MARÍA DEL CARMEN DÍAZ DE MERA GÓMEZ-LIMÓN - 100383384
MARINA TORELLI POSTIGO - 100383479

PROCESS SCHEDULING

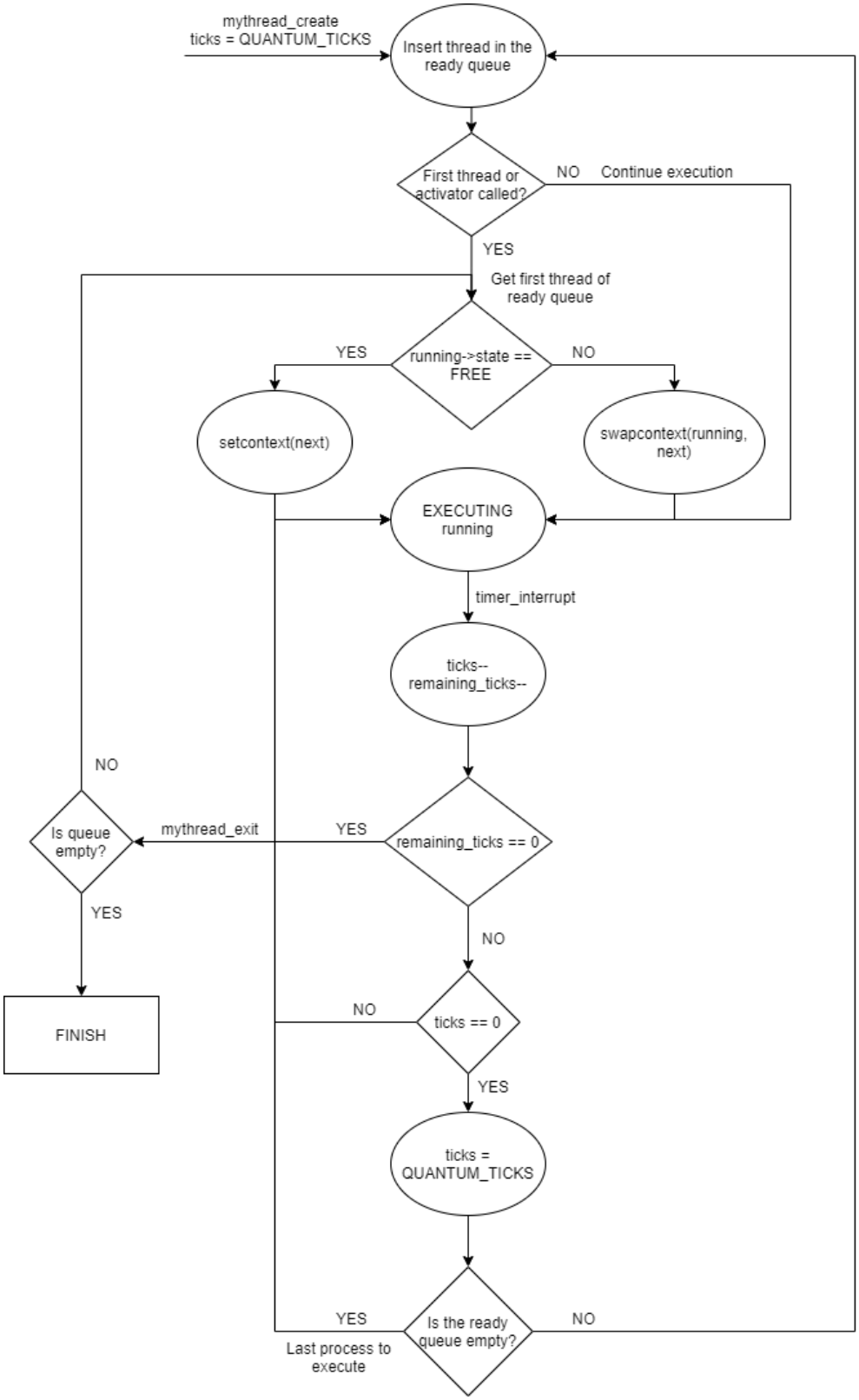
GROUP 89
PROGRAMMING ASSIGNMENT 1
OPERATING SYSTEMS DESIGN
COMPUTER SCIENCE AND ENGINEERING
UNIVERSIDAD CARLOS III DE MADRID

REPORT

TABLE OF CONTENTS

1. ROUND-ROBIN

1.1. Design



In the design we included the call to mythread_exit when the remaining ticks reach 0, without considering a possible timeout. This however was done differently in the code, as mythread_exit is called from function_thread and we used mythread_timeout in the timer_interrupt in the case there was some error and a thread exceeded its execution time and its remaining_ticks went below 0.

