CSC9006 Real-Time Systems (Spring 2021) Homework 3

** **Submission deadline: 9PM, April 18th, 2021.** **

1. Introduction

Throughout this homework, we will conduct some empirical evaluation for the Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP). To do this, we will use POSIX threads, i.e., pthreads, to coordinate multiple real-time tasks in the same program. No worry if you've never done multi-threaded programming before. In the following, sufficient detail will be supplied in the context.

For interested readers, you can learn a lot more about pthreads via `man phtreads` and related man pages. Here you can find a comprehensive tutorial on multi-threaded programming, using pthreads as a particular implementation. Finally, this book and its example code are good references for pthreads programming.

Let's begin :)

2. A quick guide on the use of pthreads

The following is an example program that spawns two pthreads to perform some work (adopted from the book mentioned above):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void *do_one_thing(void *);
void *do_another_thing(void *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

extern int
main(void)
{
  pthread_t thread1, thread2;
  if (pthread_create(&thread1,
         NULL,
         do_one_thing,
         (void *) &r1) != 0)
    perror("pthread_create"), exit(1);
```

```
  if (pthread_create(&thread2,
         NULL,
         do_another_thing,
         (void *) &r2) != 0)
    perror("pthread_create"), exit(1);

  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);

  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

  do_wrap_up(r1, r2);

  return 0;
}

void *do_one_thing(void *pnum_times)
{
  int i, j, x;
  for (i = 0;  i < 4; i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*(int *)pnum_times)++;
  }
  return(NULL);
}

void *do_another_thing(void *pnum_times)
{
  int i, j, x;
  for (i = 0;  i < 4; i++) {
    printf("doing another \n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*(int *)pnum_times)++;
  }
  return(NULL);
}

void do_wrap_up(int one_times, int another_times)
{
  int total;
  total = one_times + another_times;
  printf("All done, one thing %d, another %d for a total of %d\n",
    one_times, another_times, total);
}
```

The idea is simple: the new thread will run the code in the function referenced by `pthread_create`. The original process will call `pthread_join` to wait for the completion of the spawned thread.

The above code can be compiled as follows (name it as "simple_threads.c"):

```
g++ -o simple_threads ./simple_threads.c -pthread
```

Here we link the pthreads library via `-pthread` (or `-lpthread`).

Here is a template for you to create two critical sections and to protect them by a mutex:

```
#include <pthread.h>
//...
// In thread1:
    // some workload
    // entering the critical section ...
    {
    pthread_mutex_lock (&mutex_section1);
        // some workload
    pthread_mutex_unlock (&mutex_section1);
    }
    // leaving the critical section ...
    // some workload
//...
// In thread2:
    // some workload
    // entering the critical section ...
    {
    pthread_mutex_lock (&mutex_section1);
        // some workload
    pthread_mutex_unlock (&mutex_section1);
    }
    // leaving the critical section ...
    // some workload
//...
```

Here is how to plug in the Priority Inheritance Protocol (PIP):

```
...
    pthread_mutexattr_t mutexattr_prioinherit;
    int mutex_protocol;
    pthread_mutexattr_init (&mutexattr_prioinherit);
    pthread_mutexattr_getprotocol (&mutexattr_prioinherit,
                                   &mutex_protocol);
    pthread_mutexattr_setprotocol (&mutexattr_prioinherit,
                                   PTHREAD_PRIO_INHERIT);
    // finally, associate the PIP to our mutex of interest
    pthread_mutex_init (&mutex_section1, &mutexattr_prioinherit);
...
```

Here is how to plug in the Priority Ceiling Protocol (PCP), very similar to the way we use PIP:

```
...
    pthread_mutexattr_t mutexattr_prioceiling;
    int mutex_protocol, high_prio;
    high_prio = sched_get_priority_max(SCHED_FIFO);
    pthread_mutexattr_init (&mutexattr_prioceiling);
    pthread_mutexattr_getprotocol (&mutexattr_prioceiling,
                                   &mutex_protocol);
    pthread_mutexattr_setprotocol (&mutexattr_prioceiling,
                                   PTHREAD_PRIO_PROTECT);
    pthread_mutexattr_setprioceiling (&mutexattr_prioceiling,
                                      high_prio);
    // finally, associate the PCP to our mutex of interest
    pthread_mutex_init (&mutex_section1, &mutexattr_prioceiling);
```

