Operating Systems 2016 Assignment 2: Processes

Deadline: Friday April 1 before 23:59 hours.

1 Introduction

Process scheduling is an important part of the operating system and has influence on the achieved CPU utilization, system throughput, waiting time and response time. Especially for real-time and modern interactive systems (such as smart phones), the scheduler must be tuned to perfection. The task of the scheduler is to decide which process will be run next, based on a list of processes which are ready to run (and are not blocking on for example I/O).

In this assignment you will implement two schedulers in a real, functional operating system and perform a number of experiments. The operating system is supplied to you with a First Come First Serve (FCFS) scheduler. You will implement a Round Robin (RR) scheduler and a Multi-Level Feedback Queue (MLFQ) scheduler that combines FCFS and RR.

To evaluate the performance of these schedulers you will implement the necessary code in the kernel to compute the total waiting time for each process. Finally, you will perform an experimental evaluation of the schedulers using three (simulated) workloads. Each workload executes a certain process mix and you will be benchmarking the average waiting time for the different schedulers that have been implemented. Your findings are presented in a concise report.

Note that next to the lab sessions you will have to work most of the time on your own. Therefore it is **very** important that you are able to access the University systems, where the necessary tools are installed, remotely or that you do your own installation of the required software on for example your laptop. We will also make a Linux virtual machine image that includes all software available.

2 Requirements

We expect the following to be achieved:

- Implement a working Round Robin scheduler. This scheduler must adhere to the following requirements:
  - Each process on the ready queue is run in turn for a fixed time quantum.
  - A process whose time quantum has expired must be taken off the CPU (preemption).
  - The idle process may only be scheduled if there is no other process to run.
  - Implement the scheduler in such a way that it is easy to switch between the given FCFS and RR scheduler for the experimentation part of the assignment.
- Implement a Multi-Level Feedback Queue (MLFQ) scheduler consisting of two queues.
  - Make use of two ready queues and optionally also a queue for the idle process.
  - The first queue is meant to be used for interactive processes and is scheduled using RR.
  - The second queue is scheduled using FCFS.
  - New processes enter the system through the first queue. If a process exhausts its time quantum it is demoted to the second queue.
  - No rule is implemented for promotion.

  Note that you can re-use the implementations of FCFS and RR in in your implementation of MLFQ
- Instrument the system by implementing accounting of "wait time" per process (in system ticks). This is the time a process spends in the ready queue. Display the accumulated wait time when the process is unloaded (e.g., in `proc_unload_sched()`).

- Perform a number of experiments with the schedulers and write a concise report about your findings. We would like to see answers to the following questions:
  - Which scheduler is preferred for each workload and why? Determine the average waiting time for the different workloads when running under the different schedulers.
  - What is the influence of different time quantum settings in RR on the average waiting time and on the interactivity of the system?
  - How would you tune the MLFQ scheduler? What time quantum works best?

Note that because you have to modify existing operating system source code written in C you will be writing C code. All modifications that you need to make should be constrained to the files `kernel/src/process-sched.c` and `kernel/include/process.h`.

3 Submission and Grading

You may work in teams of at most 2 persons. A single tar.gz file should be handed in that contains:

- `kernel/src/process-sched.c`
- `kernel/src/process.h`
- The report in PDF format.

If you have modified any other source files, please include these too and state in a `README` file why you had to modify these files. Please *do not* hand in any other files. In particular do not send us any object files and binaries.

Make sure that the files that you have modified contain your names and student IDs. Also make sure that the filename of the tar.gz file contains your student IDs like so:

`assignment2-sYYYYYYY-sXXXXXXX.tar.gz`

Mail your tar files to *os2016 (at) handin (dot) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2016 Assignment 2". Include your names and student IDs in the e-mail.

Deadline: We expect your submissions *before* Friday, April 1, 23:59 hours. No exceptions; deliveries after the deadline *will not be graded!*.

The grade is determined based on whether the program correctly implements the functionalities listed in the specification above, the quality of the report and whether the source code looks adequate: good structure, consistent indentation, error handling, correct memory handling and comments where these are required. Comments are usually required if the code is not immediately obvious, which often means you had to make a deliberate decision or trade-off. Document these decisions, trade-offs and why in the source code. Commenting on the obvious is superfluous and bad style. Note that we may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.

The maximum grade that can be obtained for this assignment is 10. The points are distributed as follows: Code Layout & Quality (1 / 10), RR scheduler (2 / 10), MLFQ scheduler (3 / 10), "wait time" instrumentation (1 / 10), Experiments & Report (3 / 10).

4 Programming language

Because you will be extending an existing operating system kernel, which has been written in C, you will have to complete this assignment using the C language. This means that you cannot

use C++ features such as classes, virtual methods and `cout` and `cin` for I/O. Also note that a C library is not available in the kernel! Only a very small amount of familiar C library functions are available. Some notes:

- For outputting text, use `puts` or `printf`.
- For this assignment you should not have a need to dynamically allocate memory. (Otherwise look at uses of `slab_alloc` and `slab_free`).
- When you make a mistake using pointers in your kernel code, you will not get a segmentation fault, but instead the kernel will crash and cannot continue. The kernel will tell you which instruction made the faulty memory access, which can be used in debugging.
- You will most likely want to temporarily include `printf` statements to debug the code and verify that it works as expected.
- Ask the assistants for help if you have problems!

