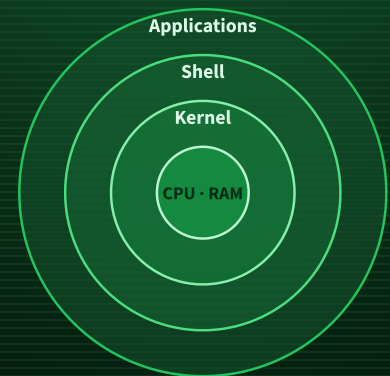


OS



Operating Systems · Interview-Ready

Processes, threads, scheduling, deadlocks, paging, virtual memory, file systems, containers and Linux, with worked numericals and 90+ interview questions with answers.

```
shreyash@kernel: ~ $ top
```

```
PID  USER      PR  %CPU %MEM  S  COMMAND
1337  you        20   99.9  4.2   R  interview_prep
2048  you        20   12.0  1.1   S  scheduler (round robin)
4096  you        20    3.4  0.8   S  page_replacement --lru
8080  you        20    0.0  0.0   Z  self_doubt (zombie, reaped)
$ kill -9 $(pgrep fear) # offer.exe starting...
```

[Processes](#)[Threads](#)[CPU Scheduling](#)[Semaphores](#)[Deadlock](#)[Banker's Algo](#)[Paging](#)[TLB](#)[LRU](#)[File Systems](#)[Docker Internals](#)[Linux](#)

WRITTEN BY

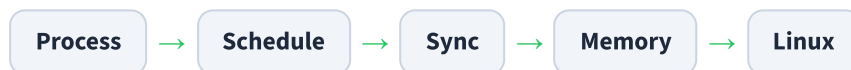
Shreyash Tripathi

JOB-READY SERIES • SHORT EDITION



Operating Systems

Interview-Ready Guide for developers preparing for their next job



WRITTEN BY

Shreyash Tripathi

Short Edition, 2026

© Shreyash Tripathi. All rights reserved.

Preface

You already have one job. You write code, you push to production, and sometimes things break in ways nobody explained in college. Now you want the next job, and the interviews ask the same "basic" Operating Systems questions again, except now the interviewer also expects you to connect them to real work. This book is written for exactly that moment.

It follows the style of a school textbook on purpose: numbered chapters, simple language, worked examples step by step, a check with answers after every topic, and full exercises with an answer key at the end of every chapter. Every chapter also has an **Interview Corner** with questions interviewers really ask, and an **At Work** box that shows where the concept appears in a real job. There is humour too, because a joke is easier to remember than a definition.

Read one chapter a day, solve the questions before you look at the answers, and revise the Interview Corners the night before your interview. You will walk in prepared.

Shreyash Tripathi

Features of Every Chapter

DO YOU KNOW?

Real facts and history.

FUNNY SIDE

A joke that makes the idea stick.

KEY POINT

The line to remember.

INTERVIEW CORNER

Questions interviewers really ask, with model answers.

AT WORK

How the concept shows up in a real job.

COMMON MISTAKE

Traps that cost marks and cause bugs.

CHECK YOUR PROGRESS

Short questions with answers after every topic.

Each chapter ends with **What Have We Learnt?**, **Key Terms**, **Exercises** and a complete **Answer Key**. The last chapter is a full **Interview Question Bank**.

Contents

PART I: PROCESSES AND THREADS

1	Operating System Basics	6
2	Processes and IPC	20
3	Threads and Concurrency	35

PART II: CPU AND SYNCHRONIZATION

4	CPU Scheduling	50
5	Synchronization: Mutex, Semaphores and Classic Problems	63
6	Deadlocks	77

PART III: MEMORY

7	Memory Management and Paging	92
8	Virtual Memory and Page Replacement	106

PART IV: STORAGE, I/O AND CONTAINERS

9	File Systems and Disk Management	120
10	I/O, Virtualization and Containers	134

PART V: OS AT WORK AND INTERVIEWS

11	Linux for Developers	149
12	Operating Systems Interview Question Bank	163

APPENDICES

A	Linux Commands Cheat Sheet	180
B	Formulas and Numericals Quick Reference	183
C	Glossary	186
D	Sample Question Paper with Answers	189

PART I

Processes and Threads

What an OS does, processes, IPC and threads.

CHAPTER 1

Operating System Basics

"The OS is like a good stage manager: if you notice it, something has already gone wrong."

After studying this chapter, you will be able to

- Explain what an operating system does and why programs cannot run safely without one.
- Compare batch, multiprogramming, time-sharing, multiprocessing and real-time systems.
- Explain kernel mode vs user mode and how a system call crosses the boundary.
- Differentiate interrupts, traps and exceptions with examples.
- Compare monolithic and microkernel designs and name real OSes for each.
- Describe the boot process from power button to login prompt (BIOS/UEFI, bootloader, kernel, systemd).

Think of a busy railway station like Mumbai CST. Hundreds of trains (programs) want platforms (CPU), tracks (memory) and signals (devices). The station master decides who gets which platform, for how long, and makes sure one train cannot drive onto another's track. That station master is the operating system.

1.1 What Does an Operating System Do?

An **operating system (OS)** is the software layer between hardware and user programs. It has two big jobs:

1. **Resource manager:** it shares CPU, memory, disk and network fairly and safely between many programs.
2. **Extended (abstract) machine:** it hides ugly hardware details behind clean abstractions. You call `write(fd, buf, n)`; you do not program the SSD controller.



Figure 1.1: The OS sits between applications and hardware

1.1.1 Main functions

Function	What it means	Example you have seen
Process management	Create, schedule, stop processes	<code>ps</code> , <code>top</code> , <code>kill</code>
Memory management	Give each process its own address space	OOM killer ending a Java process
File system	Files, directories, permissions	<code>chmod 644</code> , "Permission denied"
Security and protection	Users, permissions, isolation	Running a container as non-root

KEY POINT

The OS gives three illusions: each process thinks it has its own CPU (scheduling), its own memory (virtual memory), and simple files instead of disk blocks (file system).

INTERVIEW CORNER

Q: What is an operating system?

A: System software that manages hardware resources (CPU, memory, I/O) and provides services and abstractions (processes, files, sockets) to applications. It acts as both a resource allocator and a control program that prevents misuse.

CHECK YOUR PROGRESS 1.1

1. Name the two views of an OS.

Answer: Resource manager (allocates CPU, memory, devices fairly and safely) and extended machine (provides easy abstractions like files and processes over raw hardware).

2. Which OS function is involved when you see "Permission denied" on a file?

Answer: File system plus protection/security. The kernel checks permission bits before allowing access.

1.2 Types of Operating Systems

OS types are best understood as history: each type solved a problem of the previous one.

