INF-2201: Computer architecture and organization

Exam 1 / Assignment 2

Tarek Lein

UiT id: Tle044@uit.no

GitHub users: TrakeLean

February 20, 2023

1 Introduction

An operating system is made up of many different components and subsystems that work together to provide a wide range of functionality and services, my task was to implement some of these. Mainly we had to implement context switching, which allows our Operating system to simulate threading. To help with the "simulation" we also added locks that allows the Operating system to use multiple threads for a single task.

2 Technical Background

2.1 Context Switching

Context switching is a an important feature of modern operating systems that allows different applications or threads to execute concurrently on a single CPU. When a context switch happens, the operating system saves a running process/thread's state, also knows as context, and then restores the state/context of a previously stopped process/thread, allowing it to resume where it last left off.

2.2 Scheduling policies

There are generally two categories of scheduling policies, Preemptive and Non-preemtive scheduling. I'm only going to talk about Non-preemptive scheduling, since that is what we used for this task. A Non-preemptive

scheduling policy is used in some operating systems where a process/thread can't be interrupted by the operating system until it has completed or been set to a blocked state. The process/thread is allowed to run until it gets interrupted by a yield or an external event like user inputs such as a keyboard press.

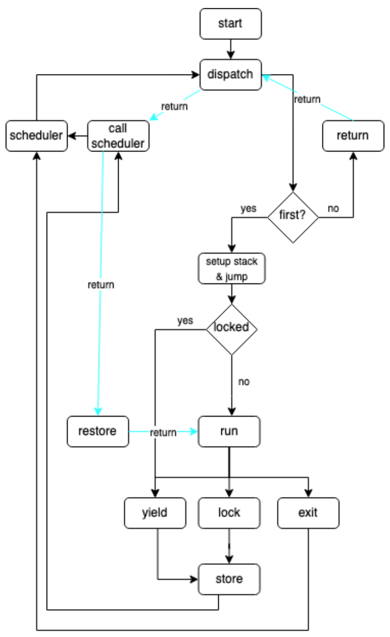
2.3 Round-Robin

This is a simple algorithm to choose who's next. Round-Robin could be seen as a simple list where the item at the top pops out and returns to the bottom, allowing the next item to do the same and so on, effectively creating a never ending loop. This has been used in my Operating system to give the CPU privilege to the next process/thread.

2.4 Locks

Locks are used to prevent issues that may arise from multiple threads accessing shared resources. They enforce a hierarchical order on access to avoid collisions. There are different types of locks, such as spin locks, sleep locks, and adaptive locks. Spin locks use busy waiting, sleep locks put threads to sleep until the lock is available, and adaptive locks change strategy depending on the system's workload.

3 Design



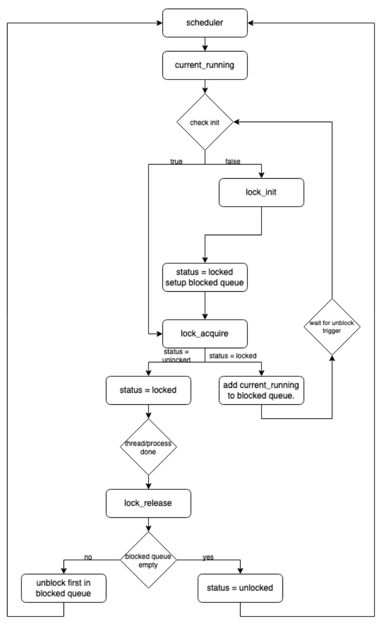
There are multiple methods to choose from when designing an operating system. The first choice that was made is how the flow of data is handled, it's quite tricky to wrap your head around how this works at first, that's why there's a picture included. The path for a thread/process that hasn't been ran yet is pretty straight forward, you're able to see and tell the program to go where you want with functions, but that all goes away when it comes to a process/thread that has already been ran once and is back for round two. You're now going to have to keep track of where the process/thread was and what it called, because it wants to return back, you can look at it as if it's climbing back up a return ladder (drawn with blue lines in the picture). The reason for this is that we want to keep running the process/thread from where we last left of.

