OBL3-OS

1. Synchronisation

1. (a)

One common example of inter-process communication is the use of pipes in Unix-like operating systems. Pipes allow data to be passed from one process to another.

In the command cat file.txt | grep “search_term”, the cat command reads the contents of file.txt and sends it to the standard output, which is then piped into the input of the grep command. The grep command searches the data for the string "search_term" and prints any matching lines to its standard output.

(b)

There are several methods for IPC:

1. Pipes: As described above, they allow one process to send data to another process. Data written to the write-end of a pipe can be read from the read-end of the same pipe.

2. Message Queues: Processes can send and receive messages to/from a message queue. The operating system maintains this queue, ensuring data integrity and delivery.

3. Shared Memory: A block of memory is set up so that multiple processes can read and write to it. Access to this block is controlled by synchronization mechanisms like semaphores to ensure data consistency.

The operating system provides mechanisms for setting up, using, and tearing down these communication methods, ensuring they work correctly and securely.

(c) Problems that can result from inter-process communication:

1. Deadlock: Two or more processes might be waiting for each other to release a resource, causing all of them to be stuck indefinitely.

2. Concurrency Issues: Without proper synchronization mechanisms, processes that access shared resources (like shared memory) concurrently might produce unpredictable results.

3. Overhead: Setting up and maintaining IPC can consume system resources and degrade performance.

4. Security Concerns: Improperly configured IPC mechanisms can be exploited. For example, shared memory regions without proper access controls could be accessed by unauthorized processes.

5. Complexity: Writing code that uses IPC can be more complicated than writing single-process code. Developers need to handle the communication, error cases, and synchronization.

2. Critical Region (or Critical Section): A critical region is a section of code where a process accesses shared resources and must not be accessed by other processes simultaneously to prevent data inconsistency or other concurrency issues.

Can a process be interrupted while in a critical region? Ideally, no. When a process is inside a critical region, it should not be interrupted by another process that might want to access the same resources. Techniques like disabling interrupts or using locks are employed to ensure that only one process accesses the shared resource at a given time, preventing potential concurrency problems.

3. Busy Waiting (Polling):

A process continuously checks a condition and waits in a loop until it can enter the critical section. This consumes CPU cycles as the process remains active but doesn't make progress.

Blocking (Wait/Signal):

A process is put to sleep (blocked) if it cannot enter the critical section. It's awakened (signaled) when access is possible. This approach frees up CPU cycles for other tasks while the process waits.

Difference:

Busy waiting wastes CPU time actively checking access, while blocking releases the CPU to handle other tasks until access is granted.

4. Race Condition:

A race condition occurs when the behavior of a system depends on the relative timing of events, such as the order in which processes or threads are scheduled. If these events don't occur in the expected order, unexpected and often undesirable outcomes can result.

Real-world Example:

Consider an online ticket booking system where two users are trying to book the last available seat on a flight at the same time. If the system doesn't handle simultaneous requests properly, both users might get a confirmation, leading to an overbooking. The race here is between the two booking requests trying to claim the last seat. Proper synchronization would ensure that once a seat is booked by one user, it's no longer available to others.

5. Spin-lock:

A spin-lock is a type of synchronization mechanism used to protect critical sections. When a process or thread attempts to acquire a spin-lock that is already held by another entity, it will "spin" in a loop, continuously checking until the lock becomes available, rather than relinquishing the CPU and going to sleep.

Why and Where It's Used:

1. Low Overhead: Spin-locks have a lower overhead compared to more complex locking mechanisms, making them efficient for short durations.

2. Short Wait Times: They are beneficial in scenarios where the expected wait time for the lock is very short, often shorter than the time it would take to put a thread to sleep and then wake it up again.

4. Multiprocessor Systems: Spin-locks are often used in multiprocessor or multi-core systems where one processor can spin while waiting for another processor to release the lock.

However, spin-locks can be wasteful in scenarios where the lock is held for more extended periods, as the spinning process consumes CPU cycles without doing useful work. In such cases, other synchronization mechanisms that block the waiting process might be more appropriate.

6. Issues Involved with Thread Synchronization in Multi-core Architectures:

1. Cache Coherence: Multiple cores have their own local caches. Ensuring that all cores have a consistent view of shared data (when one core modifies data in its cache) is a challenge.
2. False Sharing: Even if different threads on different cores modify independent data, if that data lies on the same cache line, it can cause cache invalidations, leading to performance degradation.
3. Contention: When multiple threads compete for a shared resource, the lock protecting that resource can become a bottleneck.
4. Memory Visibility: Ensuring that writes by one core are immediately visible to another core.

5. Deadlock: Situations where threads are waiting for resources in a cycle, causing a standstill.
6. Starvation: Some threads might not get access to resources for extended periods if other threads dominate access.
7. Load Balancing: Distributing work evenly across threads and cores can be challenging, leading to some cores being underutilized.

MCS (Mellor-Crummey and Scott) Lock:

- Problem Addressed: The MCS lock aims to reduce the contention and overhead associated with locks in multi-core systems, especially when many threads contend for the same lock.
- Hardware Mechanism: The MCS lock relies on atomic operations and memory ordering semantics. The core idea is a queue-based locking mechanism where each waiting thread spins on its own local variable (reducing cache coherence traffic) until it acquires the lock.

RCU (Read-Copy-Update):

- Problem Addressed: RCU addresses the issue of reading shared data without locks while still allowing updates to the data. It's particularly beneficial for situations with many reads and infrequent updates.
- Hardware Mechanism: RCU primarily relies on memory barriers to ensure the correct ordering of operations. When updates occur, they don't overwrite the original data immediately. Instead, they create a new copy, and readers can continue reading the old version until it's safe to reclaim, ensuring that readers are not blocked by writers.

