Concepts of Programming Languages

Lecture 18 - Concurrency

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Assignments:

Programming #4 : due 04.28

Final Exam: 05.01 4:00-5:50p

Reading:

Chapter 13

Two roads diverged in a yellow wood,
And sorry I could not travel both ...

Robert Frost

Concurrency

Concurrency occurs at many levels

- Machine instruction level
- High-level language statement level
- Unit level
- Program level

More realistic

Can be more efficient

Carries unique, fundamental complexities

Traditionally studied in the context of operating systems

Concurrency Concepts

Example: client-server application such as web browsing.

Example

Web browser rendering a page

- Page is a shared resource
- Thread for each image load
- Thread for text rendering
- Cannot all write to page simultaneously
- Thread for user input; e.g., Stop button

Timeline of Multiprocessor Architectures

Late 1950s - one general-purpose processor and one or more special-purpose processors for input and output operations

Early 1960s - multiple complete processors, used for program-level concurrency

Mid-1960s - multiple partial processors, used for instruction-level concurrency

Single-Instruction Multiple-Data (SIMD) machines

Multiple-Instruction Multiple-Data (MIMD) machines

A primary focus of this chapter is shared memory MIMD machines (multiprocessors)

Categories of Concurrency

1. Physical concurrency - Multiple independent processors (multiple threads of control)
2. Logical concurrency - The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)

Co-routines (quasi-concurrency) have a single thread of control

A thread of control in a program is the sequence of program points reached as control flows through the program

Motivations for the Use of Concurrency

Multiprocessor computers capable of physical concurrency are now widely used

Even if a machine has just one processor, a program written to use concurrent execution can be faster than the same program written for nonconcurrent execution

Involves a different way of designing software that can be very useful-many real-world situations involve concurrency

Many program applications are now spread over multiple machines, either locally or over a network

Concurrency Concepts

Definition

Multiprogramming: several programs loaded into memory and executed in an interleaved manner

Concurrency Concepts

Definition

Multiprogramming: several programs loaded into memory and executed in an interleaved manner

Definition

Scheduler: switches from one program or thread to another

Concurrency Concepts

Definition

Multiprogramming: several programs loaded into memory and executed in an interleaved manner

Definition

Scheduler: switches from one program or thread to another

Definition

Time-sharing: allow multiple users to communicate with a computer simultaneously

Concurrency Concepts

Definition

Multiprogramming: several programs loaded into memory and executed in an interleaved manner

Definition

Scheduler: switches from one program or thread to another

Definition

Time-sharing: allow multiple users to communicate with a computer simultaneously

Definition

Process: an execution context, including registers, activation stack, next instruction to be executed, etc.

Concurrency Concepts

Definition

A ***concurrent program*** is a program designed to have two or more execution contexts. Such a program is said to be multithreaded, since more than one execution context can be active simultaneously.

Concurrency Concepts

Definition

A ***concurrent program*** is a program designed to have two or more execution contexts. Such a program is said to be multithreaded, since more than one execution context can be active simultaneously.

Definition

Parallel program: Two or more threads simultaneously active.

Concurrency Concepts

Definition

A ***concurrent program*** is a program designed to have two or more execution contexts. Such a program is said to be multithreaded, since more than one execution context can be active simultaneously.

Definition

Parallel program: Two or more threads simultaneously active.

Definition

Distributed program: designed so that different pieces are on computers connected by a network.

Concurrency Concepts

Definition

A ***concurrent program*** is a program designed to have two or more execution contexts. Such a program is said to be multithreaded, since more than one execution context can be active simultaneously.

Definition

Parallel program: Two or more threads simultaneously active.

Definition

Distributed program: designed so that different pieces are on computers connected by a network.

Definition

Concurrency: a program with multiple, active threads

Tasks

Definition

A ***task*** or ***process*** or ***thread*** is a program unit that can be in concurrent execution with other program units

Tasks

Definition

A ***task*** or ***process*** or ***thread*** is a program unit that can be in concurrent execution with other program units

Tasks differ from ordinary subprograms in that:

- A task may be implicitly started
- When a program unit starts the execution of a task, it is not necessarily suspended
- When a task's execution is completed, control may not return to the caller

Tasks usually work together

Tasks

Definition

Heavyweight tasks execute in their own address space.

