BSc in Computing Science

Dorset College Dublin
Educating since 1983

Module 2.3: Operating Systems & Administration 1

Lesson 3: **Process Management**

Lesson 3:

Process Management

Lesson 3 Objectives

- Define what a process is and distinguish it from a program.
- Explain the process lifecycle and states (new, ready, running, waiting, terminated).
- Describe the role of the Process Control Block (PCB) in managing processes.
- Illustrate how context switching enables multitasking.
- Analyse how OSs create, schedule, and terminate processes in practice.

What is a Process?

What is a Process?
Introduction

- A program is a **static entity**: an executable file stored on disk.
- A **process** is a **dynamic entity**: a program in execution.
- Both Tanenbaum and Silberschatz emphasise: a *process = program + execution context*.

- **Silberschatz et al. (2019):**

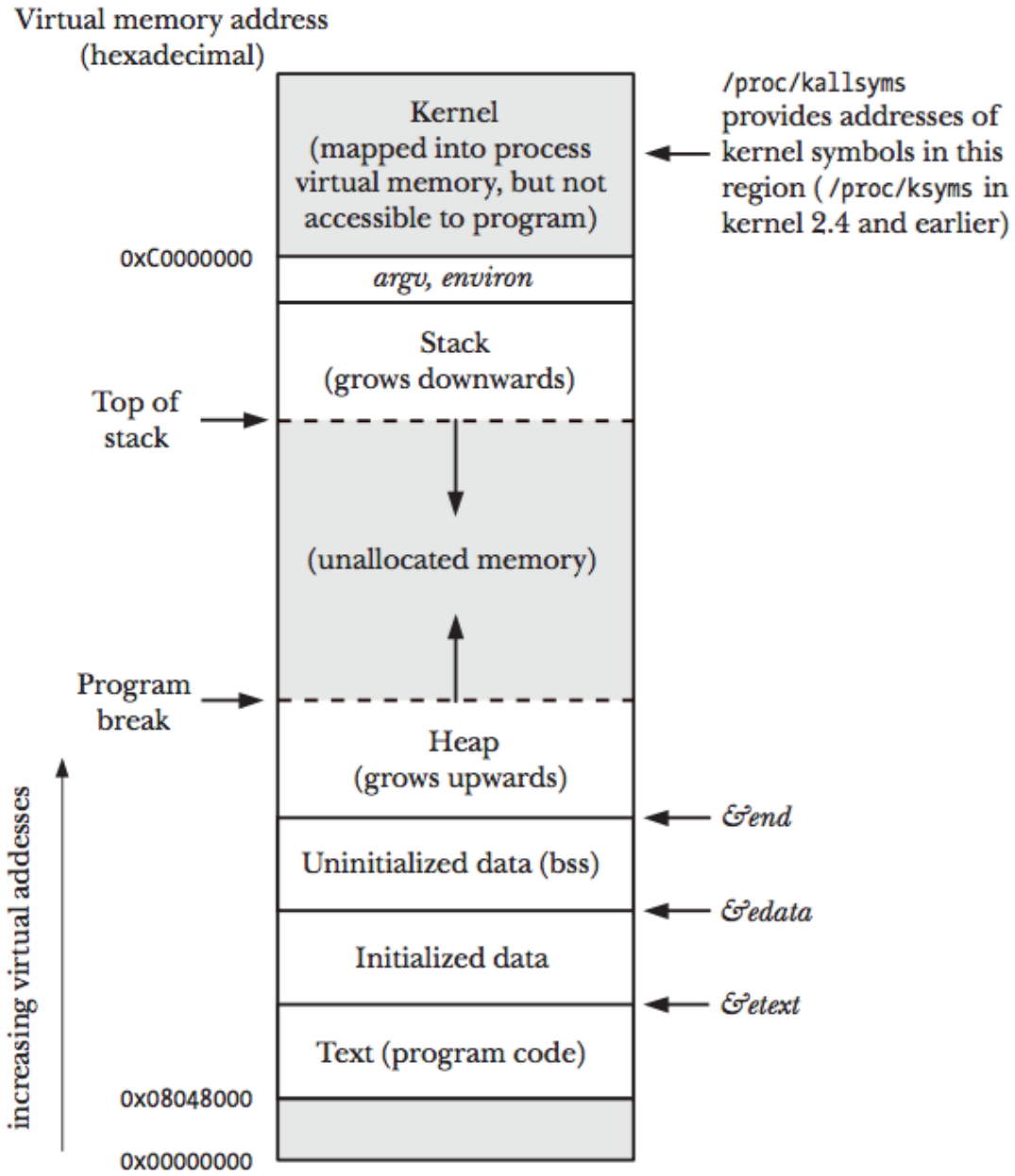
  > "A process is a program in execution; it is the unit of work in a system."

- **Tanenbaum & Bos (2015):**

  > "A process is an instance of a program, together with the current values of the program counter, registers, and variables."

Components of a Process

1. **Program code (text section)**: the instructions being executed.
2. **Data section**: global variables.
3. **Stack**: function calls, local variables, return addresses.
4. **Heap**: dynamically allocated memory during execution.



Components of a Process

Execution Context:

The execution context isn't inside the process's virtual memory space (the diagram). It refers to the dynamic state of the process that the OS must save and restore during a context switch.

It includes:

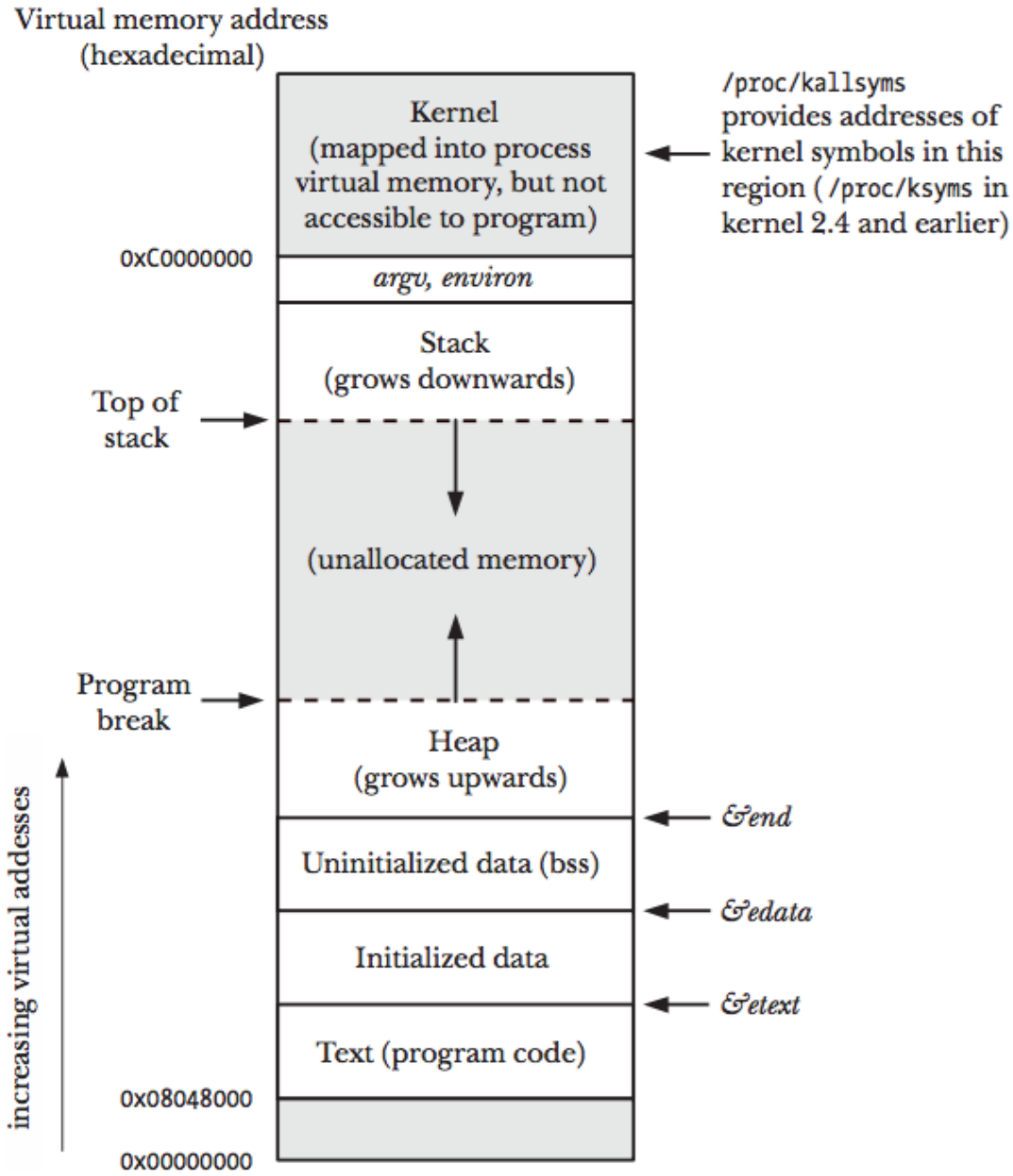
Program counter (PC) → points to the next instruction in the text section.

