

# Multithread Computing

## About This Lecture

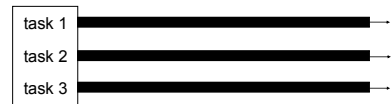
- ◆ Purpose
  - To learn multithread programming in Java
- ◆ What You Will Learn
  - Benefits of multithreading
  - Class Thread and interface Runnable
  - Thread methods and thread state diagram
  - Synchronization problem and its solution
  - Thread groups

## Thread vs. Process

- ◆ A *thread* is a function or method in execution. It is represented by code, program counter value, register values, and its own method activation stack
- ◆ A *process* is allocated system resources like disk/memory space to perform a task. It generates threads to execute code
- ◆ Threads in a process can share data. Synchronization and data protection are programmers' responsibility. The benefit is better performance due to finer user control.

## Benefits of Multithreading (1/2)

- ◆ Multithreading on multiprocessor system leads to parallel computing
- ◆ Multithreading on a single processor may also benefit from overlapping of running time on different function units



## Benefits of Multithreading (2/2)

- ◆ On a single processor, multithreading supports the illusion that several threads are executing in parallel. The programmer can thus focus on the subtask of each thread, and let OS take care of detail how the threads share the CPU
- ◆ For interactive systems, multithreading is almost the only way to make a system remain responsive when heavy computing is going on

## Threads are System-Dependent (1/2)

- ◆ The scheduling of threads is system dependent
- ◆ On any system, a thread of higher priority can preempt the CPU from a running thread of lower priority
- ◆ On Windows, round-robin time-sharing is used. Each thread of the highest priority is allocated a small time slice to run. When its time slice is up, it gives up the CPU to another thread of the same priority, and waits for its next turn.

## Threads are System-Dependent (2/2)

- ◆ On Sun Solaris, the running thread will not give up CPU to threads of the same priority. As a result, programmers have to use `yield()` to let some thread voluntarily give up CPU from time to time to share CPU
- ◆ Java code should try to accommodate both thread models
- ◆ Our discussion will be based on round-robin time-sharing, which represents the future

## Class *java.lang.Thread*

- ◆ For a simple thread that doesn't need to inherit from an existing class, extend the thread class from `java.lang.Thread`, and implement thread computation in `Thread` method "`void run()`"

```
//User Defined Thread Class
class UserThread extends Thread {
    .....
    public UserThread() { .....}
    public void run() { ..... }
}
```

```
//Client class
public class Client {
    .....
    main() {
        UserThread ut = new UserThread();
        ut.start();
        .....
    }
}
```

## Interface *java.lang.Runnable*

- ◆ To make a derived class a thread, implement interface `Runnable`

```
//User Defined Thread Class
class UserThread extends X
implements Runnable {
    .....
    public UserThread() { .....}
    public void run() { ..... }
}
```

```
//Client class
public class Client {
    .....
    main() {
        Thread ut =
            new Thread(new UserThread());
        ut.start();
        .....
    }
}
```

## TestThreads.java (1/3)

```
public class TestThreads {
    public static void main(String[] args) {
        // Create threads
        PrintChar printA = new PrintChar('a', 100);
        PrintChar printB = new PrintChar('b', 100);
        PrintNum print100 = new PrintNum(100);
        // Start threads
        print100.start();
        printA.start();
        printB.start();
    }
}
```

## TestThreads.java (2/3)

```
class PrintChar extends Thread {
    private char charToPrint; // The character to print
    private int times; // The times to repeat
```

```
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }
```

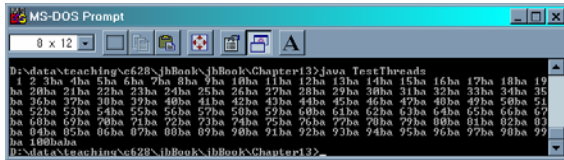
```
    public void run() {
        for (int i=0; i < times; i++)
            System.out.print(charToPrint);
    }
}
```

## TestThreads.java (3/3)

```
class PrintNum extends Thread {
    private int lastNum;
```

```
    // Construct a thread for print 1, 2, ... i
    public PrintNum(int n) {
        lastNum = n;
    }
    public void run() {
        for (int i=1; i <= lastNum; i++)
            System.out.print(" " + i);
    }
}
```

## Run TestThreads



## Useful Methods of a Thread (1/4)

- ◆ **public void run()**
  - Execution of the thread starts from Java runtime system invoking run() in a new thread, as a result of user code calling method start(). Never invoked directly from user code
- ◆ **public void start()**
  - User code invokes it to start a new thread, thereby causing the run() be invoked by Java runtime system in the new thread

