Operating Systems Midterm Study Guide

30 select multiple
2 long answer

1. Introduction

The mode bit determines what? User or kernel mode
Asymmetric multiprocessing involves what? A single boss processor
The initial program that runs at startup is called what? The bootstrap
The operating system is a process that _____? Is always running
Which of the following is not a resource the OS allocates?
a. File storage system
b. User input
c. Computer memory
d. CPU
An interrupt is triggered by software only, hardware only, or both? Both, hardware and software
In order to prevent a user program from dominating the CPU what can be utilized? A timer
Storage device hierarchy?

1. Registers
2. Cache
3. Main memory
4. Solid state disk

Mainframe - designed primarily to optimize utilization of hardware
PC - support complex games, business applications, everything in between
Mobile - provide an environment for easy user interface

2. OS Structures

- ## Operating System Services

User Interface

- **Command-line interface (CLI):** Users interact by typing text commands.
- **Batch interface:** Users enter commands into files, which are then executed.
- **Graphical User Interface (GUI):** Window-based interaction using pointing or touch devices.

Program Execution

The OS loads programs into memory, runs them, and handles both normal and abnormal termination.

I/O Operations

The OS manages I/O devices, abstracting control for users to ensure efficiency and protection.

File-System Manipulation

Allows users to read, write, create, delete files, and manage permissions for access control.

Communications

Processes exchange information either via **shared memory** (both processes access a shared memory section) or **message passing** (packets exchanged between processes).

Error Detection

The OS handles various errors like memory faults, power failures, disk parity issues, and illegal operations.

Resource Allocation

Manages resources like CPU cycles, memory, and file storage, distributing them among processes.

Accounting

Tracks resource usage for users, useful for billing or analyzing usage patterns.

Protection and Security

Ensures that processes do not interfere with one another, and enforces authentication for users to access the system securely.

3. Processes

The heap section of a process in memory refers to what? The memory dynamically allocated at runtime
When a process is forked, what's the code to determine which process is the child? pid = fork(); if (pid == 0);
Interprocess communication methods:

1. Sockets
2. Message passing (pipes)
3. Shared memory

How many processes are spawned when fork() is called? 1
A process control block does what? It's used to save the context of a process when swapped
Pipes involve...
a. A read end and two write ends
b. A read and a write end
c. Multiple read and write ends
The number of multicore refers to what? The number of CPU's in a program
The degree of multiprogramming refers to what? The number of processes in a program
Which of the following is not a valid process state?
a. Ready
b. Waiting
c. Running
d. Completed

The text section of a process in memory refers to what? The program code

- ## States

  New - Process is being created
  Running - Instructions are executed
  Waiting - The process is waiting for some event to occur

Ready - The process is waiting to be assigned to a processor
Terminated - The process has finished execution

• Process Control Block

Process State: New, Ready, Running, Waiting, Terminated
Program Counter: Indicates the address of the next instruction to be executed
CPU Registers: Saves the state of CPU registers for when an interrupt occurs (e.g., accumulators, index registers, stack pointers)
CPU-Scheduling Information: Process priority, pointers to scheduling queues
Memory-Management Information: Base and limit registers, page tables, segment tables
Accounting Information: CPU time used, time limits, account numbers, process numbers
I/O Status Information: List of I/O devices allocated, list of open files

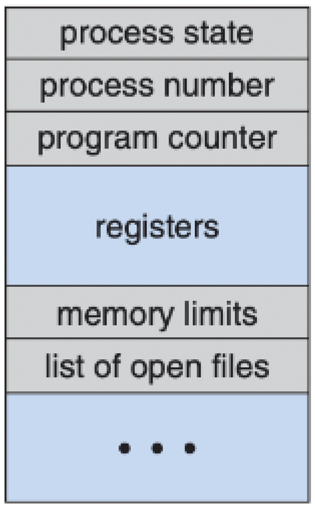

Figure 3.3 Process control block (PCB).

• Context Switching

Switching the CPU to another process, and saving the state of the current process is known as a context switch.

