CG2271 Real Time Operating Systems

Lab 5 – Task Coordination

1. Introduction

In the previous labs you did all of your programming on the Arduino. For this and the next (few) labs we'll have a change of gears; you'll be programming in UNIX. ☺ If you're not familiar with Unix, work through the Lab 5 Supplement first.

You will learn about threads in Lecture 8, but meanwhile we will use them for our labs because they're fairly decent approximations of real-time tasks. The primary aim of these labs is to teach you about task coordination and communication, and threads will do nicely. The main advantage is that you can now focus on getting the task logic right instead of trying to get LEDs and buzzers to flash and buzz correctly.

You will also return to the joys of using printf, instead of using LEDs to show that your program works. ☺

2. Submission Instructions

You will submit a HARDCOPY of the answer book to your tutor at the start of the session for lab 6. There will be a demo of the final versions of lab5a.c and lab5b.c to your tutor at the start of lab 6. The report is worth 30 marks and both demos together are worth 5 marks, giving a total of 35 marks.

3. Introduction to POSIX Threads

POSIX threads (henceforth called "pthreads") is an API specification for threads on Unix based systems. Pthreads packages are available for non-Unix systems as well. The basic thread creation, joining and destruction calls are:

Call	Description
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg)	Creates a new thread. Returns 0 if successful. Arguments: thread - A data structure which will contain information about the created thread. attr - Thread attributes. Can be NULL. start_routine - Pointer to the thread's starting function. Must be declared as void *fun(void *) arg - Argument passed to the starting routine.
void pthread_exit(void *value_ptr)	Exits from a thread. The value in value_ptr is passed to another thread that joins with this exiting thread.

void pthread_join(pthread_t thread, void **value_ptr)	Suspends execution until the thread specified by "thread" completes execution. If value_ptr is not NULL, it will point to a location containing the value passed by "thread" when it exits using pthread_exit. pthread_join returns 0 if successful.

You need to `#include <pthread.h>` to use these. We will now look at an example of how to use threads. Type out the program below on a Unix machine with pthreads installed (if unsure do this lab on Sunfire) and call it lab5a.c. Compile it using "gcc lab5a.c -o lab5a", and execute using "./lab5a". Run this program several times (at least 8-10 times) and observe the output.

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

// Global variable.
int ctr=0;
pthread_t thread[10];

void *child(void *t)
{
      // Print out the parameter passed in, and the current value of ctr.
      printf("I am child %d. Ctr=%d\n", t, ctr);
      // Then increment ctr
      ctr++;
      pthread_exit(NULL);
}

int main()
{
      int i;

      // Initialize ctr
      ctr=0;

      // Create the threads
      for(i=0; i<10; i++)
            pthread_create(&thread[i], NULL, child, (void *) i);

      // And print out ctr
      printf("Value of ctr=%d\n", ctr);
      return 0;
}
```

Question 1 (2 marks)

Do the threads print out in order? I.e. does it go "I am child 1.", "I am child 2.", etc? Or are the thread outputs mixed up? In either case explain why.

Question 2 (3 marks)

Based on your observation on the values of ctr printed by each thread, do threads share memory or do they each have their own portions of memory? Explain your answer, with reference to ctr.

Question 3 (3 marks)

Are the values of ctr as printed out by the child threads correct? Explain why or why not.

Question 4 (4 marks)

The variable "i" in main is effectively the index number of the child thread. Explain why it must be cast to (void *) before passing to the child thread, and why the child thread can successfully print out "i" without recasting it back to an int.

You will notice that the "printf("Value of ctr=%d\n", ctr);" statement in main sometimes executes even before the last thread completes. We will now use "pthread_join" to ensure that the 10th thread completes before this statement executes. To do this, add the following statement to just before the printf:

```
  // wait for the 10th thread
     pthread_join(thread[9], null);
```

Our main would therefore now look like this, with the added lines in **bold.**

```
int main()
{

      ...

