Overview

• What are threads?
• Thread scheduling, data races, and synchronization
• Thread mechanisms in Java

Computation Abstractions

Processes vs. Threads

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

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
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
- The thread API differs with each, but most have the basic features we will now present

Thread Applications

- Web browsers
  - one thread for I/O
  - one thread for each file being downloaded
  - one thread to render web page
- Graphical User Interfaces (GUIs)
  - Have one thread waiting for each important event, like key press, button press, etc.

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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 2</td>
<td>p1</td>
<td>p2</td>
</tr>
</tbody>
</table>

One process per CPU

<table>
<thead>
<tr>
<th>p2 threads:</th>
<th>p1 threads:</th>
</tr>
</thead>
</table>

Scheduling Example (2)

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

Threads shared between CPU’s

<table>
<thead>
<tr>
<th>p2 threads:</th>
<th>p1 threads:</th>
</tr>
</thead>
</table>

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
Data Races

- Data can be shared by threads
  - Scheduler can interleave threads arbitrarily
  - Can lead to unexpected data corruption
  - This is a data race

- Need to avoid such data accesses.
  - Use synchronization.

Data Race Example

```c
int cnt = 0; Shared state  cnt = 0
void thread1() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
void thread2() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```c
int cnt = 0; Shared state  cnt = 0
void thread1() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
void thread2() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
```

Thread1 is pre-empted. Thread2 executes, grabbing the global counter value into y.

Data Race Example

```c
int cnt = 0; Shared state  cnt = 1
void thread1() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
void thread2() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
```

Thread2 executes, storing the incremented cnt value.

Data Race Example

```c
int cnt = 0; Shared state  cnt = 1
void thread1() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
void thread2() {
  int y = cnt;  y = 0
  cnt = y + 1;
}
```

Thread2 completes. Thread1 executes again, storing the old counter value (1) rather than the new one (2)!
What happened?

• Thread1 was preempted after it read the counter but before it stored the new value.
• 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.

Applying synchronization

```java
int cnt = 0;
Object lock = new Object();
void thread1() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

```
Thread1 reads cnt into y
Thread1 runs, assigning to cnt
```
Applying synchronization

```java
int cnt = 0;
Object lock = new Object();
void thread1() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state: cnt = 1

Thread1 releases the lock and terminates

Thread2 reads cnt into y.

Applying synchronization

```java
int cnt = 0;
Object lock = new Object();
void thread1() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state: cnt = 1

Thread2 now can acquire lock l.

Thread2 assigns cnt, then releases the lock

Synchronization not a Panacea

• Two threads can block on locks held by the other; this is called deadlock

```java
Object lock = new Object();
Object lock2 = new Object();
void thread1() {
    synchronized (lock) {
        synchronized (lock2) {
            ... 
        }
    }
}
void thread2() {
    synchronized (lock) {
        synchronized (lock2) {
            ... 
        }
    }
}
```

Other Thread Operations

• **Condition variables**: `wait` and `notify`
  – Alternative synchronization mechanism

• **Yield**
  – Voluntarily give up the CPU

• **Sleep**
  – Wait for a certain length of time
Thread Lifecycle

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

Java Threads

- The class java.lang.Thread
  - Implements the basic threading abstraction
  - Can extend this class to create your own threads
- The interface java.lang.Runnable
  - Can create a thread by passing it a class that implements this interface
  - Favors composition over inheritance; more flexible

Extending class Thread

- Can build a thread class by extending java.lang.Thread
- Must supply a public void run() method
- Start a thread by invoking the start() method
- When a thread starts, executes run()
- When run() returns, thread is finished/dead

Example: Synchronous alarms

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

- Extending Thread means can’t extend any other class
- Instead implement Runnable
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)

Thread example revisited

```java
public class AlarmRunnable implements Runnable {
    private String msg = null;
    private int timeout = 0;

    public AlarmRunnable(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);
    }
}
```

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