Understanding, Scripting and Extending GDB

Kevin Pouget
Jean-François Méhaut, Fabrice Rastello

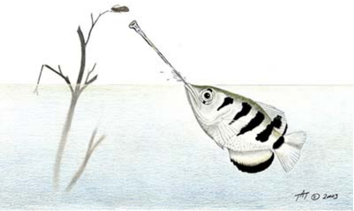
Université Grenoble Alpes / LIG, INRIA, CEA

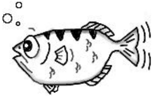
Séminaire CORSE, Aussois, France
4 janvier 2017

What is a debugger?

What is a debugger?

It's not a tool to remove bugs!





(not even to shoot them like the Archerfish of GDB's logo ;-)

What is a debugger?

It's not a tool to remove bugs!

Tools like GDB have the ability to ...

- access the program state
  - read and write memory cells and CPU registers ...
  - in the language's type system
- control the application execution
  - execute debugger-side code on specific events
  - execute process-side code on user demand

What is a debugger?

It's <u>not</u> a tool to <u>remove</u> bugs!

Tools like GDB have the ability to ...

- access the program state
  - read and write memory cells and CPU registers ...
  - in the language's type system
- control the application execution
  - execute debugger-side code on specific events
  - execute process-side code on user demand

Like python QEMU Valgrind ?

What is a debugger?

It's <u>not</u> a tool to <u>remove</u> bugs!

Tools like GDB have the ability to ...

- access the program state
- control the application execution

Like python QEMU Valgrind ?

Nope!

- the execution is <u>100% native</u>
- everything done through collaboration between ...
  - the OS, the compiler, the CPU ... and ol' hackers' tricks!

What is a debugger?

It's not a tool to remove bugs!

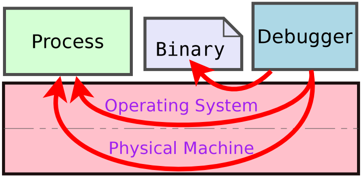
Tools like GDB have the ability to ...

- access the program state
- control the application execution

Like python QEMU Valgrind ?

Nope!

- the execution is 100% native
- everything done through collaboration between ...
  - the OS, the compiler, the CPU ... and ol' hackers' tricks!



Help from the compiler

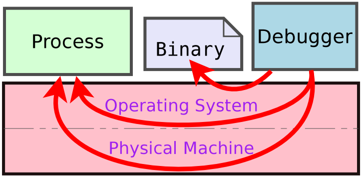
- DWARF debug info: type system and calling conventions

Help from the CPU

- not much (mainly watchpoint and instruction-level step-by-step)

Help from the OS

- ... the rest (access to the memory/registers + scheduler)



Help from the compiler

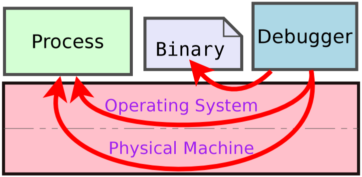
- DWARF debug info: type system and calling conventions

Help from the CPU

- not much (mainly watchpoint and instruction-level step-by-step)

Help from the OS

- ... the rest (access to the memory/registers + scheduler)



Help from the compiler

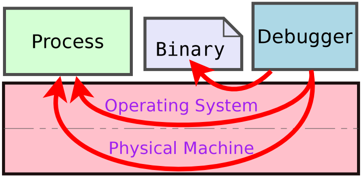
- DWARF debug info: type system and calling conventions

Help from the CPU

- not much (mainly watchpoint and instruction-level step-by-step)

Help from the OS

- ... the rest (access to the memory/registers + scheduler)



Help from the compiler

- DWARF debug info: type system and calling conventions

Help from the CPU

- not much (mainly watchpoint and instruction-level step-by-step)

Help from the OS

- ... the rest (access to the memory/registers + scheduler)

Agenda

Compiler Optimization and Runtime SystEms

1. GDB Under the Hood
2. Programming GDB in Python
3. New GDB Functionnalities

GDB Under the Hood: Definitions

- Stopping the execution ...

breakpoint	on an address execution
watchpoint	on an address access (read or write)
catchpoints	on particular events (signals, syscalls, fork/exec, ...)

- Controlling the execution:

`next/i`	go to next line/instruction
`step/i`	step into the current line's function call (if any)
`finish`	continue until the end of the current function
`return`	abort the current function call

GDB Under the Hood: Definitions

- Stopping the execution ...

breakpoint	on an address execution
watchpoint	on an address access (read or write)
catchpoints	on particular events (signals, syscalls, fork/exec, ...)

