Java Programming 2
Lecture #15

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
11 November 2019

"juice" by Lindsey Turner
https://flic.kr/p/4npCkA

Outline

Concurrent programming

Threads in Java

Concurrent programming: basics

Process

- Self-contained execution environment – own memory space
- Often synonymous with programs or applications – however, an application may consist of several processes (e.g., Google Chrome)

Thread

- Lightweight processes: shared memory space
- Every process has at least one thread

https://www.howtogeek.com/124218/why-does-chrome-have-so-many-open-processes/

Why use concurrent techniques?

Tasks can be divided into subtasks; subtasks can be executed in parallel

Theoretical possible performance gain (Amdahl's Law):

If F is the percentage of the program which cannot run in parallel and n is the number of processes, then the maximum performance gain is $\frac{1}{F+\left(\frac{1-F}{n}\right)}$

Why NOT use concurrent techniques?

Threads can access shared data – two main potential problems

Visibility

Thread A reads shared data which is later changed by thread B without thread A being aware of the change

Access

Several threads access and change shared data at the same time

Can lead to

Liveness failure – program does not react any more due to problems in shared access

Safety failure – program creates incorrect data

Concurrent programming in Java

Java uses **multithreaded** programming within a single process

Basic building block: **Thread** class

Useful helper package: **java.util.concurrent** (since Java 1.5)

Note: all work we have done so far is taking place in the context of a single main **Thread** object – can be accessed and manipulated just like other Threads

Creating a Thread

Preferred technique: implement the **Runnable** interface and define **run()** method

(Can also subclass Thread but not as flexible/general)

Create thread based on **Runnable** class and use **Thread.start()** to start it

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from a thread");
    }

    public static void main(String[] args) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
    }
}
```

Based on https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Pausing a Thread

Use **Thread.sleep()** to pause execution for a specified period

- Lets other threads use system resources
- Wait for something time-dependent to finish

Duration can be specified in milliseconds or nanoseconds – not guaranteed to be precise, depends on underlying OS and might be interrupted

Thread.sleep() throws **InterruptedException** (checked) – indicates that another thread interrupted the sleep and the thread should terminate

- *If you do not have another thread that can interrupt, you can usually just ignore this exception*

Thread.sleep() example (main thread)

```
String[] importantInfo = {
    "Mares eat oats",
    "Does eat oats",
    "Little lambs eat ivy",
    "A kid will eat ivy too" };

for (String info : importantInfo) {
    // Pause for (approximately) 4 seconds
    Thread.sleep(4000);
    // Print a message
    System.out.println(info);
}
```

Waiting for a thread to terminate

Use **Thread.join()** method

Can also throw **InterruptedException** – must be caught

You should do this at the end of any multithreaded program to be sure it terminates

Interrupting a Thread

From outside: call **Thread.interrupt()** on the **Thread** object

Inside the **Thread**

- If you call a method that could throw **InterruptedException** (e.g, **sleep**(), **join**()), just return after it is caught
- Otherwise, periodically check **Thread.interrupted()** and return if it is true

SimpleThreads example (Java tutorial)

https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html

Thread interference details

Interference happens when two operations running in different threads but on the same data **interleave**

Two operations have multiple steps, and the steps overlap

Seemingly simple statements can translate to multiple steps in the virtual machine:

`i++` turns into:

1. Retrieve the current value of `i`
2. Increment the retrieved value by 1
3. Store the incremented value back into `i`

Example

```
public class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Thread interference

Possible sequence with two threads both accessing memory:

1. Thread A: Retrieve `i`
2. Thread B: Retrieve `i`.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in `i`; `i` is now 1.
6. Thread B: Store result in `i`; `i` is now -1.

Avoiding interference: impose an ordering

Establish a **happens-before** relationship between two statements

Actions that create **happens-before**:

- Every statement before a **Thread.start()** happens before every statement executed by that thread
- When a thread terminates and causes **Thread.join()** to return, every statement in the terminated thread happens before every statement following the join

```
int counter = 0;

counter++;

System.out.println
    (counter);
```

Synchronized methods

Additional keyword:
`synchronized`

Add to method header

Ensures that:

- Two calls to **synchronized** methods **on the same object** cannot interleave
- When a synchronized method exits, it **happens-before** any other **synchronized** method calls **on the same object**

Constructors cannot be synchronized

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Under the hood: **Intrinsic locks**

Every Java object has a lock associated with it

A thread that needs consistent access to an object fields must **acquire** the lock before access, and **release** the lock when it is done

- In between, the thread **owns** the lock – no other thread can acquire it (will block on attempt)
- Note that a thread can access the same lock multiple times (**re-entrant**)

Synchronized methods make implicit use of the lock

More fine-grained option: **synchronized statements**

Synchronized statements example

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 =
      new Object();
    private Object lock2 =
      new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Atomic access

Atomic action

Effectively happens all at once – cannot stop in the middle

Reads and writes are atomic for most types (except long/double)

Increments like c++ are **not** atomic

Avoids need for synchronized code

Liveness problems

Liveness: concurrent program's ability to execute in a timely fashion

Potential problems:

- **Deadlock**: two or more threads are blocked forever, waiting for each other
- **Starvation:** a thread cannot gain access to a shared resource and is unable to make progress
- **Livelock:** threads too busy responding to each other to make progress

Deadlock is by far the most common problem

Next time

Higher-level concurrent API

Immutable classes

Annotations