Chapter 11: Input/Output

Operating Systems

I/O hardware and buses

Device registers, port-mapped and memory-mapped I/O

Polling (programmed I/O)

Interrupts, vectors, MSI-X

DMA, scatter-gather, IOMMU

I/O software layers

Buffering, caching, spooling

Blocking and asynchronous I/O, io_uring

Clocks and timers

Disks and disk scheduling

SSD, NVMe, blk-mq

RAID and stable storage

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_11_Input_Output.html

Learning Outcomes

Hardware

Explain controllers, device registers, PCIe/USB/NVMe buses, and port-mapped versus memory-mapped I/O.

Transfer methods

Compare polling, interrupt-driven I/O and DMA; state the role of MSI-X and the IOMMU.

Software layers

Trace a `read()` through the interrupt handler, the driver, the device-independent layer and the user library.

Interfaces

Distinguish blocking, nonblocking and asynchronous I/O; tell `epoll` from `io_uring`.

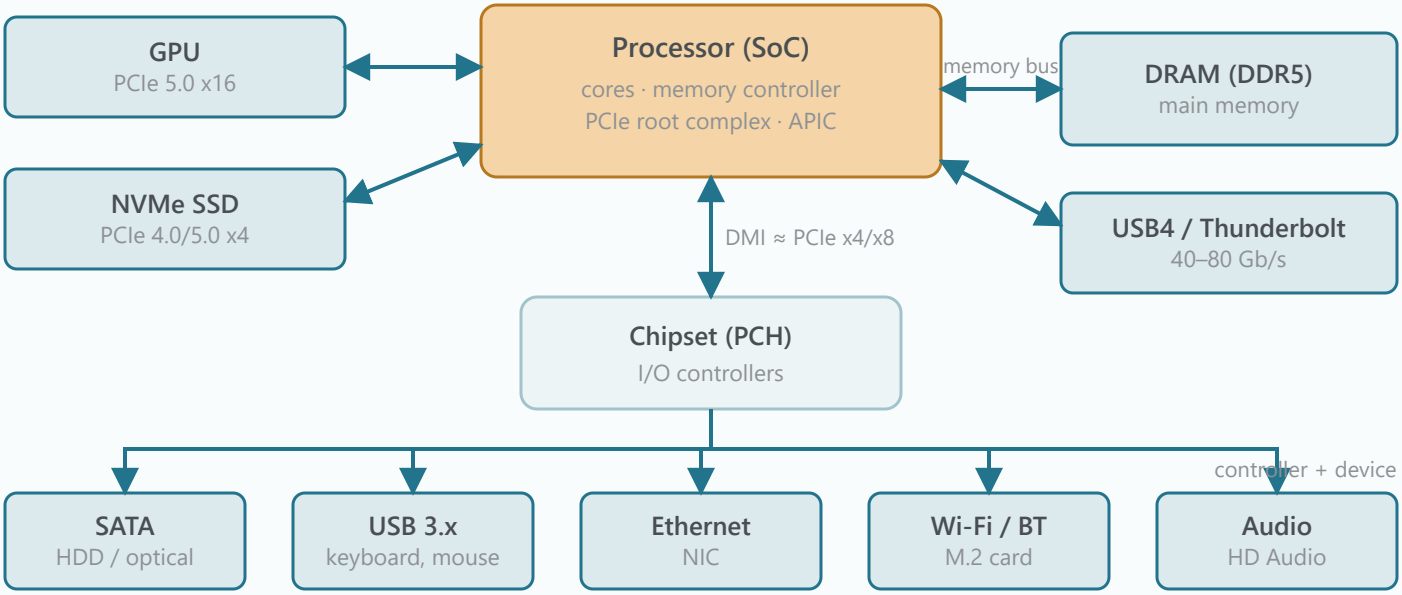
Disks

Compute access time; find the total head movement of FCFS, SSTF, SCAN, C-SCAN, LOOK and C-LOOK.

SSD and RAID

Explain the FTL, wear leveling, TRIM and blk-mq; RAID 0/1/5/6/10 and recovery with XOR parity.

I/O System Overview



- The OS **issues commands** to devices, **reads their status**, catches **interrupts** and handles **errors**.
- It gives applications a uniform, layered, device-independent interface.
- Most kernel code is **device drivers** (about 60–70% of the Linux source lives in `drivers/`).

Buses: PCIe, USB4, SATA

- **Bus**: a set of wires carrying data, address and control signals between components. Old PCs used shared parallel buses (ISA, PCI); today we use **point-to-point serial links** (PCIe, USB, SATA).
- A PCIe lane is full duplex; a link bundles x1, x4, x8 or x16 lanes. Bandwidth is counted per direction.

Link	Raw rate	Effective data rate	Note
PCIe 4.0	16 GT/s / lane	≈ 1.97 GB/s / lane; x4 ≈ 7.9 GB/s	128b/130b encoding; NVMe SSDs
PCIe 5.0	32 GT/s / lane	≈ 3.94 GB/s / lane; x16 ≈ 63 GB/s	GPUs, 14 GB/s-class SSDs
PCIe 6.0	64 GT/s (PAM4)	≈ 7.6 GB/s / lane; x16 ≈ 121 GB/s	FLIT + FEC; growing in servers
USB 2.0 / 3.2	480 Mb/s / 5–20 Gb/s	≤ 60 MB/s / ≤ 2.5 GB/s	Host-polled protocol
USB4 v1 / v2	40 / 80 Gb/s	PCIe and DisplayPort tunneling	Thunderbolt 4/5 compatible
SATA III	6 Gb/s	≈ 600 MB/s (8b/10b)	AHCI: one queue, 32 commands
Ethernet	1 / 10 / 100 / 400 Gb/s	125 MB/s … 50 GB/s	Data center: 400–800 GbE

Math: a PCIe 4.0 lane gives 16 GT/s × 128/130 ÷ 8 = 1.97 GB/s. FireWire, 56K modems and 10/100 Mb/s Ethernet are history. The Pentium-era northbridge is gone: the memory controller moved into the CPU (AMD 2003, Intel 2008); fast devices attach to the CPU over PCIe, slow ones to the chipset.

Device Data Rates



A keyboard and PCIe 5.0 x16 differ by a factor of $\approx 6 \times 10^9$: the same OS must handle a few keystrokes per second and millions of NVMe requests per second. Values are typical upper limits; real rates depend on signal quality, protocol overhead and the device.

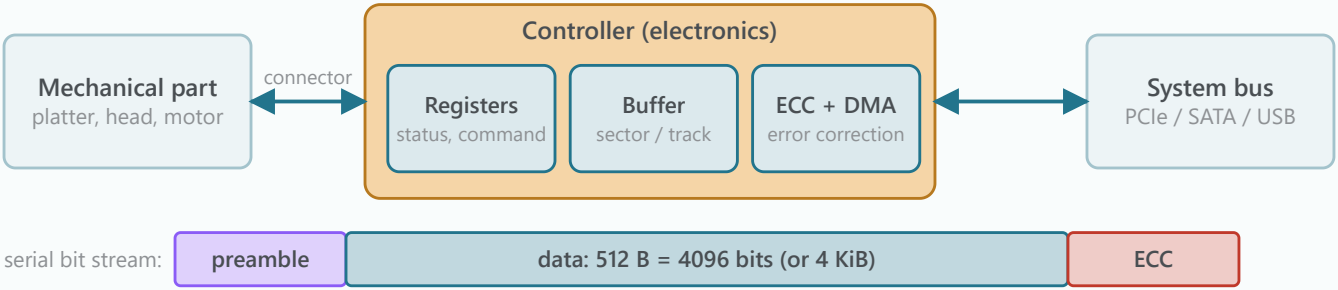
Device Classes: Block, Character, Network

Class	Example	Property	Typical interface
Block	HDD, SSD, USB stick, SD card	Fixed-size, *addressable* blocks (512 B / 4 KiB); each block is read or written independently	`read`, `write`, `lseek`; file system, `mmap`
Character	Keyboard, mouse, serial port, printer, audio	A byte *stream*; not addressable, no seek	`read` / `write`, `ioctl`; `/dev/ttyS0`
Network	Ethernet, Wi-Fi	Packet based; neither block nor character	Socket API (`send`, `recv`); no file in `/dev`
Other	Clocks, timers, GPUs	Neither stream nor block: they raise interrupts or are memory-mapped	`timerfd`, DRM/KMS, `mmap`

```
$ ls -l /dev/nvme0n1 /dev/ttyS0
brw-rw---- 1 root disk    259, 0  nvme0n1   b = block, major/minor 259,0
crw-rw---- 1 root dialout   4, 64 ttyS0     c = character
```

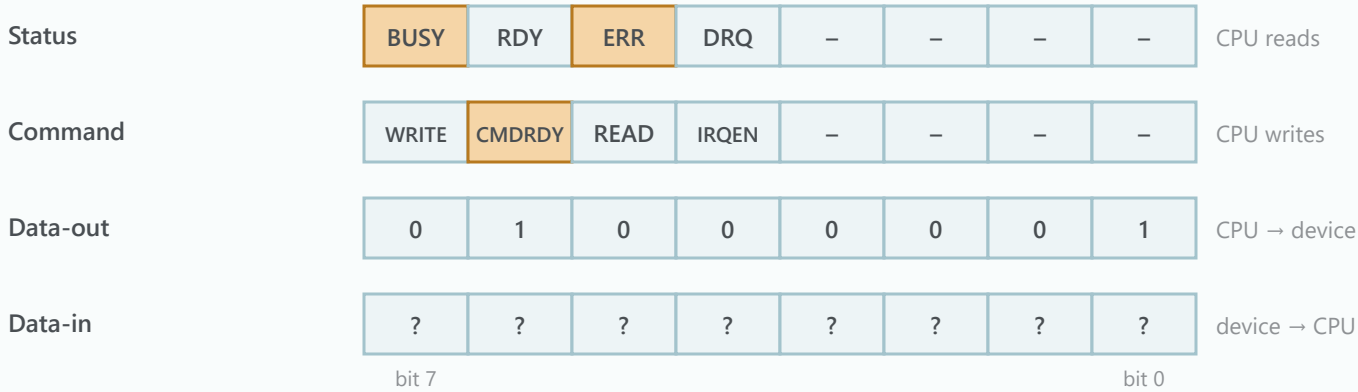
Network interfaces are not character devices; in Unix/Linux networking is a class of its own. Block size today is mostly 512 B (emulated) or 4 KiB. The classification is not airtight: tapes and clocks do not fit neatly.

Device Controllers



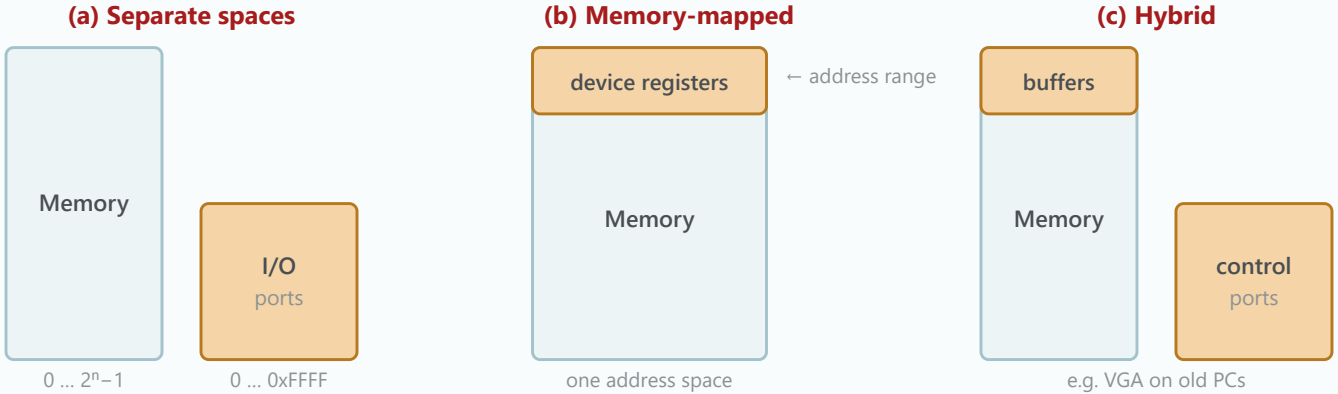
- An I/O unit = a **mechanical/device** part + an **electronic controller** (a chip or card, linked to the device by a connector).
- The **preamble** carries a sync pattern and the cylinder/sector number. The controller turns the bit stream into a block, corrects errors with **ECC**, keeps the block in its own buffer, then copies it to memory.
- At boot and on hot-plug the OS **enumerates** the buses (PCIe config space, USB descriptors), reads each device ID and loads the matching driver: **plug and play**.
- The OS talks to the **controller**, not the device; standard interfaces such as SATA and NVMe let one driver run devices from many vendors.
- Modern disks use 4 KiB physical sectors (Advanced Format); drives that expose 512 B sectors are called *512e*.

Device Registers: Status, Command, Data



- A controller typically has four registers: **status** (busy, ready, error), **command** (what to do), **data-in** and **data-out**.
- Writing a register = *issuing a command*; reading one = *learning the status*. Devices may also have large data buffers (e.g. a frame buffer, NIC rings).
- Register layout is device-specific; only the **driver** knows it. Bit names here are illustrative.

Port-Mapped and Memory-Mapped I/O



Port (x86)	Device
0x20–0x21	8259 interrupt controller
0x40–0x43	8254 timer (PIT)
0x60, 0x64	8042 keyboard controller
0x3F8–0x3FF	COM1 serial port
0xCF8–0xCFF	PCI configuration

```
in  al, 0x60          ; port: special instruction
mov eax, [rbx+0x14]   ; MMIO: ordinary load

Linux: inb()/outb()  ·  ioremap() + readl()/writel()
/proc/ioports  ·  /proc/iomem
```

Port-mapped I/O uses special `IN` / `OUT` instructions and a separate address space; with memory-mapped I/O a device register is an ordinary memory address that the address decoder routes to the device. ARM and RISC-V use only MMIO; PCIe devices expose registers as MMIO via BARs.

Memory-Mapped I/O: Pros and Cons

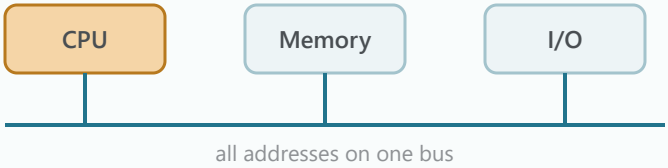
Memory-mapped I/O: pros

+ No special instructions; a driver can be written entirely in C (a `volatile` pointer).
+ Protection comes from the page table: if the register page is not mapped into user space it is unreachable; it can also be handed to a single process (VFIO, DPDK).
+ Every memory instruction works on registers: `TEST`, `AND` and the like operate on a register directly.

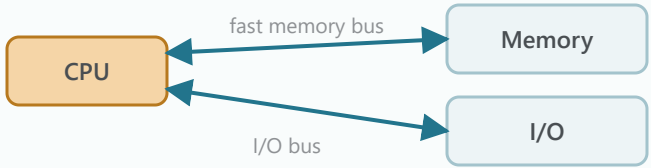
Cons

– **Caching**: a cached status register hides the current value → the page is marked *uncacheable* (PAT/MTRR, `ioremap`).
– **Multiple buses**: if memory sits on its own fast bus, I/O devices never see the address → the bridge or root complex routes by address range.
– The compiler may reorder accesses: `volatile` and memory barriers are required.

(a) single bus

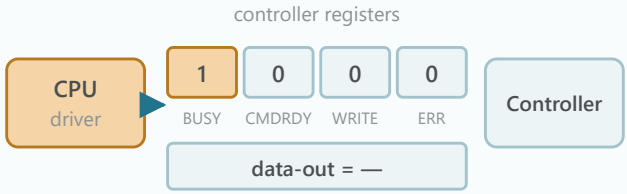


(b) dual bus



Polling: The Handshake Step by Step

```
▶ while (status & BUSY)
▶     ;          /* busy-wait */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



The driver **reads the status register over and over**: BUSY = 1 because the controller has not finished the previous job. Meanwhile the CPU can do nothing else.

Step 1 / 8

VARIABLES

`BUSY = 1` `CMDRDY = 0` `WRITE = 0` `ERR = 0`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;           /* busy-wait */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



BUSY dropped to 0: the loop exits. How long this takes depends on the device, anywhere from a few cycles to milliseconds.

Step 2 / 8

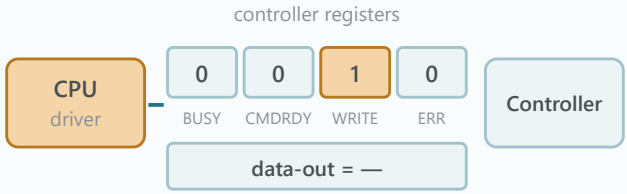
VARIABLES

`BUSY = 0` `CMDRDY = 0` `WRITE = 0` `ERR = 0`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;          /* busy-wait */
▶ command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



The **WRITE** bit is set in the command register: the job is a write.

Step 3 / 8

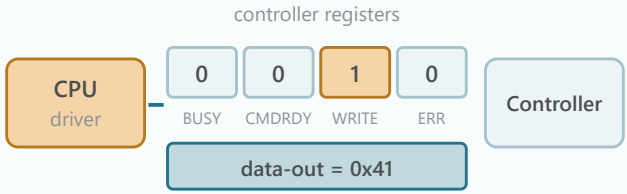
VARIABLES

`BUSY = 0` `CMDRDY = 0` `WRITE = 1` `ERR = 0`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;           /* busy-wait */
3 command = WRITE;
▶ data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



The byte to send ('A' = 0x41) goes into the **data-out** register.

Step 4 / 8

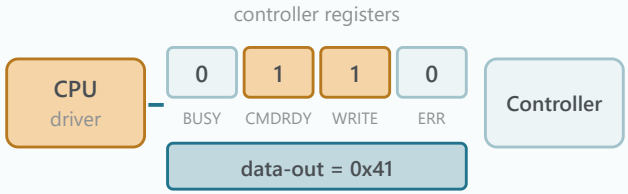
VARIABLES

`BUSY = 0` `CMDRDY = 0` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;           /* busy-wait */
3 command = WRITE;
4 data_out = 'A';
▶ command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



CMDRDY = 1: “command ready”. This bit tells the controller it may start.

Step 5 / 8

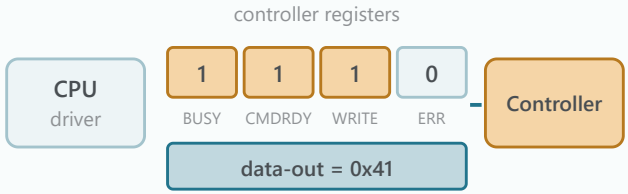
VARIABLES

`BUSY = 0` `CMDRDY = 1` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;           /* busy-wait */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
▶ /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



The controller sees CMDRDY and sets **BUSY = 1**, telling the host it is now busy.

Step 6 / 8

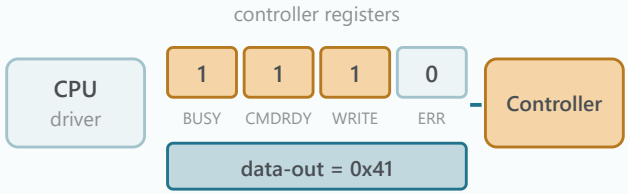
VARIABLES

`BUSY = 1` `CMDRDY = 1` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;           /* busy-wait */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
▶ /* read command, send byte */
9 /* CMDRDY = ERR = BUSY = 0 */
```



The controller reads the command register (WRITE), takes the byte from data-out and sends it to the device.

Step 7 / 8

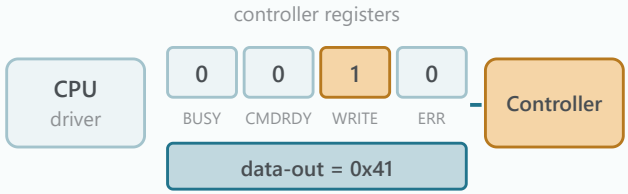
VARIABLES

`BUSY = 1` `CMDRDY = 1` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Polling: The Handshake Step by Step

```
1 while (status & BUSY)
2     ;           /* busy-wait */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- controller side --- */
7 /* sees CMDRDY → BUSY = 1 */
8 /* read command, send byte */
▶ /* CMDRDY = ERR = BUSY = 0 */
```



Done: CMDRDY, ERR and BUSY are cleared (ERR = 0 → success). The loop starts over for the next byte.

Step 8 / 8

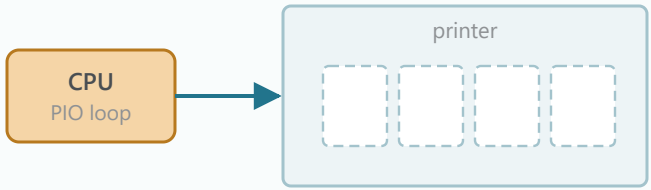
VARIABLES

`BUSY = 0` `CMDRDY = 0` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programmed I/O (PIO): the CPU runs this handshake itself for every byte. It is simple, but the CPU spins while it waits; if the CPU switches to another process and misses a byte (e.g. a buffer overrun), data can be lost.

Programmed I/O: Printer Example

```
▶ copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* busy-wait */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



The user's “ABCD” string is copied into a kernel buffer (`copy_from_user`); user memory is never used directly.

Step 1 / 10

VARIABLES

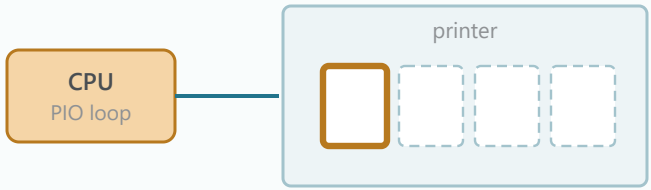
`i = –` `count = 4` `status = READY`

OUTPUT

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* busy-wait */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 0: the loop spins until the status is READY.

Step 2 / 10

VARIABLES

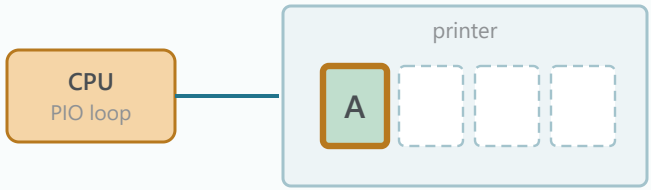
`i = 0` `count = 4` `status = BUSY`

OUTPUT

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* busy-wait */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'A' is written to the data register; the printer prints it and goes busy again.

Step 3 / 10

VARIABLES

`i = 0` `count = 4` `status = READY`

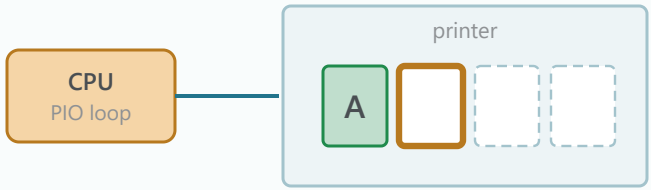
OUTPUT

```
A
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* busy-wait */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 1: the printer is busy, so the CPU **polls** the status register.

Step 4 / 10

VARIABLES

`i = 1` `count = 4` `status = BUSY`

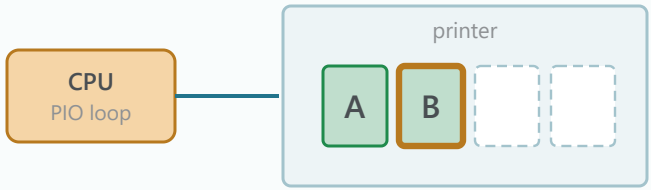
OUTPUT

```
A
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* busy-wait */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'B' is written to the data register; the printer prints it and goes busy again.

Step 5 / 10

VARIABLES

`i = 1` `count = 4` `status = READY`

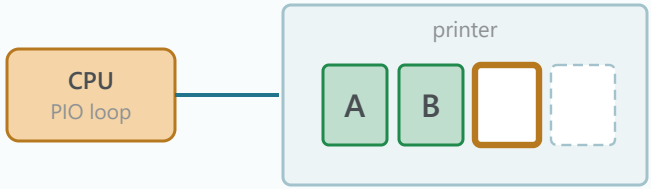
OUTPUT

```
AB
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* busy-wait */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 2: the printer is busy, so the CPU **polls** the status register.

Step 6 / 10

VARIABLES

`i = 2` `count = 4` `status = BUSY`

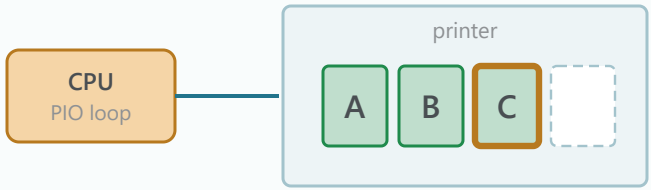
OUTPUT

```
AB
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* busy-wait */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'C' is written to the data register; the printer prints it and goes busy again.

Step 7 / 10

VARIABLES

`i = 2` `count = 4` `status = READY`

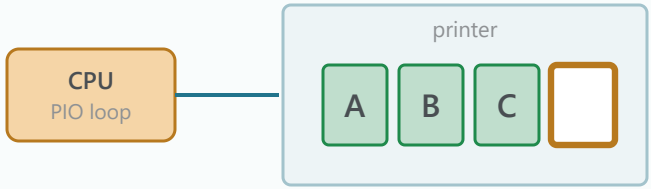
OUTPUT

```
ABC
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* busy-wait */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 3: the printer is busy, so the CPU **polls** the status register.

Step 8 / 10

VARIABLES

`i = 3` `count = 4` `status = BUSY`

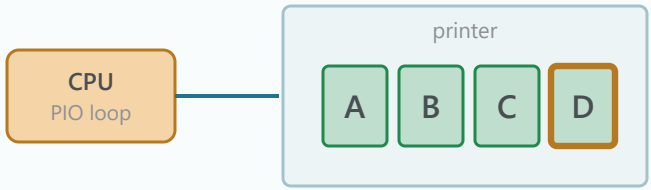
OUTPUT

```
ABC
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* busy-wait */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'D' is written to the data register; the printer prints it and goes busy again.

Step 9 / 10

VARIABLES

`i = 3` `count = 4` `status = READY`

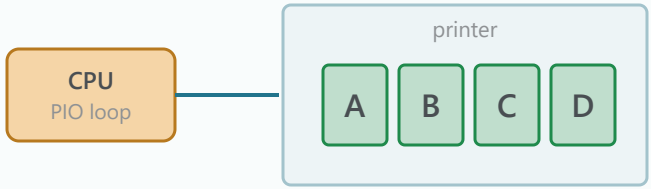
OUTPUT

```
ABCD
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Programmed I/O: Printer Example

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* busy-wait */
5     *printer_data_register = p[i];
6 }
▶ return_to_user();
```



Loop done; control returns to the process. The CPU was tied to this process the whole time: a 4-character job runs at printer speed.

Step 10 / 10

VARIABLES

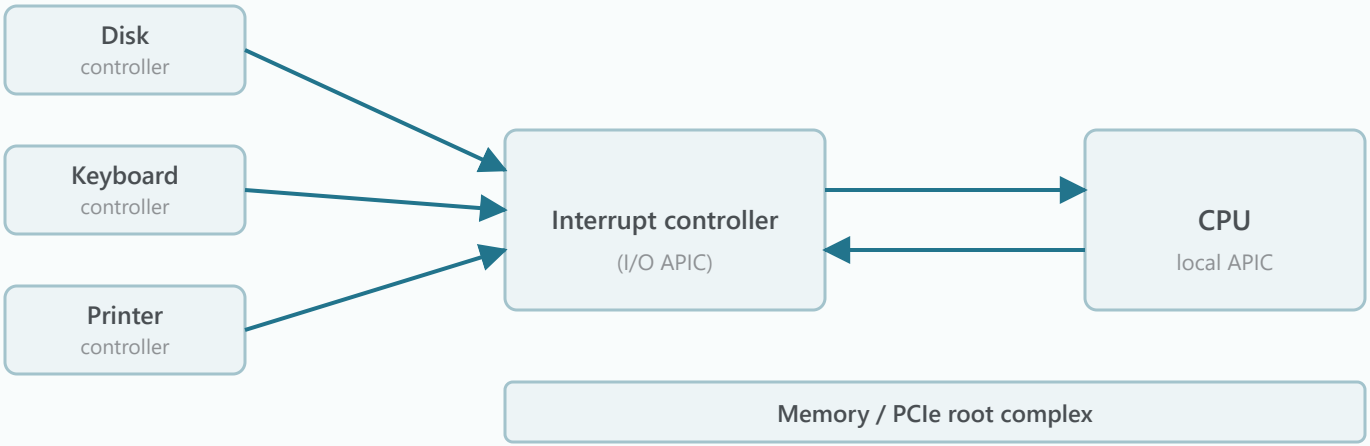
`i = 4` `count = 4` `status = READY`

OUTPUT