Schedulers

Multiprogramming: Goal is to have a process running at all times
Time Sharing: Switch processes frequently so that users can interact with them
Process Scheduler: Helps manage and switch between processes

Job Queue: All processes in a system
Ready Queue: Processes in main memory, ready and waiting to execute
Device Queue: Processes waiting for an I/O device; each device has its own queue

Long-term Scheduler:
- Selects processes from a pool to be executed
- Loads selected processes into memory
- Runs infrequently, takes longer to make decisions, not available on all systems

Short-term Scheduler:
- Selects from processes already in memory
- Runs frequently, needs to make faster decisions

Process Creation

A process (or task) can create another process, known as a **child process**, with the creating process called the **parent process**. This relationship can be represented as a **tree**, showing the parent and its children.

Resource Access: A child process accesses resources (memory, files, I/O devices, CPU time) directly from the Operating System or through resources partitioned by the parent.

Parent Behavior: The parent can either wait for the child to execute and terminate or continue executing concurrently.

Child Process Characteristics: A child can be a duplicate of the parent process (same program/data) or load a new program using **exec**, replacing its memory space.

Forking Limitations: The number of children a process can fork is limited by available memory and the maximum process ID (PID), typically around **4 million** on a 64-bit system, but often constrained by resources.

Interprocess communication

Cooperating Processes: Any process that can be affected by another process. A process that shares data with another process is a cooperating process.

Independent Processes: Any process that cannot affect or be affected by another process.

Reasons for Cooperation:

- **Information Sharing:** Allows sharing of data (e.g., shared files).
- **Computation Speedup:** Enables running tasks in parallel.
- **Modularity:** Organizes system functions into separate processes.
- **Convenience:** Facilitates working on multiple tasks simultaneously.

Two Fundamental Models for IPC:

- **Shared Memory:** Involves reading and writing data to a shared region of memory.
- **Message Passing:** Useful for exchanging smaller amounts of data.

4. Threads

• Benefits

Responsiveness: Threads improve the responsiveness of applications by allowing multiple tasks to be performed simultaneously.

Resource Sharing: Threads share the same memory space and resources, facilitating communication and data exchange.

Economy: Creating and managing threads is more efficient than processes. For example, in Solaris OS, creating a thread is **30 times faster**, and context switching is **5 times faster** with threads compared to processes.

Scalability: Threads can run on multiple processors, enhancing performance and scalability in multi-core systems.

• Data parallelism

Same task on different data.

Focus: Involves distributing *data* across multiple processors.
How it Works: The same operation is applied to different chunks of data simultaneously. Each processor works on a subset of the data, performing the same task.
Example: If you have a large array, each processor might handle a portion of the array, performing identical operations like summing or sorting.

Ideal For: Scenarios where the dataset is large, and the same computation needs to be performed on each portion of the data.

- ### Task parallelism

  Different tasks on the same or different data.

  Focus: Involves distributing *tasks* (or different parts of a program) across multiple processors.
  How it Works: Different processors execute different tasks or threads at the same time, which may or may not work on the same data.
  Example: One processor might handle data input, another processes that data, while another outputs the results.
  Ideal For: Situations where a program can be broken down into distinct tasks that can run independently or semi-independently of one another.

- ### Multithreading Models
  - #### One to One

    In this model, each user-level thread is mapped to a kernel thread. Creating a user-level thread creates a corresponding kernel thread. However, the number of threads per process may be restricted due to overhead. Examples include Windows, Linux, and Solaris 9 and later.

  - #### Many to One

    This model maps many user-level threads to a single kernel thread. If one thread blocks, all threads are blocked, meaning that multiple threads may not run in parallel on multicore systems since only one can be in the kernel at a time. Few systems use this model, including Solaris Green Threads and GNU Portable Threads.

  - #### Many to Many

    This model allows many user-level threads to be mapped to many kernel threads. It enables the operating system to create a sufficient number of kernel threads, facilitating better resource utilization and parallelism. This model was used in Solaris prior to version 9 and Windows with the ThreadFiber package.

