Gloria Nduka
March 2021
Project 2

The files I edited in the xv6-kernel-threads folder are bolded below.

exec.c
I implemented kill_others() helper function lines 10-21. This function kills all alive threads but itself. I called kill_others() in the exec() function on line 108. I call kill_others() because the thread doing exec should be telling other threads within the same process to kill themselves and then complete the exec task only themselves so the other threads in the same process don't attempt to perform exec.

kthread.h
I included function prototypes for kthread_create, kthread_id, kthread_join, kthread_exit, kthread_mutex_alloc, kthread_mutex_dealloc, kthread_mutex_lock, kthread_mutex_unlock in kthread.h and also defined max_mutexes to be 64 all on lines 5-13.

proc.c
This is where I did a majority of the function implementation. In the growproc() function I acquired and released locks. I acquired the ptable lock in line 170 and released the lock in lines 175, 181, and 188.This growproc() function is responsible for allocating more memory when the process requests for some more memory.
I implemented kill_all() helper function lines 249-267. This function kills all alive threads. I did not include any acquiring and releasing of locks because that happens in exit() function and I called kill_all() in the exit() function on line 309. The exit() function should kill the process and all of its threads because when a single thread executes exit() other threads within that same process might still be running so we want to kill them all.
I implemented kthread_create() in lines 647-667, and this will create a new thread within the context of the calling process. I didn't acquire or release any locks in this function because I am just creating kthreads and not sharing any shared fields.. The new thread will be in a runnable state. The caller of this function has to allocate a user stack for the new thread to use. I created the stack in user mode and sent its pointer to the system call in order to be consistent with the current memory allocator of xv6.
I implemented kthread_id() in lines 669-677 and that returns the caller thread's id. No locks implementation because I am just returning the ID of the caller thread.
I implemented kthread_join() in lines 716-752 and this function suspends the execution of the calling thread until the target thread, within the same process, terminates. I acquired and released

locks for synchronization. Since multiple threads could be calling this function and issues can arise within the system if locks aren't implemented

I implemented kthread_exit() in lines 679-712 and this function terminates the execution of the calling thread. This implementation doesn't kill the whole process if it's called by a thread while other threads exist in that same process. If it's the last thread that's running, then the process should be terminated. Acquiring and releasing locks need to be implemented since multiple threads could be calling this function and issues can arise within the system if locks aren't implemented; each thread must explicitly call this function to terminate itself. At the end of this function in line 711, I call sched() and don't release the ptable lock after sched() because sched() scheduled the next thread to be run, and in order to call sched() you must be holding the ptable lock so the scheduling operation isn't interrupted. After scheduling the next thread, the scheduler handles the releasing of the ptable lock.

I added another struct for the mtable lock to be created; I have a spin lock on the mutex table in lines 19-23.

I implemented the kthread_mutex functions: alloc, dealloc, lock, and unlock in lines 757-892. In the kthread_mutex_alloc function I am allocating a mutex object and then initializing it. The initial state of the mutex is unlocked, and then the function returns the ID of the initialized mutex. In kthread_mutex_dealloc function I am deallocating a mutex object that is not needed anymore and then returning 0 for success or -1 for failure if the given mutex is currently locked. The kthread_mutex_lock function is used by a thread to lock the specified mutex. If the mutex is locked already by another thread, this call will block the calling thread by changing the state to TBLOCKED until the mutex is unlocked. I implemented the TBLOCKED state and still used the sleep function. I added thread->state = TBLOCKED in line 848, and then also added thread->state = TRUNNABLE after the mutex object is unlocked. In kthread_mutex_unlock function it unlocks the specified mutex if called by the thread that owns that mutex, and if there are any blocked threads, then one of the threads will acquire the mutex. If the mutex was already unlocked then there will be an error. With the mutex object, once a thread has the mutex locked, no other thread can get the mutex, until the owning thread releases (unlocks) it. I acquired and released mtable mutex locks in all the kthread_mutex functions; this is based on the implementation of spinlocks to sync. Spinlocks are used in order to synchronize kernel code.

proc.h

I created a mutexstate struct that included all the states, unused, locked, and unlocked in line 56. I added a Blocked state in the threadstate struct in line 55. I also created a kthread_mutex struct that had the mutex id and the mutex state in lines 69-73.

syscall.c

I edited syscall.c and included lines 89-96: extern int sys_kthread_create(void), extern int sys_kthread_exit(void), extern int sys_kthread_id(void), extern int sys_kthread_join(void), extern int sys_kthread_mutex_alloc(void), extern int sys_kthread_mutex_dealloc(void), extern

int sys_kthread_mutex_unlock(void), and extern int sys_kthread_mutex_lock(void). These were listed in alphabetical order along with other syscall functions. I also added [SYS_kthread_create] sys_kthread_create, [SYS_kthread_id] sys_kthread_id, [SYS_kthread_exit] sys_kthread_exit, [SYS_kthread_join] sys_kthread_join, and also the corresponding kthread_mutex_alloc/dealloc/unlock/lock in the *syscall[] array on lines 130-141.

sysproc.c

I also edited the sysproc.c file. This file is a collection of process-related system calls. The functions in this file are called from syscall. I implemented 8 syscall functions in lines 99-171. I used the argint and argptr helper functions to help obtain the input parameter values and then call the corresponding functions with those obtained inputs. For sys_kthread_id() and sys_kthread_mutex_alloc I just returned kthread_id() and kthread_mutex_alloc(), respectively, and for sys_kthread_exit() I called kthread_exit().