Konrad Radecki - KR536
Jared Cheung - JLC598

API function and scheduler logic:
We have called many global variables in our thread library. Num_finished_processes is used to help calculate the average turnaround time and response time. Each head is used as the head of individual linked lists. Also lists for threads waiting to be added and waiting to be removed. Also a main context and scheduler context. We also keep track of the current process running and the current list being run. We have 4 lists for the 4 queues used by MLFQ.

worker_create:
Firstly we have to create the thread. Here we initialize everything in the TCB for the thread, make sure every list is connected to the thread, and every context has value. We also set up the stack.

worker_yield:
For yield we simply stop the timer, change the status of the current running process to be yield. Then swap context from current running process to scheduler context.

worker_exit:
First we stop the time, then we check if join has been called on this thread before, then we find the thread from the blocked list. If join has previously been called and we found the thread then it is inserted back into the thread pool, to be used again in the future. Lastly we change the current thread to wait for the join function. Then we change the status of the thread to terminated as it is exiting. Then this thread is moved to the exited thread queue and we set the context to scheduler context.

worker_join:
First we stop the time, then we will check if there has already been a thread that has exited. If so we will look through the lists of the exited threads searching for the desired thread. If we find it we will start the timer again, since the thread will not be context switched into the scheduler and we will be continuing with our current running thread. If we do not find the desired thread we will first change the status of the current running thread to blocked. It will also be removed from the job list and added to the blocked list. Next we will traverse through the job list to find the next waiting thread. Once we find that thread we will change its wait value, then we wait until the next waiting thread calls exit and is terminated. Finally the thread is freed from memory.

Thread Synchronization:

Worker_mutex_init:
Here we initialize mutex, mid works as the mutex identifier, incrementing the mutex count to make sure each mutex gets its own identifier, locked and pid to show status of the mutex and ready_waiting for processes waiting on the mutx.

Worker_mutex_lock:
First we make sure that the mutex passed is not null, than we check if the mutex is locked, we set the status of this mutex to the status of the current running thread than waiting for the mutex to be unlocked. Once the mutex is unlocked we record the process id of the current running thread than set that there are no threads waiting for the mutex.

Worker_mutex_unlock:
First we make sure that the mutex passed is not null, than we check if the mutex is locked. If the mutex is locked we check if the process ID of the current running process matches that of the mutex, if they do not match we return an error. If they match we unlock the mutex and check for any processes waiting on the mutex and label them blocked. Than we label the mutex such that processes are waiting on the mutex.

Worker_mutex_delete:
First we make sure that the mutex passed is not null,than we check if the mutex is locked, if locked we return an error message if not we continue. Than we check to make sure no threads are waiting on the mutex, if threads are waiting on the mutex we return an error message if not we continue. Once this is all complete we free the mutex.

Scheduler:
The context switch is set for 5 usec, meaning every 5usec scheduler is called again.

PSJF:
Only one list is used for this scheduler. Whenever the current running thread exits we schedule the next shortest thread, changing the status, change the set context to the new thread and restart the timer. If the current running thread has not yet been completed (meaning it has not yet exited) we increase the counter by 1, the counter being the amount of times the thread has been run. The thread is then inserted back into the head list based on its priority. Priority is given by the number of times the thread has run, (the more it runs the lower the priority).

MLFQ:
For MLFQ we have 4 lists (4 queues). If the current thread exits we schedule the next thread to be run based off of priority. If all the threads are on the same queue it simply is run in a round robin fashion. When the thread has a status of yield the priority is not changed. If the current thread is not blocked it is inserted back into the list based on priority. Finally a new thread is scheduled.

We also have helper functions for the timer, for finding certain nodes/threads and also for repositioning threads in MLFQ and for scheduling MLFQ and PSJF.

Benchmark results:

PSJF:

```
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./parallel_cal 10
**************************
Total run time: 3321 micro-seconds
Total sum is: 83842816
Total context switches 844
Average turnaround time 2023.700000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./parallel_cal 50
**************************
Total run time: 3268 micro-seconds
Total sum is: 83842816
Total context switches 1009
Average turnaround time 116.060000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./parallel_cal 100
**************************
Total run time: 3531 micro-seconds
Total sum is: 83842816
Total context switches 934
Average turnaround time 31.590000
Average response time  0.000000
**************************
```

```
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./vector_multiply 10
**************************
Total run time: 44 micro-seconds
Total sum is: 631560480
Total context switches 14
Average turnaround time 0.300000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./vector_multiply 50
**************************
Total run time: 155 micro-seconds
Total sum is: 631560480
Total context switches 55
Average turnaround time 0.100000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./vector_multiply 100
**************************
Total run time: 492 micro-seconds
Total sum is: 631560480
Total context switches 142
Average turnaround time 1.530000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$
```

```
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./external_cal 10
**************************
Total run time: 652 micro-seconds
Total sum is: 2022620141
Total context switches 150
Average turnaround time 239.700000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./external_cal 50
**************************
Total run time: 648 micro-seconds
Total sum is: 2022620141
Total context switches 192
Average turnaround time 60.180000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./external_cal 100
**************************
Total run time: 607 micro-seconds
Total sum is: 2022620141
Total context switches 233
Average turnaround time 24.320000
Average response time  0.000000
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$
```

MLFQ:

```
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./parallel_cal 10
**************************
Total run time: 3468 micro-seconds
Total sum is: 83842816
Total context switches 934
Average turnaround time 3159.800000
Average response time  167.971391
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./parallel_cal 50
**************************
Total run time: 5077 micro-seconds
Total sum is: 83842816
Total context switches 1153
Average turnaround time 671.900000
Average response time  33.594279
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./parallel_cal 100
**************************
Total run time: 3456 micro-seconds
Total sum is: 83842816
Total context switches 930
Average turnaround time 78.520000
Average response time  16.797140
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$
```

```
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./vector_multiply 10
**************************
Total run time: 43 micro-seconds
Total sum is: 631560480
Total context switches 13
Average turnaround time 1343771235971.399902
Average response time  167.971405
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./vector_multiply 50
**************************
Total run time: 199 micro-seconds
Total sum is: 631560480
Total context switches 60
Average turnaround time 1377365519494.719971
Average response time  33.594281
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$ ./vector_multiply 100
**************************
Total run time: 548 micro-seconds
Total sum is: 631560480
Total context switches 148
Average turnaround time 957437008762.920044
Average response time  16.797141
**************************
kr536@ilab4:~/Downloads/good_stuff/benchmarks$
```

For MLFQ external_cal started to crash the server, it was working for me before but I ran out of time of trying to figure out the problem, when I ran just 1 thread the terminal would keep repeating 11, it would do so until I terminated the program using ctrl+c

Analysis:
Kernel:
Parallel 100: 163 microseconds
Vector 100: 476 microseconds
external 100: 3490 microseconds

The kernel version of the scheduler worked a lot better than the scheduler we created. This is due to the programmers working on the kernel scheduler are experts and have been working on these kinds of programs for years with many more years of knowledge on their hands, making a scheduler that is much more complex than the one we could create.