DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING
CONCORDIA UNIVERSITY

Multithreading
(Java, C#, C++)

by Emil Vassev & Joey Paquet

2007, 2009

Outline

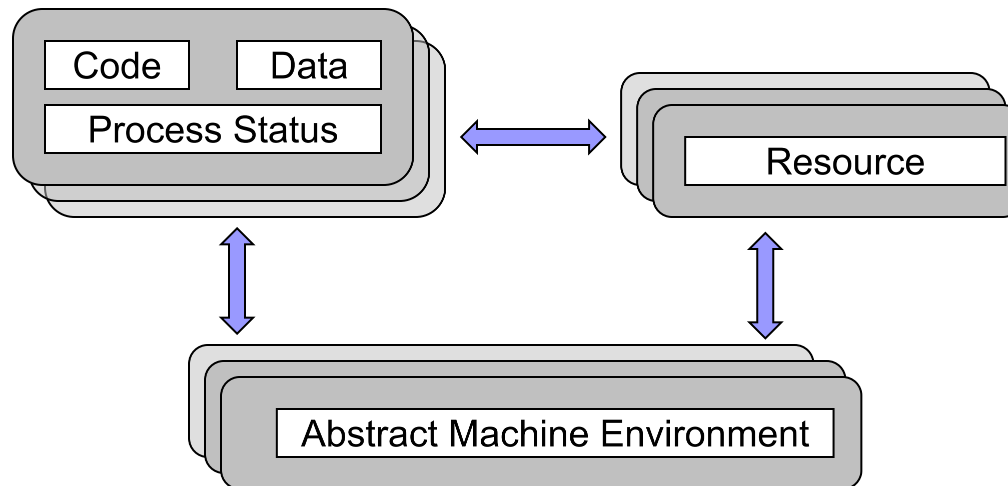
- Process versus Thread
- Synchronization
- Multithreading with Java
- Multithreading with C#
- Multithreading with C++

Process versus Thread.

Synchronization

Process Model

- A **process** is a sequential program in execution.
- A **process** is a unit of computation.
- **Process components:**
  - The program (code) to be executed.
  - The data on which the program will execute.
  - Resources required by the program.
  - The status of the process execution.
- **A process runs in an abstract machine environment (could be OS) that manages the sharing and isolation of resources among the community of processes.**



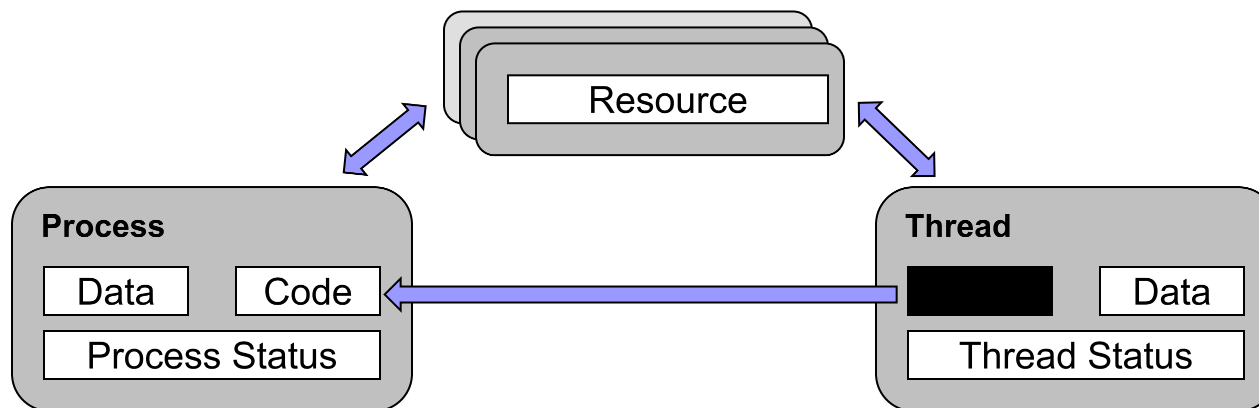
Program and Process

Program and process – distinction?

- A program is a **static entity** made up of program statements. The latter define the run-time behavior.
- A process is a **dynamic entity** that executes a program on a particular set of data.
- **Two or more processes could execute the same program**, each using their own data and resources.

Thread Model

- **A thread** is an alternative form (to the process) of schedulable unit of computation.
- In the **thread model:**
  - Each thread is associated with a process.
  - A thread is an entity that executes by **relying on the code and resources, holding by the associated process**.
  - Several threads could be associated with a single process. Those threads share the code and resources of the process.
  - A thread allocates part of the process's resources for its needs.
  - A thread has its own data and status.