## Useful Methods of a Thread (2/4)

- ◆ **public static void sleep(long millis)** throws InterruptedException
  - Puts the calling thread to sleep for a specified time in milliseconds. Takes minimum resources. The thread will be awakened by a timer at the end of the sleep
- ◆ **public boolean isAlive()**
  - Tests whether the thread is currently running or blocked
- ◆ **public void setPriority(int p)**
  - Sets priority (1 to 10) for the thread
- ◆ **public static void yield()**
  - Temporarily pause the thread to let other threads of the same priority run; only useful for non-time-sharing platforms

## Useful Methods of a Thread (3/4)

- ◆ **public String getName()**
  - Return the name of the thread
- ◆ **public static Thread currentThread()**
  - Return the reference of the current running thread object
- ◆ **public join()**
  - It will block the caller thread until the target thread dies
- ◆ **public void setDaemon(true)**
  - This thread will continue to run after the program terminates

## Useful Methods of a Thread (4/4)

- ◆ **public void interrupt()**
  - Generates InterruptedException on the target thread
- ◆ **Thread(Runnable o, String threadName)**
  - Constructor to generate a thread with the given name
- ◆ **public void dumpStack()**
  - Prints a stack trace of the current thread.

## Thread Related Methods of Object

- ◆ **public final void wait()** throws InterruptedException
  - Put the thread to waiting for a notification on the same object by another thread. Take less resources than sleep.
- ◆ **public final void notify()**
  - Awaken a single thread that is waiting on this object
- ◆ **public final void notifyAll()**
  - Awaken all threads waiting on this object
- ◆ These 3 methods must be invoked in a synchronized method

## Deprecated Thread Methods

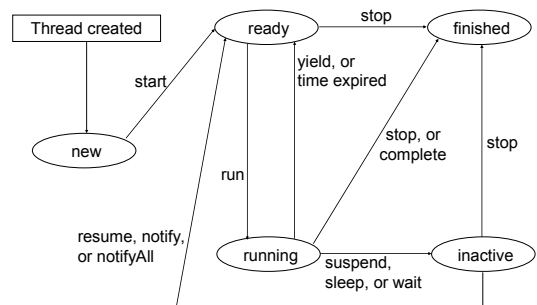
- ◆ These methods can cause dead-locks, and therefore should be avoided (more later)
- ◆ Using these methods, or overriding them, will cause compilation warnings
- ◆ `public void suspend()`
  - Suspends the thread, which will be awakened by `resume()`
- ◆ `public void resume()`
  - Awakens the suspended thread
- ◆ `public void stop()`
  - Terminates the thread without reclaiming resources

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-19

## Thread State Diagram



Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-20

## Thread Group

- ◆ A thread group is a set of threads that can be manipulated together
  - Create a new group

```
ThreadGroup g =
    new ThreadGroup("UniqueGroupName");
```
  - Add a thread to the group

```
Thread t = new Thread(g, new ThreadClass(),
    "thread label");
```
  - Find out number of active threads

```
g.activeCount()
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-21

## Thread Synchronization

- ◆ When multiple threads access a shared resource simultaneously, the shared resource may be corrupted. Keyword **synchronized** may help
- ◆ If a method is qualified by "synchronized" but not by "static", any time only one of such methods can be invoked on a particular object of the class type; but two invocations of such synchronized methods can happen at the same time if they are on different objects
- ◆ If a method is qualified by "static synchronized", any time only one of such methods can be invoked on any object of this class type

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-22

## Example for Resource Conflict

- ◆ `PiggyBankWithoutSync.java` contains an instance of `PiggyBank`, and 100 threads of `AddAPennyThread`
- ◆ Due to shared data access, different run has different final balance

```
MS-DOS Prompt
8 x 12
Final balance is 28
D:\data\teaching\c628\test\threads>java PiggyBankWithoutSync
Final balance is 24
D:\data\teaching\c628\test\threads>java PiggyBankWithoutSync
Final balance is 29
D:\data\teaching\c628\test\threads>
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-23

## PiggyBankWithoutSync.java (1/3)

```
public class PiggyBankWithoutSync {
    private PiggyBank bank = new PiggyBank();
    private Thread[] thread = new Thread[100];

    public static void main(String[] args) {
        PiggyBankWithoutSync test = new PiggyBankWithoutSync();
        System.out.println("Final balance is " + test.bank.getBalance());
    }

    public PiggyBankWithoutSync() {
        ThreadGroup g = new ThreadGroup("group");
        boolean done = false;

        for (int i = 0; i < 100; i++) {
            thread[i] = new Thread(g, new AddAPennyThread(), "t");
            thread[i].start();
        }
    }
}
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-24

## ■ PiggyBankWithoutSync.java (2/3)