- Controlling the execution:

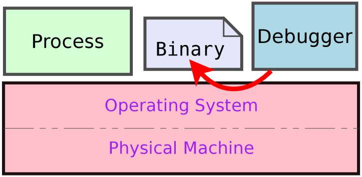
`next/i`	go to next line/instruction
`step/i`	step into the current line's function call (if any)
`finish`	continue until the end of the current function
`return`	abort the current function call

Agenda

Compiler Optimization and Runtime SystEms

Everything GDB knows about the language (DWARF)

- the type system
- the calling conventions and local variables
- the address-to-line mapping



`$ dwarfdump prodconsum` (see docker machine)

Under the Hood: Help from the Compiler

Everything GDB knows about the language (DWARF)

- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
struct Context {
        pthread_cond_t *cond;
        ...
};

void *consumer(void *_context){
       struct Context *context = ...;
       ...
}
```

Under the Hood: Help from the Compiler

Everything GDB knows about the language (Dwarf)

- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
DW_TAG_subprogram
   DW_AT_name          consumer
   DW_AT_decl_file     prodconsum.c
   DW_AT_type          <0x00000094> # void *
   DW_AT_low_pc        0x00400d47
   DW_AT_high_pc       <offset-from-lowpc>237
   ...
```

Everything GDB knows about the language (Dwarf)

- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
DW_TAG_subprogram
   DW_AT_name              consumer
   ...
   DW_TAG_formal_parameter
      DW_AT_name              _context
      DW_AT_decl_file         0x00000001 prodconsum.c
      DW_AT_decl_line         0x0000007b   # 123
      DW_AT_type              <0x00000094> # void *
      DW_AT_location          len 0x0002: 9158: DW_OP_fbreg -40
      ...
```

Under the Hood: Help from the Compiler

Everything GDB knows about the language (DWARF)

- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
DW_TAG_subprogram
   DW_AT_name             consumer
   ...
   DW_TAG_variable
      DW_AT_name             context
      DW_AT_decl_file        0x00000001 prodconsum.c
      DW_AT_decl_line        0x0000007d   # 125
      DW_AT_type             <0x00000596> # struct Context *
      DW_AT_location         len 0x0002: 9168: DW_OP_fbreg -24
      ...
```

Under the Hood: Help from the Compiler

Everything GDB knows about the language (Dwarf)

- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
DW_TAG_pointer_type               # <0x00000596> struct Context*
   DW_AT_byte_size                 0x00000008
   DW_AT_type                      <0x0000050a>
DW_TAG_structure_type             # <0x0000050a> struct Context
   DW_AT_name                      Context
   DW_AT_byte_size                 0x00000018
   DW_TAG_member
      DW_AT_name                      cond
      DW_AT_type                      <0x0000054c> # pthr_cond_t *
      DW_AT_data_member_location 0
```

Everything GDB knows about the language (DWARF)

- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
DW_TAG_pointer_type              # 0x00000094 void *
  DW_AT_byte_size                0x00000008


DW_TAG_base_type                 # 0x0000003f int
  DW_AT_name                     int
  DW_AT_byte_size                0x00000004
  DW_AT_encoding                 DW_ATE_signed
```

Everything GDB knows about the language (Dwarf)

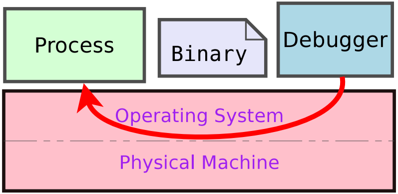
- the type system
- the calling conventions and local variables
- the address-to-line mapping

```
<pc>         [lno,col] NS BB ET PE EB IS= DI= uri: "filepath"
0x00400aa6   [  44, 0] NS uri: "prodconsum.c"
0x00400aae   [  46, 0] NS
0x00400abc   [  47, 0] NS
0x00400aca   [  48, 0] NS
0x00400ad1   [  50, 0] NS
0x00400ae2   [  51, 0] NS
0x00400af3   [  56, 0] NS
0x00400afd   [  57, 0] NS
...
```

Agenda

Under the Hood: Help from the OS

Everything GDB knows about the execution

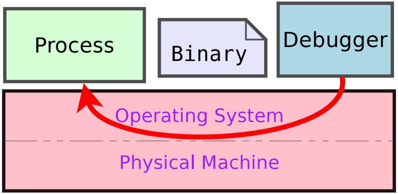


In LINUX: the `ptrace` API

- read/write access to memory addresses
- read/write access to CPU registers
- start/stop/interrupt the process
- a few more notifications...
  - catching syscalls

