Operating Systems Design

Programming assignment 1:
Process Scheduling

Application of process planification algorithms with the use of queues

Group 13
Luis Rodríguez Rubio - 100383365
Cristian Sevillano Vicente - 100383513
Ignacio Talavera Cepeda - 100383487

Index

Introduction

This document will serve to keep record of the work done in the first programming assignment, Process Scheduling, a Round-Robin algorithm will be created following and further extending the code provided in Aula Global.

Three versions of the algorithm will be defined, in each section, both the code description and design will be thoroughly explained:

- Plain Round-Robin:
  The simplest of the algorithms, where time slices (also known as time quanta) are assigned in equal portions to each of the processes following a circular order.

- Round-Robin with priority / SJF:
  New version of the round-robin, with the feature of priority differentiation. Now with two queues instead of one, one will apply a round-robin scheduler to low priority processes and treating the high priority process with a Shortest Job First algorithm.

- Round-Robin with (potential) voluntary context switching / SJF:
  This scheduling policy is an extended version of the later, with the added *read_disk()* system call which allows the thread to perform a potential voluntary context switch. The threads may have to access data that is not loaded into the cache memory, so we simulate the interruption that would appear in this case, performing a voluntary context switch to load this data into memory.

Once a version is finished and functional, a series of unique test cases will be carried on to identify errors and incorrect behavior in order to improve the code and a more polished program.

At the end of the document a conclusion can be found, detailing the main problems of the assignment as well any personal comment that the group may add.

Part 1: Plain Round-Robin

Description

As it was introduced, this first approach includes only one scheduler: the round-robin. It is very popular, employed mainly by process and network schedulers, and its concept is very simple, yet very powerful: time slices, or quanta, are assigned to each process in equal amounts and in circular order. Doing so, it is able to handle processes in a cyclic executive. Round-robin scheduling is simple, easy to implement, starvation-free, but without the possibility to add prioritization by itself. The name was given after the famous term dates from the 17th-century French *ruban rond*, which was the practice of signatories to petitions against authority (usually the Crown) by appending several names on a document in a non-hierarchical circle or ribbon pattern so no one was identified as a leader.

By giving each process a time slot, the round-robin is able of interrupting the job if its not completed when it finishes. The job is resumed when a full circle has been made through the rest of the processes, and it is again the turn of that specific process.

In our implementation, we depend on three main functions in order to make the scheduler work. Prior to them, we need to initialize the threads. Thread 0 will perform that, and then the circular process will start. Each thread , when created, is given the state 'INIT', and its remaining ticks are setted. After that, each thread is introduced in our queue, the data structure used to control the scheduler. Each time a thread is executed, it is dequeued. When it finishes, the thread is terminated, but when it is expelled because of the quanta, it is enqueued again.

The scheduler is the function that returns the next thread to be executed. It returns the same thread if the slice has not finished, and the next in the queue if the quanta or the thread itself has finished. If the quanta has finished, it will enqueue the thread that was running, again in the queue; but if it the thread is the one that has ended, that thread will be terminated.

Working along with the scheduler, the activator controls the contexts of the threads. They are the information used by the thread to operate, and it must be saved, setted and swapped correctly. If the current thread has finished, the activator will set the context of the next one, but if it the quanta is the one that has finished, we perform a swapcontext() which consist of saving the actual context in its corresponding thread structure and restoring the context of the next one, as it will be resumed sooner or later. It will also control the behaviour of the contexts when neither the thread nor the quanta has finished, and no dequeuing nor swapping must be made.

