Project 1: Multithreaded Programming and Synchronization

Carolina Karthik, Christian Canizares

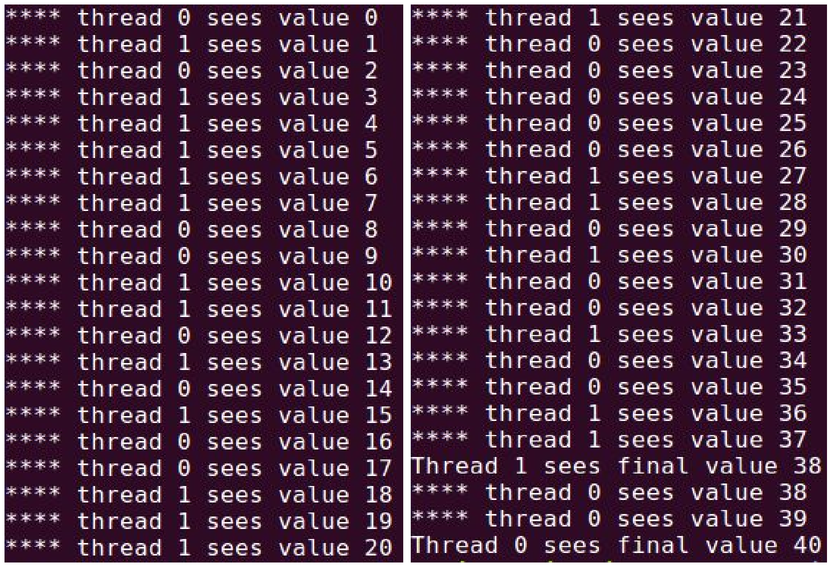
This program tests what happens when multiple Pthreads are created and then perform both synchronized and unsynchronized access to shared data. Each thread runs a function that increments a shared variable, *SharedVariable*, and prints out the value that each thread sees.

Unsynchronized Threads

```
void SimpleThread(int which){

    int num, val;

    for(num = 0; num < 20; num++){
        if(random() > RAND_MAX / 2)
        usleep(500);
        val = SharedVariable;
        printf("**** thread %d sees value %d\n", which, val);
        SharedVariable = val + 1;
    }
    val = SharedVariable;
    printf("Thread %d sees final value %d\n", which, val);

    pthread_exit(0);
}
```

Each thread runs this *SimpleThread* function. The threads iterate through a for-loop 20 times, each time incrementing the value of *SharedVariable* and printing what it sees. When the for-loop returns the thread prints the final value it sees. Finally, each thread exits.

Creating two threads and passing them through the function concurrently yields the following results:



```
**** thread 0 sees value 0
**** thread 1 sees value 1
**** thread 0 sees value 2
**** thread 1 sees value 3
**** thread 1 sees value 4
**** thread 1 sees value 5
**** thread 1 sees value 6
**** thread 1 sees value 7
**** thread 0 sees value 8
**** thread 0 sees value 9
**** thread 1 sees value 10
**** thread 1 sees value 11
**** thread 0 sees value 12
**** thread 1 sees value 13
**** thread 0 sees value 14
**** thread 1 sees value 15
**** thread 0 sees value 16
**** thread 0 sees value 17
**** thread 1 sees value 18
**** thread 1 sees value 19
**** thread 1 sees value 20
```

```
**** thread 1 sees value 21
**** thread 0 sees value 22
**** thread 0 sees value 23
**** thread 0 sees value 24
**** thread 0 sees value 25
**** thread 0 sees value 26
**** thread 1 sees value 27
**** thread 1 sees value 28
**** thread 0 sees value 29
**** thread 1 sees value 30
**** thread 0 sees value 31
**** thread 0 sees value 32
**** thread 1 sees value 33
**** thread 0 sees value 34
**** thread 0 sees value 35
**** thread 1 sees value 36
**** thread 1 sees value 37
Thread 1 sees final value 38
**** thread 0 sees value 38
**** thread 0 sees value 39
Thread 0 sees final value 40
```

