03-assignment.md

OBL3-OS – Assignment

1. Synchronisation

1.1 Process Isolation and Communication

(a) Example of communication:

A process sends a message to another process using inter-process communication (IPC) mechanisms like pipes, message queues, or shared memory.

(b) How it works:

The sending process writes data to a communication channel, while the receiving process reads from it. The OS manages the data transfer and ensures that processes can communicate without interfering with each other's memory.

(c) Possible problems:

Issues may include message loss, deadlocks if processes wait indefinitely for messages, and increased complexity in managing communication protocols.

Explicit Example:

When I write the command `ls | grep "txt"` how is this example of process communication?

In this example, the output of the `ls` command is sent through a pipe (`|`) to the `grep` command. This is a form of inter-process communication where the two processes (ls and grep) communicate by passing data through the pipe.

Terms:

Inter-process communication (IPC): Mechanisms that allow processes to communicate and share data.

Pipe: A unidirectional communication channel that connects the output of one process to the input of another.

1.2 Critical Region

What is a critical region? Can a process be interrupted while in a critical region?

A `critical region` is a section of code that accesses `shared resources` and must not be executed by more than one process or thread at a time. This is crucial for preventing `race conditions` and ensuring data consistency.

A process can be interrupted while in a critical region, but this can lead to `inconsistencies` and `race conditions` . To prevent this, mechanisms like locks or semaphores are used to ensure that only one process can enter the `critical region` at a time.

Terms:

Race condition: When software behavior depends on the unpredictable order of events, causing errors if multiple processes access shared resources at the same time.

Critical section / critical region: Code that accesses shared resources and must be protected to avoid concurrent execution by multiple processes or threads.

Inconsistency: A situation where shared data is not in a predictable state due to concurrent access.

Shared resources: Resources (like memory, files, or devices) that can be accessed by multiple processes or threads.

1.3 Busy Waiting vs Blocking

> Explain the difference between `busy waiting (polling)` versus `blocking (wait/signal)` in the context of a process trying to get access to a critical section.

Busy Waiting (Polling): In `busy waiting`, a process repeatedly checks if a condition is true (if a lock is available) in a tight loop. This consumes CPU cycles and can lead to inefficiency, especially if the wait time is long.

Blocking (Wait/Signal): In `blocking`, a process that cannot enter a critical section is put to sleep (blocked) until the condition is met (the lock is released). This is more efficient as it frees up the CPU for other processes.

Terms:

> **Busy waiting (polling):** A technique where a process continuously checks for a condition to be true, consuming CPU resources.
>
> **Blocking (wait/signal):** A technique where a process is put to sleep until a condition is met, allowing other processes to use the CPU.

1.4 Race Condition

What is a race condition? Give a real-world example.

> A real world example of a `race condition` is when two bank account transactions occur simultaneously on the same account. If both transactions read the account balance at the same time before either has updated it, they may both proceed to withdraw funds based on the same initial balance, leading to an incorrect final balance.
>
> In my previous subject, *databases*, where we used MySQL: if two transactions try to update the same row at the same time without proper locking mechanisms, it can lead to inconsistencies and lost updates. Where you could choose the isolation level to avoid this.

Terms:

> **Race condition:** A situation where the behavior of software depends on the timing or sequence of uncontrollable events, leading to unpredictable results if multiple processes access shared resources concurrently without proper synchronization.

1.5 Spin-lock

What is a spin-lock, and why and where is it used?

> A `spin-lock` is a synchronization mechanism used to protect shared resources in multi-threaded environments. It works by having a thread repeatedly check (or "spin") on a lock variable until it becomes available. Spin-locks are typically used in scenarios where the wait time for the lock is expected to be very short, as they avoid the process of putting a thread to sleep and waking it up.

1.6 Multi-core Thread Synchronisation

List the issues involved with thread synchronisation in multi-core architectures. Two lock algorithms are `MCS` and `RCU` (`read-copy-update`). Describe the problems they attempt to address. What hardware mechanism lies at the heart of each?

> Issues involved with thread synchronization in multi-core architectures include:
>
> 1. `Cache coherence`: Ensuring that all cores have a consistent view of memory.
> 2. `Contention`: Multiple threads competing for the same lock can lead to performance bottlenecks.
> 3. `Scalability`: As the number of cores increases, traditional locking mechanisms may not scale well, leading to increased latency.
>
> `MCS` (`Mellor-Crummey and Scott`) Lock:
>
> - Problem Addressed: Reduces `contention` and improves `scalability` by using a queue-based approach where each thread spins on its own node.
> - Hardware Mechanism: Utilizes `atomic operations` like `compare-and-swap (CAS)` to manage the lock state.
>
> `RCU` (`Read-Copy-Update`):

- Problem Addressed: Allows multiple readers to access data concurrently while writers update copies, minimizing `contention`.
- Hardware Mechanism: Relies on `memory barriers` and `atomic operations` to ensure safe updates.

