Threads and Synchronization
October 11th, 2007

(thanks to Doug Lea for some slides)
Overview

• What are threads?
  – Concept
  – Basic Java mechanisms

• Thread concerns
  – Safety and Liveness
  – Use of synchronization and signalling

• Threading design patterns
A computer

Computation Abstractions

Processes
(e.g., JVM’s)

Threads
Processes vs. Threads

Processes do not share data

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

Threads share data within a process

```c
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
Why Not Multiple Threads?

• Complexity:
  – **Dealing with safety, liveness, composition**

• Nondeterminism
  – Even with "correct" code, timing differences may result in different output each time it is run
  – Not necessarily bad, just makes it a pain to test
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 \texttt{main()} method
• The code executed by \texttt{main()} can create other threads
  – Explicitly, using the \texttt{Thread} class
  – Implicitly, by calling libraries that create threads as a consequence
    • RMI, AWT/Swing, Applets, etc.
Java Threads: Creation

• 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
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 \texttt{yield()} the CPU
  – Call to yield may be ignored; don’t depend on it
• \textit{Preemptive schedulers} can de-schedule the current thread at any time
  – Almost all JVMs use preemptive scheduling, but a few (embedded) JVM’s do not, so a thread stuck in a loop may \textit{never} yield by itself. Therefore, put \texttt{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
  - But don’t depend on priority for correctness
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
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
Concurrency Issues

• Threads allow concurrent activities, which can be both good and bad!
• Two opposing design forces
  – Safety: “Nothing bad ever happens”
  – Liveness: “Something (useful) eventually happens”
• A safe system may not be live and a live system may not be safe. Balance is key.
Safe Objects

• Perform actions only when in consistent states
  – Don’t want one thread to access an object while another thread is modifying its internal state.

• This boils down to ensuring object invariants in the face of concurrent access
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)
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;  // Shared state  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.*
Data Race Example

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

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

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

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

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

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

\( \text{T1 is pre-empted. T2 executes, grabbing the global counter value into y.} \)
static int cnt = 0;   // Shared state cnt = 1

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

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

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

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

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

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

T2 completes. T1 executes again, storing the old counter value (1) rather than the new one (2)!
But When I Run it 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

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

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

t2.run() {
    int y = cnt;  // T1 executes, grabbing
    cnt = y + 1; // the global counter value into y.
}
```
Data Race Example

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

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

t2.run() {
    int y = cnt;   // T1 executes again, storing the
    cnt = y + 1;  // counter value
}
```
Data Race Example

```java
static int cnt = 0;

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

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

*Shared state*  \( cnt = 1 \)

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

```
static int cnt = 0;  

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

Shared state  cnt = 2

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

T2 executes, storing the incremented cnt value.
```
What Happened?

- In the first example, \texttt{t1} was preempted after it read the counter but before it stored the new value.
  - Depends on the idea of an \textit{atomic action}
  - Violated an object invariant
- A particular way in which the execution of two threads is interleaved is called a \textit{schedule}. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.
• If instead of
  \[\text{int } y = \text{cnt};\]
  \[\text{cnt} = y+1;\]

• We had written
  \[-\text{cnt}++;\]

• Would the result be any different?

• Answer: NO!
  \[-\text{Don’t depend on your intuition about atomicity}\]
• 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

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

```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 = 0

T1 acquires the lock
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  \*\* T1 reads cnt into y \*\*
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

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

\begin{itemize}
\item \texttt{t1.run()} {
\begin{itemize}
\item \texttt{synchronized(lock) { \{ \}
\item \texttt{int y = cnt; \}
\item \texttt{cnt = y + 1; \}} y = 0
\end{itemize}
\end{itemize}

\item \texttt{t2.run()} {
\begin{itemize}
\item \texttt{synchronized(lock) { \{ \}
\item \texttt{int y = cnt; \}
\item \texttt{cnt = y + 1; \}}}
\end{itemize}
\end{itemize}

\textit{Shared state} \quad cnt = 0

\textit{T1 is pre-empted.} \quad \textit{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() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    } cnt = y + 1;
}
```

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

*T1 runs, assigning to \( \text{cnt} \)*
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* \( \text{cnt} = 1 \)