Design

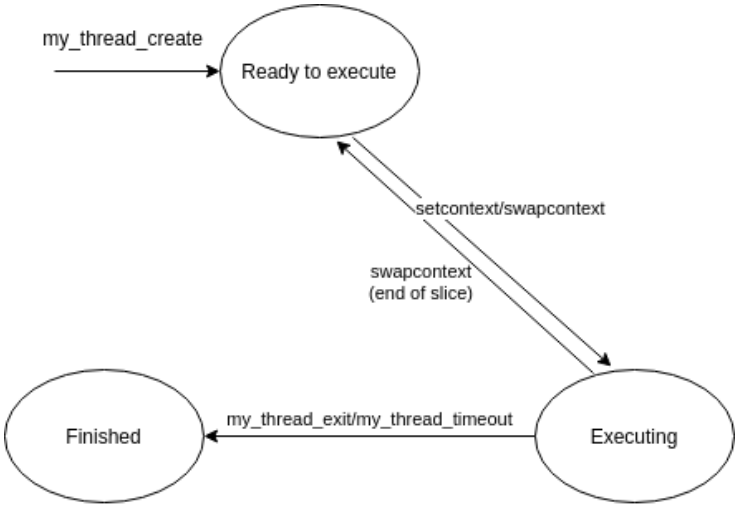

Figure 1: Round-Robin execution design

This Finite State Machine represent the different states that a thread can adopt in our scheduler. Once created, all threads are ready to execute. When the scheduler returns them, they move into executing by a setcontext. Once in executing, they could return to ready to execute if the quanta finishes (by a swap context, and then it could come again to executing by another swap context), or it could go to finished by a my_thread_exit or my_thread_timeout.

Test Cases

Test	Description	Expected Result	Obtained Result
Basic test	Functionality test	Threads are executed	Same as expected

		following RR scheduler without any error.	result.
Remaining ticks less than Quantum	Remaining ticks are now smaller than the quantum.	As now the threads can be completed, FIFO-lie behaviour is expected	Same as expected result.
Quantum = 2	Quantum is set to a very small value.	A lot of swap context must be done in order to complete the threat	Same as expected result.
Quantum = -1	Quantum is set to a negative value	The threads are terminated directly as the remaining tick is a fraction of the quanto	Same as expected result.
Remaining ticks equal than Quantum	Remaining ticks are now the same as the quantum.	The threads should be terminated in just one quantum	Same as expected result.

Part 2: Round-Robin with priority / SJF

Description

In this case we will be using two queues, one for low priority threads and the other for high priority threads, then the thread 0 is called, it enqueues the threads in its correspondent queue. However, in this case we are working with a low priority thread 0, meaning that if a high priority thread is enqueued, the context is immediately swapped to that thread and once it is checked as 'FREE', meaning that it has finished, the context is swapped back to the thread 0 (to keep enqueuing the rest of the threads for its future executing).

If all the threads are already queued, the program starts executing the thread with fewer remaining ticks in the high priority queue until this is empty, from this point on it works as the plain Round-Robin algorithm that has been explained in Part 1 of the document for the low priority queue.

The high-priority queue works with another type of scheduler: the Shortest-Job First. This scheduling policy selects the waiting process with the smallest time-to-finish (remaining_ticks). We are working with a preemptive version of this algorithm, and if a high priority thread with a lower execution time comes, the running process must be queued in the high-priority queue, and the new process will acquire the CPU immediately.

As it can be observed in the output, it appears that the thread 0 and 1 finishes, with the last one finishing 5 times, this happens when thread 0 enqueues a high priority thread and instantly changes context to that same thread, executing it until it calls **my_thread_exit** and it is marked as 'FREE'. Having said that, once the program goes back to thread 0 in order to enqueue the rest of the threads, it actually enqueues them in thread 1 as it is marked as 'FREE'. This process is completely normal. The process tables have a limited size, so the logical performance implies that the processes reuse entries. It's just an incorrect visual representation as the program is running as expected.

The scheduler now is divided into two functions: a Round-robin function with the same behaviour as the first scheduler, and the implemented SJF function for the high-priority queue. The scheduler decides which function needs to be called checking if the high priority queue is empty or not. The activator is very similar, but now it covers the cases bringed by the two queues.

We use the sorted_enqueue function in order to ensure that the high-priority queue is always sorted, as on it the SJF will depend to function correctly.