Under the Hood: Help from the OS

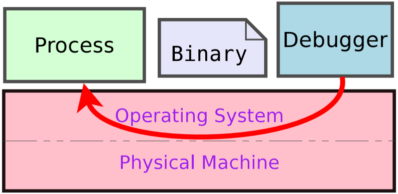
Everything GDB knows about the execution



- read/write access to memory addresses
  - `PTRACE_PEEKTEXT, PTRACE_PEEKUSER, PTRACE_POKE...`
  - `copy_to_user()`, `copy_from_user()`
- read/write access to CPU registers
- start/stop/interrupt the process
- a few more notifications...
  - catching syscalls
  - handling signals

Under the Hood: Help from the OS

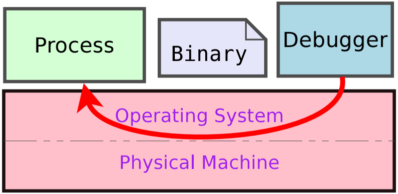
Everything GDB knows about the execution



- read/write access to memory addresses
  - `PTRACE_PEEKTEXT, PTRACE_PEEKUSER, PTRACE_POKE...`
  - `copy_to_user()`, `copy_from_user()`
- read/write access to CPU registers
  - registers are saved in the scheduler's `struct task_struct`
  - `copy_regset_to`, `copy_regset_from_user`
- start/stop/interrupt the process
- a few more notifications

Under the Hood: Help from the OS

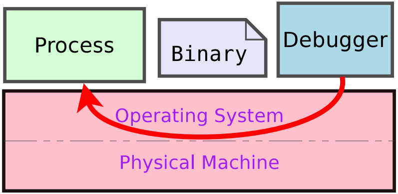
Everything GDB knows about the execution



- read/write access to memory addresses
- read/write access to CPU registers
- start/stop/interrupt the process
  - basic scheduler operations
  - ie: put it on the run-queue, send a signal-like interruption request, ...
- a few more notifications...
  - catching syscalls
  - handling signals

Under the Hood: Help from the OS

Everything GDB knows about the execution

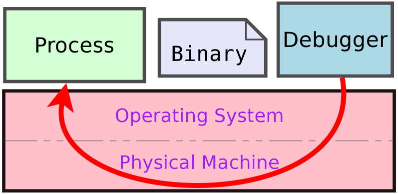


- read/write access to memory addresses
- read/write access to CPU registers
- start/stop/interrupt the process
- a few more notifications...
  - catching syscalls
  - handling signals
  - ...

Agenda

Compiler Optimization and Runtime SystEms

~~Everything GDB ...~~ Single-stepping and Watchpoints

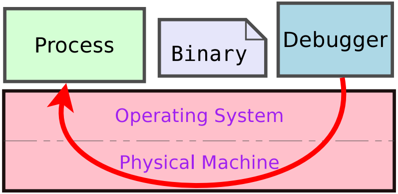


Single-stepping execute one CPU instruction

Watchpoint stop on memory-address reads and writes

- it's inefficient to implement in software
- main CPUs only have 4 debug registers

~~Everything GDB ...~~ Single-stepping and Watchpoints



Single-stepping execute one CPU instruction

Watchpoint stop on memory-address reads and writes

- it's inefficient to implement in software
- main CPUs only have 4 debug registers

Agenda

Compiler Optimization and Runtime SystEms

1. GDB Under the Hood
   - Help from the Compiler
   - Help from the OS
   - Help from the CPU
   - **Internal algorithms**
2. Programming GDB in Python
   - Python Interface Capabilities
   - Ex. 1: (re)discovering gdb-cli and gdb.py
   - Ex. 2: gdb simple scripting
3. New GDB Functionnalities
   - Section breakpoint
   - Return `true` breakpoint
   - Register watchpoint
   - Step into next call
   - Faking function execution

GDB Under the Hood: Internal algorithms

Callstack
- current frame saved on CPU registers (IP, FP, BP)
- older frames computed with calling conventions
  (⇔ where registers are stored)

Finish
- set a temporary breakpoint on the upper-frame PC
  (+ exception handlers / setjumps)

Step
- get current line's address boundaries in DWARF info
- single-step until out / in a new frame

Next
- same as `step`, but invoke `finish` in new frames

Callstack

- current frame saved on CPU registers (IP, FP, BP)
- older frames computed with calling conventions
  (⇔ where registers are stored)