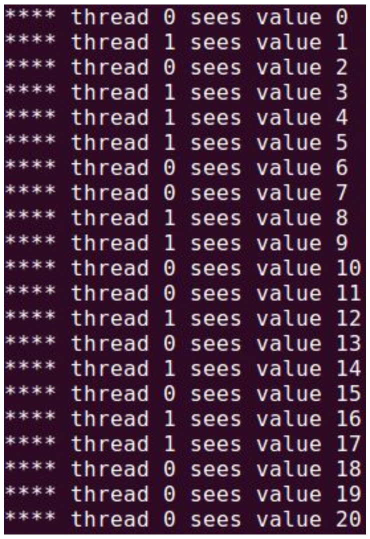
Since the threads aren't synchronized, each thread sees a different final value. Each thread accesses the *SharedVariable* without being blocked so each thread can increment it without caring for the other thread to finish.

Synchronized Threads

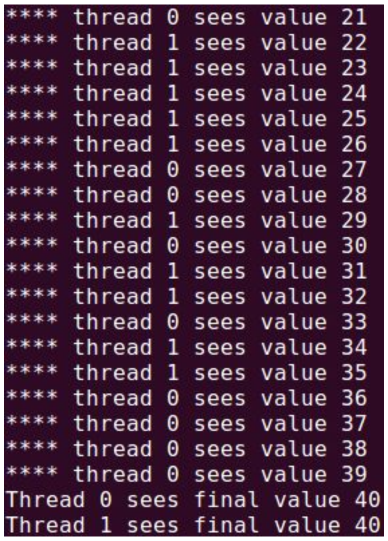
```
void SimpleThread(int which){

    int num, val;

    for(num = 0; num < 20; num++){
        if(random() > RAND_MAX / 2)
        usleep(500);
        val = SharedVariable;
        printf("**** thread %d sees value %d\n", which, val);
        pthread_mutex_lock (&mutexsum);
        SharedVariable = val + 1;
        pthread_mutex_unlock (&mutexsum);
    }
    #ifdef PTHREAD_SYNC
    int rc = pthread_barrier_wait(&barr);
    if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD)
    {
        printf("Could not wait on barrier\n");
        exit(-1);
    }
    #endif
    val = SharedVariable;
    printf("Thread %d sees final value %d\n", which, val);
    pthread_exit(0);
}
```

Like in the unsynchronized threads version of *SimpleThread,* the threads iterate 20 times through a for-loop, each time incrementing and reading the value of *SharedVariable*. However, we added a mutex lock before we increment so that no other thread can access *SharedVariable.* We also created a barrier so that the threads wait for the last thread to exit the loop.

Creating two threads and passing them through *SimpleThread* yields the following results:



```
**** thread 0 sees value 0
**** thread 1 sees value 1
**** thread 0 sees value 2
**** thread 1 sees value 3
**** thread 1 sees value 4
**** thread 1 sees value 5
**** thread 0 sees value 6
**** thread 0 sees value 7
**** thread 1 sees value 8
**** thread 1 sees value 9
**** thread 0 sees value 10
**** thread 0 sees value 11
**** thread 1 sees value 12
**** thread 0 sees value 13
**** thread 1 sees value 14
**** thread 0 sees value 15
**** thread 1 sees value 16
**** thread 1 sees value 17
**** thread 0 sees value 18
**** thread 0 sees value 19
**** thread 0 sees value 20
```



```
**** thread 0 sees value 21
**** thread 1 sees value 22
**** thread 1 sees value 23
**** thread 1 sees value 24
**** thread 1 sees value 25
**** thread 1 sees value 26
**** thread 0 sees value 27
**** thread 0 sees value 28
**** thread 1 sees value 29
**** thread 0 sees value 30
**** thread 1 sees value 31
**** thread 1 sees value 32
**** thread 0 sees value 33
**** thread 1 sees value 34
**** thread 1 sees value 35
**** thread 0 sees value 36
**** thread 0 sees value 37
**** thread 0 sees value 38
**** thread 0 sees value 39
Thread 0 sees final value 40
Thread 1 sees final value 40
```

Since the threads are synchronized, each thread sees the same final value. The final value that each thread sees is also printed at the end because the barrier makes each thread wait until all the others end. The synchronization resolves the race condition caused by two threads accessing the same data concurrently like in the unsynchronized version of *SimpleThread*.