Ser321 Principles of Distributed Software Systems

5. Threads

- [Section 5.a](#) Background on Programming with Concurrency
- [Section 5.b](#) Threading in the Context of Runtime Environment
- [Section 5.c](#) Java Reflection API
- [Section 5.d](#) Threading in Java
- [Section 5.e](#) Shared Objects
- [Section 5.f](#) C# Uses Monitors
5.a Background on Programming with Concurrency

5.a.1 Motivation

- How does a Web Browser allow you to continue to browse pages while it is simultaneously downloading a large file?

- What does Word or another desktop publishing application do to allow you to edit a document while it is processing a large print request?

- How does a server, for protocols such as the Web (http), or Secure Shell (ssh), allow multiple clients to be simultaneously connected to the server?

- A similar problem exists at the application or program-level. That is, when multiple programs need to be active simultaneously. For example, how does a smartphone play music while you are using another application. Or, receive a phone call while you are creating a text message?

- Programming languages and platforms (operating system) often support different mechanisms. We’ll look at Threading in Java as one example.

- In Graphics and Game Development you’ll see another approach. In native mobile application development, Android and iOS support different approaches for app developers.
5.a.2 What Should You Get from this Section?

• Outcomes

- To know when and how to implement multi-threaded applications that can share objects and synchronize access to those objects.

- To be able to describe the difference between threading and concurrency, and how threads and other related program structures fit into the design of programming languages.

- To be able to use the Java API classes and methods involved with threaded programming. `java.lang.Thread` and `java.langRunnable`

- To be able to design and implement thread-safe data structures, and to understand the use of threading in user-interface programming.

- To describe the information and organization of the Java runtime environment including the interactions between the runtime and the elements of an executing Java program.

- To be able to use the Java Reflection API to explore aspects of classes that have been loaded into a Java runtime.
5.a.3  Readings and Index to Examples

- In *Java Network Programming and Distributed Computing* Text:
  - Chapter 7: Multi-threaded Applications

- In *Java Network Programming 4th Edition* Text:
  - Chapter 3: Threads

- Other Reference Material
  - The Java Tutorial, linked below, see *Essential Java Classes* --> Threads
  - [http://java.sun.com/docs/books/tutorial/](http://java.sun.com/docs/books/tutorial/)

- Example programs demonstrating threading
  - Java Reflection: [Section 5.c Java Reflection API](#)
  - Java: [Section 5.d.2 Creating Multi-Threaded Programs in Java](#)
  - Java: [Section 5.e.4 Java’s Solution to Shared Access](#)
5.b Threading in the Context of Runtime Environment

5.b.1 What is a Thread?

• An Execution context, sometimes called a lightweight process
  - A thread is a single sequential flow of control within a program
  - **Threads** have an instruction pointer and a data/object environment
    that the processor (Java virtual machine) uses to execute a program.

• In a multi-threaded program
  - Two or more threads (instruction pointers and data environments) are managed and seemingly active simultaneously
  - A program has a single starting and ending point (for all threads). A multi-threaded program as depicted in the Java Tutorial.
5.b.2 Runtime Environment Components

- **Activation Stack** - A store for locals (temporary variables/values) defined within a method, and a record of the current method call nesting of a program.

  - The stack contains an entry (frame) for each active method. When a method is called, a frame containing local storage for the method is pushed onto the stack. When the method returns, the frame is popped from the stack.

- **Code-cells** - Provide the byte-code (Java) or machine-code (C++/Objective-C) of the executable instructions for all methods in the program.

- **Heap Storage** - Where objects are stored: created, accessed, deallocated and garbage collected. Each time an object is created, a reference to it is placed either on the stack (as a local within a method), or within another object on the Heap.

