OBL3-OS

1. Synchronization

1.
    a. One common example of process communication is in a chat application, where multiple processes need to exchange messages or data with each other.

    b. This communication could work by using shared memory which is an inter-process mechanism. Shared memory allows processes to access and modify a common region of memory. This allows processes to share data more efficiently and communicate with each other (Lenovo, 2023).

       Another way this communication could work is by using message passing. In message passing, processes typically use messaging interface/library provided by the operating system or programming language to send and receive messages (Lenovo, 2023).

    c. Problems with inter-process communication:
        - Increasing system complexity: Making it harder to design, implement, and debug.
        - Data Consistency: Ensuring that data shared between processes is consistent and doesn't lead to conflicts.
        - Deadlocks: Processes may block each other, waiting for resources that the other has.
        - Security: Unauthorized access to another process's data.
        - Performance: Overhead in message passing or sharing data can impact system performance.
          (GeeksforGeeks, 2023)

2. A critical region refers to a segment/section of code or data structure where shared variables can be accessed. Only one process can execute in its critical region at a time. The other processes must wait to execute in this region. Critical regions are utilized to prevent concurrent entry to shared sources, that allow you to maintain information integrity and keep away from race conditions. This is achieved by using synchronization mechanisms such as mutexes or semaphores. (Geeksforgeeks, 2023)

   A process can be interrupted while in a critical region. This could be done by the operative system kernel schedular. It can preempt a process in order for processes with higher priority to execute. This is done to maintain fairness, and to prevent process monopolization and process starvation.

3. In busy waiting a process is consistently waiting for another process to finish while consuming CPU resources. This could lead to high CPU utilization even when the resource isn't available until a process is preempted to re-schedule (Operating Systems, page 193). In the context of a process trying to access a critical section, a process continuously waits for another process to finish its execution in the critical section.

   In blocking a process gives up the CPU and is awakened later from an event such as when a condition variable is met. It doesn't consume CPU time, which makes it more efficient (Stackoverflow, 2014). In the context of a process trying to access a critical section, a process efficiently waits for the condition that the critical section is available.

4. A race condition is a situation in which the behavior of a system depends on the interleaving of operations of different threads, without proper synchronization. The result of the of the program execution essentially depends on who wins the race. (Operating Systems, page 187).

   Real-world example: A bank account shared by two threads - one for depositing and one for withdrawing. If not properly synchronized, a race condition can occur, where both threads access the account balance simultaneously, leading to incorrect results which could either be beneficial for or troubling for a user.

5. A spin-lock is a synchronization mechanism where a thread, instead of blocking or going to sleep when it cannot access a critical section, repeatedly checks if the lock is available. It busy waits, or "spins" in a tight busy-wait loop until some other lock releases the lock. Spin-locks is inefficient if locks are held for long periods. However, for locks that are hold in a short amount of time, spin-lock is more efficient then context switching (Operating Systems, page 240). It is typically in these situations where we would like to use spinlocks.

6. Issues involved with thread synchronization in multi-core architectures:

   a. Locking may remain a bottleneck to good performance.
   b. Locking could introduce the risk of deadlocks.
   c. Overhead of acquiring and releasing a lock can increase dramatically with high lock contention.

   The problem that the MCS (Mellor-Crummey and Scott) locks are attempting to address, is the problem of overhead of acquiring and releasing a lock when there are multiple threads contending for the lock. (Operating Systems, page 272). MCS is more efficient than test-and-set and test and test-and-set in this case. The MCS lock takes advantage of an atomic read-modify-write hardware-mechanism called compare-and-swap that is supported on most modern multiprocessor architectures. Essentially a MSC lock maintains a linked list of threads waiting for a lock without a separate spinlock. The front of the list holds the lock, while the tail is the last thread in the list. When a thread releases the lock, it signals that it is the next thread's turn. In scenarios where are few waiting threads, MCS locks are less efficient than normal spinlock (Operating Systems, page 274).

RCU is a synchronization mechanism that is particularly useful in the case where there are many more readers than writers. RCU reduces the overhead for readers at a cost of increased overhead for writers (Operating Systems, page 272). The hardware-mechanism for this lock is also the compare-and-swap instruction. This is used by a writer-thread which must public its changes to a shared data-structure. RCU allows multiple readers to access data simultaneously while providing a mechanism for writers to safely update data without affecting ongoing reads.

2. Deadlocks

1. Resource starvation occurs when a thread fails to make progress for an indefinite period of time, but it's not necessarily waiting forever (Operating Systems, page 288).

   A deadlock is a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some actions, causing all threads to be blocked indefinitely. It is a more severe and permanent form of resource contention (Operating Systems, page 285).

2. The four necessary conditions for a deadlock:

   a. Bounded resources: There are finite number finite number of threads that can simultaneously use a resource.

   b. No preemption: Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.

   c. Wait while holding: A thread holds one resource, while waiting for another. This condition is sometimes called multiple independent requests because it occurs when a thread first acquires one resource and then tries to acquire another.

   d. Circular waiting: There is a set of waiting threads such that each thread is waiting for a resource held by another.
   (Operating Systems, page 289)

   Of all the necessary condition for a deadlock, only bounded resources and no-preemption inherits properties from the operating system since it is the operating system that limits the resources and executes preemption.

3. An operating system can detect a deadlock in several ways. If there are several resources and only one thread can hold each resource at a time, then the operating system can detect a deadlock by analyzing a simple graph (Operating Systems, page 303). These graphs are often called Resource Allocation Graph (RAG) which shows which resource is held by which process and which process is waiting for a resource of a specific kind (GeeksforGeeks, 2023). This can simply be explained as graph with threads and resources represented as nodes, and edges indicating which resource is owned by which thread, and which thread is waiting for a resource. There is a deadlock if and only if there is a cycle in this graph.