1.2.1 Batch systems

In the 1950s and 60s, users gave jobs (punched cards) to an operator, who grouped similar jobs into batches and ran them one after another. No interaction. Problem: when a job waited for slow I/O (tape, printer), the costly CPU sat idle.

1.2.2 Multiprogramming

Keep **several jobs in memory**. When one waits for I/O, the CPU switches to another. Goal: **maximise CPU utilisation**. Analogy: a chai stall owner who puts milk to boil and, instead of staring at it, makes

sandwiches for the next customer.

1.2.3 Time-sharing (multitasking)

An extension of multiprogramming where the CPU switches between jobs **very frequently** (every few milliseconds, a "time slice"), so many interactive users each feel they own the machine. Goal: **fast response time**. UNIX was designed as a time-sharing system. Your laptop running VS Code and Chrome together is time-sharing.

1.2.4 Multiprocessing (parallel) systems

More than one CPU (or core) shares memory and runs processes truly at the same time. **Symmetric multiprocessing (SMP)**: every core is equal and runs the same OS (Linux, Windows today). (Asymmetric multiprocessing, where one master CPU assigns work, is rare today.)

1.2.5 Real-time systems (RTOS)

Correctness depends on **meeting deadlines**, not just on producing the right answer.

- **Hard real-time**: missing a deadline is a failure. Airbag controller, pacemaker, ABS brakes.
- **Soft real-time**: missing a deadline lowers quality but is tolerable. Video streaming, online gaming, voice calls.

Examples: FreeRTOS, VxWorks, QNX.

⚠️ COMMON MISTAKE

Saying "multiprogramming and multiprocessing are the same". Multiprogramming shares **one** CPU by switching; multiprocessing uses **many** CPUs in parallel. Modern OSes do both.

🎯 INTERVIEW CORNER

Q: Difference between multiprogramming and multitasking?

A: Multiprogramming switches jobs mainly when the running job blocks for I/O, to maximise CPU use. Multitasking (time-sharing) also switches on a timer, giving each task a short time slice, to minimise response time for interactive users.

Q: Is a real-time OS a "fast" OS?

A: Not necessarily. Real-time means **predictable** and deadline-bound. An RTOS may have lower throughput than Linux but guarantees a worst-case response time.

😄 FUNNY SIDE

Batch OS: you submit a job, come back tomorrow, and find a syntax error on line 3. Time-sharing OS: you get the syntax error in 2 seconds. Modern IDE: it underlines the error before you finish typing. Progress means getting rejected faster.

✓ CHECK YOUR PROGRESS 1.2

1. Which OS type was invented to solve CPU idling during I/O?

Answer: Multiprogramming.

2. What is SMP?

Answer: Symmetric multiprocessing: all CPUs are peers, share memory and run the same OS kernel; any CPU can run any process. Linux and Windows use SMP.

1.3 Kernel Mode vs User Mode

If any program could directly write to disk or disable interrupts, one buggy app could wipe the disk or freeze the machine. So the CPU hardware supports at least two privilege levels, indicated by a **mode bit**:

- **Kernel mode** (supervisor/privileged, mode bit 0): can execute every instruction, access all memory and devices. The kernel runs here.
- **User mode** (mode bit 1): restricted. Privileged instructions (I/O, changing page tables, halting the CPU) cause a fault. Your apps run here.

On x86 these are "rings": ring 0 for the kernel, ring 3 for user programs (rings 1 and 2 are rarely used).

Analogy: in a bank, customers (user mode) stand at the counter. Only the cashier (kernel) can open the vault. You fill a slip (system call) and the cashier does the job for you after checking it.

🔑 KEY POINT

Dual mode is a **hardware** feature. The OS relies on it for protection. The only way for a user program to enter kernel mode is through a controlled entry point: a system call, an interrupt or an exception.

✓ CHECK YOUR PROGRESS 1.3

1. What happens if a user program runs a privileged instruction?

Answer: The CPU raises an exception (general protection fault). The kernel handles it, usually by killing the process with a signal such as SIGSEGV or SIGILL.

2. Which x86 ring does the Linux kernel use?

Answer: Ring 0. User programs run in ring 3.

1.4 System Calls and Traps

A **system call** is the programming interface through which a user program requests a service from the kernel: open a file, create a process, send data on a socket. Normally you call a library wrapper (glibc's `read()`, Java's `FileInputStream`) which internally makes the system call.

1.4.1 How a system call works (Linux x86-64)

1. The program calls `read(fd, buf, 100)` in `libc`.
2. `libc` puts the system call number (for `read`, 0 on x86-64) in register `rax` and the arguments in other registers.
3. `libc` executes the `syscall` instruction. This is a **trap**: the CPU switches to kernel mode and jumps to a fixed kernel entry point.
4. The kernel looks up the handler in the **system call table**, validates arguments (is `buf` really your memory?) and does the work.
5. The result (or a negative error, such as `-EBADF`) is put in `rax`; the CPU returns to user mode.
6. `libc` returns the value; on error it returns `-1` and sets `errno`.



Figure 1.2: Path of a system call

1.4.2 Categories of system calls

Category	Linux examples
Process control	<code>fork</code> , <code>execve</code> , <code>exit</code> , <code>wait4</code> , <code>kill</code>
File management	<code>open</code> , <code>read</code> , <code>write</code> , <code>close</code> , <code>lseek</code>
Device management	<code>ioctl</code> , <code>read</code> , <code>write</code>
Information	<code>getpid</code> , <code>time</code> , <code>uname</code>
Communication	<code>pipe</code> , <code>socket</code> , <code>connect</code> , <code>mmap</code> , <code>shmget</code>
Protection	<code>chmod</code> , <code>chown</code> , <code>setuid</code>

ACTIVITY 1.1

On Linux, run `strace -c ls`. It prints a table of every system call `ls` made and how many times.

COMMON MISTAKE

Thinking `printf` is a system call. `printf` is a C library function that formats text into a user-space buffer; only when the buffer is flushed does it call the `write` system call. That is why a program that crashes may "lose" its last `printf` lines.

AT WORK

A Node.js API is slow and you suspect disk. `strace -f -T -p <pid>` shows each system call with its time; you notice thousands of tiny `write` calls for logging, each costing a mode switch. Fix: buffer logs (or use an async logger like `pino`). Every system call crosses the user/kernel boundary, so batching I/O is a real win.

INTERVIEW CORNER

Q: What is a system call? Give an example.

A: A controlled request from a user program to the kernel for a privileged service. Example: `fork()` to create a process, `open()` to open a file. It is entered by a trap instruction that switches the CPU to kernel mode.

