# University of Tromsø

## INF-2201

## Project 3 - Preemtpive Scheduling

Alexander Einshøj

Department of Computer Science

March 9, 2016

## 1 Introduction

A non-preemptive operating system kernel is transformed into a preemptive kernel. This implies changing the current system call mechanism to an interrupt based mechanism as used in contemporary operating systems. The design and implementation of synchronization primitives such as barriers, semaphores, and condition variables are described. Lastly, the dining philosophers problem is described.

## 2 Technical Background

The possibly most important concept to comprehend when implementing synchronization primitives and avoiding race conditions, is how critical section works.

### 2.1 Critical Section

A critical region or critical section is the part of the program where the shared memory is accessed. When a process wants to do this, it's vital that no more than one process can be in critical region at once, given that it can, and probably will, easily lead to races. For critical regions to work correctly, it's not enough that two processes can't be inside them at once. A process outside of its critical region can't block another process.

### 2.2 Semaphores

Semaphores are manipulative objects with an initial value that determines its behavior. Semaphores can be divided into two groups. Counting semaphores can be at any real value, without any real limits.Semaphores can also be implemented as locks, also called binary semaphores. They have two basic operations as proposed by Dijkstra, up and down. The down operation checks if its value is greater than 0. That being the case, it decrements the value and continues. If it is 0(or less), the process is blocked. The up operation increments the value, then checks whether the value is above 0, before unblocking the first waiting process, allowing for it to use the semaphore. If not, the process just continues.

### 2.3 Condition Variables

A condition variable is a queue that threads can put themselves on when a condition is not what they desire. Another thread can, when it changes state, wake one or more of the awaiting threads allowing the to continue. This is done by signaling on the condition.

### 2.4 Barriers

Barriers come to use in applications that are logically or in another way divided into phases. If the developer wants no process to continue until all phases have reached the point where they are ready to continue. A barrier can be placed at the end of each phase. When a process reaches the phase, it blocks itself until all barriers have reached this stage. This allows for synchronization.

When the last processes reaches this point, all processes are unblocked.

### 2.5 Monitors

A monitor i sa higher-level synchronization primitive that contatins procedures, variables, and data structures that are all grouped together in a package. Processes can call on monitors, but not access its data. Only one process can use a monitor at once, which achieves mutual exclusion. If a calling process finds out that the monitor is already in use, it blocks itself until the monitor is free.

## 3 Design

The synchronization primitives in this case, locks, conditions, semaphores and barriers are all implemented in the same fashion. They all begin by entering critical region, because they need access to shared content that only allows one process to use them at once. This is for mutual exclusion purposes. A pcb is defined in all the implemented structs. Helper functions are made for all the synchronization primitives, allowing for easier viewing and debugging. When the helper functions are done, the process leaves its critical region and lets another process use the primitive.

There is an interrupt triggered every 10ms by a timer, for which an interrupt handler had to be developed. The handler instantly enters its critical region, issues EOI, before it switches to kernel stack, then makes the scheduler move on to the next task.
When it resumes, it switches back to user stack(will only occur for processes), and leaves its critical region.

Locks in this paper have stayed unchanged since the non-preemptive implementation, with the exception of entering its critical region. Everything that's implemented in this case, is done because of synchronization and mutual exclusion purposes.

A thread can get to a point where it needs another thread to continue for it to continue itself. It can then issue a condition wait, where it "lends" the lock it's using to the other thread, then acquires it back when the other thread is done using it.

A condition also has the ability to signal, which simply unblocks the waiting thread first in the queue.

The broadcast ability of conditions unblocks all other waiting queues.

The implemented semaphores are initiated with a value decided by the caller of the function. The semantics of the semaphore functions are as described in technical background.

The implemented solution to dining philosophers currently uses semaphores and isn't fair. The current implementation favors caps ahead of scroll and num. The reason for this is that both caps and scroll grab their right fork before the left fork, but num grabs its left fork first. This means in 2/3 cases, the fork between scroll and num will be attempted to be grabbed first, but only one of them can run at a time(either num or scroll) resulting in caps running alot more, because its forks are more frequently free to use.

A simple switch, making num grab the right fork first, as the others do, results in a fair solution. The problem doing it this way, is the occurance of race conditions, which will happen pretty fast. This is the main reason of having to implement a solution using condition variables, as they have the possibility of solving that particular problem.

### 3.1 Implementation

The assignment is written in C.

## 4 Results & Discussion

The implementation as is, doesn't fulfill all given requirements. All the synchronization primitives are implemented, and work as they should with preemptive scheduling. The interrupt handler works as it should.

The solution of dining philosophers is not implemented due to lack of time. The given implementation is just edited to make it fair, while being aware of the race conditions that do appear. Although the solution for philosophers is not yet implemented, I have a plan for how to do it. The assignment says "Implement a fair solution to the dining philosophers, and use condition variables in your implementation.".

The idea was to make a monitor with conditions and a lock. The lock would tell each philosopher whether they could pick up the forks or not. In order to try and pick up the forks, a philosopher would first need to acquire the lock. Once the forks are picked up, even before it's begun eating, the lock would be given to the next philosopher in queue. This solves the problem of race conditions if implemented correctly. It's also vital that any philosopher is in its critical region every time it tries to grab forks, and that two doesn't try to grab at once.

Statistically, this solution will be fair. There doesn't necessarily need to be an order of who gets to pick up next, and the amount of philosophers should be arbitrary as the solution is flexible. Once a philosopher is done with the forks, it can signal or broadcast(race condition) to let someone else eat.

## 5 Conclusion

Even though the philosopher part of the assignment isn't implemented as wished, I still believe the understanding of the theory behind it is shown convincingly. There are currently no race conditions, except if one chooses to change which fork is grabbed first by num.