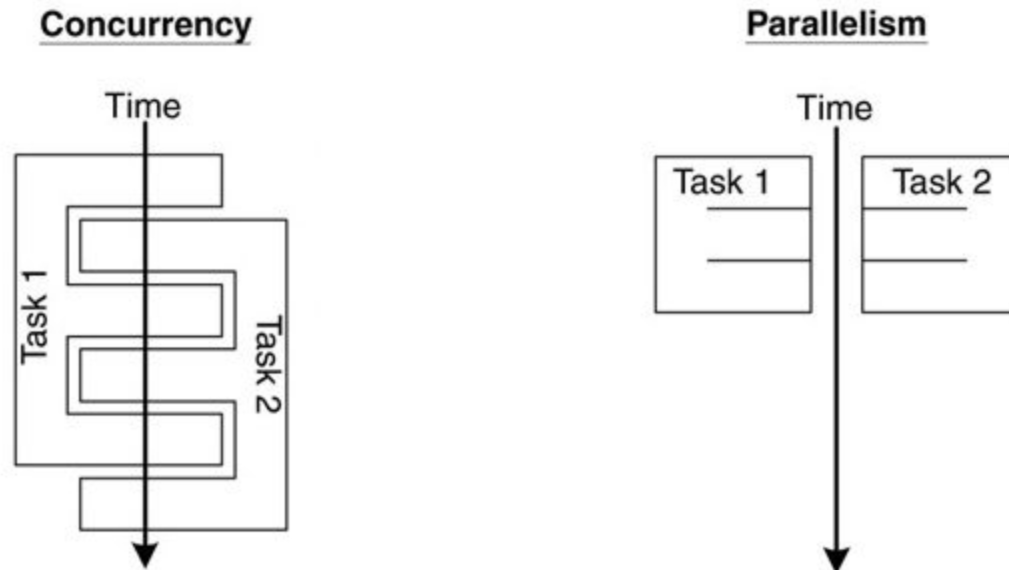


Thread Model

- Control in a normal program usually follows a ***single thread of execution***.
- What differentiates threads from normal processes is the **shared memory** (objects), which is **visible** to all threads in a multi-threaded program.
- A thread **has much less overhead** than a process so is sometimes called as ***light-weight process***.
- **Multithreading** allows **an application to have multiple** threads of execution running **concurrently**.

Concurrency and Parallelism

- Concurrent multithreading systems **give the appearance of several tasks executing at once**, but these tasks are actually split up into chunks that share the processor with chunks from other tasks.
- In parallel systems, two tasks are actually performed simultaneously. **Parallelism requires a multi-CPU system.**

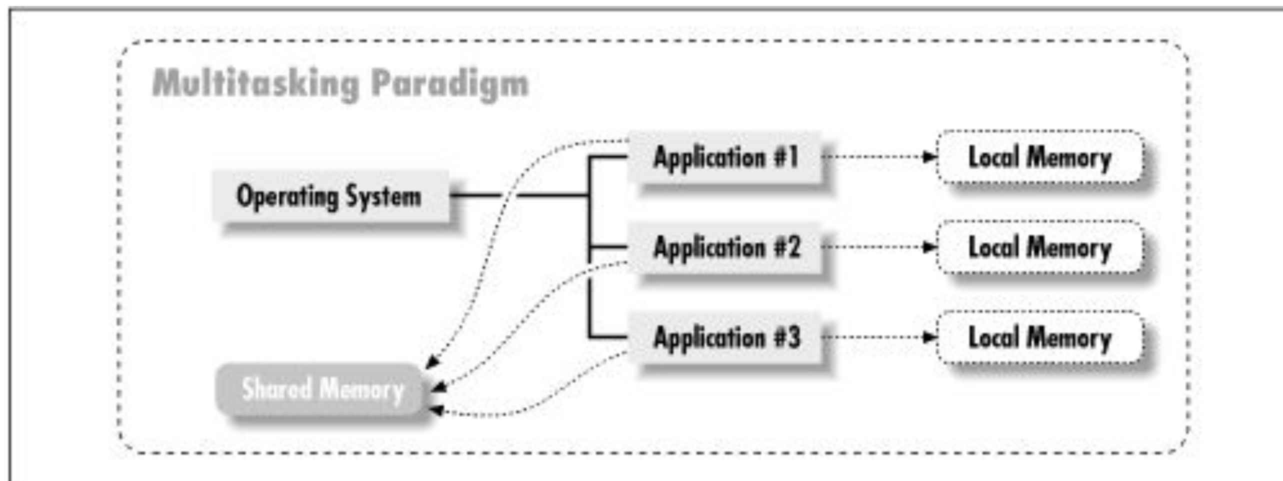


Multitasking

Multitasking operating systems **run multiple programs simultaneously**.

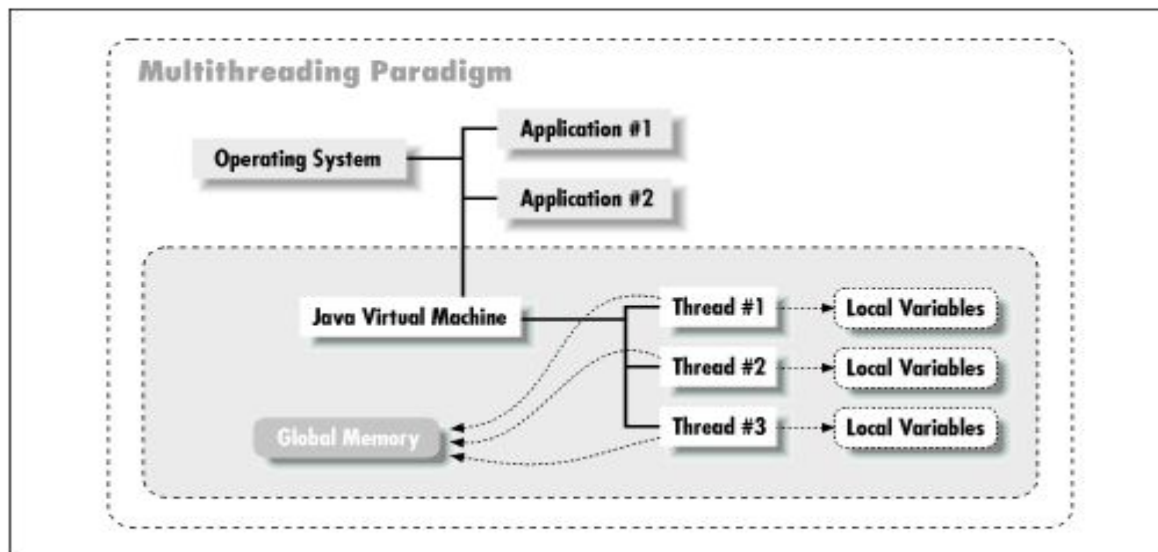
Each of these programs has at least one thread within it - **single-threaded** process:

- **The process begins execution** at a **well-known** point. In Java, C# or C++, the process begins execution at the first statement of the function called ***main()***.
- **Execution of the statements** follows in a completely **ordered, predefined sequence** for a given set of inputs.
- While executing, the process has access to certain data – local, global, static etc.



Multithreading

- **A program with multiple threads** running within a single instance **could be considered** as a multitasking system within an OS.
- In a multithreading program, threads have the following properties:
  - A thread begin execution at a predefined, well-known location. For one of the threads in the program, that location is the ***main()*** method; for the rest of the threads, it is a **particular location** the programmer decides on when the code is written.
  - A thread executes code in an **ordered, predefined sequence**.
  - A thread executes its code **independently** of the other threads.
  - The threads appear to have a **certain degree of simultaneous execution**.



Threading Models

There are typically two threading models supported by OS:

- **Cooperative Threading Model;**
- **Preemptive Threading Model.**

Cooperative Threading Model

- In a **cooperative** system, a thread **retains control** of the processor until it decides to give it up (**which might be never**).
- Supporting OS – Windows 3.x, Solaris, Mac OS.
- The various threads **have to cooperate** with each other. If not, some of them **will be starving** (never given a chance to run).
- Scheduling in most cooperative systems is done strictly by priority level - when the current thread gives up control, the highest-priority waiting thread gets control.

Preemptive Threading Model

- In a **preemptive** system, some **sort of timer** is used by the operating system itself to cause a context swap.
- Supporting OS – Windows 9x, XP, NT (2000), Solaris, Linux.
- When the timer "ticks" the OS can abruptly take control away from the running thread and give control to another thread.
- The interval between timer ticks is called a **time slice**.
- To get to concurrency, the OS must do the **thread scheduling**.
- Preemptive systems are **less efficient** than cooperative ones because the **thread management must be done by the OS' kernel**, but they are **easier to program** (except their synchronization).

Synchronization

Background

- **Concurrent access** to shared data may result in **data inconsistency**.
- Maintaining data consistency requires mechanisms to ensure the **orderly execution** of cooperating processes (or threads).

When do we need synchronization?

When two or more processes (or threads) work on the same data simultaneously.

Synchronization

Example:

Two threads are trying to **update the same shared variable simultaneously:**

- The result is **unpredictable**.
- The result depends on which of the two threads was the last one to change the value.
- The competition of the threads for the variable is called **race condition.**
- The **first** thread is the one who **wins the race to update** the variable.

Classical Synchronization Problems

Mutual exclusion

- Only one process executes a piece of code (**critical section**) at any time.
- **OS examples:** access to shared resources, e.g., a printer.

Sequencing

- A process waits for another process to finish executing some code.
- OS examples: waiting for an event, e.g., **ls (dir)** command suspends until there is some data to read from the file system.

Bounded-buffer
(also referred to as the Producer-Consumer problem)

- A pool of n buffers.
- **Producer** processes put items into the pool.
- **Consumer** processes take items out of the pool.
- Issues: **mutual exclusion, empty pool, and full pool.**
- OS examples: buffering for pipes, file caches, etc.

Readers-Writers

- Multiple processes access a **shared data object X**.
- Any number of **readers** can access X at the same time.
- No **writer** can access it at the same time as a **reader** or another **writer**.
- Mutual exclusion is too constraining. **Why?**
- Variations:
  - reader-priority: a reader must not wait for a writer;
  - writer-priority: a writer must not wait for a reader,
- OS examples: file locks.

Classical Synchronization Problems

Dining Philosophers

- 5 philosophers with 5 chopsticks placed between them.
- To eat requires two chopsticks.
- Philosophers alternate between thinking and eating.
- OS examples: simultaneous use of multiple resources.

Many examples, along with Java code

- http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency.html

The Critical Section Problem

Definition:

A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

Conditions:

- n processes (or threads) all competing to use some shared data.
- Each process has a code segment, called **critical section**, in which the shared data is accessed.

Problem:

How to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section?

The Critical Section Problem - Example

Suppose that two processes are trying to increment the same variable. They both execute the statement

$$x := x + 1;$$

To execute this statement each process **reads the variable x, then adds one to the value, then write it back.**

Suppose the value of **x** is 3.

- If both processes read x at the same time then they would get the same value 3.
- If they then both added 1 to it then they would both have the value 4.
- They would then both write 4 back to x.
- The result is that both processes incremented x, but its value is only 4, instead of 5.

