CG2271 Real Time Operating Systems

Lab 6 – Synchronization Mechanisms

1. Introduction

In the previous lab you worked with mutexes and thread joins. In this lab we will look at two other synchronization mechanisms available: condition variables and semaphores. In addition you will implement a queue-based inter-thread communication library.

2. Submission Instructions

As before, fill in your answers in the attached answer book, and submit in HARDCOPY to your lab tutors BEFORE the start of the session for Lab 7. There will also be a demo for this lab conducted at the start of the session for Lab 7. The answer book is worth 30 marks and the demo is worth 5 marks, totalling 35 marks.

3. Condition Variables

A condition variable is a synchronization mechanism with the pthreads library that is very similar to the condition variables used in monitors in the lecture notes. In both the monitor and pthread implementations, condition variables must be used in a mutex. The key difference is that in the monitor implementation, the mutex is provided by the monitor. In the pthreads implementation, you need to create a mutex variable to use with the condition variable. The table below summarizes the functions associated with condition variables:

Condition variables are declared as type pthread_cond_t. E.g.:

```
pthread_cond_t cv;
```

The key functions are:

Function	Parameters	Comment
pthread_cond_init(&cv, attr)	cv = Condition variable attr = Attributes for the condition variable This is normally NULL.	Initializes the given condition variable.
pthread_cond_wait(cv, cm)	cv = Condition variable. cm= Mutex	Waits on condition variable cv until there is a signal. The mutex cm must be locked. Before the signal: This call will block, and cm will be released. After the signal: This call exits back to the thread and cm is automatically re-locked.
pthread_cond_signal(cv)	cv=Condition variable	Signals and wakes a thread that is waiting on cv.
pthread_cond_broadcast(cv)	cv=Condition variable	Signals and wakes up ALL threads that are waiting on cv.
pthread_cond_destroy(&cv)	cv = Condition variable	Condition variable cv is destroyed.

We will now see how to use condition variables. Key in the following program calling it lab6_1.c, compile and execute it, and answer the questions that follow:

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

#define MAX_COUNT 10

pthread_mutex_t ctr_mutex;
pthread_cond_t ctr_cond;
int ctr;

void *add(void *p)
{
      /* Get a mutex to update ctr */

      while(ctr < MAX_COUNT)
      {
            pthread_mutex_lock(&ctr_mutex);
            ctr++;
            printf("ADD: New value of ctr is %d\n", ctr);
            if(ctr>=MAX_COUNT)
            {
                  printf("ADD: Reached limit of %d! Waking up print
thread\n", MAX_COUNT);
                  pthread_cond_signal(&ctr_cond);
            }
```

```
            pthread_mutex_unlock(&ctr_mutex);
            sleep(1);
      }
      pthread_exit(NULL);
}

void *print(void *p)
{
      pthread_mutex_lock(&ctr_mutex);
      printf("PRINT: Current value of ctr is %d\n", ctr);

      if(ctr < MAX_COUNT)
      {
            printf("PRINT: ctr is too small. Zzzz.. g'night!\n");
            pthread_cond_wait(&ctr_cond, &ctr_mutex);
            printf("PRINT: ctr is now %d! Exiting thread.\n", ctr);
      }

      pthread_mutex_unlock(&ctr_mutex);
      pthread_exit(NULL);
}

int main()
{
      pthread_t threads[2];

      ctr=0;
      pthread_mutex_init(&ctr_mutex, NULL);
      pthread_cond_init(&ctr_cond, NULL);
      pthread_create(&threads[0], NULL, add, NULL);
      pthread_create(&threads[1], NULL, print, NULL);

      pthread_join(threads[1], NULL);
      pthread_mutex_destroy(&ctr_mutex);
      pthread_cond_destroy(&ctr_cond);
      pthread_exit(NULL);
}
```

Question 1 (2 marks)

Run your program, copy and paste the output to your answer book, and describe what this program does.

Question 2 (3 marks)

Although the "print" thread is blocked at the pthread_cond_wait call while still holding on to the mutex "ctr_mutex", this program does not deadlock. Explain why.

4. Semaphores

POSIX also specifies a semaphore interface for synchronization. This interface is shown below. To access this interface, use "#include <semaphore.h>"

To declare a semaphore called "sema", use:

```
sem_t sema;
```

Function	Parameters	Comment
sem_init(&sema, flag, initval)	sema = semaphore flag = Whether this semaphore can be shared with forked processes. 0=No, 1=Yes. initval = Initial value.	The semaphore sema is initialized to initval.
sem_wait(&sema)	sema = semaphore	Decrements sema and blocks if sema becomes negative.
sem_post(&sema)	sema=semaphore	Increments the semaphore and wakes up a blocked process.
sem_destroy(&sema)	sema=semaphore	Destroys the semaphore.

Type in the following program calling it lab6_2.c. Compile the program, execute it and answer the questions that follow:

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

sem_t sema1;

void *child(void *p)
{
      sem_wait(&sema1);
      printf("Child %d woken up!\n", p);
      pthread_exit(NULL);
}

int main()
{
      pthread_t threads[10];
      int i;

      sem_init(&sema1, 0, 1);

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

      for(i=0; i<10; i++)
      {
            printf("Sending wake up call number %d\n", i);
            sem_post(&sema1);
      }
```

```
        sem_destroy(&sema1);
        pthread_exit(NULL);
}
```

Question 3 (2 marks)

Compile and run your program using:

```
gcc lab6_2.c -lrt lab6_2.c -o lab6_2
```

Notice the "-lrt" which links in the semaphore library. Without this the program will not compile.