```
ABCD
```

Tanenbaum's printer example. At 100 characters per second the CPU waits ≈ 10 ms per character, time in which it could run millions of instructions. In embedded systems where the CPU has nothing else to do, PIO still makes sense.

Interrupt Hardware: From Lines to MSI

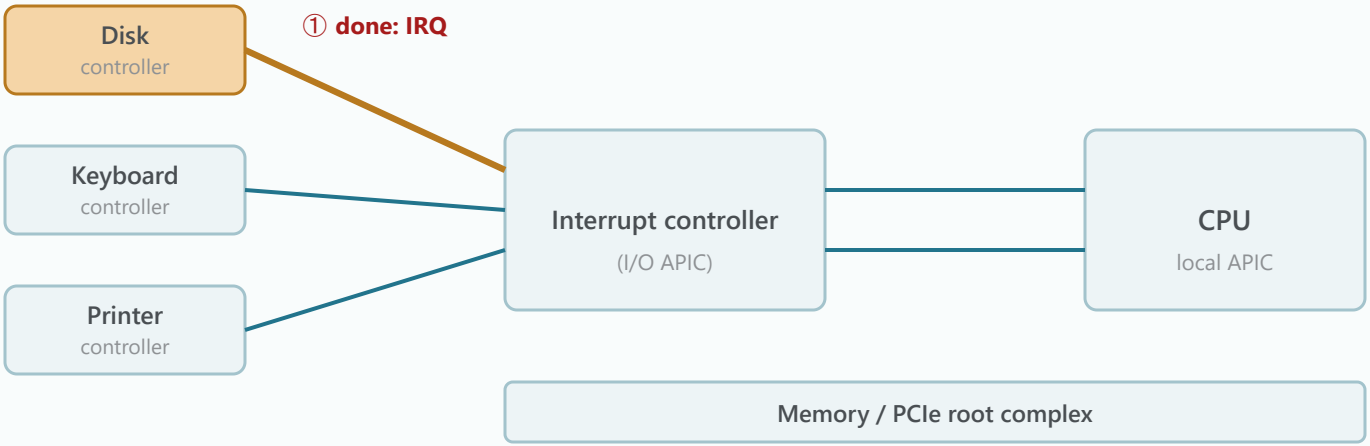


Every device is wired to the interrupt controller; the controller shares the CPU's single *interrupt input* and sets priorities.

Step 1 / 5

Old PCs had two 8259 PICs and 15 IRQ lines. Today x86 has an I/O APIC plus a local APIC per core (x2APIC); ARM has the GIC. The “controller puts a number on the address lines” story applies only to that historical PIC setup.

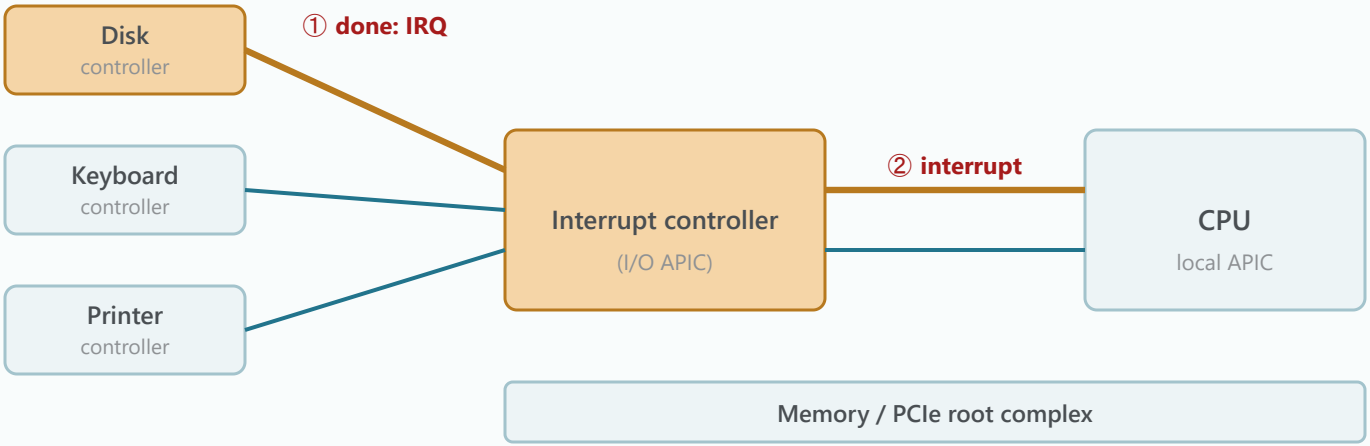
Interrupt Hardware: From Lines to MSI



① When the disk controller finishes, it signals on its **interrupt line** (IRQ).

Old PCs had two 8259 PICs and 15 IRQ lines. Today x86 has an I/O APIC plus a local APIC per core (x2APIC); ARM has the GIC. The “controller puts a number on the address lines” story applies only to that historical PIC setup.

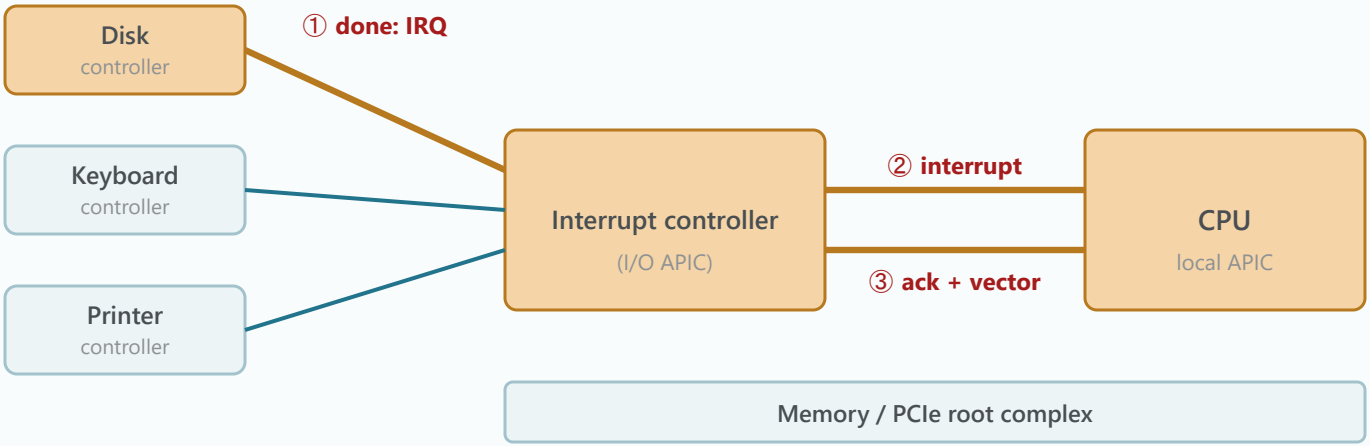
Interrupt Hardware: From Lines to MSI



② If no other interrupt is in progress, or this one has higher priority, the controller interrupts the CPU; otherwise the request waits (the line stays asserted until the CPU acknowledges).

Old PCs had two 8259 PICs and 15 IRQ lines. Today x86 has an I/O APIC plus a local APIC per core (x2APIC); ARM has the GIC. The “controller puts a number on the address lines” story applies only to that historical PIC setup.

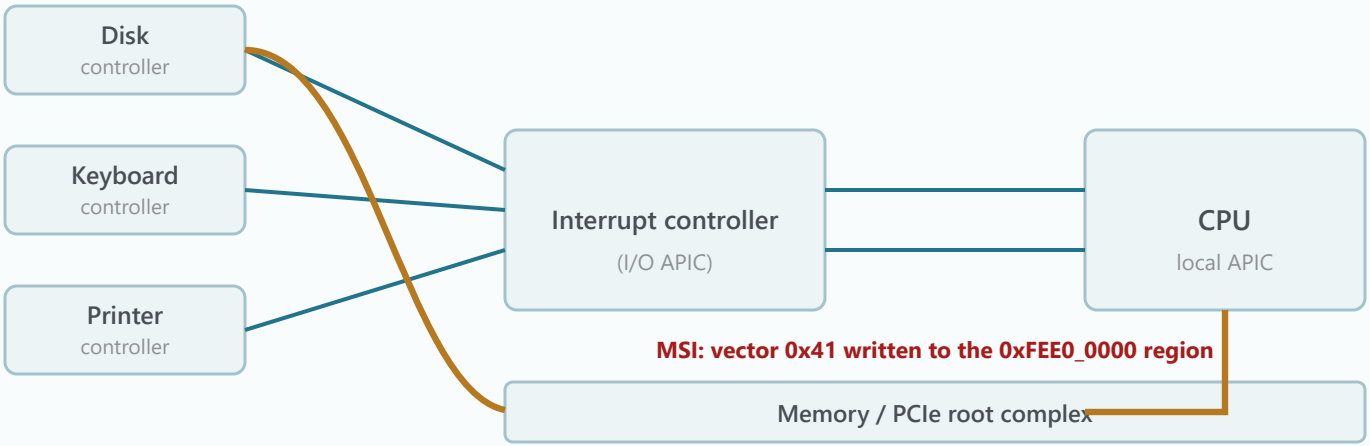
Interrupt Hardware: From Lines to MSI



③ The CPU accepts the interrupt; the controller puts the **vector number** on the bus. The CPU uses it as an index into the interrupt vector table.

Old PCs had two 8259 PICs and 15 IRQ lines. Today x86 has an I/O APIC plus a local APIC per core (x2APIC); ARM has the GIC. The “controller puts a number on the address lines” story applies only to that historical PIC setup.

Interrupt Hardware: From Lines to MSI

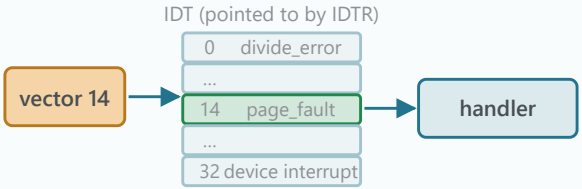


Modern PCIe uses **MSI/MSI-X**. No wire: the device sends a **memory write** containing the vector to a special address (the 0xFEE0_0000 region on x86), and the local APIC turns it into an interrupt.

Old PCs had two 8259 PICs and 15 IRQ lines. Today x86 has an I/O APIC plus a local APIC per core (x2APIC); ARM has the GIC. The “controller puts a number on the address lines” story applies only to that historical PIC setup.

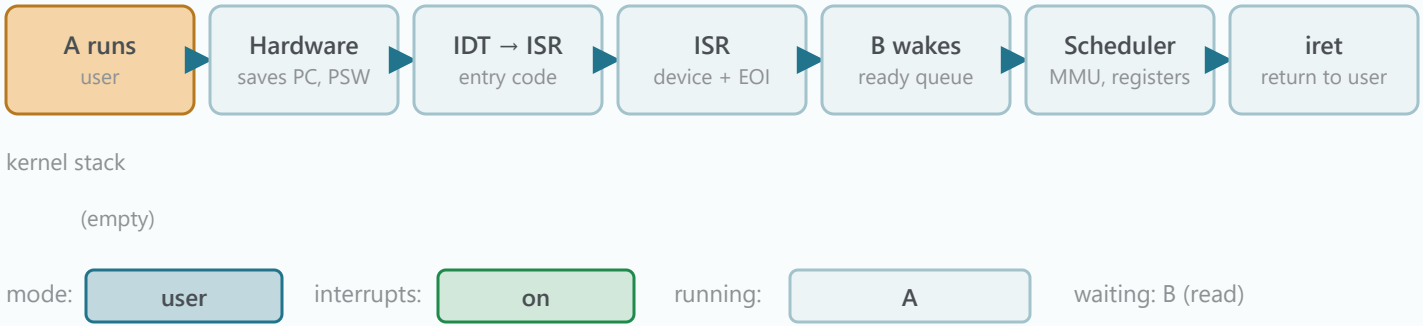
Vectors, Priority, NMI and Exceptions

Vector	Event	Type
0	Divide error	fault
1 / 3	Debug / breakpoint	trap
2	NMI (non-maskable interrupt)	interrupt
6	Invalid opcode	fault
13	General protection fault	fault
14	Page fault	fault
18	Machine check	abort
32–255	Device interrupts, IPIs, APIC timer	maskable



- **Interrupt vector table** (x86: the IDT, 256 entries): vector → handler address. Vectors 0–31 are reserved for processor exceptions.
- **Maskable** interrupts can be deferred inside a critical section (`cli` / `sti`); an **NMI** cannot: hardware errors, memory ECC, watchdog.
- **Priority**: the APIC orders interrupts by vector class; a high-priority interrupt can preempt a low-priority handler.
- **Interrupt chaining**: with more devices than vectors, each entry points to a list of handlers (shared IRQ).
- A **fault** (page fault) cannot complete the instruction; after the handler *the same instruction* runs again. A **trap** (system call, `int3`) resumes at the next instruction.
- **MSI-X**: up to 2048 vectors per function; NVMe and NICs take one vector per queue and steer each interrupt to that queue's core (`/proc/interrupts` , `irqbalance`).

Interrupt Handling Step by Step

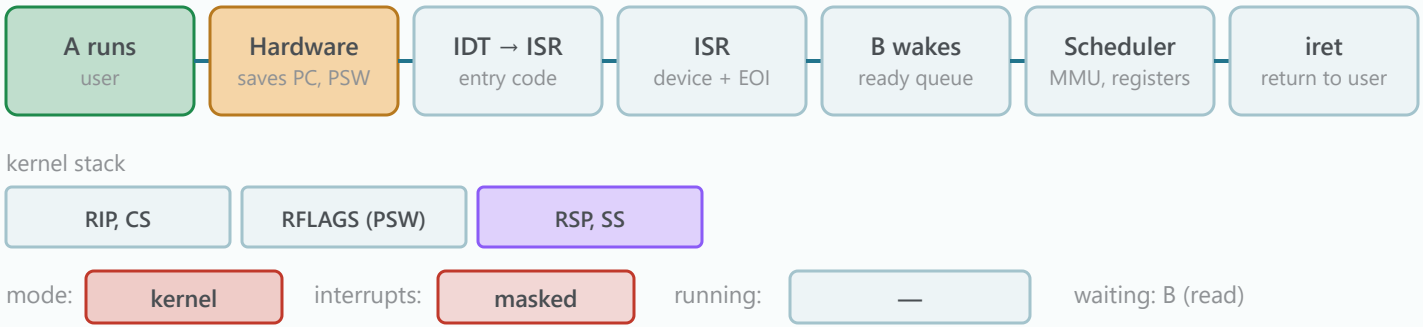


Process A is running; B is waiting for a disk read. A disk interrupt arrives. The CPU **finishes the current instruction**, then accepts the interrupt.

Step 1 / 7

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

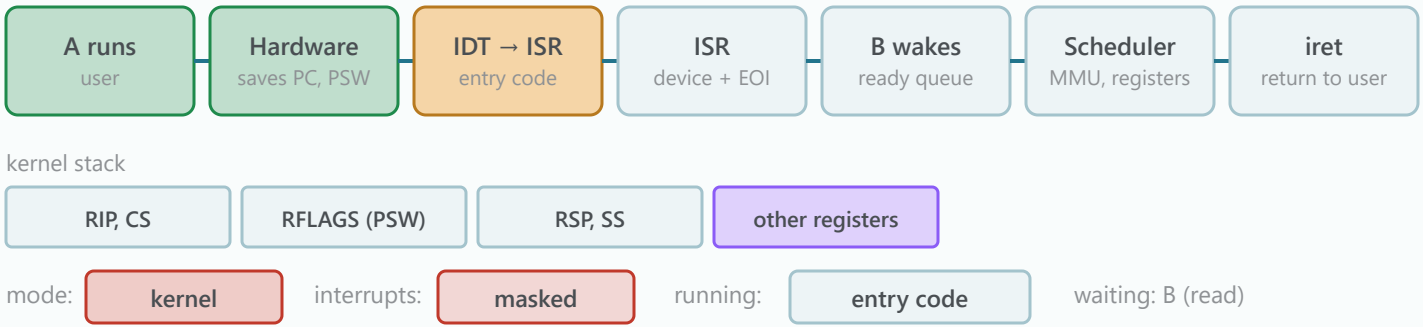
Interrupt Handling Step by Step



The **hardware** pushes the PC (RIP), PSW (RFLAGS) and stack pointer onto the **kernel stack**, switches to kernel mode and masks interrupts.

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

Interrupt Handling Step by Step

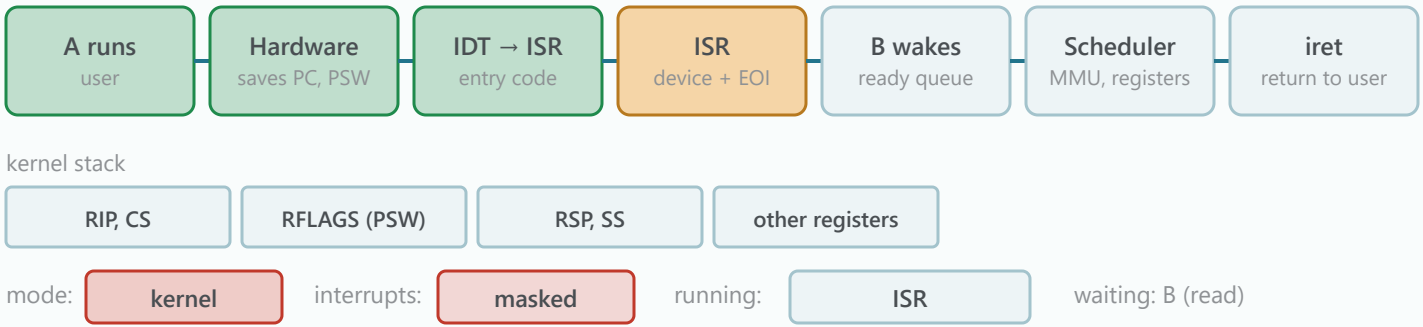


The vector number indexes the IDT → handler address. The entry code (assembly) saves the registers the hardware did not; context and stack are ready for the ISR.

Step 3 / 7

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

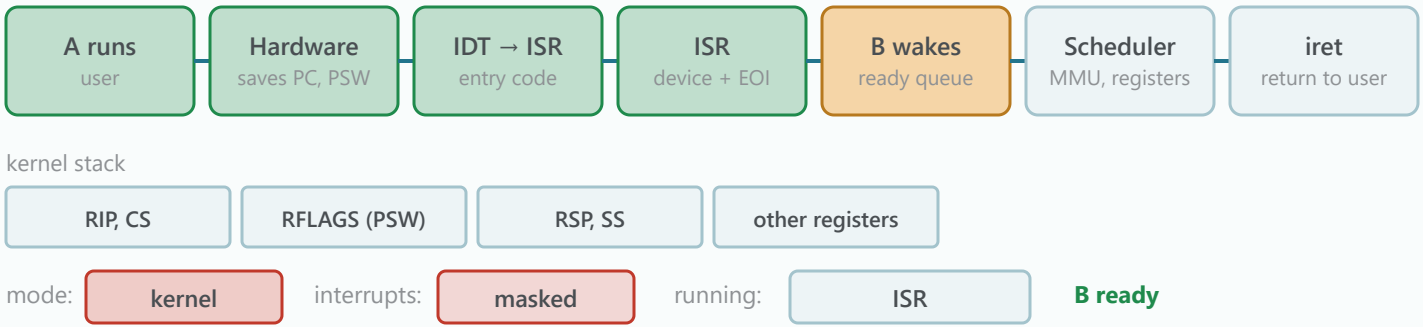
Interrupt Handling Step by Step



The ISR reads the device status, takes the data and sends the controller an **acknowledgment (EOI)**; new interrupts may now arrive. Long work is deferred to the bottom half.

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

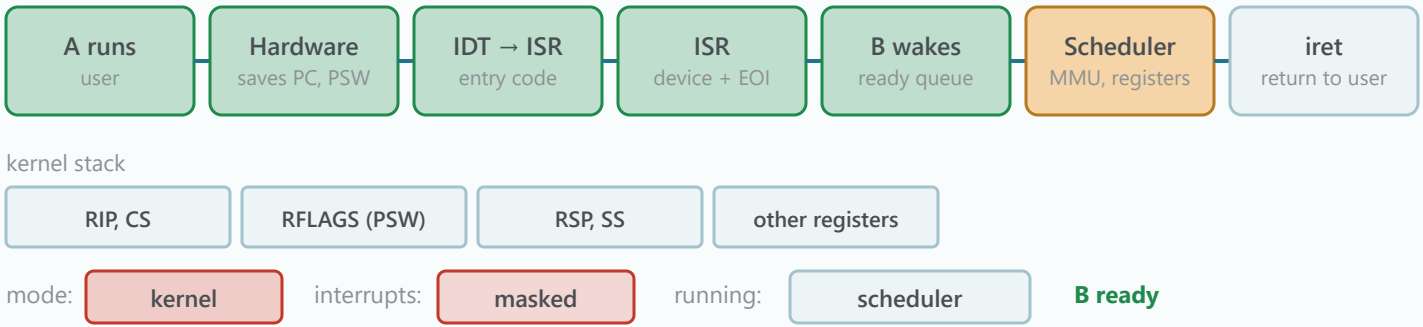
Interrupt Handling Step by Step



The waiting process B moves from **blocked** to the **ready** queue.

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

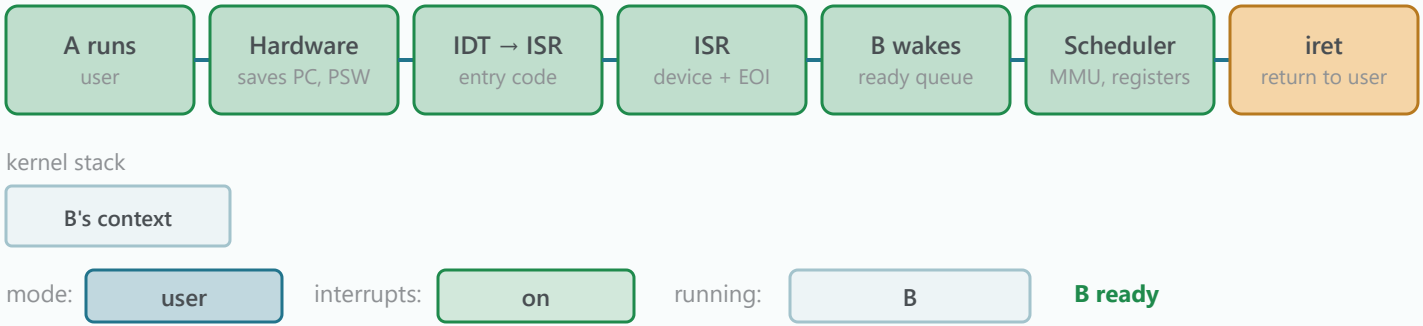
Interrupt Handling Step by Step



The scheduler picks the next process (here B). B's page table is loaded into the MMU (x86: CR3) and its registers are restored from the process table / stack.

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

Interrupt Handling Step by Step

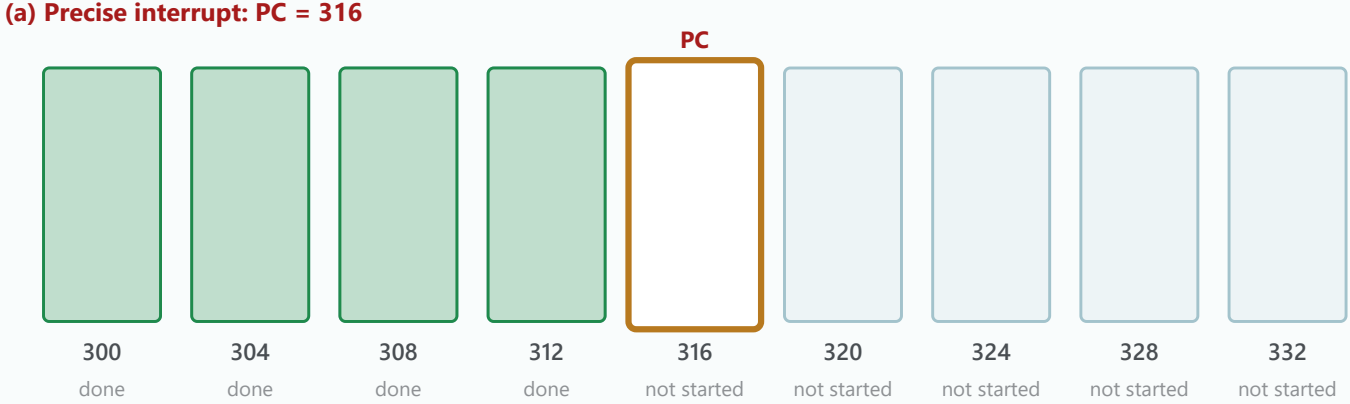


`iret`: the saved PC and PSW are restored, the CPU returns to user mode with interrupts enabled. B resumes where it left off.

Step 7 / 7

This sequence follows Tanenbaum's interrupt-handler steps. The scheduler could also pick A; the choice depends on the policy.

Precise and Imprecise Interrupts



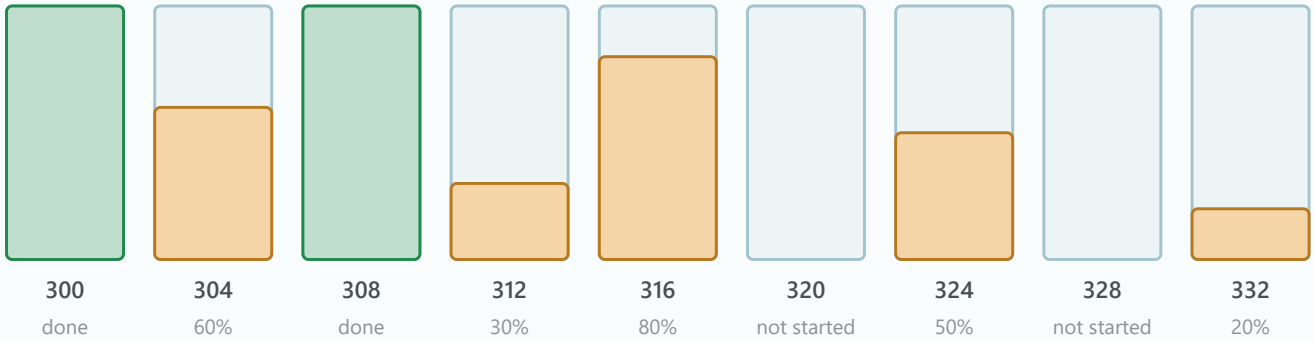
Precise interrupt: the PC is saved in a known place; every instruction before the PC has **fully** completed, **none** after it has started, and the state of the one at the PC is known.

Step 1 / 2

- A pipeline holds many instructions in different stages; a superscalar CPU completes them out of order. The “PC” alone does not describe the state.
- Modern x86, ARM and RISC-V processors provide **precise interrupts**: the reorder buffer (ROB) retires instructions in program order, and on an interrupt the effects of later instructions are discarded. The price is chip area and complexity.

Precise and Imprecise Interrupts

(b) Imprecise: instructions in different stages



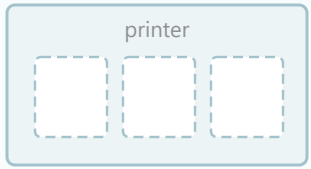
Imprecise interrupt: with pipelining or out-of-order execution, instructions are half done. To restart, either the hardware rolls back or the OS must save and decode a lot of internal state.

Step 2 / 2

- A pipeline holds many instructions in different stages; a superscalar CPU completes them out of order. The “PC” alone does not describe the state.
- Modern x86, ARM and RISC-V processors provide **precise interrupts**: the reorder buffer (ROB) retires instructions in program order, and on an interrupt the effects of later instructions are discarded. The price is chip area and complexity.

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
▶  copy_from_user(buffer, p, count);
▶  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* block process */
8  /* interrupt service routine */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



process: running

CPU: process

"ABC" is copied into a kernel buffer and printer interrupts are enabled.

Step 1 / 7

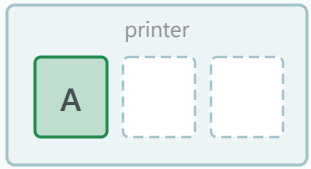
VARIABLES

`i = –` `count = 3`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
▶  while (*printer_status_reg != READY) ;
▶  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* block process */
8  /* interrupt service routine */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



process: running

CPU: process

When the printer is ready, **only the first character** is written.

Step 2 / 7

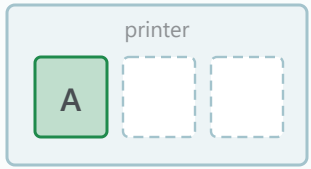
VARIABLES

`i = –` `count = 3`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
▶  i = 1; count = count - 1;
▶  scheduler();      /* block process */
8  /* interrupt service routine */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



process: blocked

CPU: other process

The counters are updated and the scheduler is called: the process **blocks** and the CPU runs another process.

Step 3 / 7

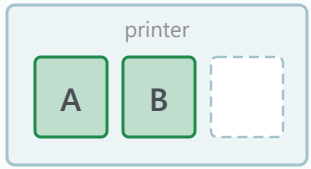
VARIABLES

`i = 1` `count = 2`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* block process */
8  /* interrupt service routine */
9  if (count == 0) {
10     unblock_user();
▶  } else {
▶      *printer_data_register = p[i];
▶      count = count - 1;
▶      i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



process: blocked

CPU: ISR

Interrupt 1: 'A' printed. count ≠ 0 → p[1] = 'B' is written.

Step 4 / 7

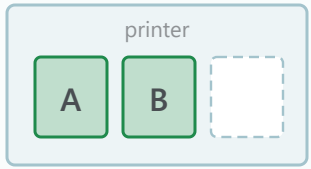
VARIABLES

`i = 2` `count = 1`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* block process */
8  /* interrupt service routine */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
▶  acknowledge_interrupt();
▶  return_from_interrupt();
```



process: blocked

CPU: ISR

The interrupt is acknowledged and the interrupted process resumes. The CPU did not spin while the printer worked.

Step 5 / 7

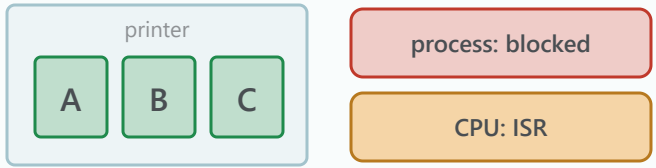
VARIABLES

`i = 2` `count = 1`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* block process */
8  /* interrupt service routine */
9  if (count == 0) {
10     unblock_user();
11 } else {
▶      *printer_data_register = p[i];
▶      count = count - 1;
▶      i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



Interrupt 2: 'B' printed → p[2] = 'C' is written.

Step 6 / 7

VARIABLES

`i = 3` `count = 0`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Interrupt-Driven I/O: Printer Example

```
1  /* system call */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* block process */
8  /* interrupt service routine */
▶  if (count == 0) {
▶      unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



Interrupt 3: count == 0 → all characters printed, so the user process moves to the **ready** queue.

Step 7 / 7

VARIABLES

`i = 3` `count = 0`

One interrupt per character: 3 characters → 3 interrupts. Lines 13 and 14 of the ISR update `i` and `count` after each character. The cost: saving context and running the handler on every interrupt (≈ 1–2 µs).

Top Half, Bottom Half and Hybrid Polling

Top half (hardirq)

The real ISR: kept short, runs with interrupts partly disabled. It acknowledges the device, buffers the data and schedules the bottom half.

Bottom half

Deferred work: in Linux a *softirq*, *tasklet*, *workqueue* or threaded IRQ (`request_threaded_irq`); in Windows the ISR + DPC.

Interrupt coalescing

NICs and NVMe report several completions in one interrupt: fewer interrupts, slightly more latency.

NAPI: interrupts + polling

Under heavy network traffic the driver disables the interrupt and *polls* the queue, then returns to interrupts when traffic drops. The hybrid wins.

When pure polling?

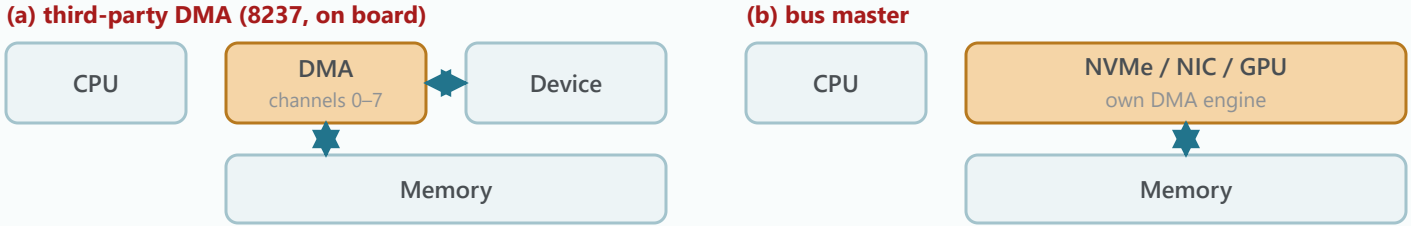
When the event rate is very high and latency is critical: DPDK/SPDK, NVMe polled queues, `io_uring` IOPOLL. One core is dedicated to polling.

Interrupt storm

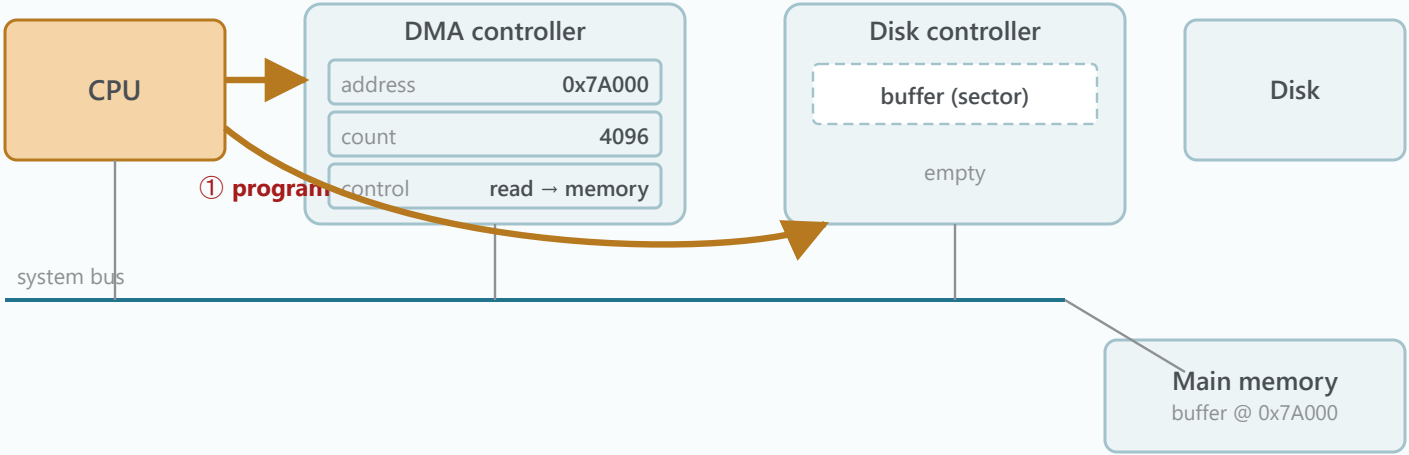
A faulty device that interrupts nonstop locks up the system; the kernel disables the line (“nobody cared”) and rate-limits interrupts.

Direct Memory Access (DMA)

- Byte-by-byte transfer wastes CPU time; a **DMA** controller moves whole blocks without the CPU. The CPU only sets up the transfer and handles the completion interrupt.
- Registers: **memory address**, **byte count**, **control** (I/O port, direction: read/write, unit: byte/word, burst size). A multichannel controller handles many transfers at once.
- A single DMA controller on the motherboard is only the historical 8237 design. Today most devices have **their own DMA engine** and write to memory directly over PCIe.



DMA Transfer Step by Step

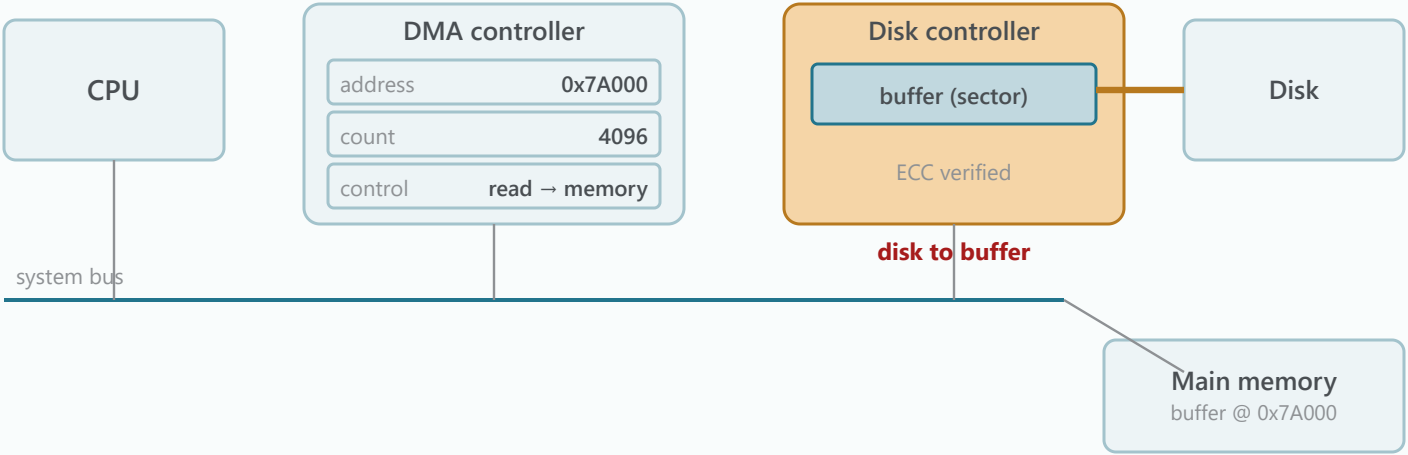


① The CPU programs the DMA controller (address = 0x7A000, count = 4096, direction) and the disk controller (read the sector), then goes back to other work.

Step 1 / 6

Cycle stealing: DMA grabs the bus briefly for each word, so the CPU slows a little when it accesses memory. In **burst** mode the bus is held for a whole block. In fly-by mode data goes straight from device to memory; otherwise the DMA controller reads first and then writes.

DMA Transfer Step by Step

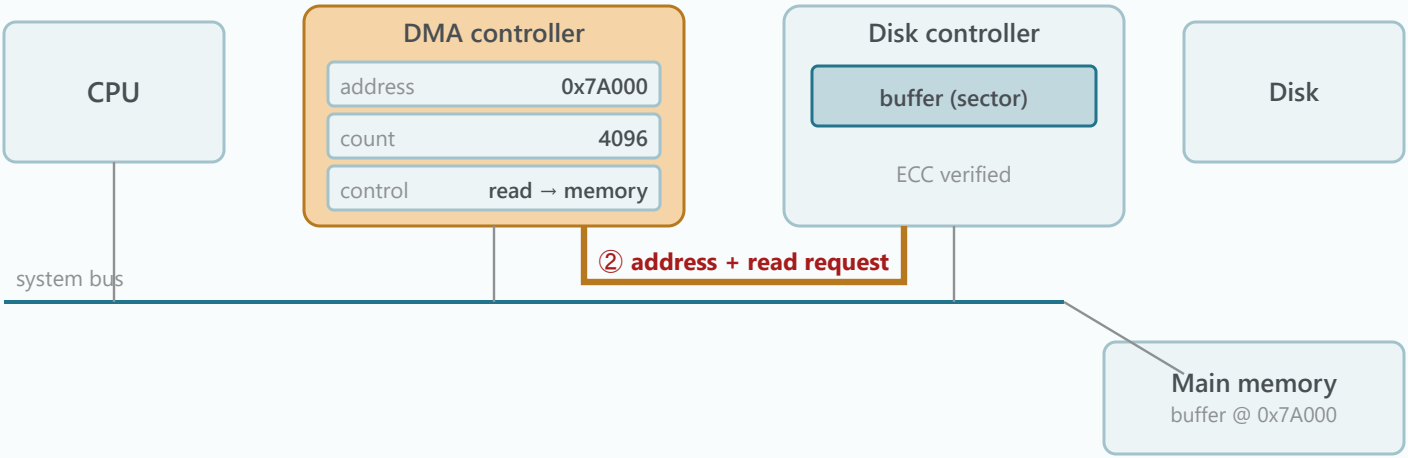


The disk controller reads the sector into its own buffer and checks it with ECC. Thanks to the buffer, no data is lost while the bus is busy.

Step 2 / 6

Cycle stealing: DMA grabs the bus briefly for each word, so the CPU slows a little when it accesses memory. In **burst** mode the bus is held for a whole block. In fly-by mode data goes straight from device to memory; otherwise the DMA controller reads first and then writes.

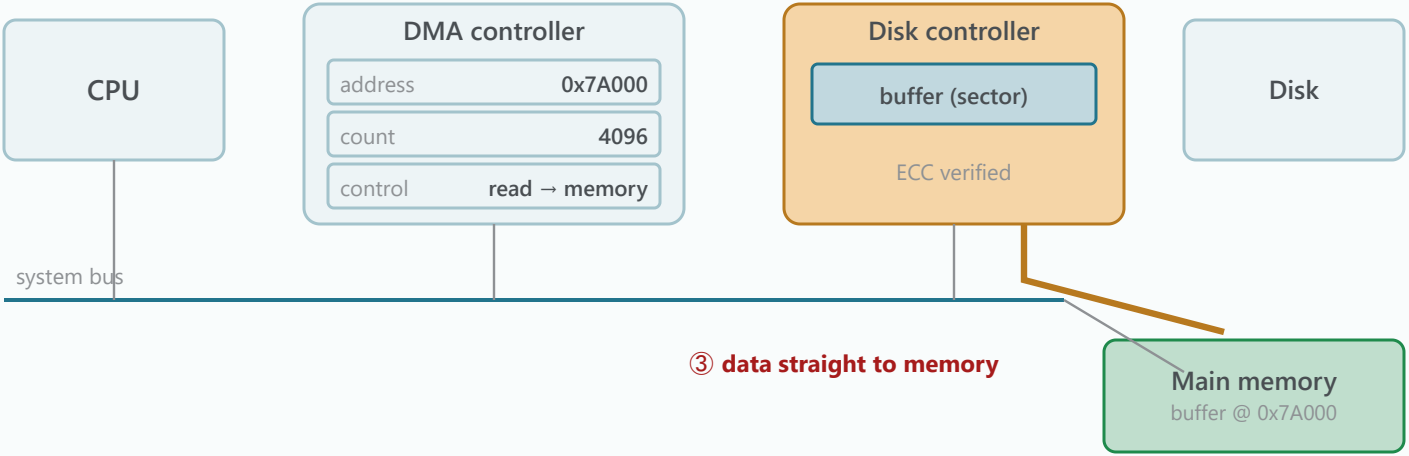
DMA Transfer Step by Step



② The DMA controller requests the bus, puts the memory address on it and sends a read request to the disk controller.

Cycle stealing: DMA grabs the bus briefly for each word, so the CPU slows a little when it accesses memory. In **burst** mode the bus is held for a whole block. In fly-by mode data goes straight from device to memory; otherwise the DMA controller reads first and then writes.

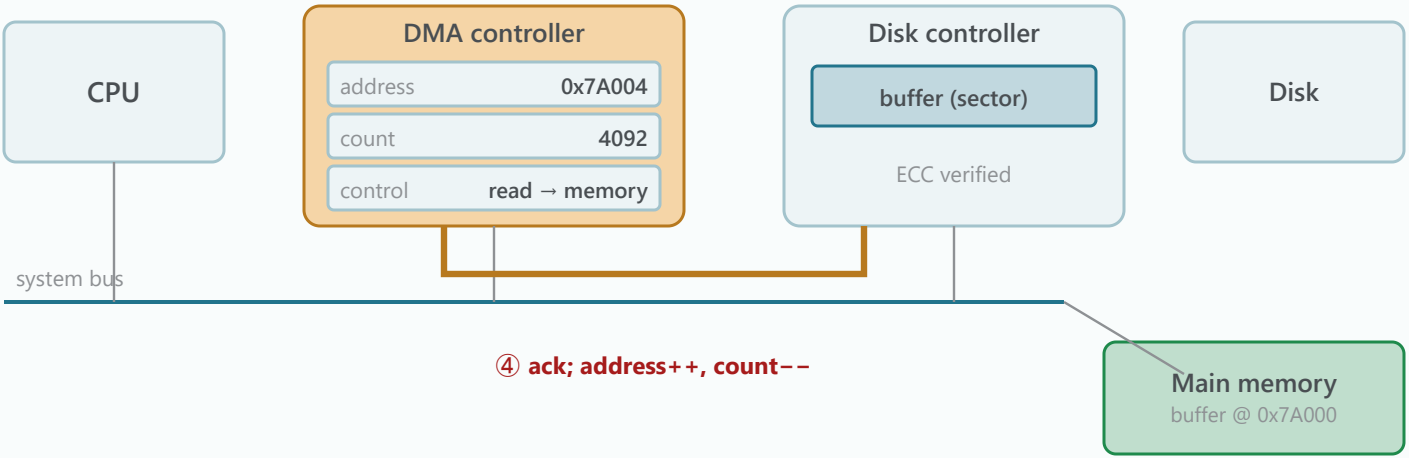
DMA Transfer Step by Step



③ The disk controller writes the word **directly to memory**; the CPU is not involved on the bus.

Cycle stealing: DMA grabs the bus briefly for each word, so the CPU slows a little when it accesses memory. In **burst** mode the bus is held for a whole block. In fly-by mode data goes straight from device to memory; otherwise the DMA controller reads first and then writes.

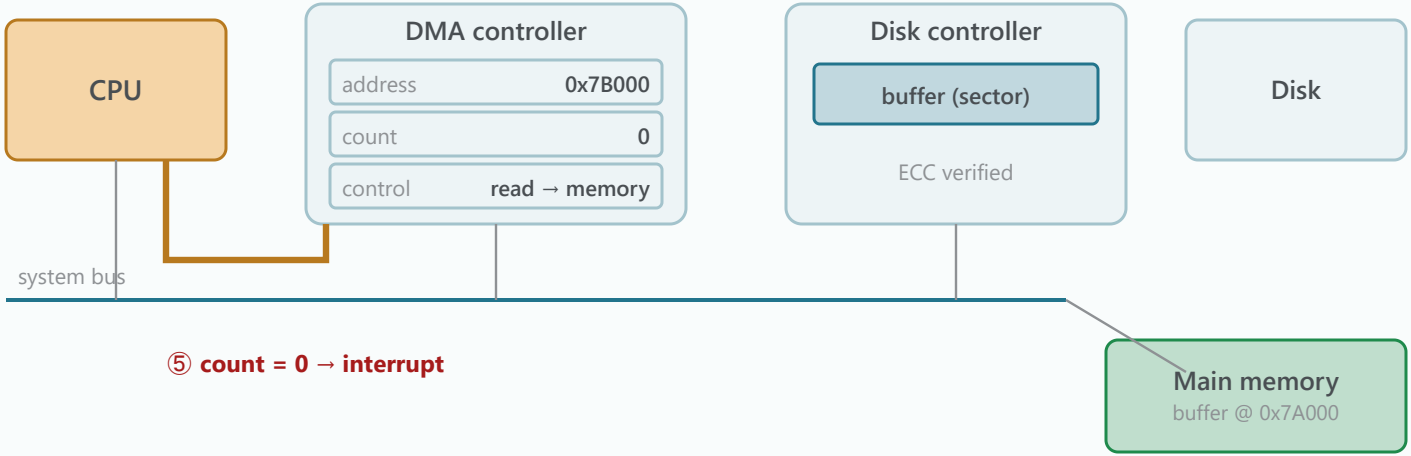
DMA Transfer Step by Step



④ The disk controller sends an acknowledgment; the DMA controller increments the address and decrements the count. Steps ②–④ repeat until the count reaches 0.

Cycle stealing: DMA grabs the bus briefly for each word, so the CPU slows a little when it accesses memory. In **burst** mode the bus is held for a whole block. In fly-by mode data goes straight from device to memory; otherwise the DMA controller reads first and then writes.

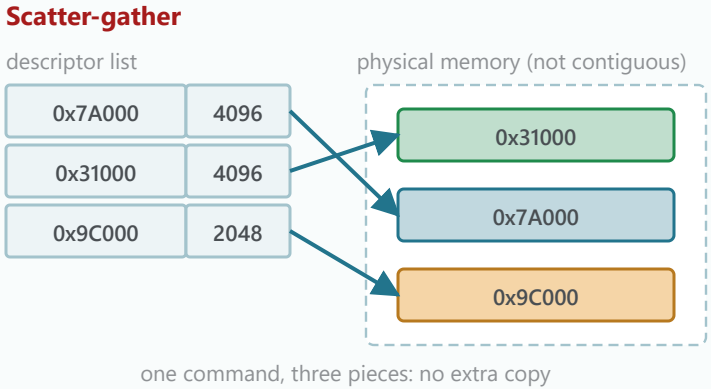
DMA Transfer Step by Step

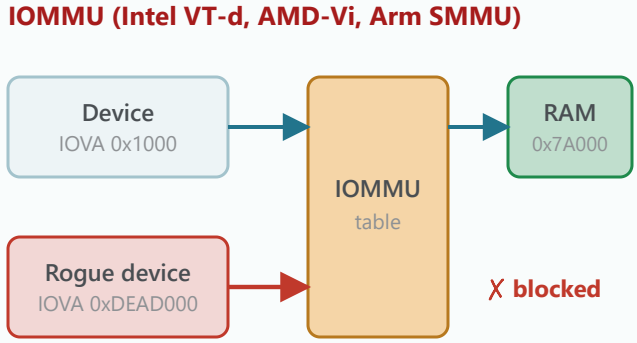


⑤ Count is 0: the DMA controller **interrupts** the CPU. By the time the OS looks, the data is already in memory.

Cycle stealing: DMA grabs the bus briefly for each word, so the CPU slows a little when it accesses memory. In **burst** mode the bus is held for a whole block. In fly-by mode data goes straight from device to memory; otherwise the DMA controller reads first and then writes.

Scatter-Gather and the IOMMU





Scatter-gather

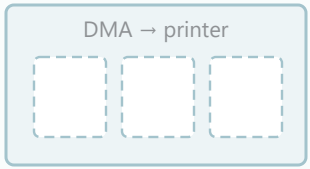
The device gets a list of (address, length) pairs and writes to noncontiguous pages in one operation (`readv`, network packets, NVMe PRP/SGL).

IOMMU

Translates the device-visible address (IOVA) to a physical one through a page table: *protection* (Thunderbolt DMA attacks), assigning devices to VMs (VFIO), high memory for 32-bit devices.

I/O Using DMA: Printer Example

```
1 /* system call */
▶ copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
4 scheduler();     /* block process */
5
6 /* interrupt service routine */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



process: running

CPU: process

The data is copied into a kernel buffer.

