(ks, "Number of register used: %d\n", gpr_nr);
255     kp_verbose_printf(ks, "Start FFI call on %p\n", csf->addr);
256     ffi_call_assem_x86_64(stack_p, tmp_p, csf->addr, rvalue, rtype);
257 }
258
259 #else /* non-supported platform */
260
261 #define ffi_call(ks, cf, rvalue) ffi_call_unsupported(ks, cf, rvalue)
262
263 static void ffi_call_unsupported(ktap_state_t *ks,
264         csymbol_func *csf, void *rvalue)
265 {
266     kp_error(ks, "unsupported architecture.\n");
267 }
268
269 #endif /* end for platform-specific setting */
270
271
272 static int ffi_set_return(ktap_state_t *ks, void *rvalue, csymbol_id ret_id)
273 {
274     ktap_cdata_t *cd;
275     ffi_type type = csym_type(id_to_csym(ks, ret_id));
276
277     /* push return value to ktap stack */
278     switch (type) {
279     case FFI_VOID:
280         return 0;
281     case FFI_UINT8:
282     case FFI_INT8:
283     case FFI_UINT16:
284     case FFI_INT16:
285     case FFI_UINT32:
286     case FFI_INT32:
287     case FFI_UINT64:
288     case FFI_INT64:
289         set_number(ks->top, (ktap_number)rvalue);
290         break;
291     case FFI_PTR:
292         cd = kp_cdata_new_ptr(ks, rvalue, -1, ret_id, 0);
293         set_cdata(ks->top, cd);
294         break;
295     case FFI_STRUCT:
296     case FFI_UNION:
297         cd = kp_cdata_new_record(ks, rvalue, ret_id);
298         set_cdata(ks->top, cd);
299         break;
300     case FFI_FUNC:
301     case FFI_UNKNOWN:
302         kp_error(ks, "Error: Have not support ffi_type %s\n",
303                 ffi_type_name(type));
304         return 0;
305     }
306     incr_top(ks);
307     return 1;
308 }
309
310 /*
311 * Call C into function
312 * First argument should be function symbol address, argument types
313 * and return type.
314 * Left arguments should be arguments for calling the C function.
315 * Types between Ktap and C are converted automatically.
316 * Only support x86_64 function call by now
317 */
318 int ffi_call(ktap_state_t *ks, csymbol_func *csf)
319 {
320     int i;
321     int expected_arg_nr, arg_nr;
322     ktap_closure_t *cl;
323     void *rvalue;
```

```
324
325     expected_arg_nr = csymf_arg_nr(csf);
326     arg_nr = kp_arg_nr(ks);
327
328     /* check stack status for C call */
329     if (!csf->has_var_arg && expected_arg_nr != arg_nr) {
330         kp_error(ks, "wrong argument number %d, which should be %d\n",
331                 arg_nr, expected_arg_nr);
332         goto out;
333     }
334     if (csf->has_var_arg && expected_arg_nr > arg_nr) {
335         kp_error(ks, "argument number %d, which should be bigger than %d\n",
336                 arg_nr, expected_arg_nr);
337         goto out;
338     }
339
340     /* maybe useful later, leave it here first */
341     cl = clvalue(kp_arg(ks, arg_nr + 1));
342
343     /* check the argument types */
344     for (i = 0; i < arg_nr; i++) {
345         if (ffi_type_check(ks, csf, i) < 0)
346             goto out;
347     }
348
349     /* platform-specific calling workflow */
350     ffi_call_arch(ks, csf, &rvalue);
351     kp_verbose_printf(ks, "Finish FFI call\n");
352
353 out:
354     return ffi_set_return(ks, rvalue, csymf_ret_id(csf));
355 }
```

One Level Up Top Level

runtime/kp_bcread.h - ktap

Macros defined

- __KTAP_BCREAD_H__

Source code

```
1 #ifndef __KTAP_BCREAD_H__
2 #define __KTAP_BCREAD_H__
3
4 ktap_proto_t *kp_bcread(ktap_state_t *ks, unsigned char *buff, int len);
5
6 #endif /* __KTAP_BCREAD_H__ */
```

One Level Up Top Level

runtime/kp_mempool.h - ktap

Macros defined

- __KTAP_MEMPOOL_H__

Source code

