# Operating Systems 2014 Assignment 2: Process Scheduling

**Deadline:** April 6, 2014, at 23:59.

## 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 write your own scheduler in a real, functional, operating system. The operating system is supplied to you with a First Come First Serve (FCFS) scheduler. We will see that this scheduler is not sufficient for an interactive system when jobs are being run in the background. It is your task to improve this situation by writing a Round Robin scheduler with preemption. In this case, all processes get to run in turn for a given period of time.

The Round Robin scheduler does make the system work, however, the wait time for interactive processes is not as good as it could be. Therefore, the next part of the assignment is to implement a Multi-Level Feedback Queue scheduler. This scheduler will keep the system interactive while computation jobs are going on in the background.

Next to the abovementioned schedulers for interactive systems, embedded systems often deploy hard real-time schedulers, to guarantee that applications will meet their given tight deadlines. Missing deadlines will, otherwise, cause system failure, or even life danger. This can be seen for example in air-bag systems and engine control used in cars. To this end, Earliest Deadline First (EDF) scheduling is an optimal scheduling policy. In this assignment, you will implement a simplified version of an EDF scheduler.

You will have three lab sessions to complete this work. Next to these three lab sessions you will have to work most of the time on your own. **So it is extremely important that you do your own installation of the required software, for example on your laptop.** We can provide help with this during the lab hours.

## 2 Requirements

You may work in teams of at most 2 persons. You have to modify existing operating system source code written in C and therefore you will be writing C code. Your submissions must adhere to the following requirements:

1. Submit the source code of the operating system with your functioning Round Robin scheduler. The source code should:
    - Make use of a single run queue for non-idle processes.
    - Run each process on the run queue in turn, for a fixed time quantum.
    - Preempt the process (take it off the CPU) when its allocated time quantum has been exceeded.

- Only schedule the idle process when there is no other process to run.
- Provide a function to resume a process and immediately run this process by preempting the currently running process. This function should be used in the interrupt handler of the serial port (UART) driver.

2. Submit the source code of the operating system with your functioning Multi-Level Feedback Queue scheduler. The source code should:

   - Make use of three run queues (high, normal and low), optionally also a queue for the idle process.
   - Run high priority processes for 1 time quantum, normal priority for 2 time quanta and low priority for 5 time quanta.
   - Only schedule the idle process when there is no other process to run (e.g. the low priority queue is empty).
   - Once a process has used its full allocated time quantum, it should be demoted to a lower priority run queue (high is demoted to normal, normal is demoted to low).
   - After any job has run for a single time quantum, it must be checked whether the job has run through its entire allocated time quantum, in which case it is taken off the CPU (unless there is no other process ready). Otherwise, if the job is not of the highest priority, it must be checked whether any job is ready in the run queues of a higher priority than the current job. If so, the process should be preempted.

3. Submit the source code of the operating system with your functioning Earliest Deadline First scheduler. The source code should:

   - Correctly compute a process' release time and deadline based on the given parameters at the appropriate times.
   - Place processes which release time has not passed in the unreleased queue, processes which release time has passed in the released queue. On each run of the scheduler, eligible processes should be moved from the unreleased to the released queue.
   - Consistently schedule the process in the released queue with the earliest deadline to run first.
   - Check at every clock tick if the deadline of the current process has passed and whether there is another process ready to run with an earlier deadline than the current process. Note that this does not apply to the idle process, this process should always be preempted if there is another process to run.
   - Launch *edftest* instead of *init* during boot.

4. This means that we will receive three tar files from each team, one with the Round Robin scheduler, one with the Multi-Level Feedback Queue scheduler and one with the EDF scheduler.

When grading the submissions, we will look at whether your source code fulfills these requirements and the source code looks adequate: good structure, consistent indentation 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. Note that we may always invite teams to elaborate on their submission in an interview in case parts of the source code are not clear.

*Make sure that the files you have modified contain your names and student IDs.* Remove any object files and binaries by removing the build directory. Create a gzipped tar file of the directories:

```
tar -czvf assignment2-round-robin.tar.gz assignment2-rr/
tar -czvf assignment2-mlfq.tar.gz assignment2-mlfq/
tar -czvf assignment2-edf.tar.gz assignment2-edf/
```

Mail your tar files to *krietvel (at) liacs (dot) nl* and make sure the subject of the e-mail contains *OS Assignment 2*. Include your names and student IDs in the e-mail.