CPU registers → hold working variables and intermediate values.

Stack pointer → points into the stack region.

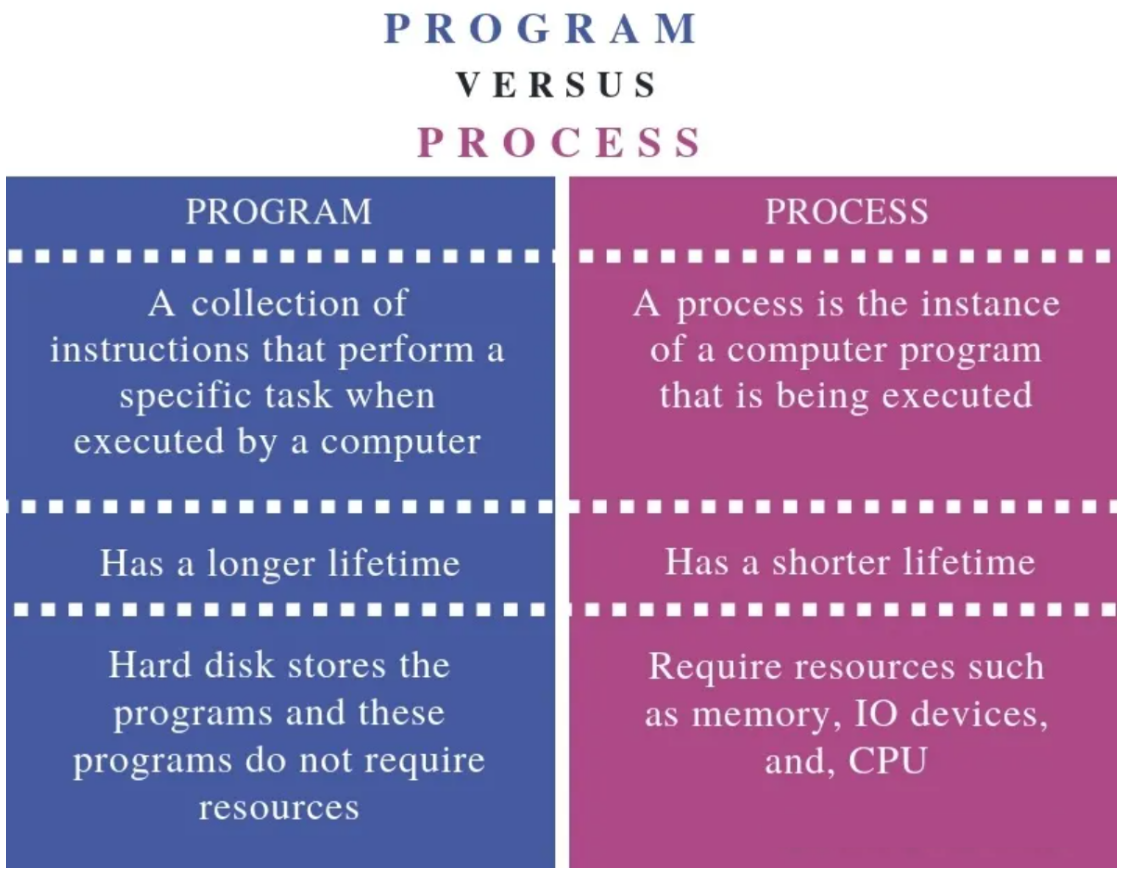
Open file descriptors → tracked by the OS, not stored inside this memory map.

Memory mappings → link the process to these regions (text, data, heap, stack).



Processes vs. Programs

- **Program:** Passive, stored on disk, contains instructions only.
- **Process**: Active, loaded into memory, has an execution state.
- One program can spawn multiple processes (e.g., opening multiple browser windows).



Process Abstraction

- Processes are an abstraction of the CPU.
- They give each program the illusion of exclusive access to the processor.
- The OS handles scheduling and context switching to multiplex the CPU between processes.

- When you run a .EXE file, the OS creates a process = a running program
- OS timeshares CPU across multiple processes: virtualises CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU:
  - Policy: which process to run
  - Mechanism: how to “context switch” between processes.

Multiple Processes in Action

- Example: A running laptop may have hundreds of processes:
  - System daemons (e.g., `systemd`, `csrss.exe`).
  - User applications (browser, word processor).
  - Background tasks (indexing service, updater).
- Each process has its own address space and resources, isolated by the OS.