```
1  #ifndef __KTAP_MEMPOOL_H__
2  #define __KTAP_MEMPOOL_H__
3
4  void *kp_mempool_alloc(ktap_state_t *ks, int size);
5  void kp_mempool_destroy(ktap_state_t *ks);
6  int kp_mempool_init(ktap_state_t *ks, int size);
7
8  #endif /* __KTAP_MEMPOOL_H__ */
```

One Level Up Top Level

runtime/kp_transport.h - ktap

Macros defined

- __KTAP_TRANSPORT_H__

Source code

```
1  #ifndef __KTAP_TRANSPORT_H__
2  #define __KTAP_TRANSPORT_H__
3
4  void kp_transport_write(ktap_state_t *ks, const void *data, size_t length);
5  void kp_transport_event_write(ktap_state_t *ks, struct ktap_event_data *e);
6  void kp_transport_print_kstack(ktap_state_t *ks, uint16_t depth, uint16_t skip);
7  void *kp_transport_reserve(ktap_state_t *ks, size_t length);
8  void kp_transport_exit(ktap_state_t *ks);
9  int kp_transport_init(ktap_state_t *ks, struct dentry *dir);
10
11  int _trace_seq_puts(struct trace_seq *s, const char *str);
12
13  #endif /* __KTAP_TRANSPORT_H__ */
```

runtime/ffi/call_x86_64.S - ktap

```
1  /*
2  * call_x86_64.S - assembly code to call C function and handle return value
3  *
4  * This file is part of ktap by Jovi Zhangwei
5  *
6  * Copyright (C) 2012-2013 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
7  *
8  * ktap is free software; you can redistribute it and/or modify it
9  * under the terms and conditions of the GNU General Public License,
10  * version 2, as published by the Free Software Foundation.
11  *
12  * ktap is distributed in the hope it will be useful, but WITHOUT
13  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
14  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
15  * more details.
16  *
17  * You should have received a copy of the GNU General Public License along with
18  * this program; if not, write to the Free Software Foundation, Inc.,
19  * 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
20  */
21
22
23 #ifdef __x86_64
24
25     .file "call_x86_64.S"
26     .text
27
28 /*    ffi_call_assem_x86_64(void *stack, void *temp_stack,
29 *        void *rvalue, void *func_addr, ffi_type rftype)
30 *    @stack: base address of register values and new stack
31 *    @temp_stack: stack to store temporary values
32 *    @func_addr: Function address
33 *    @rvalue: where to put return value
34 *    @rftype: FFI type of return value
35 */
36     .align 2
37     .globl    ffi_call_assem_x86_64
38     .type    ffi_call_assem_x86_64,@function
39
40 ffi_call_assem_x86_64:
41     movq    (%rsp), %rax    /* save return address */
42     /* move stuffs to temp memory region(void *temp_stack) */
43     movq    %rcx, (%rsi)    /* save pointer to return value */
44     movq    %r8, 8(%rsi)    /* save return_ffi_type */
45     movq    %rbp, 16(%rsi)    /* save %rbp */
46     movq    %rax, 24(%rsi)    /* save return address */
47     movq    %rsp, 32(%rsi)    /* save %rsp */
48     movq    %rsi, %rbp    /* point %rbp to temp memory region */
49
50     movq    %rdx, %r11    /* move function address to %r11 */
51
52     movq    %rdi, %r10    /* set %r10 point to register region */
53     movq    (%r10), %rdi    /* load registers */
54     movq    8(%r10), %rsi
55     movq    16(%r10), %rdx
56     movq    24(%r10), %rcx
57     movq    32(%r10), %r8
58     movq    40(%r10), %r9
59     xorq    %rax, %rax
60
61     leaq    48(%r10), %rsp
62
63     callq    *%r11
64
65     movq    32(%rbp), %rsp    /* restore %rsp */
66     movq    24(%rbp), %rcx    /* restore return address */
67     movq    %rcx, (%rsp)
68
69     movq    (%rbp), %rcx    /* get pointer to return value */
70     movq    8(%rbp), %r8    /* get return_ffi_type */
71     movq    16(%rbp), %rbp    /* restore rbp */
```

