Exam 1 Notes

Operating System Components

An operating system (OS) is a crucial component of any computer system, acting as an intermediary between hardware and application software. Its primary function is to manage the system's resources, ensuring that hardware is utilized efficiently while providing an interface for user programs to execute. The OS abstracts the complexities of hardware components, such as the CPU, memory, and I/O devices, allowing applications to perform tasks without needing to interact with the hardware directly. By providing a controlled environment, it ensures the smooth execution of multiple applications, manages hardware resources, and prevents conflicts between programs.

The key goals of an operating system include abstraction, protection, and arbitration. Abstraction simplifies hardware interactions, protection ensures security and stability between processes, and arbitration manages shared resources. The OS is responsible for critical functions such as process management, memory management, file systems, and device management. These functions enable the system to handle multiple tasks concurrently, efficiently allocate memory, and manage input/output operations, thereby ensuring the smooth operation of both hardware and software components. In modern systems, the OS also plays a role in security, ensuring that only authorized programs can access sensitive system resources.

Goals of an OS

The operating system has several important goals to ensure the efficient and secure functioning of a computer system. Abstraction provides applications with a simplified view of complex hardware, allowing developers to focus on functionality without dealing with hardware intricacies. Protection ensures that different processes are isolated from one another, preventing accidental or malicious interference. Arbitration deals with managing shared resources, ensuring fair access and preventing resource conflicts. These goals collectively enhance the stability and performance of a system, especially in multi-tasking environments.

- **Abstraction**: Simplifying complex hardware details for applications.
- **Protection**: Ensuring different processes do not interfere with each other.
- **Arbitration**: Managing access to shared resources.

Note: Connection is not a direct goal of an OS.

OS Responsibilities

The OS is responsible for managing various system resources efficiently. Memory management involves allocating and freeing memory as required by applications. Process management includes creating, scheduling, and terminating processes, as well as ensuring synchronization and communication between them. File system management is responsible for storing, retrieving, and organizing files, while device management ensures the smooth functioning of input/output devices like keyboards, printers, and disks. By handling these tasks, the OS ensures that hardware and software can work together seamlessly.

- **Memory Management**: Allocation and deallocation of memory.
- **Process Management**: Scheduling, synchronization, and creation/deletion of processes.
- **File System Management**: Handling file storage, retrieval, and organization.
- **Device Management**: Managing device input/output operations.

System Calls

System calls are the primary mechanism by which user-level applications access OS services. These calls allow user applications to request various low-level services provided by the OS, such as file access, process control, and communication with devices. System calls provide a controlled way for applications to switch from user mode to kernel mode, ensuring that the OS can mediate access to critical resources, and enforce security and stability.

System Call Parameter Passing

When a system call is made, parameters are passed from the user program to the OS in various ways. Common methods include placing a pointer to the data in a register, using the stack, or directly placing the value in a register. The system then processes these parameters in kernel mode, performing the requested action before returning control to the user program.

- **Pointer placed in a register**

- **Value placed on stack**
- **Value placed in a register**

Interrupts and Traps

Interrupts and traps are mechanisms used by the OS to respond to events that require immediate attention. An interrupt is a signal to the processor indicating an event, such as input/output completion, requiring the processor to pause its current task and handle the interrupt. Traps are similar, but they occur when a program deliberately switches to kernel mode, often to execute privileged instructions via system calls. These mechanisms are crucial in multitasking systems, allowing the OS to efficiently manage time-sensitive operations.

Definition of Interrupts

Interrupts are signals that cause the CPU to stop the current instruction and switch to a different task. These are used for:

- **Handling events**: E.g., I/O completion.
- **Switching to kernel mode**: A trap instruction switches the CPU into kernel mode for system calls.

The mode bit distinguishes user mode from kernel mode. **Mode bit = 1** indicates user mode, whereas kernel mode allows the OS to execute privileged operations.

Context Switches

A context switch occurs when the CPU changes from executing one process to another. This involves saving the state of the current process, including the program counter and register values, and loading the state of the next process. Context switching is essential for multitasking, allowing the OS to share the CPU among multiple processes, improving overall system efficiency. While context switches allow multiple processes to appear as if they are running simultaneously, they also add overhead to the system.

Definition of Context Switch

A context switch refers to the act of switching the active Process Control Block (PCB) from one process to another. This operation is critical in multitasking operating systems and is managed by the process scheduler.