Terms:

Cache coherence: The consistency of data stored in local caches of a shared resource, ensuring all processors see the same values.

Contention: A situation where multiple threads or processes compete for the same resource, such as a lock, leading to delays.

Scalability: The ability of a system or algorithm to maintain performance as the number of cores or threads increases.

Atomic operation: An operation that completes in a single step relative to other threads, preventing race conditions.

Compare-and-swap (CAS): An atomic instruction that updates a variable only if it matches an expected value, used for synchronization.

Memory barrier: A hardware or software mechanism that ensures memory operations are completed in a specific order, preventing reordering by the CPU.

MCS lock: A scalable, queue-based spinlock where each thread spins on its own node, reducing contention.

RCU (Read-Copy-Update): A synchronization mechanism that allows concurrent reads and defers updates, improving performance for read-heavy workloads.

2. Deadlocks

2.1 Starvation vs Deadlock

Starvation Occurs when a process is perpetually denied access to resources it needs to proceed, often due to other higher-priority processes continuously taking precedence. This can happen in scheduling algorithms that favor certain processes over others.

Example ABC:
Suppose you have three processes: `A` (high priority), `B` (medium priority), and `C` (low priority). In a priority-based scheduler, if processes `A` and `B` keep arriving or getting new work, process `C` will always be preempted and never get CPU time.

Explicit Example: For instance, imagine a print server where urgent print jobs (high priority) and normal print jobs (medium priority) keep coming in. A background maintenance task (low priority) is always ready to run, but because new print jobs keep arriving, the maintenance task never gets scheduled. This is starvation: the maintenance task is perpetually denied CPU time, even though it is ready and waiting.

Deadlock

On the other hand, is a situation where a group of processes are all waiting for resources held by each other, resulting in a standstill where none of the processes can proceed.

Example of Deadlock: Consider two processes, `P1` and `P2`. `P1` holds Resource `R1` and waits for Resource `R2`, while `P2` holds Resource `R2` and waits for Resource `R1`. Neither process can proceed because each is waiting for the other to release the resource it needs, resulting in a deadlock.

Explained in a "real-world" context: Imagine two kids are drawing. Kid `A` has a pencil, and needs an eraser to continue. Kid `B` has an eraser, but needs a pencil to continue. Neither kid can proceed because each is waiting for the other to give up their resource. The only way out is for one kid to voluntarily give up their resource, but if neither does, they are deadlocked.

2.2 Four Necessary Conditions

What are the four necessary conditions for a deadlock? Which of these are inherent properties of an operating system?

The four necessary conditions for a deadlock are:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode; only one process can use the resource at any given time. (Inherent to OS)
2. **Hold and Wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes. (Not inherent to OS)
3. **No Preemption**: Resources cannot be forcibly taken away from a process holding them; they must be released voluntarily. (Inherent to OS)
4. **Circular Wait**: A set of processes are waiting for each other in a circular chain, where each process holds a resource needed by the next process in the chain. (Not inherent to OS)

2.3 Deadlock Detection

How does an operating system detect a deadlock state? What information does it have available to make this assessment?

An operating system can detect a deadlock state by maintaining a `resource allocation graph` or using algorithms like the `Banker's algorithm`. The OS keeps track of the resources allocated to each process, the resources requested by each process, and the total available resources in the system. By analyzing this information, the OS can determine if there is a `circular wait` condition among processes, indicating a `deadlock`. **Terms**: **Resource allocation graph:** A directed graph that represents the allocation of resources to processes and the requests made by processes for resources.

3. Scheduling

3.1 Uniprocessor Scheduling

(a) When is `first-in-first-out (FIFO)` scheduling optimal in terms of average response time? Why?

`FIFO` scheduling is optimal in terms of average response time when all processes have the same execution time. In this scenario, since all processes require the same amount of time to complete, the order in which they arrive does not affect the overall average response time. Each process will finish in the order they arrive, leading to a fair and predictable scheduling outcome.

(b) Describe how `Multilevel Feedback Queues (MFQ)` combines `first-in-first-out`, `shortest job first`, and `round robin` scheduling in an attempt at a fair and efficient scheduler. What (if any) are its shortcomings?

`Multilevel Feedback Queue (MFQ)` scheduling is a complex scheduling algorithm that combines elements of `First-In-First-Out (FIFO)`, `Shortest Job First (SJF)`, and `Round Robin (RR)` to create a more dynamic and responsive scheduling system. In `MFQ`, processes are assigned to different queues based on their behavior. New processes typically start in the highest priority queue, which may use `FIFO` scheduling for short bursts. If a process uses up its time slice without completing, it is moved to a lower priority queue, which may use `Round Robin` scheduling to ensure fairness among longer-running processes. Additionally, `MFQ` can incorporate `SJF` principles by promoting shorter jobs to higher priority queues. Shortcomings of `MFQ` include complexity in implementation:

- Difficulty in tuning parameters such as the number of queues and time slices.
- Potential for `starvation` of lower-priority processes if higher-priority queues are frequently populated.

