

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 6

The Linux Kernel and Booting

what a kernel is, and how it gets started

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

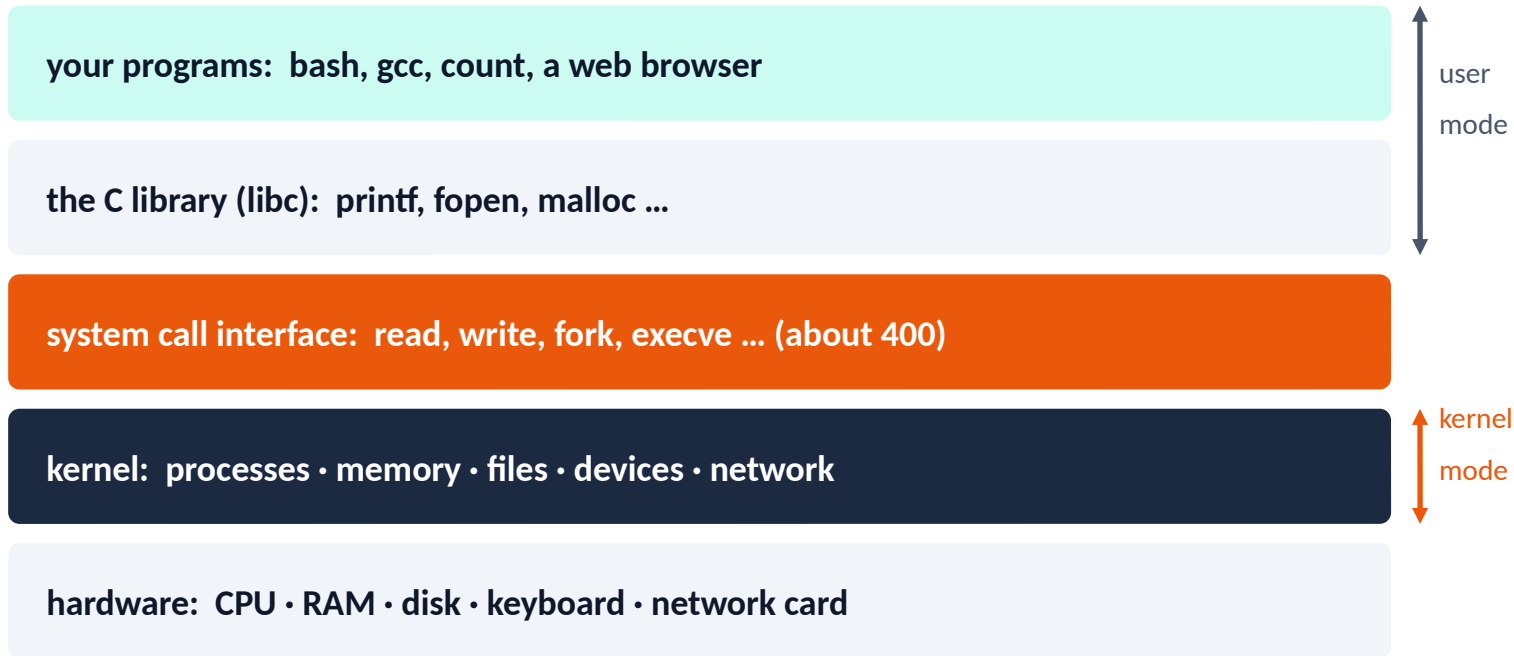
CLASS

6 / 10

19 slides, taken from the full course:

Lecture 7.1, Lecture 7.2

Where the kernel sits



The rule

Only the kernel touches hardware. A program that wants to read a file, print, or start another program must **ask the kernel** with a system call.

libc wraps those calls in friendly C functions.

C link: your `printf("hi")` (libc) ends in `write(1, "hi", 2)` (a system call), which the kernel turns into work for the terminal driver. You saw it with `strace` in Lecture 1; Lecture 9 opens the system call itself.

Book: Love, LKD Ch. 1 (Fig. 1.1); Bovet & Cesati Ch. 1

Five jobs, each with a real number from our booted kernel



Processes

Creates, schedules and ends programs. Even before init did anything: 57 kernel threads.

Lectures 8, 10



Memory

Hands out RAM and protects each process's memory. MemTotal: 469 252 kB of a 512 MB machine.

Lecture 11



Files

One interface, many file systems: proc, sysfs, ext4, vfat ... /proc/filesystems lists 30 types.

