CS 416: Project 2 Report

Sanjana Pendharkar (sp1631) and Vaishnavi Manthena (vm504)

March 11, 2021

1 Part 1: Detailed logic of how you implemented each API functions and scheduler

A central data structure that we used was the runqueue, which was a linked list of all rpthreads. Each rpthread contains a thread control block (tcb). A tcb has the parameters 'tid' (thread ID), 'status' ('READY': when a thread can be pushed onto runqueue or 'BLOCKED': when a thread is unavailable for further action), 'cntx' (thread context), and priority (this was used for the MLFQ implementation to keep track of the level the thread was most recently acquired from). The mutex has the following flags that describe its status: (1) 1 - mutex is unavailable, (2) 0 - mutex is available, with a pointer to the thread that it belongs to (this thread holds the mutex lock).

Other basic data structures that we used were:

- Scheduler: A pointer to a runqueue which holds all the ready jobs in case of RR. For MLFQ, the scheduler is holding all the ready jobs of the top level in a linked list of nodes. This is used for both MLFQ and RR.
- Level1: pointer to runqueue which holds all the ready jobs of level 1. This is used only for MLFQ.
- Level2: pointer to runqueue which holds all the ready jobs of level 2. This is used only for MLFQ.
- Level3: pointer to runqueue which holds all the ready jobs of level 3. This is used only for MLFQ.
- blockedList: a pointer to runqueue which holds all the blocked jobs for both MLFQ and RR implementations.
- val: an rpthread_t insttance that is used to assign unique thread values for each thread created.
- current: pointer to the current rpthread that has been scheduled and is running.

- isBlocked: an integer that is 0 if the current thread has not been blocked, else 1.
- didYield: an integer that is 0 if the current thread did not yield yet, else 1.
- didExit: array that tells us whether or not an array with a particular thread id has exited.
- exitValues: array that tells us the value returned by a thread that exited. This is an array of pointers to void. Each index of the array points to the value returned by the thread whose thread id is equal to this index.

One of the main helper function we used is 'addToRunQueue'. In this function, we have the following inputs: (1) pointer to a thread, and (2) pointer to our runqueue struct. addToRunQueue's main functionality is that it creates a node with the parameter thread, and adds it to the end of the runqueue. This function is important, as it is used to implement RR, each level of the MLFQ implementation and the blocked list.

Function we used to handle the timer interrupts: ring(int signum). This function just swaps the context back to the scheduler so that another thread could be scheduled.

rpthread_create: In the case that this is the first create that is being called, we follow the below steps to initialize important global structures for the library:

1. We initialize the exitValues and didExit arrays.
2. Registering the signal handler and pointing it to the ring function for whenever a timer interrupt occurs.
3. Allocating memory for all the runqueues (scheduler, different levels of MLFQ, blocked list).
4. Setting up the scheduler context using make context and get context.

All of the above steps are part of the helper function 'initialize().' In case this is the first thread, the main context is set up using getcontext. The main thread is created and added to the scheduler runqueue and we shift to the scheduler's context. The purpose of this is so that the main thread could get scheduled correctly and we could continue the creation of the first user level thread from this point. This way when the first user level thread is created we also ensure that a main thread is created so that later on it could participate similarly to all the other threads.

Now, once we take care of the main thread (which only happens on the first call to rpthread_create function), we set up the new thread that has to be created. We allocate memory for the thread structure, the tcb, and context stack. We set up the context using make context. Also we initialize all other attributes

of the thread control block including STATUS, PRIORITY, THREAD ID, and CONTEXT. Now that thread is set up we add it to the scheduler (a runqueue pointer). This process is applicable for both RR and MLFQ. For RR we add to 'scheduler' because that is the job queue for RR, and we also reused it as the top priority level for MLFQ. In MLFQ, every new thread is added to top priority.

rpthread_yield: In this function, our implementation involves first setting the current thread's control block's status to READY, which means that our thread can be scheduled. We then set our integer value of didYield to 1, which lets our scheduler know that the thread has yielded (finished) before its time slice. Finally, the thread context is swapped out using the swap_context() function from the current thread to the scheduler thread, so that the current thread is enqueued back to the runqueue and the next thread that is ready to run can be accessed.

