

Lecture 7.1

The Linux Kernel an overview from the inside

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

● ● ● ps inside a freshly booted kernel (real)

```
PID  COMMAND
  1 {init} /bin/sh /init
  2 [kthreadd]
 16 [ksoftirqd/0]
 44 [kswapd0]
... 57 kernel threads
before any program ran
```

What you will learn today



Say what the kernel does, and where it sits between programs and hardware.



Compare monolithic kernels and microkernels; explain loadable modules.



Find your way in the Linux source tree, and say how kernel C differs from your C.



Ask a running kernel about itself through `/proc`, `/sys` and `dmesg`.

Today's plan (60 min)

0-5

Recap of Part A

5-15

What a kernel is

15-30

Kernel design

30-45

Inside the source

45-55

Talking to the kernel

55-60

Checkpoint

Lab 7.1: `kinfo.c`, a C program that reads `/proc`.

Quick recap: Part A in three questions

Q1

Who runs `./count` after `ld` has made the file?

The kernel (`execve`) maps its segments, and `ld.so` links the libraries. Part B opens that box.

Q2

What does `syscall` in `tiny` do?

Switches the CPU from user mode to kernel mode and asks the kernel for a service (`exit 42`).

Q3

Who calls `main`?

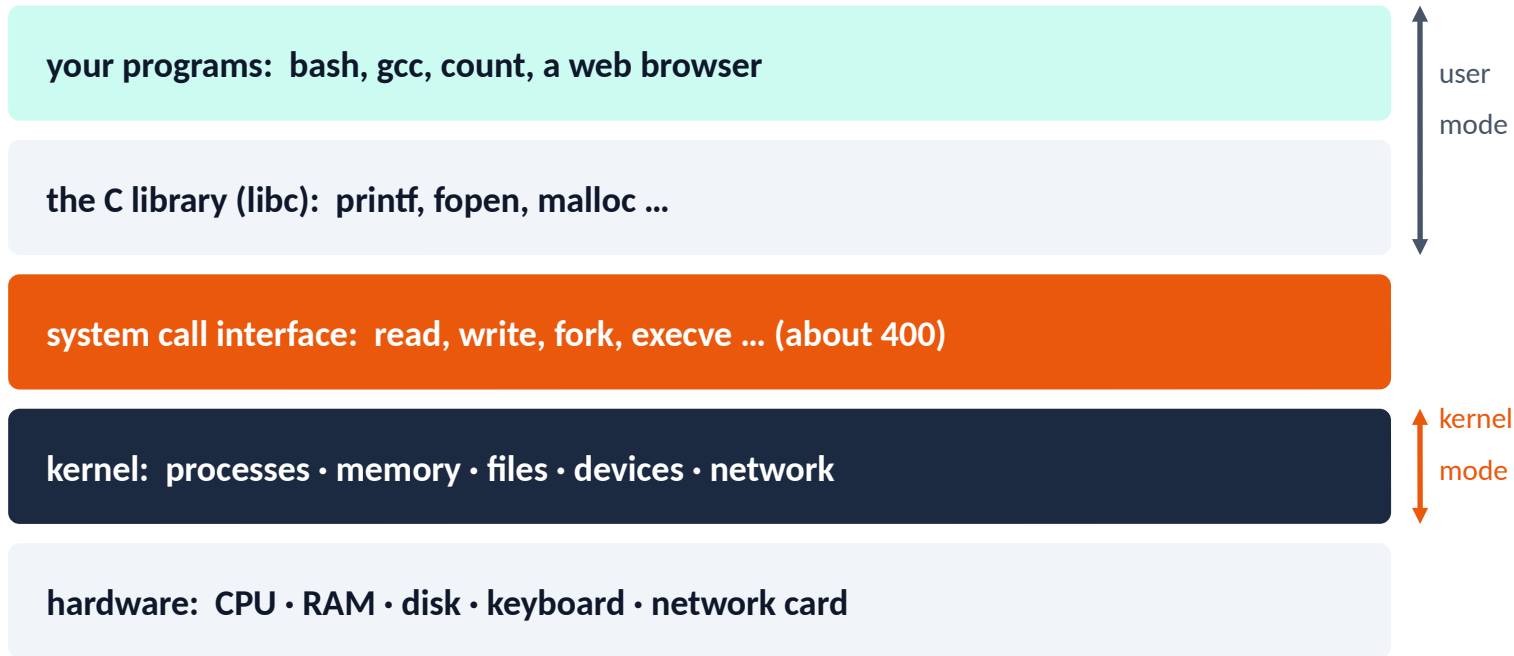
`__libc_start_main`, from `_start`. And who starts the very first program of all? Lecture 7.2.

