Lab 07 - Synchronization Mechanisms

Objectives

1. Learn how to create threads using the pthread library.
2. Get an introduction to synchronization issues.
3. Solve the bounded buffer problem.

Pre-reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Pthreads Library

Historically, each hardware vendor had its own proprietary threads implementation. This made writing portables threaded application a very challenging and difficult task and created the need for a standardized programming interface. Such a programming interface was specified for UNIX systems by the IEEE POSIX 1003.1c standard (1995) and it's called the POSIX Threads library (pthreads library).

Pthreads library consists of a set of C programming types and function calls implemented in the `pthread.h` header file. These function calls can be grouped into three major classes:

- **thread management** – class of functions responsible with creating, joining, detaching threads, setting and querying thread attributes.
- **mutexes** – functions dealing with a synchronization primitive called a mutex (mutual exclusion): creating, destroying, locking and unlocking mutexes.
- **condition variables** – class of functions that facilitate the communication between threads sharing a mutex. They include functions for creating, destroying, waiting and signaling condition variables.

Creating a thread

A new thread is created using the following function:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

This will create a new thread in the current process staring with **`start_routine`**. This function must take a **`void*`** parameter and return void*. The **`arg`** parameter will be passed to the **`start_routine`** function when the thread starts. The first argument is a thread handler set by the **`pthread_create`** function that can be used to reference the thread later and the second argument is used to specify various thread attributes. Usually this argument can be **`NULL`** in which case the default thread attributes are used.

A thread terminates when it returns from its **`start_routine`** or may be explicitly terminated using the following function:

```
void pthread_exit(void *value_ptr);
```

The **`value_ptr`** argument is the termination value and it's made available on any subsequent join with the thread. A thread may be joined using the following function call:

A call to pthread_join blocks the calling thread until the thread referenced by the first argument terminates and if the second argument is not NULL the thread return value is stored at that address.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Synchronization primitives

Every modern kernel provides one or more synchronization primitives to enable application programmers to coordinate the behavior of multithreaded software. Classic UNIX systems did not provide mechanisms for fine-grained multithreaded operation, the primary means for synchronizing processes was pipes (and files).

Mutexes are basically locking mechanisms: to ensure only one thread operates in the critical section, a thread must first acquire the lock. After the thread has the lock, no other thread can access the shared resource until the other thread releases it first.

A mutex example is:

```
// notice the scope of variable and mutex are the same
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

int counter = 0;

void someFunction() {

      pthread_mutex_lock( &mutex1 );

      // the counter variable is incremented in this critical section
      counter++ ;

      pthread_mutex_unlock( &mutex1 );
}
```

So, what to do if the lock can't be obtained because it is in use? A simple solution is called a 'busy wait' (or spin locking). Checking in a tight loop until the lock can be obtained – not a very efficient use of CPU time. It is therefore preferable to block the thread until the lock has been released, but how do we coordinate between the threads? The thread that has just unlocked the mutex needs a way to signal that the other thread -- currently blocked -- that it may now acquire the lock. Looking at the pthread library you will notice the **`pthread_cond_signal()`** function call. Reading **`man`** pages you find that this call will unblock at least one of the threads that are blocked on a specified condition variable (supplied as an argument).

In this example, the conditional signaling argument is of type **`pthread_cond_t`**. The **`man`** page will give you the information on how to initialize it.

```
{
	...

	// acquire lock
	pthread_mutex_lock(&mutex);

	// do some work on the shared buffer here
	...

	// signal the other thread to un-block
	pthread_cond_signal(&cond);

	// release the lock
	pthread_mutex_unlock(&mutex);
}
```

If we think now at the problem we are trying to solve in this lab (the bounded-buffer problem), we need to find a way to synchronize the access to the shared buffer, to respect the filling and emptying of the data in the buffer? How do we block the producer thread if the buffer is full? What about the case where the consumer is trying to read from a buffer that is empty? Again we go back to the pthread library and find two methods for blocking, or waiting: **`pthread_cond_wait()`** and **`pthread_cond_timedwait()`**. It's your job to read the manual pages and figure out how they work and how they must be used.

