Anne Whitman : alh220
Michael Mazzola : mjm706
Henry Starman : hfs16

Quick Important Notes:

-Maximum number of threads: 128 (2^7)
    -This means a total of 127 threads can be created (on first create an extra one is created/used to store main's context.)
-Thread stack size set to be 16384 (2^14)
-Quantum set to 25MS
-Number of priority levels in MPQ: 4 (numbered 0-3)
-Quantum per priority:
    Level 0: 2^0 * Quantum (1 Quantum = 25MS)
    Level 1: 2^1 * Quantum (2 Quantum = 50MS)
    Level 2: 2^2 * Quantum (4 Quantum = 100MS)
    Level 3: 2^3 * Quantum (8 Quantum = 200MS)
-Cycles before promotion: 5.

Scheduling

Priorities and Quantums/TimeSlices

We implemented a multi-level priority queue with 4 levels total (0-3). All newly created threads start in level 0. Each level received 2^priority quantums (1, 2, 4, and 8 respectively). Every maintenance cycle we put 38 from level 0, 17 from level 1, 7 from level 2, and 2 from level 3 into the running queue.

Promotion/Demotion

After each thread finishes its quantum, it is demoted to the next queue (unless it is already in the lowest queue). If a thread yields, it does not demote. If a thread is placed onto a waiting queue (through a join or mutex lock), it is placed back into the same priority queue it was in prior to locking. If a thread has been on a given priority queue for more than 5 maintenance cycles, it is promoted up one level.

Maintenance

The maintenance cycle is used to increase priority levels of threads who have been in the cycle for too long. We determined that any thread that has been inactive for 5 cycles will have their priority level increased. Once it is increased it is relocated in the multi-level priority queue. After all threads have been placed into their appropriate location, the createRunning() method is called, which pulls a specified amount of

threads from each priority level and adds them into a queue which will be used for execution.

Handling Priority Inversion

Priority inversion was handled using Priority Inheritance (PInh). With PInh, when there is a low priority thread locking a mutex and a higher priority thread enters, then the low priority thread "inherits" the higher one's priority and runs inside the higher priority queue (never being demoted to a lower queue) until it unlocks the mutex to allow the higher priority thread into the lock. This will prevent the high priority thread from waiting a large amount of time for the low priority thread to run and release the lock.

Implementation

Inside the mutex_lock function, when a thread (A) is passed to the waiting queue, the priority of A is compared to the priority of the mutex owner (B). If A's priority is less than B's then A will be added to the end of the waiting queue. However if A's is greater than B's then PInh is set in place. Thread A will be placed at the head of the waiting queue instead of the tail and B will be forced to run at level 0 (highest priority) without being demoted to a lower priority over time, until it is able to finish and release the mutex. At that point A is a able to gain control of the mutex and and can run normally.

Analysis

In addition to verifying our threads against the benchmark provided, we also did other testing.

Basic Testing

We ran some basic tests including making sure that when my_pthread_yield, my_pthread_exit, and my_pthread_join are called without my_pthread_create being called, everything doesn't blow up. Instead they return a -1 (with the exception of exit) and return gracefully back to the calling function.

Threads

We ran the same tests using regular pthreads (using the linux implementation), and then compared it across our own threads trying different strategies. Our test was creating an even number of threads. Half of the threads called a function that incremented i 1,000,000,000 times (ensuring preemption), and the other half of the

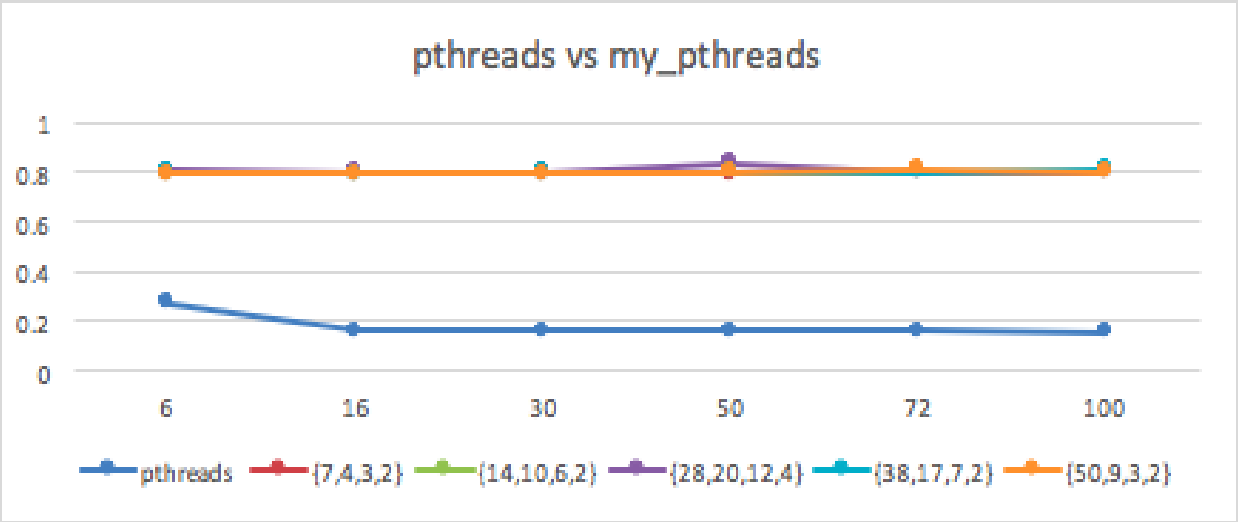
functions simply assigned a number into a return value. We called these in a way where we switched back and forth between them so every odd numbered thread called the shorter function, and every even numbered thread called the longer function.

