CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Threads in Java

Department of Computer Science
University of Maryland, College Park

Problem

- Multiple tasks for computer
  - Draw & display images on screen
  - Check keyboard & mouse input
  - Send & receive data on network
  - Read & write files to disk
  - Perform useful computation (editor, browser, game)
- How does computer do everything at once?
  - Multitasking
  - Multiprocessing
Multitasking (Time-Sharing)

- Approach
  - Computer does some work on a task
  - Computer then quickly switch to next task
  - Tasks managed by operating system (scheduler)
- Computer seems to work on tasks concurrently
- Can improve performance by reducing waiting

Multitasking Can Aid Performance

- Single task

```
Busy ← 1 sec → Busy ← 1 sec → Busy ← 1 sec → Busy
Total Execution Time = 7 seconds
Total Time Executing Code: 4 seconds
Total Time Waiting: 3 seconds
Time Executing Code: 57%  Time Waiting: 43%
```

- Two tasks

```
P1: Busy ← 1 sec → Busy ← 1 sec → Busy ← 1 sec → Busy
Total Time Executing Code: 8 seconds
Total Time Waiting: 0 seconds
Time Executing Code: 100%  Time Waiting: 0%

P2: Busy ← 1 sec → Busy ← 1 sec → Busy ← 1 sec → Busy
```
Multiprocessing (Multithreading)

• Approach
  • Multiple processing units (multiprocessor)
  • Computer works on several tasks in parallel
  • Performance can be improved

Dual-core AMD Athlon X2
32 processor Pentium Xeon
Titan at ORNL

Perform Multiple Tasks Using Processes

• Process
  • Definition → executable program loaded in memory
  • Has own address space
    • Variables & data structures (in memory)
  • Each process may execute a different program
  • Communicate via operating system, files, network
  • May contain multiple threads
Perform Multiple Tasks Using Threads

- Thread
  - Definition → sequentially executed stream of instructions
  - Has own execution context
    - Program counter, call stack (local variables)
  - Communicate via shared access to data
  - Also known as "lightweight process"

Motivation for Multithreading

- Captures logical structure of problem
  - May have concurrent interacting components
  - Can handle each component using separate thread
  - Simplifies programming for problem

- Example

Web Server uses threads to handle ...

Multiple simultaneous web browser requests
Motivation for Multithreading

- Better utilize hardware resources
  - When a thread is delayed, compute other threads
  - Given extra hardware, compute threads in parallel
  - Reduce overall execution time

- Example

![Multiple simultaneous web browser requests...](image1)

![Handled faster by multiple web servers](image2)

Programming with Threads

- Concurrent programming
  - Writing programs divided into independent tasks
  - Tasks may be executed in parallel on multiprocessors

- Multithreading
  - Executing program with multiple threads in parallel
  - Special form of multiprocessing
Creating Threads in Java

- **Two approaches to create threads**
  - Extending Thread class *(NOT RECOMMENDED)*
  - Runnable interface approach *(PREFERED)*
- **Approach 1: Extending Thread class**
  - We overload the Thread class run() method
  - The run() methods defines the actual task the thread performs
  - **Example**
    ```java
    public class MyT extends Thread {
        public void run() {
            ...
            // work for thread
        }
    }
    MyT t = new MyT();  // create thread
    t.start();  // begin running thread
    ...
    // thread executing in parallel
    **Example**: message, messageThreadExtends packages

- **Approach 2: Runnable Interface**
  - Define a class (worker) that implements the Runnable interface
    ```java
    public interface Runnable {
        public void run();  // work done by thread
    }
    ```
  - Create thread to execute the run() method
    - Alternative 1: Create thread object and pass worker object to Thread constructor
    - Alternative 2: Hand worker object to an executor
  - **Example**
    ```java
    public class Worker implements Runnable {
        public void run() {
            // work for thread
        }
    }
    Thread t = new Thread(new Worker());  // create thread
    t.start();  // begin running thread
    ...
    // thread executing in parallel
    **Example**: message, messageThreadRunnable packages
Why Extending Thread Approach Not Recommended?

