Mini Project Report

Christopher Vargas – 007412039

Jacky Li – 014628729

Daman Patel – 029346666

CECS 526

Hailu Xu

YouTube Link: https://youtu.be/uq5qFREFTJg

The purpose of this assignment is to understand how to synchronize a multi-threaded program using POSIX threads. There are two versions to the program: one without synchronization and one with synchronization using mutex locks. The former shows how threads with no synchronization, can create a race condition inside our critical section, and change the shared variable at the same time another process is changing or using it; it is called a race condition because we do not know the order of the threads that will access the data. This leads to inconsistent values for the shared variable, which are shown in our screenshots. The latter shows how mutex locks are used to synchronize the critical section in order to ensure that only one thread has access to the shared variable, while the other threads wait. The barrier is added after unlocking the critical section so each thread waits for all threads to finish executing the critical section. Now all threads see the same final value for a consistent output shown in our screenshot.

Description and flow of execution without synchronization.

1. **Validation** – the program validates the command line inputs to make sure it is a number and not an arbitrary string. First, we check that the number of inputs is 2. Then, the *ValidateNumber* function takes the input and iterates through each char in *num[]* to make sure the ascii value is equal to the value of a digit 0-9.

```
int ValidateNumber(char num[]) {
	// Checks the char array if each char is a digit between 0 to 9
	// Returns 1 if true; 0 if false
	int i;
	for(i = 0; i < strlen(num); i++) {
	    if (num[i] < 48 || num[i] > 57)
			return 0;
	}
	return 1;
}
...
int main (int argc, char *argv[]) {
	// Validate Input
	if(argc != 2) {
		printf("Invalid number of arguments.\n");
		return -1;
```

```
    }
    if(!ValidateNumber(argv[1])) {
          printf("Not a valid positive integer.\n");
          return -1;
    }
    // Get the int version of char array using atoi
    int num_threads = atoi(argv[1]);
    if (num_threads <= 0) {
          printf("Not a positive integer or value is greater than 2^31.\n");
          return -1;
    }
```

2. **Creating the threads** – Next, we create the number of threads in main based on the input. We use an array of *pthread_t threads*. The return code is the exception for anything that goes wrong while thread execution is taking place. In the for loop, we call *pthread_create* and pass the address of *thread[i]*, the function *SimpleThread*, and *i* (which is a simplified representation of a thread id to distinguish the individual threads in *SimpleThread*).

```
    // Create num_threads of threads
    pthread_t threads[num_threads];
    // return code is used to store error code.
    int return_code;
    long i;

    for (i = 0; i < num_threads; i++) {
        printf("creating thread %ld\n", i);
        return_code = pthread_create(&threads[i], NULL, SimpleThread, (void*
)i);
        // If there is an error, we exit the program
        if (return_code) {
        printf("ERROR: return code from pthread_create() is %d\n",
return_code);
                exit(-1);
          }
     }
```

3. **SimpleThread Function** – We modified the function to be able to work with threads; other than that, it is the same as the handout. We added a generic pointer type for the parameter.

```
void *SimpleThread(void *args) {
	// Thread ID is passed as arguments.
	int num,val;
	// Thread number.
	long which = (long)args;

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

Test Cases and Results

1 thread – with just one thread it runs by itself with no fear of other threads entering the critical section. The final value should be 20 because when the thread finishes its loop, there is no other thread modifying SharedValue. We have included invalid inputs as well.

```
christopher@christopher-VirtualBox:~/Desktop$ ./a abc123
Not a valid positive integer.
christopher@christopher-VirtualBox:~/Desktop$ ./a 123abc
Not a valid positive integer.
christopher@christopher-VirtualBox:~/Desktop$ ./a string
Not a valid positive integer.
christopher@christopher-VirtualBox:~/Desktop$ ./a 1
creating thread 0
*** thread 0 sees value 0
*** thread 0 sees value 1
*** thread 0 sees value 2
*** thread 0 sees value 3
*** thread 0 sees value 4
*** thread 0 sees value 5
*** thread 0 sees value 6
*** thread 0 sees value 7
*** thread 0 sees value 8
*** thread 0 sees value 9
*** thread 0 sees value 10
*** thread 0 sees value 11
*** thread 0 sees value 12
*** thread 0 sees value 13
*** thread 0 sees value 14
*** thread 0 sees value 15
*** thread 0 sees value 16
*** thread 0 sees value 17
*** thread 0 sees value 18
*** thread 0 sees value 19
Thread 0 sees final value 20
```

3 threads – without synchronization, the 3 threads will enter the critical section and change SharedVariable while other threads are possibly using SharedValue at the same time. This results with an inconsistent output. All three threads see a different final value because in this version of the code there is no thread barrier preventing the threads from finishing before all threads have a chance to finish; that's why thread 0 sees a final value

of 37, thread 1 sees a final value of 40, and thread 2 sees a final value of 43. The expected final value is 60.

