CSE2101: Object Oriented Programming-II (Java)

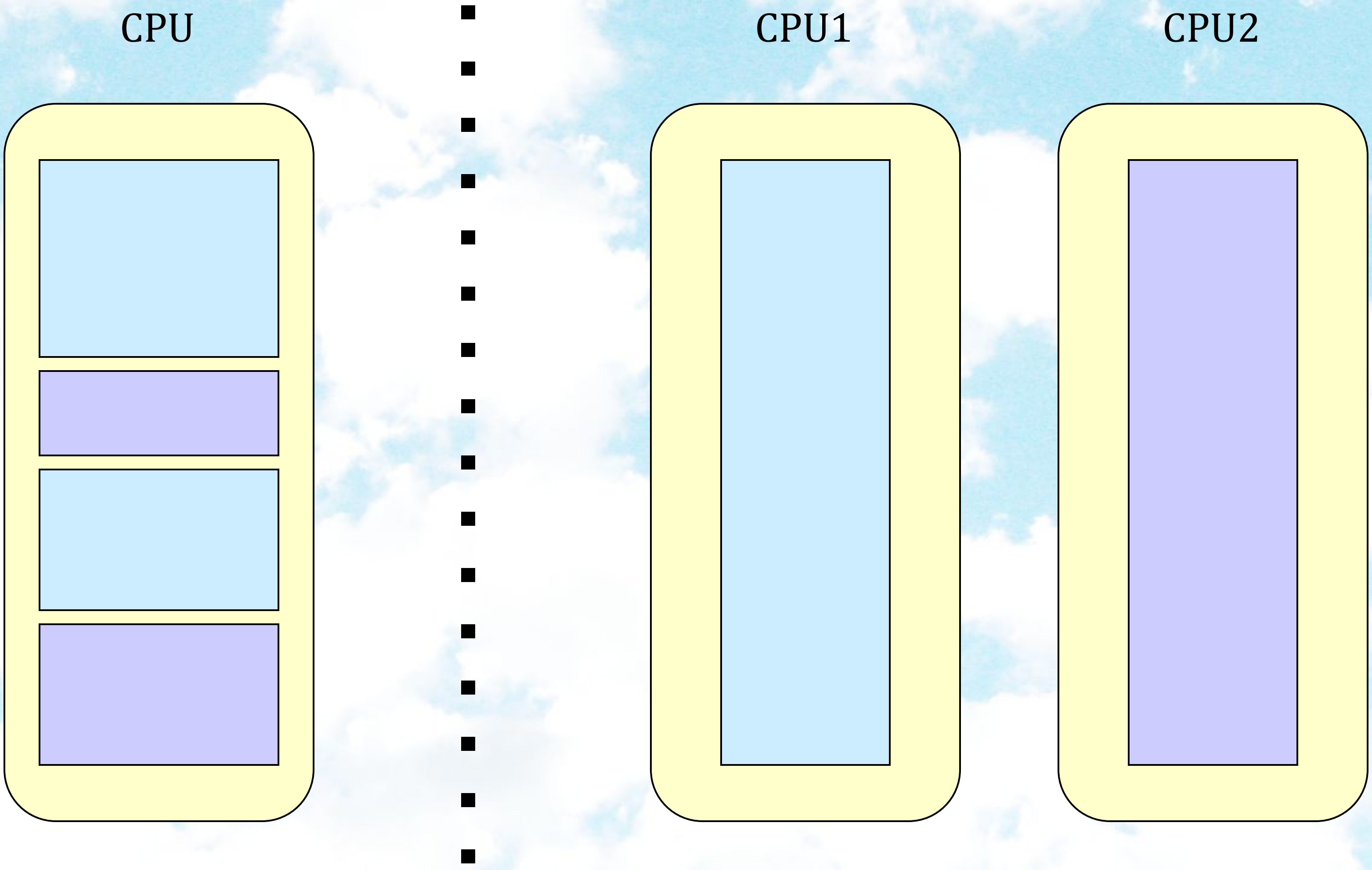
Lecture 13

Threads

Multitasking and Multithreading

- Multitasking refers to a computer's ability to perform multiple jobs concurrently
  - more than one program are running concurrently, e.g., UNIX
- A thread is a single sequence of execution within a program
- Multithreading refers to multiple threads of control within a single program
  - each program can run multiple threads of control within it, e.g., Web Browser

Concurrency vs. Parallelism



Threads Overview

- Threads allow the program to run tasks in parallel
- In many cases threads need to be synchronized, that is, be kept not to handle the same data in memory concurrently
- There are cases in which a thread needs to wait for another thread before proceeding

What are Threads Good For?

- To maintain responsiveness of an application during a long running task.
- To enable cancellation of separable tasks.
- Some problems are intrinsically parallel.
- To monitor status of some resource (DB).
- Some APIs and systems demand it: Swing.

