Serial, Concurrent, Parallel

Threads

What is “concurrent” as opposed to “serial”?

Serial (or sequential) programs:
- have a single “thread” of control
- basically, assuming the compiler doesn’t play with things, the flow of execution matches the code you’ve written

Concurrent programs:
- have (literally) multiple threads of control
**Concurrent versus Parallel?**

*Concurrent programs:*
- have “logically” simultaneous processing models
- does not automatically imply multiple physical processing elements (PEs)
  - ie: things might not actually happen simultaneously or on different chips or on different machines

*Parallel programs:*
- are assumed to actually involve multiple, possibly independent PEs
- just about all things really can happen simultaneously

We will generally ignore this distinction from here on out unless explicitly stated otherwise.

---

**Benefits of Concurrency: Fitting your problem**

There are algorithms that just “naturally” fit a concurrent programming model, often when the state (data) can be partitioned well:
- some simulations
- some divide and conquer problems
- server applications

If you want to isolate and simplify your tasks, trying to think about things as small modules might help, and if they don’t directly depend on each other (or don’t for a while) then you might easily go with a concurrent model.
**Benefits of Concurrency: Using available power**

*Exploiting the power of multiple processors!*

– to “work around” the limits of trying to speed up individual processors:
  - more machines were given multiple CPUs
  - common CPUs started to move to multi-core

– without concurrent software, how could you take full advantage of this?
  the answer is… you really couldn’t…

---

**Benefits of Concurrency: Servers**

*If you wanted to have a server application that interacted with multiple clients, how would you do it?*

– non-concurrent implementations have to explicitly multiplex the interaction with different clients
– concurrent servers can simply handle each client in a separate thread of control
  - even if this is done on a single-processor machine, you are handing the management of the different threads over to the operating system rather than having to orchestrate it yourself
  - a lot of work and expertise has gone into writing operating systems, so let them do their job

Example: Have a server thread make a system call to accept a connect and then have the thread block until it gets one, then have the server thread make a system call to read data and have it block until it gets data, repeat last step…
**Concurrency Risks**

These risks come from having multiple observers and actors interacting with a shared state.

Yes, concurrency is notoriously complex, largely due to the fact that system behavior depends on the specific timing of events and the timing/order can change from execution to execution (in other words, bugs can be hard to reproduce).

**Over the next few weeks we will discuss**
- what specific difficulties arise from concurrent programming (some of which are issues with sequential programming as well, but are just magnified here).
- how we can follow certain practices to work to build and test concurrent systems that operate correctly.
- how operations that look atomic might not be (x++; any math on a 64 bit number when run on a 32 bit architecture)

---

**Implementing Concurrent Systems**

What are threads?
- Conceptually?
- As available via standard Java mechanisms?
  - How do different approaches look in Java code?
  - What are the differences on how the threads act and interact?

What are some major thread concerns?
- Scheduling issues and consequences in general
- Safety of shared data
- “Liveness” (deadlock, starvation, livelock)

What are some common methods in the Thread class?
Some Computation Abstraction Concepts

• hardware threads
• software threads
• interrupts
• operating system (single task versus multiple)
• virtual machine (from the view of the process)
• multiple hardware threads

Computation Abstractions (I)

Processes (p1, p2, p3, p4)
in our case could be JVMs

A dual-CPU computer with processes whose threads aren’t crossing processors.
Computation Abstractions (II)

Processes (p1, p2, p3, p4) in our case could be JVMs

Threads
- p1’s threads t1, t2
- p2’s threads t1, t2, t3
- p3’s thread t1
- p4’s threads t4, t5 (we can assume its t1, t2, t3 already ended)

A dual-CPU computer with processes whose threads are crossing processors.

Sharing data: Processes vs. Threads

Processes do not directly share data.

Threads within a single process can directly share data (y) but can also have unshared data (z).
**Java Memory Model**

**Stack**
- Local variables
- Method parameters

**Heap**
- Objects
- Every call to `new` allocates space on heap

Class-typed variables reference heap or null

---

**Two Single-Threaded Processes**

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Main memory</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IPC

---

9/5/2012 ©2012 University of Maryland
What Is a Thread?

Conceptually: within a multi-threaded process, it can be thought of as a parallel track of computations occurring within that process.

Implementation view: internally the “thread” is really just a program counter and a stack that shares the heap and any statically allocated data with all the other threads.

In reality, all programs have at least one thread; the “main” thread. Often that’s the only thread there is.

One Multi-threaded Process
Side Question: Sharing a File System

What would happen if you had two processes on a machine and the first was reading a file and the second issued an instruction to delete that file?

What would happen if one process was reading a file and the other wrote to the end of that file in the middle of things?

Why utilize multiple threads?

Performance:
- Parallelism on a multiprocessor machine.
- Concurrency of computation and I/O on either a single or multiprocessor machine.
  - assumes direct memory access I/O

Can easily express some programming paradigms
- Event processing
- Simulations
- Servers

Keep unrelated computations separate, as in an OS
- Why not use processes for things other than the OS?
Why might we NOT utilize multiple threads?

