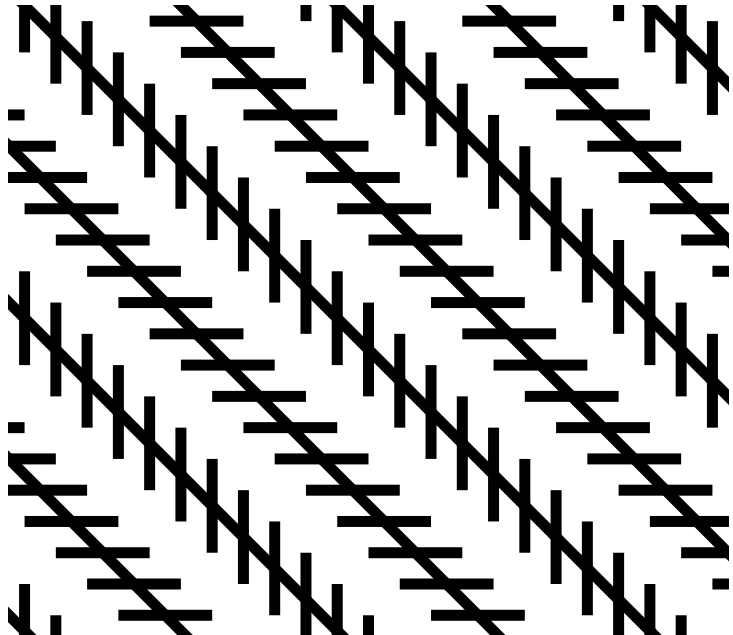
Parallel Programming in OpenMP and Java

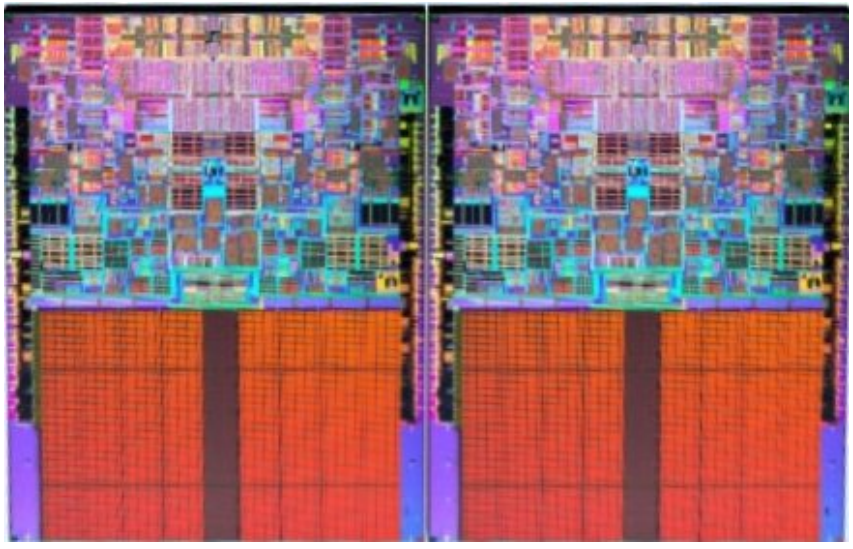
Stephen A. Edwards

Columbia University

Summer 2014



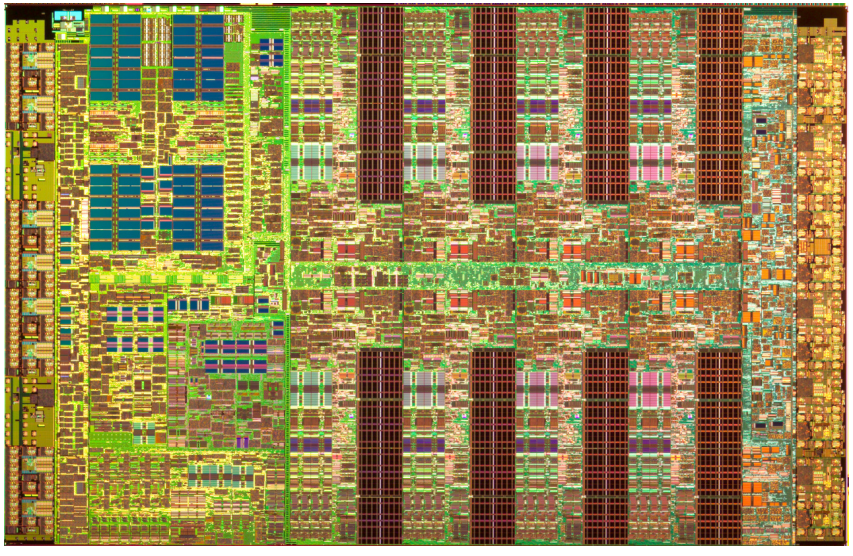
Parallel Computer Architectures



Intel Quad Core Duo

Four identical x86 processor cores

Separate L1 caches, shared L2 cache and main memory



IBM Cell Broadband Engine

One general-purpose PPE w/ 32K L1 caches and 512K L2 cache

Eight vector SPEs, each with individual 256K memories

Processes and Threads

Process

Separate address space

Explicit inter-process communication

Synchronization part of communication

Operating system calls: `fork()`, `wait()`

Thread

Shared address spaces

Separate PC and stacks

Communication is through shared memory

Synchronization done explicitly

Library calls, e.g., pthreads

Part I

OpenMP

Dot Product

```
#include <stdio.h>

int main()
{
  double a[256], b[256], sum;
  int i, n = 256;

  for (i = 0 ; i < n ; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  sum = 0.0;

  for (i = 0 ; i < n ; i++ )
    sum += a[i]*b[i];

  printf("sum = %f\n", sum);
  return 0;
}
```

Dot Product

```
#include <omp.h>
#include <stdio.h>

int main()
{
  double a[256], b[256], sum;
  int i, n = 256;

  for (i = 0 ; i < n ; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  sum = 0.0;
# pragma omp parallel for reduction(+:sum)
  for (i = 0 ; i < n ; i++ )
    sum += a[i]*b[i];

  printf("sum = %f\n", sum);
  return 0;
}
```

OpenMP version

`gcc -fopenmp -o dot dot.c`

OpenMP pragma tells the compiler to

- Execute the loop's iterations in parallel (multiple threads)
- Sum each thread's results

What's going on?