In the following code fragment, the producer has just acquired the lock, and needs to add new items to the buffer. Notice where the **`pthread_cond_wait()`** is used. Why so you think the call to **`pthread_cond_wait()`** must be placed within a while loop?

```
{
	...

	// debug info
	if( buffer.occupied >= BSIZE)
		printf("producer waiting, full buffer ... \n");

	// wait condition
	while( buffer.occupied >= BSIZE)
		pthread_cond_wait(&(buffer.less), &(buffer.mutex) );

	// add to the buffer
	buffer.buf[buffer.nextin++] = item[i];
	buffer.nextin %= BSIZE;
	buffer.occupied++;

	...
}
```

Procedure

Part A

Solve the bounded buffer problem. You may use the following code skeleton to get you started:

File: bbuffer.c

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

// shared buffer, adjust this size and notice the interleaving
#define BSIZE 3
typedef struct {
      char buf[BSIZE];
      int occupied;
      int nextin, nextout;
      pthread_mutex_t mutex;
      pthread_cond_t more;
      pthread_cond_t less;
} buffer_t;
buffer_t buffer;

// running flag for thread termination
int done = 0;
int sleep_time;

// threads, and pointers to associated functions
void* producerFunction(void *);
void* consumerFunction(void *);
pthread_t producerThread;
pthread_t consumerThread;

//
// program entry point
//
int main( int argc, char *argv[] ){

      // sanity check
      if( argc != 2 ){
            printf("useage: bound sleeptime \n");
            return(0);
      }

      // get parameters
      sleep_time = atoi(argv[1]);

      pthread_cond_init(&(buffer.more), NULL);
      pthread_cond_init(&(buffer.less), NULL);

      pthread_create(&consumerThread, NULL, consumerFunction, NULL);
      pthread_create(&producerThread, NULL, producerFunction, NULL);

      pthread_join(consumerThread, NULL);
      pthread_join(producerThread, NULL);
```

```
	printf("main() exiting properly, both threads have terminated. \n");
	return(1);
}


void* producerFunction(void * parm){

	printf("producer starting... \n");

	// objects to produce, place in buffer for the consumer
	char item[]= "More than meets the eye!";

	int i;
	for( i=0 ;; i++){

		// done producing when end of null terminated string
		if( item[i] == '\0') break;

		// acquire lock
		if( ... ) printf("producer has the lock. \n");

		// debug info
		if( ... ) printf("producer waiting, full buffer ... \n");

		// wait condition
		while( ... )
			pthread_cond_wait(&(buffer.less), &(buffer.mutex) );

		// add to the buffer
		...

		// debug info
		printf("producing object number: %i [%c]\n", i, item[i]);

		// signal the producer, release the lock
		....

		// tell consumer we are no longer producing more items
		// by setting the done flag if this is the last element
		if( ... )
			done = 1;

		// impose a delay to show mutual exclusion
		sleep(sleep_time);
	}

	printf("producer exiting. \n");
	pthread_exit(0);
}


void* consumerFunction(void * parm){

	printf("consumer starting \n");

	char item;
	int i;
	for( i=0 ;; i++ ){
```

```
            // is the producer still running?
            if( ... ) break;

            // acquire lock
            if( pthread_mutex_lock(&(buffer.mutex)) == 0 )
                  printf("consumer has the lock. \n");

            // debug info
            if ( ... ) printf("consumer waiting, empty buffer ... \n");

            // wait condition
            while(...)
                  pthread_cond_wait(...);

            // consume from buffer by displaying to the terminal
            item = buffer.buf[buffer.nextout++];
            printf("consuming object number %i [%c]\n", i ,item);

            // now there is room in the buffer for the producer to add
            ...

            // signal the producer, and release the lock
            ...
      }

      printf("consumer exiting. \n");
      pthread_exit(0);
}
```

Part B

Now modify the BSIZE value and explain the difference in the output in a short paragraph. Next reverse the order of these lines in both producer and consumer, and explain the difference in output in a short paragraph. Which order makes more sense and works properly?

```
// signal the other thread to un-block
pthread_cond_signal(&(buffer.xxx));

// release the lock
pthread_mutex_unlock(&(buffer.mutex));
```

Part C

To ensure that the sleep call in the producer thread doesn't alter the mutual exclusion, create a batch/script file to run your program with different delay times.

Example only, your program may take different augments.

```
> ./bound 0 > t0
> ./bound 1 > t1
> diff t0 t1
```

If a device was producing data at random and in bursts, you must be sure that the rest of the system has no effect on your code. Modify your sleep call to be random to simulate a real device. Does it produce a different output file?

You will call your program as follows for a random sleep time.

```
> bound -r 3
```

which would use a random delay between 0 and 3 seconds.

To do this, use the **`diff`** utility against your output from your program. If you're not familiar with this utility, check the **`man`** pages.

Deliverables

You should have only one source code file, a make file and a testing script to submit. The questions from part B should be answered in a plain text file called partB.txt. Do not submit your output files; your test script is all the marker will require.