OPERATING SYSTEMS DESIGN

BACHELOR’S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

CARLOS III UNIVERSITY OF MADRID

Programming assignment 1

Process Scheduling

Ramiro Pinto Prieto
100346042

Tomas Mataitis
100390517

Aurimas Klimavicius
100390516

March 19th, 2018

TABLE OF CONTENT

1 INTRODUCTION

Main purpose of our 1st programming assignment of Operating Systems Design is to develop a process scheduler using C programming language. Our assignment tasks are mainly divided in 4 parts:

- Initial code
- Round-Robin
- Priority-Based Round-Robin
- Priority-Based Round-Robin with voluntary context switching

Final report after fulfilling the tasks should have 2 main result sections:

- Description of the codes. In this section we will have 4 subsections for each part describing each of the code with modifications made and policies implemented of scheduling
- Test cases. This part of report will help us to see if our codes works as they are supposed to and outputs are good

Each of those parts had their own scheduling policies, starting from the initial code we have to develop 3 different schedulers for them. By following the requirements we need to fulfill each task by developing C code.

All the schedulers will be made from starting point – initial code, which is provided by our professor. With the initial code scheduler and activator functions need to be modified in purpose to implement three different policies of scheduling.

Lastly we will need to try all the inputs of separate parts to see if everything works as expected just by executing the initial code and write all the obtained results in tables with brief description behind the tests.

2 Description of the code

2.1 Initial code (FIFO)[1]

The whole practical assignment was started from initial code which was provided by our professor and contained 4 C source files, 3 header files and Makefile. 3 different schedulers: Round-Robin, Priority-Based Round-Robin and Priority-Based Round-Robin with voluntary context switching were developed by writing each of them different policies from initial code. We worked with these files:

- 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
- Interrupt.c - code file containing functions for managing interruptions
- Interrupt.h - header file containing functions for managing interruptions
- Queue.c - code file to create and manipulate thread queues
- Queue.h - header file to create and manipulate thread queues
- Makefile

More information about initial code description can be found in references.

2.2 Round-robin

This exercise was developed using as starting point the initial provided code. So that the main structures and functions used have been the same. The initial code has been described in the previous point (See point 2.1.) and in the practical assignment statement. We will focus on describing the changes implemented to carry out the round-robin scheduling policy.

This policy is based on assigning equal time slices to each process in order to execute them in a non-priority circular order. In our case, the time slice is set to 40 ticks and it is defined in the *QUANTUM_TICKS* constant (*mythread.h*).

One of the main differences of this exercise is that we use a queue for storing the ready processes, which are the processes ready to execute that has not been completed. This queue is called *processes_q*. It is declared as a global variable and it is initialized within *init_mythreadlib* function. We also added some error handling to this function.

Another functions needed to be modified were: *mythread_create*, which after inserting the process into the *t_state* waiting matrix and setting its attributes, it enqueue the process to the ready queue; and *mythread_exit*, where we need to add the condition that if there is no processes in queue the program stops, otherwise, we execute next process.

[1] When we talk about processes we are referring to the threads which independently execute the functions described in *main.c*. These threads are used to simulate the process scheduling.

The core functions to implement the new scheduler policy are: *scheduler*, *timer_interrupt* and *activator*.

- **Scheduler:** This function returns the next thread to be executed. In order to do this for the round-robin policy, first, we enqueue the running process in case that is has not ended in the ready queue (*processes_q*). Then, if ready queue is not empty, we dequeue the next element and return it. Otherwise, we return *idle* process.

- **Timer_interrupt:** This function is used to manage time slices of each process. Each time we invoke this function by a clock interrupt, we reduce by one the ticks remaining to finish the process. If no ticks remains we invoke the scheduler and activator to execute the next instruction. In order to protect the access to queues, before invoking the scheduler and activator, we disable the interrupts (*disable_interrupt* and *disable_network_interrupt*).

- **Activator:** This function perform the corresponding context swapping to the thread received as argument. To do it, first, we declare a temporal TCB struct (*temp*) to store the running process. Then we update the running process to the next one. If *temp* state is *FREE*, that means that the process has finished its execution, so we set the context of next process (*setcontext*). Otherwise, we swap context from temp to next process, in order to been able to continue the execution of the previous process (*temp*). We also add some conditions to avoid context swapping of same process and to differentiate a standard process from *idle* process.

2.3 PRIORITY-BASED ROUND-ROBIN/FIFO

This exercise was created according round-robin scheduling policy from previous exercise (2.2) but with ability to set priority of executing threads.