5. Process synchronization

Which of the following is not a multi threaded model: User to kernel

In data parallelism: the task is split into subsets
A multithreaded process does not share what with the other threads? Stack
Which of the following is not a benefit of multithreaded programming: improved semaphores handling
Which of the following is not a thread library: linux
Which are: windows, pthreads, java
In multicore programming, identifying tasks to separate can be difficult

- ## Critical Section Problem
  - ### Criteria

    Mutual Exclusion: If process PiP_iPi is executing in its critical section, then no other processes can be executing in their critical sections.

    Progress: If no process is executing in its critical section and there are processes that wish to enter, the selection of the next processes to enter cannot be postponed indefinitely.

    Bounded Waiting: A limit must exist on the number of times other processes can enter their critical sections after a process has requested entry and before that request is granted.

  - ### Test and Set

    This is a hardware-based atomic instruction used to achieve mutual exclusion. It works by checking the value of a lock variable and setting it to "locked" if it is currently "unlocked." If multiple processes attempt to test and set the lock variable simultaneously, only one will succeed in acquiring the lock, while the others will be forced to retry.

  - ### Compare and Swap

    This is another hardware-based atomic instruction that compares the value of a variable to a given value and, if they match, swaps it with a new value. It is useful for implementing synchronization in a multi-threaded environment.

- ## Race Conditions

  Race Conditions:

  A **race condition** occurs when multiple processes or threads access shared data simultaneously, leading to unpredictable outcomes based on the timing of their execution.

Key Characteristics:

- **Shared Resources:** Arises from concurrent access to shared variables without synchronization.
- **Non-Deterministic Behavior:** The result varies depending on the execution order of processes.
- **Difficult to Debug:** Often intermittent and hard to reproduce.

Example:
Two processes, (P_1) and (P_2), increment a shared variable (X) (initially 5). If both read (X) before either writes back, they both may write 6 instead of the expected 7.

Prevention:
Use synchronization mechanisms like mutexes or locks to ensure that only one process accesses the critical section at a time, maintaining data integrity.

• Preemption

Preemption:

Preemption refers to the ability of an operating system to temporarily interrupt a currently running process or thread to allow another process to run. This is a key feature in multitasking systems, enabling fair resource allocation and improved responsiveness.

Key Points:

- **Context Switching:** When a process is preempted, the operating system saves its current state (context) and switches to another process. This involves storing the process's register values and program counter.
- **Time Sharing:** Preemption allows multiple processes to share CPU time, enabling a system to be more responsive to user interactions and ensuring that no single process monopolizes CPU resources.
- **Priority Scheduling:** Preemption often works in conjunction with scheduling algorithms, where higher-priority processes can preempt lower-priority ones.

Example:
In a time-sharing system, if a process is running and its allocated time slice expires, the scheduler can preempt it, saving its state and allowing a higher-priority process to run.
Importance:
Preemption enhances system responsiveness and ensures efficient CPU utilization by enabling the operating system to manage multiple processes effectively.

• Mutex Locks

Mutex locks are used to protect critical sections by ensuring that only one process or thread can access the critical section at a time.
Key Features:

- **Acquisition and Release:** A lock must be acquired before entering a critical section and released upon completion.
- **Atomic Operations:** Calls to `acquire()` and `release()` must be atomic, meaning they cannot be interrupted to maintain consistency.
- **Busy Waiting:** This solution often involves busy waiting, also known as a **spin lock**, where a process repeatedly checks if the lock is available.

Example Implementation:

```
acquire() {
    while (!available) { } /* busy wait */
    available = false;
}

release() {
    available = true;
}

do {
    acquire lock;
    // critical section
    release lock;
    // remainder section
} while (true);
```

Summary:
Mutex locks provide a straightforward mechanism for ensuring mutual exclusion in concurrent programming, although busy waiting can lead to inefficiencies.

• Semaphores

Semaphores:

A **semaphore** is an integer variable used for controlling access to a shared resource in a concurrent system. It is accessed through two atomic operations:
Key Operations:

- **wait() (proberen):** Tests the semaphore and potentially blocks the process if the resource is not available.

