CptS 370
Program 4: Synchronization

Due Date: 13 November, 2020 (Midnight)

---

1. Purpose

In this assignment, you will implement the monitors utilized by ThreadOS. While standard Java monitors are only able to wake up one (using *notify*) or all (using *notifyall()*) sleeping threads, the ThreadOS monitors (implemented in *SynchQueue.java*) allows threads to sleep and wake up on a specific condition. These monitors are used to implement two separate but key aspects of ThreadOS.

Firstly, using ThreadOS' s SynchQueue.java monitor, you will implement SysLib.join() and SysLib.exit() system calls. The SysLib.join() system call will allow the parent thread to wait until a child thread terminates. Secondly, in the Disk IO subsystem, the monitors are used to prevent threads from busy waiting on disk read and write operations. Specifically, in the assignment you will preempt the current thread if it attempts to execute a disk read or write operation on a busy disk, put it into the SynchQueue's FIFO list, and continue with other scheduled threads for execution. In this way, one can prevent I/0-bound threads from wasting CPU power.

2. Disk.java

Disk.java simulates a slow disk device composed of 1000 blocks, each containing 512 bytes. Those blocks are divided into 100 tracks, each of which thus includes 10 blocks. This disk provides the following commands (all of which return a Boolean value):

Operations	Behaviors
read(int blockId,byte buffer[])	reads data into buffer[] from the disk block specified by blockId
write(int blockId,byte buffer[])	writes the buffer[] data to the disk block specified by blockId
sync( )	writes back all logical disk data to the file DISK
testAndResetReady( )	tests the disk status to see if the current disk command such as read,write,or sync has been completed or not,and resets it if the command has been completed.
testReady( )	tests the disk status to see if the current disk command such as read,write,or sync has been completed or not.(The disk status will however not be reset.)

The DISK is created when first booting up TheadOS (java Boot). Upon this first invocation, Disk.java checks if your current directory includes the file named DISK. If such a file exists, Disk.java reads data from this file into its private array that simulates disk blocks. Otherwise, Disk.java creates a new DISK file in your current directory. Every time it accepts a sync command, Disk.java writes all its internal disk blocks into the DISK file. When ThreadOS Kernel is shut down, it automatically issues a sync command to Disk.java, so that Disk.java will retrieve data from the DISK file upon the next invocation. You can always start ThreadOS with a clean disk by removing the DISK file from the directory (rm DISK).

The disk commands read, write, and sync return a Boolean value indicating if Disk.java has accepted the command or not. The command may not be accepted if another read, write, or sync is in process. In this case false will be returned and the command will need to be reissued until the disk accepts the command and returns true. Note that when a true is returned the read, write, or sync has been accepted and is being executed. It is not necessarily complete. Disk IO is expensive and ThreadOS simulates the time it takes for the Disk to complete the operation and write or read from the buffer[].

When the read and the write have completed and the data has been read or written from the buffer[] the disk completion status will be set to true. Similarly, the sync command does not guarantee that all disk blocks have been written back to the DISK file when it returns; one has to check the disk completion status to determine when this is ready as well. The Disk provides two commands to check the completion status: testAndResetReady() and testReady( ). When the read, write, or sync have been completed by the disk

the commands will return true, otherwise false. The testAndResetReady() will reset the completion status to false if the status had been true. The testReady() call simply tests the current status but does not reset it.

The following code shows one way to do a clean disk read operation. Notice that testAndResetReady() command is used to set the disk status so another operation can be performed.

```
Disk disk = new Disk( 1000 );

// a disk read operation:
while ( disk.read( blockId,buffer ) == false )
        ; // busy wait
while ( disk.testAndResetReady( ) == false )
        ; // another busy wait
        // now you can access data in buffer
```

