INF-2201, Exam
Non-Preemptive Scheduler

Name: Magnus Dahl-Hansen
GitHub username: MagnusDH
Email: mda105@uit.no

Introduction:

This report describes the implementation of a non-preemptive scheduler for a multi-programming kernel supporting both processes and threads.

Technical background:

Those reading this report should have a general idea of what a non-preemptive scheduler, process control block and lock is, and also knowing the difference between processes and threads for this assignment.

A scheduler is a mechanism that decides which of the processes in memory that the central processing unit (CPU) should execute. Non-preemptive scheduling is used in this case to switch between execution so that multiple processes and threads may run simultaneously. A non-preemptive scheduler works by assigning the CPU to a process or thread and let it execute until it gets terminated or reaches a waiting state. When this happens the scheduler will allocate CPU time to another process/thread.

A process control block (PCB) is a data structure that contains information and data related to a process. In this assignment the PCB also contains information about a thread. The information a PCB contains may vary from different operating systems and computers, but store in general the state of a process, some identification, its memory and registers.

Locks (also know as mutexes) is a concept that is used to solve problems that can occur during the execution of multiple threads. Different threads can have access to and alter the same section of code during execution which can cause inconsistency in the shared data. A lock in this case acts as a gate keeper that only allows one thread access to the shared section of code and have the ability to block other threads trying to access the section of code in question. Other threads trying to access the locked code is placed in a blocked queue and is only released when the lock is open again.

In computer programming a thread is a line of code that can be executed by the CPU and a process is an overhead program which can contain several threads. In this assignment however both processes and threads are executable programs, each performing different tasks. The main difference is that threads run in kernel space while processes can operate in both user space and kernel space.

Design & Implementation:

In order to run multiple processes and threads each process and thread have to be assigned a PCB. The PCB in this implementation is a data structure containing variables that points to other PCB's, identifies the a thread/process, give information about its state, define its type (informing if it is assigned to a process or thread) and variables that points to where its kernel stack and potentially user stack is located. Since the PCB's are linked together via next and previous pointers they form a list. The first two PCB's in the list are assigned to have properties for a process and the last seven are identified as thread PCB's. Before calling the scheduler the last PCB in the list is set to be the current running PCB.

After setting up the PCB's the scheduler is called. The scheduler will set the next PCB in the list as the current running PCB and allow it to execute. The scheduler will also perform a status check on the current running PCB to see if it is in a "exited" state, meaning it has ended its execution completely. If so this PCB will not be scheduled to run and the scheduler will loop through the list of PCB's and select one that is not in a exited state before it calls the dispatch function.

The responsibility of the dispatch function is to prepare a PCB to either start execution or to continue execution. The first step in doing so is to checking what type the scheduled PCB is because a thread PCB can be in a blocked state and therefore have to be processed different than a process PCB. However, in both cases dispatch will continue to check if the PCB has been run before or if this is the first time it is being run. If it is the first time, the current running PCB will be loaded into the %eax register using inline assembly allowing its kernel stack-pointer to be loaded into the %esp register. After doing so a jump to its start address where the process/thread is located is performed and allowing this code to take control.
In a case where the scheduled PCB has run before, dispatch will first move the PCB to the %eax register, move its kernel stack-pointer to %esp, restore the floating point registers, pop all arguments from its stack back to its previous state and return to where it was before, using inline assembly.

