# INF-2201-1 23V: Operating system fundamentals

## Assignment 2

Amund Strøm

Email: amund.strom@gmail.com

GitHub user: ast283

February 19, 2023

## 1 Introduction

This report describes the design and implementation of a multi-programming kernel with a non-preemptive scheduler, using the Round-robin algorithm to schedule the next process. Topics that will be discussed is scheduling, context switching, threads and processes, the stack, and synchronization primitives.

## 2 Technical Details

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

- A process contains steps and information about a program and how it should be executed, it also contains different memory spaces (such as user- and kernel level) and does not share this space with other processes. On the contrary a thread is a lightweight process that exists in one memory space (the kernel level) and may share its space with other similar threads. A thread could be looked at as a subpart of a process. This assignment is only meant to simulate a process operating in two different memory spaces, meaning that user- and kernel level in reality runs on the same flat memory space. The only difference is that a process uses two different memory limits, while a thread uses a single memory limit. And they have unique entry points into the kernel.
- Program Control Block (PCB) generally contains information about a process in execution, but in this assignment it will also be used for threads. A PCB may hold information such as the process ID, process state, memory limits, and in this case a pointer to the next and previous process/task.
- Scheduling involves arranging, removing, and picking the next task to execute based on a particular algorithm. The algorithm used in this assignment is Round-robin which treats each task equal by giving them the same amount of time in the CPU before it picks the next task. Therefore the ready queue (contains task which is ready to execute) must loop around so that if a task did not finish, it will simply be put at the end of the ready queue. This assignment only required looping around the ready queue and not assigning equal time to each task.
- Non-preemptive scheduling is the process of picking the next ready task after a task in the CPU either exited/yielded or is moved from the ready state to the waiting state.
- Context switching is the routine of saving the context of a task after it loses control over the CPU (determined by the scheduler or the task itself) and restoring the context of the next task that is picked by the scheduler. The context of a task contains the value in the registers and the address where it previously stopped its execution.
- Synchronization primitives is a type of software that allows a platform to support process and thread synchronization. This grants the possibility of mutual exclusion which involves protecting the critical section of a thread. The critical section is the part of the thread which uses a ”shared variable”, therefore there must only be a single thread utilizing this variable.

# 3 Design

The assignment was divided into 5 tasks and 3 extra credits. This section should give an high-level view of how every task and extra credit were completed.

Initializing the program control blocks (PCB) is done at the start of the program. This is done by giving each PCB information about each thread and process such as memory limits, process ID, process state, a variable to differentiate processes and threads, the address for the task, and linking each task with a next and previous pointer creating a circle.

In this assignment a process and thread must explicitly call yield or exit to tell the scheduler to pick the next task. Therefore to implement a simple non-preemptive scheduler we must act according to the system calls. When such an event happens, the first order of business is to save the context of the currently running task. This is the start of the context switch and involves pushing the general purpose register and flags onto the stack of the still currently running task. Some threads use floating point registers and therefore these must also be stored by saving them onto a 108 byte large memory space in the PCB for the currently running task[1]. Now most importantly, preserve the value in the base and stack pointers by also saving them on the PCB, so we can restore the context for this task next time it is scheduled.

Lastly call the scheduler to check the state of the still currently running task, if it is either blocked (discussed later) or exited/terminated, then remove it from the ready queue. This ensures that it will not be scheduled in the future. Finally pick the next task to run which is simply the next in the ready queue, and call the dispatcher for handing over the control to the picked task.

Handing over the control is done via two different methods, the picked task is either executed for the first time or not. If it is executed for the first time there is no context to restore therefore it is needed to setup the stack frame using the memory limits stored in the PCB, and jumping to the address of the task which starts the execution. If it is NOT executed for the first time the dispatcher must restore the context of the task. This is done by restoring the values of the base and stack pointers stored in the PCB, and popping the general purpose register and flags from the stack, and restore the floating point registers stored in the PCB. This is the last step of the context switch and the task can finally return to the address where it previously stopped its execution. In both of these cases the dispatcher must also differentiate between processes and threads, since processes uses two memory spaces (user- and kernel level) and threads only uses one (kernel level). When operating with processes it uses the kernel level to store its context and the user level to execute, and must therefore restore the user stack before starting execution/returning to the address where it left of. On the contrary the thread must restore the kernel stack before execution/returning.

