

Lecture 7.2

Booting Linux from power-on to your own PID 1

Md Mahbub E Noor

Assistant Professor, Department of CSE

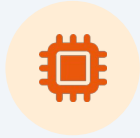
University of Barishal

```
● ● ● QEMU: firmware → GRUB → kernel → our  
init  
SeaBIOS (version 1.16.3-debian-1.16.3-  
2)  
Booting from DVD/CD...  
GNU GRUB version 2.12  
GRUB: loading the kernel...  
GRUB: loading the initramfs...  
*** myinit: I am PID 1, started by  
the kernel as /init ***
```

What you will learn today



Name the stages from power-on to the first user program, and what each one loads.



Compare BIOS and UEFI booting, and say what a boot loader does.



Follow `start_kernel`, `rest_init` and `kernel_init` in the real source.



Write your own PID 1 in C, boot it in QEMU, and read a kernel panic.

Today's plan (60 min)

0-5 Recap of 7.1

5-15 Firmware

15-25 Boot loader

25-45 Kernel start-up

45-55 PID 1

55-60 Checkpoint

Lab 7.2: myinit.c, your own init, booted in QEMU.

Quick recap of Lecture 7.1

Q1

Who is PID 1, and who is PID 2?

PID 1: the first user program (init). PID 2: kthreadd, the parent of all kernel threads. Today: who creates them, and in which order.

Q2

What does /proc show before it is mounted?

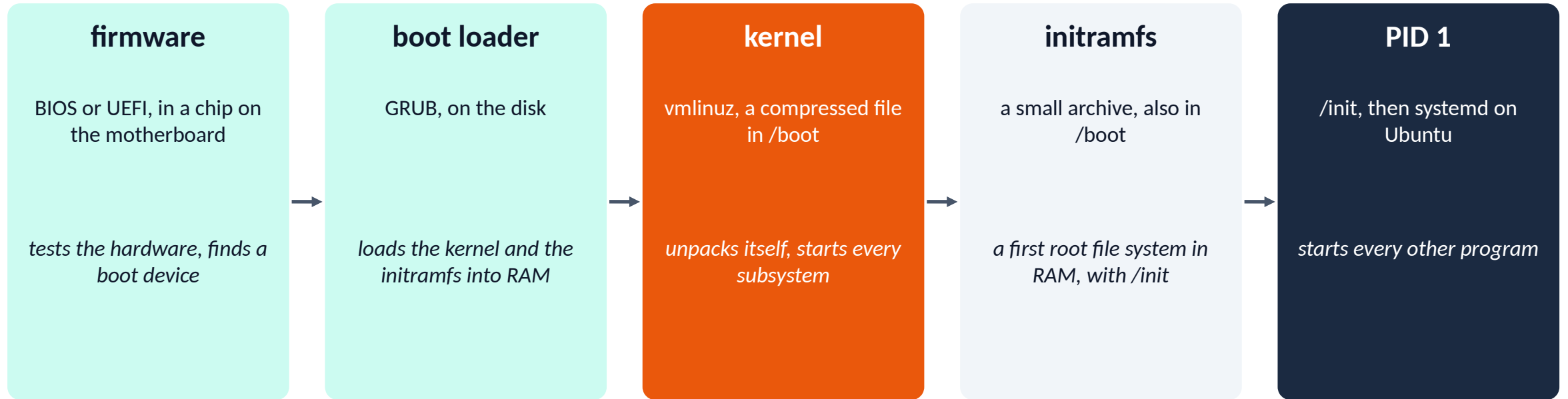
Nothing: an empty folder. Today your own init must mount it.

Q3

What did the bootstrap loader of Lecture 5.1 do?

A tiny absolute loader at a fixed address that loads a bigger program. A PC does the same, in several stages.

The boot chain: each stage loads the next



SAME AS C

Beck's bootstrap, in stages

Lecture 5.1: a tiny loader at a fixed address loads a bigger program. A PC does this three times: firmware → GRUB → kernel.

