CSE 305 Introduction to Programming Languages
Lecture 19 – Concurrency

CSE @ SUNY-Buffalo
Zhi Yang
Courtesy of Professor Vitaly Shmatikov
Courtesy of Professor Michael Hicks and Dave Levin
Notice Board

• Next class, August 6(Tuesday), 2013, we will be having practice-final-exam.
Our objective

• The first objective of our class, is to comprehend a new programming language within very short time period, and because you have this ability to shorten your learning curve, you are going to manipulate the language with an insight learning.

• The second objective is to even engineer your own language!
Review what we’ve learnt and see future

Number System

$1^{st}$ Generation language: Machine Code

Basic Calculation System

What’s next?

$2^{nd}$ Generation language: Assembly Code

Regular Expression

Type Checking

$3^{rd}$ Generation Language: Macro function

 Compiler System

Virtual Machine

Macros function

Push Down Automata

Context-Free Grammar

Lambda Calculus Theory
A family tree of languages

- Fortran
- BASIC
- Cobol
- LISP
- Scheme
- ML
- Prolog
- PL/1
- Algol 60
- Algol 68
- Pascal
- Simula
- C
- Modula 3
- Ada
- Dylan
- Ruby
- Perl
- Java
- C++
- C#
- Python
- PHP
- JavaScript
- PHP
Multiprocessing

- **Multiprocessing**: The use of multiple parallel computations

- We have entered an era of multiple cores...
  - Hyperthreading
  - Dual-core and quad-core consumer processors
  - Symmetric Multi-Processing (SMP) machines

- ...and multiple nodes
  - Computational clusters
  - Grid and cloud computing
Technologies

• Multiple cores:
  – SIMD arithmetic and GPUs
  – Java and Ruby threads
  – POSIX threads
  – OpenMP

• Multiple nodes:
  – MPI
  – MapReduce
Amdahl’s Law

- *Informally*: The theoretical maximum speedup using multiprocessing is limited by a program’s sequential performance.

\[
\frac{1}{(1 - P) + \frac{P}{S}}
\]

- Sequential portion
- Parallel portion
Computations Abstractions

A computer

Processes

Threads
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
Thread Creation

execution (time)

main thread

thread starts

thread starts

thread ends

thread join
• Per-thread stack and instruction pointer
  – Saved in memory when thread suspended
  – Put in hardware esp/eip when thread resumes
Programming Threads

• Threads are available in most 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

• Thread and Monitor modules in Ruby
Example: Rendering a Web page

- Page is a shared resource
- Multiple concurrent activities in the Web browser
  - Thread for each image load
  - Thread for text rendering
  - Thread for user input (e.g., “Stop” button)
- Cannot all write to page simultaneously!
  - Big challenge in concurrent programming: managing access to shared resources
Outline/Challenge of concurrency

– Communication – send or receive information
– Synchronization – wait for another process to act
– Atomicity – do not step in the middle and leave a mess
Inter-Process Communication

• Processes may need to communicate
  – Process requires exclusive access to some resources
  – Process need to exchange data with another process

• Can communicate via:
  – Shared variables
  – Message passing
  – Parameters
Problem 1: cobegin / coend

- Limited concurrency primitive
  - Concurrent Pascal [Per Brinch Hansen, 1970s]

\[
x := 0;
\]
\[
cobegin
  begin x := 1; x := x+1 end;
  begin x := 2; x := x+1 end;
\]
\[
coend;
\]
\[
print(x);
\]

Atomicity at level of assignment statement
Properties of cobegin/coend

- Simple way to create concurrent processes
- Communication by shared variables
- No mutual exclusion
- No atomicity
- Number of processes fixed by program structure
- Cannot abort processes
  - All must complete before parent process can go on
Race Conditions

• Race condition occurs when the value of a variable depends on the execution order of two or more concurrent processes (why this is bad?)

• Example

  procedure signup(person)
  begin
    number := number + 1;
    list[number] := person;
  end;
  signup(joe) || signup(bill)
Solve Problem 1: Critical Section

- Two concurrent processes may access a shared resource
- Inconsistent behavior if processes are interleaved
- Allow only one process in **critical section**
- Issues
  - How to select which process is allowed to access the critical section?
  - What happens to the other process?
Locks and Waiting

<initialize concurrency control>

Process 1:

  <wait>
  signup(joe);  // critical section
  <signal>

Process 2:

  <wait>
  signup(bill);  // critical section
  <signal>

Need atomic operations to implement wait
Solution of Problem 1 gives rise to Problem 2: Deadlock

- **Deadlock** occurs when a process is waiting for an event that will never happen.
- Necessary conditions for a deadlock to exist:
  - Processes claim exclusive access to resources.
  - Processes hold some resources while waiting for others.
  - Resources may not be removed from waiting processes.
  - There exists a circular chain of processes in which each process holds a resource needed by the next process in the chain.
- Example: “dining philosophers”
Implementing Mutual Exclusion

• Semaphore
  – Keep queue of waiting processes
    • Avoid busy-waiting loop
  – Scheduler has access to semaphore; process sleeps
  – Disable interrupts during semaphore operations
    • OK since operations are short
Semaphores

