Overview

- What are threads?
  - Concept
  - Basic Java mechanisms
- Thread concerns
  - Safety and Liveness
  - Use of synchronization and signalling
- Threading design patterns

Administrivia

- Project 3
  - Due Nov. 8
- Midterms handed back now
  - Please pick yours up
- Project 1 regrades
  - Please see message on newsgroup
Processes vs. Threads

Processes do not share data

Threads share data within a process

So, What Is a Thread?

- **Conceptually**: it is a parallel computation occurring within a process
- **Implementation view**: it’s a program counter and a stack. The heap and static area are shared among all threads
- All programs have at least one thread (main)

Why Multiple Threads?

- **Performance**:
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Can easily express some programming paradigms
  - Event processing
  - Simulations
- Keep computations separate, as in an OS
  - Java OS

Why Not Multiple Threads?

- **Complexity**:
  - Dealing with safety, liveness, composition
- **Overhead**
  - Higher resource usage
- We’ll compare threads to their alternatives a bit later …
Programming Threads

- Threads are available in many languages
  - C, C++, Objective Caml, Java, SmallTalk ...
- In many languages (e.g., C and C++), threads are a platform specific add-on
  - Not part of the language specification
- Part of the Java language specification

Java Threads

- Every application has at least one thread
  - The “main” thread, started by the JVM to run the application’s main() method.
- The code executed by main() can create other threads
  - Explicitly, using the Thread class
  - Implicitly, by calling libraries that create threads as a consequence
    - RMI, AWT/Swing, Applets, etc.

Java Threads: Creation

- To explicitly create a thread
  - Instantiate a Thread object
    - An object of class Thread or a subclass of Thread
  - Invoke the object’s start() method
    - This will start executing the Thread’s run() method concurrently with the current thread
  - Thread terminates when its run() method returns
Running Example: Alarms

- Goal: let us set alarms that will be triggered in the future
  - Input: Time t (seconds) and message m
  - Result: We'll see m printed after t seconds

Example: Synchronous alarms

```java
while (true) {
  System.out.print("Alarm> ");
  // read user input
  String line = b.readLine();
  parseInput(line); // sets timeout
  // wait (in secs)
  try {
    Thread.sleep(timeout * 1000);
  } catch (InterruptedException e) { }
  System.out.println("(+timeout+) " + msg);
}
```

Making It Threaded (1)

```java
public class AlarmThread extends Thread {
  private String msg = null;
  private int timeout = 0;

  public AlarmThread(String msg, int time) {
    this.msg = msg;
    this.timeout = time;
  }

  public void run() {
    try {
      Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("(+timeout+) " + msg);
  }
}
```

Making It Threaded (2)

```java
while (true) {
  System.out.print("Alarm> ");
  // read user input
  String line = b.readLine();
  // creates AlarmThread to wait timeout secs
  Thread t = parseInput(line);
  // start alarm thread
  if (t != null)
    t.start();
}
```
Alternative: The **Runnable** Interface

- Extending **Thread** prohibits a different parent
- Instead implement **Runnable**
  - Declares that the class has a `void run()` method
- Construct a **Thread** from the **Runnable**
  - Constructor `Thread(Runnable target)`
  - Constructor `Thread(Runnable target, String name)`

Change is in **parseInput**

- Old `parseInput` does
  - `return new AlarmThread(m,t);`
- New `parseInput` does
  - `return new Thread(new AlarmRunnable(m,t));`
- Code in while loop doesn’t change

Notes: Passing Parameters

- **run()** doesn’t take parameters
- We “pass parameters” to the new thread by storing them as private fields
  - In the extended class
  - Or the **Runnable** object
  - Example: the time to wait and the message to print in the AlarmThread class
Thread Scheduling

- Once a new thread is created, how does it interact with existing threads?
- This is a question of scheduling:
  - Given N processors and M threads, which thread(s) should be run at any given time?

Thread Scheduling

- OS schedules a single-threaded process on a single processor
- Multithreaded process scheduling:
  - One thread per processor
    - Effectively splits a process across CPU’s
    - Exploits hardware-level concurrency
  - Many threads per processor
    - Need to share CPU in slices of time

Scheduling Example (1)

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 2</td>
<td>p1</td>
<td>p2</td>
</tr>
</tbody>
</table>

One process per CPU

Scheduling Example (2)

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 2</td>
<td>p1</td>
<td>p2</td>
</tr>
</tbody>
</table>

Threads shared between CPU’s
Scheduling Consequences

- Concurrency
  - Different threads from the same application can be running at the same time on different processors
- Interleaving
  - Threads can be pre-empted at any time in order to schedule other threads

Thread Scheduling

- When multiple threads share a CPU, must decide:
  - When the current thread should stop running
  - What thread to run next
- A thread can voluntarily yield() the CPU
  - Call to yield may be ignored; don’t depend on it
- Preemptive schedulers can de-schedule the current thread at any time
  - Not all JVMs use preemptive scheduling, so a thread stuck in a loop may never yield by itself. Therefore, put yield() into loops
- Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep

Thread Lifecycle

- While a thread executes, it goes through a number of different phases
  - New: created but not yet started
  - Runnable: is running, or can run on a free CPU
  - Blocked: waiting for I/O or on a lock
  - Sleeping: paused for a user-specified interval
  - Terminated: completed

Which Thread to Run Next?

