CMSC 330: Organization of Programming Languages

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 processes have at least one thread (main)
  – Programs vs. processes
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

execution (time)

main thread

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: To 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
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);
    }
}
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)
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);
    }
}
```
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 class that extends **Runnable**
  - 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)

One process per CPU
Scheduling Example (2)

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;
}
```

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

Shared state \( cnt = 0 \)
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*  
```
cnt = 0
```

```
y = 0
```

*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.*
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**  \( cnt = 1 \)

\( y = 0 \)

\( y = 1 \)

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

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

Shared state  cnt = 2

T2 executes, storing its incremented 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
Data Race Example

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

Shared state  
cnt = 0

y = 0

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*  \( \text{cnt} = 0 \)

\( y = 0 \)

\( y = 0 \)

*\( T1 \) is preempted. \( T2 \) 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;
}
```

```plaintext
Shared state  cnt = 1
y = 0

T2 executes, storing the incremented 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;
}
```

**T2 completes. T1 executes again, storing the incremented original counter value (1) rather than what the incremented updated value would have been (2)![1]
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
  ```c
  int y = cnt;
  cnt = y+1;
  ```

• We had written
  ```c
  – cnt++;
  ```

• Would the result be any different?

• Answer: NO!
  ```c
  – 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
What’s Wrong with the Following?

```java
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
Aside: Functional Programming

• No side effects
• No memory access
• No data races!
• Easier to parallelize functional programs
Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- Different languages have adopted different mechanisms to allow the programmer to synchronize threads.
- Java has several mechanisms; we'll look at locks first.
Locks (Java 1.5)

```java
interface Lock {
    void lock();
    void unlock();
    ... /* Some more stuff, also */
}

class ReentrantLock implements Lock { ... }
```

- Only one thread can hold a lock at once
  - Other threads that try to acquire it block (or become suspended) until the lock becomes available

- Reentrant lock can be reacquired by same thread
  - As many times as desired
  - No other thread may acquire a lock until has been released same number of times it has been acquired
Avoiding Interference: Synchronization

public class Example extends Thread {
    private static int cnt = 0;
    static Lock foo = new ReentrantLock();
    public void run() {
        foo.lock();
        int y = cnt;
        cnt = y + 1;
        foo.unlock();
    }
}

Lock, for protecting the shared state
Acquires the lock; Only succeeds if not held by another thread
Releases the lock
Applying Synchronization

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

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

*T1 acquires the lock*
Applying Synchronization

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

*Shared state*  
\[ \text{cnt} = 0 \]

*\( y = 0 \)*

*T1 reads \( \text{cnt} \) into \( y \)*
Applying Synchronization

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

**Shared state**  \( cnt = 0 \)

\( y = 0 \)

**T1 is preempted.**

**T2 attempts to acquire the lock but fails because it’s held by T1, so it blocks**
Applying Synchronization

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

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

\( y = 0 \)

\( T1 \) runs, assigning to \( \text{cnt} \)
Applying Synchronization

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

Shared state  cnt = 1

\[ y = 0 \]

\textit{T1 releases the lock and terminates}
Applying Synchronization

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

*Shared state*  
\( cnt = 1 \)

\( y = 0 \)

*T2 now can acquire the lock.*
Applying Synchronization

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

`Shared state` \( cnt = 1 \)

\( y = 0 \)

\( T2 \) reads \( cnt \) into \( y \).

\( y = 1 \)
Applying Synchronization

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

**Shared state**  
\( cnt = 2 \)

\( y = 0 \)  
\[
\begin{array}{cccccccc}
\text{Red} & \text{Blue} & \text{Green} & \text{Red} & \text{Blue} & \text{Green} & \text{Red} & \text{Blue} \\
\end{array}
\]

\( T2 \) assigns \( cnt \),  
then releases the lock

\( y = 1 \)
Different Locks Don’t Interact

static int cnt = 0;
static Lock l =
    new ReentrantLock();
static Lock m =
    new ReentrantLock();

void inc() {
    l.lock();
    cnt++;
    l.unlock();
}

void inc() {
    m.lock();
    cnt++;
    m.unlock();
}

• This program has a race condition
  – Threads only block if they try to acquire a lock held by another thread
Reentrant Lock Example

- Reentrancy is useful because each method can acquire/release locks as necessary
  - No need to worry about whether callers have locks
  - Discourages complicated coding practices

```java
static int cnt = 0;
static Lock l =
    new ReentrantLock();

void inc() {
    l.lock();
    cnt++;
    l.unlock();
}

void returnAndInc() {
    int temp;
    l.lock();
    temp = cnt;
    inc();
    l.unlock();
}
```