Process Lifecycle

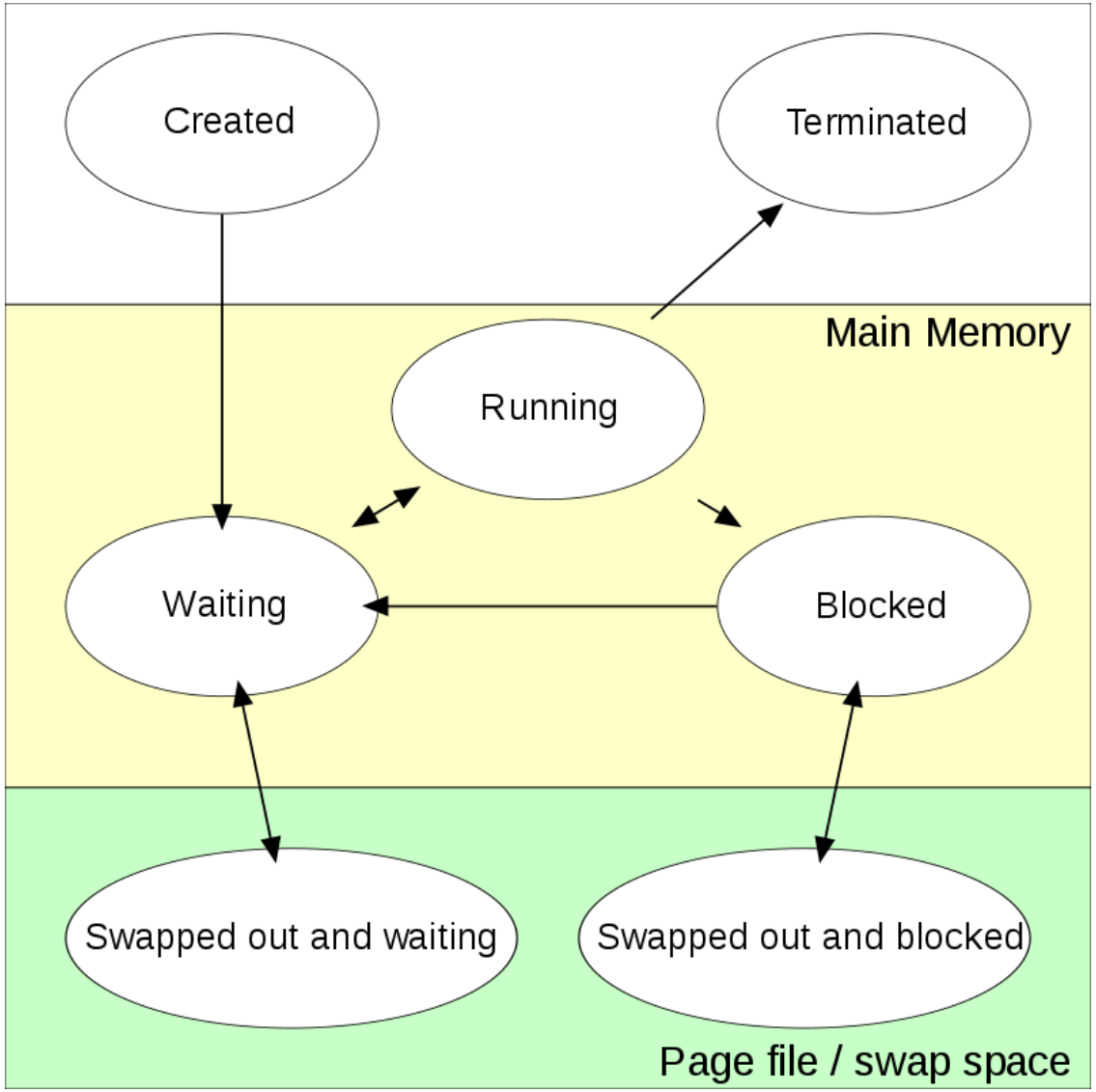
Process Lifecycle
Introduction

- Processes are dynamic entities: they are created, move between states, and eventually terminate.
- The process lifecycle models these state transitions.
- Both *Tanenbaum* and *Silberschatz* emphasise that the lifecycle allows the OS to efficiently allocate CPU, memory, and I/O resources.

Process Lifecycle

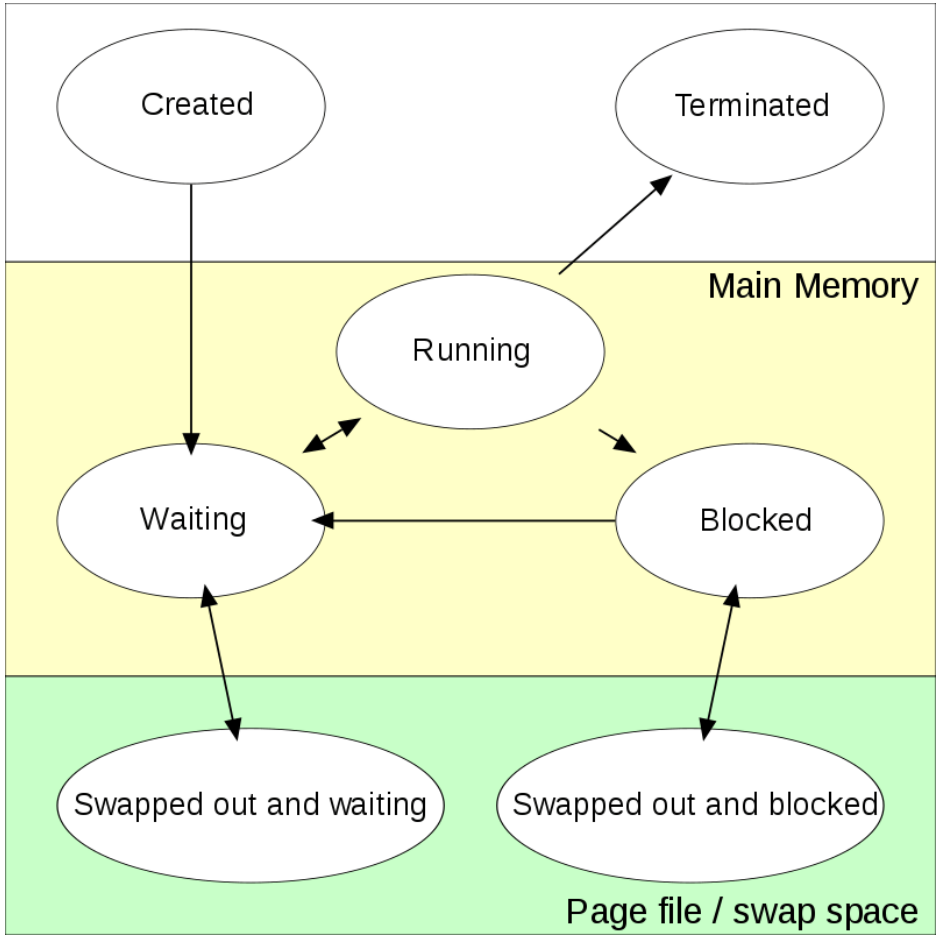
Process States (Classic 5-State Model)

1. New: process is being created.
2. Ready: process is loaded in memory, waiting for CPU.
3. Running: process is actively executing instructions on the CPU.
4. Waiting (Blocked): process cannot proceed until an event occurs (e.g., I/O completion).
5. Terminated: process has finished execution and is awaiting cleanup.



Process Lifecycle
State Transitions

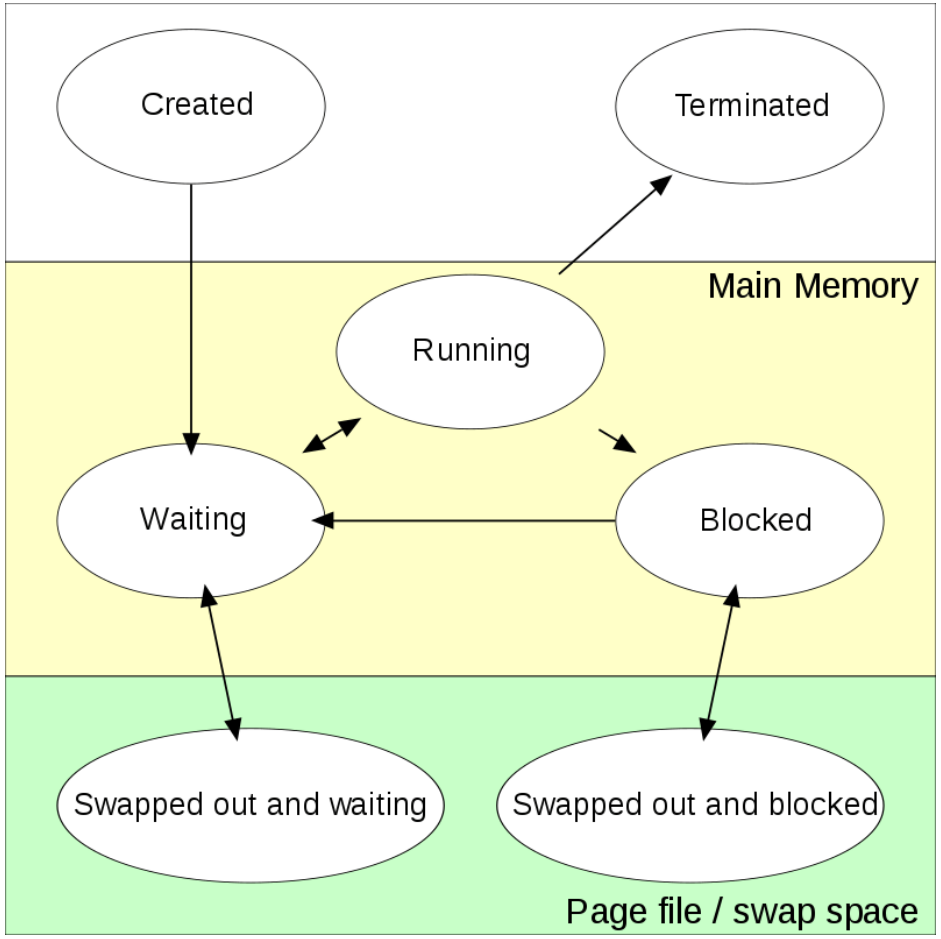
- Created → Waiting: When a new process is admitted into the system, it moves into memory and waits to be scheduled.
- Waiting ↔ Running: The scheduler selects a waiting process for CPU execution (Waiting → Running). When a running process is **pre-empted** or yields the CPU, it returns to Waiting (Running → Waiting).
- Running → Blocked: A process requests I/O or some event it must wait for. It cannot continue until the event completes.
- Blocked → Waiting: Once the I/O or event completes, the process becomes ready to run again.
- Running → Terminated: The process finishes execution or is killed by the OS/user.