5 Quick start guide to modifying the kernel

All software and additional files required to complete this assignment are installed on the lab machines in room 411 and in the usual computer rooms 302/304, 306/308. For rooms 302/304, 306/308 you must first execute the following command before you can use the tools:

```
source /vol/share/groups/liacs/scratch/os2016/os2016.bashrc
```

If you want to install the software on your own machine, refer to the software installation guide which is available on the website. Once the software has been setup, your work flow will be:

- Compile the kernel: create a `build` directory in the kernel source directory, use cmake (see the README file and getting started guide) and *make*.
- Copy `kernel.bin` to `sdcard.img` using: `mcopy -i sdcard.img@@1048576 kernel.bin ::`
- Run qemu, refer to the getting started guide to see which arguments to pass to qemu.
- The kernel will launch automatically, or when you give the `boot` command at the command prompt.
- Once the kernel is running, it will give you a shell. You can try to run `ls`, `ps` and `cat`.

Important: when using the machines in room 411, make sure to take a back up of your assignment with you (on a USB key or through e-mail) at the end of the day. Your data is also available from "sshgw" and the "huisuil" machines by connecting with ssh to *ssh411.liacs.nl*.

During the lab hours it is possible to test your code on the BeagleBoard hardware.

6 First Come First Serve Scheduling

Before you start making modifications, try the kernel as has been supplied to you. You can start the program `stress` to launch a couple of background processes (it will launch 4 more background processes every 10 seconds). What happens as soon as `stress` starts running? Can you explain this behavior? You can also try the `workload1`, `workload2` and `workload3` programs, each of which launches a mix of processes with different characteristics.

7 Round Robin Scheduling

In Round Robin scheduling, each process is run in turn for a fixed time quantum. The time quantum is already defined for you as `PROC_TIME_QUANTA` in the file `kernel/include/process.h`. This header file also contains the definition of the process structure (`struct _proc_t`), in which all information related to a process is stored.

To implement a scheduling algorithm, you will mainly be modifying the code in `kernel/src/process-sched.c`. All functions in this file have been documented for you. The functions `proc_enqueue()` and `proc_dequeue()` are used to put processes on the ready queue and to remove them. The function `proc_schedule()` is supposed to make a decision on which process to run next. Currently, `proc_schedule()` will select a next process to run if the current process has been blocked or when the current process is the idle process (`kernel_proc`, which does nothing and puts the system to sleep until it is woken again by an interrupt), otherwise the current process will continue to run.

It is your task to change this function to select a next process to run from the ready queue. It should remove the process to run next from the queue and place the process that will be suspended (which is the process that is currently running) at the tail of the ready queue. Beware that when the current process is blocked or zombie (`current->state` equals `PROC_BLOCKED` or `PROC_ZOMBIE`), it is not ready to run and should not be placed on the ready queue. The process that is selected to run next should be granted a time quantum.

Important is that the idle process should get special treatment. It is not placed on the usual ready queue, because we only want to run the idle process when there is no useful work to do. Therefore, you probably want to implement an exception for the idle process in `proc_enqueue()`, `proc_dequeue()` and `proc_schedule()` to ensure this. The `is_idle_process` function will be useful to use.

Secondly, you want to make sure that a process is taken off the CPU when it has used its full time quantum. The time quantum is stored in the field `time_quanta` of `proc_t`. It is your task to modify the function `proc_tick()` to do this. The argument `p` of `proc_tick()` points at the current process. On each call to `proc_tick()` (which is triggered by a clock tick), you should decrease the `time_quanta` field. When this field reaches zero, you have to schedule a next process and preempt the current one. Note that the idle process is always ready to run.

If you try your kernel now, you should notice that as soon as you start `stress`, the system continues to be interactive. The response times should also improve when running the different `workload` programs.

8 Multi-Level Feedback Queue

In your implementation of the Multi-Level Feedback Queue, you can re-use the implementations of FCFS and RR you have by now. First, you need a mechanism to determine to which queue or priority each process belongs. To do so, you could for example add a field to the process control block found in `process.h`. Make sure the initialize the process priority in `proc_create_sched()` if necessary. As a second step, you could change the code so that `proc_enqueue()` and `proc_dequeue()` operate on the correct ready queue corresponding to the process' priority.

Third, modify `proc_schedule()` to take all ready queues into account and to invoke the RR and FCFS "sub-schedulers". Of course, any process that is ready in the "interactive" RR queue gets precedence over the lower priority FCFS queue.

Finally, you want to implement process demotion. All processes start in the "interactive" queue. Once a process has used its full time quantum, demote the process to the lower priority FCFS queue.

9 "Wait time" instrumentation

We would like to instrument the kernel to maintain the total wait time (in system ticks) for each process. The wait time is the time a process spends "waiting" in the ready queue to be put on the CPU. We will not describe the implementation but instead give a number of hints:

- You may want to add fields to the process control block in `process.h`, feel free to do so!
- The current system time in ticks can be obtained by calling `timer_get_system_ticks()`.
- Remember that `proc_enqueue()` is called when a process must enter the ready queue and `proc_dequeue()` when a process must be removed from the ready queue. And note that `proc_dequeue()` may even be called for a process `p` that is currently not on the ready queue! Take this into account!
- You can put a `printf` to display the waiting time upon process termination in `proc_unload_sched`.

10 Report

After thoroughly testing your implementation, it is time to conduct some experiments. We have provided three example workloads to use in the experimentation: `workload1`, `workload2`, `workload3`. Each of these workloads executes a mix of processes with different CPU and I/O bursts. Design and perform a number of experiments such that you are able to discuss the following issues:

1. Which scheduler is preferred for each workload and why? Determine the average waiting time for the different workloads when running under the different schedulers.
2. What is the influence of different time quantum settings in RR on the average waiting time and on the interactivity of the system?
3. How would you tune the MLFQ scheduler? What time quantum works best?

In your report, describe the experiments that have been conducted, the results and a discussion of the results.