Memory Management and Segmentation

Memory is divided into various segments, with different types of variables stored in different sections. Local variables are typically stored in the stack, which grows and shrinks as functions are called and return. Dynamically allocated variables are stored in the heap, allowing memory to be allocated and deallocated during runtime. Global variables reside in the data segment, and their values persist throughout the program's execution. Effective memory management is crucial for preventing issues such as memory leaks and buffer overflows.

Memory Segments

Different types of variables are stored in various memory segments:

- **Local variables**: Stored in the stack.
- **Dynamically allocated variables**: Stored in the heap.
- **Global variables**: Stored in the data segment.

Lab Assignment Overview

In the lab assignment, students are tasked with implementing unbounded data structures that can dynamically handle an unknown amount of data. This requires efficient memory management using dynamic allocation functions like 'malloc()' and 'realloc()'. The primary goal is to minimize memory usage by only allocating memory when needed and ensuring that all allocated memory is properly freed to prevent memory leaks.

Dynamic Memory Allocation

You will use the heap for dynamically allocating space as needed. The functions to implement include:

- **get_unbounded_line()**: Dynamically reads a line of text from the user.
- **get_value_table()**: Parses the line into a set of integer values and dynamically allocates space for them.

These functions use `malloc()`, `realloc()`, and `free()` for memory management, ensuring no memory leaks.

Lab Objectives and Implementation Steps

The lab aims to teach students how to handle dynamically allocated memory in a way that is both space-efficient and error-free. By following a step-by-step approach to implementing the required functions, students will gain hands-on experience in managing heap memory. This includes checking return values, reallocating memory as needed, and using debugging tools to verify the correctness of their implementation.

Objectives of the Lab

- Efficiently allocate memory for unknown quantities of data.
- Use `malloc` and `realloc` for dynamic memory management.
- Implement memory leak checks.

Step-by-Step Implementation

1. Start by creating a basic structure for the `main()` function.
2. Use stub functions to test various components.
3. Implement dynamic allocation in small increments, testing each step.

Process Management in Operating Systems

Process management is a core function of operating systems, responsible for handling the lifecycle of processes. A process is an instance of a program in execution, and the OS manages the creation, execution, and termination of processes. This also includes handling multiple processes simultaneously, managing resources for each, and ensuring efficient communication between them. The OS allocates CPU time, memory, and I/O devices to processes, which allows multiple applications to run concurrently without interference.

Effective process management allows for multitasking, wherein multiple processes share system resources without degrading system performance. It also ensures security by isolating processes, so they do not interfere with each other. To achieve this, the OS maintains a process control block (PCB) for each process, which stores important information about the process, such as its process ID (PID), state, and allocated resources. Overall, process management is essential for modern computing, enabling the system to run multiple applications smoothly and efficiently.

Process Control Block (PCB)

Every process in an OS is represented by a **Process Control Block (PCB)**, which contains information such as the process ID, program counter, CPU registers, memory limits, and open files. The PCB is used by the OS to track and manage processes, especially during context switching. The PCB plays a crucial role in maintaining the state of a process when it is not running, ensuring that it can resume seamlessly.

- **Process ID (PID)**: A unique identifier for each process.
- **Process State**: Current status, such as running, waiting, or terminated.
- **CPU Registers**: Contents of the CPU registers when the process is not executing.

Context Switching

Context switching is a mechanism that allows the CPU to switch between processes by saving the current state of a process and loading the saved state of another process. This is essential for multitasking, as it allows multiple processes to share the CPU without interfering with one another. A context switch occurs when the OS scheduler

decides to allocate CPU time to a different process, saving the current process's state in its PCB and loading the new process's state.

Definition of Context Switch

A context switch involves switching the CPU from one process to another. The state of the old process is saved in its PCB, and the state of the new process is loaded from its PCB. This allows the system to maintain multiple processes and run them efficiently by allocating CPU time fairly. The process scheduler determines when a context switch should occur.

- **CPU Registers**: Contents saved to the PCB for restoration.
- **Process Control**: Switching between processes using the PCB.

Process Creation and Termination

In most operating systems, processes are created using the `fork()` system call, which creates a new process by duplicating an existing one. The new process, called the child, inherits many of the attributes of the parent process but has its own unique process ID. After creation, the child process can either execute the same program or replace itself with a new program using the `exec()` family of system calls. Processes may terminate voluntarily using `exit()` or may be terminated by the OS due to errors or policy violations.

