Threads

Threads versus Processes

Spring 2013
Processes and Threads

Process 1

- Stack
- CPU
- State
- Heap
- Code
- Environment

Process 2

- Stack
- CPU
- State
- Heap
- Code
- Environment

Processes and Threads

Process 1

- Thread 1
  - CPU
  - State
  - Stack
- Heap
- Code
- Environment

- Thread 2
  - CPU
  - State
  - Stack

- Thread 3
  - CPU
  - State
  - Stack

Spring 2013

CSE 370/369

4/17/2013
Tabs in Browsers

- In the past, in most browsers (e.g., Firefox, IE, Safari) tabs were implemented by using threads
- In Chrome, IE8 tabs are implemented using processes

- Which approach is better?

Terminology

- A **thread** is a single sequence of execution within a program
- **Multiprogramming**:
  - Running multiple processes concurrently
- **Multithreading**:
  - Running multiple threads within one process concurrently
  - That is, having several code executions within a single process
- Concurrency is achieved either by using several processors or by time-slicing over one processor
Multiple Threads in an Application

- Each thread has its own run-time stack and CPU state (i.e., register content, next instruction, etc.)
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
  - Why?
- However, all threads see the same dynamic memory, i.e., heap
  - Which variables are stored on the heap?
- Two different threads can act on the same object and same static fields concurrently
Why Threads?

- Improve the responsiveness of applications
  - i.e., allows user interaction while a task is running
- Improve utilization of resources
  - e.g., one thread can run while the other waits for I/O
- Provide a convenient programming technique
  - run resource cleaners in the background
  - good for monitoring
  - easy to separate tasks
  - graphical applications

Java Threads

API and how to use Threads
Class Thread

- We use the class `Thread` in order to create and run threads
- Two ways to create `Thread` objects:
  - Implementing `Runnable` and wrapping with `Thread`
  - Extending `Thread`
- `Thread` itself also implements `Runnable`, so in both ways you implement the method `run()`

Implementing Runnable

```java
public class MinusPrinter implements Runnable {
    public void run() {
        for (int i=0; i<1000; ++i) {
            System.out.print("-";
        }
    }

    Runnable r = new MinusPrinter();
    Thread t1 = new Thread(r);
}
```

When should we use this method of creating threads?
Extending Thread

```java
public class PlusPrinter extends Thread {
    public void run() {
        for (int i = 0; i < 1000; ++i) {
            System.out.print("+");
        }
    }
}
Thread t2 = new PlusPrinter();
```

Running Threads

- A Thread's execution starts by invoking its method `start()`
- This method invokes its method `run()` in a new thread

```java
public static void main(String argv[]) {
    Runnable r = new MinusPrinter();
    Thread t1 = new Thread(r);
    Thread t2 = new PlusPrinter();
    t1.start();
    t2.start();
}
```
Running Threads

- A Thread’s execution starts by invoking its method `start()`

```java
public static void main(String argv[]) {
    Runnable r = new MinusPrinter();
    Thread t1 = new Thread(r);
    Thread t2 = new PlusPrinter();
    t1.start();
    t2.start();
}
```

How many threads do we have here?

What would happen if we’ll change these lines to `t1.run(); t2.run();`
Java Thread Scheduling

Thread Scheduling

- Usually, threads run one at a time
  - Unless several processors are being used
- Thread execution is merged in order to provide concurrency
Scheduling Threads

- Currently executed thread
- Ready queue
- I/O operation completes
- Ready queue
- Blocked queue
- Waiting for I/O operation to be completed
- Waiting to be notified
- Sleeping
- Waiting to enter a synchronized section

What happens when a program with a ServerSocket calls accept()?

Thread State Diagram

- Alive
- Running
- Runnable
- Blocked
- Dead
- New
- Runnable
- Blocked
- Thread.sleep()
- Object.wait()
- Don’t get confused with Runnable interface

new PlusPrinter();
for (...) { ... }
thread.start();
run() method returns

Spring 2013
Thread Scheduling

- Thread **scheduling** is the mechanism used to determine how Runnable threads (in the ready queue) are allocated CPU time.
- Java scheduling is preemptive, priority based.
- Each thread has a priority - an integer number.
  - Use `thread.getPriority()`/`setPriority()` to control priorities.
- In principle, the runtime system chooses the thread that has the highest priority.