```
...
```

Finally, to set a task's priority level, recall that in Homework 2 we've mentioned that there are two ways to do so:

1. Use the `sched_setscheduler` system call (there was a typo in Homework 2);
2. Use the `chrt` utility.

We used the second way in Homework 2, because it allows us to make on-line changes to how a task will be scheduled. Now we will use the first way, which allows us to make changes at the precise logical timing in the program; in other word, we can control the change so that it would happen before (or after) the execution of some specific code segment in our program. Working with pthreads, instead of using `sched_setscheduler` we will use `pthread_setschedparam`; similarly, we will use `pthread_getschedparam` instead. You may find details in the man pages of these system calls and APIs. In summary, you can use the following helper function to set a thread's real-time scheduling policy and real-time priority level:

```
#include <pthread.h>
void setSchedulingPolicy (int newPolicy, int priority)
{
    sched_param sched;
    int oldPolicy;
    if (pthread_getschedparam(pthread_self(), &oldPolicy, &sched)) {
        perror("pthread_setschedparam");
        exit(EXIT_FAILURE);
    }

    sched.sched_priority = priority;

    if (pthread_setschedparam(pthread_self(), newPolicy, &sched)) {
        perror("pthread_setschedparam");
        exit(EXIT_FAILURE);
    }
}
```

Similar to the way we use PIP/PCP, we call *get* to populate a variable structure (here, `sched`), and then we call *set* to use that structured information to setup our target.

For example, calling `setSchedulingPolicy(SCHED_FIFO, 99)` we can set the current thread's scheduling policy to `SCHED_FIFO` and run it at priority 99. You will need the root privilege to make this work.

3. Empirical evaluation of PIP and PCP

Now let's apply what we've learned since the beginning of this semester. We will use the POSIX timers (Homework 2) to create a set of periodic real-time tasks (implemented using pthreads). We will follow the RM scheduling policy and begin with two real-time tasks, one being of a higher priority level, and have them share a mutex. Because of the mutex sharing, the high-priority task at run-time could have a response time equal to its execution time *plus* the interval the low-priority task held the mutex, the so-called priority inversion.

Here is the specification of the two tasks:

Task	Period (ms)	WCET (ms)
τ_1	139	30 (with 10 ms in the critical section protected by the mutex)
τ_2	257	100 (with 20 ms in the critical section protected by the mutex)

Your first job is to create some middle-priority periodic tasks (use the RM scheduling policy) that do not use the mutex but will occasionally cause a further delay to the high-priority task, *so that the response time of the high-priority task could exceed 100 ms*. Recall that this happens when the high-priority task is blocked by the low-priority task (due to the access of mutex) and that the low-priority task is preempted by the middle-priority task(s). As a sanity check, always make sure that the total CPU utilization of your task set is under 100%.

Write a program and give empirical evidence that the response time of the high-priority task could exceed 100 ms. Name your code `priority_inversion.cpp` and the experiment result `priority_inversion.out`, which should contain at least 5000 results of the response time information of the high-priority task. I've made a base version for you, named `starting_point.cpp` in the homework resource folder, and you are encouraged to build your version on top of it. The base version uses the POSIX timer in a way different than what we did in Homework 2: now, upon timer expiration, it will work as if we create a new thread to run the workload. Check out the code and `man timer_create` to learn more about this.

Explain how you achieve so and your analysis. For example, what's the period and the WCET of your middle-priority tasks? how many middle-priority tasks? how do you decide the period and the WCET for the middle-priority tasks? how do you decide the needed number of those tasks? does the result meet your expectation, and why or why not? Write your answer in a plain text file and name it `experiment_setup.txt`.

Then, following what you've created, now plug in PIP, and give empirical evidence that the response time of the high-priority task never exceeds 100 ms. Name your code `withPIP.cpp` and the experiment result `withPIP.out` (again, at least 5000 results).

Then, replace PIP by PCP, and give empirical evidence that the response time of the high-priority task also never exceeds 100 ms. Name your code `withPCP.cpp` and the experiment result `withPCP.out` (again, at least 5000 results).

IMPORTANT: remember to fix the CPU frequency for your experiments; otherwise, the result will be misleading.

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) `experiment_setup.txt`;
- (50 points) `priority_inversion.cpp`;
- (35 points) `priority_inversion.out`, `withPIP.out`, and `withPCP.out`.

Should you have any questions, feel free to post them on the Moodle forum. Enjoy exploring this aspect of real-time systems :)