Finish

- set a temporary breakpoint on the upper-frame PC
  (+ exception handlers / setjumps)

Step

- get current line's address boundaries in DWARF info
- single-step until out / in a new frame

Next

- same as `step`, but invoke `finish` in new frames

GDB Under the Hood: Internal algorithms

Compiler Optimization and Runtime SystEms

Callstack
- current frame saved on CPU registers (IP, FP, BP)
- older frames computed with calling conventions
  (⇔ where registers are stored)

Finish
- set a temporary breakpoint on the upper-frame PC
  (+ exception handlers / setjumps)

Step
- get current line's address boundaries in DWARF info
- single-step until out / in a new frame

Next
- same as `step`, but invoke `finish` in new frames

GDB Under the Hood: Internal algorithms

Compiler Optimization and Runtime SystEms

Callstack

- current frame saved on CPU registers (IP, FP, BP)
- older frames computed with calling conventions
  ($\Leftrightarrow$ where registers are stored)

Finish

- set a temporary breakpoint on the upper-frame PC
  ($+$ exception handlers / setjumps)

Step

- get current line's address boundaries in DWARF info
- single-step until out / in a new frame

Next

- same as `step`, but invoke `finish` in new frames

GDB Under the Hood: Internal algorithms

Callstack

- current frame saved on CPU registers (IP, FP, BP)
- older frames computed with calling conventions

  (⇔ where registers are stored)

Finish

- set a temporary breakpoint on the upper-frame PC

  (+ exception handlers / setjumps)

Step

- get current line's address boundaries in DWARF info
- single-step until out / in a new frame

Next

- same as `step`, but invoke `finish` in new frames

GDB Under the Hood: Internal algorithms

Catchpoint

- Kernel notification (via `ptrace`)

Watchpoint

- CPU notification to the kernel (trap)
- Kernel notification to GDB (`ptrace`)

or

- Instruction-by-instruction execution
- Instruction parsing to figure out reads and writes
  - $\Rightarrow$ very slow!

Breakpoint

- it's a bit more complicated ...

GDB Under the Hood: Internal algorithms

Catchpoint
- Kernel notification (via `ptrace`)

Watchpoint
- CPU notification to the kernel (trap)
- Kernel notification to GDB (`ptrace`)

or

- Instruction-by-instruction execution
- Instruction parsing to figure out reads and writes
  - $\Rightarrow$ very slow!

Breakpoint
- it's a bit more complicated ...

GDB Under the Hood: Internal algorithms

Catchpoint
- Kernel notification (via `ptrace`)

Watchpoint
- CPU notification to the kernel (trap)
- Kernel notification to GDB (`ptrace`)

or

- Instruction-by-instruction execution
- Instruction parsing to figure out reads and writes

  $\Rightarrow$ very slow!

Breakpoint
- it's a bit more complicated ...

GDB Under the Hood: Internal algorithms

Catchpoint

- Kernel notification (via `ptrace`)

Watchpoint

- CPU notification to the kernel (trap)
- Kernel notification to GDB (`ptrace`)

or

- Instruction-by-instruction execution
- Instruction parsing to figure out reads and writes

  $\Rightarrow$ very slow!

Breakpoint

- it's a bit more complicated ...

The algorithm behind breakpoints

- original_insn = *addr_to_breakpoint
- *addr_to_breakpoint = <special instruction>
- continue && wait(signal)
  - SIGTRAP if ISA has a breakpoint instruction (0xcc in x86)
  - SIGILL if illegal instruction
- if PC $\notin$ set(bpts): deliver(signal); done;
- otherwise: # breakpoint hit
  - cancel(signal)
  - **stop** if bpt.cli_condition() || bpt.py.stop() || ...
  - *addr_to_breakpoint = original_insn
  - cpu(single_step)
  - *addr_to_breakpoint = <special instruction>
  - continue && wait(...)

The algorithm behind breakpoints

- original_insn = *addr_to_breakpoint
- *addr_to_breakpoint = <special instruction>
- continue && wait(signal)
  - SIGTRAP if ISA has a breakpoint instruction (0xcc in x86)
  - SIGILL if illegal instruction
- if PC $\notin$ set(bpts): deliver(signal); done;
- otherwise: # breakpoint hit
  - cancel(signal)
  - **stop** if bpt.cli_condition() || bpt.py.stop() || ...
  - *addr_to_breakpoint = original_insn
  - cpu(single_step)
  - *addr_to_breakpoint = <special instruction>
  - continue && wait(...)

