Laura Sizemore, John Ly

# Lab 5: Real-Time Scheduling

## Description of Solution

Data Structures

- *realtime_t*: This struct is defined in *realtime.h*. It maintains two interger values, *sec* and *msec*.
- *process_t* : This struct is the same as in lab 3 except with the addition of a few new key values. The int *realtime* maintained the status of process as 1 for realtime and 0 for non-realtime. In addition two *realtime_t* pointers were added, containing start and deadline times specifically for any realtime process.

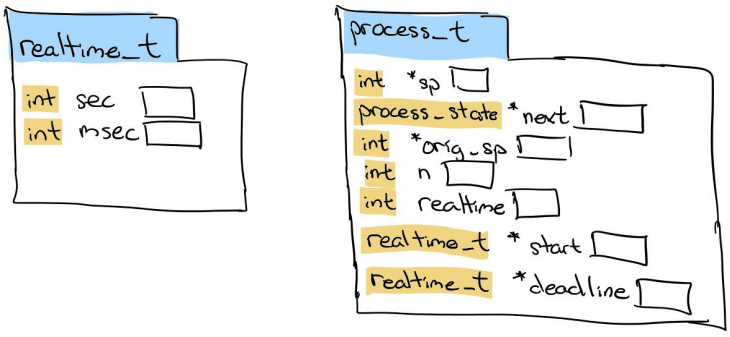


**Figure 1.** *realtime_t* and *process_t* Data Structures

*getTotalMsec*: This function converts the deadline time from being relative to start to being relative to current_time, and converts it from a realtime_t to an int. To do this, it took a realtime_t* start as input. It then multiplied start's seconds by 1000, added that value to start's msec values. It then does the same thing for the deadline. Finally, it adds the start milliseconds to the deadline's milliseconds, and returns that total.

*Can_start*: This function returned a boolean showing whether a process with a start time start could begin. To do this, the function converted both start and current_time into milliseconds. Then, using those values to compare, it returned 0 if start was greater than current_time, and 1 otherwise.

*rt_sort* : This function is used for handling new processes in *realtime_process_queue*. It has one parameter, a *process_t *proc*. In this function we first checked for any existing realtime queues. If this value was null, we simply set *realtime_process_queue* to this new *proc* and updated its *next* value to null. When there are already existing realtime processes in the queue, we instead must determine the position where the process belongs in the queue, sorting them by highest priority first. These priorities are determined by comparing the total msec of Start + Deadline times with lower counts equating to higher priorities as deadlines in EDF are to be considered relative to current times. To accomplish this comparison, we used our *getTotalMsec* function. To start this sorting process, we defined two local variables, *temp* and *prev*, to keep track of our current iterator position as well as the previous node for the purpose of properly inserting a new process in the middle of the linked list while still maintaining the proper pointers. In the case

there is only one process in the queue, we compare the new process using *getTotalMsec* and only insert the new process in front of the currently queued process when the new process has a smaller deadline time than the one in the queue. Then in a while loop, we continuously iterate through the *realtime_process_queue* whilst maintaining the previous node in *prev* until either we have reached the end of the queue or we have found a proper position for the new process as a result of its deadline priority. In the case that there are no more additional nodes to iterate through, we break out of the while loop and enqueue the process to the very end of the queue. In the case that we need to insert a node into the middle of the queue, we use our *prev* variable to insert the new process node whilst still maintaining the proper order for the *next* pointers.