```
christopher@christopher-VirtualBox:~/Desktop$ ./a 3
creating thread 0
creating thread 1
creating thread 2
*** thread 1 sees value 0
*** thread 0 sees value 1
*** thread 1 sees value 2
*** thread 1 sees value 3
```

...

```
*** thread 2 sees value 32
*** thread 2 sees value 33
*** thread 2 sees value 34
*** thread 0 sees value 35
*** thread 2 sees value 36
Thread 2 sees final value 37
*** thread 1 sees value 36
*** thread 1 sees value 37
*** thread 1 sees value 38
*** thread 1 sees value 39
Thread 1 sees final value 40
*** thread 0 sees value 40
*** thread 0 sees value 41
*** thread 0 sees value 42
Thread 0 sees final value 43
```

Description and flow with synchronization.

1. **Validation** – validation is the same as the previous code.

2. **Creating the threads with Synchronization** – first we must declare our mutex and barrier objects.

```
// Mutex lock
pthread_mutex_t mutex;
// Barrier is used to stop thread's last execution.
pthread_barrier_t barrier;
```

In main, we initialize our barrier just like we initialize our pthread object. Then the SimpleThread functions run with each individual thread.

```
	// return code is used to store error code.
	int return_code;
	long i;

	 // If any error pops up when using barrier1.
	 return_code = pthread_barrier_init(&barrier, NULL, num_threads);

	 // strerror is used to check string error.
	 if (return_code) {
		fprintf(stderr, "pthread_barrier_init: %s\n",
strerror(return_code));
		return -1;
	 }

	 for (i = 0; i < num_threads; i++) {
		printf("creating thread %ld\n", i);
		return_code = pthread_create(&threads[i], NULL, SimpleThread,
	(void*)i);
	// If there is an error we exit the program
		if (return_code) {
		printf("ERROR: return code from pthread_create() is
%d\n",return_code);
			return -1;
		 }
	 }
```

After all threads have completed the SimpleThread function we join all threads to synchronize all outputs at the end, after that we terminate all threads and destroy the mutex object.

```
	// Synchronize and join all threads.
	// Terminate threads and destroy mutex object
	for (i = 0; i < num_threads; i++) {
		pthread_join(threads[i], NULL);
	}
	pthread_mutex_destroy(&mutex);
	return 0;
```

3. **SimpleThread Function with Mutex Locks and Barrier** – It is very similar to the previous function except for the addition to the mutex locks around the critical section. *#ifdef* works if all the threads are working at the same time, we pick a random thread and suspend it for 500ms. Before using SharedVariable we must lock our mutex; this ensures that no other thread will use that variable at the same time. Once our thread is finished, it unlocks the mutex. Barrier is used so that a thread will wait for all other threads to finish so that all threads see the same value for the final value.

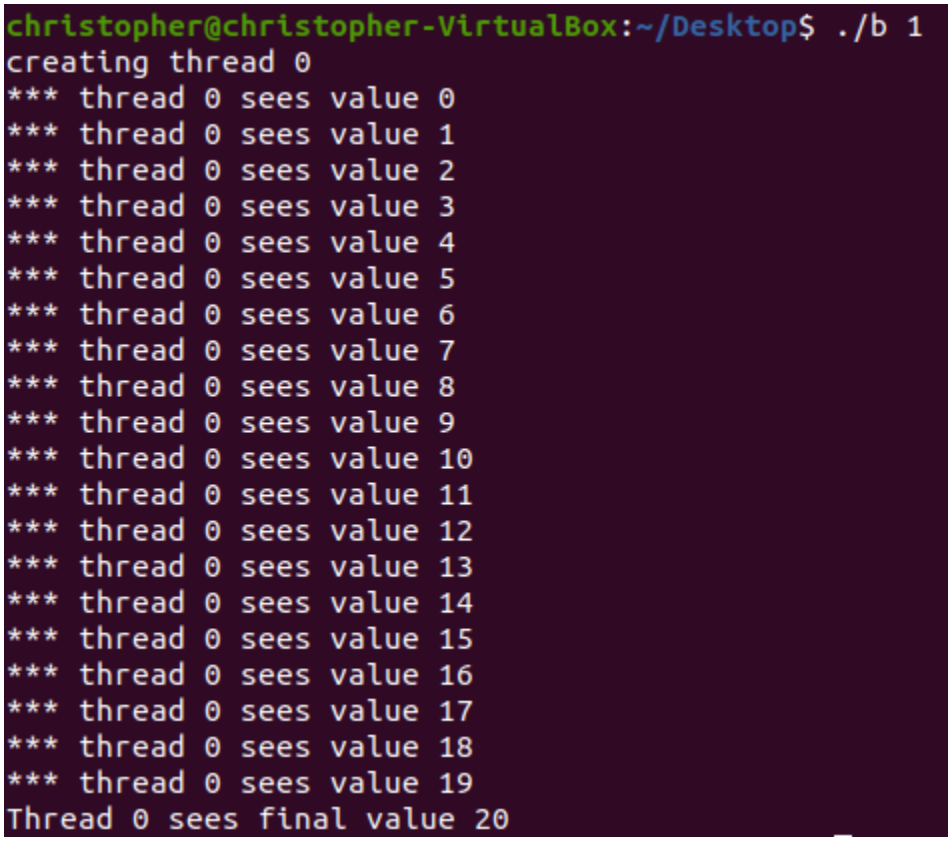
```
void *SimpleThread(void *args) {
	// Thread ID is passed as argument.
	int num,val;
	// Thread number.
	long which = (long)args;