```
72
73      leaq    .Lreturn_table(%rip), %r11    /* start address of return_table */
74      movslq    (%r11, %r8, 8), %r11    /* fetch target address from table */
75      jmpq    *%r11            /* jump according to value in table */
76
77      .align 8
78  .Lreturn_table:
79      .quad    .Lreturn_void        /* FFI_VOID */
80      .quad    .Lreturn_uint8        /* FFI_UINT8 */
81      .quad    .Lreturn_int8        /* FFI_INT8 */
82      .quad    .Lreturn_uint16        /* FFI_UINT16 */
83      .quad    .Lreturn_int16        /* FFI_INT16 */
84      .quad    .Lreturn_uint32        /* FFI_UINT32 */
85      .quad    .Lreturn_int32        /* FFI_INT32 */
86      .quad    .Lreturn_uint64        /* FFI_UINT64 */
87      .quad    .Lreturn_int64        /* FFI_INT64 */
88      .quad    .Lreturn_ptr        /* FFI_PTR */
89      .quad    .Lreturn_func        /* FFI_FUNC */
90      .quad    .Lreturn_struct        /* FFI_STRUCT */
91      .quad    .Lreturn_unknown    /* FFI_UNKNOWN */
92
93      .align 8
94  .Lreturn_void:
95  .Lreturn_func:
96  .Lreturn_unknown:
97      retq
98      .align 8
99  .Lreturn_uint8:
100     movzbq    %al, %rax
101     movq    %rax, (%rcx)
102     retq
103     .align 8
104 .Lreturn_int8:
105     movsbq    %al, %rax
106     movq    %rax, (%rcx)
107     retq
108     .align 8
109 .Lreturn_uint16:
110     movzwq    %ax, %rax
111     movq    %rax, (%rcx)
112     retq
113     .align 8
114 .Lreturn_int16:
115     movswq    %ax, %rax
116     movq    %rax, (%rcx)
117     retq
118     .align 8
119 .Lreturn_uint32:
120     movl    %eax, %eax
121     movq    %rax, (%rcx)
122     retq
123     .align 8
124 .Lreturn_int32:
125     movslq    %eax, %rax
126     movq    %rax, (%rcx)
127     retq
128     .align 8
129 .Lreturn_uint64:
130 .Lreturn_int64:
131 .Lreturn_ptr:
132     movq    %rax, (%rcx)
133     retq
134 /* Struct type indicates that struct is put into at most two registers,
135  * and 16 bytes space is always available
136  */
137     .align 8
138 .Lreturn_struct:
139     movq    %rax, (%rcx)
140     movq    %rdx, 8(%rcx)
141     retq
142
143 #endif /* end for __x86_64 */
```

One Level Up Top Level

userspace/kp_parse.h - ktap

```
1
2 ktap_proto_t *kp_parse(LexState *ls);
3 ktap_str_t *kp_parse_keepstr(LexState *ls, const char *str, size_t l);
4
```

include/ktap_errmsg.h - ktap

Macros defined

- ERRDEF

Source code