- **Class Storage** -- In languages like Java and C#. Part of the heap is used to store runtime information about API and user-defined classes. The storage is accessible by user programs through the Reflection API.
5.b.3  Runtime Storage for an Object-Oriented Language

- Example Hello World program in a simplified object-oriented programming language. This program has a single class (Main) with a single method (main), which has no parameters and returns a boolean:

```java
class Main () extends Object () {
    def main () : Bool = {
        this = new Main();
        "Hello World".println();
        return true
    }
}
```

- Note that objects are created on the heap for both program defined objects and for program Classes. Class objects are created for both API classes and for Programmer-Defined Classes. For further information, see Section 5.c Java Reflection API
5.b.4 Runtime Storage for an Object-Oriented Language

- Principle components: Activation Frames, Heap, and Structure of Loaded Classes.

```
main, e1
T1:
T2:
T3:
T4:
This: Dynamic: nil
Return: system

Processor (Ip, Ep) (5, e1)

Heap
Object for Main
members: nil
parent:

Object for String
members: “Hello World”
parent:

Object for Object
members: nil
parent: nil

Object for Object
members: nil
parent: nil

Class Structure

Class for Main
parent:
members: nil
methods:
main - code cell 1

Class for String
parent:
members:
char * str
methods: toString ...

Class for Object
parent: nil
members:
methods: ...

Code Cells
1 enterMethod main, (4, *Object, nil)
2 newObj Main, T1
3 assignThis T1
4 newStringToTmp "Hello World", T2
5 dispatch T2, putln
6 popResultTo T3
7 newObjToTmp true, T4
8 return T4
```
5.5.5 Threads and the Runtime Environment

- Threads can be understood in the context of the Runtime Environment for an Object-Oriented programming language, by considering the ip,ep pair.
  - In Java, threads are objects and their storage is placed on the Heap.
  - Each thread object includes its own processor - instruction pointer (ip) and environment pointer (ep).
  - The Java Virtual Machine is responsible for managing the allocation of the physical processor(s) to threads. While a thread is running, the corresponding physical processor is advancing the thread’s instruction pointer and environment pointer thereby executing the program using a fetch, decode and execute model, similar to what you may have learned in a course on machine organization and assembly language programming.
  - Each time a new thread is created, in the program, a thread object is allocated and the JVM schedules it to run.

- As threads call methods, new entries are pushed onto the runtime stack, and a reference to thread’s prior frame is placed in the new frame.
  - As a result, threads have their own runtime stacks, and also are able to share information.
5.b.6 Issues With Multi-Threaded Programs

- **Synchronization** - “I can’t go past this statement until another thread reaches that statement”
- **Synchronizing access to shared data.** - Enabling multiple threads have access (safely) to the same data (object). **Shared Data**.
- **Concurrency**, parallelism and asynchronous behavior.
- **Deadlock** - When two or more threads are waiting for each other to finish before they can proceed. None are able to progress beyond the block.
- **Scheduling** and blocking operations, such as input and output, access to shared data, and synchronization.
- Underlying support for **implementing threads**
- **Performance**
- Is there a better language mechanism than threads to enable concurrent and asynchronous behavior in software?
5.c Java Reflection API

• **Class Loader:**
  - At runtime there is a storage structure which represents the details of all classes which are available at runtime. API-defined and program-defined.
  - Like all Java objects, this representation of class information is kept on the heap. It is used by the runtime system, and is also available to the program.
  - Classes (.class files) can also be loaded at runtime under program control. Some are loaded automatically, and others may be loaded at the discretion of the running program (the programmer can control).

• The class **java.lang.Class**
  - The class **Class** provides access to the runtime representation of all loaded Java classes. At runtime you can discover information such as class’s inheritance structure, any interfaces implemented, the methods supported and their parameters.
  - Collectively the information accessible through class **Class** is referred to as the **Reflection API**
  - Here’s an example of Java Reflection. See: **reflection.jar**
5.c.1 Reflection’s Use in Serialization

- Java Reflection supports query of
  - Inheritance structure,
  - Fields names and types of a class
  - Field values of an class object.

- These provide the ability to serialize an object with code that is not specific to the class being serialized.
  - Example that includes Json serialization using reflection in both Java and Swift.
    - reflectJson.jar
5.d Threading in Java

5.d.1 Programming: Extend Thread or Implement Runnable

• First approach: Extend the Thread class (java.lang.Thread)

```java
class PrimeThread extends Thread {
  long minPrime;
  PrimeThread(long minPrime) {
    this.minPrime = minPrime;
  }
  public void run() {
    //compute primes larger than minPrime
  }
}
```

- Create a Thread with:

```java
PrimeThread p = new PrimeThread(143);
p.start();
```

• Use this approach when your thread class doesn’t need to extend another class.
5.d.2 Creating Multi-Threaded Programs in Java

- Alternative: Implement the Runnable interface (`java.lang.Runnable`)

  ```java
  class PrimeRun implements Runnable {
      long minPrime;
      PrimeRun(long minPrime) {
          this.minPrime = minPrime;
      }
      public void run() {
          // compute primes larger than minPrime
      }
  }
  ```

- Create a thread with:

  ```java
  PrimeRun p = new PrimeRun(143);
  new Thread(p).start();
  ```

- Use this approach when your class can’t extend Thread.

