NEPAL COLLEGE OF INFORMATION TECHNOLOGY

Balkumari, Lalitpur

Affiliated to Pokhara University

ASSIGNMENT FOR OPERATING SYSTEM

ASSIGNMENT 2

Submitted by:

Name: Arpan Adhikari

Roll No.: 231309

BE-CE (3rd SEM)

Submitted to:

Amit Shrivastava

Name : Arpan Adhikari
Roll no. 231309

Assignment 2

Q1) List five services provided by an operating system that are designed to make it more convenient for users to use the computer system. In what cases it would be impossible for user-level programs to provide these services ? Explain.

→

Here are five services provided by OS, with explaination why userlevel programs can't independently provide these services.

1. User Interface Management
The OS provides a standardized interface for users to interact with the systems. This includes visual components like windows, menus, or text-based input.

Without Os level support for graphics rendering or terminal control, User level programs can't consistently access or modify hardware components like the display or keyboard.

2. File system Management
The OS handles file storage, organization, retrieval, and access permissions on storage device.

User-level programs can't directly interact with the raw hardware or storage medium due to the absence of standardized methods for accessing sectors on disks or enforcing security.

3. Process scheduling and Management

The OS manages the execution of multiple programs by allocating CPU time and resources. It also handles process creation, termination and, multitasking.

Only the OS has direct control over the CPU and memory. User level programs can't interrupt the CPU or enforce multitasking across independent applications.

4. Memory Management

The OS allocates, deallocates and protects memory used by various programs. This includes virtual memory management.

Programs can't directly manage memory without risking conflicts or corruption since they lack access to hardware level memory protection mechanisms.

5. Device Management and Drivers

The OS facilitates interaction with hardware devices like printers, network cards, and hard drives through standardized drivers and APIs.

Direct interaction with hardware requires privileged instructions and knowledge of specific device interfaces, which user level programs aren't allowed to execute due to security restrictions.

Q2) What are the advantages and disadvantages of using the same system call interface for manipulating both files and devices?

→

Advantages:

1. Consistency
A unified interface (open, read, write etc.) simplifies development and usage for both files and devices.

2. Code Reusability
similar handling for files and devices allows code reuse across different applications.

3. Portability
Abstracting hardware differences makes program easier to port.

4. Simplified OS Design
Reduces the need for multiple specialized APIs

5. Hardware Abstraction
Hides device specific complexities from developers.

Disadvantages:

1. Loss of Specificity
Device-specific operations may not fit ~~like~~ into a file like interface.

2. Performance Overhead
Abstraction can slow down device operations requiring high speed access.

3. Complex Error Handling
Different error semantics for files and devices increase complexity.

4. Limited Advanced Features
Advanced device functionalities may need additional APIs.

5. Misuse Risk
Developers might incorrectly treat devices like files, leading to inefficiencies.

Q3) Describe the actions taken by a kernel to context switch between processes.

→ A context switch occurs when the kernel switches the CPU from one process to another. The steps are as follows:

1. Save the current Process's state
The kernel pauses the running process and saves its current state, which includes:
- The values in the CPU's registers.
- Information about its memory setup, like page tables.
- Other details stored in the process control block (PCB)

2. Update the Process Status
The kernel changes the status of the current process to indicate it is no longer running. For example, it might move the process to a "Ready" or "Waiting" state, depending on whether it's waiting for something or ready to resume later.

3. Choose the Next Process to Run
The kernel uses the CPU scheduler to decide which process should run next. This decision is based on scheduling policies like priority, round-robin, or first-come-first-served.

4. Restore the New Process's State
The kernel loads the saved state of the process chosen to run. It restores the CPU registers, sets the program counter to the process's last saved instruction, and configures its memory settings.

5. Switch control to the New Process
The kernel gives control of the CPU to the new process by starting its execution where it left off. If necessary, the CPU mode is switched from kernel mode to user mode.