Implementing a system call mechanism required only a small jump table that called different functions depending on the system call. A system call is a technique for a user level program to request kernel level services such as yield or exit of a task. This means that only processes in this assignment benefit from this mechanism. Whenever a process makes a system call, simply call the corresponding method used by a thread. This is only meant to simulate a system call since processes and threads operate in the same address space, as mentioned in the technical background.

Implementing two synchronization primitives for mutual exclusion involved creating a lock that would prevent any other tasks to interfere with the critical section of a task that owns the lock. This required an waiting queue where the tasks that would like to operate in the same critical section must be sorted in a queue following the first in first out (FIFO) rule.

Whenever a task requests a lock it must also release the same lock whenever it is done with the critical section. There are two scenarios when a task request a lock, either the lock is available or not. If it is available, update the status of the lock to *taken* and proceed as normal. If it is not available, then this task is placed in the waiting queue following the FIFO rule and removed from the ready queue so it will not be scheduled in the future. On the occasion of when a task releases a lock, there is also two scenarios, either the waiting queue is empty or not. If it is empty, update the status of the lock to *available*. If it is not empty, then remove the first task in the waiting queue and place it back into the ready queue at the end.

Measuring the overhead (time used) of context switches required observing the time (in clock cycles) before a task starts the context switch and observe the time right after the context switch is done and calculate the

[1] source for 108 bytes, https://www.felixcloutier.com/x86/fsave:fnsave

difference between these values. This is achieved whenever a task calls the yield method, since every task must call yield explicitly to start a context switch. In other words what is measured is the clock cycles it takes the program to stop working with a PCB and start working with another PCB.

The first extra credit involved keeping track of user and system time each thread and process used, and print it on the screen. Since threads only operate in the kernel level they will only use system time, while processes will use system time for context switching and user time to execute the task. Therefore each task must record the time after it gets scheduled and before the next task is scheduled. Processes must also record the time when it arrives in the kernel.

The last extra credits involved creating a new thread and process, which only meant creating a new task and allocating new PCB's and memory limits.

This is an illustration that shows the datapath for a running task. The dotted lines at the top represent optional paths a task can take during the execution, while the dotted lines in the middle represent the task being removed from the ready queue. The other lines represent the paths a task must take.