Step 1 / 6

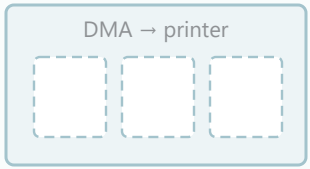
VARIABLES

`DMA count = –` `irq = 0`

DMA frees the CPU from I/O, but the DMA controller is usually slower than the CPU. If the device is very slow or the CPU has nothing else to do, interrupt-driven I/O or even PIO can be simpler and good enough (Tanenbaum).

I/O Using DMA: Printer Example

```
1 /* system call */
2 copy_from_user(buffer, p, count);
▶ set_up_DMA_controller();
4 scheduler();      /* block process */
5
6 /* interrupt service routine */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



process: running

CPU: process

The DMA controller is programmed: buffer address, count = 3, target printer port.

Step 2 / 6

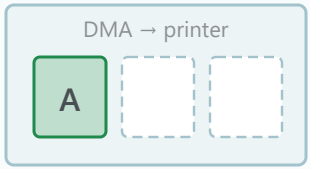
VARIABLES

`DMA count = 3` `irq = 0`

DMA frees the CPU from I/O, but the DMA controller is usually slower than the CPU. If the device is very slow or the CPU has nothing else to do, interrupt-driven I/O or even PIO can be simpler and good enough (Tanenbaum).

I/O Using DMA: Printer Example

```
1 /* system call */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
▶ scheduler();     /* block process */
5
6 /* interrupt service routine */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



process: blocked

CPU: other process

The process blocks; the CPU does other work. DMA feeds the characters to the printer **without the CPU**.

Step 3 / 6

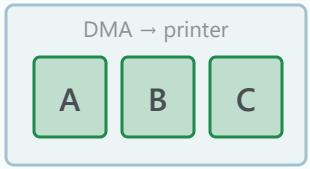
VARIABLES

`DMA count = 2` `irq = 0`

DMA frees the CPU from I/O, but the DMA controller is usually slower than the CPU. If the device is very slow or the CPU has nothing else to do, interrupt-driven I/O or even PIO can be simpler and good enough (Tanenbaum).

I/O Using DMA: Printer Example

```
1 /* system call */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
▶ scheduler();     /* block process */
5
6 /* interrupt service routine */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



process: blocked

CPU: other process

All characters transferred, and the CPU saw no interrupt along the way.

Step 4 / 6

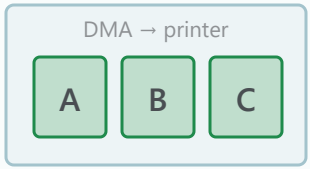
VARIABLES

`DMA count = 0` `irq = 0`

DMA frees the CPU from I/O, but the DMA controller is usually slower than the CPU. If the device is very slow or the CPU has nothing else to do, interrupt-driven I/O or even PIO can be simpler and good enough (Tanenbaum).

I/O Using DMA: Printer Example

```
1 /* system call */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
4 scheduler();     /* block process */
5
6 /* interrupt service routine */
▶ acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



process: blocked

CPU: ISR

DMA raises a single interrupt; the handler acknowledges it.

Step 5 / 6

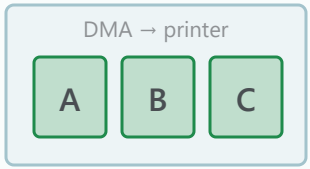
VARIABLES

`DMA count = 0` `irq = 1`

DMA frees the CPU from I/O, but the DMA controller is usually slower than the CPU. If the device is very slow or the CPU has nothing else to do, interrupt-driven I/O or even PIO can be simpler and good enough (Tanenbaum).

I/O Using DMA: Printer Example

```
1 /* system call */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
4 scheduler();     /* block process */
5
6 /* interrupt service routine */
7 acknowledge_interrupt();
▶ unblock_user();
▶ return_from_interrupt();
```



process: ready

CPU: ISR

The user process is woken and the handler returns. **One interrupt per buffer**, not per character.

Step 6 / 6

VARIABLES

`DMA count = 0` `irq = 1`

DMA frees the CPU from I/O, but the DMA controller is usually slower than the CPU. If the device is very slow or the CPU has nothing else to do, interrupt-driven I/O or even PIO can be simpler and good enough (Tanenbaum).

Cost of the Three Methods

The printer prints **1000 characters** per second; we print 4096 characters. Handling an interrupt costs **2 µs**, setting up DMA **5 µs**.

Polling: the CPU waits on every character: busy for 4096 × 1 ms = **4.096 s**.

With programmed I/O the CPU is tied to the device's speed.

Cost of the Three Methods

The printer prints **1000 characters** per second; we print 4096 characters. Handling an interrupt costs **2 µs**, setting up DMA **5 µs**.

Polling: the CPU waits on every character: busy for 4096 × 1 ms = **4.096 s**.

Interrupt-driven: 4096 interrupts × 2 µs = **8.19 ms** (0.2% of the job's time).

The waiting is gone; the price is the number of interrupts.

Cost of the Three Methods

The printer prints **1000 characters** per second; we print 4096 characters. Handling an interrupt costs **2 µs**, setting up DMA **5 µs**.

Polling: the CPU waits on every character: busy for 4096 × 1 ms = **4.096 s**.

Interrupt-driven: 4096 interrupts × 2 µs = **8.19 ms** (0.2% of the job's time).

DMA: 5 µs setup + 1 interrupt × 2 µs = **≈ 7 µs**, about 1170 times less.

But DMA steals bus cycles, and its setup does not pay off for small transfers.

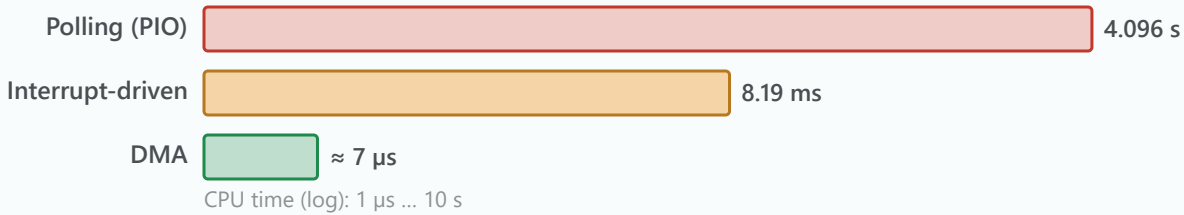
Cost of the Three Methods

The printer prints **1000 characters** per second; we print 4096 characters. Handling an interrupt costs **2 µs**, setting up DMA **5 µs**.

Polling: the CPU waits on every character: busy for 4096 × 1 ms = **4.096 s**.

Interrupt-driven: 4096 interrupts × 2 µs = **8.19 ms** (0.2% of the job's time).

DMA: 5 µs setup + 1 interrupt × 2 µs = **≈ 7 µs**, about 1170 times less.



More than four orders of magnitude apart. With short transfers, slow DMA or an idle CPU, the DMA advantage disappears.

Goals of I/O Software

Device independence

Programs should not care which device they use: `sort < input` works the same from a file, an SSD or a USB stick.

Uniform naming

Names should not depend on the device type: `/dev/nvme0n1`, `/mnt/usb/report.txt`; mounting builds a single tree.

Error handling

Errors should be handled as close to the hardware as possible: the controller corrects with ECC, the driver retries; upper layers hear about it only if they must.

Synchronous / asynchronous

Most hardware transfers are asynchronous (interrupt-driven), yet a *blocking* interface makes programming much easier for the user.

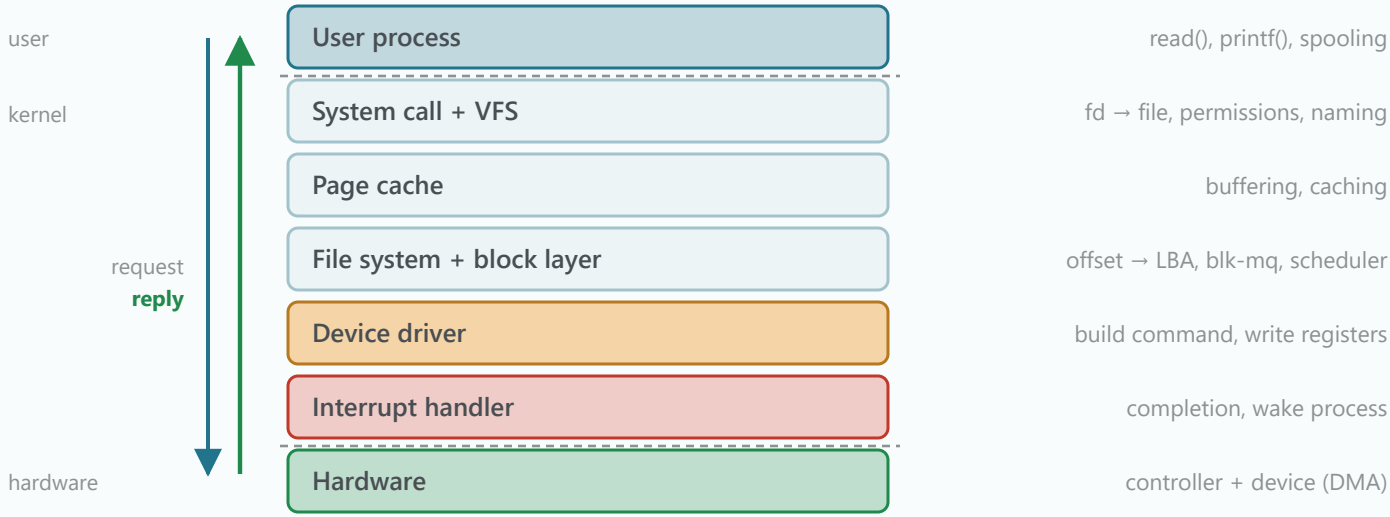
Buffering

An incoming packet must be stored somewhere before we know where it goes; real-time devices such as audio also rely on buffers.

Shared / dedicated

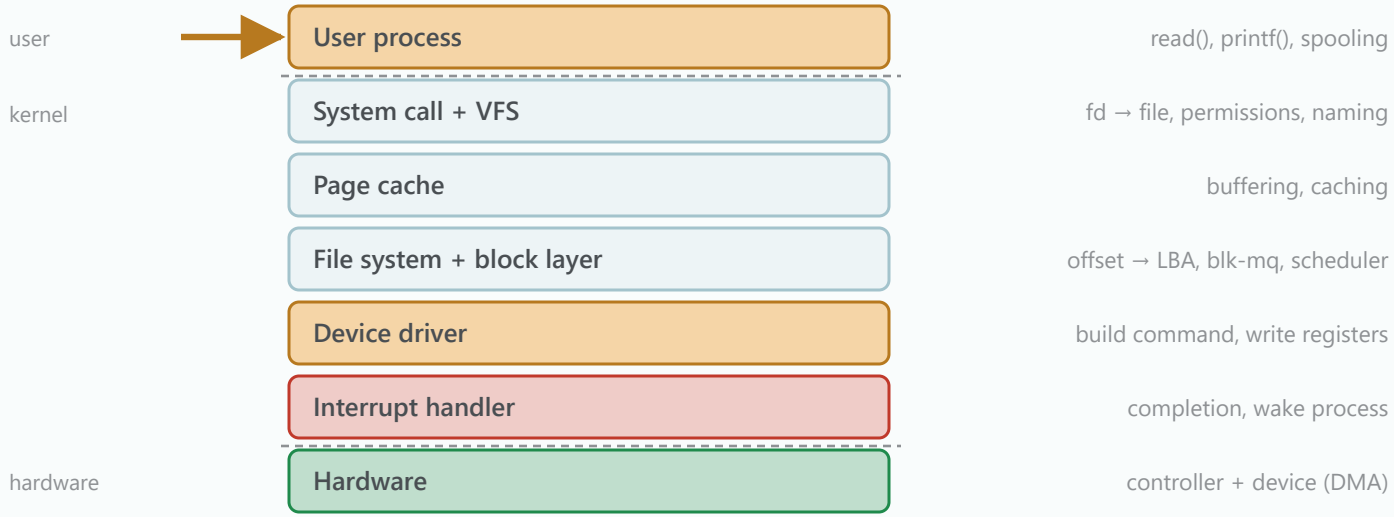
A disk serves many users at once; a printer or scanner is *dedicated* to one user (which opens the door to deadlock).

I/O Software Layers



Each layer has a well-defined job and an interface to the layer above. Tanenbaum counts four layers: user-level I/O software, device-independent software, device drivers and interrupt handlers. In Linux the “device-independent” part splits into the VFS, the page cache, the file system and the block layer.

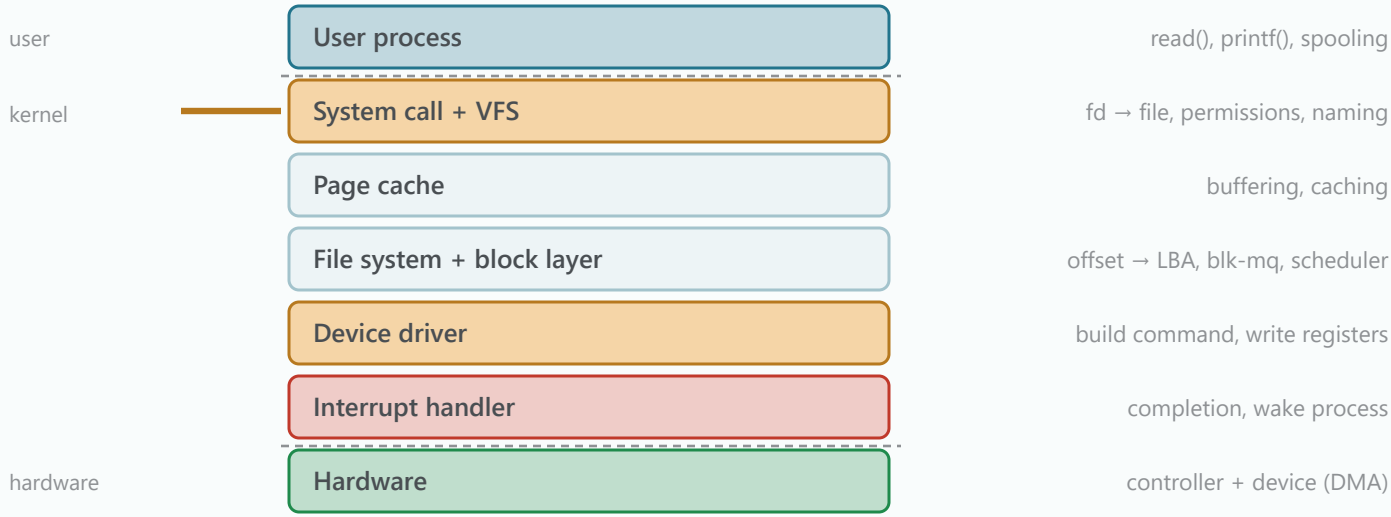
The Path of a read() Call



n = read(fd, buf, 4096)

The process calls `read(fd, buf, 4096)` ; the libc wrapper enters the kernel with the `syscall` instruction.

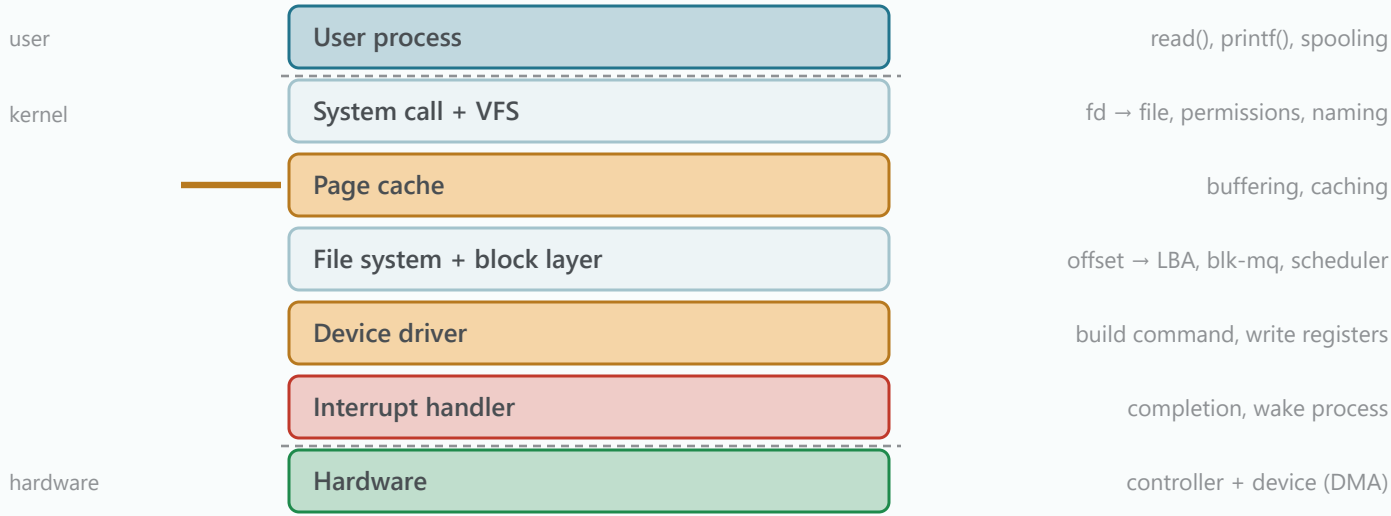
The Path of a read() Call



fd → struct file → inode

The VFS maps the descriptor to the open file and the file to its i-node; permissions were already checked at open time.

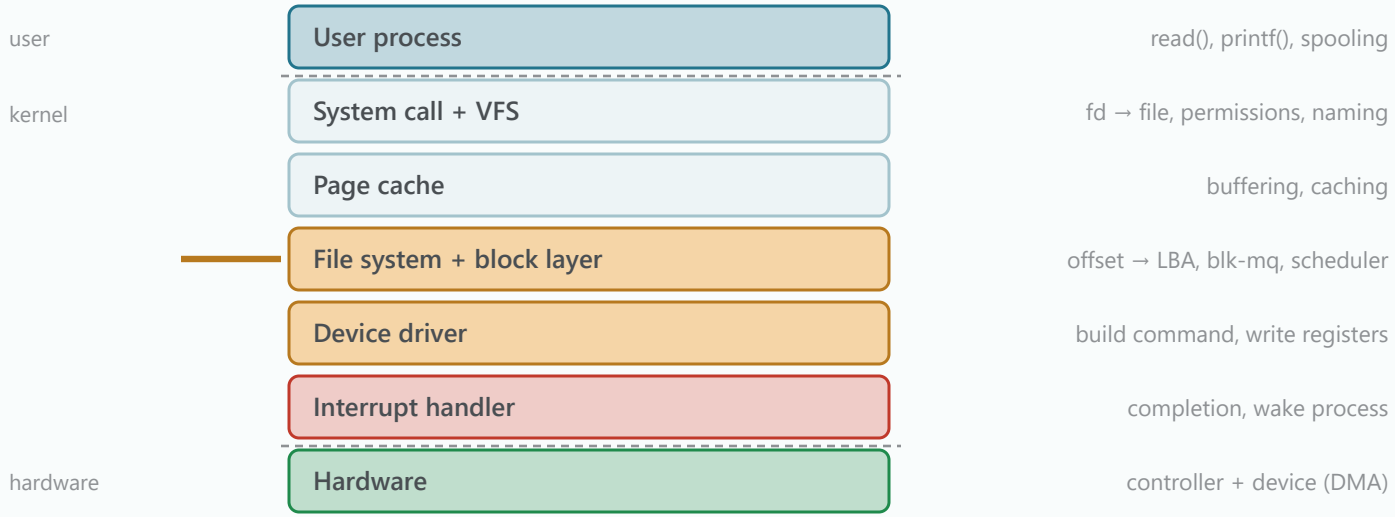
The Path of a read() Call



not cached: miss

The page cache is checked: the page is **not there**. A cache page is allocated and a read is started.

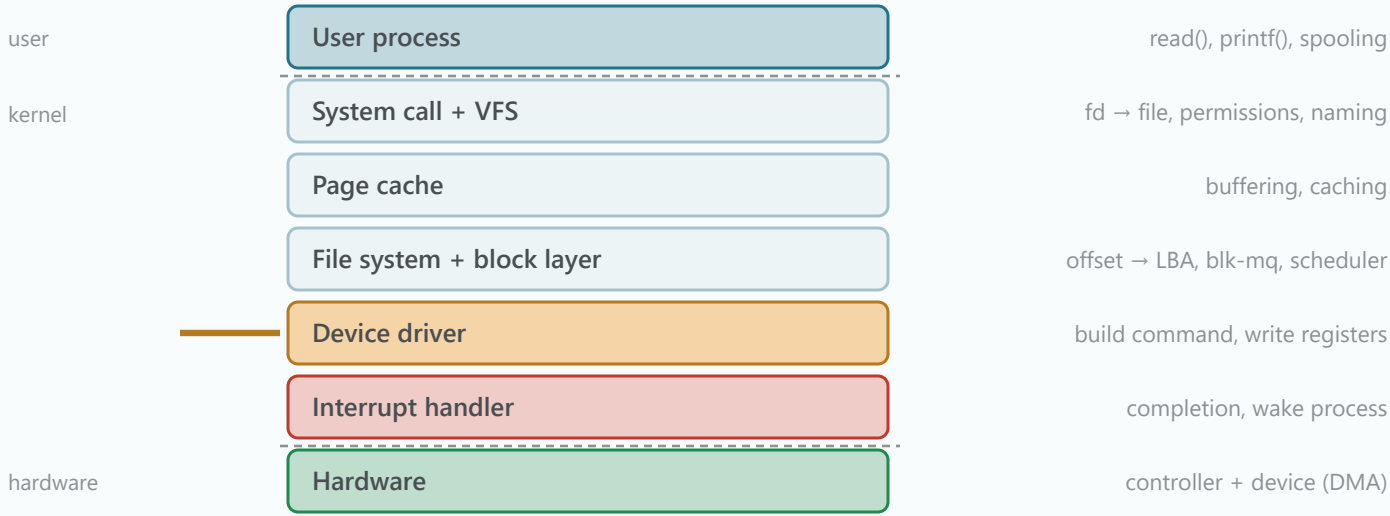
The Path of a read() Call



offset 0 → LBA 2048

The file system maps the file offset to a disk block (LBA); the block layer builds a `bio` /request and hands it to a hardware queue.

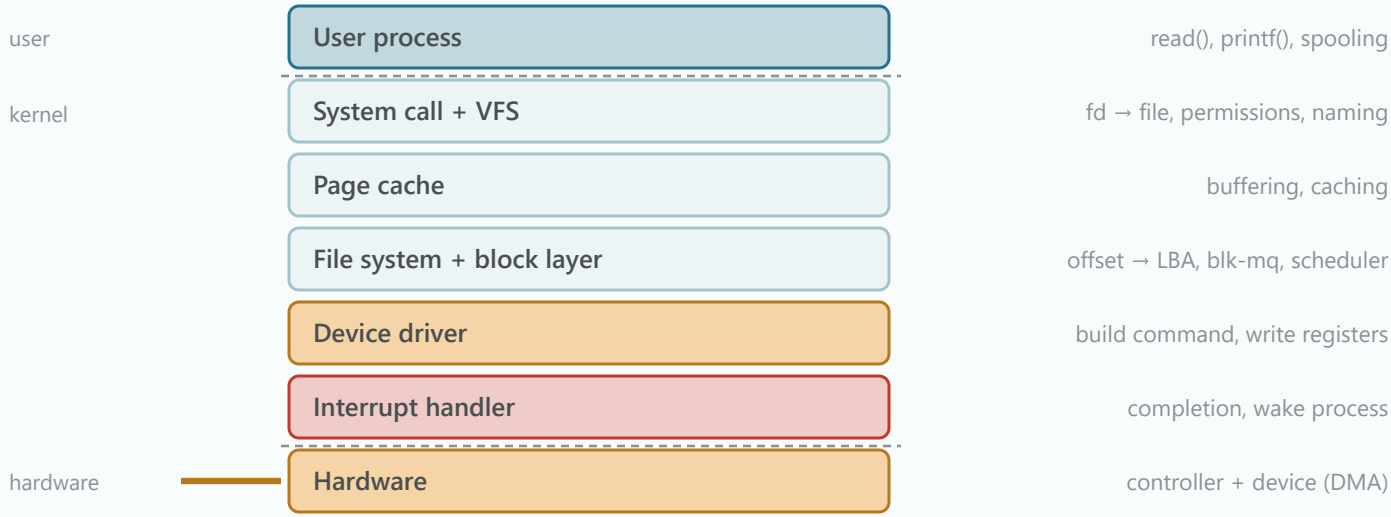
The Path of a read() Call



command to SQ + doorbell

The NVMe driver writes the read command into the submission queue and rings the doorbell register. The process **blocks** (state D).

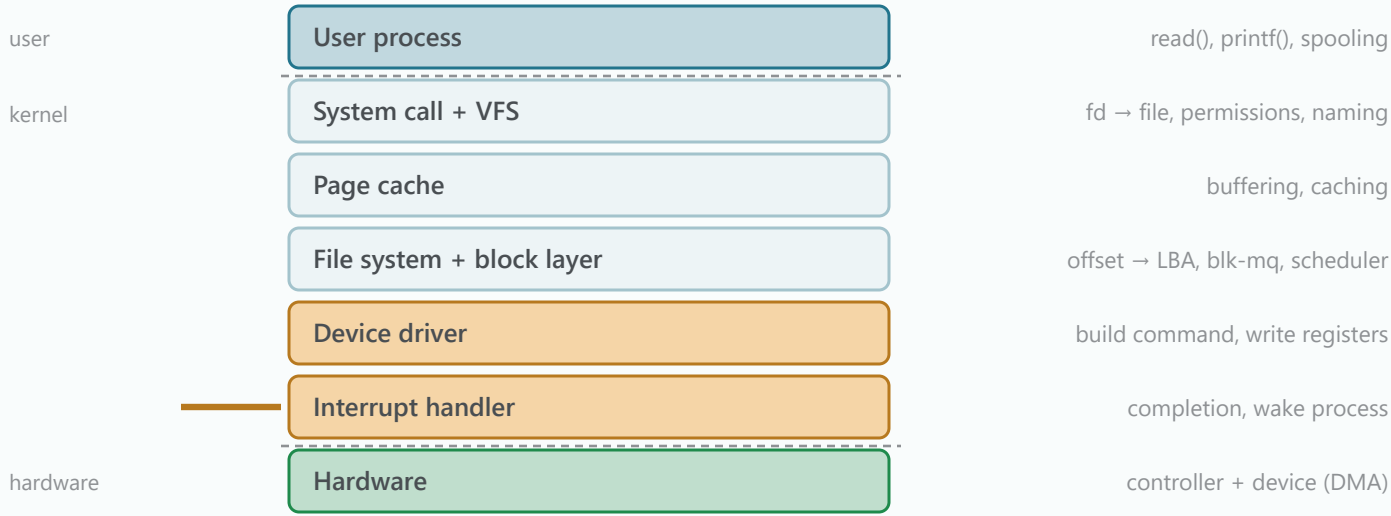
The Path of a read() Call



DMA → cache page, MSI-X

The SSD DMAs the data into the cache page, posts an entry to the completion queue and raises an MSI-X interrupt.

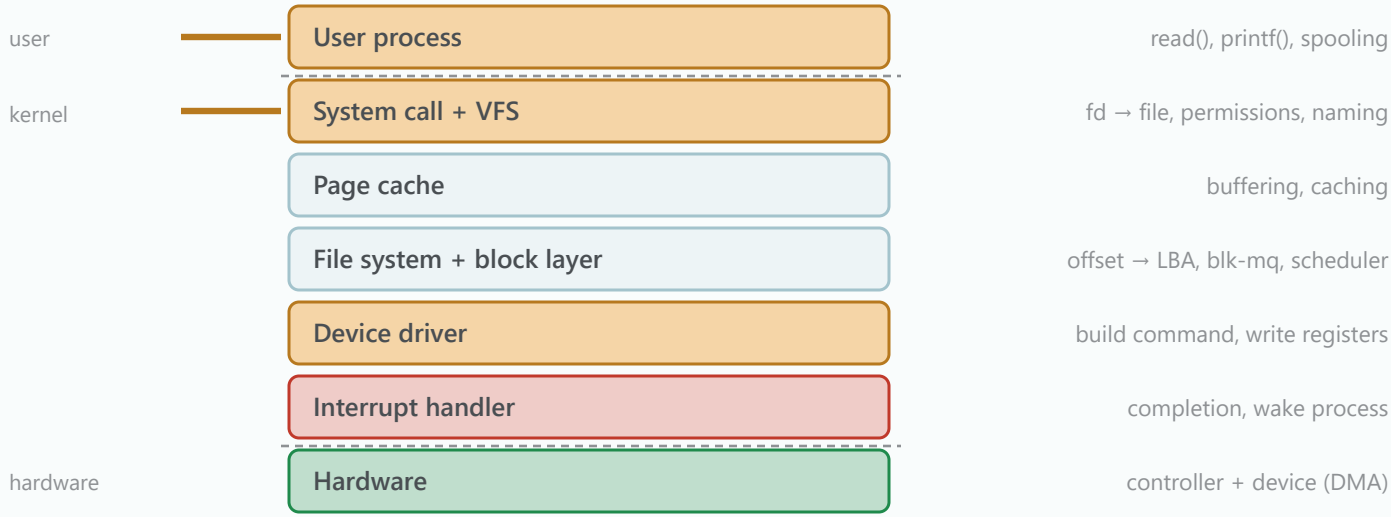
The Path of a read() Call



done: wake process

The interrupt handler processes the completion: the page is marked “up to date” and the waiting process becomes **ready**.

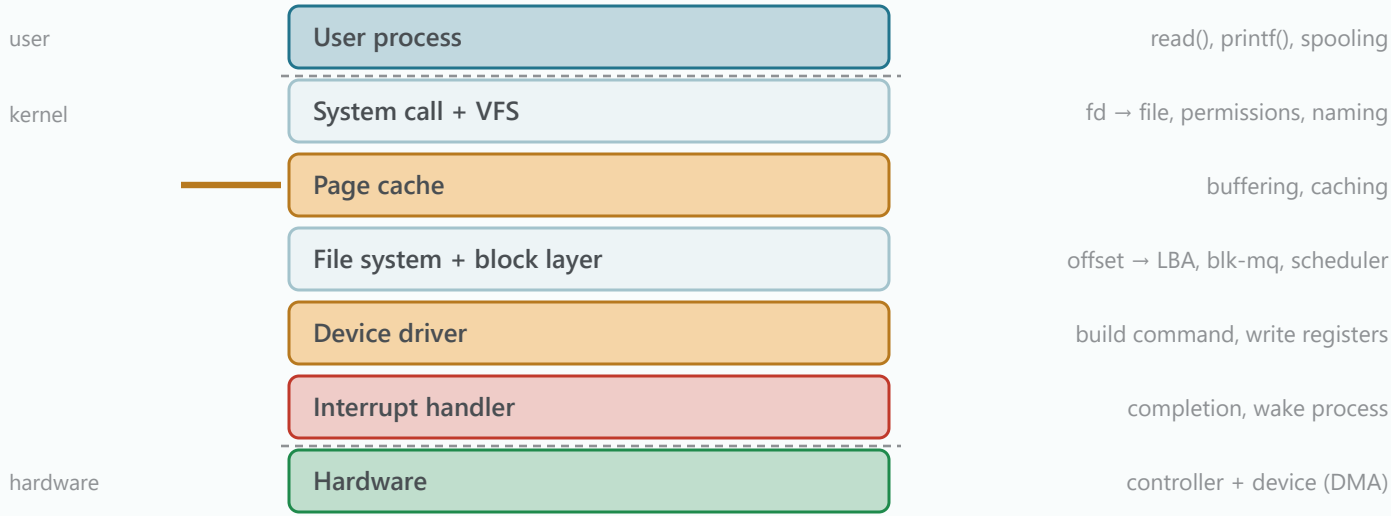
The Path of a read() Call



copy_to_user → n = 4096

When the process runs, the VFS copies the data from the cache into the user buffer and returns 4096.

The Path of a read() Call

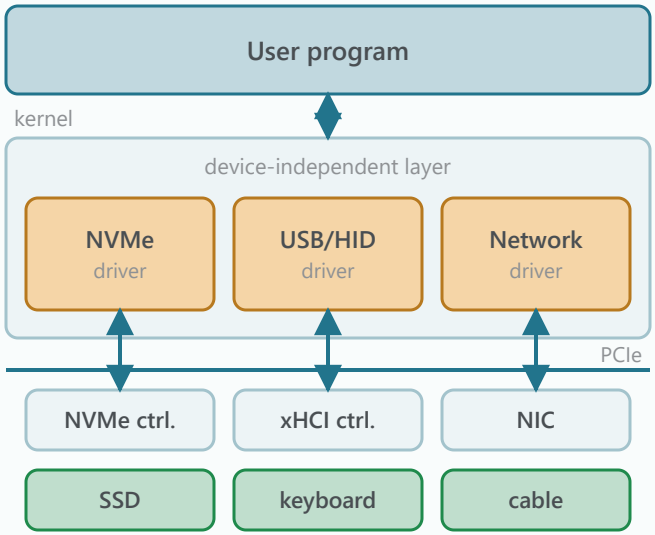


second read: hit

If the same data is read again, it is a page-cache **hit**: the device is never touched and the call takes microseconds.

Device Drivers

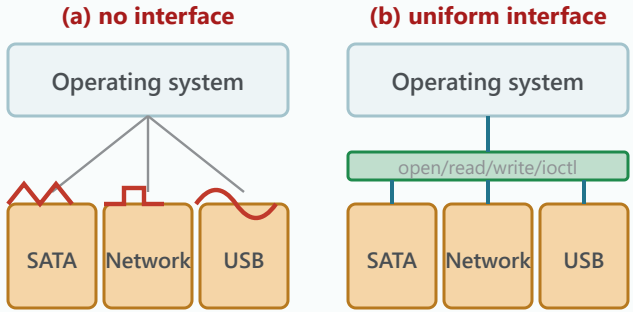
- A **driver** encapsulates device-specific code: only it knows the registers, commands and quirks. Most Linux drivers live in the kernel tree (*in-tree*); some come from vendors.
- Its jobs: validate parameters, turn the request into device commands (LBA, queue entry), initialize the device if needed, write the registers, wait for the interrupt (block), check for errors, return the result.
- It must be reentrant: a new request or interrupt can arrive while it runs. With hot-plug, the device can vanish while in use.
- A buggy driver in the kernel crashes the whole system → **user-space drivers**: MINIX 3, FUSE, VFIO/UIO, DPDK/SPDK; Windows UMDF. Loadable modules: `modprobe`, `lsmod`.



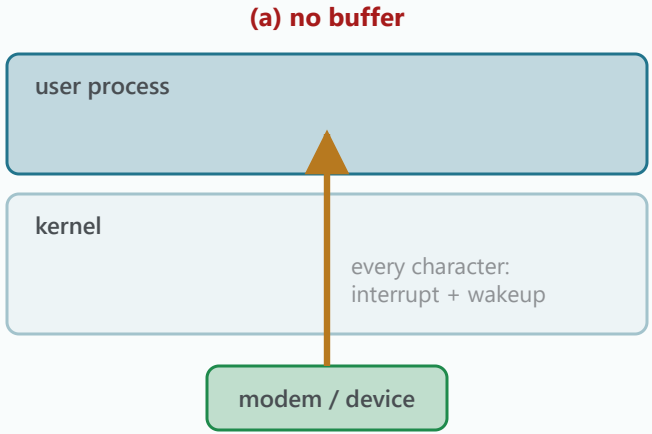
Device-Independent I/O Software

- A **uniform interface** for all drivers: every class offers the same functions, and the kernel calls them through a *table of function pointers*.
- **Naming**: `/dev/nvme0n1` → major number (driver) + minor number (unit). **Protection**: permissions on the device file.
- A device-independent **block size**, buffering, error reporting, and allocating and releasing dedicated devices all live in this layer.

```
struct file_operations nvme_fops = {
  .open           = nvme_open,
  .release        = nvme_release,
  .read_iter      = nvme_read,
  .unlocked_ioctl = nvme_ioctl,
  .poll           = nvme_poll,
};
```



Buffering

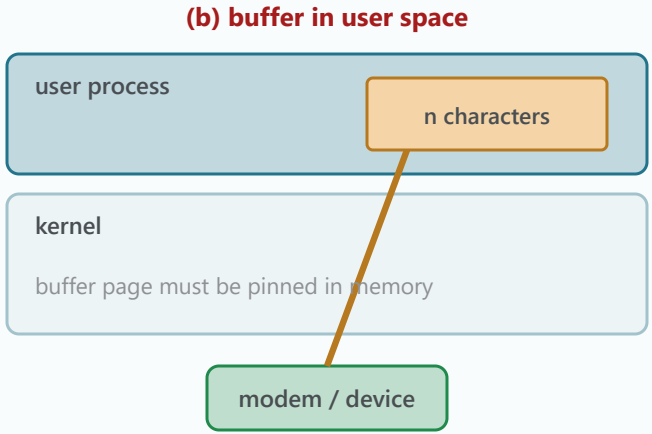


Unbuffered: every character raises an interrupt and wakes the process. Lots of context switches; inefficient.

Step 1 / 5

Three reasons to buffer (Silberschatz): a **speed mismatch** between producer and consumer, a **transfer-size mismatch** (reassembling packets), and **copy semantics** (the user may change its buffer once `write` returns).

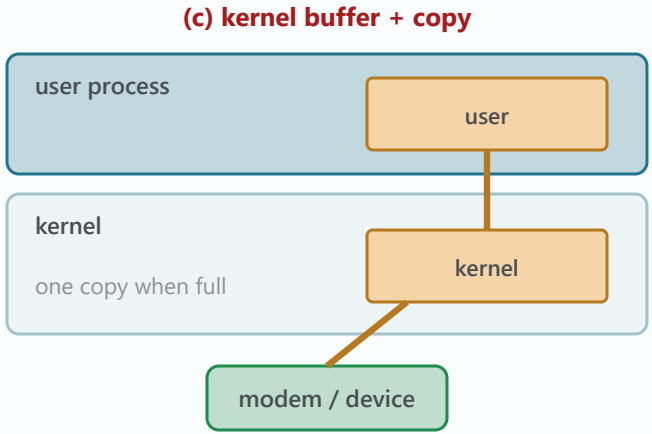
Buffering



User buffer: the interrupt handler writes characters straight into the process's buffer. What if that page is swapped out? It must be pinned, and if many processes pin pages, memory runs short.

Three reasons to buffer (Silberschatz): a **speed mismatch** between producer and consumer, a **transfer-size mismatch** (reassembling packets), and **copy semantics** (the user may change its buffer once `write` returns).

