Thread communication and synchronization

There are two main aspects to multithreaded programming in Java:

• thread lifecycle, and
• thread synchronization.

Thread lifecycle, which we considered last lecture, is usually the easier part.

Thread synchronization involves challenges like

• avoiding deadlock and starvation,
• preventing data corruption, and
• sharing resources effectively.

Need for synchronization: If threads comprise parts of our software systems, then they must communicate.

For two or more independent tasks to communicate, there must be:

• A medium for information exchange
• Synchronization between the tasks

In an OO system, typically a shared object plays both roles.

Suppose we have two threads in our bank program, each representing a separate teller. One is processing a check deposited in the customer’s account, while the other is processing a withdrawal by the customer.

Here is a set of actions that might occur:
\[ p_1: \text{balance} := \text{balance} + 10; \quad p_2: \text{balance} := \text{balance} - 5; \]

(1a) \( R1 := \text{balance}; \) \quad (2a) \( R2 := \text{balance}; \)

(1b) \( R1 := R1 + 10; \) \quad (2b) \( R2 := R2 - 5; \)

(1c) \( \text{balance} := R1; \) \quad (2c) \( \text{balance} := R2; \)

Assuming that the initial balance is 100, what are the possible ending balances?

Clearly, only one of these answers is correct.

To prevent this situation, there must be some way to prevent one transaction from starting while the other is in progress.

Java provides two ways to control concurrency:

- synchronized methods, and
- synchronized statements.

That is, the responsibility to protect an object from corruption can be placed on the threads via techniques like critical sections.

Or alternatively, responsibility can be associated with the object itself.

**synchronized methods:**

Let us first consider the case of synchronized methods.

Any method can be marked as synchronized.

If so, it is always executed in its entirety (unless explicitly suspended, for example, via a `wait`) before any other synchronized method of the object is allowed to start.

When a thread is executing a synchronized method on an object, the object is locked.

Another thread invoking a synchronized method on the same object will block until the lock is released; in other words, until
Here is an example of an `Account` class for a multithreaded environment.

```java
class Account {
    private String name;
    private float balance;

    public Account(float initialBalance) {
        balance = initialBalance;
    }

    public synchronized float balance() {
        return balance;
    }

    public synchronized void balance(float newBalance) {
        balance := newBalance;
    }

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }
}
```

Note that the `Account` constructor didn’t need to be synchronized. Why?

Also, note that the access methods are synchronized. This is another reason to write getters and setters instead of making the instance variable public.

If a subclass overrides a synchronized method, the overriding method can be synchronized or not.

The overridden method is still synchronized, however, so if the code invokes `super <method_name>`, the object will become locked.

If all methods in a class were marked synchronized, then an object of that class would behave like a `monitor`.
Informally, objects of a class that defines *synchronized* methods are said to be using a monitor. Note that each instance of a class has its own monitor.

In Java, only a subset of the interfaces on an object (its synchronized methods) are protected.

Each object (*java.lang.Object*) has an associated synchronization lock.

A thread invoking a synchronized method must acquire the object’s lock upon entering and release it upon leaving the method.

A synchronized method, once invoked will run to completion before any other thread gets a chance to run *any* synchronized method of the same object.

To investigate synchronized methods (and the subsequent guarded actions), let’s consider a bounded buffer.

The bounded buffer is implemented using a fixed-sized array. We need to have methods to insert an object, to delete an entry, and allow simple indexed access to support traversal.

Surely, a `BoundedBuffer` object should only be doing one of these things at a time, so each of these should be marked as synchronized.

```java
class BoundedBuffer {
    static private final int CAPACITY = 100;
    private String items[] = new String[CAPACITY];
    private int len = 0;

    public synchronized void add(String obj) {
        if (len >= CAPACITY) return;
        items[len] = obj; ++len;
    }

    public synchronized int size() { return len; }

    public synchronized String nth(int i) {
        if (i >= len) return null;
        return items[i];
    }
}
```
public synchronized void remove(String obj) {
    for (int i = 0; i < len; ++i) {
        if (items[i].equals(obj)) {
            items[i] = items[len-1];
            items[len-1] = null;
            --len; return;
        }
    }
}

Code for a **synchronized** method may call itself or another **synchronized** method of the same object.

