Booting: From Power Up to Login Prompt

CSCI780: Linux Kernel Internals
Summer 2004

Scott Schneider

Big Picture

The Linux boot process is conceptually broken up into five main stages:

- **BIOS**: hardware initialization, gets boot loader up
- **Boot Loader**: loads kernel into executable state
- `setup`, `setup_32`: pre-kernel initialization
- `start_kernel()`: kernel initialization proper
- `init`: user level initialization process

The exact boot process differs depending on the architecture and the storage device Linux boots from. We're going to trace through an i386 boot from a hard drive; for us, that's the most common case.

My emphasis is on the kernel initialization itself, which means I'm not going to cover the BIOS or LILO in detail.

BIOS

The absolute beginning. Immediately after the power button is pressed:

- RESET pin of the CPU is raised.
- Registers are set to default values.
- The BIOS code at physical address `0xfffffff0` is executed.

The BIOS then:

- Power On Self Test (POST) which checks for and tests hardware.
- Initialize hardware devices, display PCI device list.
- Find an OS to boot; we're going to assume it found LILO on a hard drive
- Copies contents of the first sector into RAM and jumps into that code.

The primary goal here is to get the boot loader loaded into memory, and then jump to that code. Linux doesn't depend on the initializations that the BIOS does at this point; it redos everything, so we're not going to cover it in any more detail.

The Boot Loader: LILO

LILO (LInux LOader) is a single stage boot loader that has no knowledge of filesystems or Linux itself. It achieves everything through calls to the BIOS. Since it knows nothing of filesystems, it has to request files on the disk through the BIOS using physical disk addresses, provided by a map file. The map file is configured when LILO is installed.

After loaded, LILO:

- Loads the map file using the BIOS and displays the boot message.
- Prompts user to select a kernel.
- Loads the kernel using BIOS calls and the values in the map file.
- Executes the kernel.

The Boot Loader: GRUB

While LILO is easy to understand, no one uses it anymore. The most common boot loader used for Linux is GRUB (GRand Unified Bootloader).

GRUB starts out similar to LILO, but it loads more code in stages so that it understands the filesystem and even the network. A command line is eventually exposed, as opposed to a simple menu to select a kernel as LILO does.

There's quite a lot more to the BIOS and boot loaders, but we're going to end our discussion of them here so we can get down to the kernel code.

`setup` *or* Pre-Pre-Pre-Kernel

This is it. This is the beginning, the entry point for all i386 kernel code. Keep in mind that at this point, we still don't have an executable kernel image in memory, and we still have to start up just about all hardware. Which the BIOS mostly already did.

The Linux kernel is paranoid. It doesn't trust the BIOS to initialize everything, so it goes through and does it again.

From (arch/i386/boot/setup.S):

```
start:
        jmp     trampoline
 . . .
trampoline:     call    start_of_setup

 start_of_setup:
 . . .
# Check if an old loader tries to load a big-kernel
        testb   $LOADED_HIGH, %cs:loadflags     # Do we have a big kernel?
        jz      loader_ok                       # No, no danger for old loaders.

        cmpb    $0, %cs:type_of_loader          # Do we have a loader that
                                                # can deal with us?
        jnz     loader_ok                       # Yes, continue.
```

This is the first time we encounter the concept of a "low" and "high" loaded kernels. A low loaded kernel is a smaller image, produced with `make zImage` and a high loaded kernel is larger, produced with `make bzImage`. We're going to assume we have a bzImage, high loaded kernel.

Still in (arch/i386/boot/setup.S):

```
# Get memory size (extended mem, kB)

        xorl    %eax, %eax
        movl    %eax, (0x1e0)
        movb    $0x88, %ah
        int     $0x15                   Send interrupt to BIOS to get memory size.
        movw    %ax, (2)                Memory size is stored in %ax register.
```

There's three other methods for determining the memory size. The above method is the "Ye Olde Traditional Methode" which runs when `STANDARD_MEMORY_BIOS_CALL` is defined. The others are hairy.

