Project 2 Report

CS436 - OS Design

Omer Sen: os226
Reuben Thomas: rmt135

1. What structures you used to implement the TCB, mutex, runqueues, any other queues or structures, etc.

For the TCB, I included a worker_t TID as a unique identifier, an int status flag to keep a thread's status, a ucontext_t context for swapping contexts, and an int retval for storing any return value provided by the user. I also created a node struct as a wrapper for the TCB that can be added and removed from a queue. The queue is also a struct that contains a head node pointer, tail node pointer, and size int.

For the worker_mutex_t struct I included an int called mid for identifying a mutex, a mutex int variable for locking/unlocking, a node pointer to the current user of the mutex, and a waitQ queue struct to maintain a list of all threads waiting for the mutex. I also created a separate list of unique nodes called "mutNode's", and is a list of all currently active mutexes. This comes in handy when searching for a mutex or for waiting threads, determining the validity of a mutex given by the user, and determining whether a thread actually has the lock to a mutex.

Besides the waitQ's within each mutex, I also had a global runQ that keeps a list of all threads in the READY state. I also had a termQ or terminated queue that keeps a list of all exited threads. This comes in handy for determining when to free a thread.

I also created many helper functions to help manage the runQ, termQ, and each waitQ, which were searchQ, enqueue, dequeue, and removeNode. Because the mutex queue was made up of a different type of node, I also made separate functions for them that do the same thing as the regular functions.

I also had restartTimer(), pauseTimer(), and resumeTimer() functions. I paused the timer before executing any code in any worker function, and resumed it accordingly. This ensures that the timer can't interrupt the thread while it's manipulating any shared queues, TCB, mutex, etc. The signal_handler function also just swaps contexts to the scheduler.

2. What logic you used for implementing the thread and mutex API functions.

In my implementation, the worker functions handle most of the necessary queue, TCB, etc. manipulation rather than the scheduler in the scheduler context. I found it easier because we have direct access to the currentThread, relevant mutex, etc. before we swap contexts to scheduler, which is necessary for a lot of the struct manipulation.

Worker_create:

On the first call to worker_create, it initializes the scheduler context and TCB, queues, timers, and main context, tcb, and node, and pushes the main node to the runQ and sets it as the currentThread. I chose to give the scheduler a TCB because I thought it would be good to be able to identify it by a tid, though it's not necessary. Then, as on any call to worker_create, it creates a new thread context for the user, including a TCB and node, and pushes to runQ. If it's the first call, it then swaps to the scheduler, otherwise it returns back to the caller.

Worker_yield:

This function simply pauses the timer and swaps to the scheduler, which will handle the queue changes.

Worker_exit:

Worker_exit() pauses the timer, updates the return value in the currentThread's TCB to that which is given by the user, if at all. It then updates the currentThread's status to TERMINATED, enqueues it to the termQ, sets currentThread to NULL, and setcontext()'s to the scheduler.

Worker_join:

This function first pauses the timer, and checks the termQ for the given thread. If it's not there it will check the runQ and every mutex waitQ for it. If these return NULL, the process exits with the error, "attempt to join with nonexistent thread". Otherwise it will enter a spinlock and wait until the thread's state is TERMINATED. Once this happens, the function stores the child thread's return value in the location provided by the user, frees the child, resumes the timer and returns.

This is the only place where I free a thread. This means that it's the user's responsibility to make sure every thread is joined, otherwise there will be memory leaks. This is a fair responsibility to put on the user, because otherwise we would have to free "unjoined" threads as soon as they exit as there's no way to tell if someone might or might not call worker_join on them in the future. This means we lose return value data which could be useful. Also the main thread blocks, waiting to join the first created thread, which could happen after many other created threads exit and their return values are lost. Therefore, I decided to only free threads in the worker_join function.

Worker_mutex_init:

This function initializes a worker_mutex_t struct, including its waitQ. It also creates a mutNode struct for the mutex and pushes it onto the mutQ or mutex queue.

Worker_mutex_lock:

This function first ensures that the mutex given by the calling thread is valid, not currently in use, and that the calling thread is not already the current owner. Otherwise the process exits with error "invalid mutex lock". Then the function will use `__sync_lock_test_and_set()` to try and lock the mutex. If it succeeds, we update the currentUser of the mutex, and return. If not, we enqueue the currentThread onto the mutex's waitQ and swap to the scheduler. When the thread is run again, we try again to obtain the lock. If it fails again, we enqueue back to the waitQ and repeat in a loop. This can happen because I have implemented worker_mutex_unlock() such that a waiting thread is not immediately given cpu control after the unlock, but enqueued onto the end of the runQ. Therefore it's possible that another thread ahead in the runQ could have obtained the lock. Once the thread eventually gets the lock, the mutex's currentUser is updated, and returned.

Worker_mutex_unlock:

On an unlock, the function first ensures that the mutex given by the calling thread is valid, currently in use, and that the calling thread is truly the current owner of the mutex. Otherwise the process exits with error "invalid mutex unlock". Then the function will use `__sync_lock_release()` to unlock the mutex and sets the currentUser to NULL. Then it dequeues the mutex's wait queue, sets this thread's state to READY and enqueues it to the end of the runQ.

Worker_mutex_destroy:

The function first tests whether mutex currently exists, else the process exits with error "invalid mutex destroy". Then it checks whether this mutex is currently in use or if any threads are currently waiting to use it. If so, the process exits with error "attempt to destroy an occupied or requested mutex". Otherwise, the mutex is dequeued from the mutQ and freed, including its waitQ.

3. What logic you used for implementing the scheduler(s). (How it keeps track of which threads are in what states, how it handles transitions between states, how it chooses which thread to pick next, etc.)

The schedule function first pauses the timer, and then enqueues the current thread back into the runQ. If the currentThread is NULL (i.e. currentThread just exitted) or if the main thread was just created (so it has already been enqueued by worker_create and is the currentThread) then schedule() does not enqueue the currentThread to the runQ. After this schedule() calls sched_rr(), which dequeues the runQ and sets the resulting thread as the currentThread. Then it restarts the timer and setcontext()'s to this thread.

4. Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

I consulted the man pages for ucontext, itimerval, and signal. I also asked questions to my TA Abhilash about how to implement the queues and other logic. I used these resources to get a clearer understanding about how the thread library should function.