CmpE 322 – Operating Systems

Project 2 – Modifying the Process Scheduling in Minix

(Due Date: December 26th, 2016 – 23:59)

This document includes the details of your second project assignment. You will modify the user space and kernel space scheduling algorithms in Minix to implement a lottery scheduling algorithm with kernel privilege (more appropriate Turkish term: torpil) support. This project is an individual assignment, please do not attempt it as a group. It will make up 100/1000 points of your course grade.

- Please submit you project by sending an e-mail to cantunca@gmail.com.
- The subject of your email should be in the format **"CmpE322-Project2 Surname Name StudentID"**. Emails not following this subject format will not be considered!
- You need to prepare a **zip archive** containing the project files and include it as an attachment to your submission email. The details of the required project files are given at the end of this document.
- You are also required to include a project report in your project archive. The content and the format of the report are also given at the end.

Minix scheduling is divided into two parts: kernel and user space. The kernel part provides 16 priority queues (numbered 0-15 with lower numbered queues having higher priorities) and simple round robin scheduling. The user space part is responsible for deciding on which queue a process should be put into and the amount of quantum each process is assigned. Therefore, the high-level scheduling decisions can be made in the user space. In this project, you will modify the user space scheduling code to implement a lottery scheduling algorithm. However, the kernel (the big boss) will be able to override the results of the lottery to give some privilege (torpil) to specific processes. Some general pointers to get you started:

- User space scheduling is done in the subserver SCHED (part of PM) which is located under src/minix/servers/sched. The file schedule.c is responsible for the majority of the scheduling decisions. Before attempting this project, please make sure that you understand when the functions in this file are called and what they exactly do.
- Kernel space scheduling code is present under src/minix/kernel. The file proc.c contains the function responsible for handling the priority queues and round robin scheduling.
- Beware: These two directories may not contain all the code you may need to modify. Like the first project, one of the aims of this project is for you to explore and find the necessary code locations and files.
- A comprehensive document on how Minix scheduling works can be found in: http://www.minix3.org/docs/scheduling/report.pdf

Part 1: Implementing the Lottery Scheduling Algorithm with Kernel Torpil

Lottery scheduling (as presented in the course slides) relies on distributing tickets to different processes, and deciding on if a process is to be scheduled by randomly selecting a ticket. The higher the number of tickets a process has, the more chance it gets to be scheduled. The basics of the lottery scheduling algorithm is outlined below:

- Each process should start with a fixed number of tickets, say 10. The processes will be able to modify the number of tickets they have with a system call (details can be found below).
- You should set minimum and maximum limits for the number of tickets a process may have: say 1 and 30, respectively. The ticket setting function should consider these limits.
- The execution times each process gets should be proportional to the number of tickets it has. You may ensure that by ordering the tickets of the processes according to the individual ticket numbers, selecting a random number between 0 and total number of tickets – 1, and scheduling the process holding that specific ticket.
- You should repeat the lottery each time the user space scheduler is invoked after a process has depleted its quantum.
- This algorithm should be applied to only user processes. The system processes should continue to be scheduled normally. Therefore, you need to find a way to distinguish between user and system processes (remember: unlike the system processes, the user processes are forked from the process "init").

In addition to the regular operation of the lottery scheduling algorithm outlined above, you need to implement a mechanism to enable kernel to give torpil to specific processes. When a process with torpil exists, it should be executed until it finishes, without allowing other user processes to receive any execution time.

Below are some specifics on how to implement this algorithm. We also provide the points you will receive for the successful completion of each objective (though we reserve the right to change them in the future, if deemed necessary). In summary, you need to implement two system calls and make some modifications in both the user and kernel space schedulers:

The system calls (10 pts: 5 each):

- First, you are required to implement two system calls in PM (similar to your first project) that have the user library functions `settickets` and `settorpil`.
- `settickets(int numtickets)` should take an integer as input which sets the number of tickets of the calling process (considering the min. and max. ticket limits).
- `settorpil(int torpil)` should take 0 or 1 as input, with 1 meaning that the calling process should have torpil. You must basically set a flag for the processes that have torpil.

In the user space scheduler (30 pts):

- You must select 3 consecutive queues among the 16 available queues in the kernel: say queues numbered 12, 13, 14. (we advise you to select queues higher than 7 and lower than 15, since the former queues are reserved for system processes and the latter is reserved for the process "idle").
- Queue 12 (the lowest numbered queue among the ones you selected) should be the winner's queue. The process winning the lottery should be put into this queue.
- Queue 13 should be the losers' queue. All the other loser processes should be put here.
- Queue 14 (the highest numbered queue among the ones you selected) should be the torpil queue.
- In this scheme, if the kernel behaved normally, we would expect the winner's queue to be prioritized over the other queues, while the torpil queue gets the least priority (remember: the lower numbered queues have higher priorities). So, we can think of the user space scheduler as the good guy, trying not to allow torpil.

In the kernel scheduler (10 pts):

- In this project, as opposed to the good user space scheduler, the kernel is the bad guy with power. So, it will to provide torpil to the processes in the torpil queue, overriding the preferences of the user space scheduler.
- The queues up to the three selected queues should operate normally (Queues 0-11 should be prioritized over the queues you selected no matter what). This would ensure that we do not alter the behavior of the system processes and torpil does not affect them.
- To enable kernel torpil, you must modify the next process selection code in the kernel space to check if the torpil queue (Queue 14) has any process in it. If so, you should select this process regardless of the state of the other two queues (Queues 12 and 13). If Queue 14 is empty, you may select the process in Queue 12 and allow the lottery winner to run, as expected.

