Block 2: Processes

What is a Process?

A process is a **program in execution**. It consists of:

1. Program code (text segment)
2. Data (global/static variables)
3. Heap (dynamic memory via `malloc`)
4. Stack (function calls, local variables)
5. CPU state (program counter, registers)
6. OS-managed metadata (PID, state, open files, etc.)

A program is **passive** (file on disk); a process is **active** (running instance in memory).

Program vs Process

A program is static and stored on disk. A process is dynamic, loaded into memory, and multiple processes can be created from the same program.

Why Processes Exist

- **Isolation**: one process cannot corrupt another
- **Multiplexing**: many programs share one CPU
- **Protection**: enforced via user/kernel mode
- **Abstraction**: each process behaves as if it owns the machine

Process States and Transitions

$$\text{created} \rightarrow \text{ready} \rightarrow \text{running} \rightarrow \text{blocked} \rightarrow \text{ready} \rightarrow \text{running} \rightarrow \text{exited}$$

- **ready**: can run, waiting for CPU
- **running**: executing on the CPU
- **blocked**: waiting for an event (I/O, signal, child)
- **exited**: process has terminated

`fork()` and `exec()`

`fork()` creates a new process by duplicating the calling process. It returns:

- `0` in the child
- the child's PID (> 0) in the parent
- `-1` on error (no child created)

After `fork()`, parent and child continue execution independently; execution order is undefined. After n consecutive `fork()` calls, up to 2^n processes exist.

`exec()` replaces the current program in a process while keeping the PID and process metadata. After a successful `exec()`, the old program no longer exists.

Process Termination, Zombies, Orphans

A process terminates via `exit()`, returning from `main`, or by a signal.

- **Zombie**: child exited, parent has not yet called `wait()` to collect the exit status
- Zombies become a problem if parents never call `wait()` and create many children
- **Orphan**: parent terminates while child is still running; orphans are adopted by `init/systemd`

Blocked vs Ready

Blocked processes cannot run even if the CPU is free because they are waiting for an event. Ready processes can run immediately but are waiting only for the CPU.

Process Groups

A process group is a set of related processes.

- each process belongs to exactly one process group
- each group has a PGID equal to the PID of the group leader
- the group leader may exit; the group continues as long as members exist

Process groups allow the OS to control multiple related processes as a unit (e.g. pipelines).

Why Pipelines and Signals Use Process Groups

Pipelines place all involved processes into the same process group so they can be controlled together. Signals (e.g. Ctrl+C) are sent to process groups so all related processes are affected consistently.

Scheduling Basics

Scheduling decides which ready process runs, when it runs, and for how long.

- **Scheduler**: selects the next process to run
- **Dispatcher**: performs the context switch

Context Switch

A context switch saves the state of the running process (PC, registers, state) and restores the state of another process. It can occur when a process blocks, exits, is preempted, or due to a timer interrupt.

Preemptive Scheduling and Timer Interrupts

UNIX/Linux uses preemptive scheduling.

- the OS can interrupt a running process
- timer interrupts enforce time sharing and trigger scheduling decisions

Preemptive scheduling prevents CPU monopolization.

Starvation and Aging

Starvation occurs when a ready process never receives CPU time due to priority. Aging prevents starvation by gradually increasing the priority of waiting processes.

Round-Robin Scheduling

Round-robin scheduling provides fairness through time slicing. Time slicing requires preemptive scheduling.