CS 518: Project 2 Report

Gona Srikar (gs943) and Abhilash Kolluri (ak2048)

March 15, 2022

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

The central Data Structure we used is RunQueue. We implemented it using LinkedList. Which has the push and pop operations, It consists of all the worker\_threads**.** Each thread consists of thread-controlled block also called as TCB.A TCB has the parameters ‘tid’ (thread ID), ’cntx’ (thread context), ’status’ (’READY’: when a thread can be pushed onto runqueue or ’BLOCKED’: when a thread is unavailable for further action), priority (this was used for the MLFQ implementation to keep track of the level the thread was most recently acquired from).

We also implemented the mutex, It has the 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 we used are –

* Scheduler - A pointer to a runqueue which holds all the ready jobs in case of RR (Round Robin). For MLFQ, the scheduler is holding all the ready jobs of the top level in a linked list of nodes. This Scheduler Data Structure is used for both MLFQ and RR.

We also created some Data Structures which are used only for MLFQ, because it has multiple levels unlike Round Robin.

* Level-1 (mlfq\_level\_1): pointer to runqueue which holds all the ready jobs of level 1. This is used only for MLFQ.
* Level-2 (mlfq\_level\_2): pointer to runqueue which holds all the ready jobs of level 2. This is used only for MLFQ.
* Level-3 (mlfq\_level\_3): pointer to runqueue which holds all the ready jobs of level 3. This is used only for MLFQ.
* blocked\_thread\_list : a pointer to runqueue which holds all the blocked jobs for both MLFQ and RR implementations.
* id: an worker\_t instance that is used to assign unique thread values for each thread created. Also can be called as thread\_id.
* running\_thread: pointer to the current worker\_thread that has been scheduled and is running.
* is\_current\_thread\_blocked: an integer that is 0 if the current thread has not been blocked, else 1.
* did\_thread\_yield: an integer that is 0 if the current thread did not yield yet, else 1.
* thread\_exit\_status: array that tells us whether or not an array with a particular thread id (id) has exited or not.
* thread\_exit\_value: 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 ‘insert\_queue’. In this function we add the thread to the end of the run\_queue. This function has two parameters. 1. Pointer to the thread. 2. Pointer to the run\_queue struct. It creates a new node from the function parameter - Pointer to the thread and adds it to the end of run\_queue. It is used for 3 functionalities:

1. Implement RR (Round Robin)
2. Implement each level of MLFQ
3. Implement the blocked list.

We used timer\_handler function to handle timer interrupts. It swaps context to scheduler when the timer triggers.

**worker\_create:** This is the first create function we call for the creation of thread. Inside this we first call the initialize() function. It consists of the following steps :

1. Initialize and register the signal handler for timer interrupt.
2. Creating Run Queues for both Schedulers (RR and MLFQ) to hold list of jobs.
3. If a thread requires a mutex and mutex is not available, keeping those threads into blocked list.
4. Initializing arrays which will hold all the exit information of threads.
5. Creating all the pre-requisites for Scheduler context and to setup it to start running the Schedule function.
6. Setting the global context so that program goes to schedule function.

All the above steps are part of the initialize() function. After that we do the following steps. If it is the first thread, the main context is setup using the getcontext() function. After that we push the main thread into run\_queue. This main thread is given an id of 1, and all the other threads starting from here are given an id starting from 2. Following that we update that the Scheduler is started and transfer the control to the Scheduler. By doing the above steps the main thread gets scheduled correctly and afterwards we could create the first user level thread. This way we ensure that the main thread gets created and it could participate similarly to the other threads created thereafter.

Now, once we take care of the main thread (which only happens on the first call to worker create function), we set up the new thread that has to be created. We allocate memory for the thread structure, the context stack, and tcb. We set up the context using make context function. At the same time we also initialize various other attributes of the TCB such as Thread\_ID, context, Priority, Status. Now the thread is ready we add it to the run\_queue. This process is applicable for both RR and MLFQ. For RR we add to ‘RR’ (pointer to the run\_queue) 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. We also store the thread start time in the thread\_start\_time array.

**worker\_yield**: In this function we first set the worker thread state in the TCB to READY. There is a variable called did\_thread\_yield, this is useful to determine whether thread should be in same priority or should priority be reduced in MLFQ scheduling. Before context swiching the thread we disable the timer for this thread. Then using swapcontext() function we context switch this thread to the scheduler thread. Before doing that we save the context of this thread to the TCB of it. The current thread is enqueued back to the run queue and the next thread that is ready to run can be accessed. In this function, we context switch from thread context to scheduler context.