	for(num = 0; num < 20; num++) {
	#ifdef PTHREAD_SYNC
		if(random() > RAND_MAX / 2)
		usleep(500);
	#endif
	// Entering the critical section and SharedVariable is now locked.
	pthread_mutex_lock(&mutex);
	val = SharedVariable;
	printf("*** thread %ld sees value %d\n", which, val);
	SharedVariable = val + 1;
	//Exiting the critical section and SharedVariable is now unlocked.
	pthread_mutex_unlock(&mutex);
	}

	// Prevent thread's last execution so thread sees final
	ShavedVariable value
	pthread_barrier_wait(&barrier);
	val = SharedVariable;
	// Thread's last execution.
	printf("Thread %ld sees final value %d\n", which, val);
}
```

Test Cases and Results

1 thread – with one thread the output is the same as the output of the non-synchronized program with 1 thread.



200 threads – this is an edge case to show that we can synchronize up to 200 threads and get the expected output for our SharedVariable of 4000 (the output was very large we cropped some of it to only show relevant information)

```
*** thread 1 sees value 0
*** thread 1 sees value 1
*** thread 1 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 1 sees value 8
*** thread 1 sees value 9
*** thread 1 sees value 10
*** thread 1 sees value 11
*** thread 1 sees value 12
*** thread 1 sees value 13
*** thread 1 sees value 14
*** thread 1 sees value 15
*** thread 1 sees value 16
*** thread 1 sees value 17
*** thread 1 sees value 18
*** thread 1 sees value 19
*** thread 3 sees value 20
*** thread 3 sees value 21
```

…

```
Thread 187 sees final value 4000
Thread 189 sees final value 4000
Thread 190 sees final value 4000
Thread 191 sees final value 4000
Thread 188 sees final value 4000
Thread 192 sees final value 4000
Thread 193 sees final value 4000
Thread 194 sees final value 4000
Thread 195 sees final value 4000
Thread 196 sees final value 4000
Thread 197 sees final value 4000
Thread 198 sees final value 4000
Thread 199 sees final value 4000
Thread 173 sees final value 4000
```

Summary of Contribution

While we had individual teammates lead certain aspects of the project, we each equally shared and contributed our work for the completion of this project. As for the coding portion of the project, we each took the lead on different parts of the project. Additionally, we all collaborated on each part of the code to ensure the entire project compiles and runs as intended through code reviewing and debugging.

Breakdown of Contribution:

- **Report**: [Lead] Christopher Vargas; Jacky Li; Daman Patel
- **README.md**: [Lead] Daman Patel; Jacky Li; Christopher Vargas
- **Makefile**: [Co-Lead] (Daman Patel, Jacky Li); Christopher Vargas
- **Video**: [Lead] Jacky Li; Christopher Vargas; Daman Patel
- **mini_project.c**
    - **ValidateNumber**: [Lead] Christopher Vargas; Jacky Li; Daman Patel
    - **Multithreading**: [Lead] Jacky Li; Daman Patel; Christopher Vargas
    - **Synchronization**: [Lead] Daman Patel; Jacky Li; Christopher Vargas