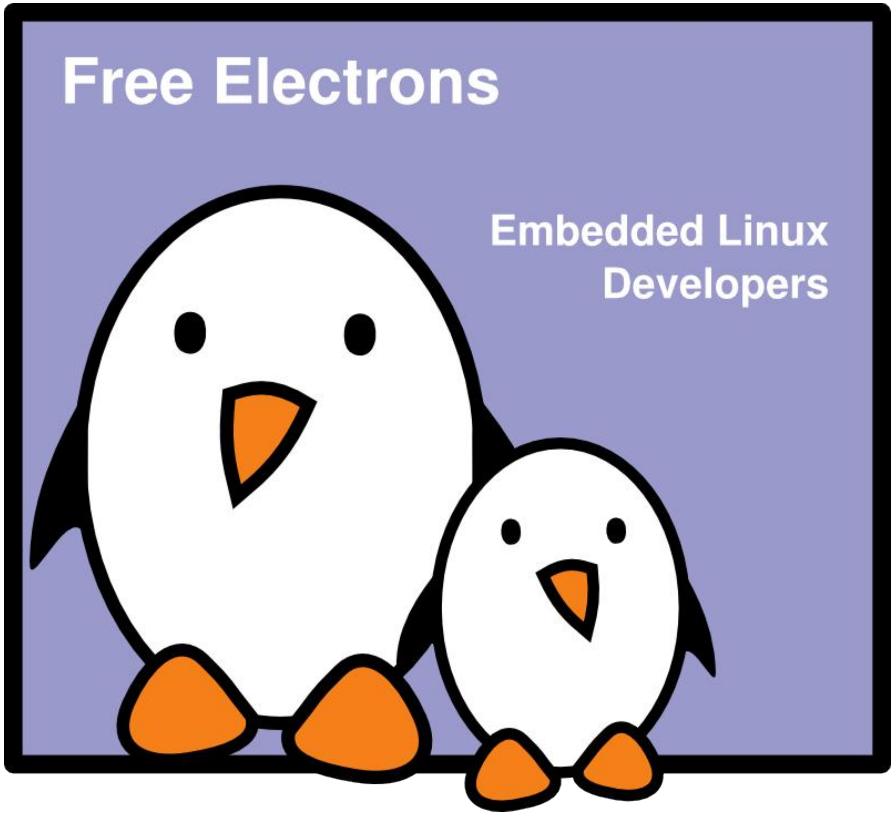
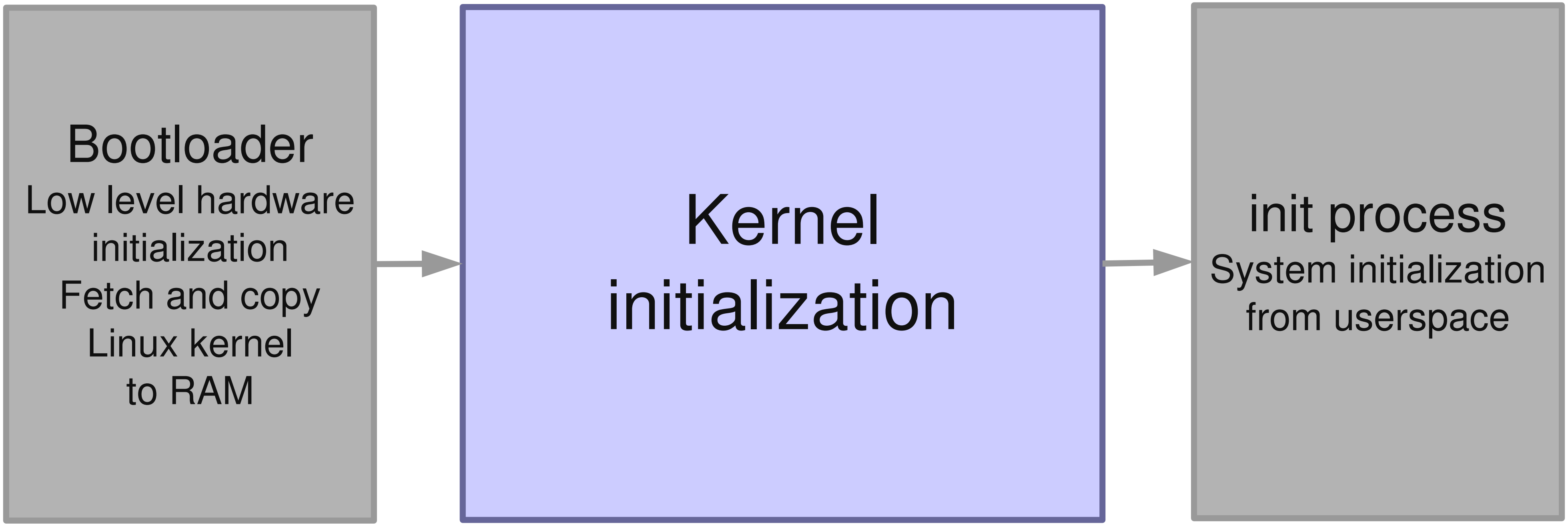
Kernel initialization