All disk operations are initiated through system calls from user threads. They are handled like all system calls by the ThreadOS Kernel. In Kernel.java the code which executes the system call forwards the disk operations to the disk device, (i.e., Disk.java). You can see the above read pattern used in Kernel.java. The following code is a portion of Kernel.java related to disk operations:

```
import java.util.*;
import java.lang.reflect.*;
import java.io.*;

public class Kernel {
    // Interrupt requests
    public final static int INTERRUPT_SOFTWARE = 1; // System calls
    public final static int INTERRUPT_DISK = 2; // Disk interrupts
    public final static int INTERRUPT_IO = 3; // Other I/O interrupts

    // System calls
    public final static int BOOT = 0; // SysLib.boot( )
    ...
    public final static int RAWREAD = 5; // SysLib.rawread(int blk,byte b[])

    public final static int RAWWRITE= 6; // SysLib.rawwrite(int blk,byte b[])

    public final static int SYNC = 7; // SysLib.sync( )

    // Return values
    public final static int OK = 0;
    public final static int ERROR = -1;

    // System thread references
    private static Scheduler scheduler;
    private static Disk disk;
```

```
    // The heart of Kernel
    public static int interrupt( int irq,int cmd,int param,Object args ) {

        TCB myTcb;
        switch( irq ) {
            case INTERRUPT_SOFTWARE: // System calls
                switch( cmd ) {
                    case BOOT:
                        // instantiate and start a scheduler
                        scheduler = new Scheduler( );
                        scheduler.start( );

                        // instantiate and start a disk
                        disk = new Disk( 100 );
                        disk.start( );
                        return OK;
                        ...

                    case RAWREAD: // read a block of data from disk
                            while (disk.read( param,( byte[] )args )==false)

                                ; // busy wait
                            while ( disk.testAndResetReady( ) == false )
                                ; // another busy wait
                            return OK;

                    case RAWWRITE: // write a block of data to disk
                            while (disk.write( param,( byte[] )args)==false )

                                ; // busy wait
                            while ( disk.testAndResetReady( ) == false )
                                ; // another busy wait
                            return OK;

                    case SYNC: // synchronize disk data to a real file
                            while ( disk.sync( ) == false )
                                ; // busy wait
                            while ( disk.testAndResetReady( ) == false )
                                ; // another busy wait
                            return OK;
                }
                return ERROR;

            case INTERRUPT_DISK: // Disk interrupts

            case INTERRUPT_IO: // other I/O interrupts (not implemented)

        }
        return OK;
    }
}
```

3. Wait and Notify

The above code has two severe performance problems. First a user thread must repeat requesting a disk call until the disk accepts its request. Second, the user thread needs to repeatedly check if its request has been served. Each of these operations are done in a *spin loop*. While functionally correct, these spin loops will cause the user thread to waste CPU until either relinquishing CPU upon a context switch or receiving a response from the disk. In order to avoid this performance penalty, you will utilize a monitor so that the thread can be enqueued and wait until the appropriate disk operation has occurred.

In Java, all objects come equipped with monitors. A thread can put itself to sleep inside the monitor by obtaining the monitor with the *synchronized* keyword and calling the *wait( )* method. A thread can be signaled and woken-up by obtaining the monitor and calling the *notify( )* method. However, Java monitors do not allow for different conditions to be signalled. Therefore, in order to wake up a thread waiting for a specific condition, we want to implement a more generalized monitor. We will call this monitor, SyncQueue, and the implementation will be in *SyncQueue.java*. The class should provide the following methods:

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 must 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	enqueueAndSleep(int condition)	allows a calling thread to sleep until a given *condition* is satisfied.
public	dequeueAndWakeup(int condition)	dequeues and wakes up a thread waiting for a given *condition*. 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.

With the *SyncQueue.java*, we can now code more efficient disk operations.

```
Disk disk = new Disk( 1000 );
SyncQueue ioQueue = new SyncQueue( );
...
...
// a disk read operation:
while ( disk.read( blockId, buffer ) == false )
{
   ioQueue.enqueueAndSleep( 1 );  // relinquish CPU to another ready thread
}
// now check to ensure disk is not busy
while ( disk.testAndResetReady( ) == false )
{
   ioQueue.enqueueAndSleep( 2 ); // relinquish CPU to another ready thread
}
// now you can access data in buffer
```

In the above example, the condition 1 stands for that a thread is waiting for the disk to accept a request, whereas the condition 2 stands for that a thread is waiting for the disk to complete a service, (i.e., waiting for the *buffer[]* array to be read or written). The remaining problem is who will wake up a thread sleeping on this *ioQueue* under the condition 1 or 2. It is the *ThreadOS Kernel* that will receive an interrupt from the disk device. Therefore, the kernel can wake up a waiting thread. You can consult the Kernel.java file to see how this is done.

4. 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 to 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.

In the first part of this lab you will use the SyncQueue Monitor to implement the *SysLib.join()* call in ThreadOS. Users of *ThreadOS* should not use the Java *join()* call directly but should utilize the *SysLib.join()* system call. We would like the *SysLib.join()* system call to follow similar semantics as the Unix/Linux *join()*

system call. *SysLib.join()* will permit the calling thread to sleep until one of its child threads terminates by calling *SysLib.exit()*. It should return the ID of the child thread that woke up the calling thread.

As noted, you will apply *SyncQueue.java* for implementing the *SysLib.join()* system call. *Kernel.java* should instantiate a new *SyncQueue* object called *waitQueue*. This *waitQueue* will use each thread ID as an independent waiting condition.

