INF-2201-1 23V: Operating system fundamentals

Assignment 3

Amund Strøm

Email: amund.strom@gmail.com

GitHub user: ast283

March 12, 2023

1 Introduction

This report goes into detail about the design and implementation of a preemptive scheduled kernel, some synchronisation primitives for mutual exclusion and a solution to the classic dining philosophers problem. It will also discuss some choices made and what tradeoffs these involve.

2 Technical Details

These are some concepts that is important to have an common understanding of:

- Preemptive scheduling is used when a process switches from the waiting state to ready state or from the running state to ready state. This is mainly dictated by the CPU clock cycles, which suggests that each process is assigned an equal amount of time in the system. A process can either finish its job within the time frame and be terminated, or it will be interrupted before it finishes and be placed back into the ready queue. That process stays in the ready queue until it is finished executing.
- Interrupt request is a signal sent to the CPU which temporally halts the current running process to do something with high priority. The most common interrupt request is sent by an external timer that tells the system to switch to the next process in the ready queue (preemptive scheduling).
- Atomic operation is a section of the code that runs completely independent of any other processes. When an atomic operation is being executed there is no other process that can modify the data used by the operation. In other words, there is not possible to interrupt the operation via external or internal methods.
- Types of synchronization primitives used in this assignment consists of locks, barriers, semaphores and condition variables. All of these serve different purposes, but achieve the same goal that is mutual exclusion.
- Monitors is yet another synchronisation primitive that is used for mutual exclusion. It is a combination of locks and zero or more condition variables. It achieves that only one process can be active in the monitor at a time.
- Dining philosophers is a classic problem that is a metaphor for distributing few resources among many processes. The idea is that there are 5 philosophers and 5 forks, each philosopher can be in the thinking state or the eating state. When a philosopher is *thinking* it requires nothing, but when it wants to *eat* it require 2 forks. Therefore the problem emerges, no more than two philosophers can eat at the same time, since there are only 5 forks and 5 philosophers. A strategy to feed the philosophers fairly is the solution to this problem.

3 Design

The assignment was divided into 6 tasks and 2 extra credits. This section should give an high-level view of how every task was completed but not the extra credits, since they were not implemented.

1. An interrupt handler that invokes the scheduler is performed as an atomic operation, since it is very important to not be interrupted when switching between jobs. Therefore we enter and leave the critical region of the code when starting and ending the operation. The system also requires a signal that tells the hardware that the interrupt is being handled. If this signal is not given, it would not be possible to perform this exact interrupt a second time and the system would be stuck on the same job, until it is either terminated or explicitly calls yield. Before invoking the scheduler the interrupt handler must switch to the kernel stack if possible, since storing the context on the kernel stack grants better protection. Finally invoke the scheduler. Then after invoking the scheduler the handler must switch back to the user stack if possible, to maintain the importance of the kernel stack.

2. The yield system call should work correctly if the interrupt handler is implemented correctly. Even though this is a preemptive system, some jobs still use the yield system call.

3. Use a software interrupt to perform system calls, these system calls should perform correctly even in the presence of the timer interrupt. All of the software interrupts are given as pre-code and works as intended if the interrupt handler is implemented correctly.

4. The synchronization primitive *lock* should continue to work in the presence of preemptive scheduling. A lock is used for mutual exclusion which protects the critical section of a job by preventing more than one job to enter the critical section. A lock supports two operations, *lock acquire* which attempts to take hold of the lock if it is available otherwise it is put on a waiting queue if the lock is not available, and *lock release* which releases the lock and unblocks the first job in the waiting queue if a job is waiting. To make the lock continue to work in the presence of preemptive scheduling, both of these operations must be done atomically, meaning no interruption can occur during the execution of these operations.

5. Add semaphore, condition variable and barrier functionalities to the existing synchronization primitives. All of these synchronization primitives is used for mutual exclusion and supports different amount of operations, each operation must be done atomically to achieve no interruption during the execution of these operations.

- Semaphore uses an integer value to dictate if a job should be allowed to enter the critical section or not. This divides semaphores into two different groups, binary semaphores and counting semaphores. If only a maximum of one job is allowed to enter the critical section, than it is a binary semaphore. If more than one job is allowed, then it is a counting semaphore. A semaphore supports two different operations *wait* and *signal*, that works the same regardless of what kind of semaphore it is. The *wait* operation decrements the integer value, if it falls below zero block the current running job moving it to a waiting queue. The *signal* operation increments the integer value, if it is less or equal to zero unblock the first job in the waiting queue.

- Condition variable waits for a specific condition to be fulfilled before allowing a job to enter the critical section. In this assignment condition variables is used to create a monitor by combining *locks* in the implementation of operations supported by the condition variables. These operations are *wait*, *signal* and *broadcast*. The *wait* operation demands that the caller has *acquired* a lock, it is also the callers responsibility to state the condition that must be fulfilled to enter the critical section. If this condition is not met, the *wait* operation is executed. the *wait* operation is done atomically and releases the lock from the current running and blocks it, moving it to a waiting queue. Eventually this job will be unblocked and moved back into the ready queue, when this happens it will *acquired* a lock and test again if the condition is met.

  The *signal* operation tells the first job to be moved from the waiting queue back into the ready queue. The *broadcast* operation moves the whole waiting queue back into the ready queue.

- Barrier waits for every job to reach a specific point in the code, before letting them all pass this point at the same time. Barrier support one operation that is *wait*, it checks if the specified amount of jobs has called this operation. If the amount is not met, block the current running moving it to a waiting queue,

otherwise if the amount is met, unblock every job in the waiting queue moving them back into the ready queue.

6. Implement a fair solution to the dining philosophers problem, using condition variables instead of semaphores. A "fair" solution is when all philosophers can eat for the same amount of time. To achieve this a philosopher can only eat if none of its neighbours is eating. In other words, while at least one of the neighbour of the current philosopher is eating, that philosopher must wait. This require the philosopher to enter a monitor after it is done thinking so that no other job can be active in the same monitor (same lock). When it has entered the monitor, do the check if neighbours is eating, this is the condition that must be met to be allowed to execute the critical section. When a philosopher is done eating it must signal the neighbours that it is done eating and that they are allowed to eat, and leave the monitor (lock) so other can enter the monitor.

4 Discussion

The tests given as pre-code that checks the functionality of the different synchronization primitives works as intended, this is the only testing that is done on the synchronization primitives in the implementation. The dining philosophers also works as intended, it is a fair solution. But this is not the work of the condition variables, rather the locks. The assignment required the condition variables to be used in an implementation that would be fair, which this solution is, but the condition variables works as an fail-safe rather than dictating which philosopher can eat. The lock ensures that only one philosopher can eat at a time, since if a philosopher all ready acquired the lock, will cause other philosophers that tries to acquire the lock to be put on a waiting queue. This means that there will never be a scenario where the condition "while neighbours is eating, wait" will be true, making the condition variables redundant.

5 Conclusion

This report has given a high-level view of how each task in the assignment was completed, this includes the timer interrupt request, some synchronization primitives and a solution to the dining philosophers problem. Although the solution is not perfect, it is a fair solution nonetheless.

6 Sources

References

[1] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015. Chapter 2.3 - 2.5.

[2] GeeksForGeeks contributors. (13 Dec, 2021). Mutual Exclusion in Synchronization. In GeeksForGeeks. from `https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/`

[3] GeeksForGeeks contributors. (27 May, 2022). Preemptive and Non-Preemptive Scheduling. In GeeksForGeeks, from `https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/`