The Linux 2.4 Kernel's Startup Procedure

Table of Contents

Copyright
About the Author

Overview

This paper describes the Linux 2.4 kernel's startup process, from the moment the kernel gets control of the host hardware until the kernel is ready to run user processes. Along the way, it covers the programming environment Linux expects at boot time, how peripherals are initialized, and how Linux knows what to do next.

Next >>>
The Big Picture

The Big Picture

Figure 1 is a function call diagram that describes the kernel's startup procedure. As it shows, kernel initialization proceeds through a number of distinct phases, starting with basic hardware initialization and ending with the kernel's launching of `/bin/init` and other user programs. The dashed line in the figure shows that `init()` is invoked as a kernel thread, not as a function call.

Figure 1. The kernel's startup procedure.

Figure 2 is a flowchart that provides an even more generalized picture of the boot process, starting with the bootloader extracting and running the kernel image, and ending with running user programs.

Figure 2. The kernel's startup procedure, in less detail.

The following sections describe each of these function calls, including examples taken from the Hitachi SH7750/Sega Dreamcast version of the kernel.

In The Beginning...

The Linux boot process begins with the kernel's `_stext` function, located in `arch/<host>/kernel/head.S`. This function is called `_start` in some versions. Interrupts are disabled at this point, and only minimal memory accesses may be possible depending on the capabilities of the host hardware.

The code to invoke `_stext` is found in the host's bootloader, code that is technically not part of Linux itself. In most kernel implementations the bootloader is a standalone program that drags the kernel image from the storage media (flash memory, hard disk, floppy, CD, etc.) into RAM, decompresses it, and jumps to `_stext`.

The bootloader extracts the compressed kernel image to a predefined location that is appropriate for the target architecture; `_stext` is, by convention, located 0x1000 bytes after the begininning of the image.

In the Hitachi SH Linux kernel, the linker places the start of the kernel image at the address `0x80000000` and defines a symbol called `_text` there. Once the kernel image is decompressed, the bootloader jumps to the address `(_text + 0x1000)`. Figure 3 shows the relevent portions of the bootloader's command file and startup code. The excerpts shown are taken from `arch/sh/vmlinux.lds.S` and `arch/sh/boot/compressed/head.S`.

```
/* linker configuration script */
SECTIONS
{
  . = 0x80000000 + CONFIG_MEMORY_START + 0x1000;
  _text = .;                    /* Text and read-only data */
  ...
}


/* bootloader code */
startup:
  ...

  /* Jump to the start of the decompressed kernel */
  mov.l   kernel_start_addr, r0
  jmp     @r0
```

```
  nop

kernel_start_addr:
        .long   _text+0x1000

```

Figure 3. The Hitachi SH kernel's linker configuration script and bootloader code.

<<< Previous | Home | Next >>>

The Big Picture | | The `_stext` Function

The _stext Function

The function `_stext` (called `_start` in some versions) is usually located in `arch/<host>/kernel/head.S`.

`_Stext` sets the initial stack pointer and performs any other functions necessary to create a minimal C runtime environment, like clearing the BSS memory section. `_Stext` then jumps to `start_kernel()`.

Figure 4 shows part of the code for the Hitachi SH version of `_stext`.

```
ENTRY(_stext)
   !    Initialize Status Register
   mov.l 1f, r0       ! MD=1, RB=0, BL=0, IMASK=0xF
   ldc   r0, sr
   !    Initialize global interrupt mask
   mov   #0, r0
   ldc   r0, r6_bank
   !
   mov.l 2f, r0
   mov   r0, r15      ! Set initial r15 (stack pointer)
   mov   #0x20, r1    !
   shll8 r1           ! r1 = 8192
   sub   r1, r0
   ldc   r0, r7_bank ! ... and init_task

   !    Initialize fpu
   mov.l   7f, r0
   jsr     @r0
    nop

   !    Enable cache
   mov.l 6f, r0
   jsr   @r0
   nop
   !    Clear BSS area
   mov.l 3f, r1
   add   #4, r1
   mov.l 4f, r2
   mov   #0, r0
9: cmp/hs       r2, r1
```

```
   bf/s   9b  ! while (r1 < r2)
   mov.l r0,@-r2
   !    Start kernel
   mov.l 5f, r0
   jmp   @r0
    nop

        .balign 4
1:      .long   0x400000F0              ! MD=1, RB=0, BL=0, FD=0, IMASK=0xF
2:      .long   SYMBOL_NAME(stack)
3:      .long   SYMBOL_NAME(__bss_start)
4:      .long   SYMBOL_NAME(_end)
5:      .long   SYMBOL_NAME(start_kernel)
6:      .long   SYMBOL_NAME(cache_init)
7:      .long   SYMBOL_NAME(fpu_init)
```

Figure 4. The Hitachi SH's `_stext` function.

