CMSC 330: Organization of Programming Languages

Multithreaded Programming in Java

Multiprocessors

- Description
  - Multiple processing units (multiprocessor)
  - From single microprocessor to large compute clusters
  - Can perform multiple tasks in parallel simultaneously

Dual-core AMD Athlon X2
Dual-core AMD Athlon X2

32 processor Pentium Xeon

106K processor IBM BlueGene/L

Computation Abstractions

Processes do not share data

Threads share data within a process

Processes vs. Threads

A computer

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

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

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

int x;
foo() {
...x...
}
So, What Is a Thread?

- Conceptually
  - Parallel computation occurring within a process

- Implementation view
  - Program counter and stack
  - Heap and static area 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++, OCaml, Java, Ruby, ...

- In many languages (e.g., C and C++), threads are a platform specific add-on
  - Not part of the language specification
  - Implemented as code libraries (e.g., pthreads)

- 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 joins

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);
}
```

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();
    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);
    }
}
```

Thread Example Revisited (2)

```java
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
Concurrent and Shared Data

- Concurrency is easy if threads don’t interact
  - Each thread does its own thing, ignoring other threads
  - Typically, however, threads must communicate
- In Java, communication via *shared data*
  - Static and heap data can be accessed by all threads
  - Called *shared memory concurrency*
- Potential pitfalls
  - 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.

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

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;
}
```

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 cnt = 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;
}
```

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

But When it's Run Again?

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

T1 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;
}

T1 is preempted. 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;
}

T2 executes, storing the incremented cnt value.

Data Race Example

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)!

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!
  – 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

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

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

T1 reads cnt into y

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: `cnt = 1`

```
t1 runs, assigning to cnt
```

```
y = 0
```

```
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();
}
```

`cnt = 1`

```
y = 0
```

```
T1 reads cnt into y.
```

```
y = 1
```
Applying Synchronization

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();
}

Different Locks Don’t Interact

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

What’s Wrong with the Following?

• 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

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
Deadlock

- **Deadlock** occurs when no thread can run because all threads are waiting for a lock
  - No thread running, so no thread can ever release a lock to enable another thread to run
    ```java
    Lock l = new ReentrantLock();
    Lock m = new ReentrantLock();
    Thread 1
    l.lock();
    m.lock();
    ...
    m.unlock();
    l.unlock();
    
    Thread 2
    m.lock();
    l.lock();
    ...
    l.unlock();
    m.unlock();
    ```

Deadlock (cont’d)

- Some schedules work fine
  - Thread 1 runs to completion, then thread 2

- But what if...
  - Thread 1 acquires lock \( l \)
  - The scheduler switches to thread 2
  - Thread 2 acquires lock \( m \)

- Deadlock!
  - Thread 1 is trying to acquire \( m \)
  - Thread 2 is trying to acquire \( l \)
  - And neither can, because the other thread has it

Wait Graphs

- Thread T1 holds lock \( l \)
- Thread T2 attempting to acquire lock \( m \)

Deadlock occurs when there is a cycle in the graph

Wait Graph Example

- T1 holds lock on \( l \)
- T2 holds lock on \( m \)
- T1 is trying to acquire a lock on \( m \)
- T2 is trying to acquire a lock on \( l \)
Another Case of Deadlock

```java
static Lock l = new ReentrantLock();
void f () throws Exception {
    l.lock();
    FileInputStream f =
        new FileInputStream("file.txt");
    // Do something with f
    f.close();
    l.unlock();
}

• l not released if exception thrown
  – Likely to cause deadlock some time later
```

Solution: Use Finally

```java
static Lock l = new ReentrantLock();
void f () throws Exception {
    l.lock();
    try {
        FileInputStream f =
            new FileInputStream("file.txt");
        // Do something with f
        f.close();
    }
    finally {
        // This code executed no matter how we
        // exit the try block
        l.unlock();
    }
```

Synchronized

• This pattern is really common
  – Acquire lock, do something, release lock under any
    circumstances after we’re done
    • Even if exception was raised etc.

• Java has a language construct for this
  – synchronized (obj) { body }
    • Every Java object has an implicit associated lock
  – Obtains the lock associated with obj
  – Executes body
  – Release lock when scope is exited
    • Even in cases of exception or method return

Example

