CS 520
Theory and Practice of Software Engineering
Fall 2022

Concurrent and distributed programs

November 8, 2022
Why use parallelism?

• Provide service for multiple requests, e.g.,

• Decompose a computation, e.g.,
  – Java Swing event handling (https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html)

• Collaborative work, e.g.,
  – Operating room team composed of multiple specialty teams

• ...
Distributed versus Concurrent Systems

• **Distributed system**
  – Usually supported by the operating system
  – Autonomous parallel processors that do not share memory
  – Heavy weight processor context switching

• **Concurrent system**
  – Usually supported by programming languages, libraries, or both (e.g., Java)
  – Autonomous parallel threads that do share memory
  – Lighter weight thread scheduling
Support for concurrency

• **Design modeling languages**, e.g.,

• **Programming languages**
  – Language feature, e.g., Java Thread (https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Thread.html)
Java Threads: Core Fields

- long getId()
- String getName()
- ThreadState.State getState()

```
public static enum Thread.State
    extends Enum<Thread.State>

A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action
  is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified
  waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.
```

A thread can be in only one state at a given point in time. These states are virtual machine
states which do not reflect any operating system thread states.
Java Threads: Core Methods

- `Thread(...)`
- `void start()`
- `void run()`
Java Threads: Core Methods

- Thread(...)
- void start()
- void run()

NOTE) The Java JVM creates a Thread to run the main method.
Example 1

public class Timer {
    public int ticks;
}

// Will have two Threads increment this Timer
Java Thread: Extends Thread

```java
public class IncrementThread extends Thread {
    private Timer timer;

    public IncrementThread(Timer timer) {
        this.timer = timer;
    }

    public void run() { this.timer.ticks++; }

    // new IncrementThread().start();
}
```
Java Thread: Implements Runnable

**run**

```java
void run()
```

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

**See Also:**

`Thread.run()`
Java Thread: Implements Runnable

<table>
<thead>
<tr>
<th>run</th>
</tr>
</thead>
<tbody>
<tr>
<td>void run()</td>
</tr>
</tbody>
</table>

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

The general contract of the method run is that it may take any action whatsoever.

See Also:
Thread.run()

NOTE) The Thread class implements the Runnable interface.
public class IncrementRunner implements Runnable {
    private Timer timer;

    public IncrementRunner(Timer timer) {
        this.timer = timer;
    }

    public void run() { this.timer.ticks++; }
}

// new Thread(new IncrementRunner()).start();
public class IncrementRunner implements Runnable {
    private Timer timer;

    public IncrementRunner(Timer timer) {
        this.timer = timer;
    }

    public void run() {
        this.timer.ticks++; // Increment timer ticks.
    }

    // new Thread(new IncrementRunner()).start();
}
Potential thread interleavings (1)

- There are two IncrementThreads called T1 and T2.
- The Ti.timer.ticks++ corresponds to Ti.timer.ticks = Ti.ticks + 1;
Potential thread interleavings (1)

- There are two IncrementThreads called T1 and T2.
- The Ti.timer.ticks++ corresponds to Ti.timer.ticks = Ti.ticks + 1;
  // WRITE     // READ
Potential thread interleavings (1)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2.
Potential thread interleavings (1)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 1, write ticks 2

3. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 0, write ticks 1

4. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 0, write ticks 1
Potential thread interleavings (1)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 1, write ticks 2

3. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 0, write ticks 1

4. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 0, write ticks 1
Potential thread interleavings (1)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 1, write ticks 2

3. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 0, write ticks 1

4. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 0, write ticks 1

This illustrates a synchronization issue called a data race.
Support for synchronization

• **Monitor**
  – In Java, *synchronized* keyword
  – In java.util.concurrent.locks, **Lock** interface

• **Fork/Join**

• ...


Monitor: Overview

Goal: Mutually exclusive access to data

Thread 1 ... Thread n

Monitor
(at most one thread at the same time)
Monitor:
Java Object synchronized block

```java
synchronized (Ti.timer) {
    Ti.timer.ticks++;
}
```

NOTE) For Java Monitors, each Object is associated with a lock.
Monitor:
Java Object synchronized method

```java
public synchronized void increment() {
    this.ticks++;
}
```

NOTE) For Java Monitors, each Object is associated with a lock.
Potential thread interleavings (1)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 1, write ticks 2

3. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 0, write ticks 1

4. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 0, write ticks 1
Potential thread interleavings (2)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 1, write ticks 2
The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — Collection, Set, List, Map, SortedSet, and SortedMap — has one static factory method.