Complexity:
• Maintaining data safety.
• Guaranteeing “liveness” of threads.
• Being able to deal with the composition of methods.

Note: the root of these problem is essentially the shared state…

Overhead:
• Higher resource usage to support multiple threads.
• May limit performance compared to direct event processing
  – time spent on context switching, locking of things, etc.

Programming in languages that support threads

Threads are available in many languages
• C, C++, C#, OCaml, Java, SmallTalk, Python, …
• Some support it in an abstract manner as part of the language
  specification (like Java) while others give you access to system-
  level APIs as a platform-specific add-on (like C).
• Different languages may have different models.
Threads in Java
As mentioned before, every application has at least one thread.
• The “main” thread in this case is the essentially one started by the JVM to run the application’s `main()` method.
  – We won’t go into thinking about JUnit testing’s main thread here…

The code running in the main thread can then choose to create and activate other threads.
• This can be done explicitly, by using Java’s `Thread` class.
• It can also be done implicitly, by invoking methods from libraries which themselves create and activate new threads as part of what they do.
  – AWT/Swing for graphics
  – Applets
  – Remote Method Invocation (RMI)
    • Initially Java-only, CORBA changed that.
  – etc.

Threads in Java: Explicit creation
To explicitly create and activate a new thread
• Instantiate a `Thread` object
  – Create a new object of class `Thread` or of a subclass of `Thread`.
• Invoke that object’s `start()` method
  – Doing this will have the system begin executing the `Thread` object’s `run()` method *concurrently* with the current thread.

This new `Thread` object terminates execution when its `run()` method returns, but note that the `Thread` object itself still exists.
Notice that in this image there isn’t anything that tells the main thread (execution seen in green) to ever wait for the new thread (execution seen in purple) to finish. In Java, by default, the main thread will stick around when it’s done if spawned threads are still running.

**Example: Setting time-delayed alarms**

**Goal:** set alarms that will be triggered in the future

**Input:** integer time $t$ (seconds) and string message $m$.

**Result:** We’ll see $m$ displayed after $t$ seconds.
**Example: Setting alarms - synchronous**

```java
int secondsDelay;
String msg;

while (true) {
    System.out.print("Alarm length in seconds: ");
    secondsDelay = s.nextInt();  s.nextLine();
    System.out.print("Message: ");
    msg = s.nextLine();

    // wait (in msecs)
    try {
        Thread.sleep(secondsDelay * 1000);
    } catch (InterruptedException e) { }

    // sound the alarm
    System.out.println("\[" + secondsDelay + "] " + msg);
}
```

**Example: Setting alarms - threaded (Part 1)**

```java
final class AlarmThread extends Thread {
    int secondsDelay;
    String msg;

    public AlarmThread(String msgIn, int delayIn) {
        msg = msgIn;
        secondsDelay = delayIn;
    }

    public void run() {
        // wait (in msecs)
        try {
            Thread.sleep(secondsDelay * 1000);
        } catch (InterruptedException e) { }

        // sound the alarm
        System.out.println("\[" + secondsDelay + "] " + msg);
    }
}
```
**Example: Setting alarms - threaded (Part 2)**

```java
int secondsDelay;
String msg;

while (true) {
    System.out.print("Alarm length in seconds: ");
    secondsDelay = s.nextInt();  s.nextLine();
    System.out.print("Message: ");
    msg = s.nextLine();

    // start alarm thread
    Thread t = new AlarmThread(msg, secondsDelay);
    t.start();
}
```

**Example: Setting alarms - using Runnable (Part 1)**

Extending the `Thread` class in Java removes the ability to extend a different parent class.

**Instead you can implement the Runnable interface.**

Doing this guarantees that the new class has a `void run()` method.

**Now, you can construct a Thread object on the fly from the class that implements Runnable.**

Constructor Option #1 `Thread(Runnable target)`
Constructor Option #2 `Thread(Runnable target, String name)`
Example: Setting alarms - using Runnable (Part 2)

```java
final class AlarmRunnable implements Runnable {
    int secondsDelay;
    String msg;

    public AlarmRunnable(String msgIn, int delayIn) {
        msg = msgIn;
        secondsDelay = delayIn;
    }

    public void run() {
        // wait (in msecs)
        try {
            Thread.sleep(secondsDelay * 1000);
        } catch (InterruptedException e) { }

        // sound the alarm
        System.out.println("[" + secondsDelay + "] " + msg);
    }
}
```

Example: Setting alarms - using Runnable (Part 3)

```java
int secondsDelay;
String msg;

while (true) {
    System.out.print("Alarm length in seconds: ");
    secondsDelay = s.nextInt();  s.nextLine();
    System.out.print("Message: ");
    msg = s.nextLine();

    // start alarm thread
    Thread t = new Thread(new AlarmRunnable(msg, secondsDelay));
    t.start();
}
```
Parameters to threads

Notice that the run() method doesn’t take parameters.

