NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Lab 2

Scheduling

Handout: 11.02.2025
Deadline: 28.02.2025, 23:59

Introduction

Hello, and welcome to the second lab. In the labs you have the opportunity to practically try out things you learned in the lectures and the theoretical exercises. The labs are compulsory coursework, you need to get at least **27 points from four labs** approved to sit in the exam. The labs are also subject to NTNU's plagiarism rules. More on that in Section 3.

If you haven't done yet, please sign up for one of the lab sessions under https://s.ntnu.no/labsessions24. You need to be logged in with your NTNU account in order to access the file. The lab sessions will help you get started

1 Task Description

This week we are diving into the scheduler of XV6. The plan is to analyse the influence of the scheduler on the execution time of processes. As you might expect, the way we schedule processes to run can have a huge impact on perceived and measured performance of a system. While perceived performance can be very important, it is not a reliable measurement. As engineers we want to measure performance reliably, such that we can compare different approaches objectively. However, measuring system performance is not always easy: There are many different things we might want to consider, and most likely the objectives change depending on the application.

Tools

In this weeks lab handout code, we included a simple utility tool called `time` that allows you to measure the execution time of a process in the number of ticks (ticks being the number of timer interrupts occurred). While this is a very coarse measurement, it is good enough to compare the different implementations in the context of this lab.

To make testing easier, we also added two additional commands, `schedls` and `schedset`, to quickly switch between different scheduler implementations. `schedls` lists all registered schedulers (see `proc.c` for the sample round-robin scheduler and how it needs to be registered). `schedset` can then be used to set the current scheduler. While `schedset` sets the pointer to the corresponding scheduler function, the old scheduler needs to return first for the change to take place. See again the sample round-robin scheduler for how to return on a scheduler change. See below for a small sample of all the tools.

```
$ schedls
  [ ]     Scheduler Name  Scheduler ID
  ====================================
  [*]     round-robin     1
          MLFQ Sched      2

  *: current scheduler

$ schedset 2
  Scheduler successfully changed to MLFQ Sched
$ schedls
  [ ]     Scheduler Name  Scheduler ID
  ====================================
          round-robin     1
  [*]     MLFQ Sched      2

  *: current scheduler

$ time ls
  .              1 1 1024
```

```
..              1 1 1024
README          2 2 2305
cat             2 3 32704
echo            2 4 31592
forktest        2 5 15568
grep            2 6 36048
init            2 7 32024
kill            2 8 31576
ln              2 9 31384
ls              2 10 34592
mkdir           2 11 31648
rm              2 12 31640
sh              2 13 55160
stressfs        2 14 32368
usertests       2 15 181664
grind           2 16 47560
wc              2 17 33696
zombie          2 18 31032
ps              2 19 32008
time            2 20 32000
congen          2 21 32480
schedls         2 22 30960
schedset        2 23 31472
load.sh         2 24 24
console         3 25 0
Executing ls took 4 ticks
```

Listing 1: Sample in/output for `schedls`, `schedset` and `time`

1.1 Task 1: Baseline – Measuring Performance

Before we start implementing any new schedulers, we first need to establish the baseline that we are working with. To do that, we need a benchmark. For that specific purpose we included a small shell script in this weeks handout, called `load.sh`. You find it in the root directory of XV6 and you can run it by typing `sh load.sh`. You see a sample below. Be aware that this shell script might take some time to run.

```
$ sh load.sh
  3305 9966 47185
```

The script itself executes three programs, the first one generates output by starting up five child processes and counting up to a certain constant. It prints the current state of the counter together with the information if it is comming from a CHILD process or the PARENT process. The second program filters the output and only lets lines through that contain "CHILD". The last program then counts the number of lines, words and characters. When running the `load.sh` script, we only get to see the output of the last program.

Remark: *If you want to know in more detail what each of the programs does, or how the benchmark runs, feel free to look at the code - all the source is available, so you should be able to read through it and understand what it does.*

In order to measure how long this benchmark takes, we can run the script under the time command as follows.

```
$ time sh load.sh
  3305 9966 47185
  Executing sh took 1351 ticks
```

As you can imagine, measuring the execution time only once does not yield very reliable numbers. Therefore we ask you to measure the time 10 times and collect the execution time to get an idea of how the runtime of the benchmark is distributed. If the benchmark takes too long to run, you can also adjust the benchmark a bit, but please ensure that in the end you test all the scheduler implementations with the same benchmark.

