Agenda

- Project 5 Questions
- Multithreading & Synchronization Introduction
  - General Intro
  - Java specifics
Concurrent Programs

- Concurrency in real-world:
  - Complex systems & tasks → collection of simpler/smaller sub-tasks
  - E.g. House-Construction → brick-laying, carpentry, plumbing, wiring etc
  - Do all activities need to occur sequentially?

- Concurrency in Programs
  - Computer programs → collection of sub-tasks (methods)
  - Sequential program → sub-tasks DO NOT overlap in time
  - Concurrent → sub-tasks occur in parallel

- Concurrency → means overlapping in time
How can Programs be Concurrent?

- **Process**: Definition \(\Rightarrow\) A process is a program in execution: can be unit of CPU utilization
- Different processes can be *in execution* in the same program
- Concurrent program \(\Rightarrow\) modeled using multiple processes: a process per sub-task
- Multiple CPUs required?
  - Not really. On single-CPU *interleaving*
  - On Multiple-CPU *True concurrency*. 
Why care about Concurrent Program?

- When program needs to model (or interact with) concurrent real-world environment (e.g. house-building helper-program?)
- True concurrency increases throughput (amount of work done per unit time)
- More interactivity (response to user), as other actions done in other processes.
- E.g. Graphical UI’s need to repaint multiple windows in response to different events (AWT , Swing ?)
- E.g. Simple chat application : needs to respond to local-user & wait for data from remote-user
Multiple processes can always produce concurrency

Disadvantages: due to isolation between processes
- Process Creation & State maintenance Costly
- Sharing, Communication Costly

Thread:
- Often called Light-Weight-Process (LWP)
- Thread (like process) represents a portion of code in execution
  ➔ A flow of control, many flows at same time possible
- Not isolated like process: Related threads share code-segment, data-segment, OS resources (open files, signals etc)

Advantage: Because of this sharing in Threads,
- Switching & Communication less costly
Threads in Java

- Java multi-threading support at language-level (means language recognizes threading)
- Many other languages don’t → C, C++ ...
  - Does not mean multi-threading not possible in C/C++. Library support.
- No standard implementation → JVM specific
  - Is Java thread mapped to OS thread or process
  - Or does JVM do its own time-sharing?
- Garbage Collection is a Thread → So all Java programs threaded!
Threads in Java (continued…)

- **Class java.lang.Thread**: creation, initialization & control of threads
- **Interface java.lang.Runnable**: interface implemented by ANY class requiring threading (including Thread)
  - Runnable has single abstract method: `run()`
  - `run()` method: supposed to carry the code we want to run as a thread
- **Thread**: has a dummy `run()` method
public interface Runnable {
    abstract public void run();
}

public class Thread implements Runnable {
    ...
    public void run() {
        // dummy implementation of run
    }
}

The run() method is the actual code that will be run by the thread. We need to make run() use the code we want. We will see how.
Thread Creation in Java: 2 ways

- Method 1: ‘extending’ Thread class
  - Class MyThread extends Thread {
    
    String str;
    public MyThread(String str) {
      this.str = str;
    }
    public void run() {
      //repeatedly print str
    }
  }
  
  MyThread t1 = new MyThread("Howdy");  //creation
  t1.start();  //starting it
Thread Creation in Java: 2 ways

- Method 2: *implementing* Runnable interface
  - Class MyRun implements Runnable {
    - String str;
    - public MyRun(String str) {
      - this.str = str;
    }
    - public void run() {
      - //repeatedly print str
    }
  }
  - MyRun r1 = new MyRun("Bye");
  - new Thread(r1).start();
Thread Creation in Java: 2 ways

- We want to use Thread as it already has the threading functionality.
- Method 1 uses single-inheritance from a concrete class (Thread) & over-rides run().
- But what if your class already needs to derive from another concrete class? → multiple inheritance!
- Method 2 avoids multiple inheritance by using a type of containment (r1 is inside Thread now & can use / be used by Thread).
- Thread’s run() method is not that dummy.
Java Thread Lifecycle

- Lifecycle tracks the different stages from Thread creation to termination

- States:
  - “New” (Just created)
  - “Dead” (Terminated)
  - “Running” (is currently running on JVM)
  - “Runnable” (not Running, but can be scheduled)
  - “Non-Runnable” (cannot be scheduled Running)

- Thread has functions that can move a Thread from one state to another: start, sleep, yield, suspend, resume etc (see javadocs.org)
Java Thread Lifecycle (continued ...)