The Critical Section Problem

Solution – three requirements:

- Only one process is allowed to be in its critical section at a time. Hence, the execution of critical sections is **mutually exclusive**.

- If there is no process in its critical section, but some processes are waiting to enter their critical sections, only the waiting processes may compete for getting in. Ultimately, there must be **progress** in the resolution and one process must be allowed to enter.

- Processes waiting to enter their critical sections must be allowed to do so in a **bounded timeframe**. Hence, processes have **bounded waiting**.

The Critical Section Problem

Critical sections are **General Framework** for process (thread) synchronization:

ENTRY SECTION

CRITICAL SECTION CODE

EXIT SECTION

- The **ENTRY SECTION** controls access to make sure no more than one process Pi gets to access the critical section at any given time. It acts as a *guard*.
- The **EXIT SECTION** does bookkeeping to make sure that other processes that are waiting know that Pi has exited.

Semaphores

- The Semaphores are a **solution** to the Critical Section Problem.
- Help in making the Critical Section **atomic**.

A semaphores is:

- a single integer variable S;
- accessed via two atomic operations:
  - **WAIT** (sometimes denoted by **P**)

    while S <= 0 do wait();

    S := S-1;
  - **SIGNAL** (sometimes denoted by **V**)

    S := S+1;
  - wake up a waiting process (if any);
- WAITing processes cannot "lock out" a SIGNALing process.

Mutual Exclusion Semaphore

```
//**** initially S = 1

P( S )  //**** WAIT
CRITICAL SECTION
V( S )  //**** SIGNAL
```

Binary semaphores - S is restricted to take on only the values 0 and 1.

Multithreading with Java

Threads in Java

- There are two ways to create a java thread:
  - By extending the **java.lang.Thread** class.
  - By implementing the **java.lang.Runnable** interface.
- The ***run()*** method is where the action of a thread takes place.
- The execution of a thread starts by calling its ***start()*** method.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
      this.minPrime = minPrime; }
    public void run() {
// compute primes larger than minPrime  . . .
    }
}
```

- The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Implementing the Runnable Interface

- In order to create a new thread we may also provide a class that implements the **java.lang.Runnable** interface.
- Preferred way in case our class has to subclass some other class.
- A Runnable object can be wrapped up into a Thread object:
  - **Thread**(Runnable target)
  - **Thread**(Runnable target, String name)
- The thread's logic is included inside the **run()** method of the **runnable** object.

```
class ExClass
  extends ExSupClass
  implements Runnable {
    …
    public ExClass (String name) {
    }
    public void run() {
        …
    }
}
```

```
class A {
    …
    main(String[] args) {
        …
        Thread mt1 = new Thread(new ExClass("thread1”));
        Thread mt2 = new Thread(new ExClass("thread2”));
        mt1.start();
        mt2.start();
    }
}
```

Implementing the Runnable Interface

- Constructs a new thread object associated with the given ***Runnable*** object.
- The new Thread object's ***start()*** method is called to begin execution of the new thread of control.
- The reason we need to pass the runnable object to the thread object's constructor is that the thread must have some way to get to the ***run()*** method we want the thread to execute. Since we are no longer overriding the ***run()*** method of the ***Thread*** class, the default ***run()*** method of the ***Thread*** class is executed:

```
public void run() {
        if (target != null) {
                target.run();
        }
}
```

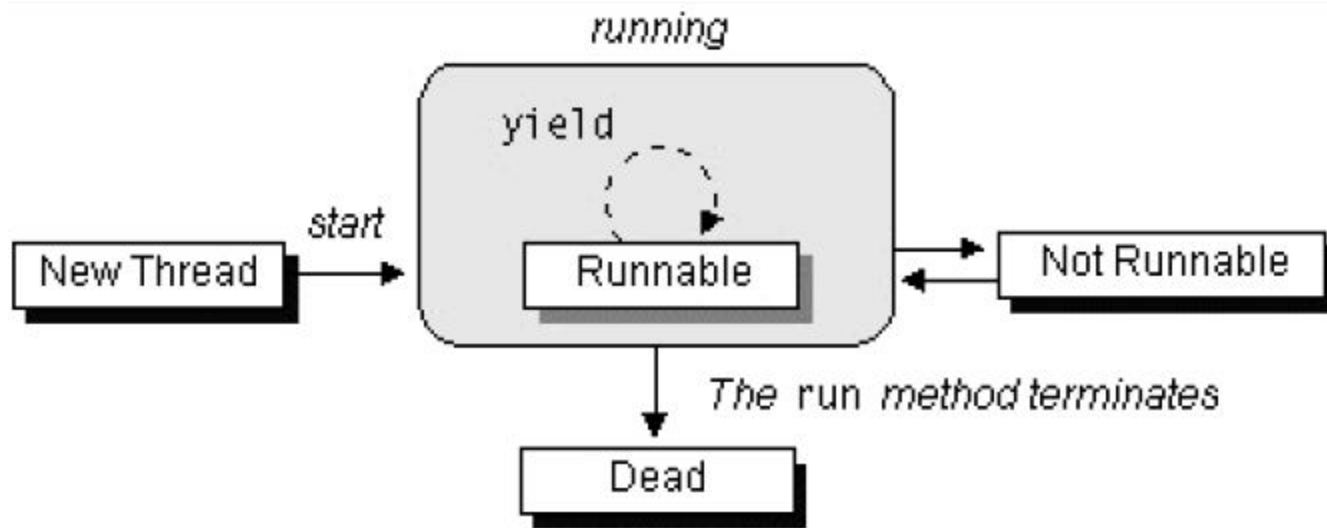
- Here, target is the runnable object we passed to the thread's constructor. So the thread begins execution with the ***run()*** method of the ***Thread*** class, which immediately calls the ***run()*** method of our runnable object.

Sleep, Yield, Notify & Wait Thread's Functions

- ***sleep(long millis)*** - causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- ***yield()*** - causes the currently executing thread object to temporarily pause and allow other threads to execute.
- ***wait()*** - causes current thread to wait for a condition to occur (*another thread invokes the **notify()** method or the **notifyAll()** method for this object*). This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notify()*** - notifies a thread that is waiting for a condition that the condition has occurred. This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notifyAll()*** – like the ***notify()*** method, but notifies all the threads that are waiting for a condition that the condition has occurred.

The Lifecycle of a Thread

- The ***start()*** method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's ***run()*** method.
- A thread becomes **Not Runnable** when one of these events occurs:
  - Its ***sleep()*** method is invoked.
  - The thread calls the ***wait()*** method.
  - The thread is blocked on I/O operations.
- A thread dies naturally when the ***run()*** method exits.



Thread Priority

- On a single CPU, threads actually run one at a time in such a way as to provide an **illusion of concurrency**.
- Execution of multiple threads on a single CPU, in some order, is called **scheduling**.
- The Java runtime supports a very simple **scheduling algorithm** (fixed priority scheduling). This algorithm schedules threads based on their priority relative to other runnable threads.
- The runtime system chooses the runnable thread with the **highest** priority for execution.

Thread Priority

- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a **round-robin fashion** - each process is guaranteed to get its turn at the CPU at every system-specified time interval.

- The chosen thread will run until:
  - A higher priority thread becomes runnable.
  - It yields (calls its ***yield()*** method), or its ***run()*** method exits.
  - On systems that support **time-slicing**, its time allotment has elapsed.

- You can modify a thread's priority at any time after its creation by using the ***setPriority()*** method.

Synchronization of Java Threads

- In many cases **concurrently** running threads share data and must consider the state and activities of other threads.
- If two threads can both execute a method that modifies the state of an object then the method should be declared to be ***synchronized***, those allowing only one thread to execute the method at a time.
- If a class has at least one ***synchronized*** method, each instance of it has a **monitor**. A monitor is an object that can **block** threads and **notify** them when the method is available.

Example:

```
public synchronized void updateRecord() {
//**** critical code goes here …
}
```

- Only one thread may be inside the body of this function. A second call will be blocked until the first call returns or ***wait()*** is called inside the synchronized method.

Synchronization of Java Threads

- If you don't need to protect an entire method, you can synchronize on an object:

```
public void foo() {
        synchronized (this) {
        //critical code goes here …
        }
        …
}
```

- There are **two syntactic forms** based on the **synchronized** keyword - blocks and methods.
- **Block synchronization** takes an argument of which object to lock. This allows **any method to lock any object.**
- The most common argument to synchronized blocks is **this.**
- Block synchronization is considered more fundamental than method synchronization.

Applying Synchronization (Example)

Consider the following class:

```
class Even {
        private int n = 0;
        public synchronized int next(){
        ++n;
        ++n;
        return n; //**** next is always even
    }
}
```

Declaring the next method as synchronized would resolve such conflicting problems.

Without synchronizing, the desired postcondition may fail due to a storage conflict when two or more threads execute the next method of the same Even object.

Here is one possible execution trace:

Thread A	Thread B
read 0	
write 1	
	read 1
	write 2
read 2	read 2
	write 3
write 3	return 3
return 3	

Synchronization of Java Threads

- To program the synchronization behavior we use the Object class' methods ***wait(), notify()*** and ***notifyAll().***

- With these methods we allow objects to wait until another object notifies them:

```
synchronized( waitForThis ) {
        try { waitForThis.wait();}
        catch (InterruptedException ie) {}
}
```

- To wait on an object, you must first **synchronize** on it.
- ***InterruptedException*** is thrown when a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt method in class Thread.

Synchronization of Java Threads

- A thread may call ***wait()*** inside a **synchronized** method. A timeout may be provided. If missing or zero then the thread waits until either ***notify()*** or ***notifyAll()*** is called, otherwise until the timeout period expires.

- ***wait()*** is called by the thread owning the lock associated with a particular object.

- ***notify()*** or ***notifyAll()*** are only called from a **synchronized** method. One or all waiting threads are notified, respectively. It's probably better (safer) to use ***notifyAll().*** These methods don't release the lock. The threads awakened will not return from their ***wait()*** call immediately, but only when the thread that called ***notify()*** or ***notifyAll()*** finally relinquishes ownership of the lock.

Synchronization of Java Threads

- The ***wait()*** method releases the lock prior to waiting, and reacquires the lock prior to returning from the ***wait()*** method.

- It is possible a **synchronized method to make a self-call** to another synchronized method on the same object without freezing up.

- <u>**Methods that are not synchronized may still execute at any time, even if a synchronized method is in progress.**</u> In other words, **synchronized is not equivalent to *atomic*, but synchronization can be used to achieve atomicity.**

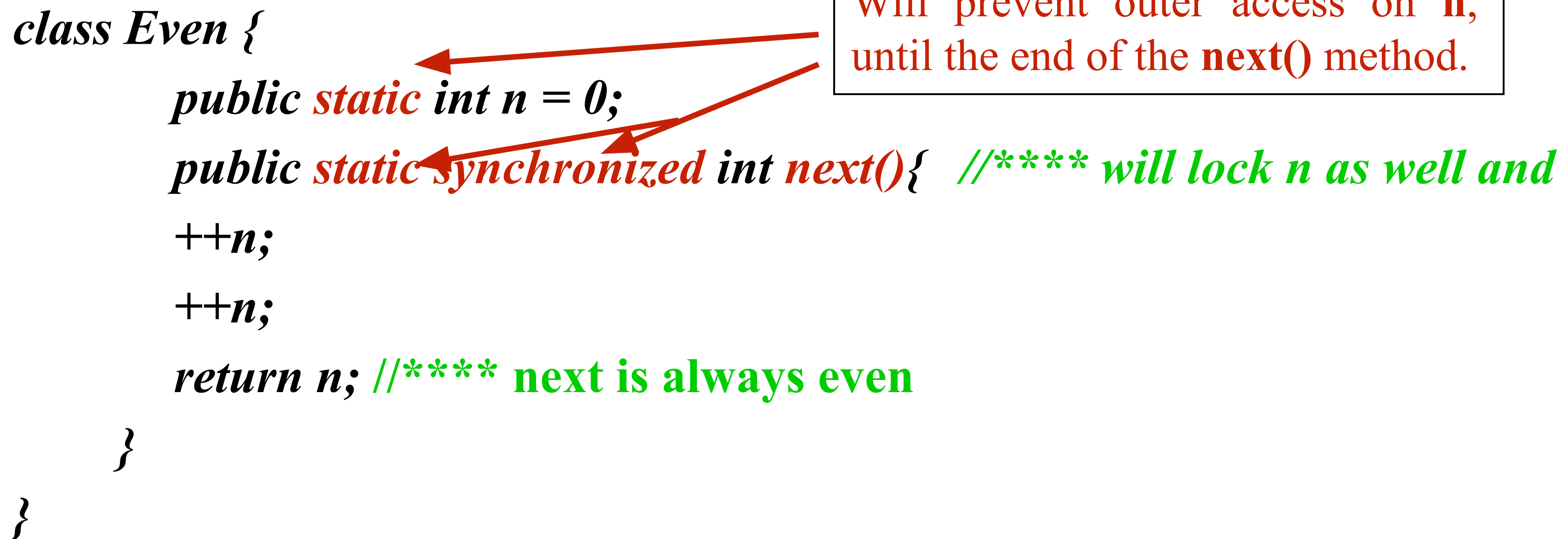
Java Semaphore - Example

```
class Semaphore {
  private int value;
  Semaphore (int value1) {
    value = value1;
  }
  public synchronized void Wait () {
    while( value <= 0 ) {
      try { wait (); }
      catch (InterruptedException e) { };
    }
    value--;
  }

  public synchronized void Signal () {
    ++value;
    notify ();
  }
}
```

Protecting Static Fields

- Locking an object does not automatically protect access to the **static fields** of that object's class or any of its superclasses.
- Access to static fields is instead protected via **static synchronized** methods and blocks.



:: Stopping Threads

- The Thread class does contain a ***stop()*** method that allows you to stop a thread immediately: no matter what the thread is doing, it will be terminated.
- However, the ***stop()*** method is very dangerous. In Java 2, the ***stop()*** method is deprecated.

Why?

- If a thread holds a lock at the time it is stopped, the lock will be released when the thread stops.
- But if the thread that is being stopped is in the middle of updating a linked list, for example, the links in the list will be left in an inconsistent state.
- Hence, if we were able to interrupt a thread in the middle of this operation, we would lose the benefit of its obtaining the lock.
- The reason we needed to obtain a lock on the list in the first place was to ensure that the list would not be found by another thread in an inconsistent state.

Java Threading API

:: The suspend() and resume() Methods

- The ***suspend()*** and ***resume()*** methods are very dangerous and they became deprecated.
- The problem with using the ***suspend()*** method is that it can conceivably lead to cases of lock starvation - including cases where the starvation shuts down the entire virtual machine.
- If a thread is suspended while it is holding a lock, that lock remains held by the suspended thread. As long as that thread is suspended, no other thread can obtain the lock.
- There is no danger in the ***resume()*** method itself, but since the ***resume()*** method is useful only with the ***suspend()*** method, it too has been deprecated.
- Java Thread primitives deprecation: http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Java Threading API

:: Thread Naming

- It is possible to assign a String name to the Thread object itself:

***void setName(String name)* //assigns a name to the Thread instance**

***String getName()* //gets the name of the Thread instance**

- The system does not use this string for any specific purpose.
- We can use it for debugging. With an assigned name, the debugger and the ***toString()*** method display thread information in terms of a “logical" name instead of a number.
- The naming support is also available as a constructor of the Thread class:
  - ***Thread(String name)*** constructs a thread object with a name that is already assigned. This constructor is used when threading by inheritance.
  - ***Thread(Runnable target, String name)*** constructs a thread object that is associated with the given Runnable object and is created with a name that is already assigned. This constructor is used when threading by interfaces.

Java Threading API

:: Thread Access – The currentThread() Method

- ***static Thread currentThread()*** gets the ***Thread*** object that represents the current thread of execution. The method is static and may be called through the ***Thread*** class name.

Why is this method important?

- The ***Thread*** object for the current thread may not be saved anywhere, and even if it is, it may not be accessible to the called method.
- In this code we are assuming that reader threads are threads whose names start with "Reader." This name could have been assigned by the ***setName()*** method earlier or when the threads were constructed.
- To obtain a name, we need simply to call the ***getName()*** method. However, since we do not have the ***Thread*** object reference of the caller, we must call the ***currentThread()*** method to obtain the reference.