Process Creation

The operating system creates a new process using the `fork()` system call, which duplicates an existing process. This creates two nearly identical processes that run concurrently, one being the parent and the other the child. Once the fork is successful, the child process may replace its program with a new one using `exec()`.

- `fork()`: Creates a child process by duplicating the parent process.
- `exec()`: Replaces the process's memory space with a new program.

Process Termination

Processes terminate either voluntarily, through the `exit()` system call, or involuntarily, by the operating system due to errors or policy violations. When a process terminates, it sends an exit status to its parent process, which can retrieve this status using the `wait()` system call. The parent may then clean up resources associated with the child process.

- `exit()`: Terminates the calling process and returns an exit status.
- `wait()`: Parent process waits for the termination of a child process.

Inter-Process Communication (IPC)

Processes often need to communicate with each other during execution, especially in multitasking environments. Operating systems provide mechanisms for Inter-Process Communication (IPC) to facilitate this. IPC mechanisms include pipes, shared memory, message queues, and sockets. These allow processes to exchange data, synchronize actions, and coordinate the use of shared resources. IPC is vital for applications that involve multiple processes working on related tasks, such as client-server models.

IPC Mechanisms

There are several mechanisms provided by the OS for inter-process communication, enabling processes to share data or signals. These mechanisms allow processes to synchronize or exchange information efficiently. Some commonly used IPC techniques include:

- **Pipes**: Unidirectional communication channels between processes.
- **Shared Memory**: A region of memory accessible to multiple processes.
- **Message Queues**: Allow processes to exchange messages.

Lab Assignment Overview

In this lab, you will implement a program that demonstrates process creation and management using system calls like `fork()`, `exec()`, and `wait()`. You will create multiple processes, each running a different executable, and

the parent process will wait for each child to terminate, reporting their exit statuses. This lab provides practical experience in understanding how processes are created and how they interact with the OS during their lifecycle.

Lab Objectives

The lab requires you to create child processes using `fork()`, execute new programs using `exec()`, and synchronize the parent with the child processes using `wait()`. These objectives will help you gain a deeper understanding of how operating systems manage process execution and termination.

- **Use `fork()`**: To create child processes.
- **Use `exec()`**: To execute new programs within child processes.
- **Use `wait()`**: To synchronize the parent and child processes.

Step-by-Step Implementation

1. Create a basic program that uses `fork()` to create child processes.
2. Modify the child process to execute new programs using `exec()`.
3. Use `wait()` in the parent process to wait for each child to terminate.
4. Report the exit status of each child process in the parent process.

Multi-Threaded Computer Systems

Multi-threaded computer systems allow multiple threads to be executed within the same process. Threads, often referred to as "lightweight processes," share the same address space, code, and data of the parent process, making them more resource-efficient than separate processes. This efficiency is because threads use less memory and have faster context switching, as they do not require as many system resources as full processes. However, the use of threads introduces complexities like race conditions, synchronization issues, and the need for thread-safe code to avoid errors in shared data.

Threads are used in applications that require parallelism or concurrent processing of tasks. Multi-threading is particularly beneficial when a program has tasks that can be divided into independent units of work that run simultaneously. This can significantly reduce execution time and improve system responsiveness. However, unlike processes, threads lack isolation, meaning a bug in one thread can potentially impact the entire process, making fault tolerance a concern.

Threads vs. Processes

Threads and processes are different in terms of how they share resources. While multiple processes have separate memory spaces and offer fault isolation, threads within the same process share memory and data, allowing for easier communication. This shared memory space is ideal for tasks that require frequent data exchange. However, since threads share memory, they must be carefully synchronized to prevent race conditions.

- **Threads**: Share code, data, and heap; less resource-intensive.
- **Processes**: Have separate address spaces; offer fault isolation.

Thread Safety and Synchronization

Thread safety refers to the property of a block of code that ensures it functions correctly during concurrent execution by multiple threads. When multiple threads access shared resources like variables or data structures, thread safety is crucial to prevent race conditions. A race condition occurs when two or more threads attempt to access and modify shared data simultaneously, leading to unpredictable outcomes. Synchronization mechanisms, such as mutexes, semaphores, and condition variables, are used to control thread access to shared resources and ensure safe execution.

Definition of Thread Safety

Thread safety is achieved when a piece of code can be executed by multiple threads concurrently without causing data corruption or inconsistency. Common methods for ensuring thread safety include using synchronization primitives to serialize access to critical sections of the code.