| Priority has range 1-10 | What is the priority of the main thread? |

Thread Scheduling (cont)

- If several threads have the same priority, an arbitrary one is chosen.
- Scheduling may violate the priority-based policy to avoid starvation of low-priority threads.
- Java’s scheduling also uses time slicing, when it is supported by the operating system.
  - mainly used for sharing CPU among equal highest-priority threads.
Thread Scheduling (cont)

A thread runs until one of the following occurs:

- The thread dies (e.g., run() completes)
- The thread becomes not Runnable (e.g., sleeps or waits for an IO operation to complete)
- A higher-priority thread becomes Runnable
- On systems that support time-slicing, its time allotment has expired
- The thread yields (discussed later)

Scheduling is OS Dependant!

- Java maps Java threads to OS threads
- In particular, Java relies on the operating system for
  - Time slicing
  - Priorities
- Thus, scheduling differs from one system to another
- Do not count on scheduling and priorities for algorithm correctness!
Relinquishing the CPU

- A running thread can explicitly relinquish the CPU for other threads to use.
- The static method `Thread.yield()` temporarily pauses the currently executing thread and allows other threads to execute.
- A thread can block for a while using the method `Thread.sleep(milliseconds)`.
- Could thread A call `Thread.yield()` and cause thread B to stop running?

```java
public class MinusPrinter implements Runnable {
    public void run() {
        for(int i=0; i<1000; ++i) {
            System.out.print("-");
            Thread.yield();
        }
    }
}

public class PlusPrinter extends Thread {
    public void run() {
        for(int i=0; i<1000; ++i) {
            System.out.print("+");
            Thread.yield();
        }
    }
}
```
public class MinusPrinter implements Runnable {
    public void run() {
        try {
            Thread.sleep(5);
        } catch (InterruptedException e) {}
        for (int i = 0; i < 1000; ++i) {
            System.out.print("-");
            Thread.yield();
        }
    }
}

public class PlusPrinter extends Thread {
    public void run() {
        for (int i = 0; i < 1000; ++i) {
            System.out.print("+");
            Thread.yield();
        }
    }
}
Daemon Threads

- There are two types of threads:
  - daemon threads (like the garbage-collection thread)
  - non-daemon threads (like the thread running main())
- JVM will let all non-daemon threads complete (i.e., complete the execution of run())
- When only daemon threads stay alive, they are killed and JVM exits
- Controlled by thread.isDaemon() and thread.setDeamon()
Garbage Collection

- The garbage collector of Java runs on a separate (daemon) thread
- An object is a candidate for garbage collection if this object *can no longer be accessed by any living thread*

Notes

- Threads inherit their priority and daemon properties from their creating threads
- The method `thread.join()` blocks and waits until the thread completes running
- A thread can have a `name` for identification
- Stopping a running thread was possible in old versions of Java, but it is now deprecated
  - Instead, `interruption` mechanisms should be used
Multithreading
Client-Server

Server

Request Handler

Client

Request Handler

Client
Multithreading Client-Server

- When a new request arrives, it is served in a new thread
- The server thread continues to listen
- Next, we will show an **EchoServer** that can handle concurrent requests

```java
import java.net.*; import java.io.*;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8000);
        
        while (true) {
            try {
                Socket socket = serverSocket.accept();
                new EchoRequestHandler(socket).start();
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }
        }
    }
}
```
public class EchoRequestHandler extends Thread {
    Socket socket = null;
    
    public EchoRequestHandler(Socket socket) {
        this.socket = socket;
    }
    
    public void run() {
        ... next slide ...
    }
}

public void run() {
    try {
        BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintStream writer = new PrintStream(socket.getOutputStream());
        String lineRead = null;
        while ((lineRead = reader.readLine()) != null) {
            writer.println("You wrote: " + lineRead);
            writer.flush();
        }
    } catch (IOException exp) {
        System.err.println("Error: " + exp.getMessage());
    } finally {
        try {
            if (!socket.isClosed()) socket.close();
        } finally {
            try {
                // do something
            } finally {
                // do something
            }
        }
    }
}
Thread Synchronization

- Consider the following consumer-producer scenario
- A single cookie can be placed in a jar and taken from it later.
public class CookieJar {
    int contents; boolean hasCookie = false;

    public void putCookie(String who, int value) {
        while (hasCookie) {}
        contents = value; hasCookie = true;
        System.out.println(who + " put cookie " + value);
    }

    public int getCookie(String who) {
        while (!hasCookie) {}
        hasCookie = false;
        System.out.println(who + " got cookie " + contents);
        return contents;
    }
}

horribly inefficient!!