DIFFERENT FROM C

But the hardware is bigger

Before anything can be loaded, memory, disks and screens must be found and set up. That is most of the work, and most of the time.

Book: Bovet & Cesati, Appendix A (system startup); Beck §3.4.3 (bootstrap loaders)

PART 1

Power-on: the firmware

BIOS, UEFI, and the 512-byte boot sector you wrote in Lecture 5.1.

BIOS: load sector 0 to 7C00 and jump

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+02FCB050+02F0B050 CA00

Booting from Hard Disk...
Hello from the boot sector_
```

Real: boot.bin from Lecture 5.1 as a QEMU hard disk. SeaBIOS (the BIOS in QEMU) loads it and it prints its message.

- 1 Power on: the CPU starts in 16-bit **real mode**, running firmware code from the chip.
- 2 **POST** (power-on self test): find RAM, disks, keyboard, screen.
- 3 Read **sector 0** (512 bytes) of the boot disk to address **7C00**.
- 4 Last two bytes **55 AA**? Jump to 7C00. If not, try the next device.

512 bytes is too small for a kernel. So the boot sector loads more of the boot loader from the disk: GRUB's first stage does exactly that.

Book: Bovet & Cesati, Appendix A (the BIOS, the boot loader)

UEFI: the modern firmware, and how to tell which one you have

	BIOS (legacy)	UEFI
Since	1981 (IBM PC)	about 2010 on PCs
CPU mode at hand-over	16-bit real mode	32- or 64-bit mode
Finds the boot loader in	sector 0 of a disk (512 bytes)	a file on a FAT partition: \EFI\...*.efi
Boot loader format	raw machine code	a PE program, the Windows .exe format
Extra	–	Secure Boot: runs only signed boot loaders

```
● ● ● same kernel booted with UEFI
      firmware (OVMF) in QEMU (real)

$ ls /sys/firmware
acpi dmi efi memmap
$ ls /sys/firmware/efi
config_table efivars ...
$ dmesg | grep efi | head -1
efi: EFI v2.7 by Ubuntu
      distribution of EDK II
```

Try it on the lab PC: `ls /sys/firmware/efi`. If the folder exists, the PC booted with UEFI; if not, with BIOS (or UEFI in legacy mode). Our BIOS boot in QEMU had no `efi` folder: `acpi dmi memmap` only. The kernel shows the firmware's world as files in `/sys` (Lecture 7.1).

PART 2

The boot loader: GRUB

Load two files into RAM, pass a command line, jump.

GRUB loads the kernel and the initramfs (real boot)

```
iso/boot/grub/grub.cfg

serial --unit=0 --speed=115200
terminal_output serial
set timeout=3
menuentry "CSE 4105: Ubuntu kernel + our init" {
    linux /boot/vmlinuz console=ttyS0 quiet panic=-1
    initrd /boot/initramfs.cpio.gz
}
```

the boot, on the serial console (real)

```
Booting from DVD/CD...
GNU GRUB version 2.12
The highlighted entry will be executed
  automatically in 0s.
GRUB: loading the kernel...
GRUB: loading the initramfs...
... our init prints /proc/cmdline:
BOOT_IMAGE=/boot/vmlinuz console=ttyS0
quiet panic=-1
```

linux

loads the kernel file into RAM and remembers the **command line** after it.

initrd

loads the initramfs archive next to it. GRUB reads files, so it understands file systems (here ISO 9660; on a PC ext4).

boot

jumps into the kernel. GRUB adds `BOOT_IMAGE=` to say which file it was: the kernel prints it in `/proc/cmdline`.

vmlinuz: one file, three ways to be started

```
● ● ● Ubuntu's kernel file (real)

