## CE0418 Operating Systems - Detailed Exam Notes (Units I-III)

This document provides detailed notes for your Operating Systems exam, covering Units I-III of the syllabus. Each topic includes explanations, examples, and code snippets where applicable. Remember that this is for study and understanding; you should also refer to your textbooks and lecture notes.

**Unit I: Introduction to Operating Systems & Process Management [12 lectures]**

**1.1 Introduction to Operating Systems:**

**Definition: An OS is system software that manages computer hardware and software resources and provides common services for computer programs. It acts as an intermediary between the user and the hardware.**

**Types of Operating Systems:**

**Batch OS: Processes jobs sequentially in batches. Example: Early mainframe systems. No interactive user input during processing.**

**Multiprogramming OS: Executes multiple programs concurrently by switching between them rapidly. Improves CPU utilization. Example: Early time-sharing systems.**

**Multitasking OS: Allows multiple tasks (programs or processes) to run seemingly simultaneously on a single processor through time-slicing. Example: Windows, macOS, Linux.**

**Multiuser OS: Allows multiple users to access and use the system simultaneously. Example: Unix-like systems.**

**Parallel OS: Utilizes multiple processors to execute a single program or multiple programs concurrently, leveraging true parallelism. Example: Systems with multiple cores or CPUs.**

**Distributed OS: Manages a network of computers as a single system. Example: Network File Systems (NFS).**

**Real-time OS: Guarantees a response within a specific time constraint. Crucial for applications requiring immediate action, like industrial control systems. Example: Embedded systems in automobiles.**

**Operating System Objectives and Functions:**

**Resource Management: Efficiently allocate and manage CPU, memory, I/O devices, and files.**

**Abstraction: Hide hardware complexity from applications, providing a simpler interface.**

**Concurrency: Manage the execution of multiple programs concurrently.**

**Protection: Protect user programs and data from each other and from unauthorized access.**

**Reliability: Ensure system stability and fault tolerance.**

**Virtual Computers: An OS can create virtual machines (VMs), each with its own isolated operating environment. This enhances resource management and allows running different OSes on the same hardware. Examples: VirtualBox, VMware.**

**Interaction of OS & Hardware Architecture: The OS interacts directly with hardware components (CPU, memory, I/O controllers) through system calls and interrupts. This interaction is crucial for resource allocation and management.**

**Evolution of Operating Systems: The evolution showcases a progression from simple batch processing systems to complex, multitasking, and networked systems. Each generation improved upon efficiency and capabilities.**

**System Calls: Functions provided by the OS that allow programs to request services (e.g., file I/O, memory allocation).**

**OS Shell: A command-line interpreter that allows users to interact with the OS. Examples: Bash (Linux), PowerShell (Windows).**

**Example (Shell Script - Linux): A simple shell script to display current date and time:**

```bash

#!/bin/bash

date

```

**1.2 Process Management:**

**Process: A program in execution. It has its own memory space, registers, and other resources.**

**Process Description: A process is defined by its state (running, ready, blocked), program counter, memory allocation, and open files.**

**Process States:**

**New: Process is being created.**

**Ready: Process is waiting for the CPU.**

**Running: Process is currently executing.**

**Blocked (Waiting): Process is waiting for an event (e.g., I/O completion).**

**Terminated: Process has finished execution.**

**Process Control Block (PCB): A data structure that contains information about a process (PID, state, memory pointers, etc.).**

**Threads: Lightweight units of execution within a process. Multiple threads can run concurrently within the same process, sharing the same memory space. This improves concurrency and responsiveness.**

**Processes vs. Threads: Processes have separate memory spaces, while threads share the same memory space. Creating a new process is more resource-intensive than creating a new thread.**

**Uniprocessor Scheduling: Algorithms that determine which process gets the CPU on a single-processor system.**

**Scheduling Algorithms:**

**First-Come, First-Served (FCFS): Processes are executed in the order they arrive. Simple but can lead to long waiting times.**

**Shortest Job First (SJF): Processes with the shortest expected execution time are executed first. Optimal in terms of average waiting time but requires knowing execution times in advance.**

