Assignment 1: Actual User Threads
Surekha Gurung (sg1089)
Krishna Patel (kap335)
Baljit Kaur (bk404)

Pthreads

int my_pthread_create (…)
The my_pthread_create takes my_pthread_t pointer, an attr pointer which is set to null, function pointer of type void* and a void* argument. First, it checks if it is initialized or not. If not initialized, then it initializes a list of nodes, the run and promo queues, and creates a context. The create function returns 0 if successfully done.

int my_pthread_yield ()
In this function, no argument is passed. It first checks if the list is initialized or not. Then, the timer is set for all the threads created by the users. The action of all the threads is set to yielding. After, the scheduler is called to put the threads together in the priority queue.

void my_pthread_exit (void *value_ptr)
This function takes a void pointer argument and checks for initialization. If the initialization is done, the timer is stopped and the status is changed to exiting thread. Then, the scheduler is called and the threads are enqueued or dequeued in the priority queue.

int my_pthread_join (my_pthread_t thread, void **value_ptr)
The join function takes the thread and void** pointer as argument. First initialization is checked for the list. Then the thread is checked if it is dead or not, if not then threads are moved to the temp nodes and resumes the timer. If not dead then the thread's status is set to joining and then temp list is incremented.

Mutexes

int my_pthread_mutex_init (…)
This function takes argument as pointer to a mutex that is created by the user. In addition, it also takes argument as pointer to mutexattr that is set to null. First it checks for the mutex. If not null, the mutex lock is initialized to the threads and the flag is kept for the track of the locks.

int my_pthread_mutex_lock (my_pthread_mutex_t *mutex)
This function takes an argument as pointer to the mutex that is created by the users. Then after getting the mutex lock it checks the test and set condition and calls my_pthread_yield () for single thread and the yielding of threads are done.

int my_pthread_mutex_unlock (my_pthread_mutex_t *mutex)
This function takes an argument as pointer to the mutex that is created by the users. After the work is done the lock is unlocked and the flag is dropped to 0.

int my_pthread_mutex_destroy (my_pthread_mutex_t *mutex)
This function takes an argument as pointer to the mutex that is created by the users.
It checks if the flag is dropped to 0 or not. If it is then the lock is destroyed and returns 0.

Scheduler

The scheduler struct has a pointer that points to the current node that is running. The node struct contains a pointer to a specific thread, its context, the joining threads, priority, and other information about that particular thread. We also included a list of all threads as well as a list of threads waiting for others to finish. This makes it easier to implement join and exit. The scheduler struct also has a multilevel priority queue of 5 levels. Each thread runs a time slice based on its level. Each level gets to run its threads for 50ms(priority level+1) ms in our implementation. As advised in the project description, threads are moved down to the next level queue if it runs for the full time.

The scheduler function has conditionals based on which on the thread's actions (joining, Exiting, yielding). After going through each level and doing the joining, exiting and yielding it gives back the threads. It also keeps tracks of the time taken and the cycles used by each thread. To avoid starvation, we used a check() function that is called every hundredth call to the scheduler. The check() function increases the priority of threads that are in the lower levels based on when it was created.

Testing

We tested our assignment using the Benchmarks provided by the TA. Our results for parallelCal.c matched the TA's results. For instance, running ./parallelCal 4 gives

```
sum is: 1726371200
verified sum is: 1726371200
```

In addition, ./vectorMultiply 4 gives

```
Res is: 332833500
Verified res is: 332833500
```

Hence, it is safe to say that our implementation gives the correct results, and it seems to be working well.

We also calculated that the runtime of our implementation if we ran the following:
`[-bash-4.2$ ./parallelCal 13456]` is 0.76ms.
`[-bash-4.2$ ./vectorMultiply 13456` is 0.60ms.