Explicit Example:

Consider three processes: `P1` (burst time 5), `P2` (burst time 3), and `P3` (burst time 8). In an `MFQ` system:

- `P1` arrives first and is placed in the highest priority queue, where it gets executed first (`FIFO`).
- `P2` arrives next and is also placed in the highest priority queue. Since `P1` is still running, `P2` waits.
- After `P1` completes, `P2` gets executed next (`FIFO`).
- `P3` arrives last and is placed in the highest priority queue. However, since `P2` is running, `P3` waits.
- If `P3` uses up its time slice without completing, it is moved to a lower priority queue, where it will be scheduled using `Round Robin`.

Explicit example in real life, with children drawing:

Imagine a group of children drawing at a table. Each child represents a process, and their drawing time represents the burst time.

- Child 1 starts drawing first and has a lot of space (high priority queue). They finish their drawing quickly.
- Child 2 starts drawing next but has to wait for Child 1 to finish.
- Once Child 1 is done, Child 2 gets their turn.
- Child 3 arrives last and has to wait for both Child 1 and Child 2 to finish.
- If Child 3 takes too long, they might be moved to a different table (lower priority queue) where they have to share space with other children (`Round Robin`).

Explicit shortcomings example:

In the same drawing scenario, if Child 1 and Child 2 keep arriving with short drawing times, Child 3 may never get a chance to draw, leading to `starvation` .

Terms:

First-In-First-Out (FIFO): A scheduling algorithm where the first process to arrive is the first to be executed.

Shortest Job First (SJF): A scheduling algorithm that selects the process with the shortest execution time next.

Round Robin (RR): A scheduling algorithm that assigns a fixed time slice to each process in a cyclic order.

Multilevel Feedback Queue (MFQ): A scheduling algorithm that uses multiple queues with different priority levels and scheduling algorithms to manage process execution dynamically.

3.2 Multi-core Scheduling

(a) Similar to thread synchronisation, a uniprocessor scheduler running on a multi-core system can be very inefficient. Explain why (there are three main reasons). Use `MFQ` as an example.

1. **Load Imbalance:** In an `MFQ` system, processes may not be evenly distributed across cores. If all high-priority processes are assigned to a single core, it can become a bottleneck, leading to inefficient CPU utilization. *Fix this by balancing the load across cores.*
2. **Context Switching Overhead:** `MFQ` may require frequent context switching between processes in different queues. On a multi-core system, this can lead to increased overhead as the scheduler must manage multiple queues and their associated processes, potentially causing delays. *Fix this by minimizing context switches and optimizing queue management.*
3. **Inflexible Scheduling:** `MFQ` uses fixed queues for different priorities, which can be too rigid for multi-core systems. If one core gets too many high-priority tasks, other tasks (especially lower-priority ones) might have to wait a long time. This can make the system less efficient because some cores are overloaded while others are underused. *Fix this by implementing dynamic priority adjustments and better load balancing.*

(b) Explain the concept of `work-stealing` .

`Work-stealing` is a technique used in multi-core systems to improve load balancing and resource utilization. In this approach, each core maintains its own queue of tasks to execute. When a core finishes its tasks and becomes idle, it can "steal" tasks from the queues of other busy cores. This helps to ensure that all cores remain active and reduces the likelihood of some cores being overloaded while others are idle.

Explicit Example:

Imagine a multi-core system with four cores (`C1` , `C2` , `C3` , `C4`). Each core has its own queue of tasks to execute. If `C1` finishes its tasks and becomes idle, it can look at the queues of `C2` , `C3` , and `C4` . If `C2` has a long queue of tasks, `C1` can "steal" some of those tasks and execute them, helping to balance the workload across all cores.

Explicit real-life example using children drawing:

Consider a scenario where four children (`C1` , `C2` , `C3` , `C4`) are drawing pictures. Each child has their own set of drawing materials and space to work. If `C1` finishes their drawing and has extra time, they can look at the other children's workspaces. If `C2` is struggling with a complex drawing and has a lot of unfinished work, `C1` can offer to help by taking some of `C2` 's materials or even drawing parts of the picture themselves. This way, all the children can stay engaged and productive, rather than having one child overwhelmed while others are idle.

Terms:

Work-stealing: A load balancing technique where idle processors (or cores) can "steal" tasks from busy processors to improve overall system efficiency and resource utilization.

Load balancing: The process of distributing tasks evenly across multiple processors or cores to maximize resource utilization and minimize response time.

Context switch: The act of saving the state of a currently running process or thread so that another can be executed, introducing overhead.

Questions for Improvement

- Do you prefer specific examples in the answers, or is general explanation sufficient? Right now im trying to mix both. And is reallife examples preferred to give a better understanding? (examples with kids drawing)
- Do you prefer code snippets in C (fork+pipe, mutex/race demo) to show actual results of theory to prove concepts, or is a general explanation acceptable?
- Do you prefer the inclusion of each term in the explanations? Or do you prefer for me to just explain the concepts?