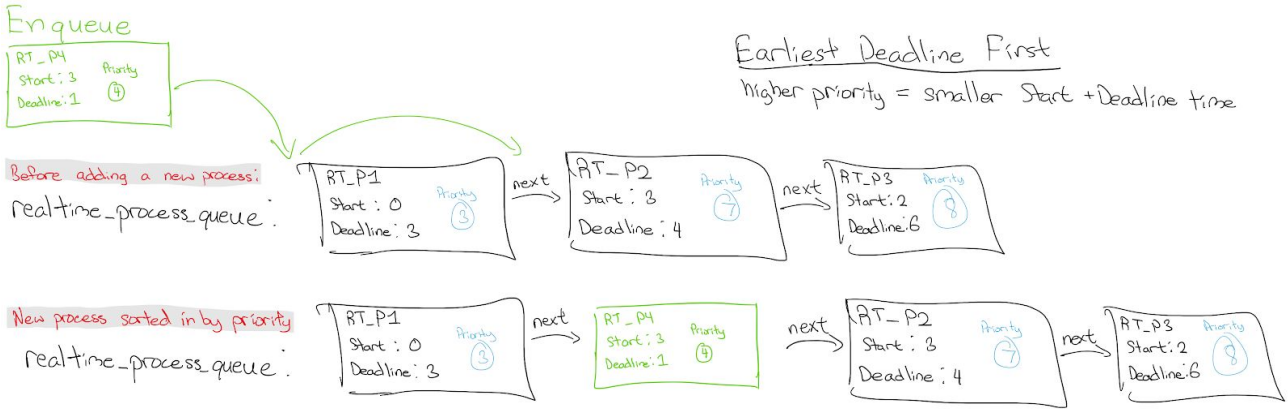


**Figure 1.** Sorting process for when adding real time processes to queue

*Find_ready:* This function returns either the process with the earliest deadline that can start, or null if none are ready. Because we use rt_sort to insert all realtime processes into the queue, we know that the realtime queue is sorted by earliest deadline. Due to this, we used a Linear Search. find_ready goes through the queue, and only compares if the current process has a start time after or at the current_time, and the sorted queue ensures that the first ready process found will also be the one with the earliest deadline. If the first ready process is the first item in the queue, then find_ready will essentially act like a dequeue, and it will update the realtime_process_queue pointer to be the popped process's next pointer. Otherwise, a local variable prev will be used to keep track of the process before the one to be popped. The prev.next will then be set to the popped process's next and realtime_process_queue will not be changed.

*Process_rt_create*: This function initializes a new realtime process, and adds it to the realtime process queue. To do this, it first uses process_stack_init and malloc to initialize stack and heap space for this new process. If that fails, this function returns a -1 to indicate that it failed. Otherwise, it updates the keys of the process_t object. So it sets n to value of parameter n, sp and orig_sp to be the sp generated by process_stack_init, realtime to be 1, as this is a realtime process, and start and deadline to the values of parameters start and deadline, respectfully. Finally, it calls rt_sort to add this new realtime process to the realtime_queue and updates the next pointer in that new process, and returns a 0 to indicate this process was created successfully.

*Process_start:* This function's structure is similar to Lab 3. However, we also had to initialize current_time's seconds and milliseconds to 0, set the priority of SV_Call, PIT0, and PIT1 so PIT1 has the highest priority. In addition, we had to initialize and start PIT1, as process_begin only starts PIT0.

*Process_Select:* This function updates the current_process and returns the current process stack pointer, or returns null if both the realtime queue and the non realtime queue are empty. To do this, it first reenables PIT1 interrupts, as the interrupt handler for PIT0 disables all interrupts and the PIT1 timer has to keep going throughout the context switch so the time is updated if a busy wait occurs. Then, the function checks if the current current_process is finished or not. If current_process is not finished, current_process's realtime value is used to correctly put that current_process back in its respective queue. If the process is done, it is freed using process_free (from lab 3), and if it's realtime, the global variable process_deadline_met or process_deadline_missed is updated, depending on whether the current_time is before/equal to or after that process's deadline respectfully. After this, if the realtime_process_queue is not empty, find_ready is used to get the next process. If find_ready returns null, the code either dequeues and returns a process from the non realtime queue, or if that's empty, it busy waits until a realtime process is ready. If realtime_process_queue is empty, then it dequeues from the non realtime process queue, and if that is empty as well, it returns NULL, signalling that all processes are done.

PIT1_IRQHandler: This function incremented current_time whenever an interrupt from PIT1 occurred. As it is an interrupt handler, there were no parameters, and the function returns nothing. First, all interrupts are disabled in this function to ensure it is atomic. Then, the current time's msec is increased by one. As current_time.msec cannot be greater than 1000, after it is incremented, we check if current_time.msec is greater or equal to 1000. While this check and the atomic interrupts should prevent current_time.msec from ever being greater than 1000, we decided there was no time saved to check greater or equal as opposed to equal, and it was better to be safe than sorry. If current_time.msec was greater or equal to 1000, we set it to 0, and incremented current_time.sec by 1. Finally, we reset the PIT1.TFLG flag, set the LDVAL to be 1 millisecond, and reenabled all interrupts.