The algorithm behind breakpoints

- original_insn = *addr_to_breakpoint
- *addr_to_breakpoint = <special instruction>
- continue && wait(signal)
  - SIGTRAP if ISA has a breakpoint instruction (0xcc in x86)
  - SIGILL if illegal instruction
- if PC $\notin$ set(bpts): deliver(signal); done;
- otherwise: # breakpoint hit
  - cancel(signal)
  - **stop** if bpt.cli_condition() || bpt.py.stop() || ...
  - *addr_to_breakpoint = original_insn
  - cpu(single_step)
  - *addr_to_breakpoint = <special instruction>
  - continue && wait(...)

GDB Under the Hood: Internal algorithms

Compiler Optimization and Runtime SystEms

The algorithm behind breakpoints

- original_insn = *addr_to_breakpoint
- *addr_to_breakpoint = <special instruction>
- continue && wait(signal)
  - SIGTRAP if ISA has a breakpoint instruction (0xcc in x86)
  - SIGILL if illegal instruction
- if PC $\notin$ set(bpts): deliver(signal); done;
- otherwise: # breakpoint hit
  - cancel(signal)
  - **stop** if bpt.cli_condition() || bpt.py.stop() || ...
  - *addr_to_breakpoint = original_insn
  - cpu(single_step)
  - *addr_to_breakpoint = <special instruction>
  - continue && wait(...)

The algorithm behind breakpoints

- `original_insn = *addr_to_breakpoint`
- `*addr_to_breakpoint = <special instruction>`
- `continue && wait(signal)`
  - `SIGTRAP` if ISA has a breakpoint instruction (`0xcc` in x86)
  - `SIGILL` if illegal instruction
- `if PC` $\notin$ `set(bpts): deliver(signal); done;`
- `otherwise: # breakpoint hit`
  - `cancel(signal)`
  - **stop** `if bpt.cli_condition() || bpt.py.stop() || ...`
  - `*addr_to_breakpoint = original_insn`
  - `cpu(single_step)`
  - `*addr_to_breakpoint = <special instruction>`
  - `continue && wait(...)`

Agenda

Compiler Optimization and Runtime SystEms

Extending (not for today)

- pretty-printers — custom variable printing based on its type
- frame decorators — custom display of the callstack
- frame unwinders — tell GDB how your callstacks are structured
- more to come, eventually:
  - thread management and process abstractions
    - bypass existing process access mechanisms
    - access to embedded systems, virtual machines, core files ...
    - already possible but in C !

Scripting (for today)

GDB Python interface

Compiler Optimization and Runtime SystEms

Extending (not for today)

Scripting (for today)

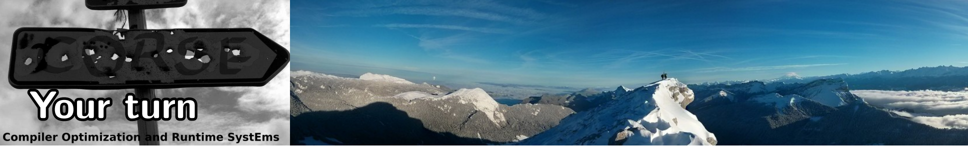
- values and types manipulation
- access the callstack and local variables, registers, ...
- create new commands
- action on breakpoints
- action on events (exec. stop/cont/exit, library loading, ...)
- ...
- for the rest: `gdb.execute("command", to_string=True)`

GDB Python interface

Extending (not for today)

Scripting (for today)

- values and types manipulation
- access the callstack and local variables, registers, ...
- create new commands
- action on breakpoints
- action on events (exec. stop/cont/exit, library loading, ...)
- ...
- for the rest: `gdb.execute("command", to_string=True)`



Your turn

Interactive part!

- `https://github.com/kpouget/tuto-gdb.py`
  - kpouget/tuto-gdb.py/blob/master/home/exercices.md
- `docker run -it`
  - `-v $HOME/gdb.py_debug:/home/gdb.py/host`
  - `-e GROUPID=$(id -g) -e USERID=$(id -u)`
  - `--cap-add sys_ptrace # or --priviledged`
  - `pouget/gdb-tuto`
- edit in `host@$HOME/gdb.py_debug` or `docker@~/host`
- consider adding this line in your `$HOME/.gdbinit`
  - `source $HOME/gdb.py_debug/gdbinit`

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable `print i`

```
(gdb) p context
$1 = {
  cond = 0x400e40 <__libc_csu_init>,
  mutex = 0x4009b0 <_start>,
  holder = -128,
  error = 32767
}
```

