Process Scheduling

Andreas van Cranenburgh (0440949)
Job Jonkergouw (5613949)

October 2008

Contents

1 Simulation **1**

2 The Operating System **2**

3 Scheduling algorithms **2**
3.1 First Fit First Serve 2
3.2 First Come First Serve 3
3.3 Shortest Job First 3
3.4 Mixed mode 3
3.5 Monte Carlo 3

4 Optional features **3**
4.1 Queueing strategy 3
4.2 Time slices 4
4.3 Priorities 4

5 Results **4**

6 Discussion **4**

Abstract

Scheduling is an important part in the design of operating systems. At any time many applications are running on a computer with many of them needing access to system recourses such as memory or I/O services. Because a computer's resources are limited not all applications can have access to their resources at the same time. A system to assign the available resources to the applications is needed. This is called scheduling. We will use a simulated operating system to test different types of scheduling. This paper describes an implementation of different methods of scheduling and measures their performance.

1 Simulation

Instead of testing the scheduling directly on a real operating system we used a simulator. This has the advantage that we have to worry less about the technical side of the operating system but instead we can focus on the scheduling alone. This simulator was provided as the object-file `simul.o`,

which is the simulator itself, the random-number generator `amt19937ar.o` and `mem_alloc.o` which is used to simulate the memory management. Those provided files were closed sources to prevent us from cheating the measurements of the scheduling; we could have adjusted the simulator to create bugged processes which could be handled very easy by some schedulers. Moreover, to give be able to edit only a few files makes it easier to focus our attention in the right way. However, there was a downside of the closed sources: we didn't get access to the main() function so direction of input and output was difficult to handle. Instead, we needed write additional scripts to manage the in- and output.

The provided simulator gives detailed output of the performance. To simplify matters we have focussed on only two variables of performance: turn around time and the number of finished processes. Turn around time is the total time from submission of a process until time of completion.

2 The Operating System

The scheduler itself was written in the files `schedule.c` and `schedule.h`, which were provided. The simulated operating system acts according to the following queues: `new_proc`, `ready_proc`, `io_proc` and `defunct_proc`. These queues are pointers to the their first element with the queue itself being a linked list. Processes move from queue to queue according to this diagram:

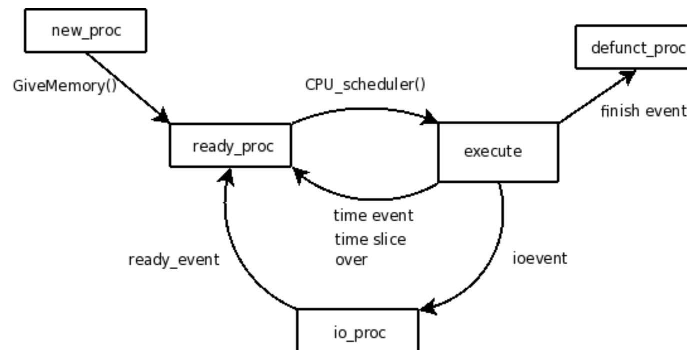

Figure 1: State transitions of processes

The functions for enqueueing and dequeueing are used to handle the switching of a process to one queue to another and are defined in in `schedule.c`.

3 Scheduling algorithms

Our scheduler has different modes, which can be chosen at run-time. This makes it possible to combine different parameters as needed.

3.1 First Fit First Serve

The first algorithm allocates memory to processes and skips over processes which do not fit in memory. The whole queue is processed in this fashion, after which they will be executed.

The obvious downside to this is that processes requesting large amounts of memory will be skipped, even though it should be their turn to run. This can lead to starvation when a lots of memory are requested.

3.2 First Come First Serve

This option is similar to the previous one but instead of skipping over processes that do not fit in memory it will defer further allocation and execute those processes which have already been admitted.

3.3 Shortest Job First

Another method is to make the bias for small processes explicit. This algorithm actively searches for the process requesting the least amount of memory in order to allocate as many processes as possible.

A small modification was made, however, when we discovered that visiting each process in the queue only once yielded much better results than restarting at the front of the queue after dequeueing the smallest process (so even though the second smallest process might be before the smallest procces, with this modification only processes after the smallest proces will be evaluated).

3.4 Mixed mode

As an experiment an option was added to use the preceeding three algorithms repeatedly, in turn (using increment modulus 3).