Next design choice is how we're going to stop a process/thread when it's running so we can give run the next process/thread that's lined up, in this exam we had to implement Non-preemptive scheduling, which brings us onto another design choice. Which is what sort of algorithm you're going to choose from when it comes to giving a process/thread run time on the CPU, We were told to implement round-robin for this exam, so that's exactly what has been implemented.

Stacks are also things that we have to setup. All the kernel stacks "grow" from 0x20000 and towards 0x10000, while the user stacks "grow" the opposite way, from 0x10000 to 0x20000. You have be careful when choosing what the base and stack pointer is and where you place them because if done wrong it'll overwrite data.

Process control blocks are also needed for each process/thread, this is because we need to keep track of multiple things while everything is running, like where is the current process/threads stack pointer or what state are we currently in? block? ready? have we been ran before or not? this is all stored in a structure called PCB

4 Implementation



There were plenty of ways to implement locks, but a few stuck out as being more optimal than the others. The first implementation tried was taking use of the PCB structs next and previous to create a queue, but this seemed to be difficult because it was hard to keep track of all the pointers while also not losing or overwriting them. This method of creating a queue was then given up and the next one was tried, The next implementation was easier but it used more time and space. This is because we setup an array where PCB's was pushed into it from the back and taken out from the front (LIFO). After taking something out we shifted the queue one to the left so that the next one in the queue got its place in the front, by doing this we could always say that Queue[0] was the front of the queue. When it came to adding items, it was just as simple. You have to write a simple loop that check the queue one by one, and when it hits a place that is NULL you just add the process/thread into that index and break out. Don't forget to set every index in the queue to NULL when initialising it, just to make sure you don't have any garbage values in the new queue

Round-robin was implemented with the PCB structs built in linked-list, this list is linked as a circle where every process/thread within it are ready. We simply remove the process/thread from this queue if it ever gets blocked or exited, The scheduling part of the operating system is made a lot easier by implementing

The process control block was implemented as a struct that could act as a linked-list, it also stored important information like, stack limits, current state, where the process/thread's address was located in memory, the process identifier and more. These multiple PCB's was then added one by one into an array, where they later got their next and previous pointers assigned to their next and previous in the array.

The ready queue is simple, it consist of a pointer named "currently_running" that always points to the process/thread that is currently running. It gets changed to "currently_running- next" when we enter the scheduler, it is allowed to always just become next. The reason for this is because everything else that isn't ready gets removed from the next and previous pointer within this queue. There is really no end or start to this queue since when it was initialised, it's decided that the last pcb's next points to the first one, and the first pcb's previous points to the last one, with this we create a loop, where you technically could interpret the start of the queue as current_running, and the end to be what current_running's previous is. The implementation of this wrap around is quite neatly thought out, We're in a loop that loops as many times as there are threads/processes, let's call that amount for NUM_TOTAL. next up we say that the current pcb's next is pcb[NUM_TOTAL+i+1]%NUM_TOTAL. it would look like this if there were 3 total threads and processes.
pcb[0] = pcb[3+0+1]%3 = pcb[1]
pcb[1] = pcb[3+1+1]%3 = pcb[2]
pcb[2] = pcb[3+2+1]%3 = pcb[0]

Here you can see that the last one wraps around back and points to the first one. The same logic is used for previous, but instead of adding 1 inside the square brackets, you just subtract by one, and the first pcb's previous will point to the last possible pcb.

Context switching overhead was relatively easy to add, all that had to be done was to use the function "get_time" before entering the kernel in yield right before "scheduler_entry" and again after it has returned, the problem that arose was that we've switched the current running process when we get back from "scheduler_entry" which means that we have the start time for the last process and the end time for the process that just got back. To fix this I had store the time before we entered the kernel onto current_running's next, this fixed the problem and behaved expected. The code shows the overhead in clock cycles.

