INF-2200: Computer architecture and organization

Exam 1 / Assignment 2

Tarek Lein

UiT id: Tle044@uit.no

GitHub users: TrakeLean

March 12, 2023

1 Introduction

In this project we were assigned many small tasks, the first thing that had to be done was change our system kernel from non-preemptive and to preemptive, then we had to implement a new type of interrupt for our processes/threads. After everything was up and running, we got started on some synchronization primitives, like semaphores, barriers and condition variables. The primitives were used in a few different threads, dining philosophers was one of them. we had to make it "fair"

2 Technical Background

When we are dealing with an interrupt based mechanism to change between threads/processes we have to be careful, because what happens if we're interrupted while doing something important? Well that could and would cause a lot of problems that you might not notice straight away, so you'll probably be stuck thinking about why your code crashes randomly. To prevent this we have to tell our code if it's allowed to interrupt or not, this is where "enter_critical" and "leave_critical" comes in. when we call the enter_critical function we're telling the computer that we are in a important part of the code that can't be interrupted. the same logic is applied to leave_critical, which is used to tell the computer that we are done within this important section of code, you are now allowed to interrupt again.

These two functions are used at the beginning of every synchronization primitive, and they would not work without it, because we would meet upon a famous problem called "race conditions" which happens when two or more threads/processes tries to access the same resource at the "same" time.

3 Design

The design parts of these implementations are straight up and simply done, i feel like there aren't really that many ways of implementing these things at this level without be "extra" and over the top. Anyhow, i tackled interrupt_request0 first. Here we have store everything that has been used and that we're going to need for later, then we chose the correct stack and tell the computer that we're dealing with an interrupt ourselves, after that we just switch process/threads, stacks and leave.

For the dining philosophers problem, we had to use conditions. The end goal was to make the eating fair, which they were when we got them implemented with semaphores, but there was a potential deadlock lurking behind the scenes, which is why we had to fix it. The fix was simple, we know when someone is eating because we increment a variable form 0 to 1 when they are. So all we had to do was check if either of the 2 other philosophers were eating, if they were we would have to condition wait, which adds us to a list, where we end up getting a round robin distribution because its first in first out and 2 philosophers are always blocked.

With this implementation there are no way of countering a deadlock and the philosophers won't end up starving :)

4 Implementation

4.1 Lock Acquire

We used the same lock acquire as the last project, the only change that was added on was entering and leaving critical.

4.2 Condition

4.2.1 wait

The condition_wait function used for synchronization between threads or processes. It releases a lock and waits for a condition variable to be signaled. Once signaled, the lock is acquired again.

4.2.2 signal

The condition_signal simply checks if there is anything within the current queue, if there is we unblock the first process/thread, if not we just leave.

4.2.3 broadcast

Condition_broadcast uses the same logic as signal, but instead of only unblocking a single process/thread, we unblock everything in the queue.

4.3 Semaphore

4.3.1 up

This functions job is to increment the semaphore value by one and check if there are any blocked threads/processes that needs to be unblocked. we unblock the first thread/process in the queue if the semaphore value is positive.

4.3.2 down

the down functions job is to decrement the semaphore value by one, then check if the current process/thread needs to be blocked. We block the current thread/process if the semaphore value is negative.

4.4 Barrier

Every time a process/thread hits a barrier, we increment the waiting value by one and block if the waiting value is still below the max number of threads/processes. We then release all the blocked processes/threads when the last one has hit the barrier. This ensures us safe data access progress when it comes to a system based interrupting kernel with multiple threads.

5 Discussion

This project got a whole lot easier when we finally understood the importance of enter and leave critical, and that it was a key part of all the synchronization techniques. The rest of the project went smoothly after that. Something that was noticed while doing this project, was that we didn't really research the given pre-code, we got stuck at barriers trying to understand how to make them wait, when it was as simple as using a predefined "block" function. That's something that we should have already been familiar with, but it didn't really hit before this project.

6 Conclusion

In conclusion, this project was a challenging yet rewarding experience in implementing synchronization primitives and designing an interrupt based kernel with preemptive threads. We learned the importance of using "enter_critical" and "leave_critical" functions to prevent race conditions and ensure safe data access between threads. We also tackled the dining philosophers problem using semaphores and conditions, and solved a potential deadlock issue by implementing a round-robin distribution.
Overall, this project provided valuable insights into the technical background and design considerations of synchronization primitives such as semaphores, barriers, and conditions. It also helped us solidify our understanding of interrupt based kernels and preemptive threads.

7 Sources

References

[1] Teacher