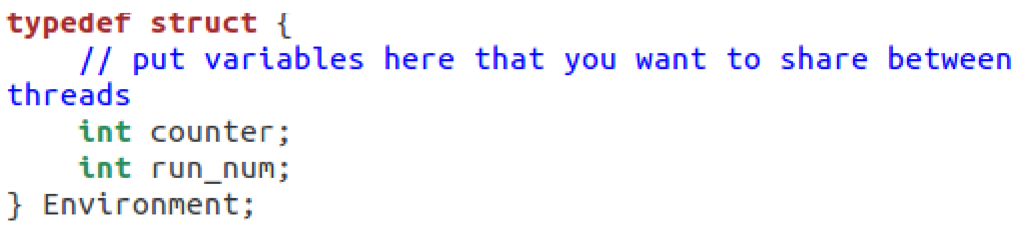
SoftSys Homework 7

Orion Taylor, Joe Gibson
9/23/2012
Olin College of Engineering

Experiment 1

The purpose of this experiment is to measure how three different implementations of a mutex affect the problem of thread racing. Thread racing is a phenomenon that can occur when multiple threads, that are running at the same time, have to read and write from a resource that is shared among them. Essentially, what happens is that a thread can be preempted by the scheduler while it is in the midst of a read/write operation. If another thread does a read/write operation while the first thread is still blocked, erroneous data may ultimately be written into the shared memory location. This is called thread racing. Mutexes (mutual exclusion locks) are a solution to thread racing. Mutexes allow a thread to lock a shared data resource, barring all other threads reading or writing to it, until the first thread has unlocked it.

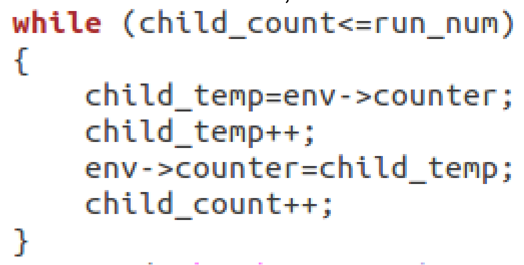
To measure the effectiveness of a mutex implementation, a scenario in which thread racing is pervasive must be created. In this program, two threads are spawned, each sharing a resource, the struct environment.



```
typedef struct {
    // put variables here that you want to share between
threads
    int counter;
    int run_num;
} Environment;
```

Fig 1: This specific implementation of environment does not have locks.

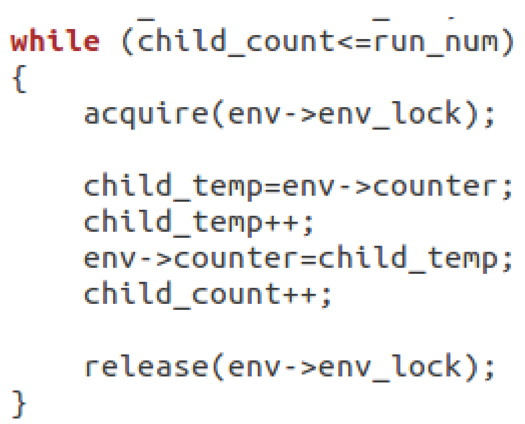
The control does not have any implementation of mutex. In this, each thread would iterate a fixed number of times, that number being determined by command line input. Each iteration, the child thread would read the value of counter, store it in a temporary variable, increment the variable, then store the result back into counter.



```
while (child_count<=run_num)
{
    child_temp=env->counter;
    child_temp++;
    env->counter=child_temp;
    child_count++;
}
```

Fig 2: The loop inside the thread first reads counter, storing into child_temp. It then increments child_temp. Finally, it writes child_temp back onto counter.

The variable group is written in a very similar manner as the control. For each test of an implementation of mutex, each thread will iterate a fixed number of times, that number being determined by command line input. Each iteration, the child thread will lock the shared resource, read the value of counter, store it in a temporary variable, increment the variable, store the result back into counter, then unlock the shared resource.



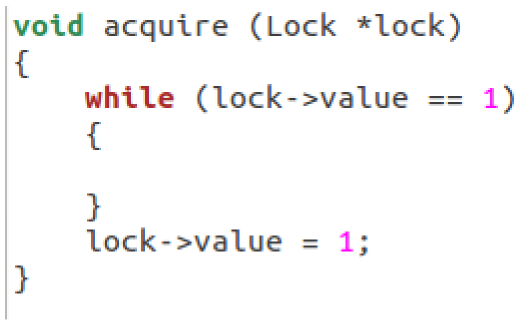
```
while (child_count<=run_num)
{
    acquire(env->env_lock);

    child_temp=env->counter;
    child_temp++;
    env->counter=child_temp;
    child_count++;

    release(env->env_lock);
}
```

Fig 3: The primary difference here is that the shared resource is being locked before the read/write occurs, and unlocked afterwards.

The first implementation of mutex being tested is not so great. In this implementation, an int stored in the mutex is used to determine if the mutex is locked or unlocked. 1 corresponds to a locked mutex, while 0 corresponds to an unlocked mutex. To lock this mutex, the thread will engage in busy waiting, until it reads a value of 0 from the mutex, writing a value of 1 immediately thereafter. To unlock the mutex, the thread simply writes a value of 0 into the mutex.