Design

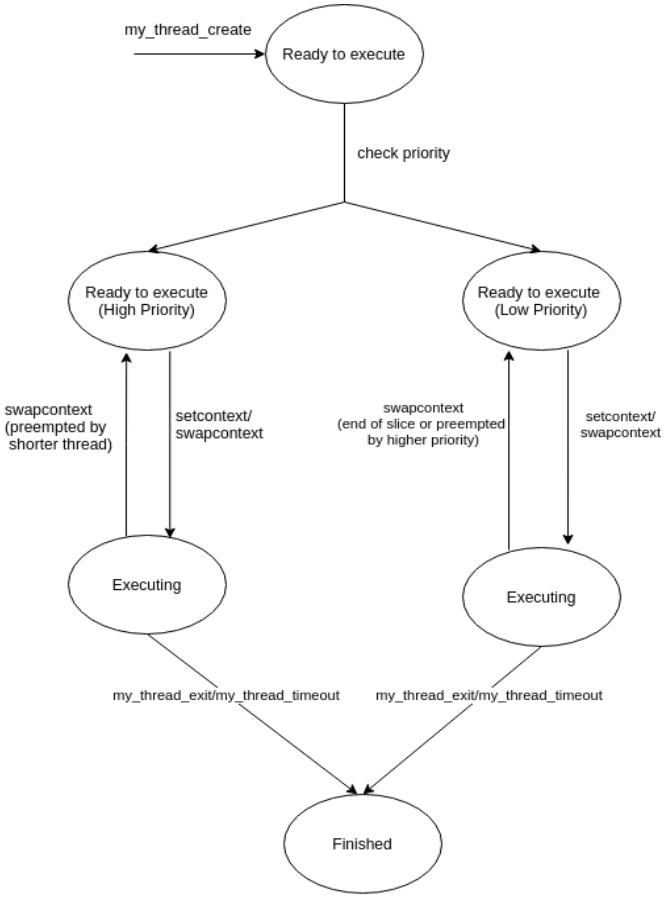

Figure 2: Priority based Round-Robin execution design

This Finite State Machine represent the different states that a thread can adopt in our scheduler. We have decided to make different states for the two priorities, even though the state is technically the same (ready to execute). The part of the low priority is very similar to the FSM performed in the Round-robin scheduler. The high-priority, ready to execute threads will go into the CPU by a set context (if no other thread was on the CPU before) or by a swap context (if other thread is expelled from the CPU). Once in execution, it could be preempted by a high-priority thread with lower execution time (and sent back to the ready to execute state), or it could be terminated if the thread has ended (using my_thread_exit or my_thread_timeout).

Test Cases

Test	Description	Expected Result	Obtained Result
Basic test	Functionality test	Threads are executed following RR scheduler without any error.	Same as expected result.
Remaining ticks less than Quantum	Remaining ticks are now smaller than the quantum.	As now the threads can be completed, FIFO-lie behaviour is expected	Same as expected result.
Quantum = 2	Quantum is set to a very small value.	A lot of swap context must be done in order to complete the threat	Same as expected result.
Quantum = -1	Quantum is set to a negative value	The threads are terminated directly as the remaining tick is a fraction of the quanto	Same as expected result.
Remaining ticks equal than Quantum	Remaining ticks are now the same as the quantum.	The threads should be terminated in just one quantum	Same as expected result.
Only low-pri threads	There will be only low priority threads created	The scheduler should behave as the basic test of the Round-robin scheduler	Same as expected result.
Only high-pri for the last thread	There will be only a high-priority thread (as the last)	All threads are added to the queue, and then the high-pri is executed. Lastly, RR terminates the rest.	Same as expected result.

Part 3: Round-Robin with voluntary context switching / SJF

Description

This scheduling policy is an extended version of the previous one, and suppose the culmination of this work. If implemented in a real-life scenario, this scheduler would behave sophisticatedly, allowing threads with different priorities and voluntary context switch due to a miss on cache memory. Voluntary context switching consist on a process explicitly yielding the CPU to another. In our case, this may happen because of a simulation of a cache miss: the CPU will need to acquire a process to read some data from main memory.

