Intro to Operating Systems

Role of OS

An operating system (OS) is a software that acts as an intermediary between users and hardware.

It manages computer hardware resources and provides common services for computer programs.

The primary role of an operating system includes managing processes, memory, storage, and input/output devices.

Computer System Architecture

Single Processor Systems

In a single processor system, there is only one central processing unit (CPU) executing instructions.

The operating system manages the CPU's scheduling, ensuring fair access to resources for running processes.

Multiprocessor Systems

It consist of multiple CPUs working together, sharing resources.

The operating system must efficiently distribute tasks among CPUs to maximize performance and resource utilization.

Clustered Systems

Clustered systems connect multiple independent computers (nodes) to work together as a single system.

The OS coordinates communication and resource sharing among nodes to achieve high availability and scalability.

Structure

Operations:

The OS performs various operations to manage system resources and provide services to user programs.

These operations include process management, memory management, storage management, and input/output management.

Components of OS

1. **Process Management:**

Manages processes, including creation, scheduling, termination, and communication.

2. **Memory Management:**

Allocates and deallocates memory space for processes and handles memory protection.

3. **Storage Management:**

Manages storage devices, file systems, and storage allocation.

4. **I/O Management:**

Controls input/output devices, handles device communication, and manages data transfer between devices and memory.

OS Services

User and OS Interface

The OS provides a user interface for interacting with the system.

This interface can be a command-line interface (CLI) or a graphical user interface (GUI).

System Calls/API

System calls are interfaces provided by the operating system for applications to request OS services.

They act as entry points into the operating system's kernel.

Applications use system calls through an Application Programming Interface (API) provided by the OS.

Types of System Call

1. **Process Control**:

Create, terminate, and manage processes.

2. **File Management:**

   Open, close, read, write, and manipulate files.

3. **Device Management:**

   Control and communicate with input/output devices.

4. **Information Maintenance**:

   Get system information, set system parameters, etc.

5. **Communication:**

   Establish and maintain communication between processes.

System Program

These are software programs provided by the operating system to perform system-related tasks.

Examples include compilers, editors, debuggers, and system utilities.

Conclusion:

OS play a crucial role in managing computer hardware resources and providing services to user programs.

They come in different architectures, from single processor systems to clustered systems.

OS structure involves various operations and components like process management, memory management, storage management, and I/O management.

OS offer services through user interfaces, system calls, and system programs, facilitating efficient interaction between users and hardware.

Process Concept

Process Scheduling:

Process scheduling is the act of determining which process to execute next on the CPU.

Schedulers aim to optimize system performance by maximizing CPU utilization, throughput, and response time.

Scheduling algorithms include First Come, First Served (FCFS), Shortest Job Next (SJN), Round Robin (RR), Priority Scheduling, and Multilevel Queue Scheduling.

Operations on Processes:

1. **Creation:**

Processes are created through the `fork()` system call on Unix-like systems or `CreateProcess()` on Windows.

2. **Termination:**

Processes can terminate voluntarily by calling `exit()` or involuntarily due to errors or signals.

3. **Execution:**

Processes execute instructions loaded from executable files or binaries.

4. **Suspension and Resumption:**

Processes can be suspended and resumed by the operating system, typically to allow other processes to execute.

5. **Communication:**

Processes can communicate with each other using interprocess communication mechanisms.

Interprocess Communication (IPC):

IPC allows processes to share data and synchronize their actions.

IPC mechanisms include pipes, sockets, message queues, shared memory, and semaphores.

These mechanisms enable processes to cooperate, coordinate, and exchange information effectively.

Threads

Overview

A thread is a lightweight process that shares the same memory space and resources as other threads within the same process.

Threads allow for concurrent execution of tasks within a single process, improving efficiency and responsiveness.

Multicore Programming

It involves leveraging multiple CPU cores to execute tasks concurrently.

Threads can be distributed across CPU cores, allowing for parallel execution and better utilization of hardware resources.

Multithreading Models

1. **Many-to-One Model**:

Many user-level threads mapped to a single kernel thread. Simple but lacks parallelism.

