Joachim Grimen Westgaard

Assignment 3

1 Synchronization

1. Inter-process communication

a)

Two processes can communicate with each other without having direct access to each other's memory spaces by utilizing "shared memory", where both processes can access a shared memory space. Another way of inter-process communication is called "message passing", which includes the use of the kernel to retrieve and deliver messages between the processes.

b)

Memory sharing works by initializing an empty memory address space, and then attaching the processes to it before they then can copy messages from and to the processes into the shared memory space. By using this method, we only need to copy the data two times, one when copying from process 1 into the shared memory space, and then once more when copying from the shared memory space into process 2.

c)

When multi-threading, we might encounter issues like deadlocks and race conditions, amongst others. These happen due to the nature of multi-

threading but can be made more likely when we have a lot of communication between different processes.

2.

A critical region is a section of code or a data structure where we only want a single process or thread to operate at any given time. This is to prevent issues like race conditions from occurring. This region is treated as an indivisible unit, where its execution cannot be interrupted.

3.

When a thread is waiting for another thread that is working inside of a critical region, the waiting thread can either busy-wait or be blocked by the Condition Variable. We know that the waiting thread will not be able to access the critical region until the other thread has finished, which might make busy-waiting unnecessary, due to the CPU usage that it "steals" from other threads. If the waiting thread is controlled by the CV rather than constantly polling, the thread would not consume CPU resources and be notified the instant the critical region it wants access to, is available.

4.

A race condition is an event that takes place when the result or behavior of a program relies on which thread gets to execute its code first. We say that there is a race condition because there is a sort of "race" between the operations of the relevant threads in a program. This type of event can take place when we multi-thread in a program where the threads are working on

the same objects, for example manipulating and computing the value of the same integer.

An example from real life can be a situation where two people want to cooperate cooking food. The order of operations when cooking can be detrimental for the outcome of the dish. If we for example start cooking the chicken before marinating it, we might not end up with the same tasty results.

5.

A spinlock is a lock that uses the method "test_and_set" on a value that represents whether the lock that the thread wants to get is busy or free. While this value is set to 1 (BUSY), the thread busy-waits in a tight loop until the lock is free and acquired.

6.

Issues that we encounter with thread synchronization in multi-core architectures:

MCS attempts to address the issue where the lock is usually BUSY instead of FREE. In other words, the issue of when there is high lock contention between the threads. The overhead created when acquiring and releasing locks increase dramatically with the number of threads contending for the lock.

2 Deadlocks

1.

Resource starvation is when a process never gets access to the CPU due to higher priority processes being prioritized. If there is a constant stream of high priority processes being scheduled and executed, the processes with lower priority will indefinitely not have access to the CPU and therefore 'starve'.

A deadlock is a form of starvation, where multiple threads all starve. A deadlock implies starvation, but starvation does not necessarily imply a deadlock. It occurs when there is a cyclical queue of threads where every thread is waiting for another thread. This leads to none of the threads being executed, due to them all patiently waiting for each other. This also causes the program to appear halted/stuck. For a deadlock to be possible, certain conditions must be met:

2.

Bounded resources: there is a limit to how many threads that can be concurrently assigned to and work on the same resource.

No preemption: when the thread has acquired a resource, its ownership cannot be revoked until the thread is done with its work.

Wait while holding: the threads can hold a resource while it is waiting for the availability of another resource.

Circular waiting: there is a set of threads where every thread is waiting for a resource held by another thread.

The bounded resources and the rules around preemption are both inherent properties of the operating system. The operating system can define the maximum number of threads that can concurrently be assigned to and work on the same resource. It also decided when to interrupt a thread or process, with the intention of continuing their work later.

3.

In the case of a thread only being able to hold one resource at a time, we can analyze a graph. We represent the resources and threads as nodes, with directed edges between them, indicating the threads' ownerships over the resources. When this graph is cyclical, we have a deadlock.
Another possible way of detecting deadlocks is a variation of Dijkstra's Banker's algorithm (described by Coffman, Elphick and Shoshani in 1971). In this algorithm, we cannot assess whether some future sequence can result in a deadlock. We can, however, look at the current set of resources, granted requests and pending requests. With this information, we can determine whether it is possible for the current set of requests to eventually be satisfied, assuming we do not receive any more requests and all threads complete. If this is the case, we do not have a deadlock. Otherwise, we have a deadlock.

3 Scheduling

1. Uniprocessor scheduling

a)

In terms of the average response time, FIFO is the optimal scheduling algorithm when we handle tasks of equal length. In FIFO, we have no way of re-ordering the received tasks such that we can optimize the average response time before execution. In addition to this, FIFO creates virtually no overhead when handling these equally long tasks.

b)

The MFQ scheduling attempts to be an efficient and starvation-free scheduling algorithm that creates low overhead. It creates low overhead by its focus on FIFO, where it minimizes the number of preemptions. It is efficient due to its ability to complete shorter tasks quickly as in SJF, and it is starvation-free due to its desire that all tasks should make progress as in RR.

MFQ does not perfectly implement all the advantages of FIFO, SJF and RR. Instead, it is a compromise of all of them, sort of like a "jack-of-all-trades, but master of none".

MFQ is an extension of RR, where it has multiple queues instead of only one. Each queue has a priority level and time quantum. Tasks with higher priority will preempt lower priority ones, while tasks with the same priority is scheduled in the same RR. Higher priority queues have shorter time quanta

than lower levels. Tasks can be moved between the different queue levels, to favor shorter tasks over longer ones. Every time a task uses up its time quantum in the CPU, it drops in priority level. If the task yields to wait for e.g. I/O, it stays at the same priority level.
There are some disadvantages with MFQ, namely its increased overhead and the fact that it is more complex than the other scheduling algorithms.

2. Multi-core scheduling

a)

In the case of a MFQ designed for a uniprocessor applied on a multi-core processor, we might encounter three main issues:

Contention for scheduler spinlock

The MFQ will be subject to lock contention, due to processes and threads running concurrently on different CPUs might want to access the same resources, which are protected by locks, for example mutexes or semaphores. Due to the possibility of tasks being split across different cores, the lock contention might also be higher. This can lead to a significant performance bottleneck, as the scheduler might need to access the same shared resources on individual cores. Thus, we might also increase the overhead of thread synchronization, further hindering the performance.

Cache slowdown

When cores access the MFQ, they might need to modify its state. This leads to the other cores having to erase and update their cached information about

the MFQ's state. It is important that all the cores have the same updated view of shared structures (when they are stored in their own private caches). This further slows down the performance of the CPU.

Limited cache reuse

An MFQ designed for a uniprocessor might not fully utilize the different caches of each core when applied to a multi-processor architecture. When a core stores popular resources for its threads in its cache, we can drastically improve the performance of the CPU. When we with a scheduler designed for a uniprocessor, have the same threads moving between multiple different cores, they lose quick access to their wanted resources. This slows down the performance even more, as that resource now must be fetched from memory (slow compared to cache-access).

b)

When a processor is working on a task with a series of instructions, this task may generate more sequences that could be executed in parallel. While the processor is busy with its original sequence, it puts these new sequences in its queue. Another processor might be done with its workload, and therefore begin searching for work to do. It might see that the other processor has sequences that could be executed at once and can therefore "steal" these sequences from the other processor.

Work-stealing is a form of scheduling strategy for multi-threaded programs running on multi-core processors.