$ file vmlinuz
Linux kernel x86 boot executable bzImage,
  version 6.8.0-146-generic ... R0-rootFS ...
$ ls -l vmlinuz      → 15 067 528 bytes
$ od -A x -t x1z -N 2 vmlinuz
000000 4d 5a                >MZ<
$ od -A x -t x1z -j 0x1fe -N 6 vmlinuz
0001fe 55 aa 00 00 48 64 72 53 >U...HdrS<
```

MZ at byte 0

the start of a PE (Windows .exe) header: UEFI can run the kernel directly as an EFI program.

55 AA at byte 510

a boot-sector signature, from the days when the kernel could boot itself from a floppy.

HdrS at byte 0x202

the Linux boot protocol header: a C struct (setup_header) where GRUB writes the command line and initramfs address.

SAME AS C

Like the ELF header

A header the loader reads to know what to load and where (Lecture 6). GRUB is the loader here.

DIFFERENT FROM C

But compressed

vmlinuz is the kernel compressed, plus a small program that unpacks it in RAM first: "z" in the name.

PART 3

The kernel starts itself

`start_kernel()`, `rest_init()`, and the birth of PID 1 and PID 2.

start_kernel(): every subsystem, in order

init/main.c, Linux 6.8 (real lines, shortened)

```
874 void start_kernel(void)
875 {
895     pr_notice("%s", linux_banner);
897     setup_arch(&command_line);
905     pr_notice("Kernel command line: ...
931     trap_init();
932     mm_core_init();
944     sched_init();
976     init_IRQ();
984     time_init();
1008     console_init();
1056     vfs_caches_init();
1072     arch_call_rest_init();
    } /* never returns */
```

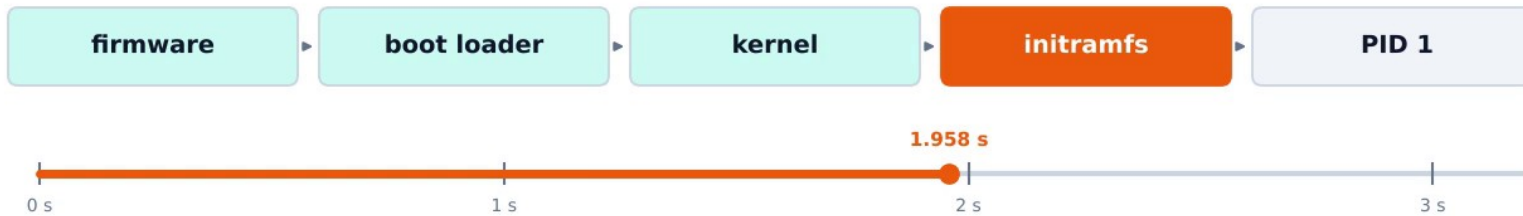
Call	Starts	Lecture
setup_arch	CPU, firmware memory map (e820)	7.2
trap_init	exception handlers (divide by 0, page fault)	13
mm_core_init	the memory allocator	11
sched_init	the process scheduler	8
init_IRQ, time_init	interrupts and the timer tick	13
vfs_caches_init	the virtual file system	12

C link: start_kernel is the kernel's main(): a long list of function calls, one per subsystem. Part B of the course is basically this list.

Book: Love, LKD Ch. 1-2; Bovet & Cesati, Appendix A

Watch one real boot, message by message

From power-on to PID 1 (a real boot, Ubuntu kernel in QEMU)



kernel clock (the [...] number in each message); the firmware runs before the clock starts

serial console (real output)

```
[ 0.000000] Linux version 6.8.0-146-generic (build@lcy02-amd64-111) (x86_64-linu
[ 0.000000] Command line: console=ttyS0 panic=-1
[ 0.000000] BIOS-e820: [mem 0x0000000000100000-0x0000000001ffdf0000] usable
[ 0.099079] Memory: 459696K/523768K available (22528K kernel code, 4442K rwd
[ 1.154871] smpboot: CPU0: AMD QEMU Virtual CPU version 2.5+ (family: 0xf, model:
[ 1.212470] devtmpfs: initialized
[ 1.534838] ACPI: Interpreter enabled
[ 1.958338] Trying to unpack rootfs image as initramfs...
```

where in the code

