Operating Systems 2015 Assignment 2: Processes

Deadline: Sunday April 5 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 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 Multi-Level Feedback Queue scheduler with preemption. This scheduler will keep the system interactive while computational jobs are going on in the background.

In the second part of the assignment, you will implement blocking, direct, symmetric message passing. Using the provided test programs, we will see how this variant of message passing can be used as a process synchronization primitive.

You will have three lab sessions to work on this assignment. 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.** A Linux virtual machine image that includes all software is also available. We can provide help with software installation during the lab hours.

2 Requirements

You have to modify existing operating system source code written in C and therefore you will be writing C code. The implementation of the scheduler should adhere to the following requirements:

- Make use of three run queues (high, normal and low), optionally also a queue for the idle process.
- Processes in each of the run queues should be scheduled using round robin scheduling.
- When scheduled, high priority processes are granted 1 time quantum, normal priority 2 time quanta and low priority 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.

The implementation of the message passing primitive should adhere to the following requirements:

- The message passing primitive should be direct and symmetric, so a sender must always explicitly specify a recipient and a recipient a sender.
- The payload of a message is a character string of at most 64 bytes.

- No buffering of messages should be done.
- The message passing primitive should be blocking. If a message cannot be received or sent at the time of the call, the process should be blocked. Once the receive or send operation can be carried out, the blocking sender or recipient process must be resumed.
- The implementation must properly handle locking.
- The implementation must make correct execution of the given *pingpong* and *abc* test cases possible.

3 Submission and Grading

You may work in teams of at most 2 persons. A single tar file should be handed in of the modified operating system containing the Multi-Level Feedback Queue scheduler and the message passing implementation. To hand in your work, remove any object files and binaries by removing the build directory. *Make sure that the files you have modified contain your names and student IDs.* Create a gzipped tar file of the source code directory:

```
tar -czvf assignment2.tar.gz assignment2/
```

Mail your tar files to *krietvel (at) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2015 Assignment 2". Include your names and student IDs in the e-mail.

Deadline: We expect your submissions *before* Sunday, April 5, 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 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.

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, which you can use during the lab hours on Wednesdays. Virtual machine images are available with the same software installation as the lab machines. During the lab hours we can help you to install the software on your own machine. Refer to the getting started guide, which is available on the website, for many more details. 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`.
- 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. Your data is also available from the "huisuil" machines through *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?

7 Round Robin Scheduling

We recommend to gradually work towards the final Multi-Level Feedback Queue scheduler and start by implementing Round Robin Scheduling as an intermediate step. 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.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 or zombie (`current->state` equals `PROC_BLOCKED` or

PROC_ZOMBIE), it is not ready to run and should not be placed on the run 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 run 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. 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. Can you explain why this is the case? To further improve the scheduler, you will continue to implement a more advanced Multi-Level Feedback Queue scheduler.

8 Multi-Level Feedback Queue

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 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 responsive.

9 Message Passing

The operating system that has been provided to you has already been extended with *msgsend* and *msgrecv* system calls. The calls take a *pid* and buffer location as arguments. *pid* indicates to which process to send or from which process to receive a message. If the specified process is not ready to send or receive, the calling process should be blocked. We will keep the implementation simple and ignore the problems of access control (should we allow a process to send messages to any other process?) and pid reuse (what happens if a process we sent a message to never tries to receive it, terminates and eventually another process will appear with the same pid which will call receive?).

You will have to complete the implementation of these two system calls by filling the `proc_msgsend()` and `proc_msgrecv()` functions at the bottom of `process.c`. A rough sketch of the implementation is as follows:

1. Check function arguments.
2. Check if the designated sender or recipient is blocked waiting to send or receive a message to the process currently executing send or receive. In this case, the message can be transferred immediately and after that the sender or recipient can be woken up by setting the process state to `PROC_READY` and putting it on the run queue.
3. Otherwise, the process currently executing send or receive needs to block until the designated sender or recipient will communicate with the process. Before blocking, correctly set the `msg_pid` and `msg_waiting` member fields in the process structure. This way the designated sender or recipient can determine whether this is the correct process to wake up.

Note that it is *strongly* recommended to first study the above sketch using pencil and paper before attempting to implement it!

To help your implementation, the process structure has been extended with a number of member fields: `msg_pid`, `msg_waiting`, `msg_buffer` and `msg_len`. `msg_waiting` must be set to one of `PROC_MSG_NO`, `PROC_MSG_RECEIVE`, `PROC_MSG_SEND`.

To test your implementation, two test programs are provided. Without implementing `proc_msgsend()` and `proc_msgrecv()` these tests will not work correctly. Verify this! Also study the source code of the test programs (`apps` subdirectory) to see that when implemented correctly, it is impossible for the processes to all be in their respective "critical" sections at the same time.

In *pingpong* process A will send *ping* to process B, which will reply to A with a *pong* message. If the message passing has been implemented correctly, A and B should be running one after the other. In *abc* three processes are created: A, B and C. Each process will print its name five times and then hand over control to the next process using message passing. So, you should see A run first, then B, then C and then A again. For more thorough testing, you can launch *stress* first and then launch *pingpong* or *abc*.