```
// Get the string already read from the socket so far.
// Only allows "Reader" threads to execute this method.
public String getResult()
{
    String reader = Thread.currentThread().getName();
    if (reader.startsWith("Reader"))
    {
        String retval = result.toString();
        result = new StringBuffer();
        return retval;
    }
    else
    {
        return "";
    }
}
```

Java Threading API

:: Thread Access – Enumerating Threads in JVM

The ***Thread*** class provides methods that allow you to obtain a list of all the threads in the program:

- ***static int enumerate(Thread threadArray[])*** gets all the thread objects of the program and stores the result into the thread array. The value returned is the number of thread objects stored into the array. The method is static and may be called through the ***Thread*** class name.
- ***static int activeCount()*** returns the number of threads in the program. The method is static and may be called through the ***Thread*** class name.

```
public void printThreads()
{
    Thread ta[] = new Thread[Thread.activeCount()];
    int n = Thread.enumerate(ta);
    for (int i = 0; i < n; i++)
    {
        System.out.println("Thread " + i + " is " +
            ta[i].getName());
    }
}
```

Multithreading with C#

C# Namespace System.Threading

- **Sytem.Threading** is a powerful namespace for:
  - programming Threads in C#;
  - thread Synchronization in C#.

- The most important class inside this namespace for manipulating threads is the class **Sytem.Threading**.**Thread**.
  - It can run other thread in our application process.

- Threads in C# does not require a *run()* method;
- A thread in C# is not considered as an object;
- C# provides similar to Java set of primitives for operating on threads.

Java versus C#

Java Code

```
public class ThreadingExample
{
    public static void main( String args[] )
    {
        Thread[] threads = new Thread[2];

        for( int count=1;count<threads.length;count++ )
        {
            threads[count] = new Thread( new Runnable()
            {
                public void run()
                {
                    count();
                }

            } );
            threads[count].start();
        }
    }

    public static void count()
    {
        for( int count=1;count<=5;count++ )
            System.out.print( count + " " );

    }
}
```

C# Code

```
using System.Threading;

public class ThreadingExample : Object {

    public static void Main()
    {
        Thread[] threads = new Thread[2];
        for( int count=1;count<threads.Length;count++ )
        {
            threads[count] = new Thread( new ThreadStart( Count ) );
            threads[count].Start();

        }
    }

    public static void Count()
    {
        for( int count=1;count<=5;count++ )
            Console.Write( count + " " );

    }
}
```

Java's java.lang.Thread – C#'s System.Threading.Thread

Java	C#
setDaemon(boolean on) method	*IsBackground* set property
isDaemon() method	*IsBackground* get property
isAlive() method	*IsAlive* get property
yield() method	*Interrupt()* method
isInterrupted() method	n/a
sleep(long millis) and *sleep(long millis, int nanos)*	*Sleep(int millisecondTimeout)* and *Sleep(System.TimeSpan)* methods
join(), and *join(long millis),* and *join(long millis, int nanos)* methods	*Join(), Join(int millisecondTimeout),* and *Join(System.TimeSpan)* methods
suspend() method	*Suspend()* method
resume() method	*Resume()* method
stop() method	*Abort()* method

Thread Synchronization

Java	C#
synchronized	*lock*
Object.wait() method	*Monitor.Wait(object obj)* method
Object.notify() method	*Monitor.Pulse(object obj)* method
Object.notifyAll() method	*Monitor.PulseAll(object obj)* method

- In addition to the lock construct, C# has provided access to its internal methods to acquire and release locks:
  - *Monitor.Enter(object obj);*
  - *Monitor.Exit(object obj).*

- Using these methods can buy a programmer the same benefits as using the lock construct, but it can also provide more elaborate locking abilities, such as being able to **lock variables** in one method and **have them released at different times** or different points in the code, **depending on the code path**.

Example: Thread Synchronization

```
using System;
using System.Threading;

namespace ThreadLauncher
{
    class Launcher {
        public Launcher(){}
        public void Coundown() {
            lock(this) {
                for(int i=4;i>=0;i--) {
                    Console.WriteLine("{0} seconds to start",i);
                }
                Console.WriteLine("GO!!!!!");
            }
        }
    }

    class Test {
        [STAThread]
        static void Main(string[] args) {
            Launcher la = new Launcher();