Feel free to automate collecting the ten measurements (e.g. by writing an additional small shell script that runs the time command several times or by extending `test-lab-l2` Python script). Save those ten measurements persistently, so we can compare to them to the Multi-Level Feedback Queue implementation.

1.2 Task 2: Multi-Level Feedback Queue

The default scheduler of XV6 uses the round-robin method. We are now going to implement an alternative scheduler that uses the Multi-Level Feedback Queue approach. Here is a quick reminder of the rules that the MLFQ scheduler follows:

1. If Priority(A) > Priority(B), A is scheduled to run next
2. If Priority(A) = Priority(B), A and B will run according to the round-robin scheduling policy
3. A job is first placed in the highest priority queue when it enters the system
4. If a job uses up the entire allocated time slice, its priority is reduced (moved to a lower priority queue)
5. A job is moved to the topmost queue after a time period `S`

MLFQ is defined by a few parameters. It is up to you how you implement the MLFQ, as long as your implementation has at least **two (2) priority levels** and **S $\geq$ 2**. The other parameters are up to you to determine. Feel free to play around with some of them to find a more optimal configuration for the benchmark at hand.

Once you wrote your MLFQ implementation, you should be able to run the same commands as in Listing 1. Please register your scheduler as the second (with ID 2) scheduler in the system. That is required for automated testing purposes.

To test your implementation, you also need to implement a syscall that returns the current priority of the current process. You find a stub called `sys_getprio` in `sysproc.c`.

Files that you might want to take a look at for this task:

`kernel/proc.c`: That file contains the current scheduler (`rr_scheduler`) and the structures to register new schedulers. Further interesting are the functions `allocproc`, which sets up new processes and `yield`, which calls the `sched` function (which organizes the scheduling).

`kernel/proc.h`: Contains the declarations for `proc.c`. You might find it useful to modify the `struct proc`, adding scheduling metadata.

`kernel/trap.c`: This file contains the trap handler, which also gets invoked when a timer interrupt arrives.

`kernel/sysproc.c`: Implement the `sys_getprio` syscall there, which should return the current priority of the process that invoked the syscall.

If you find it difficult to start with the task, see the optional tasks below. Some of the easier ones might help you to break down the problem into more manageable pieces.

1.2.1 Optional Tasks

EASIER: Implement a scheduler with two fixed queues. A process is placed either in the high or low-priority queue and never changes its priority. The rest of the rules stay the same as for the MLFQ scheduler (Rules 1 & 2 in the list above).

EASIER: Implement the `sys_getprio` syscall to test the priorities of your processes.

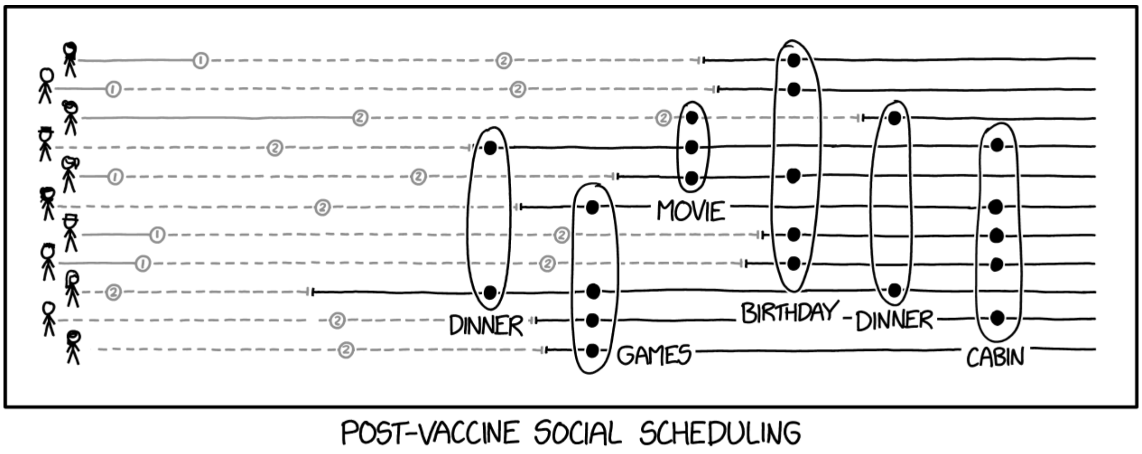
EASIER: Where does XV6 invoke the scheduler and how is control returned to the kernel? How can you distinguish between a program that used up the entire time slice it got and one that returned control on its own earlier? Note that we currently call the scheduler at each timer interrupt, so timing information is relatively coarse-grained and thus not necessarily useful for determining if a process used up the entire quota.