```
do_initcalls()
    populate_rootfs()
```

The initramfs, a cpio archive loaded by the boot loader, is unpacked into a RAM file system. It holds /init.

Watch for

1. The firmware has **no timestamp**: the kernel clock starts at 0.
2. Each message is a **printk** from one function.
3. **Memory** is known after 0.1 s.
4. The initramfs is unpacked at about 2 s.
5. Run /init ends the kernel's own work.

Click the video to play or pause. About 1½ minutes, no sound.

rest_init(): why init is PID 1 and kthreadd is PID 2

init/main.c, Linux 6.8 (real, shortened)

```
683 void rest_init(void)
684 {
689     /* We need to spawn init first so that it
        obtains pid 1, however the init task will
        end up wanting to create kthreads ... */
695     pid = user_mode_thread(kernel_init, NULL, CLONE_FS);
        ...
708     pid = kernel_thread(kthreadd, NULL, NULL, ...);
        ...
730     cpu_startup_entry(CPUHP_ONLINE);    /* idle */
731 }
```

PID 0: the boot thread itself

After creating the other two, it becomes the **idle task**: it runs when nothing else can. That is the PPID 0 you saw in 7.1.

PID 1: kernel_init

Created first, on purpose (the comment says so). It finishes the kernel's set-up, then becomes your /init.

PID 2: kthreadd

Created second. Every later kernel thread is made by it: PPID 2.

C link: `user_mode_thread(kernel_init, ...)` is like `pthread_create(&t, NULL, func, NULL)`: start a new thread of control that runs a C function. The PID is given in creation order: 1, then 2.

kernel_init(): looking for a program to become PID 1

```
init/main.c, Linux 6.8 (real, shortened)

159 static char *ramdisk_execute_command = "/init";
... in kernel_init():
1466     if (ramdisk_execute_command) {
1467         ret = run_init_process(ramdisk_execute_command);
1468         if (!ret) return 0;
... init= from the command line, then:
1497     if (!try_to_run_init_process("/sbin/init") ||
1498         !try_to_run_init_process("/etc/init") ||
1499         !try_to_run_init_process("/bin/init") ||
1500         !try_to_run_init_process("/bin/sh"))
1501         return 0;
1503     panic("No working init found. ...");
```

1 /init in the initramfs (or rdinit=...)

2 init=... from the command line

3 /sbin/init, /etc/init, /bin/init, /bin/sh

4 nothing found: **panic**

run_init_process() ends in `kernel_execve("/init", ...)`: the same `exec` that a shell uses to start a program (Lecture 8). PID 1 stops being kernel code and becomes your C program.

The initramfs: a first root file system in RAM

```
● ● ● mkinitramfs.sh (lab)

mkdir -p rootfs/proc rootfs/sys rootfs/dev rootfs/bin
cp init rootfs/init
cp kinfo_static rootfs/bin/kinfo
cd rootfs && find . | cpio -o -H newc \
    | gzip > ../initramfs.cpio.gz
```

```
● ● ● zcat initramfs.cpio.gz | cpio -tv (real)

drwxr-xr-x  proc
drwxr-xr-x  dev
drwxr-xr-x  sys
-rwxr-xr-x  init      826008 bytes
-rwxr-xr-x  bin/kinfo 968048 bytes
```

Why it exists

The real root disk may need a driver that is a module (Lecture 7.1), on the disk itself. The initramfs carries those modules and a small /init that loads them, mounts the real root, then hands over.

cpio

An old, simple archive format (like tar). The kernel has its own unpacker: "Trying to unpack rootfs image as initramfs" in the boot log.

Why -static

The initramfs has no libc.so and no ld.so (Lecture 5.2). A static program needs nothing: 826 KB for a small init.

PART 4

PID 1: the first user program

Yours, in the lab. And what happens when it goes wrong.

Our own init: 50 lines of ordinary C

● ● ● myinit.c (lab reference, shortened)