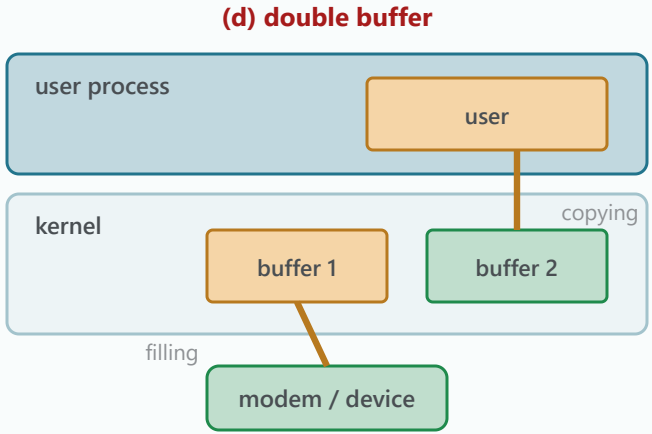
Buffering



Kernel buffer: characters collect in the kernel and are copied to the user in one go when the buffer fills. But where does data go that arrives during the copy?

Three reasons to buffer (Silberschatz): a **speed mismatch** between producer and consumer, a **transfer-size mismatch** (reassembling packets), and **copy semantics** (the user may change its buffer once `write` returns).

Buffering

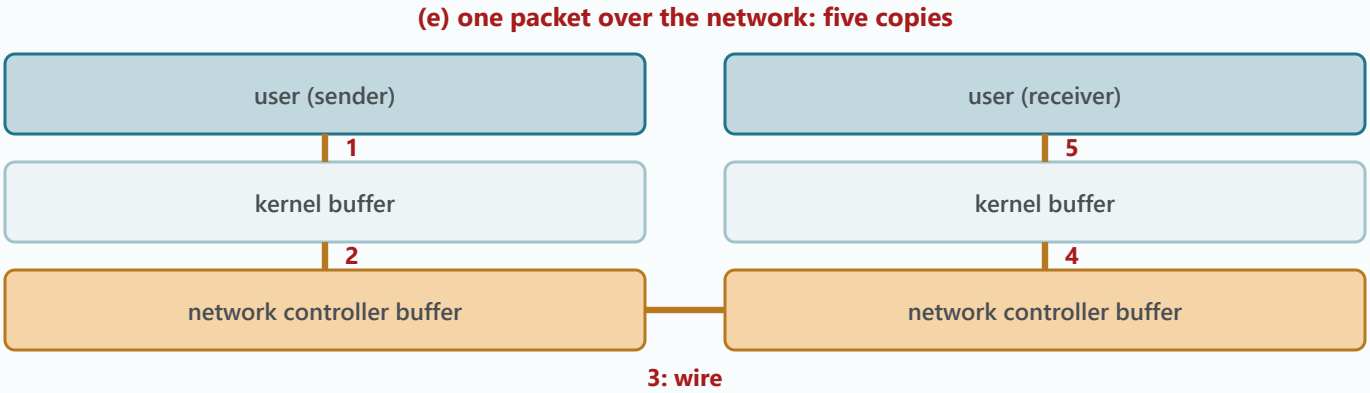


Double buffering: one buffer fills while the other is copied, and the roles alternate. The general form is the **circular buffer**.

Step 4 / 5

Three reasons to buffer (Silberschatz): a **speed mismatch** between producer and consumer, a **transfer-size mismatch** (reassembling packets), and **copy semantics** (the user may change its buffer once `write` returns).

Buffering

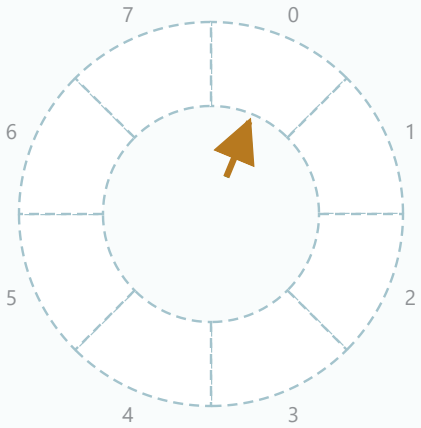


zero copy: sendfile, splice, MSG_ZEROCOPY, io_uring

A packet may be copied 5 times on its way across the network; each copy costs memory bandwidth and latency. Hence zero-copy techniques.

Three reasons to buffer (Silberschatz): a **speed mismatch** between producer and consumer, a **transfer-size mismatch** (reassembling packets), and **copy semantics** (the user may change its buffer once `write` returns).

Circular Buffer



head = 0 tail = 0

used slots: 0 / 8

data (producer: ISR / DMA)

tail: next slot to write

head: next slot to read

start: buffer empty

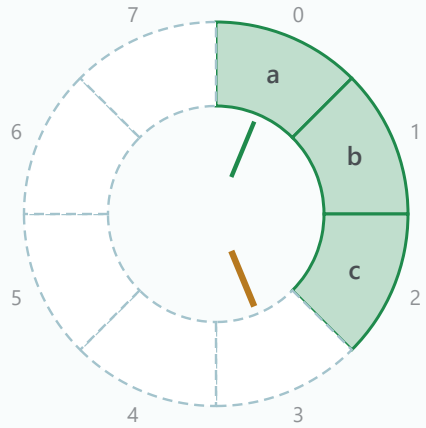
empty: head == tail full: (tail + 1) % 8 == head

Head and tail point to the same slot: the buffer is **empty**.

Step 1 / 5

Circular buffers are everywhere: keyboard and serial drivers, NIC receive/transmit rings, NVMe submission/completion queues, `io_uring`. With a single producer and a single consumer no lock is needed, only memory barriers.

Circular Buffer



head = 0 tail = 3

used slots: 3 / 8

data (producer: ISR / DMA)

tail: next slot to write

head: next slot to read

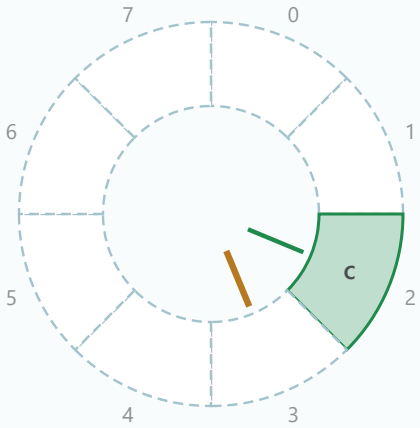
producer wrote 3 bytes

empty: head == tail full: (tail + 1) % 8 == head

The interrupt handler writes 3 incoming bytes at the tail, and the tail advances. The consumer has not read yet.

Circular buffers are everywhere: keyboard and serial drivers, NIC receive/transmit rings, NVMe submission/completion queues, `io_uring`. With a single producer and a single consumer no lock is needed, only memory barriers.

Circular Buffer



head = 2 tail = 3

used slots: 1 / 8

data (producer: ISR / DMA)

tail: next slot to write

head: next slot to read

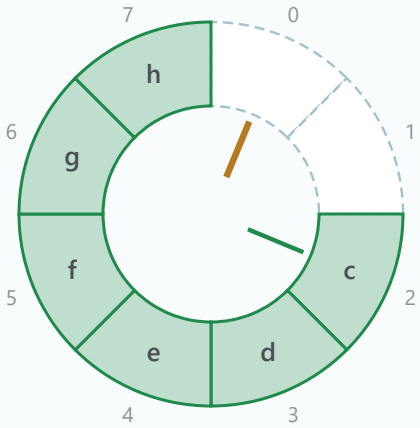
consumer read 2 bytes

empty: head == tail full: (tail + 1) % 8 == head

The process reads 2 bytes and the head advances. Slots that were read can be reused.

Circular buffers are everywhere: keyboard and serial drivers, NIC receive/transmit rings, NVMe submission/completion queues, `io_uring`. With a single producer and a single consumer no lock is needed, only memory barriers.

Circular Buffer



head = 2 tail = 0

used slots: 6 / 8

data (producer: ISR / DMA)

tail: next slot to write

head: next slot to read

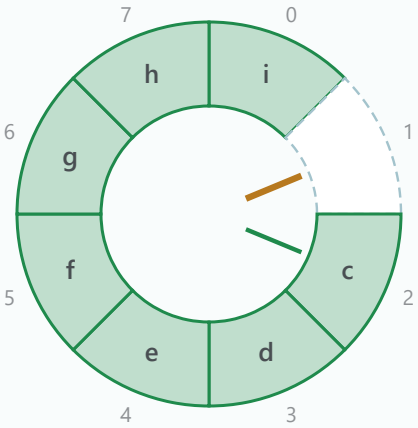
wrap: tail 7 → 0

empty: head == tail full: (tail + 1) % 8 == head

The producer writes 5 more bytes; at the end the tail **wraps around** (mod 8): tail = 0.

Circular buffers are everywhere: keyboard and serial drivers, NIC receive/transmit rings, NVMe submission/completion queues, `io_uring`. With a single producer and a single consumer no lock is needed, only memory barriers.

Circular Buffer



head = 2 tail = 1

used slots: 7 / 8

data (producer: ISR / DMA)

tail: next slot to write

head: next slot to read

one more byte: buffer full

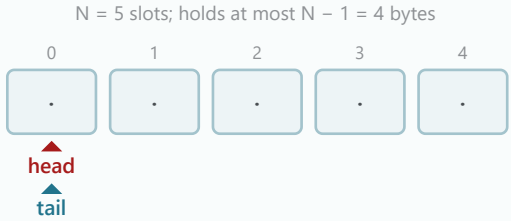
empty: head == tail full: (tail + 1) % 8 == head

After 'i', tail = 1 and (tail + 1) mod 8 == head: the buffer is **full**. Leaving one slot empty tells full from empty. When full, the producer waits or drops the data (lost network packets).

Circular buffers are everywhere: keyboard and serial drivers, NIC receive/transmit rings, NVMe submission/completion queues, `io_uring`. With a single producer and a single consumer no lock is needed, only memory barriers.

Circular Buffer: Code Trace

```
▶ int put(char c) {          /* producer */
2   int s = (tail + 1) % N;
3   if (s == head) return -1; /* full */
4   buf[tail] = c;
5   tail = s;
6   return 0;
7 }
8 int get(char *c) {          /* consumer */
9   if (head == tail) return -1; /* empty */
10  *c = buf[head];
11  head = (head + 1) % N;
12  return 0;
13 }
```



Buffer empty: head = tail = 0. The interrupt handler writes with `put`, the process reads with `get`; each changes only its own index.

Step 1 / 9

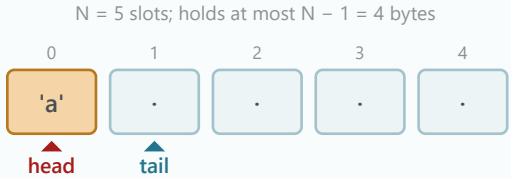
VARIABLES

`op = –` `head = 0` `tail = 0` `used = 0`

OUTPUT

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
2    int s = (tail + 1) % N;
3    if (s == head) return -1; /* full */
4    buf[tail] = c;
5    tail = s;
6    return 0;
7  }
8  int get(char *c) {         /* consumer */
9    if (head == tail) return -1; /* empty */
10   *c = buf[head];
11   head = (head + 1) % N;
12   return 0;
13 }
```



`put('a')` : buf[0] = 'a', tail = 1.

Step 2 / 9

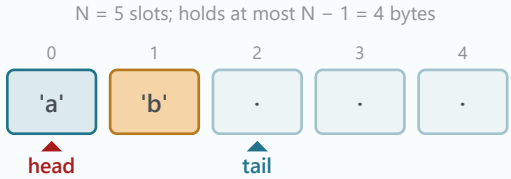
VARIABLES

`op = put('a')` `head = 0` `tail = 1` `used = 1`

OUTPUT

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
   int s = (tail + 1) % N;
3  if (s == head) return -1; /* full */
   buf[tail] = c;
   tail = s;
6  return 0;
7  }
8  int get(char *c) {         /* consumer */
9  if (head == tail) return -1; /* empty */
10 *c = buf[head];
11 head = (head + 1) % N;
12 return 0;
13 }
```



`put('b')` : buf[1] = 'b', tail = 2.

Step 3 / 9

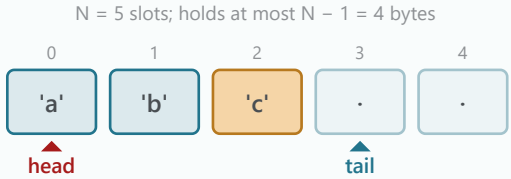
VARIABLES

`op = put('b')` `head = 0` `tail = 2` `used = 2`

OUTPUT

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
2    int s = (tail + 1) % N;
3    if (s == head) return -1; /* full */
4    buf[tail] = c;
5    tail = s;
6    return 0;
7  }
8  int get(char *c) {         /* consumer */
9    if (head == tail) return -1; /* empty */
10   *c = buf[head];
11   head = (head + 1) % N;
12   return 0;
13 }
```



`put('c')` : buf[2] = 'c', tail = 3.

Step 4 / 9

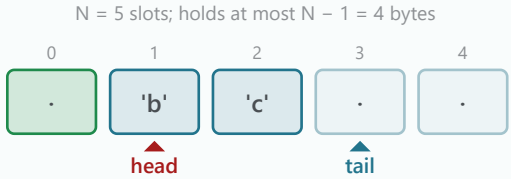
VARIABLES

op = put('c') | head = 0 | tail = 3 | used = 3

OUTPUT

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
2    int s = (tail + 1) % N;
3    if (s == head) return -1; /* full */
4    buf[tail] = c;
5    tail = s;
6    return 0;
7  }
8  int get(char *c) {         /* consumer */
▶    if (head == tail) return -1; /* empty */
▶    *c = buf[head];
▶    head = (head + 1) % N;
12   return 0;
13 }
```



`get`: head ≠ tail, so the buffer is not empty. buf[0] = 'a' is read, head = 1. Slot 0 can be reused.

Step 5 / 9

VARIABLES

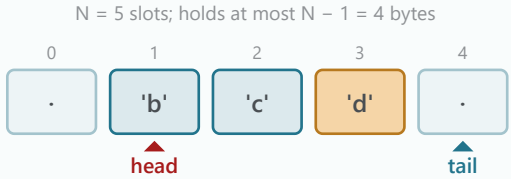
`op = get` `head = 1` `tail = 3` `used = 2`

OUTPUT

```
get: a
```

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
   int s = (tail + 1) % N;
3  if (s == head) return -1; /* full */
   buf[tail] = c;
   tail = s;
6  return 0;
7  }
8  int get(char *c) {         /* consumer */
9  if (head == tail) return -1; /* empty */
10 *c = buf[head];
11 head = (head + 1) % N;
12 return 0;
13 }
```



`put('d')` : buf[3] = 'd', tail = 4.

Step 6 / 9

VARIABLES

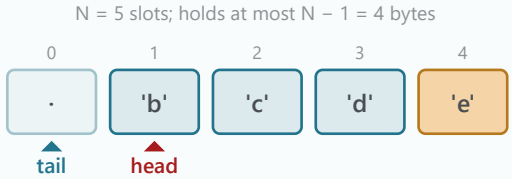
op = put('d') | head = 1 | tail = 4 | used = 3

OUTPUT

```
get: a
```

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
   int s = (tail + 1) % N;
3  if (s == head) return -1; /* full */
   buf[tail] = c;
   tail = s;
6  return 0;
7  }
8  int get(char *c) {         /* consumer */
9  if (head == tail) return -1; /* empty */
10 *c = buf[head];
11 head = (head + 1) % N;
12 return 0;
13 }
```



`put('e')`: buf[4] = 'e', tail = 0. The tail reached the end and **wrapped around** (mod 5).

Step 7 / 9

VARIABLES

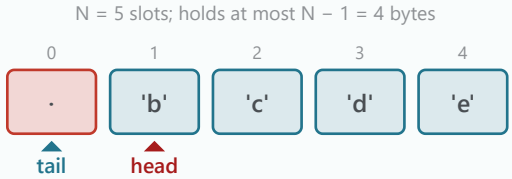
`op = put('e')` `head = 1` `tail = 0` `used = 4`

OUTPUT

```
get: a
```

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
2    int s = (tail + 1) % N;
3    if (s == head) return -1; /* full */
4    buf[tail] = c;
5    tail = s;
6    return 0;
7  }
8  int get(char *c) {         /* consumer */
9    if (head == tail) return -1; /* empty */
10   *c = buf[head];
11   head = (head + 1) % N;
12   return 0;
13 }
```



(tail + 1) mod 5 = 1 = head → full

`put('f')` : s = (0 + 1) mod 5 = 1 = head. One slot always stays empty; otherwise head == tail would mean both empty and full. The byte is rejected (−1).

Step 8 / 9

VARIABLES

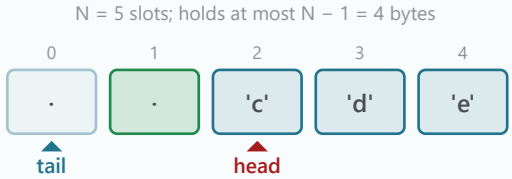
`op = put('f')` `head = 1` `tail = 0` `used = 4`

OUTPUT

```
get: a
put('f'): full
```

Circular Buffer: Code Trace

```
1  int put(char c) {          /* producer */
2    int s = (tail + 1) % N;
3    if (s == head) return -1; /* full */
4    buf[tail] = c;
5    tail = s;
6    return 0;
7  }
8  int get(char *c) {         /* consumer */
▶    if (head == tail) return -1; /* empty */
▶    *c = buf[head];
▶    head = (head + 1) % N;
12   return 0;
13 }
```



`get`: head ≠ tail, so the buffer is not empty. buf[1] = 'b' is read, head = 2. Slot 1 can be reused.

Step 9 / 9

VARIABLES

`op = get` `head = 2` `tail = 0` `used = 3`

OUTPUT

```
get: a
put('f'): full
get: b
```

Caching, Spooling and Error Handling

Cache ≠ buffer

A buffer temporarily holds the *only* copy of data; a cache is a fast *copy* of data that lives elsewhere. The Linux page cache combines both; free RAM goes to the cache (`free -h` : buff/cache).

Spooling

A queue for a dedicated device: users drop files into a spool directory and a single daemon (CUPS, `/var/spool/cups`) prints them in turn. No process ever holds the printer.

Error handling

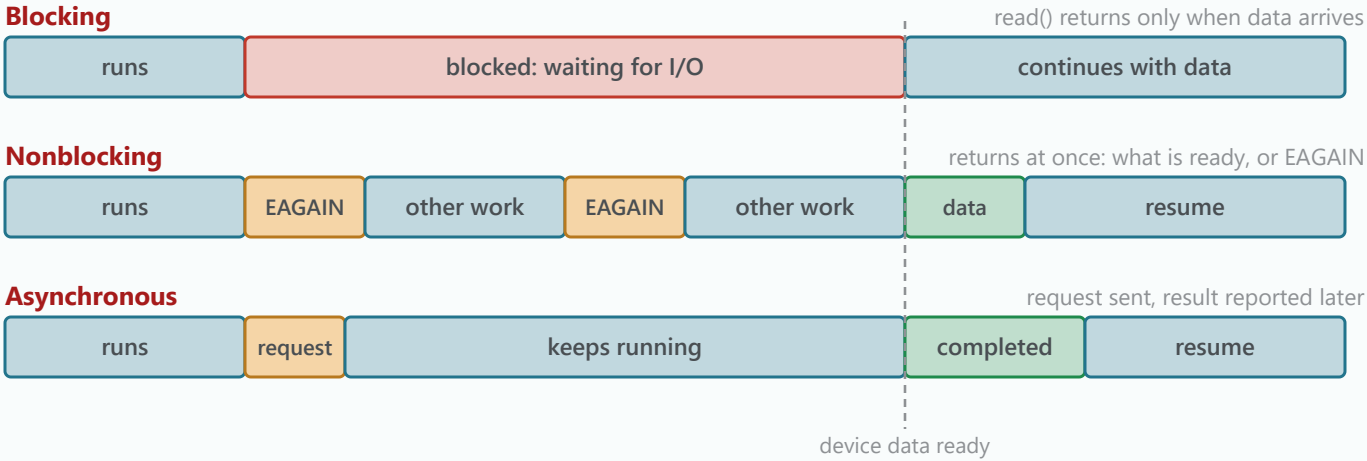
A programming error (bad fd) → an error code (`EBADF` , `EINVAL`). A real I/O error → the driver retries; if it persists, `EIO` and a log entry (`dmesg`).

Allocation and protection

A dedicated device is claimed with `open` ; if busy, the request is queued or `EBUSY` is returned. I/O instructions are privileged; users do I/O only through system calls.

User-space I/O software: library routines such as `printf` / `scanf` format data and buffer it (`stdio` line/block buffering), then call the `write` / `read` system calls.

Blocking, Nonblocking and Asynchronous I/O



Interface	Model	Note
`select` / `poll`	readiness notification	scans the whole fd list on every call: O(n)
`epoll` (Linux), `kqueue` (BSD)	readiness notification	interest list in the kernel; only ready fds return
`io_uring`, Windows IOCP	completion notification	true asynchronous I/O, disks included

Event Loop: epoll

```
▶ int ep = epoll_create1(0);
2 for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3     epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4 for (;;) {
5   int n = epoll_wait(ep, evs, 64, -1);
6   for (int i = 0; i < n; i++) {
7     int fd = evs[i].data.fd;
8     ssize_t r = read(fd, buf, sizeof buf);
9     if (r < 0 && errno == EAGAIN) continue;
10    handle(fd, buf, r);
11  }
12 }
```

interest list (in kernel)

ready list

(empty)

An epoll instance is created: a kernel object holding the interest and ready lists.

Step 1 / 6

VARIABLES

`ep = 3` `n = –` `fd = –` `r = –`

One thread manages tens of thousands of connections (nginx, Node.js, Redis). `epoll` says “you may read”, but `read` still does the work; disk files always look “ready”, so true asynchrony for disks comes with `io_uring`.

Event Loop: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
5    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```



Three sockets (5, 6, 7) are added to the interest list in **nonblocking** mode. This happens once; unlike `select`, they are not passed again every round.

Step 2 / 6

VARIABLES

`ep = 3` `n = –` `fd = –` `r = –`

One thread manages tens of thousands of connections (nginx, Node.js, Redis). `epoll` says “you may read”, but `read` still does the work; disk files always look “ready”, so true asynchrony for disks comes with `io_uring`.

Event Loop: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
▶    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```

interest list (in kernel)

fd 5 | fd 6 | fd 7

ready list

(empty)

`epoll_wait` **blocks** the process while no socket is ready; the CPU does not spin.

Step 3 / 6

VARIABLES

`ep = 3` `n = –` `fd = –` `r = –`

One thread manages tens of thousands of connections (nginx, Node.js, Redis). `epoll` says “you may read”, but `read` still does the work; disk files always look “ready”, so true asynchrony for disks comes with `io_uring`.

Event Loop: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
▶    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```



A packet arrives on socket 7: NIC interrupt → network stack → fd 7 joins the ready list and the process wakes. n = 1.

Step 4 / 6

VARIABLES

`ep = 3` `n = 1` `fd = –` `r = –`

One thread manages tens of thousands of connections (nginx, Node.js, Redis). `epoll` says "you may read", but `read` still does the work; disk files always look "ready", so true asynchrony for disks comes with `io_uring`.

Event Loop: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
5    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
▶      int fd = evs[i].data.fd;
▶      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```



Only the ready fd is handled: 512 bytes are read. Cost grows with the number of ready fds, not the total.

Step 5 / 6

VARIABLES

`ep = 3` `n = 1` `fd = 7` `r = 512`

One thread manages tens of thousands of connections (nginx, Node.js, Redis). `epoll` says "you may read", but `read` still does the work; disk files always look "ready", so true asynchrony for disks comes with `io_uring`.

Event Loop: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
5    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
▶      if (r < 0 && errno == EAGAIN) continue;
▶      handle(fd, buf, r);
11   }
12 }
```

interest list (in kernel)

fd 5 | fd 6 | fd 7

ready list

(empty)

Data handled; when the next `read` returns EAGAIN there is nothing left. The loop goes back to `epoll_wait`.

Step 6 / 6

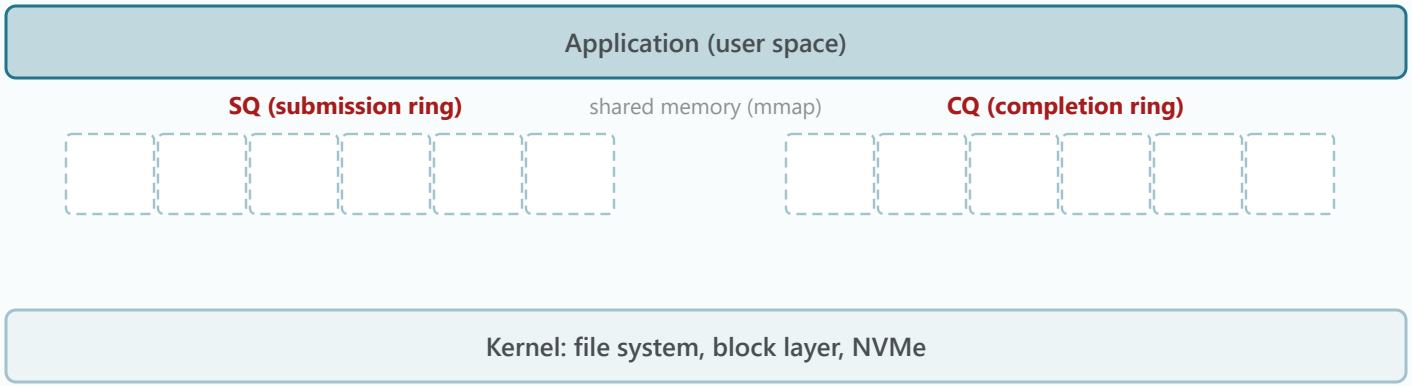
VARIABLES

`ep = 3` `n = 1` `fd = 7`

`r = -1 (EAGAIN)`

One thread manages tens of thousands of connections (nginx, Node.js, Redis). `epoll` says “you may read”, but `read` still does the work; disk files always look “ready”, so true asynchrony for disks comes with `io_uring`.

io_uring: Shared Rings

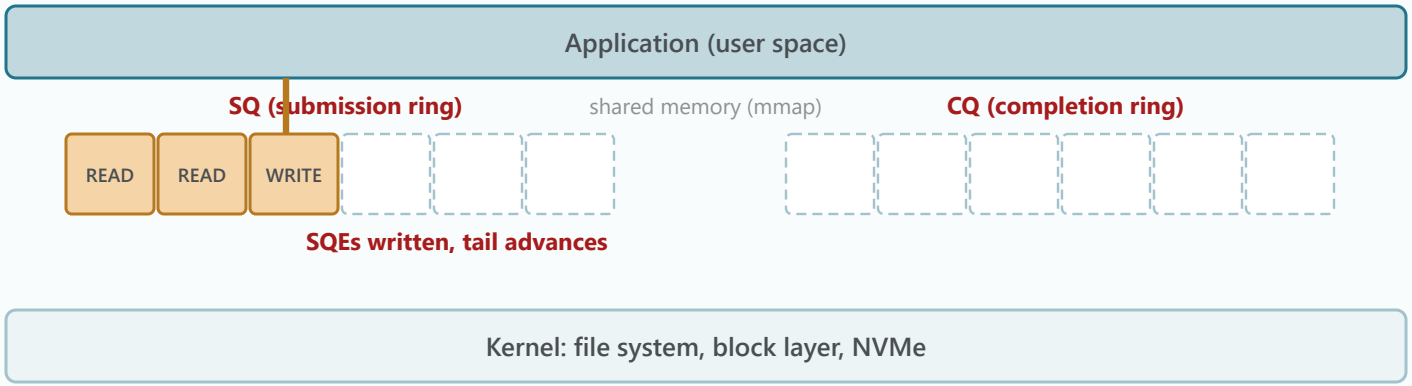


io_uring (Linux 5.1, 2019): the application and the kernel see two rings in **shared memory**; requests and results are never copied.

Step 1 / 5

This is a **completion model** (like Windows IOCP): you are told “your job is done”. Hundreds of operation types exist, including network, file, `accept` and timers. Because of security bugs it is disabled in some environments (e.g. Android, some containers).

io_uring: Shared Rings

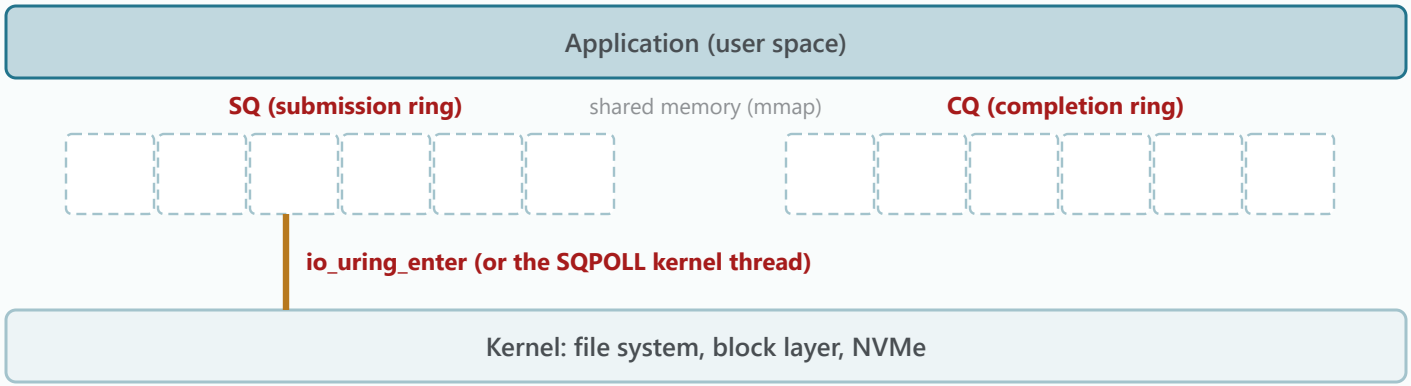


The application writes three submission entries (SQEs): two reads and a write. Each carries an opcode, fd, buffer and offset.

Step 2 / 5

This is a **completion model** (like Windows IOCP): you are told “your job is done”. Hundreds of operation types exist, including network, file, `accept` and timers. Because of security bugs it is disabled in some environments (e.g. Android, some containers).

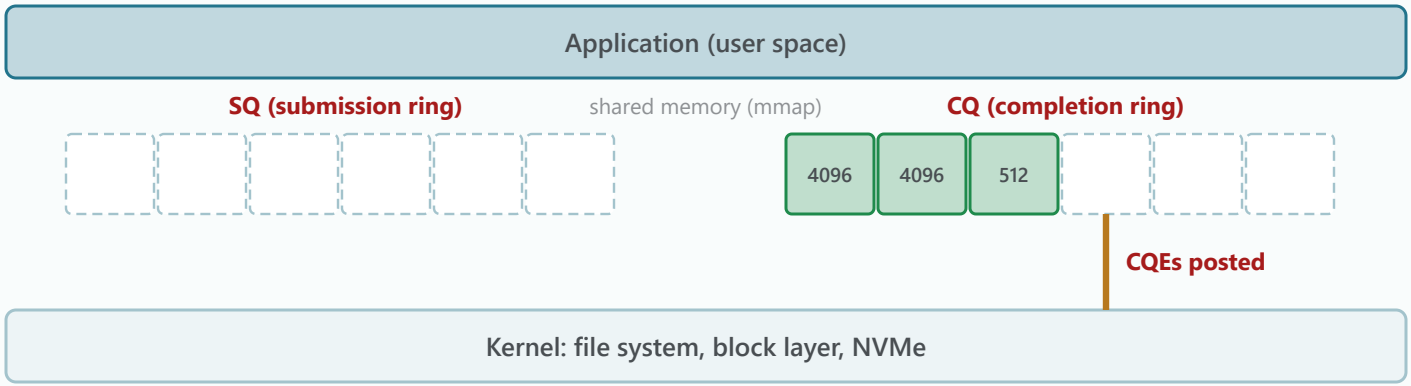
io_uring: Shared Rings



A single `io_uring_enter` call submits all three requests. In SQPOLL mode a kernel thread polls the ring: **no system call** at all.

This is a **completion model** (like Windows IOCP): you are told “your job is done”. Hundreds of operation types exist, including network, file, `accept` and timers. Because of security bugs it is disabled in some environments (e.g. Android, some containers).

io_uring: Shared Rings

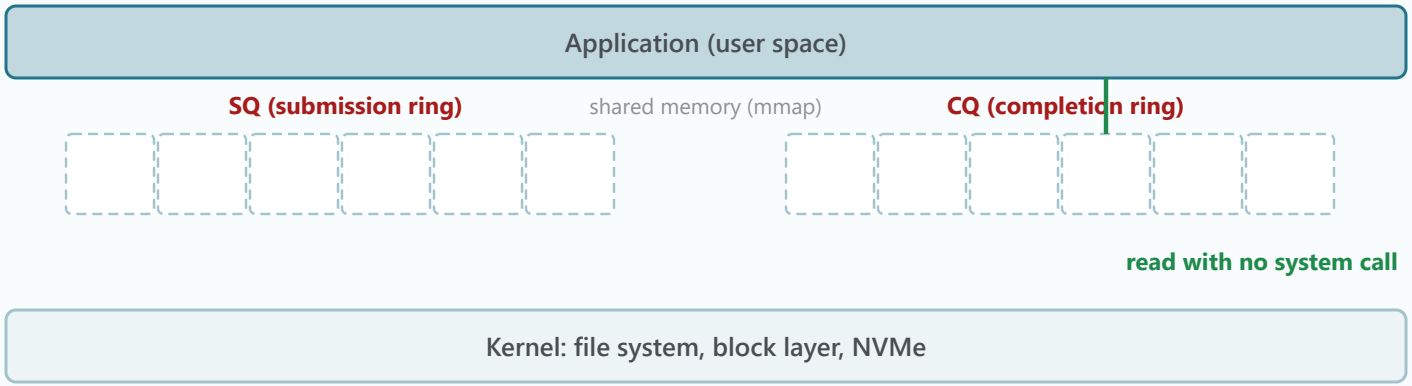


As devices finish, the kernel posts completion entries (CQEs) to the ring; the result field holds bytes transferred or −errno.

Step 4 / 5

This is a **completion model** (like Windows IOCP): you are told “your job is done”. Hundreds of operation types exist, including network, file, `accept` and timers. Because of security bugs it is disabled in some environments (e.g. Android, some containers).

io_uring: Shared Rings



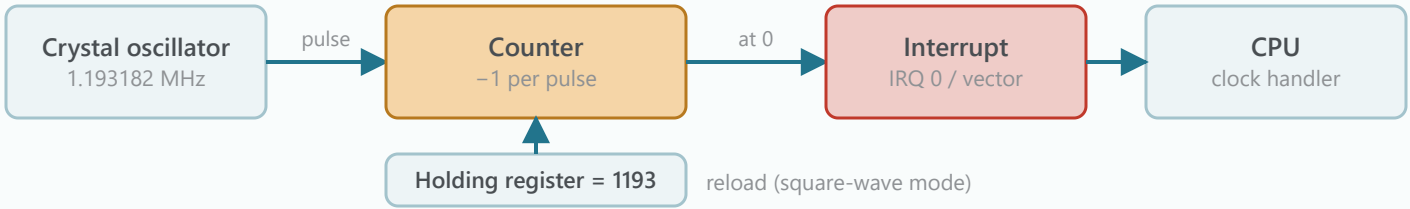
The application reads CQEs straight from memory and advances the head. Batched submission + batched reaping → very few context switches.

Step 5 / 5

This is a **completion model** (like Windows IOCP): you are told “your job is done”. Hundreds of operation types exist, including network, file, `accept` and timers. Because of security bugs it is disabled in some environments (e.g. Android, some containers).

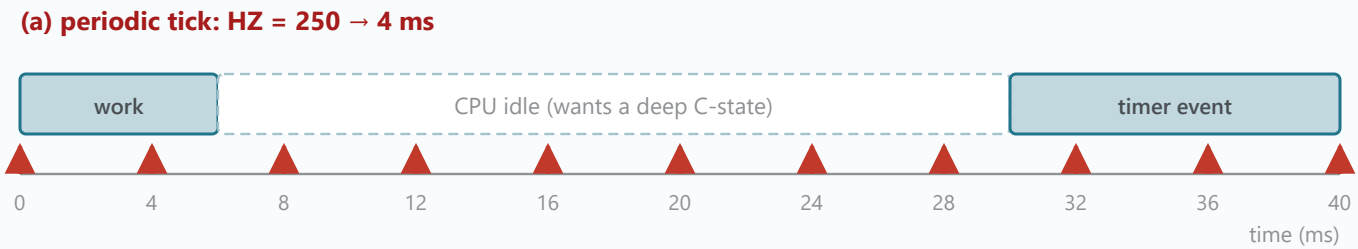
Clock Hardware

modes: one-shot · square-wave (periodic)



- Example (8254 PIT): for a 1 kHz interrupt, counter = 1,193,182 / 1000 ≈ **1193** → actual frequency 1000.15 Hz.
- Modern x86: the **TSC** (time-stamp counter), **HPET** and the **local APIC timer** (TSC-deadline mode); ARM: the generic timer. A battery-backed **RTC** keeps the date while powered off.
- Clock software: time of day, quantum expiry, CPU usage accounting, `alarm` /timers, watchdogs, profiling.
- Keep the time of day in a 64-bit counter: a 32-bit `time_t` overflows on 19 January 2038.

Clock Software: Ticks and Tickless Kernels



6 useless wakeups while idle: each pulls the CPU out of deep sleep

hrtimer: ns-resolution timers kept sorted in a red-black tree

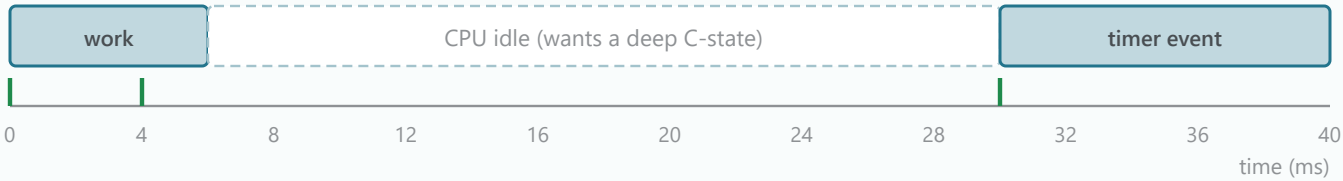
Periodic tick: the timer interrupts every 1/HZ seconds (Linux HZ = 100, 250 or 1000 → 10, 4, 1 ms). The handler advances time, charges the quantum and fires expired timers.

Step 1 / 3

Tickless operation saves energy and CPU time on laptops and VMs. For short (µs) waits, *soft timers* and polling are used together.

Clock Software: Ticks and Tickless Kernels

(b) tickless (NO_HZ): one-shot to the next event



no tick while idle: the timer is programmed once, for 30 ms

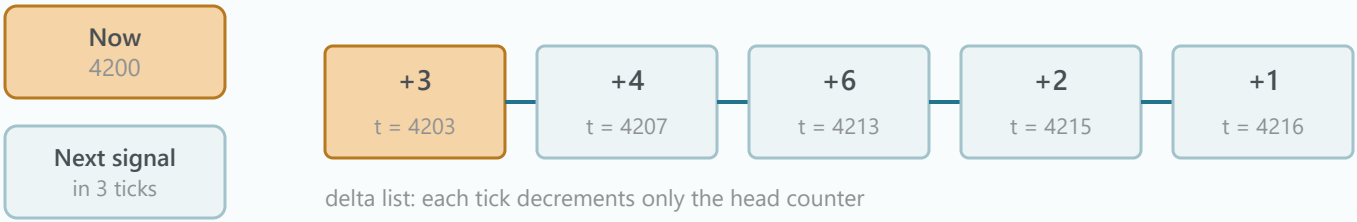
hrtimer: ns-resolution timers kept sorted in a red-black tree

Tickless kernel (NO_HZ_IDLE, 2007): when the CPU is idle the tick stops and the clock is set one-shot to the next event. `nohz_full` removes the tick even while a single process runs on a core (HPC, real time).