- **Semaphore** is an integer variable and an associated process queue

- **Operations:**
  - $P(s)$ if $s > 0$ then $s--$
    
    else enqueue process
  - $V(s)$ if a process is enqueued then dequeue it
    
    else $s++$

- **Binary semaphore**
- **Counting semaphore**
Example of Semaphore: Simple Producer-Consumer

```
program SimpleProducerConsumer;
var buffer : string;
  full : semaphore = 0;
  empty : semaphore = 1;
begin
  cobegin
    Producer; Consumer;
  coend;
end.

procedure Producer;
var tmp : string
begin
  while (true) do begin
    produce(tmp);
    P(empty);  { begin critical section }
    buffer := tmp;
    V(full);   { end critical section }
  end;
end;

procedure Consumer;
var tmp : string
begin
  while (true) do begin
    P(full);   { begin critical section }
    tmp := buffer;
    V(empty);  { end critical section }
    consume(tmp);
  end;
end;
```
**Producer-Consumer**

program ProducerConsumer;
const size = 5;
var buffer : array[1..size] of string;
  inn  : integer = 0;
  out  : integer = 0;
  lock : semaphore = 1;
  nonfull : semaphore = size;
  nonempty : semaphore = 0;

procedure Producer;
  var tmp : string
  begin
    while (true) do begin
      produce(tmp);
      P(nonfull);
      P(lock);  { begin critical section }
      inn := inn mod size + 1;
      buffer[inn] := tmp;
      V(lock);  { end critical section }
      V(nonempty);
    end;
  end;
end;

procedure Consumer;
  var tmp : string
  begin
    while (true) do begin
      P(nonempty);
      P(lock);  { begin critical section }
      out = out mod size + 1;
      tmp := buffer[out];
      V(lock);  { end critical section }
      consume(tmp);
    end;
  end;
end;
Monitors

- **Monitor** encapsulates a shared resource (monitor = “synchronized object”)
  - Private data
  - Set of access procedures (methods)
  - Locking is automatic
    - At most one process may execute a monitor procedure at a time (this process is “in” the monitor)
    - If one process is in the monitor, any other process that calls a monitor procedure will be delayed
Example of a Monitor

```pascal
monitor Buffer;
const size = 5;
var buffer : array[1..size] of string;
in : integer = 0;
out : integer = 0;
count : integer = 0;
nonfull : condition;
nonempty : condition; ...

procedure put(s : string);
begin
  if (count = size) then wait(nonfull);
in := in mod size + 1;
buffer[in] := tmp;
count := count + 1;
signal(nonempty);
end;

function get : string;
begin
var tmp : string

  if (count = 0) then wait(nonempty);
out = out mod size + 1;
tmp := buffer[out];
count := count - 1;
signal(nonfull);
get := tmp;
end;
```
Examples of concurrency

• We will be using 3 examples to demonstrate the use of concurrency:

1) Java
2) Ruby
3) C#
Example 1: Java Threads

- **Thread**
  - Set of instructions to be executed one at a time, in a specified order
  - Special Thread class is part of the core language
    - In C/C++, threads are part of an “add-on” library
- **Methods of class Thread**
  - **start**: method called to spawn a new thread
    - Causes JVM to call run() method on object
  - **suspend**: freeze execution (requires context switch)
  - **interrupt**: freeze and throw exception to thread
  - **stop**: forcibly cause thread to halt
Java.lang.Thread

```java
public class Thread implements Runnable {
    private char name[];
    private Runnable target;
    ...
    public final static int MIN_PRIORITY = 1;
    public final static int NORM_PRIORITY = 5;
    public final static int MAX_PRIORITY = 10;
    ...
    private void init(ThreadGroup g, Runnable target, String name) {...}
    public Thread() { init(null, null, "Thread-" + nextThreadNum()); }
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum());
    }
    public Thread(Runnable target, String name) { init(null, target, name); }
    
    public synchronized native void start();
    public void run() {
        if (target != null) {
            target.run();
        }
    }
}
```

What does this mean?

Create Native Code

Creates execution environment for the thread (sets up a separate run-time stack, etc.)
Methods of Thread Class

```java
public class Thread implements Runnable {
    ...
    public static native Thread currentThread();
    public static native void yield();
    public static native void sleep(long millis) throws InterruptedException;
    public static int enumerate(Thread tarray[])

    public static boolean interrupted() { ... }
    public boolean isInterrupted() { ... }
    public final native boolean isAlive();
    public String toString() {
        public void interrupt() { ... }
    public void interrupt() { ... }
    public final void stop() { ... }
    public final void suspend() { ... }
    public final void resume() { ... }
    public final void setPriority(int newPriority) {
    public final int getPriority() {
    public final void setName(String name) { ... }
    public final String getName() { return String.valueOf(name); }
    public native int countStackFrames();
    public final synchronized void join() throws InterruptedException {...}
    public void destroy() { throw new NoSuchMethodError(); }
}
```
Runnable Interface

- Thread class implements Runnable interface
- Single abstract (pure virtual) method `run()`
  ```java
  public interface Runnable {
      public void run();
  }
  ```
- Any implementation of Runnable must provide an implementation of the `run()` method
  ```java
  public class ConcurrentReader implements Runnable {
      ...
      public void run() {
          ...
          ... code here executes concurrently with caller ...
      }
  }
  ```
Two Ways to Start a Thread

• Construct a thread with a runnable object
  ConcurrReader readerThread = new ConcurrReader();
  Thread t = new Thread(readerThread);
  t.start();  // calls ConcurrReader.run() automatically

... OR ...

• Instantiate a subclass of Thread
  class ConcurrWriter extends Thread {  ...
      public void run() {  ...  }  }
  ConcurrWriter writerThread = new ConcurrWriter();
  writerThread.start();  // calls ConcurrWriter.run()
Why Two Ways?