PART 1

What a kernel is

The one program that is always running, in kernel mode, and owns the hardware.

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

The kernel lives at the top of every address space

System.map of Ubuntu's kernel (real)

```
ffffffff81000000 T _text
ffffffff811b84b0 T _printk
ffffffff83ab16f0 T start_kernel
ffffffff8225d370 t kernel_init
ffffffff84430000 B _end
```

where.c from Lecture 5.2 (user program, real)

```
main    0x55f030bf3199
printf  0x7fdf0be60100
local   0x7ffdf3617b8c
```

0xffff... top

kernel: code, data, every process's kernel stack. Same in every process, but only usable in kernel mode.

0x0000 7fff ... and below

user space: your program, its libraries, heap and stack. Different in every process.

C link: System.map has the same format as nm output (Lecture 5): address, type letter T/t/B, name. The kernel is a linked ELF program too.

PART 2

How much belongs inside the kernel?

Monolithic kernels, microkernels, and Linux's loadable modules.

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).

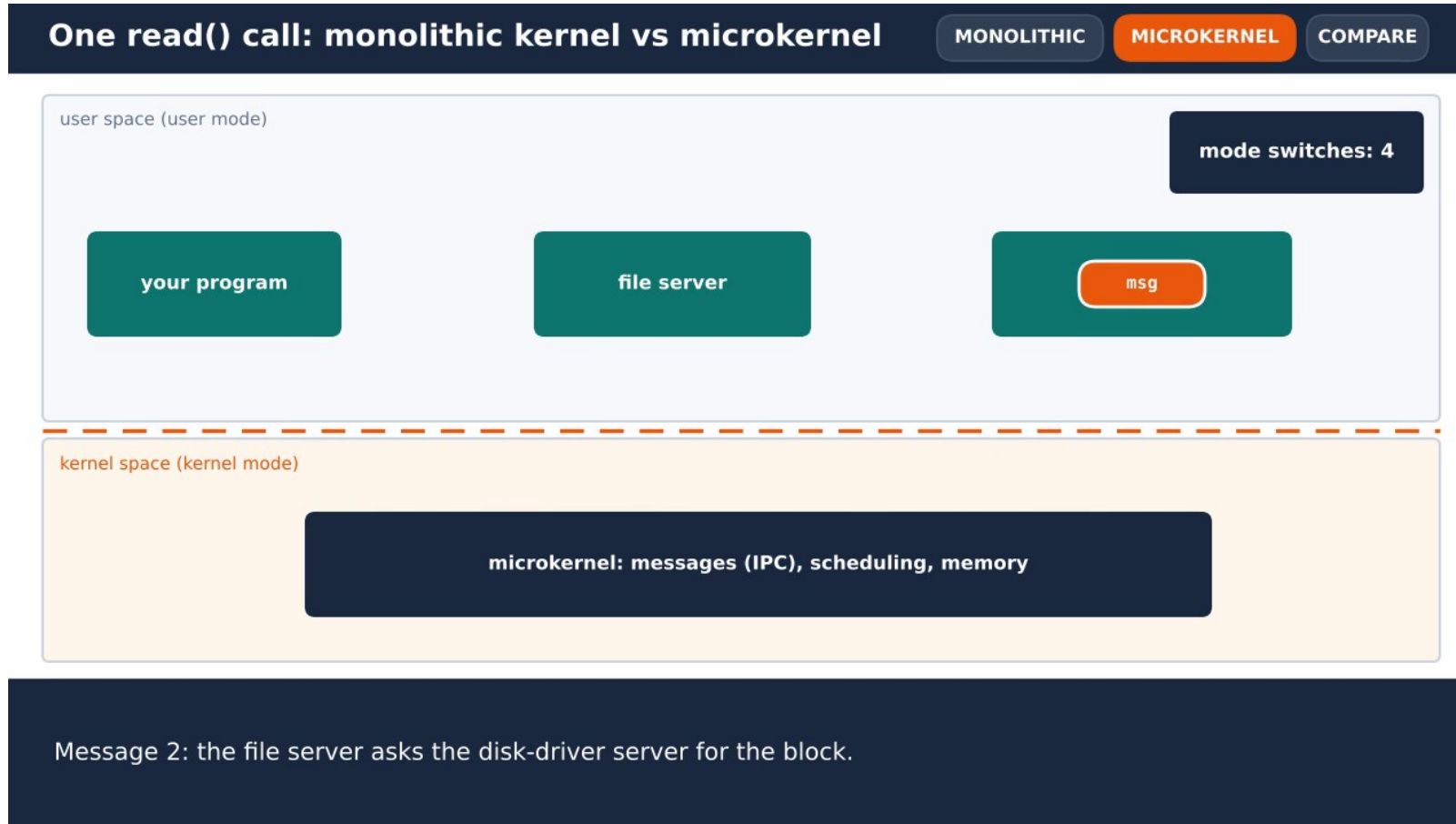
Trade-off: robust: a crashed driver is restarted; but messages cost time

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).