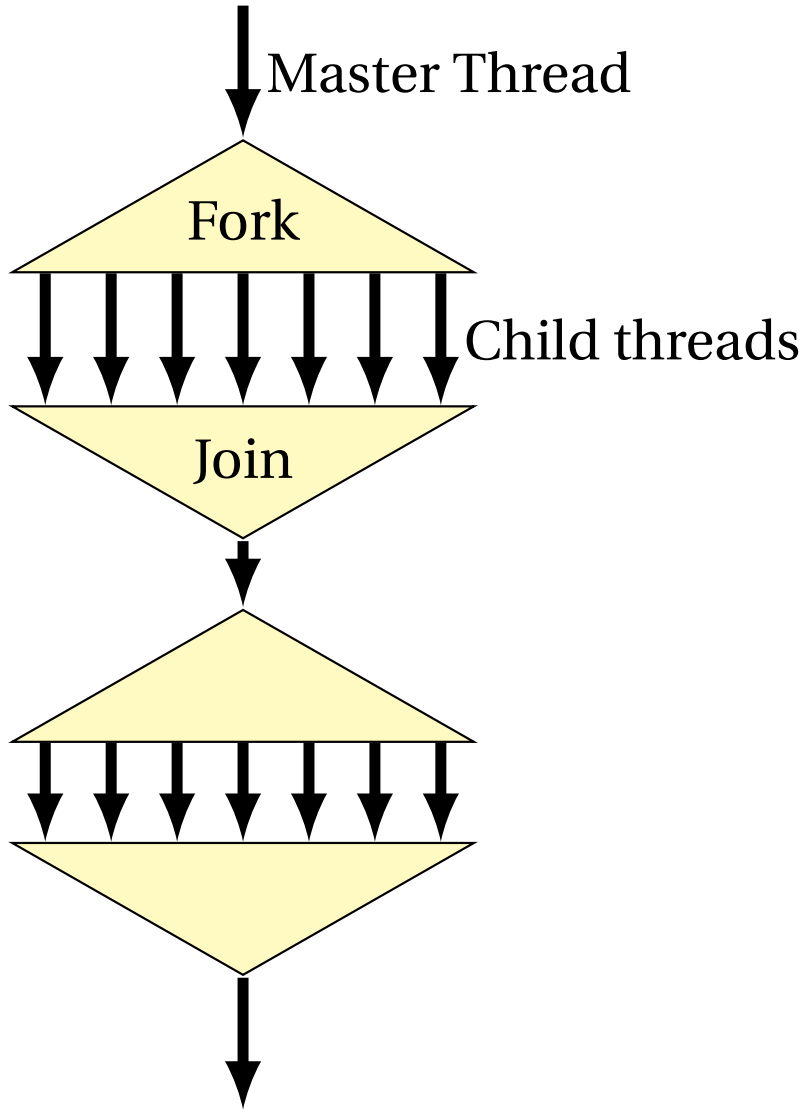
```
#include <omp.h>
#include <stdio.h>

int main() {
  int i, n = 16;

# pragma omp parallel for \
    shared(n) private(i)
  for (i = 0 ; i < n ; i++)
    printf("%d: %d\n",
      omp_get_thread_num(), i);

  return 0;
}
```

"parallel for": Split for loop iterations into multiple threads
"shared(n)": Share variable n across threads
"private(i)": Each thread has a separate i variable

Run 1	Run 2	Run 3
1: 4	3: 12	0: 0
1: 5	3: 13	0: 1
1: 6	3: 14	0: 2
1: 7	3: 15	0: 3
2: 8	1: 4	3: 12
2: 9	1: 5	3: 13
2: 10	1: 6	3: 14
2: 11	1: 7	3: 15
0: 0	2: 8	1: 4
3: 12	2: 9	1: 5
3: 13	2: 10	1: 6
3: 14	2: 11	1: 7
3: 15	0: 0	2: 8
0: 1	0: 1	2: 9
0: 2	0: 2	2: 10
0: 3	0: 3	2: 11

Fork-Join Task Model

Spawn tasks in parallel, run each to completion, collect the results.



Parallel Regions

```c
#include <omp.h>
#include <stdio.h>

int main()
{

# pragma omp parallel
  { /* Fork threads */

    printf("This is thread %d\n",
           omp_get_thread_num());

  } /* Join */

  printf("Done.\n");

  return 0;
}
```

```
$ ./parallel
This is thread 1
This is thread 3
This is thread 2
This is thread 0
Done.
$ ./parallel
This is thread 2
This is thread 1
This is thread 3
This is thread 0
Done.
$ OMP_NUM_THREADS=8 ./parallel
This is thread 0
This is thread 4
This is thread 1
This is thread 5
This is thread 3
This is thread 6
This is thread 2
This is thread 7
Done.
```

Work Sharing: For Construct

```
#include <omp.h>
#include <stdio.h>

int main()
{
  int i;
# pragma omp parallel
  {
#   pragma omp for
    for (i = 0 ; i < 8 ; i++)
      printf("A %d[%d]\n",
        omp_get_thread_num(), i);
#   pragma omp for
    for (i = 0 ; i < 8 ; i++)
      printf("B %d[%d]\n",
        omp_get_thread_num(), i);
  }

  return 0;
}
```

```
$ ./for
A 3[6]
A 3[7]
A 0[0]
A 0[1]
A 2[4]
A 2[5]
A 1[2]
A 1[3]
B 1[2]
B 1[3]
B 0[0]
B 0[1]
B 3[6]
B 3[7]
B 2[4]
B 2[5]
```