Application Thread

- When we execute an application:
  - The JVM creates a Thread object whose task is defined by the **main()** method
  - It starts the thread
  - The thread executes the statements of the program one by one until the method returns and the thread dies

Multiple Threads in an Application

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory (heap)
- Two different threads can act on the same object and same static fields concurrently

Creating Threads

- There are two ways to create our own **Thread** object

1. Subclassing the **Thread** class and instantiating a new object of that class
2. Implementing the **Runnable** interface

- In both cases the **run()** method should be implemented

Example (Subclassing the Thread)

```
public class CounterThread extends Thread {
  public void run()  {
    for ( int i=0; i<10; i++)
      System.out.println(“Count:  “ + i);
  }

  public static void main(String args[]) {
    CounterThread ct = new CounterThread();
    ct.start();
  }
}
```

Example (Implementing the Runnable)

```
public class DownCounter implements
Runnable {
  public void run() {
    for (int i=10; i>0; i--)
      System.out.println(“Down:  “+ i);
  }

  public static void main(String args[]) {
    DownCounter ct = new DownCounter();
    Thread t = new Thread(ct);

    t.start();
  }
}
```

Thread Name

- t.getName();
  - – Obtain a thread’s name
- t.setName();
  - – Change the name of the thread

Thread Methods

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

Thread Methods

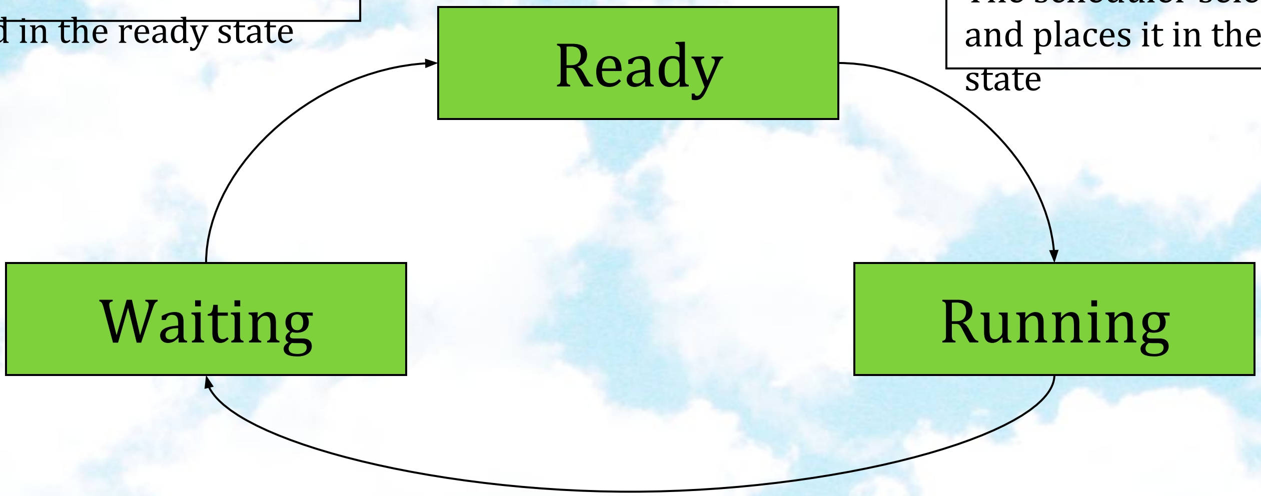
- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute
  - Allow only threads of the same priority to run
- **sleep(int *m*)/sleep(int *m*,int *n*)**
  - The thread sleeps for *m* milliseconds, plus *n* nanoseconds

Thread Scheduling

- Threads are scheduled like processes
- Thread states
  - Running
  - Waiting, Sleeping, Suspended, Blocked
  - Ready
  - Dead
- When you invoke start() the Thread is marked ready and placed in the thread queue

Thread States

The start() method places a
thread in the ready state



The scheduler selects a thread and places it in the running state

A thread that is waiting for I/O, was suspended, is sleeping, blocked, or otherwise is unable to do any more work is placed in
the waiting state

Scheduling Implementations

- Scheduling is typically either:
  - non-preemptive
  - preemptive
- Most Java implementations use preemptive scheduling.
  - the type of scheduler will depend on the JVM that you use.
  - In a non-preemptive scheduler a thread leaves the running state only when it is ready to do so.

Thread Priorities

- Threads can have priorities from 1 to 10 (10 is the highest)
- The default priority is 5
  - The constants Thread.MAX_PRIORITY, Thread.MIN_PRIORITY, and Thread.NORM_PRORITY give the actual values
- Priorities can be changed via setPriority() (there is also a getPriority())

isAlive()

- The method isAlive() determines if a thread is considered to be alive
  - A thread is alive if it has been started and has not yet died.
- This method can be used to determine if a thread has actually been started and has not yet terminated

isAlive()

```
public class WorkerThread extends Thread {
  static private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() );
    System.out.println( result );
  }
}
```

sleep()

- Puts the currently executing thread to sleep for the specified number of milliseconds
  - sleep(int milliseconds)
  - sleep(int millisecs, int nanosecs)
- Sleep can throw an InterruptedException
- The method is static and can be accessed through the Thread class name

sleep()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() )
      try {
        sleep( 100 );
      } catch ( InterruptedException ex ) {}

    System.out.println( result );
  }}
```

Joining Threads

- Calling isAlive() to determine when a thread has terminated is probably not the best way to accomplish this

- What would be better is to have a method that once invoked would wait until a specified thread has terminated

- join() does exactly that
  - join()
  - join(long timeout)
  - join(long timeout, int nanos)

- Like sleep(), join() is static and can throw an InterruptedException

join()

```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated time consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    try {
        t.join();
    } catch ( InterruptedException ex ) {}

    System.out.println( result );
  }
}
```

Thank you