CS416 Project 2: User Thread Library and Scheduler Implementation

Aditya Verma (175007700)
Joshua Siegel (169007779)

March 2, 2019

Abstract

We were required to write a single core user-level thread library and implement 2 different variations of popular scheduler policies. These policies are commonly known as **MLFQ** and **STCF**. The goal was to be able to use this thread library to mimic the pthread library, in which a user may create multiple threads, create and utilize mutexes, and customize the scheduler to follow one of the 2 different paradigms.

1 Introduction

To create this thread library, we started off by designing a First In First Out (FIFO) scheduler. Once that was fully implemented, we abstracted the ideas to MLFQ and STCF paradigms. We decided that a modular approach was best, since the schedulers would have similar core functionality and need to call similar methods.

For our MLFQ, we used a data structure that kept track of 4 levels of FIFO queues. Each level represented a priority from 0 to 3, with 0 being the highest priority and 3 being the lowest.

For our STCF, we kept a timeRan variable in the TCB and added to it everytime we swapped it's context out. We then kept a priority queue to make sure the top of the queue was always the thread that had the lowest timeRan value.

We started our scheduler when the first call to pthread_create was called by creating a queue to store the threads, and by starting a signal timer that would interrupt our program every 10ms. This timer would simulate "trapping" into the OS and allow us to switch contexts to other non-blocked threads. Of course we did not want our scheduler to be interrupted while it was in the middle of doing the logic for context switches, so we created a test and set variable to ignore the signal when it occurs at an inconvenient time. When a thread's status is DONE we remove it from the queue, and when its status is BLOCKED we don't schedule it unless it is available to be unblocked.

Our mutexes were implemented by keeping a variable called isLocked inside the mutex struct, and using the built in __sync_lock_test_and_set() function to test and set the variable whenever needed. We used this method since it is atomic and cannot be interrupted.

We kept track of a threadQueue variable that was responsible for keeping track of the threads in our scheduler, and we implemented many of the common queue functions such as insertion, removal, search, and update. We abstracted these functions to work for all types of schedulers, and inside the functions we decide what to do based on the scheduler type.

To keep track of the scheduler type, we used a macro in our header file called SCHED which mapped to an enumeration of the type of scheduler. We used this macro and enumeration pair in our functions to separate out the logic.

1.0.1 Assumptions

For the main codebase mentioned above, we made the following assumptions:

1. No more than **16384** threads will be created by the user.

2. As discussed with the professor, the main is not a thread to be scheduled. However, the main is returned to when a thread that is being

'joined on' finishes, or there are no threads left to run.

3. All mutex orderings, locks, unlocks are respected by the user. In case of a **DEADLOCK** (all threads are locked), it is detected and the program is terminated.

2 Implementation

For this project, our primary goal was to create an efficient and correct scheduler that would manage threads and manage the context switching between these threads according to the MLFQ or STCF policies. The main difficulty involved ascertaining what the next thread to run was according to the current state and once determined, carrying out a smooth and error free switch to this new thread. Furthermore, we needed to keep track of the state of multiple mutexes and the threads that were attached to these mutexes. All these mechanisms were implemented as described below.

There are 2 main pieces of code. These are:

1. **my_pthread.c:** This file contains all the various API's called by the user when a thread is to be created, destroyed, joined on, etc. All the various mutex functions are also declared and written in this file. Our main goal in this library is to abstract the various data structures used to maintain the threads and their synchronizations, and allow the user to seamlessly interact with said threads.

2. **my_pthread_t.h:** This header file declares all the data stuctures, macros, functions, enumerations, constants etc, that are used by my_pthread.c.

3 Code and APIs

The final list of methods used by us can be found in the comprehensive list below:

1. **void threadWrapper(void * arg, void *(*function)(void*), int threadId):** This function behaves as a wrapper function to retrieve the return value from the function pointed to by the thread, and store

it in an array data structure before handling the termination of the thread.

2. **int my_pthread_create(my_pthread_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg):**
This API is called by the user when a thread is to be created. The function pointer passed in the function argument is processed as the thread to be created. The 'arg' variable is the argument passed into the function to be created as the thread. The context space, stack, and properties for this thread are initialized. We also point a uc_link to a function called ProcessFinishedJob() which handles the termination of the thread if pthread_exit is not calld. This thread is then stored in a thread control block and added to the thread queue at the very top of the running priority (regardless of scheduler). In order to retrieve the return value from this function, the thread is created around a wrapper function (as mentioned above).
In addition to creating and adding the threads to the queues, in the case of a first time creation, the queue(s) are initialized and the scheduling timer is started with the value stored in *TIME_QUANTUM* which we set as a constant to 10ms. A 0 is returned on successful creation of the thread, and -1 otherwise.