Q: Why are system calls slower than normal function calls?

A: They need a mode switch, saving and restoring registers, argument validation and sometimes cache/TLB effects (and extra cost from security mitigations). A normal function call stays in user mode and just jumps.

✓ CHECK YOUR PROGRESS 1.4

1. How does the kernel know which system call was requested?

Answer: By the system call number placed in a register (rax on x86-64) before the trap. The kernel uses it as an index into the system call table.

2. Which tool shows the system calls made by a running process on Linux?

Answer: `strace` (for example `strace -p PID`).

1.5 Interrupts, Traps and Exceptions

The CPU cannot keep asking every device "are you done yet?" (that is **polling**, wasteful). Instead, devices send an **interrupt**: a hardware signal that makes the CPU pause the current work, run an **interrupt service routine (ISR)** in kernel mode, then resume.

1.5.1 Steps in interrupt handling

1. A device (say, the network card) raises an interrupt line.
2. The CPU finishes the current instruction, saves the program counter and flags.
3. It switches to kernel mode and looks up the handler in the **interrupt vector table** (IDT on x86).
4. The ISR runs (for example, copies the packet into a kernel buffer).
5. The saved state is restored; the interrupted program continues (or the scheduler picks another process).

1.5.2 Types

Event	Source	Synchronous?	Example
Hardware interrupt	Device or timer	No (asynchronous)	Keypress, packet arrival, timer tick
Trap (software interrupt)	Intentional instruction	Yes	<code>syscall</code> , <code>int 0x80</code> , debugger breakpoint

Event	Source	Synchronous?	Example
Exception (fault)	Error in current instruction	Yes	Divide by zero, page fault, invalid opcode

The **timer interrupt** is the hero of time-sharing: it fires periodically so the kernel gets control back even if a program runs `while(true){}`. Without it, a single infinite loop could own the CPU forever.

INTERVIEW CORNER

Q: Difference between an interrupt and a trap?

A: An interrupt is asynchronous and comes from hardware (device, timer). A trap is synchronous and is caused deliberately by the running program (system call instruction). Both switch the CPU to kernel mode and run a handler.

Q: Is a page fault an error?

A: Not always. A page fault is an exception raised when a page is not in memory. Usually the kernel loads it and the instruction is retried (normal demand paging). Only an access to an invalid address becomes a segmentation fault.

CHECK YOUR PROGRESS 1.5

1. Why is the timer interrupt essential for a time-sharing OS?

Answer: It returns control to the kernel at regular intervals, so the scheduler can preempt a running process even if it never makes a system call or blocks.

2. Classify a divide-by-zero.

Answer: A synchronous exception (fault) caused by the current instruction. Linux delivers SIGFPE to the process.

1.6 Kernel Architectures: Monolithic vs Microkernel

1.6.1 Monolithic kernel

All core services (scheduler, memory manager, file systems, drivers, network stack) run in **one big program in kernel mode**, sharing one address space. Services call each other with plain function calls, so it is **fast**. But a bug in one driver can crash the whole kernel. Examples: Linux, traditional UNIX, MS-DOS.

Linux softens this with **loadable kernel modules** (`lsmod`, `modprobe`): drivers can be loaded at runtime, but they still run in kernel mode.

1.6.2 Microkernel

Only the bare minimum runs in kernel mode: basic scheduling, low-level memory management and **inter-process communication (IPC)**. File systems, drivers and networking run as **user-space**

servers that talk by message passing. A crashed driver can be restarted. Cost: more context switches and messages, so historically slower. Examples: Mach, QNX, MINIX 3, seL4.

1.6.3 Hybrid and others

Hybrid kernels mix both: Windows NT and macOS XNU (Mach plus BSD parts) are usually called hybrid.

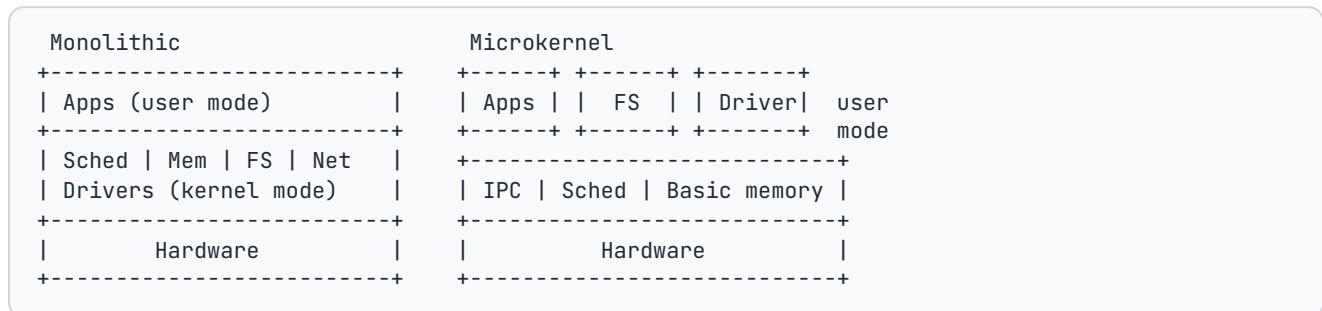


Figure 1.3: Monolithic vs microkernel

💡 DO YOU KNOW?

In 1992, Andrew Tanenbaum (creator of MINIX) and Linus Torvalds had a famous Usenet debate titled "LINUX is obsolete", arguing microkernel vs monolithic. Linux stayed monolithic and went on to run most of the world's servers and every Android phone.

✅ CHECK YOUR PROGRESS 1.6

1. Name two OSes usually called hybrid.

Answer: Windows NT family and macOS (XNU kernel).

2. Why is a monolithic kernel faster?

Answer: All services share one address space in kernel mode, so they call each other with direct function calls instead of IPC messages and context switches.

1.7 The Boot Process

What happens between pressing the power button and seeing the login screen? A favourite "explain end to end" question.

1. **Power on:** the CPU starts executing firmware at a fixed address.
2. **Firmware (BIOS or UEFI):** runs **POST** (power-on self-test), initialises hardware, then finds a boot device.
 - **Legacy BIOS:** reads the first 512-byte sector (MBR) of the disk and runs the tiny bootloader code there. MBR partitioning supports disks up to about 2 TiB and 4 primary partitions.
 - **UEFI:** modern replacement. Reads a FAT-formatted EFI System Partition, runs a `.efi` bootloader file. Uses GPT partitions, supports much larger disks and **Secure Boot** (only signed bootloaders run).