rpthread_exit: In this function, the parameter value_ptr points to the value returned by this thread. So, we correspondingly update the exitValues array, so that later this returned value can be viewed by the 'join' function. Next, we set the value in index = thread id of the thread exiting of the didExit array to 1, so that later the join function will know that this thread has exited. We free the memory associated with this thread (the thread structure, the tcb structure, and the context's stack) by using the helper function 'freeThread.' Now we set the global variable 'current' to NULL to indicate that no thread is currently running. Now, since this thread stopped running we set the context back to the schedule's context so that we can schedule another thread to run. *Note: for both yield and exit we also disarm the timer so that the scheduler would not be affected by timer interrupts.*

rpthread_join: By using a while loop we wait for the thread with thread id given by the parameter 'rpthread_t thread' to exit. We would know if the thread has exited using the global didExit array. If the parameter 'value_ptr' is not equal to NULL, then it should also contain the returned value of the thread it was waiting for. So, once we come out of the while loop (meaning the thread we are waiting for has exited) we assign the value that it returns (this value is obtained from the global 'exitValues' array) to *value_ptr. Finally we return 0, indicating join function has completed.

Thread Synchronization: To make sure that access to data across threads is synchronized, we had to revise the mutex structure. In our rpthread_mutex_t, we first created an int mutex initialization variable, flag, which would be set to 0 when the mutex is available for use and 1 when the mutex is not. Then, we created a pointer to the thread holding the mutex, so that we can find which thread is holding the lock. Below are the methods that we implemented to make sure that thread synchronization works:

Thread Mutex Initialization: In this function, we are being asked to initial-

ize a mutex created by the calling thread. Here, we first check for an invalid pointer to mutex. We initialize the mutex's flag to 0 to indicate its initial state where it is available for locking. Initially, we wanted to initialize the thread of this mutex through malloc, but we opted not to use this line because when we create a thread, we have already allocated memory for it.

Thread Mutex Lock: In this function, we attempt to lock the mutex for a caller thread. To do so, we first use the built-in test-and-set atomic function to check if the mutex is available and can be acquired. If the atomic test-and-set function returns 1 after setting it to one, this means that the lock has already been acquired by some other thread. In this case, we find that the lock cannot be acquired. We first push the current thread into the blockedList, and set its status to BLOCKED (as it is not ready to run yet). Next, we set isBlocked to 1, to ensure that the scheduler does not enqueue the current thread once again. Then, we use swapcontext to make a context switch from the current thread to the scheduler thread, after we save the current thread's context. In the other case (the atomic test-and-set function returns 0 after setting it to one), we find that the lock has been successfully acquired by the current thread. We set the mutex's thread to the current thread, and set the flag to 1, making it unavailable.

Thread Mutex Unlock: In this function, we are unlocking a given mutex, so that once it is released, other threads are able to access and lock the mutex. Here, we first set the mutex flag to 0, making it available and we set the mutex's thread to null (because a thread no longer locks the mutex). Then, we call 'blockToRun', which is a helper function that adds all threads in the block queue to the runqueue. blockToRun takes each thread from blockedList and adds it to runqueue based on implementation (RR or MLFQ) and priority (which is only needed for MLFQ), and then it frees that node in the blockedList.

Thread Mutex Destroy: According to our implementation we did not have to do anything in this function. Our mutex only has an rpthread pointer and an integer flag. The mutex is destroyed by the user. The integer does not have to be destroyed. The thread will be destroyed once it exits. So, we simply just check that the mutex is not NULL. We return -1 if the mutex is null (invalid since it does not exist), else we just return 0.

Here is a detailed description of blockToRun:
In blockToRun, we first set a pointer to blockedList's head, as we will be traversing through the blockedList and adding each thread to a job queue. We also have a prev ptr which is allows us to free the nodes of the blockedList as we move on. If the our library uses RR, then we add the thread at every node to the 'scheduler' runqueue. If or library has been implemented in MLFQ, we set p as an integer that describes the thread's control block's priority. If p = 1, we add the thread to level 1 (the same goes for p = 2, p = 3, and p = 4). Finally, once we are done with this loop, we set blockedList's head and tail to null because there are no more threads to iterate over and the blockedList does

not contain any more threads.

Schedule: In the scheduler function, we are being asked to choose and run a new thread, essentially whenever a timer interrupt occurs. To do so, we implement simple if-then to determine if we should call sched_rr (which schedules Round Robin), or sched_mlfq (which schedules MLFQ).When we call sched_rr, we send it the parameter 'scheduler' since this is the only job queue RR deals with.

sched_RR: This function is modular and applicable for both RR and MLFQ purposes.