Definition

Lightweight task all run in the same address space - more efficient

Definition

A task is ***disjoint*** if it does not communicate with or affect the execution of any other task in the program in any way

Task Synchronization

A mechanism that controls the order in which tasks execute

Two kinds of synchronization

- Cooperation synchronization
- Competition synchronization

Task communication is necessary for synchronization, provided by:

- Shared nonlocal variables
- Parameters
- Message passing

Kinds of Synchronization

Definition

Cooperation: Task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer-consumer problem

Definition

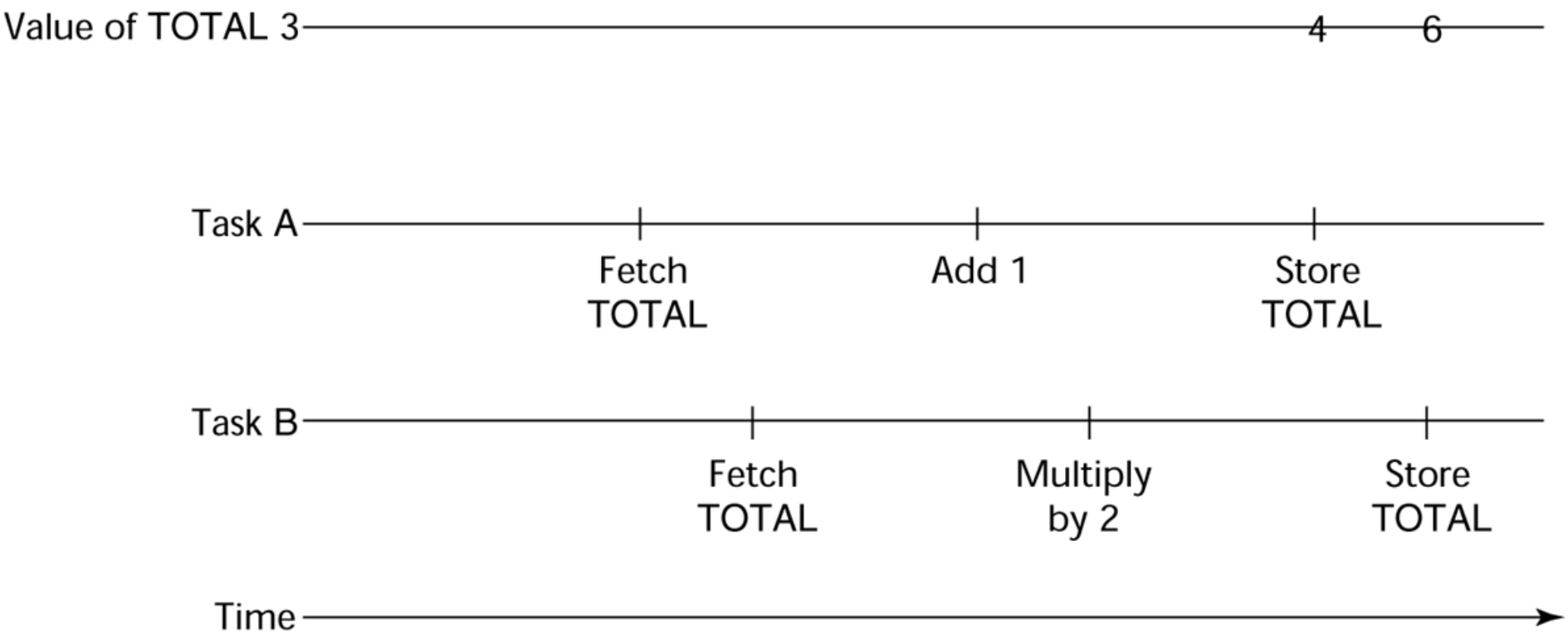
Competition: Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter

- Competition is usually provided by mutually exclusive access (approaches are discussed later)