This policy is based on priorities of threads, which are set in the mythread_create function. They are executing by priorities. First thread is executing if it has HIGH_PRIORITY, then goes thread with LOW_PRIORITY.

One of the main differences of this exercise is that we use two queues (hp_q - for high priority, lp_q - for low priority) for threads in order to ensure their executing by priority. They are declared as a global variables and they are initialized inside *init_mythreadlib* function. We also added some error handling to this function.

Another functions needed to be modified were: *mythread_create* - there was added code to put threads in appropriate queue by their priority; and *mythread_exit*, where we need to add the condition that if there is no processes in two queues the program stops, otherwise, we execute next process, timer_interrupt - here we added only one condition to the if that only threads with low priority could be released in order to execute high priority thread.

The core functions to implement the new scheduler policy are: *scheduler* and *activator*.

- **Scheduler:** This function returns the next thread to be executed. In order to do this for the priority-based round-robin/fifo policy, first, we enqueue the running process, according to its priority, in case that is has not ended in the ready queue (*hp_q, lp_q*). Then, if ready queue is not empty, according to the priority, we dequeue the next element and return it. Otherwise, we return *idle* process.

- **Activator:** This function perform the corresponding context swapping to the thread received as argument. It is similar to the to the round-robin, however it includes context swaping by priorities: if current thread has low priority and the next one has high priority, then we allow context swap.

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

This exercise was created according priority-based round-robin/fifo scheduling policy from previous exercise (2.3) but with voluntary context switching using network interruptions.

This policy is still based on priorities of threads as in the previous one and it also has two new functions (network_read, network_interrupt) which allows us to fulfil the policies.

One of the main differences of this exercise is that we use two new methods such as network_read - for putting given thread into a waiting list and leave the cpu and network_interrupt - for withdrawing thread from waiting list and putting it into ready queue according to its priority. Moreover, we have one additional waiting queue for that - w_q it puts threads there when the network_read function is called.

Scheduler function was left as it is in order to preserve the logic of priority-based round-robin/fifo policy. The main functionality was implemented in network_read and network_interrupt functions.

Activator function was also left as it is in order to preserve the logic of priority-based round-robin/fifo policy.

3 Tests cases

3.1 Initial code (FIFO)

	Input	Reason	Expected output	Result
1	main.c	Ready message	No message shown	Success
2	main.c	Swap context	There is no swap context	Success
3	main.c	Thread finish and set context to next thread.	**** THREAD N FINISHED* but there is no SET CONTEXT message	Success
4	main.c	Execution order. The program should follow the arrival order.	EXecution of threads in arrival order (FIFO)	Success
5	main.c	Finish message when there is no process to execute	No message implemented	Success
6	Infinite process	Starvation	There is starvation	Success
7	main.c	Read from network message	Not supposed to be shown	Success
8	main.c	Context switching from the idle thread to a thread ready to run.	Not message shown	Success

Table 1

3.2 RR

	Input	Reason	Expected output	Result
1	main.c	Ready message when enqueueing into ready queue	**** THREAD N READY* (N from 1 to 7)	Success
2	main.c	Swap context of all processes when a process completes its time slice inside the CPU	**** SWAPCONTEXT FROM N TO M* Of all processes which *FINISHED* message has not been shown	Success
3	main.c	Thread finish and set context to next thread.	**** THREAD N FINISHED* **** THREAD N FINISHED: SET CONTEXT OF M*	Success
4	main.c	Execution order. The program should follow the arrival order.	Ready, swap, finish and set messages showing threads in arrival order (1,2,3,4,5,6,7)	Success
5	main.c	Finish message when there is no process to execute	*FINISH*	Success
6	Infinite process	Starvation	No starvation as Round robin is supposed to be starvation-free	Success
7	main.c	Read from network message	Not supposed to be shown	Success
8	main.c	Context switching from the idle thread to a thread ready to run.	**** THREAD READY: SET CONTEXT TO <ready>*	This situation never happens.

