Joseph Jaeschke (jjj93), Crystal Calandra (crystaca), Adeeb Kebir (ask171)
Tested on the cray1 iLab machine
Asst1: Adventures in Scheduling
CS 416

my_pthread_t.h:

For assignment one, we implement a user level thread library. To accomplish this a variety of constructs were needed. Since threads are essentially separate execution contexts, having a different context for each thread was how we differentiated different threads. In order to store the threads, a multilevel priority queue was used. Every thread started at the highest priority level when it was created. Every time a thread finished its time-slice, it would drop to the next lowest priority level. Now that the threads were stored properly a scheduler can now decide what threads to switch to and when. The scheduler would run the first thread that was ready to run by searching from the highest priority level to the lowest. After a certain amount of threads were run, a maintenance cycle was run to help prevent starvation of the threads by boosting all ready threads to the highest priority level. Since the execution of one thread may be swapped with another at any given time, a synchronization construct was needed. For this we implemented mutex locks that utilized atomic operations to prevent more than one thread from entering a critical section at a time. More details on the implementations of these can be found below.

mutex struct:

In the header file we have a my_pthread_mutex_t struct defined for new mutexes. It contains:

- an integer value "locked" which can be set to 0 for unlocked or 1 for locked
- a pointer to a thread control block called "waiting" which contains a linked list to all threads waiting for this particular mutex
- an integer value for "maxP" which contains the maximum (aka lowest) possible priority level
- a pointer to another my_pthread_mutex_t called "next" which will create a linked list of all mutexes

my_pthread_mutex_t* mutexList is defined here. It is a global pointer to the beginning of the linked list of mutexes.

Thread control block struct:

The struct for a thread control block is also defined here. It contains:

- a thread id
- a thread state. States include embryo state where thread is not fully constructed, a running state, a ready state, a waiting state, and a joining state.
- a void pointer to store the return value
- a context for the thread
- a time-slice for the thread’s
- a priority of the thread
- an old priority for the thread in case it’s priority was modified by another function like my_pthread_yield, it could still be found in the queue
- a pointer to the next thread. This could be either the next thread in the run-queue, terminating

list, or mutex wait list.

The tcb** queue is defined here. This is the multilevel priority queue which will contain the threads that are ready to run.

The tcb* terminating is defined here. This is the list of terminating threads that have all finished running and have made a call to my_pthread_exit. Once a thread is in the terminating list, it is no longer in the multilevel priority queue.

Function declarations:

Contains function declarations for all functions used in the library including:

- alarm_handler
- scheduler
- maintenance
- my_pthread_create
- my_pthread_yield
- my_pthread_exit
- my_pthread_join
- my_pthread_mutex_init
- my_pthread_mutex_lock
- my_pthread_mutex_unlock
- my_pthread_mutex_destroy

(all functions explained in my_pthread.c section)

my_pthread.c:

my_pthread_mutex_init:

This function takes a my_pthread_mutex_t pointer from the user and populates the struct variables with appropriate values. The "locked" is initially set to 0 for unlocked, the "maxP" is set to the maximum priority level in the priority queue. The "waiting" pointer is set to NULL, the "maxP" is set to the highest priority in the priority queue, and the "next" is also set to NULL. This also adds the mutex to the "mutexList".

my_pthread_mutex_lock:

This function uses an "atomic" instruction to set the value of "locked" in the specified mutex passed by the user. If the value of "locked" is 1 for locked, the calling thread is placed in "waiting" status, removed from the run queue, and placed at the end of the "waiting" queue for the given mutex. This allows the mutexes to be handled in a first come first served manner. We decided not to take the threads original priority into consideration when placing it on the waiting list. Once the thread is in waiting status, it calls my_pthread_yield to allow another thread to run while it waits. If the value of "locked" was 2, it prints an error message stating that the mutex had already been destroyed. If the value of "locked" was 0, the calling thread is allowed to continue running.

my_pthread_mutex_unlock:

This function checks the status of the specified mutex (just as a safeguard) and then uses another "atomic" instruction to return the "locked" value to 0 for unlocked. It then takes the first thread from the waiting queue for the specified mutex and sets it's priority back to 0 and places it back in the run queue. We decided that the thread would be given highest priority since we do not know how long it had been waiting for the mutex to be freed and may have other threads waiting for the same mutex. It does not, however, yield the remaining time for the calling thread.

my_pthread_mutex_destroy:

This function checks to make sure that the given mutex is not in locked status first, resulting in an error if it is. Then it changes the "locked" status to 2 for destroyed and removes the mutex from the "mutexList". This should render the mutex no longer usable unless it is re-initialized through a new call to my_pthread_mutex_init().

my_pthread_create:

This function takes a pointer to a my_pthread_t, a pointer to a pthread_attr_t, a function pointer, and a void pointer to the users arguments. The attribute pointer is ignored. Upon the first call to my_pthread_create, the initialization of a main context is made, an initialization of a scheduler context is made, an a timer is set that corresponds to the first time-slice. The context initialization is accomplished with the use of ucontext and the timer is accomplished with the use of itimerval. If the call to my_pthread_create was not the first, then it will initialize a context for the new thread that is being created if the max thread limit has not been reached. At this point there will be a call to makecontext to finish making the context. Finally the new thread is added to the multilevel priority queue at the highest priority.

