

Processes and Threads

“The OS illusion: making one CPU act like it’s 20.”

Big Idea

Every running program isn't *just code* — it's a **process**, an isolated world with its own memory, resources, and identity.

The OS juggles hundreds of these, switching between them so smoothly you think it's multitasking.

Process: The Living Program

A **process** is a running instance of a program with:






- Its own memory space
- CPU time slices
- Open files / I O handles
- Unique process ID (PID)

Analogy:

A person at a desk, working with their own supplies, taking turns at the CPU coffee machine.



Process Lifecycle

States

-  **New** – being created
-  **Ready** – waiting for CPU
-  **Running** – currently executing
-  **Waiting** – blocked for I/O or event
-  **Terminated** – finished or killed

Transitions: scheduler decisions, I/O events, system calls.

Visual Model

New → Ready → Running → Terminated
 Waiting 

Each arrow represents a system call or interrupt that changes who holds the CPU.

Threads: The Mini-Processes

A **thread** is a lightweight process inside another process.

All threads share:

- Memory space
- Code / data sections
- File descriptors

Each thread has:

- Own stack
- Own program counter
- Own schedule

Process vs Thread Comparison

Feature	Process	Thread
Memory space	Separate	Shared
Communication	Slow (IPC)	Fast (shared vars)
Overhead	High	Low
Isolation	Strong	Weak
Typical use	Whole programs	Tasks inside program

Why Threads Exist

- To do multiple tasks within one program (browser tabs)
- To overlap computation with I/O
- To scale across multiple cores
- To confuse CS students since 1970

Context Switching

When the CPU switches threads/processes:

1. Save current CPU state (registers, PC, etc.)
2. Load next thread's state
3. Resume execution

Overhead: switching too often wastes cycles — scheduling aims to minimize that.

Scheduling Overview (Concept Only)

Schedulers decide:

- Who runs next
- For how long (quantum)
- When to pre-empt

Common strategies:

- **FCFS** – First Come, First Served
- **RR** – Round Robin
- **SJF** – Shortest Job First

System Calls — The OS Entry Points

Processes interact with the OS via **system calls**:

`fork()` , `exec()` , `wait()` , `read()` , `write()` ...

They're the controlled bridge from **user mode** → **kernel mode**.

Process Creation (UNIX Style)

```
pid = fork()
if pid == 0:
    exec("child_program")
else:
    wait(pid)
```

- `fork()` duplicates the process
- `exec()` replaces it with new code
- `wait()` synchronizes parent and child

Visualization Idea – “Process Zoo”

Simulate multiple processes competing for CPU:

- Random burst and I/O wait times
- Log state transitions
- Scheduler: Round Robin for fairness

Shows the lifecycle in motion instead of static theory.

Why It Matters

Understanding processes and threads explains:

- Multitasking
- Deadlocks (next week)
- Virtual memory
- Containers and virtualization

Every higher-level abstraction depends on these primitives.

Exit Prompt

Finish the sentence:

| "A thread is like a __ **because it** __."

Examples

- " ...like a roommate — shares space but has its own schedule."
- " ...like a worker bee — independent but part of one hive."

Summary

Concept	Core Idea
Process	Running program with own resources
Thread	Lightweight execution unit
Scheduler	Decides who runs next
System Call	Bridge to kernel operations
Context Switch	Save/restore between tasks

Next Topic

Week 01 Topic 03 → Scheduling and Context Switching 

Week 01 Topic 03 → Scheduling and Context Switching 

References & Credits

topic: "Operating Systems Week 1 Lecture Slides"

focus: "Processes and Threads"

format: "Markdown-based slide content"

author: "T. Griffin and OpenAI GPT-5"

credit: "Concept scaffolding with ChatGPT (OpenAI GPT-5)"