- Not a big problem for getting started
  - But a bad habit for industrial strength development
- Methods of worker and Thread class intermixed
- Hard to migrate to more efficient approaches
  - Thread Pools

Thread Class

```java
public class Thread extends Object implements Runnable {
    public Thread();
    public Thread(String name);  // Thread name
    public Thread(Runnable R);
    public Thread(Runnable R, String name);

    public void run();  // if no R, work for thread
    public void start();  // thread gets in line so it eventually it can run
    ...
}
```
More Thread Class Methods

```
public class Thread extends Object {
    ...
    public static Thread currentThread()
    public String getName()
    public void interrupt()  // alternative to stop (deprecated)
    public boolean isAlive()
    public void join()
    public void setDaemon()
    public void setName()
    public void setPriority()
    public static void sleep()
    public static void yield()
}
```

Creating Threads in Java

- **Note**
  - Thread eventually starts executing **only if start() is called**

- Runnable is interface
  - So it can be implemented by any class
  - Required for multithreading in applets

- **Do not call the run method directly**
Threads – Thread States

- Java thread can be in one of these states
  - **New** → thread allocated & waiting for start()
  - **Runnable** → thread can begin execution
  - **Running** → thread currently executing
  - **Blocked** → thread waiting for event (I/O, etc.)
  - **Dead** → thread finished
- Transitions between states caused by
  - Invoking methods in class Thread
    - new(), start(), yield(), sleep(), wait(), notify()…
  - Other (external) events
    - Scheduler, I/O, returning from run()…
- In Java states defined by Thread.State
  [http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html)

---

Threads – Thread States

- State diagram

**Running** is a logical state → indicates runnable thread is actually running
Daemon Threads

- Java threads types
  - User
  - Daemon
    - Provide general services
    - Typically never terminate
    - Call `setDaemon()` before `start()`
- Program termination
  - All user threads finish
  - Daemon threads are terminated by JVM

Threads – Scheduling

- Scheduler
  - Determines which runnable threads to run
    - When context switching takes place
  - Can be based on thread priority
  - Part of OS or Java Virtual Machine (JVM)

- Scheduling policy
  - Non-preemptive (cooperative) scheduling
  - Preemptive scheduling
Threads – Non-preemptive Scheduling

- Threads continue execution until
  - Thread terminates
  - Executes instruction causing wait (e.g., IO)
  - Thread volunteering to stop (invoking yield or sleep)

Threads – Preemptive Scheduling

- Threads continue execution until
  - Same reasons as non-preemptive scheduling
  - Preempted by scheduler
Thread Scheduling Observations

- Order thread is selected is **indeterminate**
  - Depends on scheduler
- Scheduling may not be fair
  - Some threads may execute more often
- Thread can block indefinitely (starvation)
  - If other threads always execute first
- Your code should work correctly regardless the scheduling policy in place

Java Thread Example

```java
public class ThreadNoJoin extends Thread {
    public void run() {
        for (int i = 0; i < 3; i++) {
            try {
                sleep((int)(Math.random() * 5000)); // 5 secs
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(i);
        }
    }
    public static void main(String[] args) {
        Thread t1 = new ThreadNoJoin();
        Thread t2 = new ThreadNoJoin();
        t1.start();
        t2.start();
        System.out.println("Done");
    }
}
```

To understand this example better, let's assume we want to make a sandwich
Java Thread Example – Output

- Possible outputs
  - 0,1,2,0,1,2,Done // thread 1, thread 2, main()
  - 0,1,2,Done,0,1,2 // thread 1, main(), thread 2
  - Done,0,1,2,0,1,2 // main(), thread 1, thread 2
  - 0,0,1,1,2,Done,2 // main() & threads interleaved

Thread Class – join( ) Method

- Can wait for thread to terminate with join( )
- Method prototype
  - public final void join( )
    - Returns when thread is done
    - Throws InterruptedException if interrupted
Java Thread Example (Join)

```java
public class ThreadJoin extends Thread {
    public void run() {
        for (int i = 0; i < 3; i++) {
            try {
                sleep((int)(Math.random() * 5000)); // 5 secs
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        Thread t1 = new ThreadJoin();
        Thread t2 = new ThreadJoin();
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Done");
    }
}
```

About Join

- Important: You will limit the concurrency level if you do not start/join correctly
- Suppose you want to run many threads concurrently. **Start them all and then execute the join for each one. Do not start one thread, then join on that thread, start the second one, join on that thread, etc.**
- The following is **WRONG!**
  ```java
t1.start()
t1.join()
t2.start()
t2.join()
```
- Feel free to use arrays, sets, etc., to keep track of your threads
Terminating Threads

- A thread ends when the run() method ends
- Sometimes we may need to stop a thread before it ends
  - For example, you may have created several threads to find a problem solution and once one thread finds it, there is no need for the rest
- How to stop thread?
  - Using **stop()** method → **WRONG!** This is a deprecated method. Using it can lead to problems when data is shared
  - Using **interrupt()** method
    - This method does not stop the thread. Instead, it notifies the thread that it should terminate. The method sets a boolean variable in the thread and that value can be checked by the thread (by using the method interrupted())
    - It is up to the thread to terminate or not
    - public void run() {
        while(!Thread.interrupted()) {
            // work
        }
        // release resource, cleaning tasks
    }

Thread Example

- Swing uses a single-threaded model
- Long computations in the EDT freezes the GUI
- Example: Progress Bar Example