Student Details

Field	Value
Name	أحمد علي أحمد علي عثمان
Code	20240592
Section	1
Number	15

Operating System Assignment 2

Task 1

1. Describe the concept of process synchronization and its purpose.

Process synchronization is the coordination of multiple processes or threads to ensure orderly execution and maintain data consistency when accessing shared resources. Its primary purposes are:

- To prevent **race conditions**, where concurrent access to shared data leads to inconsistent results.
- To enforce **mutual exclusion**, ensuring only one process accesses a critical resource at a time.
- To enable **cooperation** among processes through signaling and waiting mechanisms.
- To prevent **deadlocks** and **starvation**.

2. Discuss the classic process synchronization problems and compare the different approaches used to solve the critical section problem.

Classic Synchronization Problems:

1. **Bounded-Buffer Problem:**
   - Involves a producer and consumer sharing a fixed-size buffer.
   - Synchronization ensures the producer doesn't overflow and the consumer doesn't read empty buffers.
2. **Readers-Writers Problem:**
   - Multiple readers can read simultaneously, but writers require exclusive access.
   - Solutions must prevent starvation and ensure fairness.
3. **Dining-Philosophers Problem:**
   - Philosophers share chopsticks; deadlock occurs if each picks up one chopstick and waits.
   - Solutions include resource ordering, limiting concurrent access, or asymmetric picking.

Approaches to Critical Section Problem:

1. **Software Solutions**: Peterson's algorithm for two processes (uses `turn` and `flag` variables).
2. **Hardware Solutions**: Atomic instructions like `test_and_set()` and `compare_and_swap()`.
3. **OS-Level Solutions**:
   - **Mutex Locks**: Simple binary locks with `acquire()` and `release()` operations (busy waiting).
   - **Semaphores**: Integer-based signaling mechanism with `wait()` (P) and `signal()` (V) operations.

- **Monitors**: High-level abstraction providing mutual exclusion and condition variables.

Comparison:

Approach	Pros	Cons
Peterson's Solution	No hardware support needed.	Limited to two processes; busy waiting.
Hardware Solutions	Simple, efficient for small sections.	Busy waiting; low-level complexity.
Mutex Locks	Easy to use; supported by OS.	Busy waiting in spinlocks.
Semaphores	Flexible; can solve various problems.	Incorrect use may lead to deadlock.
Monitors	Reduces programmer errors; language-level support.	Not available in all languages.

3. Define the following terms:

- **Critical Section:**
  A segment of code in which a process accesses shared resources. Only one process should execute its critical section at a time.
- **Mutual Exclusion Locks (Mutex):**
  A synchronization mechanism that ensures only one thread or process can enter a critical section at a time. It provides `lock()` and `unlock()` operations.
- **Semaphores:**
  An integer synchronization variable that controls access to shared resources via `wait()` (decrement) and `signal()` (increment) operations. Operations are atomic and can be **binary** (0/1) or **counting** (≥0). Used for signaling between processes

Task 2

4. Describe the characteristics of deadlock and explain how deadlocks can be detected and avoided.

Characteristics (Necessary Conditions):

1. **Mutual Exclusion**: Resources cannot be shared.
2. **Hold and Wait**: Processes hold resources while waiting for others.
3. **No Preemption**: Resources cannot be forcibly taken.
4. **Circular Wait**: A cycle exists in the resource-wait graph.

Deadlock Detection:

- **Resource-Allocation Graph (RAG):**
  If the graph contains a cycle, deadlock may exist (certain if only one instance per resource type).
- **Wait-for Graph:**
  A simplified version of RAG for single-instance resources.
- **Banker's Algorithm (Detection Version):**
  Use the **detection algorithm** with `Available`, `Allocation`, and `Request` matrices ($O(m \times n^2)$ complexity). to check for safe states.
- **Multiple Resource Types**:
  Use the **detection algorithm** with `Available`, `Allocation`, and `Request` matrices (O(m×n²)

complexity).

Deadlock Avoidance:

- **Banker's Algorithm**: Requires processes to declare maximum resource needs in advance. It checks for **safe states** before allocating resources (i.e., there exists a sequence where all processes can finish).
- **Resource-Allocation Graph Algorithm**: For single-instance resources, denies requests that create cycles.

5. Discuss the main techniques used to recover from deadlocks.

1. **Process Termination**:
   - **Abort all deadlocked processes:** simple but drastic.
   - **Abort one process at a time** until deadlock is resolved. Selection criteria include priority, execution time, and resources held.
2. **Resource Preemption**:
   - **Victim Selection**: Choose a process based on cost, priority, or resource usage.
   - **Rollback**: Restore the preempted process to a safe state and restart.
   - **Starvation Prevention**: Limit how often a process is selected as a victim.

6. Describe the low-level implementation of memory management, including the ideas of page tables and swapping.

Page Tables:

A data structure used by the MMU (Memory Management Unit) to translate **logical addresses** to **physical addresses**. Each entry maps a page number to a frame number. Can be hierarchical (multi-level) to save space.

- Map logical addresses to physical frames.
- Stored in main memory; accessed via **Page Table Base Register (PTBR)**.
- **Translation Lookaside Buffers (TLBs)** cache recent translations to speed up access.

Swapping:

A process can be temporarily moved out of main memory to disk (backing store) and brought back later. Used when memory is overcommitted.

- Moves processes between main memory and secondary storage (backing store).
- Used when memory is overcommitted; enables **virtual memory**.
- **Roll-out, roll-in:** Used in priority-based scheduling.
- **Swap time** depends on transfer speed and memory size.

7. Explain how segmentation and paging work and outline the different memory allocation strategies.

Segmentation:

- Divides memory into **logical segments** (code, data, stack, etc.).
- Uses a **segment table** with base and limit registers for address translation.
- Logical address: `<segment-number, offset>`.
- Supports protection and sharing but leads to **external fragmentation**.
- **External fragmentation** is a problem.

Paging:

- Divides memory into fixed-size **frames** and logical memory into **pages** of the same size.
- Uses a **page table** for address translation.
- Supports **shared pages** (e.g., code segments).
- **Internal fragmentation** can occur (last page may not be fully used).

Memory Allocation Strategies:

Strategy	Description	Pros	Cons
First-Fit	Allocate the first hole large enough.	Fast, simple.	External fragmentation.
Best-Fit	Allocate the smallest hole that fits.	Reduces wasted space.	Slow; leaves small fragments.
Worst-Fit	Allocate the largest hole.	Leaves large leftover holes.	Poor utilization; slow.
Buddy System	Divide memory into power-of-two blocks.	Reduces external fragmentation.	Internal fragmentation.
Compaction	Move processes to collect free space into one block.	Reduces external fragmentation.	Overhead; requires dynamic relocation.