**worker\_exit:** In this function we de-allocate any dynamic memory created when starting this thread. At first we get the index of the exiting thread and then we update the thread\_exited\_status to 1. If the passed value\_ptr in the function parameter is not null we populate it the exit status. We free the memory associated with this thread (the thread structure, the tcb structure, and the context’s stack) by using the helper function ‘free\_thread\_memory’ Now we set the global variable ‘running\_thread’ to NULL to indicate that scheduler is aware not to pick up new thread from the queue. We then disable the timer for this thread before exiting. At the last step we switch the scheduler context to run new threads by using setcontext() function.

**worker\_join:** In this function we wait for thread termination. We de-allocate any dynamic memory created by the joining thread. The exit array will be 1 if thread has exited, we need to wait until the thread exited to perform a thread\_join. If the value provided by argument (function argument) (value\_ptr) is not null, populate the exit status. 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:

**worker\_mutex\_init:** We use this function to initialize the mutex lock. Here, first we do NULL check for passed parameter of mutex, If it is NULL we return -1. Generally, Mutex is allocated memory at user space and given a pointer to same. Later we set the mutex\_flag to 0 and return 0 to the function. Initially, we wanted to initialize the thread of this mutex through malloc, but we opted not to use this because when we create a thread, we have already allocated memory for it.

**worker\_mutex\_lock**: We use this function to acquire the mutex lock. We use the built-in test-and-set atomic function to test the mutex, if the mutex is acquired successfully, we enter the critical section, if acquiring mutex fails, push current thread into block list and context switch to the scheduler thread. We use atomic\_test\_and\_set to set the mutex flag. We have 2 cases where the mutex acquire failed and the other one being is successful. In the first case, We need to add this thread to the blocked list. After that, we check to know if a thread that is waiting on the mutex is blocked. We do not schedule the thread in that case. If acquiring lock was successful, then we assign the thread pointer in mutex to this thread.

**worker\_mutex\_unlock:** We use this function to release the mutex lock. At first, we release mutex and make it available again. After that, we put threads in block list to run queue so that they could compete for mutex later. The sequence of steps to achieve that are Updating the flag, updating the thread pointer, then we release whatever from blocked list to run queue. Finally we return 0, indicating unlock function has completed.

**worker\_mutex\_destroy:** According to what we have implemented, we didn’t have much to implement in this function. Our mutex which is defined has rp\_thread pointer and an integer flag, the mutex is destroyed by the user. In this function, We return -1 if the mutex is null (invalid since it does not exist), else we just return 0.

**Schedule**: every time a timer interrupt occurs, your worker thread library should be context switched from a thread context to this schedule() function. We invoke scheduling algorithms according to the policy (RR or MLFQ). In this 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. Also, in this function we added a condition to calculate the average\_response\_time and average\_turn\_around\_time.

**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() function was

called) back to the tail of the ‘scheduler queue’ using the insert\_queue function,

given that the thread has not exited and is not blocked. This is accordance with how the RR scheduling algorithm works, by enqueuing the last thread back on to the end of the jobs queue. In case of MLFQ the step we implement the logic in sched\_mlfq() function.

The next step in sched\_rr () is applicable for both RR and MLFQ schedulers. We do scheduling only if there is a job present in the queue. At first we Pop the head from the scheduler, which is to be run, then we move the head pointer to next job. If only 1 job is present in the queue and it has been popped, then tail must also be pointed to null implying that queue is empty. Temp is the popped head node. Then we remove the reference to next jobs that temp is pointed, so we can schedule it. After that we populate the current thread to the popped job from run queue, i.e., we assign the popped job from run queue to the current thread. We update the variable is\_current\_thread\_blocked = 0 indicating that the current running thread is not blocked. Since the node pointing to thread is scheduled, we can free temp. In the subsequent step we configure the timer such that it expires after the time slice. We then set the context as the new thread context using the setcontext function.

**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, it appears only for mlfq. At first we check if current thread is not null and it is not in blocked state. Afterwards we retrieve the priority of the of the thread and change the status of it to READY. If that thread yielded we put it into the same priority level. If the thread is not yielded, the priority should be reduced by 1. Hence we need to place it in a low priority queue. In MLFQ there are different priority levels. We perform RR scheduling approach for each priority level.

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.

We have defined the maximum number of threads the code supports. We have set that value to 250. It is defined in the top of the worker.c file.

