Program 3

Wednesday, February 3, 2016 3:31 PM

UNDER CONSTRUCTION: description is fully usable and complete, I've just been working on redaction improvements for your convenience.

Synchronization Using Monitors

Please click here before emailing the professor for any question regarding this assignment.

1. Goals

In this assignment, you are asked to implement the ThreadOS monitors (implemented by *SynchQueue.java and QueueNode.java*), which allow threads to sleep and wake up ***on a specific condition***. In addition to implementing monitors, you are also asked to implement some system functions (e.g., SysLib.join() and SysLib.exit()) and modify *Kernel.java* to effectively use the monitors implemented in this assignment.

2. Implementing SyncQueue.java, QueueNode.java, SysLib.join( ) and SysLib.exit( )

2.1 Background

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.

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 signaled. Java monitors is only able to wake up one (using *notify*) or all (using *notifyall()*) sleeping threads. Therefore, we will need to be a bit more crafty in the ThreadOS scenario.

Java monitors are only able to wake up one (*notify*) or all (*notifyall*) sleeping threads. In ThreadOS, monitors should instead allow threads to sleep and wake up on specific conditions. ThreadOS monitors are implemented in ***SyncQueue.java*** and are used to implement the following key aspects:

1. *SysLib.join( ), SysLib.exit( ) and SysLib.wait()* system calls. The *SysLib.join( )* call in particular, allows a parent thread to wait until a child thread terminates.
2. Prevent IO-bound threads from wasting CPU cycles, i.e., avoid busy waiting on read/write disk operations in ThreadOS's Disk IO subsystem. A thread should be blocked when it attempts a read or write operation on a busy disk; the thread should be queued into the *SyncQueue*, and the system should continue to execute other scheduled threads.

2.2 **Implementation** of the SysLib.join() and SysLib.exit() calls 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 *wait()* 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.

In order to implement the *SysLib.join()* system call, *Kernel.java* should instantiate a new *SyncQueue* object called *waitQueue*. **This *waitQueue* will use each calling thread's ID as an independent waiting condition.** Class *SyncQueue* is a ThreadOS monitor class to be implemented.

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

When *SysLib.join()* is invoked, *Kernel.java* will put the calling thread to sleep and utilize the *waitQueue* to keep track of this thread. **The condition used by *waitQueue* to track the thread is equal to the thread's ID,** i.e., each condition in the wait queue corresponds to a ThreadOS thread, which can sleep on its corresponding condition variable.

When *SysLib.exit( )* is invoked, *Kernel.java* utilizes the *waitQueue* to wake up the thread waiting under the condition, which is equal to the current thread's parent ID. In this way, the exiting thread (child, who is calling *SysLib.exit()*) will notify the waiting thread (parent, who called *SysLib.join()*). Please note that *SysLib.join( )* must return to the calling thread the ID of the child thread that has woken it up. See the code below for details:

```
    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;
```

SyncQueueWAIT_EXIT.java hosted with ♥ by GitHub

Note that one key feature of *SysLib.join( )* is to return to the calling thread the ID of the child thread that has woken it up. For reason, we need to define *SyncQueue*'s *enqueueAndSleep( )* and *dequeueAndWakeup( )*.

2.3 Implementation of SyncQueue.java and QueueNode.java.

What type of underlying support do we need to block and wake up the threads for *SysLib.join()* and *SysLib.exit()*?

In order to wake up a thread waiting for a specific condition, we want to implement a more generalized ThreadOS monitor compared to Java monitors. We will call this ThreadOS monitor *SyncQueue*. You can use the java monitor support (*wait() / notify()*) to implement this class.

You should also create a class *QueueNode,* which is utilized by *SyncQueue*. There will be one *QueueNode* object per condition supported by the *SyncQueue*. The SyncQueue class should provide the following methods:

Private / Public	Methods/Data	Description
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)`
		Constructors that create a queue and allow threads to wait for a default condition number (of 10) or a condMax number of condition/event types.

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.

3. Updating Kernel.java to implement Asynchronous Disk I/O (Disk.java)

Please click here before emailing the professor for any question regarding this assignment.

3.1 Background

Disk.java simulates a slow hard drive 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):

Operation	Behavior
`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) and it might take up some time (maybe a couple of minutes.) 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 flight. In this case *false* will be returned and the command will need to be re-issued 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* or *write* operations 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.

```java
    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
```

raw DiskIO_Snippet.java hosted with ❤ by GitHub

All disk operations are initiated through system calls from user threads. There 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:

```java
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;

```

```java
    // 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 )
                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;
    }
}
```

view raw KernelSnippet.java hosted with ❤ by GitHub

The above code has two severe performance problems:

1. A user thread must repeat requesting a disk call until the disk accepts its request.
2. 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.