3. **Bootloader (GRUB on Linux, Windows Boot Manager):** shows a menu, loads the kernel image (`vmlinux`) and an initial RAM disk (`initramfs`) into memory, passes kernel parameters.
4. **Kernel:** decompresses itself, sets up memory management, interrupts, scheduler, drivers; uses `initramfs` to load drivers needed to mount the real root file system; mounts root.
5. **Init process (PID 1):** the kernel starts the first user-space process. On most modern Linux distributions this is **systemd** (older: SysV init, Upstart).
6. **systemd:** starts services in parallel based on unit files and dependencies (network, sshd, docker, nginx), reaching a target such as `multi-user.target` or `graphical.target`.
7. **Login:** getty or a display manager shows the login prompt.

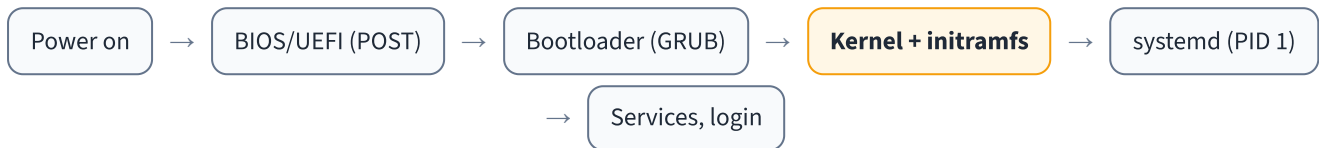


Figure 1.4: Linux boot sequence

ACTIVITY 1.2

On a Linux machine or VM, run `systemd-analyze` to see how long boot took, and `systemd-analyze blame` to see which services were slowest. Run `ps -p 1 -o comm=` to confirm what PID 1 is. Inside a Docker container, run the same command: PID 1 is usually your app, not systemd.

AT WORK

You deploy a Node.js app in Docker and notice `docker stop` takes 10 seconds and the app never runs its graceful shutdown. Reason: your app runs as PID 1 in the container. PID 1 is special in Linux: signals it has not installed a handler for (like `SIGTERM`) are ignored by default. Docker waits 10 seconds, then sends `SIGKILL`. Fixes: handle `SIGTERM` in code (`process.on('SIGTERM', ...)`), or run with `docker run --init (tini)` so a tiny init forwards signals and reaps zombies.

INTERVIEW CORNER

Q: Explain what happens when you switch on a computer.

A: Firmware (BIOS/UEFI) runs POST and finds a boot device; the bootloader (GRUB) loads the kernel and `initramfs`; the kernel initialises hardware, memory and scheduler, mounts the root file system, and starts PID 1 (systemd), which starts services and finally the login prompt.

Q: What is PID 1 and why is it special?

A: The first user-space process (init/systemd). It starts services, adopts orphan processes and reaps zombies. If PID 1 dies, the kernel panics on a host; in a container, the container stops.

✓ CHECK YOUR PROGRESS 1.7**1. What does POST stand for and who runs it?**

Answer: Power-On Self-Test, run by the firmware (BIOS/UEFI) to check that basic hardware like RAM and CPU works before booting.

2. Name two advantages of UEFI over BIOS.

Answer: Supports GPT and disks larger than about 2 TiB, and supports Secure Boot.

What Have We Learnt?

- An OS is a resource manager and an extended machine between applications and hardware.
- Batch systems ran jobs in groups; multiprogramming keeps the CPU busy during I/O waits.
- Time-sharing switches often using time slices to give quick response to many users.
- Multiprocessing uses many CPUs; SMP treats all cores as equals.
- Real-time systems are about meeting deadlines: hard (airbag) vs soft (video).
- Dual mode (kernel/user) is a hardware feature that protects the system.
- System calls are the controlled gate into the kernel, entered via a trap instruction.
- Interrupts are asynchronous (hardware); traps and exceptions are synchronous.
- The timer interrupt makes preemptive multitasking possible.
- Monolithic kernels (Linux) are fast; microkernels (QNX, MINIX 3) are more isolated.
- Boot: firmware (BIOS/UEFI) → bootloader (GRUB) → kernel + initramfs → systemd (PID 1) → services.

Key terms: operating system, kernel, batch system, multiprogramming, time-sharing, multiprocessing, SMP, real-time OS, kernel mode, user mode, mode bit, system call, trap, interrupt, exception, ISR, interrupt vector table, timer interrupt, monolithic kernel, microkernel, hybrid kernel, loadable module, BIOS, UEFI, POST, MBR, GPT, bootloader, GRUB, initramfs, systemd, PID 1

Exercises

A. Multiple Choice Questions

- Which mechanism lets the kernel regain control from a program stuck in an infinite loop?
 - System call
 - Timer interrupt
 - Page fault
 - Bootloader
- Which of these is NOT a system call on Linux?
 - fork
 - read
 - printf
 - execve
- A car's airbag controller needs a:
 - Batch OS
 - Soft real-time OS
 - Hard real-time OS
 - Time-sharing OS
- Which kernel design runs file systems and drivers as user-space servers?
 - Monolithic
 - Microkernel
 - Batch
 - Hybrid only
- On a modern Linux host, PID 1 is usually:
 - GRUB
 - bash
 - systemd
 - the kernel

B. Fill in the Blanks

- On x86, the kernel runs in ring _____ and user programs in ring _____.
- UEFI uses the _____ partition scheme, while legacy BIOS uses MBR.
- The Linux tool to trace system calls of a process is _____.

C. True or False

- Multiprogramming requires more than one CPU.
- A page fault always means the program has a bug.
- Linux kernel modules run in kernel mode.

D. Short Answer Questions

- Differentiate between an interrupt and a trap.
- Why is a system call more expensive than a function call?
- What is the role of the bootloader?

E. Long Answer Questions

- Explain the steps of a system call from a C program calling `read()` to the value being returned. Why is dual-mode operation needed?

2. Describe the Linux boot process in detail, comparing BIOS and UEFI.

F. Design Challenge

A Python worker in Docker loses in-flight jobs on every deploy, and old containers take 10 seconds to stop. Using PID 1, signals and init, propose a fix.

Answer Key: Chapter 1

A. MCQ

1. (b) The timer interrupt fires periodically and hands control to the kernel scheduler.
2. (c) `printf` is a C library function; it calls `write` internally.
3. (c) Missing the deadline would be a failure, so it is hard real-time.
4. (b) Microkernels keep only IPC, basic scheduling and memory in the kernel.
5. (c) `systemd` is the init system on most modern distributions.

B. Fill in the Blanks

1. 0, 3
2. GPT
3. `strace`

C. True or False

1. False. Multiprogramming switches between programs on one CPU; multiple CPUs is multiprocessing.
2. False. Most page faults are normal demand paging; only invalid accesses become segmentation faults.
3. True. Modules are loaded into kernel space and have full privilege.