Part 2: Testing the Algorithm

You will perform several experiments to measure the effect of the new scheduling algorithm under three different scenarios:

1. The lottery scheduling algorithm without torpil and ticket setting. Each process will have the same fixed amount of tickets during their lifetimes and there will be no process with torpil. We refer to this scenario as the *base scenario*. **(10 pts)**
2. Base scenario + processes with different number of tickets (ticket numbers modified using the `settickets` library function from the test programs' code). The number of tickets should be assigned at the beginning of process execution and should not change (as the base scenario suggests). This scenario should demonstrate the direct effect of the number of tickets. We leave the choice of the initial number of tickets to you. **(15 pts)**
3. Base scenario + torpil. An additional process should use the `settorpil` library function to receive torpil and hence finish its execution before the other user processes. **(15 pts)**

To be able to create the processes necessary for the test scenarios, you need to write two programs, one CPU-bound and one IO-bound. Example: A CPU-bound program that iteratively factorizes a large number, and an IO-bound program that iteratively calculates Fibonacci numbers up to a large number and writes them to a file at each iteration. You may use the programs in this example or come up with your own, provided that you explain why the programs you come up with are either CPU or IO bound, in your report. Please note that these programs should run for a reasonable amount of time to get meaningful test results (the number of iterations should be reasonably large). You must create at least one CPU-bound and one IO-bound process for the scenarios 1 and 2. In scenario 3, you must additionally create one or more torpil processes which can be CPU-bound.

We advise you to create shell scripts for different scenarios, to be able to execute multiple processes at the same time. You must also include these scripts in your submission package.

How to measure the execution times of the processes for the test scenarios are up to you. In addition to the total running times of the processes, we are also interested in the change of the times allocated to the processes during their execution (especially for the bonus part, explained below). To achieve that each process may create a log file including entries for process ID/name, iteration number and time, recorded at different times during execution. Please note that, recording log entries at each iteration may alter the CPU/IO-boundedness of the program, so you need to make sure that the programs do not write to the log files frequently. Using a measurement method other than creating a log file is also acceptable, as long as it is reasonable and you explain it in your report.

Bonus (20 pts)

To get 20 extra points, you are required to improve the lottery scheduling algorithm by coming up with a method to dynamically increase the number of tickets of IO-bound processes and decrease the number of tickets of CPU-bound processes. Even though the kernel does not directly inform the user space scheduler when a process gets blocked, the information to determine if a process is more likely to be IO-bound is available (which can be found in the pdf document linked above). How to decide on when and which amount to increase/decrease the tickets of a process are left to you. We will accept any thoughtful solution. If you choose to implement this bonus part, you are also required to run a fourth test scenario (without torpil) to show its effect.

Report (10 pts)

You should write a short report with separate sections for each of the two parts, and address the following items. You should include this report in your project zip archive as well.

Part 1:
- Where exactly are the source files that you modified? List the directories, filenames and the brief locations of modifications within the files.
- Briefly discuss the modifications you did and what they exactly do.

Part 2:
- Explain what your test programs do, and why they are CPU or IO bound.
- Provide instructions on how to execute each of the three (or four, if you did the bonus part) test scenarios.
- Explain how you chose to measure the timing of the processes, and possibly the format of the log files that are produced.
- Present the input parameters/conditions you selected for the algorithm and the tests (number of the selected queues, initial number of tickets, ticket min/max. limits, etc.).
- Summarize the results of the experiments: The numeric timing results for each scenario. There are no limitations or minimum requirement for the variety of the results you present in the report. You should simply present enough results to convince us of the algorithm's effect (If presenting only a single numeric result does that, it is OK).

Submission Package

Your zip archive should contain the following items:
- The diff patch of the src directory containing all the modifications. (we also accept git patches, if diff patch produces a very large file that does not fit into mail attachment; but getting an unreasonably big patch file could mean that something else is wrong.)
- The test programs you wrote for Part 2.
- The shell script files to run your test programs for each test scenario.
- The output log files (or any other file as the output of the measurement method you chose), for each test scenario.
- Project Report in .pdf or .docx format.

Demo

We will arrange a demo session for you to come and demonstrate the work you have done. This session will primarily determine your grades. The date and the details of this demo session are to be announced.

Final Remarks

- This project is much more difficult than the first project, so you really are advised to start early (no kidding this time).
- Even though there are many online resources to help you through this homework (and we encourage you to make use of them, without plagiarism of course), searching for the right information, understanding and implementing it may take a considerable amount of time. Moreover, executing tests and gathering the results may also take time.
- In the demo, we may ask you theoretical questions to assess how well you understand the scheduling in Minix and the algorithm you implemented, so make sure you fully understand each step.
- We will accept partial work, so if you have any trouble implementing a part of the scheduling algorithm exactly as specified in Part 1, you may slightly modify it to something you can implement. In short, submitting a modified version of the project is better than submitting nothing. Interesting alternative ideas may even result with no grade reduction (though we reserve the right to assess what is interesting and what is not). Of course, you should explain exactly what you modified and why you did it, in your report and during the demo.