2. **One-to-One Model**:

Each user-level thread corresponds to a kernel-level thread. Provides better parallelism but may have higher overhead.

3. **Many-to-Many Model:**

Multiple user-level threads mapped to a smaller or equal number of kernel threads. Balances simplicity and parallelism.

Threading Issues

1. **Thread Safety:**

Ensuring that shared data structures are accessed safely by multiple threads to avoid data corruption and race conditions.

2. **Deadlock:**

Occurs when two or more threads are blocked indefinitely, waiting for each other to release resources.

3. **Starvation:**

Happens when a thread is perpetually denied access to resources it needs to execute.

4. **Priority Inversion:**

Occurs when a low-priority thread holds a resource required by a high-priority thread, leading to inefficient resource utilization.

Conclusion

Processes are managed by the OS scheduler, and interprocess communication mechanisms enable cooperation between processes.

Threads allow for concurrent execution within a single process, improving performance on multicore systems.

However, managing threads requires careful consideration of threading models and addressing issues such as thread safety, deadlock, starvation, and priority inversion.

CPU Scheduling

It is the process of selecting which process should be executed next by the CPU from the ready queue.

It aims to maximize system performance by improving CPU utilization, throughput, response time, and fairness.

Scheduling Criteria

CPU Utilization:

Keep the CPU busy to maximize throughput.

Throughput:

Maximize the number of processes completed per unit of time.

Turnaround Time:

Minimize the time taken to execute a process from submission to completion.

Waiting Time:

Minimize the time processes spend waiting in the ready queue.

Response Time:

Minimize the time taken for the system to respond to user interactions.

Scheduling Algorithms

First In, First Out (FIFO):

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

Simple and easy to implement.

It may result in poor turnaround time for long-running processes (convoy effect).

Shortest Job First (SJF)

Selects the process with the shortest burst time next.

Minimizes average waiting time and turnaround time.

Requires knowledge of the burst time of each process, which may not be available.

Priority Scheduling:

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

Can be preemptive (priority can change during execution) or non-preemptive (priority fixed throughout execution).

May suffer from starvation (low priority processes may never execute).

Round-Robin Scheduling:

Each process is given a fixed time quantum to execute on the CPU.

If the process does not complete within its time quantum, it is preempted and placed at the end of the ready queue.

Provides fair scheduling and prevents starvation.

It may result in higher average waiting time for CPU-bound processes.

Multilevel Queue Scheduling:

Processes are categorized into multiple queues based on priority.

Each queue may use a different scheduling algorithm.

Processes move between queues based on their priority or characteristics.

Process Synchronization

It ensures that concurrent processes or threads cooperate and coordinate their actions to avoid undesirable outcomes.

Common issues include race conditions, deadlock, and starvation.

The Critical-Section Problem:

Critical sections are segments of code where shared resources are accessed.

It involves ensuring that only one process can execute its critical section at a time to avoid race conditions.

Two-Process Solution:

Solutions to the critical-section problem include using mutual exclusion mechanisms such as locks, semaphores, or atomic operations.

Multiple-Process Solution:

Extends mutual exclusion mechanisms to handle multiple processes, ensuring that only one process accesses the critical section at a time while allowing other processes to execute non-critical sections concurrently.

Synchronization Hardware:

Some hardware architectures provide atomic instructions or special CPU instructions (like test-and-set) to support synchronization primitives.

Semaphores

Semaphores are integer variables used for controlling access to shared resources.

They can be used to implement mutual exclusion, synchronization, and signalling between processes.

Classic Problems of Synchronization:

1. **Producer-Consumer Problem:**

Coordinating the production and consumption of data by multiple threads or processes.

2. **Readers-Writers Problem:**

Managing concurrent access to shared data by multiple readers and writers.

3. **Dining Philosophers Problem:**

Coordinating the resource allocation among multiple processes to avoid deadlock.

Critical Regions and Monitors

Critical Regions:

Sections of code where shared resources are accessed and need to be protected against concurrent access.

