Mectron/Sfwr Eng 4AA4 - Lab 6

Scheduling of real-time tasks

Introduction:

When a number of real time tasks run concurrently they need to be synchronized otherwise unintended behaviour may result. The results are also dependent on the scheduling policy used. In this lab you will learn about the consequences of un-synchronized concurrent tasks and how to synchronize them using semaphores and mutexes. You will also learn about the features of different scheduling policies implemented by an operating system.

Goals:

- Learn how to create multiple periodic tasks as threads and observe why synchronization is required.
- Learn how to synchronize concurrent tasks using semaphores.
- Learn how to use different scheduling policies and observe their consequences.

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

- Pthreads API:
  https://hpc-tutorials.llnl.gov/posix/pthreads_api/
- Pthreads and pthread mutexes at the link:
  https://hpc-tutorials.llnl.gov/posix/mutex_variables/
- Posix semaphores at the link:
  http://www.csc.villanova.edu/~mdamian/threads/posixsem.html
- More details on Pthreads and Semaphores at link:
  http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html

Part 1: Consequences of un-synchronized resource sharing [15]

- Create a new C-project in your work space as done in Lab 3 Part 1 ("Hello World") . This lab will not use the FPGA of MyRIO therefore it is not necessary to create the project from template as done in earlier labs.
- You have earlier created a real time periodic task as a thread. For this part of the lab you need to create three threads. Each thread will execute its own function. You may give them names such as "`void* tfun1(void*n)`", "`void* tfun2(void*n)`", etc.

  You can use the file named "`lab6-0.c`" as a starting point if you like.

- Before you build the project, make sure that following properties are set:

  Under **C/C++ Build>> Settings >Cross GCC Compiler>>Symbols**, add symbol: ***_GNU_SOURCE***

  Under **C/C++ Build > Settings >>Cross GCC Linker >Libraries**, add library: ***pthread***

- In the lab we have multiprocessor workstations. Make sure for all work in this lab to force your program to run on a single CPU.

- Build the project, and run it on MyRIO for several times to observe how the tasks are scheduled.

- Show the result to your TA.

- Modify your code such that task 1 increments a global variable named "`count`" of type int by one in each period. **You must use the following lines of code for this purpose:**

```
int num; // (declare it at the start of function to be executed by task 1)

//inside the while loop:
num = count;
printf("count in thread 1:%d\n", num);

clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);

num = num + 1;
count=num;
printf("---count in thread 1, after increment: :%d\n", count);

//sleep for one more interval after the global variable is updated.
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
```

- Modify your code for task 2 such that it decrements the "`count`" by one in each period. **Again you must use the following lines of code for this purpose:**

```
int num; //(declare it at the start of function to be executed by task 2)

//inside the while loop:
num = count;
printf("\t\t\t\t\t\t count in thread 2:%d\n", num);

clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);

num = num - 1;
count = num;
printf("\t\t\t\t\t\t ---count in thread 2, after decrement: :%d\n", count);
```

```
//sleep for one more interval after the global variable is updated.
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
```

- Make no changes to the code for task 3. Before the function "`main()`" returns, add a line:

```
printf("Value of count %d\n", count);
```

- Save your modified code.
- What value of "count" do you expect at the end of function "`main()`"? The "`count` was initialized to zero, task 1 was created first which incremented it 20 times. Task 2 was created next that decremented it 20 times. The result should be a zero!!
- Build your project and run it on MyRIO.
- Run the project three times. Observe and note the outcome, take a screen print for including in your report. Can you explain why the "`count`" is not zero at the end of program?

Part 2: Synchronize tasks using semaphores [10]

- Modify your final running code in part 1 of the lab as follows:
- Declare a semaphore of type "`sem_t`" with global scope and initialize it using "`sem_init()`" function in your "`main()`" function before the threads are created. Destroy the semaphore before exitng the main() function.
- Surround the statements that read and update the shared variable "`count`" in your code for both task 1 and task 2 by functions `sem_wait()` and `sem_post()`.
- Build the project and run it on MyRIO. Show the result to your TA and take screen print of the output.

Part 3: Using priority based scheduling [15]

- In parts 1 and 2 of this lab you used a scheduling policy called: "**SCHED_OTHER**" that does not assign any priority to a particular task. Real time tasks are generally assigned priorities based on their importance and the scheduling policy should ensure that tasks with higher priority run before the lower priority tasks. Real time Linux provides priority based scheduling policies named: "**SCHED_FIFO**" and "**SCHED_RR**". In this part of the lab you will use one of these two policies.
- Create a new project in your workspace, similar to the one in part 1 of this lab. that creates three threads each with a `printf` statement that prints the thread PID and its priority.
- Assign different values to "MY_PRIORITY1, MY_PRIORITY2, and MY_PRIORITY3", for example 45, 42 and 40 respectively.
- Instead of "**SCHED_OTHER**", use "**SCHED_FIFO**".

- Ensure that all three tasks are able to run at the same time. This can be done by using a "condition variable" along with a "mutex" (Read more about the condition variables and mutexes in the references provided at the start of this document, if necessary).
- Declare a mutex of type: "`pthread_mutex_t`" and a condition variable of type: "`pthread_cond_t`", both with global scope.
- Initialize them in the "`main()`" function or at the time of creation by using macros: "`PTHREAD_COND_INITIALIZER`" and "`PTHREAD_MUTEX_INITIALIZER`".
- Make all three tasks to wait on the condition variable at a suitable moment after creation, e. g. after the call to function "`stack_prefault()`", by adding the following lines of code:

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock(&mutex);
```

- In the "`main()`" function, after the code for creating the three threads, release the condition for all threads waiting on the condition variable simultaneously. This can be done by inserting the following code:

```
sleep(1); // wait for one second. Requires inclusion of <unistd.h>.

pthread_mutex_lock(&mutex);
pthread_cond_broadcast(&cond); // Releases all threads simultaneously
pthread_mutex_unlock(&mutex);
```

- Build your project and run if on MyRIO. Show the output to one of your TAs and note the order of execution of tasks.

Part 4: Modify the above program as follows [5]

- Try different orders of creation of tasks. Note that the task with the highest priority always runs first.
- Depending upon the period of tasks, the lowest priority task may never get to run! This condition is called "Starvation".
- Modify your code by adjusting the periods of tasks. Try the task periods with: 200000, 100000, 50000, 10000 and/or other values. Run your project and observer its output.
- Modify your code to run each periodic task forever instead of "num_runs" as given in sample code, and adjust the periods of tasks to create "Starvation" for the lowest priority task and show the output to a TA and take a screen print for the report. (**You will have to use the red rectangle in console window to terminate your program!!**)

Part 5: Modify the above program as follows [5]

- Assign a priority of 45 to task 1, but assign equal priorities of 40 to both task 2 and task 3
- Re-build the project and run on MyRIO.
- The task with the highest priority always runs first. Which of the two tasks with equal priorities runs next? Try different orders of creation of tasks, with different periodic task intervals, for example, with interval to be 200,000 or to be 10,000, to determine how the scheduler decides it?

Report:[10]

- In Part 1 of this lab, you got the final value of "count" other than a zero. Attach the screen print of results and explain why the final value of "count" was not zero? Also you were asked to write the code that incremented or decremented the shared variable in a certain way. Explain its significance.
- Submit the screen print of the results obtained in part 2 of the lab and explain why the final value of "count" was zero this time.
- Submit the screen print of result obtained in part 4 and explain this behavior.
- How are the tasks with equal priority selected for execution using FIFO scheduling policy?
- What is the difference between FIFO and RR scheduling policies?