Asst 1: Adventures in Scheduling

Overview (TL;DR)

We used the ucontext library to keep track of the different stages of progression for each thread when they were cycled in/out to run. We store all jobs not currently being run on a multi-level priority queue that adds/retrieves jobs following the 5 rules listed in the Multi-Level Feedback chapter in the textbook.

Structs && Data Structures

1. Scheduler (scheduler.h)
    a. Curr: tcb = currently running thread
    b. M_queue = multi level priority queue (defined later in section)
    c. Terminated: hashtable<thread id, tcb> = stores threads that have ended
    d. unlockJobs: hashtable<mutex id, min heap> = stores threads waiting on “mutex id” in a minheap
    e. joinJobs: hashtable<thread id, min heap> = stores threads waiting on “thread id” in a minheap
2. Mutex (my_pthread_t.h)
    a. Id: int = unique identifier for a mutex. We implement unique ids for our mutexes by simply assigning it the value of a global int, which is initially set to 0, and then increment the global int.
    b. Locked: int = set to 0 if unlocked and 1 if locked
3. TCB (Thread Control Block) (scheduler.h)
    a. Context: ucontext_t = context for the thread.
    b. Id = unique identifier for thread. Implemented similarly to unique ids for mutexes (see above)
    c. P_level: int = priority level (0 being highest)
    d. Retval: void* = pointer to threads return value
4. Hash table (data_structure.h)
    a. Functions like a standard hash table for the most part. We do not anticipate any hash table to be under a large load at any time so for simplicity’s sake we do not have a load factor by which we rebalance and our mod function is “id % table size”.
5. Min-heap (data_structure.h)
    a. We chose a min heap because the lower int integer value of a priority, the higher the priority (0 being the highest priority). Functions like a standard min heap.
6. Single Queue (data_structure.h)

    a. Located in data_structures.h, the single queue does exactly what one would think a queue does. It consists of nodes with prev and next pointers and holds one void * piece of data (typically a TCB *).
7. Multi-level Priority Queue (scheduler.h)
    a. Located in scheduler.h, the multi-level priority queue allows the user to define different parameters of the queue, including:
        i. The number of N levels (0 being the highest priority and N being the lowest)
        ii. The base time interval for the signal handler to fire
        iii. The time delta between each priority level to allow for longer time run time the lower the priority
    b. The “dequeue” method (get_next_job() in scheduler.c) tries to find a job to run from the highest priority to the lower
    c. The “enqueue” method (add_job() in scheduler.c) adds the job back into the respective TCB’s priority level (see TCB->p_level)
    d. To run the maintenance cycle, bump_old_jobs() in scheduler.c bumps all of the jobs on a certain X percentage of the lowest levels to the highest priority. This maintenance cycle is run every Y context switches. Both the values of X and Y are defined in the constants in my_pthread.c.

Mutexes

The definition of our mutex struct can be found in the “Struct && Data Structures” section. The “init” method sets locked to 0 and sets the id as described above. Since both of the fields are not dynamic in size, we did not have anything to do in our “my_pthread_mutex_destroy()” method. Psuedocode for our unlock and lock methods are as follows:

Unlock()
- Set lock flag in mutex to true
- Remove thread from “lockOwners”
- Check to see if someone is waiting on the mutex
- If someone is waiting on the mutex, swap them in next, else, continue running the current thread

Lock()
- If its unlocked:
    - Add thread to “lockOwners”
    - Mark lock flag in mutex to true
    - Continue running current thread
- If its locked:
    - Add thread to “unlockJobs”
    - Swap in next thread

- When this thread gets swapped in again, we know that the mutex must be unlocked. So the part of the function after when we swap ourselves out, acquires the mutex as per the steps laid out in the “if its unlocked” branch

Priority Inversion (EC)

To handle priority inversion, we followed the one variant of Priority Ceiling Protocol, Original Ceiling Priority Protocol (OCPP) shown here. The protocol dictates the following:

- Let LOW = a thread with a low priority
- Let MEDIUM = a thread with a medium priority
- Let HIGH = a thread with a high priority
- Let R = some shared resource amongst the 3 previously described threads
- If LOW acquires R and then HIGH waits on LOW to release R, LOW’s priority should temporarily be raised to the priority of HIGH. This way, MEDIUM could not indirectly preempt HIGH, by running before LOW.

