Laksh Kotian : lk596
Arnab Halder : ash186

## Implementation detail.

**Important structure and variable:**

Tcb
{
  mypthread_t* id
  ucontext_t* context
  thread_status status
  void** return_val: stores value returned by thread after it returns
  queue* join_queue; : stores queue of threads that are waiting to join on this one
  int quant_count; : number of quantums for which this thread has executed
  time_t entry_time; timestamp of the most recent execution of this thread completed
  int mlfq_lvl; level of mlfq the thread is currently in
  bool is_blocking; flag that is set if thread is holding a lock

}

Mutex
{
  Int lock;
  mypthread_t* owner_id;
  queue* blocked_queue; // queue of threads waiting on this lock
}

Current_tcb : holds tcb of  thread that is currently executing.

Thread can have one of the following statuses : READY, RUNNING, BLOCKED, FINISHED

Before scheduler is called sched_event is set which can have the following values:
0: RUN_NEXT ,2:  REMOVE, 3: BLOCK

**Functions:**

Mypthread_create:

Allocates context for new thread using getcontext. The function passed as an argument is wrapped in the return_wrapper function that calls the pthread_exit function with the return value of the function. This wrapped function is attached to the context.

Initializes the a tcb for the new thread and stores the newly created context in it.

The tcb status is set to READY and it is pushed in ready_queue

In case this is the first time the thread is executing we initialize a context and tcb for main and save it as current_tcb.

Sched_event is set to 0 and scheduler is called

Mypthread_join:

The current thread is pushed into the join_queue of the thread passed in as the argument

The current thread then needs to be blocked so sched_event is set to 3 and scheduler is called.

mypthread_mutex_lock

If mutex lock is not set , the current thread is assigned as the owner of the lock and the lock is set
If the mutex lock is set , current thread is put into the block queue of the lock. The current thread then needs to be blocked so sched_event is set to 3 and scheduler is called.

mypthread_mutex_unlock
If the thread calling unlock is not the owner , this function returns.
If it is the owner , the first thread in the block queue of the mutex is put into the readyqueue.
In case the block queue is empty, the lock is unset

Scheduler

- Handling sched_event 0 (RUN_NEXT):
    - The current_tcb is set to READY state from RUNNING and pushed into queue.
    - The thread at top of queue is popped and set to running state and swapcontext is done between these two threads.
- Handling sched_event 2 (REMOVE):
    - This is called when current thread is done executing.
    - If there are any threads waiting to join on this thread we pop all of them and add them to ready queue.

    - The current thread structures are then freed.
- Handling sched event 3(BLOCK):
    - This is called when a thread is blocked either due to mutex_lock or thread_join.
    - The current thread is set to blocked state and the thread at top of queue is set as current tcb.
- In all three cases the scheduler makes a call to set_timer function that invokes the scheduler after a time quantum.

**SJF details**

The scheduling decisions in SJF are done based on two factors
Quant_count : Every time the scheduler is invoked by the timer , the current thread has run for one quantum of time so the quant_count in its tcb is incremented by 1.
Entry_time: At the start of the thread and every time when the thread gets scheduled the entry_time gets updated to the current timestamp.

Whenever a thread is pushed into the ready queue we ensure that it takes its place in th queue in such a way that the threads are arranged in ascending order of quant_count. Ties are broken by checking which entry time is older i.e. the thread which has been waiting longer gets priority.

Therefore whenever we pop from the queue we ensure that the proper decision is taken based on these parameter.

**MLFQ details**

Our implementation takes the following parameters.
NUM_MLFQ Number of queues in the MLFQ (Default 3)
PROMOTION_DELAY  Number of quants after which promotion will be attempted (Default 10)
TIME_QUANTUM quantum for the first queue (Default 500 ms)

All levels of the queue follow the same strategy as SJF for their insert i.e. they take into account quant_count and entry time and the queue is always kept in sorted order
The first NUM_MLFQ-1 follow non-preemptive RR for scheduling and the time quantum for a level is given by level_number *  TIME_QUANTUM.
The last level is a preemptive FCFS queue in  which threads will run as long as a higher priority threads comes along.

<u>promotion/demotion</u>
After a thread is executed for its assigned quantum, it gets demoted to the next level of the queue.
If a thread is holding a lock , it does not get demoted till it releases the lock . This is to ensure it gets high priority so that other threads can access the locked resource.

After PROMOTION_DELAY number of cycles an update level function is run that checks if any of the threads has been waiting for more than PROMOTION_DELAY * TIME_QUANTUM duration. If it finds such threads they are moved up a level.

**New Benchmark 1: prime_sum.c, void prime_sum(void* arg)**

This .C file is used to send thread_num numbers of random integers between the Lower and Upper integer range defined in the file to the function prime_sum that is handled by the threads. We used this protocol due to the extensive nature of computing required by the threads and the fact that different threads will run for a different amount of time, where each thread finds the prime number in the range from 0 to counter[i] and stores the sum of prime numbers in that range in the variable res.

**New Benchmark 2: matrix_multiplication, void matrix_multiplication(void* arg)**

This .C file is used to calculate the sum of all elements in the product matrix. The product matrix is the result of multiplication of two NxN matrices (r and s). This protocol was used as matrix multiplication takes O(N^3) time to compute the multiplication of 2 matrices, due to which the computations are large as N grows larger.

