A0: Uthreads!

Himanshu Rathi (hr393)   Shashwenth Muralidharan (sm2785)

1 Introduction

This document is a comprehensive report of our implementation of a user-level thread library that mimics the working of the Pthread library. The POSIX thread library provides a set of interfaces for the user to implement thread programming. Our user-level thread library provides various methods for threading, multi-threading, synchronization, and scheduling operations.

In this report we will extensively detail the methods and API implemented, the data structures used and the performance comparison between our library and POSIX library.

2 Methods

This section will provide all the necessary information regarding the functions we implemented in this library.

For simplicity, we have split this section into the following subsections based on their usage.

2.1 For Thread Execution

This section elucidates all the functions implemented for the purpose of thread execution, this includes:

1. mypthread_create()
2. mypthread_join()
3. mypthread_yield()
4. mypthread_exit()

2.1.1 mypthread_create()

The mypthread_create() function allows the user to create new threads.During the creation of new threads, a new thread control block will be initialized for each of the threads being created. The parameters in the thread control block are updated based on the information of the thread. These parameters include the thread ID, that is unique to each threads, current status of the thread, the thread's context, etc. Also, these threads are added to a global list that stores the thread control block of all the new threads to keep track of them. During the first mypthread_create call a scheduler thread and main thread will also be created each of them representing the scheduler and the main block of the program respectively. Additionally, we have also created a exit context that is pointed by all the threads in their UC link context variable. This exit context is used to return to the pthread_exit() function once the thread execution is completed.

2.1.2 mypthread_join()

The main purpose of the mypthread_join() function is to check whether a thread has completed its execution. If the thread is terminated it will return the pointer to the return values of the joining thread. If the thread is still in execution, the join function will wait for the thread to get terminated by calling the scheduler function by a context switch. The current thread which is executing currently will be blocked during a join call and will be added back to the running queue once the joining thread finishes its execution.

2.1.3 mypthread_yield()

The mypthread_yield() function is used when the user wants a thread to voluntarily relinquish its run-time so that other threads can execute in the meantime. This is done by switching the context of the yielding thread to the context of the scheduler. The status of the thread control block is updated accordingly for the yielding thread and is pushed in the running queue.

2.1.4 mypthread_exit()

The primary purpose of this function is to terminate the calling thread. . Here the status of the thread is changed to TERMINATED and it is added to the finished queue. The termination of the thread refers to the deallocation of the calling thread's stack and any memory associated with the calling thread.

2.2 For Mutex Operations

This section elucidates all the functions implemented for the mutex operations, this includes:

1. mypthread_mutex_init()
2. mypthread_mutex_lock()
3. mypthread_mutex_unlock()
4. mypthread_mutex_destroy()

2.2.1 mypthread_mutex_init()

This initialization function allocates the mutex struct to the mutex passed to this function. Here the mutex is set to unlock ('0' in our case). Also, the lockholder variable is used to represent the thread that is currently holding the mutex lock and accessing the critical section. The waiting queue variable is used to keep track of the threads that tried to access the mutex when it was held by a different owner.

2.2.2 mypthread_mutex_lock()

The mypthread_mutex_lock() function is used to acquire the lock to the critical section by a thread. The status of the mutex (0 or 1) is checked using a __sync_lock_test_and_set() function. There are two possible outcomes to this function: A)The thread successfully acquires the lock. This situation occurs when the mutex is in an unlocked state when the thread tried to acquire the lock. B)The thread was unsuccessful in acquiring the mutex lock. This situation occurs when the mutex is already held by another thread. During this case, the thread that tried to get the lock will be pushed to the waiting queue and the scheduler's context will be called.

2.2.3 mypthread_mutex_unlock()

This method allows the mutes lock to be released (i.e) set the status of the mutex lock to unlocked (0). Once the lock is released, threads in the waiting queue are transferred to the running queue and the waiting queue is cleared.

2.2.4 mypthread_mutex_destroy()

This function deallocates all the memory associated with the mutex. In simple terms, it destroys the mutex.