Process Lifecycle
State Transitions

Swapping (Suspended States):

- Waiting → Swapped Out and Waiting: If memory is scarce, the OS may swap a waiting process out to disk.
- Swapped Out and Waiting → Waiting: The process is swapped back into memory when resources are available.
- Blocked → Swapped Out and Blocked: If a blocked process cannot remain in main memory, the OS may suspend it to swap space.
- Swapped Out and Blocked → Blocked: When swapped back into memory, it remains blocked until its event completes.



Process Lifecycle
Why Lifecycle Matters

- The lifecycle explains how multitasking is possible.
- OS must quickly switch between states via scheduling and context switching.
- Ensures fairness (CPU shared between processes) and responsiveness (I/O not blocked by CPU tasks).

Process Control Block (PCB)

Process Control Block (PCB)
Introduction

- To manage multiple processes, the OS must keep track of each process's state.
- Both Tanenbaum and Silberschatz emphasise the Process Control Block (PCB) as the fundamental data structure that stores all the information needed to manage and resume a process.
- Without the PCB, multitasking and context switching would not be possible.

Definition

- **Silberschatz**: A PCB is a repository for all information associated with a process.
- **Tanenbaum**: It is the "*roadmap*" for a process – containing everything the OS needs to suspend and later restart it.

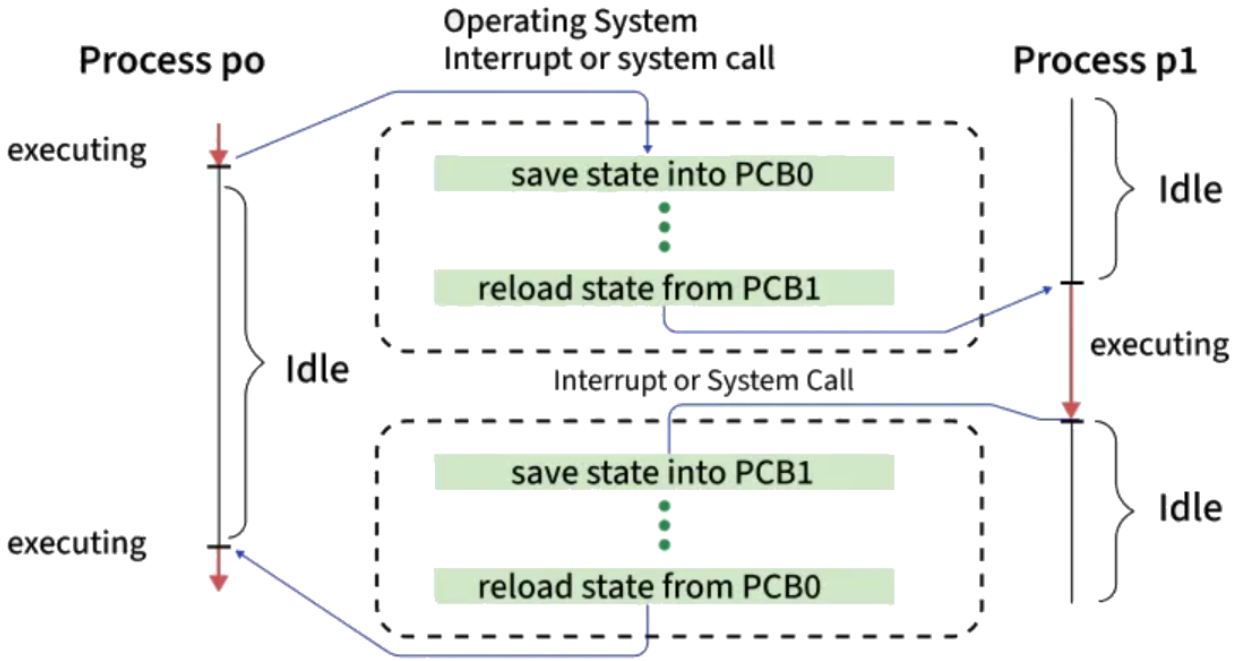
Contents of a PCB

A PCB typically includes:

1. Process Identification
   - Process ID (PID)
   - Parent process ID
   - User ID (owner of the process)
2. Processor State Information
   - Program Counter (PC): next instruction to execute
   - CPU Registers: general-purpose, stack pointer, etc.
   - Status word/flags
3. Scheduling Information
   - Process state (ready, running, waiting, etc.)
   - Priority level
   - CPU scheduling info (time slice, queue pointers)
4. Memory Management Information
   - Base and limit registers
   - Page tables or segment tables
   - Memory allocation details
5. Accounting Information
   - CPU time used
   - Execution time limits
   - Process creation time
6. I/O Status Information
   - List of open files
   - I/O devices allocated
   - Pending I/O operations

Process Control Block (PCB)
PCB and Context Switching

- When a process is pre-empted (interrupted by the scheduler), the OS saves the CPU state into the PCB.
- To resume the process, the OS restores the saved values from the PCB.
- This mechanism enables multitasking and ensures processes continue seamlessly.



Process Control Block (PCB)
Example Across OS

Linux: PCBs are implemented as the `task_struct` data structure.
It contains scheduling information, process state, memory maps, and links to parent/child processes.

Windows: Uses an **EPROCESS** structure in kernel space. It stores scheduling, memory, and handle table information. Each thread also has its own **ETHREAD** structure.

macOS (XNU kernel): Uses a combination of Mach tasks and BSD processes.

- **Mach task**: Represents the resources allocated to a process (virtual memory, threads).
- **BSD process structure (proc)**: Provides Unix/POSIX semantics such as file descriptors, credentials, and signals.
- Together, they serve the same role as a PCB: maintaining all state needed for process execution and management.

