CSC9006 Real-Time Systems (Spring 2021) Homework 2

** **Submission deadline: 9PM, April 7th, 2021.** **

1. Introduction

Throughout this homework, we will

- review what we've learned about real-time scheduling;
- explore Linux's system support for scheduling, and implement the rate-monotonic (RM) scheduling policy;
- revisit our elevator control system, considering the existence of a higher-priority task as well as a requirement of response time.

Let's begin :)

2. Lecture reviews

Answer the following three questions. Write your answers into one plain text file and name it `review_questions.txt`. Restrict your analysis to a single-core CPU system.

- (5 points) Explain how anytime algorithms may help improve both the responsiveness and the utilization of a system, especially when the system needs to serve multiple applications? Your answer should touch both aspects.
- (5 points) Prove or disprove: the RM scheduling policy can schedule the following task set:
  - Task 1 has period $p_1 = 5$ and WCET $c_1 = 1$;
  - Task 2 has period $p_2 = 4$ and WCET $c_2 = 2$.
- (10 points) Specify a task set of two tasks that cannot be scheduled using the RM policy. Your answer should give their periods and WCETs. The total CPU utilization of your task set should be less than 1.

3. Linux's scheduling environment

Linux Programmer's Manual provides an overview of CPU scheduling support. Type `man sched` to take a look at it. Based on the `sched` man page and related man pages, answer the following two questions and put your answer in a plain text file named `sched.txt`:

- (5 points) Give a brief comparison between the SCHED_OTHER scheduling policy and the SCHED_FIFO scheduling policy.
- (5 points) Following the previous question, can we use `setpriority()` to change the parameter `sched_priority` ? If not, what is the purpose of `setpriority()` ?

In the following, we will learn how to use the SCHED_FIFO policy and we will see how the policy affects the response time of tasks. First, let's learn some tools to better control our experimental setting.

3a. Fix the CPU frequency

In modern laptops, the CPU frequency may change over time to save energy. But that will essentially change a program's execution time and, as a consequence, response time. So, for our purpose, we should fix the CPU frequency instead. We will use a tool to help us achieve so:

`$ sudo apt install cpufrequtils`

This package provides three utilities: `cpufreq-aperf`, `cpufreq-info`, and `cpufreq-set`. Read the corresponding man pages to learn more. In the homework 2 folder, there are two scripts for you to set/unset CPU frequency. Use `setCPUFreq.sh` to fix the frequency of all CPU cores on your machine, and use `unsetCPUFreq.sh` to release it. Remember to adjust the parameters in the scripts according to your machine's hardware specification.

A curious mind should read this document for further detail on this subject.

3b. Set the CPU affinity of a task

For our evaluation purpose, we need to pin tasks to the same CPU core. This can be achieved by `sched_setaffinity`. Here is an example function:

```c
#define _GNU_SOURCE
#include <sched.h>
...
void pinCPU (int cpu_number)
{
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(cpu_number, &mask);
    if (sched_setaffinity(0, sizeof(cpu_set_t), &mask) == -1)
    {
        perror("sched_setaffinity");
        exit(EXIT_FAILURE);
    }
}
```

Calling `pinCPU(0)` at the beginning of your program execution would ensure that the execution will stay on CPU core 0.

3c. Implementing the RM scheduling policy

With all the tools at our disposal, it is now straightforward to implement the RM scheduling policy: we use SCHED_FIFO and set task's `sched_priority` value according to the reciprocal of the task period. There are two ways to set `sched_priority`:

1. Use the `sched_scheduler` system call;
2. Use the `chrt` utility.

Here we will follow the second approach (The first approach is nevertheless useful, and we will come back to it in the future assignments). For example, given two executables, p1 and p2, we may write a simple script to run them using the `SCHED_FIFO` policy, with the priority of p1 set to 99 and that of p2 set to 98:

```
#!/bin/bash
chrt -f 99 ./p1 &
chrt -f 98 ./p2 &
```

The `-f` option sets the scheduling policy to `SCHED_FIFO`. The number before the command-to-run specifies the priority for that command. The highest priority available is 99. Read `man chrt` and `man sched_get_priority_max` for detail.