- Java only has single inheritance
- Can inherit from some class, but also implement Runnable interface so that can run as a thread

```java
class X extends Y implements Runnable {
    public synchronized void doSomething() {
    }
    public void run() {
        doSomething();
    }
}
X obj = new X();
obj.doSomething(); // runs sequentially in current thread
Thread t = new Thread(new X()); // new thread
t.start(); // calls run() which calls doSomething()
```
Interesting “Feature”

[Allen Holub, “Taming Java Threads”]

- Java language specification allows access to objects that have not been fully constructed

```java
class Broken {
    private long x;
    Broken() {
        new Thread() {
            public void run() { x = -1; }
        }.start();
        x = 0;
    }
}
```

Thread created within constructor can access partial object
Interaction Between Threads

- Shared variables and method calls
  - Two threads may assign/read the same variable
    - Programmer is responsible for avoiding race conditions by explicit synchronization!
  - Two threads may call methods on the same object

- Synchronization primitives
  - All objects have an internal lock (inherited from Object)
  - Synchronized method locks the object
    - While it is active, no other thread can execute inside object
  - Synchronization operations (inherited from Object)
    - Wait: pause current thread until another thread calls Notify
    - Notify: wake up waiting thread
Synchronized Methods

- Provide mutual exclusion
  - If a thread calls a synchronized method, object is locked
  - If another thread calls a synchronized method on the same object, this thread blocks until object is unlocked
- Unsynchronized methods can still be called!
  - “synchronized” is not part of method signature
    - Subclass may replace a synchronized method with unsynchronized method
public class Object {

    public final native void notify();
    public final native void notifyAll();

    public final native void wait(long timeout) throws InterruptedException;
    public final void wait() throws InterruptedException { wait(0); }
    public final void wait(long timeout, int nanos)
    throws InterruptedException { ... }

    • **wait()** releases object lock, thread waits on the object’s internal queue
    • **notify()** wakes the highest-priority thread closest to the front of the object’s internal queue
    • **notifyAll()** wakes up all waiting threads
      – Threads non-deterministically compete for access to object
      – May not be fair (low-priority threads may never get access)
    • May only be called when object is locked (when is that?)
Using Synchronisation

```java
public synchronized void consume() {
    while (!consumable()) {
        wait();  // release lock and wait for resource
    }
    // have exclusive access to resource, can consume
}

public synchronized void produce() {
    // do something that makes consumable() true
    notifyAll();  // tell all waiting threads to try consuming
    // can also call notify() and notify one thread at a time
}
Example: Shared Queue

class SharedQueue {
    private Element head, tail;

    public boolean empty() { return head == tail; }

    public synchronized Element remove() {
        try {
            while (empty()) wait();
        } // wait for an element in the queue
        catch (InterruptedException e) { return null; }
        Element p = head; head = head.next;
        if (head == null) tail = null;
        return p;
    }

    public synchronized void insert(Element p)
        if (tail == null) head = p;
        else tail.next = p;
        p.next = null;
        tail = p;
        notify(); // let one waiter know something is in the queue
    }
}
Example: Producer-Consumer

- Method call is synchronous
- How do we do this in Java?
A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

1. PRODUCER

2. CONSUMER

3. **PROBLEM**
   - Consumer takes from buffer before producer is done adding to it - trouble!
   - This is solved by

4. **BUFFER**
   - One-way buffer
   - The consumer must wait for producer to produce before it can consume...

5. **BUFFER**
   - Token buffer
   - Infinite # of mugs
   - Fixed # of mugs
   - If the consumer is busy (can't consume), the producer must wait, if the buffer is full; for the consumer to start consuming again. The processes are now
Solving Producer-Consumer

- **Cannot be solved with locks alone**
- Consumer must wait until buffer is not empty
  - While waiting, must sleep (use wait method)
  - Need condition recheck loop
- Producer must inform waiting consumers when there is something in the buffer
  - Must wake up at least one consumer (use notify method)
Implementation in Stack<T>

```java
public synchronized void produce (T object) {
    stack.add(object); notify();
}

public synchronized T consume () {
    while (stack.isEmpty()) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    int lastElement = stack.size() - 1;
    T object = stack.get(lastElement);
    stack.remove(lastElement);
    return object; }
```

Why is loop needed here?
Condition Rechecks

- Want to wait until condition is true
  ```java
  public synchronized void lock() throws InterruptedException {
    if ( isLocked ) wait();
    isLocked = true; }
  
  public synchronized void unLock() {
    isLocked = false;
    notify(); }
  ```

- Need a loop because another process may run instead
  ```java
  public synchronized void lock() throws InterruptedException {
    while ( isLocked ) wait();
    isLocked = true; }
  ```
Nested Monitor Lockout Problem

• Wait and notify used within synchronized code
  – Purpose: make sure that no other thread has called method of same object
• Wait causes the thread to give up its lock and sleep until notified
  – Allow another thread to obtain lock and continue processing
• Calling a blocking method within a synchronized method can lead to deadlock
Nested Monitor Lockout Example

class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
        synchronized(list) {
            list.addLast( x ); notify();
        }
    }
    public synchronized Object pop() {
        synchronized(list) {
            if( list.size() <= 0 ) wait();
            return list.removeLast();
        }
    }
}

Releases lock on Stack object but not lock on list; a push from another thread will deadlock
Preventing Nested Monitor Deadlock

• No blocking calls in synchronized methods, OR
• Provide some nonsynchronized method of the blocking object

• No simple solution that works for all programming situations
Synchronized Blocks

• Any Java block can be synchronized
  
  ```java
  synchronized(obj) {
    ... mutual exclusion on obj holds inside this block ...
  }
  ```

• Synchronized method declaration is just syntactic sugar for synchronizing the method’s scope
  
  ```java
  public synchronized void consume() {
    ... body ...
  }
  ```
  
is the same as
  
  ```java
  public void consume() {
    synchronized(this) {
      ... body ...
    }
  }
  ```
Locks Are Recursive

- A thread can request to lock an object it has already locked without causing deadlock

```java
public class Foo {
    public void synchronized f() { ... }
    public void synchronized g() { ... f(); ... }
}
```

```java
Foo f = new Foo;
synchronized(f) { ... synchronized(f) { ... } ... }
```
Synchronizing with Join()

• Join() waits for thread to terminate
  
  ```java
class Future extends Thread {
    private int result;
    public void run() { result = f(...); }
    public int getResult() { return result; }
  }
  ...