The `start_kernel()` Function

`Start_kernel()` is located in `kernel/init/main.c`.

The `start_kernel()` function orchestrates all of Linux's startup procedure. Prior to invoking all the other functions needed to get the kernel into an operational state, `start_kernel()` prints the familiar Linux startup banner and parses the command line.

The following sections describe each of the functions called by `start_kernel()`, in the order of their invocation. The code in Figure 5 shows the first few lines of `start_kernel()`.

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    unsigned long mempages;
    extern char saved_command_line[];

    lock_kernel();
    printk(linux_banner);
    setup_arch(&command_line);
    printk("Kernel command line: %s\n", saved_command_line);
    parse_options(command_line);
    trap_init();
    init_IRQ();
    ...
```

Figure 5. The `start_kernel()` function.

The `setup_arch()` Function

`Setup_arch()` is usually located in `arch/<host>/kernel/setup.c`.

The `setup_arch()` function is responsible for initial, machine-specific initialization procedures. These include setting up the *machine vector* for the host, and determining the locations and sizes of available memory. `Setup_arch()` also initializes a basic memory allocator called *bootmem* to use during the boot process, and for most processors, calls `paging_init()` to enable the host's Memory Management Unit (MMU).

The host's *machine vector* is a data structure containing the name of the host, and function pointers for host-specific functions to read and write i/o ports. The machine vector reduces the number of configuration points in the kernel, by allowing host-specific operations in generic kernel code to use a common API.

<<< Previous

Next >>>

The `trap_init()` Function

`Trap_init()` is usually located in `arch/<host>/kernel/traps.c`.

`Trap_init()` initializes some of the processor's interrupt handling capability. In particular, it aims the processors interrupt vector table pointer to the address of the actual vector table, if necessary. Interrupts are not enabled until later on, just before the `calibrate_delay()` function is run.

The code in Figure 6 shows `trap_init()` for the Hitachi SH.

```
void __init trap_init(void)
{
   extern void *vbr_base;
   extern void *exception_handling_table[14];

   exception_handling_table[12] = (void *)do_reserved_inst;
   exception_handling_table[13] = (void *)do_illegal_slot_inst;

   asm volatile("ldc  %0, vbr"
                  : /* no output */
                  : "r" (&vbr_base)
                  : "memory");
}
```

Figure 6. The `trap_init()` function.

<<< Previous | Home | Next >>>

The `setup_arch()` Function | The `init_IRQ()` Function

The init_IRQ() Function

Init_IRQ() is usually located in arch/<host>/kernel/irq.c. The Hitachi SH version is in arch/<host>/kernel/irq_ipr.c.

Init_IRQ() initializes the hardware-specific side of the kernel's interrupt subsystem. Interrupt controllers are initialized here, but their input lines are not enabled until drivers and kernel modules call request_irq().

[<<< Previous] [Next >>>]

The sched_init() Function

Sched_init() is located in kernel/sched.c.

Sched_init() initializes the kernel's pidhash[] table, a lookup table for quickly mapping process IDs to process descriptors used by the kernel. The sched_init() function then initializes the vectors and bottom-half handlers used by the kernel's various internal timers.

<<< Previous	Home	Next >>>
The init_IRQ() Function		The softirq_init() Function

The softirq_init() Function

Softirq_init() is located in kernel/softirq.c.

Softirq_init() initializes the kernel's *softirq* subsystem. Softirqs are the 2.4 kernel's replacement for bottom-half handlers used in version 2.2.x, and are used to improve interrupt handling performance for things like network packet transmission and reception.

Softirqs are managed by the kernel's **ksoftirqd** thread.

The `time_init()` Function

`Time_init()` is usually located in `arch/<host>/kernel/time.c`.

The `time_init()` function initializes the host's system tick timer hardware. It installs the timer's interrupt handler, and configures the timer to produce a periodic tick. The tick interrupt handler is usually called `do_timer_interrupt()`.

A portion of the Hitachi SH version of `time_init()` is shown in Figure 7. The timer interrupt handler `timer_interrupt()` (which calls `do_timer_interrupt()`) is installed, then after determining the proper clock frequency (not shown), the code starts the chip's TMU0 periodic timer.