- The scheduler looks at all of the runnable threads, including threads that were unblocked because
  - A lock was released
  - I/O became available
  - They finished sleeping, etc.
- Of these threads, it considers the thread’s priority. This can be set with setPriority(). Higher priority threads get preference.
  - Oftentimes, threads waiting for I/O are also preferred.
Simple Thread Methods

- void start()
- boolean isAlive()
- void setPriority(int newPriority)
  - Thread scheduler might respect priority
- void join() throws InterruptedException
  - Waits for a thread to die/finish

Simple Static Thread Methods

- void yield()
  - Give up the CPU
- void sleep(long milliseconds)
  - throws InterruptedException
  - Sleep for the given period
- Thread currentThread()
  - Thread object for currently executing thread
- All apply to thread invoking the method

Example: Threaded, Sync Alarm

```java
while (true) {
    System.out.print("Alarm> ");
    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);
    // wait (in secs) asynchronously
    if (t != null) 
        t.start();
    // wait for the thread to complete
    t.join();
}
```

Daemon Threads

- void setDaemon(boolean on)
  - Marks thread as a daemon thread
  - Must be set before thread started
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons
# Concurrency Issues

- Threads allow concurrent activities, which can be both good and bad!
- Two opposing design forces
  - **Safety**: “Nothing bad ever happens”
  - **Liveness**: “Something (useful) eventually happens”
- A safe system may not be live and a live system may not be safe. Balance is key.

# Systems = Objects + Activities

- **Safety** is a property of objects, and groups of objects, that participate across multiple activities.
  - Can be a concern at many different levels: objects, composites, components, subsystems, hosts, ...
- **Liveness** is a property of activities, and groups of activities, that span across multiple objects.
  - Levels: Messages, call chains, threads, sessions, scenarios, scripts workflows, use cases, transactions, data flows, mobile computations, ...

# Safe Objects

- Perform actions only when in consistent states
  - Don’t want one thread to access an object while another thread is modifying its internal state.
- This boils down to ensuring *object invariants* in the face of concurrent access

# Violating Safety

- Data can be shared by threads
  - Scheduler can interleave or overlap threads arbitrarily
  - Can lead to *interference*
    - Storage corruption (e.g., a *data race/race condition*)
    - Violation of representation invariant
    - Violation of a protocol (e.g., A occurs before B)
public class Example extends Thread {
    private static int cnt = 0;  // shared state
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }
    public static void main(String args[]) {
        Thread t1 = new Example();
        Thread t2 = new Example();
        t1.start();
        t2.start();
    }
}

static int cnt = 0;  // shared state
Start: both threads ready to run. Each will increment the global count.

t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

cnt = 0
T1 executes, grabbing the global counter value into y.

cnt = 1
T1 executes again, storing the counter value
Data Race Example

```c
static int cnt = 0;  // Shared state  cnt = 1

t1.run() {
    int y = cnt;   y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;   y = 1
    cnt = y + 1;  // T1 finishes. T2 executes, grabbing the global counter value into y.
}
```

Data Race Example

```c
static int cnt = 0;  // Shared state  cnt = 2

t1.run() {
    int y = cnt;   y = 0
    cnt = y + 1;
}

t2.run() {
    int y = cnt;   y = 1
    cnt = y + 1;  // T2 executes, storing the incremented cnt value.
}
```

But When I Run it Again?

```c
static int cnt = 0;  // Shared state  cnt = 0

t1.run() {
    int y = cnt;
    cnt = y + 1;
}

t2.run() {
    int y = cnt;
    cnt = y + 1;  // Start: both threads ready to run. Each will increment the global count.
}
```
Data Race Example

static int cnt = 0;   // Shared state  cnt = 0
t1.run() {           
    int y = cnt;      // T1 executes, grabbing
    y = 0             // the global counter value into y.
    cnt = y + 1;     // cnt = y + 1;
}                     

T1 executes, grabbing
the global counter value into y.

shared state
y = 0
y = 0

t2.run() {           
    int y = cnt;      // T2 executes, storing the
    y = 0             // incremented cnt value.
    cnt = y + 1;     // cnt = y + 1;
}                     

T1 is pre-empted. T2
executes, grabbing the global
counter value into y.

shared state
y = 0
y = 0
What Happened?

- In the second example, t1 was preempted after it read the counter but before it stored the new value.
  - Depends on the idea of an atomic action
  - Violated an object invariant
- A particular way in which the execution of two threads is interleaved is called a schedule. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

Question

- If instead of
  ```java
  int y = cnt;
  cnt = y+1;
  ```
  We had written
  ```java
  cnt++;
  ```
  Would the result be any different?
- Answer: NO!
  - Don’t depend on your intuition about atomicity

Question

- If you run a program with a race condition, will you always get an unexpected result?
  - No! It depends on the scheduler
  - ...i.e., which JVM you're running
  - ...and on the other threads/processes/etc that are running on the same CPU
- Race conditions are hard to find