```
int main(int argc, char *argv[]) {
    printf("*** myinit: I am PID %d ...", getpid(), ...);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);
    show("kernel:", "/proc/sys/kernel/osrelease");
    show("command line:", "/proc/cmdline");
    pid_t pid = fork();
    if (pid == 0) {                /* child */
        execv("/bin/kinfo", args);
        _exit(127);
    }
    waitpid(pid, &status, 0);      /* parent */
    reboot(RB_POWER_OFF);         /* never return */
}
```

● ● ● ./boot.sh (lab, real)

```
*** myinit: I am PID 1, started by
    the kernel as /init ***
kernel:      6.8.0-146-generic
command line: console=ttyS0 quiet panic=-1
... kinfo's output (Lecture 7.1) ...
myinit: child 78 (kinfo) exited
           with status 0
myinit: all done, powering off.
[ 2.420903] reboot: Power down
```

C link: fork + exec + wait is how **every** program after PID 1 is born, including the ones your shell starts. Lecture 8 opens it up.

When PID 1 goes wrong: three real kernel panics

1. init returns

```
● ● ● serial console (real)
starter: main() ends with return 0;
myinit: all done.
Kernel panic - not syncing:
Attempted to kill init!
exitcode=0x00000000
```

PID 1 must never end: every orphaned process is given to it. kernel/exit.c line 840 panics.

2. no /init, no root disk

```
● ● ● serial console (real)
initramfs without /init
VFS: Cannot open root device ""
Please append a correct "root="
Kernel panic - not syncing:
VFS: Unable to mount root fs
on unknown-block(0,0)
```

No /init, so the kernel tried a root disk; none was given. The fix: an init, or root=/dev/...

3. the wrong PID 1

```
● ● ● serial console (real)
boot.sh with EXTRA=rdinit=/bin/kinfo
Processes: 0 user processes,
0 kernel threads
Kernel panic - not syncing:
Attempted to kill init!
PID: 1 Comm: kinfo
```

kinfo became PID 1: /proc was never mounted, so it saw nothing; then it ended.

The kernel is relocated too: KASLR

```
● ● ● the last line of four panics (real)
Kernel Offset: 0x9e00000 from 0xffffffff81000000
Kernel Offset: 0x25e00000 from 0xffffffff81000000
Kernel Offset: 0x27200000 from 0xffffffff81000000
booted with nokaslr on the command line:
Kernel Offset: disabled
```

0xffffffff81000000

where System.map says `_text` is (Lecture 7.1): the address the kernel was linked for.

+ a random offset

chosen at every boot, so `start_kernel` is somewhere else each time. An attacker cannot know where kernel code is.

SAME AS C

Same as ASLR and M records

Lecture 5.2: a PIE program gets a random base at every run. Lecture 5.1: a relocating loader adds the load address to address fields. The kernel's decompressor does both to the kernel itself.

DIFFERENT FROM C

But who relocates the relocater?

There is no `ld.so` before the kernel runs. The small decompressor program in `vmlinuz` picks the offset and applies the kernel's own relocation table, then jumps in.

On the lab PC: systemd is PID 1

● ● ● commands to run on the lab PC (Ubuntu)

```
$ ps -o pid,ppid,comm -p 1,2
# 1 0 systemd
# 2 0 kthreadd
$ ls -l /sbin/init
# /sbin/init -> /lib/systemd/systemd
$ systemd-analyze
# firmware + loader + kernel + userspace
$ systemctl list-units --type=service | head
```

Same job as our init

Mount /proc and /sys, start programs with fork + exec, wait for them, never exit.

Much more

Starts hundreds of services in parallel, restarts them when they crash, logs their output (journalctl), shuts down cleanly.

systemd-analyze

Prints how long each boot stage took on that PC: firmware, loader, kernel, user space. Compare with our 2.4 s QEMU boot.

	SIC bootstrap (5.1)	PC with Linux
Started by	hardware, at address 0	firmware chip, then GRUB
Loads	object code from device F1, to address 80	kernel + initramfs from disk, anywhere (KASLR)
Then jumps to	80	the kernel, then /init, then systemd

LAB 7.2

Boot your own init in QEMU

myinit.c: three TODOs, one deliberate kernel panic.

Lab 7.2 tasks, step by step

1

Get a kernel