2.3 Scheduler Methods

This section elucidates all the functions implemented for the scheduler and the various scheduling algorithms, this includes:

1. schedule()
2. sched_RR()
3. sched_MLFQ()
4. sched_PSJF()

2.3.1 schedule()

The schedule function is used to schedule the threads in the running queue based on the algorithm specified by the user. The scheduler thread and its associated context are used to access the schedule() function. In our implementation we use the following algorithms:
A)Round Robin algorithm
B)Preemptive Shortest Job First algorithm
C)Multilevel Feedback Queue algorithm

2.3.2 sched_RR()

This function is used to implement the round robin scheduling function. In round robin scheduling algorithm each thread is cyclically assigned a fixed runtime slot. This runtime slot is called a time slice or time quantum. It is a preemptive algorithm as it stops the current running thread and transfers the run time to the next thread in the running queue once the time quantum is completed.

In our implementation, we will first check if the current thread is in active state, if not we will execute the next thread. If it is in the running queue, will rotate the running queue to shift the run-time to the next thread and execute it. By rotation we refer to the pop of the front thread in the running queue and pushing it to the end of the running queue. Then we will change the status of the thread in the front to RUNNING and execute it. This process happens until the running queue is empty or the user stops the execution for any other reason.

2.3.3 sched_MLFQ()

In the MLFQ scheduling algorithm there are 'n' number of queues also known as levels, and represent the priority of execution. Initially, all the threads are placed at the highest priority queue (i.e 0) and are moved between queues based on the execution runtime of each thread.

In our implementation, we have 4 levels of queues. At first, the status of the thread is checked and if the status is not active it is removed from the queue. Now, if the ready thread is currently executing at the lowest level it will be appended to the same queue. Otherwise, the running thread will be moved down a level and appended to the new low-priority queue. Finally, all the levels are scheduled sequentially based on their priorities in the RR method. During execution at each levels the time quantum is updated for each levels (i.e TIME QUANTUM = TIME_QUANTUM * Level). The time quantum will be the least for the highest priority and will gradually increase as the priority decreases. Finally, after a particular time quantum we move all the threads to the highest priority level.

2.3.4 sched_PSJF()

The idea behind PSJF is to execute the thread that has the least rum-time of all the threads. In our implementation we have used a quantum variable in the thread's TCB that allows us to keep track of the number of time quantums utilized by a particular thread. During the schedulers run time it will sort the running threads based on their time quantum and it will run the thread with the least quantum usage.

2.4 Signals And Timer Methods

This section elucidates the functions necessary for raising signals, handling signals and other signal related operations. The functions are,

1. boot_timer()
2. sigg_handler()

2.4.1 boot_timer()

The boot_timer() function is used to allocate the time quantum and initialize the timer. Once the timer is initialized the signal is raised and the signal handler is called. Here the setitimer() is used to raise a signal and the sigaction() function's sa_handler() is used to handle the signal. The timer we use in our implementation is the SIGVTALRM which is a virtual timer that sends a signal when a process or thread finishes its current CPU run-time. The boot_timer is used to call the scheduler at regular intervals specified by the user.

2.4.2 sigg_handler()

Once the signal handler function is called after the timer executes, the signal handler function calls this sigg_handler() function. This function is used to switch the context to the context of the scheduler. Furthermore, to make the function mutex compatible, we disable the handler function so that the mutex operation will not be interrupted.

2.5 Queue Methods

This section elucidates all the functions implemented for the Queue implementation, this includes:

1. Initializequeue()
2. push()
3. pop()
4. clearQ()

2.5.1 Initializequeue()

This function is used to initialize a new queue. It allocates the necessary size in the heap and assigns the default parameters like the front index, back index, size of the queue, and a list to store the threads.

2.5.2 push()

The push function is used to add an element to the queue in our case it inserts the thread to the back of the queue.

2.5.3 pop()

The queue data structure follows FIFO so the pop function is used to remove the element from the front of the queue.

2.5.4 clearQ()

The clearQ() function removes all the elements from the queue and reassigns the queue to its default parameters.