Q4) Provide two programming examples in which multithreading doesnot provide better performance than a single threaded solution.

→

Here are two of the programming examples,

1. CPU-Intensive Tasks on a single core Processor
When a program is performing tasks that require a lot of computation, like finding prime numbers, simulating physics, or processing large datasets, these tasks are called CPU-bound because they rely entirely on the CPU for progress.

On a single core processor, only one thread can run at a time, no matter how many threads are created. If multiple threads are used, the processor has to switch between them, which adds overhead for managing the threads. This content switching reduces the performance compared to single threaded approach, where the

CPU can fully on completing one task at a time.
E.g.: calculating a large list of prime numbers or rendering complex graphics on a single-core machine.

2. Tasks that require extensive Synchronization

If multiple threads in a program need to frequently coordinate with each other or share data, the performance can suffer due to the time spent managing this coordination. To ensure threads donot corrupt shared data, synchronization mechanism like locks, semaphores or condition variables are used.
These mechanisms often make threads wait until they are allowed to access the shared resource. If this waiting happens too often, the threads end up spending more time waiting for locks to release or coordinating with each other than doing actual work. This makes the program slower than a single-threaded version, where no such synchronization is needed.

E.g. A program with multiple threads updating the same database or shared file.

Q5) Describe the actions taken by a thread library to content switch between user level threads.

→

When a thread library content switches between user level threads, it performs the following action:

1. Save the current Thread's state
The library saves the CPU registers, program counter, stack

pointer and thread specific data in thread control block (TCB).

2. Select the Next Thread
It picks the next thread to run based on a scheduling algorithm like round robin or priority based.

3. Restore the Next Thread's state
The saved state of the chosen thread (registers, program counter, stack pointer) is loaded.

4. Switch Execution
The library jumps to the program counter of the selected thread; resuming its execution.

This is faster than kernel-level switching since it avoids system calls, but it doesn't provide true parallelism on multi core systems.

Q6) Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single processor system?

→

A multithreaded solution using multiple kernel threads can provide better performance than a single threaded solution on a single processor system in following situations:

1. I/O Bound task
When a program spends significant time waiting for I/O operations (e.g. reading from disk or waiting network responses), multithreading allows other threads to execute while one thread is blocked. This increases overall throughput.

E.g. A webserver handling multiple client requests where some threads process data while others wait for network responses.

2. Overlapping Computation and I/O:
Programs that alternate betⁿ computation and I/O benefit from multithreading, as one thread can handle computation while another manages I/O. This minimizes CPU idle time.
E.g. A file processing application that reads data chunks from disk while simultaneously processing prev. read chunks.

3. Interactive Applications
Multithreading helps keep interactive applications responsive by separating user interaction from background tasks. One thread can handle user input while others manage computations or data processing.

4. Blocking system calls
In single threaded programs, blocking system calls halt the entire process. While multithreading, other threads can continue execution, improving performance.

Q7) Can a multithreaded solution using multiple user level threads achieve better performance on a multiprocessor system than on a single processor system?

→

Yes, a multithreaded solution with user level threads can achieve better performance on a multiprocessor system because of the following reasons.

1. Parallel Execution
Multiple threads can run on different processors simultaneously, allowing true parallelism, unlike single processor systems where threads must take turns.

2. Reduced Blocking
If one thread is blocked, other threads can still execute on different processors, improving efficiency.

3. Better scalability
Multiprocessor systems handle more threads effectively, making them ideal for multi-threaded applications.

However, since the kernel is unaware of user-level threads, managing them efficiently across processors can be challenging.

Q8) Discuss how the following pairs of scheduling criteria conflict in certain settings.

a) CPU utilization and response time

→

CPU utilization : This measures how much time the CPU spends doing productive work. Maximizing CPU utilization means ensuring the CPU is always busy and not idle, which improves overall system performance.

Response time : This measures how quickly the system begins processing a request after it has been submitted.