Tickless operation saves energy and CPU time on laptops and VMs. For short (µs) waits, *soft timers* and polling are used together.

Clock Software: Ticks and Tickless Kernels

(c) timer list: current time 4200



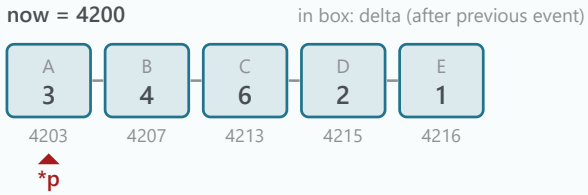
Tanenbaum's example: pending signals at 4203, 4207, 4213, 4215 and 4216. The list stores only the differences; when the head reaches 0, its event fires.

Step 3 / 3

Tickless operation saves energy and CPU time on laptops and VMs. For short (µs) waits, *soft timers* and polling are used together.

Timer List: Code Trace

```
▶ void add(Timer *z, int t) {  /* t: ticks */
▶   Timer **p = &head;
3   while (*p && (*p)->delta <= t) {
4     t -= (*p)->delta;
5     p = &(*p)->next;
6   }
7   z->delta = t; z->next = *p;
8   if (*p) (*p)->delta -= t;
9   *p = z;
10 }
11 void tick(void) {
12   if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



Pending events at 4203, 4207, 4213, 4215, 4216; the list stores only the deltas: 3, 4, 6, 2, 1. New timer X fires 10 ticks from now (4210): `add(X, 10)`.

Step 1 / 8

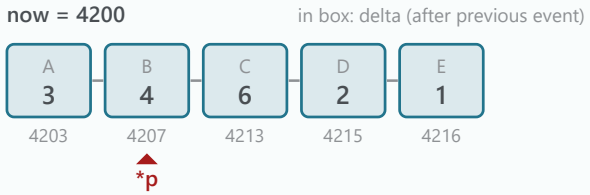
VARIABLES

`now = 4200` `t = 10` `head.delta = 3`

OUTPUT

Timer List: Code Trace

```
1  void add(Timer *z, int t) {  /* t: ticks */
2    Timer **p = &head;
▶    while (*p && (*p)->delta <= t) {
▶      t -= (*p)->delta;
▶      p = &(*p)->next;
6    }
7    z->delta = t; z->next = *p;
8    if (*p) (*p)->delta -= t;
9    *p = z;
10 }
11 void tick(void) {
12   if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



A's delta 3 ≤ t: X goes after it. t = 7, and *p moves to the next node.

Step 2 / 8

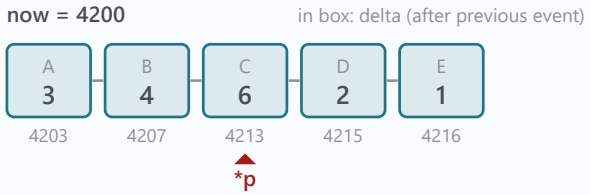
VARIABLES

`now = 4200` `t = 7` `head.delta = 3`

OUTPUT

Timer List: Code Trace

```
1  void add(Timer *z, int t) {  /* t: ticks */
2    Timer **p = &head;
▶    while (*p && (*p)->delta <= t) {
▶      t -= (*p)->delta;
▶      p = &(*p)->next;
6    }
7    z->delta = t; z->next = *p;
8    if (*p) (*p)->delta -= t;
9    *p = z;
10 }
11 void tick(void) {
12   if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



B's delta 4 ≤ t: X goes after it. t = 3, and *p moves to the next node.

Step 3 / 8

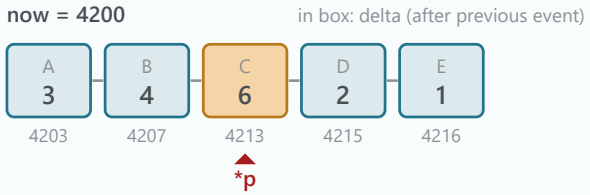
VARIABLES

`now = 4200` `t = 3` `head.delta = 3`

OUTPUT

Timer List: Code Trace

```
1  void add(Timer *z, int t) {  /* t: ticks */
2    Timer **p = &head;
▶    while (*p && (*p)->delta <= t) {
4      t -= (*p)->delta;
5      p = &(*p)->next;
6    }
7    z->delta = t; z->next = *p;
8    if (*p) (*p)->delta -= t;
9    *p = z;
10 }
11 void tick(void) {
12   if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



C's delta 6 > t = 3: X must go **before** C. The loop ends.

Step 4 / 8

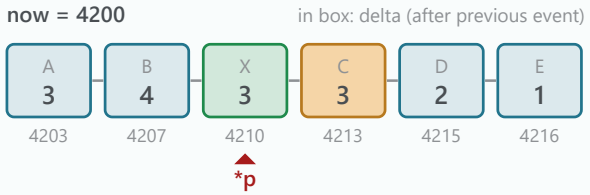
VARIABLES

`now = 4200` `t = 3` `head.delta = 3`

OUTPUT

Timer List: Code Trace

```
1  void add(Timer *z, int t) {  /* t: ticks */
2    Timer **p = &head;
3    while (*p && (*p)->delta <= t) {
4      t -= (*p)->delta;
5      p = &(*p)->next;
6    }
▶    z->delta = t; z->next = *p;
▶    if (*p) (*p)->delta -= t;
▶    *p = z;
10 }
11 void tick(void) {
12   if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



X is spliced in with delta = 3; C's delta becomes 6 − 3 = 3. C's absolute time (4213) is unchanged; only one node was updated.

Step 5 / 8

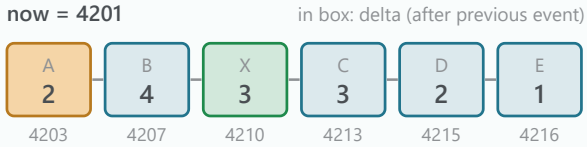
VARIABLES

`now = 4200` `t = 3` `head.delta = 3`

OUTPUT

Timer List: Code Trace

```
1  void add(Timer *z, int t) {  /* t: ticks */
2    Timer **p = &head;
3    while (*p && (*p)->delta <= t) {
4      t -= (*p)->delta;
5      p = &(*p)->next;
6    }
7    z->delta = t; z->next = *p;
8    if (*p) (*p)->delta -= t;
9    *p = z;
10 }
11 void tick(void) {
▶    if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



bottom row: absolute event time

Tick: only the head delta drops → 2.

Step 6 / 8

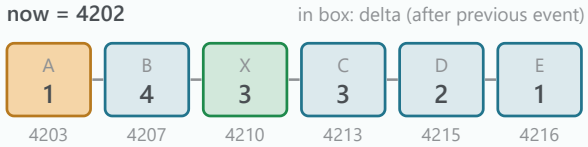
VARIABLES

now = 4201 | t = 3 | head.delta = 2

OUTPUT

Timer List: Code Trace

```
1  void add(Timer *z, int t) {  /* t: ticks */
2    Timer **p = &head;
3    while (*p && (*p)->delta <= t) {
4      t -= (*p)->delta;
5      p = &(*p)->next;
6    }
7    z->delta = t; z->next = *p;
8    if (*p) (*p)->delta -= t;
9    *p = z;
10 }
11 void tick(void) {
▶    if (head) head->delta--;
13   while (head && head->delta == 0)
14     head = run(head);   /* remove */
15 }
```



bottom row: absolute event time

Tick: only the head delta drops → 1.

Step 7 / 8

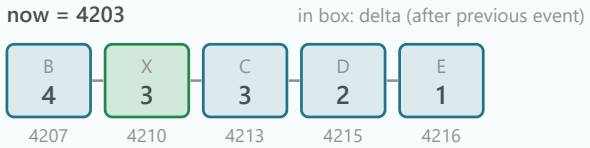
VARIABLES

now = 4202 | t = 3 | head.delta = 1

OUTPUT

Timer List: Code Trace

```
 1 void add(Timer *z, int t) {  /* t: ticks */
 2   Timer **p = &head;
 3   while (*p && (*p)->delta <= t) {
 4     t -= (*p)->delta;
 5     p = &(*p)->next;
 6   }
 7   z->delta = t; z->next = *p;
 8   if (*p) (*p)->delta -= t;
 9   *p = z;
10 }
11 void tick(void) {
▶  if (head) head->delta--;
▶  while (head && head->delta == 0)
▶    head = run(head);   /* remove */
15 }
```



bottom row: absolute event time

4203: the head delta hit 0; A runs and leaves the list. The work per tick is just decrementing the **head** counter: O(1), however long the list.

Step 8 / 8

VARIABLES

now = 4203 | t = 3 | head.delta = 4

OUTPUT

```
4203: A ran
```

Terminals, Input Devices and Power Management

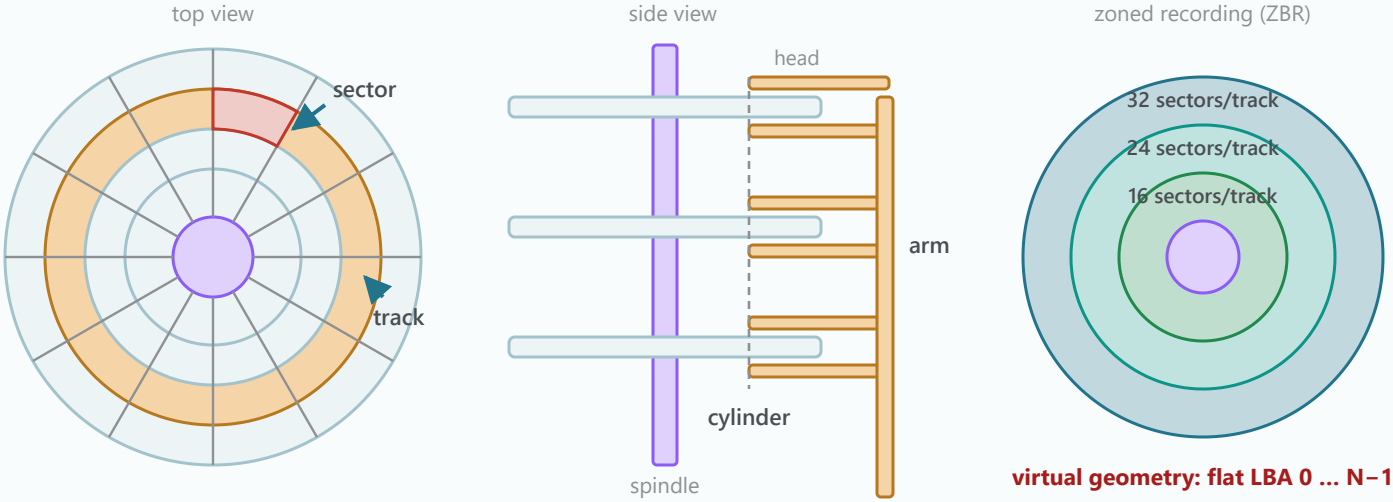
Terminals and GUI input

- **tty** + **line discipline**: canonical mode edits the line (backspace, deliver on Enter); raw mode delivers every key at once (`stty -icanon` , editors).
- **Pseudoterminal** (pty): terminal emulators, `ssh` ; screen control through ANSI escape sequences.
- The keyboard sends *scan codes* (press/release); the driver maps them to characters with a keymap. USB HID devices are *polled* by the host at 125–1000 Hz.
- Linux: input subsystem → `/dev/input/event*` (evdev) → libinput → Wayland compositor → window.

Power management

- ACPI system states: **S0** working (S0ix modern standby), **S3** suspend to RAM, **S4** hibernate (to disk), **S5** off.
- CPU: **C-states** (idle sleep depth), **P-states** (frequency/voltage; Linux `schedutil`).
- Devices: **D0–D3** states, runtime power management; NVMe APST, PCIe ASPM, Wi-Fi power saving, screen dimming.
- The OS puts devices to sleep and wakes them in the right order, and chooses the wakeup sources (keyboard, network, clock). Battery life depends mostly on idle time.

Magnetic Disk Geometry



- Platters spin at 5400–7200 rpm (servers: 10–15k), with one head per surface. Tracks at the same radius form a **cylinder**. Outer zones hold more sectors per track.
- The OS sees **LBAs**, not CHS (cylinder, head, sector); the controller inside the disk maps them to the real geometry, spare sectors and cache. The old CHS limit: 1024 × 255 × 63 × 512 B ≈ 8.4 GB.

Access Time: An Example

Access time = **seek** + **rotational latency** + **transfer**. Disk: 7200 rpm, 8 ms average seek, 200 MB/s media rate; 4 KiB block.

All three components count, transfer time included.

Access Time: An Example

Access time = **seek** + **rotational latency** + **transfer**. Disk: 7200 rpm, 8 ms average seek, 200 MB/s media rate; 4 KiB block.

One revolution = 60,000 / 7200 = **8.33 ms**; average rotational latency = half a revolution = **4.17 ms**.

On average the target sector is half a turn away.

Access Time: An Example

Access time = **seek** + **rotational latency** + **transfer**. Disk: 7200 rpm, 8 ms average seek, 200 MB/s media rate; 4 KiB block.

One revolution = 60,000 / 7200 = **8.33 ms**; average rotational latency = half a revolution = **4.17 ms**.

Transfer = 4096 B ÷ 200 MB/s = **0.020 ms**. Total = 8 + 4.17 + 0.02 = **12.19 ms**.

Mechanical delay dominates: almost all the time goes to seek and rotation. The CPU could run millions of instructions meanwhile; the OS does not wait, it runs another process.

Access Time: An Example

Access time = **seek** + **rotational latency** + **transfer**. Disk: 7200 rpm, 8 ms average seek, 200 MB/s media rate; 4 KiB block.

One revolution = 60,000 / 7200 = **8.33 ms**; average rotational latency = half a revolution = **4.17 ms**.

Transfer = 4096 B ÷ 200 MB/s = **0.020 ms**. Total = 8 + 4.17 + 0.02 = **12.19 ms**.

Random 4 KiB: 1000 / 12.19 ≈ **82 IOPS** → 82 × 4 KiB ≈ **0.34 MB/s**. Sequential 1 MiB: 8 + 4.17 + 5.24 = 17.41 ms → **60 MB/s**.

The same disk is 180 times faster or slower depending on the access pattern, which is why locality and scheduling matter. (The math assumes a constant media rate and a fresh seek + rotation per chunk; a long sequential read seeks once and approaches the media rate.)

Access Time: An Example

Access time = **seek** + **rotational latency** + **transfer**. Disk: 7200 rpm, 8 ms average seek, 200 MB/s media rate; 4 KiB block.

One revolution = 60,000 / 7200 = **8.33 ms**; average rotational latency = half a revolution = **4.17 ms**.

Transfer = 4096 B ÷ 200 MB/s = **0.020 ms**. Total = 8 + 4.17 + 0.02 = **12.19 ms**.

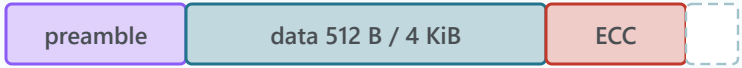
Random 4 KiB: 1000 / 12.19 ≈ **82 IOPS** → 82 × 4 KiB ≈ **0.34 MB/s**. Sequential 1 MiB: 8 + 4.17 + 5.24 = 17.41 ms → **60 MB/s**.



Compare: an NVMe SSD does a random 4 KiB read in ≈ 50–100 µs and over a million IOPS with parallel queues.

Disk Formatting and Cylinder Skew

sector format (low-level format)



skew: don't miss sector 0 when switching tracks

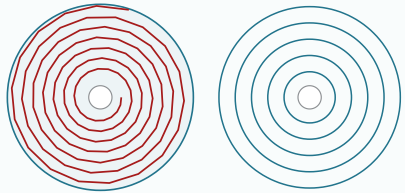
track 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
track 1	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
track 2	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9

- **Low-level formatting** is done at the factory: tracks, sectors, preamble, ECC and spare sectors. Formatted capacity is less than raw capacity. **High-level formatting** writes an empty file system to a partition (boot block, free-space management, root directory).
- Skew math (Tanenbaum): 10,000 rpm → 6 ms per revolution; 300 sectors/track → 20 µs per sector. With an 800 µs track switch, skew = 800 / 20 = **40 sectors**.
- **Interleaving** once spaced consecutive sectors apart for slow controllers; today the controller buffers the whole track, so it is unnecessary.
- Today's disks: 4 KiB physical sectors (Advanced Format); 512e disks emulate 512 B, and unaligned writes need read-modify-write.

Optical Disks (Briefly)

	CD (1982)	DVD (1996)	Blu-ray (2006)
Capacity	650–700 MB	4.7 / 8.5 / 9.4 / 17 GB	25 / 50 GB; BDXL 100–128 GB
Laser	780 nm (infrared)	650 nm (red)	405 nm (blue-violet)
Smallest pit	0.83 µm	0.40 µm	0.15 µm
Track pitch	1.6 µm	0.74 µm	0.32 µm

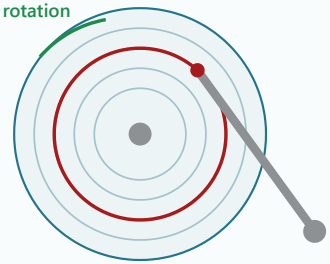
- A single **spiral track**; a *transition* between pit and land is a 1 bit. On pressed discs the pits are molded into polycarbonate, not burned by a laser. On CD-R/DVD-R the laser heats an organic dye layer to change its reflectivity.
- On a CD-ROM each byte becomes a 14-bit channel symbol via **EFM** (a modulation code, not ECC); error correction uses **CIRC**. 98 frames × 24 data bytes = 2352 B; in mode 1, 2048 B of that is user data and the rest is sync, header and extra ECC/EDC.
- A DVD's two layers are semitransparent; the laser reads the lower one by refocusing. Optical media now serve mainly archiving and distribution; their density trails magnetic and flash media.



optical: one spiral track magnetic: concentric

Disk Arm Scheduling

- The goal is to cut arm movement (seek time) and raise throughput; the driver keeps a list of (cylinder, request time). Metric: **total head movement** (cylinders).
- Today the LBA → physical mapping lives inside the disk, and the disk itself reorders 32 requests by rotational position with **NCQ**. The OS schedules approximately, by "LBA proximity".



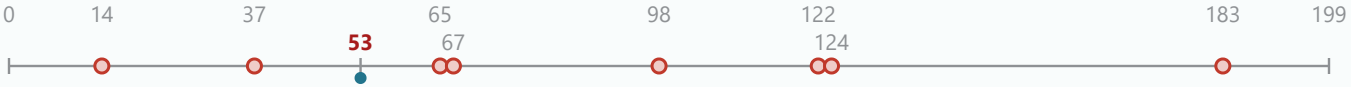
seek: arm changes cylinder
red: the head's cylinder

Algorithm	Idea	Problem
FCFS	Arrival order	Fair, but the arm wanders a lot
SSTF	Request closest to the head (greedy)	Requests at the edges can *starve*
SCAN (elevator)	Sweep to the end in one direction, then reverse	Edges and middle do not wait equally
C-SCAN	Sweep one way, jump from the end back to the start	The return trip is wasted
LOOK / C-LOOK	SCAN/C-SCAN, but only up to the last request	The forms used in practice

The examples use Silberschatz's queue: head at 53; requests 98, 183, 37, 122, 14, 124, 65, 67. Cylinders 0–199.

FCFS

First come, first served: requests in arrival order.



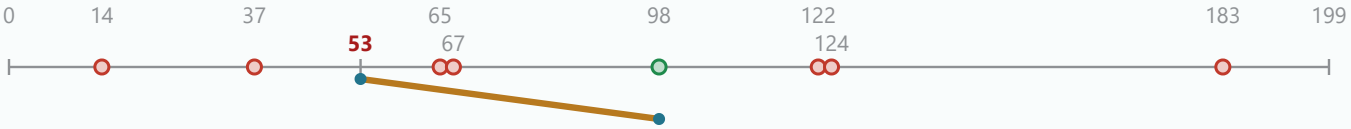
served waiting **total: 0 cylinders**

The head is at **53**. Queue: 98, 183, 37, 122, 14, 124, 65, 67 (cylinders 0–199). Red: waiting, green: served.

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.



served waiting

total: 45 cylinders

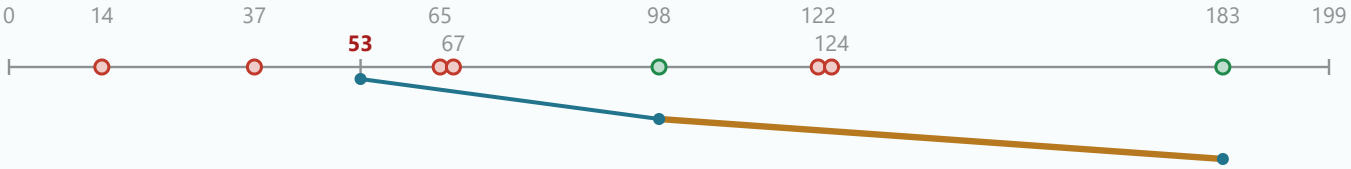
53 → 98: **45** cylinders · total **45**

Step 2 / 9

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.



served waiting

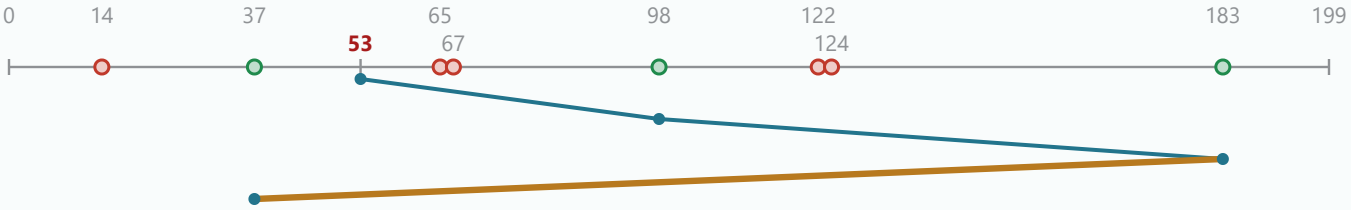
total: 130 cylinders

98 → 183: **85** cylinders · total **130**

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.

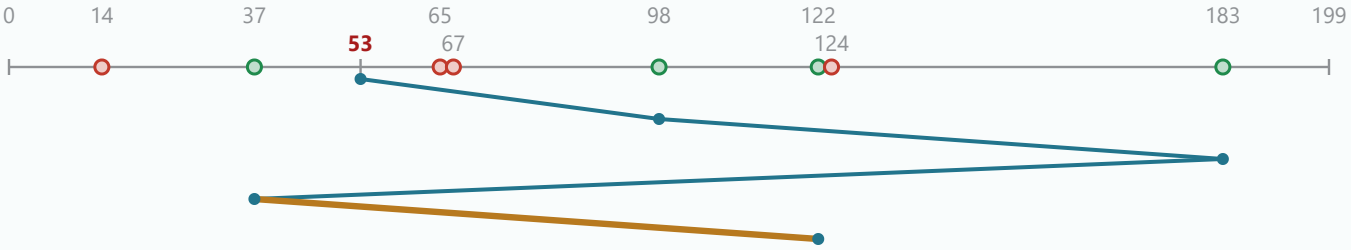


183 → 37: **146** cylinders · total **276**

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.



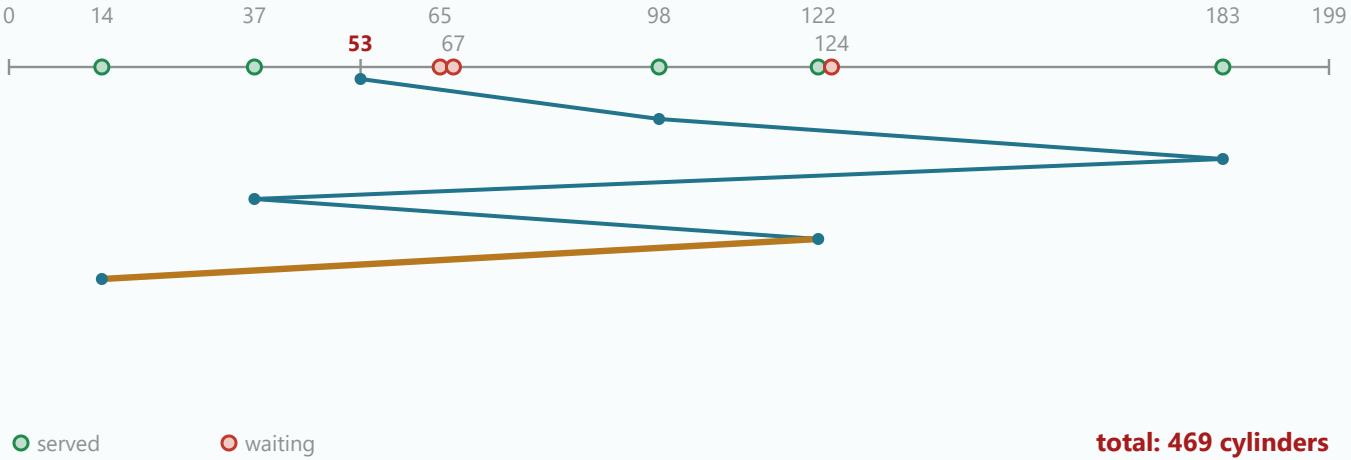
37 → 122: **85** cylinders · total **361**

Step 5 / 9

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.

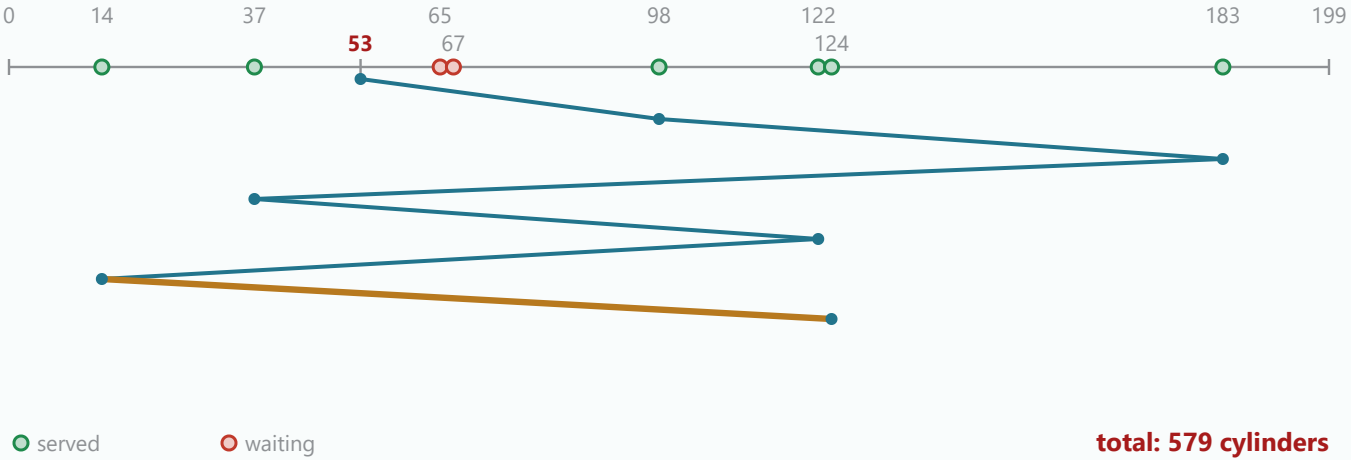


122 → 14: **108** cylinders · total **469**

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.

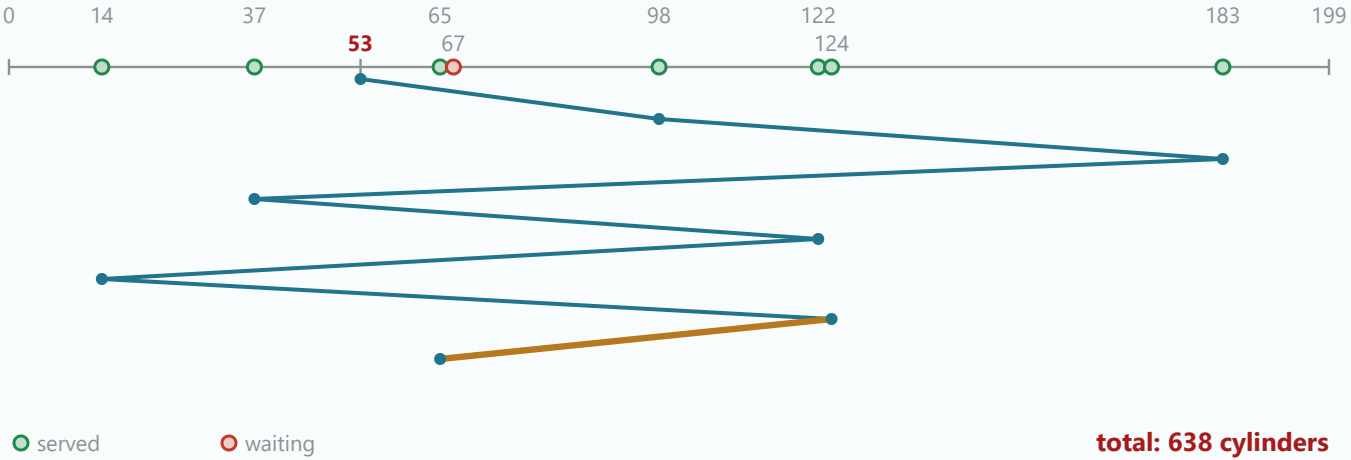


14 → 124: **110** cylinders · total **579**

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.

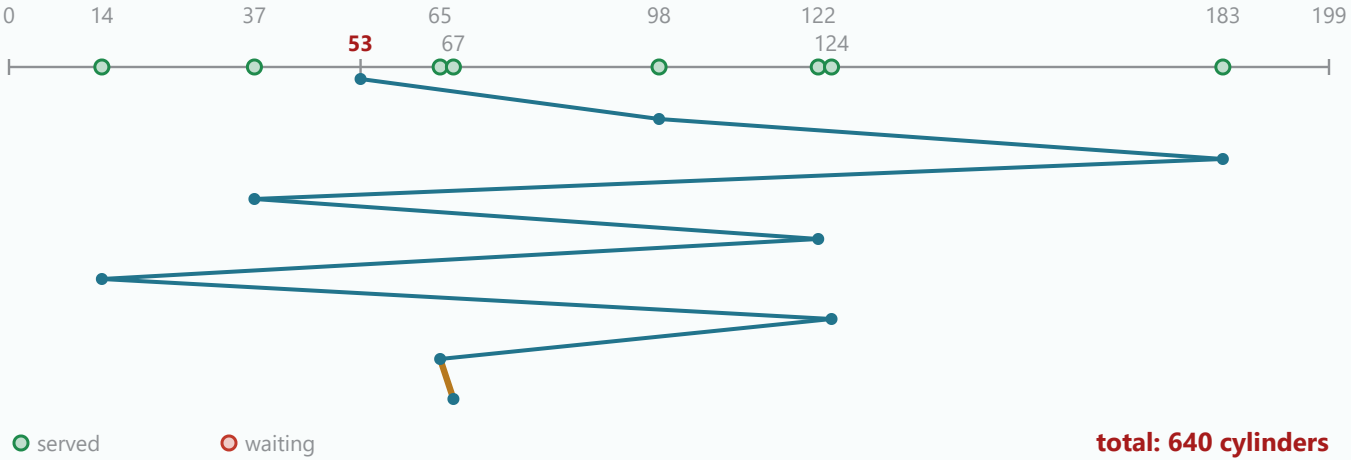


124 → 65: **59** cylinders · total **638**

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

FCFS

First come, first served: requests in arrival order.



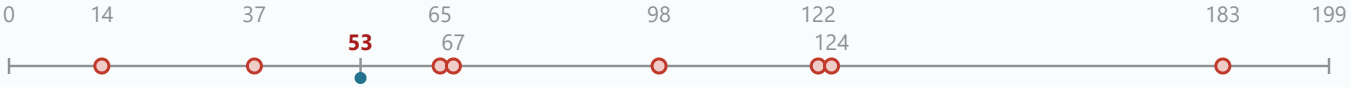
65 → 67: **2** cylinders · total **640**

Step 9 / 9

640 cylinders in all. Fair and starvation-free, but swings like 122 → 14 → 124 are very costly.

SSTF

Shortest seek time first: at each step, the request closest to the head.



served waiting **total: 0 cylinders**

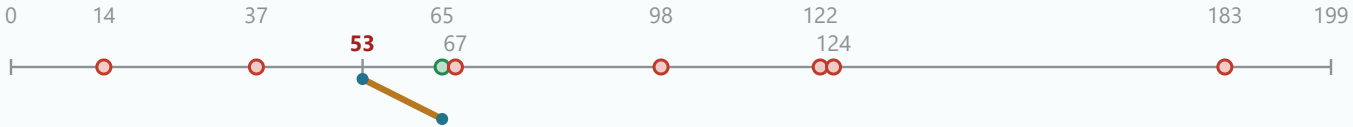
The head is at **53**. Queue: 98, 183, 37, 122, 14, 124, 65, 67 (cylinders 0–199). Red: waiting, green: served.

Step 1 / 9

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



served waiting

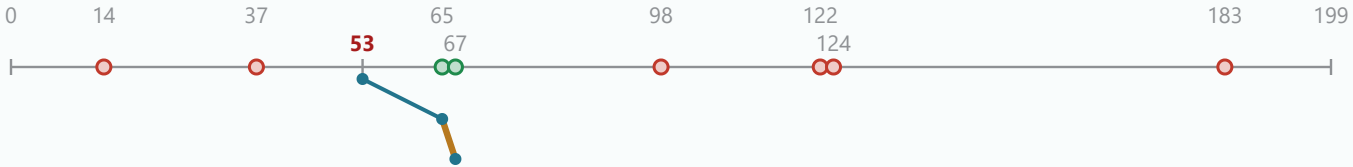
total: 12 cylinders

53 → 65: **12** cylinders · total **12**

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



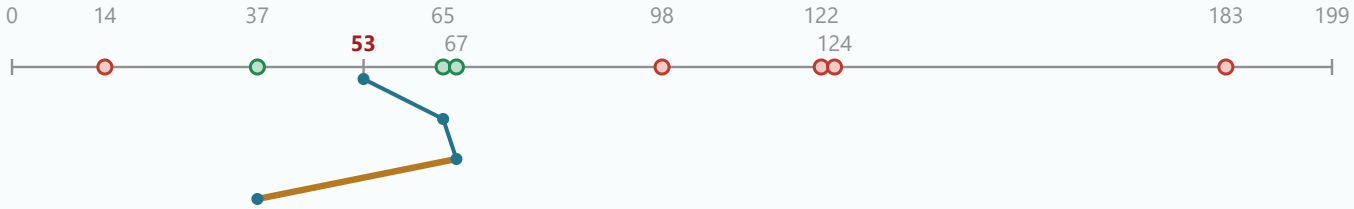
served waiting **total: 14 cylinders**

65 → 67: **2** cylinders · total **14**

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



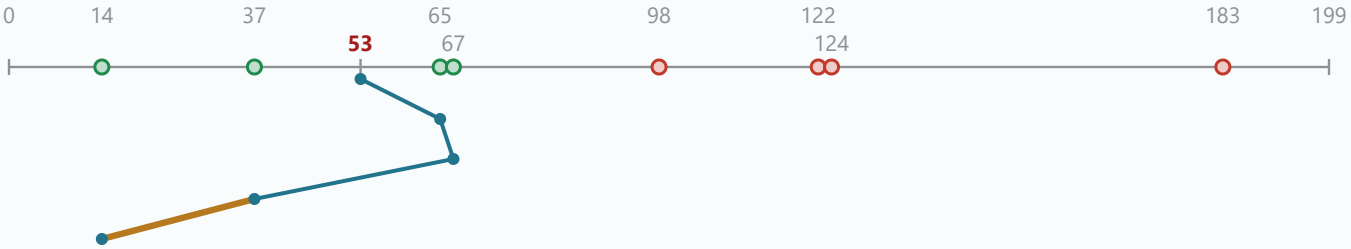
total: 44 cylinders

67 → 37: **30** cylinders · total **44**

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



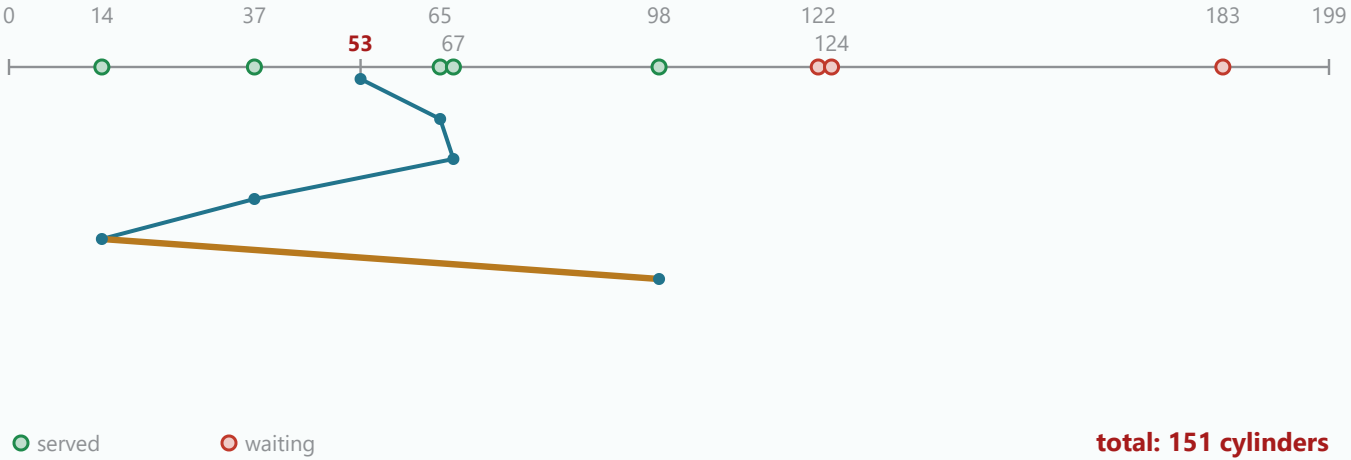
37 → 14: **23** cylinders · total **67**

Step 5 / 9

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



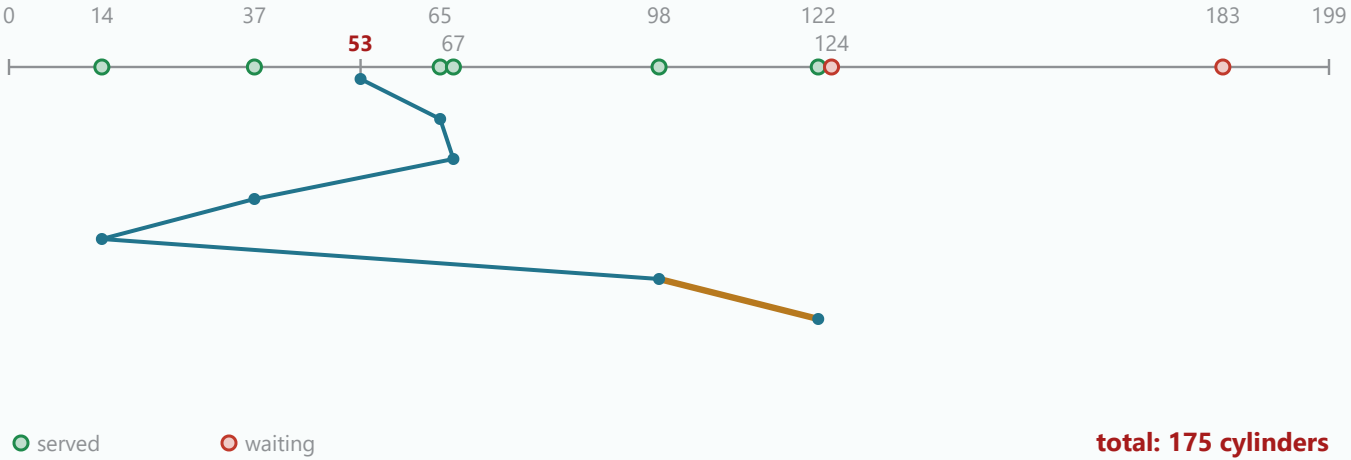
14 → 98: **84** cylinders · total **151**

Step 6 / 9

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.