  Future t = new future;
  t.start();  // start new thread
  ...
  t.join();  // wait and get result
  x = t.getResult();
  ```
States of a Java Thread

- Non-existing
  - create thread object
- New
  - start
  - destroy
- Runnable
  - run method exits
  - notify, notifyAll
  - thread termination
- Blocked
  - wait, join
- Terminated (Dead)
  - garbage collected and finalization
  - destroy
- Non-Existing

Thread may not execute any “finally” clauses, leave objects locked.
Concurrent Garbage Collection

• Need to stop thread while mark-and-sweeping
  – Do other threads need to be stopped?
• Problem: objects may change during collection
• Solution: prevent read/write to memory area
  – Subtle!
  – Generational GC distinguishes short-lived and long-lived objects
  – Copying collectors allows reads from old area if writes are blocked...
POSIX Threads

- Pthreads library for C

`pthread_create` - create a new thread giving it a “starting” procedure to run along with a single argument.
`pthread_self` - ask the currently running thread for its thread id.
`pthread_join` - join with a thread using its thread id (an integer value)

`pthread_mutex_init` - initialize a mutex structure
`pthread_mutex_destroy` - destroy a mutex structure
`pthread_mutex_lock` - lock an initialized mutex, if already locked suspend execution and wait
`pthread_mutex_trylock` - try to lock a mutex and if unsuccessful, do not suspend execution
`pthread_mutex_unlock` - unlock a mutex that was locked by the current thread

`pthread_cond_init` - initialize a condition variable structure
`pthread_cond_destroy` - destroy a condition variable structure
`pthread_cond_wait` - block the currently running thread on a condition variable indefinitely
`pthread_cond_timedwait` - block the currently running thread on a condition variable for a specific time
`pthread_cond_signal` - wake up one thread blocked on a condition variable
`pthread_cond_broadcast` - wake up all threads blocked on a condition variable
Example of Using POSIX Threads

```c
#include <pthread.h>
#include <unistd.h> /* sleep declaration */
#include <stdio.h>  /* printf declaration */
const int NUM_THREADS = 5;

void* sleeping(void* st) {
    int sleep_time = (int) st; /* cast void* to an int */
    printf("thread \%d sleeping \%d seconds ...\n", pthread_self(), sleep_time);
    sleep(sleep_time);
    printf("\nthread \%d awakening\n", pthread_self());
}

main( int argc, char *argv[] ) {
    pthread_t tid[NUM_THREADS]; /* array of thread IDs */
    int i;

    for ( i = 0; i < NUM_THREADS; i++ )
        pthread_create (&tid[i], NULL, sleeping, i+2);

    for ( i = 0; i < NUM_THREADS; i++ )
        pthread_join (tid[i], NULL);

    printf("main() reporting that all \%d threads have terminated\n", i);
} /* main */
```
Thread Stacks

Main thread and run-time stack

```c
main() {
    for(i=0; i<n; i++)
        pthread_create(.....)
    ...
}
```

Multiple thread run-time stacks, each a separate “thread of execution”

- **Thread 1**
  ```c
  run()
  f()
  ...
  ```

- **Thread 2**
  ```c
  run()
  g()
  ...
  ```

- **Thread n**
  ```c
  run()
  h()
  ...
  ```

Stack Activation Frames

1MB Thread Stack

- Underflow stack frame raises exception if “popped”
- Starting procedure activation frame
- Nested procedure call activation frames
- "Red-lined" overflow page to generate memory segmentation violation (SEGV)
  if an overflow occurs from trying to create a frame beyond the end of the stack
Java-Style Synchronization in C++

class Synchronized {
    pthread_mutex_t m;  // mutex variable
    pthread_cond_t c;   // condition variable

protected:

    /* use this class to associate the mutex lock/unlock with the scope of a procedure */
    class Scope {
        Synchronized* obj;

    public:
        Scope(Synchronized* s) : obj(s) { pthread_mutex_lock(&obj->m); }
        ~Scope() { pthread_mutex_unlock(&obj->m); }
    };

public:

    Synchronized() { // initialize the mutex and condvar on construction
        pthread_mutex_init(&m, 0);
        pthread_cond_init(&c, 0);
    }

    ~Synchronized() { // destroy the mutex and condvar on destruction
        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&c);
    }

    // map Java-like wait, notify and notifyAll onto pthread equivalents

    void wait() { pthread_cond_wait(&c, &m); }
    void notify() { pthread_cond_signal(&c); }
    void notifyAll() { pthread_cond_broadcast(&c); }
};
Using C++ Threads

class MySynchronizedClass : public Synchronized {
   .. // private instance variables
public:
   
   // when this classes constructor is called, it first invokes the 
   // constructor of the Synchronized class, which initialized the 
   // the mutex and condition variable by calling the corresponding 
   // pthread_{mutex,cond}_init library procedures
   MySynchronizedClass() { ... }

   // Likewise on destruction, the destructor of the Synchronized class is 
   // automatically called and it destroys the mutex and condition variable 
   ~MySynchronizedClass() { ... }

   // to make a method “synchronized” we declare a local variable of type 
   // Synchronized::Scope, which locks the mutex on entry to the procedure scope 
   // and automatically unlocks the mutex on exit from the procedure scope

   int some_method(...) 
   {
      Synchronized::Scope mx(this); // automatically locks the mutex
      ... // execute code under mutual exclusion
   } // mx is automatically destructed, which unlocks the mutex
   ... 
};
Thread Safety of Classes

- Fields of an object or class must always be in a valid state, even when used concurrently by multiple threads
  - What’s a “valid state”? Serializability ...

- Classes are designed so that each method preserves state invariants on entry and exit
  - Example: priority queues represented as sorted lists
  - If invariant fails in the middle of a method call, concurrent execution of another method call will observe an inconsistent state
public class RGBColor {
    private int r; private int g; private int b;
    public RGBColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r; this.g = g; this.b = b;
    }
    private static void checkRGBVals(int r, int g, int b) {
        if (r < 0 || r > 255 || g < 0 || g > 255 ||
            b < 0 || b > 255) {
            throw new IllegalArgumentException();
        }
    }
    public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r; this.g = g; this.b = b;
    }
    public int[] getColor() {
        // returns array of three ints: R, G, B
        int[] retVal = new int[3];
        retVal[0] = r;
        retVal[1] = g;
        retVal[2] = b;
        return retVal;
    }
    public void invert() {
        r = 255 - r; g = 255 - g; b = 255 - b;
    }
}

What goes wrong with multi-threaded use of this class?
Problems with RGBColor Class

- **Write/write conflicts**
  - If two threads try to write different colors, result may be a “mix” of R,G,B from two different colors

- **Read/write conflicts**
  - If one thread reads while another writes, the color that is read may not match the color before or after
Making Classes Thread-Safe

• Synchronize critical sections
  – Make fields private, synchronize access to them
• Make objects immutable
  – State cannot be changed after object is created
    
```java
public RGBColor invert() {
    RGBColor retVal = new RGBColor(255 - r, 255 - g, 255 - b);
    return retVal;
}
```
  – Examples: Java String and primitive type wrappers Integer, Long, Float, etc.
  – Pure functions are always re-entrant!
• Use a thread-safe wrapper
Thread-Safe Wrapper

- Define new class which has objects of original class as fields, provides synchronized methods to access them

