

CSC9006 Real-Time Systems (Spring 2021) Homework 1

CSC9006 Real-Time Systems (Spring 2021) Homework 1

1. Introduction
2. A Tutorial for Some Related Tools
3. A Time-Triggered Control for an Elevator
4. Things to Submit to Moodle
5. Concluding Remarks

**** Submission deadline: 9PM, March 21th, 2021. ****

1. Introduction

In this homework assignment, we will implement a program to simulate part of a time-triggered control system for an elevator. Review the example right before Section 1.6 in the textbook for such a time-triggered control.

Throughout this homework, we will learn to

- use Linux's signaling facility;
- use `chrono` the C++ library to measure the interval between the time we hit the calling button and the time the elevator responded;
- conduct some experiments to evaluate the latency performance of our program.

Let's begin :)

2. A Tutorial for Some Related Tools

In real-time systems research and development, we often need to measure the timing performance of our program. Among various time utilities, C++'s `chrono` library is in particular suitable for our need. Here are two great C++ reference sites where you may learn how to use the library: cppreference.com and cplusplus.com. Spend some time to get familiar with the sites. You will need them in both this and the future homework assignments.

To measure the response time of a task, we first use type

`std::chrono::system_clock::time_point` to declare time variables. We then use

`std::chrono::system_clock::now()` to get the current instant. We may determine the response time by subtracting the end time of a job execution by its start time, using

`std::chrono::duration`. Check out the [documentation](#) to learn more.

The following C++ snippet is an example:

```

#include <iostream>
#include <chrono>

int main (void)
{
    std::chrono::system_clock::time_point startTime =
std::chrono::system_clock::now();
    // TODO: add your workload here
    std::chrono::system_clock::time_point endTime =
std::chrono::system_clock::now();
    const int delta = std::chrono::duration_cast<std::chrono::milliseconds>
(endTime - startTime).count();
    std::cout << "response time of our workload = " << delta << " ms" <<
std::endl;
    return 0;
}

```

To compile with some older version of g++, we may need to make the compiler know that we are using C++11's feature:

```
g++ -std=c++11 main.cpp
```

As discussed in class, a task's workload essentially determines its *execution time*. But because a modern operating system is typically multitasking, the *response time* of a task could be much larger than the execution time of it.

In real-time systems research, sometimes we find it useful to create some controlled *synthetic workloads* for us to validate our systems design from the timing aspect. For example, suppose that our target application is elevator control, and that we would like to see how our system would behave should it run a control algorithm with 50 milliseconds execution time. In this case, instead of implement an exact algorithm, we may first create a simple task that spends 50 milliseconds looping over some simple arithmetic operation.

Assignment Item 1: Modify and add some synthetic workload to the C++ snippet above. The amount of workload you added should account for about 50 milliseconds of *response time*. Name your code `item1.cpp`. Think about why we say *response time* here, not *execution time*.

After several attempts to tune the workload, you may still find that the response time vibrates quite a bit. Don't worry. There are factors that will affect the timing performance of a program running in Linux, and we will learn how to deal with them in the future homework assignments.

Now let's visualize the distribution of the response time of our task. We do so by plotting the cumulative distribution function (c.d.f.) of the response time using [Matplotlib](#). Run your code several times and save the record of response times in a file named `data.out`, one record per line. Install Matplotlib according to [the official instruction](#). Use the following Python script to plot the c.d.f. of the value of interest:

```

import numpy as np
import matplotlib.pyplot as plt

rt = np.loadtxt('./data.out')

fig1, ax1 = plt.subplots()

n = np.arange(1, len(rt)+1) / np.float(len(rt)) * 100

```

```
rtSorted = np.sort(rt)
ax1.step(rtSorted, n, color='k', label='task1')

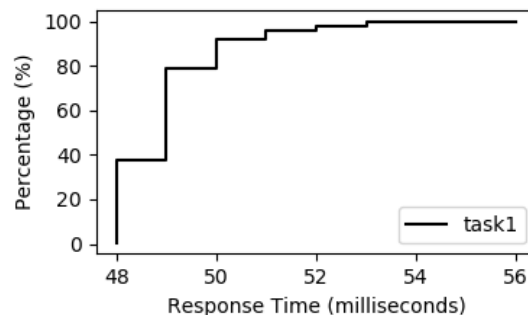
ax1.set_xlabel('Response Time (microseconds)')
ax1.set_ylabel('Percentage (%)')

plt.legend(loc=4)
plt.tight_layout()
plt.show()
```

Name the script `cdf.py` and we can plot by typing

```
python3 cdf.py
```

which will give us a figure similar to the following:



From the control panel at the bottom of the window, you may tweak the presentation of the plot and save the result as a .png file.