In summary, both MCS and RCU are mechanisms to deal with the challenges posed by multi-core architectures, with each focusing on optimizing specific synchronization scenarios.

2. Deadlocks

1. Resource Starvation:

Resource starvation occurs when a process or thread is perpetually denied the resources it needs to perform its task. Even though the system may be functioning and processing other tasks, a specific process might not get the resources it needs, causing it to wait indefinitely. Starvation can result from factors like scheduling algorithms favoring certain processes over others or priority inversion.

Deadlock:

Deadlock is a specific condition where two or more processes are each waiting for another to release a resource, and none of them can proceed. In a deadlock, the involved processes are stuck in a circular waiting condition, and without external intervention, none can move forward.

Difference:

The primary distinction is that in resource starvation, a process waits indefinitely for a resource due to various reasons but not necessarily because other processes hold the resources in a circular wait

pattern. In contrast, a deadlock specifically refers to a situation where processes are stuck because they are all waiting for resources held by others in the set.

2. The four necessary conditions for a deadlock are:

1. Mutual Exclusion: At least one resource must be non-shareable (only one process can use it at a time).
2. Hold and Wait: A process must be holding at least one resource and requesting additional resources that are currently held by other processes.
3. No Preemption: Resources cannot be forcibly taken away from a process. They must be released voluntarily.
4. Circular Wait: A set of processes exists where each process in the set is waiting for a resource held by the next process in the set, creating a cycle.

Of these conditions, Mutual Exclusion and No Preemption are often inherent properties of many operating systems, especially when considering resources like printers or some types of memory. The other conditions arise from specific choices in process scheduling, resource allocation strategies, or program behavior.

3. To detect a deadlock, an operating system often uses a resource allocation graph (RAG) or other data structures that keep track of which processes hold and request resources.

Resource Allocation Graph (RAG):

This is a directed graph where nodes represent processes and resources. An edge from a resource to a process indicates that the resource is allocated to the process. An edge from a process to a resource shows the process has requested that resource.

Deadlock Detection:

A deadlock exists if and only if the RAG has a cycle, and if each resource in the cycle can have only one instance. If a cycle exists, but some resources can have multiple instances, further analysis is needed to determine if a deadlock is present.

Information Available:

1. Allocation Matrix: Represents the number of each resource currently allocated to each process.

2. Request Matrix: Represents the number of each resource a process is currently requesting.

3. Available Vector: Represents the number of available instances of each resource type.

By examining this information and comparing the number of resources requested to the number of available resources, the OS can identify potential deadlock situations.

Periodically, the OS will use this information to check for deadlocks. If a deadlock is detected, recovery actions, such as terminating or rolling back processes, might be initiated.

3. Scheduling

1. (a) First-in-first-out (FIFO) scheduling:

Optimal Condition: FIFO (or FCFS, First-Come-First-Serve) scheduling is optimal in terms of average response time when all jobs are of the same length or when all jobs arrive at the same time.

Why: Because when all jobs have equal lengths, each job will wait for exactly the same amount of time before it starts executing, ensuring the fairest distribution of CPU time. If they arrive at the same time and have varied lengths, the response time average is minimized, though it might not be optimal for individual tasks.

(b) Multilevel Feedback Queues (MFQ):

Description: MFQ is a complex scheduling algorithm designed to adapt to a process's behavior and estimation of its CPU burst time. It does so by:

1. First-in-first-out (FIFO): Used for jobs in higher-priority queues, typically for short tasks that have just entered the system.

2. Shortest Job First (SJF): By continually adjusting a job's position in the queue based on its observed characteristics, MFQ can approximate SJF. Shorter jobs get quickly promoted to higher-priority queues.

3. Round Robin (RR): Used for jobs in lower-priority queues, especially for long-running or I/O-bound tasks. This ensures they get a share of the CPU but without monopolizing it.

Shortcomings of MFQ:

1. Complexity: MFQ is more complex to implement than simpler schedulers.

2. Starvation: If not carefully managed, lower-priority jobs might suffer starvation.

3. Tuning Required: MFQ often requires careful tuning of parameters like time quantum or the criteria for promoting/demoting jobs between queues.

In summary, while MFQ attempts to combine the benefits of several scheduling algorithms to adapt to a variety of job types and loads, it's not without its complexities and potential pitfalls.

2. (a) Inefficiencies of a Uniprocessor Scheduler on a Multi-core System:

1. Lack of Parallelism Utilization: A uniprocessor scheduler, like MFQ, when applied directly to a multi-core system, doesn't naturally distribute work among available cores. The scheduler would not exploit the full potential of multiple cores by running multiple threads/tasks simultaneously.

2. Contention Over Shared Structures: When multiple cores try to access and modify the scheduler's shared structures (like queues in MFQ), it can lead to contention. This contention can cause delays, as cores may need to wait to access these structures.

3. Cache Inefficiencies: On a multi-core system, each core typically has its own cache. A uniprocessor scheduler doesn't account for cache locality. If tasks frequently switch between cores, it can lead to cache misses, degrading performance.

Using MFQ as an example: If MFQ, designed for a uniprocessor, is directly applied to a multi-core system, the feedback queues become contention points. Also, if a process is moved between cores, the cache warm-up benefits on one core are lost, causing inefficiencies.

(b) Work-stealing Concept:

Work-stealing is a scheduling strategy where idle cores ("thieves") take (or "steal") tasks from the queues of busy cores. The primary objective is to keep all cores in a multi-core system as busy as possible by dynamically redistributing tasks. When a core finishes its local tasks, instead of remaining idle, it tries to find work from other overloaded cores, ensuring better load balancing and utilization across the system.