```java
static Object o = new Object();
void f() throws Exception {
    synchronized (o) {
        FileInputStream f =
            new FileInputStream("file.txt");
        // Do something with f
        f.close();
    }

    – Lock associated with o acquired before body
      executed
    • Released even if exception thrown
```
Discussion

• An object and its associated lock are different!
  – Holding the lock on an object does not affect what you can do with that object in any way
  – Ex:

    ```java
    synchronized(o) { ... }  // acquires lock named o
    o.f();                  // someone else can call o’s methods
    o.x = 3;                // someone else can read and write o’s fields
    ```

Example: Synchronizing on this

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
    void dec() {
        synchronized (this) {
            cnt--;
        }
    }
}
```

• Does this program have a data race?
  – No, both threads acquire locks on the same object before they access shared data

Example: Synchronizing on this (cont’d)

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
    void dec() {
        synchronized (this) {
            cnt--;
        }
    }
}
```

• Data race?
  – No, threads acquire locks on the same object before they access shared data

Example: Synchronizing on this (cont’d)

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
    void dec() {
        synchronized (this) {
            cnt--;
        }
    }
}
```

• Does this program have a data race?
  – No, threads acquire different locks, but they write to different objects, so that’s ok
Synchronized Methods

- Marking method as synchronized same as synchronizing on this in body of the method
  - The following two programs are the same

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
}
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
    }synchronized void dec() {
        cnt--;
    }
}
```

Synchronized Methods (cont’d)

- Data race?
  - No, both acquire same lock

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
}
class C {
    int cnt;
    synchronized void inc() {
        cnt++;
    }
    synchronized void dec() {
        cnt--;
    }
}
```

Synchronized Static Methods

- Warning: Static methods lock class object
  - There’s no `this` object to lock

```java
class C {
    static int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
    static synchronized void dec() {
        cnt--;
    }
}
```

Thread Scheduling

- When multiple threads share a CPU...
  - When should the current thread stop running?
  - What thread should run next?
- A thread can voluntarily `yield()` the CPU
  - Call to yield may be ignored; don’t depend on it
- **Preemptive schedulers** can switch threads at any time
  - Extremely common, but not guaranteed for Java
  - In theory, should always include `yield()` in loop
  - In practice, don’t bother
- Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep
Thread Lifecycle

- Java thread can be in one of these states
  - **New** — thread allocated and waiting for start()
  - **Runnable** — thread can begin execution
    - **Running** — thread currently executing
  - **Blocked** — waiting for event (I/O, unlock, etc)
  - **Dead** — completed
- Transitions between states caused by
  - Invoking methods in class Thread
    - `new()`, `start()`, `yield()`, `sleep()`, `wait()`, `notify()`, ...
  - Other (external) events
    - Scheduler, I/O, returning from `run()`, ...

Which Thread to Run Next?

- Look at all runnable threads
  - A good choice to run is one that just became unblocked because
    - A lock was released
    - I/O became available
    - It finished sleeping, etc.
- Pick a thread and start running it
  - Can try to influence this with `setPriority(int)`
  - Higher-priority threads get preference
  - But you probably don’t need to do this

Some Thread Methods

- `void join()` throws `InterruptedException`
  - Waits for a thread to die/finish
- `static void yield()`
  - Current thread gives up the CPU
- `static void sleep(long milliseconds)` throws `InterruptedException`
  - Current thread sleeps for the given time
- `static Thread currentThread()`
  - Get `Thread` object for currently executing thread
Example: Threaded, Sync Alarm

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

    // read user input
    String line = b.readLine();
    parseInput(line);

