Usman Ghafoorzai

NTNU

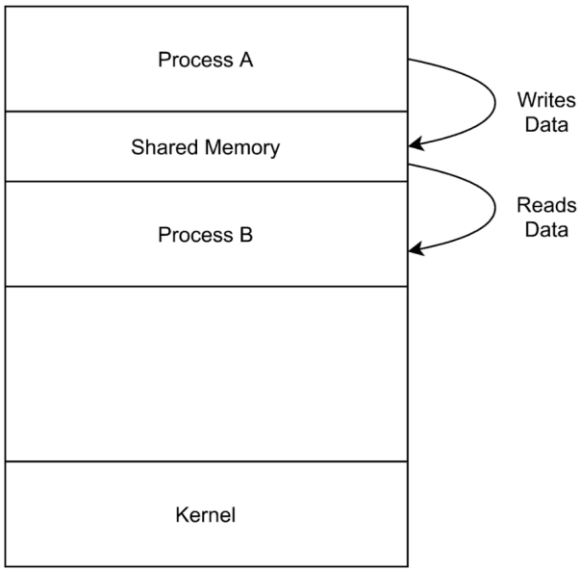
OBL3-OS

1 Synchronization

1.1 The principle of process isolation in an operating system means that processes must not have access to the address spaces of other processes or the kernel. However, processes also need to communicate.

(a) Give an example of such communication

The processes need to communicate with each other to exchange data and information; they do so via the mechanism known as Inter-Process Communication (IPC). An example of inter-process communication is through **shared memory**, which requires communicating processes to establish a shared memory region. Generally, the processes that wish to communicate to this process need to attach their address space to this shared memory segment. The figure below shows that process A and process B established a shared memory segment and exchanges information through the shared memory region:



The OS usually prevents processes from accessing other process memory, therefore this model requires processes to agree to remove this restriction. Furthermore, as

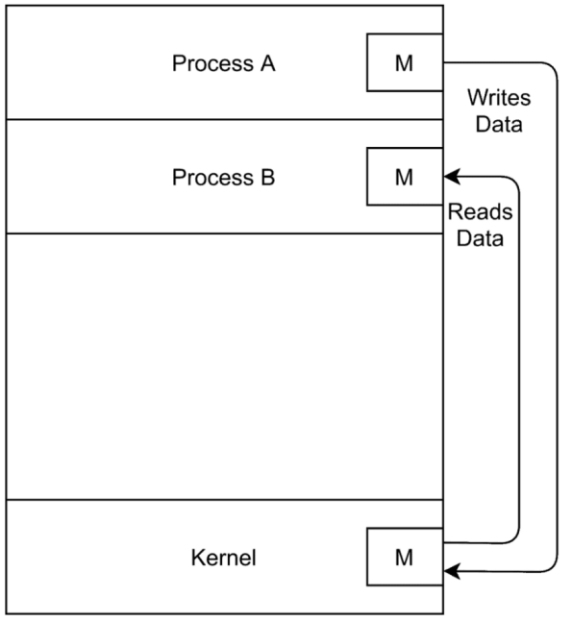
shared memory is established, the processes are also responsible to ensure synchronisation so that both processes are not writing on the same location at the same time.

(b) How does this communication work?

IPC works by providing a set of communication mechanisms and protocols. The processes then use that to exchange information. Categorized into various types, including shared memory, message passing, pipes, sockets and remote procedure calls (RPC). Each one of these has its own characteristics and usage patterns, but they all serve the purpose of facilitating communication between processes. For **shared memory**, look at 1.1 (a). **Pipes** creates a unidirectional communication channel between two processes. One writes to it, and the other reads from it. Commonly used between related processes like a parent and child. **Semaphores** are synchronisation tool that lets processes to signal each other or manage access to shared resources. Can be used to control how many processes can access a particular resource at the same time, ensuring coordination and preventing race condition. **Message Passing –** in this mode, processes exchange information through messages, which are handled by the operating system. The process that wants to send data puts it into a message and passes it to the OS, which then delivers it to the receiving process. Unlike shared memory, message passing can be used across different systems, making it suitable for distributed environments. This method ensures data isolation and can be synchronized or asynchronous, depending on the requirements. Direct or indirect communication can be used, where messages are sent either directly between processes or via mailboxes. Pipes, message queues and sockets are a part of message passing. For **Sockets**, they establish a bidirectional communication channel between processes. They establish a bidirectional communication channel between processes. Processes can send and receive messages (packets) over the network using sockets. Sockets are widely used in client-server architectures, such as web browsers communicating with web servers. Data sent through sockets is considered a form of

message passing because one process sends a message, and another process receives it.