```
while (!done)
    if (g.activeCount() == 0)
        done = true;
}
class AddAPennyThread extends Thread {
    public void run() {
        int newBalance = bank.getBalance() + 1;
        try {
            Thread.sleep(5);
        }
        catch (InterruptedException e) {
            System.out.println(e);
        }
        bank.setbalance(newBalance);
    }
}
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-25

## ■ PiggyBankWithoutSync.java (3/3)

```
class PiggyBank {
    private int balance = 0;

    public int getBalance() {
        return balance;
    }

    public void setbalance(int balance) {
        this.balance = balance;
    }
}
```

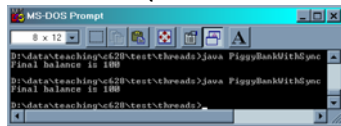
Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-26

## ■ PiggyBankWithSync.java (modified part)

```
public class PiggyBankWithSync {
    ..... // next line "static" can be dropped
    private static synchronized void addAPenny(PiggyBank bank) {
        int newBalance = bank.getBalance() + 1;
        try { Thread.sleep(5); }
        catch (InterruptedException e) { System.out.println(e); }
        bank.setbalance(newBalance);
    }
    class AddAPennyThread extends Thread {
        public void run() {
            addAPenny(bank);
        }
    }
}
```



Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-27

## ■ Monitor (1/4)

- ◆ To ensure object data integrity, some methods cannot be executed by more than one thread at a time
- ◆ Java uses *monitor* to support data protection. Each class has one *class monitor* for the class, and one *object monitor* for each object of the class
- ◆ The *class monitor* controls the access of threads to the *static synchronized methods* of the class
- ◆ An *object monitor* controls the access of threads to the non-static synchronized methods of an object of the class

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-28

## ■ Monitor (2/4)

- ◆ Each monitor has one lock and one key to control access to its synchronized methods. Only a thread holding the key can access these methods
- ◆ When no thread is executing in a monitor, the key is available. A thread grabs the key to execute in the monitor, holding the key. Upon exiting from the monitor, the thread returns the key
- ◆ Each monitor has two queues: the *entry queue* for threads waiting for the key to access the monitor, and the *wait queue* for threads that are blocked by invoking `wait()` when they were executing in a synchronized method

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-29

## ■ Monitor (3/4)

- ◆ When a thread invokes a synchronized method of a class, it first waits in the *entry queue* of the corresponding monitor
- ◆ If the key is available, the thread takes it and starts executing in the monitor. Otherwise it waits
- ◆ If the running thread exits the method, it returns the key, and the thread with the highest priority in the entry queue will take the key and start its execution in the monitor
- ◆ If the running thread calls `wait()`, it suspends itself on the wait queue, and releases the key for other threads in the entry queue to execute in the monitor

Java for C++ Programmers

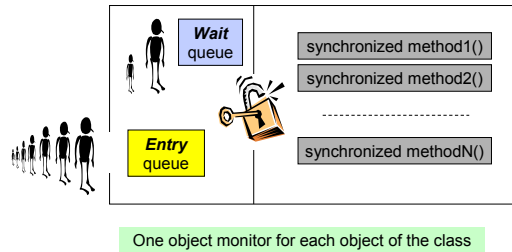
© Dr. Lixin Tao, 2000

08-30

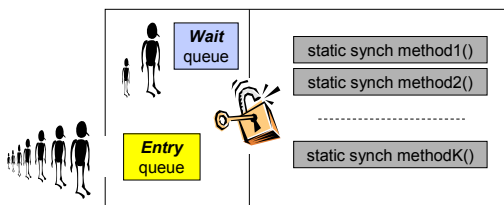
## Monitor (4/4)

- ◆ If the running thread calls `notify()` (`notifyAll()`), one (all) of the threads in the wait queue will be awakened and moved to the entry queue (for `notify()`, the one awakened is implementation-specific)
- ◆ `wait()`, `notify()`, and `notifyAll()` are methods of the monitor class (not of the thread class), and must be invoked in a synchronized method of a monitor of the class
- ◆ If the wait queue is empty, `notify()` and `notifyAll()` have no effect

## Monitor for Non-Static Synchronized Methods



## Monitor for Static Synchronized Methods



## Synchronized Block

- ◆ Synchronizing methods will increase execution time; minimizing units of synchronization can minimize the synchronization penalty
- ◆ A synchronized block has syntax
  - **synchronized** (object) statement;
 the statement will be executed in the monitor of the specified object
- ◆ For monitors, a synchronized block is equivalent to a non-static synchronized method