- In a Graphical User Interface (GUI) application, threads are used to perform actions in the background.

  - Example of a thread class to select and copy a file: `fileCopy.jar`
  
  - Example of playing music in the background. Note that the threads communicate to signal when to stop: `musicThreadStudents.jar`
5.d.3 Other Threading Capabilities

- **Join**, and **interrupt** methods - join waits for a thread to complete.
  - `anotherThread.join()` //waits for `anotherThread` to complete.
- (class method) **sleep** -- suspend for a number of milliseconds
  - `Thread.sleep(2000);` //sleep for two seconds

- **Thread States**

  - **Priorities** - Java VM supports 1 to 10 lowest to highest priorities
  - **Volatile** modifier for attributes (don’t cache) multi-CPU one RAM.
5.e Shared Objects

5.e.1 Multiple Threads May Access the Same Data Objects

- From the Runtime Environment perspective.
  - Each active thread has a separate `run` method on the activation stack. This is similar to recursive calls to a method, in that each recursive call will create a new activation frame for the same method.
  - The heap may contain objects referred to (pointed at) by several activation frames or other objects.
  - When multiple `run` methods can get to the same object on the heap, then they may concurrently try to access or change that object.
- If an Object is accessible to multiple threads (called a **Shared Object**) then a special mechanism is needed to synchronize access to the object.
  - Monitors perform this duty.
5.e.2 Thread Safe Classes in Java

- Compare `java.util.ArrayList`, whose javadocs state:
  - Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.

- With `java.util.Vector`, whose javadocs state:
  - Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

Other similar collections exist in Java with the same difference. For example, consider `java.util.HashMap` and `java.util.Hashtable`.

- If multiple threads of a program simultaneously access a Vector, then their access to the vector is properly synchronized. This is not the case when multiple threads access an ArrayList.
5.e.3 Sharing Objects: Synchronization

- What is a **Shared Object**?
  - Multiple threads of single program may **access** or **change** the same object.

- When should access to shared objects be **controlled**?
  - When multiple threads only **read** the object - **no synchronization**.
  - If any thread **modifies the object** - **synchronization** is necessary.

- **Why must access be controlled** when some thread changes the object?
  - consider: `stack.push(item)`, for example
  - **multiple actions are necessary** to achieve consistent state, for example:
    - (1) increment a stack pointer;
    - (2) place the new item into the collection using the stack pointer.
  - If one thread is interrupted **after step 1, but before step 2**, then another thread may see an inconsistent stack state (visualize `a = stack.top()`).

- **Monitor** - Used by some languages. Locks protect a critical section of code
  - Each thread calls **Enter()** to get the lock before **push(item)**
  - Each thread calls **Exit()** to release the lock after **push(item)** completes
5.e.4  Java’s Solution to Shared Access

- Java Monitors - programmers don’t need to explicitly get and release locks.
  - Instead, use the synchronized modifier to acquire and release the lock.

```java
public class Stack {
    public synchronized void push (String item) {...}
    public synchronized String pop ( ) {...}
    public synchronized String peek ( ) {...}
}
```

- A **monitor** is associated with each object of a class that has **synchronized** methods. **Synchronized** methods get and release the **monitor’s** lock.

- A **monitor** is associated with each class having one or more **synchronized** static methods. Synchronized methods get and release the monitor’s lock.

- **Recursive** synchronized methods are fine (monitors are reentrant).

- When should you use a monitor (**synchronized**)? When there are two or more threads accessing the same object, and at least one method modifies the object data, then any method that either modifies or accesses the object data should use the **synchronized** modifier.