```
1  /*
2  * VM error messages.
3  * Copyright (C) 2012-2014 Jovi Zhangwei <jovi.zhangwei@gmail.com>.
4  * Copyright (C) 2005-2014 Mike Pall.
5  */
6
7  /* Basic error handling. */
8  ERRDEF(ERRMEM,    "not enough memory")
9  ERRDEF(ERRERR,    "error in error handling")
10
11  /* Allocations. */
12  ERRDEF(STROV,    "string length overflow")
13  ERRDEF(UDATAOV,    "userdata length overflow")
14  ERRDEF(STKOV,    "stack overflow")
15  ERRDEF(STKOVM,    "stack overflow (%s)")
16  ERRDEF(TABOV,    "table overflow")
17
18  /* Table indexing. */
19  ERRDEF(NANIDX,    "table index is NaN")
20  ERRDEF(NILIDX,    "table index is nil")
21  ERRDEF(NEXTIDX,    "invalid key to " KTAP_QL("next"))
22
23  /* Metamethod resolving. */
24  ERRDEF(BADCALL,    "attempt to call a %s value")
25  ERRDEF(BADOPRT,    "attempt to %s %s " KTAP_QS " (a %s value)")
26  ERRDEF(BADOPRV,    "attempt to %s a %s value")
27  ERRDEF(BADCMPT,    "attempt to compare %s with %s")
28  ERRDEF(BADCMPV,    "attempt to compare two %s values")
29  ERRDEF(GETLOOP,    "loop in gettable")
30  ERRDEF(SETLOOP,    "loop in settable")
31  ERRDEF(OPCALL,    "call")
32  ERRDEF(OPINDEX,    "index")
33  ERRDEF(OPARITH,    "perform arithmetic on")
34  ERRDEF(OPCAT,    "concatenate")
35  ERRDEF(OPLEN,    "get length of")
36
37  /* Type checks. */
38  ERRDEF(BADSELF,    "calling " KTAP_QS " on bad self (%s)")
39  ERRDEF(BADARG,    "bad argument #%d to " KTAP_QS " (%s)")
40  ERRDEF(BADTYPE,    "%s expected, got %s")
41  ERRDEF(BADVAL,    "invalid value")
42  ERRDEF(NOVAL,    "value expected")
43  ERRDEF(NOCORO,    "coroutine expected")
44  ERRDEF(NOTABN,    "nil or table expected")
45  ERRDEF(NOLFUNC,    "ktap function expected")
46  ERRDEF(NOFUNCL,    "function or level expected")
47  ERRDEF(NOSFT,    "string/function/table expected")
48  ERRDEF(NOPROXY,    "boolean or proxy expected")
49  ERRDEF(FORINIT,    KTAP_QL("for") " initial value must be a number")
50  ERRDEF(FORLIM,    KTAP_QL("for") " limit must be a number")
51  ERRDEF(FORSTEP,    KTAP_QL("for") " step must be a number")
52
53  /* C API checks. */
54  ERRDEF(NOENV,    "no calling environment")
55  ERRDEF(CYIELD,    "attempt to yield across C-call boundary")
56  ERRDEF(BADLU,    "bad light userdata pointer")
57
58  /* Standard library function errors. */
59  ERRDEF(ASSERT,    "assertion failed!")
60  ERRDEF(PROTMT,    "cannot change a protected metatable")
61  ERRDEF(UNPACK,    "too many results to unpack")
62  ERRDEF(RDRSTR,    "reader function must return a string")
```

```
ERRDEF(PRTOSTR,    KTAP_QL("tostring") " must return a string to " KTAP_QL("print"))
ERRDEF(IDXRNG,    "index out of range")
ERRDEF(BASERNG,    "base out of range")
ERRDEF(LVLRNG,    "level out of range")
ERRDEF(INVLVL,    "invalid level")
ERRDEF(INVOPT,    "invalid option")
ERRDEF(INVOPTM,    "invalid option " KTAP_QS)
ERRDEF(INVFMT,    "invalid format")
ERRDEF(SETFENV,    KTAP_QL("setfenv") " cannot change environment of given object")
ERRDEF(CORUN,    "cannot resume running coroutine")
ERRDEF(CODEAD,    "cannot resume dead coroutine")
ERRDEF(COSUSP,    "cannot resume non-suspended coroutine")
ERRDEF(TABINS,    "wrong number of arguments to " KTAP_QL("insert"))
ERRDEF(TABCAT,    "invalid value (%s) at index %d in table for " KTAP_QL("concat"))
ERRDEF(TABSORT,    "invalid order function for sorting")
ERRDEF(IOCLFL,    "attempt to use a closed file")
ERRDEF(IOSTDCL,    "standard file is closed")
ERRDEF(OSUNIQF,    "unable to generate a unique filename")
ERRDEF(OSDATEF,    "field " KTAP_QS " missing in date table")
ERRDEF(STRDUMP,    "unable to dump given function")
ERRDEF(STRSLC,    "string slice too long")
ERRDEF(STRPATB,    "missing " KTAP_QL("[") " after " KTAP_QL("%f") " in pattern")
ERRDEF(STRPATC,    "invalid pattern capture")
ERRDEF(STRPATE,    "malformed pattern (ends with " KTAP_QL("%") ")")
ERRDEF(STRPATM,    "malformed pattern (missing " KTAP_QL("]") ")")
ERRDEF(STRPATU,    "unbalanced pattern")
ERRDEF(STRPATX,    "pattern too complex")
ERRDEF(STRCAPI,    "invalid capture index")
ERRDEF(STRCAPN,    "too many captures")
ERRDEF(STRCAPU,    "unfinished capture")
ERRDEF(STRFMT,    "invalid option " KTAP_QS " to " KTAP_QL("format"))
ERRDEF(STRGSRV,    "invalid replacement value (a %s)")
ERRDEF(BADMODN,    "name conflict for module " KTAP_QS)
ERRDEF(JITOPT,    "unknown or malformed optimization flag " KTAP_QS)

