Multithreading in Java

Fawzi Emad
Chau-Wen Tseng

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

Two tasks
Multiprocessing (Multithreading)

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

---

Perform Multiple Tasks Using...

1. **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...

2. Thread
   - Definition – sequentially executed stream of instructions
   - Shares address space with other threads
   - Has own execution context
     - Program counter, call stack (local variables)
   - Communicate via shared access to data
   - Multiple threads in process execute same program
   - Also known as “lightweight process”

Motivation for Multithreading

1. 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

2. 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… Handled faster by multiple web servers

Multithreading Overview

- Motivation & background
- Threads
  - Creating Java threads
  - Thread states
  - Scheduling
- Synchronization
  - Data races
  - Locks
  - Wait / Notify
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
  - Thread class
    ```java
    public class Thread extends Object { … }
    ```
  - Runnable interface
    ```java
    public interface Runnable {
        public void run(); // work ⇒ thread
    }
    ```
Thread Class

```java
public class Thread extends Object
    implements Runnable {
    public Thread();
    public Thread(String name); // Thread name
    public Thread(Runnable R); // Thread -> R.run()
    public Thread(Runnable R, String name);
    
    public void run(); // if no R, work for thread
    public void start(); // begin thread execution
    
    }
```

More Thread Class Methods

```java
public class Thread extends Object {
    ... 
    public static Thread currentThread()
    public String getName()
    public void interrupt()
    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

1. Thread class
   - Extend Thread class and override the run method
   
   **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
   ```

2. Runnable interface
   - Create object implementing Runnable interface
   - Pass it to Thread object via Thread constructor
   
   **Example**
   ```java
   public class MyT implements Runnable {
       public void run() {
           ... // work for thread
       }
   }
   
   Thread T = new Thread(new MyT()); // create thread
   T.start(); // begin running thread
   ...
   // thread executing in parallel
   ```
Creating Threads in Java

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

Runnable is interface
- So it can be multiply inherited
- Required for multithreading in applets

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()...
Threads – Thread States

State diagram

- **new** → **Runnable**
- **Runnable** → **Running** by scheduler
- **Running** → **Blocked** by IO, sleep, wait, join
- **Blocked** → **Runnable** by yield, time slice
- **Runnable** → **Dead** by terminate
- **Dead** → **new**
- **Runnable** → **notify, notifyAll, IO complete, sleep expired, join complete**
- **Running** → **terminate**

Daemon Threads

- **Java threads types**
  - **User**
  - **Daemon**
    - Provide general services
    - Typically never terminate
    - Call setDaemon() before start()
  
- **Program termination**
  1. All user threads finish
  2. Daemon threads are terminated by JVM
  3. Main program finishes
Threads – Scheduling

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

Scheduling policy
- Nonpreemptive (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

Java Thread Example

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

    public static void main(String[] args) {
        new ThreadExample().start();
        new ThreadExample().start();
        System.out.println("Done");
    }
}
```
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

main(): thread 1, thread 2, println Done
thread 1: println 0, println 1, println 2
thread 2: println 0, println 1, println 2

Data Races

```java
public class DataRace extends Thread {
    static int x;
    public void run() {
        for (int i = 0; i < 100000; i++) {
            x = x + 1;
            x = x - 1;
        }
    }
    public static void main(String[] args) {
        x = 0;
        for (int i = 0; i < 100000; i++)
            new DataRace().start();
        System.out.println(x); // x not always 0!
    }
}
```
Thread Scheduling Observations

- Order thread is selected is \textit{indeterminate}
  - Depends on scheduler

- Thread can block indefinitely (starvation)
  - If other threads always execute first

- Thread scheduling may cause \textit{data races}
  - Modifying same data from multiple threads
  - Result depends on thread execution order

- Synchronization
  - Control thread execution order
  - Eliminate data races