Still in (`arch/i386/boot/setup.S`):

```
# Set the keyboard repeat rate to the max
        movw    $0x0305, %ax
        xorw    %bx, %bx
        int     $0x16
```

Send interrupt to the keyboard.

We initialize the keyboard very early - notice that we haven't even touched the CPU yet. Why?

Still in (`arch/i386/boot/setup.S`):

```
# Check for video adapter and its parameters and allow the
# user to browse video modes.
        call    video                           # NOTE: we need %ds pointing
                                                # to bootsector
```

Determine the video mode, set it correctly and test it. After this call, we have a monitor that can at least display some characters. Detailed monitor information has been tucked away so that the kernel proper can use it. All of this code is in `arch/i386/boot/video.S`; it's quite long and hairy. We're going to skip it, but feel free to take a look if you want to see what user menus look like when coded in assembly.

And those menus are why we need to initialize the keyboard so early; even at this early stage, we're depending on user input.

Still in (`arch/i386/boot/setup.S`):

```
# Get hd0 data...
        xorw    %ax, %ax
        movw    %ax, %ds                      %ds points to the boot sector.
        ldsw    (4 * 0x41), %si               Load boot sector information into memory.
        movw    %cs, %ax                          # aka SETUPSEG
        subw    $DELTA_INITSEG, %ax               # aka INITSEG
        pushw   %ax
        movw    %ax, %es
        movw    $0x0080, %di
        movw    $0x10, %cx
        pushw   %cx
        cld
        rep
        movsb
```

Now we know about our hard drive.

Still in (`arch/i386/boot/setup.S`):

```
# Check for PS/2 pointing device
        movw    %cs, %ax                          # aka SETUPSEG
        subw    $DELTA_INITSEG, %ax               # aka INITSEG
        movw    %ax, %ds
        movw    $0, (0x1ff)                       # default is no pointing device
        int     $0x11                             # int 0x11: equipment list
        testb   $0x04, %al                        # check if mouse installed
        jz      no_psmouse

        movw    $0xAA, (0x1ff)                    # device present
```

A general purpose "So, what do we have here?" interrupt is issued, and we check the corresponding bit for a PS/2 mouse.

Still in (`arch/i386/boot/setup.S`):

```
# Now we want to move to protected mode ...
        cmpw    $0, %cs:realmode_swtch
        jz      rmodeswtch_normal

. . .

rmodeswtch_normal:
        pushw   %cs
        call    default_switch

default_switch:
        cli                                     # no interrupts allowed !
        movb    $0x80, %al                      # disable NMI for bootup
                                                # sequence
        outb    %al, $0x70
        lret
```

This code doesn't actually switch us to protected mode. It just sets things up so that when we want to (right before we jump into the kernel), we're ready.

Still in (`arch/i386/boot/setup.S`):

```
# we get the code32 start address and modify the below 'jmpi'
# (loader may have changed it)
        movl    %cs:code32_start, %eax
        movl    %eax, %cs:code32
```

We need to make sure we're going to jump to the correct address when it's time to load the kernel.

```
# Now we move the system to its rightful place ... but we check if we have a
# big-kernel. In that case we *must* not move it ...
        testb   $LOADED_HIGH, %cs:loadflags
        jz      do_move0                        # .. then we have a normal low
                                                # loaded zImage
                                                # .. or else we have a high
                                                # loaded bzImage
        jmp     end_move                        # ... and we skip moving
```

And that did the actual move of the kernel to the correct place. Now we know the kernel is in a good and known place.

Still in (`arch/i386/boot/setup.S`):

```
 # set up gdt and idt
        lidt    idt_48                          # load idt with 0,0
        xorl    %eax, %eax                      # Compute gdt_base
        movw    %ds, %ax                        # (Convert %ds:gdt to a linear ptr)
        shll    $4, %eax
        addl    $gdt, %eax
        movl    %eax, (gdt_48+2)
        lgdt    gdt_48                          # load gdt with whatever is
```

Set up the Global Descriptor Table and Local Descriptor Table. Note that we have to be in real mode to do this, as protected mode depends on these.

