README

Thomas Daub, Eli Minkin, Marshal Nink

Document scheduler, priority queues

The multi-level pirority queue implemented consists of 3 separate priority queues (PQ for short): priority_Queue0,priority_Queue1,priority_Queue2. PQ0 is the hgihest priority queue with a time quanta of 25ms and is the first PQ that a TCB is loaded into. PQ1 has a time quanta of 50ms and PQ2 has 100ms.

A running queue of TCBs was implemented and pulls from all three priority queues and runs each TCB sequencially. If a TCB does not finish in its specific time, it is places into a higher PQ and waits to be put back into the running queue to run again. When the running queue picks TCBs from the priority queues, it attempts to fill in the order of 4 PQ0 TCBs, 2 PQ1 TCBs, and 1 PQ2 TCB and if it cannot it attempts to get as close to 300ms of total time as possible. The running queue implementation is used to help avoid starvation so no matter what a low priority job can always get a chance to run.

The scheduler() function chooses which tcb from the running queue to swap to the context of. If it the running queue is empty it runs createrunninqueue() which does the above selection process to fill the running queue and then swaps to the head of the list. If a function every exits, returns, or times out the scheduler is called to swap context. The scheduler always checks if the function had exited and if it did it's tcb is removed enitirely and not put back in any priority queue. It also checks if a function ran its whole time slice and if it did the tcb is put at a lower priority. If the function instead called yield because it was waiting for a mutex then its priority is not lowered. To help avoid starvation a tcb keeps track of how long it has been running. Everytime a context returns to the scheduler the scheduler increases how long it has been running based on which priority queue it came from. If the time it has been running is greater than 375 ms the tcb is set to priority 0 and put in priority_Queue0. Scheduler() also sets the timer for each tcb before the context is swapped. The timer given to the timer depends on the priority of the tcb being swapped to.

Yield, Signal Handler, Exit, Join

my_pthread_yield() is essentially used to call scheduler() but it makes sure the timer is stopped so that the scheduler will not get interrupted. Yield is called by my_pthread_create to ensure that the scheduler will invoked so that the jobs will start to be run again. Yield is also called when a thread checks to see if a mutex is locked and the mutex ends up being locked. This is done so the thread is not wasting time doing nothing.

We implemented a signal handler to detect when the timer goes off for a thread. This signal handler invokes the scheduler but passes 1 to it so the scheduler knows a thread had run its whole time slice without exiting. It stops the timer as well.

When my_pthread_exit is called by a thread a flag is set for the tcb that was currently running its context saying the tcb had exited so he scheduler knows to remove it. Then it checks a list of tcbs being waited on by other tcb's to see if it is there. If it is then it sets a flag in the waiting list to say it has exited. If exit was given a non-null return value then the return value given is put on a list of return values so a thread that calls join on the current thread exiting can get that return value. Every tcb has a uc link set to my_pthread_exit so if a thread happens to return it can exit properly and thus be taken out of the running queue and priority queues.

Note: There is a struct made to hold the data of whether or not a thread has exited and a linked list of those structs was also made. There is also a struct made to hold the value a struct had returned by passing a parameter to exit. Again, a linked list of these structs was also created.

When my_pthread_join is called the function checks if the thread being waited on is still in any of the priority queues or in the running queue. If it is not the thread continues from where it left off. If it is then a struct holding the data of whether or not the thread exited is initialized and put on a linked list of such structs then the thread enters a loop so it can wait for the thread its waiting for to exit. If a non-null pointer is passed to join then the thread checks through a list of return values to find if the thread its waiting on had returned a value. If so it puts the return value in the non-null pointer it was passed.

Mutual Exclusion

The struct my_pthread_mutex_t has two fields. An integer called destroyed which is 1 if the mutex was destroyed and 0 otherwise and another integer which is called flag which is 1 if the mutex has been locked and 0 if it is unlocked.

my_pthread_mutex_init() will check if the mutex had been destroyed before. If it was destroyed destroyed will be set to 0 and the flag will already be 0 because a mutex will not be destroyed if flag is 1. If it wasn't destroyed, it will initialize the mutex flag to 0.

my_pthread_mutex_lock() will first check if the mutex is destroyed and if so the function will just return -1 as an error. If the mutex is not destroyed the function will enter a loop which will keep looping if the atomic test and set function returns 1. Test and set is given the mutexes flag and one as arguments. The loop is escaped once the thread that locked the mutex unlocks it. However every time the loop is entered the thread yields so it does not waste computation time. Once unlocked the thread sets the mutex flag back to one so it can be locked.

my_pthread_mutex_unlock() will return -1 as an error if the mutex was destroyed already. If it was not destroyed the thread will set the flag to 0 so other threads can use the mutex or so that it can be destroyed.

my_pthread_mutex_destroy() will return -1 as an error if the mutexes flag is set to 1 because one cannot destroy a mutex that is locked(because it is in use). If the flag is 0 the function will set destroyed to 1. If the flag is 0 that means the mutex is not locking any data so it is free to destroy.