- **Mutexes**: Locks that allow only one thread to access a section of code.
- **Semaphores**: Synchronization tools that control access to resources.
- **Condition Variables**: Used to signal state changes to other threads.

Race Conditions and Reentrant Code

A race condition is a type of concurrency issue where the outcome of a program depends on the timing of uncontrollable events such as thread scheduling. Race conditions occur when multiple threads access shared data without proper synchronization, leading to unexpected results. Reentrant code, on the other hand, is a block of code that can be interrupted in the middle of its execution and safely called again. Such code does not rely on shared or static data and can be safely used by multiple threads without synchronization.

Definition of Race Conditions

Race conditions arise when the correctness of a program depends on the relative timing of threads. Proper use of synchronization primitives is essential to prevent race conditions, ensuring that shared data is accessed in a safe and predictable manner.

- **Critical Section**: A section of code where shared resources are accessed.
- **Race Condition**: Anomalous behavior due to unexpected timing of events.

Definition of Reentrant Code

Reentrant code can be interrupted and safely executed again without adverse effects. Reentrant functions do not modify shared variables and only rely on local variables, making them inherently safe for concurrent execution.

- **Local Variables**: Used in reentrant code to avoid shared state issues.
- **No Static Data**: Reentrant code avoids using static or global variables.

Inter-Process Communication (IPC)

Inter-process communication (IPC) mechanisms allow multiple processes to exchange data and synchronize their actions. IPC is used in scenarios where separate processes need to share information, synchronize actions, or coordinate tasks. Some common IPC mechanisms include shared memory, pipes, message queues, and sockets. IPC plays a crucial role in systems that require processes to work together while maintaining isolation.

IPC Mechanisms

IPC provides the means for processes to communicate and share data safely. Some commonly used IPC techniques include:

- **Pipes**: Unidirectional communication channels.
- **Sockets**: Allow for two-way communication across machines.
- **Shared Memory**: Enables fast communication by sharing a region of memory.

Lab Assignment Overview

In this lab, you will implement a program using pipes to redirect the output of one process as the input to another. Pipes are a form of IPC that allows the flow of data between processes in a unidirectional manner. You will create a parent process that starts one or more child processes, connecting them with pipes to enable communication. This lab provides hands-on experience with process creation, file descriptor manipulation, and basic IPC using pipes.

Objectives of the Lab

The lab involves understanding how to use pipes and system calls like `fork()`, `dup2()`, and `pipe()`. By implementing these techniques, you will gain practical experience in managing data flow between processes and redirecting input/output streams.

- **Use `pipe()`**: Create a pipe for communication between processes.

- **Use `fork()`**: Create child processes that can communicate using pipes.
- **Use `dup2()`**: Redirect file descriptors to control the flow of data.

Step-by-Step Lab Implementation

This lab focuses on creating a pipeline of processes using pipes. The first step involves creating a child process that executes a command and redirects its output to a pipe. In the second step, another child process is created to read from this pipe, completing a simple pipeline of two commands connected by a pipe. In the final step, implement synchronization mechanisms to ensure proper execution and termination of each process.

Step-by-Step Implementation

1. Create a pipe using the `pipe()` system call.
2. Use `fork()` to create the first child process and redirect its `stdout` to the write end of the pipe.
3. Use `fork()` again to create the second child process and redirect its `stdin` to the read end of the pipe.
4. Close unnecessary file descriptors in both parent and child processes.
5. Implement `wait()` to synchronize the parent process with its children.

Virtual Machines and Distributed Systems

Virtual machines (VMs) are a key technology in modern computing, allowing multiple operating systems to run simultaneously on a single physical machine. VMs are created by a software layer called the hypervisor, which abstracts the hardware resources and allows each VM to act like a fully independent computer. This separation provides strong isolation between different VMs, making virtual machines useful for cloud computing, testing environments, and resource allocation. Additionally, VMs offer benefits like fault isolation, resource efficiency, and the ability to easily migrate between physical machines using techniques like live migration.

Distributed systems, on the other hand, refer to a collection of independent computers that work together as a unified system. In such systems, communication occurs across a network, and resources are shared between machines. This allows for improved fault tolerance, scalability, and performance, but introduces challenges in synchronization, message passing, and ensuring consistency between nodes. Distributed systems are commonly used in large-scale applications, cloud infrastructure, and internet services.