Michael Opdenacker
Free Electrons



Latest update: Dec 20, 2010,
Document sources, updates and translations:
http://free-electrons.com/docs/kernel-init
Corrections, suggestions, contributions and translations are welcome!

From bootloader to userspace



Bootloader

Low level hardware initialization

Fetch and copy Linux kernel to RAM

Kernel initialization

init process

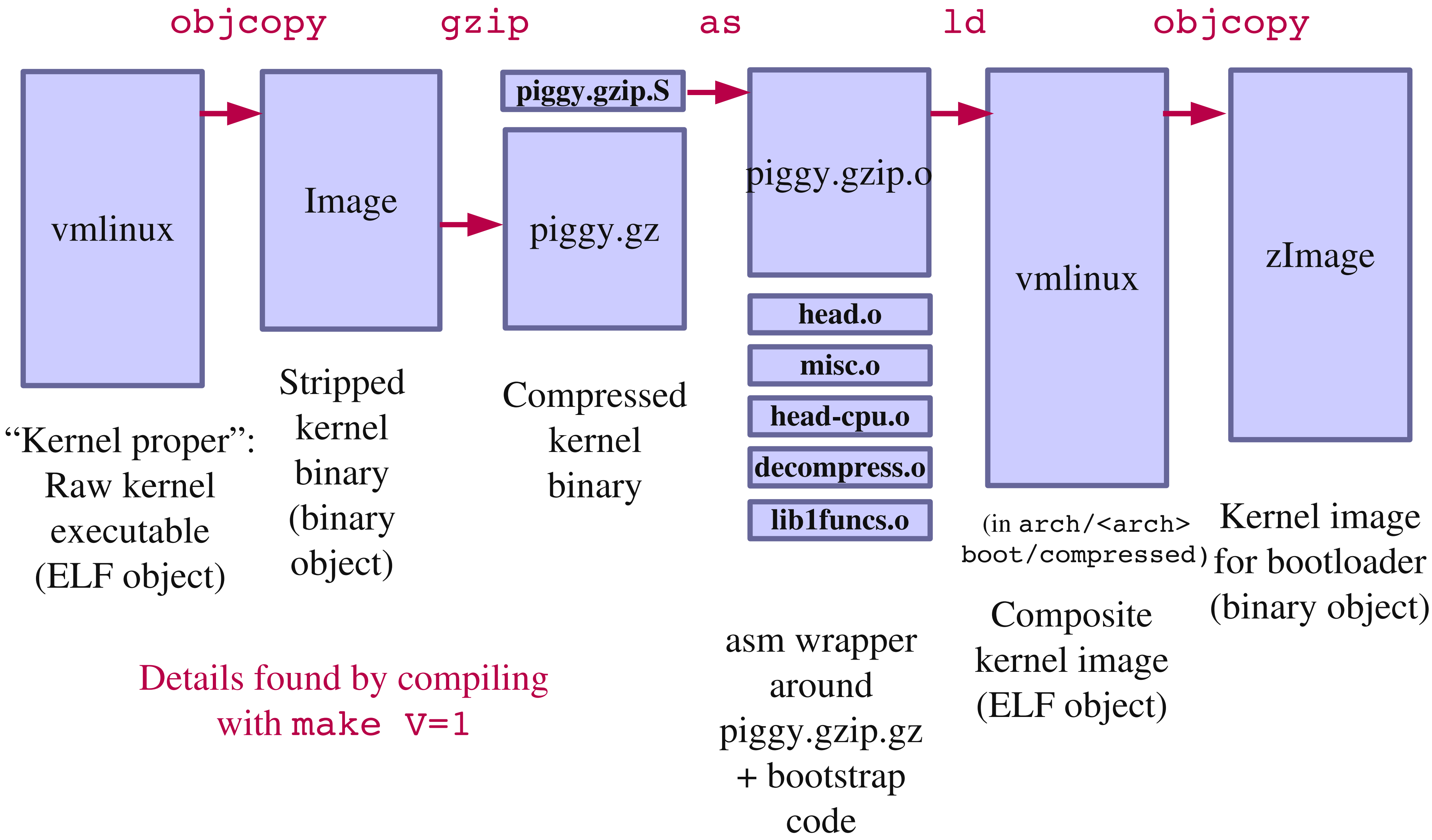
System initialization from userspace

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)



`objcopy` `gzip` `as` `ld` `objcopy`

vmlinux

"Kernel proper": Raw kernel executable (ELF object)

Image

Stripped kernel binary (binary object)

piggy.gzip.S

piggy.gz

Compressed kernel binary

piggy.gzip.o

head.o

misc.o

head-cpu.o

decompress.o

lib1funcs.o

asm wrapper around piggy.gzip.gz + bootstrap code

vmlinux

(in `arch/<arch> boot/compressed`)

Composite kernel image (ELF object)

