Operating Systems Design

Bachelor's Degree in
Computer Science and Engineering

University Carlos III of Madrid

Programming Assignment 1: Process Scheduling

Manuel F. Dolz Zaragozá
David del Rio Astorga
Alejandro Calderón Mateos

February 12, 2018

Contents

1 Introduction

The aim of this programming assignment is to develop a process scheduler in C language. This programming assignment is composed of several tasks. This tasks correspond to the different schedulers, each one having a different policy and features. For each task, the student must develop a C code fulfilling a set of given requirements. A final report about the assignment must also be written. Instructions on how to submit the work are indicated in the corresponding section.

2 Starting Code

An initial code is provided in the ZIP file *dssoo_p1_material.zip*. Inside, the following files can be found:

- *main.c*: main program which creates a number of threads with different priorities.
- *mythread.h*: header file which contains the TCB structure and several declarations of functions used in *mythreadlib.c*.
- *mythreadlib.c*: thread library code which contains a scheduler implementing a FIFO policy. **The student must use this file as starting point in order to develop the different schedulers. To do so, both scheduler and activator functions should be modified to implement the different scheduling policies.**

```
TCB* scheduler(){
...
}
void activator(TCB * next){
...
}
```

  The scheduler function should return the next thread to be executed while the activator function should perform the corresponding context swapping.

- *interrupt.c* and *interrupt.h*: Header and code files containing functions for managing interruptions.

```
void init_interrupt();
void enable_interrupt();
void disable_interrupt();
```

- *queue.c* and *queue.h*: Header and code files to create and manipulate thread queues.

```
struct queue * queue_new ();
struct queue* enqueue( struct queue*, void *);
void* dequeue( struct queue*);
int queue_empty (struct queue* s );
```

An example of creating, inserting and getting elements from a queue is the following:

```
struct queue *q = queue_new (); // Creates an empty queue
TCB *t = & t_state[0]; // take the first element from the TCB table
enqueue(q, t); // insert a TCB into the queue
TCB *s = dequeue (q); // remove a TCB from the queue
```

3 Student Work

3.1 Scheduling policies

Starting from the initial code provided, the student must develop three different schedulers:

3.1.1 Round-robin

Each thread will be executed for a number of ticks (slice), and afterwards, the scheduler will preempt the thread, i.e., it will leave the CPU. Then, other thread will be chosen to run on the CPU. Only when all other threads have been executed for a slice of time, the thread that was enqueued first will be executed again. The requirements of this scheduler are the following:

1. The number of ticks is determined by the constant `QUANTUM_TICKS` defined in *mythread.h* header file. The number of ticks remaining must be stored in the current thread entry of the TCB table, in the field `tick`.
2. After the thread consumes its quantum, the number of ticks for that thread is reset (in order to set up the next slice). Afterwards, the current thread should be preempted.
3. **The resulting file will be RR.c**

The starting code only implements a FIFO scheduler, i.e., only when the current thread finishes, the scheduler is called. In order to implement RR, the student must invoke the scheduler on each tick so as to check if a context switch is necessary.

3.1.2 Priority-based round-robin/FIFO

There will be two priorities (`HIGH_PRIORITY` and `LOW_PRIORITY`). The priority of the thread is a parameter of the *mythread_create* function. Threads with high priority (high-pri) will be scheduled following a FIFO policy, whereas threads with low priority (low-pri) will be scheduled following a Round-Robin policy. The requirements of this scheduler are the following:

1. If a high-pri thread is created on the system:
   - If the current (on execution) thread is a low-pri, then the high-pri thread will acquire the CPU immediately. The preempted thread (the low-pri) has to be enqueued in the low priority ready-state queue. Additionally, its ticks or quantum will also be reset.
   - If the current (on execution) thread is a high-pri, then the new thread should be enqueued in the high priority ready-state queue.
2. When a high-pri thread acquires the CPU, it will be the owner of the CPU from the beginning to the end of its execution (following the FIFO policy).

3. When there are no high-pri threads left, the scheduler will start dispatching low-pri threads following the Round-Robin policy.

4. **The resulting file will be RRF.c**

If a high-pri thread is created on the system, and the current (on execution) thread is a low-pri, then the high-pri thread will acquire the CPU immediately. The preempted thread (the low-pri) has to be enqueued in the low priority ready-state queue. Additionally, its ticks or quantum will also be reset.

3.1.3 Priority-based round-robin/FIFO with voluntary context switching

This scheduling policy is an extended version of the previous one. In this case, the scheduler should incorporate a new feature that allow the threads to perform voluntary context switching by using the *read_network* system call. The requirements of this scheduler are the following:

1. When the function *read_network* is called by a given thread, it should leave the CPU and introduced in a waiting queue.

2. Similarly to the clock interruption handler, the function *network_interrupt()* that emulates a NIC receiving a packet each second. When a new packet arrives, this handler should dequeue the first thread from the waiting queue and enqueue it in the ready queue corresponding to its priority. If there is no thread waiting, the packet should be discarded.

3. If there is no thread ready to run, but there are threads that have not finished their execution yet, the *idle* thread should run. This thread executes an infinite loop, so that the scheduler can check if there is any thread ready to run.