```java
public synchronized void setColor(int r, int g, int b) {
    color.setColor(r, g, b);
}

public synchronized int[] getColor() {
    return color.getColor();
}

public synchronized void invert() {
    color.invert();
}
```
public class Buffer {
    protected Object[] buf;  
    protected int MAX;    protected int current = 0;
    Buffer(int max) {
        MAX = max;
        buf = new Object[MAX]; }
    public synchronized Object get()  throws Exception {
        while (current<=0) { wait(); }
        current--;
        Object ret = buf[current];
        notifyAll();
        return ret; }
    public synchronized void put(Object v) throws Exception {
        while (current>=MAX) { wait(); }
        buf[current] = v;
        current++;
        notifyAll(); } }
Problems in Derived Class

```java
public class HistoryBuffer extends Buffer {
    boolean afterGet = false;
    public HistoryBuffer(int max) { super(max); }

    public synchronized Object gget() throws Exception {
        while ((current<=0)||(!afterGet)) { wait(); }
        afterGet = false;
        return super.get(); }

    public synchronized Object get() throws Exception {
        Object o = super.get();
        afterGet = true;
        return o; }

    public synchronized void put(Object v) throws Exception {
        super.put(v);
        afterGet = false; }
}
```

- New method, can be called only after get
- Must be redefined to keep track of last method called
- Need to redefine to keep track of last method called
util.concurrent

• Doug Lea’s utility classes
  – A few general-purpose interfaces
  – Implementations tested over several years
• Principal interfaces and implementations
  – Sync: acquire/release protocols
  – Channel: put/take protocols
  – Executor: executing Runnable tasks
Sync

• Main interface for acquire/release protocols
  – Used for custom locks, resource management, other common synchronization idioms
  – Coarse-grained interface, doesn’t distinguish different lock semantics

• Implementations
  – Mutex, ReentrantLock, Latch, CountDown, Semaphore, WaiterPreferenceSemaphore, FIFOSemaphore, PrioritySemaphore
  – ObservableSync, LayeredSync to simplify composition and instrumentation
Channel

- Main interface for buffers, queues, etc.

- Implementations
  - LinkedQueue, BoundedLinkedQueue, BoundedBuffer, BoundedPriorityQueue, SynchronousChannel, Slot
Executor

- Main interface for Thread-like classes
  - Pools
  - Lightweight execution frameworks
  - Custom scheduling
- Need only support `execute(Runnable r)`
  - Analogous to `Thread.start`
- Implementations
  - `PooledExecutor`, `ThreadedExecutor`, `QueuedExecutor`, `FJTaskRunnerGroup`
  - Related `ThreadFactory` class allows most Executors to use threads with custom attributes
java.util.Collection

- Adapter-based scheme
  - Allow layered synchronization of collection classes
- Basic collection classes are unsynchronized
  - Example: java.util.ArrayList
  - Except for Vector and Hashtable
- Anonymous synchronized Adapter classes
  - Constructed around the basic classes, e.g.,
    `List l = Collections.synchronizedList(new ArrayList());`
Java Memory Model

- Multithreaded access to shared memory
  - Competitive threads access shared data
  - Can lead to data corruption
- Memory model determines:
  - Which program transformations are allowed
    - Should not be too restrictive
  - Which program outputs may occur on correct implementation
    - Should not be too generous
  - Need semantics for incorrectly synchronized programs
Old memory model placed complex constraints on read, load, store, etc.
Program and Locking Order

Thread 1

1. \( y = 1 \)
2. lock M
3. \( x = 1 \)
4. unlock M

Thread 2

1. lock M
2. \( i = x \)
3. unlock M
4. \( j = y \)

Program order

Lock sync

[Manson, Pugh]
Example Program with Data Race

[Manson, Pugh]

Can we end up with $i = 0$ and $j = 0$?
Sequential Reordering + Data Race

[start threads]

Can we end up with $i = 0$ and $j = 0$? Yes!

Java definition considers this OK since there is a data race

[Thread 1]
- $x = y = 0$
- $x = 1$
- $j = y$

[Thread 2]
- $y = 1$
- $i = x$

OK to reorder single thread

[Manson, Pugh]
Want To Prevent This

- Must not result in $r_1 = r_2 = 42$
  - Imagine if 42 were a reference to an object!
- Value appears “out of thin air”
  - Causality run amok
  - Legal under a simple “happens-before” model of possible behaviors

\[
x = y = 0
\]

\[
r_1 = x
\]

\[
y = r_1
\]

\[
r_2 = y
\]

\[
x = r_2
\]

[Manson, Pugh]
Concurrent operations
  - read: no problem
  - read/write: OK if different lists
  - read/write to same list: clever tricks sometimes avoid locking
ConcurrentHashMap Tricks

- List cells immutable, except for data field
  - Read thread sees a linked list, even if concurrent write in progress
- Add to list by inserting at the head
- Remove from list: set data field to null, rebuild list to skip this cell
  - Unreachable cells eventually garbage collected
Limitations of Race-Freedom

(1)

class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
        }
        synchronized (this) {
            i = t+1;
        }
    }
    ...
}

Ref.inc()
- Race-free
- Behaves \textit{incorrectly} in a multithreaded context

Race freedom \textbf{does not} prevent errors due to unexpected interactions between threads

[Flanagan]
Limitations of Race-Freedom

(2)