```
static struct irqaction irq0 = {timer_interrupt, SA_INTERRUPT, 0,
                                "timer", NULL, NULL};

void __init time_init(void)
{
   unsigned long interval;

   ...

   setup_irq(TIMER_IRQ, &irq0);

   ...

   interval = (module_clock/4 + HZ/2) / HZ;
   printk("Interval = %ld\n", interval);

   ...

   /* Start TMU0 */
   ctrl_outb(0, TMU_TSTR);
   ctrl_outb(TMU_TOCR_INIT, TMU_TOCR);
   ctrl_outw(TMU0_TCR_INIT, TMU0_TCR);
   ctrl_outl(interval, TMU0_TCOR);
   ctrl_outl(interval, TMU0_TCNT);
   ctrl_outb(TMU_TSTR_INIT, TMU_TSTR);
}
```

Figure 7. An excerpt from the Hitachi SH's `time_init()` function.

[<<< Previous] [Home] [Next >>>]
The `softirq_init()` Function
The `console_init()` Function

<<< Previous

Next >>>

The `console_init()` Function

`Console_init()` is located in `drivers/char/tty_io.c`.

The `console_init()` function performs early initialization of the kernel's serial console device, if one is configured for use. This console device is used to display kernel boot messages before the formal, complete virtual console system is initialized.

Once some basic TTY information is recorded, `console_init()` calls a host-specific console initialization function like `sci_console_init()`, which uses the Hitachi SH's `SCI` peripheral as a serial console.

In most cases, the kernel's console device is the host's VGA display hardware, or a serial port. By creating your own terminal initialization function for `console_init()`, however, just about any primitive interface is possible. Consoles that talk to network hosts can't be used here, since the kernel's networking subsystem has not yet been initialized.

<<< Previous | Home | Next >>>

The `time_init()` Function | | The `init_modules()` Function

The `init_modules()` Function

`Init_modules()` is located in `kernel/module.c`.

The `init_modules()` function initializes the kernel module subsystem's `nsyms` parameter.

<<<
Previous

Next >>>

The `kmem_cache_init()` Function

`Kmem_cache_init()` is located in `mm/slab.c`. This function initializes the kernel's SLAB memory management subsystem. SLABs are used for dynamic memory management of internal kernel structures.

<<< Previous | Home | Next >>>
The `init_modules()` Function | | The `calibrate_delay()` Function

The calibrate_delay() Function

Calibrate_delay() is located in init/main.c.

The calibrate_delay() function performs the kernel's infamous *BogoMips(tm)* calculation. A *BogoMip* is a unitless number that calibrates Linux's internal delay loops, so that delays run at roughly the same rate on processors of different speeds.

The BogoMips calculation depends on the value of jiffies, the number of timer ticks since system startup. If the system tick is not working, the BogoMips calculation will freeze.

The mem_init() Function

Mem_init() is located in arch/<host>/mm/init.c.

Mem_init() initializes the kernel's memory management subsystem. It also prints a tabulation of all available memory and the memory occupied by the kernel.

The `kmem_cache_sizes_init()` Function

`Kmem_cache_sizes_init()` is located in `mm/slab.c`.

The `kmem_cache_sizes_init()` function finishes the SLAB subsystem initialization started by `kmem_cache_init()`.

<<< Previous Next >>>

The `fork_init()` Function

`Fork_init()` is located in `kernel/fork.c`.

`Fork_init()` initializes the kernel's `max_threads` and `init_task` variables. This information is used by the kernel during `fork()` system calls.

<<< Previous Home Next >>>
The `kmem_cache_sizes_init()` Function
The `proc_caches_init()` Function

The proc_caches_init() Function

Proc_caches_init() is located in kernel/fork.c.

Proc_caches_init() initializes the SLAB caches used by the kernel. This is analogous to initializing malloc()-style heaps in a user program.

The `vfs_caches_init()` Function

`Vfs_caches_init()` function is located in `fs/dcache.c`.

`Vfs_caches_init()` initializes the SLAB caches used by the kernel's Virtual File System subsystem.

[<<< Previous]

[Next >>>]

The `buffer_init()` Function

`Buffer_init()` is located in `fs/buffer.c`.

`Buffer_init()` initializes the kernel's buffer cache hash table. The buffer cache holds blocks of adjacent disk data, and is used to improve the performance of reads and writes to hard disks and other slow media.

<<< Previous	Home	Next >>>
The `vfs_caches_init()` Function		The `page_cache_init()` Function

<<< Previous Next >>>