try {
    Thread.sleep(1000);
} catch (InterruptedException e) {}
The Simpson Simulation (cont)

```java
public class Marge implements Runnable {
    CookieJar jar;

    public Marge(CookieJar jar) {
        this.jar = jar;
    }

    public void bake(int cookie) {
        jar.putCookie("Marge", cookie);
    }

    public void run() {
        for (int i = 0; i < 10; i++) bake(i);
    }
}
```

The Simpson Simulation (cont)

```java
public class RunSimpsons {

    public static void main(String[] args) {
        CookieJar jar = new CookieJar();
        Homer homer = new Homer(jar);
        Marge marge = new Marge(jar);
        new Thread(homer).start();
        new Thread(marge).start();
    }
}
```
Oops! Missed a Cookie!

```java
public class CookieJar {
    int contents; boolean hasCookie = false;

    public void putCookie(String who, int value) {
        while (hasCookie) { sleep(); }
        contents = value;
        hasCookie = true;
    }

    public int getCookie(String who) {
        while (!hasCookie) { sleep(); }
        hasCookie = false;
        return contents;
    }
}
```

<table>
<thead>
<tr>
<th>hasCookie</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>4</td>
</tr>
</tbody>
</table>

We start at a point where there's a cookie in the jar...

Race Condition Example

Put green pieces

How can we have alternating colors?

Put red pieces
Race Condition

- **Race condition** –
  - Two threads are simultaneously reading or modifying some shared data
  - The outcome of a program is affected by the order in which the program's threads are allocated CPU time
  - Both threads “race” for accessing shared data
  - When undesired, *synchronization* is required

Monitors and Locks

- **Monitors** are key elements in Java's thread synchronization
- **Every object** has a monitor
  - An object’s monitor is used as a guardian that watches a block of code (called a *critical section*) and enables *only one thread to enter that code*
  - To enter a critical section, a thread must first *acquire an ownership* over the corresponding monitor
Unique Lock Ownerships

- Only one thread can own a specific monitor
- If a thread $A$ tries to enter a block under a monitor and a different thread $B$ has already entered that block, $A$ will wait until $B$ releases the monitor and (hopefully) that monitor will be passed to $A$
  - Hence, monitors are related to as *locks*
- When a thread leaves the critical section, the monitor is automatically released
- Threads awaiting a monitor are *blocked* and queued

The *synchronized* keyword

- To monitor a block code using the monitor of Object $o$, use the *synchronized* keyword as follows:
  
  ```java
  synchronized(o) {
  \texttt{critical-section}
  }
  ```
- *synchronized method() {\texttt{critical-section}}* is a shorthand for

  ```java
  method() {
  synchronized(this) {
  \texttt{critical-section}
  }
  }
  ```
An Example

```java
public class BankAccount {

    private float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }

    public synchronized void transfer(float amount, BankAccount target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```

Critical Sections

[Diagram showing critical sections and BankAccount]
private String a = "hello";
private Date b = new Date();

void a() {
    synchronized (a) {
        System.out.println("In A 1");
        System.out.println("In A 2");
    }
}

void b() {
    synchronized (b) {
        System.out.println("In B 1");
        System.out.println("In B 2");
    }
}

Can we get A and B prints alternately?

synchronized void a() {
    System.out.println("In A 1");
    System.out.println("In A 2");
}

synchronized void b() {
    System.out.println("In B 1");
    System.out.println("In B 2");
}

static synchronized void a() {
    System.out.println("In A 1");
    System.out.println("In A 2");
}

static synchronized void b() {
    System.out.println("In B 1");
    System.out.println("In B 2");
}

static String c = "world";

void a() {
    synchronized (c) {
        System.out.println("In A 1");
        System.out.println("In A 2");
    }
}

void b() {
    synchronized (getClass()) {
        System.out.println("In B 1");
        System.out.println("In B 2");
    }
}

What about here?

Uses the monitor of the Class object

Can we get A and B prints alternately?
Synchronization Scopes

What will happen here?