- print its type `ptype i`
- print it as another type `print (unsigned int) i`
- print its address / target `print &i; print *i`

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable — `print i`
- print its type — `ptype i`

```
(gdb) ptype context
type = volatile struct Context {
    pthread_cond_t *cond;
    thread_mutex_t *mutex;
    char holder;
    int error;
}
```

- print it as another type — `print (unsigned int) i`
- print its address / target — `print &i; print *i`

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable — `print i`
- print its type — `ptype i`
- print it as another type — `print (unsigned int) i`

```
(gdb) print (unsigned int) context.holder
$3 = 4294967168
```

- print its address / target — `print &i; print *i`
- evaluate C expression — `i + 1; i & 0x4`
- evaluate functions — `f(i)`

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable `print i`
- print its type `ptype i`
- print it as another type `print (unsigned int) i`
- print its address / target `print &i; print *i`

```
(gdb) p &context.mutex
$5 = (pthread_mutex_t **) 0x7fffffffe588

(gdb) p *context.mutex
$6 = {
  __data = {
    __lock = -1991643855,
  ...
```

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable — `print i`
- print its type — `ptype i`
- print it as another type — `print (unsigned int) i`
- print its address / target — `print &i; print *i`

```
# access to variables
i = gdb.parse_and_eval("i")                  <gdb.Value(int)>
i.type                                      <gdb.Type(int)>
uint = gdb.lookup_type("unsigned int")      <gdb.Type(uint)>
i.cast(uint)                                <gdb.Value(uint)>
gdb.newest_frame().read_var("i")
```

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable `print i`
- print its type `ptype i`
- print it as another type `print (unsigned int) i`
- print its address / target `print &i; print *i`

- evaluate C expression `i + 1; i & 0x4`
- evaluate functions `f(i)`

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

- print a variable `print i`
- print its type `ptype i`
- print it as another type `print (unsigned int) i`
- print its address / target `print &i; print *i`

- evaluate C expression `i + 1; i & 0x4`
- evaluate functions `f(i)`

```
(gdb) p puts("creating first thread") # print or call
creating first thread
$10 = 23
```

Your turn! print, evaluate, access, ...

```
# frame register access
gdb.newest_frame().older().read_reg("pc")
# function call
ret = gdb.parse_and_eval("puts")("text")     <gdb.Value()>
text
```

- disassemble a specified section of memory `disassemble main`
- in Python: `gdb.execute("disa fct", to_string=True)` or

```
frm = gdb.selected_frame()
frm.architecture().disassemble(frm.read_register("pc"))
[{'addr': 4595344, 'asm': 'sub $0x28,%rsp', 'length': 4}]
```

Your turn! print, evaluate, access, ...

Exercise 1: (re)discovering gdb-cli and gdb.py

Time to work!

Your turn! commands, breakpoints and events

Exercise 2: Hooking into gdb.py

Your turn! commands, breakpoints and events

Exercise 2: Hooking into gdb.py

Defining new commands

CLI

```
define cmd
  ...
  ...
end
```

Python

```
class MyCommand(gdb.Command):
  def __init__(self):
    gdb.Command.__init__(self, "cmd", gdb.COM)

  def invoke (self, args, from_tty):
     ...
```

Exercise 2: Hooking into gdb.py

- **Conditional breakpoints** `break <loc> if f(i) == &j`
  - internally, the breakpoint is hit all the time
  - but GDB only notifies the user if the condition is met

CLI

```
break fct
command
  silent
  print i
  cont
end
```

Your turn! commands, breakpoints and events

Exercise 2: Hooking into gdb.py

- **Conditional breakpoints**

```
break <loc> if f(i) == &j
```

  - internally, the breakpoint is hit all the time
  - but GDB only notifies the user if the condition is met

CLI

```
break fct
command
  silent
  print i
  cont
end
```

Python

```python
class MyBreakpoint(gdb.Breakpoint):
  def __init__(self):
    gdb.Breakpoint.__init__(self, "fct",
                            internal=True)
    self.silent = True
  def stop(self):
    print(gdb.parse_and_eval("i"))
    return True or False
```

Executing code on events

```
def say_hello(evt): print("hello")

gdb.events.stop.connect(say_hello) # then disconnect
gdb.events.cont
gdb.events.exited

gdb.events.new_objfile # shared library loads, mainly
gdb.events.clear_objfiles

gdb.events.inferior_call_pre/post
gdb.events.memory/register_changed # user-made changes

gdb.events.breakpoint_created/modified/deleted
```

Your turn! commands, breakpoints and events

Exercise 2: Hooking into gdb.py

Time to work!

