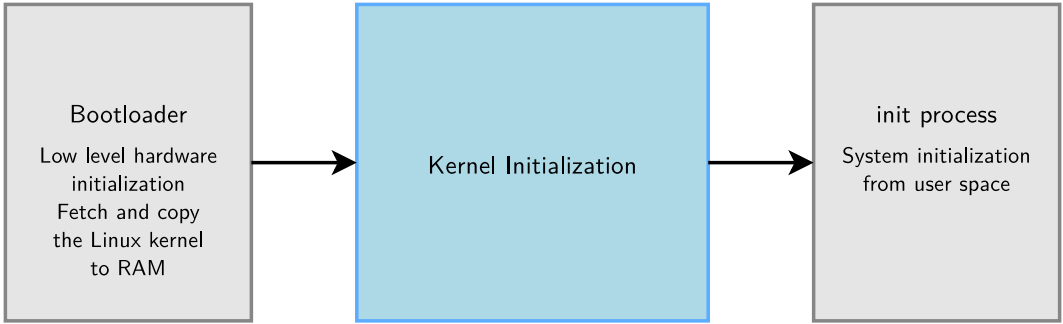
Linux kernel initialization

This file is an old chapter of Bootlin' embedded Linux kernel and driver development training materials (`https://bootlin.com/training/kernel/`), which has been removed and is no longer maintained.

PDF version and sources are available on
`https://bootlin.com/doc/legacy/kernel-init/`

Rights to copy

From Bootloader to user space

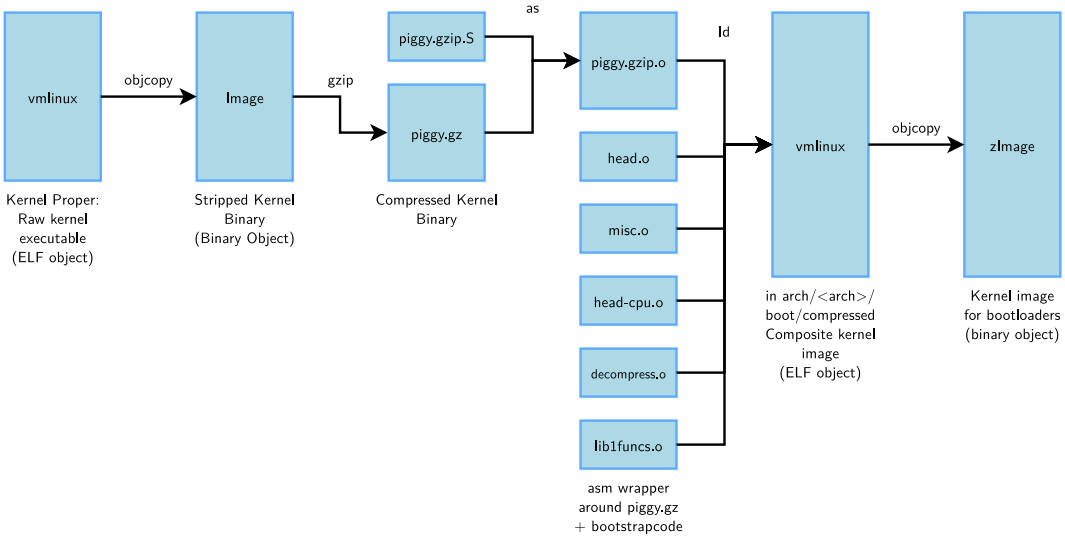


Kernel Bootstrap (1)

How the kernel bootstraps itself appears in kernel building. Example on ARM (pxa cpu) in Linux 2.6.36:

```
...
  LD      vmlinux
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  AS      arch/arm/boot/compressed/head.o
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  AS      arch/arm/boot/compressed/head-xscale.o
  SHIPPED arch/arm/boot/compressed/lib1funcs.S
  AS      arch/arm/boot/compressed/lib1funcs.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
```

Kernel Bootstrap (2)



Bootstrap Code for Compressed Kernels

- Located in `arch/<arch>/boot/compressed`
  - `head.o`
    - Architecture specific initialization code.
    - This is what is executed by the bootloader
  - `head-cpu.o` (here `head-xscale.o`)
    - CPU specific initialization code
  - `decompress.o`, `misc.o`
    - Decompression code
  - `piggy.<compressionformat>.o`
    - The kernel itself
- Responsible for uncompressing the kernel itself and jumping to its entry point.

Architecture-specific Initialization Code

- The uncompression code jumps into the main kernel entry point, typically located in `arch/<arch>/kernel/head.S`, whose job is to:
  - Check the architecture, processor and machine type.
  - Configure the MMU, create page table entries and enable virtual memory.
  - Calls the `start_kernel` function in `init/main.c`.
  - Same code for all architectures.
  - Anybody interested in kernel startup should study this file!

start_kernel Main Actions

- Calls `setup_arch(&command_line)`
  - Function defined in `arch/<arch>/kernel/setup.c`
  - Copying the command line from where the bootloader left it.
  - On arm, this function calls `setup_processor` (in which CPU information is displayed) and `setup_machine`(locating the machine in the list of supported machines).
- Initializes the console as early as possible (to get error messages)
- Initializes many subsystems (see the code)
- Eventually calls `rest_init`.