Now explain why there is at least one child thread that does not block at the "sem_wait" before the main thread does its first "sem_post".

Question 4 (2 marks)

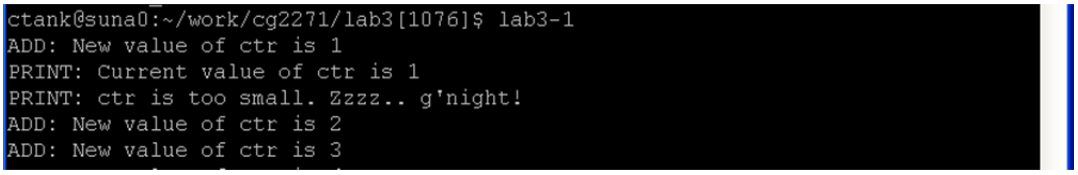
How many threads are woken up each time the main thread calls sem_post?

Question 5 (2 marks)

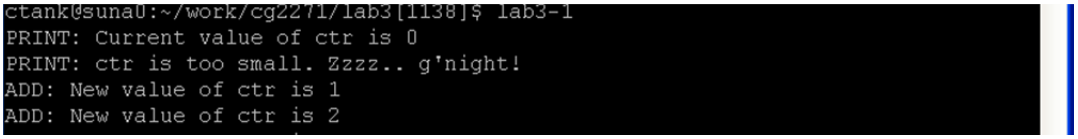
Explain how POSIX semaphores and different from POSIX mutexes.

Question 6 (3 marks)

Run lab6_1 again. You will notice that sometimes ADD runs before print, giving an output like this:



Modify your lab6_1.c so that PRINT always executes before ADD, giving the following output each time the program is run:



Summarize the changes you made to lab6_1.c to achieve this.

5. Putting it All Together

We will now put everything we've learnt in labs 6 and 7 to build a communication library for POSIX threads. You are not allowed to use any other synchronization structures (e.g. reader-writer locks, barriers) other than those that have already been covered.

The specification for this library is given below:

Function	Parameters	Comments
pq_t *pq_create(int size);	size = size of the queue.	Creates a queue of size "size" in integers, where "size" can be arbitrarily large. Returns the queue if successfully created, or NULL otherwise.
void pq_put(pq_t *q, int b);	b = data to be added to the queue. q=A queue created using pq_create.	This function blocks if the queue is full, otherwise "b" is added to the queue.
int pq_get(pq_t *q);	q = A queue created using pq_create.	This function blocks if the queue is empty, otherwise it de-queues the first item in the queue and returns it.
void pq_destroy(pq_t *q)	q = A queue to destroy.	This function destroys queue q.

A skeleton implementation has been done for you in pcomm.c, and the necessary headers, etc are in pcomm.h. In addition there is a test routine in lab6_3.c.

Question 7 (2 marks)

The test program consists of 5 writer threads, with thread 0 writing 0 to 9 to the queue, thread 1 writing 10 to 19, etc, with a single reader thread reading and summing the numbers written to the queue. Assuming that all the implementation is correct, what is the final sum of the numbers?

Question 8 (3 marks)
Compile your program using the following command:

```
gcc pcomm.c lab6_3.c –o lab6_3
```

Then execute the lab6_3 program. Does it give the correct sum? Explain why or why not.

Note about Question 8: It may take up to a minute or two for the writer threads to exit. this is not a bug.

Question 9 (6 marks)

Locate the following statement in main:

```
queue=pq_create(100);
```

and change it to:

```
queue=pq_create(10);
```

Now make modifications to pcomm.c and pcomm.h so that the message passing mechanism works correctly. In particular pq_put should block when full, pq_get should block when empty, and you should get the correct sum when running lab6_3. Summarize the changes made.

NOTE:
Your pcomm implementation MUST support multiple queues. I.e. it is possible that in a single program you might have:

```
pq_t *q1, *q2;
...
q1=pq_create(10);
q2=pq_create(15);
...
pq_put(q1, 10);
...
val=pq_get(q2);
```
etc.

When this happens the queues should not be mutually blocking. I.e. an access to q1 MUST NOT block any accesses to q2!

Question 10 (5 marks)

At the moment pq_destroy at the end of main thread is commented out because it will destroy the queue before all the threads exit. Use a semaphore so that pq_destroy is called only after ALL threads have finished using the queue. Explain the changes you have made here.

Compile and demonstrate your working program lab6_3 to your tutor at the start of the lab session for Lab 7.