Lecture 12



Devices

Drivers talk to hardware. /proc/interrupts: timer 522, serial port ttyS0 164 interrupts so far.

Lectures 13–15



Network

Sockets and protocols. Boot log: "NET: Registered PF_INET protocol family".

beyond this course

Monolithic kernel or microkernel?

Monolithic

All services (scheduling, memory, file systems, drivers, network) run in kernel mode, in one address space, and call each other directly.

Trade-off: fast: a function call; one bug can crash everything

Examples: Linux, FreeBSD, classic Unix

Microkernel

Only the minimum runs in kernel mode: messages (IPC), scheduling, memory. File systems and drivers are user programs (servers).

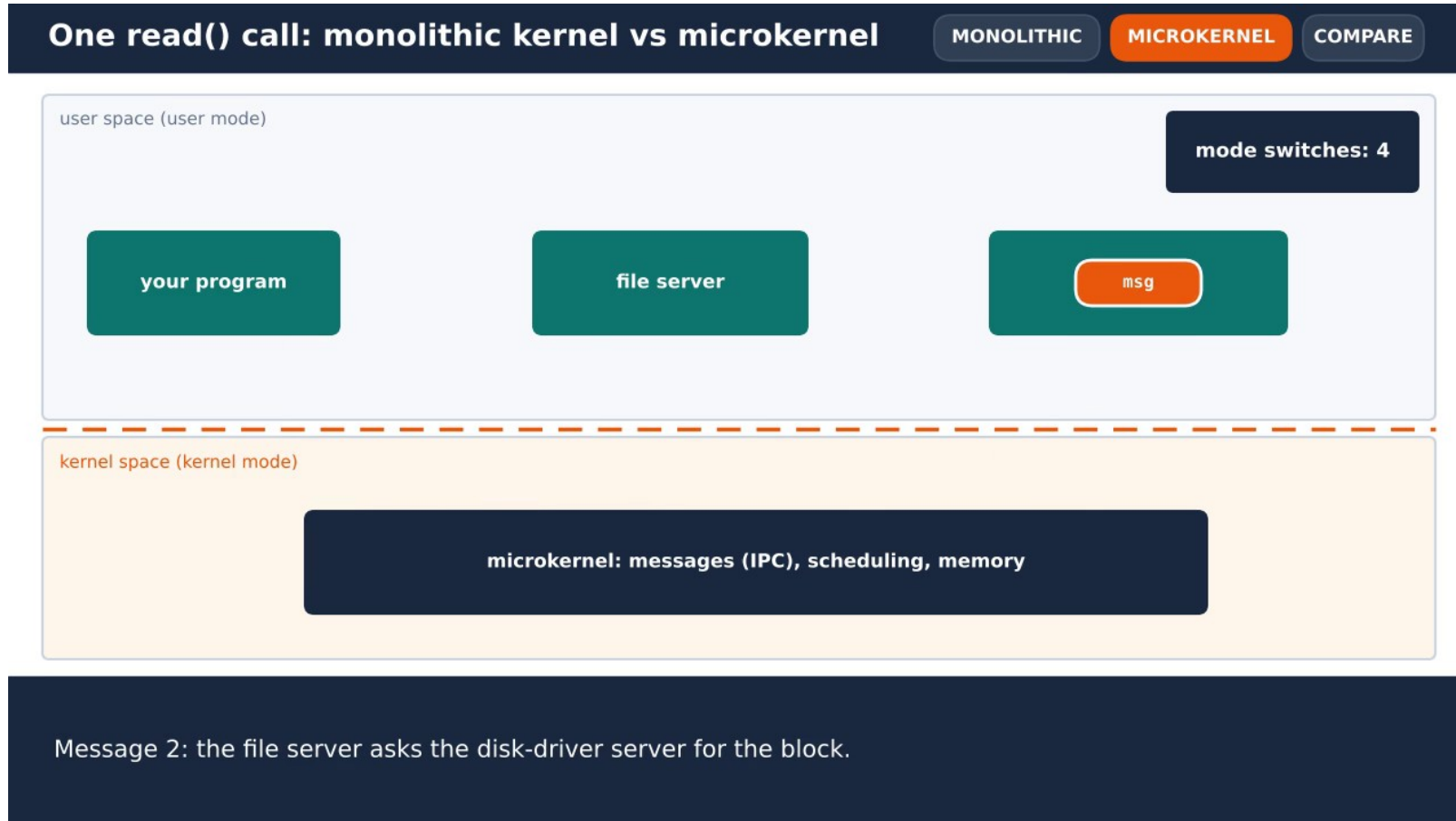
Trade-off: robust: a crashed driver is restarted; but messages cost time