Detailed description of RR:
We check if the libraries implementation is RR or MLFQ. If it is RR, then we add the current thread (the one which was just running before schedule() was called) back to the tail of the 'scheduler queue' given that the thread has not exited and is not blocked. This is accordance with how the RR enqueues the last thread back on to the end of the job queue. In case of MLFQ the step we described (adding job back to queue) will be accounted for in sched_mlfq().

The next step in sched_rr() is applicable for both RR and MLFQ. In this step we pop the head of the queue and get ready to schedule the thread of this node. To schedule the thread of this node, we first update certain global variables or structures. We set the global value rpthread * current to point to this new thread. We reset isBlocked back to 0 (to iitialize that this new thread has not been blocked so far). We set its status as scheduled. We also free the node we popped from the queue. Then we start the timer and set the context as the new thread's context. In case of RR, the queue that we are talking about is the 'scheduler.' In case of MLFQ, the queue that we are talking about is the either level1, level2, level3, or the scheduler. The queue we want to deal with is correctly passed onto as the parameter to sched_rr.

sched_MLFQ: This method is used to schedule threads in case of mlfq.

Detailed description of MLFQ: This method does not apply to the RR implementation, only for mlfq. Again if the current thread (the one which was just running before schedule() was called) has not been blocked or has not exited, based on whether it has yielded or not, we set a new priority to this thread and add it to the end of the job queue at the corresponding level. If it has yielded (the global variable didYield is 1), then we don't change it priority and we add it to the same level from which it was removed before. Note: we know this level because we store it in the thread's tcb. We reset didYield to 0. In case the didYield was 0, we reduce the priority by 1 (unless priority was originally 1) and add it to that corresponding job queue.

In the final step we use RR to schedule a thread from a particular queue. This

particular queue is the non-empty queue at the highest priority level.

As a final note, just to clarify, in our implementation the queues only have the user threads and the main thread. The schedule's context is not part of the job queues.

Also, since the project description and professor specified that the number of threads we will be tested on will be 200 or below, our code only works up until 249 threads. This is just because we only allocated that much space to store thread return values.

2 Part 2: Benchmark results of your thread library with different configurations of thread number

Parallel_calc.c

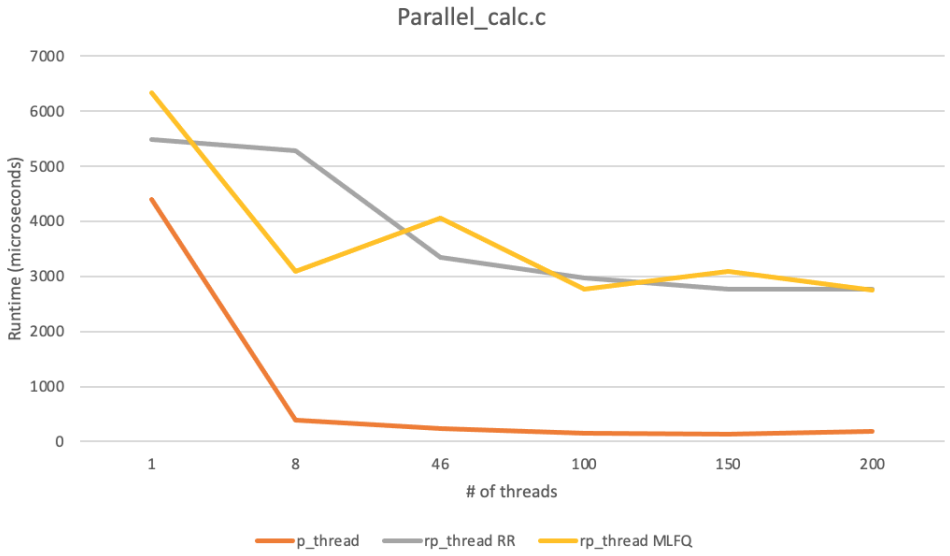
Number of threads	1	8	46	100	150	200
p_thread library runtime	4391 μs	398 μs	231 μs	146 μs	132 μs	191 μs
rp_thread library runtime with RR	5484 μs	5283 μs	3351 μs	2963 μs	2770 μs	2770 μs
rp_thread library runtime with MLFQ	6333 μs	3089 μs	4056 μs	2762 μs	3089 μs	2752 μs

