**Task 1: KThread**

**Introduction**

Task 1 implements the KThread.join() function to allow one or more threads to join other threads. A queue will be used to store the threads while they are waiting to join the current thread. Before a thread can be joined to the current thread Its state must be checked to ensure that it is available.

Pseudocode – KThread.java

PROCEDURE KThread() :

joinCalled = false; // tracks if thread is called already

if(not current thread) // prevent thread from joining itself

join() {

if (joinCalled == true)

return;

if (joinThreadQueue not created)

create joinThreadQueue();

disable interrupts;

if (status not finished) {

joinThreadQueue.aquire(); // gives access to thread if no other thread holds access  
 joinThreadQueue.nextThread(); // give access to next thread   
 joinThreadQueue.waitForAccess(currentThread); // take access from the current thread and pass it to the next waiting thread

}

enable interrupts;

joinCalled = true;   
 }

finish() {

disable interrupts;

if(threads waiting to join)

ready next thread to join queue;

sleep();

}

END PROCEDURE

**Test Cases**

1. Ensure the join method is working by calling join

thread1:

thread2.join()

1. Join should only be called once per thread instance. If join is called more than once for the same thread, the second call will do nothing.

thread1:

thread2.join()

thread2.join() // second join should do nothing

1. A thread should allow multiple threads to be joined to the current thread. Once thread1 finishes, thread2 will start, and after thread2 finishes thread3 will start.

thread1:

thread2.join()

thread3.join()

1. A sleeping thread cannot join the current thread unless it is awoken first. If thread2 is sleeping

thread1:

thread2.join() // won’t join

thread2.wake() // wake up thread2

thread2.join() // thread2 is now able to join

**Task 2: Implement Condition2 Variables**

**Introduction**

Each thread will need a signal to allow it to sleep in the critical section and wait for the child process to finish before waking, re-acquiring the lock and proceeding. The queue will consist of sleeping threads that will wake the same way that one would when a semaphore condition is met. Threads sleeping in the critical section will be waiting to be put into the ready queue. Interrupts are disabled whenever a thread is running and enabled when the thread is no longer active to achieve atomicity.

Pseudocode – Condtion2.java

PROCEDURE Condition2() :

sleep() {

disable interrupts;

release lock;

add to sleeping queue;

wait for signal;

acquire lock;

enable interrupts;

}

wake() {

disable interrupts;

remove from sleeping queue;

move to ready queue;

enable interrupts;

}

wakeAll() {

disable interrupts;

while(sleeping queue is not empty)  
 wake();

enable interrupts;

}

END PROCEDURE

**Test Cases**

1. Calling sleep() on a sleeping thread should result with the function call exiting because the lock is not held.
2. Calling sleep() on a thread that is awake will put it in a waiting state and add it to a queue to be woken up.
3. Call sleep() on multiple threads and ensure they are queued to be woken up.
4. Calling wake() and wakeAll() with an empty queue should do nothing.
5. Calling wake() with a non-empty queue should result with the thread waking up one thread after acquiring the lock.
6. Calling wakeAll() with a queue of more than one thread. Ensure all threads are moved into the ready queue.