rest_init: Starting the Init Process

```
static noinline void __init_refok rest_init(void)
        __releases(kernel_lock)
{
        int pid;

        rcu_scheduler_starting();
        /*
         * We need to spawn init first so that it obtains pid 1, however
         * the init task will end up wanting to create kthreads, which, if
         * we schedule it before we create kthreadd, will OOPS.
         */
        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
        numa_default_policy();
        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
        rcu_read_lock();
        kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
        rcu_read_unlock();
        complete(&kthreadd_done);

        /*
         * The boot idle thread must execute schedule()
         * at least once to get things moving:
         */
        init_idle_bootup_task(current);
        preempt_enable_no_resched();
        schedule();
        preempt_disable();

        /* Call into cpu_idle with preempt disabled */
        cpu_idle();
}
```

kernel_init

- `kernel_init` does two main things:
  - Call `do_basic_setup`
  - Once kernel services are ready, start device initialization (Linux 2.6.36 code excerpt):

```
static void __init do_basic_setup(void)
{
        cpuset_init_smp();
        usermodehelper_init();
        init_tmpfs();
        driver_init();
        init_irq_proc();
        do_ctors();
        do_initcalls();
}
```

  - Call `init_post`

do_initcalls

Calls pluggable hooks registered with the macros below. Advantage: the generic code doesn't have to know about them.

```
/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 */
#define pure_initcall(fn)               __define_initcall("0",fn,1)

#define core_initcall(fn)               __define_initcall("1",fn,1)
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)             __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)        __define_initcall("4s",fn,4s)
#define fs_initcall(fn)                 __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)            __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)             __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)             __define_initcall("6",fn,6)
#define device_initcall_sync(fn)        __define_initcall("6s",fn,6s)
#define late_initcall(fn)               __define_initcall("7",fn,7)
#define late_initcall_sync(fn)          __define_initcall("7s",fn,7s)
```

Defined in `include/linux/init.h`

initcall example

From `arch/arm/mach-pxa/lpd270.c` (Linux 2.6.36)

```
static int __init lpd270_irq_device_init(void)
{
    int ret = -ENODEV;
    if (machine_is_logicpd_pxa270()) {
        ret = sysdev_class_register(&lpd270_irq_sysclass);
        if (ret == 0)
            ret = sysdev_register(&lpd270_irq_device);
    }
    return ret;
}

device_initcall(lpd270_irq_device_init);
```

init_post

- The last step of Linux booting
  - First tries to open a console
  - Then tries to run the init process, effectively turning the current kernel thread into the user space init process.

init_post Code: init/main.c

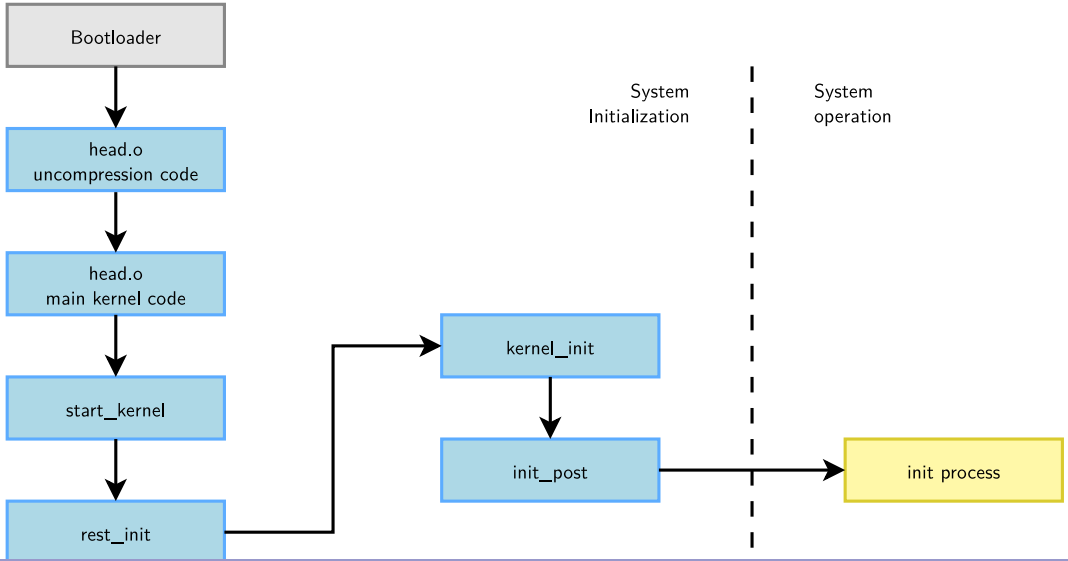
```c
static noinline int init_post(void) __releases(kernel_lock) {
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    current->signal->flags |= SIGNAL_UNKILLABLE;
    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n", ramdisk_execute_command);
    }

    /* 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) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel. See Linux Documentation/init.txt");
}
```

Kernel Initialization Graph



Kernel Initialization - Summary

- The bootloader executes bootstrap code.
- Bootstrap code initializes the processor and board, and uncompresses the kernel code to RAM, and calls the kernel's `start_kernel` function.
- Copies the command line from the bootloader.
- Identifies the processor and machine.
- Initializes the console.
- Initializes kernel services (memory allocation, scheduling, file cache...)
- Creates a new kernel thread (future init process) and continues in the idle loop.
- Initializes devices and execute initcalls.