      // wait for the 10th thread
      pthread_join(thread[9], NULL);
      // And print out ctr
      printf("Value of ctr=%d\n", ctr);
      return 0;
}
```

Compile and run the program, and verify that the printf no longer executes until the 10th thread completes. However you will notice that the threads still do no execute in order.

Question 5 (4 marks)

Modify the program so that ALL the threads execute in order. I.e. thread 0 executes first, then thread 1, etc. Compile your program and run it several times to verify that it works correctly, then describe the changes that you made, and cut-paste the code into your answer book. You will demo this program at the start of the next lab session.

4. Introduction to Mutexes

In multi-threaded applications there are sections of code where no more than one thread can execute at a time. For example, code that updates global variables should not be executed by more than one thread or the updating may go wrong. Such sections of code are called "critical sections".

The word "mutex" stands for "Mutual Exclusion", and the idea here is that when a thread wants to enter a critical section the thread must first successfully obtain a lock called a "mutex". When it has

the mutex, the thread can safely enter the critical section. Once it exits the critical section, the thread frees the mutex.

Only one thread can obtain this mutex, and other threads that are trying will block until the mutex is freed.

The mutex must be shared by several threads and must therefore be "global", and of type pthread_mutex_t. So to create a mutex call "my_mutex", the statement would be:

```
pthread_mutex_t my_mutex=PTHREAD_MUTEX_INITIALIZER;
```

This creates a new mutex lock, and initializes it to a default starting value. The functions below are used to manipulate the mutex:

Call	Description
int pthread_mutex_lock(pthread_mutex_t *mutex)	Locks the mutex. Blocks and does not return if mutex is already locked. If mutex locks successfully, this function returns with a 0. This function returns a non-0 value if something goes wrong.
int pthread_mutex_unlock(pthread_mutex_t *mutex)	Unlocks the mutex. Returns 0 if successful.
int pthread_mutex_destroy(pthread_mutex_t *mutex)	Destroys the mutex. Returns 0 if successful.

Type out the program below, calling it lab5b.c.

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

int glob;

void *child(void *t)
{
      // Increment glob by 1, wait for 1 second, then increment by 1 again.
      printf("Child %d entering. Glob is currently %d\n", t, glob);
      glob++;
      sleep(1);
      glob++;
      printf("Child %d exiting. Glob is currently %d\n", t, glob);
}
```

```
int main()
{
      int i;
      glob=0;

      for(i=0; i<10; i++)
            child((void *) i);

      printf("Final value of glob is %d\n", glob);
      return 0;
}
```

Question 6 (1 mark)

What is the value of glob printed at the end of main?

Question 7 (3 marks)

Now modify main so that it spawns each call to "child" as a thread, giving us 10 threads in total. Describe the changes you made to the program.

Question 8 (3 marks)

Are the values of glob now correct? Explain why or why not.

Now modify your program as shown below. Statements in **bold underline** are newly added.

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

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

int glob;

void *child(void *t)
{
      // Increment glob by 1, wait for 1 second, then increment by 1 again.
      printf("Child %d entering. Glob is currently %d\n", t, glob);
      pthread_mutex_lock(&mutex);
      glob++;
      sleep(1);
      glob++;
      pthread_mutex_unlock(&mutex);
      printf("Child %d exiting. Glob is currently %d\n", t, glob);

      ... Other code you may have added ...
}
```

```
int main()
{
      int i, quit=0;
      glob=0;

      ... Codes and declarations that make your program multi-threading

      printf("Final value of glob is %d\n", glob);
      pthread_mutex_destroy(&mutex);
      ... Other code you may have added ...
}
```

Question 9 (3 marks)

Do your threads now update glob correctly? Explain your answer, and why the updates are correct/incorrect.

Question 10 (4 marks)

You will notice that the statement "printf("Final value of glob=%d\n", glob);" in main often executes before all the threads are done, causing the wrong value of glob to be printed. Modify your program so that this statement executes only after all threads complete.

Describe the modifications made, and cut and paste the code into your answer book. You will demo that your program works at the start of the next lab session.