3.5 Monte Carlo

Monte Carlo scheduling (also known as lottery scheduling) simply picks a random process from the queue of waiting processes until a process is encounterd which does not fit in memory. The downside to this is that it is obviously unfair: processes which have been waiting for longer might not get a turn before others which have just arrived in the queue. On the other hand it solves the problem of starvation, every process will have an equal oppurtunity.

4 Optional features

The following three features can be enabled or disabled with any of the preceeding scheduling algorithms.

4.1 Queueing strategy

Processes which have been granted memory can be queued either at the back or at the front of the ready queue. Normally they should be queued at the back, because the other processes in the queue have been waiting longer, but queueing at the front can be used as a trick to achieve very low turn-around times. This will have detrimental results if the queue is large (unfairness will rise exponentially).

4.2 Time slices

When enabled processes will get a specific time slice, based on the number of processes (this is to ensure that scheduling has a constant frequency, irrespective of number of processes). The formula for this is `100 / number_of_processes`, in "units of simulation" (the actual units were not documented in the simulator, so this number was arrived at by educated guessing). The advantage of this is that processes will not hog the CPU for long amounts of time, and this benefits the responsiveness of the system. The downside is that many more context switches will be generated, especially with a large number of concurrent processes.

4.3 Priorities

When this feature is enabled processes will be assigned a priority depending on their I/O behavior. In order to determine the priority of a process two timers are maintained in an extra struct in the process control block, the first for cumulative CPU time, the other for cumulative I/O time. When a process starts to do I/O, its CPU timer is stopped and its I/O time starts; and vice versa when its I/O is finished.

Processes that spend a relatively long time doing I/O will be granted a higher priority than other processes (the actual condition is `cputime > 1000 * iotime`). The rationale for this is that processes waiting a lot for I/O should be compensated by not having to wait for the CPU, as well as the heuristic that I/O bound processes will generally not need a lot of CPU power.

5 Results

	FCFS	FFFS	SJF	FFFS+slices	SJF+prio	SJF+frontq+prio
processes	893	1003	934	989	929	969
turn-around	4770	1584	1498	889	823	788

Table 1: Some measurements. Processes is the number of finished processes (higher is better). Turn-around is the total turn around time (lower is better)

See the graphs on the following pages.

6 Discussion

We have compared the different scheduling methods by their performance as function of the CPU load. We considered the normal methods such as FCFS, FFFS, SJF, Mixed and Monte Carlo but also two configurations we think give good results. We see that by far FCFS is almost always the slowerst with Monte Carlo coming close, but that is not something we can trust giving its random nature.

When comparing the use of round robin with FFFS we see that time slices only gain the upper hand under a high CPU load, which is as expected because round robin is only useful with CPU heavy processes or for time sharing systems. A notable winner is SJF at the higher CPU load with priority checking giving it an even better result. This may indicate a correlation between memory size and CPU load of a process.

From the graphs it is clear that as the number of processes increases, only Shortest Job First, Mixed mode and Monte Carlo degrade gracefully. The other scheduling algorithms show faster growth of turn around time.

If the CPU load is increased instead only Shortest Job First shows promising performance, the other algorithms break down at high loads. The clear winner is Shortest Job First combined with priorities.

Enabling priorities and time slices yields the best performance, as should be expected.

We have also compared the effect of the Priority and Time-slice settings on SJF in particular (since that algorithm seemed to perform the best). We can see that Time slices are notably slower under lower loads. Which is as expected because Round-Robin only has a benefit at high CPU loads, otherwise the context switching time will work as a big disadvantage. Moreover, it seems that the enabling priorities gives best performance even at the highest CPU load.

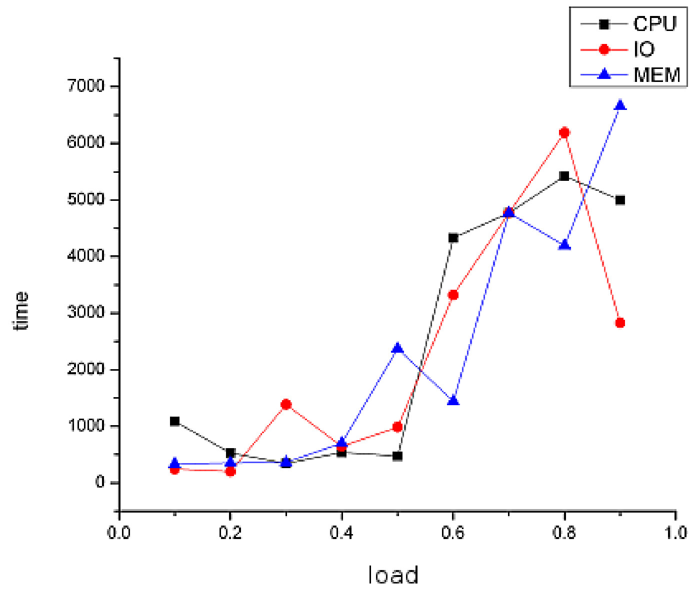

Figure 2: Turn around time of FCFS under different CPU, memory and IO loads (lower is better).