EASIER: Extend the scheduler from the previous version to one that always switches the priority when a process uses up the entire quota (decrease priority) or not (increase priority).

SIMILAR: To move from the aggressive switching priority scheduler closer towards MLFQ, add promotion to the topmost priority queue only after it spent some time `S` in the lower priority queue.

HARDER: Implement MLFQ for more than just two queues. How does your implementation change, and does it affect performance? Which number of queues gives you optimal results? Is the increased complexity worth the performance improvement?

HARDER: If you have already implemented the MLFQ but think you can do better than MLFQ for the given benchmark, write a third scheduler optimized to run the benchmark used for this lab. How much faster are you than round-robin? How much faster than MLFQ? Don't forget to take at least ten measurements to show the difference in performance.



"As if these problems weren't NP-hard enough."
Source: Post Vaccine Social Scheduling by Randall Munroe / CC BY-NC 2.5 DEED

1.3 Task 3: MLFQ – Measuring Performance

Similar to the baseline we ask you to take ten measurements of the benchmark, now using your newly written MLFQ scheduler. Is the performance better, worse or comparable? Can you find an explanation for the behaviour?

2 Handing In

Before handing in, we advise you to use the command `make test`, which will test your implementation with a small set of test cases. If you want to see the tests in greater detail, you find the commands that were executed and the expected output in the file `test-lab-l2`. The test script also tests for some of the optional tasks, so you don't need to pass all the tests. We marked the test cases that belong to the mandatory tasks with a tag `[MANDATORY]`. Try to make sure that all the mandatory tests pass before handing in.

After you tested and potentially debugged and fixed your solution, you can run `make prepare-handin` to create an archive of your solution. The archive will be written in the current directory and the file is called `lab-l2-handin.tar.gz`. Take that file and upload it to blackboard on the submission page for lab 2. You can submit multiple solutions, we are going to grade the last submission we received before the deadline.

Additionally, you will find a short form for lab 2, where we will ask you about the performance of your solutions. That number is not considered for grading, but we expect you to provide the measured information in the form. **Otherwise, the submission is not complete.** We will use that to provide you with a small statistic about the performance of all the solutions handed in.

Deliverables to Hand in

- **lab-l2-handin.tar.gz**: The output of the command `make prepare-handin`. Please do not package up your code manually, as this command runs some tests on your side and ensures that we see if it ran on your machine. Additionally, the format it packages it up is the one expected by our automated tests.
- **Performance Measurement Survey**: You find a survey on Blackboard in the `Labs/Lab 2` folder. Please enter your measurements for the base scheduler (`rr_scheduler`) and for your MLFQ scheduler there.

Missing Deadlines

We expect you to start working early on the labs, to ensure submitting before the deadline. The deadlines are fixed and there is little we can do to move them. However, if you encounter a problem and notice that you can't make it in time, please contact us as soon as you notice. If you contact us early we have time to come up with a solution.

Last lab: Since we need to hand in a list of people that get to sit in the exam, handing in late for the last lab is not possible. We need a bit of time to grade the labs and if you hand in late, we cannot grade your assignment in time before the list must be handed over to administration.

3 Plagiarism

You must submit your **own work**. You must write your **own code** and **not copy** it from anywhere else, including your classmates, internet, and textbooks[1]. Failure to do so might be considered plagiarism. Detailed guidelines on what constitutes plagiarism can be found at: https://innsida.ntnu.no/wiki/-/wiki/English/Cheating+on+exams.

We check all submitted code for similarities to other submissions and online sources. Plagiarism detection tools have been effective in the past at finding similarities. If we suspect that a student copied code, we will involve administration to follow up on the case.

[1]This is not an exhaustive list. Don't copy code from any source