3. **int my_pthread_join(my_pthread_t thread, void **value_ptr):**
Called by the parent of a thread, this API functions as a 'wait-er' on the child thread. Once the thread passed as the argument finishes, the control returns to the parent that called my_pthread_join. The thread to be joined on is first found in the thread queue. If it isn't found, the control returns to the calling parent since it must have finished execution or is not a valid threadId. Else, the 'join_boolean' flag inside the thread's control block is set to true. Once the thread finishes, the scheduler detects this flag and returns control to this join function. The thread is then detected to have completed, and the subsequent actions eventually return the control to the calling parent. If value_ptr is not NULL, we store the return value of the terminated thread, that we got in threadWrapper(), in the memory address pointed to by value_ptr. A 0 is returned on successful joining of the thread, and -1 otherwise.

4. **void my_pthread_exit(void *value_ptr):**
The thread that calls this function will be terminated. The thread is

found as the currently running thread in the queue. It's status is set to DONE, which will be detected by the scheduler and this thread will be terminated. The value_ptr pointer holds the return value from said thread. This value is retrieved and kept in an array data structure so that it may be returned at some point (when the my_pthread_join function is called).

5. **int my_pthread_yield():**
   The thread that calls this function gives up the rest of it's scheduled *TIME_QUANTUM*. This thread is context switched out of by calling the signal handler, and the scheduler finds the next thread to be run. In the case of an MLFQ scheme, this operation results in the priority of the thread staying constant in the multi-level queue. We set the "yielded" bit to 1, signifying that the last thread yielded, so we can keep its priority the same. A 0 is returned on successful yielding of the thread, and -1 otherwise.

6. **int my_pthread_mutex_init(my_pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr):**
   Initializes the state of a mutex passed in as the argument. The mutex properties are stored in a my_pthread_mutex_t struct that holds the mutexId and a boolean bit called isLocked. This is then pointed to by a mutexNode, which is added to a global list of mutexes. This mutex can then be locked and unlocked until my_pthread_mutex_destroy is called on it. A 0 is returned on successful initialization of the mutex, and -1 otherwise.

7. **int my_pthread_mutex_lock(my_pthread_mutex_t *mutex):**
   Locks the mutex that is passed as the argument. This bars access to other threads trying to access the critical section that follows this call. If the mutex is already locked, the thread's status is set to BLOCKED, and it's mutex_from variable is set to point to the mutex blocking it. The thread will not be scheduled until is unlocked, and then locks the mutex again and gains access into the following execution. This check is done at an atomic level using the __sync_lock_test_and_set() atomic instruction. A 0 is returned on successful lock of the mutex, and -1 otherwise.

8. **int my_pthread_mutex_unlock(my_pthread_mutex_t *mutex):**

Searches the mutex list to find the mutex argument. If found, performs a __sync_lock_test_and_set() to set the isLocked bit of the mutex to 0. If it was already 0, then we raise an error, since the user called unlock() twice. A 0 is returned on successful unlock of the mutex, and -1 otherwise.

9. **int my_pthread_mutex_destroy(my_pthread_mutex_t *mutex):**
   The mutex to be destroyed is found in the global mutexList by a call to findMutex(). We then check if the mutex was locked, and if so, we unlock it. After that we deallocate, remove and destroy the mutex passed in as the argument to this function. If this mutex is not found, an error is thrown and -1 is returned. As before, a 0 is returned on successful unlock of the mutex, and -1 otherwise.

10. **static void schedule():**
    This is where the logic shared by all schedulers takes place. This function has its own context that should be switched into from the function SIGALRM_Handler(). First, we get the time since the last schedule, which signifies how long the previous thread has been running. We add this to the thread's time_ran value in the TCB, which is used in the STCF scheduler. We call getRunningThread to get the previously executing thread. If it is NULL, then main was running, so we call getNextToRun(). If the next thread to run is NULL, there are no threads left in the queue, so we schedule main. If there is a thread to run, we change its status to RUNNING and switch into its context. If the previously executing thread's status is DONE, then we remove it from the queue by calling removeFromQueue(), and then if the thread was waiting on by Join(), we schedule main. Otherwise, we get the next thread to run. If the next thread to run is NULL, we have no threads left to run, so we schedule main. Otherwise, we schedule the next thread to run and set its status to RUNNING. If the previously executing thread's status was RUNNING or BLOCKED, we set its status to Ready, save its state, update its position in the queue with a call to updateThreadPosition(), and schedule the next thread to run.