Assignment Item 2: Run the compiled code of your `item1.cpp` for 200 times and record the response time of each execution. Name your record `data.out`. Plot the c.d.f of the data in `data.out` using `cdf.py`, and save the figure as `cdf.png`. You may use the following bash script to help accomplish this assignment item:

```
#!/bin/bash
for i in `seq 1 200`
do
    ./a.out >> data.out
done
```

Finally, let's review how to use Linux's signaling facility to send signals to a program. We will use that in our time-triggered elevator control program to simulate a press of the calling button. The following example implements an *event-triggered* task that prints "Got SIGUSR1":

```
// Note: The following code is derived from the example
//      in the man page of mprotect(2).
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void handler(int sig, siginfo_t *si, void *unused)
```

```

{
    // !!!! Read the following note from mprotect(2):
    /* Note: calling printf() from a signal handler is not safe
       (and should not be done in production programs), since
       printf() is not async-signal-safe; see signal-safety(7).
       Nevertheless, we use printf() here as a simple way of
       showing that the handler was called. */
    printf("Got SIGUSR1\n");
}

int main(int argc, char *argv[])
{
    char *p;
    struct sigaction sa;

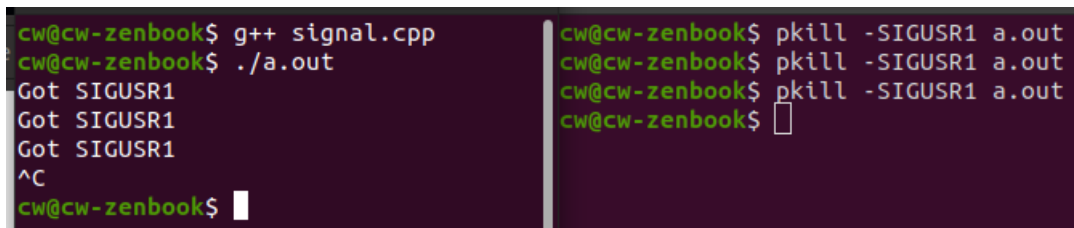
    sa.sa_flags = SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = handler;
    if (sigaction(SIGUSR1, &sa, NULL) == -1)
    {
        handle_error("sigaction");
    }
    while (1)
    {
        pause();
    }
    return 0;
}

```

Run this code at one terminal, and at another terminal type the following to send the signal to it:

```
kill -SIGUSR1 a.out
```

Hit Ctrl+C to terminate the program. Here is an example of execution:



```

cw@cw-zenbook$ g++ signal.cpp
cw@cw-zenbook$ ./a.out
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
^C
cw@cw-zenbook$

cw@cw-zenbook$ kill -SIGUSR1 a.out
cw@cw-zenbook$ kill -SIGUSR1 a.out
cw@cw-zenbook$ kill -SIGUSR1 a.out
cw@cw-zenbook$

```

Read `man sigaction`, `man pause` and related man pages to learn more.

Assignment Item 3: Read the man page `man 7 signal` and explain (1) what is `SIGUSR1`, and (2) what is its default action, and why our code above behaved differently. Write your answer in a plain text file and name it `sigusr1.txt`.

3. A Time-Triggered Control for an Elevator

Review the example right before Section 1.6 in the textbook for our target time-triggered control functionality. Then write a C++ program to meet the following requirements:

1. The program use a boolean variable `pressed` to denote the status of the calling button of the elevator. The variable is initialized to `false`.

2. The program catches signal `SIGUSR1`, and in handling the signal it flips variable `pressed` to `true`.
3. The program would use a periodic timer to check whether variable `pressed` equals `true`, at the interval of 500 milliseconds, and if the condition is met the program will (1) execute a 50-millisecond synthetic workload, followed by printing "Sent the control command", and (2) flip variable `pressed` to `false`. You are required to use Linux's system call `timer_create` to implement the periodic timer. You may learn how to do this by reading the man page of the system call (there is an example code in it). In your implementation, use `CLOCK_REALTIME` for the `clockid` argument when creating the timer.

Name your code as `control1.cpp`. Send 200 `SIGUSR1` to your program at changing intervals. Plot the c.d.f. of response times between the time your program caught `SIGUSR1` and the time it printed "Sent the control command". Name the plot `resp1.png`. Now, change the timer's interval from 500 milliseconds to 100 milliseconds, and repeat the above procedure. Name your new plot as `resp2.png`. Based on your plots, give reasons for the timing behavior of your program. Write your analysis in a plain text file and name it `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:

- (15 points each) `item1.cpp`, `data.out`, `cdf.png`, and `sigusr1.txt`;
- (10 points each) `control1.cpp`, `resp1.png`, `resp2.png`, and `analysis.txt`.

5. Concluding Remarks

What we have done in this homework assignment is a typical process of experimental research. We study how a design works by implementing it and experimenting with different parameters. We learn by making our hands dirty.

For a general-purpose operating system, timeliness of program execution is not the solely goal. For example, power-saving is important for laptops, and therefore the OS may perform [CPU frequency scaling](#) that to dynamically adjust the CPU speed to prolong the battery life. Practical concerns like this often make it harder to study the timing performance of a system. As another example, to prevent long-running real-time tasks from freezing the system, in Linux those real-time tasks by default can only use 95% of all CPU cycles (type `man sched` and read the section 'Limiting the CPU usage of real-time and deadline processes' to learn more about this). Being aware of your system's configuration is important when doing systems research.