Still in (`arch/i386/boot/setup.S`):

```
# well, that went ok, I hope. Now we mask all interrupts - the rest
# is done in init_IRQ().
        movb    $0xFF, %al                      # mask all interrupts for now
        outb    %al, $0xA1
        call    delay

        movb    $0xFB, %al                      # mask all irq's but irq2 which
        outb    %al, $0x21                      # is cascaded
```

Being interrupted now would be bad; we're about to do the switch to protected mode, and then the jump the kernel.

Still in (`arch/i386/boot/setup.S`):

```
# Well, that certainly wasn't fun :-(. Hopefully it works, and we don't
# need no steenking BIOS anyway (except for the initial loading :-).
# The BIOS-routine wants lots of unnecessary data, and it's less
# "interesting" anyway. This is how REAL programmers do it.
```

Read: masochistic and paranoid. We basically just redid most of the work the BIOS did.

```
# Well, now's the time to actually move into protected mode. To make
# things as simple as possible, we do no register set-up or anything,
# we let the gnu-compiled 32-bit programs do that. We just jump to
# absolute address 0x1000 (or the loader supplied one),
# in 32-bit protected mode.
#
# Note that the short jump isn't strictly needed, although there are
# reasons why it might be a good idea. It won't hurt in any case.
        movw    $1, %ax                         # protected mode (PE) bit
        lmsw    %ax                             # This is it!
        jmp     flush_instr
```

Now we're in protected mode.

Still in (`arch/i386/boot/setup.S`):

```
flush_instr:
        xorw    %bx, %bx                        # Flag to indicate a boot
        xorl    %esi, %esi                      # Pointer to real-mode code
        movw    %cs, %si
        subw    $DELTA_INITSEG, %si
        shll    $4, %esi                        # Convert to 32-bit pointer
        .byte 0x66, 0xea                        # prefix + jmpi-opcode
code32: .long   0x1000                          # will be set to 0x100000
                                                # for big kernels
.word   __KERNEL_CS
```

And that concludes our re-implementation of the BIOS. Which brings us to the . . .

`startup_32`, Take 1 *or* Pre-Pre-Kernel

We're still not in the kernel proper. Remember, our kernel image is still compressed. We need to decompress it before we can jump into it (`arch/i386/boot/compressed.S`):

```
startup_32:
 /*
  * Do the decompression, and jump to the new kernel..
  */
        subl $16,%esp   # place for structure on the stack
        movl %esp,%eax
        pushl %esi      # real mode pointer as second arg
        pushl %eax      # address of structure as first arg
        call SYMBOL_NAME(decompress_kernel)
        orl  %eax,%eax
        jnz  3f
        popl %esi       # discard address
        popl %esi       # real mode pointer
        xorl %ebx,%ebx
        ljmp $(__KERNEL_CS), $0x100000
```

`decompress_kernel()` is an interesting digression, so let's take a look inside.

`decompress_kernel()`

From (`arch/i386/boot/compressed/misc.c`):

```
asmlinkage int decompress_kernel(struct moveparams *mv, void *rmode)
{
        . . .
        puts("Uncompressing Linux... ");
        gunzip();
        puts("Ok, booting the kernel.\n");
        if (high_loaded) close_output_buffer_if_we_run_high(mv);
        return high_loaded;
}
```

`gunzip()` is a library routine in `lib/inflate.c`. Several standard functions have to be redefined in `misc.c` in order for `gunzip()` to work correctly, such as `error()`, `memcpy()`, `memset()`, `puts()`, `malloc()` and `free()`. The reimplementation of `malloc()` just uses a `long` to keep track of what physical memory has already been used; `free()` is even simpler:

```
static void free(void *where)
{       /* Don't care */
}
```

startup_32, **Take 2** *or* **Pre-Kernel**

This is the second setup_32 assembly function, from (arch/i386/kernel/head.S):

```
startup_32:
. . .
/*
 * Initialize page tables
 */
        movl $pg0-__PAGE_OFFSET,%edi /* initialize page tables */
        movl $007,%eax          /* "007" doesn't mean with right to kill, but
                                   PRESENT+RW+USER */
2:      stosl
        add $0x1000,%eax
        cmp $empty_zero_page-__PAGE_OFFSET,%edi
        jne 2b
```