/* Lexer/parser errors. */
ERRDEF(XMODE,    "attempt to load chunk with wrong mode")
ERRDEF(XNEAR,    "%s near " KTAP_QS)
ERRDEF(XLINES,    "chunk has too many lines")
ERRDEF(XLEVELS,    "chunk has too many syntax levels")
ERRDEF(XNUMBER,    "malformed number")
ERRDEF(XLSTR,    "unfinished long string")
ERRDEF(XLCOM,    "unfinished long comment")
ERRDEF(XSTR,    "unfinished string")
ERRDEF(XESC,    "invalid escape sequence")
ERRDEF(XLDELIM,    "invalid long string delimiter")
ERRDEF(XTOKEN,    KTAP_QS " expected")
ERRDEF(XJUMP,    "control structure too long")
ERRDEF(XSLOTS,    "function or expression too complex")
ERRDEF(XLIMC,    "chunk has more than %d local variables")
ERRDEF(XLIMM,    "main function has more than %d %s")
ERRDEF(XLIMF,    "function at line %d has more than %d %s")
ERRDEF(XMATCH,    KTAP_QS " expected (to close " KTAP_QS " at line %d)")
ERRDEF(XFIXUP,    "function too long for return fixup")
ERRDEF(XPARAM,    "<name> or " KTAP_QL("...") " expected")
ERRDEF(XAMBIG,    "ambiguous syntax (function call x new statement)")
ERRDEF(XFUNARG,    "function arguments expected")
ERRDEF(XSYMBOL,    "unexpected symbol")
ERRDEF(XDOTS,    "cannot use " KTAP_QL("...") " outside a vararg function")
ERRDEF(XSYNTAX,    "syntax error")
ERRDEF(XFOR,    KTAP_QL("=") " or " KTAP_QL("in") " expected")
ERRDEF(XBREAK,    "no loop to break")
ERRDEF(XLUNDEF,    "undefined label " KTAP_QS)
ERRDEF(XLDUP,    "duplicate label " KTAP_QS)
ERRDEF(XGSCOPE,    "<goto %s> jumps into the scope of local " KTAP_QS)
ERRDEF(XEVENTDEF,"cannot parse eventdef " KTAP_QS)

/* Bytecode reader errors. */
ERRDEF(BCFMT,    "cannot load incompatible bytecode")
ERRDEF(BCBAD,    "cannot load malformed bytecode")

#undef ERRDEF

```

One Level Up Top Level

ktapvm.mod.c - ktap

Global variables defined

- __this_module
- __used

Source code

```
1  #include <linux/module.h>
2  #include <linux/vermagic.h>
3  #include <linux/compiler.h>
4
5  MODULE_INFO(vermagic, VERMAGIC_STRING);
6
7  struct module __this_module
8  __attribute__((section(".gnu.linkonce.this_module"))) = {
9      .name = KBUILD_MODNAME,
10     .init = init_module,
11 #ifdef CONFIG_MODULE_UNLOAD
12     .exit = cleanup_module,
13 #endif
14     .arch = MODULE_ARCH_INIT,
15 };
16
17 static const char __module_depends[]
18 __used
19 __attribute__((section(".modinfo"))) =
20 "depends=";
21
```

vim/ - ktap

- ftdetect/
- syntax/

vim/ftdetect/ - ktap

- ktap.vim

One Level Up Top Level

vim/ftdetect/ktap.vim - ktap

```
augroup filetype
  au BufNewFile,BufRead *.kp   set filetype=ktap
augroup end
```

One Level Up Top Level

vim/syntax/ - ktap

- ktap.vim

vim/syntax/ktap.vim - ktap

Global variables defined

- b:current_syntax
- did_lua_syntax_inits

Source code

```
" Vim syntax file
" Language:     ktap
" Maintainer:    Jovi Zhangwei <jovi.zhangwei@gmail.com>
" First Author:    Jovi Zhangwei <jovi.zhangwei@gmail.com>
" Last Change:    2013 Dec 19