```java
void a() {
    Date d = new Date();
    synchronized (d) {
        System.out.println("In A 1");
        System.out.println("In A 2");
    }
}
```

Constructors

- Can we add a `Synchronized` keyword before a constructor?
- Who *should* have access to an object while it is being constructed?
- Who *can* have access to an object while it is being constructed?
- That is, a constructor should not provide a reference to itself (i.e., `this`) to data structures shared by other threads
Back to the Simpsons

```java
public synchronized void putCookie(String who, int value) {
    while (hasCookie) { sleep(); }
    contents = value;
    hasCookie = true;
}

deadlock!

public synchronized int getCookie(String who) {
    while (!hasCookie) { sleep(); }
    hasCookie = false;
    return contents;
}
```

Another Deadlock Example

```java
public class BankAccount {

    private float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }

    public synchronized void transfer(float amount, BankAccount target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```
Deadlocks

wait() and notify()

- Suppose that an object has some monitor, but conditions disable it from completing the critical section
- The wait/notify mechanism enables that object to release the monitor and wait until conditions are changed
wait()

- The method `Object.wait()` requires the current thread to own the monitor of the object
- When called, the current thread
  - releases ownership on the object's monitor
  - stops and waits until some other thread will wake it up
    and the monitor will be re-obtained

notify()

- Like `wait`, requires the object to own the monitor
- The method `Object.notify()` wakes up an arbitrary thread that waits on the monitor of the object
- `Object.notifyAll()` wakes all such threads
- When a thread is waken up, it regularly waits for the monitor to be available (since it called `Object.wait()`)
- The thread calling `notify` should release the monitor for the waiting thread to continue (e.g. exit the synchronized scope)
Waiting and Notifying

```java
synchronized (lock) {
    while (!resourceAvailable()) { lock.wait(); }
    consumeResource();
}

produceResource();
synchronized (lock) { lock.notifyAll(); }
```

Wait/Notify Sequence

1. synchronized(lock){
2.   lock.wait();
3.   produceResource();
4.   lock.notify();
5.     lock.notifyAll();
6.    Reacquire lock
7.  Return from wait()
8.}

Consumer Thread

Producer Thread
Wait/Notify Sequence

1. synchronized(lock){
2.   lock.wait();
3.   produceResource();
4.   synchronized(lock) {
5.     lock.notify();
6.   }
7.   Reacquire lock
8.   Return from wait()
9.   consumeResource();
10. }

Spring 2013  cs236369

Wait/Notify Sequence

1. synchronized(lock){
2.   lock.wait();
3.   produceResource();
4.   synchronized(lock) {
5.     lock.notify();
6.   }
7.   Reacquire lock
8.   Return from wait()
9.   consumeResource();
10. }

Spring 2013  cs236369
Wait/Notify Sequence

1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }

3. produceResource();
4. synchronized(lock) {
5.   lock.notify();
6. }

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread
1. synchronized(lock) {
2.   lock.wait();
9.   consumeResource();
10. }
3. produceResource();
4. synchronized(lock) {
5.   lock.notify();
6. }
7. Reacquire lock
8. Return from wait()
1. `synchronized(lock){`
2.   `lock.wait();`
9.   `consumeResource();`
10. `}`

3. `produceResource();`
4. `synchronized(lock) {
5.   `lock.notify();`
6. }

7. Reacquire lock
8. Return from `wait()`

36
1. synchronized (lock) {
2.   lock.wait();
3. }
4. produceResource();
5. synchronized (lock) {
6.   lock.notify();
7. }
8. return from wait();

Consumer Thread

Producer Thread

Wait/Notify Sequence
Using Interruptions for Communication Between Threads

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.
- A thread sends an interrupt by invoking `interrupt` on the Thread object for the thread to be interrupted.

```java
public synchronized void putCookie(String who, int value) {
    while (hasCookie) {
        try { wait(); } catch (InterruptedException e){} 
    }
    contents = value; hasCookie = true;
    System.out.println(who + " put cookie " + value);
    notifyAll(); }

public synchronized int getCookie(String who) {
    while (!hasCookie) {
        try { wait(); } catch (InterruptedException e){} 
    } hasCookie = false;
    System.out.println(who + " got cookie " + contents);
    notifyAll();
    return contents; }
```
More Methods

- `isInterrupted()` – can be used by a thread to check if another thread has been interrupted
- `isAlive()` – can be used by a thread to check if another thread is alive
- `join()` – allows one thread to wait for the completion of another

Links


http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html