PAVITRA PATEL - php51
KUSH PATEL - kp1085
CS 416 – Report Project 2
rlab2.rutgers.edu

1. Detailed logic of how you implemented each API function and the scheduler logic.

```
static void signal_handle(int signum);
```

- This function is registered with sigaction. Every time there is a timer interrupt, this method is executed. This method gets the context of the current running thread and swaps the control to the scheduler.

```
int worker_create(worker_t * thread, pthread_attr_t * attr,
                  void *(*function)(void*), void * arg);
```

- This function initializes a new thread control block for a new user thread that needs to be created. The status of the new thread created is set to THREAD_CREATED. Using getcontext, the context of the new thread is saved. makecontext is called on this new thread context to determine which function will be executed. The new thread is added to the scheduler run queue. We record the time when the thread is created so that it can be used to calculate average turnaround and response time.
- When a new thread is created for the first time:
  - a main thread control block is initialized, and its status is set to THREAD_RUNNING.
  - There is a global variable of type ucontext_t that is used to execute the schedule function. This is achieved by using getcontext and makecontext.
  - Here, we initialize different queues for terminated, blocked_join, and scheduler_runqueue threads.
  - The timer is set and started. The timer expires at every 50,000 ms by default.

```
int worker_yield();
```

- This function uses a global variable whose value can be 0 or 1. Here, we set the value to 1, which indicates that current running thread wants to yield. The status of the current running thread is set to THREAD_READY, the context is saved using getcontext, and we transfer the control to scheduler using swapcontext.

```
void worker_exit(void *value_ptr);
```

- The status of the current running thread is set to THREAD_EXITED. There is a global queue that is used to stored threads which are blocked for join reason. In other words, if there is any other thread waiting for this thread to exit, we notify that thread by changing it's status to THREAD_READY that it can proceed for execution. The current running thread is added to terminated threads queue. We record the turnaround end time for this thread and update the average turnaround time accordingly. We free up the stack used by the exited thread and transfer the control to scheduler using setcontext.

```
int worker_join(worker_t thread, void **value_ptr);
```

- This function blocks the current running thread and adds it to the blocked_join queue. In the parameter, a thread is specified which the current running thread will wait for. As soon as the parameter passed thread exits, the current running thread will be freed. We also check if the thread that we were waiting for has already exited in the terminated threads queue. Finally, we transfer the control to the scheduler.

```
int worker_mutex_init(worker_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);
```

- This method initializes a mutex structure. It sets the internal state variables of the mutex to their initial values. Here we set the holdingTCB value to Null. Which we are indicating the no threads currently holding in mutex. Used an atomic flag clear which shows a muted is unlocked. Finally we are initializing the waiting queue using the Queue data structure of ini_runqueue.

```
int worker_mutex_lock(worker_mutex_t *mutex);
```

- This method is used to acquire the mutex lock. TO check if the flag status is lock or unlock, implemented the atomic glass test status. If there is current thread which is enqueued in waiting queue and we set the status to THREAD_MUTEX_BLOCK. In other words, it's waiting for the mutex. Then we use the swap context to scheduler to take over of the control. On the other hand, if the mutex lock available then we set the holdingTCB to itself which indicates that it has got the lock.

```
int worker_mutex_unlock(worker_mutex_t *mutex);
```

- This method is used to acquire the mutex unlock. We clear the status or unlock using the atomic flag clear. More over, we are also checking in the holdingTCB that is no longer holds the lock. Also, checking if there is any threads are waiting in the waiting queue, and set the their status to THREAD_READY, so it can run.

```
int worker_mutex_destroy(worker_mutex_t *mutex);
```

- This method is use to clean up and destroy the mutex. We first set the holdingTCB to NULL, and clear the atomic flag to make sure of the it's in unlocked state. Then we free the memory we use for the waiting queue used by the mutex, to avoid any memory leaks.

```
static void schedule();
```

- This function determines the scheduling policy. If the scheduling policy is PSJF then the method with PSJF logic is executed. If the scheduling policy is MLFQ, then the method with MLFQ logic is executed.

```
static void sched_psjf();
```

- Here the status of the current running thread is updated to THREAD_READY and is added to the tail of the scheduler run queue. The scheduler run queue is sorted based on number of times every thread is scheduled. We dequeue the head of the scheduler run queue and set it’s status to THERAD_RUNNING. Before we set the status, we check if the dequeued thread

status was THREAD_CREATED or not. If it was thread created, we record the response time for that thread and update the average response time accordingly. We update the global variable that keeps track of total context switches here. We also update the number of times the dequeued thread is scheduled. Finally, we set the context of the dequeued thread which will now be current running thread.

```
static void sched_mlfq();
```