1.2. Description of the code

To perform round-robin we need a queue to store the ready processes, which is declared globally and initialized in init_mythreadlib(). When a new process is created we store the value of QUANTUM_TICKS in the 'ticks' field of the TCB and enqueue the new thread in the ready queue. Each time we access the queue to do an enqueue or dequeue operation we have to disable the timer interrupt and disk interrupts. This is done because those functions could potentially access and modify the queue (in the case of disk interrupt it could only happen in the scheduler of section 3 but we disabled them in the other two schedulers as a good practice), causing undefined behavior if several functions access the queue at the same time.

In the timer interrupt we update the value of the 'ticks' and 'remaining_ticks' fields of the running thread, subtracting one to them. First we check that there wasn't any error and the running thread exceeded its executing time. For that, if the value of remaining_ticks reaches a value smaller than 0, we call the timeout function to eject the thread. Otherwise, if the value of 'ticks' reaches 0, we restore the quantum and store the running process again in the ready queue. We get the next process from the scheduler and perform the change with the activator. We only do this if the ready queue is not empty, if it is empty then the running process is the only one remaining, so we do not need to perform a swap.

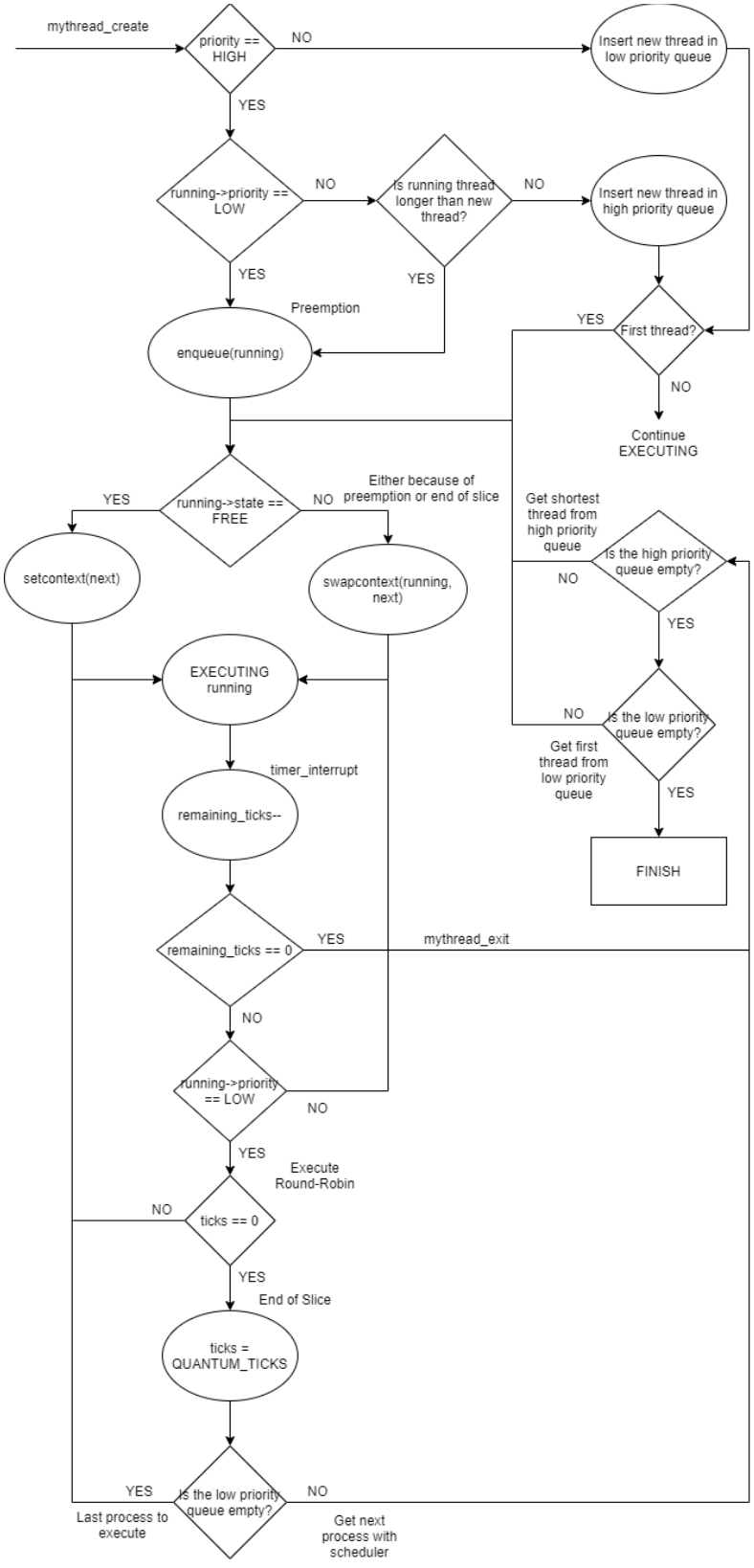
In the function activator we update the global variables running and current and set the context of the next process. If this function was called from mythread_exit then the value of the state of the running thread will be FREE, which means it finished. The function prints the corresponding message and uses setcontext with the next thread, checking if that function returns any error. In the rest of the cases the activator was called due to the end of the slice of the running process, so the function prints the corresponding message and performs a swapcontext, also checking for errors.