Work Sharing: Nested Loops

Outer loop's index private by default.
Inner loop's index must be set private.

```
#include <omp.h>
#include <stdio.h>

int main()
{
  int i, j;
# pragma omp parallel for private(j)
    for (i = 0 ; i < 4 ; i++)
      for (j = 0 ; j < 2 ; j++)
        printf("A %d[%d,%d]\n",
          omp_get_thread_num(), i, j);

  return 0;
}
```

```
p$ ./for-nested
A 3[3,0]
A 3[3,1]
A 1[1,0]
A 1[1,1]
A 2[2,0]
A 2[2,1]
A 0[0,0]
A 0[0,1]

$ ./for-nested
A 3[3,0]
A 3[3,1]
A 1[1,0]
A 1[1,1]
A 0[0,0]
A 0[0,1]
A 2[2,0]
A 2[2,1]
```

Summing a Vector: Critical regions

```
#pragma omp parallel for
for (i = 0 ; i < N ; i++)
  sum += a[i]; // RACE: sum is shared
```

```
int main()
{
  double sum = 0.0, local, a[10];
  int i;
  for ( i = 0 ; i < 10 ; i++)
    a[i] = i * 3.5;
# pragma omp parallel \
    shared(a, sum) private(local)
  {
    local = 0.0;
#   pragma omp for
    for (i = 0 ; i < 10 ; i++)
      local += a[i];
#   pragma omp critical
    { sum += local; }
  }
  printf("%g\n", sum);
  return 0;
}
```

Without the `critical` directive:

```
$ ./sum
157.5
$ ./sum
73.5
```

With `#pragma omp critical`:

```
$ ./sum
157.5
$ ./sum
157.5
```

Summing a Vector: Reduction

```
#include <omp.h>
#include <stdio.h>

int main()
{
  double sum = 0.0, a[10];
  int i;
  for ( i = 0 ; i < 10 ; i++)
    a[i] = i * 3.5;

# pragma omp parallel \
    shared(a) reduction(+:sum)
  {
#   pragma omp for
    for (i = 0 ; i < 10 ; i++)
      sum += a[i];
  }

  printf("%g\n", sum);
  return 0;
}
```

```
$ ./sum-reduction
157.5
```

Barriers

```
#include <omp.h>
#include <stdio.h>

int main()
{
  int x = 2;
# pragma omp parallel shared(x)
  {
    int tid = omp_get_thread_num();
    if (tid == 0)
      x = 5;
    else
      printf("A: th %d: x=%d\n",
             tid, x);

#   pragma omp barrier

    printf("B: th %d: x=%d\n",
           tid, x);
  }
  return 0;
}
```

```
$ ./barrier
A: th 2: x=2
A: th 3: x=2
A: th 1: x=2
B: th 1: x=5
B: th 0: x=5
B: th 2: x=5
B: th 3: x=5

$ ./barrier
A: th 2: x=2
A: th 3: x=5
A: th 1: x=2
B: th 1: x=5
B: th 2: x=5
B: th 3: x=5
B: th 0: x=5
```

Old value of x

New value of x

Summary of OpenMP

Shared memory model

Explicit private and shared directives

Fork/Join Concurrency

Good support for splitting up simple loops over arrays.

