CMSC 330: Organization of Programming Languages

Reminders

- Homework 2 due on Oct. 30
- Project 3 due Oct. 31
- Midterm 2 on Nov. 1

- Done with OCaml… now onto Threads…
Computation Abstractions

A computer

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)

**Implementation View**

- Per-thread stack and instruction pointer
  - Saved in memory when thread suspended
  - Put in hardware esp/eip when thread resumes
Tradeoffs

- Threads can increase performance
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Natural fit for some programming patterns
  - Event processing
  - Simulations
- But increased complexity
  - Need to worry about safety, liveness, composition
- And higher resource usage

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
- They're 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

- `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.

Thread Creation

![Thread Creation Diagram]

- `main thread`
- `thread starts`
- `thread starts`
- `thread ends`
- `thread join`
Thread Creation in Java

- 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's set alarms which 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);
}
```

like phone calls

thrown when another thread calls interrupt

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)

while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);
    if (m != null) {
        // start alarm thread
        Thread t = new AlarmThread(m,tm);
        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)
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");
    }
}

while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);
    if (m != null) {
        // start alarm thread
        Thread t = new Thread(
            new AlarmRunnable(m,tm));
        t.start();
    }
}
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

Concurrency

• A concurrent program is one that has multiple threads that may be active at the same time
  – Might run on one CPU
    • The CPU alternates between running different threads
    • The scheduler takes care of the details
      – Switching between threads might happen at any time
  – Might run in parallel on a multiprocessor machine
    • One with more than one CPU
    • May have multiple threads per CPU

• Multiprocessor machines are becoming more common
  – Multi-CPU machines aren't that expensive any more
  – Dual-core CPUs are available now
Scheduling Example (1)

CPU 1  
| p1  
| p2

CPU 2  
| p1  
| p2

One process per CPU

Scheduling Example (2)

CPU 1  
| p1  
| p2

CPU 2  
| p1  
| p2

Threads shared between CPUs
Concurrency and Shared Data

- Concurrency is easy if threads don’t interact
  - Each thread does its own thing, ignoring other threads
  - Typically, however, threads need to communicate with each other

- Communication is done by *sharing* data
  - In Java, different threads may access the heap simultaneously
  - But the scheduler might interleave threads arbitrarily
  - Problems can occur if we’re not careful.

Data Race Example

```java
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();
    }
}
```
Data Race Example

```java
static int cnt = 0;

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

*Shared state*  
\( cnt = 0 \)

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

---

T1 executes, grabbing the global counter value into its own \( y \).
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

**Shared state**  \( \text{cnt} = 1 \)

\( y = 0 \)

\( T1 \) executes again, storing its value of \( y + 1 \) into the counter.

\( y = 1 \)

\( T1 \) finishes. \( T2 \) executes, grabbing the global counter value into its own \( y \).
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state \(\text{cnt} = 2\)

\(y = 0\)

\(y = 1\)  
*\(T2\) executes, storing its incremented \(\text{cnt}\) value into the global counter.*

But When it's Run Again?
Data Race Example

```java
static int 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.

Shared state cnt = 0

T1 executes, grabbing the global counter value into its own y.

y = 0
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state \( \text{cnt} = 0 \)

T1 is preempted. T2 executes, grabbing the global counter value into its own \( y \).

```
cnt = 1
```

T2 executes, storing the incremented \( \text{cnt} \) value.
Data Race Example

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

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

**Shared state**  
**cnt = 1**

T2 completes. T1 executes again, storing the incremented original counter value (1) rather than what the incremented updated value would have been (2)!

What Happened?

- Different schedules led to different outcomes
  - This is a data race or race condition
- A thread was preempted in the middle of an operation
  - Reading and writing cnt was supposed to be atomic-to happen with no interference from other threads
  - But the schedule (interleaving of threads) which was chosen allowed atomicity to be violated
  - These bugs can be extremely hard to reproduce, and so hard to debug
    - Depends on what scheduler chose to do, which is hard to predict
Question

• If instead of
  ```java
  int y = cnt;
  cnt = y+1;
  ```
• We had written
  ```java
  - cnt++;
  ```
• Would the result be any different?
• Answer: NO!
  ```java
  - 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?
  ```java
  - 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
What’s Wrong with the Following?

```
static int cnt = 0;
static int x = 0;
```

```
Thread 1
  while (x != 0);
  x = 1;
  cnt++;
  x = 0;

Thread 2
  while (x != 0);
  x = 1;
  cnt++;
  x = 0;
```

- Threads may be interrupted after the `while` but before the assignment `x = 1`
  - Both may think they “hold” the lock!
- This is *busy waiting*
  - Consumes lots of processor cycles