# REPORT

## (1) Description of the code

**[RR]**
The implementation of the round-robin policy is done by the means of a “ready queue” in which processes are enqueued at the end of it immediately when they are created. The *timer_interrupt* handler checks if the quantum of time of the running process is expired. In that case, a context switch is performed. The next process to be executed is chosen by the *scheduler.*
The scheduler dequeues the first process in the *ready_queue* and returns a pointer to this process.
The *activator* performes the context switch.

**[RRF]**
For the implementation of different priorities two different queues are needed: *ready_queue* and *high_priority_queue*. The scheduler picks the process from the *high_priority_queue* (following the FIFO policy) until it is empty and then begins to dequeue from the *ready_queue* (low priority).

**[RRFN]**
The *read_network* syscall can be used from threads to trigger a voluntary context switch. This syscall saves the current context, enqueues the running process in the *waiting_queue* and then calls the *scheduler* to allow the next unblocked process to execute.
The *network_interrupt* function handles the case of receiving a signal from the network: it extracts the first process of the *waiting_queue* (that is waiting for the network) and enqueues it in the right queue (high/low priority).

NOTE: The purpose of protecting regions of code with disable_interrupts is to avoid the situation in which an interrupt disrupts a system call, or an interrupt disrupts another interrupt handler.
That’s why we put the control only in the last file, since it is the only file with multiple interrupt handlers (both timer and network). With only timer interrupts, it should not be the case that the timer interrupts itself.

## (2) Tests performed

**[RR]**
At first, our round robin crashed after all threads finished. We used gdb to help step through and find that the *running* and *current* global variables were not getting updated after a thread exited. This meant that threads that had already finished (their state was FREE) were being put back into the *ready_queue*, and getting scheduled again.
To fix this, we added code to the *activator* function that would update the global variables before performing the context switch.

**[RRF]**
To check that this file worked, we ran the main file, which creates a number of low and high priority threads. Thread 0 must be high priority, otherwise it would be preempted by the first high priority thread it creates. All the high priority threads run, and then all of the low priority threads run in a round-robin fashion.

**[RRFN]**
To test the behavior of the system under different loads, we created a file called **test_stress.c**. The purpose of this file is to see what happens when the execution times of the processes is similar to the delay between the network interrupts. What we found was that when the network interrupts were infrequent, and the execution time of the process was quick, the idle process would run for several clock intervals. This is reasonable behavior, because while the blocked processes haven't finished, the system needs to wait.
When the network interrupts happen frequently, and the processes run long, the idle process is not invoked as often. It still runs when the first thread blocks before creating any new threads, but after that there are enough threads in the ready queue to keep the CPU busy between network interrupts.

# (3) Conclusion

The trickiest parts of this assignment were making sure that the finished threads were not enqueued, and that getcontext worked correctly. The manipulation of the global queues and running variables had to be done carefully with respect to interrupts.