**Deadline:** We expect your submissions *before* April 6, 2014, at 23:59. No exceptions; points are deducted for late delivery.

## 3 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!

## 4 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, which you can use during the lab hours on Wednesdays. During the lab hours we can help you to install the software on your own machine. Also refer to the getting started 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 make.
- Copy `kernel.bin` to `sdcard.img`: `mcopy -i sdcard.img 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`.
- The secondary window displays a graph of the CPU time used by each process.

**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.

Testing your code on the BeagleBoard hardware is possible during the lab hours.

## 5 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?

## 6 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 this scheduling algorithm, you will mainly be modifying the code in `kernel/src/process.c`. Notice that the functions `proc_enqueue()` and `proc_dequeue()` are used to put processes on the run 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 run queue. It should remove the process to run next from the run queue and place the process that will be suspended (which is the process that is currently running) at the tail of the run queue. Beware that when the current process is blocked (`current->state` equals `PROC_BLOCKED`), it is not ready to run and should not be placed on the run queue.

Important is that the idle process should get special treatment. It is not placed on the usual run queue, because we only want to run the idle process when there is no useful work to do. Therefore, you should 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. From the displayed graph you can observe that the groups of 4 processes launched by `stress` all get equal time to run.

However, the response and waiting times are not great. One explanation for this is that as soon new input comes in, it has to wait until the process acting on this input may run next. When the number of processes pending in the ready queue is high, this takes a long time. To alleviate this, we want to modify the kernel so that when new interactive input (e.g., from keyboard) comes in, the process blocking for (waiting for) this input is immediately run next (the current process is preempted).

To do this, make a new function called `proc_resume_and_preempt()` based on `proc_resume()` that immediately runs process `p` (using `proc_preempt`) instead of putting `p` on the ready queue. queue.

Secondly, modify the input driver (`kernel/src/arch/arm/serial.c`) to call `proc_resume_and_preempt()` instead of `proc_resume()` in the function `uart_irq_handler()`.

Try to run the kernel now and see if response time for input has improved after you have launched `stress`.

## 7 Multi-Level Feedback Queue

The Round Robin scheduler has made the system functional and we also implemented a trick that made the response time for input reasonable. Nevertheless, when starting a new command from the shell while `stress` is running, there still is a noticeable delay before the program will actually start. To solve this, we will implement a more advanced Multi-Level Feedback Queue scheduler.

Three ready queues have already been defined at the top of `process.c`: high, normal and low priority. Each process is assigned to one of these queues. The time quanta handed out to a process depends on which queue the process is part of. Again, we make an exception for the idle process, which should not appear in the low priority queue but should be treated on its own.

We suggest that you first change the code so that `proc_enqueue()` and `proc_dequeue()` do not append the process to the normal priority ready queue, but to the ready queue that belongs to the process' priority. Make sure to initialize the process' priority in `proc_create()` and also in the definition of `static proc_t kernel_proc` at the top of the file.

Second, when you reset the time quantum (in `proc_tick()` and `proc_enqueue()`), use the correct time quantum according to the process' priority. Processes in the high priority queue get a single time quantum (`PROC_TIME_QUANTA`), normal priority 2 times the default time quantum and low priority 5 times the default time quantum. For the idle process we will use the same quanta allocation as for low priority processes.

Third, modify `proc_schedule()` to take all three ready queues and how you desire to handle the idle process into account. Of course, processes in the higher priority queues get precedence over the lower priority queues.

Fourth, you want to implement process demotion. All processes start in the highest priority (modify `proc_create()` and `kernel_proc` for this). In `proc_tick()` bookkeep whether or not a process has used its full time quantum. Based on this you should demote the process to a lower priority run queue.

If you try to run the kernel now and launch `stress`, you should observe that the processes spawned by `stress` are demoted to lower priorities. While the interactive processes, like the shell and `ls`, always remain in the higher priority queues. Due to this prioritized scheduling, the groups of four processes will still be visible, but not as clear as before.