We did, however, essentially “pass parameters” to the a thread via (in our example) the constructor (though you could use a set method) and then storing them as private fields within the Thread-extended or Runnable object itself.

Thread Scheduling: General Idea

Once a new thread is created and started, how does it execute relative to other threads?

– Of the same process?
– Of other processes?
– On which CPU if there are several available?

This is a question of scheduling: Given N processors and M threads, which thread(s) should be assigned to run on which CPU at any given time?

Answering this is sometimes the job of the operating system and sometimes the job of the programmer.
**Thread Scheduling: Multi-threaded processes**

If there are more processors than threads…
- Could have each thread run on its own processor.
- This would split a process across CPUs.
  - Exploits hardware-level concurrency.
  - Parallelism potentially creates issues since multiple threads from a single process could be accessing shared data at the same exact moment.

If there are more threads than processors…
- Threads would need to share access to the CPU.
  - Each would be given a slice of time to run, and then paused (pre-empted).
  - Interleaving potentially creates issues since a thread might be pre-empted in the middle of what would appear to be an atomic operation.

---

**Scheduling Example (1)**

**One process and all its threads on a single CPU**

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU 2</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`p2 threads: ![Color1]  p1 threads: ![Color2]`
**Scheduling Example (2)**

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>p1</th>
<th>p1</th>
<th>p1</th>
<th>p2</th>
<th>p2</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 2</td>
<td>p1</td>
<td>p1</td>
<td>p1</td>
<td>p2</td>
<td>p2</td>
<td>p2</td>
</tr>
</tbody>
</table>

*Threads of a process allowed to run on either CPU*

**Thread Scheduling**

If multiple threads share a CPU, must somehow decide:
- When the current thread should stop running.
- What thread to run next.

A thread *can* voluntarily offer to yield the CPU:
- However, a call to yield() may be ignored so one should not depend on it actually leading to the thread pausing.

*Preemptive schedulers* can de-schedule the current thread at any time!

Threads can be de-scheduled by the scheduler when they block (such as when they are waiting for a locked resource or are waiting for I/O to complete) or go to sleep.
**Lifecycle of a Thread**

While a thread executes, it can pass through a number of different phases:

- **New**: created but not yet started
- **Run-able**: is running or is ready to run if a CPU becomes free
- **Blocked**: waiting for I/O or for a lock
- **Sleeping**: paused for a user-specified interval
- **Terminated**: completed

**Deciding which Thread will get to run next**

The scheduler looks at all of the run-able threads, including threads that were recently unblocked because:

- A lock was released
- I/O became available
- They finished sleeping, etc.

Of these threads, it will consider the thread’s priority as part of its decision.

- This can be set with the `setPriority()` method.
- Higher priority threads can get preference.
- Threads that are waiting for I/O might also be preferred.
**Thread’s basic instance methods**

void **start()**
- Invoke the object’s run method in a concurrent thread.

**boolean isAlive()**
- Check to see whether the object’s thread is still running.

**void setPriority(int newPriority)**
- Note: the scheduler may or may not actually respect priorities…

**void join(int) throws InterruptedException**
- Has the current thread wait for the object’s thread to finish (successfully or unsuccessfully) before continuing itself.
- If the integer is provided (and is not zero) then it will only wait that many milliseconds before throwing the exception.

---

**Example: Setting alarms - threaded (Part 2) w/sync**

```java
int secondsDelay;
String msg;

while (true) {
    System.out.print("Alarm length in seconds: ");
    secondsDelay = s.nextInt();  s.nextLine();
    System.out.print("Message: ");
    msg = s.nextLine();

    // start alarm thread
    Thread t = new AlarmThread(msg, secondsDelay);
    t.start();
    // but wait for that thread to complete before going on
    try {
        t.join();
    } catch (InterruptedException e) { }
}
```
Thread’s basic static methods

void yield()
• Hint that the current thread should give up the CPU.

void sleep(long milliseconds) throws InterruptedException
• The current thread will sleep for the given period of time.

Thread currentThread()
• Returns a reference to the Thread object for the currently executing thread.

Note that these all apply to current thread that is invoking the method.

Mark a Thread as a daemon – instance method

void setDaemon(boolean on)
• Sets whether the thread should be treated as a daemon thread.
• Note: This must be set before thread is started.

By default a new thread acquires the status of thread that is spawning it.

Program execution terminates when no threads are left running except for daemon threads.

Consider garbage collection or if you have a pool of worker threads waiting for data and all the data has been processed and the program is ready to end.
Zero’th Project
Download the code base.

Read and understand how the provided code works and what is required to create a fully working project.

Write several test cases to better observe its performance.
• How does this code perform given different numbers of threads?
• What might account for any performance differences?
• Are there any problems with the general algorithm presented for max?