Akira DeMoss
CPRE 308
Lab 4
Lab Section M

Summary

This lab has been a good review of mutex, conditional variables and intro to semiphores. It was interesting learning about semiphores especially and how you are able to implement them in a queue. Getting hands on with the API has been helpful in my learning of how semaphores work. In regards part 4 of the lab, I basically used the semaphores example to solve the problem of printing the jobs from the queue. Again similarly to project 1 the hardest part was dealing with pointers, but this time we had the added complexity of also understanding a large amount of other code on top of this. I think this lab would be a lot better if we had a partner to work with and collaborate just because there is so much information to digest and if you miss a small detail you could easily wind up stuck for several hours.

Review from lab 3

1. Memory management with threads

In the previous lab, the subjects of creating and joining threads were explored. There are quite a few applications that these ideas could be applied to, but these applications are not exactly the most interesting. Most applications that do interesting things that use threading will most likely have to share variables between threads at some point. This lab will cover the subject of how to share information & variables between threads in a way that produces the desired result.

2. If you like it, then you shoulda put a mutex lock on it

For the task in the previous lab, there was no reason for variables & information to be shared between multiple threads as each thread computed its own value in the result matrix without need of input from another thread. Though great, it's not a very interesting problem to solve. For more interesting problems, there is a need to share memory between threads. A trivial example of this would be to increment a shared variable between multiple threads. The program to do that is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
void *threadCounter(void *param){
   int *args = (int *)param;
   int i;
   for(i = 0; i < 1000000; i++){
       (*args)++;
   }
}
int main(int argc, char** argv){
   pthread_t t1;
   pthread_t t2;
   int shared = 0;
   int err;
   err = pthread_mutex_init(&lock, NULL);
   if(err != 0){
       errno = err;
       perror("pthread_create\n");
       exit(1);
   }
     err = pthread_create(&t1, NULL, threadCounter, (void *)&shared);
   if(err != 0){
       errno = err;
       perror("pthread_create");
       exit(1);
   }
     err = pthread_create(&t2, NULL, threadCounter, (void *)&shared);
   if(err != 0){
       errno = err;
       perror("pthread_create");
       exit(1);
   }
   err = pthread_join(t1, NULL);
   if(err != 0){
       errno = err;
       perror("pthread_join");
       exit(1);
   }
     err = pthread_join(t2, NULL);
   if(err != 0){
       errno = err;
       perror("pthread_join");
       exit(1);
   }
   printf("After both threads are done executing, `shared` = %d\n", shared);
   return 0;
}
```

Save this code snippet as *threaded_count.c*, compile it, and run the program a couple of times, observe
the outputs and answer the following questions:
(a) What is the expected output?
2,000,000

(b) What is the calculated output?
1057980

(c) What caused the discrepancy between the expected and calculated values?
At a lower level, the threads are able to access (read and write) the shared value at the same time because this program does not contain a lock, hence why the actual calculated value is lower than our expected output.

3. Ride into the critical section

One way to avoid the error that occurs in the threaded counter program is for the individual threads to lock the area of code where the accumulation of shared is performed, and unlock it once the accumulation is complete for that individual thread. This area is known as the critical section. To maximize performance, it is preferred that the critical section is as small as possible. To perform these locks, the following lines of code are needed:

```
pthread mutex t lock ;
void ∗ threadFunction ( void ∗ args ) {
...
p t h r e a d m u t e x l o c k (& l o c k ) ;
// s t a r t o f c r i t i c a l s e c t i o n
...
// end o f c r i t i c a l s e c t i o n
p t h r e a d m u t e x u n l o c k (& l o c k ) ;
...
}
i n t main ( i n t argc , c h a r ∗∗ argv ) {
...
e r r = p t h r e a d m u t e x i n i t (& l o c k , NULL ) ;
...
e r r = p t h r e a d m u t e x d e s t r o y (& l o c k ) ;
return 0;
}
```

For more information regarding the init, lock, and unlock calls, consult man 3p *pthread_mutex_init*, man 3p *pthread_mutex_lock*, and man 3p *pthread_mutex_unlock* respectively. As seen above, the variable lock is declared as a global variable so that all the threads can get access to it. It is initialized in the main thread using *pthread_mutex_init*, and the threads use it to lock critical sections using pthread mutex lock. Once the critical section is complete, the critical section is unlocked using *pthread_mutex_unlock*. Finally, before the program exits, destroy the mutex using the *pthread_mutex_destroy* function call.

Now, add the mutex and the calls to pthread mutex lock and pthread mutex unlock to the counting thread functions in *threaded_count.c*, compile it, run it, and answer the following questions:

(a) Did this fix the issue with the original code?
Yes, output: After both threads are done executing, `shared` = 2000000

4. See the threads in the streets, with not enough to do

Using mutexes to lock and unlock are great for avoiding race conditions, like what was shown in *threaded_count.c*, but it doesn't do a very good job of keeping threads from executing when we do not want them to.

Suppose that a program has one producer thread P, and two consumer threads C1 and C2. To ensure correctness of this program and to avoid duplicit computations, the critical sections of this program should be when P writes elements to the queue, and when either C1 or C2 reads from the queue. This would work fine, but there is a problem.

Suppose that the program was implemented naively, making the critical section of P, C1, and C2 are quite lengthy. During execution, P locks the mutex, produces data, writes to the queue, and releases the mutex. Then C1 gets to execute, going in to its critical section. While C1 is in its critical section, C2 gets scheduled to execute. Due to C1 still being in the critical section, C2 cannot get the lock, and thus cannot execute. A bit later, C1 finishes the execution of its critical section, and unlocks the mutex. Then P executes and goes into its critical section. While P is in its critical section, C2 gets scheduled to execute. Since P is in its critical section, C2 cannot get the lock and cannot execute. Then P finishes execution in its critical section, and unlocks the mutex. Then C1 goes next, and the cycle repeats. We see that C2 is never able to do anything, due to the fact that either P or C1 has the lock when C2 tries to get it. This situation is known as starvation. Since that is not desirable, conditional variables and semaphores should be used to avoid starvation.

5. Waiting on the conditional variable to change

Conditional variables are used to ensure that threads wait until a specific condition occurs. Using the example presented in the previous section, answer the following questions:
(a) What is the minimum number of conditions needed for the example to work as intended?
We'll need at least 3 signals & a conditional variable.

(b) What would those conditions be, and which thread(producer or consumer) should wait on that condition?

We’ll need at least 3 signals, one signalling that P is finished and C1 can lock in, another to signal that C1 has finished and C2 can lock in, and a final to signal that C2 is finished and P can produce.

To use conditional variables, the following function calls are needed:

