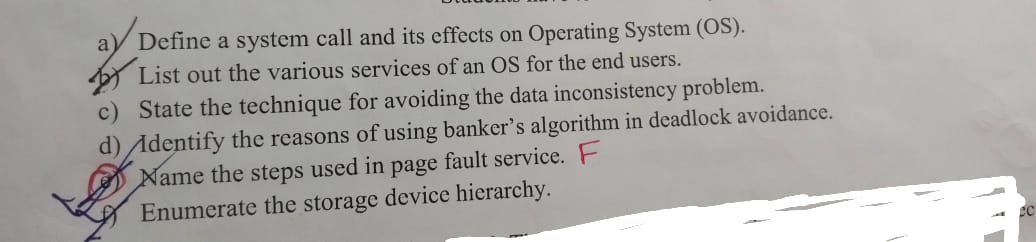
Ques:2022 All

(1)



1. A **system**
2. **call** is a request made by a user program to the operating system (OS) for services such as file manipulation, process control, or hardware access. It acts as an interface between the user space and kernel space of an operating system.

**Effect on the Operating System:**

* When a system call is made, the control is transferred from user mode to kernel mode, allowing the OS to execute privileged operations.
* It can alter the state of the system, such as creating processes, accessing I/O devices, or managing memory, thus enabling the OS to perform essential tasks for the application.

In essence, system calls provide a controlled interface for programs to interact with the operating system.

**b) List out the various services of an OS for the end users:**

The operating system (OS) provides several services to end users, including:

1. **Process Management**: Creating, scheduling, and terminating processes.
2. **Memory Management**: Allocating and managing memory resources.
3. **File Management**: Storing, retrieving, and organizing files.
4. **Device Management**: Managing input and output devices like printers and disk drives.
5. **Security and Access Control**: Ensuring proper access control, authentication, and authorization.
6. **User Interface**: Providing a command-line interface (CLI) or graphical user interface (GUI) for interaction.

c) State the technique for avoiding the data inconsistency problem?

To avoid data inconsistency problems, the technique involves using synchronization mechanisms to ensure orderly execution of processes when accessing shared data. Some of the common solutions include:

1. **Mutex Locks**: These locks prevent multiple processes from entering critical sections simultaneously by requiring a process to acquire the lock before entering the critical section and release it afterward.
2. **Semaphores**: Semaphores manage access to shared resources by allowing processes to signal (increment) and wait (decrement) on an integer value, ensuring proper synchronization between processes.
3. **Atomic Instructions**: Instructions like test\_and\_set and compare\_and\_swap ensure that certain operations on shared variables are completed atomically, preventing race conditions and ensuring consistencyes are essential for ensuring that multiple processes or threads do not interfere with each other while accessing shared data, thus avoiding data inconsistencies.

**d) Identify the reasons for using the Banker's algorithm in deadlock avoidance:**

The Banker's algorithm is used for deadlock avoidance because it:

1. **Ensures Resource Allocation Safety**: It checks whether resource allocation will leave the system in a safe state, preventing deadlock.
2. **Prevents Circular Wait**: It avoids situations where processes are waiting for each other in a cycle.
3. **Maximizes Resource Utilization**: It allocates resources only when it is guaranteed that all processes can eventually complete.

**e) Name the steps used in page fault service:**

The steps involved in handling a page fault are:

1. **Page Fault Detection**: The OS detects that the requested page is not in memory.
2. **Select a Victim Page**: If memory is full, select a page to be swapped out.
3. **Load the Page from Disk**: Fetch the required page from secondary storage (disk) into memory.
4. **Update Page Tables**: Modify page tables to reflect the new page in memory.
5. **Restart the Process**: Resume the execution of the process that caused the page fault.

**f) Enumerate the storage device hierarchy:**

The storage device hierarchy, from the fastest and smallest to the slowest and largest, is as follows:

1. **Registers** (CPU cache)
2. **Cache Memory**
3. **Main Memory (RAM)**
4. **Solid-State Drives (SSD)**
5. **Hard Disk Drives (HDD)**
6. **Optical Disks (CDs, DVDs)**
7. **Magnetic Tapes**

(2)

(a) compare the working principle of Mutex locks with that Semaphore.

**Comparison of Mutex Locks and Semaphores:**

| **Aspect** | **Mutex Locks** | **Semaphores** |
| --- | --- | --- |
| **Definition** | A **Mutex (Mutual Exclusion)** is a locking mechanism that ensures only one thread or process can access a critical section of code at a time. | A **Semaphore** is a synchronization primitive that uses a counter to control access to shared resources by multiple processes or threads. |
| **Ownership** | A mutex lock is **owned by the thread** that locks it, meaning only the thread that locked the mutex can unlock it. | A semaphore is **not owned by any specific process**. Any process can signal or wait on the semaphore, irrespective of which process previously changed its value. |
| **Value Type** | Mutexes are **binary**, having only two states: **locked (1)** or **unlocked (0)**. | Semaphores are **counting variables**, with values ranging from 0 to the maximum count, which defines the number of processes that can concurrently access a resource. |
| **Usage** | Used to protect critical sections of code that should be accessed by only **one thread at a time**. | Used to control access to a **pool of resources**, ensuring that a defined number of processes can access the resource at once. |
| **Blocking** | A thread will **block** if it attempts to lock a mutex that is already locked by another thread. | A process will **block** if it attempts to decrease a semaphore's value when it is 0 (indicating no available resources). |
| **Deadlock Prevention** | **Deadlock** can occur if multiple threads are using mutexes in an improper order, causing circular waits. | Semaphores can help manage resources, but improper handling can also lead to **deadlocks**, especially when combined with conditions like low semaphore values. |
| **Types** | Only **binary** mutex locks exist, which can be in two states: locked or unlocked. | There are two main types of semaphores: **binary** (0 or 1) and **counting semaphores**, which can have values greater than 1. |
| **Complexity** | Mutexes are **simpler** and used for exclusive access to resources. | Semaphores are **more flexible** and can manage multiple resources, but they require careful handling to avoid problems like race conditions and deadlocks. |