1.3. Test cases

ID	Objective	Procedure	Expected Output
RR-01	General functionality test	Execute the *main.c* file provided by the statement	There is a swap between the threads in the order they were created until they all finish.
RR-02	Thread 0 must finish within the first slice	Remove the for loop at the end of main.c to shorten the execution time of thread 0.	Thread 0 will finish immediately before the end of its slice after creating the rest of the threads.
RR-03	There must be no swaps from one thread to itself	Increase the execution time from 2s to 5s of thread 3.	Thread 3 will still run for a few slices after the rest of the threads have finished, without swaps.
RR-04	Functionality of TimeOut function	Modify the function_thread and enter one thread 2 into an infinite loop where the exit function is never called.	When the execution time of thread 2 is exceeded it is ejected from the CPU and the next thread is executed.

2. PRIORITY-BASED ROUND-ROBIN/SJF

2.1. Design



In this design we will use a sorted enqueue every time we enqueue a high priority thread in its corresponding queue. This queue will be sorted according to the remaining ticks of each thread, so to get the shortest job first we just have to do a dequeuer operation.

Furthermore, one of the consequences of performing preemption when each thread is created is that if thread 0 has low priority each time it creates a high priority thread it will be removed from the CPU, not being able to create any more threads. Only when the created high priority thread finishes it will be able to resume creating the threads. The next created thread will have the same tid than the one that just finished, as its TCB was freed before creating the new thread.

2.2. Description of the code

Instead of one queue, we will declare globally two queues, one for high priority processes and one for low priority, which will be initialized in init_mythreadlib(). When we create a new thread we have to check its priority. If the new process has low priority no preemption needs to be considered, so we enqueue it in its corresponding queue. If it is a high priority process, we have to consider all the cases where preemption would occur as explained in the statement.

If the running process has low priority, the new process will acquire the CPU immediately, so we restore the quantum of the running process and enqueue it in the low priority queue, activating the newly created process. If the running process has also high priority, we check if the total execution ticks of the new process is smaller than the remaining ticks of the running process. If the new process is shorter then it will also acquire the CPU immediately, performing a sorted enqueue in the high priority queue of the preempted process according to its remaining ticks and activating the new process. If we don't need to consider preemption we perform a sorted enqueuer of the new process in the high priority queue according to its execution total ticks.

The timer interrupt updates the remaining ticks of the running process and in the case it has low priority, executes the round-robin as in the same function of the first section. The activator works the same as in round-robin, only checking before the swapcontext if the priority of the next thread is high. If it is, then that means that the previous thread is preempted, as it still has not finished, so we print the corresponding message about preemption. In the statement this message is stated only in the cases of preemption of a low priority thread by a high priority one, but we also printed it in the case of preemption of a high priority thread by a shorter high priority thread.