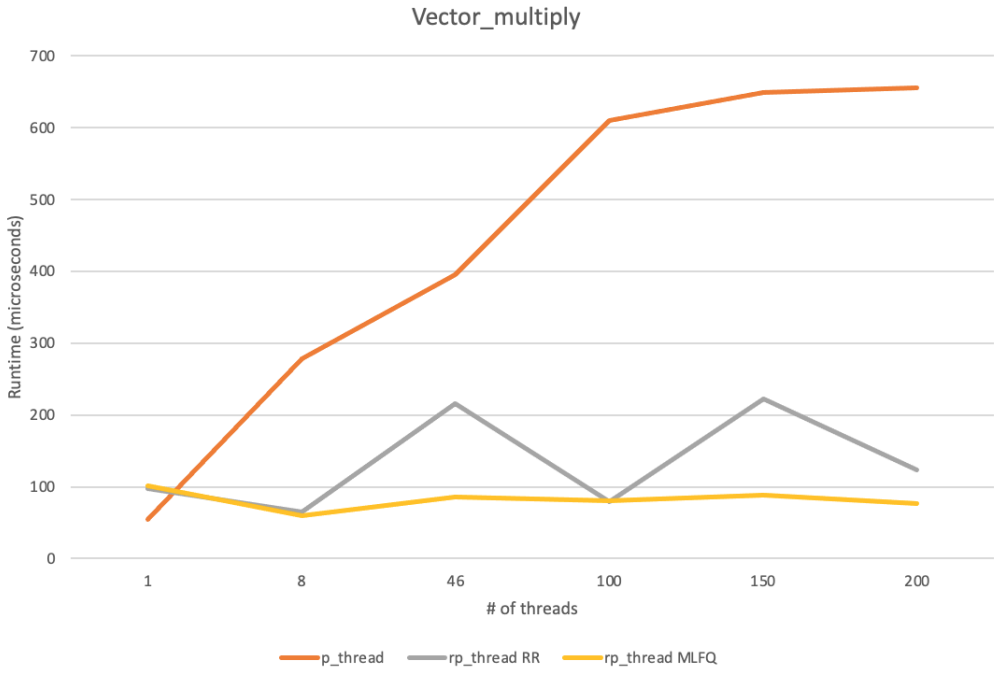
Vector_multiply.c

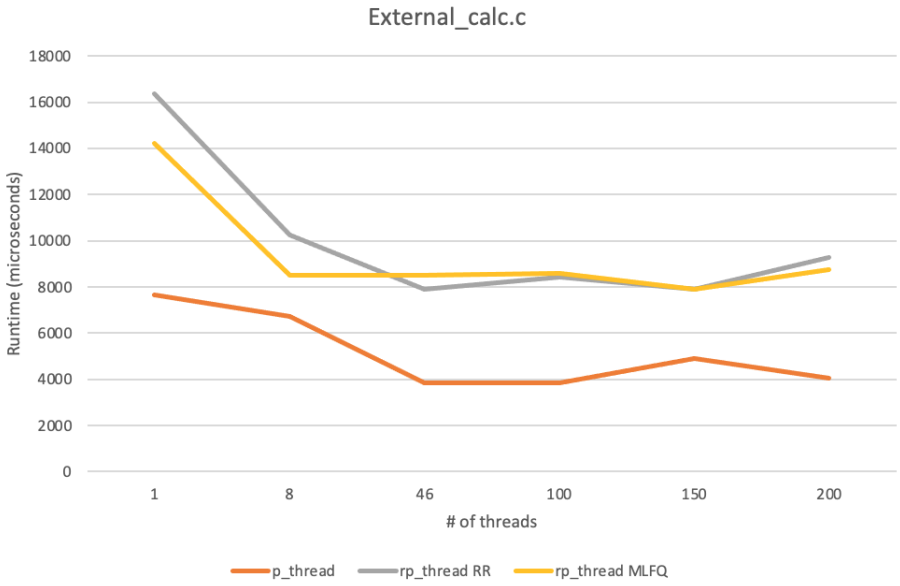
Number of threads	1	8	46	100	150	200
p_thread library runtime	55 μs	278 μs	395 μs	610 μs	649 μs	655 μs
rp_thread library runtime with RR	98 μs	65 μs	216 μs	79 μs	222 μs	123 μs
rp_thread library runtime with MLFQ	101 μs	60 μs	86 μs	80 μs	89 μs	77 μs

