OS Assignment 1

Derek Mao mm2180, Quzhi Li ql88, Kevin Pei ksp98
Tested on ls.cs.rutgers.edu

Introduction: This is a program meant to simulate the pthread and mutex libraries. The functions allow you to create new threads, yield, exit, and join them, and create mutexes which you can lock and unlock.

Priority queues: There are three run queues in our program, corresponding to different priority levels. The first level runs for 25 ms round-robin style, the second level runs for 50 ms round-robin style, and the third and final level runs each thread until it's completed FIFO-style. Each queue contains threads from highest priority to lowest priority, and the thread with the highest priority out of all 3 queues is always run. If there is a tie, then the thread in the fastest queue is run, with level one being the fastest, level two being medium, and level three being the slowest. After a thread is finished running, all other threads in all run queues have their priorities increased by 1 to age them up and prevent starvation.

All threads are initially added to the first level queue with priority 100. If, after 25 ms a first-level thread hasn't finished running, then it is added to the second level run queue with a priority of 50. This means that new threads in the first level are run more often than older threads in the second level, with threads in the second level having to age up before having more priority than threads in the first level. If a thread in the second level runs for 50 ms without finishing, then it is added to the third level queue with a priority of 0. This means that third level threads must age up even more than second level queues before being run. This is to ensure that shorter threads are run much more often than longer threads.

In addition to the three levels of run queue, there are also two waiting queues. The first waiting queue is for threads that are waiting for mutex locks. When a thread is waiting for a lock, its priority does not change. When the lock is released, then all threads waiting for that lock are put back in the run queue in the first priority level. The second waiting queue is for threads that are waiting to join with a thread. When a thread exits, all threads joining on it are put back into the priority queue at the first level. Once again, their priorities are unchanged.

Our maintenance/aging cycle is entirely based on when a thread finishes. Whenever a thread is swapped, then all threads have their priorities increased by one. This is true regardless of whether the thread actually finished, it was moved to a different priority level, or it was put in a wait queue. This means that the frequency of priority increases is very dependent on which threads are running.

Functions: The functions we implemented were as follows:

my_pthread_create: It mallocs the space and creates a new pthread, initializes the factors in the pthread, then set its priority to 100, and add it to the end of the first running queue.

my_pthread_yield: our scheduler function, it allows another thread to run. Depending on why yield is called, it can move the currently running thread to next in the queue, to another priority level, or remove it from the queue entirely. The thread with the next highest priority is always run next.

my_pthread_exit: End the pthread that is called and save the return value the thread have.

My_pthread_join: Wait for the thread that it pass to exits, and it will pass back all the return values of the children while it is joining.

my_pthread_mutex_init: initialize the mutex lock, set the initial lock to open.

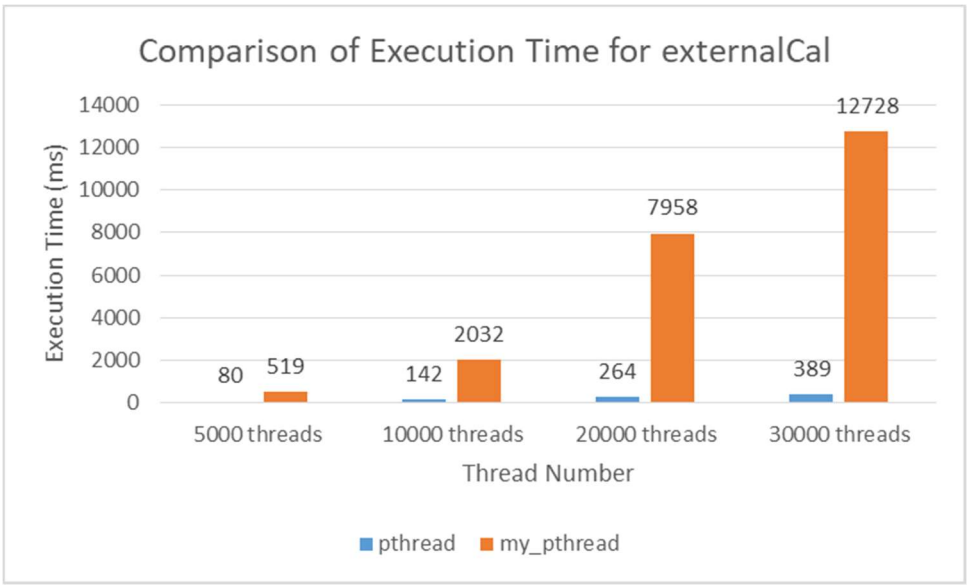
my_pthread_mutex_lock: if the mutex is not locked then lock it and acquire the mutex. Otherwise, move it to the wait queue.