Monitors:

High-level synchronization constructs that encapsulate shared data and operations on it, ensuring mutual exclusion and data integrity.

These concepts and mechanisms are essential for efficient and reliable concurrent programming, ensuring correct and synchronized access to shared resources in multi-threaded or multi-process environments.

Deadlock

Deadlock occurs when two or more processes are unable to proceed because each is waiting for the other to release a resource.

Necessary Conditions for Deadlock:

1. **Mutual Exclusion:**

   At least one resource must be held in a non-sharable mode.

2. **Hold and Wait:**

   Processes hold resources while waiting for others.

3. **No Preemption:**

   Resources cannot be forcibly taken from processes.

4. **Circular Wait:**

   A set of processes waits for resources held by others in a circular chain.

Methods for Handling Deadlocks:

1. **Deadlock Prevention:**

   Ensures that at least one of the necessary conditions for deadlock cannot hold.

2. **Deadlock Avoidance:**

Dynamically decides whether granting a resource request will result in deadlock.

3. **Deadlock Detection:**

Periodically checks the system for deadlocks and takes corrective actions if one is detected.

4. **Recovery from Deadlocks:**

Terminate processes, rollback transactions, or preempt resources to resolve deadlocks.

Deadlock Prevention

Techniques include:

Mutual Exclusion:

Ensure that at least one resource type is non-shareable.

Hold and Wait:

Require processes to request and hold all necessary resources before execution.

No Preemption:

Preempt resources if necessary to avoid deadlock.

Circular Wait:

Impose a total ordering of all resource types and require processes to request resources in increasing order.

Deadlock Avoidance

- Banker's Algorithm is an example of deadlock avoidance.

- Processes must declare their maximum resource requirements upfront.

- Resource allocation is granted only if it satisfies the system's safety criteria and avoids deadlock.

Deadlock Detection

Utilizes resource allocation graphs or wait-for graphs to detect cycles indicating deadlock.

When a deadlock is detected, the system may take actions such as aborting processes or preempting resources to resolve it.

Recovery From Deadlocks

- Terminate processes involved in the deadlock.

- Rollback transactions to a consistent state before the deadlock occurred.

- Preempt resources from processes to break the deadlock.

Memory Management

It is the process of managing computer memory resources.

It involves allocating memory to processes, managing memory hierarchies, and ensuring efficient use of available memory.

Swapping:

It involves moving entire processes between main memory and secondary storage (disk) to free up space for other processes.

Contiguous Memory Allocation:

Memory is allocated to processes contiguously in physical memory.

Techniques include single-partition allocation and multiple-partition allocation (fixed or variable partitioning).

Segmentation:

Divides memory into logical segments based on the process's memory requirements.

Each segment can grow or shrink dynamically as needed.

Paging:

Divides memory into fixed-size blocks called pages.

Processes are divided into equal-sized blocks called frames.

Provides flexibility in memory allocation and efficient use of memory space.

Segmentation with Paging:

Combines the segmentation and paging techniques to provide the benefits of both.

Processes are divided into logical segments, and each segment is further divided into fixed-size pages.

Virtual Memory

It allows processes to use more memory than physically available by using disk storage as an extension of RAM.

It provides a uniform memory abstraction to processes, hiding the complexities of physical memory management.

Demand Paging:

Only the necessary pages of a process are loaded into memory at runtime.

Reduces initial memory overhead and improves memory utilization.

Page Replacement Algorithms:

Determines which pages to replace when a new page needs to be brought into memory.

Examples: FIFO, LRU (Least Recently Used), Optimal, and Clock algorithms.

Allocation of Frames:

Allocates physical memory frames to processes based on their memory requirements and system policies.

Uses algorithms such as proportional allocation or fixed allocation.

Thrashing:

Occurs when the system spends a significant amount of time swapping pages between main memory and disk due to excessive paging activity.

Causes severe performance degradation as CPU time is spent swapping rather than executing processes.

Mass Storage Structure:

Mass storage devices (such as hard drives) provide non-volatile storage for large amounts of data.