**Priority Scheduling: Processes are assigned priorities, and higher-priority processes are executed first. Can lead to starvation for low-priority processes.**

**Round Robin (RR): Each process gets a time slice (quantum) of CPU time. Fair but performance depends on quantum size.**

**Multilevel Feedback Queue Scheduling: Processes are placed in different queues with different priorities and time quantum. Attempts to balance fairness and performance.**

**Example (C code - Process Creation): A simple C program that creates a child process using `fork()`:**

```c

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed\n");

return 1;

} else if (pid == 0) {

// Child process

printf("Child process\n");

} else {

// Parent process

printf("Parent process\n");

}

return 0;

}

```

**Multiprocessor Scheduling: Scheduling algorithms designed for systems with multiple processors. These algorithms need to consider assigning processes to different processors efficiently.**

**Real-Time Scheduling: Scheduling algorithms that meet deadlines with predictable behavior. Often used in embedded systems and real-time applications.**

**Unit II: Concurrency & Deadlock [12 lectures]**

**2.1 Concurrency:**

**Principles of Concurrency: Managing multiple processes or threads running seemingly simultaneously. Requires mechanisms to avoid race conditions and ensure data consistency.**

**Critical Section: A section of code that accesses shared resources. Only one process or thread should be in the critical section at a time.**

**Mutual Exclusion: A mechanism that ensures only one process or thread can access a critical section at a time. Hardware or software solutions can be used.**

**Hardware Support: Atomic instructions (e.g., test-and-set) that ensure indivisible operations.**

**Software Approaches: Semaphores, mutexes, monitors, message passing.**

**Semaphores: Integer variables used for synchronization. Two operations: `wait()` (decrements the semaphore; blocks if negative) and `signal()` (increments the semaphore).**

**Mutexes (Mutual Exclusion): A binary semaphore (value 0 or 1) used to protect a critical section.**

**Message Passing: Processes communicate by sending and receiving messages. Provides a way for processes to synchronize and share data.**

**Monitors: High-level synchronization constructs that encapsulate shared data and synchronization operations.**

**Classical Problems of Synchronization:**

**Readers-Writers Problem: Multiple readers can access shared data simultaneously, but only one writer can access the data at a time. Requires careful synchronization to avoid data corruption.**

**Producer-Consumer Problem: One process (producer) produces data, and another process (consumer) consumes the data. Requires a buffer to store data and synchronization to avoid buffer overflow or underflow.**

**Dining Philosophers Problem: A classic problem illustrating the challenges of deadlock and starvation in concurrent systems. Five philosophers are seated at a circular table, each with a chopstick to their left and right. Each philosopher needs both chopsticks to eat. The problem highlights the difficulty of designing resource allocation algorithms that prevent deadlock and starvation.**

**Example (C code - Semaphore): A simple example using semaphores (implementation may vary depending on OS and library used):**

```c

// This code requires a semaphore library, the exact implementation will vary based on your system.

// This example is for illustrative purposes only.

// ... semaphore initialization ...

// Producer

wait(semaphore); // Acquire semaphore

// Access shared resource

signal(semaphore); // Release semaphore

// Consumer

wait(semaphore); // Acquire semaphore

// Access shared resource

signal(semaphore); // Release semaphore

```

**2.2 Deadlock:**

**System Model: A system model for deadlock analysis usually involves resources, processes, and their relationships.**

**Deadlock Characterization: A deadlock occurs when two or more processes are blocked indefinitely, waiting for each other to release the resources that they need. Four necessary conditions for deadlock:**

**Mutual Exclusion: Only one process can use a resource at a time.**

**Hold and Wait: A process holds at least one resource and is waiting for another resource held by another process.**

**No Preemption: Resources cannot be forcibly taken away from a process.**

**Circular Wait: A circular chain of processes exists, where each process is waiting for a resource held by the next process in the chain.**