Using this test we performed tests on the following data:
-first column is the linux implemented pthreads.
-every other column is the number of threads scheduled at a time from each priority level of the MPQ (across top of table)
{level0, level1, level2, level3}
-on the left margin are the number of threads used in this program.
-the data is calculated in seconds and marks how long each thread took on average.

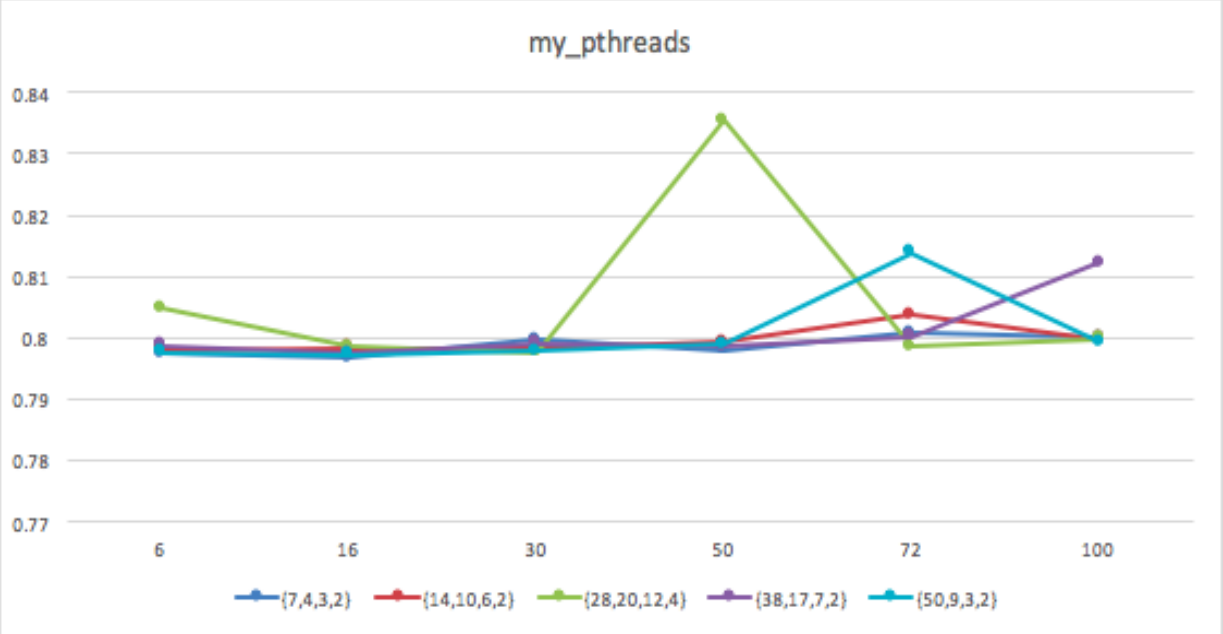
While this data may seem high, keep in mind that only half of these threads completed in one cycle. The other half of these threads included many preemptions/context switches in order to complete.

	actual	test1	test2	test3	test4	test5
	pthreads	{7,4,3,2}	{14,10,6,2}	{28,20,12,4	{38,17,7,2}	{50,9,3,2}
6	0.2751032	0.7974307	0.7980128	0.8047057	0.7987185	0.797578
16	0.1603173	0.7966988	0.7982865	0.798514	0.7974275	0.7973026
30	0.1581756	0.7995908	0.798112	0.797678	0.7991534	0.7978338
50	0.1618587	0.7979024	0.799289	0.835382	0.7986009	0.7988784
72	0.1574179	0.8007072	0.8036753	0.7984355	0.8002498	0.8137925
100	0.1555841	0.7999349	0.7998052	0.7998319	0.8122398	0.7992076

As you can see from our preliminary graph below, we didn't quite hit the same benchmarks as LInux. But we were only about 4 times higher, and averaged less than a second, for each thread.



We decided to remove LInux implemented pthreads from the equation and only look at our own:



-As you can see we did not get much improvement as we tried changing the number of threads scheduled per cycle per priority. However, even with these minimal changes, the best run times (although minimally different) seemed to come from the {38, 17, 7, 2} split. We ended up using this in our project since it had slightly better advantages and helped give lower priorities a better shot as well without sacrificing runtime.