Still in (arch/i386/kernel/head.S):

```
/*
 * Enable paging
 */
3:
         movl $swapper_pg_dir-__PAGE_OFFSET,%eax
         movl %eax,%cr3          /* set the page table pointer.. */
         movl %cr0,%eax
         orl $0x80000000,%eax
         movl %eax,%cr0          /* ..and set paging (PG) bit */
         jmp 1f                  /* flush the prefetch-queue */
1:
         movl $1f,%eax
         jmp *%eax               /* make sure eip is relocated */
1:
         /* Set up the stack pointer */
         lss stack_start,%esp
```

There's lots more initialization we're going to skip over, such as detecting what kind of CPU we're running.

Still in (arch/i386/kernel/head.S):

```
        xorl %eax,%eax
        lldt %ax
        cld                     # gcc2 wants the direction flag cleared at all times
#ifdef CONFIG_SMP
        movb ready, %cl
        cmpb $1,%cl
        je 1f                   # the first CPU calls start_kernel
                                # all other CPUs call initialize_secondary
        call SYMBOL_NAME(initialize_secondary)
        jmp L6
1:
#endif
        call SYMBOL_NAME(start_kernel)
L6:
        jmp L6                  # main should never return here, but
                                # just in case, we know what happens.
```

And that's it, we're now in the kernel proper.

start_kernel() *or* Kernel Proper

From (init/main.c):

```
asmlinkage void __init start_kernel(void)
{
 /*
  * Interrupts are still disabled. Do necessary setups, then
  * enable them
  */
        lock_kernel();                      Ensure only one CPU does initialization.
        printk(linux_banner);
        setup_arch(&command_line);          Setup drive, screen, paging, ACPI and device memory.
        printk("Kernel command line: %s\n", saved_command_line);
        parse_options(command_line);        kernel ... ro root=/dev/hda4 hdc=ide-scsi vga=791
        trap_init();
        init_IRQ();
        sched_init();
        softirq_init();
        time_init();
```

```
        console_init();                 Done early for debugging purposes.
#ifdef CONFIG_MODULES
        init_modules();                 Empty for i386. So what happens? See slide 31.
#endif
        . . .
        kmem_cache_init();
        sti();                          Enable interrupts again.
        calibrate_delay();              Determine jiffies.
. . .
        mem_init();
        kmem_cache_sizes_init();
        pgtable_cache_init();
        . . .
        fork_init(num_mappedpages);     Ensures thread structures don't take up more than half of memory.
        proc_caches_init();
        vfs_caches_init(num_physpages);
        buffer_init(num_physpages);
        page_cache_init(num_physpages);
        . . .
        signals_init();
#ifdef CONFIG_PROC_FS
        proc_root_init();                /proc file system initialization.
#endif
```



```
#if defined(CONFIG_SYSVIPC)
        ipc_init();                     Initializes System V semaphores, shared memory and messages.
#endif
        check_bugs();                   Lots of error checking, including the Pentium divide bug.
        printk("POSIX conformance testing by UNIFIX\n");

        /*
         *      We count on the initial thread going ok
         *      Like idlers init is an unlocked kernel thread, which will
         *      make syscalls (and thus be locked).
         */
        smp_init();
        rest_init();                    Initialize everything else - see next slide.
}
```

rest_init()

Still in (init/main.c):

```
/*
 * We need to finalize in a non-__init function or else race conditions
 * between the root thread and the init thread may cause start_kernel to
 * be reaped by free_initmem before the root thread has proceeded to
 * cpu_idle.
 */

static void rest_init(void)
{
        kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
        unlock_kernel();
        current->need_resched = 1;
        cpu_idle();
}
```

This thread is process 0 and becomes swapper, aka the idle process; forked thread is process 1 and becomes init (not to be confused with `init()`, which we are about to get into).

init()

Still in (init/main.c):

```
static int init(void * unused)
{
        lock_kernel();
        do_basic_setup();              See next slide.

        prepare_namespace();           See slide 33.
```