There is still a single problem: you will notice that as soon as new background processes are started, it takes one or two seconds before the system becomes fully interactive again. You will resolve this as follows. Even though the lower priority processes will be allowed to run for 2 or 5 time quanta, we will check if there is a higher priority process pending in the ready queue each time after a single quantum has passed. You should modify `proc_tick()` to implement this behavior.

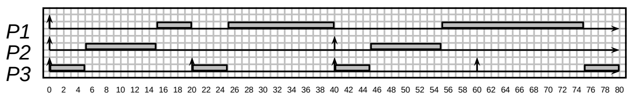
When you try the kernel with these last changes, you will notice that the system stays fully responsive.

## 8 Earliest Deadline First

In Earliest Deadline First (EDF) scheduling, we will focus on a simplified version of EDF scheduling, which schedules a set of periodic processes. A periodic process has several parameters, i.e., start time $S$ (when the process should initially start running), period $T$ (the process is released every time $T$), and deadline $D$ (once a process is released, it must must finish an execution before its deadline). For each process, the absolute release time $r_i$ of the i$th$ execution can be computed as: $r_i = S + i \cdot T$. The absolute deadline $d_i$ of i$th$ execution can be thus obtained as: $d_i = r_i + D$. Here we see an example consisting of three processes with their parameters shown in Figure 1(a). An EDF schedule is shown in Figure 1(b). The upper arrows denote the release time and the gray

| Processes | start time | Period | Deadline |
|---|---|---|---|
| *P1* | 0 | 80 | 80 |
| *P2* | 0 | 40 | 40 |
| *P3* | 0 | 20 | 20 |

(a)



(b)

Figure 1: (a) Three processes and their parameters and (b) an EDF schedule.

bars indicate the execution. Note that multiple feasible EDF schedules might exist depending on the implementation. In general, a good schedule should have the least amount of preemption.

To implement an EDF scheduler, two priority queues are maintained, i.e., the unreleased queue and the released queue. The unreleased queue contains processes whose release time has not arrived. This is compared to the system time, which can be obtained with `timer_get_system_ticks()`. The released queue contains processes whose release time has passed. Because both queues are priority queues, the process with the smallest absolute release time or absolute deadline, respectively, is at the front of the queue. As data structure we will use a heap, which has already been implemented. You will use the variables `released_queue` and `unreleased_queue` in `process.c`. For how to use the heap, refer to `kernel/include/heap.h`. Note that the heaps are already initialized in `proc_schedule_init()`.

It is your task to implement a functional EDF scheduler. First, you need to make sure the EDF parameters (refer to `kernel/include/process.h`) are initialized to sane defaults for a new process. This is done in `rt_proc_create`. For example, you need to initialize start time, deadline, and period. As start time, you can use a value of 100 for all processes. Also, initialize `n_periods_executed` and compute the absolute release time and deadline using `proc_update_edf_parameters`. A process has finished its period when it performs the `sleep(-1)` system call, which will call `proc_yield` in the kernel. In this case, the parameters must be updated to reflect the next period.

Secondly, in `proc_enqueue` you must place the process `p` in the correct priority queue.

Third, `proc_schedule` should move all eligible processes from the unreleased queue to the released queue. The current process should be placed back in the queues using `proc_enqueue`, with as exception the idle process, which we will not place in the priority queues. After that, find out if the released queue contains a process to run. If there is no process to run, schedule the idle process.

Fourth, `proc_tick` should invoke `proc_schedule` to determine the next scheduled process. If it is different from the current one, the current one should be preempted. An idle process must be pre-empted if there is another process to run.

To test your code we have provided an executable called *edftest*. You should modify `kernel/src/boot.c` to launch this executable instead of *init*. This executable will fork 5 processes, with the following parameters (values are in system ticks):

1. $D = 500, T = 500$
2. $D = 700, T = 700$
3. $D = 900, T = 900$
4. $D = 1000, T = 1000$
5. $D = 4000, T = 4000$

Each process will indicate on the console when it gets a turn to perform its useful work in the infinite loop. To validate your scheduler, work out in which order the processes should run and check this with the output from the operating system.