```java
class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t+1;
        }
    }
    void read() { return i; }
    ...
}
```

Ref.read()
- Has a race condition
- Behaves **correctly** in a multithreaded context

Race freedom is not necessary to prevent errors due to unexpected interactions between threads
Atomicity

An easier-to-use and harder-to-implement primitive:

```java
void deposit(int x) {
    synchronized(this) {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

```java
void deposit(int x) {
    atomic {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

semantics:
lock acquire/release

semantics:
(behaves as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)
Second Example: Ruby

Multi-paradigm: object-oriented, imperative, functional, reflective

Ruby is a dynamic, reflective, general-purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features. It was also influenced by Eiffel and Lisp. Ruby was first designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

The standard and already retired 1.8.7 implementation was written in C, as a single-pass interpreted language. Starting with the 1.9 branch, and continuing with the current 2.0 branch, YARV has been used, and will eventually supersede the slower Ruby MRI. The language specifications for Ruby were developed by the Open Standards Promotion Center of the Information-Technology Promotion Agency (a Japanese government agency) for submission to the Japanese Industrial Standards Committee and then to the International Organization for Standardization. It was accepted as a Japanese Industrial Standard (JIS X 3017) in 2011 and an international standard (ISO/IEC 30170) in 2012. As of 2010, there are a number of complete or upcoming alternative implementations of Ruby, including YARV, JRuby, Rubinius, IronRuby, MacRuby (and its IOS counterpart, RubyMotion), mruby, HotRuby, Topaz counterpart, RubyMotion), mruby, HotRuby, Topaz and Opal. Each takes a different approach, with IronRuby, JRuby, MacRuby and Rubinius providing just-in-time compilation and MacRuby and mruby also providing ahead-of-time compilation.
Ruby Threads

• Create thread using `Thread.new`
  – `New` method takes code block argument
    
    ```ruby
    t = Thread.new { ...body of thread... }
    t = Thread.new (arg) { | arg | ...body of thread... }
    ```
  – `Join` method waits for thread to complete
    ```ruby
t.join
    ```

• Example:
  ```ruby
  myThread = Thread.new {
    sleep 1    # sleep for 1 second
    puts "New thread awake!"
    $stdout.flush # flush makes sure output is seen
  }
  ```
Thread Lifecycle

• While a thread executes, it goes through a number of different phases
  – **New**: created but not yet started
  – **Runnable**: can run on a free CPU
  – **Running**: currently executing on a CPU
  – **Blocked**: waiting for I/O or on a lock
  – **Sleeping**: paused for a user-specified interval
  – **Terminated**: completed
Thread Lifecycle

New

Runnable

Running

Blocked

Sleeping

Terminated
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 (we’ll see this in a minute)
    • I/O became available
    • It finished sleeping, etc.

• Pick a thread and start running it
  – Higher-priority threads get preference
  – Handled by the system or VM scheduler
Scheduling

One process per CPU

CPU 1
- p1
- p2

CPU 2
- p1
- p2

p2 threads: [Diagram]

p1 threads: [Diagram]
Scheduling

Threads shared between CPUs
Problem: Data Race

• Multiple processes may attempt to modify the same value at the same time, and their edits may conflict

• Concept: *Atomicity*
  – Atomic operations appear to happen instantaneously
  – Guaranteed to be isolated from other threads
  – Usually *succeed* or *fail*; no partial success
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* \( \text{cnt} = 0 \)

\( y = 0 \)

*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;
}

Shared state  cnt = 1

T1 executes again, storing its value of y + 1 into the counter.

y = 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*  
\[cnt = 1\]

\[Shared state\]
\[\text{y = 0}\]

\[\text{y = 1}\]

*T1 finishes. T2 executes, grabbing the global counter value into its own y.*
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

*y = 0*

Shared state  cnt = 2

*y = 1*

T2 executes, storing its incremented cnt value into the global counter.
But When it's Run Again?

• Suppose the second thread has a higher priority, and the first thread gets paused during execution
**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*  \(cnt = 0\)

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

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 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;
}

Shared state  cnt = 1

\[
y = 0
\]

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;
}
```

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

\( y = 0 \)

\( y = 0 \)

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
  – The second schedule (interleaving of threads) allowed atomicity to be violated
  – These bugs can be extremely hard to reproduce and debug
Question

• If instead of
  
  \[
  \text{int } y = \text{cnt} ; \\
  \text{cnt } = y + 1 ; \\
  \]

• We had written
  
  \[
  \text{cnt} \text{++} ; \\
  \]

• Would the result be any different?

• Answer: NO!
  