zImage

Kernel image for bootloader (binary object)

Details found by compiling with `make V=1`

Bootstrap code

- `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
- `lib1funcs.o`:
  Optimized ARM division routines (ARM only)

Bootstrap code tasks

Main work done by `head.o`:

- 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 interesting 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

Starting a new kernel thread which will later become 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();
}
```

Source: Linux 2.6.36

kernel_init

kernel_init does two main things:

- Call do_basic_setup
  Now that 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

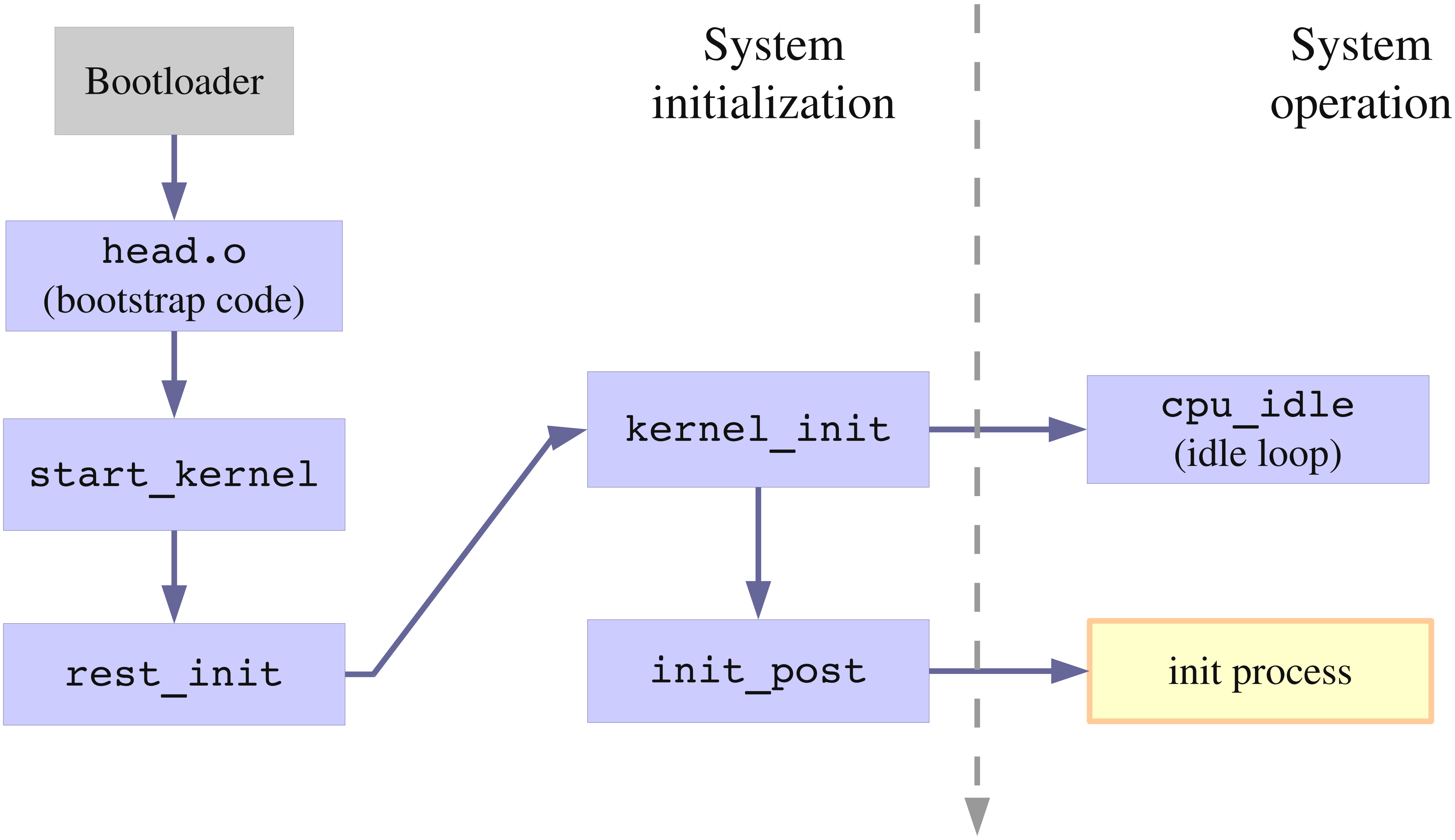
- First tries to open a console
- Then tries to run the init process, effectively turning the current kernel thread into the userspace init process.

init_post code

