Chapter 2 – Processes and Threads (Notes)

1. Processes

- **Definition:** A program in execution. It's more than just code — it has a current state, memory, and resources.
- **Process Table:** Maintained by the OS; stores info like program counter, registers, state (ready, running, blocked).
- **Process States:**
  - **Running** – actually using the CPU.
  - **Ready** – waiting for CPU.
  - **Blocked** – waiting for I/O or event.
- **Context Switch:** Saving current process state and loading another. Costly because it involves kernel work.

2. Threads

- **Definition:** A smaller unit of execution inside a process ("lightweight process").
- Each process can have **multiple threads** sharing code and data but with their own program counter, stack, and registers.
- **Why threads?**
  - Easier to overlap computation and I/O.
  - Good for servers (one thread per request).
- **Pros:** Faster to create/switch than processes.
- **Cons:** Harder to program; bugs in one thread can crash the whole process.

3. User vs Kernel Threads

- **User-Level Threads:**
  - Managed by a library, not the OS.
  - Fast switching (no kernel call needed).
  - Problem: one thread blocks → whole process blocks.
- **Kernel-Level Threads:**

    - Managed by the OS.
    - True parallelism on multiprocessors.
    - Slower (more overhead).
- **Hybrid models:** Some systems use both.

4. Interprocess Communication (IPC)

Processes/threads need to talk to each other. Two main ways:

a) Shared Memory

- Multiple processes access the same memory area.
- Must use **synchronization** to avoid conflicts.

b) Message Passing

- Processes send and receive messages.
- Safer, but slower than shared memory.
- Example in UNIX: **pipes**, message queues.

5. Synchronization Problems & Solutions

a) Critical Section Problem

- Only **one process** should be in the critical section at a time.
- Requirements: **mutual exclusion, progress, bounded waiting.**

b) Software Solutions

- **Peterson's Algorithm** (two processes, uses turn + flag variables). Works but depends on atomic instructions.

c) Hardware Solutions

- **Disabling interrupts**: simple, but only works on single CPU.
- **Atomic instructions** (e.g., TSL – Test and Set Lock).

d) Semaphores

- Integer variable + two atomic operations:
    - **wait (P)** – decrease, block if < 0.
    - **signal (V)** – increase, wake process if needed.
- Used for mutual exclusion and synchronization.

e) Monitors

- High-level construct (language support).
- Ensures only one process executes monitor procedures at a time.

6. Classic Synchronization Problems

- **Producer–Consumer:** Producers add items to a buffer; consumers remove them. Use semaphores (empty, full, mutex).
- **Dining Philosophers:** Philosophers alternate between eating and thinking, need two forks. Demonstrates deadlock risk.
- **Readers–Writers:** Multiple readers can access database, but writers need exclusive access. Variations: reader-priority, writer-priority, fair.

7. Scheduling

How the OS picks which process/thread runs next.

- **First-Come, First-Served (FCFS):** Simple but can cause long waits.
- **Shortest Job First (SJF):** Minimizes average wait but needs knowledge of run times.
- **Round Robin (RR):** Each process gets a time slice (good for time-sharing).
- **Priority Scheduling:** High-priority jobs run first; risk of starvation.
- **Multilevel Queue:** Different queues for different types of jobs.
- **Multilevel Feedback Queue:** Jobs can move between queues (fairer).

8. Predicting CPU Bursts (Aging Algorithm)

- **Exponential Averaging Formula:**
  $T(n+1) = \alpha T(n) + (1 - \alpha)T(n-1)$

- Recent bursts have more weight.
- Example with α = 0.5 → balances old and new behavior.

9. Deadlocks

- **Conditions for Deadlock:**
  - Mutual Exclusion.
  - Hold and Wait.
  - No Preemption.
  - Circular Wait.
- Solutions: prevention, avoidance (Banker's Algorithm), or detection + recovery.

✅ Key Takeaways

- **Processes** are heavy; **threads** are light.
- **User threads** are faster but weaker than **kernel threads**.
- **IPC** = shared memory (fast, risky) vs message passing (safe, slower).
- **Synchronization** is about protecting critical sections.
- **Semaphores, monitors, and hardware instructions** help manage concurrency.
- Classic problems (Producer–Consumer, Philosophers, Readers–Writers) teach synchronization pitfalls.
- **Scheduling** tries to balance efficiency and fairness.
- **Deadlocks** happen if the four conditions hold; OS must prevent or handle them.