We're going to inerrupt the `init()` function to take a look at `do_basic_setup()` and `prepare_namespace()`. We shall return, however, as kernel level initialization finishes up in the `init()`.

do_basic_setup()

Still in (init/main.c):

```
/*
 * Ok, the machine is now initialized. None of the devices
 * have been touched yet, but the CPU subsystem is up and
 * running, and memory and process management works.
 *
 * Now we can finally start doing some real work..
 */
static void __init do_basic_setup(void)
{
        /*
         * Tell the world that we're going to be the grim
         * reaper of innocent orphaned children.
         *
         * We don't want people to have to make incorrect
         * assumptions about where in the task array this
         * can be found.
         */
        child_reaper = current;
```



```
        /*
         * Ok, at this point all CPU's should be initialized, so
         * we can start looking into devices..
         */
        . . .                           Lots of device initialization wrapped in conditionals.

        /* Networking initialization needs a process context */
        sock_init();

        start_context_thread();         keventd, initializes bottom half.
        do_initcalls();                 See next slide.
        . . .
#ifdef CONFIG_PCMCIA
        init_pcmcia_ds();               /* Do this last */
#endif
}
```

We're going to digress into `do_initcalls()` because it's an interesting aside.

do_initcalls()

Throughout the kernel, initialization functions are labeled with `__init`. That label tells the linker that these functions (and data) are to be treated differently than other functions. Specifically, the function's address needs to be placed, along with other initialization functions, in a well known place so that the kernel can call them on boot. From (`include/linux/init.h`):

```
typedef int (*initcall_t)(void);
. . .
extern initcall_t __initcall_start, __initcall_end;
. . .
#define __init          __attribute__ ((__section__ (".text.init")))
```


Back to (`init/main.c`):

```
static void __init do_initcalls(void)
{
        initcall_t *call;

        call = &__initcall_start;
        do {
              (*call)();
              call++;
        } while (call < &__initcall_end);

        /* Make sure there is no pending stuff from the initcall sequence */
        flush_scheduled_tasks();
}
```

After `do_initcalls()` all of the special initialization functions have been called. That's how the `init_modules()` function can be empty for i386; they get initialized here.

prepare_namespace()

This function sets up a properly mounted root filesystem. Very soon after this call, we depend on the filesystem to `exec` the `init` process.

From (`init/do_mounts.c`):

```
/*
 * Prepare the namespace - decide what/where to mount, load ramdisks, etc.
 */
void prepare_namespace(void)
{
        int is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;
        . . .
        sys_mkdir("/dev", 0700);
        sys_mkdir("/root", 0700);
        sys_mknod("/dev/console", S_IFCHR|0600, MKDEV(TTYAUX_MAJOR, 1));
        . . .
        create_dev("/dev/root", ROOT_DEV, NULL);
```



```
        if (mount_initrd) {
                if (initrd_load() && ROOT_DEV != MKDEV(RAMDISK_MAJOR, 0)) {
                        handle_initrd();
                        goto out;
                }
        } else if (is_floppy && rd_doload && rd_load_disk(0))
                ROOT_DEV = MKDEV(RAMDISK_MAJOR, 0);
        mount_root();
out:
        sys_umount("/dev", 0);
        sys_mount(".", "/", NULL, MS_MOVE, NULL);
        sys_chroot(".");
        mount_devfs_fs ();
}
```

Back in `init()`

We've looked at `do_basic_setup()` and `prepare_namespace()`, so we're back to finish up `init()` in (`init/main.c`):

```
        /*
         * Ok, we have completed the initial bootup, and
         * we're essentially up and running. Get rid of the
         * initmem segments and start the user-mode stuff..
         */
        free_initmem();
        unlock_kernel();

        if (open("/dev/console", O_RDWR, 0) < 0)
                printk("Warning: unable to open an initial console.\n");

        (void) dup(0);
        (void) dup(0);

        /*
         * We try each of these until one succeeds.
         *
```



```
         * The Bourne shell can be used instead of init if we are
         * trying to recover a really broken machine.
         */

        if (execute_command)
                execve(execute_command,argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/bin/sh",argv_init,envp_init);
        panic("No init found.  Try passing init= option to kernel.");
}
```

Notice that the `execve` depends on the filesystem. Also note that the path for `init` is hardcoded into the kernel. At this point, we have a fully initialized kernel. But the boot process is not over . . .