D. Short Answers

1. An interrupt is an asynchronous signal from hardware (timer, keyboard, NIC) that can arrive at any time. A trap is a synchronous, intentional transfer to the kernel caused by an instruction in the running program, such as `syscall`. Both save state, switch to kernel mode and run a handler.
2. A system call switches the CPU from user to kernel mode, saves and restores registers, validates arguments and may disturb caches and TLB; security mitigations add more cost. A function call is just a jump within user mode.
3. The bootloader (for example GRUB) is loaded by firmware. It lets the user choose an OS, loads the kernel image and `initramfs` into memory, passes boot parameters and jumps to the kernel's entry point.

E. Long Answers

1. Model answer:
 - Program calls the libc wrapper `read(fd, buf, n)`; libc puts the `syscall` number in `rax` and arguments in registers.
 - The `syscall` instruction traps: CPU switches to kernel mode and jumps to the kernel entry point.
 - Kernel saves registers, indexes the system call table, validates `fd` and `buf`.
 - It reads the data (the process may block until the device interrupts).

- Result or negative error goes in `rax`; CPU returns to user mode; `libc` returns the count or `-1` with `errno`.
- Dual mode is needed so user programs cannot run privileged instructions or touch devices and other processes' memory directly.

2. Model answer:

- Firmware runs POST and initialises hardware.
- BIOS runs code from the 512-byte MBR; UEFI runs a `.efi` file from the EFI System Partition.
- UEFI adds GPT, disks beyond about 2 TiB and Secure Boot.
- GRUB loads `vmlinuz` and `initramfs`; the kernel sets up memory, interrupts, scheduler and drivers.
- `initramfs` helps mount the real root file system; the kernel starts PID 1 (`systemd`).
- `systemd` starts services in parallel to reach a target; `getty` or a display manager shows login.

F. Design Challenge: Model Solution

The worker runs as PID 1, so it ignores `SIGTERM` unless it installs a handler; after the 10-second grace period Docker sends `SIGKILL` and in-flight jobs die. Fix: (1) handle `SIGTERM`: stop taking new jobs, finish or re-queue the current one, exit; (2) run with `docker run --init (tini)` so signals are forwarded and zombies reaped; (3) use the exec form `CMD ["python", "worker.py"]` so a shell does not swallow signals; (4) raise the stop grace period if jobs are long.

CHAPTER 2

Processes and IPC

"A program is a recipe. A process is the kitchen on fire because someone is actually cooking."

After studying this chapter, you will be able to

- Differentiate a program from a process and draw a process's memory layout.
- List what a Process Control Block (PCB) stores and why.
- Draw the process state diagram and explain each transition.
- Explain what a context switch costs and why it matters.
- Predict the output of fork/exec/wait programs, including "n forks" questions.
- Explain zombie and orphan processes and fix them in real code.
- Compare IPC methods: pipes, FIFOs, message queues, shared memory, sockets, signals.

Think of a wedding caterer. The recipe book (program) sits quietly on a shelf. On the wedding day, five cooks pick up the same biryani recipe and start cooking in five big vessels. Each vessel has its own rice, its own fire and its own stage of cooking. Those five running cookings are five processes of one program.

Interviewers love this chapter: tricky fork output questions, zombies, and real-world links like Nginx workers and Redis snapshots. Let us cook.

2.1 Program vs Process

A **program** is a passive file on disk (for example `/usr/bin/python3`). A **process** is a program **in execution**: it has its own memory, CPU register values, open files and a state. One program can have many processes: open three terminals and you have three bash processes from one binary.

2.1.1 Memory layout of a process

Each process gets its own **virtual address space**, divided into regions:

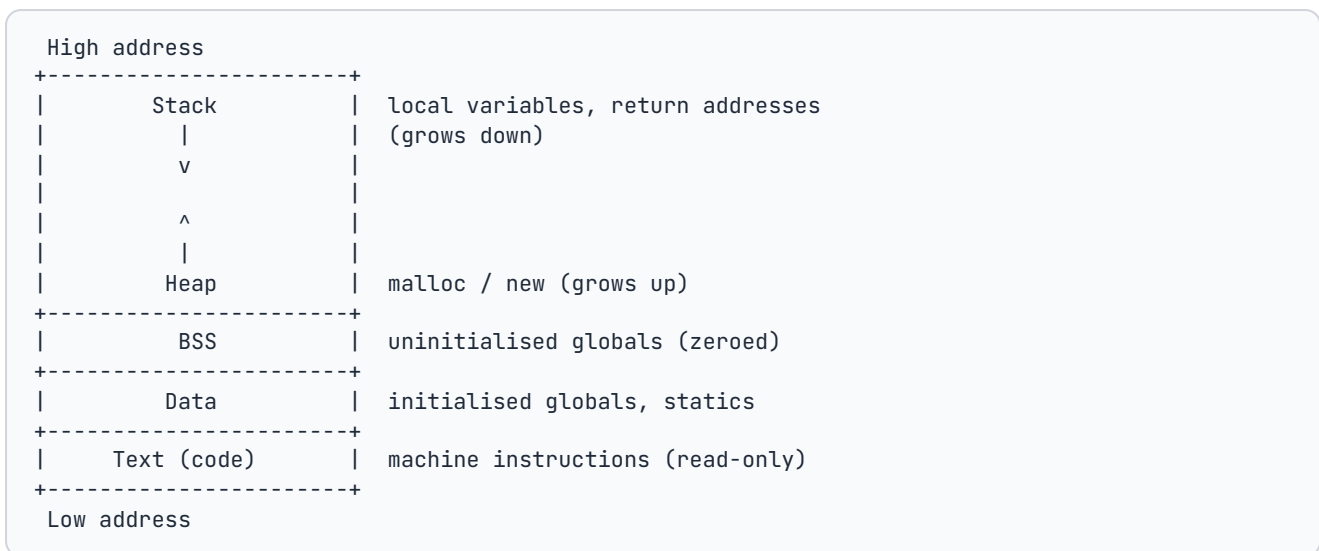


Figure 2.1: Memory layout of a process

Shared libraries and memory-mapped files usually sit in the gap between heap and stack.

AT WORK

Stack overflow (the error, not the website) comes from deep or infinite recursion filling the stack (Linux default main-thread stack is usually 8 MB, see `ulimit -s`). A **memory leak** is heap memory that is allocated but never freed (or, in Java/Node, objects still referenced from a global cache). Leaks show up as RSS slowly rising in Grafana until the container hits its memory limit and gets OOM-killed. Knowing which segment grows tells you what kind of bug you have.

INTERVIEW CORNER

Q: Difference between a program and a process?