11. **void SIGALRM_Handler():**
    Handles the simulated trapping into OS from hardware interrupts.

Switches context from the thread or main context to the scheduler context.

12. **void processFinishedJob(int):**
    If a thread does not explicitly call pthread_exit(), then this function will be invoked when the thread exits, and has the same functionality as pthread_exit().

13. **tcb* searchMLFQ(int):**
    searches through a MLFQ for the thread with the given threadId. Returns NULL if not found.

14. **tcb* findThread(int):**
    Searches for the thread in our queue. If the scheduler is MLFQ it calls searchMLFQ(), otherwise it calls findThreadHelper().

15. **tcb* findThreadHelper(int):**
    searches through a STCF or FIFO queue for the thread with the given threadId. Returns NULL if not found.

16. **queueNode* getRunningThread():**
    Gets the thread that was executing, if main was executing we return NULL. This is done by returning the global variable runningThread.

17. **queueNode *getNextToRun():**
    Gets the next thread to run. Begins by looking at the top of the queue and checking if the thread is blocked. If the thread is blocked, it checks if the mutex it was blocked from is unlocked. If so, the function unblocks the thread and return it to be scheduled. Otherwise it is still blocked and we check the next thread. If no threads are available to run it returns NULL.

18. **int removeFromQueueHelper(queueNode*):**
    Removes queueNode from STCF or FIFO queue and frees it. returns -1 on failure.

19. **void removeFromQueue(queueNode*):**
    Removes the queueNode from the Queue and frees the node. If scheduler is MLFQ, it calls removeFromMLFQ, otherwise calls removeFromQueueHelper. Returns the removed node.

20. **void removeFromMLFQ(queueNode*):**
    Removes the queueNode from the MLFQ and frees it.

21. **int removeFromQueueHelper_NoFree(queueNode*):**
    Removes queueNode from STCF or FIFO queue without freeing. returns -1 on failure.

22. **queueNode* removeFromQueue_NoFree(queueNode*):**
    Removes the queueNode from the Queue without freeing the node. If scheduler is MLFQ it calls removeFromMLFQ_NoFree(), otherwise calls removeFromQueueHelper_NoFree(). Returns the removed node.

23. **void removeFromMLFQ_NoFree(queueNode*):**
    Removes the queueNode from the MLFQ without freeing the node.

24. **void updateThreadPosition(queueNode*):**
    Takes a thread that was just running, removes it from the queue, and then searches for the appropriate place to re-insert the thread. This is where threads change priority levels in the MLFQ.

25. **void start_timer(int):**
    starts/restarts the timer.

26. **mutexNode *findMutex(int mutexId):**
    Searches our Mutex list for the mutex with the given mutexID.

27. **void freeQueueNode(queueNode*):**
    Frees the QueueNode and everything allocated inside it. Calls freeTcb.