*\( T1 \) releases the lock and terminates*
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;
    }
}
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;  y = 1
    }
}
```

*Shared state*  \( cnt = 1 \)

*\( T2 \) reads \( cnt \) into \( y \).*
Applying Synchronization

```
int cnt = 0;

// T1 thread
void t1() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
        y = 0
    }
}

// T2 thread
void t2() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
        y = 1
    }
}
```

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

T2 assigns \( \text{cnt} \),
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
  - *in Java 1.4*
    - 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
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 for the class
    • Accessible directly, e.g. `Foo.class`
  – Not the same as `this`!
Synchronization Example

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

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, …
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

A

T1

T2

B

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
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 and arrays can be shared
  – Need to prevent interference
    • Synchronization is one way, but not the only way
  – Overuse use of synchronization can create deadlock
    • Violation of liveness
Guaranteeing Safety

• Ensure objects are accessible only when in a **consistent** and appropriate state
  – All invariants are maintained
  – Presents subclass obligations

• Use locks to enforce this
  – **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
Guaranteeing Liveness

• Ensuring availability of services
  – Called methods eventually execute

• Ensuring progress of activities
  – Managing resource contention
  – Freedom from deadlock
  – Fairness
  – Fault tolerance
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
- How do we implement this?
  - Use wait and notify
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;
        value = null; // why do we do this?
        notifyAll();
        return o;
    }
}
Wait and Notify

• Both must be called while lock is held on a
• a.wait()
  – Releases the lock on a
    • But not any other locks acquired by this thread
  – Adds the thread to the wait set for a
  – Blocks the thread
• a.wait(int m)
  – Limits wait time to m milliseconds
• `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 performs the reacquire automatically
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 shared 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();
}
```
Use This Design

• This is the right solution to the problem
  – Tempting to try to just use locks directly
  – Very hard to get right
  – Problems with other approaches often very subtle
    • E.g., double-checked locking is broken
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

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

    synchronized Object consume() {
        while (!valueReady) {};
        valueReady = false;
        Object o = value;
        value = null;
        return o;
    }
}
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

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

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

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

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

    synchronized Object consume() throws InterruptedException {
        if (!valueReady) wait();
        valueReady = false;
        Object o = value;
        value = null;
        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 one thread
    • Doesn’t matter which thread it is
  – Handle exceptions correctly
    • Including InterruptedException
• For this course, just use notifyAll()
Wait and Notify Gotcha’s

• *wait* *must* be in a loop
  – Don’t assume that when *wait* returns conditions are met

• Avoid holding other locks when waiting
  – *Wait* only gives up locks on the object you *wait* on
Preventing Data Races

• One programming technique to prevent races:
  – Ensure that for every shared field x, there is some lock l such that no thread accesses x without holding lock l

• Note: This is not the only way to avoid races
  – There are fancier, more complicated techniques
  – But this is what you should do for your project
Detecting Races with checkSync

• Algorithm
  – Locks_held(t) = set of locks held by thread t
  – For each shared field x, C(x) := { all locks }  
  – On each access to x by thread t,
    • C(x) := C(x) ∩ locks_held(t)
    • If C(x) = ∅ then issue a warning

An Improvement

- Unsynchronized reads of a shared location are OK
  - As long as no one writes to the field after it becomes shared
- Track state of each field
  - Only enforce locking protocol when it becomes shared and written
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
• 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?

Thread 1

\[ x = 1 \]

unlock M

Thread 2

lock M

\[ i = x \]

Must be the \textit{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
- Semantics for volatile have been strengthened in JSR-133, Java 5
  - Many 1.4 VM’s compliant
- If you don’t try to be too clever
  - Declaring it volatile just works
Using Volatile

• 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
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.destroy()
  – 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!
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
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
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.
synchronized (this) {
    while (!ready) {
        try {
            wait();
        } catch (InterruptedException e) {
            // make shared state acceptable
            notifyAll();
            return; // quit processing
        }
        // do whatever
    }
}
Why No Thread.destroy()?

• 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
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
• Look at the methods that might be invoked while a lock is held
• Are any of them methods that might:
  – obtain a lock
  – perform I/O
  – block waiting for another thread to do something
• For example, invoking .hashCode() on an Object passed to your method is an open call
  – the hashCode method could do anything
• Open calls can't always be avoided, just worry about them