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

When *SysLib.join()* is invoked, *Kernel.java* will put the current thread to sleep in *waitQueue* under the condition that is equal to the thread's ID. When *SysLib.exit()* is invoked, *Kernel.java* searches the *waitQueue* for and wakes up the thread waiting under the condition which is equal to the current thread's parent ID. In this way, the exiting thread (child) will signal the waiting thread (parent).

```
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;
```

One more feature of *SysLib.join()* is to return to the calling thread the ID of the child thread that has woken it up. For this purpose, we need to define *SyncQueue*'s *enqueueAndSleep()* and *dequeueAndWakeup()* as follows:

Private/Public	Methods/Data	Descriptions
public	enqueueAndSleep(int condition)	enqueues the calling thread into the queue and sleeps it until a given *condition* is satisfied. It returns the ID of a child thread that has woken the calling thread.
public	dequeueAndWakeup(int condition), dequeueAndWakeup(int condition, int tid)	dequeues and wakes up a thread waiting for a given *condition*. 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. This function can receive the calling thread's ID, (*tid*) as the 2nd argument. This *tid* will be passed to the thread that has been woken up from *enqueueAndSleep*. If no 2nd argument is given, you may regard *tid* as 0.

5. Statement of Work

Per usual, create a '~/src/java/prog4' working directory and make soft links to the class files in '~/TOS'. Copy only the Java files which you will modify for your solutions from '~/TOS/' to your working directory, '~/src/java/prog4'.

Part 1: Implementing SysLib.join( ) and SysLib.exit( )

Design and code *SyncQueue.java* following the above specification. Modify the code of the *WAIT* and the *EXIT* 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), as shown below:

```
public class Kernel
{
private static SyncQueue waitQueue;
...

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

Compile your implementations of SyncQueue.java and Kernel.java, and 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.

Use the ThreadOS-original version of *Shell.class*.

Part 2: Implementing Asynchronous Disk I/O*

Before going onto Part 2, save your *Kernel.java* as *Kernel.old*. Thereafter, modify *Kernel.java* to use *SyncQueue.java* in the three case statements: *RAWREAD*, *RAWWRITE*, and *SYNC*, so that disk accesses will no longer cause spin loops. Note that *SyncQueue.java* should be instantiated as *ioQueue* upon a boot, (i.e., in the *BOOT* case).

```
public class Kernel
{
private static SyncQueue ioQueue;
...

public static int interrupt( int irq, int cmd, int param, Object args ) {
        TCB myTcb;
        switch( irq ) {
                case INTERRUPT_SOFTWARE: // System calls
                        switch( cmd ) {
                                case BOOT:
                                ...
                                        ioQueue = new SyncQueue( );
                                ...
}
```

Write a user-level test thread called *Test3.java* which spawns and waits for the completion of **X pairs** of threads (where **X = 1 ~ 4**), one conducting only numerical computation and the other reading/writing many blocks randomly across the disk. Those two types of threads may be written in *TestThread3a.java* and *TestThread3b.java* separately or written in *TestThread3.java* that receives an argument and conducts computation or disk accesses according to the argument. *Test3.java* should measure the time elapsed from the spawn to the termination of its child threads.

Measure the running time of *Test3.java* when running it on your *ThreadOS*. Then, replace *Kernel.java* with the older version, *Kernel.old* that does not

permit threads to sleep on disk operations. Compile and run this older version to see the running time of *Test3.java*. Did you see the performance improved by your newer version? Discuss the results in your report.

To verify your own test program, you may run the professor's Test3.class that receives one integer, (i.e., X) and spawns X pairs of computation and disk intensive threads. **Since Sig machines include multi-cores, you need to increase X up to 3 or 4 or more for observing the clear difference between interruptible and busy-wait I/Os.**

```
[Sig1]$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test3 3
l Test3 3
...
elapsed time = 85162 msec.
-->q
```

6. What to Turn In

- Softcopy (source files and documents in electronic form):

    1. Part 1:
        - Kernel.old (which you have modified for Part 1)
        - SyncQueue.java
        - Any other java programs you coded necessary to implement SyncQueue.java (QueueNode.java or whatever you named them)
        - Result when running Test2.class from Shell.class.
        - Specifications and descriptions about your java programs
    2. Part 2:
        - Kernel.java (which you have modified for Part 2)
        - Test3.java as well as TestThread3a.java / TestThread3b.java if created.
        - Performance result when running Test3.java on your Kernel.old (implemented in Part 1)
        - Performance result when running Test3.java on your Kernel.java (implemented in Part 2)
        - Specifications and descriptions about your java program

3. Report:
    - Discussions about performance results you got for part 2.

The grade guide for the assignment 3 is included in assignment on Blackboard.

7. FAQ

“Program 3 FAQ.pdf” on Blackboard could answer your questions. Please consult it before emailing the professor or the TA.

*** RACE CONDITION(?):**

There *may* be a race condition in the Disk class which is found in the initial ThreadOS directory. This race may be encountered in your testing for part 2 of this assignment. No other labs are susceptible to this race. It is not required that you understand it for this lab.

If you hit this race condition, ThreadOS will hang and a simple re-cycle will get it started again. The race is infrequent enough that you can complete your assignment as stated above with current implementations.

The distributed Kernel.java's code does not map directly to the to the one produced in the Kernel.class as the student is required to write some of this code.