            Thread firstThread = new Thread(new ThreadStart(la.Coundown));
            Thread secondThread = new Thread(new ThreadStart(la.Coundown));
            Thread thirdThread = new Thread(new ThreadStart(la.Coundown));

            firstThread.Start();
            secondThread.Start();
            thirdThread.Start();
        }
    }
}
```

Multithreading with C++

C++ Has No Build-in Multithreading

- **C++ does not contain any built-in support for multithreaded applications**. Instead, it relies entirely upon the operating system to provide this feature.

- Using operating system functions to support multithreading gives you access to the full range of control offered by the execution environment.

- Consider Windows. It defines a rich set of thread-related functions that enable finely grained control over the creation and management of a thread.

  Example: Windows has several ways to control access to a shared resource - **semaphores**, **mutexes**, **event objects**, **waitable timers**, and **critical sections**.

Windows Thread Functions - CreateThread

- Windows offers a wide array of Application Programming Interface (API) functions that support multithreading.
- To use Windows' multithreading functions, you must include ***<windows.h>*** in your program.
- To create a thread, use the Windows API ***CreateThread()*** function. Its prototype is shown here:

```
HANDLE CreateThread(
        LPSECURITY_ATTRIBUTES secAttr,
                SIZE_T stackSize,
                LPTHREAD_START_ROUTINE threadFunc,
                LPVOID param,
                DWORD flags,
                LPDWORD threadID);
```

Windows Thread Functions - CreateThread

- ***secAttr*** - a pointer to a set of security attributes pertaining to the thread. If ***secAttr*** is NULL, then the default security descriptor is used.
- Each thread has its own stack – the ***stackSize*** parameter. If this integer value is zero, then the thread will be given a stack that is the same size as the creating thread.
- Each thread of execution begins with a call to a function, called the ***thread function***, within the creating process (like in C#).
- Execution of the thread continues until the thread function returns.
- The address of this function (that is, the entry point to the thread) is specified in ***threadFunc***.

DWORD WINAPI threadfunc(LPVOID param);

Windows Thread Functions - CreateThread

- ***param*** – specifies any argument that you need to pass to the new thread.
- ***flags*** - determines the execution state of the thread:
  - If it is zero, the thread begins execution immediately.
  - If it is CREATE_SUSPEND, the thread is created in a suspended state, awaiting execution.
  - It may be started using a call to ***ResumeThread().***
- ***threadID*** - the identifier associated with a thread is returned in this long integer pointer.
- The function returns a handle to the ***thread*** if successful or NULL if a failure occurs.
- The thread handle can be destroyed:
  - manually by calling ***CloseHandle();***
  - automatically when the parent process ends.

Windows Thread Functions – TerminateThread

- A thread terminates when its entry function returns.
- We can also terminate threads manually:
  - ***TerminateThread( );***
  - ***ExitThread( );***

BOOL TerminateThread(HANDLE thread, DWORD status);
VOID ExitThread(DWORD status);

- ***thread*** - the handle of the thread to be terminated.
- ***status*** - the termination status.
- ***ExitThread()*** - terminates the thread that calls ***ExitThread().***
- ***TerminateThread()*** returns nonzero if successful and zero otherwise.

Visual C++ Threading Model

- The Visual C++ alternatives to ***CreateThread()*** and ***ExitThread()*** are listed below. Both require the header file ***<process.h>.***
  - ***_beginthreadex( );***
  - ***_endthreadex( );***

```
uintptr_t _beginthreadex(
   void *secAttr,
   unsigned stackSize,
   unsigned (__stdcall *threadFunc)(void *),
   void *param,
   unsigned flags,
   unsigned *threadID);

void _endthreadex(unsigned status);
```

Suspending and Resuming Threads

- A thread of execution can be suspended by calling ***SuspendThread().***
- It can be resumed by calling ***ResumeThread().***

DWORD SuspendThread(HANDLE hThread);

DWORD ResumeThread(HANDLE hThread);

Windows Synchronization Objects

- ***classic semaphore*** - when using a semaphore, the access to a resource can be completely synchronized.
- ***mutex semaphore*** (***mutex***) - synchronizes a resource such that **one and only one** thread or process can access it at any one time.
- ***event object*** - can be used to block access to a resource until some other thread or process signals that it can be used. An event object signals that a specified event has occurred.
- ***waitable timer*** - blocks a thread's execution until a specific time.
- ***timer queues -*** lists of timers.
- ***critical section*** - prevents a section of code from being used by more than one thread at a time.

Using Mutex

- ***CreateMutex()*** – creates a mutex object.

```
HANDLE CreateMutex(
   LPSECURITY_ATTRIBUTES secAttr,
   BOOL acquire,
   LPCSTR name);
```

- Once you have created a semaphore, you use it by calling two related functions: ***WaitForSingleObject()*** and ***ReleaseMutex().***
- To use a ***mutex*** to control access to a shared resource, wrap the code that accesses that resource between a call to ***WaitForSingleObject()*** and ***ReleaseMutex().***

```
If (WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
 { //**** handle time-out error }
 //**** access the resource
ReleaseMutex(hMutex);
```

References

- Scott Oaks and Henry Wong, “Java Threads”, 2nd edition, O’Reilly
- Bruce Eckel, "*Thinking in Java*", 3d edition
- Sun Microsystems, “The Java Tutorial, http://java.sun.com/docs/books/tutorial/essential/threads/
- Sun Microsystems, JavaTM 2 Platform, Standard Edition, v 1.3.1 API Specification, http://java.sun.com/j2se/1.3/docs/api/overview-summary.html
- Mike Gold October , Introduction to Multithreading in C#, C# Corner, June 2005
- Multithreading in C++, Contributed by McGraw-Hill/Osborne, http://www.devarticles.com/c/a/Cplusplus/Multithreading-in-C/