External_calc.c

Number of threads	1	8	46	100	150	200
p_thread library runtime	7638 μs	6722 μs	3839 μs	3860 μs	4920 μs	4067 μs
rp_thread library runtime with RR	16375 μs	10268 μs	7892 μs	8416 μs	7911 μs	9295 μs
rp_thread library runtime with MLFQ	14209 μs	8491 μs	8505 μs	8571 μs	7908 μs	8751 μs


Parallel_calc.c
7000
6000
5000
4000
3000
2000
1000
0
Runtime (microseconds)
1
8
46
100
150
200
# of threads
p_thread
rp_thread RR
rp_thread MLFQ


Vector_multiply
700
600
500
400
300
200
100
0
Runtime (microseconds)
1
8
46
100
150
200
# of threads
p_thread
rp_thread RR
rp_thread MLFQ



3 Part 3: A short analysis of the benchmark results and comparison of your thread library with pthread library.

Some patterns that we noticed:

In case of parallel_cal.c and external_cal.c, our implementation took more time overall as compared to the pthread library. We speculate that this is due to the way we implemented our main thread. In our case, whenever we schedule the main thread, in the case that it is waiting for a join to return, it just waits and does nothing during the entire timeslice. Therefore, all of such timeslices pass by with no work being done. However, in case of the pthread library it may be using certain mechanisms to avoid this waste of time.

In case of all the benchmarks our RR implementation took more time overall as compared our MLFQ implementation. We think this is due to how MLFQ reduces the priority of those jobs which use up the entire time slice. So, it might be reducing the priority of main function during join so that we would schedule the main function later, and by that time the other threads might have exited and the join may take less time. On the other hand, the user level threads end up staying in the priority level because they get blocked before their time slice. So, MLFQ' s prioritization might be helping its runtime.

In case of vector_multiply.c, our implementation takes less time than the ac-

tual pthread library. This might be due to the vector_multiply being slightly simple and the complicated mechanisms used by pthread library may add extra overheads. This might also be due to the timeslice values used by the pthread library. If they use values less than our time slice this might be extra overhead as well.

Finally, the 2 pictures below are some test results that we got by adding some helper functions to check the RR functionality. Note: the code to generate this has been commented out. In addition to the normal result we also printed out how many times each thread is being scheduled for RR. Note the first entry with tid = 0, is the main thread. It is being scheduled for a greater number of times since it does work before thread creation and potentially after thread exit (in case there is a join). However, the rest of the threads are scheduled for about the same number of times.

Below are some examples of how RR responded to different threads:

```
Timeslice changed from 15 to 5:

./parallel_cal 10

Result:


running time: 2995 micro-seconds
sum is: 83842816
verified sum is: 83842816
tid=  0 : number of times it ran: 272
tid=  1 : number of times it ran: 23
tid=  2 : number of times it ran: 23
tid=  3 : number of times it ran: 23
tid=  4 : number of times it ran: 23
tid=  5 : number of times it ran: 23
tid=  6 : number of times it ran: 23
tid=  7 : number of times it ran: 23
tid=  8 : number of times it ran: 23
tid=  9 : number of times it ran: 23
tid=  10 : number of times it ran: 23
```

```
With time slice in makefile:

./parallel_cal 20

Result:

running time: 3743 micro-seconds
sum is: 83842816
verified sum is: 83842816
tid=  0 : number of times it ran: 302
tid=  1 : number of times it ran: 9
tid=  2 : number of times it ran: 9
tid=  3 : number of times it ran: 9
tid=  4 : number of times it ran: 10
tid=  5 : number of times it ran: 9
tid=  6 : number of times it ran: 9
tid=  7 : number of times it ran: 9
tid=  8 : number of times it ran: 9
tid=  9 : number of times it ran: 9
tid=  10 : number of times it ran: 9
tid=  11 : number of times it ran: 10
tid=  12 : number of times it ran: 10
tid=  13 : number of times it ran: 10
tid=  14 : number of times it ran: 10
tid=  15 : number of times it ran: 10
tid=  16 : number of times it ran: 10
tid=  17 : number of times it ran: 10
tid=  18 : number of times it ran: 10
tid=  19 : number of times it ran: 10
tid=  20 : number of times it ran: 10
```

```
With time slice equal to 500

./parallel_cal 1 |

Result:

running time: 5813 micro-seconds
sum is: 83842816
verified sum is: 83842816
tid=  0 : number of times it ran: 12
tid=  1 : number of times it ran: 6
```

This last picture above is just to show how are implementation works with round robin, one thread and time slice 500.