CMSC 433 – Programming Language Technologies and Paradigms
Spring 2003

Threads and Synchronization
April 3, 2003

Key Ideas
• Multiple threads can run simultaneously
  – Either truly in parallel on a multiprocessor
  – Or can be scheduled on a single processor
    • A running thread can be pre-empted at any time
• Threads can share data
  – In Java, only fields can be shared
  – Use synchronization to prevent data races

Data Races
int cnt = 0;
void thread1() {
  cnt++;
}
void thread2() {
  cnt++;
}

• Assuming thread1() and thread2() are separate threads, does this program have a data race?
  – Yes! Do not assume any operations are atomic

Avoiding Data Races
int cnt = 0;
Object lock;
void thread1() {
  synchronized(lock) {
    cnt++;
  }
}
void thread2() {
  synchronized(lock) {
    cnt++;
  }
}

• Use synchronization when modifying shared data

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

Creating Threads using extend
[From Java 1.4.1 API Doc]
class PrimeThread extends Thread {
  long minPrime;
  PrimeThread(long m) { minPrime = m; }
  public void run() {
    // do a lot of computation
  }
  PrimeThread p = new PrimeThread(143);
p.start();
Running Threads

- Thread not started until start() method called
- start() method executes run()
- Thread terminates when run() returns

Creating Threads using Runnable

[From Java 1.4.1 API Doc]

class PrimeRun implements Runnable {
    long minPrime;
    PrimeThread(long m) { minPrime = m; }
    public void run() {
        // do a lot of computation
    }
    PrimeRun p = new PrimeRun(143);
    new Thread(p).start();
    // Thread calls p.run() when it starts

Question

- Why use Runnable? Why not just extend thread?
  - Can implement many interfaces; can only extend one class

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

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
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();
}
```

Simple static thread methods

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

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

Synchronization Topics

- Locks
- `synchronized` statements and methods
- `wait` and `notify`
- Deadlock

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
  - 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
Recasting earlier example

```java
public class State {
    public int cnt = 0;
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) {
        this.s = s;
    }
    public void main(String[] args) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
}
```

Unsynchronized access to shared data!

Adding Synchronization

```java
public class State {
    public int cnt = 0;
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) {
        this.s = s;
    }
    public void main(String[] args) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
    synchronized (s) {
        int y = s.cnt;
        s.cnt = y + 1;
    }
}
```

Uses s as a lock, forces exclusive access

Synchronized methods

- A method can be synchronized
  - add `synchronized` modifier before return type
- Obtains the lock on object referenced by `this` before executing method
  - releases lock when method completes
- For a static synchronized method
  - locks the class object

Synchronization example

```java
public class State {
    private int cnt = 0;
    public synchronized incCnt(int x) {
        cnt += x;
    }
    public synchronized getCnt() {
        return cnt;
    }
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) {
        this.s = s;
    }
    public void main(String[] args) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
    synchronized (s) {
        s.incCnt(1);
    }
}
```

Synchronization occurs in State object itself, rather than in its caller.

Synchronization Style

- Design decision
  - Internal synchronization (class is thread-safe)
    - Have a stateful object synchronize itself (e.g. with synchronized methods)
  - External synchronization (class is thread-compatible)
    - Have callers perform synchronization before calling the object
- Can go both ways:
  - Thread-safe: Random
  - Thread-compatible: ArrayList, HashMap, …

Condition Variables

- Want access to shared data, but only when some condition holds
  - Implies that threads play different roles in accessing shared data
- Examples
  - Want to read variable v, but only when it is non-null
  - Want to insert myself in a data structure, but only if it is not full
CVs: Use Wait and Notify

To wait for a condition to become true:

```java
synchronized (obj) {
    while (condition does not hold)
        obj.wait();
    ... perform appropriate actions
}
```

To notify waiters that a condition has changed:

```java
synchronized (obj) {
    ... perform actions that change condition
    obj.notifyAll();
    or obj.notify(); // using notify() is a tricky optimization
}
```

Wait and Notify (cont.)

- `a.notify()` resumes one thread from a’s wait set
  - no control over which thread
- `a.notifyAll()` resumes all threads on a’s wait set
- resumed thread(s) must reacquire lock before continuing
  - (Java inserts the reacquire automatically)

Broken Producer/Consumer Example

```java
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

    synchronized void produce(Object o) {
        while (valueReady) {};
        synchronized (this) {
            value = o; valueReady = true;
        }
    }

    synchronized Object consume() {
        while (!valueReady) {};
        synchronized (this) {
            if (!valueReady) continue;
            valueReady = false;
            Object o = value; value = null;
            return o;
        }
    }
}
```
Producer/Consumer Example

```java
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

    synchronized void produce(Object o) throws InterruptedException {
        while (valueReady) wait();
        value = o; valueReady = true;
        notifyAll();
    }

    synchronized Object consume() throws InterruptedException {
        while (!valueReady) wait();
        valueReady = false;
        Object o = value; // why do we do this?
        value = null; // why do we do this?
        notifyAll();
        return o;
    }
}
```

notify() vs. notifyAll()

- Very tricky to use notify() correctly
  - notifyAll() generally much safer
- To use notify() correctly, should:
  - have all waiters be equal
  - each notify only needs to wake up 1 thread
  - doesn’t matter which thread it is
  - handle exceptions correctly
    - including InterruptedException
- For this course, just use notifyAll()

Thread Cancellation

- Example scenarios: want to cancel thread
  - whose processing the user no longer needs (i.e. she has hit the “cancel” button)
  - that computes a partial result and other threads have encountered errors, … etc.
- Java used to have Thread.kill()
  - But it and Thread.stop() are deprecated
  - Use Thread.interrupt() instead

Thread.interrupt()

- Tries to wake up a thread
  - Sets the thread’s interrupted flag
- Won’t disturb the thread if it is working

Cancellation Example

```java
public class CancellableReader extends Thread {
    private FileInputStream dataFile;
    public void run() {
        try {
            while (!Thread.interrupted()) {
                try {
                    int c = dataFile.read();
                    if (c == -1) break;
                    process(c);
                } catch (IOException ex) { break; }
            }
        } finally { // cleanup here }
    }
}
```

What if the thread is blocked on a lock or wait set, etc.

InterruptedException

- Thrown if interrupted while doing a wait, sleep, or join
  - Also thrown when interrupt flag is set and attempt to do a wait, sleep, or join
  - Not thrown when blocked (or blocking on) on a lock or I/O
- Must reset invariants before cancelling
  - E.g., closing file descriptors, notifying other waiters, etc.
InterruptedException Example

• Threads t1 and t2 are waiting
• Thread t3 performs a notify
  – thread t1 is selected
• Before t1 can acquire lock, t1 is interrupted
• t1’s call to wait throws InterruptedException
  – t1 doesn’t process notification
  – t2 doesn’t wake up

Handling InterruptedException

```java
synchronized (this) {
    while (!ready) {
        try { wait(); } 
        catch (InterruptedException e) {
            // make shared state acceptable
            notifyAll();
            // cancel processing
            return;
        }
        // do whatever
    }
}
```

Why no Thread.kill()?  

• What if the thread is holding a lock when it is killed? The system could
  – free the lock, but the data structure it is protecting might be now inconsistent
  – keep the lock, but this could lead to deadlock
• A thread needs to perform its own cleanup
  – Use InterruptedException and isInterrupted() to discover when it should cancel