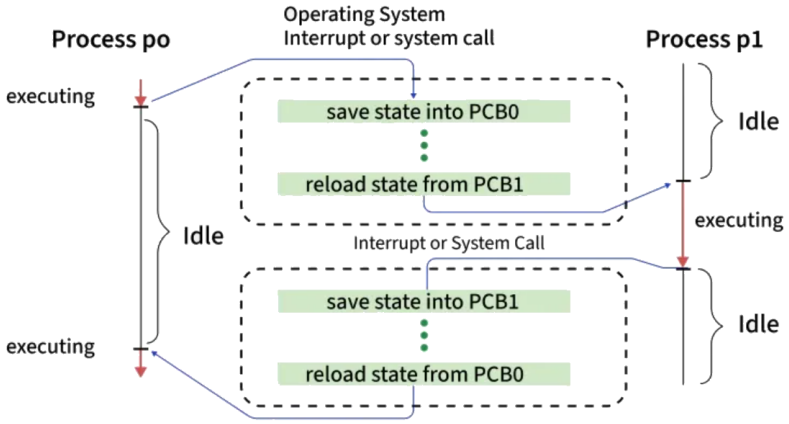
Process Control Block (PCB)
Why PCBs Matter

The PCB ensures:

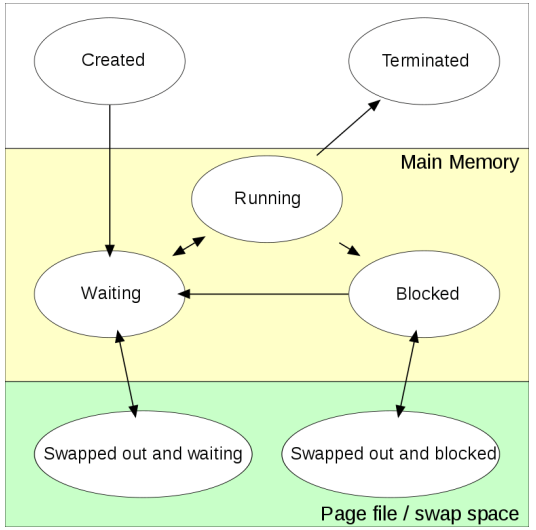
- Correctness: Processes resume exactly where they left off.
- Isolation: Each process has a private execution context.
- Efficiency: Enables pre-emptive multitasking and fair scheduling.

Process ID
Process state
Process priority
Accounting information
Program counters
CPU registers
PCB pointers
List of open files
Process I/O status Information
.....

Process Control Block (PCB)



Multitasking through Context Switches



Processes (and their threads) transition between states

Context Switching

Context Switching
Introduction

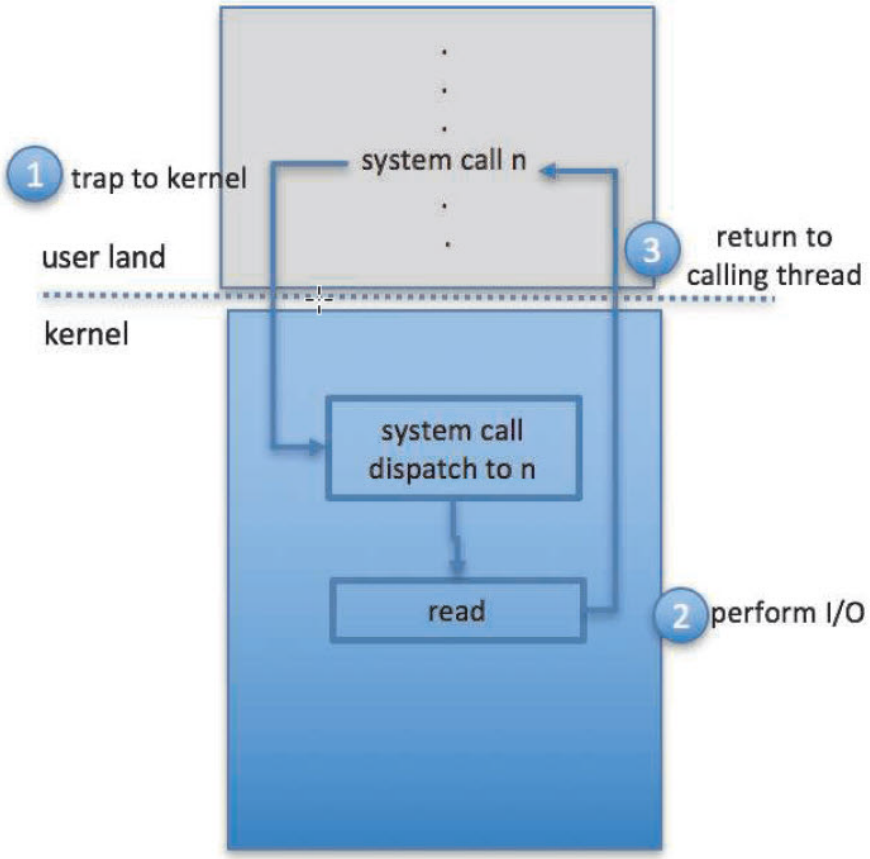
- In a multiprogramming OS, many processes compete for the CPU.
- The OS gives the illusion of **concurrent execution** by switching rapidly between them.
- Both *Tanenbaum* and *Silberschatz* describe **context switching** as the mechanism that enables multitasking.

Definition

- A **context switch** is the act of saving the state of the currently running process and loading the state of the next scheduled process.
- The “context” includes all information needed to **resume execution later**.

Kernel/User Switch
Not the same as a **Context Switch!**

- A kernel/user switch happens when the CPU changes from user mode (running application code) to kernel mode (running OS code), or back again.
- This is triggered by:
  - System calls (e.g., `open()`, `read()` → need OS services).
  - Interrupts (hardware event → OS must handle).
- What changes?
  - CPU switches privilege level (from user to kernel).
  - Execution jumps to kernel code (e.g., system call handler).
- What doesn't change?
  - **The same process is still running**. Its PCB, memory context, and process ownership remain unchanged.



Use of a system call to perform I/O.

Context Switching
What Is Saved in a Context Switch

The OS stores process state in the Process Control Block (PCB):

- Program Counter (next instruction to execute).
- CPU Registers (general-purpose, stack pointer, instruction registers).
- Memory management info (page/segment tables).
- Process state (ready, waiting, running).
- Accounting info (CPU usage time).

Context Switching
When Context Switching Occurs

- Timer interrupt (time slice expires) → pre-emptive scheduling.
- I/O request → process blocks, OS switches to another.
- Interrupts → hardware event requires attention.
- System calls → a process requests OS service and may block.
- Priority scheduling → higher-priority process becomes ready.

Context Switching
Costs of Context Switching

- Overhead: The CPU does no useful work during the switch:
- Context switching activities involve saving/restoring registers, flushing pipelines, cache disruption.
- More frequent context switches = less efficiency. Why?
  - Short CPU time slices → high proportion of CPU time wasted in switching.
  - Long CPU time slices → low proportion wasted, but reduced responsiveness.

Context Switching
Processes vs Threads

- Processes: More expensive — involves switching memory context, page tables, file tables.
- Threads (within the same process): Lighter, since they share memory space; only registers and stack need switching.
- Tanenbaum highlights: lightweight context switches make threads attractive for concurrency.