3 Data Structures

This section explains the important data structures involved in our implementation

3.1 Thread Control Block

```
typedef struct threadControlBlock
    {
    uint tid; // Unique thread ID
    ucontext_t tcontext; // Thread context
    thread_status tstatus; // Status of the thread
    void* return_value; // Pointer to return value4
    mypthread_t blockedThread;
    int quantums_used; // Number of quantums run by a Thread
    int priority; // The current Queue level of the thread
    } tcb;
```

3.2 Queue Data Structure

```
typedef struct my_queue
    {
    mypthread_t front; // Front of the queue
    mypthread_t back; // Back of the queue
    mypthread_t max_size; // Max size of the queue
    mypthread_t curr_size; // Current size of the queue
    mypthread_t* list; // List to store thread ID's
    }my_queue;
```

3.3 Mutex structure

```
typedef struct mypthread_mutex_t
    {
    int lock; // Ststus of lock
    mypthread_t lockHolder; // Current lock owner thread
    my_queue* waiting; // Waiting queue for the mutex
    } mypthread_mutex_t;
```

3.4 Thread list Array

threadList[MAX_TCBS] // Used to store all the threads that are created based on their thread id. The 0th index is used to store the schedulers context and the 1st index is used to store the main function's context.

3.5 Running Queue

Running_q[QNUM] // It is used to store the threads that are ready for exection. The QNUM is used to represent the level for MLFQ scheduling. Initially the value of QNUM will be 0 during thread creation and insertion.

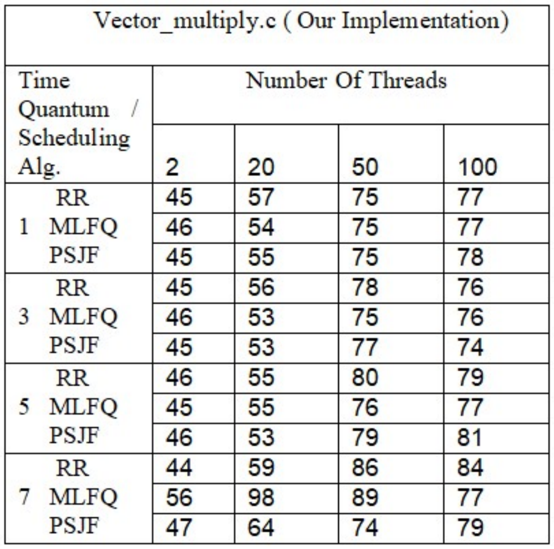
4 Performance

The below tables presents the performance of our user level thread library for each benchmarks. We have also included the performance of the POSIX library for the same benchmarks for comparison based evaluation.

4.1 vector_multiply.c Benchmark Performance

The following tables represent the performance for both the libraries. The first table specifies the time elapsed by our user level thread library for different time quantum and thread count. The second table represents the time elapses for the POSIX library to execute the vector_multiply program.

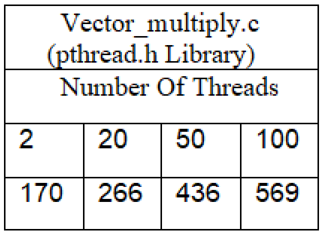
ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



Vector_multiply.c (Our Implementation)

Time Quantum / Scheduling Alg.		Number Of Threads			
		2	20	50	100
1	RR	45	57	75	77
	MLFQ	46	54	75	77
	PSJF	45	55	75	78
3	RR	45	56	78	76
	MLFQ	46	53	75	76
	PSJF	45	53	77	74
5	RR	46	55	80	79
	MLFQ	45	55	76	77
	PSJF	46	53	79	81
7	RR	44	59	86	84
	MLFQ	56	98	89	77
	PSJF	47	64	74	79

Figure 1: Vector_multiply.c performance for mypthread library



Vector_multiply.c (pthread.h Library)

