Names: Divit Shetty & Vignesh Venkat
netID: dps190 & vvv11

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

Helper Functions:
setup_scheduler_context: This function initializes the scheduler's context. We made sure that the memory was allocated for the scheduler's stack, we set up the scheduler context, make the main thread, set up the timer, and switch to the scheduler's context to begin scheduling.

Start_timer: the function starts a timer using a system call, where we have set the intervals to microseconds and seconds. We use it to trigger context switches in the thread library.

Stop_timer: this function stops the running timer, and we use it to disable context-switching signals. Essentially the function is used to pause scheduling when the thread is inside the critical section.

Timer_setup: This function configures the signal handler, initializes the timer with intervals for scheduling, and we call it during the initialization of the scheduler.

Enqueue: Adds a TCB to the rear of the queue. We have logic to handle cases where the queue is empty, contains elements, or is empty.

Dequeue: Removes the thread from the front of the queue and returns it. If the queue is empty, we update the queue head and rear pointers.

Dequeue_psjf: This function finds and dequeue a thread with the shortest elapsed thread in any queue. We iterate through the queue to find the thread with the least time and then we dequeue it.

Dequeue_mlfq: removes a thread from the highest available priority level in the MLFQ. We start from the highest priority and go downwards until we find a viable thread.

Dequeue_blocked: This function dequeues a thread from the blocked_queue and moves it to the highest priority level in the MLFQ. We reset the unblocked thread's priority to prevent starvation.

Print_Queue: Just prints the contents of a specified queue, used primarily for debugging.

Refresh_MLFQ: Resets the priority of all threads in the MLFQ to the highest priority, used to avoid starvation. Ensures fairness by moving all threads from lower priority back to highest priority.

Ring: The ring function is responsible for handling context switches between threads and preventing starvation in our MLFQ scheduling system. Our function checks if there is a current thread. In the case where the MLFQ system is enabled, the function addresses thread starvation. It increments the priority of the current thread unless it is already at the highest priority level. We ensure that lower-priority threads are not indefinitely starved, by tracking the function elapsed time. Once the elapsed time exceeds a defined aging threshold, the Refresh_mlfq function is called, resetting elapsed and rebalancing the MLFQ to address potential starvation. After handling starvation, the function updates our context switch counter. Finally, the function performs a context switch by saving the current thread's state and passing control back to the scheduler via swapcontext. If this context switch fails, an error message is printed, and the program terminates.

Find_thread_by_id: Find thread by id is used when we need to search for a thread with a specific ID in a given queue.

Find_thread_by_id_all_queues: This function searches all the queues in the MLFQ, the blocked queue, and the finished queue for a thread with a specific id. We return the matching thread's TCB or NULL if the thread is not found.

theQueueisempty: checks if all queues in the MLFQ and PSJF are empty.

Part I:

1.1 Thread creation:

We constructed a TCB that consisted of: (1) Thread Identifier (2) Priority Status for the Scheduler (3) Thread Status, that was defined by an enumerator that had four values: READY, SCHEDULED, BLOCKED, FINISHED (4) A context variable with type ucontext to save the registers information (5) A stack variable to help setting the uc_stack.ss_sp (6) A pointer to hold the return value if need be (7) An integer to hold the number of seconds that has elapsed, utilized for the scheduler (8) A list of next TCB's, that was utilized for the run queue and scheduler threads. Since the scheduler context has to be initialized the first time a worker thread is created, we created a function called setup_scheduler_context() that instantiates all of the contexts for the scheduler and we set our Boolean to True so that the scheduler does not get initialized again. We then allocate some memory for our new thread and make sure that there was no error with the memory allocation. We then get the context of the new thread and once again set up some debugging to ensure no errors in the code. We then set up the context's stack using the sample-code instructions! we then make the context to execute the function with the given argument. We then add the new thread to the run queue and initialize some of our other parameters for the control block (i.e. status, elapsed time, and the waiting time). Based on the macro value, the thread is enqueued to the respective scheduler. We also defined our global variables to be next_thread_id as a counter for unique thread IDs. A runqueue_head to define the current running thread and a scheduler_context to define the scheduler context. We then defined scheduler_initialized to help setup scheduler context and mlfq_queues as an array of run queues for MLFQ. Lastly a time quantum to define our time quanta for MLFQ levels (lowest to highest i.e. 8 =HIGHPRIO).

1.2 Thread Yield:

We check the context of the current thread to ensure there is a viable context for us to operate upon and check if status is equal to FINISHED. If there is viable context and not equal to FINISHED, then we set the status to READY and swap the context to the scheduler context. Based on the macro value, the thread is enqueued to the respective scheduler.