Critical regions

Barriers

Reduction operators

Part II

Java's Support for Concurrency

Thread Basics

How to create a thread:

```
class MyThread extends Thread {
  public void run() { // A thread's "main" method
    /* thread body */
  }
}

MyThread mt = new MyThread; /* Create thread */
mt.start();  /* Start thread at run() */
// Returns immediately}
```

Threads are multiple contexts/program counters running within the same memory space

All objects shared among threads by default

Any thread can access any object/method: no notion of ownership

implements Runnable vs. extends Thread

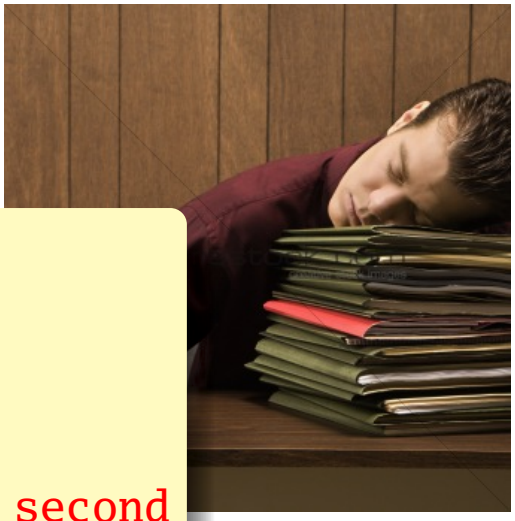
An alternative:

```
class MyRunnable implements Runnable {
  public void run() {
    /* thread body */
  }
}

Thread t = new Thread(new MyRunnable());
t.start();  /* Starts thread at run() */
// Returns immediately}
```

Advantage: class implementing Runnable can be derived from some other class (i.e., not just from Thread).

Disadvantage: “this” is not a Thread so, e.g., sleep() must be called with Thread.currentThread().sleep().

The Sleep Method



```
class Sleeper extends Thread {
  public void run() {
    for (;;) {
      try {

        sleep(1000); // Pause for at least a second

      } catch (InterruptedException e) {
        return;  // caused by thread.interrupt()
      }

      System.out.println("tick");
    }
  }
}
```

Does this print “tick” once a second?

A Clock?

```
class PrintingClock implements Runnable {
  public void run() {
    for (;;) {

      java.util.Date now = new java.util.Date();
      System.out.println(now.toString());

      try {

        Thread.currentThread().sleep(1000);

      } catch (InterruptedException e) {}
    }
  }
}

public class Clock {
  public static void main(String args[]) {

    Thread t = new Thread(new PrintingClock());
    t.start();

  }
}
```

What does the clock print?

```
$ java Clock
Sat Sep 14 13:04:27 EDT 2002
Sat Sep 14 13:04:29 EDT 2002
Sat Sep 14 13:04:30 EDT 2002
Sat Sep 14 13:04:31 EDT 2002
```

What happened to 13:04:28?

Motivation for Synchronization

Something you might want to implement:

```
class ScoreKeeper {

  int _score = 0;

  void score(int v) {
    int tmp = _score;
    tmp += v;
    _score = tmp;
  }
}
```

What could the final score be if two threads simultaneously call `score(1)` and `score(2)`?

Non-atomic Operations

Java guarantees 32-bit reads and writes to be atomic

64-bit operations may not be

Therefore,

```
int i;
double d;
```

Thread 1

```
i = 10;
d = 10.0;
```

Thread 2

```
i = 20;
d = 20.0;
```

Per-Object Locks

Each Java object has a lock that may be owned by at least one thread

A thread waits if it attempts to obtain an already-obtained lock

The lock is a counter: one thread may lock an object more than once

The Synchronized Statement

A synchronized statement waits for an obtains an object's lock before running its body

```
Counter mycount = new Counter;

synchronized(mycount) {
  mycount.count();
}
```

Releases the lock when the body terminates.

Choice of object to lock is by convention.

Synchronized Methods

```
class AtomicCounter {

  private int _count;

  public synchronized void count() {
    _count++;
  }
}
```

`synchronized` attribute equivalent to enclosing body with `synchronized (this)` statement.

Most common way to achieve object-level atomic operations.

Implementation guarantees at most one thread can run a synchronized method on a particular object at once

Deadlock

```
synchronized(Foo) {
  synchronized(Bar) {
    // Asking for trouble
  }
}
```

```
synchronized(Bar) {
  synchronized(Foo) {
    // Asking for trouble
  }
}
```

Rule: always acquire locks in the same order.