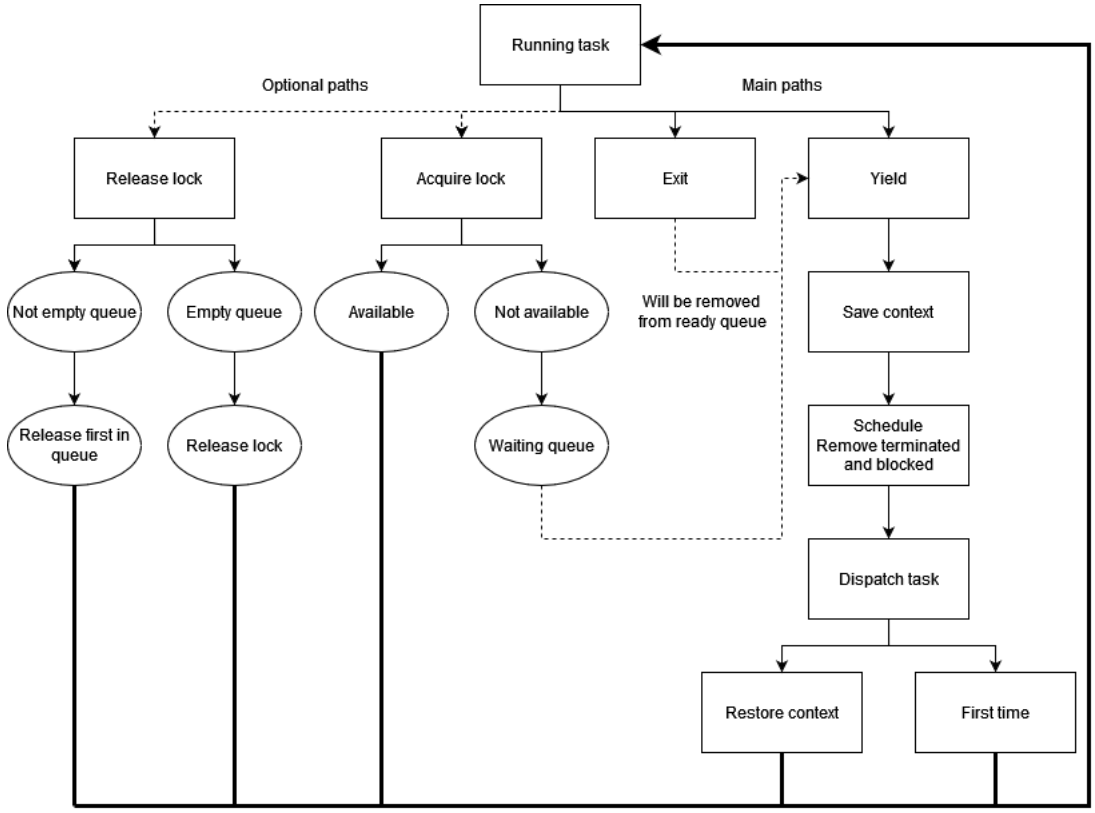


Figure 1: Datapath for a running task

## 4 Results

The results in this assignment is what is produced on the screen while the program runs. It is shown that the locks works as intended according to the simple threads given in the pre-code that checks the functionality of the locks. The same goes for the processes, every process runs without issue. There is also no issue running both threads and processes at the same time or any other known bugs.

The measurement of time used during context switching (described in design section) is only tested on a virtual machine and not real hardware. This may result in some inconsistent data, since measuring works best on real hardware. The results shows that context switching takes a consistent 666 clock cycles to complete.

To reiterate what is measured; it is the amount of clock cycles it takes the system to stop working with a PCB and start working with a new PCB. The result may sound a bit off since there is a small difference in operations done when handling processes and threads. If we ignore this small difference, then it makes sense that the result is constant since the number of operations stays the same each context switch.

This is a table showing the results of user and system time for each thread and process. These results ignore the 10 000 000 clock cycle delay that some tasks utilize. It is also somewhat of an average, since some tasks may flicker between other values and these are just the most common values. The result is also produced on an virtual machine and would be more accurate on real hardware.

| PID | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| System time | 3030 | 2325 | 2324 | 1270 | 1270 | 1270 | 1270 | 1264 | 122 | 122 | 122 |
| User time | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8129 | 1254 | 3852 |

Table 1: Average time used in system and user level for each task

# 5 Discussion

The important thing to grasp in this assignment was how the stack worked and changed according to the calling conventions. Before starting the execution of a task, you must change the values in the *esp* and *ebp* registers (mainly the esp register, but it is good practice to change both) according to the memory limits for this particular task. Since the task uses the stack as it wish, it must therefore have its own memory limits so it wont overwrite other data.

According to the calling conventions whenever a program *calls* another function, it creates a new stack for this function to use as it wish. But this stack contains the return address to where it got called. This logic is applied when a program calls for example yield, since we would like to continue execution right after where we left of. This is why it is very important to store the *esp* and *ebp* registers to save the context of a particular task.

Following this explanation it is generally not a good idea to use the *jmp* command since it doesn't create a new stack, so there is no way to get the return address. But in the case of between context switching, we don't want to pollute the memory with unnecessary stacks and there is no need to know the return address. Therefore it may be a good idea (but not necessary) to use the *jmp* command between context switching.

# 6 Conclusion

This report has shown how to implement a very primitive kernel with a non-preemptive scheduler. The results show that there is no problem running both processes and threads, there is no issues using locks and there is no other known bugs.

# 7 Sources

# References

[1] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015. Chapters 2.4, 3.2-3.4.

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual. (April 2022). FSAVE/FNSAVE — Store x87 FPU State from `https://www.felixcloutier.com/x86/fsave:fnsave`

[3] University of Virginia Computer Science. (08 March, 2022). x86 Assembly Guide from `https://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

[4] Tutorials point contributors. (18 February, 2023). Operating System - Process Scheduling from `https://www.tutorialspoint.com/operating_system/os_process_scheduling.html`