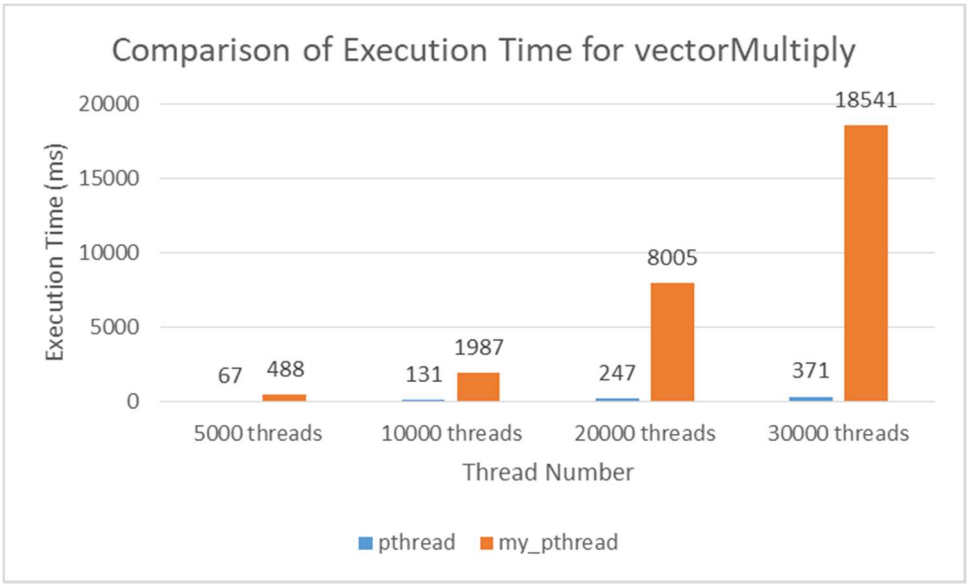
my_pthread_mutex_unlock: release the mutex lock. Check if the current thread is holding the lock, if yes, release the lock.

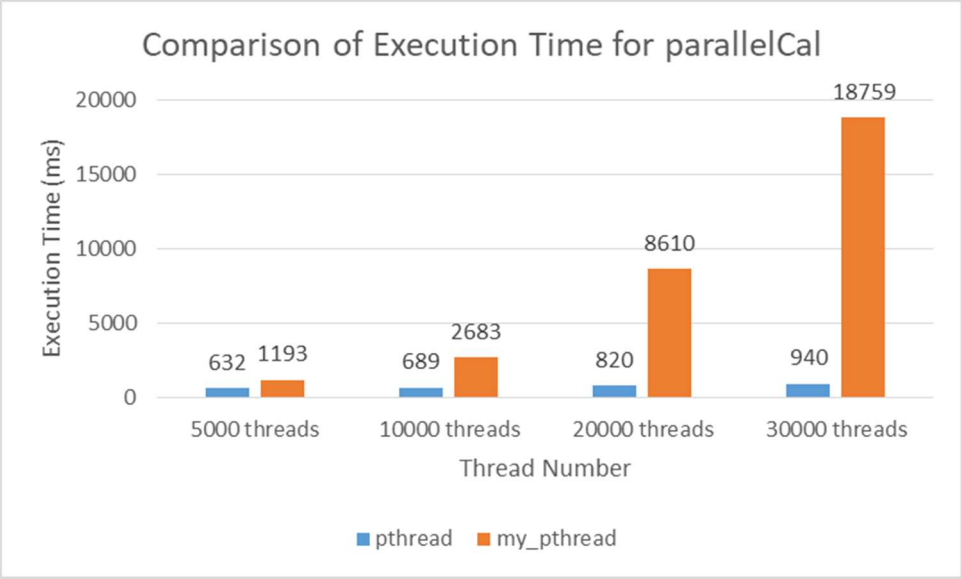
my_pthread_mutex_destroy: destroy the mutex if it is released by setting the mutex id to -1.

In addition, there are helper functions to help facilitate the above, such as functions to get the currently running thread, functions to add a thread to the end of a run or wait queue, and functions to get the highest priority thread. In particular, our yield function is simply a call to the swap_contexts() function, which itself is what handles all the context swaps and queue manipulation. We use two locks within our own code to prevent simultaneous modification of the queue. Our two locks are modifying_queue and scheduler_running. Modifying_queue is locked whenever the run or wait queues are being modified. Scheduler_running is locked whenever the swap_contexts function is running, to prevent it from running twice and interrupting its own operations. In other words, to prevent the scheduler from running inside the scheduler.

Speed: When comparing the speed of our pthread implementation to the standard c library implementation, our results were as follows:


Comparison of Execution Time for externalCal
Execution Time (ms)
14000
12000
10000
8000
6000
4000
2000
0
80
519
142
2032
264
7958
389
12728
5000 threads
10000 threads
20000 threads
30000 threads
Thread Number
pthread
my_pthread


Comparison of Execution Time for vectorMultiply
Execution Time (ms)
20000
15000
10000
5000
0
67
488
131
1987
247
8005
371
18541
5000 threads
10000 threads
20000 threads
30000 threads
Thread Number
pthread
my_pthread



As expected, the built-in pthread library is faster than my_pthread. It is interesting to note that the time taken for my_pthread seems to be geometric. ParallelCal also has some base cost independent of the number of threads, which seems to be the case for my_pthread too. However, while it dominates runtime for pthread, it is negligible for my_pthread because it represents such a small portion of the total runtime.