Pedro Torres (pt397) and Ivan Zheng (iz60)

Tested in: vi.cs.rutgers.edu, ilab1.cs.rutgers.edu

Project 2 Report

Implementation

Scheduler

Our project was structured around a central Scheduler who decided which thread got to run next. The central Scheduler has its own context as well, but stored separately from all the other threads that needed to be scheduled. Whenever a context switch occurred, it would always return to the scheduler to figure out who to schedule next.

PSJF

For the PSJF scheduler, we would schedule the jobs that did not use up its time cycles first, by allowing those jobs to remain at the front of the queue, whilst moving those who did use up its entire time cycle to the end of the queue. To do this we maintained a unique variable in the TCB called `elapsed`, which would be set if the thread did not relinquish the CPU of its own accord. If this variable has been set, the PSJF scheduler will then move that Node in the queue to the end. Then, the scheduler will move through the queue and find the item with the lowest runtime that is Runnable, and take that as the thread to run.

MLFQ

For the MLFQ scheduler, we should schedule the jobs based on the 5 rules defined in the textbook. The basis is a multiple queue system, where each priority level has a run queue. If a process is of higher priority than another process, then it runs before the other process. If two processes have the same priority, they run in a round robin. Once a job uses all of its time allotment at a given level, its priority is reduced by 1. After a certain amount of time, all jobs in the system move to the top most queue. We implement a 4 level MLFQ. Each time the scheduler checks if either the current process has used its allotment of time (one time quantum). If it has, it gets placed in a lower run queue (currPriority - 1). Afterwards, the scheduler finds a new thread to context switch to by going through the runqs. After S time, the jobs not in the topmost queue get moved up.

Queue

We made use of a circular linked list to mimic a queue operation. The circular linked list uses a custom struct node, which contains references to the TCB that the node represents and the next upcoming Node in the linked list. The Queue has multiple operations to make it modularized for ease of use across the entire program. These operations are `queue`, which inserts a given node at the end of a reference pointer to the linked list, `dequeue`, which removes a given Node from the referenced list, with the option to either free that dequeued Node or leave all its dynamic memory allocated for use in a different queue. There is then

some helper methods defined, `printList`, which prints out the Linked list, and `freeList`, which frees the memory for everything malloc-ed in the linked list.

Worker_create

Our worker_create method is one of our entries and initializers for our scheduler. If the scheduler is not already initialized, we will call a special thread_init() method which will set up the scheduler, the timers for the timer interrupts, and the main method that had called the worker_create. After everything has been initialized if it wasn't already, worker_create will create a new block for the new thread, filling it out with all the relevant context and wrapping its calling function in a custom wrapper so that terminations of that thread do not affect the entire scheduler as a whole. Once everything has been filled out, the new thread is then added to the run queue for the scheduler to take care of.

Worker_yield

Our worker_yield changes the status of the running thread back to ready, swapping back to the scheduler to run the next thread.

Worker_exit

Worker_exit will change the status of the running thread to Terminated and save the return value ptr passed in to be returned in the parent. It then calculates the turnaround time of the thread, as it has now exited, and adds the average to the overall.

Worker_join

The worker_join method will find the thread that it is waiting to join, and monitor the status of the thread, waiting for it to finally terminate. Whilst it waits, it will put itself into the Yielding state, allowing other programs to run.

Mutex

Our mutex struct makes use of an `atomic_int` variable that will tell us if the specific mutex has been locked or not. It also has a worker_t variable to keep track of the owner, a queue of its own to keep track of all the threads waiting for the mutex, as well as an id to easily identify the mutex, but to know if its been initialized or not.

Worker_mutex_init

The worker_mutex_init method initializes the actual mutex, and is the other entry point into the schedulers. This is because the mutexes can be created even when there is no other worker thread that has been created. As such, we do another check here for the initiation of the scheduler. Once that has been checked and initialized, we then start a critical section, intializing the mutex lock with no owner, assigning it a id mallocing the necessary items before exiting the critical section and continuing.

Worker_mutex_lock

The worker_mutex_lock is called in order to lock a certain mutex, giving ownership of the mutex to the thread calling the method. We enter a critical section and allow for the mutex to be checked for whether or not it is locked. If it is already locked, and the owner is not the caller of the method, we will block the thread from running and add it to the mutex's queue, keeping track of the fact that it is currently waiting for the mutex. Otherwise, if there is no owner for the mutex, it will set the owner as the calling thread and proceed to lock the mutex.

Worker_mutex_unlock

The worker_mutex_unlock will release the given mutex if the caller is the owner of that mutex. If they are the owner, then the mutex will be unlocked, and the first item in the queue will be dequued from the mutex queue and its status in the run queue will be set back to ready to allow it to continue executions.

Worker_mutex_destroy

Worker_mutex_destroy will take the current mutex and proceed to free all the items that are left in the queue. We do not change the statuses of any left in the queue if they have not been reincorporated into the scheduler queue as they will not be able to access the mutex regardless, meaning it is on the user to make sure there are no threads waiting for the mutex before destroying it.

Benchmarks and Analysis

PSJF

For our PSJF algorithm, the average runtime appears to be around 2000 milliseconds, regardless of the total threads being scheduled. We average around 26000 context switches total and an average turnaround time of 900 milliseconds. The thing that changed in between the number of threads is the average response time of the program. For 4 threads, it had a average turnaround time of 10.2 milliseconds, for 5 threads a average turnaround time of 14 milliseconds, and for 6 threads, a avg of 17.2.

For our preemptive shortest job first algorithm, the results of the test are that it appears to run 4-5 times slower than the actual pthread library. When running `parallel_cal` with 4 separate threads, we got a 1991 millisecond total runtime, whereas pthread only took 529 milliseconds. When running with 5 separate threads, we had a runtime of 1989 milliseconds, whereas pthread took 423.

Our results show a significant difference in performance between the actual pthread library and our self-built one. This can be explained by a scheduling algorithm that works in user level threads, without the ability use any kernel level privileges.

MLFQ

For our MLFQ algorithm, the average runtime appears to be around ranges from 2000 milliseconds to 15000 millisecond depending on the task at hand. We average around 500 context switches. Between numbers of threads, the response time increased from around 18 ms with 4 threads to 33 with 8 threads

When running `parallel_cal` with 4 separate threads, we got a 1996 millisecond total runtime, whereas pthread only took 520 milliseconds. When running with 5 separate threads, we had a runtime of 1989 milliseconds, whereas pthread took 396 with priority setting.

Our results show a significant difference in performance between the actual pthread library and our self-built one. This can be explained by a scheduling algorithm that works in user level threads, without the ability to use any kernel level privileges. Also there are some limitations in complexity regarding picking the right time quantum for the scheduler and also for resetting the queues.