Testing: For the EDF realtime processes, we were provided with one test case, and we wrote two more. The provided test case checked if realtime processes had a higher priority than non-realtime scheduling, and if the timing mechanisms worked as expected. To do this, this test case created two processes: one non-realtime, and one realtime. For the realtime process, the start time was set to 1 second, and the deadline was 1 msec, or 1.001 seconds. As the start time was not 0, this meant that the realtime process was not immediately ready, so the initial expected behavior was that the non-realtime process would start. Once the realtime process was ready, it would execute to completion, and finally the non-realtime process would finish. This tested if our code was correctly checking if a process was ready or not and if our current_time was working as expected, and depending on that result, either running the realtime

process or the non-realtime process. In addition, as it is impossible for a process to complete a millisecond after it started, the realtime process would miss its deadline, therefore testing our deadline_miss logic.

The first test case we made checked if our busy wait and deadline_met logic worked as expected. To do this, we had a similar set up to the provided test case. However, this time, instead of having the start time be 1 second, we made it 5 seconds. By doing this, the non-realtime process completed before the realtime process was ready, causing the code to busy wait. LED wise, we expected 4 red LEDs, or the non-realtime process, and then a pause, and then 3 blue LED's, or the realtime process. This checked if our code correctly went into busy wait, and more importantly, could leave the busy wait once a realtime process was ready. This also checked if our PIT1 timer had the highest priority, as we found if the PIT1 timer did not have the highest priority, the code would get stuck in the busy wait. In addition, we set the deadline to be 10 seconds instead of 1 millisecond, ensuring that our realtime process would meet its deadline.

Our final test case checked if our priority scheduling algorithm worked and if our code was preemptive. To do this, we created two realtime processes and no non-realtime processes. The first realtime process arrived at 1 second, and had a deadline of 10 seconds. The second process arrived at 3 seconds, but had a deadline of 5 milliseconds. Therefore, this test case started with the start time = 1 process, then the start time = 3 process cut in and ran to completion, and finally the first process finished. LED wise, this produced an expected output of B B G G G B. In addition, because of our different deadlines, we had one process that met it's deadline, and one that missed. By having this, we checked if our logic worked for multiple processes.

**Work Distribution**

How did you carry out the project?

a) design: Before starting the code, we first read over the lab instructions to ensure we understood the lab. Then, we discussed how we best wanted to design our scheduling algorithm and discussed possible solutions to the various scenarios of scheduling we drew on an iPad. We also went to office hours to ask about the specifics of the different requirements to make sure we truly understood everything.

b) coding: We used pair programming to complete this lab. Laura was the driver for part 1, while John was the observer. For part 2, Laura was the driver for the helper functions (can_start, getTotalMsec), process_start, and process_select, while John was the driver for rt_sort, rt_process_create, and find_ready, our enqueue and dequeue for realtime processes.

c) code review: For each part, the observer was reading over the code and mentioning any logic or syntax errors that he/she found while the driver typed. In addition, after each part was written, both team read over the specifications of the function written to make sure we weren't violating any pre or postconditions. Finally, we constantly compiled our code to find and fix any errors.

d) testing: We did all of our testing together. After adding the PIT1 Timer, we tested our code using a lab 3 test case to make sure the timer wasn't messing up our non realtime processes.

We did the same thing after finishing our implementation of realtime scheduling. Once that worked, we tried our code with the provided test case. After fixing those bugs and getting that test case to run, we tried the same test case again, except this time instead of the start time being 1 second, we made it 3 seconds. This made us realize that we were not setting the priorities of all 3 interrupts correctly. After we fixed that bug, we wrote our two test cases (above) and tested our code with that.
e) documenting/writing: We wrote documentation after finishing each function, so whoever was the driver for the function wrote the documentation. Laura then wrote the documentation for the test cases, the "How Did you Carry Out the Project, the testing, and the explanation of the can_start, process_start, find_ready, and process_select. John wrote the explanation and drawings of the data structures, rt_sort, process_rt_create, and the collaboration

How did you collaborate?

We divided the work evenly for both the programming and the lab report. We communicated via text and for the most part in person. Before implementing our code, we both took the time to diagram our thoughts on how we should be processing the new realtime queue differently from the non-realtime queue. To help us out we drew out diagrams on an iPad. Finally we coded everything together on Laura's laptop using pair programming where one person was the driver and the other was the observer. This helped a lot of debugging and catching logic errors in our code. For the lab report, we split the writing into parts and typed it using Google Docs with the illustrations being drawn on an iPad.