New Thread → start → Runnable

Running

yield

Runnable → Not Runnable

The run method terminates

Dead
Synchronization: Sharing & Interference

Sharing:
- Two threads can communicate & cooperate using a ‘shared-object’ (e.g. even an integer)
- One thread can write out data to the object, and another thread can read → communicate
- Multiple threads can use many shared-objects

Interference:
- Remember ‘interleaved’ operation of threads? → incorrect (or unexpected) state can be produced
- Simple example: value += 1; assume value=20 before.
- If two threads sharing value & perform above will value = 22 afterwards? Not necessarily!
Race Condition

- `value += 1; //java statement`
- Translated into asm may look as follows:
  - Read value into register
  - Increment register
  - Write register to value
- A single java statement can be many low-level statements.
- Thread can be interrupted/rescheduled between any of above statements
- `value += 1; IS NOT ATOMIC`
- Which Thread wins the race, decides out-come
Interference: Large state (continued…)

- Assume two threads A & B share an AVL-Tree! Both threads input data into AVL-Tree (to speed up)
- AVL-Tree `insert()` can consist of 100’s of java statement & 1000’s of low-level statements

- The state of Tree is ‘correct’ only BEFORE or AFTER `insert()`. NOT during `insert()` (i.e not when `insert()` is changing around references)
- Threads can be rescheduled (interleaved at any point)

Therefore two `insert()`s should NEVER be interleaved.
- An `insert()` with reference to ANY other Tree operation (on same tree) must be considered ATOMIC.
Synchronization in Java

- Mutual Exclusion is a solution to Interference.
- Mutual-Exclusion means that ONLY a single thread will be allowed access to some shared resource (can even be code – critical-section)

- Java provides keyword ‘synchronized’
- 2 ways to use ‘synchronized’
  - synchronized method1() { ....//the method }
  - synchronized(object-x) { ....//any code }
Synchronization in Java
(continued)

- 1st Method:
  - synchronized increment_value() {
    - value += 1;
  }
  - Mutual-Exclusion at method-level.
  - Does not explicitly protect every individual object in function
  - Code is shared resource here (of course the code uses shared-variable & is hence synchronized)

- What if we have:
  - synchronized int read_value() {
    - Return value;
  }
  - Does this synchronize access to value? Yes. No other synchronized method is accessible.
Synchronization in Java
(continued)

- **2nd Method:**
  - `increment_value()` {
    - `synchronized(value) { //can only be used with Objects, not ‘ints’. Just for e.g. here.`
    - `value += 1;`
    - `}
  }

- Mutual-Exclusion at shared-object level. Only single thread can get to value provided all threads do `synchronized(value).`
- Explicitly protects value

- Previous method is considered safer, than this one → any code that doesn’t do `synchronized(value)` will break everything.

- But gives finer grained locking (i.e not all synchronized methods share the same data!)
- Note: Locks not *self-blocking* → recursive synchronized methods
Deadlocks: Locking can lead to...

- Thread A:
  - Control-Flow:
  - 1. read from var1
  - 2. perform computation
  - 3. write to var2

- Thread B:
  - Control-Flow:
  - 1. read from var2
  - 2. perform computation
  - 3. write to var1

- How can we synchronize access to shared-objects var1, var2?
Deadlocks (continued …)

- **Thread A:**
  - `synchronized(var1) {
    1. read from var1
    2. perform computation
    `synchronized(var2) {
      3. write to var2
    }
  }

- **Thread B:**
  - `synchronized(var2) {
    1. read from var2
    2. perform computation
    `synchronized(var1) {
      3. write to var1
    }
  }

- Suppose Thread has locked var1 & is in step 2. Thread B has locked var2.

- ➔ A will start waiting for var2 (held by B) & B will wait for var1 (held by A)
One way to avoid deadlocks is to \textit{Strictly enforce order in which locks will be taken.}

\textbf{Thread A:}
- \texttt{synchronized(var1) \{
  1. read from var1
  2. perform computation

  \hspace{0.5cm} \texttt{synchronized(var2) \{
    3. write to var2
  \}}}

\textbf{Thread B:}
- \texttt{synchronized(var1) \{
  1. read from var2
  2. perform computation

  \hspace{0.5cm} \texttt{synchronized(var2) \{
    3. write to var1
  \}}}

Always lock var1 before var2. B cannot get in until A leaves. No Deadlock.
Multithreading & Synchronization

Conclusion

- This was only an introduction → There is a lot more to getting even moderately complex multithreaded programs working correctly.
- But this should give you the basic ideas & tools to start playing with Thread, Runnable & ‘synchronized’ keyword in Java.
- See any OS book for general synchronization problems
- Questions?
References

- Concurrency : State Models & Java Programs. Magee & Kramer.
- Operating System Concepts. Silberschatz & Galvin
- javadocs.org
- http://java.sun.com/docs/books/tutorial/essential/threads