```
#i n c l u d e <p t h r e a d . h>
int test var ;
pthread cond t generic condition ;
pthread mutex t lock ;
void ∗ genericThread0 ( void ∗ args ) {
p t h r e a d m u t e x l o c k (& l o c k ) ;
// do awesome s t u f f
p t h r e a d c o n d s i g n a l (& g e n e r i c c o n d i t i o n ) ;
test var = 1;
p t h r e a d m u t e x u n l o c k (& l o c k ) ;
}
void ∗ genericThread1 ( void ∗ args ) {
p t h r e a d m u t e x l o c k (& l o c k ) ;
w h i l e ( t e s t v a r == 0 ) {
p t h r e a d c o n d w a i t (& g e n e r i c c o n d i t i o n , &l o c k ) ;
}
// d o e s fun t h i n g s
p t h r e a d m u t e x u n l o c k (& l o c k ) ;
}
...
i n t main ( i n t argc , c h a r ∗∗ argv ) {
4test var = 0;
...
e r r = p t h r e a d m u t e x i n i t (& l o c k , NULL ) ;
...
e r r = p t h r e a d c o n d i n i t (& g e n e r i c c o n d i t i o n , NULL ) ;
...
e r r = p t h r e a d c o n d d e s t r o y (& g e n e r i c c o n d i t i o n ) ;
return 0;
}
```

For more information regarding the cond_init, cond_wait, cond_signal(and in extension cond_broadcast), and cond_destroy please consult man *pthread_cond_init*, man 3 *pthread_cond_wait*, man 3 *pthread_cond_signal*, and man 3 *pthread_cond_destroy* respectively.

genericThread1 will attempt to lock the mutex, test the value of *test_var*, and call *pthread_cond_wait* to see if the conditional variable has been signaled. If not, the thread will block, and *pthread_cond_wait* will not return. However, according to the man pages, this block does not last forever, and should be re-evaluated each time *pthread_cond_wait* returns; hence the while loop that surrounds the call to *pthread_cond_wait*. If the conditional variable has been signaled, then *pthread_cond_wait* would return, and the thread calling it would get the mutex. The value of *test_var* would then be tested, fall through, fun things are performed, and the mutex is unlocked.

One all is said and done, remove the conditional variable using *pthread_cond_destroy*.

To see an example of conditional variables, please take a look at *cond_example.c*. Make sure that everything pertaining to condition variables, such as how it's created, and used, is understood before compiling it. Run the compiled code, and put the output value of the program into your report.

6. Why not semaphore; you've got to declare yourself openly

Semaphores perform a similar task to conditional variables, and they are slightly easier to use. Semaphores come it two flavors, Named, and Unnamed. The differences between the two are in how they are created, and destroyed. For simplicity, the unnamed flavor of semaphores will be covered in this handout. To use semaphores, the following function calls and includes are needed:

```
#i n c l u d e <semaphore . h>
sem t semaphore ;
...
void ∗ genericThread0 ( void ∗ args ) {
p t h r e a d m u t e x l o c k (& l o c k ) ;
e r r = sem wait (&semaphore ) ;
...
// do awesome s t u f f
e r r = s e m p o s t (&semaphore ) ;
...
p t h r e a d m u t e x u n l o c k (& l o c k ) ;
}
void ∗ genericThread1 ( void ∗ args ) {
e r r = sem wait (&semaphore ) ;
p t h r e a d m u t e x l o c k (& l o c k ) ;
...
// d o e s fun s t u f f
5e r r = s e m p o s t (&semaphore ) ;
...
p t h r e a d m u t e x u n l o c k (& l o c k ) ;
}
i n t main ( i n t argc , c h a r ∗∗ argv ) {
...
e r r = s e m i n i t (&semaphore , 0 , 1 ) ;
...
e r r = s e m d e s t r o y (&semaphore ) ;
...
return 0;
}
```

For more information on the init, wait, post, and destroy functions, consult man 3 sem init, man 3 sem wait, man 3 sem post, and man 3 sem destroy respectively.

Note that the calls to pthread mutex lock and pthread mutex unlock do not necessarily have to be where they are shown in the above code, i.e., the mutex lock does not have to occur before the semaphore wait, and the mutex unlock doesn't have to occur after the semaphore post.

For similar reasons to lock and generic condition, semaphore is declared as a global variable. It is initialized in main using sem init with the value for pshared set to 0(meaning the semaphore is only shared between the threads of the current process), and it's initial value will be 1(last argument to sem init).

genericThread0 decrements semaphore by one using sem wait. If genericThread1 attempts to decrement semaphore when it got to its call to sem wait before genericThread0 incremented the semaphore by calling sem post, then genericThread1 will block right at its sem wait line. This is because decrementing a semaphore past 0 will not evaluate. So if a semaphore is already at a value of 0, decrementing it with sem wait will cause the thread calling sem wait to wait until the value of the semaphore is incremented to a value greater than 0.

Once everything is completed, destroy the semaphore by calling sem destroy on semaphore.

For an example of semaphores being used, take a look at sem example.c. Please make sure that everything in the program sem example.c pertaining to semaphores is understood before compiling it. Run the program, and note the order in which the buffer is read/written to. Run the program multiple times, and again not the order in which the buffer is read/written to.
(a) Do they look different? Why do you think that is the case?

They look different b/c while locking is being done on the semaphore this does not change the fact that scheduling can change the order.