The `page_cache_init()` Function

`Page_cache_init()` function is located in `mm/filemap.c`.

`Page_cache_init()` initializes the kernel's page cache subsystem. Page caches hold streams of file data, and help improve performance when reading and writing user files.

<<< Previous Home Next >>>
The `buffer_init()` Function The `signals_init()` Function

The signals_init() Function

`Signals_init()` is located in `kernel/signal.c`.

`Signals_init()` initializes the kernel's signal queue. Signals are a form of interprocess communication.

The proc_root_init() Function

Proc_root_init() is located in fs/proc/root.c.

Proc_root_init() initializes Linux's /proc filesystem, and creates several standard entries like /proc/bus and /proc/driver.

The ipc_init() Function

Ipc_init() is located in ipc/util.c.

Ipc_int() initializes the resources that implement SystemV-style interprocess communication, including semaphores (initialized in the subfunction sem_init()), messages (msg_init()) and shared memory (shm_init()).

The check_bugs() Function

`Check_bugs()` is located somewhere in the host-specific portions of the kernel source tree. For some versions, it is declared in `include/asm-<host>/bugs.h`, so that it can be statically included in `start_kernel()`.

The `check_bugs()` function is where host-specific code can check for known processor errata, and implement workarounds if possible. Some implementations of this function check for FPU bugs, opcodes that are not supported by the whole processor family, and buggy opcodes.

`Check_bugs()` also usually calls `identify_cpu()`, to detect which version of a processor family is in use. For example, the x86 kernel's `identify_cpu()` can identify and apply runtime fixes for Coppermine, Celeron, and Pentium Pro/II/III processors, as well as chips from non-Intel vendors like AMD and Transmeta.

The smp_init() Function

Smp_init() is defined in init/main.c.

If the host machine is an SMP-capable x86 processor, smp_init() calls IO_APIC_init_uniprocessor() to set up the processor's APIC peripheral. For other processor families, smp_init() is defined as a do-nothing.

The `rest_init()` Function

`Rest_init()` is located in `init/main.c`.

`Rest_init()` frees the memory used by initialization functions, then launches `init()` as a kernel thread to finish the kernel's boot process.

The `init()` Function

`Init()` is located in `init/main.c`.

`Init()` completes the kernel's boot process by calling `do_basic_setup()` to initialize the kernel's PCI and network features. The remaining memory allocated for initialization is discarded, scheduling is enabled, the standard input, output and error streams are created, and `prepare_namespace()` is called to mount the root filesystem.

With the root filesystem in place, `init()` runs `execve()` to launch the program `/sbin/init`, if it exists. If a valid program name is provided with the `init=<programname>` command line option, `init()` will `execve()` that program instead. If a suitable startup program cannot be found (the kernel also tries "/bin/init" and "/bin/sh"), the kernel panics and halts.

The code for `init()` is shown in Figure 8.

```
static int init(void * unused)
{
   lock_kernel();
   do_basic_setup();

   prepare_namespace();

   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);

   if (execute_command) execve(execute_command,argv_init,envp_init);
   execve("/sbin/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.");
```

```
}
```

Figure 8. The `init()` function.

<<< Previous | Home | Next >>>

The `rest_init()` Function | | The `do_initcalls()` Function

The do_initcalls() Function

Do_initcalls() is located in init/main.c.

Do_initcalls() runs the list of functions registered with the __initcall attribute, which usually only applies to compiled-in kernel modules and device drivers. The __initcall attribute eliminates the need for a hand-maintained list of device driver initialization functions.

The __initcall mechanism works by creating a constant function pointer in a memory section called .initcall.init, that points to the initialization function itself. When the kernel image is linked, the linker organizes all of these function pointers into a single memory section, and do_initcalls() invokes them in the order they appear there.

The macros and type definitions that implement the __initcall attribute are shown in Figure 9; the code for do_initcalls() is shown in Figure 10.