Disk Structure:

Disk storage is organized into tracks, sectors, and cylinders.

Data is stored and retrieved using read/write heads.

Disk Attachment:

Disk drives can be attached to the system via various interfaces such as SATA, SCSI, or NVMe.

Disk Scheduling:

Disk scheduling algorithms determine the order in which disk I/O requests are serviced.

Examples: FCFS (First-Come, First-Served), SSTF (Shortest Seek Time First), SCAN, and C-SCAN.

Disk Management:

Disk management involves partitioning disks into logical volumes, formatting, and maintaining file systems for data organization and storage.

Installation, Configuration & Customizations of Linux

Installation:

Linux distributions can be installed from installation media (DVD, USB) or downloaded as disk images for installation.

Follow the distribution-specific installation instructions to complete the installation process.

Configuration:

Linux configuration involves setting up system preferences, user accounts, network settings, and system services.

Configuration files are located in `/etc` directory, and tools like `vim` or `nano` can be used to edit them.

Customizations:

Users can customize their Linux environment by modifying shell configurations, desktop environments, themes, and installing additional software.

Package managers like `apt` (Debian/Ubuntu) or `yum` (Fedora/CentOS) can be used to install and manage software packages.

Introduction to GCC Compiler

GCC (GNU Compiler Collection) is a free and open-source compiler for various programming languages, including C, C++, and Objective-C.

It provides a set of command-line tools for compiling and linking programs.

Compilation of Program:

- To compile a C program with GCC, use the following command:

  gcc -o output_file source_file.c

- This command compiles the source file and generates an executable binary named `output_file`.

Execution of Program

- To execute the compiled program, use:

  ./output_file

Timestamping

GCC can include a timestamp in the compiled binary by passing the `-D` option:

gcc -o output_file source_file.c -D"BUILD_TIME=\"`date`\""

Automating Execution using Makefile:

Makefile automates the compilation and execution process by defining rules and dependencies.

Here's a simple Makefile:

```
Make program: source_file.c
    gcc -o program source_file.c

run: program
    ./program
```

Process Concepts Implementation in C

Printing Process ID:

Use the `getpid()` function to get the process ID in C.

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Process ID: %d\n", getpid());
    return 0;
}
```

Executing Linux Command as Subprocess:

Use the `system()` function to execute Linux commands as subprocesses.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    system("ls -l");
    return 0;
}
```

Creating and Executing Processes using `fork()` and `exec()` System Calls:

Use `fork()` to create a new process and `exec()` to replace the current process with a new program.

Example:

```
#include <stdio.h>
#include <unistd.h>
```

```
int main() {
    int pid = fork();
    if (pid == 0) {
        // Child process
        execl("/bin/ls", "ls", "-l", NULL);
    } else {
        // Parent process
        wait(NULL);
    }
    return 0;
}
```

Implementation of Scheduling Algorithms in C

First-Come, First-Served (FCFS):

Implements FCFS scheduling algorithm in C.

Processes are executed in the order they arrive.

Example:

```
// Implement FCFS scheduling algorithm
```

Shortest Job First (SJF):

Implements SJF scheduling algorithm in C.

Processes with the shortest burst time are executed first.

Example:

```
// Implement SJF scheduling algorithm
```

Priority Scheduling:

Implements priority scheduling algorithm in C.

Processes with higher priority are executed first.

Example:

```
// Implement priority scheduling algorithm
```

Round-Robin (RR) Scheduling:

Implements RR scheduling algorithm in C.

Each process is executed for a fixed time quantum before being preempted.

Example:

```
// Implement RR scheduling algorithm
```

Implementation of Basic and User Status Commands:

Basic Commands:

su : Switches the current user to another user.

sudo : Executes commands with superuser (root) privileges.

man : Displays the manual pages for commands.

help : Displays help information for shell built-in commands.

history : Displays the command history.

who : Shows who is logged into the system.

whoami : Prints the username of the current user.

id : Displays user and group information.

uname : Shows system information such as kernel version and system architecture.

uptime : Displays system uptime.

free : Shows memory usage and available memory.

tty : Prints the file name of the terminal connected to the standard input.

cal : Displays a calendar for a specified month or the current month.

date : Prints or sets the system date and time.

hostname : Prints or sets the system's hostname.

reboot : Restarts the system.

clear : Clears the terminal screen.