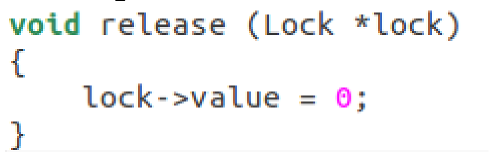


```
void acquire (Lock *lock)
{
    while (lock->value == 1)
    {

    }
    lock->value = 1;
}
```

Fig 4: The implementation of acquiring a lock for this not so great mutex.

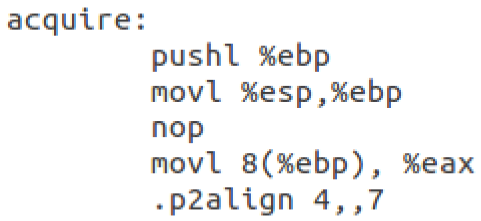
This implementation of mutex has multiple faults. First, it engages in busy waiting when trying to acquire a lock. This is very inefficient, since it means that the processor is wasting resources keeping this thread awake while it is waiting for the mutex to unlock. Second, the read and write operations to the value that determines that state of the mutex are not atomic. This means that if the thread is preempted immediately after it reads a value of 0 in the mutex, but before it can write a value of 1, it is possible for another thread to swoop in and read a value of 0 in the mutex. Both threads would then break out of the while loop, write a value of 1 to the mutex, thinking that they have acquired the lock. This means that this implementation of mutex is itself affected by the racing problem. Finally, there is nothing preventing a thread from unlocking a mutex that it does not hold the lock for, which is very very very bad.



```
void release (Lock *lock)
{
    lock->value = 0;
}
```

Fig 5: The implementation of releasing a lock for this not so great mutex.

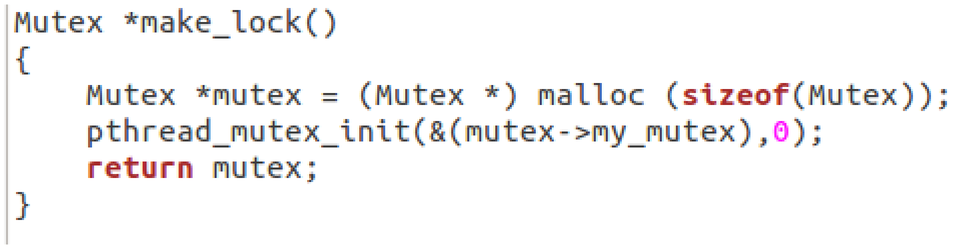
The second implementation of mutex being tested is supposedly a better implementation than the first. I cannot really comment on its implementation because it is written in assembly. I do not know assembly yet.



```
acquire:
        pushl %ebp
        movl %esp,%ebp
        nop
        movl 8(%ebp), %eax
        .p2align 4,,7
```

Fig 6: The implementation of acquiring a lock in the second implementation of mutex. It is written in assembly, which I do not understand.

The final implementation of mutex uses the implementation of mutex found in the pthreads library. This implementation has been designed by programmers greater than I. It has been tried and tested. Thus, it should avert the problem of thread racing.



```
Mutex *make_lock()
{
    Mutex *mutex = (Mutex *) malloc (sizeof(Mutex));
    pthread_mutex_init(&(mutex->my_mutex),0);
    return mutex;
}
```

Fig 7: The third and final implementation of mutex is really just the implementation of mutex found in the pthread library, wrapped in a thin, crisp, shell of code.

The predicted results for this experiments are as follows:

1. The control has no implementation of mutex, and therefore will probably have the most amount of thread interleaving in the wrong places. This will result in thread racing, meaning the counter will not be at a very high value when all is said and done.

2. The first implementation of mutex provides a barrier for thread racing, but does not completely solve the problem because it doesn't use any sort of atomic read/write operation. Thus, it will have a higher value of counter, but it still will not be very high.

3. The second implementation of mutex supposedly solves the thread racing problem, or at least is a better barrier than the first. Thus, the value for counter ought be at its maximum (two times the user input), or at least very close to that.

4. The third implementation of mutex had better solve the thread racing problem, or all linux users would be doomed. I expect that counter will be at its maximum every time.

Now its time to look at some data that we've collected. It should be noted that each data point represents the average of the measurements.