We will use `timer_create` to create a set of periodic tasks. Here is a code skeleton:

```
...
result[...]; // a buffer to keep the timing results
void workload()
{
    // Run a synthetic workload for some milliseconds
    ...
    // Measure the response time of running the
    // synthetic workload, and keep the timing result
    // in result[]. Here we do not right away dump
    // the result to a file, because doing so would
    // cause additional and unpredictable latency.
}
...
static void
handle_timeout(int sig, siginfo_t *si, void *unused)
{
    workload();
}
...
int main()
{
    // Setup a periodic timer using timer_create();
    ...
    // use handle_timeout() to catch SIGUSR2
    ...
    // Start the timer using timer_settime()
    ...
    // Perform the first run of the workload
    workload();
    while (/* Count for a certain number */)
    {
        pause(); // wait for the timer
    }
    // Only until here do we dump all the timing
    // results to a file. This is how we do file I/O:
    std::ofstream out_file;
    out_file.open("resp_tx.out", std::ios::out|std::ios::trunc);
    for (int i = 0; i < /*number of result*/; i++)
    {
        out_file << result[i] << std::endl;
    }
    out_file.close();
    return 0;
}
```

Assignment: use the above skeleton to implement two tasks:

1. **t1.cpp** : a periodic task with period=100 milliseconds and WCET being around 50 milliseconds; **resp_t1.out** is a collection of 500 response times from running this task;
2. **t2.cpp** : a periodic task with period=500 milliseconds and WCET being around 80 milliseconds; **resp_t2.out** is a collection of 100 response times from running this task.

Remember to pin them to the same CPU core. Also, remember to fix the CPU frequency. For sanity check, recall that we can use `top` to view the current CPU utilization as well as the utilization of each running process/thread, and much more.

Now, we may run these tasks using essentially the rate-monotonic policy, by the following script (use `sudo` to earn the permission to run `chrt`):

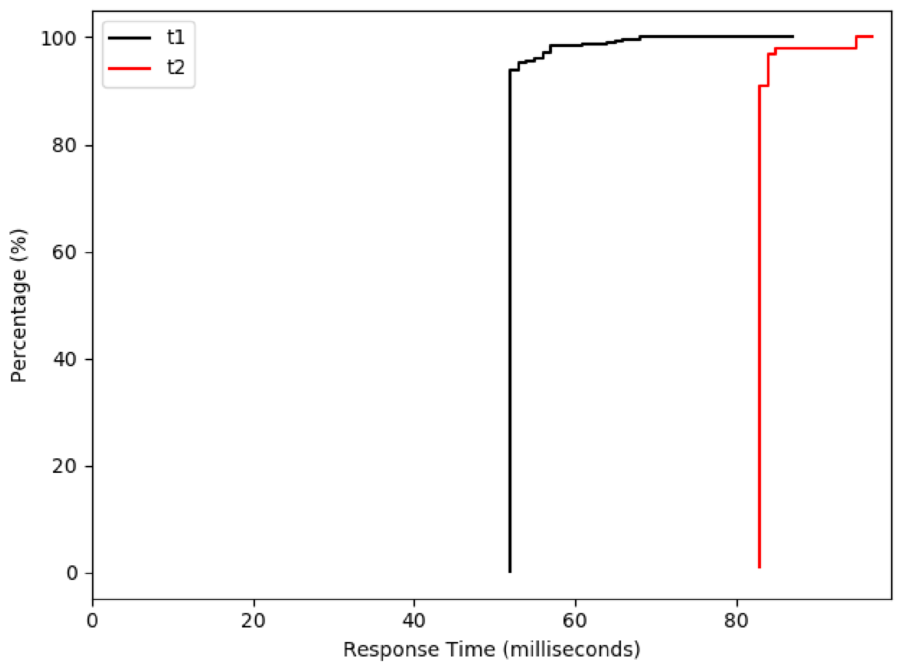
```
#!/bin/bash
chrt -f 99 ./t1 &
chrt -f 98 ./t2 &
```

Plot the c.d.f. of resp_t1.out against resp_t2.out. Name your plot `result.png` (there is a handy Python plotting script in the homework 2 folder). Based on the plot, provide your observation and analysis of the response times of t1 and t2, and write them into a plain text file and name it `result_analysis.txt`.