Implementation in C:

These commands can be implemented in C by using system calls or library functions.

For example, `system()` function can be used to execute shell commands, and standard C library functions like `gettimeofday()` can be used for getting system time.

Implementation of Deadlock in C:

Deadlock using Shared Variables:

Deadlock can be implemented in C by using shared resources and locks.

Example:

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void* thread1(void* arg) {
    pthread_mutex_lock(&mutex1);
    printf("Thread 1 acquired mutex1\n");
    sleep(1);
    printf("Thread 1 waiting for mutex2\n");
    pthread_mutex_lock(&mutex2);
```

```
    printf("Thread 1 acquired mutex2\n");
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_join(t1, NULL);
    return 0;
}
```

File System:

A file system is a method of storing and organizing computer files and the data they contain.

It provides a structured way to store, retrieve, and manage data on storage devices such as hard drives and SSDs.

Architecture:

File systems consist of:

File Allocation Table (FAT):

Tracks the allocation status of disk space.

Inodes:

Data structures that store metadata about files, such as permissions, ownership, and location on disk.

Directory Structure:

Hierarchical organization of files into directories or folders.

File Access Methods:

Techniques for accessing and manipulating files, such as sequential access, random access, or direct access.

File Types

- File systems support different types of files, including:

Regular Files:

Contain user data and can be text, binary, or executable files.

Directories:

Containers for organizing and managing files.

Special Files:

Represent system resources such as devices, pipes, and sockets.

Symbolic Links:

Pointers to other files or directories.

Understanding file system concepts and architecture is essential for efficient data storage, retrieval, and management in operating systems and applications.

Implementation of Commands for Creating and Manipulating Files:

cat:

Functionality:

Concatenates and displays the content of one or more files.

Implementation:

Use file I/O functions to read and print the contents of each file.

cp

Functionality:

Copies files or directories.

Implementation:

Open source and destination files, read from source and write to destination.

mv

Functionality:

Moves or renames files or directories.

Implementation:

Use system calls to rename or move files.

rm

Functionality:

Removes files or directories.

Implementation:

Use system calls to delete files or directories.

ls

Functionality:

Lists directory contents.

Implementation:

Use system calls to read directory entries and print them.

touch

Functionality:

Creates an empty file or updates file timestamps.

Implementation:

Use system calls to create files or update timestamps.

which, whereis, whatis:

which:

Finds the location of executable files in the user's PATH environment variable.

whereis:

Locates the binary, source, and manual page files for a command.

whatis:

Displays a brief description of a command.

These commands can be implemented by searching through the PATH environment variable and filesystem directories for executable files and manual pages.

Implementation of Directory-Oriented Commands

1. **cd**

Functionality:

Changes the current working directory.

Implementation:

Use `chdir()` system call to change the current working directory.

2**. pwd**

Functionality:

Prints the current working directory.

Implementation:

Use `getcwd()` system call to get the current working directory.

3. **mkdir**

Functionality:

Creates a new directory.

Implementation:

Use `mkdir()` system call to create a directory.

4. **rmdir**

Functionality:

Removes an empty directory.

Implementation:

Use `rmdir()` system call to remove a directory.

File System Commands:

Comparing Files using diff, cmp, comm.

diff :

Compares the contents of two files line by line and displays the differences.

cmp :

Compares two files byte by byte and displays the first differing bytes and their offsets.

comm :

Compares two sorted files line by line and displays lines unique to each file and common lines.

These commands can be implemented by reading files line by line, comparing their contents, and printing the differences or common lines.

Understanding and implementing these file manipulation and directory-oriented commands provides essential skills for working with files and directories in a Unix-like environment.