inf-2202-ob2-exam

Sigurd Uhre

February 2022

1 Introduction

The goal in this task is to implement a simple multiprogramming kernel with a non-preemptive scheduler. Processes and threads share a flat address space and the kernel must support both implementations. The processes is located in a separate object file for protection and to prevent them from easily alter data in the kernel and other processes. Meanwhile threads and the kernel are linked together in a single object file. Instead of using real mode, the kernel will use the protected mode of the x86 processors and the separation between user-mode and kernel-mode will only be logical, giving both modes the highest privilege level of ring zero (0). [2]. In this report processes and threads are referred to as tasks when including both types.

2 Technical Background

Processes
Including all its values like registers, program counter and variables, a process is simply a program in execution that shares its CPU time with other task running. The concept that the CPU switch back and forth between different task (like processes) is called multiprogramming. Although looking at each process as an independent entity, the processes often have to communicate with each other. Therefore, process properties (like states) are useful tools for solving logistics when processes are running, blocked or queued. [1]

Thread
Threads can normally be explained as a smaller and simpler process within a process. In traditional OS, threads work together with processes to solve small task's and applications so that the main process don't need to be blocked and unblocked in order to complete them. They can communicate with threads from other processes (since they have shared address space), they are faster to build and destroy (lightweight) and they increase performance. In this task they will function as separate entities and not parallel with the processes, their shared address space is the reason we need to implement locks in this program. [1]

Scheduler
The scheduler responsibility is to navigate and keep track of the logistic for the processes properties and CPU-time. In traditional OS, schedulers decide if a process is out of time, and its time to let another process have some CPU-time. In *Preemptive scheduling* the processes that the CPU is running is given a time limit, regardless of whether the process is complete or not it will interrupt the current process if it extends its time limit. In *Non-preemptive Scheduling* the CPU is allocated to the process until the process is blocked or terminated. This type of scheduling waits until the process i put in waiting queue or finished before it switches task. [3]

Process control block
The PCB is a one entry per process table containing properties to the specific process. Properties like state, stack pointer, identification and scheduling information among others, are information that needs to be saved when the task is switched from one state to another (like from running to blocked). The reason for this is so that the process pick up where it left off. [1]

3 Design

The overview design in this implementation is based around switching task states and organizations of queues. The different tasks address is hard-coded into a array where the start is connected to the end. The first task (process 1) in the array is set as the current running and since the process state is ”first time”, the scheduler will call the dispatch function to run it. The current running task will now call yield, block or exit function (depending on the task) and the scheduler will give another task (next i ready queue) some CPU-time.

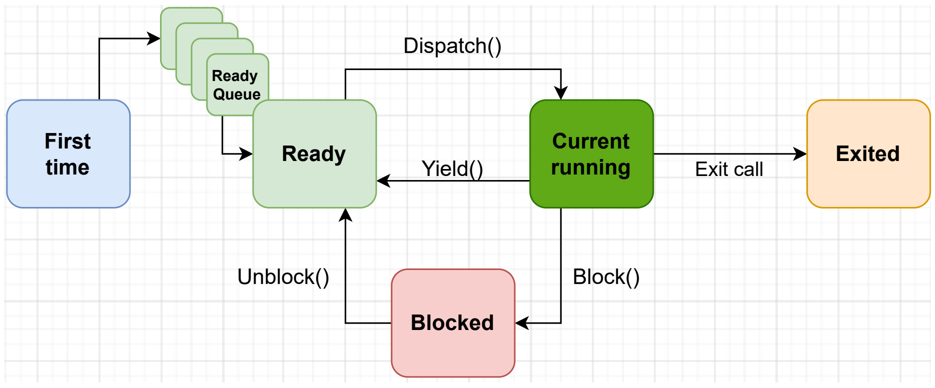


Figure 1: Architectural design

As mentioned earlier, the scheduler is a non-preemptive scheduler and does not interfere with how long the tasks have in the CPU, the tasks themselves will call their next system call. If a running task yields, the scheduler is called to run the ’next’ task in ready queue. This type of queuing system is called round-robin and is based on a circular queuing system. The task are only taken out of the queue if they are blocked or exited, which we will take a closer look at in implementation. In figure 1 above, the different functions and system calls that a task ”runs through” from one state to another is illustrated.

4 Implementation

The process control block need structural pointers in both directions (next and previous) for connecting it in the ready queue. Pointer to next pcb in waiting queue and address memory location is also implemented in the process control block structure, along with identification and state. Both user-mode stack and kernel-mode stack is given a numeric type that guarantees 32 bits, and both stacks are set up so they does not extend the given stack area.

When a running task is taken out of the circular queue system (like when blocked), the remaining task needs to ”fill the gap” left behind. Figure 2, below illustrates an example on how the scheduler function connects the pointers for the addresses next to the ”gap” left behind from the blocked or exited task. When this is done both ways, the ready queue is now fully connected with one less task in the queue system.