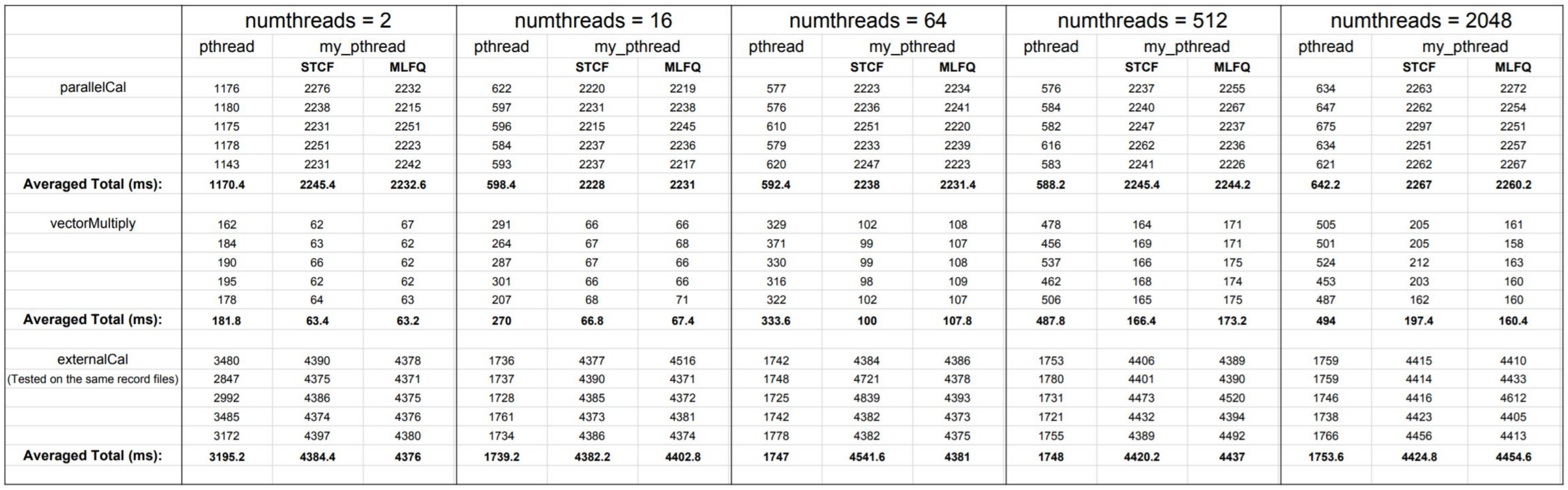
28. **void freeTcb(tcb*):**
    Frees the TCB and everything allocated inside it.

29. **void printMLFQ():**
    Prints the entire MLFQ by calling printQ() on each inner Queue.

30. **void printQ(threadQueue *queueToPrint):**
    Prints the Queue passed into the function.

4 Testing and Analysis

For this project, we needed to test both variations of our scheduler with multiple threads. The performance and runtimes were made sure to be up to par, regardless of the number of threads created. To do this, we tested our scheduling paradigms against the standard pthread library. Both scheduling paradigms were tested with all the benchmarks provided to us. We wrote shell scripts for each scheduler called makeandrun<scheduler name> which cleans both directories, makes, and runs one of drivers. The entire comprehensive analysis can be seen in the table and charts below.



	numthreads = 2			numthreads = 16			numthreads = 64			numthreads = 512			numthreads = 2048		
	pthread	my_pthread		pthread	my_pthread		pthread	my_pthread		pthread	my_pthread		pthread	my_pthread	
		STCF	**MLFQ**		**STCF**	**MLFQ**		**STCF**	**MLFQ**		**STCF**	**MLFQ**		**STCF**	**MLFQ**
parallelCal	1176	2276	2232	622	2220	2219	577	2223	2234	576	2237	2255	634	2263	2272
	1180	2238	2215	597	2231	2238	576	2236	2241	584	2240	2267	647	2262	2254
	1175	2231	2251	596	2215	2245	610	2251	2220	582	2247	2237	675	2297	2251
	1178	2251	2223	584	2237	2236	579	2233	2239	616	2262	2236	634	2251	2257
	1143	2231	2242	593	2237	2217	620	2247	2223	583	2241	2226	621	2262	2267
Averaged Total (ms):	**1170.4**	**2245.4**	**2232.6**	**598.4**	**2228**	**2231**	**592.4**	**2238**	**2231.4**	**588.2**	**2245.4**	**2244.2**	**642.2**	**2267**	**2260.2**
vectorMultiply	162	62	67	291	66	66	329	102	108	478	164	171	505	205	161
	184	63	62	264	67	68	371	99	107	456	169	171	501	205	158
	190	66	62	287	67	66	330	99	108	537	166	175	524	212	163
	195	62	62	301	66	66	316	98	109	462	168	174	453	203	160
	178	64	63	207	68	71	322	102	107	506	165	175	487	162	160
Averaged Total (ms):	**181.8**	**63.4**	**63.2**	**270**	**66.8**	**67.4**	**333.6**	**100**	**107.8**	**487.8**	**166.4**	**173.2**	**494**	**197.4**	**160.4**
externalCal	3480	4390	4378	1736	4377	4516	1742	4384	4386	1753	4406	4389	1759	4415	4410
(Tested on the same record files)	2847	4375	4371	1737	4390	4371	1748	4721	4378	1780	4401	4390	1759	4414	4433
	2992	4386	4375	1728	4385	4372	1725	4839	4393	1731	4473	4520	1746	4416	4612
	3485	4374	4376	1761	4373	4381	1742	4382	4373	1721	4432	4394	1738	4423	4405
	3172	4397	4380	1734	4386	4374	1778	4382	4375	1755	4389	4492	1766	4456	4413
Averaged Total (ms):	**3195.2**	**4384.4**	**4376**	**1739.2**	**4382.2**	**4402.8**	**1747**	**4541.6**	**4381**	**1748**	**4420.2**	**4437**	**1753.6**	**4424.8**	**4454.6**

Figure 1: Table of run times across different number of threads, library implementations, scheduling paradigms, and drivers.