**Pthread vs MyPThread:**

We ran tests using the linux implementation of the PThread library and compared it with our own threads by using the existing benchmarks provided and adding 2 new benchmarks. Of the 2 new benchmarks added, one benchmark (prime_sum) focused not only on the extensiveness of the computation but also stressed on the fact that different threads will require different amounts of time to run their critical section, while the second benchmark was mainly used to perform computation extensive tasks.

The readings are high when compared to the PThread library.

The tables below show the reading of the benchmarking harness used based on different time quantums, number of threads, MLFQ scheduling and the actual readings as observed from the PThread library.

| external_cal | Time Quantum (ms) | | | | |
|---|---|---|---|---|---|
| No of Thread | 1,000.00 | 500.00 | 128.00 | 64.00 | Pthread |
| 2 | 5,301.00 | 10,227.00 | 5,021.00 | 5,027.00 | 1,220.00 |
| 4 | 5,409.00 | 9,957.00 | 5,013.00 | 5,033.00 | 1,377.00 |
| 6 | 5,041.00 | 10,040.00 | 5,008.00 | 5,155.00 | 1,524.00 |
| 8 | 5,014.00 | 7,508.00 | 5,135.00 | 5,039.00 | 1,401.00 |
| 10 | 5,012.00 | 10,711.00 | 5,194.00 | 5,073.00 | 1,570.00 |
| 50 | 5,113.00 | 10,968.00 | 5,070.00 | 5,043.00 | 1,593.00 |
| 100 | 5,020.00 | 8,150.00 | 5,010.00 | 5,049.00 | 1,567.00 |

| parallel_cal | Time Quantum (ms) | | | | |
|---|---|---|---|---|---|
| No of Thread | 1,000.00 | 500.00 | 128.00 | 64.00 | Pthread |
| 4 | 1,928.00 | 1,908.00 | 1,938.00 | 1,919.00 | 509.00 |

| | | | | | |
|---|---|---|---|---|---|
| 6 | 1,920.00 | 1,929.00 | 1,915.00 | 1,910.00 | 360.00 |
| 8 | 1,932.00 | 1,919.00 | 1,917.00 | 1,916.00 | 273.00 |
| 10 | 1,948.00 | 1,911.00 | 1,913.00 | 1,923.00 | 246.00 |
| 12 | 1,915.00 | 1,909.00 | 1,917.00 | 1,915.00 | 311.00 |
| 50 | 1,937.00 | 1,916.00 | 1,912.00 | 1,921.00 | 197.00 |
| 100 | 1,938.00 | 1,956.00 | 1,921.00 | 1,926.00 | 199.00 |

| **vector_multiply** | Time Quantum (MS) | | | | |
|---|---|---|---|---|---|
| No of Thread | 1,000.00 | 500.00 | 128.00 | 64.00 | Pthread |
| 2 | 1,341.00 | 1,765.00 | 3,665.00 | 3,618.00 | 236.00 |
| 4 | 1,340.00 | 1,349.00 | 3,643.00 | 3,053.00 | 231.00 |
| 6 | 1,344.00 | 1,341.00 | 3,645.00 | 3,770.00 | 213.00 |
| 8 | 1,353.00 | 1,352.00 | 3,645.00 | 3,628.00 | 213.00 |
| 10 | 1,366.00 | 1,353.00 | 3,630.00 | 3,638.00 | 250.00 |

| | | | | | |
|---|---|---|---|---|---|
| 50 | 1,521.00 | 1,521.00 | 3,660.00 | 3,676.00 | 284.00 |
| 100 | 1,530.00 | 1,469.00 | 3,701.00 | 3,709.00 | 301.00 |

| **prime_sum** | Time Quantum (MS) | | | | |
|---|---|---|---|---|---|
| No of Thread | 1,000.00 | 500.00 | 128.00 | 64.00 | Pthread |
| 2 | 641.00 | 640.00 | 638.00 | 656.00 | 518.00 |
| 4 | 2,549.00 | 2,547.00 | 2,546.00 | 2,595.00 | 1,637.00 |
| 6 | 5,448.00 | 5,118.00 | 5,117.00 | 5,165.00 | 1,629.00 |
| 8 | 7,883.00 | 7,938.00 | 7,830.00 | 7,915.00 | 1,629.00 |
| 10 | 8,606.00 | 8,611.00 | 8,713.00 | 8,718.00 | 1,743.00 |
| 50 | 34,856.00 | 36,367.00 | 34,869.00 | 35,245.00 | 3,901.00 |
| 100 | 61,553.00 | 61,386.00 | 61,110.00 | 61,417.00 | 6,767.00 |

| **matrix_multiplication** | Time Quantum (MS) | |
|---|---|---|

| No of Thread | 1,000.00 | 500.00 | 128.00 | 64.00 | Pthread |
|---|---|---|---|---|---|
| 2 | 4,304.00 | 4,391.00 | 4,349.00 | 4,487.00 | 2,093.00 |
| 4 | 4,465.00 | 4,384.00 | 4,420.00 | 4,397.00 | 1,155.00 |
| 6 | 4,314.00 | 4,301.00 | 4,592.00 | 4,742.00 | 726.00 |
| 8 | 4,441.00 | 4,832.00 | 4,407.00 | 4,468.00 | 550.00 |
| 10 | 4,338.00 | 4,260.00 | 4,357.00 | 4,438.00 | 464.00 |
| 50 | 4,481.00 | 4,281.00 | 4,445.00 | 4,357.00 | 482.00 |
| 100 | 4,318.00 | 4,217.00 | 4,345.00 | 4,337.00 | 452.00 |