Context Switching
Example Across OS

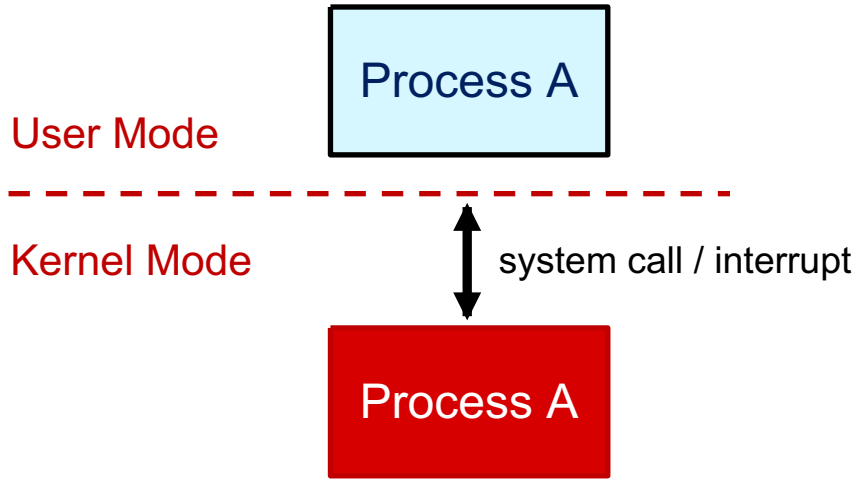
- Linux: The scheduler selects the next process; the kernel saves/restores CPU registers in the `task_struct` PCB.
- Windows: Uses the Dispatcher; context switch involves switching thread control blocks (TCBs) and updating CPU state.
- macOS: Built on Mach; context switch occurs between threads, since the Mach kernel manages threads as the basic unit of execution.

Context Switching
Why Context Switching Matters

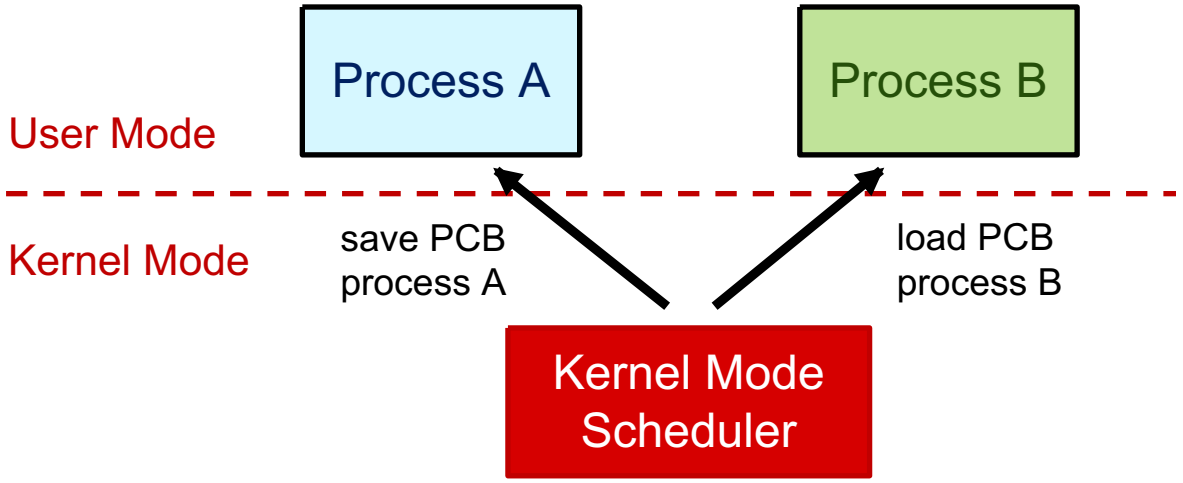
- Enables multitasking and fair CPU sharing.
- Provides responsiveness: interactive tasks can interrupt background tasks.
- Ensures process isolation: each process resumes correctly without interfering with others.
- Foundation for scheduling algorithms (to be covered in Topic 4: Threads and Scheduling).

Kernel/User Switch vs Context Switch

Kernel / User Switch



Context Switch



- **Kernel/User Switch**: Same process moves between user mode and kernel mode during a system call or interrupt (no PCB swap).

- **Context Switch**: Scheduler stops Process A and starts Process B, saving/loading PCBs.

Process Creation & Termination

Process Creation & Termination
Introduction

- An operating system must provide mechanisms to create new processes and cleanly terminate them.
- Both *Tanenbaum* and *Silberschatz* emphasise this as essential for multiprogramming, where processes dynamically start, run, and end.

Process Creation & Termination

1. Process Creation

New processes may be created for:

- System initialization (daemons, services at boot).
- User request (starting an application).
- Process spawning (a running process creates a child).
- Batch jobs or scheduled tasks.

Process Creation & Termination
Mechanisms in practice:

UNIX/Linux:

- `fork()` → duplicates the parent process, creating a child with a new PID.
- `exec()` → replaces the child's memory image with a new program.
- `wait()` → parent waits for child to finish.

Windows:

- `CreateProcess()` → single API call that creates a new process and loads an executable.
- More complex, but provides greater control (security attributes, environment, priority).

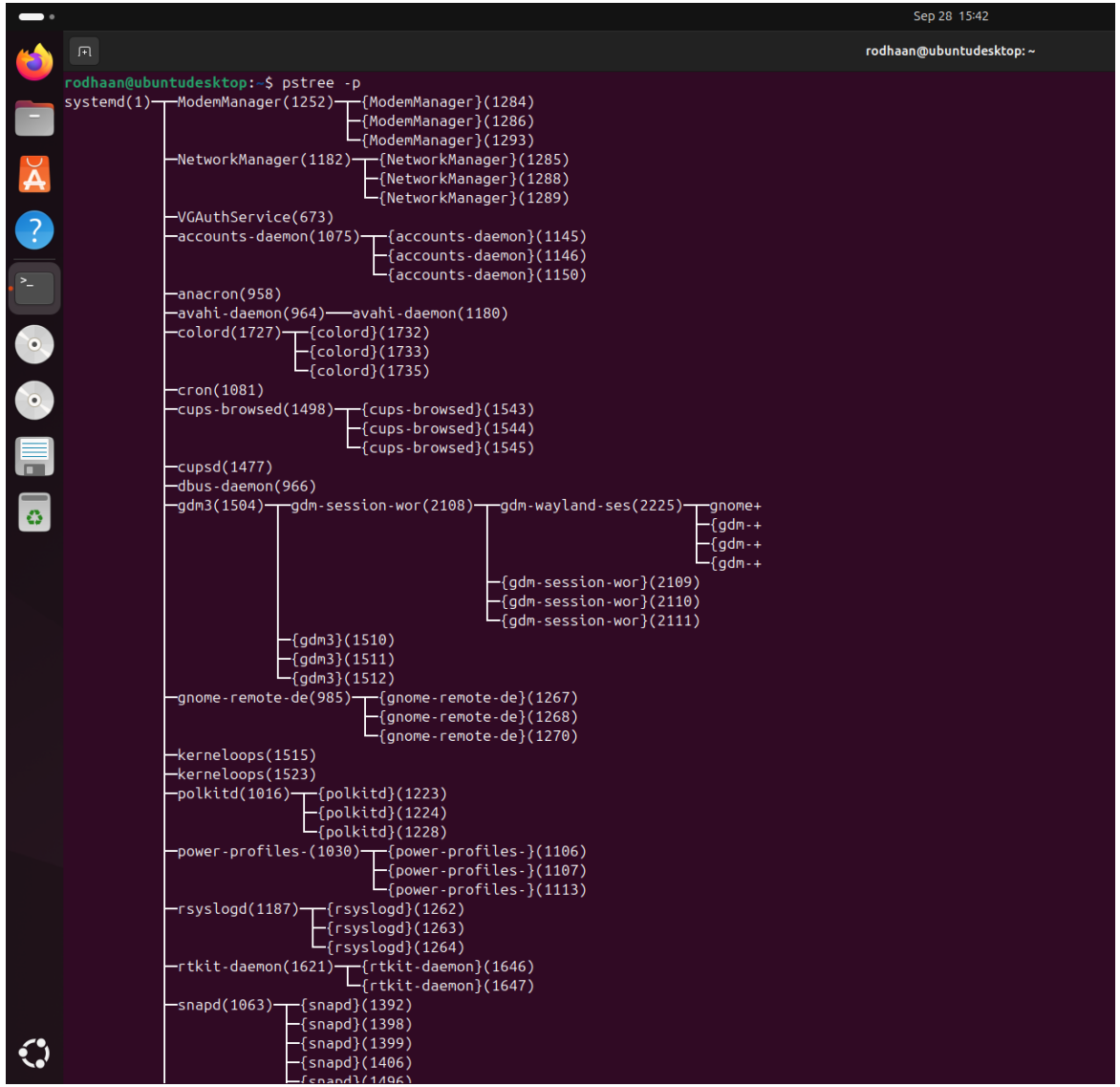
macOS (XNU):

- Follows UNIX semantics (BSD `fork`, `exec`, signals), layered on Mach tasks.