A: A program is a passive set of instructions on disk; a process is an active instance with its own address space, registers, PC, open files and state. Many processes can run the same program.

CHECK YOUR PROGRESS 2.1

1. Why is the text segment usually read-only?

Answer: To prevent a program from accidentally or maliciously modifying its own code, and so that many processes running the same program can share one copy of the code in physical memory.

2. In which segment does `static int x;` (no initial value) live?

Answer: BSS, because it is uninitialised (it is zero-filled at start).

2.2 Process Control Block and Process States

2.2.1 Process Control Block (PCB)

The kernel keeps a data structure per process called the **PCB** (in Linux, `struct task_struct`). It is the process's "Aadhaar file". It contains:

- **Process ID (PID)** and parent PID
- **Process state** (ready, running, waiting...)
- **Program counter** and **CPU registers** (saved when not running)
- **Scheduling info**: priority, time used, queue pointers
- **Memory-management info**: page table pointer, limits
- **I/O and file info**: open file descriptor table, current directory
- **Accounting**: CPU time used, start time, user ID

2.2.2 Process states



Figure 2.2: Five-state process model

The transitions in plain words:

1. **New** → **Ready**: process created and admitted, waiting for CPU.
2. **Ready** → **Running**: scheduler dispatches it to a CPU.
3. **Running** → **Ready**: preempted (time slice over or higher-priority process arrived).
4. **Running** → **Waiting (Blocked)**: it needs I/O or an event (disk read, lock, sleep).
5. **Waiting** → **Ready**: I/O completes. Note: it does **not** go straight to Running.
6. **Running** → **Terminated**: it calls `exit()` or is killed.

In Linux `ps` or `top`, the STAT column shows: **R** running/runnable, **S** interruptible sleep, **D** uninterruptible sleep (usually disk or NFS I/O), **T** stopped, **Z** zombie.

⚠ COMMON MISTAKE

Saying a process goes from Waiting directly to Running when its I/O finishes. It goes to **Ready** and must wait for the scheduler like everyone else, just like a passenger who finally gets the ticket still has to queue at the gate.

 INTERVIEW CORNER

Q: What is a PCB?

A: A kernel data structure holding everything about a process: PID, state, saved registers and PC, scheduling info, memory maps, open files. It lets the OS stop and resume a process exactly where it left off.

Q: A process shows state D in top and kill -9 does not kill it. Why?

A: D is uninterruptible sleep, usually waiting on disk or a hung NFS mount inside the kernel. Signals are not acted on until it leaves that state, so even SIGKILL waits.

 CHECK YOUR PROGRESS 2.2

1. What is the Linux name for the PCB?

Answer: `struct task_struct`.

2. When a time slice expires, where does the running process go?

Answer: Back to the Ready state (ready queue). It is preempted, not blocked.

2.3 Context Switch

A **context switch** is when the CPU stops running process A and starts running process B. Steps:

1. An interrupt, system call or block event enters the kernel.
2. Kernel saves A's registers and PC into A's PCB.
3. Scheduler picks B from the ready queue.
4. Kernel loads B's registers from B's PCB and switches the address space (page table base register).
5. CPU returns to user mode and continues B.

Context switching is **pure overhead**: no useful user work happens during it. The direct cost is typically in the range of a few microseconds; the bigger hidden cost is that CPU caches and the TLB are now "cold" for B.

Analogy: a cricket commentator switching between two matches must recall the score each time; switch every 5 seconds and nobody learns anything.

 KEY POINT

Process switch = save/restore registers + change address space (TLB flush or tagged TLB). Thread switch within the same process skips the address-space change, so it is cheaper. This is a top interview follow-up.

📁 AT WORK

Your Java service with 2,000 threads on a 4-core VM shows high CPU but low throughput. `vmstat 1` shows the `cs` (context switches per second) column in the hundreds of thousands. Too many runnable threads cause constant switching. Fix: right-size thread pools (close to core count for CPU-bound work), or use async I/O or virtual threads (Java 21).

🎯 INTERVIEW CORNER

Q: What happens during a context switch?

A: The kernel saves the current process's CPU state into its PCB, chooses the next process, restores that process's state and memory mapping, and resumes it. It is overhead, and it also cools caches and TLB.

✅ CHECK YOUR PROGRESS 2.3

1. Why is a very small time slice bad?

Answer: The CPU spends a large fraction of time on context-switch overhead instead of running user code, reducing throughput.

2. Which Linux command shows context switches per second?

Answer: `vmstat 1` (cs column). `pidstat -w` shows them per process.

2.4 Process Creation: fork, exec, wait

On UNIX/Linux, a new process is created by copying an existing one.

- `fork()` : creates a child that is a near-copy of the parent (same code, copy of memory, copy of open file descriptors). Returns **0 in the child**, the **child's PID in the parent**, and **-1 on failure**.
- `exec()` family (`execl`, `execvp`, `execve`...): replaces the current process's program with a new one. PID stays the same. On success, it **never returns**.
- `wait()` / `waitpid()` : parent blocks until a child exits and collects its exit status.
- `exit()` : terminates the process with a status code.

Modern kernels use **copy-on-write (COW)**: after fork, parent and child share the same physical pages marked read-only. Only when one writes is that page copied. That makes fork fast even for big processes.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
    } else if (pid == 0) {
        // child
        execlp("ls", "ls", "-l", NULL); // replace child with ls
        perror("exec");                // runs only if exec failed
    } else {
        // parent
        int status;
        waitpid(pid, &status, 0);
        printf("child %d done, exit=%d\n", pid, WEXITSTATUS(status));
    }
    return 0;
}
```

This fork + exec + wait pattern is exactly how your shell runs `ls -l`.

2.4.1 Output prediction: n forks

Rule: n `fork()` calls in sequence (not inside if conditions) create 2^n processes in total, so $2^n - 1$ new child processes.