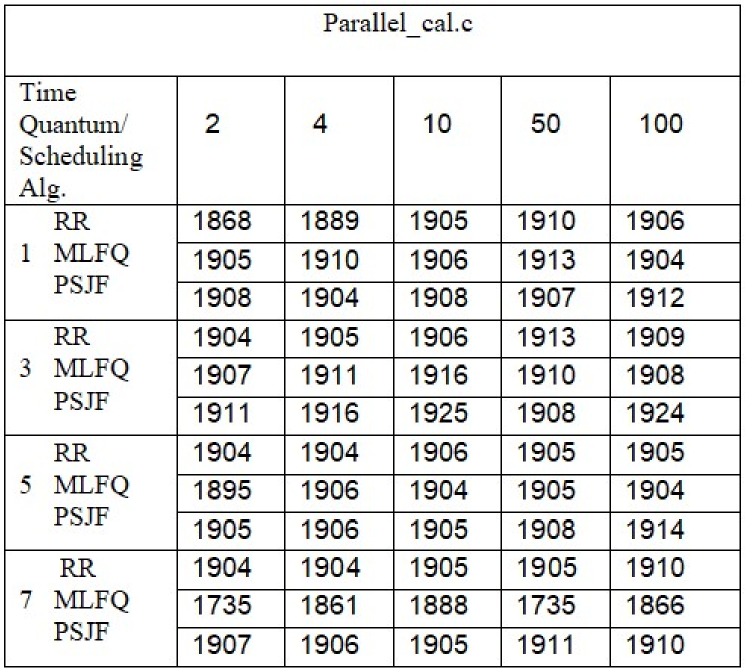
Number Of Threads			
2	20	50	100
170	266	436	569

Figure 2: Vector_multiply.c performance for POSIX library

4.2 parallel_cal.c Benchmark Performance

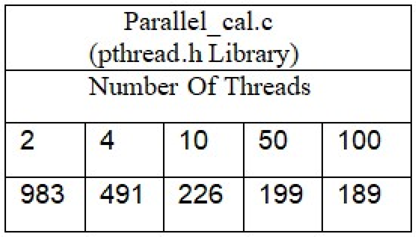
The following tables represent the performance for both the libraries. The first table specifies the time elapsed by our library different parameters. The second table represents the time elapses for the POSIX library to execute the parallel_multiply program.

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



Parallel_cal.c						
Time Quantum/ Scheduling Alg.		2	4	10	50	100
1	RR	1868	1889	1905	1910	1906
	MLFQ	1905	1910	1906	1913	1904
	PSJF	1908	1904	1908	1907	1912
3	RR	1904	1905	1906	1913	1909
	MLFQ	1907	1911	1916	1910	1908
	PSJF	1911	1916	1925	1908	1924
5	RR	1904	1904	1906	1905	1905
	MLFQ	1895	1906	1904	1905	1904
	PSJF	1905	1906	1905	1908	1914
7	RR	1904	1904	1905	1905	1910
	MLFQ	1735	1861	1888	1735	1866
	PSJF	1907	1906	1905	1911	1910

Figure 3: parallel_cal.c performance for mypthread library



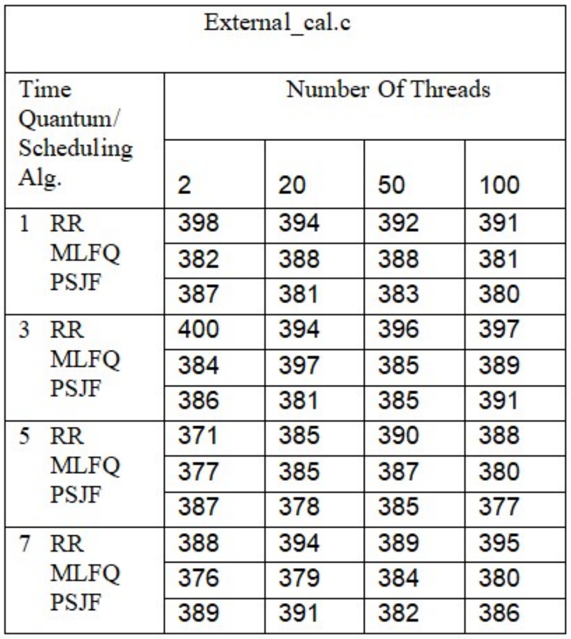
Parallel_cal.c (pthread.h Library)				
Number Of Threads				
2	4	10	50	100
983	491	226	199	189

Figure 4: parallel_cal.c performance for POSIX library

4.3 external_cal.c Benchmark Performance

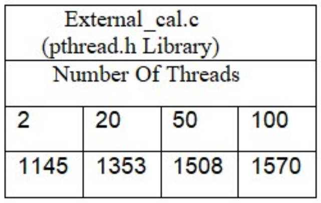
The following tables represent the performance for both the libraries. The first table specifies the time elapsed by our implementation for different time quantum and thread count . The second table represents the time elapses for the POSIX library to execute the external_multiply program.