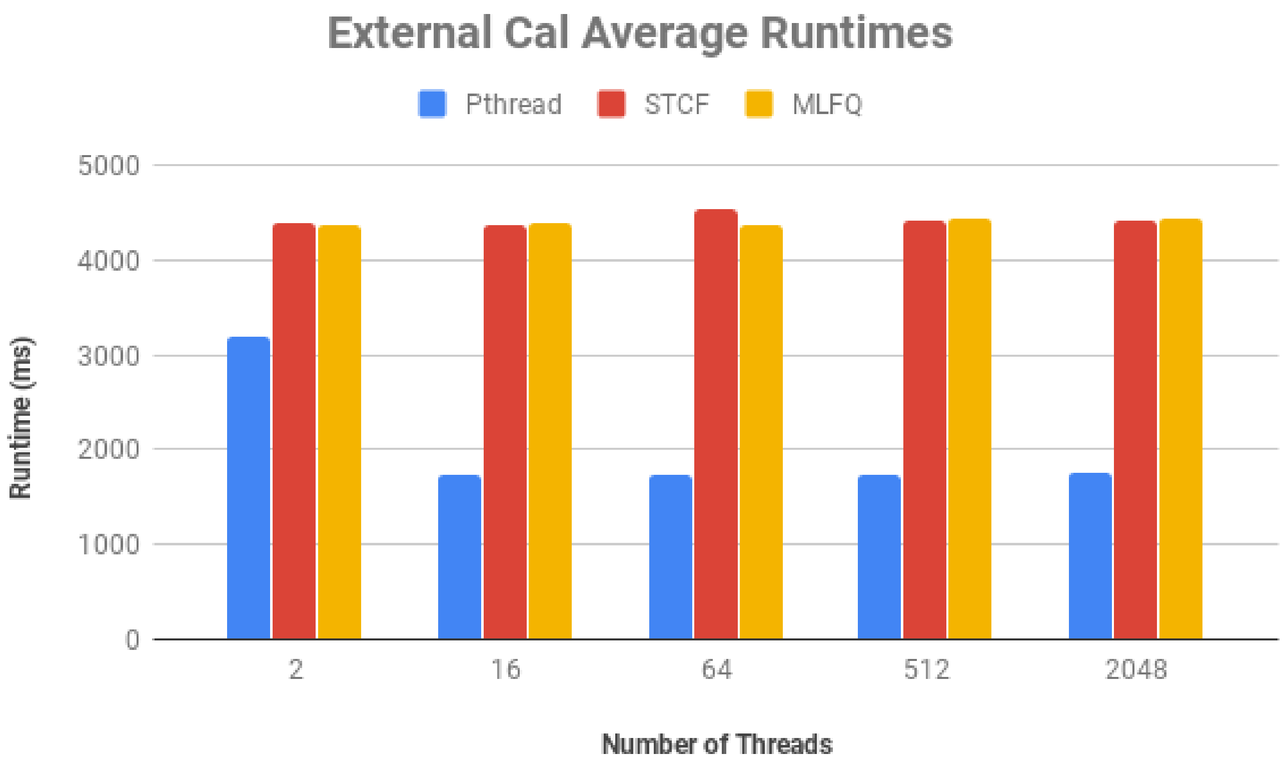


Figure 2: Chart comparing average runtimes for different scheduling paradigms from the externalCal driver.

As we can see from the chart above, the pthread library implementation's runtime decreases drastically when the number of threads increase, since it can be run simultaneously. Our library only simulates simultaneity, and so our runtime will not decrease drastically.