```
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 for guidance.");
}
```

Source:
`init/main.c`
in Linux 2.6.36

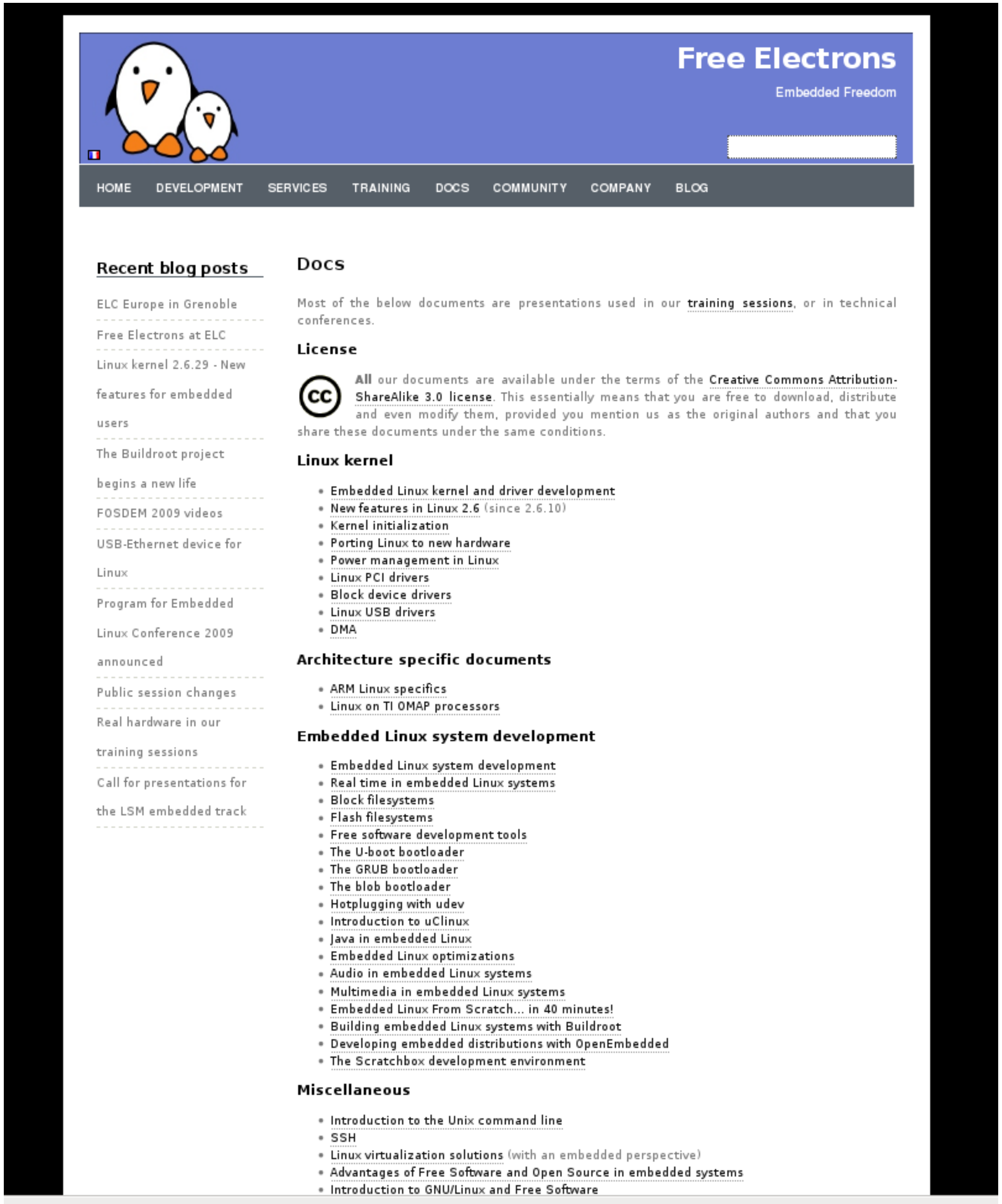
Kernel initialization graph



Kernel initialization - What to remember

- 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.

Related documents



Free Electrons

Embedded Freedom

Recent blog posts

- ELC Europe in Grenoble
- Free Electrons at ELC
- Linux kernel 2.6.29 - New features for embedded users
- The Buildroot project begins a new life
- FOSDEM 2009 videos
- USB-Ethernet device for Linux
- Program for Embedded Linux Conference 2009 announced
- Public session changes
- Real hardware in our training sessions
- Call for presentations for the LSM embedded track

Docs

Most of the below documents are presentations used in our training sessions, or in technical conferences.

License

All our documents are available under the terms of the Creative Commons Attribution-ShareAlike 3.0 license. This essentially means that you are free to download, distribute and even modify them, provided you mention us as the original authors and that you share these documents under the same conditions.

Linux kernel

- Embedded Linux kernel and driver development
- New features in Linux 2.6 (since 2.6.10)
- Kernel initialization
- Porting Linux to new hardware
- Power management in Linux
- Linux PCI drivers
- Block device drivers
- Linux USB drivers
- DMA

Architecture specific documents

- ARM Linux specifics
- Linux on TI OMAP processors

Embedded Linux system development

- Embedded Linux system development
- Real time in embedded Linux systems
- Block filesystems
- Flash filesystems
- Free software development tools