Rather than having the context of the new thread be that of the user’s provided function, there is a wrapper function that the user’s function and its arguments get passed to. The reasoning behind this is to force a call to my_pthread_exit. If the user thread does not call my_pthread_exit, the wrapper function will call it after the user’s function has ended. This way the thread that has finished can properly be added to the terminating queue.

Since the function parameters passed to makecontext are required to be integers some special handling of pointers needed to be done. To mask a pointer as an integer a union was made to hold a void pointer and a struct. In that struct were two integers. Since the void pointer and the struct were both 64 bits, setting one would set the other fully. Moreover, the first int in the struct would represent the first half of the address of the pointer and the second int would represent the second half of the address. In this way, the void pointer could be set to either the user’s function pointer or their argument pointer. This would set the two integers in the struct which could then be passed to makecontext. In the wrapper function, these two integers could be put back in the struct and then the void pointer could be read and casted to the appropriate type.

my_pthread_yield:

A call to my_pthread_yield will set the current thread’s priority to the lowest level. After this it will switch to the scheduler to run the next thread.

my_pthread_exit:

A thread will call my_pthread_exit when it is done running. The parameter is a pointer to the exiting thread's return value. When this function is called, the the thread will be marked as terminating and will be removed from the multilevel priority run queue and added to the list of terminating threads. At this point the thread will yield and forfeit its remaining time since it has nothing to run.

my_pthread_join:

A thread will call my_pthread_join when it wants to get the return value of a previous thread. This function takes two parameters, a my_pthread_t holding the thread id of the thread it wants to join with and a pointer to a void pointer to hold the return value of the joining thread. The first thing this function does is search for the thread it is looking for in the terminating list. If the thread it is looking for has not terminated it will yield since there is nothing else it can do. If the thread that called my_pthread_join finds the thread it is to join with in the terminating list, the terminated thread will be removed from the list. The terminated thread's return value will then be saved and the function will return to the user's program.

Since my_pthread_join takes a pointer to a void pointer which will eventually be dereferenced, some casting is needed. In order to dereference the pointer, it needs to be casted to a different type and then dereferenced. The type it was temporarily casted to was a pointer to a double pointer. This is because a pointer and a double are the same size. After the casting, the pointer can be dereferenced and set.

alarm_handler:

This is the signal handler that will catch the SIGALRM sent by the itimer. If the alarm went off inside the scheduler, it will simply return. If the alarm went off while the scheduler was not running, then the currently executing context will be swapped out for that of the scheduler. Since the alarm going off corresponds to the current thread's time-slice ending, when the alarm goes off, the thread will be moved to the next lowest priority and a new thread will be chosen to run.

scheduler:

The scheduler is where the previously running thread is put back in the queue and where the next thread is chosen to run. Once the scheduler is running, it will find the current thread in the multilevel priority queue and put it in the next level. An exception to this is if that thread just terminated or is in line for a mutex. If a thread made a call to my_pthread_exit, it will be marked as terminated in its state and removed from the run-queue and put in the terminated list. Similarly, if a thread is in line for a mutex, it is marked as waiting, it is removed from the multilevel priority queue, and put in the wait list for the respective mutex. In both these cases, the current thread cannot be found in the queue so if a thread comes into the scheduler and is marked as terminated or waiting, the scheduler will not try to remove it from its old priority queue. In the average case, the current thread's priority will be decrease by one level and placed in the priority queue corresponding to the updated priority. After that the scheduler will check if it is time to run the maintenance cycle. The workings of this will be explained in another section. The scheduler then picks a new thread to run. This is done by searching through the queue starting at the highest priority level. If no thread is found in the highest level it will move onto the second highest level and so on until a thread that is ready to run is found. Once a thread is found the scheduler will set a timer for the new thread. The time-slice is calculated by

25*(1 + priority level) milliseconds. Since the priorities start at zero, it is necessary to add one. Due to the calculation of the time-slice, high priority threads get a smaller time-slice than low priority threads, but high priority threads are run more often to balance out. The time-slice corresponds to the suggested 25 milliseconds proposed in the assignment specification. Once a timer is set, the scheduler switches context to the newly chosen thread.

maintenance:

After a specified amount of threads have finished their time-slice, the scheduler will call the maintenance cycle. In the maintenance cycle, all threads in the multilevel priority queue are boosted to the highest priority level. This is done to lessen the chance that a thread at a low priority level will experience starvation. If a user makes a lot of new threads then it is unlikely the threads at low levels in the queue will get runtime. Once all the threads are boosted to the highest priority level, then each of them are guaranteed to run once before any threads that are created after maintenance cycle get run. This is because new threads get added to the back of the queue. This does address thread starvation although it may not be optimal. This strategy was chosen because of its simplicity and its effectiveness and factors such as how often it is run will affect how well it preforms.