Table 2

3.3 RRF

	Input	Reason	Expected output	Result
1	main.c	Ready message when enqueueing into ready queue	*** THREAD N READY (N from 1 to 7)	Success
2	main.c	Execute first high priority processes (4,5,6,7) and FIFO execution of this processes, that means, without stopping the process execution.	**** THREAD 4 FINISHED* **** THREAD 4 FINISHED: SET CONTEXT OF 5* **** THREAD 5 FINISHED* **** THREAD 5 FINISHED: SET CONTEXT OF 6* **** THREAD 6 FINISHED* **** THREAD 6 FINISHED: SET CONTEXT OF 7* **** THREAD 7 FINISHED*	Success
3	main.c	Execute low priority processes in order (0,1,2,3)	**** THREAD 7 FINISHED: SET CONTEXT OF 1*	Success
4	main.c	Low priority processes executed in round-robin policy. Swap context of all processes when a process completes its time slice inside the CPU	**** SWAPCONTEXT FROM N TO M* Of all processes which *FINISHED* message has not been shown	Success
5	main.c	Thread finish and set context to next thread.	**** THREAD N FINISHED* **** THREAD N FINISHED: SET CONTEXT OF M*	Success
6	main.c	Finish message when there is no process to execute	*FINISH*	Success
7	Infinite high priority process	Starvation is not controlled	Error. Starvation, process stuck in CPU	Output as expected but it is not a good performance
8	High priority process while executing low priority ones	Preempted message	**** THREAD <preempted > PREEMPTED: SET CONTEXT OF <dest >*	Success
9	main.c	Read from network message	Not supposed to be shown	Success
10	main.c	Context switching from the idle thread to a thread ready to run.	**** THREAD READY: SET CONTEXT TO <ready>*	This situation never happens.

Table 3

3.4 RRFN

	Input	Reason	Expected output	Result
1	main.c	Ready message when enqueueing into ready queue	*** THREAD N READY (N from 1 to 7)	Success
2	main.c	Execute first high priority processes (4,5,6,7) and FIFO execution of this processes, that means, without stopping the process execution.	**** THREAD 4 FINISHED* **** THREAD 4 FINISHED: SET CONTEXT OF 5* **** THREAD 5 FINISHED* **** THREAD 5 FINISHED: SET CONTEXT OF 6* **** THREAD 6 FINISHED* **** THREAD 6 FINISHED: SET CONTEXT OF 7* **** THREAD 7 FINISHED*	Error. Thread 0 appears as firstly as finished, then everything is correct.
3	main.c	Read from network message	*** THREAD 0 READ FROM NETWORK *** THREAD 1 READ FROM NETWORK *** THREAD 2 READ FROM NETWORK *** THREAD 3 READ FROM NETWORK *** THREAD 4 READ FROM NETWORK *** THREAD 5 READ FROM NETWORK *** THREAD 6 READ FROM NETWORK *** THREAD 7 READ FROM NETWORK	Success
4	main.c	Context switching from the idle thread to a thread ready to run.	**** THREAD READY: SET CONTEXT TO <ready>*	This situation never happens.
5	main.c	Finish message when there is no process to execute	*FINISH*	Segmentation fault (core dumped)

Table 4

4 Conclusions

After finishing practical assignment of process scheduling we learned how to develop different schedulers and to test them. It helped us to familiarize and deepen our knowledge with C programming language. This assignment has been big opportunity to put into practice what we have learned in our theoretical classes during first half of the semester and see the applications that it has in the real world while we learn about C programming language, which is currently one of the widest used and powerful programming languages and seems to be a very powerful language for developing schedulers in this type of problems.

At the end of this assignment we have had developed 3 different process schedulers in C language just by using the initial code provided by professor. Each of those schedulers had different policies with different features. We have defined the queues containing threads which weren't running at that moment, but were ready to be scheduled, which ones we used later. Plain Round-Robin used only one queue for handling low priority tasks and Round-Robin/FIFO used two in case of high priority tasks. Protection of the access to the queues have been made by disabling interruptions during the process to obviate asynchrony, because it is a concurrent process.

We accomplished the initial goals of this practical assignment and gained a lot of knowledge in the process. However, we have had to face some problems during the development of the exercises as a consequence of not being familiar with:

- Initial code complexity
- Understanding how everything works
- The tools of implementation
- Network interruptions

Despite all these problems, we have carried out the practical assignment in the best way that we could and now we think that we have a greater understanding of the theoretical concepts seen in class because we tried to prove ourselves in the best way by using knowledge acquired there. Finally, doing this practical work we were acquired with new tools, which will help us with further tasks and practical assignments. This first assignment of process scheduling was challenging but it taught us a lot.

5 References

- Guide to Programming Assignment 1: Process Scheduling. Access through the internet: https://aulaglobal.uc3m.es/
- Wikipedia. (2018, March 10). Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Round-robin_scheduling