Mectrom/Sfwr Eng 4aa4 - Lab 7

Resource sharing protocols

Introduction:
By the time you start with this lab, most of the scheduling algorithms and resource sharing protocols would have been discussed in the class (hopefully). In this lab you will get some hands-on experience on how the priority inversion takes place and learn about protocols such as Priority Inheritance and Priority Ceiling that try to minimize the blockage of the highest priority task.

Goals:

- Understand how priority inversion may occur and use priority inheritance algorithm to avoid it
- Use a simulation software to simulate the schedule of a set of tasks
- Priority Ceiling and Priority Inheritance protocols for resource sharing

Note: Before you go for your lab sessions, please read the following documents at your own convenient time, in addition to the class notes:

- `http://linux.die.net/man/3/pthread_mutexattr_setprotocol`
- Pthreads API:

  `https://computing.llnl.gov/tutorials/pthreads/#PthreadsAPI`
- Pthreads and pthread mutexes at the link:

  `https://computing.llnl.gov/tutorials/pthreads/#Mutexes`
- Posix semaphores at the link:

  `http://www.csc.villanova.edu/~mdamian/threads/posixsem.html`

Part 1[30]
Priority Inversion:

- As discussed in class, priority inversion may occur when a task with lower priority holding a resource is preempted by a task with a higher priority. The CPU control is passed on to the task with higher priority but the lower priority task still holds the control on the resource it has previously locked. In this situation, if the higher priority task needs the same resource that is held by the lower priority task, it will be blocked and a task with a priority lower than that of this task may be allowed to run. Depending upon the number of tasks and their relative priorities, the higher priority task may be blocked for an indefinite period of time.

- One way to minimize the effects of priority inversion is the use of priority inheritance protocol which insures that the task that owns a resource executes at the priority of the highest priority task blocked on that resource. When execution is complete, the task relinquishes the resource and returns to its normal priority. Therefore, the inheriting task is protected from preemption by an intermediate priority task. In order to implement the protocol, mutexes are used to lock the shared resources. A mutex works similar to a semaphore, however it is possible to set one of the following resource sharing protocols for mutexes:

  ```
  PTHREAD_PRIO_NONE,
  PTHREAD_PRIO_INHERIT,
  PTHREAD_PRIO_PROTECT
  ```

  Follow the steps given below carefully. You are incrementally building a project that will demonstrate priority inversion.

- Start with a new C-project in your work space that creates three real time tasks as threads. This is similar to what you have done in earlier labs. However in order to keep things simple the tasks will not be periodic i. e. they will just run once. Each time a task runs it will execute a job that uses CPU time. Implement a function similar to the one given below for this purpose:

  ```
  void workfor (long n, int task) //Use CPU time
  {
     long counter=0;
     while (  counter < n)
           { counter++;
                     }
     printf("Task %d finished spinning\n", task);
  }
  ```

- Set the priorities of tasks 1, 2 and 3 as 45, 42 and 40 respectively.

- Use different values of 'n' while calling the above function in each of the three tasks to represent the work being done by each task. For example, values such as 1000000, 100000000 and 200000000 can be used for tasks 1, 2 and 3 respectively.

- You want to measure the time for which each task runs. Linux provides several time functions and clocks. Please use the CLOCK_MONOTONIC as done in earlier labs and use "clock_gettime()" function to get the time on CLOCK_MONOTONIC. This function saves the current value of time in seconds and nanoseconds in a structure of type "timespec_t". For more details refer to man pages or similar sources on the internet.

- Call the "clock_gettime()" function as soon as a task is created and save the seconds and nanoseconds when it started then print this time on console.

- Call the function again when a task has finished doing its work. The difference between this time and the start time is a good measure of the time for whic h a task runs. Print this value on the screen.
- So far in this project you have not used a semaphore, a mutex or a condition variable.
- Build your project and run it on myRIO to ensure you are on the right track so far!.
- The output should be similar to that shown below:

```
Task1 created: sec 2703, nsec 401366245
Task2 created: sec 2703, nsec 401775346
Task3 created: sec 2703, nsec 401967640
Hello! This is task1, Priority:45
Task 1 done spinning
Task1 took 0 seconds, 16928869 nsec
Hello! This is task2, Priority:42
Task 2 done spinning
Task2 took 1 seconds, 218866716 nsec
Hello! This is task3, Priority:40
Task 3 done spinning
Task3 took 4 seconds, -378185506 nsec
logout
```

- Show the output to one of your TAs.