Examples: MINIX 3, QNX, seL4

A famous argument (1992): Andrew Tanenbaum, author of MINIX, posted "LINUX is obsolete" because it was monolithic. Linus Torvalds answered that a monolithic kernel was simpler and faster to get working. Both kinds are still in use: Linux on servers and phones, QNX in cars, seL4 where a proof of correctness matters. Windows NT and macOS mix both ideas (hybrid).

Book: Love, LKD Ch. 1 (monolithic kernel versus microkernel designs)

Watch one read() in both designs



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

Watch for

1. The orange dashed line is the **user/kernel boundary**.
2. Monolithic: **one** system call, then plain function calls.
3. Microkernel: every message crosses the line **twice**.
4. Count the mode switches: **2 vs 8**.
5. What happens when a **driver** crashes?

Linux's answer: a monolithic kernel with loadable modules

```
● ● ● /boot/config-6.8.0-146-generic (Ubuntu, real)
```

```
CONFIG_MODULES=y
CONFIG_EXT4_FS=y      # built in
CONFIG_BTRFS_FS=m     # module
CONFIG_VFAT_FS=y
CONFIG_USB_STORAGE=m
CONFIG_E1000=m        # network card
```

3 074

options =y
built into vmlinux

6 275

options =m
built as modules

1 011

.ko files in
linux-modules alone

A module (.ko, an ELF relocatable file like a .o) is linked into the **running** kernel when needed: a linking loader inside the kernel (Lecture 5.1).

SAME AS C

Like a shared library

`lsmod` lists loaded modules like `ldd` lists libraries; `modprobe` loads one with its dependencies, like `ld.so`. Only drivers for hardware you have get loaded.

DIFFERENT FROM C

But still kernel mode

Unlike a microkernel server, a module runs in kernel mode with full power. A buggy module can still crash the system. You will write one in Part C (Lecture 14).

/proc: the kernel shows its data as text files

QEMU guest, busybox shell (real)

```
$ cat /proc/cmdline
console=ttyS0 quiet panic=-1
$ head -3 /proc/meminfo
MemTotal:  469252 kB
MemFree:   434512 kB
MemAvailable: 429228 kB
$ head -3 /proc/1/status
Name:  init
Umask: 0022
State: S (sleeping)
```

Not on any disk

Each read runs kernel code that writes the answer at that moment. `ls -l` shows size 0.

One folder per process

`/proc/1`, `/proc/2` ... with status, cmdline, maps, fd ... (our init had 57 entries).

Before mount: empty

`/proc` exists only after `mount -t proc proc /proc`. Your init in Lab 7.2 must do it.

SAME AS C

Same C as always

`fopen("/proc/meminfo")`, `fgets`, `sscanf`. No special API: that is the lab.

DIFFERENT FROM C

Not a real file

The data is made up on each read; two reads can give different answers.

Processes and kernel threads right after boot

● ● ● kinfo (Lab 7.1) inside the guest (real)

Kernel: Linux version 6.8.0-146-generic ...

Up for: 3.1 seconds

CPUs: 1 x QEMU Virtual CPU version 2.5+

Processes: 2 user processes, 57 kernel threads
(56 of them children of PID 2)

PID	PPID	S	kind	name
1	0	S	user	init
2	0	S	kernel	kthreadd
3	2	S	kernel	pool_workqueue_release
4	2	I	kernel	kworker/R-rcu_g

PID 1: the first user program

Parent 0: the kernel itself started it. Every user process descends from PID 1.

