CS444 2/02/06

Lab Exercise 03

Note

All the code that is found in this lab guide can be found at the following web address:
www.clarkson.edu/~jets/cs444/sp06/labs/labex03

Threading

A thread is an independent stream of instructions that is meant to execute in parallel with other independent streams of instructions. Up until now, you have probably only ever written single threaded programs. This means that within your program, there is only one thing happening at any given time, there is only one single procedure being executed at any give time. With multi-threaded applications, there can be multiple things going on. Lets take a look at a very simple single threaded application.

```
#include<stdio.h>

int main()
{
   printf("Hello.\n");
   return 0;
}
```

When executed, This small program will print "Hello" on a single terminal line and then exit. There is only one thread, and nothing is done in parallel. In contrast, a multi-threaded application could be printing things to the screen while at the same time calculating pi or playing chess. The concept of multi-threading should not be new to you. When ever you open a web browser, type in www.google.com, and hit enter you are dealing with multiple threads. One thread is contacting the web server for www.google.com while another thread is waiting for you to press a button so that it can process your input. If your web browser was not multi-threaded, you would not be able to interact with it AT ALL while it was trying to load a web page.

Multi-threading in Linux

The Linux operating system does not support the concept of threads. So instead of relying on built in OS functionality to write a multi-threaded application we must use a special library called pthreads. Lets see what a pthreads based multi-threaded application looks like. Examine the code below and see if you can figure out how it works.

[unixcode/threading/threading.c]

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

void* my_thread(void* args);

int main()
{
   pthread_t thread1, thread2;
   int arg1 = 1, arg2 = 2;

   pthread_create(&thread1, NULL, my_thread, &arg1);
   pthread_create(&thread2, NULL, my_thread, &arg2);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   return 0;
}

void* my_thread(void* args)
{
   int* arg = (int*) args;
   printf("I am thread %d\n", *arg);
   pthread_exit(NULL);
}
```

This code creates two threads and then waits for them to exit. Both of these new threads will end up executing the 'my_thread' function. The only difference between the threads is the argument that is passed to them. The first thread is passed a pointer to an integer that has the value 1 and the second thread is passed a pointer to an integer that has the value 2. When executed, both threads will print out their “I am thread %d” string and exit. The result, two printed lines will appear on the terminal. You may notice that thread1 always seems to get to print before thread2, the reason for this is that thread1 is started just a fraction of a second sooner than thread2 and the print operation is so fast that it is usually completable in a single time slice, so thread1 gets

to finish its work before thread2 even gets a chance to start. later we'll be looking at a more complicated example where this is not the case.

Before you can begin to use pthread related functions and data types, you will need to include 'pthread.h' in your program.

Take a look at the 'pthread_t' variables that we create at the top of main(). These variables will represent the two threads that we will be creating.

Next is the call to the 'pthread_create' function. The first argument is just a pointer to the pthread_t variable that will represent our new thread, the second argument is NULL, and the third argument is a pointer to a function. yes, thats right, functions (function pointers) can be passed as arguments to other functions. This function contains the code that the new thread will execute. The last argument to pthread_create must be a pointer. It can point to anything you want it to but it absolutely must be a pointer. This pointer will get passed as an argument to the function that your new thread will begin executing.

Now look at the 'my_thread' function for a moment. This function has been crafted in such a way that it can be used as a thread. What are the requirements for a function to be usable as a thread, you ask? Well, the function must return a void pointer and take a single void pointer argument. These are the only requirements. A void pointer is a generic pointer that can point to any type of variable/struct that you wish. This allows us to pass an argument to our thread, and to get a return value from the thread when it is done executing.

The last thing we must do in a multi-threaded environment is ensure that our threads are done executing before we exit. This is exactly what the 'pthread_join' function does. Joining a thread is very similar to using waitpid() to wait for a process to exit. 'pthread_join' takes as its first argument the thread that you wish to wait for. The second argument to 'pthread_join' is a pointer that will point to whatever the thread returns via the 'pthread_exit' function. If the second argument is null, the thread's return value is ignored.

Multi-threading in Windows

Multi-threading in Windows is very similar to Multi-threading in Linux. Instead of using pthreads, you need to use Window's built in thread functionality.

To create a thread in Windows you will use the 'CreateThread' function. To wait for that thread to end you will use the 'WaitForSingleObject' function. Instead of explaining all of the

differences, lets examine a Windows version of the simple multi-threading application that we looked at above.

[wincode/threading/threading.c]

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

void* my_thread(void* args);

int main()
{
   HANDLE thread1, thread2;
   int arg1 = 1, arg2 = 2;

   thread1 = CreateThread(NULL, NULL,(LPTHREAD_START_ROUTINE)
                                  my_thread, &arg1, NULL, NULL);
   thread2 = CreateThread(NULL, NULL,(LPTHREAD_START_ROUTINE)
                                  my_thread, &arg2, NULL , NULL);

   WaitForSingleObject(thread1, INFINITE);
   WaitForSingleObject(thread2, INFINITE);

   return 0;
}

void* my_thread(void* args)
{
   int* arg = (int*) args;
   printf("I am thread %d\n", *arg);
   return NULL;
}
```

Producer-Consumer

Picture an environment where you have three entities. One, a queue that can hold objects. Two, a producer whose job is to produce objects and put them in the queue as fast as he can. Third, a consumer whose job is to consume objects in the queue as fast as possible. This model is known as the Producer-Consumer model.