PART 3

Inside the Linux source

33 million lines of C, and where to find things.

The Linux 6.8 source tree (real counts)

Folder	Files	MB	What is inside
drivers/	33 771	977	device drivers: 70% of all C lines
arch/	16 985	108	CPU-specific code: x86, arm64, riscv ...
fs/	2 421	46	VFS and every file system (ext4, vfat, proc ...)
net/	1 974	34	network protocols
include/	6 072	41	header files shared by the whole kernel
kernel/	579	13	processes, scheduler, signals, printk, exit
mm/	184	5	memory management
init/	15	0.2	start_kernel(), the boot sequence (7.2)
ipc/	13	0.3	pipes' cousins: message queues, semaphores (10.1)

83 475 files

1.40 GB of source

33.3 M lines of .c and .h

The core (kernel/ + mm/ + init/) is about **0.66 M** lines: 2% of the total. Most of Linux is drivers for hardware you will never own.

Reading a kernel version number

```
● ● ● uname -r ; cat /proc/version (QEMU guest, real)
6.8.0-146-generic
Linux version 6.8.0-146-generic (buildd@lcy02-amd64-111)
(x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0,
GNU ld (GNU Binutils for Ubuntu) 2.42) ... (Ubuntu ... 6.8.12)
```

6	8	0	146	generic
major (1)	minor (1)	Ubuntu (1)	ABI / upload (2)	flavour (2)

Linux 6.8 was released on 10 March 2024 (Makefile: VERSION = 6, PATCHLEVEL = 8). Ubuntu builds its own kernel on top of the stable release 6.8.12 and numbers its own updates (146).

1991

Linus Torvalds, a student in Helsinki, announces a "hobby" kernel. Version 0.01 has about 10 000 lines.

now

A new mainline version about every 9–10 weeks, made by thousands of developers; the git history is public.

SAME AS C

Built with your compiler

/proc/version says it: gcc 13.3 and ld 2.42, the same versions as on the lab PC.

The kernel is written in C, but not your C

	Your C program	Kernel C
Library	libc: printf, strlen, malloc	no libc: its own lib/ (strlen at lib/string.c:399), printk, kmalloc
Stack	8 MB (ulimit -s: 8192)	16 KB per thread (THREAD_SIZE on x86-64)
A bad pointer	Segmentation fault: only you die	"Oops" or a panic: the whole system may die
Floating point	use double freely	avoided: the FPU state belongs to user programs
Running alone?	yes, as far as you can tell	no: interrupts and other CPUs run kernel code at the same time
Language	standard C	GNU C extensions (and some Rust since 6.1)

Why no libc? libc itself is built **on top of** system calls into the kernel. The kernel cannot call printf, because printf ends by asking the kernel to write.

Book: Love, LKD Ch. 2 (a beast of a different nature)

printk: the kernel's printf, from source to screen

● ● ● init/main.c, Linux 6.8 (real excerpt)

```
1345 static int run_init_process(const char *init_filename)
1346 {
    ...
1349     argv_init[0] = init_filename;
1350     pr_info("Run %s as init process\n", init_filename);
    ...
1357     return kernel_execve(init_filename, argv_init,
envp_init);
```

● ● ● the boot log of our QEMU guest (real)

```
[    3.059268] Run /init as init process
```

pr_info(...) = **printk(KERN_INFO ...)**

Same format string as printf: %s, %d, %x. The level (EMERG ... DEBUG) says how important the message is.

Where it goes

Into a ring buffer in kernel memory. dmesg reads it; the console shows it while booting. The [3.059268] is seconds since the kernel started.

Who can read it

On Ubuntu, dmesg needs sudo (kernel.dmesg_restrict = 1): kernel messages can leak addresses.

C link: the line `pr_info("Run %s as init process\n", init_filename);` is ordinary C. Find any boot message in the source with `grep -rn` and you know exactly where you are in the boot.

PART 4

Asking a running kernel about itself

`/proc` and `/sys`: files that are not stored anywhere.

/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.