```
int main(void) {
    fork();
    fork();
    fork();
    printf("hello\n");
}
```

Worked solution:

```
After fork #1:  P, C1                = 2 processes
After fork #2:  P, C1, C2, C3        = 4 processes
After fork #3:  each of the 4 forks again = 8 processes
printf runs in all 8 → "hello" printed 8 times
New processes created = 8 - 1 = 7
```

Example 2: fork inside a loop

```
for (int i = 0; i < 3; i++) {
    fork();
    printf("hi\n");
}
```

Count prints per iteration: after iteration 0 there are 2 processes, each prints → 2. Iteration 1: 4 processes print → 4. Iteration 2: 8 print → 8. Total = 2 + 4 + 8 = **14** "hi". General formula: $2 + 4 + \dots + 2^n = 2^{n+1} - 2$.

Example 3: fork with logical operators

```
if (fork() && fork()) {
    fork();
}
printf("x\n");
```

P: fork#1 → parent gets PID (true), child C1 gets 0 (false)
 C1: && short-circuits, skips if. C1 prints.
 P: fork#2 → parent true, child C2 gets 0 (false)
 C2: condition false, skips if. C2 prints.
 P: condition true, enters if, fork#3 → P and C3
 P prints, C3 prints.
 Total processes: P, C1, C2, C3 = 4 → "x" printed 4 times

Example 4: the buffering trap

```
printf("A");      // no newline!
fork();
printf("B\n");
```

Output when run on a terminal: `AB` printed twice. "A" is still in the stdio buffer at fork time, so the buffer is copied into the child and both flush "A" with their "B". With `printf("A\n")` on a terminal (line-buffered), "A" is printed once. Redirect output to a file and even "A\n" may print twice, because files are fully buffered.

ACTIVITY 2.1

Compile and run Example 1 with `gcc f.c -o f && ./f`. Then try Example 4 with `./f | cat`. Finally, draw the process tree for 4 forks and count (answer: 16 processes).

FUNNY SIDE

A fork bomb `:(){ :|:& };;` is a bash function that calls itself twice in the background, forever. It is the digital version of a WhatsApp family group forward: each message makes two more, until the server (or your aunt's phone) gives up. Protect servers with `ulimit -u` or the cgroup pids limit (Docker `--pids-limit`). Do not run it on your office laptop.

 INTERVIEW CORNER

Q: What does `fork()` return?

A: 0 to the child, the child's PID (positive) to the parent, -1 to the parent if it failed (no child is created).

Q: Difference between `fork` and `exec`?

A: `fork` creates a new process (a copy of the caller). `exec` replaces the program running in the current process; no new process is created and PID stays the same.

Q: How many times is "hello" printed with 4 consecutive `forks` before `printf`?

A: $2^4 = 16$ times; 15 new processes are created.

Q: What is copy-on-write?

A: After `fork`, parent and child share physical pages read-only. A page is copied only when either process writes to it. It saves memory and makes `fork` fast. Redis uses `fork` plus COW for background snapshots (BGSAVE).

 CHECK YOUR PROGRESS 2.4

1. If `execvp` succeeds, what happens to the line after it?

Answer: It never runs. The process's code is replaced by the new program; `exec` returns only on failure.

2. How many "hi" lines does a loop of 2 iterations with `fork()` then `printf()` print?

Answer: $2 + 4 = 6$ (formula $2^{n+1} - 2$ with $n = 2$).

2.5 Zombie and Orphan Processes

2.5.1 Zombie

When a child exits, the kernel frees its memory but keeps a small entry (PID, exit status) in the process table so the parent can read it with `wait()`. Until the parent does that, the child is a **zombie** (state `Z`, shown as `<defunct>` in `ps`). A zombie uses no CPU or memory, but uses a PID. Thousands of zombies can exhaust the PID limit and new processes fail with "fork: Resource temporarily unavailable".

Fixes: parent calls `wait/waitpid`; handle `SIGCHLD` and reap in the handler; or set `SIGCHLD` to `SIG_IGN` so the kernel reaps automatically; or kill the parent (then `init` adopts and reaps them). You cannot `kill -9` a zombie: it is already dead.

2.5.2 Orphan

If the **parent exits before the child**, the child becomes an **orphan**. It keeps running and is **adopted by `init` (PID 1)** or by a designated "subreaper" process. `init` reaps it when it ends, so orphans do not stay as zombies.

```
// Creating a zombie for 30 seconds (for learning only)
if (fork() == 0) {
    exit(0);          // child exits immediately
} else {
    sleep(30);        // parent does not call wait(): child is a zombie
}
// In another terminal: ps -el | grep Z
```

	Zombie	Orphan
Who died first	Child	Parent
Still running?	No (only an entry remains)	Yes
Resource used	PID and process-table slot	Normal resources
Fix	Parent must wait()	Adopted by init automatically

AT WORK

A Node.js container spawns `chromium` for PDF generation. After a week, `ps` inside the container shows hundreds of `<defunct>` processes. Reason: chromium's helper processes get orphaned, they are re-parented to PID 1, and PID 1 is your Node app, which never reaps them. Fix: run the container with `--init` (tini) or use `dumb-init` as entrypoint.

INTERVIEW CORNER

Q: What is a zombie process and how do you remove it?

A: A child that has exited but whose parent has not yet collected its exit status. Fix: parent calls `wait` (or handles `SIGCHLD`), or kill the parent so `init` reaps it.

Q: Can an orphan become a zombie forever?

A: No. Orphans are adopted by `init` (or a subreaper), which calls `wait` when they exit.

CHECK YOUR PROGRESS 2.5

1. Why does `kill -9` not remove a zombie?

Answer: The zombie has already terminated; there is nothing to kill. Only the parent's `wait()` (or the parent's death followed by `init` reaping) removes its entry.

2. Who becomes the parent of an orphan?

Answer: `init/systemd` (PID 1), or the nearest ancestor marked as a child subreaper.

2.6 Inter-Process Communication (IPC)

Processes have separate address spaces, so they need kernel help to talk. Two basic models: **shared memory** (both map the same memory; fast, but you must synchronise) and **message passing** (send/receive via the kernel; simpler, slower due to copying and system calls).

Method	Direction / scope	Speed	Typical use
Pipe (anonymous)	One-way, related processes (parent-child)	Good	<code>ps aux grep node</code>
Named pipe (FIFO)	One-way, unrelated processes on same host	Good	<code>mkfifo</code> for simple tools
Message queue	Structured messages, priority possible	Medium	POSIX mq between daemons
Shared memory	Many processes share a region	Fastest (no copy)	PostgreSQL shared buffers, Chrome rendering
Signal	Tiny notification (a number)	Fast	SIGTERM, SIGHUP to reload Nginx config
Unix domain socket	Two-way, same host	Fast	Docker CLI to dockerd (<code>/var/run/docker.sock</code>)
TCP/UDP socket	Two-way, across machines	Network bound	Microservices, databases