The figure below shows message passing.



(c) What problems can result from inter-process communication?

Problems resulting from inter-process communication (IPC) can arise due to several challenges. First, IPC increases system complexity, making it harder to design, implement, and debug, as multiple processes need to be properly synchronized. Additionally, IPC can introduce security vulnerabilities, where processes might inadvertently or maliciously access or modify data belonging to others, leading to potential breaches. Resource management is also critical, as IPC consumes system resources like memory and CPU time, and poor management can degrade overall performance. Furthermore, data inconsistencies can occur if multiple processes try to access or modify the same data simultaneously without proper synchronization, resulting in race conditions or corrupted data. While IPC is crucial for enabling processes to collaborate in modern operating systems, these challenges must be carefully addressed to ensure security, performance, and data integrity.

Usman Ghafoorzai

NTNU

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

Critical region is a part of code that must be executed exclusively by one thread or process at a time – to ensure safe access to shared resources, such as variables, data structures or devices. This prevents race conditions, which can lead to data inconsistencies or unpredictable program behaviour when multiple processes try to access shared resources simultaneously. The picture below, retrieved from geeksforgeeks.org shows some characteristics and requirements for critical region.

Critical Region Characteristics and Requirements

Following are the characteristics and requirements for critical regions in an operating system.

1. Mutual Exclusion

Only one procedure or thread can access the important region at a time. This ensures that concurrent entry does not bring about facts corruption or inconsistent states.

2. Atomicity

The execution of code within an essential region is dealt with as an indivisible unit of execution. It way that after a system or thread enters a vital place, it completes its execution without interruption.

3. Synchronization

Processes or threads waiting to go into a essential vicinity are synchronized to prevent simultaneous access. They commonly appoint synchronization primitives, inclusive of locks or semaphores, to govern access and put in force mutual exclusion.

4. Minimal Time Spent in Critical Regions

It is perfect to reduce the time spent inside crucial regions to reduce the capacity for contention and improve gadget overall performance. Lengthy execution within essential regions can increase the waiting time for different strategies or threads.

In practice, a process cannot be interrupted while it is in a critical region, since it holds a lock or uses another synchronisation mechanism that ensures it completes its task before other threads or processes can access that region.

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

While busy waiting (polling), the process continuously checks for a condition and consumes the CPU cycle continuously by actively looping and repeatedly testing the resource's availability. This then keeps the process in a "ready-to-run" state so it can provide immediate access once the critical section becomes free, but it is highly inefficient if multiple processes are waiting, since it wastes valuable processor time.

Meanwhile, in blocking a process does not use CPU by remaining inactive until it gets the desired condition, by releasing the CPU for other processes to use and waits passively until it receives a signal from another process indicating that the critical section is available. This method conserves CPU resources by putting the process to sleep and only reactivating it when access is possible.

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

A race condition occurs when a program's behaviour depends on the unpredictable order in which operations from different threads or processes execute. This "race" between threads happens when they attempt to access and modify shared resources concurrently, with the outcome depending on which thread reaches the critical section first. Since the interleaving of operations is uncertain, the results can vary with each execution, leading to inconsistent or unintended outcomes.

For a real-world example, imagine an online ticket booking system. If two users (User A and User B) try to book the last available seat at the same time, a race condition may arise. If both users are allowed to proceed without a locking mechanism, the system might accidentally double-book the seat, confirming it for both users. Proper synchronization, such as using locks, can prevent such race conditions by ensuring that only one user's transaction is processed at a time, securing accurate seat availability.

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

A spin-lock is a synchronisation mechanism designed to prevent multiple threads or processes from accessing a shared resource simultaneously. Unlike other synchronisation methods, spin-lock employ a “busy-wait” approach. As previously explained, it means that when a thread encounters a spin-lock, it repeatedly checks if the lock is available, continuously “spinning” until it can enter the critical section. This mechanism is useful for maintaining exclusive access to shared resources when the wait time is expected to be very short, as it avoids the overhead of putting threads to sleep and then waking them up again.