## Rewrite Deprecated Methods (1/4)

- ◆ Thread methods `suspend()` and `resume()` can deactivate and activate a particular thread; Object methods `wait()` and `notify()` cannot do the same thing directly
- ◆ Since `suspend()` may suspend a thread executing in a monitor without giving up its monitor key, it can lead to deadlock
- ◆ Thread methods `suspend()`, `resume()`, and `stop()` are deprecated and should be avoided
- ◆ The suggested codes are in pseudo-code to show ideas. Integrate them into your own code

## Rewrite Deprecated Methods (2/4)

- ◆ Assume a boolean variable dedicated to this thread

```
boolean suspended = false;
```

```
public synchronized void mySuspend() {
    suspended = true;
    while (suspended)
        wait();
}
```

### ■ Rewrite Deprecated Methods (3/4)

- ◆ Assume a boolean variable dedicated to this thread

```
boolean suspended = false;
```

```
public synchronized void myResume() {  
    if (suspended) {  
        suspended = false;  
        notifyAll();  
    }  
}
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-37

### ■ Rewrite Deprecated Methods (4/4)

```
public synchronized void myStop()  
{  
    thread = null;  
    notifyAll();  
}  
void run() {  
    .....  
    if (thread == null)  
        termination Processing;  
}
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-38

### ■ ThreadSynch.java (1/7)

```
// ThreadSynch.java demonstrates how to simulate the deprecated methods  
// suspend(), resume(), and stop(). It creates 3 threads, each prints random  
// letters. User can suspend or resume each thread with mouse.
```

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;
```

```
public class ThreadSynch extends JApplet  
    implements Runnable, ActionListener {  
    private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
    private JLabel outputs[];  
    private JCheckBox checkboxes[];  
    private final static int SIZE = 3;  
    private Thread threads[];  
    private boolean suspended[];
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-39

### ■ ThreadSynch.java (2/7)

```
public void init() {  
    outputs = new JLabel[SIZE];  
    checkboxes = new JCheckBox[SIZE];  
    threads = new Thread[SIZE];  
    suspended = new boolean[SIZE];  
    Container c = getContentPane();  
    c.setLayout(new GridLayout(SIZE, 2, 5, 5));  
    for (int i = 0; i < SIZE; i++) {  
        outputs[i] = new JLabel();  
        outputs[i].setBackground(Color.green);  
        outputs[i].setOpaque(true);  
        c.add(outputs[i]);  
        checkboxes[i] = new JCheckBox("Suspended");  
        checkboxes[i].addActionListener(this);  
        c.add(checkboxes[i]);  
    }  
}
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-40

### ■ ThreadSynch.java (3/7)

```
public void start() {  
    // create threads and start every time start is called  
    for (int i = 0; i < threads.length; i++) {  
        threads[i] =  
            new Thread( this, "Thread " + (i + 1) );  
        threads[i].start();  
    }  
}
```

```
public void run() {  
    Thread currentThread = Thread.currentThread();  
    int index = getIndex(currentThread);  
    char displayChar;
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-41

### ■ ThreadSynch.java (4/7)

```
while (threads[index] == currentThread) { // check for stopping the thread  
    // sleep from 0 to 1 second  
    try {  
        Thread.sleep((int)(Math.random()*1000));  
  
        synchronized(this) {  
            while (suspended[index] &&  
                threads[index] == currentThread)  
                wait();  
        }  
    }  
    catch (InterruptedException e) {  
        System.err.println("sleep interrupted");  
    }  
}
```

Java for C++ Programmers

© Dr. Lixin Tao, 2000

08-42

## ThreadSynch.java (5/7)

```
displayChar = alphabet.charAt((int)(Math.random()*26 ));
outputs[index].setText(currentThread.getName() +
    ":" + displayChar );
}
System.err.println(
    currentThread.getName() + " terminating");
}

private int getIndex(Thread current) {
    for (int i = 0; i < threads.length; i++)
        if (current == threads[i])
            return i;

    return -1;
}
```

## ThreadSynch.java (6/7)

```
public synchronized void stop() {
    // stop threads every time stop is called
    // as the user browses another Web page
    for (int i = 0; i < threads.length; i++)
        threads[i] = null;

    notifyAll();
}
```

## ThreadSynch.java (7/7)

```
public synchronized void actionPerformed( ActionEvent e ) {
    for ( int i = 0; i < checkboxes.length; i++ ) {
        if (e.getSource() == checkboxes[i]) {
            suspended[i] = !suspended[i];
            outputs[i].setBackground(
                !suspended[i] ? Color.green : Color.red);
            if (!suspended[i])
                notifyAll();
            return;
        }
    }
}
```

## ThreadSynch.html

```
<html>
<applet code="ThreadSynch.class" width=275 height=90>
</applet>
</html>
```