There were three extra credits we could implement, the first one was to add an extra thread, which was easy. all that was done was creating a fitting thread file and tell the makefile to include it.

The same logic was used for the second extra credit where we had to implement another process, but there was a few more things to add inside the makefile here, but everything worked after I followed how process2 was written in.

For the last extra credit, the task was to find out how long a process/thread spends in kernel and user mode, to do this I used the same logic as the context switch overhead then divided it by the MHZ set by process1 which is 500, but there had to be done some converting, since the program is unable to print decimal numbers onto the screen. Therefore the answer was multiplied by 1000000 to show how many microseconds the process/thread had spent.

5 Discussion

The concept of context switching was probably the hardest concept to figure out, understand and implement. The reason for that is lack of assembly x86 knowledge where things had to be done, what order and never really knowing if the context switch was wrong or some other part of the code. Tests for each part of the code would have greatly improved the progress of this exam when looking back. But after many hours of trying and failing, there was built more and more knowledge and understanding for context switching and assembly code. With this came the ease of implementing processes the correct way, where they had to have two stacks. One kernel stack and one user stack, the reason for this is that this operating system only simulates processes going into kernel mode, in reality the processes and threads acted the same. The process ran on the user stack until i wanted to yield and context switch, where the stack pointers got switched from the user stack to the kernel stack right before calling ”kernel_entry_helper” which then yielded or exited. when it came back from a yield it would then switch from the kernel stack and back to the user stack, to then run.

Another part that was more excruciating than expected to implement was locks. I implemented around 20 different versions of a sleep-wait lock, before finally seeing some progress, this was due to not understanding how the locks was supposed to work, this means that I actually managed to create multiple ”working” locks, but they were initiated wrong. The biggest mistake made was that when one process/thread was unblocked, it also unlocked the lock. Which means that the lock is released and every thread within it’s block queue got lost with it, this huge mistake somehow got trough all the lock implementations. I was successful in locking, blocking, unlocking and unblocking after this mistake was taken care of, which is really annoying since i could most likely have implemented the linked-list lock that i had in mind.

6 Conclusion

All in all I’m happy with my implementation of all the features we were asked to add, there is one thing that really bothers me though. That is a bug where if i try to make the code on the schools computer i get an error that tells me i have a memory conflict, this error is raised in the createimage file at line 257 and stops the creation of the image. I’ve asked multiple TA’s for help but none of them know why this is happening and have never seen it before. I’m able to dodge this if I don’t run all the threads and processes when making or if i comment out some code, it seems as if the file gets too big somehow. I’m also able to make the program on my Mac M1 through docker with all the threads and processes enabled or if i comment out that error in the createimage file. Which is why I’m delivering with that commented out, you could try and comment it in and see if it makes on your computer. This bug really annoys me since it seems to be a hardware issue and not anything to do with how I’ve implemented things, and it would sadden me if something out of my control would affect my grading.

I’ve decided to deliver with the docker files and docker code within the makefile included as i was tipped by a TA to do this, just in case that’s the only way you’re able to run the code.

7 Sources

References

[1] https://www.geeksforgeeks.org/process-table-and-process-control-block-pcb/ (02/02/23)

[2] https://www.geeksforgeeks.org/linux-system-call-in-detail/?ref=rp (03/02/23)

[3] https://cs.lmu.edu/ ray/notes/gasexamples/ (6/02/23)

[4] https://pdos.csail.mit.edu/6.828/2005/readings/i386/PUSHA.htm (07/02/23)

[5] https://pdos.csail.mit.edu/6.828/2005/readings/i386/POPA.htm (07/02/23)

[6] https://wiki.osdev.org/Context_Switching (07/02/23)

[7] https://www.youtube.com/watch?v=W8vM3Qn-1aM (09/02/23)

[8] https://wiki.osdev.org/Synchronization_Primitives (18/02/23)