However, as stated earlier, busy-waiting consumes CPU cycles, which can be inefficient if the critical section is held for too long. In such cases, other synchronisation mechanisms might be more appropriate. Spinlocks are especially advantageous in multi-threaded environments for small, quick sections of code, where they can prevent data inconsistency without the complexity of more substantial locking mechanisms. By limiting the time spent in critical sections and reducing context-switching overhead, spinlocks help the operating system manage resource access efficiently, ensuring order and data integrity.

1.6 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?

In multi-core architectures, thread synchronization is essential but challenging due to issues like race conditions, contention, and cache coherence, all of which can lead to inefficiencies and potential inconsistencies. Synchronization aims to ensure orderly access to shared data, maintain data integrity, and maximize parallelism across multiple cores. The MCS (Mellor-Crummey and Scott) lock and RCU (Read-Copy-Update) mechanism are two synchronization algorithms used to handle these issues, each

addressing specific problems in multi-core environments and relying on different hardware mechanisms.

Issues in Thread Synchronization in Multi-Core Architectures

1. **Race Conditions**: When multiple threads access shared data simultaneously, uncoordinated access can lead to inconsistent data.
2. **Cache Coherence**: Caches must be kept consistent across cores, but frequent synchronization creates "cache coherence traffic," slowing down performance.
3. **Contention and Scalability**: In scenarios with many threads, high contention on locks can lead to delays and reduced parallelism.
4. **Deadlocks**: Mismanagement of multiple locks can result in deadlock, where threads wait indefinitely for each other.
5. **Latency and Overhead**: Synchronization mechanisms like locks can introduce latency, especially with high-frequency locking/unlocking.

MCS Lock: Problem and Hardware Mechanism

The **MCS lock** is a scalable spinlock designed to minimize contention and cache coherence traffic by organizing waiting threads in a queue. Each thread "spins" on a private variable rather than on a shared lock variable, preventing other cores from constantly refreshing their cache with updates to the same lock variable. This approach addresses the problems of:

- **Contention** by structuring a fair, FIFO queue, reducing the constant back-and-forth between threads.
- **Cache coherence** by limiting spinning to a local variable, which minimizes coherence traffic between cores.

Hardware Mechanism: The core hardware mechanism supporting MCS locks is **atomic memory operations** (like compare-and-swap or atomic exchange). These operations ensure that a thread can acquire the lock atomically, adding itself to the queue without interference from other threads.

RCU (Read-Copy-Update): Problem and Hardware Mechanism

RCU is designed for scenarios with high read contention but relatively infrequent writes. It allows multiple readers to concurrently access a shared data structure while maintaining a low-overhead, single-writer model. Key issues RCU addresses include:

- **Read scalability** by allowing multiple readers without synchronization overhead, ensuring that reads are lock-free and do not interfere with each other.
- **Graceful update management** by serializing writes and maintaining old versions of data structures until all readers have finished accessing them (known as the *grace period*).

Hardware Mechanism: RCU relies on the hardware's **memory barriers** and **cache coherence** to ensure that updates to shared data are visible in the correct order to all threads. Memory barriers enforce order in operations, ensuring that writes are properly synchronized across processors without explicitly locking readers. In some implementations, especially in the Linux kernel, **interrupt disable/enable mechanisms** are used to ensure readers' critical sections aren't interrupted, simplifying synchronization.

Each of these mechanisms—MCS with atomic operations and RCU with memory barriers—leverages different aspects of the hardware to optimize for specific use cases: high contention in MCS and read-heavy workloads in RCU.

2 Deadlocks

2.1 What is the difference between resource starvation and a deadlock?

Resource starvation occurs when a thread is perpetually denied the resources it needs to proceed, often due to resource allocation policies that favour other threads. This can happen even in a system without deadlocks, where a particular thread might not get a chance to run because other threads are continuously prioritized. In contrast, a deadlock is a specific state where two or more threads are unable to proceed because each is waiting for the other to release resources. In a deadlock, there is a cycle of dependencies that prevents any of the involved threads from making progress, whereas starvation can occur without such cyclical waiting.