```
// Pipe: parent writes, child reads
int fd[2];
pipe(fd);                      // fd[0] = read end, fd[1] = write end
if (fork() == 0) {
    close(fd[1]);
    char buf[32];
    int n = read(fd[0], buf, sizeof buf);
    write(1, buf, n);           // prints "chai ready"
} else {
    close(fd[0]);
    write(fd[1], "chai ready\n", 11);
    close(fd[1]);
    wait(NULL);
}
```

💡 DO YOU KNOW?

Google Chrome uses a multi-process architecture: the browser, renderers (often one per site), GPU and network service run as separate processes and talk through an IPC system called Mojo. A crash in one tab's renderer does not take down the whole browser, which is the classic trade-off: isolation in exchange for more memory.

🔑 KEY POINT

Shared memory is the fastest IPC because data is not copied through the kernel after setup, but it needs synchronisation (mutex/semaphore). Message passing is easier to get right and works across machines.

 INTERVIEW CORNER

Q: Which IPC is fastest and why?

A: Shared memory, because after the initial mapping, processes read and write the same physical pages without system calls or copying. The cost is that you must synchronise access yourself.

Q: Pipe vs named pipe?

A: A pipe has no name and works only between related processes that inherit the file descriptors. A FIFO has a path in the file system, so unrelated processes can open it.

 CHECK YOUR PROGRESS 2.6

1. Which IPC would you use to tell Nginx to reload its config?

Answer: A signal: SIGHUP to the master process (this is what `nginx -s reload` sends).

2. What does `pipe(fd)` give you?

Answer: Two file descriptors: `fd[0]` for reading and `fd[1]` for writing; data written to `fd[1]` can be read from `fd[0]`.

What Have We Learnt?

- A program is passive code on disk; a process is a running instance with its own state.
- Process memory: text, data, BSS, heap (grows up), stack (grows down).
- The PCB (`task_struct` in Linux) stores PID, state, registers, memory and file info.
- States: New, Ready, Running, Waiting, Terminated; Waiting returns to Ready, not Running.
- A context switch saves one process's state and loads another's; it is pure overhead.
- `fork` returns 0 to the child and the child's PID to the parent; `exec` replaces the program.
- `n` consecutive forks give 2^n processes; a loop with `print` after `fork` prints $2^{n+1} - 2$ times.
- Copy-on-write makes `fork` cheap; `stdio` buffers are copied too (the double-print trap).
- Zombie: child dead, parent has not waited. Orphan: parent dead, child adopted by `init`.
- IPC methods: pipes, FIFOs, message queues, shared memory, signals, sockets.
- Shared memory is fastest but needs synchronisation.

Key terms: program, process, address space, text segment, data segment, BSS, heap, stack, PCB, `task_struct`, process state, ready queue, context switch, `fork`, `exec`, `wait`, `exit`, copy-on-write, fork bomb, zombie, orphan, `init`, `subreaper`, `SIGCHLD`, IPC, pipe, FIFO, message queue, shared memory, signal, Unix domain socket

Exercises

A. Multiple Choice Questions

- How many processes exist after 5 consecutive `fork()` calls (including the original)?
 - 5
 - 10
 - 31
 - 32
- `fork()` returns which value in the child?
 - Child PID
 - Parent PID
 - 0
 - 1
- A process waiting for disk I/O is in which state?
 - Ready
 - Running
 - Waiting
 - Terminated
- Memory from `malloc()` comes from the:
 - Stack
 - Heap
 - Text
 - BSS
- Which IPC method needs no copying of data through the kernel after setup?
 - Pipe
 - Message queue
 - Shared memory
 - TCP socket

B. Fill in the Blanks

- A child that has exited but not been waited for is called a _____ process.
- After a successful _____ call, the process runs a new program with the same PID.
- The optimisation where pages are copied only when written after fork is called _____.

C. True or False

- A zombie process can be removed with `kill -9`.
- An orphan process is adopted by `init` (or a subreaper).
- Context switching between threads of the same process is usually cheaper than between processes.

D. Short Answer Questions

- List five fields of a PCB.
- How many times does this print `"*"`? `fork(); fork(); printf("*");` Explain.
- Compare a pipe and a Unix domain socket.

E. Long Answer Questions

1. Draw and explain the five-state process model. Which transitions are caused by the scheduler and which by the process itself?
2. Explain how a shell runs `ls -l` using fork, exec and wait. What goes wrong if the shell never calls wait?

F. Design Challenge

Design a simple pre-fork web server (like early Apache or Nginx's master/worker model): a master process that creates 4 worker processes to handle requests. Describe how the master creates workers, how it restarts a worker that crashes, how it tells workers to shut down gracefully, and how you avoid zombies.

Answer Key: Chapter 2

A. MCQ

1. (d) $2^5 = 32$ processes total.
2. (c) The child receives 0.
3. (c) It is blocked waiting for an event.
4. (b) Dynamic allocation uses the heap.
5. (c) Shared memory maps the same pages into both processes.

B. Fill in the Blanks

1. zombie
2. exec
3. copy-on-write

C. True or False

1. False. It is already dead; the parent must reap it.
2. True. init or a subreaper adopts and later reaps it.
3. True. The address space and TLB entries stay the same.

D. Short Answers

1. PID and parent PID, process state, program counter and saved registers, scheduling info (priority), memory-management info (page table pointer), open file table, accounting info (CPU time used).
2. 4 times. The first fork makes 2 processes; each forks again giving 4; all 4 execute printf. ($2^2 = 4$.)
3. A pipe is one-way, anonymous and works only between related processes. A Unix domain socket is two-way, has a file-system path (or abstract name), works between unrelated processes, and supports multiple clients like a server socket. Both stay on one host.

E. Long Answers

1. Model answer:
 - States: New, Ready, Running, Waiting, Terminated.
 - New → Ready: OS admits the process. Ready → Running: scheduler dispatch.
 - Running → Ready: preemption by timer or higher priority (scheduler).
 - Running → Waiting: the process requests I/O or waits for an event (process).
 - Waiting → Ready: I/O completes, interrupt handler wakes it (OS).
 - Running → Terminated: exit (process) or kill (OS).
 - No direct Waiting → Running or Ready → Waiting.
2. Model answer:
 - Shell parses the line and calls fork().
 - Child sets up redirections if any (dup2), then `execvp("ls", args)`; PID unchanged.
 - Parent calls `waitpid(child)` and blocks (foreground job).
 - `ls` exits; parent collects status (available as `$?`) and prints the next prompt.

- Background jobs (&) are reaped later via SIGCHLD.
- Without wait, every finished command stays a zombie; eventually the PID table fills and fork fails.

F. Design Challenge: Model Solution

The master opens the listening socket first, then calls `fork()` 4 times; each child inherits the socket and loops on `accept()` to handle requests. The master stores worker PIDs and installs a SIGCHLD handler that calls `waitpid(-1, &status, WNOHANG)` in a loop, which reaps dead workers (no zombies) and forks a replacement if the exit was unexpected. For graceful shutdown the master catches SIGTERM, forwards SIGTERM (or SIGQUIT) to each worker, workers stop accepting, finish in-flight requests and exit; the master waits for all, then exits. This is essentially Nginx's master/worker design.