Need for Competition Synchronization

Task A: `TOTAL = TOTAL + 1`

Task B: `TOTAL = 2 * TOTAL`



Depending on order, there could be four different results

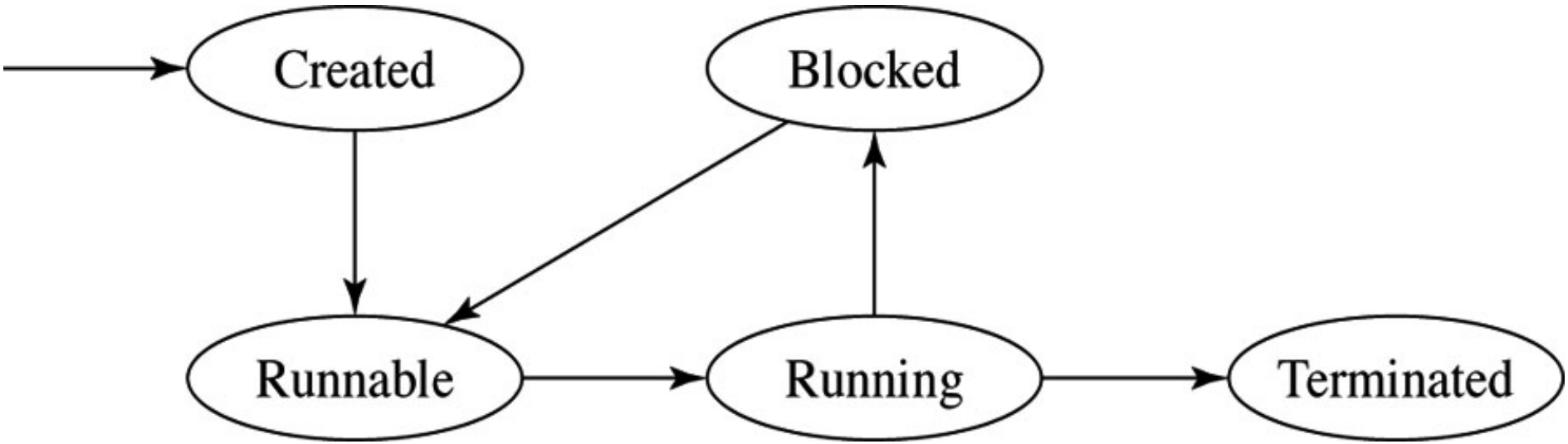
Scheduler

Providing synchronization requires a mechanism for delaying task execution

Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors

States of a Thread

1. Created (New): but not yet ready to run
2. Runnable (Ready): ready to run; awaiting a processor
3. Running: executing
4. Blocked: waiting on some resource
5. Terminated (Dead): stopped; no longer active in any sense



Threads

Inter-thread communication needs to occur:

1. Thread requires exclusive access to some resource
2. Thread needs to exchange data with another thread

Can communicate via:

1. Shared variables
2. Message passing
3. Parameters

Race Condition

Definition

A ***race condition*** occurs when the resulting value of a variable depends on the execution order of two or more threads.

Example

```
c = c + 1
```

Machine level:

1. load c
2. add 1
3. store c

If c initially 0:

- With 2 threads, can get 1 or 2
- With n threads, can get 1, 2, . . . , n

Deadlock

Definition

A ***deadlock*** occurs when a thread is waiting for an event that will never happen.

Necessary conditions for a deadlock to exist:

1. Threads claim exclusive access to resources.
2. Threads hold some resources while waiting for others.
3. Resources may not be removed from waiting threads (*preemption*).
4. A circular chain of threads exists in which each thread holds a resource needed by the next thread in the chain.

Design Issues for Concurrency

- Competition and cooperation synchronization
- Controlling task scheduling
- How can an application influence task scheduling
- How and when tasks start and end execution
- How and when are tasks created

Methods of Providing Synchronization

Semaphores - data structure used for controlling access, by multiple processes, to a common resource.

Monitors - abstract data type to encapsulate the shared data and its operations in order to restrict access.

Methods of Providing Synchronization

Semaphores - data structure used for controlling access, by multiple processes, to a common resource.

Monitors - abstract data type to encapsulate the shared data and its operations in order to restrict access.

Semaphores

Definition

A ***semaphore*** semaphore is a data structure consisting of a counter and a queue for storing task descriptors.

- A task descriptor is a data structure that stores all of the relevant information about the execution state of the task

Originally defined by Dijkstra in 1968.