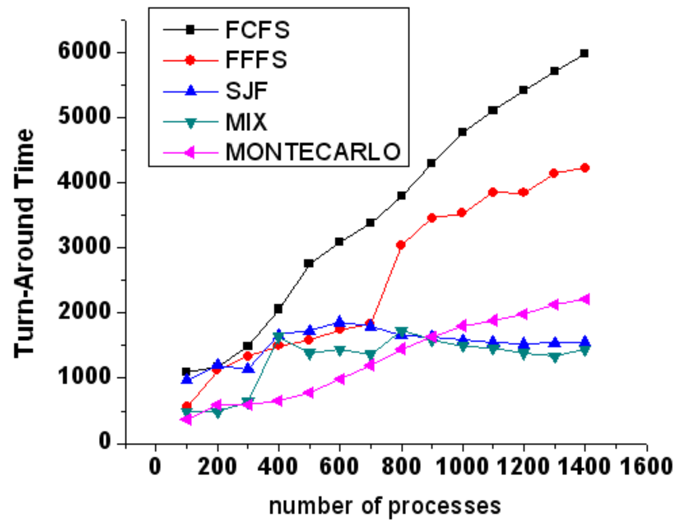

Figure 3: Turn around time for each scheduling algorithm as a function of the number of processes (lower is better)

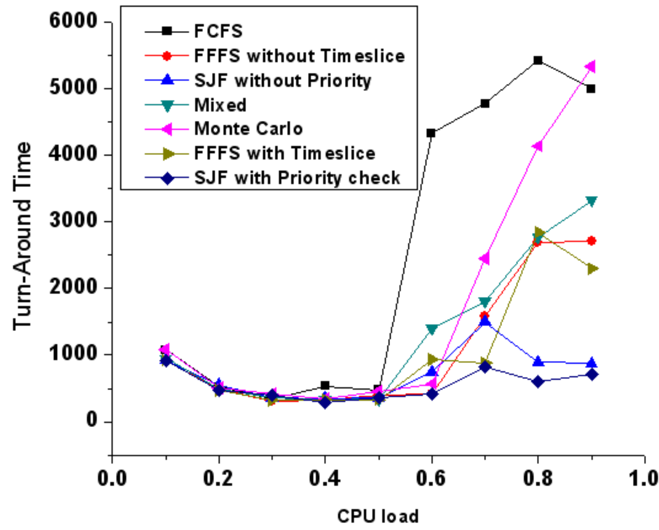

Figure 4: Turn around time for each scheduling algorithm and some combinations of options as a function of CPU load (lower is better)

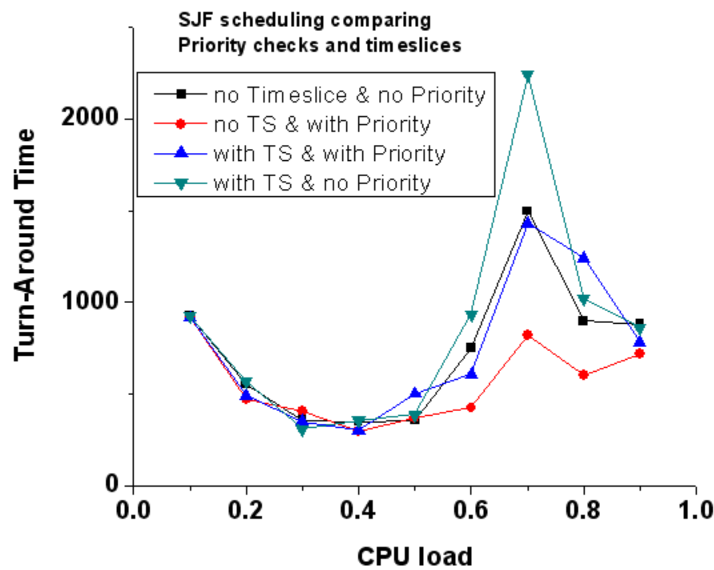

Figure 5: Turn around time for SJF with some combinations of options as a function of CPU load (lower is better)