Agenda

Compiler Optimization and Runtime SystEms

1. GDB Under the Hood
   - Help from the Compiler
   - Help from the OS
   - Help from the CPU
   - Internal algorithms
2. Programming GDB in Python
   - Python Interface Capabilities
   - Ex. 1: (re)discovering gdb-cli and gdb.py
   - Ex. 2: gdb simple scripting
3. New GDB Functionnalities
   - Section breakpoint
   - Return `true` breakpoint
   - Register watchpoint
   - Step into next call
   - Faking function execution

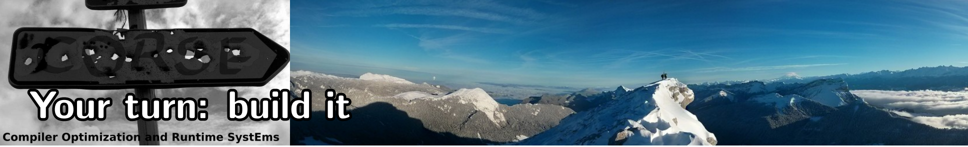
Your turn, part 2!

Adding new functionalities to GDB

1. Section breakpoint
   - `break_section start_profiling stop_profiling run`
2. Break when returned `true`
   - `break_return run 1`
3. Register watchpoint
   - `reg_watch eax main void *`
4. Step-to-next-call
   - `step-before-next-call`
   - `step-to-next-call`
5. Faking function execution
   - `skip_function run`
   - `fake_run_function`

---

https://sourceware.org/gdb/current/onlinedocs/gdb/Python-API.html



Your turn: build it

- `make all; make help`
- `make run_{section|return|watch|step|fake} DEMO={y|n}`
  - `DEMO=y` to run my code, `DEMO=n` for yours (default)

Your turn: section.c (1/2)

Compiler Optimization and Runtime SystEms

```
int main() {
  int i;

  srand(time(NULL));
  int bad = rand() % NB_ITER;

  for(i = 0; i < NB_ITER; i++) {
    if (i != bad) start_profiling();
    run(i); # calls bugs(i) if not profiling
    if (i != bad) stop_profiling();
  }
}
```

Your turn: section.c (2/2)


```
void start_profiling(void) {
  assert(!is_profiling);
  is_profiling = 1;
}
void stop_profiling(void) {
  assert(is_profiling);
  is_profiling = 0;
}
int run(int i) {
  if (!is_profiling) bug(i);

  return is_profiling;
}
```

Section breakpoint

Context

- We want to profile the function `run()`.
  - profiling starts with function `start_profiling()`
  - and stops with function `stop_profiling()`.

Problem

- `run()` is sometimes called outside of the profiling region.

⇒ we want to stop the debugger there.

```
(gdb) break_section start_profiling stop_profiling run
Section bpt set on start_profiling/run/stop_profiling
(gdb) run
Section breakpoint hit outside of section
15          if (!is_profiling) bug(i);
```

Section breakpoint

Idea:

- breakpoint on `start_profiling()` that sets a flag,
- breakpoint on `stop_profiling()` that unsets a flag,
- breakpoint on `run()` that checks the flag

Better:

- `start()` / `stop()` breakpoints enable/disable the bpt on `run()`

Return `true` breakpoint

Context

- I want to stop the execution whenever function `run()` has returned `true`.

Problem (kind of :)

- Function `run()` has many return statements
- I don't want to breakpoint all of them.

```
(gdb) break_return run 1
(gdb) run
Stopped after finding 'run' return value = 1 in $rax.
#0  0x00000000004006f7 in main () at section.c:36
```

Return `true` breakpoint

```
(gdb) break_return <fct> <expected value>
```

Idea:

- `BreakReturn_cmd.invoke`
  - parse and cast the expected value:
    ```
    gdb.parse_and_eval(<expected value>)
    ```
  - Function breakpoint on target function:
    ```
    FunctionReturnBreakpoint(<fct>, <expected value>)
    ```

- `FunctionReturnBreakpoint.prepare_before()`
  - before the function call: nothing to do

- `FunctionReturnBreakpoint.prepare_after()`
  - after the call: read register `eax`
    ```
    my_gdb.my_archi.return_value(<expected value>.type)
    ```

Register watchpoint

Context

- Inside a function, we want to see all the accesses to a register.