**Methods for Handling Deadlocks:**

**Deadlock Prevention: Preventing one or more of the four necessary conditions from occurring.**

**Deadlock Avoidance: Dynamically ensuring that the system will never enter a deadlock state. Requires knowledge of resource requests.**

**Deadlock Detection: Allowing deadlocks to occur and then detecting them. Requires algorithms to detect cycles in the resource allocation graph.**

**Recovery from Deadlock: Breaking the deadlock by preempting resources or terminating processes.**

**Unit III: Memory Management & I/O Management [12 lectures]**

**3.1 Memory Management:**

**Memory Management Requirements: The OS needs to manage memory efficiently, protect processes from each other, and provide efficient mechanisms for allocating and deallocating memory.**

**Memory Partitioning: Dividing memory into partitions for allocation to processes.**

**Fixed Partitioning: Memory is divided into fixed-size partitions. Simple but can lead to wasted memory (internal fragmentation).**

**Dynamic Partitioning: Partitions are created dynamically as needed. Reduces wasted memory but can lead to external fragmentation.**

**Contiguous Memory Allocation: A process is allocated a contiguous block of memory. Simple but suffers from fragmentation.**

**Buddy System: A memory allocation algorithm that divides memory into powers of 2. Efficient in handling varying memory requests.**

**Memory Allocation Strategies: Algorithms to select a partition for a process.**

**First Fit: The first partition large enough is selected.**

**Best Fit: The smallest partition large enough is selected. Minimizes fragmentation but can be slow.**

**Worst Fit: The largest partition is selected. Attempts to reduce fragmentation but can be inefficient.**

**Next Fit: Similar to first-fit but starts searching from the location of the last allocation.**

**Fragmentation: Wasted memory due to inefficient allocation. Internal fragmentation (within a partition) and external fragmentation (between partitions).**

**Swapping: Moving a process's entire memory image to secondary storage (disk) to free up memory for other processes.**

**Segmentation: Dividing a program's address space into logical segments. Each segment has its own size and access rights.**

**Paging: Dividing both logical and physical memory into fixed-size blocks (pages and frames). Allows non-contiguous allocation of memory.**

**Virtual Memory: A technique that allows processes to use more memory than is physically available. Uses paging and swapping to manage memory.**

**Demand Paging: Pages are loaded into memory only when needed (on demand).**

**Page Replacement Policies: Algorithms to select a page to replace when a page fault occurs.**

**First In, First Out (FIFO): The oldest page is replaced. Simple but can suffer from Belady's anomaly.**

**Least Recently Used (LRU): The least recently used page is replaced. Optimal but requires tracking usage history.**

**Optimal: The page that will not be used for the longest time is replaced. Optimal but not practical because it requires future knowledge.**

**Clock Algorithm: An approximation of LRU that uses a circular queue and a "use" bit.**

**Thrashing: A situation where the system spends more time swapping pages than executing processes, leading to poor performance.**

**3.2 I/O Management and Disk Scheduling:**

**I/O Devices: Input/output devices (keyboard, mouse, printer, disk drives, etc.).**

**Organization of I/O Functions: The OS provides an interface to manage I/O devices. This interface includes device drivers and interrupt handling mechanisms.**

**Operating System Design Issues: The OS must handle interrupts efficiently and provide a uniform interface to different I/O devices.**

**I/O Buffering: Using buffers to store data temporarily, improving I/O efficiency.**

**Disk Scheduling: Algorithms to order requests to access disk blocks.**

**First-Come, First-Served (FCFS): Requests are processed in the order they arrive. Simple but can be inefficient.**

**Shortest Seek Time First (SSTF): The request that requires the shortest seek time is processed next. Reduces average seek time but can lead to starvation.**

**SCAN: The disk head moves in one direction, servicing requests along the way. Then reverses direction.**

**C-SCAN (Circular SCAN): Similar to SCAN but the head moves circularly, avoiding the "convoy effect."**

**Disk Caches: Buffers that store frequently accessed disk blocks in memory to improve performance.**

This detailed breakdown should help in your preparation. Remember to consult your textbooks and lecture notes for additional information and examples. Good luck with your exam!