" For version 5.x: Clear all syntax items
" For version 6.x: Quit when a syntax file was already loaded
if version < 600
   syn clear
 elseif exists("b:current_syntax")
   finish
 endif

 setlocal iskeyword=@,48-57,_,$

 syn keyword ktapStatement break continue return
 syn keyword ktapRepeat while for in
 syn keyword ktapConditional if else elseif
 syn keyword ktapDeclaration trace trace_end
 syn keyword ktapIdentifier var
 syn keyword ktapFunction function
 syn match   ktapBraces "[{}\[\]]"
 syn match   ktapParens "[()]"
 syn keyword ktapReserved argstr probename arg0 arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
 syn keyword ktapReserved cpu pid tid uid execname


 syn region ktapTraceDec start="\<trace\>"lc=5 end="{"me=s-1 contains=ktapString,ktapNumber
 syn region ktapTraceDec start="\<trace_end\>"lc=9 end="{"me=s-1 contains=ktapString,ktapNumber
 syn match ktapTrace contained "\<\w\+\>" containedin=ktapTraceDec

 syn region ktapFuncDec start="\<function\>"lc=8 end=":\|("me=s-1 contains=ktapString,ktapNumber
 syn match ktapFuncCall contained "\<\w\+\ze\(\s\|\n\)*("
 syn match ktapFunc contained "\<\w\+\>" containedin=ktapFuncDec,ktapFuncCall

 syn match ktapStat contained "@\<\w\+\ze\(\s\|\n\)*("

 " decimal number
 syn match ktapNumber "\<\d\+\>"
 " octal number
 syn match ktapNumber "\<0\o\+\>" contains=ktapOctalZero
 " Flag the first zero of an octal number as something special
 syn match ktapOctalZero contained "\<0"
 " flag an octal number with wrong digits
 syn match ktapOctalError "\<0\o*[89]\d*"
 " hex number
 syn match ktapNumber "\<0x\x\+\>"
 " numeric arguments
 syn match ktapNumber "\<\$\d\+\>"
 syn match ktapNumber "\<\$#"

 syn region ktapString oneline start=+"+ skip=+\\"+ end=+"+
 " string arguments
 syn match ktapString "@\d\+\>"
 syn match ktapString "@#"
 syn region ktapString2 matchgroup=ktapString start="\[\z(=*\)\[" end="\]\z1\]" contains=@Spell

 " syn keyword ktapTodo contained TODO FIXME XXX

```

```
syn match ktapComment "#.*"

" treat ^#! as special
syn match ktapSharpBang "^#!.*"


syn keyword ktapFunc printf print print_hist stack
syn keyword ktapFunc gettimeofday_us
syn keyword ktapFunc pairs

syn match ktapFunc /\<ffi\.cdef\>/
syn match ktapFunc /\<ffi\.new\>/
syn match ktapFunc /\<ffi\.free\>/
syn match ktapFunc /\<ffi\.C\>/



" Define the default highlighting.
" For version 5.7 and earlier: only when not done already
" For version 5.8 and later: only when an item doesn't have highlighting yet
if version >= 508 || !exists("did_lua_syntax_inits")
  if version < 508
    let did_lua_syntax_inits = 1
    command -nargs=+ HiLink hi link <args>
  else
    command -nargs=+ HiLink hi def link <args>
  endif

  HiLink ktapNumber        Number
  HiLink ktapOctalZero        PreProc " c.vim does it this way...
  HiLink ktapOctalError        Error
  HiLink ktapString        String
  HiLink ktapString2        String
  HiLink ktapTodo        Todo
  HiLink ktapComment        Comment
  HiLink ktapSharpBang        PreProc
  HiLink ktapStatement        Statement
  HiLink ktapConditional    Conditional
  HiLink ktapRepeat        Repeat
   HiLink ktapTrace        Function
   HiLink ktapFunc        Function
   HiLink ktapStat        Function
   HiLink ktapFunction        Function
   HiLink ktapBraces        Function
   HiLink ktapDeclaration    Typedef
   HiLink ktapIdentifier        Identifier
   HiLink ktapReserved        Keyword

   delcommand HiLink
 endif

 let b:current_syntax = "ktap"
```

One Level Up Top Level