- The U-boot bootloader
- The GRUB bootloader
- The blob bootloader
- Hotplugging with udev
- Introduction to uClinux
- Java in embedded Linux
- Embedded Linux optimizations
- Audio in embedded Linux systems
- Multimedia in embedded Linux systems
- Embedded Linux From Scratch... in 40 minutes!
- Building embedded Linux systems with Buildroot
- Developing embedded distributions with OpenEmbedded
- The Scratchbox development environment

Miscellaneous

- Introduction to the Unix command line
- SSH
- Linux virtualization solutions (with an embedded perspective)
- Advantages of Free Software and Open Source in embedded systems
- Introduction to GNU/Linux and Free Software

All our technical presentations on http://free-electrons.com/docs

- Linux kernel
- Device drivers
- Architecture specifics
- Embedded Linux system development

How to help

You can help us to improve and maintain this document...

- By sending corrections, suggestions, contributions and translations
- By asking your organization to order development, consulting and training services performed by the authors of these documents (see http://free-electrons.com/).
- By sharing this document with your friends, colleagues and with the local Free Software community.
- By adding links on your website to our on-line materials, to increase their visibility in search engine results.

Free Electrons

Our services

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Application and interface development

Consulting and technical support

Help in decision making
System architecture
System design and performance review
Development tool and application support
Investigating issues and fixing tool bugs