Thus, the synchronization lock is per object, but is held per thread. What does this mean?

**Guarded actions**

In the above code, the **add** method does nothing at all if the bounded buffer is full.

In single-threaded code, you could raise some sort of exception in case of a full or empty buffer.

In concurrent code, however, it is also possible to wait until a subsequent **remove** is performed by some other thread.

This can easily be achieved by using **wait()** and **notify()**.

Recall that **wait()** and **notify()** are defined in class **Object**. They may only be called in **synchronized** methods.

Calls to **wait** and **notify** are normally self-calls, made within public synchronized methods of an object.

- A call to **wait()** can be made only by the thread that owns the object’s monitor.
• The thread releases ownership of the monitor, and waits until another thread notifies threads waiting on this object’s monitor to wake up.

It can do that either through a call to—

°

(What is the difference between the two?)

• The thread then waits till it re-obtains ownership of the monitor, and then resumes execution.

Like `sleep()`, `wait(long)` can also be called with one parameter, a timeout period in milliseconds. In this case, the thread waits either—

• until it is notified to wake up, or

• till the timeout period expires.

Again, the thread waits until it can regain ownership of the monitor.

These methods form only the raw basis for synchronization. To return to the bounded-buffer example, we could rewrite the methods as follows:

```java
public synchronized void add(String obj) {
    while (len >= capacity) {
        try { wait();
        } catch(InterruptedException e) {} 
    }
    items[len] = obj; ++len;
}

public synchronized void remove(String obj) {
    for (int i = 0; i < len; ++i) {
        if (items[i].equals(obj)) {
            items[i] = items[len-1];
            items[len-1] = null;
            --len; notify();  return;
        }
    }
}
```
As a general rule, condition waits should always be placed in
while loops, as in the code above.

The paradigm goes like this:

```java
synchronized void doWhenCondition() {
    while (!condition)
        wait();
    // Do what needs to be done when condition holds
}
```

This is because when the action is resumed, the waiting task
doesn’t know if the condition is actually true; it only knows that it
has been woken up. What could have happened?

Why is it safe to assume the condition is true if a while loop is
used?

Note that `wait()` *atomically* releases the lock on the object. This
means that immediately after the `wait()` is executed, the lock is
released. Nothing can intervene.

If these actions weren’t atomic, a race condition could arise.
Suppose a `notify` came along after the lock was released, but
before the thread was suspended.

Then the `notify` would have no effect on the thread. And the
thread would not reacquire the lock, raising the specter of
consistency problems.

The paradigm for notifying a thread goes something like this:
synchronized void changeCondition() {
    // Change some value used in a condition test
    notify();
}

More than one thread may be waiting on the same object.

- When a notify() is invoked, it wakes up the thread that has
  been waiting the longest.
- If a notifyAll() is invoked, all threads waiting on this object
  wake up.

*Monitor queues*

Consider this example. There is a random object manipulated
via synchronized methods.

- Thread 1 enters a synchronized method, and subsequently performs a
  wait.

- Thread 2 enters a synchronized method, and subsequently performs a
  wait. There is now a wait queue with
  1 in front of 2.

- Thread 3 enters a synchronized method (and then gets switched out
  by the scheduler).

- Thread 4 blocks trying to enter a
  synchronized method. Why?

- Thread 5 blocks trying to enter a
  synchronized method (can't acquire
  the lock).

- Thread 3 resumes, and issues a notifyAll(). What is the
  result?
• Thread 3 exits the synchronized methods.
• Thread 1 gets the lock and subsequently exits the
  synchronized method.

synchronized statement
In addition to synchronized methods, Java also has a
synchronized statement.