Synchronization

Say you want a thread to wait for a condition before proceeding.

An infinite loop may deadlock the system (e.g., if it's using cooperative multitasking).

```
while (!condition) {}
```

Calling `yield` avoids deadlock, but is inefficient:

```
while (!condition) yield();
```

Java's Solution: wait() and notify()

`wait()` like `yield()`, but requires other thread to reawaken it

```
while (!condition) wait();
```

Thread that changes the condition calls `notify()` to resume the thread.

Programmer responsible for ensuring each `wait()` has a matching `notify()`.

Wait() and Notify() in Real Life

I often have books delivered to the CS department office.

This operation consists of the following steps:

1. Place the order
2. Retrieve the book from the CS department office
3. Place book on bookshelf

Obviously, there's a delay between steps 1 and 2.

The Implementation in the CS Office

This “order, retrive, file” thread is running in me, and it needs to wait for the book to arrive.

I could check the department office every minute, hour, day, etc. to see if the book has come in, but this would be waste of time (but possibly good exercise).

Better approach would be to have the “receiving” process alert me when the book actually arrives.

This is what happens: Alice in the front office sends me email saying I have a book.

A Flawed Implementation

```
class Mailbox {}
public class BookOrder {
  static Mailbox m = new Mailbox();

  public static void main(String args[]) {
    System.out.println("Ordering book");
    Thread t = new Delivery(m);
    t.start();
    synchronized (m) {
      try { m.wait();
      } catch (InterruptedException e) {}
    }
    System.out.println("Book Arrived");
  }
}
class Delivery extends Thread {
  Mailbox m;
  Delivery(Mailbox mm) { m = mm; }
  public void run() {
    try { sleep(1000);  // one-second delivery
    } catch (InterruptedException e) {}
    synchronized (m) {
      m.notify();
    }
  }
}
```

A Flawed Implementation

What happens if the book is delivered before the main thread starts waiting?

Answer: the “notify” instruction does not wake up any threads. Later, the main thread starts to wait and will never be awakened.

As if Alice came to my office to tell me when I was not there.

Harder Problem

Sometimes I have a number of books on order, but Alice only tells me a book has arrived. How do I handle waiting for a number of books?

Last solution assumed a single source of `notify()`*; not true in general.*

Two rules:

1. Use `notifyAll()` when more than one thread may be waiting

2. Wait in a `while` if you could be notified early, e.g.,

```
while (!condition)
  wait();
```

The multi-book problem

```
class Mailbox {
  int book;
  Mailbox() { book = -1; }
  void receive_book(int b) { book = b; }
  int which_book() { return book; }
}
```

This is not thread-safe: we'll need to synchronize all access to it.

The multi-book problem

The Delivery class also tells the mailbox which book it got.

```
class Delivery extends Thread {
  Mailbox m;
  int book;
  Delivery(Mailbox mm, int b) {
     m = mm; book = b;
  }
  public void run() {
    try { sleep(1000); }
    catch (InterruptedException e) {}
    synchronized (m) {
      m.receive_book(book);
      m.notifyAll();
    }
  }
}
```

The multi-book problem

```
class BookOrder extends Thread {
  int book;
  Mailbox m;
  BookOrder(Mailbox mm, int b) { m = mm; book = b; }
  public void run() {
    System.out.println("Ordering book " +
                       Integer.toString(book) );
    Thread t = new Delivery(m, book);
    t.start();
    synchronized (m) {
      while (m.which_book() != book) {
        try { m.wait(); }
        catch (InterruptedException e) {}
      }
    }
    System.out.println("Book " +
     Integer.toString(book) + " Arrived");
  }
}
```

The multi-book problem

Finally, the main routine kicks off two ordering threads.

```
public class MultiOrder {
  static Mailbox m = new Mailbox();
  public static void main(String args[]) {
    BookOrder t1 = new BookOrder(m, 1);
    BookOrder t2 = new BookOrder(m, 2);
    t1.start();
    t2.start();
  }
}
```

```
$ java MultiOrder
Ordering book 1
Ordering book 2
Book 1 Arrived
Book 2 Arrived
```