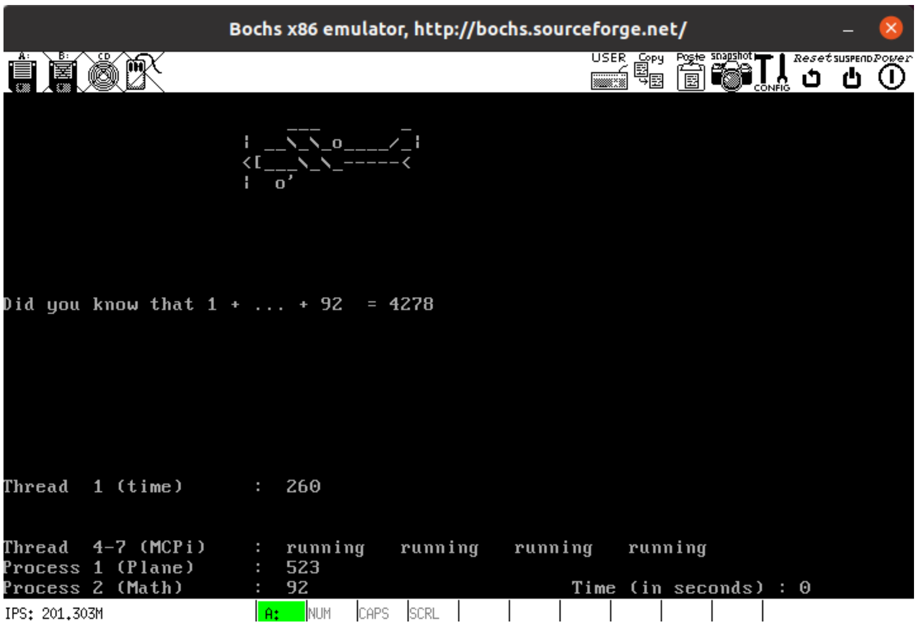
When a process or thread runs they will either do a yield call, try to acquire a lock or perform an exit call. In the case of a process executing a yield call, it will do a system yield call that provides services from the kernel. From here the current stack pointer value, all the general purpose registers and floating point registers are pushed onto the given stack for the process to save its state and the scheduler is called once again to schedule the next PCB. In the case of a thread calling yield it will not have to perform a system call like a process would to gain services from the kernel since they operate in this space already. From here however, the process of storing stack pointer, all general purpose registers and floating pointer register are the same as for a process.

During execution of a thread, it has the opportunity to try and acquire a lock. The lock is a data structure that by default is unlocked, but becomes locked when a thread acquires it. The lock data structure also contains a PCB type that is used to create a list if a PCB tries to acquire a lock that is already locked. If a PCB is denied access to a locked lock it will be removed from the list of running PCB's, its state will be set to "blocked" and it will be put into a list of blocked PCB's. From here it can not be unblocked until the thread that acquired the lock in the first place releases the lock. When the lock is released, the first PCB in the blocked list will be put back into the list of running PCB’s and granted access to the lock.

If a thread performs an exit call its state will be set to “exited” and it will be removed from the list of running PCB’s so that it can not be scheduled in the future. The scheduler is then called to schedule another PCB to execute.

Evaluation & Discussion:

The implemented code is able to run all processes and 5 out of 7 threads simultaneously. However it will only run process1, process2 and the clock thread correctly. As shown in figure 1, thread 4-7 runs but will only print the “running” message and not continue to the “i%10: 0” message as they should and thread2 and thread3 does not execute at all.