Problem

- GDB only supports memory watchpoints

```
(gdb) reg_watch eax main void *
20 watchpoints added in function main
(gdb) cont
before: (void *) 0xffffffffffffd256
   0x00000000004006a4 <+18>:        mov    %eax,%edi
after:  <unchanged>
(gdb) cont
before: (void *) 0xffffffffffffd256
   0x00000000004006be <+44>:        mov    %ecx,%eax
```

Register watchpoint

```
(gdb) reg_watch <reg name> <fct> [<fmt>]
```

Idea:

- ensure that target function exists

  ```
  if not gdb.lookup_symbol(fct)[0]:...
  ```

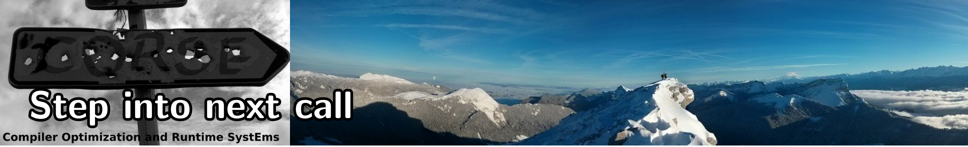
  - may through a `gdb.error` if there is no frame selected
- examine the function binary instructions
  - ```
    gdb.execute("disassemble {fct}", to_string=True)
    ```
- for all of them,
  - check if `<reg name>` appears
  - if yes, breakpoint it's address (`*addr`)
- ...
    ```

Register watchpoint

(gdb) reg_watch <reg name> <fct> [<fmt>]

Idea:

- on breakpoint hit:
  - read and print the current value of the register
    `gdb.parse_and_eval("({fmt}) ${regname}")`
  - print the line to be executed (from disassembly)
  - in `my_gdb.before_prompt`:
    - execute instruction (`nexti`)
    - re-read the register value
    - print it if different
  - mandatory stop here
    (GDB cannot `nexti` from a `Breakpoint.stop` callback)



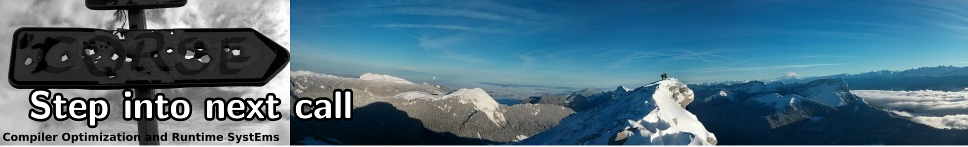
Step into next call

Compiler Optimization and Runtime SystEms

Context

- I want to step into the next function call, even if far away.
  - stop right before `step-before-next-call`
  - stop right after `step-into-next-call`

```
(gdb) step-before-next-call
step-before-next-call: next instruction is a call.
0x4006ed: callq  0x40062f <start_profiling>
(gdb) step-into-next-call
Stepped into function start_profiling
#0  start_profiling () at section.c:21
21          assert(!is_profiling);
#1  0x00000000004006f2 in main () at section.c:37
37              if (i != bad) start_profiling();
```



Step into next call

Idea:

- `step-before-next-call`:
  - run instruction by instruction
    `gdb.execute("stepi")`
  - until the current instruction contains a call
    `gdb.selected_frame().read_register("pc")`
    `arch = gdb.selected_frame().architecture()`
    `"call" in arch.disassemble(current_pc)[0]["asm"]`
- `step-into-next-call`:
  - run step by step: `gdb.execute("stepi")`
  - stop when the stack depth increases

```python
def callstack_depth():
  depth = 1; frame = gdb.newest_frame()
  while frame: frame = frame.older(); depth += 1
  return depth
```

Faking function execution

Context

- I don't want function `run()` code to execute,
- Instead I want to control its side effects from the debugger.

```
(gdb) run
BUG BUG BUG (i=<random>)

(gdb) skip_function run; run
[nothing]

(gdb) fake_run_function # calls bug(i) if not i % 10
BUG BUG BUG (i=0)
BUG BUG BUG (i=10)
BUG BUG BUG (i=20)...
```

Faking function execution

Idea:

- `skip_function <fct>`:
  - Breakpoint on `<fct>`, then call `return`:

    ```
    gdb.execute("return")
    ```

- `fake_run_function`:
  - as above, but run code before return:

    ```
    i = int(gdb.newest_frame().read_var("i"))
    ```

    ```
    if not i % 10:  gdb.execute("call bug({})".format(i))
    ```

Understanding, Scripting and Extending GDB

Kevin Pouget
Jean-François Méhaut, Fabrice Rastello

Université Grenoble Alpes / LIG, INRIA, CEA

Séminaire CORSE, Aussois, France
4 janvier 2017