Operations:

- `P(s)` – if s > 0 then s– else enqueue thread
- `V(s)` – if a thread is enqueued then dequeue it else s++

Binary semaphore

Counting semaphore

Concurrent Pascal Example

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

Concurrent Pascal Example

```
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;
```

Concurrent Pascal Example

```
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;
```

Concurrent Pascal Example

```
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;
...
```

Concurrent Pascal Example

```
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;
```

Concurrent Pascal Example

```
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 }
          V(nonfull);
          consume(tmp);
     end;
end;
```

Evaluation of Semaphores

Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer can overflow.

Misuse of semaphores can cause failures in competition synchronization, e.g., the program can deadlock if the release is left out

Methods of Providing Synchronization

Semaphores - data structure used for controlling access, by multiple processes, to a common resource.

Monitors - abstract data type to encapsulate the shared data and its operations in order to restrict access.

Monitors

Encapsulates a shared resource together with access functions.

Used in Ada, Java, C#

Locking is automatic.

Monitor implementation guarantee synchronized access by allowing only one access at a time

Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Condition – thread queue

- signal
- wait

Concurrent Pascal Example

```
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;
```

Concurrent Pascal Example

```
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;
```

Concurrent Pascal Example

```
function get : string;
var tmp : string
begin
    if (count = 0) then  wait(nonempty);
    out = out mod size + 1;
    tmp := buffer[out];
    count := count - 1;
    signal(nonfull);
    get := tmp;

end;
```

Java Threads

The concurrent units in Java are methods named run

- A run method code can be in concurrent execution with other such methods
- The process in which the run methods execute is called a thread

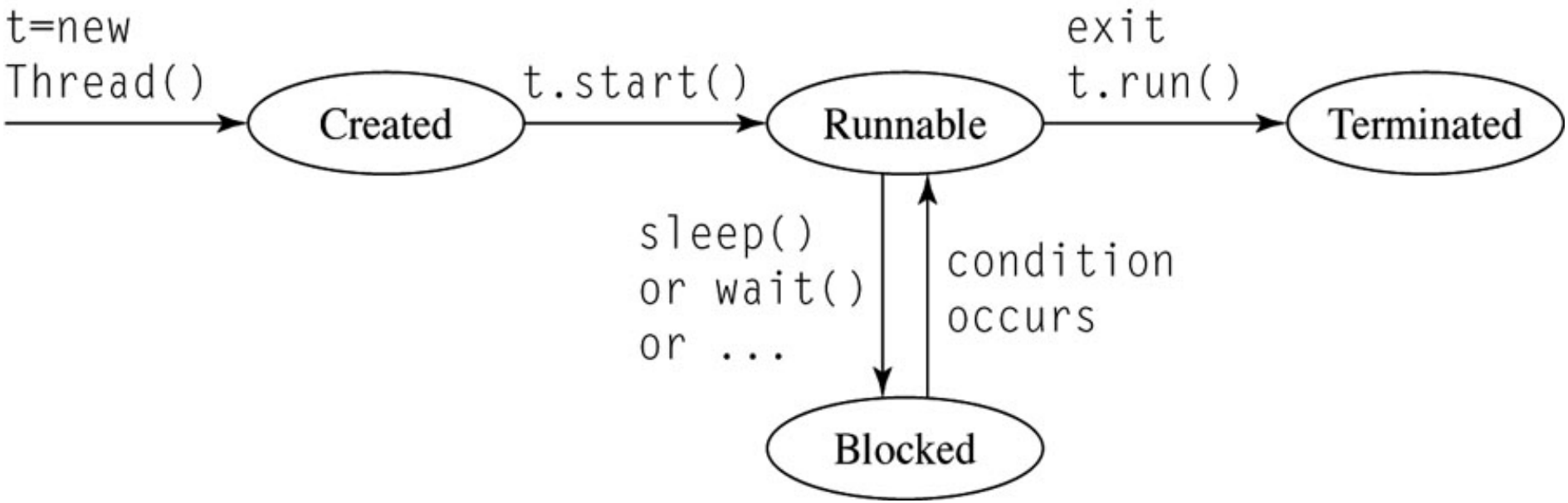
```
class myThread extends Thread
public void run () {...}
}
...
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

The `Thread` class has several methods to control the execution of threads

- The `yield` is a request from the running thread to voluntarily surrender the processor
- The `sleep` method can be used by the caller of the method to block the thread
- The `join` method is used to force a method to delay its execution until the run method of another thread has completed its execution

States of a Java Thread



Java Example: Bouncing Balls

State of a ball in motion:

- Location: x, y coordinates
- Direction and velocity: dx, dy
- Size in pixels
- Color (fun!)

Java Example: Bouncing Balls

State of a ball in motion:

- Location: x, y coordinates
- Direction and velocity: dx, dy
- Size in pixels
- Color (fun!)

Ball methods:

- Constructor
- move: one step (delta)
- paint

Java Ball Example