The Producer-Consumer model is interesting for several reasons. First, it is a very good example of a situation that requires locking. Can you understand why it requires locking? Think about it. We have two threads, one is the producer and one is the consumer. Both are trying to access objects in the queue simultaneously. What happens if the producer produces a number and begins writing it to the queue and before he gets the chance to finish writing, the consumer comes along and consumes the partially written number?!?! Instant data corruption!!

The number that the consumer just read is incomplete, and the producer will finish writing the other half of that number into the queue, so the next time the producer asks for a number, he'll get the other half. See how this works?

To avoid the data corruption mentioned in the last paragraph, we will make use of Mutexes and Semaphores. These are types of locks that are used to ensure that only one thread can access data at any given time. Don't worry too much about understanding how locking works right now. Locking will be covered in full detail during a future class. For now, just concentrate on the code that creates and manages the threads.

Here is a simplified version of producer-consumer done in Linux using pthreads. No locking code is present in this example.

[unixcode/prodcomm/prodcomm.c]

```
#define PRODUCER_SLEEP_S   1     //in seconds
#define CONSUMER_SLEEP_S   2
#define QUEUESIZE          5
#define LOOP               20

int main ()
{
   queue *fifo;
   pthread_t pro, con;
   fifo = queueInit ();

   pthread_create (&pro, NULL, producer, fifo);
   pthread_create (&con, NULL, consumer, fifo);
   pthread_join (pro, NULL);
   pthread_join (con, NULL);
   queueDelete (fifo);

   return 0;
}

void *producer (void *q)
{
   queue *fifo;
   int i;

   fifo = (queue *)q;

   for (i = 0; i < LOOP; i++)
   {
      queueAdd (fifo, i);
      printf ("producer: produced %d.\n",i);
      usleep (PRODUCER_SLEEP_S * 1000000);
   }

   return (NULL);
}
```

```
void *consumer (void *q)
{
   queue *fifo;
   int i, d;

   fifo = (queue *)q;

   for (i = 0; i < LOOP; i++)
   {
      queueDel (fifo, &d);
      printf ("consumer: recieved %d.\n", d);
      usleep(CONSUMER_SLEEP_S * 1000000);
   }

   return (NULL);
}
```

Notice that the main function creates two threads and then waits for them to finish. One thread executes the producer function that fills the shared queue with numbers. The other thread executes the consumer function which removes numbers from the shared queue.

Lab Exercise

1. Download the producer consumer code from the course webpage: www.clarkson.edu/class/cs444/cs444.sp2005/labs/lab03/code/ Ensure that you grab the portable code. [code/portcode/portcode.c]. This code will compile in Windows or Linux. Look at the code and understand how the portability works.

2. Re-write the simple “Hello” multi-threading example from [code/unixcode/threading/threading.c] so that it is portable and compiles and runs on both Linux and Windows without code modification. You may consult the Windows version of threading.c as well [code/wincode/threading/threading.c].

3. Compile and test your new code in Linux. You should compile with the following command: 'gcc -DUNIX -lpthread threading.c -o threading'

4. Compile and test your new code in Windows. You should change the project properties. Right click on your project's name in the file window and go to properties, the find the options for “command line” in the left hand view and add /D “WINDOWS” to the command line options to use when compiling.

References

www.cs.nmsu.edu/~jcook/Tools/pthreads/pc.c
www.cs.nmsu.edu/~jcook/Tools/pthreads/library.html

msdn.microsoft.com/library/en-us/dllproc/base/createthread.asp
msdn.microsoft.com/library/library/en-us/dllproc/base/waitforsingleobject.asp
msdn.microsoft.com/library/en-us/sysinfo/base/closehandle.asp
msdn.microsoft.com/library/en-us/dllproc/base/createmutex.asp
msdn.microsoft.com/library/en-us/dllproc/base/releasemutex.asp
msdn.microsoft.com/library/en-us/dllproc/base/createsemaphore.asp
msdn.microsoft.com/library/en-us/dllproc/base/releasesemaphore.asp
msdn.microsoft.com/library/en-us/dllproc/base/sleep.asp

Man Pages

pthread_create
pthread_join
pthread_mutex_lock
pthread_cond_wait
pthread_cond_signal
man usleep