    // wait (in secs) asynchronously
    if (m != null) {
        // start alarm thread
        Thread t = new AlarmThread(m, tm);
        t.start();
        // wait for the thread to complete
        t.join();
    }
}
```

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

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
  - Need to prevent interference
    - Rule of thumb 1: You must hold a lock when accessing shared data
    - Rule of thumb 2: You must not release a lock until shared data is in a valid state
  - Overuse use of synchronization can create deadlock
    - Rule of thumb: No deadlock if only one lock

Producer/Consumer Design

- Suppose we are communicating with a shared variable
  - E.g., some kind of a buffer holding messages
- One thread produces input to the buffer
- One thread consumes data from the buffer
- Rules
  - Producer can't add input to the buffer if it's full
  - Consumer can't take input from the buffer if it's empty
**Producer / Consumer Idea**

- If buffer is partially full, producer or consumer can run

  ![Diagram](producer-consumer-partially-full.png)

- If buffer is empty, only producer can run

  ![Diagram](producer-consumer-empty.png)

- If buffer is full, only consumer can run

  ![Diagram](producer-consumer-full.png)

**Broken Producer/Consumer Example**

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(object o) {
    lock.lock();
    while (!valueReady);
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    Object o = value;
    valueReady = false;
    lock.unlock();
    return o;
}
```

Threads wait with lock held – no way to make progress

**Broken Producer/Consumer Example**

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(object o) {
    boolean done = false;
    while (!done) {
        lock.lock();
        if (!valueReady) {
            value = o;
            valueReady = true;
            lock.unlock();
        } else {
            return o;
        }
    }
}

Object consume() {
    boolean done = false;
    while (!done) {
        lock.lock();
        if (valueReady) {
            Object o = value;
            valueReady = false;
            done = true;
        } else {
            lock.unlock();
            return o;
        }
    }
}
```

valueReady accessed without a lock held – race condition

**Broken Producer/Consumer Example**

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(object o) {
    boolean done = false;
    while (!done) {
        lock.lock();
        if (!valueReady) {
            value = o;
            valueReady = true;
            lock.unlock();
        } else {
            return o;
        }
    }
}

Object consume() {
    boolean done = false;
    while (!done) {
        lock.lock();
        if (valueReady) {
            Object o = value;
            valueReady = false;
            done = true;
        } else {
            lock.unlock();
            return o;
        }
    }
}
```

Constantly acquiring / releasing lock — busy wait
Solving Producer / Consumer Problem

- Difficult to use locks directly
  - Very hard to get right
  - Problems often very subtle
- Proper approach – use Condition interface
  - Condition is created from Lock object
  - Allows threads to sleep while waiting to acquire lock
  - Can wake up sleeping threads before releasing lock

```java
interface Lock { Condition newCondition(); ... }
interface Condition {
    void await();
    void signalAll(); ... }
```

Condition (Java 1.5)

- Calling `await()` w/ lock held
  - Releases the lock
  - But not any other locks held by this thread
  - Adds this thread to wait set for condition
  - Blocks the thread
- Calling `signalAll()` w/ lock held
  - Resumes all threads in condition’s wait set
  - Threads must reacquire lock
    - Before continuing (returning from await)
    - Enforced automatically; you don’t have to do it

```
Producer/Consumer Solution
```

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;

void produce(Object o) {
    lock.lock();
    while (valueReady)
        ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    while (!valueReady)
        ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
    return o;
}
```

Await and SignalAll Gotcha’s

- `await` must be called in a loop
  - Conditions may not be met when await returns
  - Some other thread may have woken first
    - And changed condition (e.g., consumed item in buffer)

- Avoid holding other locks when waiting
  - `await` only gives up locks on the object you wait on
Broken Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;

void produce(object o) {
    lock.lock();
    if (valueReady) ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    if (!valueReady) ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
    return o;
}
```

what if there are multiple producers or consumers?