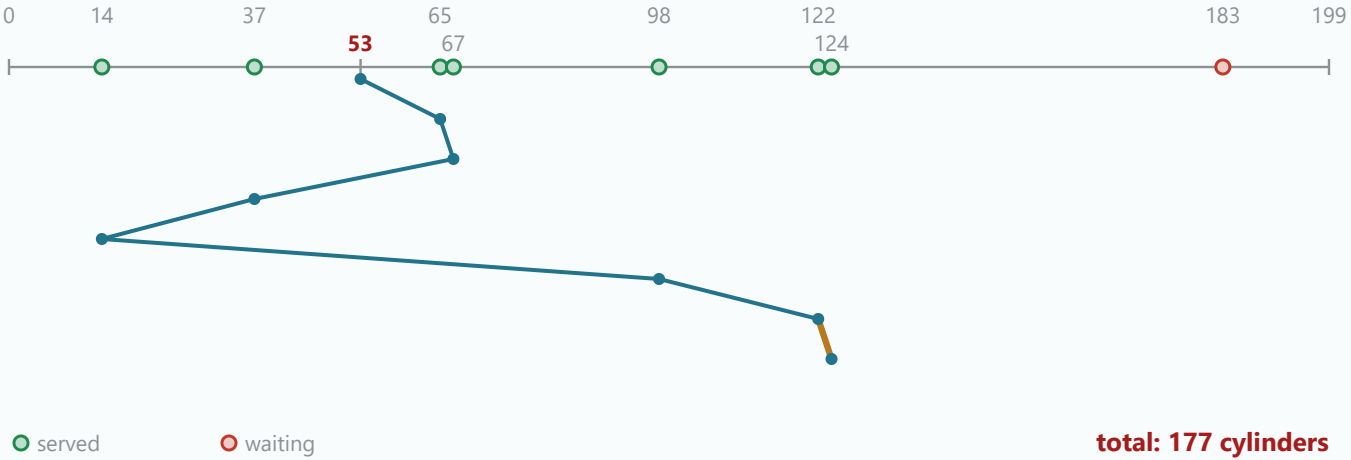


98 → 122: **24** cylinders · total **175**

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



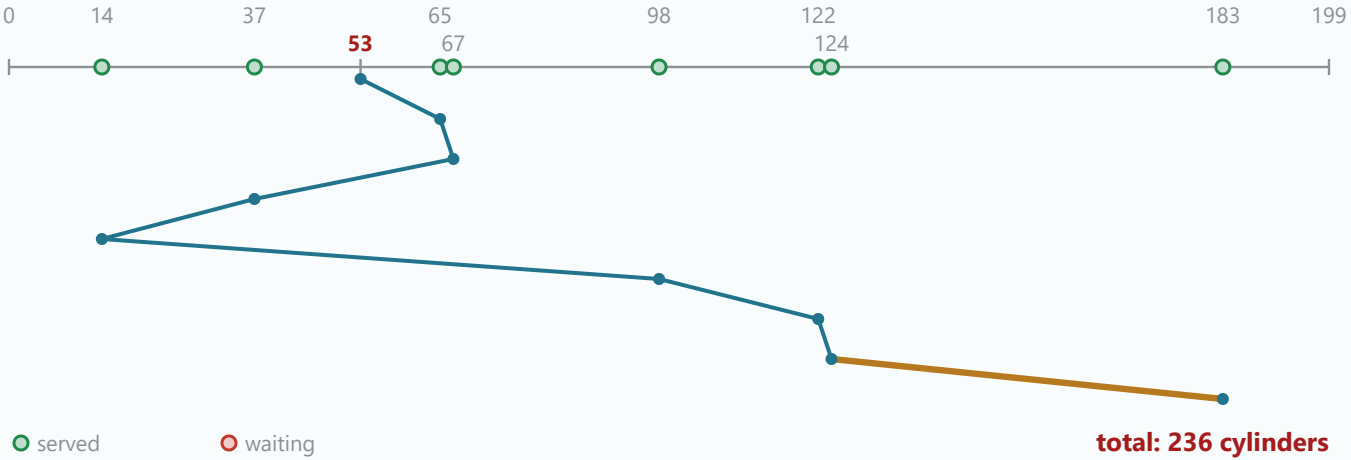
122 → 124: **2** cylinders · total **177**

Step 8 / 9

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF

Shortest seek time first: at each step, the request closest to the head.



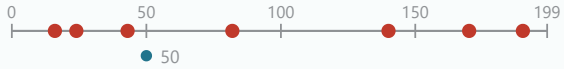
124 → 183: **59** cylinders · total **236**

Step 9 / 9

236 in all (37% of FCFS). It is greedy and not optimal: 53 → 37 → 14 → 65 … is shorter (208). Under heavy load, requests in the middle can *starve* those at the edges.

SSTF: Code Trace

```
▶ int sstf(int head, int q[], int n) {
▶   int total = 0;
3   for (int k = 0; k < n; k++) {
4     int best = -1;
5     for (int i = 0; i < n; i++)
6       if (!done[i] && (best < 0 ||
7         abs(q[i]-head) < abs(q[best]-head)))
8         best = i;          /* closest */
9     total += abs(q[best] - head);
10    head = q[best];
11    done[best] = 1;
12  }
13  return total;
14 }
```



total = 0 cylinders

The head is at cylinder 50; queue (arrival order): 82, 170, 43, 140, 24, 16, 190. Each round, the inner loop looks at every waiting request.

Step 1 / 9

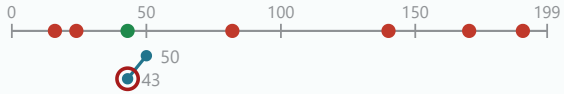
VARIABLES

`head = 50` `k = –` `best = –` `total = 0`

OUTPUT

SSTF: Code Trace

```
 1 int sstf(int head, int q[], int n) {
 2   int total = 0;
 3   for (int k = 0; k < n; k++) {
 4     int best = -1;
 5     for (int i = 0; i < n; i++)
 6       if (!done[i] && (best < 0 ||
 7         abs(q[i]-head) < abs(q[best]-head)))
 8         best = i;          /* closest */
 9     total += abs(q[best] - head);
10     head = q[best];
11     done[best] = 1;
12   }
13   return total;
14 }
```



total = 7 cylinders

k = 0: head at 50, nearest waiting request **43** (distance 7; runner-up 24, distance 26). total = 7.

Step 2 / 9

VARIABLES

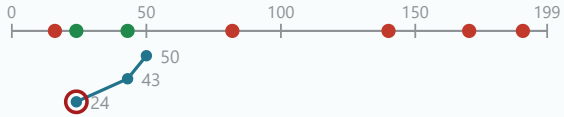
`head = 43` `k = 0` `best = 2` `total = 7`

OUTPUT

```
43
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
▶          best = i;          /* closest */
▶      total += abs(q[best] - head);
▶      head = q[best];
▶      done[best] = 1;
12   }
13   return total;
14 }
```



total = 26 cylinders

k = 1: head at 43, nearest waiting request **24** (distance 19; runner-up 16, distance 27). total = 26.

Step 3 / 9

VARIABLES

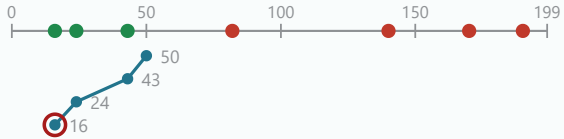
`head = 24` `k = 1` `best = 4` `total = 26`

OUTPUT

```
43 24
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
           best = i;          /* closest */
       total += abs(q[best] - head);
       head = q[best];
       done[best] = 1;
12   }
13   return total;
14 }
```



total = 34 cylinders

k = 2: head at 24, nearest waiting request **16** (distance 8; runner-up 82, distance 58). total = 34.

Step 4 / 9

VARIABLES

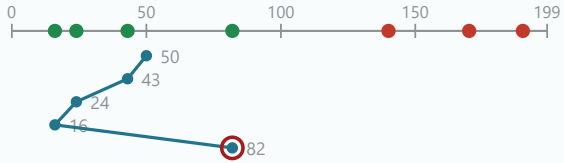
head = 16 | k = 2 | best = 5 | total = 34

OUTPUT

```
43 24 16
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
▶          best = i;          /* closest */
▶      total += abs(q[best] - head);
▶      head = q[best];
▶      done[best] = 1;
12   }
13   return total;
14 }
```



total = 100 cylinders

k = 3: head at 16, nearest waiting request **82** (distance 66; runner-up 140, distance 124). total = 100. The low requests are done, so the arm travels a long way back.

Step 5 / 9

VARIABLES

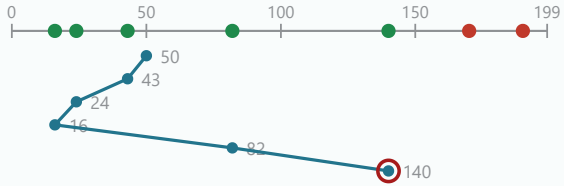
`head = 82` `k = 3` `best = 0` `total = 100`

OUTPUT

```
43 24 16 82
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
▶          best = i;          /* closest */
▶      total += abs(q[best] - head);
▶      head = q[best];
▶      done[best] = 1;
12   }
13   return total;
14 }
```



total = 158 cylinders

k = 4: head at 82, nearest waiting request **140** (distance 58; runner-up 170, distance 88). total = 158.

Step 6 / 9

VARIABLES

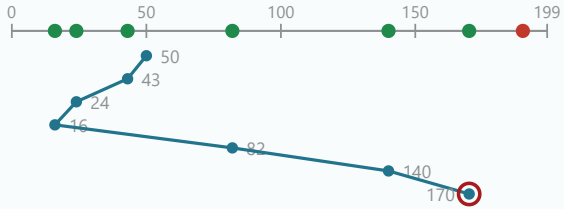
`head = 140` `k = 4` `best = 3` `total = 158`

OUTPUT

```
43 24 16 82 140
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
           best = i;          /* closest */
       total += abs(q[best] - head);
       head = q[best];
       done[best] = 1;
12   }
13   return total;
14 }
```



total = 188 cylinders

k = 5: head at 140, nearest waiting request **170** (distance 30; runner-up 190, distance 50). total = 188.

Step 7 / 9

VARIABLES

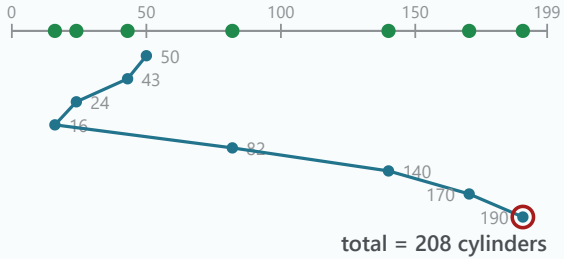
`head = 170` `k = 5` `best = 1` `total = 188`

OUTPUT

```
43 24 16 82 140 170
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
▶          best = i;          /* closest */
▶      total += abs(q[best] - head);
▶      head = q[best];
▶      done[best] = 1;
12   }
13   return total;
14 }
```



k = 6: head at 170, nearest waiting request **190** (distance 20). total = 208.

Step 8 / 9

VARIABLES

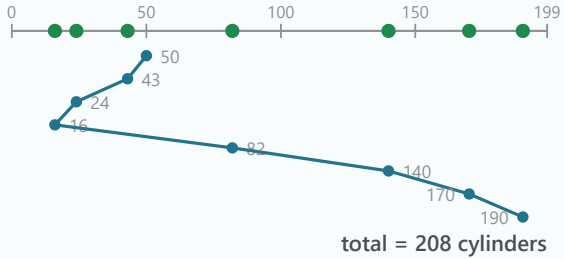
`head = 190` `k = 6` `best = 6` `total = 208`

OUTPUT

```
43 24 16 82 140 170 190
```

SSTF: Code Trace

```
1  int sstf(int head, int q[], int n) {
2    int total = 0;
3    for (int k = 0; k < n; k++) {
4      int best = -1;
5      for (int i = 0; i < n; i++)
6        if (!done[i] && (best < 0 ||
7          abs(q[i]-head) < abs(q[best]-head)))
8          best = i;          /* closest */
9      total += abs(q[best] - head);
10     head = q[best];
11     done[best] = 1;
12   }
▶    return total;
14 }
```



total = 208 cylinders

208 cylinders in all (FCFS: 642 on this queue). The greedy choice takes the shortest move each time; if requests keep arriving in the middle, those at the edges may wait.

Step 9 / 9

VARIABLES

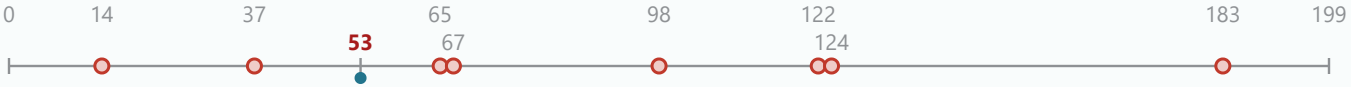
`head = 190` `k = 7` `best = –` `total = 208`

OUTPUT

```
43 24 16 82 140 170 190
```

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



served waiting **total: 0 cylinders**

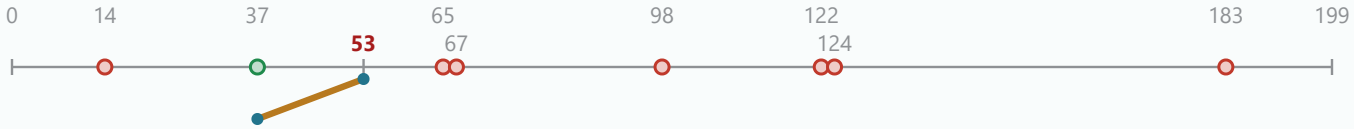
The head is at **53**. Queue: 98, 183, 37, 122, 14, 124, 65, 67 (cylinders 0–199). Red: waiting, green: served.

Step 1 / 10

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



served waiting **total: 16 cylinders**

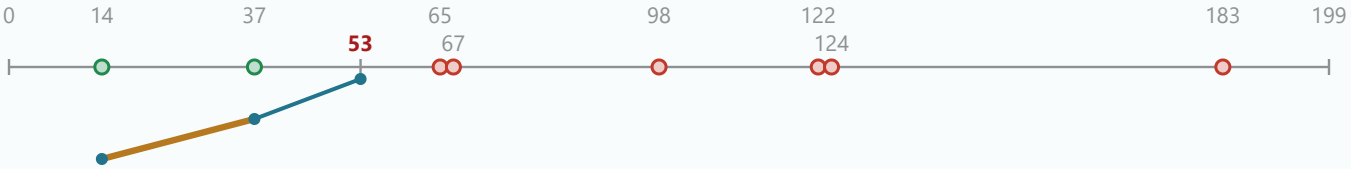
53 → 37: **16** cylinders · total **16**

Step 2 / 10

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



served waiting **total: 39 cylinders**

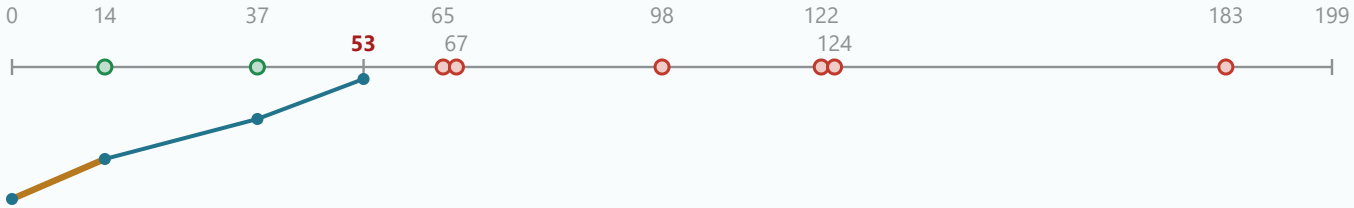
37 → 14: **23** cylinders · total **39**

Step 3 / 10

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.

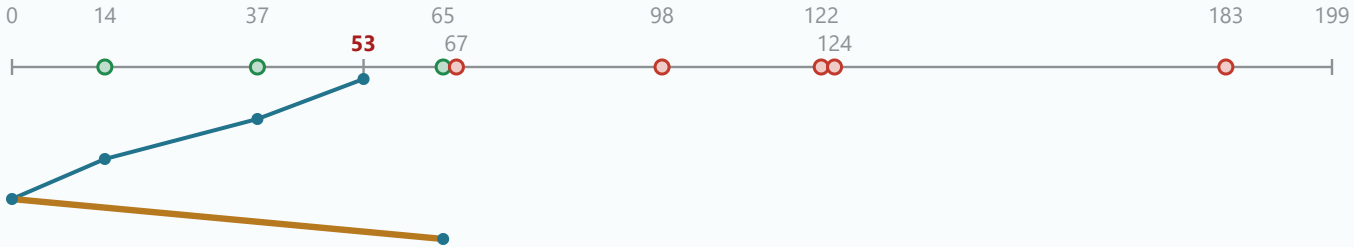


14 → 0: **14** cylinders · total **53** (no request; the arm goes to the edge)

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



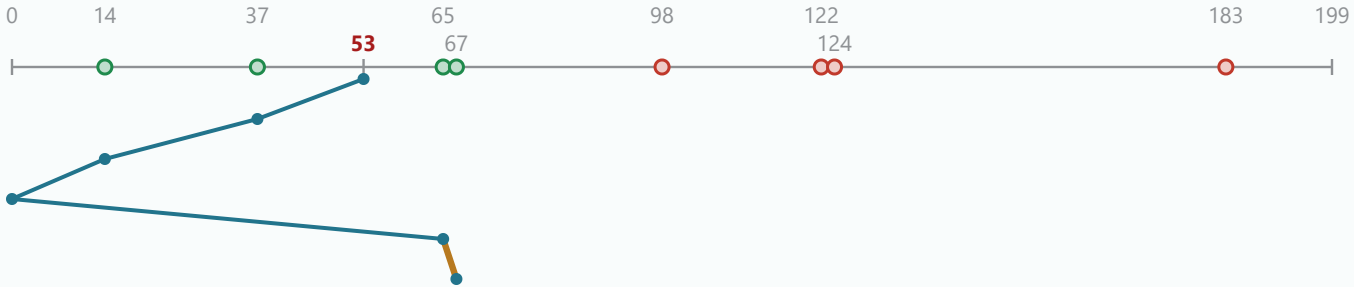
served waiting **total: 118 cylinders**

0 → 65: **65** cylinders · total **118**

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



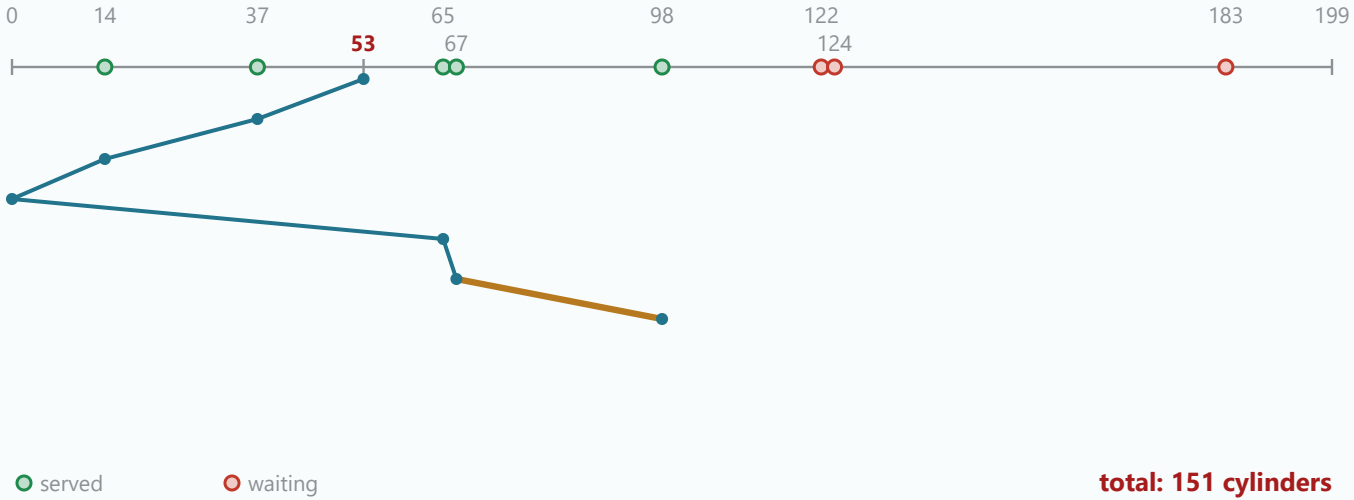
65 → 67: **2** cylinders · total **120**

Step 6 / 10

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



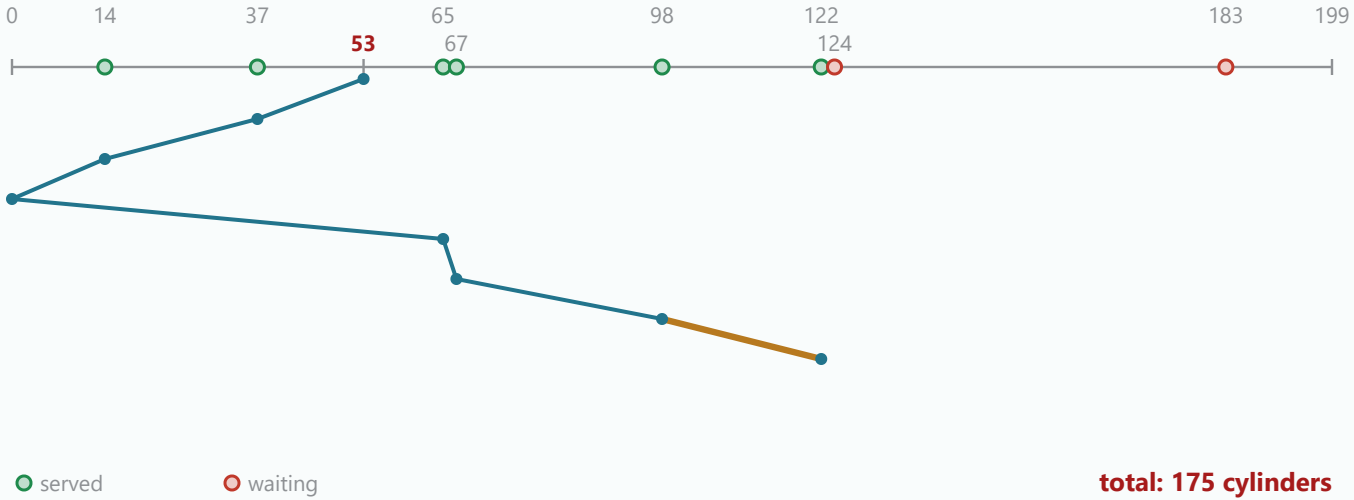
67 → 98: **31** cylinders · total **151**

Step 7 / 10

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.

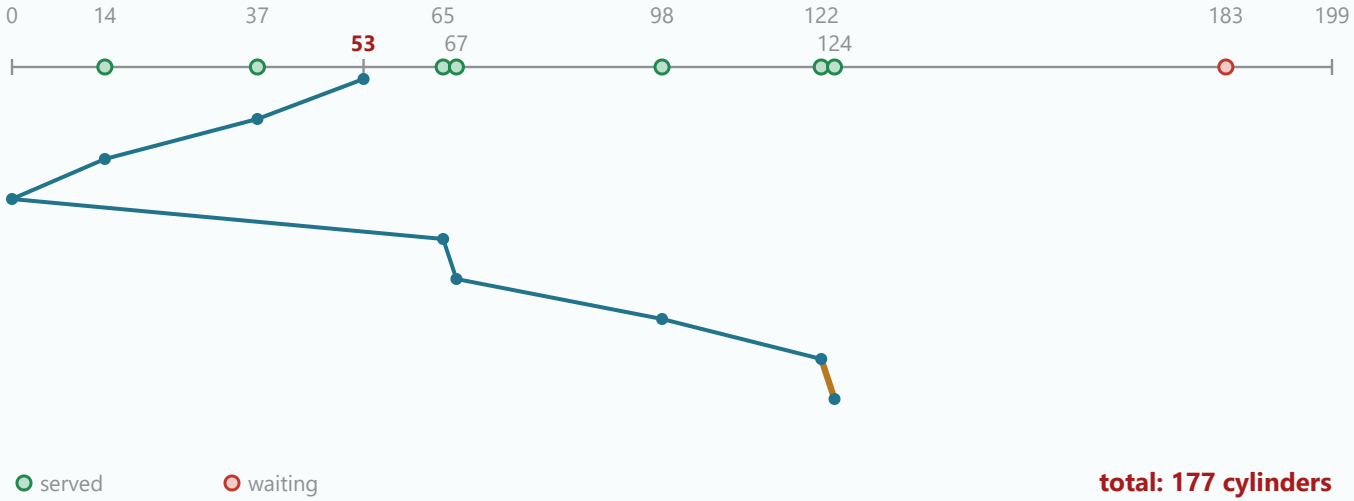


98 → 122: **24** cylinders · total **175**

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.

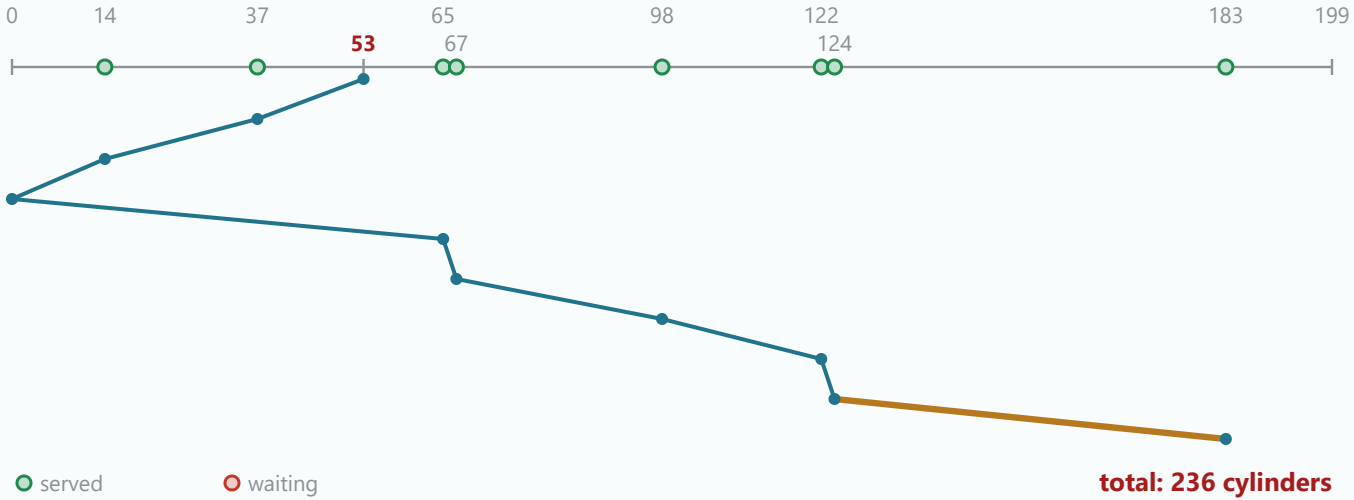


122 → 124: **2** cylinders · total **177**

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

SCAN (Elevator)

SCAN: the head sweeps in one direction (here toward 0) to the end, then reverses, like an elevator.



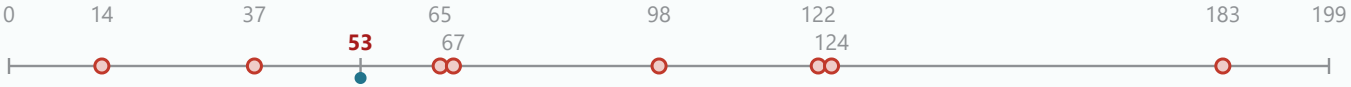
124 → 183: **59** cylinders · total **236**

Step 10 / 10

236 in all. Getting stuck in one region is SSTF's greedy flaw; SCAN prevents starvation. Going to the edge and back (14 → 0 → 14) wastes 28 cylinders.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.



served waiting **total: 0 cylinders**

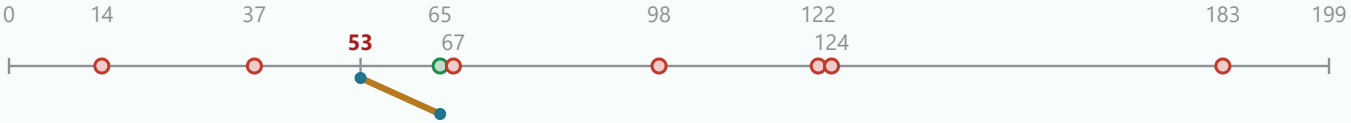
The head is at **53**. Queue: 98, 183, 37, 122, 14, 124, 65, 67 (cylinders 0–199). Red: waiting, green: served.

Step 1 / 11

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.



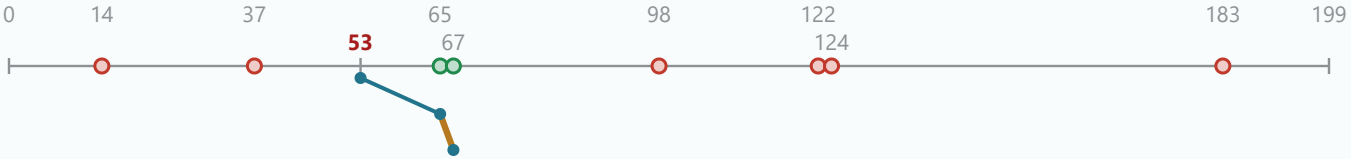
served waiting **total: 12 cylinders**

53 → 65: **12** cylinders · total **12**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.



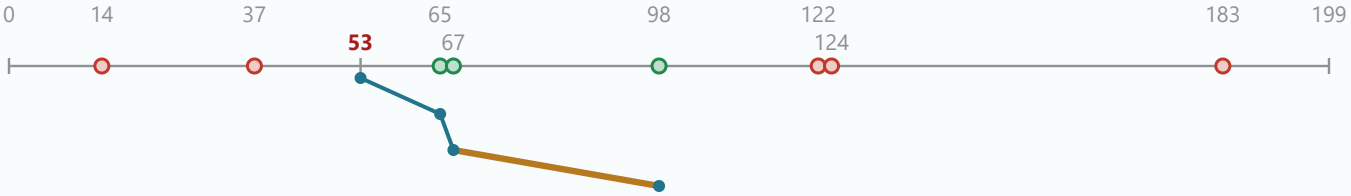
served waiting **total: 14 cylinders**

65 → 67: **2** cylinders · total **14**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.



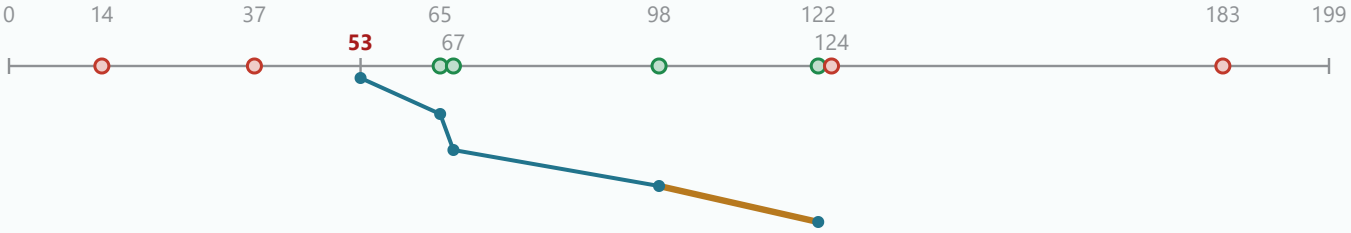
total: 45 cylinders

67 → 98: **31** cylinders · total **45**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.

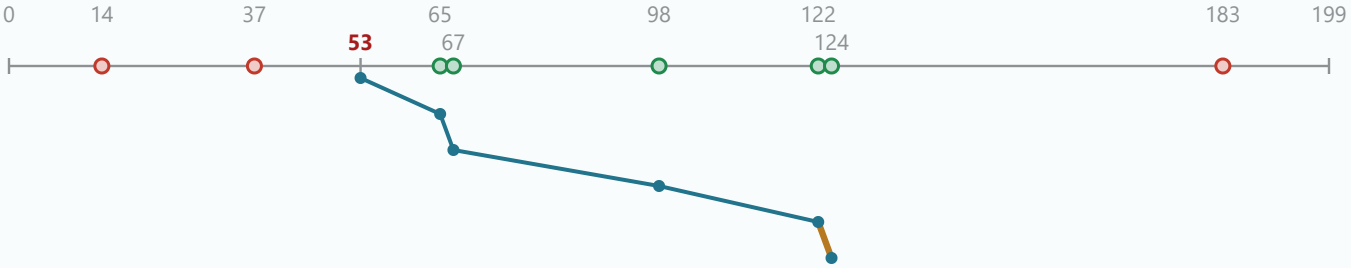


98 → 122: **24** cylinders · total **69**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.

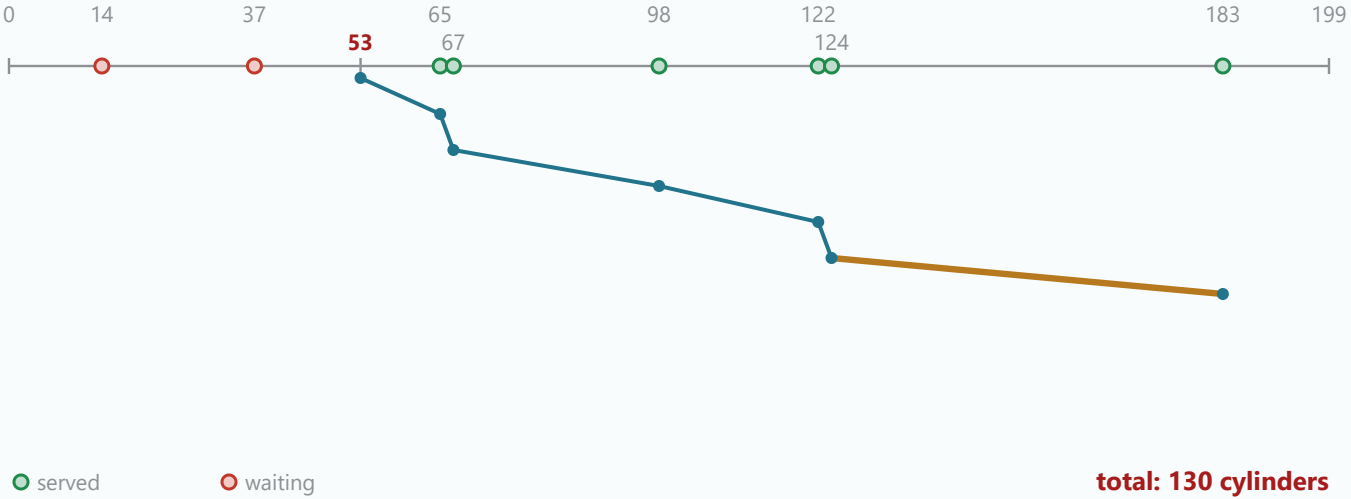


122 → 124: **2** cylinders · total **71**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.

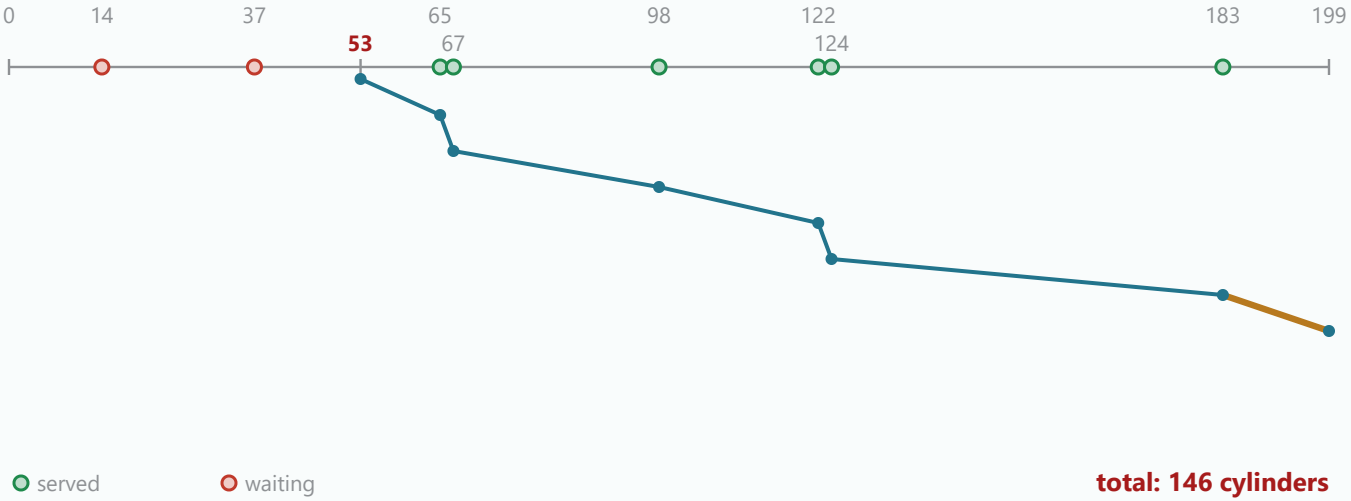


124 → 183: **59** cylinders · total **130**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.

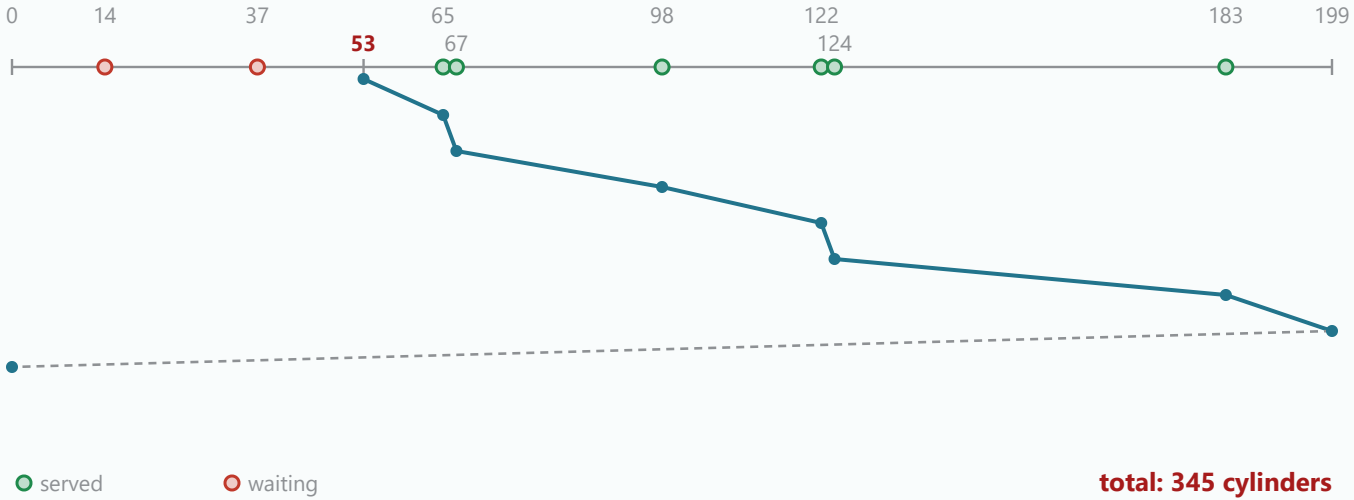


183 → 199: **16** cylinders · total **146** (no request; the arm goes to the edge)

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.