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

Each of these methods returns a synchronized (thread-safe) Collection backed up by the specified collection. To guarantee serial access, all access to the backing collection must be accomplished through the returned collection. The easy way to guarantee this is not to keep a reference to the backing collection. Create the synchronized collection with the following trick.

```java
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a Vector.

In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation. The following is the idiom to iterate over a wrapper-synchronized collection.

```java
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html
The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each collection interfaces — Collection, Set, List, Map, SortedSet, and SortedMap — has one static fact

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
```

Each of these methods returns a synchronized version of the specified collection. To guarantee serial access, you can use the following trick:

```java
List<T> myList = Collections.synchronizedList(new ArrayList<T>(myList));
```

A collection created with a wrapper is thread-safe. Remember, the wrapper is not a Vector.

In the face of changing collection state, it might be easiest to iterate over it and then compose into a single atomic statement:

```java
Collection<Type> c = synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

Which design pattern is being applied?

https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html
Java Util Concurrent:
ReentrantLock class

From javadoc:

“A reentrant mutual exclusion **Lock** with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.”

Java Util Concurrent: ReentrantLock class

Implements the **Lock** interface

**Core fields/methods:**
- protected Thread getOwner()
- protected Collection<Thread> getQueuedThreads()
- public void lock()
- public void unlock()
Java Util Concurrent:
ReentrantLock class

public class Timer {
    private int ticks = 0;
    private ReentrantLock lock = new ReentrantLock(true);

    public void increment() {
        this.lock.lock();
        try { this.ticks++; }
        finally { this.lock.unlock(); }
    }
}
Java Util Concurrent: ReentrantLock class

```java
public class Timer {
    private int ticks = 0;
    private ReentrantLock lock = new ReentrantLock(true);

    public void increment() {
        this.lock.lock();
        try { this.ticks++; }
        finally { this.lock.unlock(); }
    }
}
```

The fair locking policy helps to prevent a synchronization issue called starvation.
Potential thread interleavings (3)

1. T1.timer.ticks++ // Read ticks 0, write ticks 1
   T2.timer.ticks++ // Read ticks 1, write ticks 2

2. T2.timer.ticks++ // Read ticks 0, write ticks 1
   T1.timer.ticks++ // Read ticks 1, write ticks 2
Java Util Concurrent: Other Synchronizers

Five classes aid common special-purpose synchronization idioms.

- **Semaphore** is a classic concurrency tool.
- **CountDownLatch** is a very simple yet very common utility for blocking until a given number of signals, events, or conditions hold.
- A **CyclicBarrier** is a resettable multiway synchronization point useful in some styles of parallel programming.
- A **Phaser** provides a more flexible form of barrier that may be used to control phased computation among multiple threads.
- An **Exchanger** allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs.

Dining Philosophers: Problem

Dining Philosophers: Problem

Common synchronization issue known as deadlock

Dining Philosophers:
Possible synchronizations

Remote Method Invocation (RMI): Overview

**Goal:** Distribute Objects on different computers over the network

```
Timer.increment();
```

```
public increment {
...
}
```
Remote Method Invocation (RMI): Overview

**Goal:** Distribute Objects on different computers over the network

```java
Timer.increment();
```

**NOTE:** Pass input parameters and result by marshalling/unmarshalling them
Remote Method Invocation (RMI): Java RMI

• Java Remote Method Invocation (RMI) https://docs.oracle.com/en/java/javase/15/docs/api/java.rmi/java/rmi/package-summary.html

Middleware (e.g., CORBA)

From their website: “The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects. CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages, and operating systems. CORBA is essentially a design specification for an Object Request Broker (ORB), where an ORB provides the mechanism required for distributed objects to communicate with one another, whether locally or on remote devices, written in different languages, or at different locations on a network.”

http://www.ois.com/Products/what-is-corba.html
Middleware (e.g., CORBA)

1. Client sends a request to the Server.
2. Server receives a request from the Client.
3. Server sends a reply to the Client.
4. Client gets reply from the Server.
Other Java libraries for concurrency and synchronization

• Open MPI (Message Passing Interface) (https://www.open-mpi.org/papers/mpi-java-spec/)

• Apache hadoop (https://hadoop.apache.org)

• Scala programming language (https://www.scala-lang.org)

• ...

Advantages of concurrency

• Provide support for scalability

• Improve performance in terms of space and/or time
Disadvantages of concurrency

• Often a large number of thread interleavings

• May introduce non-determinism (e.g., blocking on locks)
Disadvantages of concurrency

• Often a large number of thread interleavings

• May introduce non-determinism (e.g., blocking on locks)

Thus many common synchronization issues such as data races, starvation, deadlock, ...