Non-Preemptive Scheduling

Alexander Einshoj

February 24, 2016

Introduction

In this paper, the design and implementation of a simple multiprogramming kernel with a non-preemptive scheduler is described. All processes, threads and the kernel share a flat address space in this paper.

Requirements

- Initialize processes and threads, eg. allocatin stacks and setting up process control blocks(PCB).

- Implement a simple non-preemptive scheduler such that every time a process or thread calls yield or exit, the scheduler picks the next thread and schedules it.

- Implement a simple system call mechanism. since all code runs in the same address space and privilege mode, the system call mechanism can be a simple jump table. But the basic single-entry point system call framework should be in place. You are required to implement two system calls: yield and exit.

- Implement two synchronization primitives for mutual exclusion: lock acquire and lock release. Their semantics should be identical to that described in Birrell's paper. To implement mutual exclusion, the scheduler needs to support blocking and unblocking of threads. This should be implemented as the two functions block and unblock, but they are not system calls and should only be called from within the Kernel.

- Measure the overhead of context switches.

Technical Background

Scheduler

A scheduling algorithm is the algorithm which dictates how much CPU time is allocated to Processes and Threads. The goal of any scheduling algorithm is to fulfill a number of criteria[1] :

- No task must be starved of resources - all tasks must get their chance at CPU time;
- If using priorities, a low-priority task must not hold up a high-priority task;
- The scheduler must scale well with a growing number of tasks, ideally being O(1). This has been done, for example, in the Linux kernel.

What's described above is the typical idea of a scheduling algorithm in the real world, which is not the same for this assignment. This will be further discussed later on.

Round-Robin

Round Robin is a simple scheduling algorithm, where only a single queue of processes is used. When the timer is up for the current running process, the scheduler switches to the next process, and the currently running process is put at the back of the queue.

This assignment is to implement a non-preemptive scheduler, which is not what Round Robin in reality is. Round Robin is a preemptive scheduler which bases itself on assigning a time slice or "quantum" to each process. The quantum is equal for all processes.

In this assignment, Round Robin is non-preemptive because the scheduling is not based on quantums, but rather by the processes and threads explicitly performing a system call, yield or exit.

Processes & Threads

Processes and threads behave differently in this assignment than they would in a real OS. All processes, threads and the kernel share a flat address space. The main difference between processes and threads in this project is that a process resides in a separate object file so that it cannot easily address data in another process (and the kernel), whereas threads are linked together with the kernel in a single object file.

In this project the separation between kernel mode and user mode is only logical, everything runs in the processors highest privilege level (ring 0). [3]

Context switch

A context is a virtual address space, and what's contained in it, such as data.

A context switch may occur due to different reasons.
The application could have been preempted (not the case in this project), a kernel function could have been called, or it has yielded its time slice.[2]

System calls

In this project, the system calls implemented were yield and exit. In both of the cases, when called by a thread or process, the scheduler picks the next thread or process to run. System calls make the application request a given service from the kernel.

Process Control Block

The PCB is a data structure in the kernel containing the information needed to manage a process. Given this, it should contain the stacks of the processes, their identification number, a relation to the other processes and their state. In a real OS, they would also need to know if they had any child processes.

Design

There were some defining design matters to handle in this project, such as choice of scheduling algorithm. The assignment papers suggests Round Robin as the algorithm, which is probably due to its simple implementation. The author has also chosen this algorithm, as it is what could be described as "the natural choice" when doing this with no prior experience in the particular subject.

When handling the lock queue, what is supposed to be the end of the queue is predefined as (*q), although the author has chosen to make this the head of the queue. This makes no practical difference, but seemed more logical and resulted in a more natural implementation.

Another element in the choice of design in this project, is how to remove a process from the ready queue to then insert it in the lock queue, and vice versa. Also implicated in this, is how the lock queue itself is built. It could be built circular so the first item in the queue has as previous the last item in the queue, or eg in a simple linked list, like the author has done. When removing the item from the ready queue, the elements that links to the current item need to redirect their pointers. When the item is then inserted into the lock queue, the queue is looped through and the item placed at the back.

Moving on to where the pcb is initiated, the loop that initializes the pcb's is hardcoded to follow the order of the array. That is, all the threads are given the lowest values, and also initialized first, whilst the processes are the last ones to be initialized. While this may not be the most flexible solution, it does provide the functionality needed for this project.

The stack initialization creates a user stack and/or a kernel stack dependent on whether it's a thread or a process.

The complexity of the lock functions are really to be found in the block & unblock functions, where the queue operations occur. A block occurs if a process tries to acquire the lock and it's locked already.

In this project, a context switch happens when yield, exit or block is called upon. As opposed to preemptive scheduling, it doesn't occur due to eg expiration of the quantum or such. When a context switch is executed, the current running process/thread is saved, and then restored later on where

it left off.

Implementation

The project was written in C, x86 Assembly and inline assembly.

Discussion

As proven by the code itself and this paper, all the requirements for this assignment were not fulfilled. What lacked to be done, was measuring the overhead of context switches. Although this is the case, I have some thoughts about how it should have been executed.

There are four different measures to be done, which are; thread to process, thread to thread, process to thread and process to process. I would have begun the implementation by running only two threads and measuring the time needed for a context switch between those, and then continued by adding processes to the measuring. What's needed to know in order to complete the implementation, is the flow in the program, what happens during a context switch.

The three cases in which this can happen are block, yield and exit. When keeping that in mind, knowing where to measure the time isn't too complicated. The hard part is displaying the information and calculating the times correctly, as the timer function included in the code returns a 64 bit unsigned integer. An idea is to constantly calculate the average time used for each kind of context switch, thereby updating the average time value.

As partially mentioned above, the initialization of the pcb struct could have been made more flexible, and shouldn't on a general basis be hardcoded to satisfy the specific order at hand.

Conclusion

The current implementation creates the expected results for the mcpi, and the rest of the threads also work as they should. Everything that has been

done seem to be running as it should.

References

[1] http://wiki.osdev.org/Scheduling_Algorithms (Visited on 20/2/16)
[2] http://wiki.osdev.org/Context_Switching (Visited on 20/2/2016)
[3] The assignment paper (Handed out at beginning of project)