- **signal() (verhogen):** Increments the semaphore, potentially waking up a blocked process.

Example Implementation:

```
wait(S) {
    while (S <= 0) {} // busy wait
    S--;
}

signal(S) {
    S++;
}
```

Types of Semaphores:

- **Counting Semaphore:** Allows multiple processes to access a lock simultaneously.
- **Binary Semaphore:** Allows only one process to access the lock.

Avoiding Busy Waiting:

To prevent busy waiting and CPU cycle wastage, when a process executes `wait`, it can block itself. This operation places the process in a waiting queue associated with the semaphore, and control is transferred to the CPU scheduler to select another process to execute.

Structure for No Busy Waiting:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

No Busy Waiting Implementation:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        Add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        Remove a process P from S->list;
        wakeup(P);
    }
}
```

Summary:
Semaphores are an effective synchronization mechanism that allows processes to coordinate their actions without busy waiting, enhancing efficiency in concurrent programming.

- ## Classical Synchronization problems
    - **Reader/Writer**
    - **Producer/Consumer**
    - **Dining Philosophers**

- ## Deadlock Characterization

Mutual Exclusion:
At least one resource must be held in a non-shareable mode; that is, only one process can use the resource at any given time. If another process requests that resource, it must be delayed until the resource is released.

Hold and Wait:
A process holding at least one resource is waiting to acquire additional resources that are currently being held by other processes. This condition allows processes to hold onto resources while waiting for others, potentially leading to deadlock.

No Preemption:
Resources cannot be forcibly taken from a process holding them; a resource can only be released voluntarily by the process holding it after it has completed its task.

Circular Wait:
A set of processes is waiting for resources in a circular chain. For example, process P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, and P3 is waiting for a resource held by P1. This creates a cycle of dependencies that cannot be resolved.

These four conditions must hold simultaneously for a deadlock to occur in a system. If any one of these conditions is not met, deadlock cannot occur.

• Starvation

Definition: Starvation occurs when a process is perpetually denied the resources it needs to proceed with its execution due to the continual allocation of those resources to other processes. This can happen even when the system is not in a deadlock state.
Causes:

- **Priority Scheduling:** In priority-based scheduling, lower-priority processes may wait indefinitely if higher-priority processes keep arriving and consuming the CPU.
- **Resource Allocation:** Limited resources (like CPU time or memory) that are heavily contested can lead to starvation if some processes continually get scheduled before others.
- **Inefficient Resource Management:** Poorly designed algorithms for resource allocation can lead to situations where certain processes are consistently overlooked.

Effects:

- Processes that are starved cannot make progress and may cause overall system performance degradation.
- Starvation can lead to user dissatisfaction as important tasks take an unexpectedly long time to complete.

Solutions:

Aging: Gradually increasing the priority of waiting processes over time can prevent starvation by ensuring that all processes eventually get CPU time.

Fair Scheduling Algorithms: Implementing scheduling algorithms that ensure fairness among processes can help to minimize starvation.

Resource Reservation: Allocating resources in a manner that guarantees that every process will eventually receive its fair share of resources.

• Resource Allocation Graph(s)

Definition: A directed graph used to model the allocation of resources in a system, helping to detect potential deadlocks.

Components:

- **Processes (P):** Represented as circles, indicating active processes in the system.
- **Resources (R):** Represented as squares, with each square having several instances (dots inside the square) corresponding to the units of the resource.

  Edges:

  > **Request Edge:** A directed edge from a process to a resource, indicating that the process is requesting that resource (`P → R`).
  >
  > **Assignment Edge:** A directed edge from a resource to a process, indicating that the resource has been allocated to the process (`R → P`).

Usage: Used to check for **deadlocks** by analyzing cycles in the graph:

- If the graph contains no cycles, no deadlock exists.