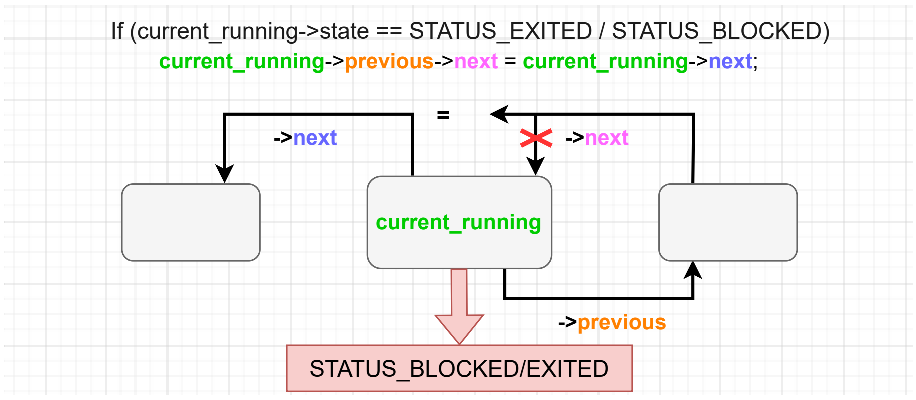

If (current_running->state == STATUS_EXITED / STATUS_BLOCKED)
current_running->previous->next = current_running->next;
=
->next
->next
current_running
->previous
STATUS_BLOCKED/EXITED

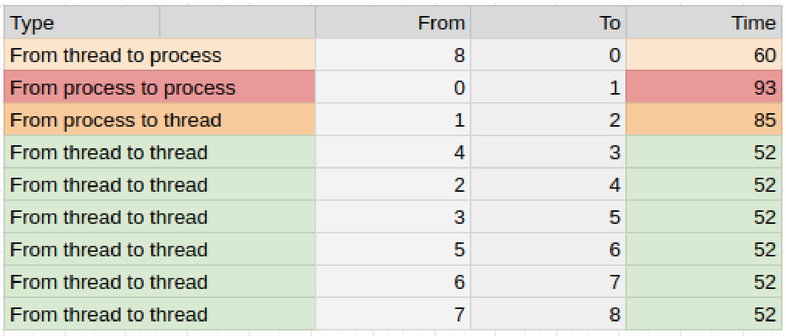
Figure 2: Example of scheduler architecture.

When a task is blocked, it depends on external situations (such as input) to be able to continue. When a task calls yield, on the other hand, it is because it gives other tasks access to the CPU, even though it could actually continue. When a task is blocked it is put in a waiting queue which is made up of a single linked list. The first process in the waiting queue (queue head) is unblocked by changing state and setting this task as the current running along with its pointers in both directions.

5 Results

By running the bochs emulator display we can see that the implementation works like it is supposed to. The different processes and tasks runs until they are completed (if they are completed) and appears to be exiting successfully. The processes calculations displayed are correct and there is a noticeable difference from when all tasks are active and when several tasks have been terminated. The data on the right part of the screen display the context switch time from one task to another.

The pid-number shows what task the context switch is switching from and to, these transitions will vary as some tasks are terminated along the way. The time is displayed at the rightmost part of the screen and show how much time the task switch costed. The first two numbers represent the address of the task and the task for which it is being replaced, followed by the time for this exact context switch. The figure below shows the time results and what type of switch that gave the results.



Type	From	To	Time
From thread to process	8	0	60
From process to process	0	1	93
From process to thread	1	2	85
From thread to thread	4	3	52
From thread to thread	2	4	52
From thread to thread	3	5	52
From thread to thread	5	6	52
From thread to thread	6	7	52
From thread to thread	7	8	52

Figure 3: Results.

6 Discussion

The switch from thread-to-thread is the least time consuming switch, followed by the thread-to-process switch which is slower. Process-to-thread is the third slowest shift, and the slowest switch is from process-to-process.

An interesting observation is that thread to process switching is significantly less time-consuming compared to process to thread switching, which is closer to a process to process switching in time-consumption.

The various task PCB address is hard-coded and customized to fit this work. Assignment of new processes or threads must be entered manually in this work and adjusted accordingly. Both the processes and threads work as they should and the relevant threads give system calls that indicate case done (exited) and case ok (passed). The calculations that the tasks make and the switch between them seem to work correctly. It is important to note that the data shown in the results may vary from machine to machine, as well as whether it is run through bochs or directly on the computer.

7 Conclusion

This work implemented an simple multiprogramming kernel with a non-preemptive schedulers. Round-robin system was selected as the ready queue implementation and single linked list was selected as the waiting queue implementation. The kernel supports both thread- and process-implementation, it distinguishes between processes and threads when it initializes the process control block and allocating stacks. The scheduler controls the task state and is responsible for running, blocking and exiting the ready queue tasks. The implementation seems to work successfully and both processes and threads seem to perform their tasks correctly.

The context switch timer measurements showed that process-to-process shifts were most time consuming and thread-to-thread was least time consuming. The same results showed that thread-to-process was less time consuming than process-to-thread, and both were somewhere between the previously mentioned shifts.

8 References:

1. Tanenbaum A, Bos H. Modern operating systems. 4th ed. Amsterdam, The Netherlands: Pearson Education INC. 2015. Page 85-92.

2. [Internet]. 2022 [cited 17 February 2022]. Available from: https://github.com/uit-inf-2201-s22/project-2-siguhr/tree/main/assignment.info

3. Preemptive and Non-Preemptive Scheduling - GeeksforGeeks [Internet]. GeeksforGeeks. 2022 [cited 17 February 2022]. Available from: https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/