**Summary:**

* **Mutex Locks** are used for **exclusive access** to a critical section, ensuring that only one thread can access the critical section at a time. They provide a simpler and more focused synchronization mechanism, but the **thread ownership** feature adds complexity in certain situations.
* **Semaphores**, on the other hand, are more **flexible** and can control access to multiple resources. They use a counter to manage the number of available resources and allow a greater number of processes to access shared resources concurrently. However, they are more complex and require careful handling to avoid deadlocks and race conditions.

In summary, **mutex locks** are ideal for **mutual exclusion** in a single-threaded context, while **semaphores** offer broader control over concurrent processes, managing resource limits effectively.

(b) illustrate the functions of valid invalid bit in page table?

The **valid-invalid bit** in a page table is a simple but important mechanism used in virtual memory systems to indicate whether a page is currently in physical memory or if it is invalid or unavailable. Here’s how it functions:

1. **Valid Bit = 1**:
   * When the bit is set to **1** (valid), it indicates that the page is currently in physical memory.
   * The page table entry contains the correct frame number or memory location, allowing the CPU to access the data directly without causing a page fault.
   * Accessing a valid page results in a fast memory access, as the required data is already loaded in memory.
2. **Invalid Bit = 0**:
   * When the bit is set to **0** (invalid), it means that the page is not in physical memory or is not available to the process.
   * Attempting to access an invalid page will trigger a **page fault**, prompting the operating system to handle it by locating the page on disk (if it exists) and loading it into memory.
   * The invalid bit also helps with protection, as it prevents access to memory addresses that don’t belong to the process (e.g., unallocated memory or protected regions).

**Functions of the Valid-Invalid Bit:**

| * + **Function** | * + **Explanation** |
| --- | --- |
| * + **Memory Access Validation** | * + Ensures that a process only accesses pages that are loaded in physical memory. |
| * + **Page Fault Handling** | * + Triggers a **page fault** if a process tries to access a page marked as invalid, allowing the OS to load the page into memory. |
| * + **Protection Against Errors** | * + Prevents accidental or unauthorized access to memory areas that are not mapped to the process. |
| * + **Efficient Memory Usage** | * + Helps the operating system track which pages are loaded in memory and which are not, optimizing memory management. |

**Key Benefits**

1. **Efficient Page Management**: Helps track which pages need to be loaded.
2. **Error Prevention**: Protects against accessing unallocated memory.
3. **Support for Virtual Memory**: Enables seamless management of pages between physical memory and secondary storage.

(3)

(d) Develop a multithreading program that outputs prime numbers.This program should work as follows: THe user will run the program amd will enter a number on the command line.THe program will then create a separate thread that outputs all the prime number less then or equal to the number entered by the user

#include <iostream>

#include <thread>

#include <cmath>

using namespace std;

// Function to check if a number is prime

bool isPrime(int n) {

if (n <= 1)

return false;

for (int i = 2; i <= sqrt(n); ++i) {

if (n % i == 0)

return false;

}

return true;

}

// Function to find and print all prime numbers up to a given number

void findPrimes(int limit) {

cout << "Prime numbers less than or equal to " << limit << ":" << endl;

for (int num = 2; num <= limit; ++num) {

if (isPrime(num)) {

cout << num << " ";

}

}

cout << endl;

}

// Main function

int main() {

int number;

// Get user input

cout << "Enter a number: ";

cin >> number;

if (number < 2) {

cout << "There are no prime numbers less than 2." << endl;

return 0;

}

// Create a thread to find and print prime numbers

thread primeThread(findPrimes, number);

// Wait for the thread to complete

primeThread.join();

return 0;

}

**Explanation:**

1. **Prime Checking Function (isPrime)**:
   * The function checks whether a number is divisible by any number from 2 to the square root of the number.
   * If divisible, it returns false (not prime); otherwise, it returns true (prime).
2. **Prime Finding Function (findPrimes)**:
   * Loops through numbers from 2 to the limit entered by the user.
   * Checks each number using the isPrime function and prints it if it is prime.
3. **Multithreading**:
   * The program uses the std::thread library to create a thread for executing the findPrimes function.
   * The user-provided number is passed as an argument to the thread.
4. **Thread Execution**:
   * primeThread.join() ensures that the main thread waits for the completion of the prime-finding thread before exiting.

Top of Form

Bottom of Form