- If a cycle exists and there is only one instance of each resource type, the system is in deadlock.

Example: If **P1** requests **R1** (`P1 → R1`), and **R1** is assigned to **P2** (`R1 → P2`), and **P2** requests **R2**, while **R2** is assigned to **P1**, the system is in a circular wait condition, leading to a potential deadlock.

Graph Examples:

- **No Deadlock:** Processes request and release resources without forming a cycle.
- **Deadlock:** A circular wait condition is present, such as `P1 → R1 → P2 → R2 → P1`, indicating a deadlock.

Key to Avoiding Deadlock:

Ensure there are no cycles in the resource allocation graph, or introduce mechanisms (like resource ordering or preemption) to handle resource requests and allocations more effectively.

6. CPU scheduling

Optimization Criteria

CPU Utilization: Maximize the usage of the CPU by keeping it as busy as possible. Ideal range is 40-90%.
Throughput: Maximize the number of processes that complete execution per time unit. Higher throughput means more processes are being completed in less time.
Turnaround Time: Minimize the time taken from the submission of a process to its completion, including all waiting and processing time.
Waiting Time: Minimize the total time a process spends waiting in the ready queue before it gets executed by the CPU.
Response Time: Minimize the time it takes from when a request is submitted until the first response is produced (not the output completion).
Fairness: Ensure that each process gets a fair share of the CPU and prevent starvation of any process.

Algorithms

First-Come, First-Served (FCFS):

Processes are executed in the order they arrive in the ready queue.

Non-preemptive: Once a process starts executing, it runs to completion.

Simple, but can cause long waiting times due to the **convoy effect** (long processes block shorter ones).

Shortest Job First (SJF):

Executes the process with the shortest burst time first.

Non-preemptive: Once a process starts, it finishes before the next one is selected.

Preemptive (Shortest Remaining Time First - SRTF): If a new process with a shorter burst arrives, the current process can be preempted.

Requires knowing the burst time in advance, which is not always feasible.

Calculating Burst Size:

The burst size can be estimated using **exponential averaging** of past bursts:
Tn+1=αtn+(1−α)TnT_{n+1} = \alpha t_n + (1 - \alpha) T_nTn+1=αtn+(1−α)Tn
where TnT_nTn is the estimated burst, tnt_ntn is the actual burst, and α\alphaα is a weighting factor.

Priority Scheduling:

Each process is assigned a priority, and the process with the highest priority is executed first.

Can be **preemptive** (if a higher-priority process arrives) or **non-preemptive**.

May cause **starvation** (low-priority processes never execute), often mitigated by **aging** (increasing priority of waiting processes over time).

Round Robin (RR):

Each process gets a small unit of CPU time (quantum) in turn.

Preemptive: If a process doesn't finish in its quantum, it is moved to the back of the queue.

Fair, but performance depends heavily on the size of the quantum. A too-small quantum leads to frequent context switching; a too-large quantum behaves like FCFS.

Multilevel Queue Scheduling:

Processes are divided into different queues based on their priority or other characteristics (e.g., foreground vs. background processes).

Each queue has its own scheduling algorithm (e.g., RR for foreground, FCFS for background).

No process can move between queues.

Multilevel Feedback Queue:

Similar to multilevel queue, but processes can move between queues based on their behavior and execution time.

If a process uses too much CPU time, it moves to a lower-priority queue.

A more dynamic and flexible approach, but more complex to implement.

- ## Metrics
  - ### Average Wait Time

    The average time that processes spend waiting in the ready queue before they get executed.

    Formula:
    Average Wait Time=Total Wait Time of all ProcessesNumber of Processes\text{Average Wait Time} = \frac{\text{Total Wait Time of all Processes}}{\text{Number of Processes}}Average Wait Time=Number of ProcessesTotal Wait Time of all Processes

    Example:
    If the wait times for 3 processes are 4 ms, 6 ms, and 8 ms, the average wait time is:
    4+6+83=6\frac{4 + 6 + 8}{3} = 634+6+8=6 ms.

Other Common Metrics (for CPU scheduling algorithms):

Throughput: The number of processes completed per unit of time.

Turnaround Time: The total time taken from the submission of a process to its completion.

CPU Utilization: The percentage of time the CPU is actively working on processes (not idle).

Response Time: The time from the submission of a request until the first response is produced.