To achieve this, we created a heap H that kept threads with the highest priorities at the top. H would be stored in a hashtable HT where the key would be the id of the mutex. This way, if a thread T had a mutex. We could quickly look up the heap containing of all threads which are waiting on said mutex. And at that point, we could get the highest priority of said threads, using a heap_peek(), and assign that to T’s priority (provided it was lower than T’s initial priority). The hash table mentioned is the field “unlockJobs” inside our scheduler struct. Implements of the heap can be found in “data_structures.c/h”.

Testing/Benchmarks

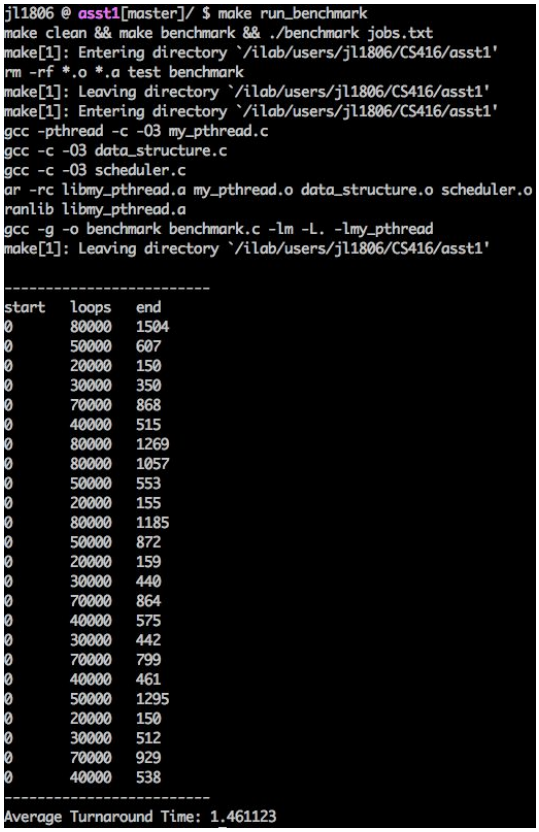
Tested on kill.cs.rutgers.edu:

- ```
  jl1806 @ Benchmark[free]/ $ ./vectorMultiply 6
  running time: 224 micro-seconds
  res is: 631560480
  verified res is: 631560480
  ```
- ```
  jl1806 @ Benchmark[free]/ $ ./parallelCal 6
  running time: 705 micro-seconds
  sum is: 83842816
  verified sum is: 83842816
  ```

Tested on python.cs.rutgers.edu:

- ```
  ┌─master|ooe4@python|~/CS416/asst1/Benchmark
  └./externalCal 6
  running time: 18629 micro-seconds
  sum is: -49745663
  verified sum is: -49745663
  ```
  ```

We also wrote a benchmarking program (benchmark.c) in order to measure the efficiency of our scheduler (see screenshot of it running below). The metric we chose to use for success was turnaround time. To measure this, we would spawn a number of jobs (as specified by a text file given as a command line argument) using our library and record their runtime. Then we would run each job without multithreading and record that runtime. With this data we could calculate the average turnaround time.



```
jl1806 @ asst1[master]/ $ make run_benchmark
make clean && make benchmark && ./benchmark jobs.txt
make[1]: Entering directory `/ilab/users/jl1806/CS416/asst1'
rm -rf *.o *.a test benchmark
make[1]: Leaving directory `/ilab/users/jl1806/CS416/asst1'
make[1]: Entering directory `/ilab/users/jl1806/CS416/asst1'
gcc -pthread -c -O3 my_pthread.c
gcc -c -O3 data_structure.c
gcc -c -O3 scheduler.c
ar -rc libmy_pthread.a my_pthread.o data_structure.o scheduler.o
ranlib libmy_pthread.a
gcc -g -o benchmark benchmark.c -lm -L. -lmy_pthread
make[1]: Leaving directory `/ilab/users/jl1806/CS416/asst1'

------------------------
start   loops   end
0       80000   1504
0       50000   607
0       20000   150
0       30000   350
0       70000   868
0       40000   515
0       80000   1269
0       80000   1057
0       50000   553
0       20000   155
0       80000   1185
0       50000   872
0       20000   159
0       30000   440
0       70000   864
0       40000   575
0       30000   442
0       70000   799
0       40000   461
0       50000   1295
0       20000   150
0       30000   512
0       70000   929
0       40000   538
------------------------
Average Turnaround Time: 1.461123
```