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



External_cal.c				
Time Quantum/ Scheduling Alg.	Number Of Threads			
	2	20	50	100
1 RR	398	394	392	391
MLFQ	382	388	388	381
PSJF	387	381	383	380
3 RR	400	394	396	397
MLFQ	384	397	385	389
PSJF	386	381	385	391
5 RR	371	385	390	388
MLFQ	377	385	387	380
PSJF	387	378	385	377
7 RR	388	394	389	395
MLFQ	376	379	384	380
PSJF	389	391	382	386

Figure 5: external_cal.c performance for mypthread library



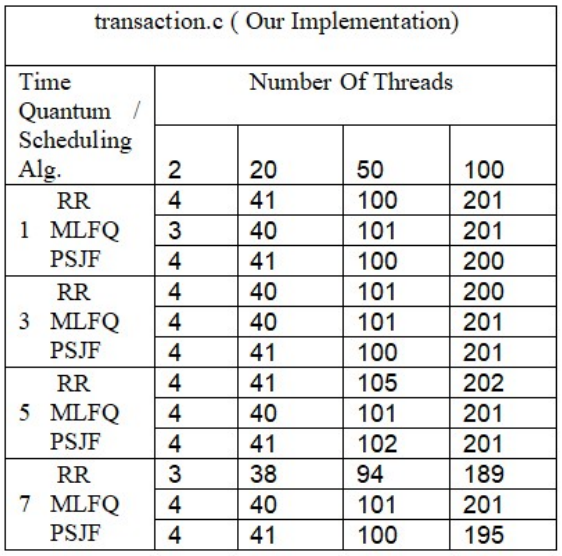
External_cal.c (pthread.h Library)			
Number Of Threads			
2	20	50	100
1145	1353	1508	1570

Figure 6: external_cal.c performance for POSIX library

4.4 Additional Benchmark 1: transaction.c Performance

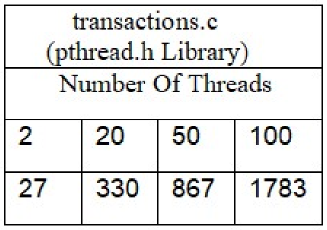
The transaction.c function benchmark is used to evaluate if the algorithm prevents race condition and allows all the threads to synchronize and access the critical section, which is increasing and decreasing the global variable "num". In this benchmark there are two functions that constantly increase and decrease the global variable. The reason we used this benchmark was to evaluate the synchronization methods. The following tables represent the performance for both the libraries. The first table specifies the time elapsed by our implementation for different time quantum and thread count . The second table represents the time elapses for the POSIX library to execute the transaction.c program.

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



transaction.c (Our Implementation)					
Time Quantum / Scheduling Alg.		Number Of Threads			
		2	20	50	100
1	RR	4	41	100	201
	MLFQ	3	40	101	201
	PSJF	4	41	100	200
3	RR	4	40	101	200
	MLFQ	4	40	101	201
	PSJF	4	41	100	201
5	RR	4	41	105	202
	MLFQ	4	40	101	201
	PSJF	4	41	102	201
7	RR	3	38	94	189
	MLFQ	4	40	101	201
	PSJF	4	41	100	195

