

CS 5523 Lecture 17: Thread Synchronization

- Questions on laboratory 2 and laboratory 3
- Java synchronization
- Monitors
- Implementations of important synchronization problems
- Priority inversions and the Mars pathfinder

Threads introduce the following problems

- Race conditions – when program outcome depends on the exact order that different threads-of-execution execute statements
- Deadlock – when one or more threads-of-execution wait on a resource that is held by another waiting process with no hope of release
- Starvation – when a thread-of-execution receives insufficient processor resources to make progress

Java synchronization

- Every Java `Object` has a monitor
- The Java runtime system assures that code that is in the monitor is executed mutually exclusively.
- Methods and blocks that are modified by the `synchronized` keyword are in the monitor. Other code is not.
- Each `Object` monitor has a single lock and a single condition variable
- There is a queue for threads waiting to enter the monitor (obtain the monitor's lock. This queue is called the entry queue for the monitor)

Java monitor ownership

A thread becomes the owner of the object's monitor in one of three ways:

- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type `Class`, by executing a synchronized static method of that class.

Java condition variables

- One condition variable per object.
- The condition variable is implemented by a queue of blocked threads along with wait and signal operations.
- Java implements the wait and signal operations with the `wait`, `notify` and `notifyAll` methods.
- No particular condition is associated with the condition variable, so the code must retest the particular condition that its thread was waiting for.

Java `wait`

- The `wait` causes a thread to block on the queue represented by the condition variable. (The Java documentation indicates blocked means "disabled for scheduling purposes")
- The thread must own the object's monitor (be executing in a `synchronized` method or block)
- When thread is unblocked (returns from the `wait`), it competes for ownership of the object's monitor.
- The `wait` can throw three exceptions:
 - `InterruptedException` – thread interrupted by another thread while waiting
 - `IllegalArgumentException` – value of timeout was negative
 - `IllegalMonitorStateException` – thread tried to execute `wait` while not in monitor

Java `wait` (continued)

- The constructor for `wait` has three forms:

```
public final void wait() throws InterruptedException;
public final void wait(long timeout) throws InterruptedException;
public final void wait(long timeout, int nanos)
                    throws InterruptedException;
```

- When the thread returns from the `wait` due to being notified, throwing an `InterruptedException` or because the timeout expires, it owns the monitor. (What are the consequences of this?)

Draw a state diagram for a thread calling `wait`

Java `notify` and `notifyAll`

- The `notify` wakes up a single thread waiting on the object's monitor. The choice of thread is system-dependent.
- If no threads are waiting, nothing happens (the notification is lost).
- The `notifyAll` wakes up all threads waiting on the object's monitor.
- The thread must own the object's monitor (be executing in a `synchronized` method or block)
- The `notify` and `notifyAll` throws the `IllegalMonitorStateException` if it tries to call these methods without owning the object's monitor

Is the Java monitor a "real" monitor?

- Per Brinch-Hansen and C.A.R. Hoare invented monitors in 1970's:
"A monitor is essentially a shared class with explicit queues"
- To satisfy the shared class in Java, ALL fields must be private and ALL methods must be synchronized.
- To satisfy explicit queues, you need to be able to associate a queue with a particular condition, Java does not have this.

This design choice for Java has not made everybody happy (See the quote from Per Brinch-Hansen from Chapter 4 of High-Performance Java Platform Computing)

Where can you find out more about monitors?

- For an excellent (and extensive) survey of monitor implementations see:

“Monitor classification”

P. Buhr and M. Fortier, *ACM Computing Surveys*, 27(1), 1995, 63-107.

- If you want to experiment with monitors, Steve Robbins has a monitor simulator which is described in:

“Starving philosophers: Experimentation with monitor synchronization”

S. Robbins, *SIGCSE 2001*, 317-321.

(See <http://vip.cs.utsa.edu/nsf/sp/index.html> for the simulator or

<http://vip.cs.utsa.edu/nsf/pubs/starving/starving.pdf> for the paper)

Classical synchronization problems:

- **Producer – consumer:** producer generate items that are used by consumers. Generally, the producers and consumers run at different speeds so they interact through a synchronized buffer.

- **Reader-Writer:** any number of threads can access an item for reading at the same time, but only one thread can access the item when it is modifying it (writing):

- Strong writer – access given to readers only if no writers are waiting
- Strong reader – access given to writers only if no readers are waiting.

Look at implementations:

- Buffer pool: Examples 3.8, 3.9 and 3.10
- Binary semaphore: Example 3.13
- Counting semaphores: Example 3.16
- Barrier synchronization: Example 3.17
- The future synchronization: Example 3.18
- Single reader-writer monitor: Example 4.14
- Readers-preferred monitor: Examples 4.15-4.18
- Writer-preferred monitor: Examples 4.19-4.22
- Alternating reader/writer monitor: Examples 4.23-4.26
- Take-a-number synchronization: Examples 4.27-4.30

from *High-performance Java Platform Computing* by Christopher and Thiruvathukal

For next time:

■ *Finish reading for threads and OS. Do assigned problems from chapter 6.*