2.2 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**: Resources cannot be shared and are allocated exclusively to one thread at a time.
2. **Hold and Wait**: Threads holding resources can request additional resources without releasing their current holdings.
3. **No Pre-emption**: Resources cannot be forcibly taken from threads; they must be voluntarily released.
4. **Circular Waiting**: There exists a cycle of threads, where each thread is waiting for a resource held by the next thread in the cycle.

2.3 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 using several methods, primarily through resource allocation graphs or variations of algorithms like Coffman et al.'s deadlock detection algorithm. The OS maintains information about:

- **Current allocations**: Which threads hold which resources.
- **Pending requests**: Resources that threads are waiting for.
- **Available resources**: Resources that are currently free and can be allocated.

By analysing this information, the OS constructs a graph where threads and resources are represented as nodes, and directed edges represent the "owns" and "waits for" relationships. A cycle in this graph indicates a deadlock. Additionally, the system can use conservative detection mechanisms, which might trigger recovery based on timeouts or by analysing the resource allocation state, leading to potential false positives but avoiding prolonged waits.

3 Scheduling

3.1 Uniprocessor scheduling

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

First-in-first-out (FIFO) scheduling is optimal in terms of average response time under specific conditions, particularly when tasks arrive in a uniform manner and have similar service times. This is because FIFO processes tasks in the order they arrive, ensuring that once a task starts executing, it runs to completion without preemption. In scenarios where all tasks are similar in length, FIFO minimizes the average waiting time for tasks, as each subsequent task begins execution immediately after the previous one completes.

However, FIFO can lead to suboptimal response times in cases of variable task lengths, as a long-running task could delay the execution of shorter tasks (the "convoy effect"), leading to increased average response times for those tasks. Nevertheless, under steady-state conditions with uniform service demands, FIFO is effective at maintaining a predictable response time.

(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 Queues (MFQ) combine different scheduling algorithms to improve both fairness and efficiency. It utilizes multiple queues, each with a different scheduling policy:

1. **First-In-First-Out (FIFO)**: At the highest priority level, this queue can quickly service tasks that are arriving in order, ensuring that tasks do not wait unnecessarily long when they are the first to arrive.

2. **Shortest Job First (SJF)**: At lower priority levels, the MFQ can implement a SJF approach, where tasks that require less processing time are prioritized. This ensures that shorter tasks are completed quickly, minimizing overall wait time.

3. **Round Robin (RR)**: Within the queues, a round-robin policy can be used to give each task a fair amount of CPU time, promoting responsiveness among tasks that are competing for resources.

The shortcomings of MFQ include its complexity in implementation and potential for unfairness. For example, if a task continually gets demoted to lower-priority queues due to long execution times or frequent context switching, it may experience starvation. Furthermore, the management of multiple queues can introduce overhead, potentially leading to scheduling inefficiencies.

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.

A uniprocessor scheduler running on a multi-core system can be inefficient for several reasons:

1. **Resource Contention**: When tasks are scheduled on a single processor, they may contend for shared resources, such as memory or I/O devices. In the case of MFQ, if all tasks are directed to a single queue managed by a single processor, it may create bottlenecks where tasks are blocked waiting for these resources, leading to increased response times.

2. **Poor Load Balancing**: A uniprocessor scheduler typically lacks the ability to balance workloads across multiple cores effectively. In an MFQ setup, if all tasks

are queued to a single processor, it undermines the potential of multi-core systems, where tasks could be spread out to optimize processing capabilities.

3. **Increased Context Switching**: On a uniprocessor system, frequent context switching can lead to overhead, which is compounded in an MFQ scheduler that might switch between multiple queues. As more tasks are added, the system's responsiveness can degrade, particularly as the processor's time is divided among various tasks without leveraging the multi-core architecture effectively.

(b) Explain the concept of work-stealing.

Work-stealing is a scheduling strategy used in parallel computing where idle processors "steal" tasks from busy processors to balance the workload dynamically. When a processor has completed its tasks and is idle, it will look for tasks in the queues of other busy processors. This approach helps to maximize resource utilization and reduce idle time by redistributing tasks as needed.

In a multi-core system, work-stealing is advantageous because it enables better load balancing without requiring a central scheduler. It effectively addresses issues of load imbalance by allowing processors to share the workload dynamically based on real-time conditions, leading to improved overall system performance and responsiveness. This approach is particularly beneficial in scenarios where task lengths are unpredictable, as it ensures that no single processor becomes a bottleneck.