Figure 7: transaction.c performance for mypthread library



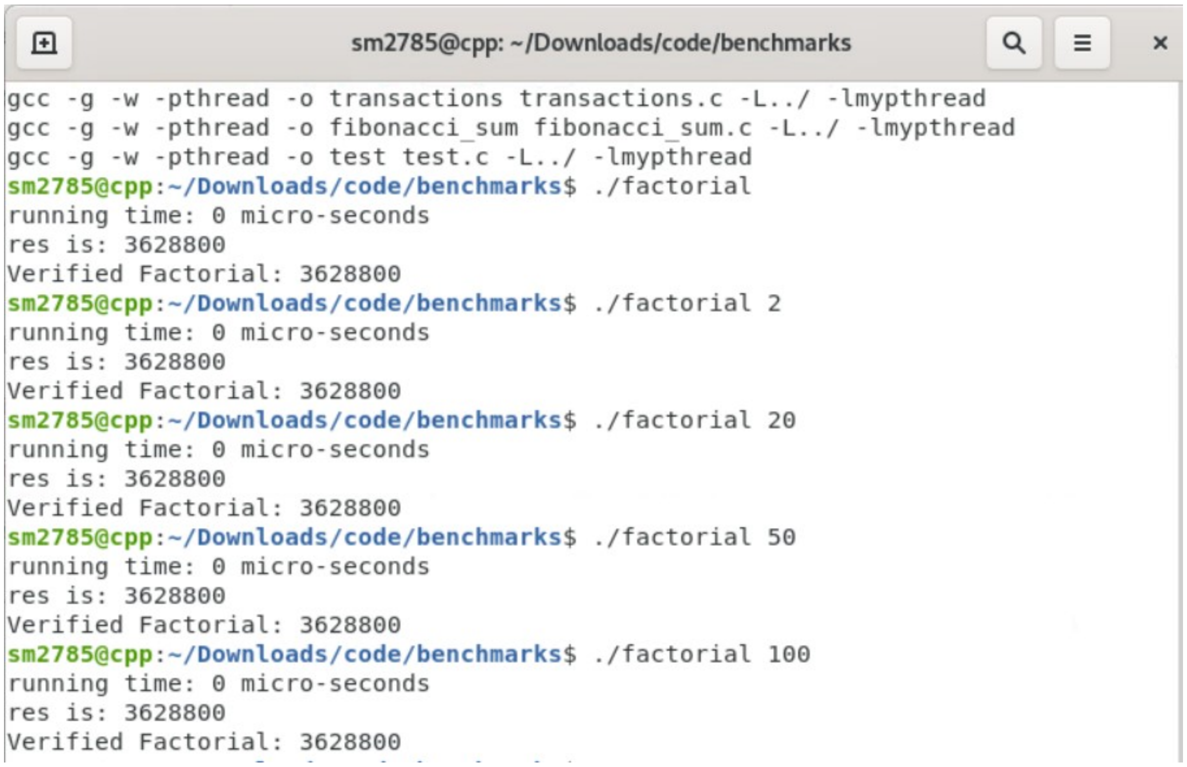
transactions.c (pthread.h Library)			
Number Of Threads			
2	20	50	100
27	330	867	1783

Figure 8: transaction.c performance for POSIX library

4.5 Additional Benchmark 2: factorial.c Performance

The factorial.c benchmark is a special case of benchmark implementation because this program uses multithreading in a interleaved manner. By interleaved we mean that each thread will execute a specific section of the code. For this reason, we have tested it without using mutex and verified the results with normal process execution. The reason this benchmark is successful is because the threads are executing different parts of the function and updating the global variable individually. When this happens there will not be any overlap or race condition between the threads and hence we can infer that our implementation supports interleaved multithreading

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



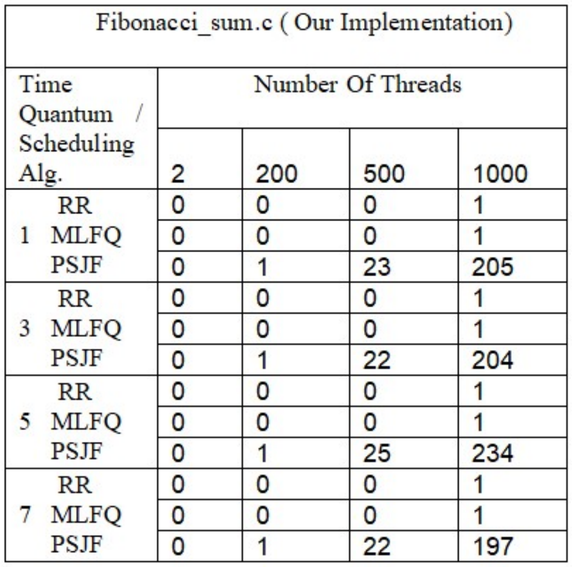
```
gcc -g -w -pthread -o transactions transactions.c -L../ -lmypthread
gcc -g -w -pthread -o fibonacci_sum fibonacci_sum.c -L../ -lmypthread
gcc -g -w -pthread -o test test.c -L../ -lmypthread
sm2785@cpp:~/Downloads/code/benchmarks$ ./factorial
running time: 0 micro-seconds
res is: 3628800
Verified Factorial: 3628800
sm2785@cpp:~/Downloads/code/benchmarks$ ./factorial 2
running time: 0 micro-seconds
res is: 3628800
Verified Factorial: 3628800
sm2785@cpp:~/Downloads/code/benchmarks$ ./factorial 20
running time: 0 micro-seconds
res is: 3628800
Verified Factorial: 3628800
sm2785@cpp:~/Downloads/code/benchmarks$ ./factorial 50
running time: 0 micro-seconds
res is: 3628800
Verified Factorial: 3628800
sm2785@cpp:~/Downloads/code/benchmarks$ ./factorial 100
running time: 0 micro-seconds
res is: 3628800
Verified Factorial: 3628800
```