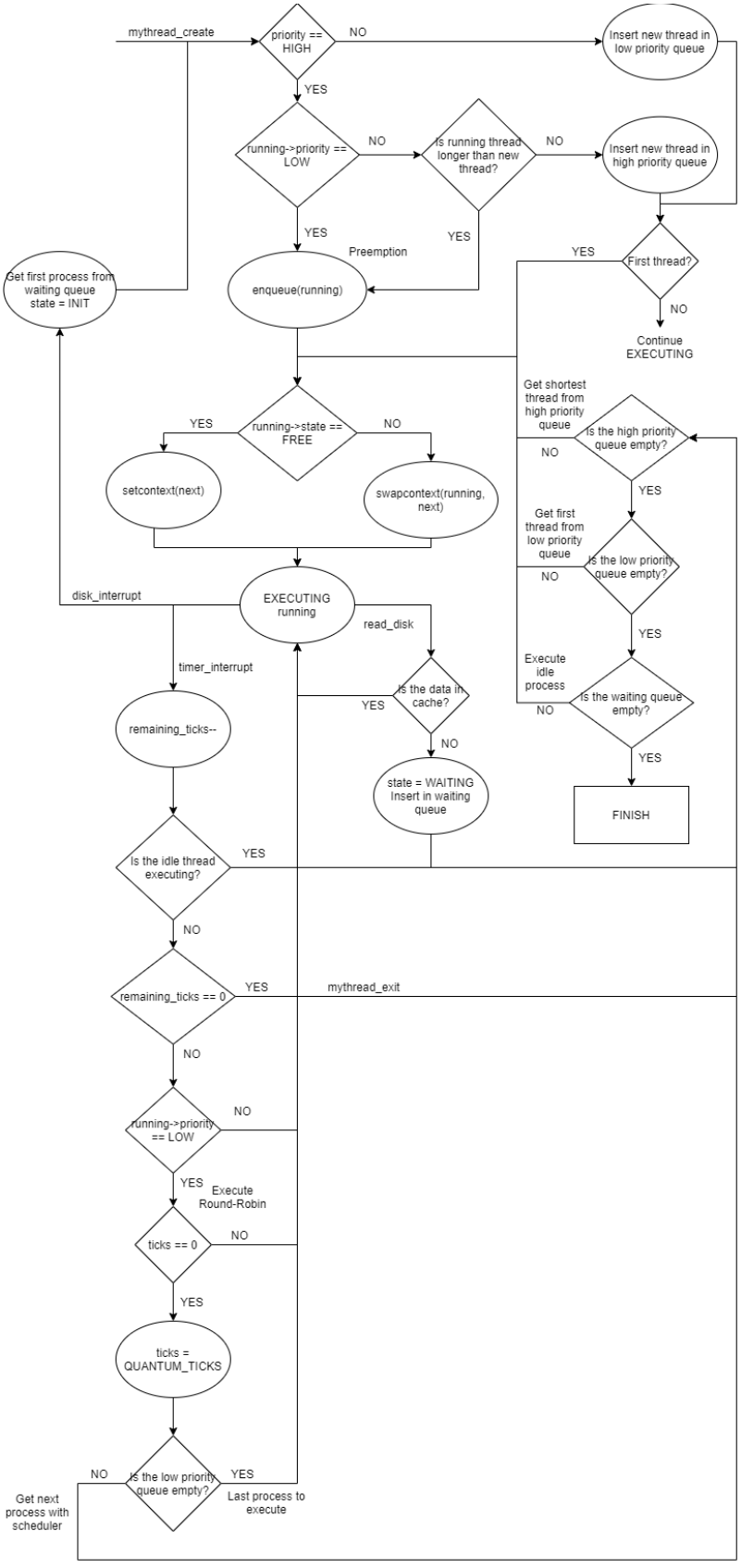
2.3. Test cases

All functionalities that have already been tested in previous cases, will not be considered for testing again. This is because we have used the same code from the first section, but it has been modified and extended.

ID	Objective	Procedure	Expected Output
RRS-01	General functionality test	Execute the *main.c* file provided by the statement	As thread 0 has low priority it is ejected every time it creates a high priority thread, which will execute without interruption until finishing. With thread 4 which has low priority it performs round-robin.
RRS-02	Check preemption of high-priority thread by other high-priority thread	Set thread 0 as a high priority thread. Give thread 2 a duration of 0s.	As thread 2 has a shorter duration than thread 0, it will acquire que CPU immediately after being created and execute until finished.
RRS-03	If all threads are high priority (including thread 0), execute with SJF	Set thread 0 to high priority and set the duration of different threads to an increasing number of seconds	As thread 0 is the shortest process it will create the rest of the threads without being preempted. After its completion, the rest of the thread will execute following a SJF policy.
RRS-04	Check that round robin is executed correctly for low priority threads after high priority threads	Set thread 0 to high priority. Then create two high priority threads and three low priority threads.	Thread 0 will create all threads without being preempted. Then the high priority threads will execute. After they have finished, the three low priority thread will perform round-robin.