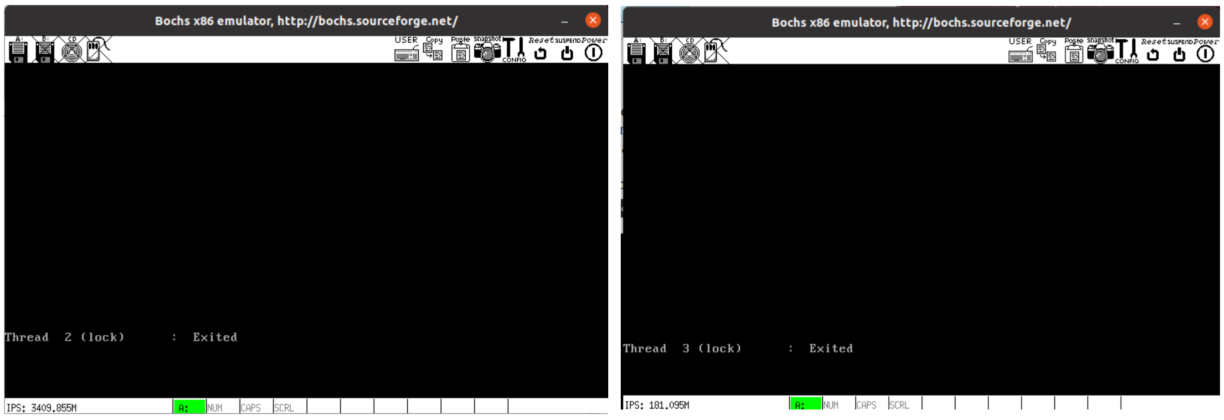
```
                       ___        _
                      |  __\_\_o____/_|
                      <[___\_\_------<
                      |   o'




Did you know that 1 + ... + 92  = 4278








Thread  1 (time)       :  260


Thread  4-7 (MCPi)     :  running   running   running   running
Process 1 (Plane)      :  523
Process 2 (Math)       :  92                        Time (in seconds) : 0
```

Figure 1. ***Running the Bochs emulator with all processes&threads.***

Running single processes and threads on the bochs emulator causes no problems and they all execute as expected according to their implementation. Also knowing that process1, process2 and the clock thread does not make use of any locks gives reason to think that the implementation of the locks causes synchronization problems. All threads that uses locks either executes incorrectly or does not execute at all. Running single threads at a time even though they make use of locks will execute correctly because they will never enter a “blocked” state. Which in turn also gives reason to think that the way threads are removed from the running list of PCB’s and put into a blocked list may a problem. However, the list implementation have been tested by printing the lists to the bochs emulator and they appear to be correct.
During the call to release a lock the “unblock” function does not seem to execute for some reason. This is tested by using a print statement inside the unblock function, but it is never run. Due to time restraints this problem have not been solved.

Another case why the threads does not run simultaneously may be caused by the scheduler or the exit function. The scheduler in this implementation will not schedule a process or thread if they are in a “exited” state, which they are set to during a call to the exit function. Process 1&2 and the clock thread will not call the exit function, which may be the reason they run correctly while the others does not.

Running thread2 and thread3 separately will count to 100 twice and print the “exited” message. However for some unknown reason the Bochs emulator will reset almost immediately after thread3 have finished executing and printed the “exited” message. These two threads are almost identical in execution, so why this is a problem is strange.


Bochs x86 emulator, http://bochs.sourceforge.net/
IPS: 3409.855M
Bochs x86 emulator, http://bochs.sourceforge.net/
IPS: 181.095M

```
Thread  2 (lock)       :  Exited
```

```
Thread  3 (lock)       :  Exited
```

Figure 2. ***Running thread2 and thread3 separately.***

To measure context switch time the idea was to start measuring time using the get_timer function when a process/thread calls yield/exit and measure the time again when another thread/process starts executing. However this was difficult to implement due to how the dispatch function is implemented. The dispatch function uses the assembly statement “ret” to start execution of processes/threads and since the get_timer is a C function it is difficult to measure time. Get_timer successfully returns the time when yield is called, but can not be called after the “ret” statement when starting the execution of a thread/process. Using less inline assembly in the dispatch function and rather writing the assembly code in entry.s may have made it easier to measure context switch time.

The implemented code is run and tested on a HP laptop with the following specs: AMD Ryzen 5 2500U processor, Linux Operating System, 8GB RAM and using the Bochs emulator.

Conclusion:

This report described the design and implementation of a non-preemptive scheduler for a multi-programming kernel and proceeded to evaluate and discuss topics and problems with the implementation. For the non-preemptive scheduler to work, process control blocks, a scheduler, locks and linked lists have been implemented. The code does not run every process and thread as expected and possible reasons why have been discussed and referred to.