Concurrency Issues in Object-Oriented Design

Part I

Why Concurrency?
- For more responsive systems
- For increased performance via parallelism

Why not concurrency?
- It brings non-determinism
- Specific knowledge and programmer discipline needed
The Process and the Thread

Process

- A program is static (dead)
- A process is dynamic (alive)
- The process is the unit of work in modern operating systems
Origin of the OS Process

Aspects of an Operating systems (OS)

A real operating system has four major parts or subsystems:
- input/output,
- filesystem,
- memory management
- process handling
  - At the very least, a "real" OS includes some form of multitasking
Multitasking OS

- A multitasking operating system requires more administration than a single-tasking OS.
- A process, or task, can be seen as a set of allocated resources: memory pages, swap pages, open files, and registers if the task is active.

A process must..

- Get some kind of ID
- Be created: have memory allocated to it
- Be given the CPU for execution
- Be removed: memory reclaimed, resources freed

On a CPU only ONE process can be running at one time.
What does a process know?

- What program it’s running
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

A Process “Maintains State”
My brain is my PCB

Process Control Block (PCB)

CPU Process Switching
The Process “Scheduler”

- The OS process scheduler assigns the CPU to a process
- Early OSs relied on each task to give back the CPU – called cooperative multitasking
- This evolved into preemptive multitasking – where the scheduler takes back the CPU
- Preemptive is preferred, as cooperative multitasking fails when a single process forgets or is unable to relinquish control of the CPU. If such a scenario occurs, the computer is "blocked".

What about Threads?
Threads vs Processes

- A process has its own address space
  - In a multitasking operating system, each program is run as a separate process
  - Process switching has overhead
  - A thread shares the address space of the program that created it
  - Minimal overhead with thread switching

Why Threads?

- Provides parallel computation with low overhead
- Use threads when a program may need to wait for some resource
  - Disk access, network connection
- When one thread is waiting, other can continue processing
Threads

- Each thread has its own program counter and call stack
- Multiple threads will share the process address space which may contain global variables which can be used for communication among the threads.

Threads are useful...

- can take advantage of multiple processors
- get work done while other threads wait for slow devices
- respond to multiple user inputs
- provide network services
Three Categories of Thread Usage

- Independent threads
  - each doing a part of an overall task
  - no interference
  - issue: how wait on all threads to complete
- Data sharing threads
  - each may interfere with data
  - issue: how to guarantee safety & liveness
- Coordinating threads
  - how to structure integrated coordinated action

Key Properties for Concurrent Software

- Safety –
  - the property that nothing bad ever happens
  - bad = random, unexpected behavior
- Liveness
  - the property that the program will make progress (and not stall)
Race conditions

A *race condition* is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled.

Race conditions can lead to unpredictable results and subtle program bugs == UNSAFE programs

Thread Safety vs. Liveness

- **Problem:**
  - easiest things that improve liveness can destroy safety and vice versa
  - Getting both right is the challenge of concurrent programming
### Sequential vs. Multithreaded Programs

<table>
<thead>
<tr>
<th>Sequential Programs</th>
<th>Multithreaded Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory is stable</td>
<td>Memory may change under you (unless private, read-only, or protected by lock)</td>
</tr>
<tr>
<td>No locks needed</td>
<td>Locks essential</td>
</tr>
<tr>
<td>OK to access two data structures</td>
<td>If structures related, locks for both must be held</td>
</tr>
<tr>
<td>Deadlock can't happen</td>
<td>Deadlock is possible if multiple unordered locks</td>
</tr>
</tbody>
</table>

**Java Threads**
Life Cycle of a Thread

New Thread

Runnable

Dead

Not Runnable

Thread \( t = \) new Thread()
Thread \( t = \) new Thread(myObj)

How to make a thread NOT Runnable -
1. invoke it's sleep method
2. thread blocks on IO
3. a thread executes a wait() method

Subclass Thread

Thread class

has run() method that does nothing

inherits

MyThread

override run() - that's what your thread will do
Two ways to create your own thread

- Extend the Thread class and override the run() method
- Implement the Runnable interface
  - define run()

Either way, there is a thread object
whatever code is in run() will be run as a separate thread

Subclassing Thread

class SimpleThread extends Thread {
  private String internalName;

  SimpleThread (String name) {
    internalName = name;
  }

  public void run() {
    for (int i=0; i<5; i++) {
      System.out.println(internalName);
    }
  }
}
Activating your thread

The Java runtime system sends the run message

the programmer sends the start message to the thread object

public class SimpleThreadTest1 {
    public static void main(String argv[]) {
        SimpleThread t1 = new SimpleThread("sun");
        SimpleThread t2 = new SimpleThread("java");
        SimpleThread t3 = new SimpleThread("beans");
        t1.start();
        t2.start();
        t3.start();
    }
}

The Operating System

The “Runnable” pool of threads (Continued →)
class SimpleThread extends Thread {
    String internalName;

    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(internalName);
        }
    }
}

Hog Prevention with yield()

- Use yield to prevent a thread from hogging the CPU

public void run() {
    for (int i=0; i<8; i++) {
        System.out.println(internalName);
        Thread.yield();
    }
}
Threads and Runnable

class PrimeRun implements Runnable {
    long minPrime;