- See: [threadsShareData.jar](threadsShareData.jar)
  - read the source code in **SynchronizedThreads.java**
5.e.5 Conditional Monitors

- **Conditional Monitors** are used when: Multiple threads have shared access to an object, and a condition should control which thread gets the monitor.

- Example: A shared bank account. Use **Wait** and **Notify** to assure that the balance does not go negative:
  - Accept a **withdrawl** only when the balance exceeds the request.
  - **Notify** changes in the balance when **deposits** occur.

- Java handles this with **wait()**, **notify()**, and **notifyAll()**
  - see **java.lang.Object**
  - These methods are only called from within synchronized methods

- **Wait** - when a thread executes a wait:
  - The monitor that it was holding is released, and the thread blocks;
  - when awakened by a notify - reacquires the monitor and continues

- **Notify(All)** - when a thread executes a notify(All):
  - signals a (all) thread(s) waiting on the same monitor to continue,
  - **notify** does not release the monitor. The monitor is released upon completing the method.
5.e.6 Example of Conditional Monitors in Java

- Java Conditional Monitors in a Banking Application
  We don’t want to accept withdrawals unless there is sufficient balance to support the withdrawl
  Try blocks are not included for simplicity

```java
public class Bank {
    private float balance;
    public synchronized void withdraw(float amount) {
        while (amount > balance) wait();
        balance = balance - amount;
    }
    public synchronized void deposit(float amount) {
        balance = balance + amount;
        notify();
    }
    public synchronized float balance() {
        return balance;
    }
}
```
5.e.7 Threads in User-Interface Programming

- Threads or other concurrency features are used in User-Interface (UI) programming to avoid freezing-up the UI during long-lasting activities.
  - Access a Web page, access a database, download a file, a JsonRPC call.
- Java Swing uses a threaded model in which user-generated events are handled on a separate thread called the Event Dispatch Thread (EDT).
  - For example, when actionPerformed is called to handle a button click, or menu item selection, it executes on the EDT.
  - The app controller should not perform long-lasting activities on the EDT. Doing so interferes with handling events for other user actions.
  - Swing provides the class javax.swing.SwingWorker which implements runnable. In addition to api javadocs, for more information see: http://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html
  - SwingWorker runs the method doInBackground on a separate thread, and then after it completes, runs the method done on the EDT.
  - doInBackground does the long-lasting or blocking activity
  - When doInBackground completes its result is available to done.
  - An example of SwingWorker in the file browser app: edtDemo.jar
5.f C# Uses Monitors

- A Monitor protects access to the Object
  - See `System.Threading.Monitor` class
  - All `Monitor` methods are static

- Basic `Monitor` methods to lock and unlock an object
  - `void Enter(object)` and `bool TryEnter(object, TimeSpan)` lock the object. A thread may be recursive (call multiple Enter’s before first Exit)
  - `void Exit(object)` indicate the calling thread is ready to unlock the object
  - Alternatively, a Thread may use the C# language construct:

```csharp
lock (object){ ... }
```

- **Java** uses monitors, but the programmer does not need to explicitly call `Enter` and `Exit` methods. Instead, use the `synchronized` modifier in defining methods.
  - `public synchronized void push(String item)`
  - The stack object’s monitor (lock) will be acquired before push or pop
5.f.1 Conditional locking in C#

- Some shared objects need the ability to conditionally lock an object based on the state of the shared structure.
- Consider a Stack that is implemented with finite storage
  - Want to accept (and lock) for a **push** only when the **stack isn’t full**.
  - Want to accept (and lock) for a **pop** only when the **stack has storage**.
- C# handles this with **Wait()**, **Pulse()** and **PulseAll()**
  - Static methods in the **System.Threading.Monitor** class
  - These are only used only within a lock (or after an **Enter**)
- **Wait, Pulse and PulseAll**
  - When a thread executes a **Wait**, the lock is released and the thread blocks (optionally, for up to a timespan). When awakened by a **Pulse** (or time expiration), the thread reacquires the lock and continues.
- In Push:
  - **lock** { if (stack_is_full) {Wait(this);} addTheItem; Pulse(this); }
- Correspondingly in Pop
  - **lock** { if (stack_is_empty) {Wait(this);} removeTheItem; Pulse(this); }