Figure 9: factorial.c performance for mypthread library

4.6 Additional Benchmark 3: fibonacci_sum.c Performance

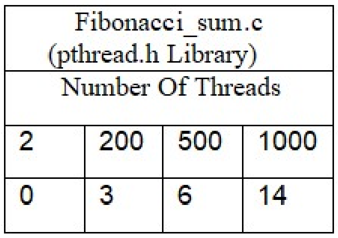
The third benchmark that we are using is fibonacci_sum.c. The output of the program is the sum of the fibonacci series until the specified 'n'. We have initialized n=200 because of the limited stack size (i.e) if we increase the value of n we will get a segmentation fault error due to stack overflow. Now, the reason we have used this particular benchmark was to test our program for recursive calls. As expected the threads have successfully implemented the recursive calls. Also, we have tested for a large number of thread counts because smaller values of number of threads often gives 0 microseconds as time elapsed. We have included the time taken by our library for the execution of this benchmark and have also attached the time taken by the POSIX library.

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



Fibonacci_sum.c (Our Implementation)					
Time Quantum / Scheduling Alg.		Number Of Threads			
		2	200	500	1000
1	RR	0	0	0	1
	MLFQ	0	0	0	1
	PSJF	0	1	23	205
3	RR	0	0	0	1
	MLFQ	0	0	0	1
	PSJF	0	1	22	204
5	RR	0	0	0	1
	MLFQ	0	0	0	1
	PSJF	0	1	25	234
7	RR	0	0	0	1
	MLFQ	0	0	0	1
	PSJF	0	1	22	197

Figure 10: transaction.c performance for mypthread library



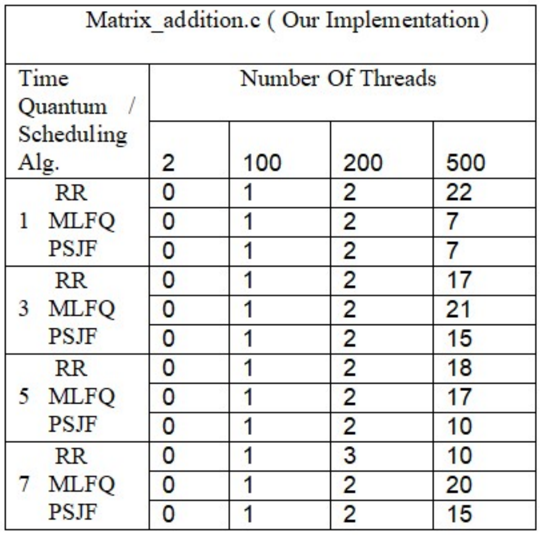
Fibonacci_sum.c (pthread.h Library)			
Number Of Threads			
2	200	500	1000
0	3	6	14

Figure 11: transaction.c performance for POSIX library

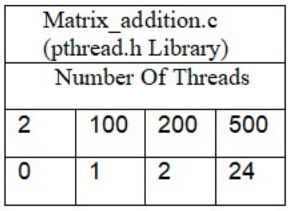
4.7 Additional Benchmark 4: Matrix_addition.c Performance

The 4th benchmark that we are using is the Matix_addition.c . This matrix_addition.c performs the sum of two matrix and storing the result in a third temp matrix. We have included the time taken by our library for the execution of this benchmark and have also attached the time taken by the POSIX library.