```
sudo cp /boot/vmlinuz-$(uname -r) vmlinuz  
&& sudo chmod a+r vmlinuz: the lab PC's own kernel.
```

2

Boot the starter

```
gcc -Wall -static -o init  
myinit_starter.c, ./mkinitramfs.sh, ./boot.sh.  
Read the panic: why?
```

3

TODO 1: mount

Mount /proc and /sys. Now the kernel version and command line appear.

4

TODO 2: fork + exec

Build kinfo with -static, run it as a child of your init, print its exit status.

5

TODO 3: never exit

sync() and reboot(RB_POWER_OFF): QEMU powers off cleanly, no panic.

6

Experiments

```
EXTRA=rdinit=/bin/kinfo ./boot.sh;  
EXTRA=nokaslr; boot with -m 128. Note what changes.
```

Quit a stuck QEMU with Ctrl-A then X. Submit: myinit.c and the serial output of your final boot.

Checkpoint: test yourself

Q1

The BIOS loads how many bytes from the boot disk, and to where?

→ 512 bytes (sector 0) to address 7C00

Q2

How can you tell whether a Linux PC booted with UEFI?

→ /sys/firmware/efi exists

Q3

Which two files does GRUB load into RAM?

→ the kernel (vmlinuz) and the initramfs

Q4

Why does init get PID 1 and kthreadd PID 2?

→ `rest_init` creates `init` first, then `kthreadd`

Q5

Your `/init` returns 0. What happens?

→ kernel panic: Attempted to kill init!

Key ideas from today



Boot = a chain of loaders: firmware → GRUB → kernel → initramfs → PID 1.



BIOS loads a 512-byte sector to 7C00; UEFI runs a .efi program from a FAT partition.



GRUB loads vmlinuz and the initramfs and passes the command line (BOOT_IMAGE=...).



start_kernel starts every subsystem; rest_init makes PID 1 (init) and PID 2 (kthreadd).



PID 1 must never exit; fork + exec + wait is how it starts everything else.



The kernel is relocated at every boot (KASLR), like a PIE program.

Homework 7.2 (due before Lecture 8.1)

1. Boot timeline

From boot1.log (lab files) find the times of: the banner, "Memory:", the initramfs unpack and "Run /init". Which stage takes longest?

2. The fallback list

Your initramfs has no /init but has /bin/sh (busybox). What does the kernel run, and why? Use the kernel_init code on slide 15.

3. Two inits

Change myinit to start kinfo twice, one after the other. Predict the two child PIDs before you boot, then check.

4. Read the header

Use od on your vmlinuz: find MZ, 55 AA and HdrS. Read the 2 bytes after HdrS: which boot protocol version is it?

5. Compare

In 8–10 sentences compare our boot with Beck's bootstrap loader (5.1): who starts it, what it loads, where, and how it ends.

Next lecture and reading

NEXT: LECTURE 8.1

Processes

What a process is to the kernel: `task_struct`, process states, `fork`, `exec`, `exit` and `wait` — the calls your `init` used today, opened up.

Main text

R. Love, **Linux Kernel Development**, 3rd ed., Ch. 3 (process management). Bovet & Cesati, Appendix A (system startup) for today.

Source

Linux 6.8: `init/main.c` (`start_kernel`, `rest_init`, `kernel_init`), `kernel/exit.c`, `init/do_mounts.c`.
Linux v6.8 source online: elixir.bootlin.com/linux/v6.8.

Tools

`qemu-system-x86`, `cpio`, `gzip`, `gcc -static`: all offline once installed.

Lab files

`lab7_files.zip`: `myinit_starter.c`, `kinfo_starter.c`, `mkinitramfs.sh`, `boot.sh`, `logs`.