```
typedef int (*initcall_t)(void);
typedef void (*exitcall_t)(void);

#define __initcall(fn)  \
    static initcall_t __initcall_##fn __init_call = fn

#define __init_call __attribute__ ((unused,__section__ (".initcall.init")))
```

Figure 9. The __initcall macros and typedefs.

```
extern initcall_t __initcall_start, __initcall_end;

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

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

    flush_scheduled_tasks();
```

```
}
```

Figure 10. The `do_initcalls()` function.

[<<< Previous] [Home] [Next >>>]
The `init()` Function The `mount_root()` Function

The mount_root() Function

Mount_root() is located in fs/super.c.

Mount_root() tries to mount the root filesystem. The identity of the root filesystem is provided as a kernel option during boot, which in workstation environments is typically the hard disk device and partition containing the system's root directory. (The root partition isn't necessarily the same as the location of the now almost-booted kernel image.)

Linux can mount root filesystems from hard disks, floppies, and over a network NFS connection to another machine. Linux can also use a *ramdisk* as a root filesystem. Mount_root() will try one or more of these sources before giving up and causing a kernel panic.

<<<
Previous

Next >>>

The `/sbin/init` Program

The program `/sbin/init` (hereafter called just **init**) is the parent of all user processes. **Init**'s job is to create other user processes by following the instructions found in the file `/etc/inittab`. Technically, the kernel itself has completely booted before **init** runs--- it has to, since **init** is a user process. Despite this, most consider **init** to be "part of the boot process".

The `inittab` script usually has entries to tell **init** to run programs like **mingetty** that provide login prompts, and to run scripts like those found in `/etc/rc.d/rc3.d` that in turn start still more processes and services like **xinetd**, **NFS**, and **crond**. As a result, a typical Linux workstation environment may have as many as 50 different processes running before the user even logs in for the first time.

Workstations usually modify system behavior by modifying the contents of `inittab`, or the contents of the subdirectories under `/etc/rc.d`. This capability makes it easy to make large-scale changes to a system's runtime behavior without needing to recompile (or in some cases, even reboot) the system. The conventions followed by `inittab` and `/etc/rc.d` scripts are well documented and pervasive (they predate Linux by a number of years), and lend themselves to automated modification during installation of user software.

To change the final stages of an embedded Linux startup process, you can either provide a modified `inittab` and run **init**, or you can replace **init** entirely, with an application of your own design--- perhaps your embedded application itself. You can even experiment a bit, by providing the names of programs like **/bin/sh** in the kernel's `init=` command line parameter of a Linux workstation. The kernel will simply run the specified program at the end of the boot process, instead of **init**.

Figure 11 shows an excerpt from a typical `inittab` file, that runs the scripts in `/etc/rc.d/rc.sysinit` and `/etc/rc.d/rc3.d`, and launches a few **mingetty**'s to provide login prompts.

```
id:3:initdefault:

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

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Things to run in every runlevel.
ud::once:/sbin/update

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# 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
```

Figure 11. Excerpt from a typical `inittab`.

<<< Previous | Home | Next >>>
The `mount_root()` Function | | Conclusion

Conclusion

Far from the mysterious entity that many claim it is, Linux's boot process is straightforward and easy to follow after spending a few minutes looking at the source code. Knowing Linux's boot process is important, because in embedded settings the generic boot procedure must almost always be modified to meet the needs of the target application. This document should help.

<<< Previous

Next >>>

Copyright

<<< Previous	Home	Next >>>
Conclusion		About the Author

About the Author

Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU-based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types.

Bill is a Contributing Editor for Embedded Systems Programming Magazine, a member of the Advisory Panel for the Embedded Systems Conference, maintainer of the Crossgcc FAQ, creator of the gdbstubs project, and a noted author and speaker.

Bill welcomes feedback and suggestions. Contact information is on his website, at http://www.billgatliff.com.