   The Banker's Algorithm uses a similar kind of resource allocation and checks for safe and unsafe states by the maximum resource requirements when a thread begins a task (Operating Systems, 295). If a thread is in a safe state, the algorithm will allow the request for resource. If it is in an unsafe state, it will not allow the thread's request, even if it theoretical won't lead to a deadlock (GeeksforGeeks, 2023).

3. Scheduling

1. Uniprocessor Scheduling:
   a. In terms of average response time first-in-first-out (FIFO) scheduling is optimal if tasks are equal in size. This is because in FIFO scheduling tasks waits on each other in the order they arrive. If all tasks are roughly equal in size, the average response time for all tasks must be equal to the response time of a single task. In that sense the average response time is optimal. FIFO is most efficient when tasks are small and equal in size (Operating Systems, page 2023).

   b. Multi-level feedback queues (MFQ) can be thought of as an extension of Round Robin. Instead of only a single queue, MFQ has multiple Round Robn ques, each with a different priority level and time quantum. The highest priorities have the lowest time quantum, while the lowest priorities have the highest quantum (Operating Systems, page 325). For a task that does not finish during a time slice expiration, the task moves down one priority. This is done to favor short tasks over long ones. If a task yields processor because it is waiting on I/O, it stays at the same level (possibly gets promoted to a higher priority). If a task completes, it leaves the system. (Operating Systems, page 325)

      In order for the MFQ to achieve the goals of responsiveness, low overhead, starvation-freedom background tasks, and fairness, it could be implemented with different scheduling algorithms for the different priority-levels. First-in-first-out (FIFO) could be used in the highest priority queue to reduce the number of preemptions. Shortest-job-first

(SJF) could be used in higher priority queues in order for shorter jobs to be executed first. At last, the Robin would be used for processes in the lowest priority-queue, preventing long processes from starving other processes.

The shortcomings with MQF are that it is not perfect at any of the goals (mentioned above) it is trying to achieve. Bottlenecks could occur, leading to starvation for tasks stuck in low priority queues, and increasing overhead for managing queues. It is rather intended to be a reasonable compromise in most real-world cases. (Operating Systems, page 327).

2. Multi-Core Scheduling:

   a. A uniprocessor schedular running on a multi-core system can lead to inefficiency by three reasons (MFQ used as an example):

      - Contention for the MFQ lock: Depending on how much computation each thread does before blocking on I/O, the centralized lock (in this case MFQ lock) may become a bottleneck, particularly as there are several cores that might try to acquire this lock. (Operating System, page 329)

      - Chache Coherence Overhead: Each processor would want to fetch the state of the MFQ from the cache of the previous processor that had the lock. Compared to a uniprocessor architecture the scheduling data structure is likely to be loaded in cache, while on a multi-core architecture the data structure will be accessed and modified by different processors in turn. Fetching data from a remote cache would take two to three of magnitude longer than accessing locally cached data. A cache miss while holding the MCQ lock, could even become more of a bottleneck. (Operating Systems, page 330).

      - Limited Cache Reuse: Threads that run on the first available processor, is likely to be assigned to a different one each time they are scheduled. This could result in that the data needed by a thread is unlikely to be cached on the current processor. The worst scenario is when the data is located on a remote cache, resulting in a slowdown as the remote data is fetched into desirable cache (Operating Systems, page 330).

   b. Work stealing is a strategy used in multi-core systems. In a work stealing scheduler, each processor in a computer system has its own queue of tasks to execute. When a processor finishes its task and has run out of work, it looks in the queues of the other processors that are still busy, and “steals” a task waiting in the queue. In effect, work stealing distributes the scheduling work over idle processors. As long as all processors have work to do, no scheduling overhead occurs (Wikipedia, 2023).

Sources:

- N/A. (2023). *What is interprocess communication (IPC)?*. Lenovo. Retrieved from: https://www.lenovo.com/us/en/glossary/ipc/
- Pandey, D. (2023, July 25.). *Inter Process Communication (IPC).* GeeksforGeeks. Retrieved from: https://www.geeksforgeeks.org/inter-process-communication-ipc/
- (User) rushi5758. (2023, September 24.). *Critical Regions in Operating System.* GeeksforGeeks. Retrieved from: https://www.geeksforgeeks.org/critical-regions-in-operating-system/
- Anderson, T. & Dahlin, M. (2014, August 21.). *Operating Systems: Principles & Practice* (2. Edition). Recursive Books.
- Dinescu, M. (2014, October 24.). what's different between the Blocked and Busy Waiting Stackoverflow. Retrieved from: https://stackoverflow.com/questions/26541119/whats-different-between-the-blocked-and-busy-waiting
- GeeksforGeeks. (2023, September 25.). *Resource Allocation Graph (RAG) in Operating System.* GeeksforGeeks. Retrieved from: https://www.geeksforgeeks.org/resource-allocation-graph-rag-in-operating-system/
- GeeksforGeeks. (2023, October 30.). *Deadlock Prevention and Avoidance*. GeeksforGeeks. Retrieved From: https://www.geeksforgeeks.org/deadlock-prevention/
- N/A. *Multi-Level Feedback* Queues. University of San Fransisco. Retrieved from: https://www.cs.usfca.edu/~mmalensek/cs326/schedule/lectures/326-L12.pdf
- N/A. (2023 July 25.). *Work stealing*. Wikipedia. Retrieved from: https://en.wikipedia.org/wiki/Work_stealing