  – Don’t depend on your intuition about atomicity
Locks (Java) and Mutexes (Ruby)

```ruby
class Lock
  void lock();
  void unlock();
end
```

- Ruby: Mutex class in Thread library
- 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
Deadlock Conditions (Coffman)

- Mutual Exclusion
  - At least one resource must be non-sharable
- Hold and Wait
  - At least one process must be simultaneously holding and requesting resources
- No Pre-emption
  - The operating system cannot (or will not) break the deadlock by killing a process
- Circular Wait
  - E.g. wait graph
Dealing with deadlock

• Ignore it
  – The “ostrich algorithm”
• Detect it and recover from it
  – Kill or roll back a process
  – Re-allocate a resource
• Avoid it
  – Don’t allow resource acquisition if it will lead to deadlock (Banker’s algorithm)
• Prevention
  – Remove all possibility of one of the Coffman conditions
Classic: Dining Philosophers Problem

- Philosophers either eat or think
- They must have two forks to eat
- Can only use forks on either side of their plate
- No talking!
- Avoid deadlock and starvation!
Bad Dining Philosophers Solution 1

- Philosophers all pick up the left fork first

- Deadlock!
  - all are holding the left fork and waiting for the right fork
Bad Dining Philosophers Solution 2

- Philosophers all pick up the left fork first
- Philosophers put down a fork after waiting for 5 minutes, then wait 5 minutes before picking it up again
- Starvation!
Possible Solutions

• The waiter solution
  – Third party arbiter (scheduler)
  – Each thread requests permission before acquiring a resource

• The resource hierarchy solution
  – Impose ordering on resources
  – Must obtain resources in order
  – Most practical solution
  – Sometimes hard to know in advance
Ruby Locks

• Monitor, Mutex
  – Intended to be used by multiple threads
  – Methods are executed with mutual exclusion
    • As if all methods are synchronized
  – Monitor is reentrant, Mutex is not
Ruby Locks

• Create lock using `Monitor.new`
  – `Synchronize` method takes code block argument

```ruby
require 'monitor.rb'
myLock = Monitor.new
myLock.synchronize {
  # myLock held during this code block
}
```
Ruby Conditions

• Condition derived from Monitor
  – Create condition from lock using `new_cond`
  – Sleep while waiting using `wait_while, wait_until`
  – Wake up waiting threads using `broadcast`

• Example

```ruby
myLock = Monitor.new
myCondition = myLock.new_cond
myLock.synchronize {
  myCondition.wait_while {  y > 0  }
  myCondition.wait_until {  x != 0  }
}
myLock.synchronize {
  myCondition.broadcast
}
```
require "monitor.rb"

class ParkingLot
  def initialize
    @numCars = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end

  def addCar
    <next slide>
  end

  def removeCar
    <next slide>
  end
end
def addCar # do work not requiring synchronization
  @myLock.synchronize {
    @myCondition.wait_until { @numCars < MaxCars }
    @numCars = @numCars + 1
    @myCondition.broadcast
  }
end

def removeCar # do work not requiring synchronization
  @myLock.synchronize {
    @myCondition.wait_until { @numCars > 0 }
    @numCars = @numCars - 1
    @myCondition.broadcast
  }
end
Parking Lot Example

garage = ParkingLot.new
valet1 = Thread.new {  # valet 1 drives cars into parking lot
  while <time/car limit> do
    # do work not requiring synchronization
    garage.addCar
  end
}
valet2 = Thread.new {  # valet 2 drives car out of parking lot
  while <time/car limit> do
    # do work not requiring synchronization
    garage.removeCar
  end
}
valet1.join
valet2.join
Ruby vs. Java Threads

• Ruby thread can access all variables in scope when thread is created, including local variables
  – Java threads can only access object fields

• Exiting
  – All threads exit when main Ruby thread exits
  – Java continues until all non-daemon threads exit

• When thread throws exception
  – Ruby only aborts current thread (by default)
  – Ruby can also abort all threads (better for debugging)
    • Set `Thread.abort_on_exception = true`
ThreadError

• To handle a threading error:

  begin
  <threading code>
  rescue ThreadError
  <error handling code>
  ensure
  <cleanup>
  end
The third Example: C#

C# (pronounced see sharp) is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, procedural, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft within its .NET initiative and later approved as a standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270:2006). C# is one of the programming languages designed for the Common Language Infrastructure. C# is intended to be a simple, modern, general-purpose, object-oriented programming language. Its development team is led by Anders Hejlsberg. The most recent version is C# 5.0, which was released on August 15, 2012.

During the development of the .NET Framework, the class libraries were originally written using a managed code compiler system called Simple Managed C (SMC). In January 1999, Anders Hejlsberg formed a team to build a new language at the time called Cool, which stood for "C-like Object Oriented Language". [15] Microsoft had considered keeping the name "Cool" as the final name of the language, but chose not to do so for trademark reasons. By the time the .NET project was publicly announced at the July 2000 Professional Developers Conference, the language had been renamed C#, and the class libraries and ASP.NET runtime had been ported to C#.
.NET today

• Java-style “monitors”
• OS shared memory primitives
• Clunky delegate-based asynchronous calling model
• Hard to understand, use and get right
  – Different models at different scales
  – Support for asynchrony all on the caller side – little help building code to *handle* messages (must be thread-safe, reactive, and deadlock-free)
Polyphonic C#

• An extension of the C# language with new concurrency constructs
• Based on the join calculus
  – A foundational process calculus like the $\pi$-calculus but better suited to asynchronous, distributed systems
• A single model which works both for
  – local concurrency (multiple threads on a single machine)
  – distributed concurrency (asynchronous messaging over LAN or WAN)
• It is different
• But it’s also simple – if Mort can do any kind of concurrency, he can do this
In one slide:

- Objects have both *synchronous* and *asynchronous* methods.
- Values are passed by ordinary method calls:
  - If the method is synchronous, the caller blocks until the method returns some result (as usual and as in Java).
  - If the method is *async*, the call completes at once and returns void.
- A class defines a collection of *chords* (synchronization patterns), which define what happens once a particular set of methods have been invoked. One method may appear in several chords.
  - When pending method calls match a pattern, its body runs.
  - If there is no match, the invocations are queued up.
  - If there are several matches, an unspecified pattern is selected.
  - If a pattern containing *only* async methods fires, the body runs in a new thread.
A simple buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