Mutexes

- Used a function that increments 100,000,000 twice (to test mutex wait queue)
- One function incremented a variable by 5
- The other function decremented the variable by 3

Mutexes

Number of Threads	Time (s)	Time/Thread
5	1.667755	.333551
10	3.248028	.324803
20	6.503427	.325171
30	9.755332	.325178
50	16.388045	.327761
75	24.954903	.332732
100	33.38882	.333888

Above are the results for the tests on the mutexes with varying numbers of threads. The last column is to show that the mutex implementation is in constant time and that the wait queue for the mutex is working properly.

Methods Definitions, Descriptions, and Return Values

int schedulerInit();

initializes the scheduler once a single thread is created. Returns 0 on success and -1 on failure.

void scheduler();

runs until no more threads exist in MPQ alternating between a maintenance cycle and running the running queue. Does not return anything.

void maintenanceCycle();

This method checks to see if the priority queue is empty. If it is empty and there are no threads, it will tell the scheduler to stop trying to run threads until a new thread is added. If it is not empty, it will iterate through each level of the priority

queue and determine which threads should have their priority level promoted. Increasing the priority level is dependent on a thread existing in the maintenance cycle for a set amount of cycles (which is currently set to five). After every thread has been checked in the while loop, the createRunning method is then called. There is no return value.

void createRunning();

This function builds the running queue that will be executed during the current maintenance cycle. It pulls a set amount of threads from each priority level and combines them together into a running queue.

void runThreads();

This method will run each thread in the running queue. As each thread preempts or yields, it will add them back to the MPQ into the respective level. If the thread finishes without exiting, this method will manually call exit.

void time_handle(int signum);

This handles the interrupts at the end of every thread's time slice. After it interrupts, it changes the priority and yields back to the scheduler (runThreads method particularly).

void timer(int priority);

Sets the time slice for the next thread.

void addMPQ(tcb* thread, queueNode** head, queueNode** tail);

Takes a thread and the head and tail of a priority queue as the arguments. This method adds the thread to the proper level of the priority queue.

int my_pthread_create(my_pthread_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg);

Takes the address of a my_pthread_t, an attributes struct (which is ignored in this implementation), a function pointer, and a single argument to pass into the function.

Upon successful creation of the first thread, adds the contexts of the main method and the first thread and then initializes the scheduler to start scheduling threads.

Return Values: 0 upon success.

-1 if there are no more threads available.
-2 if there was an issue getting or making a context.
-3 if there was an issue initializing the scheduler.

int my_pthread_yield();

This function takes no arguments. If this function is called by an executing thread, it returns control back to the scheduler. If this function is called by the scheduler, it sets the interrupt timer and immediately gives control to the next thread on the running queue.

Return Values: 0 upon success.
-1 if it is called upon before a call to pthread_create.
-2 if there was an issue swapping contexts.

void my_pthread_exit(void *value_ptr);

This function takes a single argument, a void pointer. This void pointer holds the thread's return value. If there is no return value, NULL should be passed in. The function itself has no return value.

int my_pthread_join(my_pthread_t thread, void **value_ptr);

This function takes a my_pthread_t and a pointer to a void pointer. The my_pthread_t is the thread that the calling context would like to join. Upon calling this function, the calling thread will be placed on a wait queue and will yield control back to the scheduler until the thread it's joining finishes execution. At that point this thread will be rescheduled and will find the return value it requested at the address provided in the parameters.

Return values: 0 upon successful join
-1 if it is called upon before a call to pthread_create.
-2 if the thread does not (or no longer) exists.

int my_pthread_mutex_init(my_pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

This takes the argument of a declared pthread_mutex_t. If the mutex is not NULL it gets passed through to be initialized. In initialization, the mutex is set to be unlocked, room is malloced to keep track of the mutex's owner and for the mutex wait queue. The wait queues purpose is to hold threads that are waiting for the mutex to unlock. The second argument (mutexattr) is ignored and not implemented for this assignment.

int my_pthread_mutex_lock(my_pthread_mutex_t *mutex);

This function is the meat of the mutex. When threads enter the function they immediately meet a while loop with a __sync_lock_test_and_set function as its condition. When the condition for the while loop is FALSE, a thread passes beyond the loop and is set to the mutex's owner. Then the mutex becomes locked. All other threads that pass after the owner (while the owner is still using the lock) will go inside the while loop. These threads are then enqueued to the mutex's wait queue and wait for the thread using the mutex to finish. After a thread is enqueued the scheduler is called to handle the rest of the threads. (There are some lines here dedicated to handling priority inheritance that will be explained later in the readme).

int my_pthread_mutex_unlock(my_pthread_mutex_t *mutex);

When a mutex is passed to this thread the the mutex is either set to the unlock state or is given another thread from its wait queue. If there are no threads in the mutex's queue then the mutex state is set to unlocked. However if threads do exist in the wait queue then a thread is dequeued from it and then passed back into the multi priority queue to continue running.

int my_pthread_mutex_destroy(my_pthread_mutex_t *mutex);

Mutex destroy will free a given pthread_mutex_t and make it unavailable.
First the code will check for errors. If NULL is passed to mutex_destroy the errno is set and an error is returned. The same thing will happen when the mutex is still in the lock state (signifying that the mutex is still in use).