A Better Solution

Last solution relied on threads to synchronize their own access to the Mailbox. Mailbox should be doing this itself:

```
class Mailbox {
  int book;
  Mailbox() { book = -1; }

  synchronized void deliver(int b) {
    book = b;
    notifyAll();
  }

  synchronized void wait_for_book(int b) {
    while (book != b) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
  }
}
```

A Better Solution

```
class Delivery extends Thread {
  Mailbox m;
  int book;

  Delivery(Mailbox mm, int b) {
    m = mm; book = b;
  }

  public void run() {
    try { sleep(1000); }
    catch (InterruptedException e) {}
    m.deliver(book);
  }
}
```

A Better Solution

```
class BookOrder extends Thread {
  int book;
  Mailbox m;

  BookOrder(Mailbox mm, int b) {
     m = mm; book = b;
  }

  public void run() {
    System.out.println("Ordering book " +
      Integer.toString(book) );
    Thread t = new Delivery(m, book);
    t.start();
    m.wait_for_book(book);
    System.out.println("Book " +
     Integer.toString(book) + " Arrived");
  }
}
```

A Better Solution

```
public class MultiOrder2 {
  static Mailbox m = new Mailbox();
  public static void main(String args[]) {
    BookOrder t1 = new BookOrder(m, 1);
    BookOrder t2 = new BookOrder(m, 2);
    t1.start();
    t2.start();
  }
}
```

```
$ java MultiOrder2
Ordering book 1
Ordering book 2
Book 1 Arrived
Book 2 Arrived
```

Building a Blocking Buffer

Problem: Build a single-place buffer for Objects that will block on write if the buffer is not empty and on read if the buffer is not full.

```
interface OnePlace {
  public void write(Object o);
  public Object read();
}
```

Blocking Buffer: Write Method

```
class MyOnePlace implements OnePlaceBuf {
  Object o;

  public synchronized // ensure atomic updates
  void write(Object oo) {

    try {
      while (o != null)
        wait();  // Block while buffer is full
    } catch (InterruptedException e) {}

    o = oo;       // Fill the buffer
    notifyAll(); // Awaken any waiting processes
  }
}
```

Blocking Buffer: Read Method

```
class MyOnePlace implements OnePlaceBuf {
  Object o;

  public synchronized // ensure atomic updates
  Object read() {

    try {
      while (o == null)
        wait();  // Block while buffer is empty
    } catch (InterruptedException e) {}

    Object oo = o; // Get the object
    o = null;      // Empty the buffer
    notifyAll();   // Awaken any waiting processes
    return oo;
  }
}
```

Blocking Buffer: Reader and Writer

```
class Writer extends Thread {
  OnePlaceBuf b;
  Writer(OnePlaceBuf bb) { b = bb; }
  public void run() {
    for (int i = 0 ; ; i++ ) {
      b.write(new Integer(i)); // Will block
    }
  }
}
class Reader extends Thread {
  OnePlaceBuf b;
  Reader(OnePlaceBuf bb) { b = bb; }
  public void run() {
    for (;;) {
      Object o = b.read(); // Will block
      System.out.println(o.toString());
    }
  }
}
```

Blocking Buffer: Main Routine

```
public class OnePlace {
  static MyOnePlace b = new MyOnePlace();
  public static void main(String args[]) {
    Reader r = new Reader(b);
    Writer w = new Writer(b);
    r.start();
    w.start();
  }
}
```

Summary of Java's Concurrency Support

- Thread-based shared memory model
  Fundamentally nondeterminstic; correctness the resposibility of the programmer
- Per-object locks can ensure atomic access
  Choice of locks by convention, although there is often a natural choice. Danger of deadlock.
- Wait/notify mechanism for synchronizing threads
  Undisciplined use can lead to races and deadlocks, but following a pattern avoids most problems.