Big Init

The `init` user level process is not technically a part of the kernel, but it is still an extremely important part of a working Linux system. A discussion of the boot process is not complete without it.

It is process 1. As we've seen in previous presentations, it has special status within the kernel; `init` can not be killed. It is the root of the process creation tree, and reaps zombie processes.

The version of `init` that most Linux distributions have (well, at least ours) is a version that aims to meet System V specifications, written by Miquel van Smoorenburg. Our discussion will use his code.

The `init` program is a non-trivial piece of software, so we're only going to focus on the main points and aspects related to the boot process.

A Minimal Init

Before we actually start talking about the real deal, let's take a look at a bare-bones version of `init` to get an idea of what absolutely must take place. Alessandro Rubini wrote an excellent article for the Linux Journal in 1998 titled *Take Command: Init* that starts with this shell script as an example:

```
#!/bin/sh

# avoid typing full pathnames
export PATH=/usr/bin:/bin:/sbin:/usr/sbin

# remount root read-write, and mount all
mount -n -o remount,rw /
mount -a
swapon -a

# system log
syslogd
klogd
```

```
# start your lan
modprobe eth0 2> /dev/null
ifconfig eth0 192.168.0.1
route add 192.168.0.0 eth0
route add default gw 192.168.0.254

# start lan services
inetd
sendmail -bd -q30m

# Anything else: crond, named, ...

# And run one getty with a sane path
export PATH=/usr/bin:/bin
/sbin/mingetty tty1
```

We mount the root drives, startup system and kernel logging daemons, get a working network connection, and then execute `mingetty`. But what does it do?

What's Mingetty?

From the man page:

```
DESCRIPTION
       mingetty  is  a  minimal  getty  for  use  on virtual consoles.  Unlike
       agetty(8), mingetty is not suitable  for  serial  lines.   I  recommend
       using mgetty(8) for this purpose.
```

So that wasn't much help. What's a `getty`? From the man page:

```
DESCRIPTION
     getty is a program that is invoked by init(1M).  It is the second process
     in the series, (init-getty-login-shell) that ultimately connects a user
     with the UNIX system . . . Initially getty prints the
     contents of /etc/issue (if it exists), then prints the login message
     field for the entry it is using from /etc/gettydefs, reads the user's
     login name, and invokes the login(1) command with the user's name as
     argument.
```

Init Configuration

Now that we know the basic responsibilities the `init` process has, lets take a look at the configuration file for the real thing. From the file (`/etc/inittab`):

```
# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#

id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit
```

Our systems start X by default.

rc.sysinit is the initialization script that brings up various system services.

Still in (`/etc/inittab`):

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

Recall that `mingetty` sets up login prompts on incoming connections. Saying `respawn` means that when the connection is closed (the user logs out), we want to restart the `getty` on that line.

Miquel's Init

Having seen the configuration file, and with the knowledge of what `init` is minimally responsible for, let's take a look at some of the code from Miquel's verson. From (`init.h`):

```
/* Actions to be taken by init */
#define RESPAWN          1
#define WAIT             2
#define ONCE             3
#define BOOT             4
#define BOOTWAIT         5
#define POWERFAIL        6
#define POWERWAIT        7
#define POWEROKWAIT      8
#define CTRLALTDEL       9
#define OFF             10
#define ONDEMAND        11
#define INITDEFAULT     12
#define SYSINIT         13
#define POWERFAILNOW    14
#define KBREQUEST       15
```

Child is being respawned.

Makes sure a child that is already executed isn't respawned

This child is involved in bootup procedures.

Power is failing soon.

Power was failing, is okay now.

System shutdown on ctrl+alt+del.

We have the default run level.

This child is involved in system initialization.

Power is failing, shutting down **now**.