- We get the priority of the current running thread and check if the thread was yielded or not. If the thread was yielded, then we keep the thread in the same priority queue. If the thread was not yielded we enqueue the thread to the lower priority queue indicating that it used up its time slice. The priority is stored in thread control block and will be updated. There are 4 levels of priority queues by default. If the current thread is running, we set it's status to ready here. When all threads reach the lowest priority queue they will be scheduled in Round Robin fashion. We iterate over 4 queues to find highest priority queue with threads. Now, the `static void sched_rr(int queue_priority)` method is called which will dequeue a thread from head of the priority queue. Before we set the status to THREAD_RUNNING, we check if the dequeued thread status was THREAD_CREATED or not. If it was thread created, we record the response time for that thread and update the average response time accordingly. We update the global variable that keeps track of total context switches here and then we set the context of the dequeued thread which will now be current running thread.

2. Benchmark results of your thread library with different configurations of worker thread number.

Ans:
Results for PSJF:
./parallel_cal 6
****
Total run time: 1594 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 78
Average turnaround time 1543.166667
Average response time 126.833333
****

./parallel_cal 50
****
Total run time: 1542 micro-seconds
verified sum is: 83842816

```
Total sum is: 83842816
Total context switches 171
Average turnaround time 1025.800000
Average response time  618.460000
**************************

./parallel_cal 100
**************************
Total run time: 1547 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 265
Average turnaround time 827.290000
Average response time  624.860000
**************************


./external_cal 6
**************************
Total run time: 341 micro-seconds
verified sum is: -833984588
Total sum is: -833984588
Total context switches 22
Average turnaround time 309.000000
Average response time  143.666667
**************************

./external_cal 50
**************************
Total run time: 313 micro-seconds
verified sum is: -833984588
Total sum is: -833984588
Total context switches 110
Average turnaround time 262.400000
Average response time  244.540000
**************************

./external_cal 100
**************************
Total run time: 296 micro-seconds
verified sum is: -833984588
Total sum is: -833984588
Total context switches 215
Average turnaround time 201.670000
Average response time  193.130000
**************************
```

./vector_multiply 6
****
Total run time: 16 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 12
Average turnaround time 8.666667
Average response time 6.166667
****

./vector_multiply 50
****
Total run time: 23 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 100
Average turnaround time 11.780000
Average response time 11.320000
****

./vector_multiply 100
****
Total run time: 23 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 200
Average turnaround time 11.270000
Average response time 11.050000
****

Records for MLFQ:
./external_cal 6
****
Total run time: 403 micro-seconds
verified sum is: -1356849840
Total sum is: -1356849840
Total context switches 22
Average turnaround time 354.833333
Average response time 150.500000
****

./external_cal 50
****
Total run time: 346 micro-seconds
verified sum is: -1356849840

Total sum is: -1356849840
Total context switches 114
Average turnaround time 284.220000
Average response time 263.500000
****

./vector_multiply 6
****
Total run time: 15 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 12
Average turnaround time 8.500000
Average response time 6.000000
****

./vector_multiply 50
****
Total run time: 23 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 100
Average turnaround time 11.520000
Average response time 11.060000
****

./parallel_cal 6
****
Total run time: 1547 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 76
Average turnaround time 1413.166667
Average response time 127.666667
****

./parallel_cal 50
****
Total run time: 1546 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 164
Average turnaround time 1022.920000
Average response time 616.940000
****

3. A short analysis of your worker thread library's benchmark results and comparison with the default Linux pthread library.

- The benchmark results indicate that the default Linux pthread library outperforms our worker thread library's PSJF and MLFQ implementations in terms of total run time, context switches, average turnaround time, and average response time. This suggests that there may be room for optimization and improvement in the performance of our worker thread library. It's essential to analyze and profile our library's code to identify areas where optimizations can be made to reduce overhead and improve efficiency.

4. Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

- During our project, we actively collaborated with our Teaching Assistants (TAs) to seek guidance and clarify doubts. We maintained a consistent presence on Piazza, utilizing it as a platform to engage in discussions, ask questions, and benefit from collective knowledge. Additionally, we extensively referenced our course materials and textbooks for a comprehensive understanding of MLFQ and PSJF scheduling algorithms. To gain further insights, we watched informative YouTube videos that provided in-depth explanations and practical insights related to the project's concepts and implementation. These resources played a crucial role in our learning and project development process.