1.3 Thread Exit:

First, we stop tracking the clock such that the queue time and the start time are not including the processes in the exit routine. We then set the status of the exiting thread to be FINISHED and if the value_ptr is not NULL, we set the value_ptr to the return value in the current thread. We then increment the number of completed threads, calculate the average response time and average turnaround time. We then de-allocate the stack memory allocated during thread creation. Lastly, we yield control to the scheduler context.

1.4 Thread Join:

In our worker_join() function, we find thread by the worker_id in a helper function. While the thread that we found is not finished, the current thread keeps yielding to the scheduler. We then check if there is no NULL value for the thread, as in our function did in fact find the thread. We then save the return value in a retval. Next, we free the stack memory. Lastly, we free the stack of the thread we were waiting upon to finish execution.

1.5 Thread Synchronization:
In our mutex structure, we defined: (1) initialize (2) locked, to define if it was in fact locked or not (3) owner (4) blocked_list (5) blocked_count and (6) max_blocked. We initialize the lock in the worker_mutex_init() function. In the worker_mutex_lock() function we check whether or not the lock is instantiated. We then if the user is attempting to lock an uninitialized mutex. Then we use the test_and_set atomic instruction to keep the threads spinning while the mutex is locked. If mutex is locked, then the status of the current thread gets set to blocked. We then push current thread into block list and increase the number of context switches before swapping context back to the scheduler. In the worker_mutex_unlock, we once again make sure that the user is not calling our unlock function in an inappropriate manner. We then dequeue from our blocked list, release the lock, and set the owner to null. In our helper function, dequeue_block(), we also unblock a thread and make sure to reset it's priority before enqueuing it into the run queue again. Lastly, when we coded the function:
worker_mutex_destroy(), we once again check for all the edge cases a user may try to incorrectly use. We then initialize the mutex back to uninstantiated, set the lock state to default, and set the owner to NULL. We then free all the pointers and the lock itself.

Part II:
2.1 PSJF:
If the user choses to use PSJF, then in our sched() function we call out sched_psjd() function which only sets the current thread to the highest priority thread in the run queue. We check if our current thread is NULL, and if it is not, then we initialize the start time and set our current thread to SCHEDULED (i.e. running in our code). We then increment the number of context switches. We then start the timer for the current thread's execution and swap the context from the scheduler to the current thread. We then incremented the elapsed time after the context switch had taken place. Once execution is done, we check if the current thread is NULL, BLOCKED, or FINISHED. If it is none of these states, then we set it to READY, enqueue it back into the run queue. If it is BLOCKED or FINISHED, we enqueue it into a blocked queue or a finished queue.

2.2 MLFQ:
If the user choses to use an MLFQ scheduler, then in our sched() function we call the sched_mlfq() function. We have a specific MLFQ queue that we utilize throughout our code if the MLFQ was invoked by the user. We dequeue from that MLFQ run queue, and then check if the current thread is NULL. If it isn't then, we initialize time slice based on the priority of the current thread. When the thread has exceeded its time slice and if the thread isn't at the lowest priority level, we demote the priority of the thread. We then reset the elapsed time after demoting a thread. We check if our current thread is NULL, and if it is not, then we initialize the start time and set our current thread to SCHEDULED (i.e. running in our code). We then increment the number of context switches. We then start the timer for the current thread's execution and swap the context from the scheduler to the current thread. We then incremented the elapsed time after the context switch had taken place. Once execution is done, we check if the current thread is NULL, BLOCKED, or FINISHED. If it is none of these states, then we set it to READY, enqueue it back into the run queue. If it is BLOCKED or FINISHED, we enqueue it into a blocked queue or a finished queue.

