John Quinn, jjq23

A Report on my Worker Thread Library

1. The worker API worked as such.
    a. Upon the first call of worker_create, it sets up the run queue, the main TCB, saves the main context into it, and pushes it onto the run queue. After this, it also sets up the scheduler context, the round robin timer and the signal handler for it as well. Following this, a new thread TCB is created and initialized, with a new context set up for its corresponding function. It immediately gets pushed onto the run queue, and the context is switched into the schedule function.
    b. The worker_yield function sets up a pause timer, changes the status of the current thread from RUNNING to YIELDED, pushes it back onto the run queue, and context switches back into the scheduler.
    c. The worker_exit function simply changes the status of the current thread from RUNNING to KILLED and stores the return value from the thread into the value pointer.
    d. The worker_join function takes the current thread and checks if it is killed. If it isn't killed, it calls worker_yield. The function also calls yield if the passed in thread doesn't correspond to the current thread. If the current thread is correct, and it is killed, the context is switched back into the scheduler context.
    e. The worker_mutex_init function simply checks if the mutex pointer is NULL, if not, the thread ID is set, the init flag is set to 1, and the locked flag is set to 0.
    f. The worker_mutex_lock function also checks if the mutex pointer is NULL. If not, it checks if it is initialized, if it does, it checks if the lock is already in place. If it is, the current thread is not the one that set the lock and is therefore blocked. If the lock is not set, the thread ID is set to the thread that called it, sets the lock_in_place flag to one, and returns.
    g. The worker_mutex_unlock function is simpler. If the mutex is not null and initialized, it checks if the mutex is locked. If it isn't, the function returns. If it is locked, the thread ID is reset, and the locked and lock_in_place flags are set to 0.
    h. The worker_mutex_destroy function sets the init flag to 0. If a new mutex is set up with the mutex pointer, the init flag will be turned on.
    i. For the scheduler, I primarily based it on the threads inside of the run queue. Both the Round Robin policy and the MLFQ were based on this simple procedure. The current thread is popped from the run queue, and its status is checked. If the thread is ready to run, the status is set to "Running", and the context is switched into the thread's context. The run timer is prepared and begins counting down. Once the timer runs out, the thread is placed back onto the run queue. This continues until all the threads terminate, upon which the context is switched back into the main program.

i. For the MLFQ, each thread is assigned a priority of 0 upon creation, meaning the highest priority. Upon the creation of the scheduler context, a reset timer is created. The procedure continues until the thread timer is over, where the priority is increased. If a thread is dequeued and has a lower priority, it is pushed onto a queue of lower priority, and the run queue is popped again. If every thread in the run queue is used up, the lower priority threads are pushed onto the run queue. When the reset timer runs out, all the lower priority threads are pushed onto the run queue with the highest priority.

2. Results:

2 threads	**External_Cal**	**Parallel_Cal**	**Vector_Multiply**
Running Time (RR)	490 us	2703 us	25 us
Running Time (MLFQ)	502 us	2710 us	56 us
Running Time (pthread)	2532 us	946 us	187 us

4 threads	**External_Cal**	**Parallel_Cal**	**Vector_Multiply**
Running Time (RR)	493 us	2702 us	25 us
Running Time (MLFQ)	511 us	2707 us	25 us
Running Time (pthread)	1928 us	828 us	202 us

8 threads	**External_Cal**	**Parallel_Cal**	**Vector_Multiply**
Running Time (RR)	492 us	2707 us	26 us
Running Time (MLFQ)	499 us	2706 us	25 us
Running Time (pthread)	2185 us	431 us	254 us

16 threads	**External_Cal**	**Parallel_Cal**	**Vector_Multiply**
Running Time (RR)	490 us	2789 us	38 us
Running Time (MLFQ)	501 us	2713 us	34 us

Running Time (pthread)	2280 us	290 us	331

4 threads, MLFQ	**External_Cal**	**Parallel_Cal**	**Vector_Multiply**
S = 100	569 us	2300 us	31 us
S = 250	563 us	3060 us	48 us
S = 500	812 us	3050 us	28 us
S = 1000	609 us	3544 us	31 us

3. Analysis:
    a. One interesting aspect about my thread library is that while the same result for each benchmark was computed among both my worker library and the default pthread, there was a considerable difference in time to compute. For both the external_cal and vector_multiply programs, they worked considerably faster than the linux pthread library. External_cal for both the round robin and MLFQ implementations took around 500 microseconds to complete, whereas the pthread libraries took over 2000 microseconds. Both my workers and the pthread library completed Vector_multiply relatively fast, however my workers took between 15% to 30% the amount of time it took the pthreads. When it came to parallel_cal, the reverse was true, as with the pthread library, the application completed in under a thousand microseconds, where for both my round robin and MLFQ implementations took over 2000 microseconds.