When the function read_disk is called by a given thread, another function called data_in_page_cache will determine if a memory read must be performed or not. If so, the process must leave the CPU immediately

and be introduced in another queue named waiting. After it, the function disk_interrupt emulates the arrival of a hardware interrupt from a disk controller, which notifies if that memory read is available or not. When this interrupt arrives, the handler should dequeue the first thread from the waiting queue and enqueue it in the ready queue corresponding to its priority.

We have to cover, also, the case in which there are not threads ready to run but there are threads that has not finished yet (in waiting queue). For this purpose, the idle thread (with identificator -1) is introduced. This thread executes an infinite loop so that the scheduler can check if there is any thread ready to run.

In this scheduler, the behaviour of the Round-robin and SJF parts behaves very similarly, but now the new queue has been introduced, along with the idle thread. Both schedulers have now a new case when the running thread is the idle one, in order to wait for a thread to be ready to execute when the interrupt is made.

The activator have also contemplated this case, and swaps the context to the idle thread when necessary. Also, when changing from the Idle thread to another there is no need to save the idle context as it only executes an infinite loop, it would be a waste of resources.

Design

Figure 3: Priority based Round-Robin with voluntary context switching execution design

This Finite State Machine represent the different states that a thread can adopt in our scheduler. In order to make voluntary context switches we need to make use of a new state called waiting, this state will be if a read_disk happens when a thread is in executing, it will me moved to waiting until a disk_interrupt returns it to one of the ready to execute states based on the priority assigned to that thread.

Test Cases

Test	Description	Expected Result	Obtained Result
Basic test	Functionality test	Threads are executed following RR scheduler without any error.	Same as expected result.
Remaining ticks less than Quantum	Remaining ticks are now smaller than the quantum.	As now the threads can be completed, FIFO-lie behaviour is expected	Same as expected result.
Quantum = 2	Quantum is set to a very small value.	A lot of swap context must be done in order to complete the threat	Same as expected result.
Quantum = -1	Quantum is set to a negative value	The threads are terminated directly as the remaining tick is a fraction of the quanto	Same as expected result.
Remaining ticks equal than Quantum	Remaining ticks are now the same as the quantum.	The threads should be terminated in just one quantum	Same as expected result.
Only low-pri threads	There will be only low priority threads created	The scheduler should behave as the basic test of the Round-robin scheduler	Same as expected result.

Only high-pri for the last thread	There will be only a high-priority thread (as the last)	All threads are added to the queue, and then the high-pri is executed. Lastly, RR terminates the rest.	Same as expected result.
Thread 0 will have high-priority	Only the thread 0 is a high-priority thread	Thread 0 will be executed first with an SFJ, it will enqueue every other thread before swapping context.	Same as expected result.
Read_disk = 1	Perform several read_disk() one after another	Perform correctly every swapcontext() on different states of the queues	Same as expected
Perform read_disk at the start of the execution of thread 0	Testing of idle thread behaviour and corresponding prints.	Swaps to the idle thread and it waits for the disk_interrupt in order to continue with thread 0	Same as expected result.

Conclusion

The work done about schedulers a has been very satisfying, and it is very easy to see how powerful and useful these procedures are to the design of operative system. Schedulers like the ones we coded are working in the background of our computers.

This assignment supposed a lot of investigation, documentation and trying. We developed many schedulers that, in the end, were very far from the one that did get the job done. But, as we tried, failed, a

process of debugging and rethinking lead to a better understanding of the performance of this schedulers and how they should be implemented. It was never frustrating though, as we always knew in which direction we may find the solutions, and it was just a matter of trying until we get the three of the schedulers up and running.

This lab assignment has been really enriching. Voluntary context switching was a totally new concept for us and it is always exciting to discover new fields that can have so much potential solving real-life problems; and the different implementation of schedulers was the perfect way to learn, by designing and coding, such an important algorithms. We are looking forward to your feedback and to the next labs and assignments.