Lab 5: Real-time Scheduling Report

By: Jason Russo (jr826), Eldor Bekpulatov (eb654)

Intro

The main purpose of this lab was to implement a real-time earliest-deadline-first (EDF) scheduler. An earliest-deadline-first (EDF) schedule prioritizes the processes with the earliest deadline and allows them to run to completion before any other process. We built upon the round-robin scheduler created in Lab 3 to implement our EDF scheduler that can schedule both real-time and non-real-time processes.

Design & Implementation

realtime.h

This header file was provided by the setup files and contains some function declarations to be implemented in this lab. There's however a new data structure to represent time defined in this file. It is called *`realtime_t`*, as shown in Figure 1.

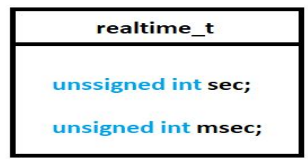

Figure 1. Structure for time.

process.c

All of our changes and new implementations were done in this file. To make things easier to understand, we will name the variable or function name and try to explain their purpose in a paragraph following the field name.

`current_time`

A `realtime_t` instance that marks the time since the beginning of execution.

`current_process`

A `process_t` instance that points to a node encapsulating the currently running process.

`process_state`

This structure encapsulates information about a process. We did not change anything from its original structure from Lab 3, but we did have to add one boolean variable `rt` and two additional pointers to `realtime_t` instances called `start` and `deadline`. The variable `rt` indicates if process is real-time or not. The `start` attribute marks the arrival time of the process, and it is relative to `current_time`. The `deadline` attribute marks the time process should finish and it is relative to the `start` attribute.

`process_queue_(head/tail)`

Contains an unordered linked list of non-real-time processes.

`rt_unready_queue_(head/tail)`

Contains an ordered linked list of real-time processes based on earliest arrival time.

`rt_ready_queue_(head/tail)`
Contains an ordered linked list of real-time processes based on earliest “absolute deadline”. “Absolute deadline” refers to the time process should finish relative to the current time. In other words, *absolute_deadline = start+deadline.*

`process_deadline_(met/miss)`
Variables counting the number of processes meeting or missing their respective deadlines.

`process_free(process_t *proc)`
Frees up the space allocated for each process node.

`process_start()`
Mostly the same as in the original implementation from Lab 3. However, we had to add lines to enable PIT1_IRQn and SVCall_IRQn, in addition to original PIT0_IRQ. PIT0 is the timer that raises interrupts for non-real-time processes. PIT1 keeps track of the current time. SVCall_IRQn is the supervisor interrupt. We also had to set the priority of each of the timers. PIT1 is set to the highest priority, as it increments the current time and that should not be affected by interrupts. SVCall_IRQn is set to the next highest priority, and PIT0 is set to the lowest priority.

`PIT1_IRQHandler(void)`
Called when PIT channel 1 raises an interrupt. It is crucial that we make this atomic function because this will increment `current_time`, so we manually disable and re-enable interrupts around the function. In the body of the function, we simply reset the channel and increment our `current_time` variable. If the `msec` attribute reaches 1000, we increment `sec` attribute and reset `msec`. This interrupt should be triggered every millisecond.

`getVal(realtime_t *time)`
Helper function that returns the msec equivalent of realtime_t instance.

`getStart(process_t *proc)`
Helper function that returns the `start` attribute in msec equivalent relative to `current_time`.

`getDeadline(process_t *proc)`
Helper function that returns *“absolute_deadline”* in msec equivalent relative to `current_time`.

`enqueue_unready(process_t *proc)`
A function that inserts a process node to an ordered linked list of real-time processes based on earliest arrival time.

`enqueue_ready(process_t *proc)`
A function that inserts a process node to an ordered linked list of real-time processes based on earliest “absolute deadline”.

`enqueue(process_t *proc)`
Simple abstract enqueue function that enqueues a process to an appropriate queue based on its attributes such as `rt`, `start`, and `deadline`. Logic goes as follows:

if(real-time)	→	check its arrival time,		
		if(past its arrival)	→	insert to ready_queue
		else	→	insert to unready_queue
else	→	add to process_queue		

`dequeue(process_t *queue_head, process_t *queue_tail)`
Simply takes the head and tail of a chosen queue and returns the head.

`process_create (void (*f)(void), int n)`
Creates new process stack on heap with size n and *enqueues* `process_t` node with initialized values. Almost exactly like Lab 3, but with `rt=0` and `start, deadline` initialized to `NULL`.

`process_rt_create(void (*f)(void), int n, realtime_t* start, realtime_t* deadline)`
Exactly like `process_create`, but with `rt=1, start, deadline` initialized to provided parameters.

`process_select (unsigned int * cursp)`
To make our logic simpler, we first start off by checking if the last process finished. If yes, then we simply increment our global variables to indicate if the process has met the deadline. If no, then we simply re-enqueue it, calling the enqueue() function.
Then we check the case where there are only unready processes available, it yes, then we idle by tuning on PIT1 interrupt until something becomes ready. Then, we repartition our unready and ready queues, until every ready process is in ready queue. Finally, we move onto selecting the next process to be run. Logic follows: if there exists rt_ready_process' we dequeue from there, else dequeue from process_queue. Lastly, we return cursp of the newly selected process, or NULL if no process was selected.

Testing

`lab5_t0.c`
This provided test case creates both a rt and non-rt process. The rt process starts after one second and has a deadline of 1 ms, so it will miss it's deadline.

`test_r1.c`
This test case creates two rt processes that will start after 5 seconds. Both have a deadline of 10 seconds. This case will test if our `process_select()` will idle properly while waiting for the rt processes to be ready to run.

`test_r2.c`

This test creates two rt processes. The first process will start after 1 sec, but has a far away deadline and takes a long time to run. The next two processes do not start right away, but have a sooner deadline. This will test our implementations ability to handle multiple real-time processes with varying deadlines.

Work Distribution

a) We start off the project by reviewing the lecture notes and discussion slide to make sure that we understand the big picture before we start coding. As we are writing the code, we also read and refer to the header files and test cases provided in this lab to get a better understanding of the implementation and expected behaviors of the test case. The major components of the lab are modifying process.c to implement real-time processes. The details of these functions are discussed in the previous section.

1) Design: We started by adding fields to process_state to handle real-time. Then, we created helper functions to keep track of start times, deadlines, current_time, and to enqueue and dequeue processes. Our goal was to have three queues: one for non-rt processes, one for rt processes ready to start, and one for rt processes that were not ready to start.
2) Coding: we implement the pair programming as suggested in lab1 before. We worked on the lab together before writing the report. We each took on an equal amount of the coding.
3) Code review: We sat together running through the code. We also went through the code line by line in debug mode, keeping track of several fields to debug.
4) Testing: two additional test cases were created: one is when there is only unready processes which tested our algorithm's ability to schedule real-time processes ahead of time. The second test shows our algorithm can schedule multiple processes based on earlier deadline priority.
5) Documenting: After we finish the lab assignments and pass all the test cases, we worked on the lab report together to make sure we are both on the same page.

b) Our collaboration consisted of in person meetings to work on the lab. We met together on across multiple days to work on the assignment together. All parts of the lab were done as a pair, with both of us discussing ideas, logic, and thoughts together before putting it into code. For the report, we outlined the various sections to be discussed and then each wrote different parts. There was no exact split here, we just kept completing parts until the report was finished.