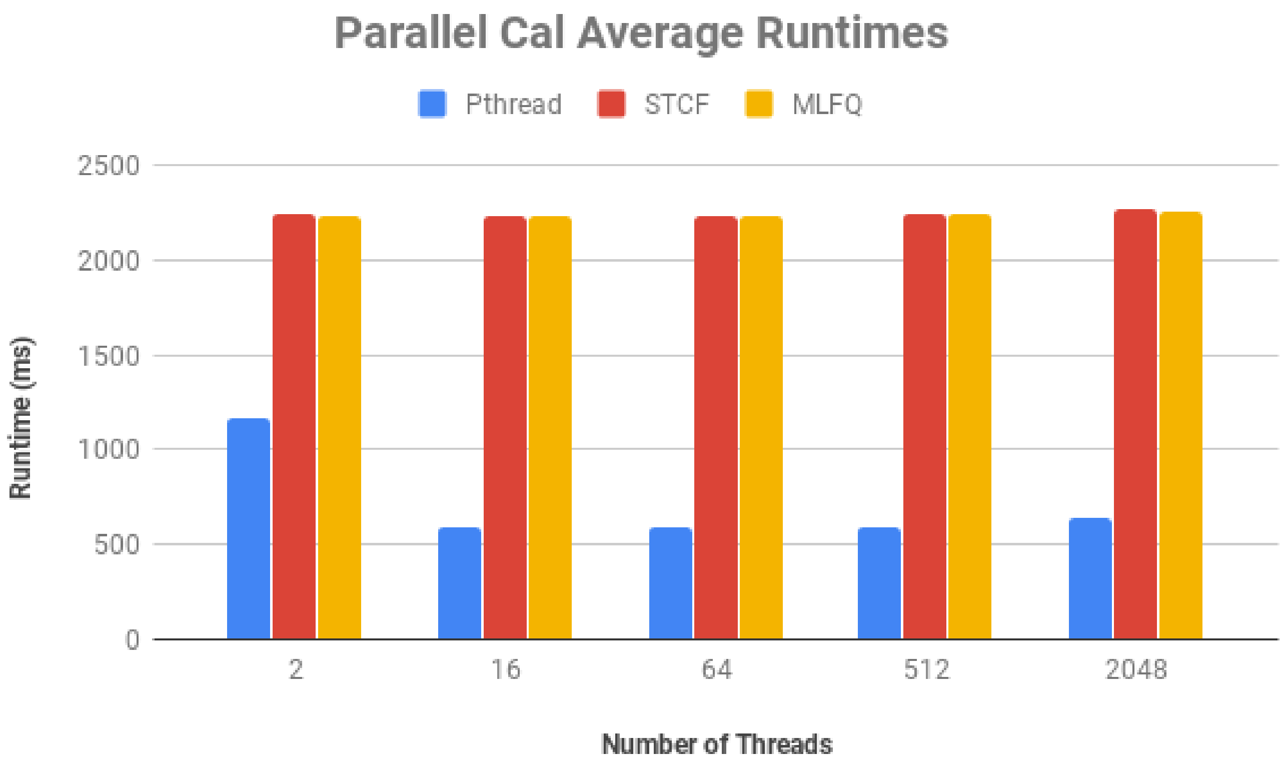


Figure 3: Chart comparing average runtimes for different scheduling paradigms from the parallelCal driver.

The chart above behaves similarly to the data collected from the externalCal driver, the pthread library decreases in runtime when the number of threads increase, but our schedulers have somewhat consistent runtimes. We can also see that the STCF and MLFQ have similar runtimes.