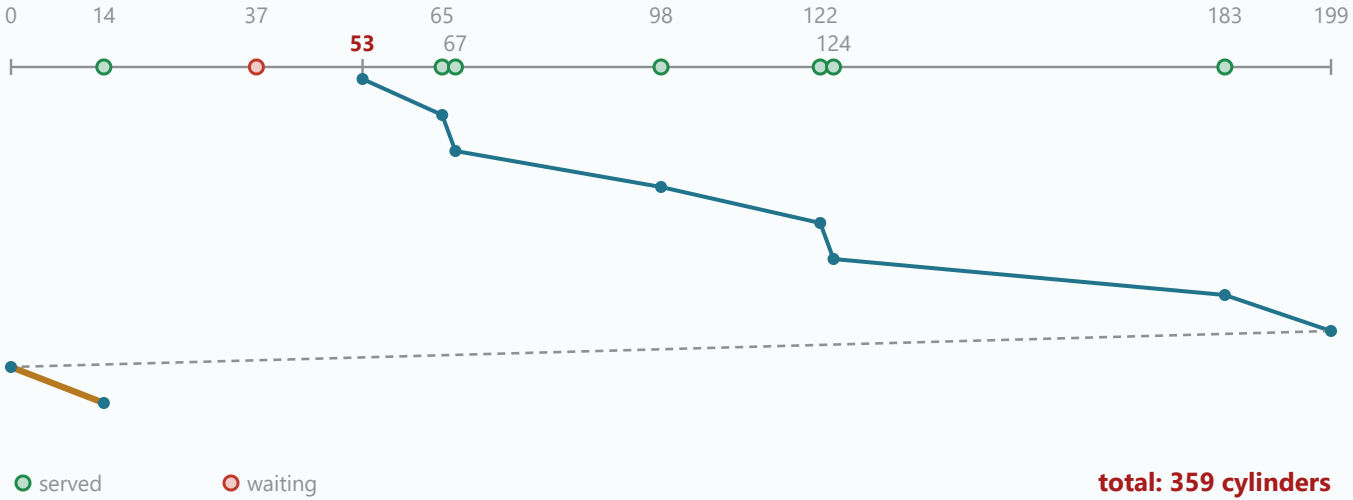


199 → 0: return jump (199 cylinders), no requests served. Counted here · total **345**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.

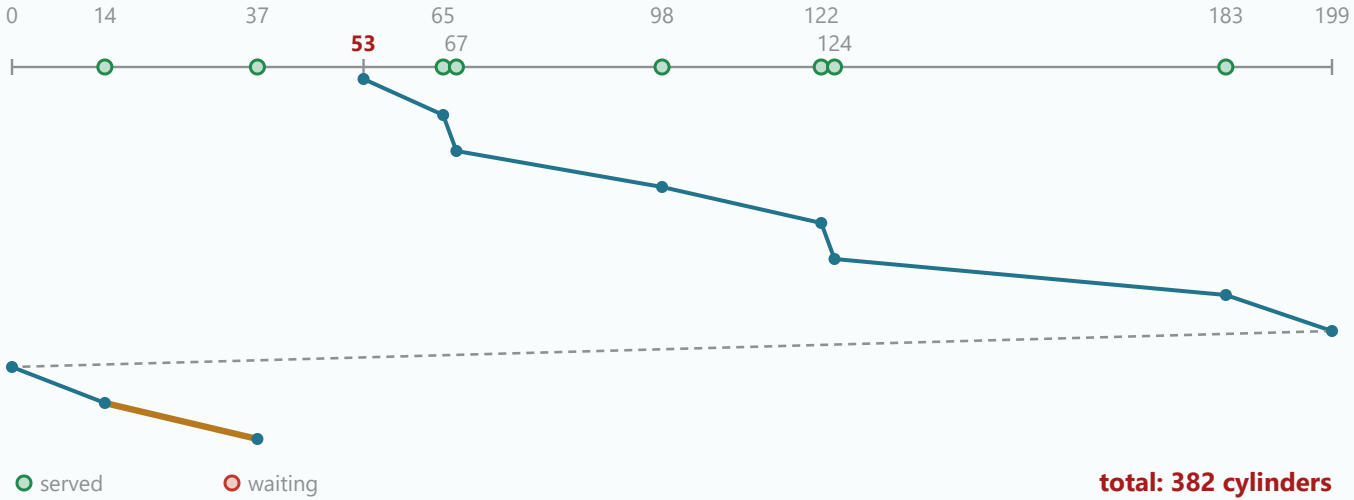


0 → 14: **14** cylinders · total **359**

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

C-SCAN

Circular SCAN: serve in the upward direction only; at the end, return to the start without serving.



14 → 37: **23** cylinders · total **382**

Step 11 / 11

382 in all (183 without the return). The disk is treated as a ring; waiting times are more uniform than with SCAN.

LOOK and C-LOOK



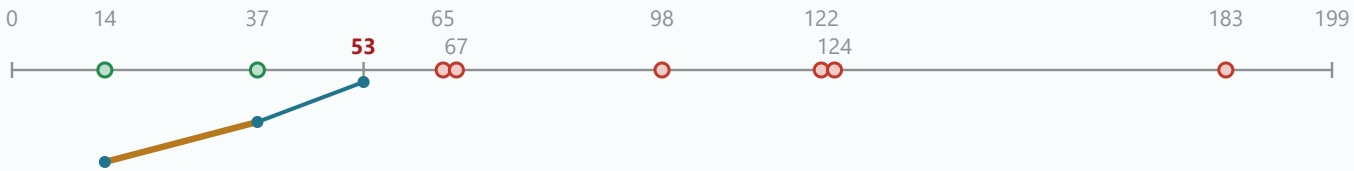
served waiting **total: 0 cylinders**

LOOK (downward): like SCAN, but it does not go to the edge; it reverses at the last request in that direction.

Step 1 / 6

Linux's old *deadline* and *CFQ* schedulers and today's *mq-deadline* dispatch in LBA order, which behaves much like C-LOOK.

LOOK and C-LOOK

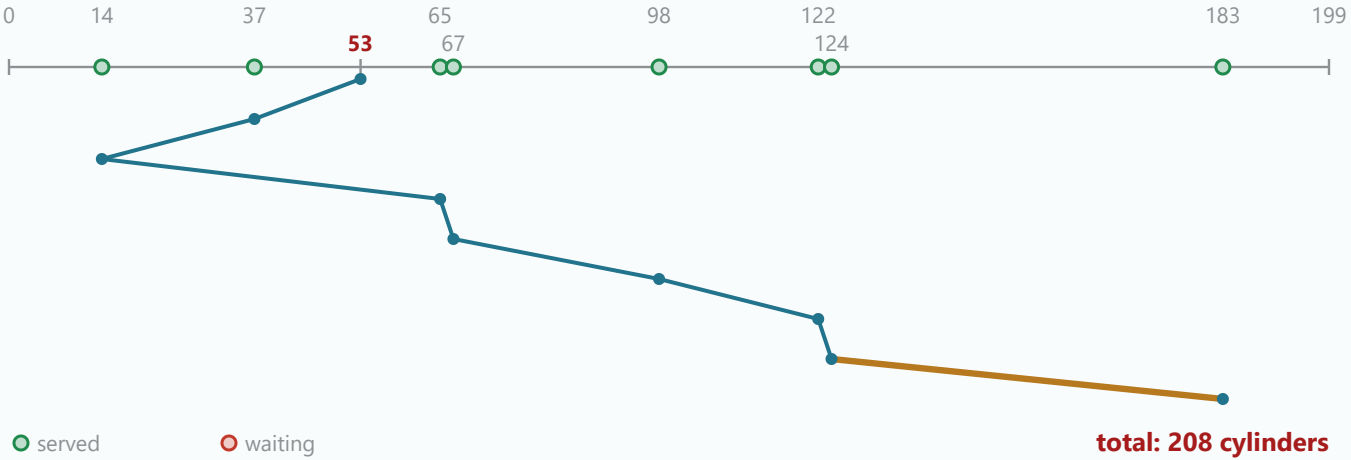


served waiting **total: 39 cylinders**

53 → 37 → 14: the low requests are done, so no trip to 0. Total 39.

Linux's old *deadline* and *CFQ* schedulers and today's *mq-deadline* dispatch in LBA order, which behaves much like C-LOOK.

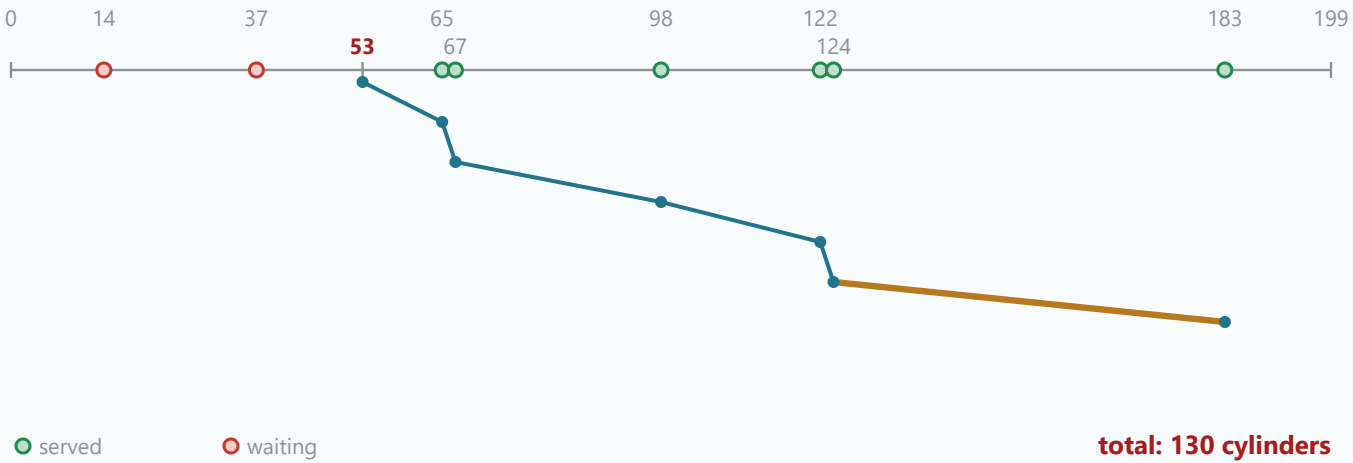
LOOK and C-LOOK



14 → 65 … 183: direction reverses. **Total 208** (28 less than SCAN). In practice, the “elevator” behavior is usually this LOOK.

Linux's old *deadline* and *CFQ* schedulers and today's *mq-deadline* dispatch in LBA order, which behaves much like C-LOOK.

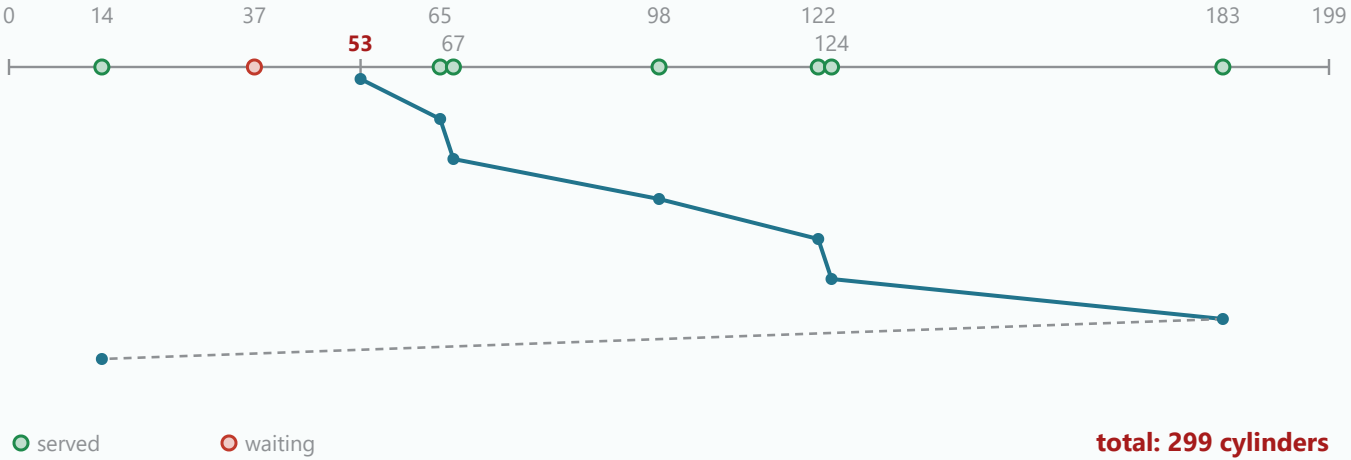
LOOK and C-LOOK



C-LOOK (upward): 53 → 65 … 183, the last request going up. Total 130.

Linux's old *deadline* and *CFQ* schedulers and today's *mq-deadline* dispatch in LBA order, which behaves much like C-LOOK.

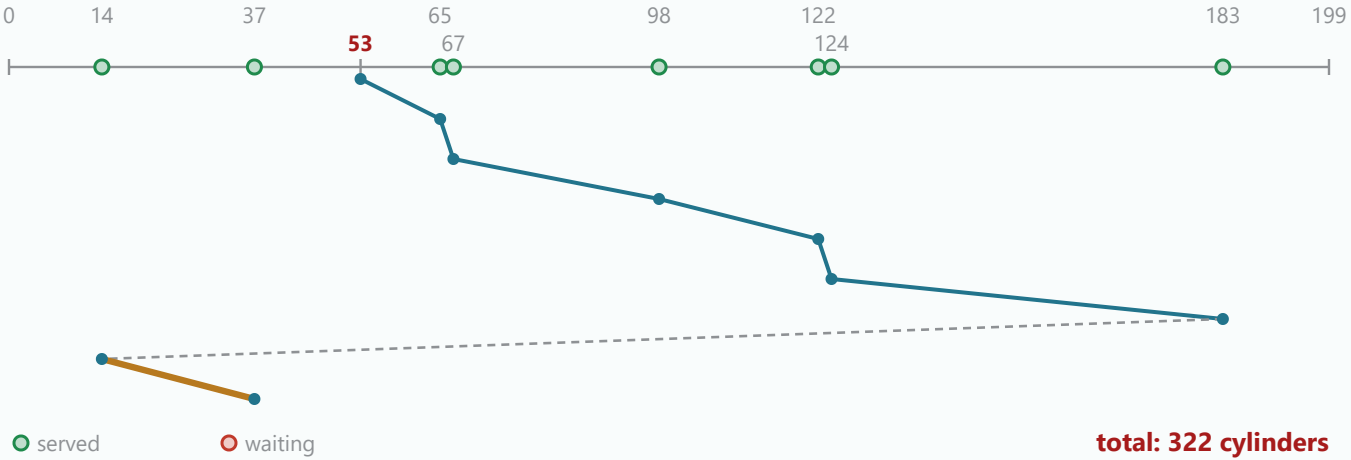
LOOK and C-LOOK



183 → 14: jumps to the lowest waiting request (169 cylinders), skipping 199 and 0.

Linux's old *deadline* and *CFQ* schedulers and today's *mq-deadline* dispatch in LBA order, which behaves much like C-LOOK.

LOOK and C-LOOK

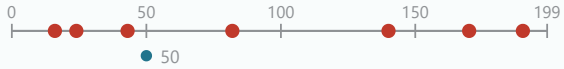


14 → 37: **total 322** (153 without the jump). A one-way sweep evens out waiting times.

Linux's old *deadline* and *CFQ* schedulers and today's *mq-deadline* dispatch in LBA order, which behaves much like C-LOOK.

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
5    for (int j = i; j < n; j++) {
6      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 0 cylinders

The same queue, sorted: s = 16, 24, 43, 82, 140, 170, 190. The head is at 50 and moving **up**.

Step 1 / 10

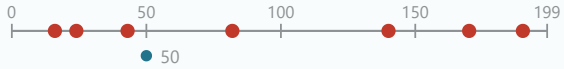
VARIABLES

`head = 50` `i = 0` `j = –` `total = 0`

OUTPUT

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
▶    while (i < n && s[i] < head) i++;
5    for (int j = i; j < n; j++) {
6      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 0 cylinders

i is the index of the first request above 50: s[3] = 82. The 16, 24, 43 below are taken on the way back.

Step 2 / 10

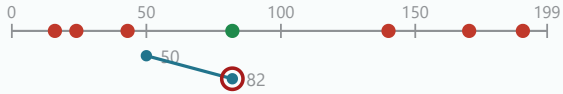
VARIABLES

`head = 50` `i = 3` `j = –` `total = 0`

OUTPUT

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
▶    for (int j = i; j < n; j++) {
▶      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 32 cylinders

j = 3: 82 (+32), total = 32.

Step 3 / 10

VARIABLES

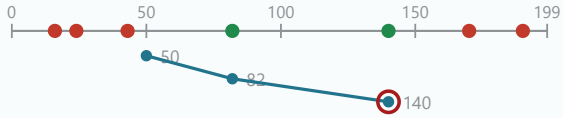
`head = 82` `i = 3` `j = 3` `total = 32`

OUTPUT

```
82
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
▶    for (int j = i; j < n; j++) {
▶      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 90 cylinders

j = 4: 140 (+58), total = 90.

Step 4 / 10

VARIABLES

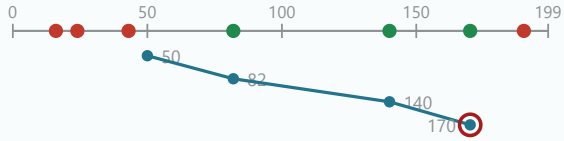
head = 140 | i = 3 | j = 4 | total = 90

OUTPUT

```
82 140
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
▶    for (int j = i; j < n; j++) {
▶      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 120 cylinders

j = 5: 170 (+30), total = 120.

Step 5 / 10

VARIABLES

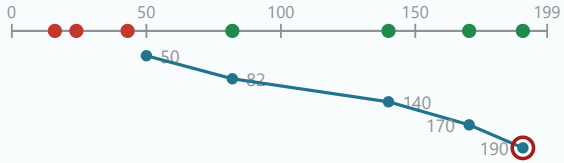
`head = 170` `i = 3` `j = 5` `total = 120`

OUTPUT

```
82 140 170
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
▶    for (int j = i; j < n; j++) {
▶      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 140 cylinders

j = 6: 190 (+20), total = 140. The last request going up: the arm turns back **without going** to 199 (unlike SCAN).

Step 6 / 10

VARIABLES

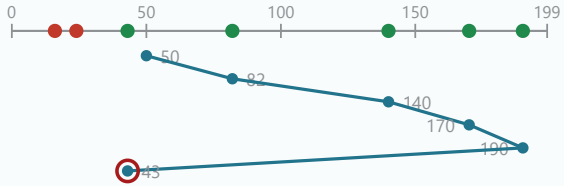
`head = 190` `i = 3` `j = 6` `total = 140`

OUTPUT

```
82 140 170 190
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
5    for (int j = i; j < n; j++) {
6      total += s[j] - head; head = s[j];
7    }
▶    for (int j = i - 1; j >= 0; j--) {
▶      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 287 cylinders

Direction reversed, j = 2: 190 → 43 (+147), total = 287. The longest move is on the way back.

Step 7 / 10

VARIABLES

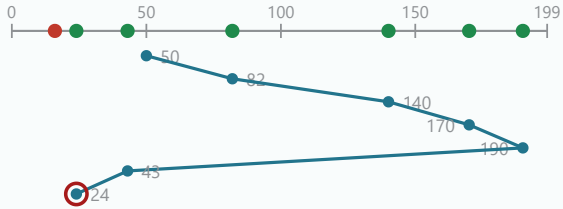
`head = 43` `i = 3` `j = 2` `total = 287`

OUTPUT

```
82 140 170 190 43
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
5    for (int j = i; j < n; j++) {
6      total += s[j] - head; head = s[j];
7    }
▶    for (int j = i - 1; j >= 0; j--) {
▶      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 306 cylinders

j = 1: 24 (+19), total = 306.

Step 8 / 10

VARIABLES

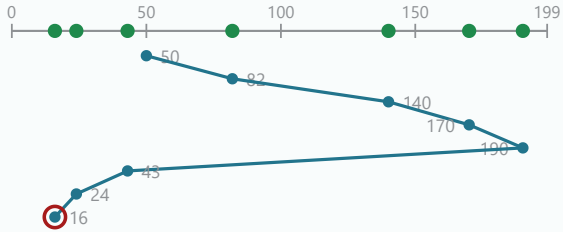
`head = 24` `i = 3` `j = 1` `total = 306`

OUTPUT

```
82 140 170 190 43 24
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
5    for (int j = i; j < n; j++) {
6      total += s[j] - head; head = s[j];
7    }
▶    for (int j = i - 1; j >= 0; j--) {
▶      total += head - s[j]; head = s[j];
10   }
11   return total;
12 }
```



total = 314 cylinders

j = 0: 16 (+8), total = 314.

Step 9 / 10

VARIABLES

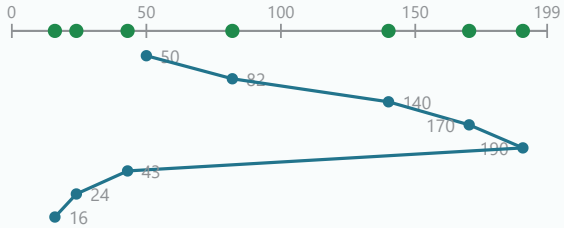
`head = 16` `i = 3` `j = 0` `total = 314`

OUTPUT

```
82 140 170 190 43 24 16
```

LOOK: Code Trace

```
1  /* s[] sorted; go up first, then down */
2  int look(int head, int s[], int n) {
3    int i = 0, total = 0;
4    while (i < n && s[i] < head) i++;
5    for (int j = i; j < n; j++) {
6      total += s[j] - head; head = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      total += head - s[j]; head = s[j];
10   }
▶    return total;
12 }
```



total = 314 cylinders

314 in all; SSTF gets 208 on the same queue. Had the head been moving down, LOOK would also give 34 + 174 = 208: the direction decides the result, but no request starves.

Step 10 / 10

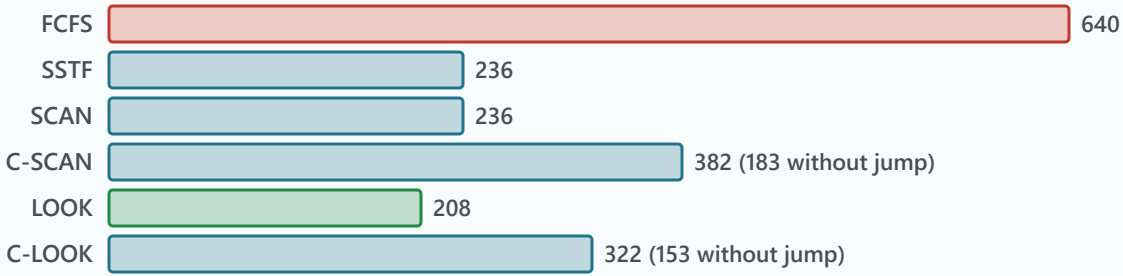
VARIABLES

`head = 16` `i = 3` `j = -1` `total = 314`

OUTPUT

```
82 140 170 190 43 24 16
```

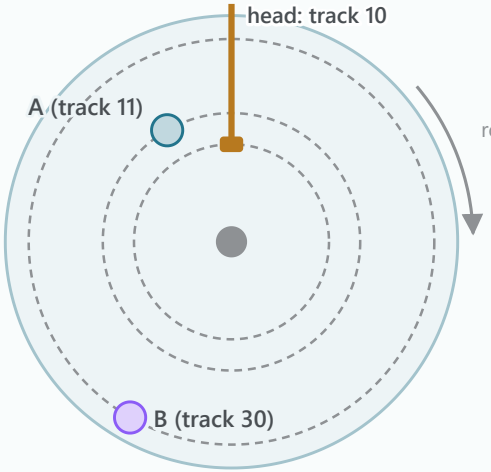
Comparing the Algorithms



Tanenbaum's example: head 11; 1, 36, 16, 34, 9, 12	Movement
FCFS: 10 + 35 + 20 + 18 + 25 + 3	**111**
SSTF: 12, 9, 16, 1, 34, 36 → 1 + 3 + 7 + 15 + 33 + 2	**61**
Elevator (up): 12, 16, 34, 36, 9, 1 → 1 + 4 + 18 + 2 + 27 + 8	**60**

On the charted queue, SCAN starting upward would give 331: the direction changes the result. Tanenbaum's elevator does not go to the edge, so it is really LOOK.

SPTF: Counting Rotational Latency Too



7200 rpm: 1 turn = 8.33 ms; the disk turns 43.2° per ms

A: 1 ms seek, sector reaches the head after 30°

B: 3 ms seek, sector reaches the head after 150°

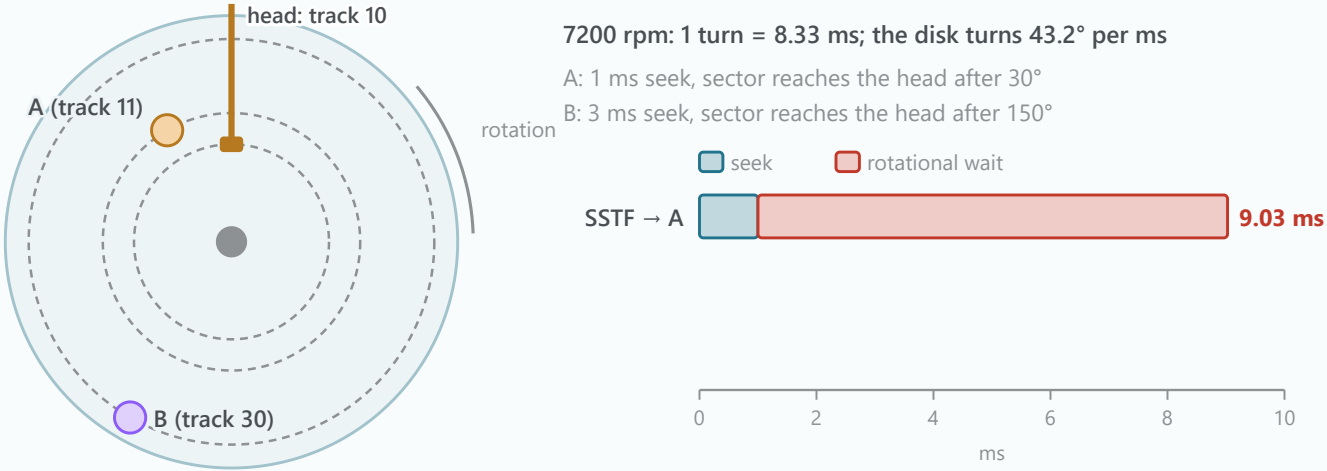


The head is on track 10 with two requests waiting. SSTF, which looks only at **track distance**, sees A as far closer: one track away. But time is set by **seek + rotation**, not seek alone.

Step 1 / 3

SPTF (shortest positioning time first) minimizes positioning time, but the OS does not know sector angles, so it is implemented inside the disk via NCQ. Against starvation, **F-SCAN** freezes the queue when a sweep starts; new requests wait for the next sweep.

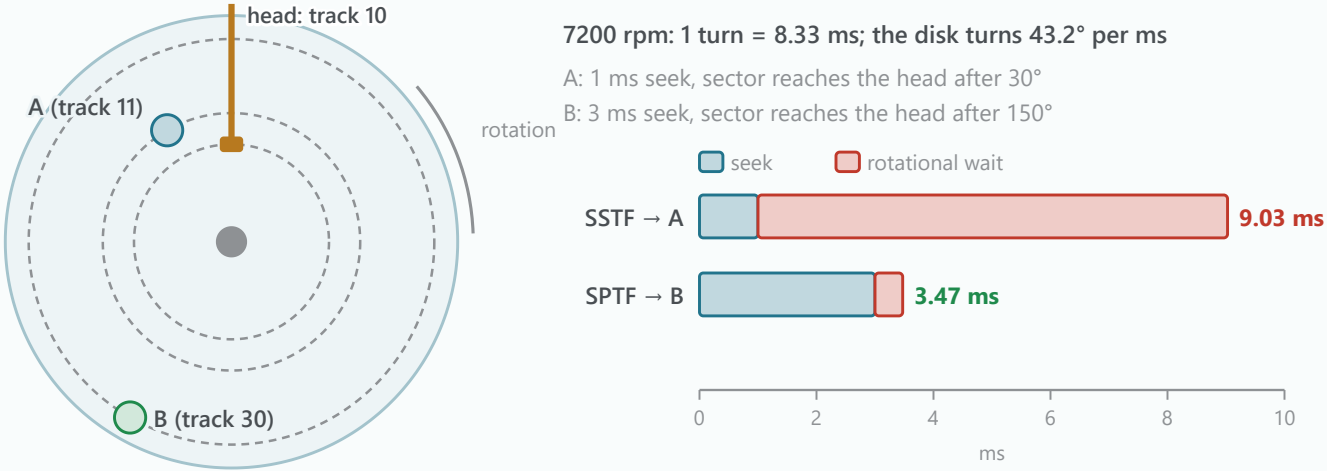
SPTF: Counting Rotational Latency Too



SSTF picks A: during the 1 ms seek the disk turns 43.2°, so the sector that was 30° away is 13.2° past the head when the seek ends. Almost a full turn of waiting: 1 + 8.03 = **9.03 ms**.

SPTF (shortest positioning time first) minimizes positioning time, but the OS does not know sector angles, so it is implemented inside the disk via NCQ. Against starvation, **F-SCAN** freezes the queue when a sweep starts; new requests wait for the next sweep.

SPTF: Counting Rotational Latency Too



SPTF picks B: during the 3 ms seek the disk turns 129.6°, leaving 20.4° to B: 3 + 0.47 = **3.47 ms**. The far track finished 2.6 times sooner than the near one.

SPTF (shortest positioning time first) minimizes positioning time, but the OS does not know sector angles, so it is implemented inside the disk via NCQ. Against starvation, **F-SCAN** freezes the queue when a sweep starts; new requests wait for the next sweep.

Controller Cache and Bad Sectors

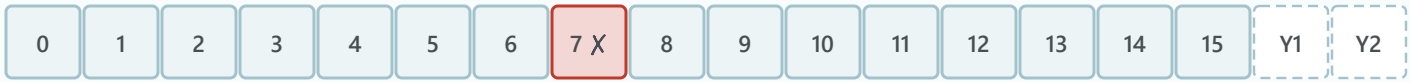
Controller cache

The disk keeps a track cache in its own DRAM (≈ 256 MB): it also reads unrequested blocks passing under the head. This is separate from the OS cache.

Write cache and NCQ

Writes may sit in the cache: durability needs `FLUSH` /FUA. NCQ orders 32 requests by rotational position.

(a) track: sector 7 bad, two spares



A sector that fails in manufacturing or in use: written data cannot be read back. The controller tracks the factory list and those found in use (SMART: *Reallocated Sectors*).

Controller Cache and Bad Sectors

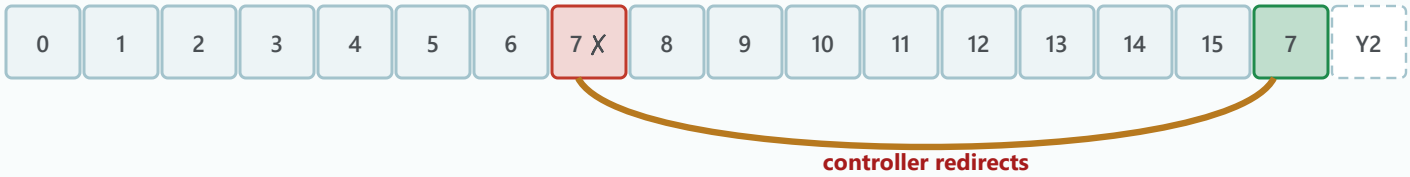
Controller cache

The disk keeps a track cache in its own DRAM (≈ 256 MB): it also reads unrequested blocks passing under the head. This is separate from the OS cache.

Write cache and NCQ

Writes may sit in the cache: durability needs `FLUSH` /FUA. NCQ orders 32 requests by rotational position.

(b) bad sector remapped to a spare



Method 1: the bad sector is **remapped** to a spare. Simple, but a sequential read through sector 7 may cost an extra revolution.

Controller Cache and Bad Sectors

Controller cache

The disk keeps a track cache in its own DRAM (≈ 256 MB): it also reads unrequested blocks passing under the head. This is separate from the OS cache.

Write cache and NCQ

Writes may sit in the cache: durability needs `FLUSH` /FUA. NCQ orders 32 requests by rotational position.

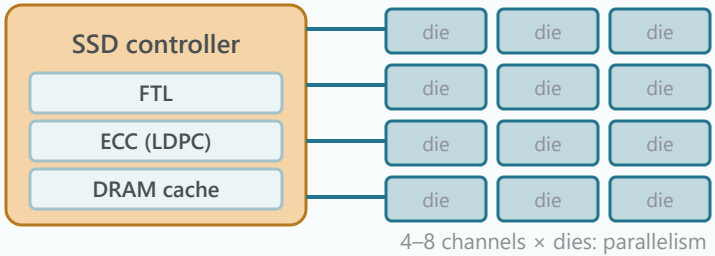
(c) sectors shifted, bad one skipped

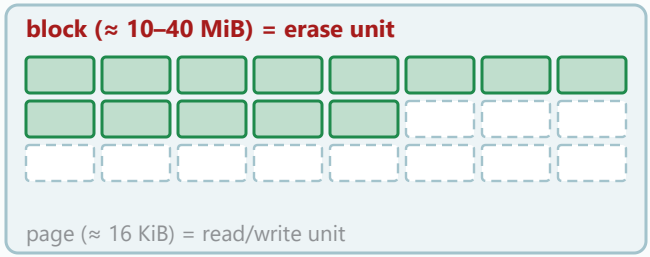


sectors 8 … 15 shift by one, one spare used; sequential reads stay intact

Method 2: the sectors are **shifted**; the whole track must be rewritten, but order is preserved. The OS can also keep a bad-block list (`badblocks` , the file system).

SSD: NAND Flash

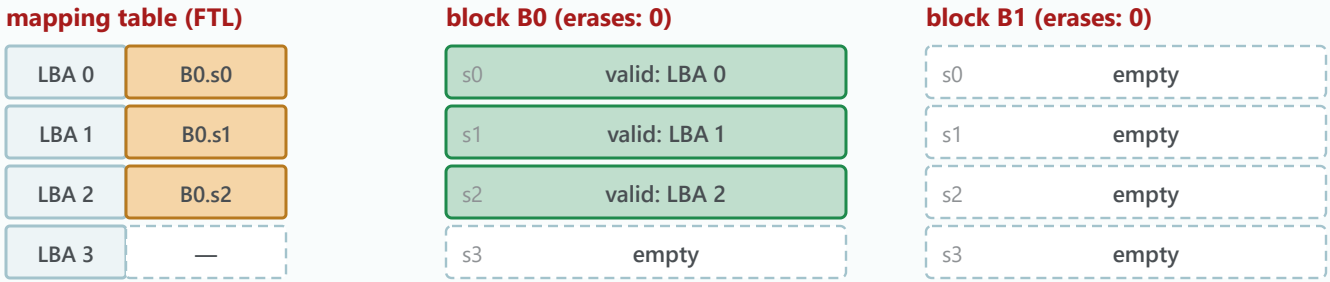




Cell	Bits/cell	Typical P/E cycles	Use
SLC	1	≈ 50–100k	cache region, industrial
MLC	2	≈ 3–10k	older enterprise
TLC	3	≈ 1–3k	most consumer and server SSDs
QLC	4	≈ 0.1–1k	high capacity, read-heavy

- No moving parts: **no seek and no rotational latency**, so random access ≈ sequential access.
- No in-place update: *erase, then write*; each cell survives a limited number of erases. 3D NAND has 200–300+ layers.
- The price-per-GB gap has narrowed; HDDs are still cheaper for archives and capacity.

FTL, Garbage Collection, TRIM, Wear Leveling

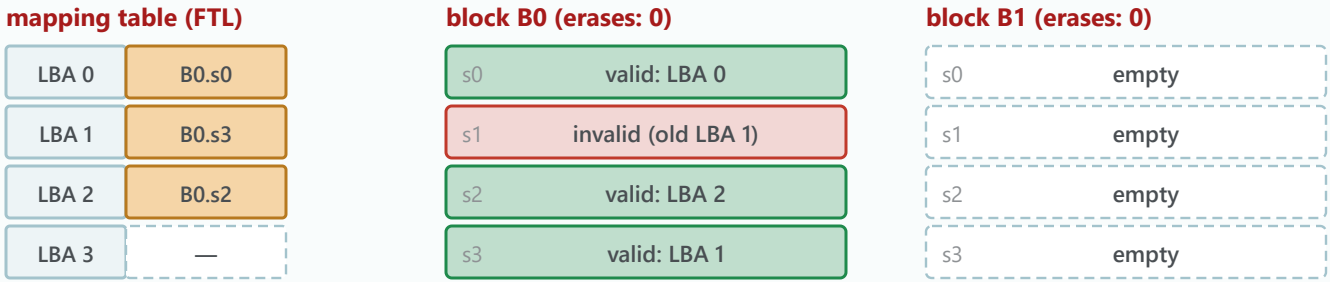


writes go only to erased (free) pages; erase is per block

Three logical blocks are written; the FTL places them in B0's free pages and records them in the table.

Windows “Optimize” does not defragment an SSD; it sends TRIM. Defragmenting an SSD only adds writes and wear. Aligning partitions to a 1 MiB boundary keeps logical blocks from straddling flash pages.

FTL, Garbage Collection, TRIM, Wear Leveling



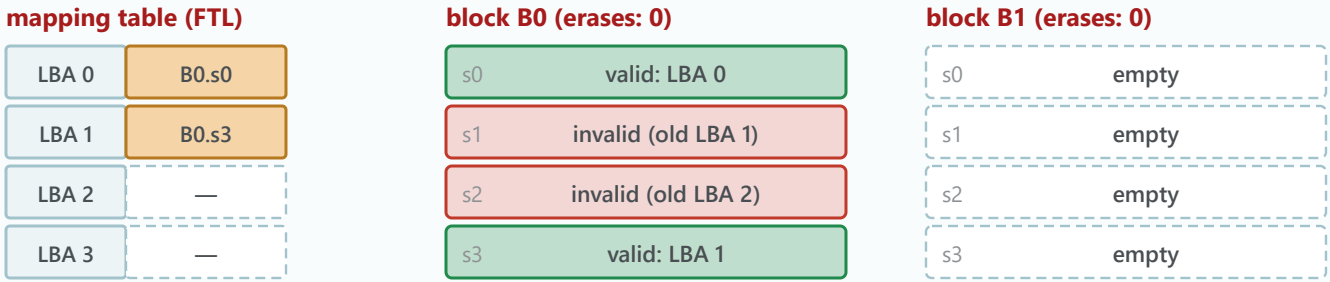
LBA 1 updated: not in place

writes go only to erased (free) pages; erase is per block

LBA 1 is rewritten: the old page cannot be overwritten. The new data goes to a free page and the old one becomes **invalid** (out-of-place update).

Windows “Optimize” does not defragment an SSD; it sends TRIM. Defragmenting an SSD only adds writes and wear. Aligning partitions to a 1 MiB boundary keeps logical blocks from straddling flash pages.

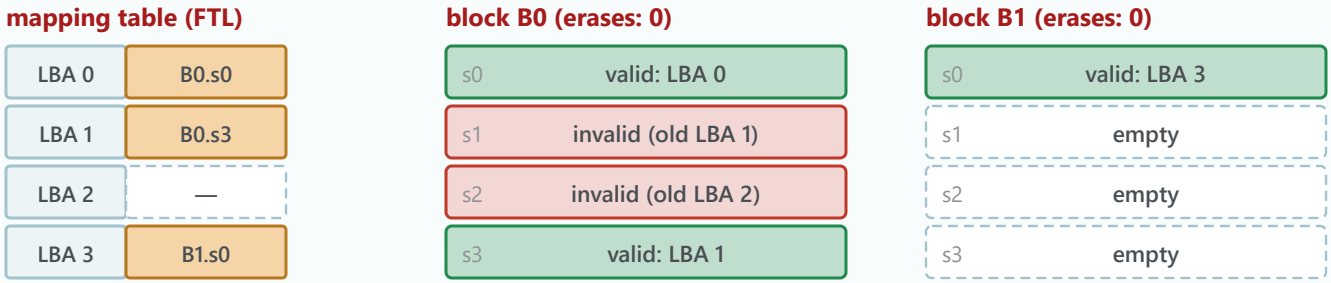
FTL, Garbage Collection, TRIM, Wear Leveling



When a file is deleted the file system sends **TRIM** (NVMe: Deallocate), so the SSD learns the page is no longer needed. Without TRIM, garbage collection would copy this data for nothing.

Windows “Optimize” does not defragment an SSD; it sends TRIM. Defragmenting an SSD only adds writes and wear. Aligning partitions to a 1 MiB boundary keeps logical blocks from straddling flash pages.

FTL, Garbage Collection, TRIM, Wear Leveling



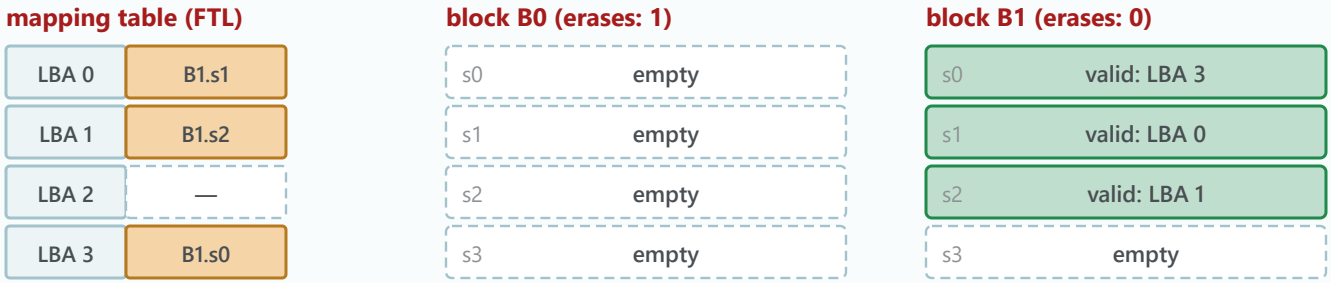
B0 full, LBA 3 goes to B1

writes go only to erased (free) pages; erase is per block

B0 has no free pages left; new writes go to B1. Half of B0 is invalid.

Windows “Optimize” does not defragment an SSD; it sends TRIM. Defragmenting an SSD only adds writes and wear. Aligning partitions to a 1 MiB boundary keeps logical blocks from straddling flash pages.

FTL, Garbage Collection, TRIM, Wear Leveling



garbage collection: 2 valid pages moved, B0 erased

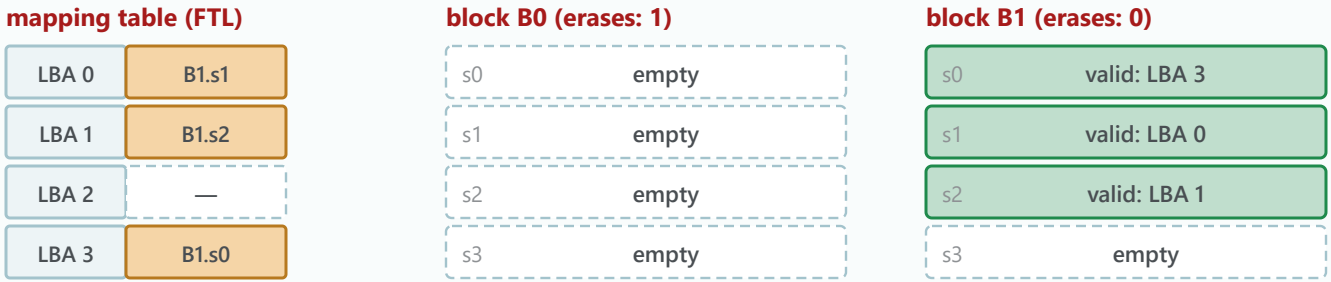
writes go only to erased (free) pages; erase is per block

Garbage collection: B0's valid pages (LBA 0, 1) are copied to B1 and B0 is erased. The 2 extra page writes are **write amplification**.

Step 5 / 6

Windows “Optimize” does not defragment an SSD; it sends TRIM. Defragmenting an SSD only adds writes and wear. Aligning partitions to a 1 MiB boundary keeps logical blocks from straddling flash pages.

FTL, Garbage Collection, TRIM, Wear Leveling



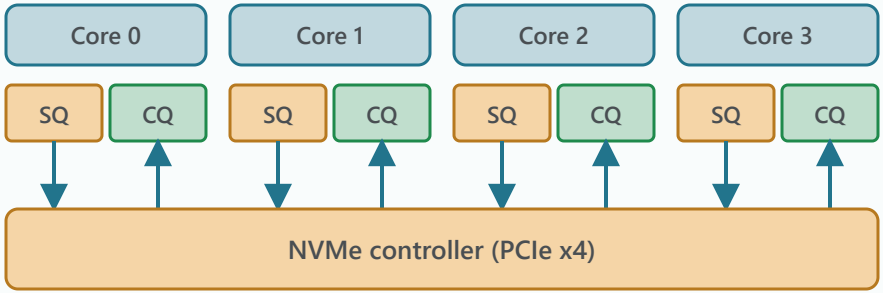
wear leveling: rarely erased blocks get used too

writes go only to erased (free) pages; erase is per block

Wear leveling: the FTL evens out erase counts, moving long-unchanged (cold) data too so that all blocks age equally. Spare area (over-provisioning) makes garbage collection easier.

Windows “Optimize” does not defragment an SSD; it sends TRIM. Defragmenting an SSD only adds writes and wear. Aligning partitions to a 1 MiB boundary keeps logical blocks from straddling flash pages.

NVMe and Multiple Queues



AHCI (SATA)

one queue · 32 commands · lock

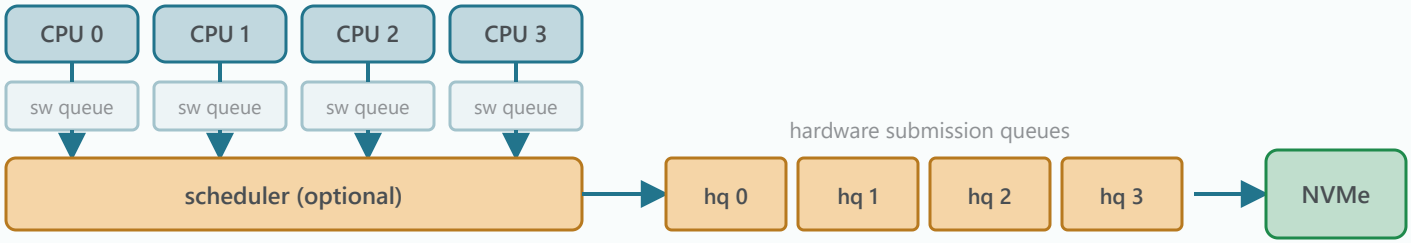
NVMe

65,535 queues × 65,536 commands

a queue per core: lock-free submission

- NVMe is a **protocol** designed for flash on PCIe; M.2 is a *form factor* (an M.2 card can also be SATA). U.2/U.3 and E1.S/E3.S are cabled/slotted server forms.
- Queue capacity: 64K queues × 64K commands per queue. Every OS has shipped a driver since 2013; the incompatibility with AHCI is at the protocol level only.
- Latency ≈ 10–100 µs; PCIe 5.0 x4 drives reach ≈ 14 GB/s and > 2 million IOPS. NVMe-oF runs it over a network; ZNS adds zoned namespaces.

Linux blk-mq and I/O Schedulers



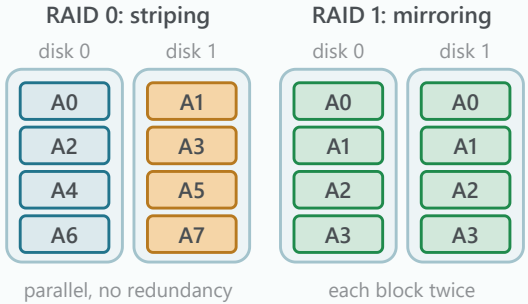
Scheduler	Idea	Best for
`none`	No sorting, dispatch directly	NVMe (default), fast SSDs
`mq-deadline`	Sorted by LBA + read/write deadlines	SATA HDD/SSD (default on most distros)
`bfq`	Per-process budgets, fair sharing	Desktops, interactivity on slow devices
`kyber`	Throttling against read/write latency targets	Fast multiqueue devices

