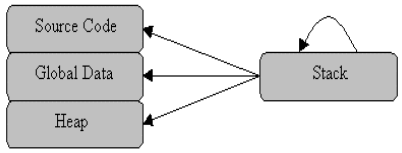
Multi-threading

- Introduction to Concurrency
- What is a **Thread**?
- Benefits of Threads
- Programming with Threads
- Pitfalls of Threads
- Synchronization

READ CHAPTER 21

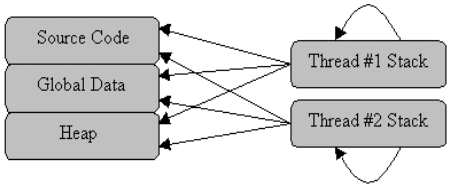
Introduction to Concurrency

- A program with a single flow of control is called a **sequential** program
- A program has four parts: *source code*, *global data*, *heap*, and *stack*



Introduction to Concurrency

- A program with multiple points of execution is called a **concurrent** program



Tasks and Processes

- A task is the execution of a sequential program (or a sequential program within a concurrent program)
- A process is used in Operating Systems (OS) as a unit of resource allocation for CPU and memory
- A traditional OS process has a single thread of control (i.e. no internal concurrency)
- Modern OS allow a process known as a heavyweight process (i.e. with multiple threads of control – concurrency within the process)

Heavyweight v. Lightweight Processes

- Each thread of control within a heavyweight process is known as a lightweight process
- BECAUSE it shares the same memory
- Multiple threads of a heavyweight process can access shared data in the process's memory
- Access must be synchronized
- "heavy" and "light" refers to the context-switching overhead (CPU and memory allocation vs. CPU allocation)

What is a Thread?

- The term *thread* derives from the phrase ***thread of execution*** in operating systems
- It is a *lightweight process*
- Threads can create other threads and kill them
- Newly created threads will run in the same address space allowing them to share data
- They have been around for quite some time
- They are built-in into Java
- Java made the use of them easy and productive

Benefits of Threads

- The ability to perform multiple tasks simultaneously
- Allow us to take advantage of computers with multiple CPUs
- Other benefits
  - Increase application throughput
  - Responsiveness
  - The ability to use system's resource efficiently

Programming with Threads

- Creating and Starting Threads
- Putting a Thread to Sleep
- Controlling Threads
- Thread Priorities
- Pitfalls of Threads
- Synchronization
- Producer/Consumer
- Scheduling

Creating and Starting Threads

- There are two ways to create a thread in Java
  - Extending the `Thread` class

```
class MyThread extends Thread {
    ....
    public void run() {
      ....
    }
    public static void main(String argv[]) {
      MyThread t1 = new MyThread();
      t1.start();
    }
}
```


Creating and Starting Threads

- The other way of creating a thread in Java is
  - By implementing the `Runnable` interface

```
class MyThread implements Runnable {
  public void run() {
    ....
  }
  public static void main(String argv[]) {
    MyThread s = new MyThread();
    Thread t1 = new Thread(s);
     t1.start();
  }
}
```


Creating and Starting Threads

- Examples:
  - MyThread.java
  - MyThread2.java
  - Counter.java

Putting a Thread to Sleep

- You may pause a thread for a specific period of time by putting it to sleep using `sleep()`

```
try {
  Thread.sleep(4000); // 4 seconds
} catch (InterruptedException ie) {
  ie.printStackTrace();
}
```

- The argument to sleep specifies the number of milliseconds the thread will sleep for

Controlling Threads

- Do not use: stop(), suspend(), resume()
- These methods have been deprecated in Java 2 and they should not be used
- Basicaly they are not thread -safe….more on this in class

Thread Priorities

- Threads will normaly be competing for processor time
- Time-critical tasks with hard deadlines can be given a higher priority than less critical tasks
- The Thread class defines three constants:
  - MAX_PRIORITY (10)
  - MIN_PRIORITY (1)
  - NORM_PRIORITY (the default 5)
- Use getPriority() and setPriority()

Pitfalls of Threads

- One pitfall of threads is data sharing
  - Examples: Alice and Bob are sharing a checkbook

```
int balance;
boolean withdraw(int amount);
  if (balance - amount >= 0) {
    balance = balance - amount;
    return true;
  }
  return false;
}
```


Pitfalls of Threads

- If Alice and Bob are executing the code segment simultaneously, we get:

Alice	Bob	Balance
If (80 - 50 >= 0)		80
	If (80 - 70 >= 0)	80
Balance = Balance - 50		30
	Balance = Balance - 70	-40

Synchronization

- Mutual Exclusion (preventing simultaneous access to shared data) can be accomplished using the `synchronized` access specifier

```
synchronized boolean withdraw(int amount) {
  ….
}
```

Or using the synchronized block:

```
boolean withdraw(int amount) {
   synchronized(this) {
     ….
   }
}
```


Producer/Consumer Problem

- The producer task produces information, which is then consumed by the consumer task
  - Transfer a data from a producer task to a consumer task
  - Synchronize between producer/consumer. If no data is available the consumer has to wait for the data to arrive from the producer
- The producer and consumer may reside on the same node or they can be distributed

wait() and notify()

- Consumer:

```
private List objects = new ArrayList();

private Object remove() throws InterruptedException
  {
 synchronized(objects) {
    objects.wait();
 }
 Object obj = objects.get(0);
 objects.remove(0);
}
```


wait() and notify()

- Producer

```
public void insert(Object obj) {
  synchronized(objects) {
    objects.add(obj);
    objects.notify(); // OR objects.notifyAll();
  }
}
```

Scheduling

- How does the Java runtime schedules CPU time among threads?
  - If a thread is blocked (I/O) or sleep, it uses no CPU time
  - The runtime chooses the thread with the highest priority
  - If all threads have the same priority (then order is not defined)
    - Preempting threads (share processor)
    - Allow one thread to run till it gives up the processor. All other threads will be starved of CPU time
    - Because of this, you should not perform long compute-bound tasks without calling Thread.yield() periodically
- Note: the use of priorities and yield() result in non-portable code. They simply offer hints to the scheduler…

Other issues

- Deadlock: two threads competing for a resource and they end up with each waiting for the other to finish
- How to avoid deadlock?
- Atomicity: can an action be interrupted by another thread
- Memory: memory allocated for a thread is not freed when the thread finishes