3. POTENTIAL VOLUNTARY SWITCHING

3.1. Design



All the cases for preemption described in the previous scheduler were also considered in this design in the case of a high priority queue going from waiting into ready state, which would acquire the CPU immediately if the running thread has low priority or is a longer high priority thread.

3.2. Description of the code

To the queues of section two we add another one, the waiting queue. We extended the code from the previous section implementing the functions read_disk and disk_interrupt and updating the functions timer_interrupt, scheduler and activator.

When a thread performs a system call in read_disk we check if the data_in_page_cache function returns a value different than 0, meaning that the data is not in the cache. If this happens the running thread goes to waiting state, so we insert it in the waiting queue. The scheduler returns the next process to execute and the activator performs the swap. If the data is in the cache the function returns immediately.

When a disk interrupt occurs the function checks if there are any processes in the waiting queue. If there are the function dequeues the first process and sets it as ready. When a new thread is ready we considered that preemption should occur in the same way as when a new thread is created. We perform then the same checks as we described in the description of the code in section 2.2 where preemption occurs if the ready thread has high priority and the running thread has low priority or has high priority and is longer than the ready thread. If preemption does not happen, the newly ready thread is enqueued in its corresponding queue (using a sorted enqueue in the high priority queue according to the remaining ticks).

In the scheduler we added a third check if both high and low priority queues are empty we check that the waiting queue is empty. If there are still waiting processes, the scheduler returns a reference to the idle thread.

In the timer_interrupt then we first check if the executing process is the idle thread. If it is, each tick it has to call the scheduler to check if there are any new ready processes. When the scheduler returns a process different than the idle thread it calls the activator.

In the activator we just added an additional check where if the previous running thread was the idle thread (state == IDLE) we print the corresponding message to indicate that we are setting the context of a new ready thread.

3.3. Test cases

ID	Objective	Procedure	Expected Output
RRSD-01	General functionality test	Execute the *main.c* file provided by the statement	Same execution as in section 3.2. After the creation of thread 1, thread 0 will perform two system calls. If any of them don't have the data in the cache, thread 0 will go into waiting state and as there are no other processes created, the idle process will execute until thread 0 is ready again. It performs another system call after creating the last thread, where thread 4 (low-pri) will acquire the CPU if thread 0 goes into waiting.
RRSD-02	Check that when there are no more threads on the ready queue, the idle thread starts executing.	Set all threads, except the last one, to high priority. On function_thread, the last thread makes a read_disk	After all threads have finished their execution, the last thread will start executing and perform a system call. As it is the last thread, the idle thread will execute if the data isn't in the cache. When the thread is ready it will resume execution.
RRSD-03	Check the correct execution of a voluntary context switch of a high priority thread, which will then acquire the CPU immediately after it is ready.	High priority thread 0 creates all the other threads, the first being high priority and the other ones low priority. The first high priority thread does a read disk.	When thread 1 reads from disk a low priority thread will start executing if the data isn't in the cache. When the thread is ready, as it is a high priority thread and the running thread has low priority, it will preempt it and acquire the CPU immediately.
RRSD-04	Check preemption of a longer high priority thread when a disk interruption happens and a new high priority thread is ready	High priority thread 0 creates all the other threads, the two first being high priority and the other ones low priority. The first high priority thread is shorter than the second and it does a read disk in function_thread	When the first thread performs a system call the next executing thread will be the second one, as it has high priority. However, when the disk interruption happens and the first thread is ready it will acquire the CPU, as it is shorter than the second one.

4. CONCLUSION

At first it was difficult to understand how everything worked together and where we had to implement each functionality described in the statement. However, once that was understood it was pretty straightforward to get the functionality we were requested.

A few design choices were more difficult to figure out, such as the preemption when we created a high priority process, which at first we didn't know if we should include in the create function as it would preempt thread 0 if it had low priority and the rest of the processes wouldn't be created. We had other difficulties understanding the difference between the exit and timeout functions as we didn't understand how a timeout could happen. This was all then clarified.

Overall it was a good assignment which made us understand better how the scheduler of an operating system works and the different type of schedulers.