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

ms – milli seconds ; µs – micro seconds

Parallel\_calc.c

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 6 | 45 | 102 | 140 | 200 |
| p thread library runtime | 2842 ms | 524 ms | 188 ms | 106 ms | 192 ms | 176 ms |
| rp thread library runtime with RR | 5415 ms | 3182 ms | 2783 ms | 2749 ms | 2761 ms | 2761 ms |
| Avg\_response\_time | 5409613 µs | 3155038 µs | 2699751 µs | 2613736 µs | 2518947 µs | 2340384 µs |
| Avg\_turnaround\_time | 5415788 µs | 3158354 µs | 2703076 µs | 2616043 µs | 2522182 µs | 2344348 µs |
| rp thread library runtime with MLFQ | 5415 ms | 3285 ms | 2828 ms | 2739 ms | 2762 ms | 2754 ms |
| Avg\_response\_time | 5413458 µs | 3275445 µs | 2733853 µs | 2626350 µs | 2545646 µs | 2335709 µs |
| Avg\_turnaround\_time | 5414573 µs | 3277515 µs | 2737238 µs | 2628535 µs | 2548748 µs | 2339797 µs |

Vector\_Multiply.c

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 6 | 45 | 102 | 140 | 200 |
| p thread library runtime | 62 ms | 261 ms | 404 ms | 552 ms | 597 ms | 641 ms |
| rp thread library runtime with RR | 88 ms | 70 ms | 80 ms | 80 ms | 72 ms | 70 ms |
| Avg\_response\_time | 85492 µs | 39990 µs | 42631 µs | 49496 µs | 38036 µs | 37783 µs |
| Avg\_turnaround\_time | 88006 µs | 44707 µs | 44334 µs | 50308 µs | 38518 µs | 38099 µs |
| rp thread library runtime with MLFQ | 88 ms | 60 ms | 76 ms | 87 ms | 90 ms | 80 ms |
| Avg\_response\_time | 85422 µs | 37003 µs | 41741 µs | 46517 µs | 48512 µs | 42786 µs |
| Avg\_turnaround\_time | 88382 µs | 42374 µs | 43289 µs | 47305 µs | 49099 µs | 43155 µs |

External\_cal.c

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 6 | 45 | 102 | 140 | 200 |
| p thread library runtime | 681 ms | 2721 ms | 2526 ms | 2394 ms | 2558 ms | 2512 ms |
| rp thread library runtime with RR | 1085 ms | 690 ms | 640 ms | 645 ms | 646 ms | 632 ms |
| Avg\_response\_time | 1078422 µs | 588440 µs | 186723 µs | 69752 µs | 54834 µs | 72170 µs |
| Avg\_turnaround\_time | 1084445 µs | 591978 µs | 187854 µs | 70135 µs | 55143 µs | 72411 µs |
| rp thread library runtime with MLFQ | 1084 ms | 699 ms | 617 ms | 613 ms | 632 ms | 692 ms |
| Avg\_response\_time | 1078542 µs | 590276 µs | 156914 µs | 81673 µs | 78586 µs | 91934 µs |
| Avg\_turnaround\_time | 1084804 µs | 593950 µs | 157838 µs | 82151 µs | 78887 µs | 92090 µs |

**Note** : By observation in the code, we found that running time is calculated in milli seconds, But in the printf() statement in external\_cal, parallel\_cal, vector\_multiply it is printed as micro seconds. So, we mentioned running time in milli seconds.

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

When we analysed the benchmark results of our scheduling algorithms we identified many patterns.

Some patterns that were noticed:

In case of parallel cal.c and external cal.c, the implementation we did took more time when compared to the linux pthread library. We thought that this might be because of the main thread implementation we did. That is, when we considered the average time of execution, parallel cal.c and external cal.c took more time than others. Whenever the main thraed is scheduled it was waiting for the join to retuen, it does nothing during that wait time. So, all such timeslices passby idle with no work being done. Whereas, in case of linux pthread library it may have some internal mechanisms to avoid this idleness or time wastage.

In case of our implementations RR (Round Robin) took more time overall as compared to MLFQ implementation. This is because there is only single level in RR Scheduler whereas in MLFQ there are Multiple priority levels. We degrade the priority when the job uses entire timeslice. For Example, if the main function takes the entire time slice then the Scheduler degrades its priority during its join so that the main function would be scheduled later. By that time other threads may have exited and the join takes less time and they would be scheduled. So, other jobs also get sometime to scheduled. On the other side, user-level threads end up staying in the priority because they are blocked just before the time slice. So, these scenarios are helping in the run-time for MLFQ.

In case of vector\_multiply.c the implementation we did is taking less time than the pthread library. This may be because vecor\_multiply being a simple logic while the complicated mechanisms used in pthread library may add overheads. The timeslice values which are used by pthread library may also have impact. The overhead will be more if the time-slices values are less and vice-versa.