Part 2:[30]
Continue working with your project in part 1.

- In order to create priority inversion, task 1 and task 3 will share a resource represented by a mutex. Declare a mutex and a mutex attribute object with global scope:

```
pthread_mutex_t mutex;
pthread_mutexattr_t mutex_attr;
```

- Both the task 1 and the task 3 put a lock on this mutex before carrying out their main work and unlock it after finishing with the work. The task 2 does not use this mutex as it does not share a resource with either of the other two tasks. Refer to links cited at the start of this document to find out how to lock and unlock a mutex.
- Now you need to control the execution of each task. For this purpose you will use a total of five semaphores. Two semaphores for each of the high and medium priority tasks to signal that they are ready to run (assume they are identified as semHi_rdy and semMid_rdy) and three semaphores for allowing each of the three tasks to run (assume they are identified as: semLow_r, semMid_r, semHi_r).

- The five semaphores are declared with global scope and are initialized to zero in the main() function so that any task that waits on one of these semaphores, gets suspended util some other task calls the "pthread_post()" function releasing the semaphore.

- The mutex is also initialized in the main() function after setting protocol for mutex_attr to PTHREAD_PRIO_NONE (i. e. no protocol):

```
if (pthread_mutexattr_setprotocol(&mutex_attr, PTHREAD_PRIO_NONE))
    {perror("mutex init"); exit(1);}

   if (pthread_mutex_init(&mutex, &mutex_attr))
    {perror("mutex init"); exit(1);}
```

- The three tasks are created one after the other in the main() function but each task gets suspended before it starts doing any useful work.

- Task1 and task 2 call "pthread_post()" function to release the semaphores indicating that they are ready to run (semHi_rdy and semMid_rdy) and then wait on their respective semaphores that stop them from running (semMid_r, semHi_r). The code in task 1 , for example may look similar to that shown below:

```
stack_prefault();
 sem_post(&semHi_rdy);
 sem_wait(&semHi_r);
```

- Task 3 waits on semLow_r and gets suspended.

- The lowest priority task is allowed to run first. This is done by relasing semLow_r in the main() function one second after all three tasks are created (use sleep() function). It puts a lock on the mutex then waits on semHi_rdy and semMid_rdy until the these two tasks are ready to run. Once task 3 gets a notification through two semaphores from task 1 and task2 that they are ready it releases the semaphores blocking them (semHi_r, semMid_r,) and lets them run. Task 3 then performs its work and releases the mutex.
  The code for task 3 for this item may look similar to that shown below:

```
  pthread_mutex_lock(&mutex);
  sem_wait(&semMid_rdy);
  sem_wait(&semHi_rdy);
  sem_post(&semHi_r);
  sem_post(&semMid_r);

      printf("Hello! This is task3, Priority:%d\n", param.sched_priority);
      workfor(200000000, 3);
      clock_gettime(CLOCK_MONOTONIC ,&t);
      printf("Task3 took %ld seconds, %ld nsec\n", (t.tv_sec - sec), (t.tv_ns
      pthread_mutex_unlock(&mutex);
```

- Re-build the project and run it on myRIO.
- Observe and note the outcome. The output should be similar to that shown below:

```
Task1 created: sec 13926, nsec 515545234
Task2 created: sec 13926, nsec 516099286
Task3 created: sec 13926, nsec 516224713
Hello! This is task2, Priority:42
Task 2 done spinning
Task2 took 1 seconds, 207566372 nsec
Hello! This is task3, Priority:40
Task 3 done spinning
Task3 took 4 seconds, -390183557 nsec
Hello! This is task1, Priority:45
Task 1 done spinning
Task1 took 4 seconds, -377406802 nsec
```

- Show the output to a TA.

Part 3[40]
Priority inversion protocol:

- Modify your code for part 2 such that before initializing the mutex, its protocol is set to "PTHREAD_PRIO_INHERIT".
- Make no other changes, build and run the project.
- Show the output to a TA.
- Modify your cofde such that the value for 'n' passed to the 'workfor()' function for task 2 is 500000000 instead of earlier 100000000.
- Use "PTHREAD_PRIO_NONE" for the mutex_attr.
- Build and run the project again and show the output to a TA.