This statement specifies—
• an object to be locked, and
• a statement to execute when the lock is obtained.

The syntax is

```
synchronized (⟨expression⟩)
  ⟨statement to execute⟩
```

The ⟨expression⟩ must evaluate to an object. Usually it is just the
name of an object, e.g., myObject.

Usually there is more than one statement to execute, so a block is
used instead
(⟨statement1⟩; ⟨statement2⟩; … , ⟨statement n⟩)

A lock is acquired on myObject and then the block is executed.

The lock is released upon exiting the block.

Here is code that updates both words of a two-element array
indivisibly:

```java
int two [] = new int[2];
...
  synchronized (two) {
    if (two[1] == 2147483647)
```
```java
    {two[0] += 1; two[1] = 0;}
    else two[1] += 1;
}
```

This construct allows any Java object to act as a binary semaphore.

There is no requirement that the object of a `synchronized` statement actually be used in the body of the statement.

In fact, we could declare an `Object` just to control access to a critical section.

Thus, in Java, there are two ways to control access to critical sections—

• `synchronized` methods, and
• `synchronized` statements.

Which is better?  It depends on the situation.

For example, suppose you are presented with a class that works only for sequential code, not for multithreaded code.

You want to extend it to work in a multithreaded environment.  You can—

• subclass this class, using `synchronized` methods to override appropriate methods, then forward method calls using the `super` construct, or

• don’t subclass, but rather, take care to use objects of this class only within a `synchronized` statement.

Which is better?
Deadlock

A well known hazard of monitors is the nested monitor-call problem. It goes like this:

A process in one monitor $M_1$ calls a procedure in another monitor $M_2$.

The mutual exclusion in $M_1$ is not released while $M_2$ is executing.

This has two implications:

- Any process calling $M_1$ will be blocked outside $M_1$ during this period.
- If the process enters a condition queue in $M_2$, deadlock may occur.

Let’s look at an example in Java. A class called Friend has synchronized methods `are_you_there` and `im_here`.

The `are_you_there` method immediately invokes `im_here` on its caller.

Let’s suppose irving and edna are instances of Friend.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. irving.are_you_there(edna)</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>edna.are_you_there(irving)</td>
</tr>
<tr>
<td>3.</td>
<td>irving.im_here</td>
</tr>
</tbody>
</table>
| 4.                      | edna.im_here

Which locks are held at the end of each step?

- After Step 1, Thread 1 has a lock on
• After Step 2, Thread 2 has a lock on

• After Step 3,

• After Step 4,

Now, neither thread can proceed until the other releases a lock. So deadlock has occurred.

Of course, we could get lucky. There’d be no deadlock if one thread completed its call to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ before the other started its.

How can we fix this problem? The simplest fix would be to use a synchronized \textit{statement} instead of synchronized methods.

The two methods could synchronize on a single (lock) object shared by all \textit{Friends}.

What would be the disadvantage of this approach?

More sophisticated strategies could prevent deadlock without incurring this disadvantage.

\textbf{Interrupting threads}

It is possible to “interrupt” a thread. This doesn’t really stop the thread from doing what it’s doing.

Rather, it sets a flag that can be tested later by the thread.

For example, a display-update loop might need to perform a database transaction to get the information it needs to display.

Suppose the user tries to stop the update thread. The display thread still doesn’t want to stop until the database transaction completes.
This can be implemented as follows:

- **The user interface issues** `displayUpdate.interrupt()`.
- When a transaction completes, the display update always checks whether it has been interrupted, via `isInterrupted()`.
- If the display update has been interrupted, it stops (exits its update loop).

Although interrupting a thread normally doesn’t affect what it is doing, there are some exceptions.

Methods like `sleep` and `wait` throw `InterruptedException` if they are interrupted.

Of course, these exceptions can be caught, and appropriate action taken.