A simple buffer

```java
class Buffer {
    String get() & async put(String s) {
        return s;
    }
}
```

• An ordinary (synchronous) method with no arguments, returning a string
A simple buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

• An ordinary (synchronous) method with no arguments, returning a string

• An asynchronous method (hence returning no result), with a string argument
A simple buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

• An ordinary (synchronous) method with no arguments, returning a string

• An asynchronous method (hence returning no result), with a string argument

• Joined together in a chord
A simple buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

- Calls to `put()` return immediately (but are internally queued if there’s no waiting `get()`).
- Calls to `get()` block until/unless there’s a matching `put()`.
- When there’s a match the body runs, returning the argument of the `put()` to the caller of `get()`.
- Exactly which pairs of calls are matched up is unspecified.
A simple buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

• Does example this involve spawning any threads?
  • No. Though the calls will usually come from different pre-existing threads.

• So is it thread-safe? You don’t seem to have locked anything…
  • Yes. The chord compiles into code which uses locks. (And that doesn’t mean everything is synchronized on the object.)

• Which method gets the returned result?
  • The synchronous one. And there can be at most one of those in a chord.
Reader/Writer (Yet another Producer and Consumer? Yes)
...using threads and mutexes in Modula 3

VAR i: INTEGER;
VAR m: Thread.Mutex;
VAR c: Thread.Condition;

PROCEDURE AcquireExclusive();
BEGIN
  LOCK m DO
    WHILE i # 0 DO Thread.Wait(m,c) END;
    i := -1;
  END;
END AcquireExclusive;

PROCEDURE AcquireShared();
BEGIN
  LOCK m DO
    WHILE i < 0 DO Thread.Wait(m,c) END;
    i := i+1;
  END;
END AcquireShared;

PROCEDURE ReleaseExclusive();
BEGIN
  LOCK m DO
    i := 0; Thread.Broadcast(c);
  END;
END ReleaseExclusive;

PROCEDURE ReleaseShared();
BEGIN
  LOCK m DO
    i := i-1;
    IF i = 0 THEN Thread.Signal(c) END;
  END;
END ReleaseShared;

An introduction to programming with threads.
public class ReaderWriter {
    public void Exclusive() & async Idle() {} 
    public void ReleaseExclusive() { Idle(); } 

    public void Shared() & async Idle() { S(1); } 
    public void Shared() & async S(int n) { S(n+1); } 
    public void ReleaseShared() & async S(int n) {
        if (n == 1) Idle(); else S(n-1); 
    } 

    public ReaderWriter() { Idle(); } 
}

A single private message represents the state:

\[
\text{none} \leftrightarrow \text{Idle()} \leftrightarrow S(1) \leftrightarrow S(2) \leftrightarrow S(3) \ldots
\]
Asynchronous requests and responses

- Service exposes an async method which takes parameters and somewhere to put the result:
  - a buffer, or a channel, or
  - a delegate

```java
public delegate async IntCB(int v);

public class Service {
    public async request(String arg, IntCB callback) {
        int result;
        // do something interesting...
        callback(result);
    }
}
```
Asynchronous requests and responses - Join

class Join2 {
    void wait(out int i, out int j)
    & async first(int r1)
    & async second(int r2) {
        i = r1; j = r2; return;
    }
}

// client code:
int i,j;
Join2 x = new Join2();
service1.request(arg1, new IntCB(x.first));
service2.request(arg2, new IntCB(x.second));
// do something useful
// now wait until both results have come back
x.wait(out i,out j);
// do something with i and j
Asynchronous requests and responses - Select

class Select {
   int wait()
   & async reply(int r) {
      return r;
   }
}

// client code:
int i;
Select x = new Select();
service1.request(arg1, new IntCB(x.reply));
service2.request(arg2, new IntCB(x.reply));
// do something useful
// now wait until one result has come back
i = x.wait();
// do something with i
The problem with inheritance

class C {
    virtual void f() & virtual async g() {...}
    virtual void f() & virtual async h() {...}
}
class D : C {
    override async g() { ...}
}

• We’ve “half” overridden f
• Too easy to create deadlock or async leakage

    void m(C x) { x.g(); x.f();}
    ...
    m(new D());
The inheritance restriction

- Two methods are *co-declared* if they appear together in a chord declaration.

> Whenever a method is overridden, every co-declared method must also be overridden.

- Hence, the compiler rejects patterns such as

  ```
  public virtual void f() & private async g() {...}
  ```

- In general, inheritance and concurrency do not mix well. Our restriction is simple; it could be made less restrictive.
Types etc.

- **async** is a subtype of **void**
- Allow covariant return types on those two:
  - An **async** method may override a **void** one
  - A **void** delegate may be created from an **async** method
  - An **async** method may implement a **void** method in an interface
- **async** methods are given the `[OneWay]` attribute, so remote calls are non-blocking
Implementation

• Translate Polyphonic C# -> C#
• Built on Proebsting & Hanson’s Icsc
• Introduce queues for pending calls (holding blocked threads for sync methods, arguments for asyncs)
• Generated code (using brief lock to protect queue state) looks for matches and then either
  – Enqueues args (async no match)
  – Enqueues thread and blocks (sync no match)
  – Dequeues other args and continues (sync match)
  – Wakes up blocked thread (async match with sync)
  – Spawns new thread (async match all async)
• Efficient – bitmasks to look for matches, no PulseAlls,...
Samples

• animated dining philosophers
• web service combinators (Cardelli & Davies)
• adaptive scheduler (cf. Larus & Parkes),
• accessing web services (Terraserver),
• active objects and remoting (stock trader)
What’s Next?

• Extension to existing language...
  – Universal Parallel C (UPC)
  – High-Performance Fortran (HPF)
  – POSIX threads
• ...or an entirely new language?
  – Chapel (Cray)
  – X10 (IBM)
• No clear consensus yet