/sys, dmesg and /proc/interrupts

```
ls /sys (guest, real)
```

```
block  class  devices  fs          kernel  power
bus    dev    firmware hypervisor module
```

```
head /proc/interrupts (guest, real)
```

```
          CPU0
0:   522  IO-APIC  2-edge  timer
1:     9  IO-APIC  1-edge  i8042
4:   164  IO-APIC  4-edge  ttyS0
8:     1  IO-APIC  8-edge  rtc0
```

```
dmesg | head -3 (guest, real)
```

```
[0.000000] Linux version 6.8.0-146-generic
          (buildd@lcy02-amd64-111) ...
[0.000000] Command line: console=ttyS0
          quiet panic=-1
[0.000000] KERNEL supported cpus:
$ dmesg | wc -l
411
```

Interface	Shows	Example
/proc	processes and kernel state	/proc/1/status, /proc/meminfo
/sys	devices, drivers, kernel objects	/sys/firmware, /sys/class/net
dmesg	kernel messages (printk)	boot log, driver errors

LAB 7.1

kinfo.c: ask the kernel with C

Read /proc with fopen, fgets, sscanf and readdir: four TODOs.

Lab 7.1 tasks, step by step

1

TODO 1: first_line

fopen, fgets, fclose, cut the '\n'. Kernel: and Command line: appear.

2

TODO 2: field_kb

Find "MemTotal:" in /proc/meminfo and read the number with sscanf.

3

TODO 3: read_stat

After the last ')' in /proc/<pid>/stat: state, ppid ... flags. One sscanf.

4

TODO 4: classify

flags & PF_KTHREAD → kernel thread. Compare your counts with ps -ef.

5

Explore the lab PC

uname -r; ls /boot; lsmod | wc -l; sudo dmesg | head; ulimit -s. Fill the table in the handout.

6

Think

Why is cat /proc/uptime different every time, though no file changed? Write 3 sentences.

Submit: kinfo.c, its output on the lab PC, and the filled table. Keep kinfo: Lab 7.2 runs it as a child of your own init.

Checkpoint: test yourself

- Q1** Why can't the kernel call printf? → **printf (libc) ends in a system call into the kernel**
- Q2** Name one advantage of a microkernel and one of a monolithic kernel. → **micro: a driver crash is contained; mono: faster (no messages)**
- Q3** CONFIG_BTRFS_FS=m. What does m mean? → **built as a loadable module (.ko), not into vmlinux**
- Q4** What is PID 2, and who are its children? → **kthreadd; all other kernel threads**
- Q5** cat /proc/meminfo reads which disk? → **none: the kernel makes the text at each read**

Key ideas from today



The kernel owns the hardware; programs ask it through about 400 system calls.



Linux is monolithic (fast, one address space) with loadable modules for flexibility.



Kernel C has no libc, a 16 KB stack, no memory protection, and printk instead of printf.



33 million lines, 70% drivers; the core (kernel/, mm/, init/) is about 2%.



/proc, /sys and dmesg show the running kernel as text you can read in C.



PID 1 is the first user program, PID 2 the parent of all kernel threads.

Homework 7.1 (due before Lecture 7.2)

1. Layers

Trace `printf("hi\n")` from your program to the screen: name every layer of slide 5 it passes through, and the mode (user or kernel) at each.

2. Design

A car maker must choose between Linux and a microkernel for the brake controller. Give two arguments for each and pick one.

3. Modules

On the lab PC: `modinfo ext4` and `modinfo btrfs`. What is different, and why must `ext4` be built in on most systems?

4. kinfo+

Extend `kinfo`: print the 5 processes using the most memory (`VmRSS` in `/proc/<pid>/status`). Kernel threads have no `VmRSS`: why?

5. Stack

Write a recursive C function with a 1 KB local array. How deep can it go in user space (8 MB)? How deep could it go on a 16 KB kernel stack?

Next lecture and reading

NEXT: LECTURE 7.2

Booting: from power-on to PID 1

Firmware (BIOS and UEFI), the boot loader, the kernel's own start-up code, the initramfs, and the first user program — which in the lab will be yours.

Main text

R. Love, **Linux Kernel Development**, 3rd ed., Ch. 1–2 (today). D. P. Bovet & M. Cesati, **Understanding the Linux Kernel**, 3rd ed., Ch. 1 and Appendix A (system startup) for 7.2.

Source

Linux 6.8: `init/main.c`, `kernel/init.c` (quoted on the slides; Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

On the lab PC

`/proc`, `/sys`, `uname`, `lsmod`, `modinfo`, `sudo dmesg`: all offline.

Lab files

`lab7_files.zip`: `kinfo_starter.c`, `myinit_starter.c`, scripts for Lab 7.2.