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



Matrix_addition.c (Our Implementation)					
Time Quantum / Scheduling Alg.		Number Of Threads			
		2	100	200	500
1	RR	0	1	2	22
	MLFQ	0	1	2	7
	PSJF	0	1	2	7
3	RR	0	1	2	17
	MLFQ	0	1	2	21
	PSJF	0	1	2	15
5	RR	0	1	2	18
	MLFQ	0	1	2	17
	PSJF	0	1	2	10
7	RR	0	1	3	10
	MLFQ	0	1	2	20
	PSJF	0	1	2	15



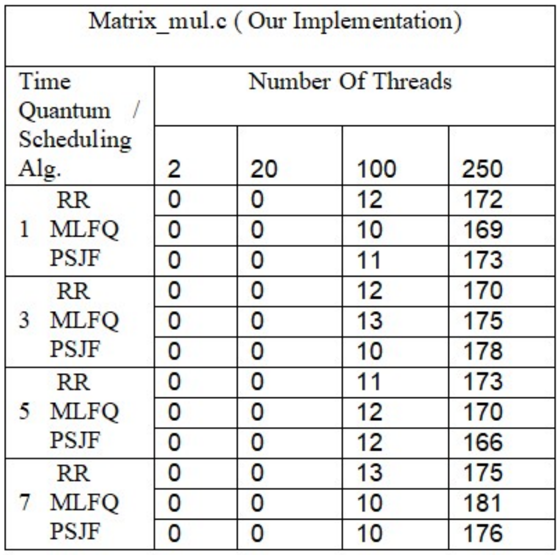
Matrix_addition.c (pthread.h Library)			
Number Of Threads			
2	100	200	500
0	1	2	24

Figure 12: Matrix_addition.c performance for mypthread library

4.8 Additional Benchmark 5: Matrix_mul.c Performance

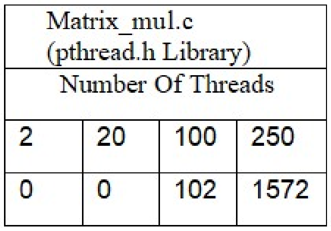
The 5th and the final benchmark that we use in our implementation is the matrix_mul.c. Here we aim at reducing the run-time by assigning each thread to a particular row in the matrix. And when the multiplication function is called by each thread it performs the multiplication of the entire row and adds it to the resultant matrix.. By this we mean that each thread will be performing the multiplication separately. This allows us to evaluate the threads running time during the execution. Also, the number of threads is the length of the rows and colums in the matrix. The output is the sum of the elements in the result matrix. We have included the time taken by our library for the execution of this benchmark and have also attached the time taken by the POSIX library.

ALL THE VALUES REPRESENTED IN THE BELOW TABLES ARE IN MICRO-SECONDS.



Matrix_mul.c (Our Implementation)					
Time Quantum / Scheduling Alg.		Number Of Threads			
		2	20	100	250
1	RR	0	0	12	172
	MLFQ	0	0	10	169
	PSJF	0	0	11	173
3	RR	0	0	12	170
	MLFQ	0	0	13	175
	PSJF	0	0	10	178
5	RR	0	0	11	173
	MLFQ	0	0	12	170
	PSJF	0	0	12	166
7	RR	0	0	13	175
	MLFQ	0	0	10	181
	PSJF	0	0	10	176

Figure 13: Matrix_mul.c performance for mypthread library



Matrix_mul.c (pthread.h Library)			
Number Of Threads			
2	20	100	250
0	0	102	1572

Figure 14: Matrix_mul.c performance for POSIX library

5 Conclusion

Based on the above performance metrics, we could conclude that our mypthread library performs better than the POSIX library for certain benchmarks like the vector_multiply.c, and the external_cal.c. However, the POSIX library performs substantially better for the parallel_cal.c benchmark. Moreover, we have shown the results for 5 other benchmarks and have shown the runtime comparison between the POSIX library and our implementation.