Process Creation & Termination
Process Trees:

- In UNIX-like systems, all processes are descendants of `init/systemd`.

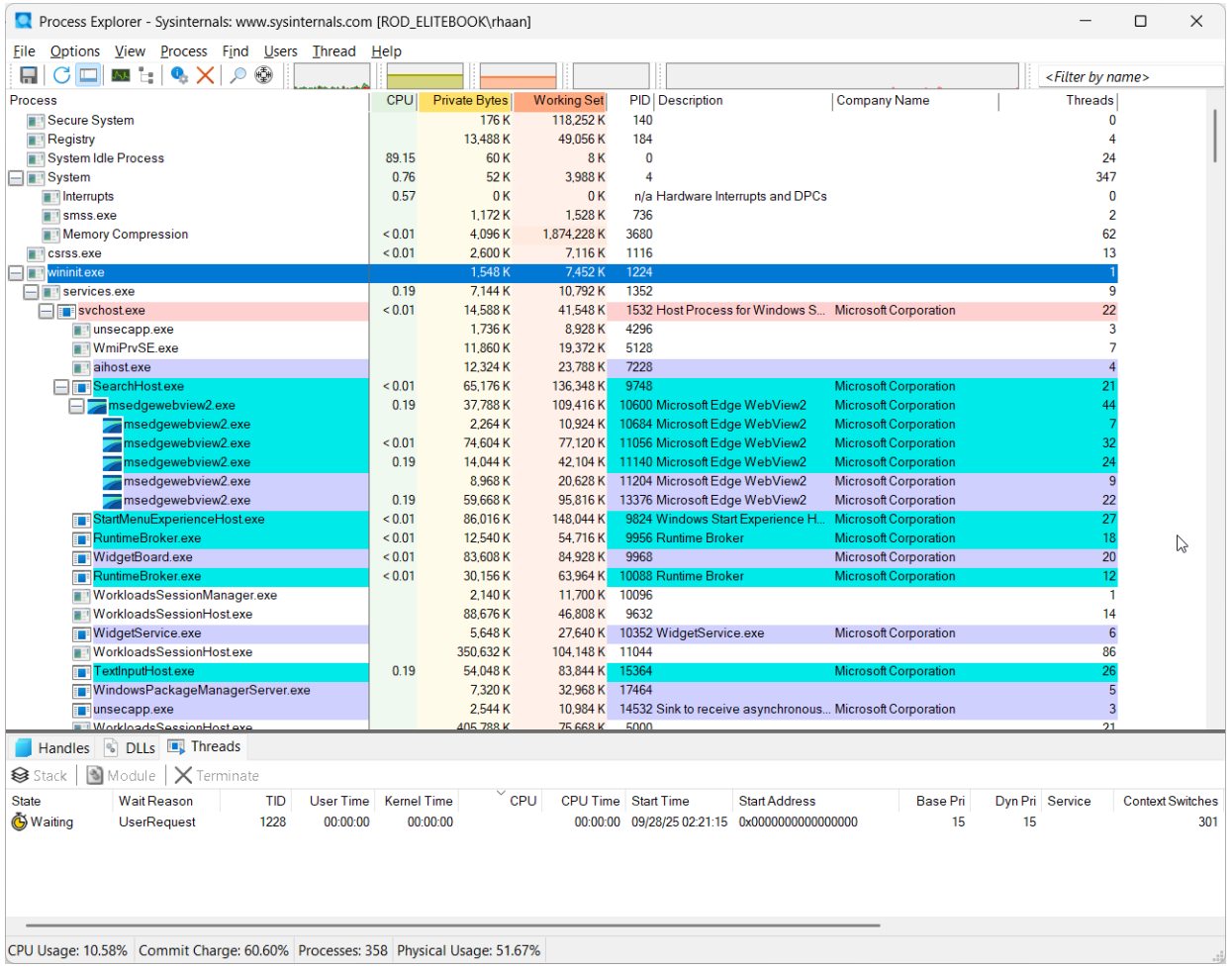
Linux
(Process hierarchy – using `pstree -p`)



Process Creation & Termination
Process Trees:

- In Windows, each user session has system processes (e.g., `smss.exe`, `wininit.exe`) that spawn application processes.

Windows
(Process hierarchy – using Process Monitor)



Process Creation & Termination
2. Process Termination

Processes may terminate:

- Normal exit (voluntary): Program completes and calls exit().
- Error exit (voluntary): Program detects an error and terminates.
- Fatal error (involuntary): Caused by illegal instruction, divide by zero, memory access violation.
- Killed (involuntary): Another process or the OS forcibly terminates it.

Mechanisms in practice:

- **UNIX/Linux**: `exit()`, `kill -9 <pid>`, signals (e.g., `SIGTERM`, `SIGKILL`).
- **Windows**: `ExitProcess()`, Task Manager "**End Task**", `TerminateProcess()` system call.

Process Creation & Termination
3. Parent–Child Relationships

- Parents may explicitly wait for children to finish (`wait()` in UNIX).
- If parent exits before child → child becomes an orphan (adopted by `init/systemd` in Linux).
- If child terminates but parent does not collect exit status → child becomes a **zombie** process (PCB remains until parent calls `wait()`).

Process Creation & Termination
Why This Matters

- Process creation and termination mechanisms ensure:
  - Resource management (CPU, memory freed when process ends).
  - System stability (zombies and orphans handled properly).
  - Security (OS enforces permission checks on who can create or kill processes).
- This is where abstract concepts (PCB, lifecycle) meet practical system calls.

Summary (Key Takeaways)