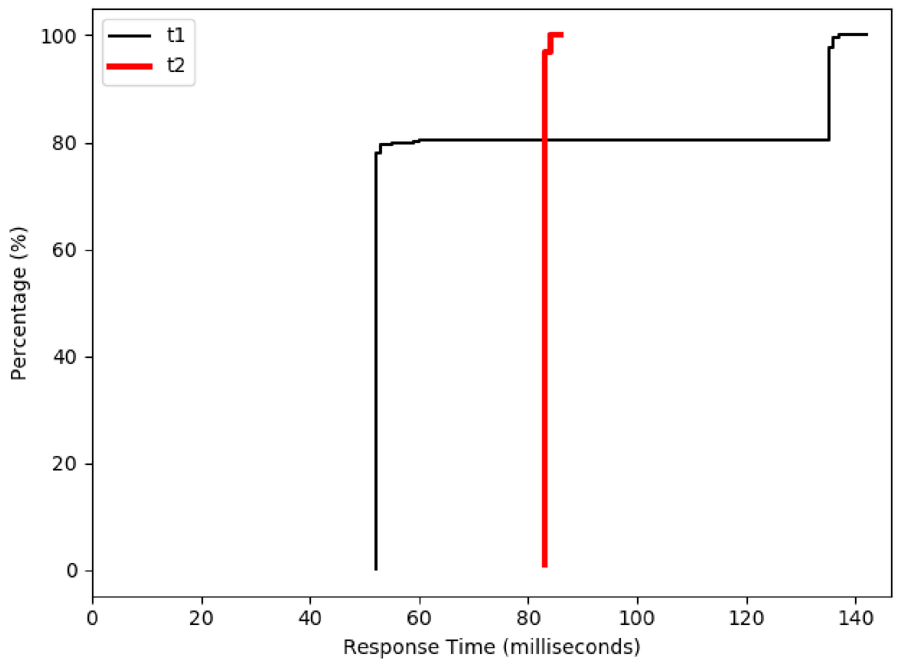
Here is an example of how to analyze experimental results. Suppose we run the following script instead (which implements the inverse of the rate-monotonic policy):

```
#!/bin/bash
chrt -f 98 ./t1 &
chrt -f 99 ./t2 &
```

Then we will have result either like this:



or like this:



or like something else.

For the first figure, it shows that the response time of t2 is pretty stable, around 80 ms; while for t1, about 5% among the distribution of the response time is 20 ms more than its WCET. It is because occasionally t1 may be delayed by t2, since t2 has a higher priority. In rare cases, t2 may have response time 10-15 ms more than it WCET, because the system also has some other critical tasks running.

For the second figure, interestingly, we observed that there are 20% chances that t1 would have a 80 ms increase in the response time. This is no accident: consider the period and the WCET of both tasks, we see that from the critical instant, every one out of five requests for t1 will be delayed by a full execution of t2, which has a WCET of 80 ms.

Finally, these results also suggest that, very often, it is a good idea to repeat an experiment many times, so that we may have enough samples to reveal the nuances in system behaviors.

With the above example, now it's your turn to analyze your experimental results :)

3. An elevator control in a busy system

Now let's revisit our elevator control implemented in Homework 1. Suppose that on this single-core CPU system we also need to run a higher-priority periodic task, with period being 70 ms and WCET being 30 ms. Recall that our time-triggered control uses 100 ms timer interval, and that the workload is 50 ms. If our goal is to have 95% probability that the response time of the control (as specified in Homework 1) will be less than 160 ms (in order to make our customers happy), can we meet that goal using our current setting along with a fixed-priority preemptive scheduling? Support your analysis by some empirical evidence. Name your c.d.f. plot of the response time as `resp_elevator.png`. Write your analysis in a plain text file and name it `elevator_analysis.txt`.

4. Things to Submit to Moodle

Package the following items into one single zip file and name the file using your student ID:

- (20 points) `review_questions.txt`;

- (65 points) `sched.txt`, `t1.cpp`, `t2.cpp`, `resp_t1.out`, `resp_t2.out`, `result.png`, and `result_analysis.txt`;
- (15 points) `resp_elevator.png` and `elevator_analysis.txt`.

5. Concluding Remarks

Let's review what we've learned:

- When doing real-time experiments, it is a good idea to fix the CPU frequency. This is achieved by using the `performance` CPU governor (the default one is `powersave`).
- We may have a better control of our real-time experiment by restricting our target tasks to run on some specified set of CPU cores.
- In Linux, we may implement the rate-monotonic scheduling policy by the use of (1) the **SCHED_FIFO** policy, and (2) the static scheduling priority, *sched_priority*.
- `top` is our friend for sanity check.
- We have seen empirically how a change in scheduling policy may affect the response time of a task.
- We see that it is necessary to repeat experiments to cover different system behaviors.

Hope you enjoy this exercise :)