More on the Condition Interface

```java
interface Condition {
    void await();
    boolean await(long time, TimeUnit unit);
    void signal();
    void signalAll();
}
```

- `await(t, u)` waits for time `t` and then gives up
  - Result indicates whether woken by signal or timeout
- `signal()` wakes up only one waiting thread
  - Tricky to use correctly
    - Have all waiters be equal, handle exceptions correctly
  - Highly recommended to just use `signalAll()`

Blocking Queues in Java 1.5

- Interface for producer/consumer pattern

```java
interface Queue<E> extends Collection<E> {
    boolean offer(E x);  /* produce */
    /* waits for queue to have capacity */
    E remove();          /* consume */
    /* waits for queue to become non-empty */
    ... }
```

- Two handy implementations
  - `LinkedBlockingQueue` (FIFO, may be bounded)
  - `ArrayBlockingQueue` (FIFO, bounded)
  - (plus a couple more)

Wait and NotifyAll (Java 1.4)

- Recall that in Java 1.4, use `synchronize` on object to get associated lock

```
object o
```

- Objects also have an associated wait set
Wait and NotifyAll (cont’d)

• `o.wait()`
  – Must hold lock associated with `o`
  – Release that lock
    • And no other locks
  – Adds this thread to wait set for lock
  – Blocks the thread

• `o.notifyAll()`
  – Must hold lock associated with `o`
  – Resumes all threads on lock’s wait set
  – Those threads must reacquire lock before continuing
    • (This is part of the function; you don’t need to do it explicitly)

Producer/Consumer in Java 1.4

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

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

    synchronized Object consume() {
        while (!valueReady) wait();
        valueReady = false;
        Object o = value;
        notifyAll();
        return o;
    }
}
```

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
  – Flag can be tested by calling
    • `interrupted()` method
      – Clears the interrupt flag
    • `isInterrupted()` method
      – Does not clear the interrupt flag

• Won’t disturb the thread if it is working
  – Not asynchronous!
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;
                    else process(c);
                } catch (IOException ex) { break; }
            }
        } finally { // cleanup here }
    }
}
```

What if the thread is blocked on a lock or wait set, or sleeping when interrupted?

Responses to Interruption

- **Early Return**
  - Clean up and exit without producing errors
  - May require rollback or recovery
  - Callers can poll cancellation status to find out why an action was not carried out
- **Continuation (i.e., ignore interruption)**
  - When it is too dangerous to stop
  - When partial actions cannot be backed out
  - When it doesn’t matter

InterruptedException

- Exception thrown if interrupted on certain ops
  - wait, await, sleep, join, and lockInterruptibly
  - Also thrown if call one of these with interrupt flag set
- **Not thrown** when blocked on 1.4 lock or I/O

Responses to Interruption (cont’d)

- Re-throw `InterruptedException`
  - When callers must be alerted on method return
- Throw a general failure exception
  - When interruption is a reason method may fail
- In general
  - Must reset invariants before cancelling
  - E.g., close file descriptors, notify other waiters, etc.
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

Aspects of Synchronization

- Atomicity
  - Locking to obtain mutual exclusion
  - What we most often think about
- Visibility
  - Ensuring that changes to object fields made in one thread are seen in other threads
- Ordering
  - Ensuring that you aren’t surprised by the order in which statements are executed

Quiz

- Can this result in i=0 and j=0?
Doesn’t Seem Possible...

• But this can happen!

How Can This Happen?

• Compiler can reorder statements
  – Or keep values in registers
• Processor can reorder them
• On multi-processor, values not synchronized in global memory

When Are Actions Visible?

Must be the same lock

Forcing Visibility of Actions

• All writes from thread that holds lock M are visible to next thread that acquires lock M
  – Must be the same lock

• Use synchronization to enforce visibility and ordering
  – As well as mutual exclusion
Volatile Fields

- If you are going to access a shared field without using synchronization
  - It needs to be volatile
- If you don’t try to be too clever
  - Declaring it volatile just works
- Example uses
  - A one-writer/many-reader value
    - Simple control flags:
      - volatile boolean done = false;
    - Keeping track of a “recent value” of something

Misusing Volatile

- Incrementing a volatile field doesn’t work
  - In general, writes to a volatile field that depend on the previous value of that field don’t work
- A volatile reference to an object isn’t the same as having the fields of that object be volatile
  - No way to make elements of an array volatile
- Can’t keep two volatile fields in sync
- Don’t use for this course

Guidelines for Programming w/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