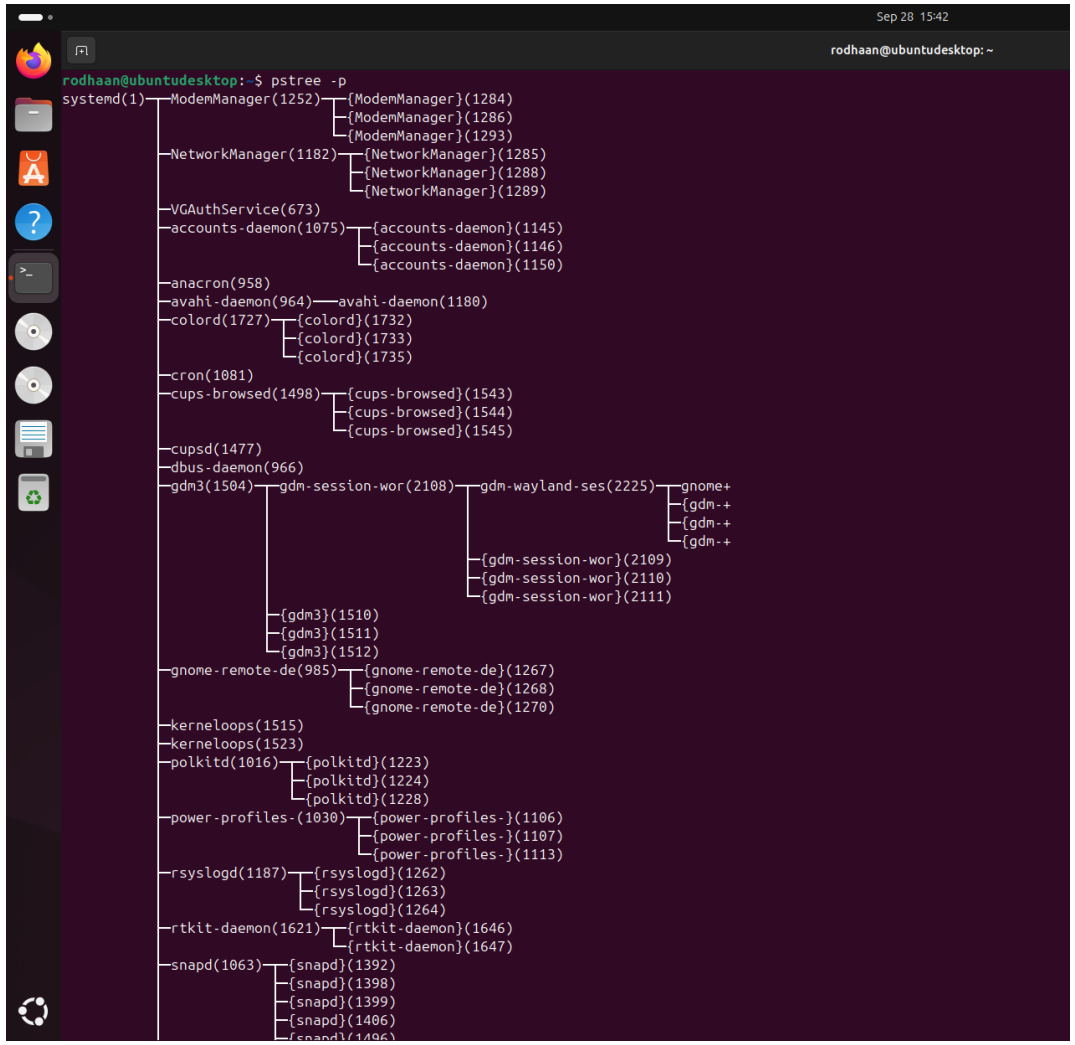
- Processes are created by system initialization, user requests, or other processes.
- UNIX uses `fork` / `exec`, while Windows uses `CreateProcess`.
- Processes terminate voluntarily (exit, error) or involuntarily (killed, crash).
- Parent–child relationships lead to orphans and zombies if not managed properly.
- Clean process creation/termination is fundamental to stable, multiprogramming systems.

Process Hierarchies & Scheduling Basics

Process Hierarchies & Scheduling Basics

Introduction

- Processes are rarely isolated — they often create other processes.
- The OS must manage these **parent–child relationships** in a process tree.
- At the same time, multiple runnable processes compete for the CPU, requiring scheduling.
- This section introduces how hierarchies are structured and the basics of scheduling decisions.



Process Hierarchies & Scheduling Basics

Parent–Child Relationship:

- A process may create another process (child).
- The parent may continue execution in parallel or wait for the child to finish.

UNIX/Linux:

- All processes descend from init (PID 1, now systemd).
- Commands: `pstree`, `ps -o pid,ppid,comm`.
- Orphans: adopted by `init`.
- Zombies: child is terminated, but parent hasn't collected exit status.

Windows:

- `CreateProcess()` creates new processes.
- Each process has a parent, though Windows does not strictly enforce process-tree termination rules like UNIX.

Process Hierarchies & Scheduling Basics

Why Process Hierarchies Matter:

- **Resource Management**: Parents may share resources with children or restrict them.
- **Security**: Access control often enforced through parent–child chains (e.g., privileges inherited).
- **Process Groups**: Used for signalling in UNIX (e.g., sending kill to a process group).

Process Hierarchies & Scheduling Basics
Introduction to Scheduling

The goal: Decide which ready process gets the CPU next.

- The OS scheduler ensures fairness, efficiency, and responsiveness.
- **Three Levels of Scheduling (Silberschatz)**:
  - **Long-term scheduling**: Admission of jobs into the system. Controls the degree of multiprogramming.
  - **Medium-term scheduling**: Suspends/resumes processes to improve mix and free resources.
  - **Short-term (CPU) scheduling**: Chooses which ready process gets the CPU. Most visible to users.

Process Hierarchies & Scheduling Basics
Non-Pre-emptive Scheduling vs Pre-emptive Scheduling

- **Non-pre-emptive**: Once a process has the CPU, it keeps it until it blocks or finishes.
  - Example: First-Come, First-Served (FCFS).
- **Pre-emptive**: Scheduler may interrupt a process to switch to another (e.g., time slice expired).
  - Example: Round Robin (covered in Week 4).
- Trade-off: Pre-emption improves responsiveness but adds context-switching overhead.

Process Hierarchies & Scheduling Basics
Key Criteria in Scheduling Decisions

(Both Tanenbaum and Silberschatz list these as fundamental goals.)

- **CPU Utilisation**: Keep CPU as busy as possible.
- **Throughput**: Maximise number of processes completed per unit time.
- **Turnaround Time**: Time to execute a given process from submission to completion.
- **Waiting Time**: How long a process spends in the ready queue.
- **Response Time**: Time from request submission until first response (important for interactive systems).

Lesson Review

- Processes form **hierarchies**: parents spawn children, creating a process tree.
- UNIX enforces hierarchy strongly; Windows is more flexible.
- Schedulers operate at **three levels** (long-, medium-, short-term).
- **Non-pre-emptive vs pre-emptive** is the core distinction in CPU scheduling.
- Basic performance metrics (utilisation, throughput, turnaround, waiting, response time) guide scheduling design.

In-Class Exercise

In-Class Exercise:
1. Observe Parent–Child Relationships
2. Kill Child Process

Observe Parent–Child Relationships

1. Start Ubuntu VM in the hypervisor.
2. Open the **Terminal** app.
3. Create a process:
   - `Sleep 200 &` [This creates a process called sleep which will run for 200 seconds (`200`) and as a background process (`&`)]
   - (note that the process ID will be displayed underneath).

```
rodhaan@ubuntudesktop:~$ sleep 200 &
[1] 3469
```

4. List the process ID (pid) of the “sleep” process and the parent process that spawned it:
   - `ps -o pid,ppid,comm | grep sleep`

```
rodhaan@ubuntudesktop:~$ ps -o pid,ppid,comm | grep sleep
   3469    3309 sleep
```

[first pid is the “sleep” process. The next is the parent pid (ppid)]

In-Class Exercise:
1. Observe Parent–Child Relationships
2. Kill Child Process

Observe Parent–Child Relationships

5. Check that the process is running:
   - `pstree -p` [scroll and find the sleep process]
   - What is the name of the parent process?

6. Also run `htop` [the process consumes almost no CPU. So sort by CPU% and find the “sleep” process.

In-Class Exercise:
1. Observe Parent–Child Relationships
2. Kill Child Process

Kill Child Process

7. Kill the “sleep” process.
   - `kill <pid>` [e.g. `kill 3469`]

8. Check that it’s no longer running:
   - `ps -aux | grep sleep` [you’ll sleep as part of the grep command but the sleep process should be shown as terminated]

```
rodhaan@ubuntudesktop:~$ ps -aux | grep sleep
rodhaan     3712  0.0  0.0   9144  2248 pts/0    S+   16:49   0:00 grep --color=auto sleep
[1]+  Terminated              sleep 200
```


Dorset
College Dublin
Educating since 1983