Conflict :

If we focus too much on keeping CPU busy, requests might have to wait longer before they even start. This increases response

time and negatively affects user experience. For e.g. In FCFS scheduling approach, a long running task might take full control of CPU, delaying shorter tasks that require quick responses. While this maximizes CPU utilization, it hurts response time.

b) Average Turnaround Time v/s Maximum waiting Time

Avg TAT : This is the average time it takes for tasks to complete, starting from when they are submitted.
Max^m waiting Time : This is the longest time any single task waits before it starts execution.

Conflict:
Reducing average turnaround time often involves prioritizing shorter tasks using algorithm like SJN. While this completes more tasks quickly, longer tasks may wait for extended periods, increasing maximum waiting time and risking starvation. Preemptive scheduling, like RR, can prevent starvation by periodically allowing longer tasks to run, but frequent switching may increase the time it takes for smaller tasks to complete, slightly raising avg. TAT.

c) I/O device utilization and CPU utilization

I/O device utilization : This measures how effectively input/output devices are being used, aiming to keep them busy and avoid idle time.
CPU utilization : This measures how much the CPU is being used for executing tasks, ensuring it remains productive.

conflict:
Prioritizing I/O-bound tasks keeps I/O devices busy but may leave the CPU idle while waiting for I/O operations. Focusing on CPU-bound tasks keep the CPU busy but can leave I/O devices idle. Balancing both is crucial to avoid underutilization of either resource.

Q9) What is the meaning of term busy waiting? What other kinds of waiting are there in operating system? Can busy waiting be avoided altogether? Explain your answer.

→

Busy waiting occurs when a process simultaneously checks for a condition to become true without relinquishing the CPU. This results in the CPU being occupied unnecessarily, even though the process is not making progress.

Other types of waiting:

1. Blocked (or sleeping) waiting
The process is placed in a waiting queue and goes into a non-active state until the desired condition is met.

2. Spinning
Similar to busy waiting, but typically used in multicore systems where a thread spins on one processor while waiting for other processor to release a resource.

Avoiding busy waiting

Yes, busy waiting can often be avoided by using more efficient synchronization mechanisms.

1. Interrupts
The operating system uses interrupts to notify a process when the condition it is waiting for is met.

2. Blocking Synchronization
Mechanisms like semaphores or condition variables put the process to sleep instead of looping.

Q10) Consider the deadlock situation that could occur in the dining-philosophers problems when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.

→

In Dining Philosophers Problem, deadlock can occur when philosophers are attempting to eat by acquiring two chopsticks, one at a time.

How ~~deadlock~~ necessary conditions are deadlock are met in the dining philosopher problem:

1. Mutual Exclusion
Each chopstick can only be used by one philosopher at a time, making it a mutual exclusion.

2. Hold and wait
Philosophers hold onto one chopstick while waiting for the second chopstick. If they cannot obtain second, they still hold the first one and wait.

3. No Preemption

chopsticks cannot be forcibly taken away from a philosopher. once a philosopher holds a chopstick, it remains with them until they finish eating and release it voluntarily.

4. Circular Wait

A circular wait happens if philosophers form a cycle. e.g. Philosopher A waits for chopstick 2, philosopher B for chopstick 3, and so on, until philosopher N waits for chopstick 1.

Avoiding Deadlock by Eliminating One condition

1. Eliminate Mutual Exclusion

This would mean allowing multiple philosophers to use the same chopstick simultaneously. This is impractical, as chopsticks must be exclusive for proper eating.

2. Eliminate Hold and wait

Philosophers could be required to pick up both chopsticks at once. This eliminates hold and wait, as philosophers won't hold one chopstick while waiting for the other. However, this approach might result in more philosophers waiting if both chopsticks are unavailable.

3. Eliminate No Preemption

Forcibly taking chopsticks from philosophers disrupts fairness and is not a practical solution.