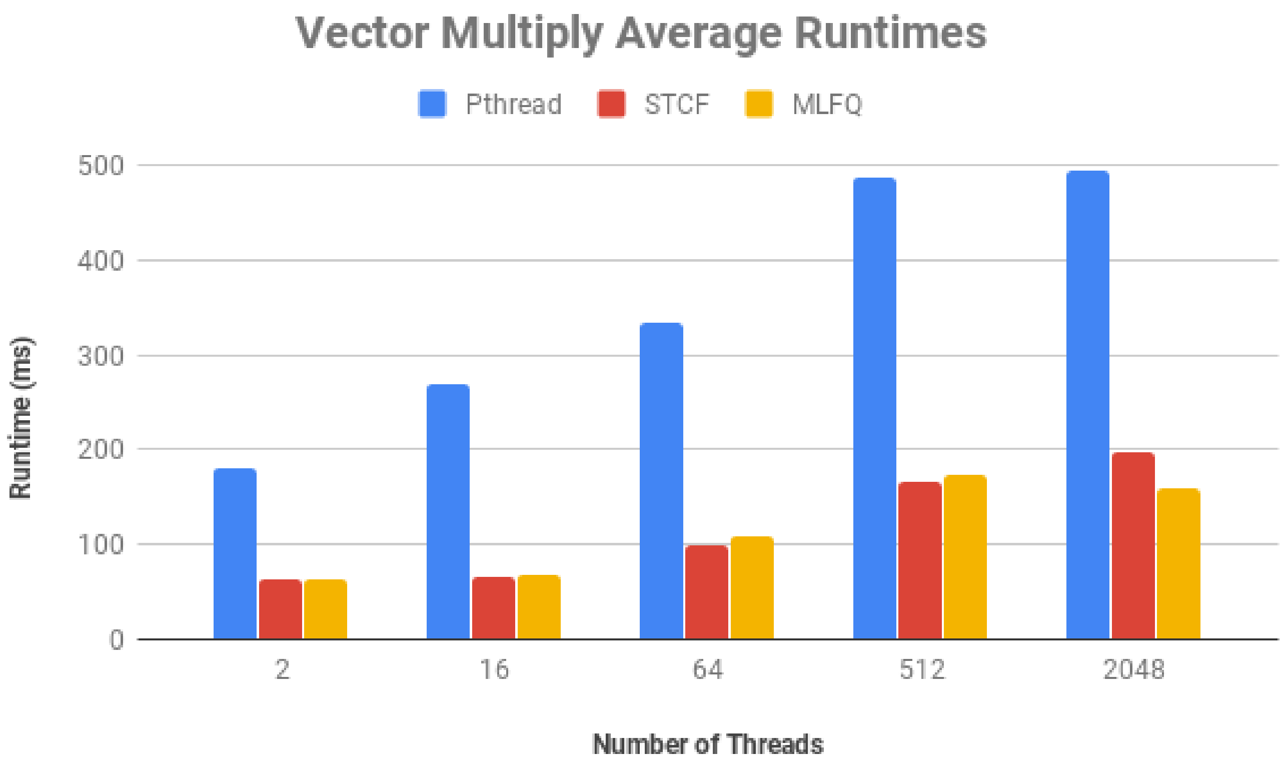


Figure 4: Chart comparing average runtimes for different scheduling paradigms from the vectorMultiply driver.

However we can see from the chart above that our scheduling paradigms outperformed the pthread library on the vectorMultiply driver. The time taken by each individual paradigm is seen to increase, however the pthread library is consistently slower than the other two.

5 Conclusion

The biggest challenge that is prevalent when trying to schedule threads efficiently is predicting how and when the threads will finish. Since there is no way to have this prescient awareness, our schedulers attempt to pick the best thread to run based on previously gathered information from the thread.

In STCF, we assume that the thread that has run for the least amount of time so far and used up the minimum number of it's entire *TIME_QUANTUM* intervals. This is a naive attempt to predict how a thread is going to run, but it is useful under the assumption that the longer a thread has already run, the longer it has yet to run as well.

The MLFQ uses a multi level, 4 pronged queue structure. Our implementation kept track of the priority of each thread. This priority value changed (priority was lowered) when the thread used up an entire time quantum. However, when the thread was detected to have yielded control to the scheduler, it's priority was maintained at the same level. The thread at the front of the highest priority queue was always selected to run. We observed that when running the three benchmark programs, the longest threads always sifted to the bottom most queue. This means that the MLFQ is a decently accurate way of predicting the run time of individual threads in certain scenarios.

This being said, we noticed that during testing, a lot of the threads sank to the bottom most queue and stayed there until the end of the program. Having observed this, a possible next step could be to implement 'boosting'. While this is out of scope for this project, as mentioned in the textbook, this would result in all threads having their priority increased to the top level queue every few milliseconds or so. Therefore, initially blocked threads would then get another opportunity to undergo scheduling and handling.

One of the main issues we faced during this project was understanding the ucontext library and the differences between setContext and swapContext, but after researching their different functionalities, we were able to design our scheduler to save contexts and swap between them as and when required. There were plenty of times where our scheduler swapped to an unknown context, but we were able to use gdb and debugging statements to fix it and learn more about switching these contexts.

Finally, once we got our FIFO scheduler to work and produce the same output as the pthread library, we implemented the STCF and MLFQ paradigms and successfully got them to work as well. We learned a lot about schedulers as well as context switching, and we are proud that our schedulers can compare to the pthread library and produce the same output.