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 example above, condition 1 stands for that a thread that is waiting for the disk to accept a request, while the condition 2 stands for that a thread that is waiting for the disk to complete a service, (i.e., waiting for the *buffer[]* array to be read or written).
There is still the problem of who will wake up a thread sleeping on the *ioQueue* under condition 1 or 2: The *ThreadOS Kernel* will received an interrupt from the disk device, so the kernel can wake up a waiting thread. Inspect Kernel.java to see how this is done.

4. Statement of Work

Part 1: Implementing SysLib.wait( ) 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;
    ...

```

```java
    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( ) );
               ...
}
```

view raw SyncQueue-Wail.java hosted with ♥ by GitHub

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.

Note: Use the ThreadOS-original version of *Shell.class* (i.e., not the one you wrote in Program 1).

Part 2: Implementing Asynchronous Disk I/Os

Before going onto Part 2, save your *Kernel.java* as *Kernel.old*.
Then, 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).

```java
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( );
               ...
}
```

view raw SyncQueue-IO.java hosted with ♥ by GitHub

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.

```java
import java.util.Date;

class Test3 extends Thread {

    private int pairs;

    public Test3 ( String args[] ) {
        pairs = Integer.parseInt( args[0] );
    }

    public void run( ) {
      String[] args1 = … // parse arguments for computationally intensive test program, e.g., "TestThread3 comp"
      String[] args2 = … // parse arguments for disk intensive test program

      long startTime = (new Date( ) ).getTime( );

      // write code to exec pairs number of computationally intensive and disk intensive
      // so you will have pairs*2 threads

      // write code to wait for each test to finish (i.e., you need to join pairs*2 times)

      long endTime = (new Date( ) ).getTime( );


      SysLib.cout( "elapsed time = " + ( endTime - startTime ) + " msec.\n" );
      SysLib.exit( );
    }
}
```

Skeleton for Test3.java

What can do for computation intensive? A bunch of mathematical operations, for example:

```
for ( int i = 0; i < Integer.MAX_VALUE / 10; i++ )
      ans = Math.pow( Math.sqrt( Math.sqrt( i ) * Math.sqrt( i ) ), 2.0 );
```

What can you do for disk intensive? Many random accesses using rawread/rawwrite, for example:

```
byte[] buffer = new byte[512];
for ( int i = 0; i < 1000; i++ )
      SysLib.rawread( i, buffer );
```

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 UW1-320 machines include multi-cores, you need to increase X up to 3 or 4 for observing the clear difference between interruptible and busy-wait I/Os.**

```
[css430@uw1-320-20 ThreadOS]$ 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
```

5. What to Turn In

Submit to Canvas:

1. Part 1:
   - **Kernel_1.java** (which you have modified for Part 1)
   - **SyncQueue.java**
   - Any other java programs you coded necessary to implement SyncQueue.java (e.g., **QueueNode.java**)
   - Results when running Test2.class from Shell.class
   - Specifications and descriptions about your java programs (this goes in the report)
2. Part 2:
   - **Kernel.java** (which you have modified for Part 2)
   - **Test3.java** as well as **TestThread3a.java / TestThread3b.java** if created.
   - Performance results when running Test3.java on your Kernel_1.java, follow grading guide (implemented in Part 1)
   - Performance results when running Test3.java on your Kernel.java (implemented in Part 2)
   - Specifications and descriptions about your java program (this goes in the report)
3. Report:
   Submit as a **docx** or **pdf** file. Your report should include:
   - Specifications and descriptions about your java programs, Part 1 and part 2.
   - Discussions about performance results you got for part 2 and other relevant design, implementation and analysis decisions you have made.
   - Outputs: screenshots of results when running tests (in the Linux lab machines)

- Your report should have the professionalism expected from a senior class, e.g., Sections should be clearly indicated, there are titles, subtitles and any figures included are clearly explained and have suitable titles/captions.

Part 1 and Part 2 should be zipped into a ZIP file. Part 3 should be submitted separately (outside the ZIP).

The grade guide for this assignment: gradeguide3.txt

6. FAQ

This website could answer your questions. Please click here before emailing the professor :-)

*** RACE CONDITION:**
There is 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. Time depending, we will go over the details of the race in class but ***it is not required that you understand it for this this lab***.
Note:

- 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.
- However, if you want to avoid the race then you can download a new version the Disk class which has this fixed (see ZIP filebelow). In order to utilize this new Disk class you will also need a new SysLib and Kernel class.

ThreadOS_...

- As with all distributed Kernel.java's the 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.