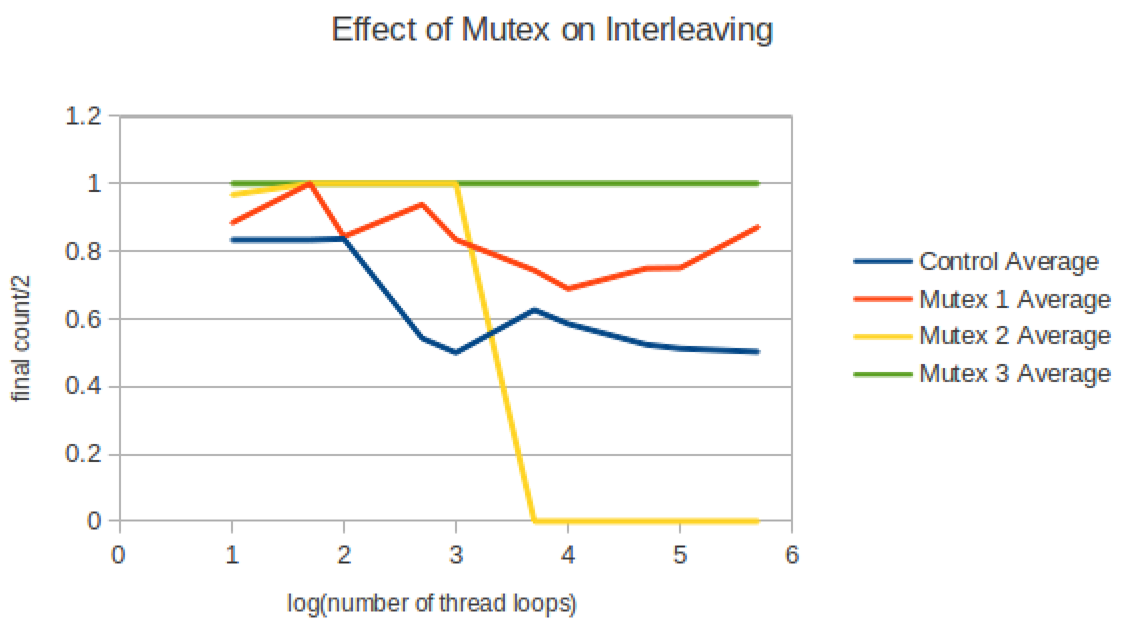


Fig 8: these results were just what I expected!

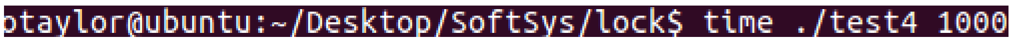
The numbers seen do not necessarily correspond to the number of times threads interleave. If one thread is able to loop multiple times while another thread is preempted, it is possible that when the blocked thread resumes, it will negate all the times that the first thread looped.

These results are consistent with my predictions, thus, I will carry the conclusion I drew from my predictions forward. The one inconsistency is that the second implementation of mutex will occasionally freeze for low values of user input, and will often/always freeze for high values. I suspect that is because this implementation of mutex, while solving the thread racing problem, introduces the possibility of thread deadlocking. In thread deadlocking, each thread is waiting for the other to unlock the mutex, leading to a perpetual cycle of waiting. This halts the program, meaning that counter is never incremented. Oh well.

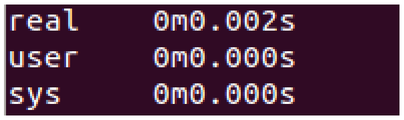
In summary, synchronization errors occur the most frequently without any use of mutex. They never occur when the pthreads implementation of mutex is used. The first implementation of mutex helps solve synchronization problems a little, and the second one changes the synchronization problem from thread racing to thread deadlocking.

Experiment 2:

The purpose of this experiment is to determine which of the three implementations of mutex is the most time efficient/fastest. To do this, the same exact code was used as before. The only difference is that each run of the program has now been timed. I don't really feel like collecting a lot more data, so I'm just going to set the user input to be 1,000 (short enough that it might be possible for implementation two of mutex not to result in deadlock).


```
otaylor@ubuntu:~/Desktop/SoftSys/lock$ time ./test4 1000
```
Fig 9: The only difference is that I am using time when I run the test

It turns out that when I ask the child threads to iterate 1000 times, I get the same exact outputs, regardless of which test I run. Each implementation takes .002s of real time, 0s of computer resources, and 0s of sys resources.



```
real    0m0.002s
user    0m0.000s
sys     0m0.000s
```

Fig 10: This result again, useless!

Clearly, I am the child threads are not iterating enough for any difference in efficiency to become apparent. Thus, I am going to up the number of iterations to 10^6. At this amount, implementation two of mutex will almost certainly deadlock, so data on it cannot be collected.

The results (after one run) were as follows:

Control (no mutex):
real: .013s
user: .020s
sys: .000s

Implementation 1 (spinlock):
real: .125s
user: .244s
sys: .000s

Implementation 2:
deadlock
deadlock
deadlock

Implementation 3 (pthreads version):
real: .151s
user: .156s
sys: .136s

As it can be seen, not using any version of mutexes is very very fast, and consumes very few resources. On the other hand, it will lead to very erroneous results. The first implementation of mutex takes less real time than the pthreads version, but it consumes more computational resources. The second implementation just deadlocks. Finally, the pthreads implementation is the only one that consumes system resources. I suspect that the reason that the pthreads implementation consumes fewer computational resources but takes more time is that it uses thread sleep to make threads wait for a mutex to unlock, as opposed to the busy waiting used by the rudimentary spinlock. Busy waiting is computationally inefficient. On the other hand, it means that a given thread is more likely to be awake when the mutex unlocks. I suspect that the reason that the pthreads implementation uses system resources is because it probably results in far more context switches than the other implementations. If I'm not mistaken, switching between threads uses up mostly system resources.