From (`init.h`):

```
/* Information about a process in the in-core inittab */
typedef struct _child_ {
  int flags;                /* Status of this entry */
  int exstat;               /* Exit status of process */
  int pid;                  /* Pid of this process */
  time_t tm;                /* When respawned last */
  int count;                /* Times respawned in the last 2 minutes */
  char id[8];               /* Inittab id (must be unique) */
  char rlevel[12];          /* run levels */
  int action;               /* what to do (see list below) */
} CHILD;

/* Values for the 'flags' field */
#define RUNNING          2       /* Process is still running */
#define KILLME           4       /* Kill this process */
#define DEMAND           8       /* "runlevels" a b c */
#define FAILING          16      /* process respawns rapidly */
#define WAITING          32      /* We're waiting for this process */
#define ZOMBIE           64      /* This process is already dead */
#define XECUTED          128     /* Set if spawned once or more times */
```

The Main Loop

Our last step is to look at the initial execution path of the `init` process.

```
int init_main()
{
        /*      Ignore all signals.
         */
        for(f = 1; f <= NSIG; f++)
                SETSIG(sa, f, SIG_IGN, SA_RESTART);

        SETSIG(sa, SIGALRM,  signal_handler, 0);
        SETSIG(sa, SIGHUP,   signal_handler, 0);
        SETSIG(sa, SIGINT,   signal_handler, 0);
        SETSIG(sa, SIGCHLD,  chld_handler, SA_RESTART);
        SETSIG(sa, SIGPWR,   signal_handler, 0);
        SETSIG(sa, SIGWINCH, signal_handler, 0);
        SETSIG(sa, SIGUSR1,  signal_handler, 0);
        SETSIG(sa, SIGSTOP,  stop_handler, SA_RESTART);
        SETSIG(sa, SIGTSTP,  stop_handler, SA_RESTART);
        SETSIG(sa, SIGCONT,  cont_handler, SA_RESTART);
        SETSIG(sa, SIGSEGV,  (void (*)(int))segv_handler, SA_RESTART);
```

`signal_handler` *queues signals.*

Does a wait for the child, cleans up.

Stops `init` *. . .*

And starts it back up again.


```
        console_init();

        /* Close whatever files are open, and reset the console. */
        close(0);
        close(1);
        close(2);
        console_stty();
        setsid();

        /*
         *      Set default PATH variable (for ksh)
         */
        if (getenv("PATH") == NULL) putenv(PATH_DFL);

        /*
         *      Start normal boot procedure.
         */
        runlevel = '#';
        read_inittab();

        start_if_needed();
```

Load the configuration file.

Run through the list of child processes and start them if needed.

```
        while(1) {

                /* See if we need to make the boot transitions. */
                boot_transitions();

                /* Check if there are processes to be waited on. */
                for(ch = family; ch; ch = ch->next)
                        if ((ch->flags & RUNNING) && ch->action != BOOT) break;

                if (ch != NULL && got_signals == 0) check_init_fifo();

                /* Check the 'failing' flags */
                fail_check();

                /* Process any signals. */
                process_signals();

                /* See what we need to start up (again) */
                start_if_needed();
        }
        /*NOTREACHED*/
}
```


And We're Done

We now have a fully working Linux system. The `while` loop executes for the entirety of when the system is up. (Well, almost. Sometimes `init` actually does a `fork-exec` on itself.)

We've discussed the BIOS, the boot loader, and completed a walkthrough of the kernel code from assembly startup, to C initialization, configuration scripts and ended with one very important user-level process.

References

- Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, 2nd Edition, O'Reilly & Associates, 2003.
- Alessandro Rubini & Jonathan Corbet. *Linux Device Drivers*, 2nd Edition, O'Reilly & Associates, 2003.
- `sysvinit` source code, Miquel van Smoorenburg.
  http://miquels.www.cistron.nl/
- Alessandro Rubini, *Take Command: Init*, The Linux Journal, November 1998.
  http://www.linux.it/kerneldocs/init/
- Kevin Boone, *Understanding the Linux Boot Process*.
  http://www.kevinboone.com/boot.html
- Linux man pages.
- Linux 2.4.21 source code.