```
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ echo bfq | sudo tee /sys/block/sda/queue/scheduler
```

Per-CPU software queues avoid contention on a single queue lock at millions of IOPS; Linux 5.0 removed the old single-queue block layer (noop, deadline, CFQ). An SSD has no seeks, so the scheduler's job is not ordering but protecting fairness and latency.

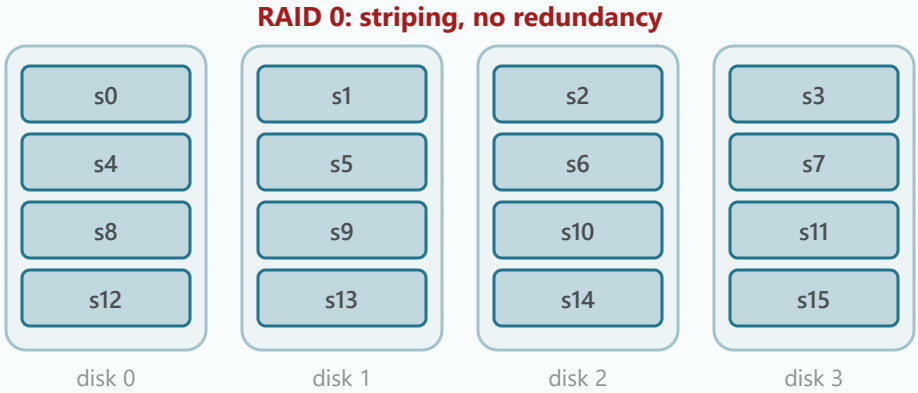
RAID: Combining Disks

- **RAID** (Patterson, Gibson, Katz, 1988) presents many disks to the OS as *one disk*; **striping** gives parallelism, **mirroring** and **parity** give redundancy. The opposite is SLED: a single large expensive disk.
- Today the acronym reads *Redundant Array of **Independent** Disks*. SCSI disks were expensive; now SAS expanders attach hundreds of disks, and the standard levels are 0–6.
- Hardware RAID cards or software RAID (Linux `mdadm`, ZFS RAID-Z, Btrfs, Windows Storage Spaces). **RAID is not a backup**: a deleted file is deleted on every disk.



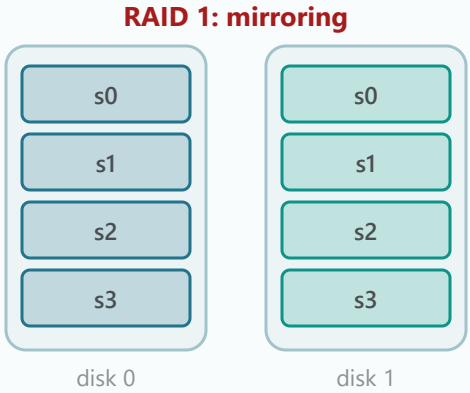
4 × 4 TB disks	Usable	Survives	Small random-write cost
RAID 0 (striping)	16 TB	0 disks	1 I/O
RAID 1 (4-way mirror)	4 TB	3 disks	4 I/Os (one per copy)
RAID 10 (mirror + stripe)	8 TB	1 (2 if in different pairs)	2 I/Os
RAID 5 (distributed parity)	12 TB	1 disk	4 I/Os (2 reads + 2 writes)
RAID 6 (dual parity)	8 TB	2 disks	6 I/Os (3 reads + 3 writes)

RAID Levels



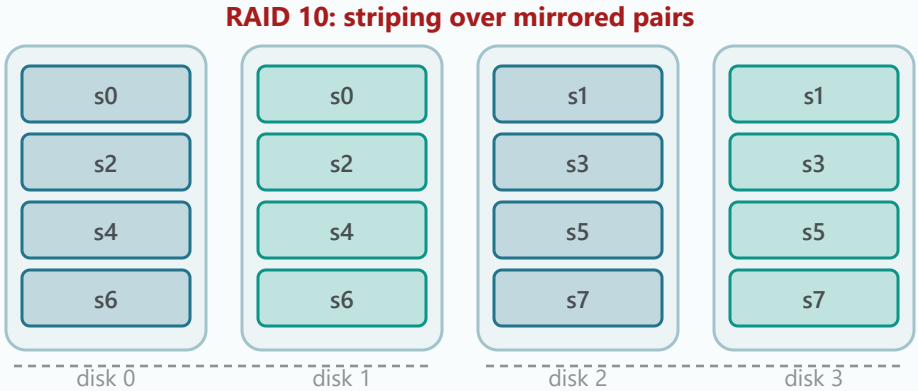
RAID 0: consecutive stripes go to different disks; n times the bandwidth for large requests. Lose one disk and you lose **everything**.

RAID Levels



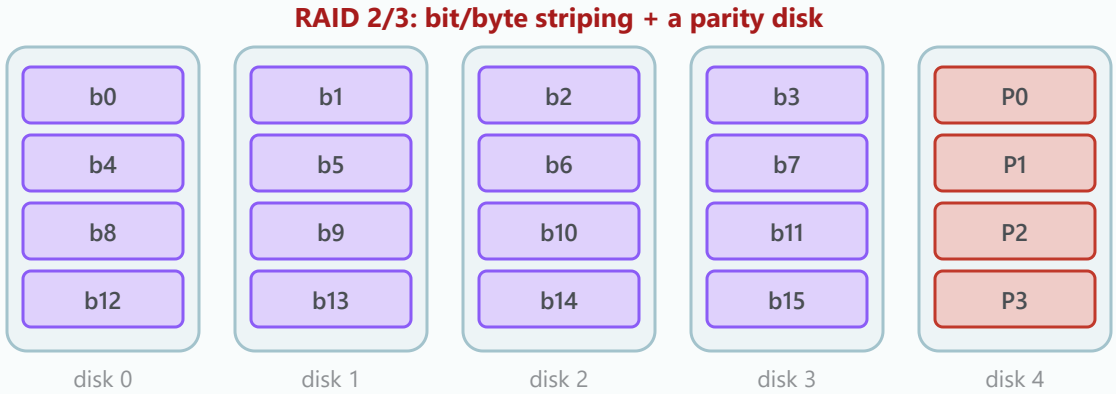
RAID 1: every write happens twice; a read comes from either disk (load is spread). Half the capacity is redundancy.

RAID Levels



RAID 10: mirror first, then stripe. Common for databases: fast writes and fast rebuilds.

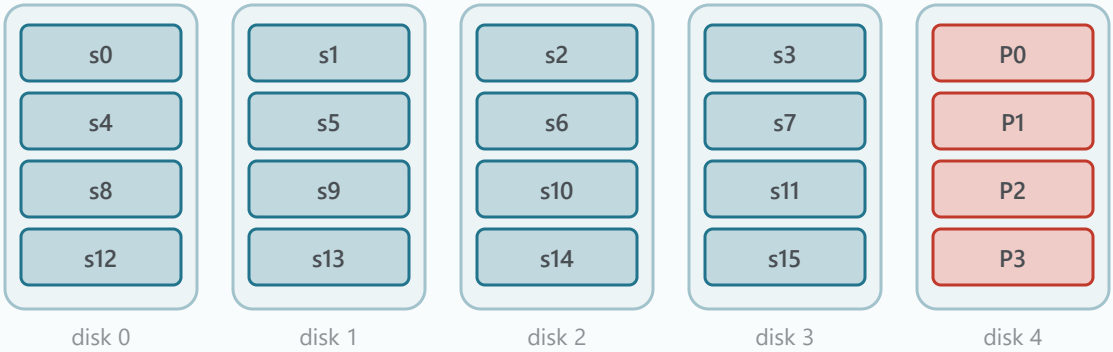
RAID Levels



RAID 2 uses words + a Hamming code, **RAID 3** bytes + a single parity disk; the disks must spin in sync. Neither is used today (disks do their own ECC).

RAID Levels

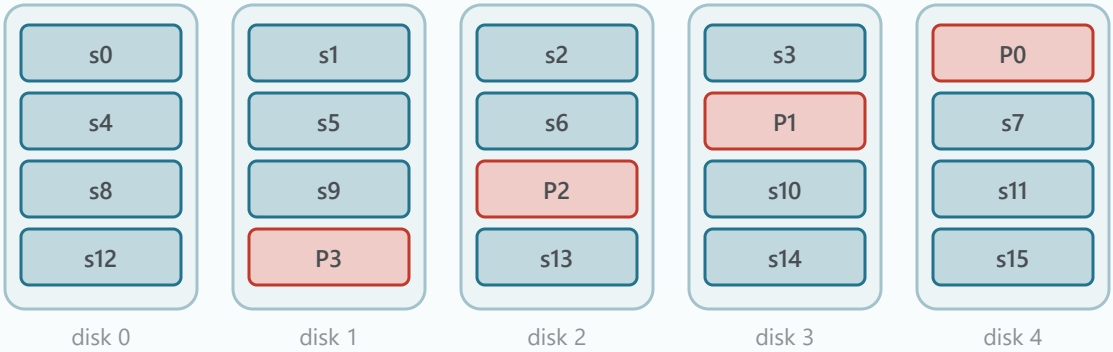
RAID 4: block striping + a dedicated parity disk



RAID 4: block stripes + a dedicated parity disk. Every small write updates the parity disk, which becomes the **bottleneck**.

RAID Levels

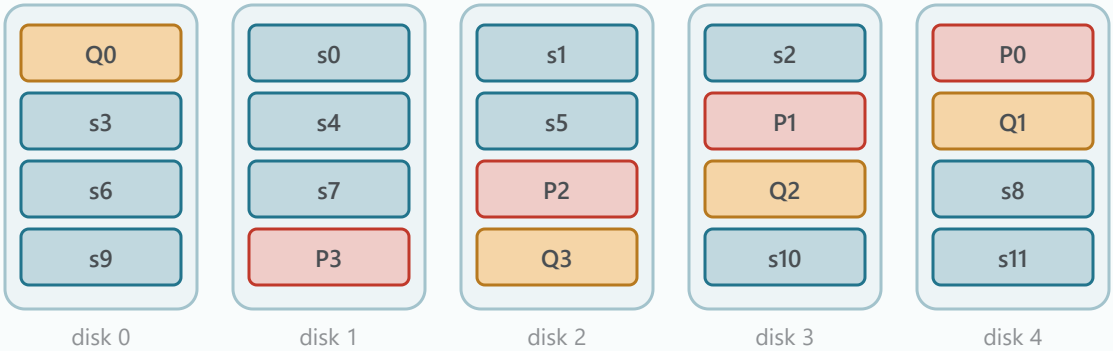
RAID 5: parity spread across the disks



RAID 5: parity blocks rotate across the disks, removing the bottleneck. Capacity of n − 1 disks; survives one disk failure.

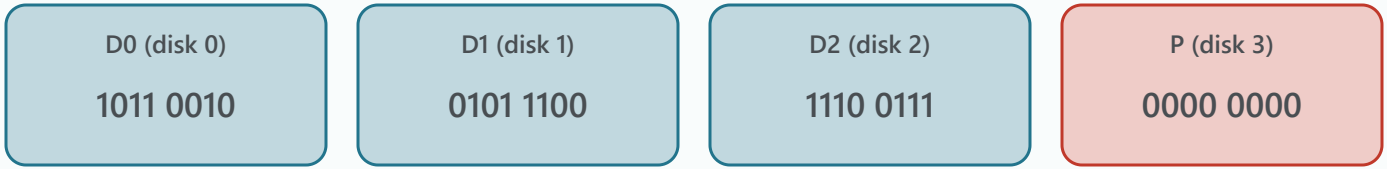
RAID Levels

RAID 6: P and Q parity spread across the disks



RAID 6: P (XOR) + Q (Reed–Solomon, GF(2^8)) parity; survives two disk failures. Preferred for large disks, where a rebuild takes hours.

Parity and XOR: An Example



Three data blocks in a stripe (an 8-bit example here). The parity block is not computed yet.

Rebuild risk: a consumer disk has an unrecoverable read error (URE) about once per 10^{14} bits. Rebuilding 12 TB gives $12\times10^{12}\times8 / 10^{14} \approx$ **0.96** expected errors, a ≈ 62% chance of at least one: a RAID 5 rebuild will likely fail, so large arrays use RAID 6.

Parity and XOR: An Example

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	0101 1100	1110 0111	0000 1001

P = D0 ⊕ D1 ⊕ D2 = 0000 1001

Parity is a bitwise XOR: each bit position is completed so the number of 1s is even.

Rebuild risk: a consumer disk has an unrecoverable read error (URE) about once per 10^{14} bits. Rebuilding 12 TB gives $12\times10^{12}\times8 / 10^{14} \approx$ **0.96** expected errors, a ≈ 62% chance of at least one: a RAID 5 rebuild will likely fail, so large arrays use RAID 6.

Parity and XOR: An Example

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	✗ failed	1110 0111	0000 1001

D1 = D0 ⊕ D2 ⊕ P = ?

Disk 1 failed. Since $x \oplus x = 0$, the lost block is the XOR of the others.

Rebuild risk: a consumer disk has an unrecoverable read error (URE) about once per 10^{14} bits. Rebuilding 12 TB gives $12 \times 10^{12} \times 8 / 10^{14} \approx$ **0.96** expected errors, a ≈ 62% chance of at least one: a RAID 5 rebuild will likely fail, so large arrays use RAID 6.

Parity and XOR: An Example

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	0101 1100	1110 0111	0000 1001

D1 = 1011 0010 ⊕ 1110 0111 ⊕ 0000 1001 = 0101 1100

The recovered value 0101 1100 equals the original D1. Every stripe is rewritten this way onto a new disk (the rebuild).

Rebuild risk: a consumer disk has an unrecoverable read error (URE) about once per 10^{14} bits. Rebuilding 12 TB gives $12\times10^{12}\times8 / 10^{14} \approx$ **0.96** expected errors, a ≈ 62% chance of at least one: a RAID 5 rebuild will likely fail, so large arrays use RAID 6.

Parity and XOR: An Example

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	0000 1111	1110 0111	0101 1010

$P' = P \oplus D1 \oplus D1' = 0101\ 1010$

Small write: D1' = 0000 1111. Read the old D1 and P, compute P', write D1' and P': **4 I/Os** (read-modify-write).

Rebuild risk: a consumer disk has an unrecoverable read error (URE) about once per 10^{14} bits. Rebuilding 12 TB gives $12 \times 10^{12} \times 8 / 10^{14} \approx$ **0.96** expected errors, a ≈ 62% chance of at least one: a RAID 5 rebuild will likely fail, so large arrays use RAID 6.

RAID 5 Small Write and Recovery: Code Trace

```
▶ void small_write(int d, uint8_t nv) {
2   uint8_t ov = rd(d);       /* I/O 1 */
3   uint8_t p = rd(P);        /* I/O 2 */
4   p = p ^ ov ^ nv;
5   wr(d, nv);                /* I/O 3 */
6   wr(P, p);                 /* I/O 4 */
7 }
8 uint8_t rebuild(int bad) {
9   uint8_t x = 0;
10  for (int d = 0; d <= P; d++)
11    if (d != bad)
12      x ^= rd(d);           /* incl. P */
13  return x;
14 }
```



disk I/Os: 0

In the stripe P = D0 ⊕ D1 ⊕ D2 = 0001 1110. D1 is to be overwritten with 0011 0011: `small_write(1, ...)`.

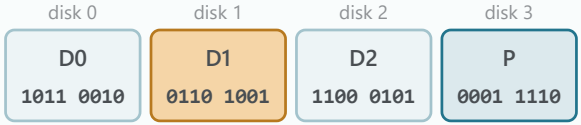
Step 1 / 11

VARIABLES

`d = 1` `nv = 0011 0011` `ov = –` `p = –`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
▶    uint8_t ov = rd(d);        /* I/O 1 */
3    uint8_t p = rd(P);         /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                 /* I/O 3 */
6    wr(P, p);                  /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);            /* incl. P */
13   return x;
14 }
```



disk I/Os: 1

The old D1 is read. The other data disks (D0, D2) are not: only 1 of the 3 data disks in the stripe changes.

Step 2 / 11

VARIABLES

`d = 1` `nv = 0011 0011` `ov = 0110 1001` `p = –`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);        /* I/O 1 */
▶    uint8_t p = rd(P);         /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                 /* I/O 3 */
6    wr(P, p);                  /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);            /* incl. P */
13   return x;
14 }
```



disk I/Os: 2

The old parity is read: p = 0001 1110.

Step 3 / 11

VARIABLES

`d = 1` `nv = 0011 0011`

`ov = 0110 1001` `p = 0001 1110`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
▶    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
6    wr(P, p);                 /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);           /* incl. P */
13   return x;
14 }
```



`0001 1110 ⊕ 0110 1001 ⊕ 0011 0011 = 0100 0100`

disk I/Os: 2

p ⊕ ov removes the old D1 from the parity (x ⊕ x = 0); ⊕ nv adds the new value. p = 0100 0100.

Step 4 / 11

VARIABLES

`d = 1` `nv = 0011 0011`

`ov = 0110 1001` `p = 0100 0100`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);        /* I/O 1 */
3    uint8_t p = rd(P);         /* I/O 2 */
4    p = p ^ ov ^ nv;
▶    wr(d, nv);                 /* I/O 3 */
6    wr(P, p);                  /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);            /* incl. P */
13   return x;
14 }
```



disk I/Os: 3

The new D1 is written.

Step 5 / 11

VARIABLES

`d = 1` `nv = 0011 0011`

`ov = 0110 1001` `p = 0100 0100`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
▶    wr(P, p);                 /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);           /* incl. P */
13   return x;
14 }
```



disk I/Os: 4

The new P is written: **4 I/Os** (2 reads + 2 writes). A crash between these two writes leaves the stripe inconsistent (the “write hole”).

Step 6 / 11

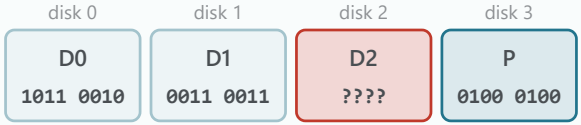
VARIABLES

`d = 1` `nv = 0011 0011`

`ov = 0110 1001` `p = 0100 0100`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
6    wr(P, p);                 /* I/O 4 */
7  }
▶ uint8_t rebuild(int bad) {
▶   uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);           /* incl. P */
13   return x;
14 }
```



disk I/Os: 0

Disk 2 failed: D2 cannot be read. `rebuild(2)` computes it as the XOR of the rest.

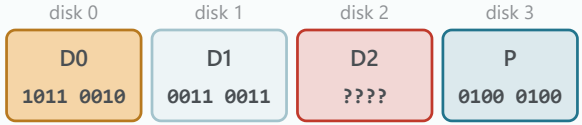
Step 7 / 11

VARIABLES

`bad = 2` `d = –` `x = 0000 0000`

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
6    wr(P, p);                 /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
▶    for (int d = 0; d <= P; d++)
▶      if (d != bad)
▶        x ^= rd(d);           /* incl. P */
13   return x;
14 }
```



x = 1011 0010

disk I/Os: 1

d = 0: x ^= D0 → x = 1011 0010.

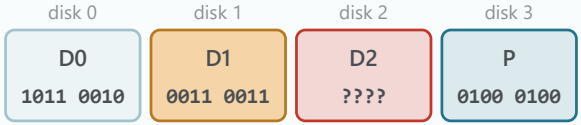
Step 8 / 11

VARIABLES

bad = **2** d = **0** x = **1011 0010**

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
6    wr(P, p);                 /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
▶    for (int d = 0; d <= P; d++)
▶      if (d != bad)
▶        x ^= rd(d);           /* incl. P */
13   return x;
14 }
```



x = 1000 0001

disk I/Os: 2

d = 1: x ^= D1 → x = 1000 0001.

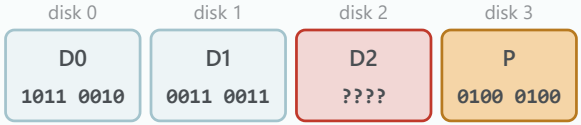
Step 9 / 11

VARIABLES

bad = **2** d = **1** x = **1000 0001**

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
6    wr(P, p);                 /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
▶    for (int d = 0; d <= P; d++)
▶      if (d != bad)
▶        x ^= rd(d);           /* incl. P */
13   return x;
14 }
```



x = 1100 0101

disk I/Os: 3

d = 3: x ^= P → x = 1100 0101. (d = 2 is skipped.)

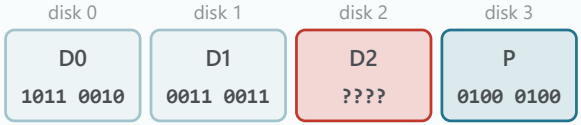
Step 10 / 11

VARIABLES

bad = **2** d = **3** x = **1100 0101**

RAID 5 Small Write and Recovery: Code Trace

```
1  void small_write(int d, uint8_t nv) {
2    uint8_t ov = rd(d);       /* I/O 1 */
3    uint8_t p = rd(P);        /* I/O 2 */
4    p = p ^ ov ^ nv;
5    wr(d, nv);                /* I/O 3 */
6    wr(P, p);                 /* I/O 4 */
7  }
8  uint8_t rebuild(int bad) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bad)
12       x ^= rd(d);           /* incl. P */
▶    return x;
14 }
```



D2 = 1100 0101

disk I/Os: 3

x = 1100 0101, the same as the lost D2. Each stripe is rebuilt onto a new disk with 3 reads; the array is unprotected meanwhile.

Step 11 / 11

VARIABLES

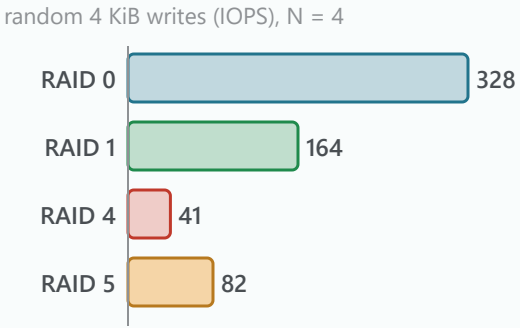
bad = **2** d = **4** x = **1100 0101**

RAID Performance: An Example

Single disk (7200 rpm, 8 ms seek, 200 MB/s): random 4 KiB = 12.19 ms → **R ≈ 82 IOPS** (0.34 MB/s); a sequential 10 MiB chunk = 8 + 4.17 + 52.43 = 64.6 ms → **S ≈ 162 MB/s**. Array: N = 4 disks. Assumes a constant media rate and a fresh seek + rotation per chunk; a long sequential read seeks once and S approaches the media rate (200 MB/s).

Level	Capacity	Sequential read/write	Random read	Random write	Write latency
RAID 0	N = 4	N·S ≈ 649 MB/s	N·R ≈ 328	N·R ≈ 328	D
RAID 1 (two-way mirror)	N/2 = 2	(N/2)·S ≈ 325 MB/s	N·R ≈ 328	(N/2)·R ≈ 164	D (slowest copy)
RAID 4	N − 1 = 3	(N−1)·S ≈ 487 MB/s	(N−1)·R ≈ 246	**R/2 ≈ 41**	2D
RAID 5	N − 1 = 3	(N−1)·S ≈ 487 MB/s	N·R ≈ 328	(N/4)·R ≈ 82	2D

- **Sequential reads on a mirror** get only (N/2)·S: the two copies split the blocks, but each disk still rotates under the blocks it skips.
- **RAID 4**: every small write needs 1 read + 1 write on the parity disk; the array drops to half the speed of one disk, and adding disks does not help.
- **RAID 5**: the same 4 I/Os spread over all disks, so N·R/4 grows with the disk count. A full-stripe write reads no old data: (N−1)·S.



Stable Storage

- **Stable storage**: a write either completes correctly or leaves the old data in place. Assumptions: the same block failing on both disks at once is negligible; bad data is detected by ECC; if the CPU crashes, the write in progress stops.
- **Stable write**: write disk 1, read it back, verify (retry n times, use a spare sector if needed); then do the same on disk 2. **Stable read**: if disk 1 cannot be read with valid ECC, read disk 2.



(a) Crash before writing disk 1: both copies are old. The write never happened.

RAID protects against failing sectors, not against a crash mid-write. Today the same idea appears as journaling, write barriers/FUA and the power-loss capacitors in enterprise SSDs.

Stable Storage

- **Stable storage**: a write either completes correctly or leaves the old data in place. Assumptions: the same block failing on both disks at once is negligible; bad data is detected by ECC; if the CPU crashes, the write in progress stops.
- **Stable write**: write disk 1, read it back, verify (retry n times, use a spare sector if needed); then do the same on disk 2.
  Stable read: if disk 1 cannot be read with valid ECC, read disk 2.



(b) Crash while writing disk 1: disk 1 is bad (ECC). The old value is restored from disk 2.

RAID protects against failing sectors, not against a crash mid-write. Today the same idea appears as journaling, write barriers/FUA and the power-loss capacitors in enterprise SSDs.

Stable Storage

- **Stable storage**: a write either completes correctly or leaves the old data in place. Assumptions: the same block failing on both disks at once is negligible; bad data is detected by ECC; if the CPU crashes, the write in progress stops.
- **Stable write**: write disk 1, read it back, verify (retry n times, use a spare sector if needed); then do the same on disk 2. **Stable read**: if disk 1 cannot be read with valid ECC, read disk 2.



(c) Disk 1 written, crash before disk 2: the new value is completed.

RAID protects against failing sectors, not against a crash mid-write. Today the same idea appears as journaling, write barriers/FUA and the power-loss capacitors in enterprise SSDs.

Stable Storage

- **Stable storage**: a write either completes correctly or leaves the old data in place. Assumptions: the same block failing on both disks at once is negligible; bad data is detected by ECC; if the CPU crashes, the write in progress stops.
- **Stable write**: write disk 1, read it back, verify (retry n times, use a spare sector if needed); then do the same on disk 2.
  Stable read: if disk 1 cannot be read with valid ECC, read disk 2.

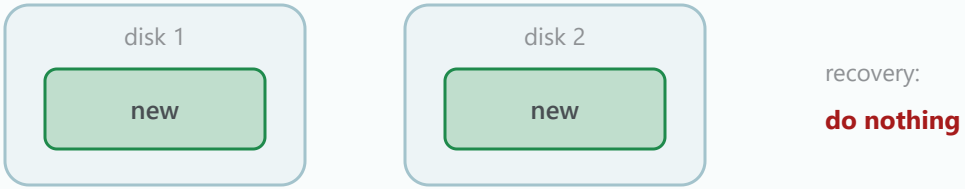


(d) Crash while writing disk 2: disk 2 is bad, so the new value is copied from disk 1.

RAID protects against failing sectors, not against a crash mid-write. Today the same idea appears as journaling, write barriers/FUA and the power-loss capacitors in enterprise SSDs.

Stable Storage

- **Stable storage**: a write either completes correctly or leaves the old data in place. Assumptions: the same block failing on both disks at once is negligible; bad data is detected by ECC; if the CPU crashes, the write in progress stops.
- **Stable write**: write disk 1, read it back, verify (retry n times, use a spare sector if needed); then do the same on disk 2.
  Stable read: if disk 1 cannot be read with valid ECC, read disk 2.



(e) Both copies new: the write is complete. In every case the result is either entirely old or entirely new.

RAID protects against failing sectors, not against a crash mid-write. Today the same idea appears as journaling, write barriers/FUA and the power-loss capacitors in enterprise SSDs.

Summary

Topic	Key idea	Today
Hardware access	Controller registers; port- or memory-mapped	MMIO through PCIe BARs
Transfer	Polling → interrupts → DMA	MSI-X, bus-master DMA, IOMMU
Software layers	User / independent / driver / interrupt	VFS + page cache + blk-mq
Interface	Blocking, nonblocking, asynchronous	epoll, io_uring
Buffering	Speed/size mismatch, double and circular buffers	NIC/NVMe rings, zero copy
HDD	Seek + rotation + transfer; scheduling	NCQ, mq-deadline, BFQ
SSD	No seeks; FTL, garbage collection, TRIM	NVMe, multiqueue, `none`
Reliability	RAID 0/1/5/6/10, stable storage	mdadm, ZFS, journaling

The essence of I/O design: decouple slow, diverse devices from the fast CPU (buffers, interrupts, DMA), hide their differences behind a uniform interface (drivers, VFS), and choose the order wisely (scheduling, batching).

SUMMARY

Review Questions

1. With port-mapped and with memory-mapped I/O, which instruction accesses a device register? How is the caching problem solved for MMIO?
2. 4096 characters go to a 1000-character-per-second printer using interrupt-driven I/O, at 2 µs per interrupt. What is the CPU cost?
3. What is cycle stealing? When does DMA not pay off?
4. How long does a random 4 KiB read take on a disk with 7200 rpm, 8 ms average seek and a 200 MB/s media rate?
5. Head at 53, queue 98, 183, 37, 122, 14, 124, 65, 67 (0–199). What is the total movement for SSTF and C-LOOK (upward)?
6. Head at 11, requests 1, 36, 16, 34, 9, 12. How many cylinders do FCFS, SSTF and the elevator (upward) move?
7. In RAID 5, D0 = 1011 0010, D2 = 1110 0111, P = 0000 1001. What is the failed D1? If one disk gives 100 IOPS, how many IOPS do 6-disk RAID 5 and RAID 4 give for random small writes?
8. Why does an SSD need TRIM? Which I/O scheduler is the Linux default for NVMe devices, and why?
9. What is the fundamental difference in model between `epoll` and `io_uring`?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 11: Mass-Storage Structure; Chapter 12: I/O Systems.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 5: Input/Output.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "I/O Devices", "Hard Disk Drives", "RAID", "Flash-based SSDs".
- D. A. Patterson, G. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", SIGMOD 1988.
- NVM Express Base Specification 2.x; PCI-SIG PCIe 5.0/6.0; USB4 Specification; UEFI Specification (GPT).
- Linux kernel documentation: "Multi-Queue Block IO Queueing Mechanism (blk-mq)", "NO_HZ: Reducing Scheduling-Clock Ticks", `io_uring` (J. Axboe, "Efficient IO with io_uring"), "Linux Device Drivers" (Corbet, Rubini, Kroah-Hartman, 3rd ed.).

The End

Chapter 11: Input/Output

Previous: Chapter 10: File Systems · Next: Chapter 12: Multiprocessor Systems