Lab 5: Real-Time Scheduling

Goal:

The purpose of this lab is to implement real-time scheduling. To successfully complete this lab, you will need to understand the following:

- Everything from Lab 3
- Initial lectures that cover scheduling

Your lab will build on your solution to Lab 3. If you are not confident in your Lab 3 solution, you can obtain one from the TAs.

Part 1: Real-Time Scheduling

For the first item in this lab (real-time clock, worth 5 points), use the following structure definition:

```
typedef struct {   unsigned int sec;   unsigned int msec; } realtime_t;
```

(it is probably a good idea to place this in 3140_concur.h). We will use this structure to represent time (in seconds and milliseconds).

Create a global variable `current_time` of type `realtime_t` that corresponds to the elapsed time on the K64F in seconds and milliseconds. Logically, the current time in milliseconds is `1000*current_time.sec + current_time.msec`, and `current_time.msec` should always be less than 1000.

Use a different timer (other than PIT_0) to implement this so that it does not impact the correct behavior of concurrent processes. The current time must be updated every millisecond. This timer should not trigger a context switch.

The rest of this is an implementation of one scheduling algorithm. For the purposes of this lab, all time will be relative to the point when `process_start()` is called, so your variables should correspond to the elapsed time since the call to `process_start()`.

You will implement "earliest deadline first" (EDF) scheduling. The EDF scheduling algorithm always selects the job with the earliest absolute deadline. This requires a modification to `process_select()`, and functions to create tasks with real-time constraints. Provide an implementation of the following functions to support real-time tasks:

```
process_rt_create(void (*f)(void), int n, realtime_t *start,
realtime_t *work, realtime_t *deadline);
```

This creates a new process that arrives after `start` milliseconds have elapsed. The task requires `work` milliseconds to complete (this is the estimate of the worst-case execution time), and has a RELATIVE deadline of `deadline` milliseconds. `n` is still the stack size for the task.

Part 2: What You Have to Do

Modify `process_select()` to implement two-level scheduling. Implement a real-time scheduling queue that has higher priority than the normal ready queue used for ordinary concurrent processes. Your implementation should also keep track of the number of tasks that missed their deadlines in a global variable `process_deadline_miss`.

Part 3: Extra Credit

Implement real-time scheduling of periodic tasks as well. Specifically, implement:

```
process_rt_periodic(void (*f)(void), int n, realtime_t *start,
realtime_t *work, realtime_t *deadline, realtime_t *period);
```

This creates a new process that arrives after `start` milliseconds have elapsed. The task requires `work` milliseconds to complete (this is the estimate of the worst-case execution time), and has a relative deadline of `deadline` milliseconds. The task is periodic, with period specified by `period`.

Modify `process_select()` to implement this functionality, once again using EDF scheduling. The primary difference is that the task is periodic so once it terminates you must have it arrive at the appropriate time in the future. You should re-use the stack by saving away the original value of the stack pointer.
Again, keep track of the number of jobs that have missed their deadlines.

Documentation:
Provide a detailed description of your implementation of real-time scheduling, keeping track of time, etc. (max 5 pages in 11pt font and single line spacing – we will not read anything after page 5. Also, don't do that little trick where you adjust the size of the periods or margins ☺).

We recommend using illustrations to describe your key data structures (processes, etc.) and key properties of the implementation that you use to ensure correct operation.

We require that you **describe in detail –** in a separate section titled "Work Distribution" – the following:
1. How did you carry out the project? In your own words, identify the major components that constituted the project and how you conducted the a) design, b) coding, c) code review, d) testing, and e) documenting/writing. If you followed well-documented techniques (e.g., peer programming), this is the place to describe it.
2. How did you collaborate? In your own words, explain how you divided the work, how you communicated with each other, and whether/how everyone on the team had an opportunity to play an active role in all the major tasks. If you used any sort of tools to facilitate collaboration, describe them here.

Submissions:

You will need to turn in your process.c and 3140_concur.h. You must also provide at least two non-trivially different test cases to demonstrate that your implementation works correctly. Name these files test_*name*.c. Your test cases should also include scenarios where you change the frequency of the timer interrupt to observe differences in interleavings. Real-time tasks will continue to behave in a predictable manner, while non-real time tasks will not.

The lab requires the following files to be submitted.

1. concurrency.zip – which includes process.c, 3140_concur.h and your test cases.
2. writeup.pdf – documentation of your implementation

The C files should be commented in a way that makes it clear why your program works.