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

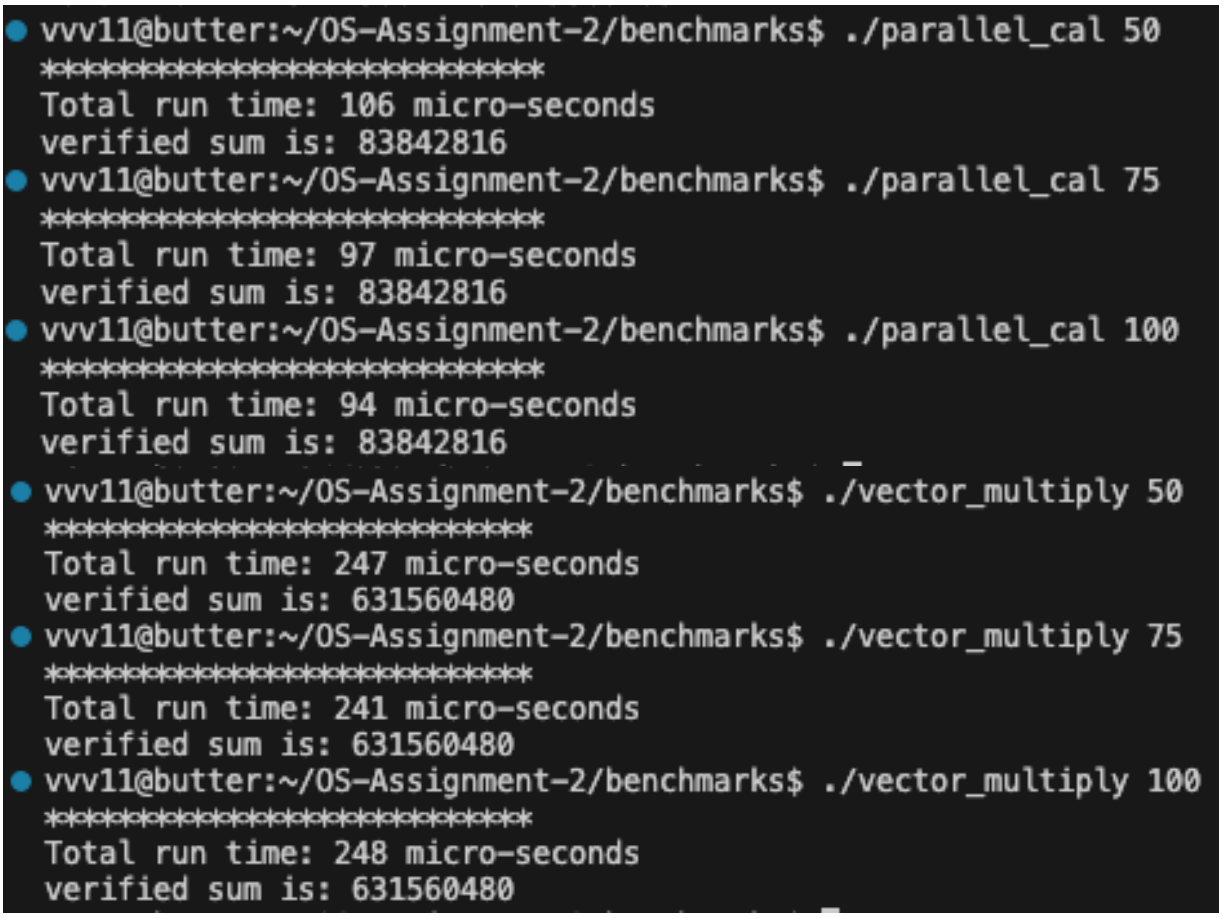
```
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./external_cal 50
**************************
Total run time: 648 micro-seconds
Total sum is: -1906762149
Total context switches 408
Average turnaround time 0.120705
Average response time  0.016106
**************************
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./external_cal 75
**************************
Total run time: 636 micro-seconds
Total sum is: -1906762149
Total context switches 385
Average turnaround time 0.206266
Average response time  0.143882
**************************
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./external_cal 100
**************************
Total run time: 638 micro-seconds
Total sum is: -1906762149
Total context switches 400
Average turnaround time 0.126685
Average response time  0.076428
**************************
```

```
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./vector_multiply 50
**************************
Total run time: 45 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 54
Average turnaround time 0.021661
Average response time  0.020758
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./vector_multiply 75
**************************
Total run time: 46 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 79
Average turnaround time 0.022219
Average response time  0.021646
**************************
```

```
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./vector_multiply 100
**************************
Total run time: 41 micro-seconds
verified sum is: 631560480
Total sum is: 631560480
Total context switches 104
Average turnaround time 0.019704
Average response time  0.019321
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./parallel_cal 50
**************************
Total run time: 2993 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 788
Average turnaround time 2.756290
Average response time  0.398018
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./parallel_cal 75
**************************
Total run time: 3002 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 817
Average turnaround time 2.736855
Average response time  0.603930
**************************
**************************
dps190@ilab3:~/Documents/OS-Assignment-2/benchmarks$ ./parallel_cal 100
**************************
Total run time: 2980 micro-seconds
verified sum is: 83842816
Total sum is: 83842816
Total context switches 774
Average turnaround time 2.447329
Average response time  0.804655
**************************
```

3. Comparison with pthreads

```
vvv11@butter:~/OS-Assignment-2/benchmarks$ ./external_cal 50
**************************
Total run time: 1357 micro-seconds
vvv11@butter:~/OS-Assignment-2/benchmarks$ ./external_cal 75
**************************
Total run time: 1384 micro-seconds
vvv11@butter:~/OS-Assignment-2/benchmarks$ ./external_cal 100
**************************
Total run time: 1306 micro-seconds
```



We get the same values as pthreads. In vector_multiply, we have achieved 41-46 ms whereas pthreads has achieved 241-247 ms. In external_cal pthreads achieves between a 1306-1384 ms run time and we have achieved 636-648 ms. However, in parallel_cal, pthread achieves 94-106 ms and we achieve 2980-3002 ms. We are not totally sure why this discrepancy is so high; however, we do get the same value as pthreads, which ensures our code functions within the given requirements.