**CMSC 433 – Programming Language Technologies and Paradigms**  
**Spring 2007**

**Threads and Synchronization**  
**May 8, 2007**

**Computation Abstractions**

**Processes vs. Threads**

- Processes do not share data
- Threads share data within a process

- int x;
  - foo() {
    - ...x...
  }

- int x;
  - foo() {
    - ...x...
  }

**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

Why Not Multiple Threads?

- Complexity
- Overhead
  - 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
- 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.
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

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)`

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
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?

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): One process per CPU

Scheduling Example (2): 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)
  - Sleep for the given period
- Thread currentThread()
  - Thread object for currently executing thread
- All apply to thread invoking the method

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)

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

static int cnt = 0;  

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;
}

T1 executes, grabbing the global counter value into y.

Shared state

y = 0

T1 executes again, storing the counter value

T1 finishes. T2 executes, grabbing the global counter value into y.

25

26

27

28
Data Race Example

```java
static int cnt = 0;

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

void t2.run() {
    int y = cnt;  // start: both threads ready to run. Each will increment the
    cnt = y + 1;
}
```

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

But When I Run it Again?

```java
static int cnt = 0;

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

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

Shared state: cnt = 2

T2 executes, storing the incremented cnt value.
Data Race Example

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

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

t2.run() {  // T1 is pre-empted.  T2 executes, grabbing the global counter value into y.
    int y = cnt;  y = 0
    cnt = y + 1;
}
```

33

Data Race Example

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

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

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

34

Data Race Example

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

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

t2.run() {  // T2 completes.  T1 executes again, storing the old counter value (1) rather than the new one (2)!
    int y = cnt;  y = 0
    cnt = y + 1;
}
```

35

Data Race Example

```
What Happened?

- In the second example, \texttt{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.
```

36
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

Avoiding Interference: Synchronization

```
public class Example extends Thread {
    private static int cnt = 0;
    static Object lock = new Object();
    public void run() {
        synchronized (lock) {
            int y = cnt;
            cnt = y + 1;
        }
    }
    ...
}
```

Applying Synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```
Applying Synchronization

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

T1 reads cnt into y

Shared state  cnt = 0

---

Applying Synchronization

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

T1 is pre-empted. T2 attempts to acquire the lock but fails because it’s held by T1, so it blocks

Shared state  cnt = 0

---

Applying Synchronization

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

T1 runs, assigning to cnt

Shared state  cnt = 1

---

Applying Synchronization

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

T1 releases the lock and terminates

Shared state  cnt = 1
Applying Synchronization

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

Shared state  cnt = 1

T2 now can acquire the lock.

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

Shared state  cnt = 1

T2 reads cnt into y.

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

Shared state  cnt = 2

T2 assigns cnt, then releases the lock.

Locks

- Any Object subclass has (can act as) a lock
- Only one thread can hold the lock on an object
  - Other threads block until they can acquire it
- If a thread already holds the lock on an object
  - The thread can reacquire the same lock many times
    - Locks are reentrant
  - Lock is released when object unlocked the corresponding number of times
- No way to only attempt to acquire a lock
  - Either succeeds, or blocks the thread
Synchronized Statement

- `synchronized (obj) { statements }`
- Obtains the lock on `obj` before executing statements in block
- Releases the lock when the statement block completes
  - Either normally, or due to a return, break, or exception being thrown in the block

Synchronization not a Panacea

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

```java
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```

Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on `A`
  - Thread 2 holds lock on `B`
  - Thread 1 is trying to acquire a lock on `B`
  - Thread 2 is trying to acquire a lock on `A`
  - Deadlock!

- Not easy to detect when deadlock has occurred
  - Other than by the fact that nothing is happening

Deadlock: Wait graphs

- Thread T1 holds lock A
- Thread T2 attempting to acquire lock B
- Deadlock occurs when there is a cycle in the graph
Wait graph example

- T1 holds lock on A
- T2 holds lock on B
- T1 is trying to acquire a lock on B
- T2 is trying to acquire a lock on A

Guidelines for Programming with Threads

- Synchronize access to shared data
- Don’t hold multiple locks at a time
  - Could cause deadlock
- Hold a lock for as little time as possible
  - Reduces blocking waiting for locks
- While holding a lock, don’t call a method you don’t understand
  - E.g., a method provided by someone else, especially if you can’t be sure what it locks
  - Corollary: document which locks a method acquires