CSS 430 Operating Systems
Program 3B: Parent and Child Threads Synchronization

1. Purpose

This assignment exercises an implementation of Java monitors. While the original Java monitors have only an ability of randomly waking up one or all sleeping threads, the implementation of Java monitor in this assignment allows threads to sleep on and wake up from a different condition specified as an integer. Using this special monitor in *SynchQueue.java*, you will implement *SysLib.wait( )* and *SysLib.exit( )* system calls, with which a parent thread can wait for one of its child threads to be terminated.

2. SyncQueue.java: a Monitor That Distinguishes Different Conditions with Integers

In Java, if an object is enclosed with the *synchronized* keyword, it can behave as a monitor. A thread can put itself to sleep inside such a monitor by calling the *wait( )* method and can wake up another thread sleeping in the monitor by calling *notify( )* method. **However, Java monitors do not distinguish different conditions.** Therefore, in order to wake up a thread waiting for a specific condition, you need to implement a more generalized monitor, named *SyncQueue.java*:

Private/Public	Methods/Data	Descriptions
private	QueueNode[ ] queue	Maintains an array of QueueNode objects, each representing a different condition and enqueuing all threads that wait for this condition. You have to implement your own QueueNode.java. The size of the **queue** array should be given through a constructor whose spec is given below.
public	SyncQueue( ) SyncQueue(int condMax)	Are constructors that create a queue and allow threads to wait for a default condition number (=10) or a *condMax* number of condition/event types.
public	int enqueueAndSleep(int cond)	Allows a calling thread to sleep until a given *condition* is satisfied. It returns the ID of a (child) thread that has woken the calling thread.
public	dequeueAndWakeup(int cond, int tid)	Dequeues and wakes up a thread waiting for a given *condition* and informing the woken thread of this calling thread ID (tid). If there are two or more threads waiting for the same *condition*, only one thread is dequeued and resumed. The FCFS (first-come-first-service) order does not matter.

You need to implement SyncQueue.java and QueueNode.java. You can find their template files from Files/code/prog3.

3. SysLib.join( ) and SysLib.exit( )

In Java, you can use the *join(Thread t)* method to have the calling thread wait for the termination of the thread pointed by the argument *t*. However, if an application program forks two or more threads and it simply wants to wait for all of them to complete, waiting for a particular thread is not a good solution. In Unix/Linux, the *wait( )* system call is based on this idea. It does not receive any arguments, (thus no PID to wait on), but simply waits for one of the child processes and returns a PID that has woken up the calling process.

Now, consider thread synchronization in *ThreadOS*. We should not allow a user thread to directly access another thread's reference. Otherwise, a user thread can take over thread management such as suspending, resuming, and kiing other threads. This is *ThreadOS'* task. Therefore, *ThreadOS* should not allow a user thread to call the Java *join(Thread t)* method. It should follow the same synchronization semantics as Unix/Linux. For such synchronization, *ThreadOS* provides two system call such as *SysLib.join( )* and *SysLib.exit( )*. *SysLib.join( )* permits the calling thread to sleep until one of its child threads calls *SysLib.exit( )* upon its termination. It returns the ID of this child thread that woke up the calling thread.

We apply *SyncQueue.java* for implementing these two system calls. First of all, Kernel.java should instantiate a new SyncQueue object, say waitQueue. This waitQueue will use each thread ID as an independent waiting condition.

```
SyncQueue waitQueue = new SyncQueue( scheduler.getMaxThreads( ) );
```

SysLib.join() interrupts *Kernel.java* with an interrupt request type = *Kernel.WAIT*. Then, *Kernel.java* puts the current thread to sleep in *waitQueue* under the condition = this thread's ID. On the other hand, *SysLib.exit()* interrupts *Kernel.java* with an interrupt type = *Kernel.EXIT*. Then, *Kernel.java* searches *waitQueue* for and wakes up the one that has been waiting under the condition = the calling thread's parent ID.

You only need to implement the following two cases in Kernel.java:

```
case WAIT:
       // get the current thread id (use Scheduler.getMyTcb( ) )
       // let the current thread sleep in waitQueue under the condition
       // = this thread id (= tcb.getTid( ) )
       return OK; // return a child thread id who woke me up
case EXIT:
       // get the current thread's parent id (= tcb.getPid( ) )
       // search waitQueue for and wakes up the thread under the condition
       // = the current thread's parent id
       return OK;
```

4. Statement of Work

Task 1: Implement *SyncQueue.java* and *QueueNode.java* in accordance with the above specification.
Task 2: Modify the code of the *WAIT* and *EXIST* case in *Kernel.java* using *SyncQueue.java*, so that threads can wait for one of their child threads to be terminated. Note that SyncQueue.java should be instantiated as waitQueue upon a boot, (i.e., in the BOOT case).

```
public class Kernel {
       private static SyncQueue waitQueue;
       ...
       public static in interrupt( int irq, int cmd, int param, Object args ) {
              TCB myTcb;
              switch( irq ) {
              case INTERRUPT_SOFTWARE: // System calls or traps
              case BOOT:
                     ...
                     waitQueue = new SyncQueue( scheduler.getMaxThreads( ) );
                     ...
}
```

Task 3: Compile your SyncQueue.java and Kernel.java, and thereafter run *Test2.java* from the *Shell.class* to confirm:

1. *Test2.java* waits for the termination of all its five child threads, (i.e., the *TestThread2.java* threads).
2. *Shell.java* waits for the termination of *Test2.java*. Shell.java should not display its prompt until *Test2.java* has completed its execution.
3. *Loader.java* waits for the termination of *Shell.java*. *Loader.java* should not display its prompt (-->) until you type exit from the *Shell* prompt.

Please use the ThreadOS-original *Shell.class* but not the one you wrote for Prog1B.

5. What to Turn in

Total 20pts.

	Materials	Points	Note
1	Task 1: SyncQueue.java and QueueNode.java a. Code organization: **+2pts** • Well organized: 2pts, • Poor comments or bad organization: 1pt, or • No comments and horrible code: 0pts b. Correctness of SyncQueue.java: **+5pts** • Use a Vector or array of QueueNodes to distinguish conditions: **+1pts** • enqueueAndSleep( ) sleeps a thread on a condition: **+2pts**, and • dequeueAndWakeup( ) wakes up a thread waiting on a condition: **2pts** c. Correctness of QueueNode.java: **+3pts**	10	Submit these two files individually. Please use file names SyncQueue.java and QueueNode.java.

	• Use a list of child TIDs if they called dequeueAndWakeup( ) before their parent called enqueueAndSleep( ): **+2pts** and • Inform a woken thread of a child TID: **+1pts**		
2	Task 2: Your modified Kernel.java a. Code organization: **+2pts** • Well organized: 2pts, • Poor comments or bad organization: 1pt, or • No comments and horrible code: 0pts b. Correctness: **+3pts** • WAIT sleeps a calling thread on its TID: **+1pts** and • EXIT wakes up a thread waiting on the calling thread's parent TID: **+2pts**	5	Submit this file individually. Please use file name Kernel.java.
3	Task 3: One or more Loader, Shell, and Test2 execution snapshots a. A correct sequence of thread executions: 5pts, b. Incorrect: 3pts, or c. No results: 0pts	5	Include the snapshots in a pdf file named: FirstNameLastName_prog3B.pdf. Submit this pdf file individually. Please clearly label each snapshot.