4. The id of the *idle* thread will be -1 and should not be introduced into a queue.

5. **The resulting file will be RRFN.c**

3.2 Output format

The following (and only the following) messages have to be printed on console:

- On swapping context between two threads due to end of slice.

```
*** SWAPCONTEXT FROM <source> to <dest>
```

- On swapping context due to the fact that current thread has finished.

```
*** THREAD <finished> FINISHED: SET CONTEXT OF <dest>
```

- On preemption of a low priority thread by a high priority thread.

```
*** THREAD <preempted> PREEMPTED: SET CONTEXT OF <dest>
```

- On a voluntary context switching when calling "read_network".

```
*** THREAD <current> READ FROM NETWORK
```

- On moving a thread from the waiting queue to its corresponding ready queue.

```
*** THREAD <ready> READY
```

- On a context switching from the idle thread to a thread ready to run.

```
*** THREAD READY: SET CONTEXT TO <ready>
```

- On finishing a thread

```
*** THREAD <finished> FINISHED
```

- Last thread finishes. No more threads remaining. Scheduler ends.

```
FINISH
```

The message regarding a thread that has finished its execution is already inserted in the code. The rest of messages should be included by the student.

- `source` refers to the thread ID whose ticks (slice) have been consumed (following the RR scheduling) or preempted by a higher priority thread, and should leave the CPU.
- `dest` refers to the ID of the new executing thread set by the scheduler.
- `finished` refers to the thread ID that finishes and leaves the CPU.
- `preempted` refers to the thread ID that is preempted from the CPU.
- `current` refers to the thread ID which is running at the moment.
- `ready` refers to the thread ID that has been moved from the waiting queue to its corresponding ready queue.

3.3 Requirements

Your solution **MUST** fulfill the following requirements:

1. You have to define and use ready queues containing all the threads that are not currently running, but ready to be scheduled.
   (a) For plain round-robin you need to use one queue and for the round robin/FIFO you need to use two queues (one for low priority tasks and other for high priority tasks).
   (b) Directly using the *t_state* array for scheduling (as in the provided code) will not be considered a correct solution. You must use queues for keeping active processes.
2. You have to protect the access to queues. In order to do that, you have use the functions *enable_interrupt* and *disable_interrupt*, disabling the interruptions during the queue processing (enqueue and dequeue operations). Explain in the report why this is necessary.
3. Your solution must use the context switching functions:
   - `setcontext`
   - `makecontext`
   - `getcontext`
   - `swapcontext`
4. The `scheduler` and `activator` functions can only be invoked by the `timer_interrupt`, `read_network`, `network_interrupt`, `mythread_exit` and, only for preemption, `mythread_create`.
   - `timer_interrupt`: when the timer interrupts
   - `read_network`: when a thread perform the system call and performs a voluntary context switch.
   - `network_interrupt`: when the NIC receives a new packet.
   - `mythread_exit`: when a thread ends.
   - `mythread_create`: when a new thread is created.
5. **IMPORTANT**: The program output will contain no more messages than those specified in section 3.2.
6. Context changes can only happen between two different process. It is forbidden (apart from inefficient) to swap the context of a thread for the same context of the same thread. Do not swap when the incoming and the outgoing threads are the same.
7. Errors control code should be included.

4 Evaluation

The final mark of this project is obtained as follows

- **Code (7 points)**
  - Round-Robin (*2 points*)
  - Round-Robin/FIFO with priorities (*2 points*)
  - Round-Robin/FIFO with voluntary context switching (*3 points*)

- **Report (3 points)**
  - Code design (*1.2 points*)
  - Test cases (*1.5 points*)
  - Quality of the report (*0.3 points*)

5 Assignment Submission

5.1 Deadline and method

You must submit your work using AulaGlobal. Submission deadline will be indicated on AulaGlobal assignment.

5.2 Submission

The submission must be done using AulaGlobal using the links available in the corresponding assignment section. The submission must be done separately for the code and the report.

5.3 Files to be submitted

You must submit the code in a zip compressed file with name:

- **dssoo_p1_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip**

where A...A, B...B, and C...C are the student identification numbers of the group. A maximum of 3 people is allowed per group. The file to be submitted must contain:

- **Makefile**
- **main.c**
- **mythread.h**
- **mythreadlib.c**
- **interrupt.c, interrupt.h**
- **queue.c, queue.h**
- **RR.c**
- **RRF.c**
- **RRFN.c**

The report must be submitted in a PDF file. Notice that only PDF files will be reviewed and marked. The file must be named:

- **dssoo_p1_AAAAAAAAA_BBBBBBBBB_CCCCCCCCC.pdf**.

A minimum report must contain:

- Description of the code detailing the main functions it is composed of. Do not include any source code in the report.
- Tests cases used and the obtained results. All test cases must be accompanied by a description with the motivation behind the tests. In this respect, there are three clarifications to take into account.

  - Avoid duplicated tests that target the same code paths with equivalent input parameters.
  - Passing a single test does guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, nor the number of tests per program.
  - Compiling without warnings does not guarantee that the program fulfills the requirements.

- Conclusions, describing the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the realization of this assignment.

Additionally, marks will be given attending to the quality of the report. Consider that a minimum report:

- Must contain a title page with the name of the authors and their student identification numbers.
- Must contain an index.
- Every page except the title page must be numbered.
- Text must be justified.
- Max. 10 pages.

The PDF file must be submitted using the TURNITIN link. Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

5.4 Rules

1. Programs that do not compile or do not satisfy the requirements will receive a grade of zero.
2. All programs should compile without reporting any warnings. In case any warning is reported during the compilation, the final mark will be reduced.
3. The assignment must be submitted using the available links in AulaGlobal. Submitting the assignments by mail is not allowed.
4. Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be applied.
5. The submitted code has to work on *guernika* server. It is recommended to test your code in there before submitting.
6. **IMPORTANT:** A group member should notify, before **March 3rd**, the lab teacher by email with the following information about the group members: name, e-mail, group and NIA.