    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime . . .
    }
}

Three Categories of Thread Usage

- Independent threads
  - each doing a part of an overall task
  - no interference
  - issue: how wait on all threads to complete

- Data sharing threads
  - each may interfere with data
  - issue: how to guarantee safety & liveness

- Coordinating threads
  - how to structure integrated coordinated action
Partitioning Work with Threads

Three Work Threads

t1.start(); ➔ returns immediately
t2.start(); ➔
t3.start(); ➔
System.out.println(“done”);

will print before threads complete their work
Using join() to wait for another Thread

- public final void join(long millis) throws InterruptedException
  - Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

- public final void join() throws InterruptedException
  - Waits for this thread to die

Three Work Threads

```java
t1.start(); // returns immediately
t2.start(); //
t3.start(); //
t1.join(); // main thread will block until t1 is no longer alive
t2.join();
t3.join();
System.out.println("done"); // will print after all threads complete their work
```
Thread Priority

- Each thread is assigned a default priority of Thread.NORM_PRIORITY. You can reset the priority using setPriority(int priority).

- Some constants for priorities include
  Thread.MIN_PRIORITY
  Thread.MAX_PRIORITY
  Thread.NORM_PRIORITY

Three Categories of Thread Usage

- Independent threads
  - each doing a part of an overall task
  - no interference
  - issue: how wait on all threads to complete

- Data sharing threads
  - each may interfere with data
  - issue: how to guarantee safety & liveness

- Coordinating threads
  - how to structure integrated coordinated action
Threads II
Safety, Liveness
and Thread Synchronization

Safety

- Similar to the notion of type safety
  - A typed –checked program may not be correct but it never encounters errors due to corruption of representation
- However, types can be checked at compile time
  - safety requires programmer discipline
Safety

- Goal: ensure all objects maintain consistent state – i.e. legal and meaningful (although this is sometimes hard to nail down)
- Safe objects may enter transient inconsistent states but never attempt to perform actions when in such states
- How can we create safe objects?

Safe Strategy
Synchronized Objects

- When visible objects change state, synchronization is necessary to ensure that changes occur only in consistent ways
- Safety preservation requires the avoidance of two kinds of conflicts that might occur
  - read/write conflict
    - reading values that are transient / inconsistent
  - write/write conflict
    - interleaving writes – e.g. ++i
      - start with 0, end result may be 1 or 2
**Solution**

- Locate a free seat
- Assign seat to customer

### Java: synchronized methods

- Only ONE thread can execute - lock other threads out
- Object is locked - only one thread can execute synchronized code at a time

```java
Seat [] seats = ...;
boolean synchronized assignSeat () {
    1. locate free seat
    2. assign seat to customer
    thread1
}
```
synchronized methods lock other threads out

only one thread can execute synchronized code at a time

object is LOCKED -

Seat [ ] seats = ....;

boolean synchronized assignSeat () {
    1. locate free seat
    2. assign seat to customer

thread1 holds the lock to the object

class Account {
    private double balance;

    public Account (double initialDeposit) {
        balance = initialDeposit;
    }

    public synchronized double getBalance () {
        return balance;
    }

    public synchronized void deposit (double amount) {
        balance += amount;
    }
}
object is locked until the synchronized block has completed

synchronized boolean foo() {
    //code goes here
}

synchronized boolean bar() {
    //code goes here
}

t1

other threads wait to execute a.foo or a.bar until t1 completes

ok to execute non-synchronized
Objects & Synchronized Methods

foo1
- synchronized methods
- non-synchronized methods
- data

foo2
- synchronized methods
- non-synchronized methods
- data

instance of Foo

```
foo1
  "i own the lock"
  all synchronized methods of this object are unavailable to other threads
  data

foo2
  data

instance of Foo
```
"I own the lock to this object"

foo1
- synchronized methods
- non-synchronized methods
- instance of Foo

foo2
- synchronized methods
- non-synchronized methods
- instance of Foo

Other objects have their own locks

foo1
- data
- instance of Foo

foo2
- data
- instance of Foo

Non-synch methods are not affected
Synchronized Block

- Allows locking an object without a synchronized method
- Form:

```java
synchronized (<some object>) {
    statements;
}
```

some object to lock

What about variables?

- Variables cannot be synchronized
- However you can control access if you:
  - declare variables private
  - provide accessor methods

```java
class Account {
    private double balance;
    synchronized void updateBalance (double amt) {
        balance += amt;
    }
}
```
Synchronized Summary

- Synchronization permits multiple threads to act concurrently without interfering with each other.
- Java’s synchronized methods use LOCKS to control thread access to code.
- Finer granularity can be obtained using the synchronized block within a method.

Cost of Synchronization

- The simple tests suggest that an uncontended synchronization is cheaper than the cost of an object creation or a HashMap lookup.
- see:
The real performance impact of synchronization is felt when threads actually contend for a lock.

The cost difference between an uncontended synchronization and a contended one is huge.

Simple test programs suggest that a contended synchronization is 50 times slower than an uncontended one.

This fact combined with the observations drawn above suggests that a contended synchronization is comparable in cost to at least 50 object creations.

End Part I (Session 3)
Concurrent OBJECT-ORIENTED Design

CSE8313