PID 2: kthreadd

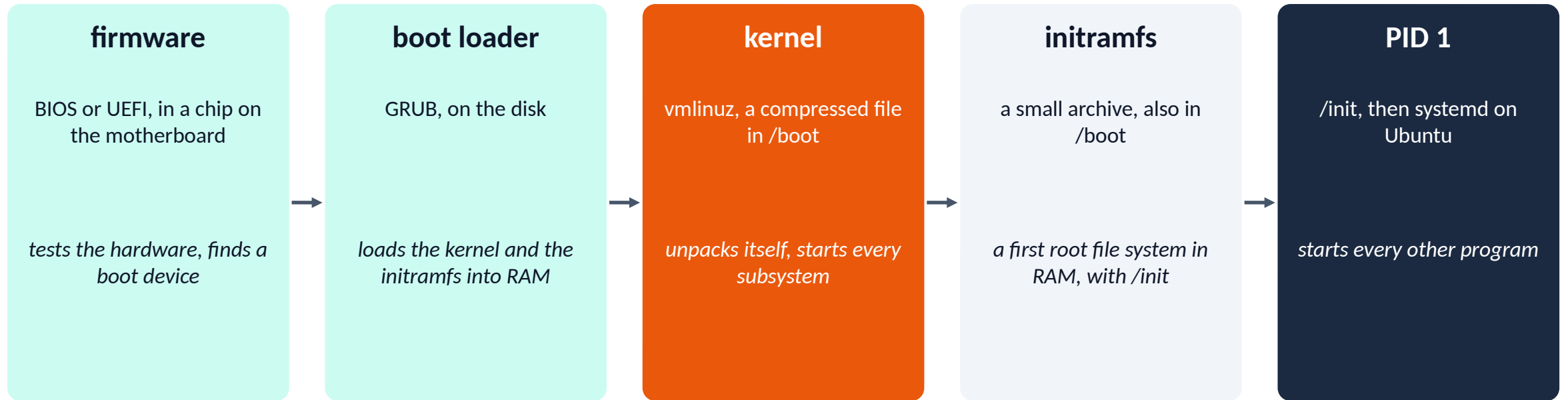
Also started by the kernel. It creates all other kernel threads: their parent is 2.

Kernel threads

Work the kernel does in the background: ksoftirqd (deferred interrupt work), kswapd (freeing memory), kworker (queued jobs). ps shows them in [brackets].

How does kinfo know? /proc/<pid>/stat has a flags field; the bit PF_KTHREAD (0x00200000) marks a kernel thread. The "2 user processes" are init and kinfo itself.

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)

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)

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**.

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 */
```

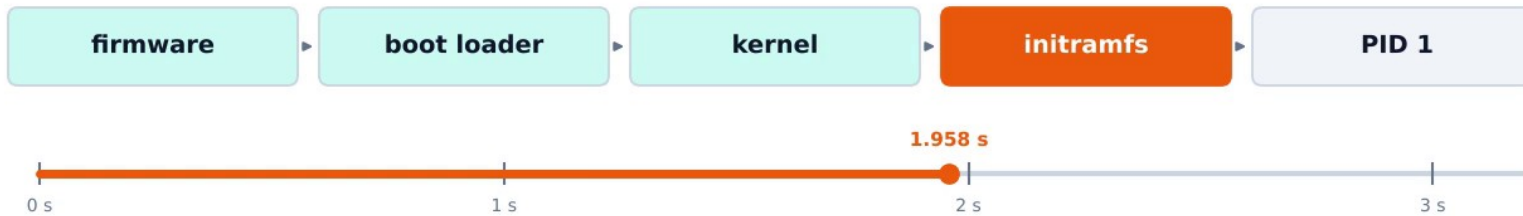
Call	Starts	Lecture
setup_arch	CPU, firmware memory map (e820)	7.2
trap_init	exception handlers (divide by 0, page fault)	13
mm_core_init	the memory allocator	11
sched_init	the process scheduler	8
init_IRQ, time_init	interrupts and the timer tick	13
vfs_caches_init	the virtual file system	12

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, m
[ 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.

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.

Key ideas from Class 6

- 1 The kernel manages processes, memory, files, devices and the network; it runs in kernel mode.
- 2 Linux is a monolithic kernel with loadable modules.
- 3 `/proc` and `/sys` show the kernel's data as text files you can read.
- 4 Boot chain: firmware (BIOS or UEFI), boot loader (GRUB), kernel, `initramfs`, then PID 1.
- 5 The kernel starts its subsystems in `start_kernel()`, then runs the first user program as PID 1.

Homework and practice

Homework 6 (bring it to the next class)

1. Put these in boot order: kernel, GRUB, PID 1, firmware, initramfs. One sentence on what each does.
2. Monolithic kernel or microkernel: what changes when a program calls read()?
3. Which file in /proc shows the kernel version, and which one shows the total memory?

Try it in the browser

/proc decoder: read a process's stat line

[labs/lab07.html](#)

On the course website: open the lab page and choose "Try it here".

Next class: Processes and Scheduling