4. Eliminate Circular Wait

One effective solution is to prevent circular waiting by imposing an ordering on how philosophers pick up chopsticks. For e.g.,

philosophers could be required to always pick up the left chopstick first, and only then the right. This way, no cycle can form, as each philosopher is only dependent on the next philosopher in a chain.

Q11) Under what circumstances would a user be better off using a timesharing system rather than a PC or single user workstation?

→

The circumstances are:

1. Cost Efficiency
Timesharing systems allow multiple users to share a single, powerful computer system. This can significantly reduce costs, as users donot need to purchase individual PCs or workstations.

2. Resource sharing
If users require access to expensive hardware, software, or high computational power that is too costly or impractical to provide on individual machines, a timesharing system is more efficient.

3. Collaboration Among Multiple Users
Timesharing systems support multiple users accessing the same system simultaneously. This facilitates collaboration, shared access to files, and joint resource usage.

4. Maintenance and Administration
Maintaining and upgrading a central timesharing system is often easier and more cost effective than managing

multiple individual PCs or workstations.

5. Limited User Needs
For users who perform light computing tasks like text editing, programming or accessing shared files, a timesharing system is sufficient and avoids the cost of dedicated workstation.

6. Remote Access
Timesharing systems enable remote access to centralized computing resources, making them suitable for users working from different locations.

7. Scalability
A timesharing system can be scaled more easily to accomodate more users by upgrading the central system, which is often ~~used~~ more cost effective than upgrading multiple PCs.

8. Large-scale Batch Processing
Timesharing systems can handle tasks that need batch processing, such as compiling large programs, data processing, or report generation, efficiently by allocating processing time.

Q12) Consider the traffic deadlock depicted in figure below.

a) Show that the four necessary conditions for deadlock indeed hold in this example.

→

The four necessary conditions for deadlock are as follows:

1. Mutual Exclusion
In the figure given, each vehicle occupies a section of the road, and other cars cannot enter the same section. This shows mutual exclusion.

2. Hold and Wait
Here, a vehicle is holding its current section and waiting for another section of the road (currently occupied by another vehicle) to become free.

3. No Preemption
A vehicle cannot be forced to leave its section of the road, it will only leave voluntarily when it moves forward.

4. Circular wait
There exists a circular chain of vehicles, where each vehicle is waiting for the section held by the next vehicle in the chain.

b) since all four conditions are satisfied, it indeed is a deadlock.

b) State a simple rule for avoiding deadlocks in this system.

→
A simple rule to avoid deadlocks in traffic system is:
"Never enter an intersection unless you are sure you can completely clear it."

By following this rule, vehicles will not block the intersection because they will only move forward when there is

enough space for them to clear the entire intersection. This breaks the Hold and wait condition, thus preventing deadlock.

Q13) In process creation, what are the possibilities in concerned
(1) Parent execution
(2) Address space of the new process (child)?

→

1. Parent Execution

When a parent process creates a child process, there are two possibilities for the execution of the parent process:

(i) Parent continues to execute concurrently with the child
- The parent process continues its execution while the child process runs concurrently.
- This allows both processes to proceed independently and simultaneously, which is common in modern Os.

(ii) Parent waits for the child to complete
- The parent process pauses its execution and waits until the child process finishes.
- This is often used when the parent process needs results or outputs from the child process before it can proceed further.

2. Address Space of the New Process (child)

When a new process (child) is created, the address space of the child process can be handled in two ways:

(i) Child process duplicates the parent's address space
- The child process is created with a duplicate copy of the parents address space. This includes the program code, data, stack and

heap.
- In many systems, an optimization called copy-on-write is used, where the child and parent initially share the same memory pages, and copies are made only when one of the processes modifies the memory.

(ii) Child process has a new, separate address space
- The child process does not inherit the parent's address space; instead, it is given an entirely new address space.
- This happens when the child process loads a new program into memory using a system call like exec() in UNIX-based systems.