Virtual Machines Overview

Virtual machines provide a way to run multiple operating systems on the same hardware by using a hypervisor to manage the system's resources. The hypervisor is responsible for allocating CPU, memory, and storage to each VM, ensuring that each VM remains isolated from others. The host OS is the operating system that runs directly on the hardware, while the guest OS refers to the operating system running inside a VM.

- **Hypervisor**: Software that creates and manages virtual machines.
- **Host OS**: The underlying operating system that supports the hypervisor.
- **Guest OS**: The operating system running within a virtual machine.

Live Migration of VMs

Live migration is a feature of virtual machines that allows the transfer of a running VM from one physical host to another without interrupting its execution. This capability is crucial for load balancing, system maintenance, and minimizing downtime. During live migration, the state of the VM, including its memory, CPU state, and storage, is transferred to the new host while the VM continues to run with minimal interruption to services.

Definition of Live Migration

Live migration refers to moving a running virtual machine from one physical machine to another without stopping the VM. This feature is used for dynamic load balancing, hardware maintenance, and improving system availability by allowing administrators to move workloads without downtime.

- **Minimal Downtime**: The VM continues running during the migration process.

- **Load Balancing**: VMs can be moved to underutilized hosts to optimize resource use.

OSI Model

The OSI (Open Systems Interconnection) model is a 7-layer framework used to understand and design network communication. Each layer is responsible for a specific aspect of network communication, from physical data transmission to application-level interactions. The OSI model helps standardize communication between different systems and ensures interoperability across various networking technologies and protocols.

OSI Model Layers

The OSI model divides network communication into seven layers, each with distinct responsibilities:

- **Layer 1 (Physical)**: Manages the physical transmission of data.
- **Layer 2 (Data Link)**: Handles error detection and frames for data transmission.
- **Layer 3 (Network)**: Manages routing and forwarding of packets.
- **Layer 4 (Transport)**: Ensures end-to-end communication and error recovery.
- **Layer 5 (Session)**: Manages connections between systems.
- **Layer 6 (Presentation)**: Ensures data is in a usable format.
- **Layer 7 (Application)**: Supports application-level protocols like HTTP and FTP.

Packetization and Routing

In network communication, large messages are often broken down into smaller packets, a process known as packetization. This makes transmission more reliable, as smaller units of data are easier to manage and resend in case of errors. Once the packets are created, they are sent through the network, potentially traveling across multiple paths before being reassembled at their destination. Routing is the process of determining the best path for packets to take across a network to ensure timely and efficient delivery.

Definition of Packetization

Packetization refers to the process of breaking down large messages into smaller packets for easier transmission over a network. Each packet contains a portion of the data, as well as metadata like the source and destination addresses, to ensure it reaches its intended destination.

- **Packets**: Small units of data that are transmitted over a network.
- **Metadata**: Information that helps identify the source, destination, and sequence of the packets.

Definition of Routing

Routing is the process of determining the best path for sending packets from one machine to another across a network. Routers use routing tables and algorithms to make decisions about where to forward packets, optimizing for speed, reliability, and load balancing.

- **Routing Algorithms**: Techniques for finding the best path in a network.
- **Routing Table**: A database in routers that stores the best routes for packet delivery.

Lab Assignment Overview

In this lab, you will explore the functionality of virtual machines and their interaction with the underlying hardware. You will configure and manage multiple VMs, ensuring resource allocation and isolation using hypervisor-based virtualization. Additionally, the lab will cover distributed system concepts such as message passing, routing, and packetization, which are essential for understanding how VMs and distributed systems interact in cloud computing environments.

Lab Objectives

The goal of the lab is to understand the creation and management of virtual machines, as well as to explore the fundamentals of distributed systems, such as message passing and network communication. This lab will also provide hands-on experience with live migration of VMs and managing communication between distributed systems.

- **Create and Manage VMs**: Use hypervisor tools to create, configure, and monitor virtual machines.
- **Live Migration**: Move VMs between hosts to maintain system availability.
- **Distributed Systems**: Set up communication between distributed machines.

Step-by-Step Implementation

1. Set up a hypervisor and configure virtual machines with specific resource allocations.
2. Experiment with live migration by moving VMs between physical hosts.
3. Implement message passing between distributed systems using standard protocols.
4. Use network packetization and routing techniques to ensure efficient communication between VMs.