```
public class Ball {

    Color color = Color.red;
    int x;
    int y;
    int diameter = 10;
    int dx = 3;
    int dy = 6;
```

Java Ball Example

```
public Ball (int ix, int iy) {
      super( );
      x = ix;
      y = iy;
      color = new Color(x % 256, y % 256,
                        (x+y) % 256);
      dx = x % 10 + 1;
      dy = y % 10 + 1;
}
```

Java Ball Example

```
public void move  () {
      if (x < 0 || x >= BouncingBalls.width)
         dx = - dx;
      if (y < 0 || y >= BouncingBalls.height)
         dy = - dy;
      x += dx;
      y += dy;
}

public void paint (Graphics g) {
      g.setColor(color);
      g.fillOval(x, y, diameter, diameter);
 }
```

Java Ball Example

```
public class BouncingBalls extends JPanel {
    public final static int width = 500;
    public final static int height = 400;
    private Ball ball = new Ball(128, 127);
    private Vector<Ball> list = new Vector();
```

Java Ball Example

```
public BouncingBalls ( ) {
        setPreferredSize(new
                Dimension(width, height));
        list.add(ball);
        addMouseListener(new MouseHandler());
        BallThread bt = new BallThread();
        bt.start( );
    }
```

Java Ball Example

```
private class MouseHandler
                    extends MouseAdapter  {
        public void mousePressed(MouseEvent e) {
            Ball b = new Ball(e.getX(), e.getY());
            list.add(b);
        }  // mousePressed
    } // MouseHandler
```

Java Ball Example

```
private class BallThread extends Thread {
        public boolean cont;
        public void run( ) {
            cont = true;
            while (cont) {
                for (Ball b : list) {
                    b.move();
                }
                repaint( );
                try { Thread.sleep(50);
                } catch
                  (InterruptedException exc) { }
            }
}}
```

Java Ball Example

```
public synchronized void
                    paintChildren(Graphics g) {
        for (Ball b : list) {
            b.paint(g);
        }
    }
```

Java Ball Example

```
public static void main(String[] args) {
        JFrame frame = new
                JFrame("Bouncing Balls");
        frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(
                        new BouncingBalls( ));
        frame.setLocation(50, 50);
        frame.pack();
        frame.setVisible(true);
}
```

C# Threads

Loosely based on Java but there are significant differences

Basic thread operations

- Any method can run in its own thread
- A thread is created by creating a Thread object
- Creating a thread does not start its concurrent execution; it must be requested through the Start method
- A thread can be made to wait for another thread to finish with Join
- A thread can be suspended with Sleep
- A thread can be terminated with Abort

Synchronizing Threads

Three ways to synchronize C# threads

The Interlocked class

- Used when the only operations that need to be synchronized are incrementing or decrementing of an integer

The lock statement:

- Used to mark a critical section of code in a thread:

```
lock (expression){...}
```

The Monitor class:

- Provides four methods that can be used to provide more sophisticated synchronization

C#'s Concurrency Evaluation

An advance over Java threads, e.g., any method can run its own thread

Thread termination is cleaner than in Java

Synchronization is more sophisticated

Summary

Concurrent execution can be at the instruction, statement, or subprogram level

Physical concurrency: when multiple processors are used to execute concurrent units

Logical concurrency: concurrent united are executed on a single processor

Two primary facilities to support subprogram concurrency: competition synchronization and cooperation synchronization

Mechanisms: semaphores, monitors, rendezvous, threads

High-Performance Fortran provides statements for specifying how data is to be distributed over the memory units connected to multiple processors