Vidhya Lakshmi Venkatarama
CS533: Concepts of Operating Systems
Assignment 4: write-up

a) The yield() method from the previous assignments has been modified to prevent blocked threads from being added to the *ready_list*. Now our *ready_list* would strictly contain only threads with state READY. As a next step, I implemented a simple blocking mutex lock. I used the structure given, however to keep track of the thread that is holding the mutex, I introduced another field in the structure: *heldBy* which potentially makes the field *held* redundant. The mutex needs to be initialized in order to be used for mutual exclusion. The *lock()* method checks that the mutex is not held by the current thread and further that it is held (then block the current thread) or free (then acquired by the current_thread). The *unlock()* method on a mutex would wake up a thread (if any) waiting on the mutex and give it the lock and put it on the *ready_list*. The next synchronization primitive implemented is the condition variable that follow MESA semantics. A condition variable is first initialized, i.e. it's field (*waiting_threads* queue) is initialized. The condition variables can be used whenever a thread wants to check a condition, carry on if it's true or wait on the waiting thread of a condition variable if the condition is false. When another thread signals the former thread, it can wake up and check for the integrity of the condition and decide to continue or go to sleep again. This is the crux of MESA semantics (the waking thread rechecks the condition again). Another method that comes with MESA semantics: *broadcast()* is also implemented, in which the signaling thread wakes up all threads waiting on the conditional variable.

The given test *counter_test.c* executed successfully and demonstrated the functioning of the mutex. In order to test the condition variables, the second test, *sort_test.c* was used. For this, I implemented a join function called by a thread on other thread. That is if thread 1 calls join with thread 2 as its parameter, then the execution of thread 1 waits till thread 2 finishes its work and the state of thread 2 is DONE. *join()* is called by thread1 on thread 2 only if thread 1 forked thread 2. Hence, in order to give thread 1 the handle of thread 2, the method *fork()* was modified to return the created thread. This test also executed successfully.

b) Currently, our program is running on a single kernel thread. The user level threads are scheduled in a co-operative manner, yet yielding in critical sections. The possibility of race conditions here is avoided by the use of mutex locks that are acquired and released atomically (the functions associated with the mutex structure do not yield). However, on introduction of a second kernel thread, these functions would no longer be executing atomically as they will be shared by both threads and could be pre-empted any time, hence leading to race conditions. Then not only the mutex locks are at risk, but also the shared data these locks were protecting. In this situation, in order to ensure the safety of our program, we will require help from the hardware. Since our user level program is not running in privileged mode, disabling interrupts while acquiring and releasing lock is not possible. Hence, a lock atomically executed by the hardware (eg: Test and Set lock) can be used to protect the user level mutex which then protect the shared data, or the hardware method (TSL) can directly be used to protect the data.