Multithreading - Synchronization

Overview

- Threads – Scheduling
- Synchronization
- Producer/consumer as communication with threads
- Deadlock

Threads – Scheduling

- Scheduler
  - Determines which *runnable* threads to run
  - Can be based on thread priority
  - Part of OS or Java Virtual Machine (JVM)
  - Many computers can run multiple threads simultaneously (or nearly so)

- Order thread selected is indeterminate
  - Depends on scheduler, timing, chance
- Scheduling is not guaranteed to be fair
- Some schedules/interleavings can cause unexpected and bad behaviors

Data Race

- Definition
    - Concurrent accesses to same shared variable, where at least one access is a write
        - variable isn’t volatile
- Can expose all sorts of really strange stuff the compiler and processor are doing to improve performance

Synchronization

- Synchronization
  - can be used to control thread execution order
- Uses
  - Marks when a block of code must not be interleaved with code executed by another thread
  - Marks when information can/must flow between threads
- Notes
  - Incurs a small amount of runtime overhead
    - if only used where you might need to communicate between threads, not significant used everywhere

Lock (mutex)

- Definition
  - Each object in a JVM has this *lock* (or *mutex*- ***mut**ual **ex**clusion*) that any program can use to coordinate multi-threaded access to the object. If any thread want to access instance attributes of that object; then thread must "own" the object's *lock* (*set some flag* in lock memory area). All other threads that attempt to access the object's attributes must *wait* until the owning thread releases the object's *lock* (*unset the flag*).
  - Entity can be held by only one thread at a time. *Locks* provide necessary support for implementing *monitors*.
- Properties
  - A type of synchronization
  - Used to enforce mutual exclusion
  - Thread can acquire/release locks
  - Please note that *lock* is acquired by a thread, when it explicitly ask for it. In Java, this is done with the *synchronized* keyword for methods or instruction (block) as a *mutual exclusion* process.
  - For *co-operation* are used *wait* and *notify* methods from *Object* class.
  - Thread would wait to acquire *lock* (stop execution), if lock held by another thread.

- The **multithreading** is usual an **asynchronous** process.
- Sometimes two or more threads must be executed in a simultaneous mode.
- The **synchronization** in Java is realized with *synchronized methods* and/or *instruction*.
- A thread that will execute a **synchronized method** will *not allow other threads* to **call** other/same *synchronized method/s* of the *same object*.
- A ***monitor*** is an object that provides that a shared resource to be accessed at a moment, is realized by not more than one thread.
- A thread will use a ***monitor*** for a ***limited time***.
- The ***monitor*** is named also a ***semaphore***, and the programmers consider that the thread keep the ***monitor*** for that time.
- If the ***monitor*** is not available, the thread will be suspended being in a *waiting* state.
- ***Monitor*** is a synchronization construct that allows threads to have both ***mutual exclusion*** (using *lock*s) and ***cooperation*** (Inter-thread communication or co-operation) i.e. the ability to make threads *wait* for certain condition to be true (using *wait*-set).

- The ***monitor*** is a control mechanism first defined by C.A.R. Hoare.
- You can think of a *monitor* as a very small box that can hold only one thread.
- Once a *thread enters a monitor*, all other threads must *wait* until that thread *exits* the monitor.
- In this way, a *monitor* can be used to *protect* a shared asset from being manipulated by more than one thread at a time.
- This mechanism is only a ***mutual exclusion*** process. Only one thread will win the competition and own the *lock*. There is no role of *wait-set* feature.
- In general, *mutual exclusion* is important only when multiple threads are sharing *data* or some *other resource*. If two threads are not working with any common data or resource, they usually can’t interfere with each other and needn’t execute in a mutually exclusive way.
- **Co-operation** is important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state e.g., **producer/consumer problem** where read thread needs the buffer to be in a “not empty” state before it can read any data out of the buffer. If the read thread discovers that the buffer is empty, it must *wait*. The write thread is responsible for filling the buffer with data. Once the write thread has done some more writing, the read thread can do some more reading. It is also sometimes called a “**Wait and Notify**” process.
- This *co-operation* requires both ***entry-set*** and ***wait-set***, and in the middle is the ***owner***, *one thread* processed. All these 3 elements with *mutual exclusion* will define the ***monitor***.

- In Java all objects have an ***own associated implicit lock (mutex)*** for mutual exclusion

-***wait( )*** method pas a thread from running in blocked state,

notify( ) and ***notifyAll( )*** methods ***reactivated*** the threads from a blocked (*wait)* state.

-The methods belongs to the ***Object*** *class*

-The methods are **able to be called only** in *synchronized* methods/blocks and will allow *co-operation*

-These methods are *different from* the ***deprecated*** *suspend( ), resume( )* or *sleep( )* and *yield( ).*

- The management of the threads could be:
  - non-preemptive
  - preemptive,

  Non-Preemptive threading model: Once a thread is started it cannot be stopped or the control cannot be transferred to other threads until the thread has completed its task (nowadays it is not so used).

  Preemptive Threading Model: The runtime can step in and hand control from one thread to another at any time. Higher priority threads are given precedence over Lower priority threads (are on the second position).

- In ***preemptive*** mode a thread will leave the running mode without using a *wait( )* or *suspend( )* method when:

  - an operation as I/O block the thread

  - CPU executes a more priority thread

- In ***preemptive time slicing***, a thread will be in a running state for a specific time, being passed in a blocked/ready state concerning the priority.
- ***Starvation*** means that it is not possible that a thread with a *lower priority* than *another thread*, will be in a running state
- This is an **indeterminist process** so that it is possible that at different executions for the same input data to have different outputs

- Now, machines are **preemptive** (**Solaris** have a different management than, **Mac OS**, and **Windows** from Vista/NT that are preemptive Round Robin machines at the end of time slicing)
- **Scheduling Algorithms:**

-First Come First Serve Scheduling

-Shortest Job First Scheduling

-Priority Scheduling

-Round-Robin Scheduling

-Multilevel Queue Scheduling

-Multilevel Feedback-Queue Scheduling

- **Other Thread Synchronization elements**:

- deadlock -> the situation in which two or more execution threads expect each other indefinitely;

- starvation -> the situation in which an execution thread expects (without result) access to shared resources;

-> may occur for the following reasons:

* Threads are locked endlessly because a thread takes too long to execute a synchronized code sequence (eg. input / output operations);

* A thread does not receive runtime from the processor because it has too little priority compared to other threads;

- ***livelock* ->** the situation in which one thread reacts to another thread (the threads do not progress due to their failing execution – mutual disposal);

-> unlike the deadlock state, in this situation the treads will not be blocked;

-> example: 2 people who want to go through the same narrow corridor simultaneously;

- ***thread pool -***

-> used to start a lot of short-term threads to efficiently use resources and thus increase performance;

-> keeps a number of inactive threads ready for the execution of the required tasks (after a thread finishes executing a task, it remains inactive in the pool and waits to be selected to execute new tasks);

-> you can define a limited number of threads in the pool, useful to prevent overloading;

-Types of thread pool-s:

***Cached thread pool*:** keep a number of active threads and creates new ones if needed;

***Fixed thread pool*:** limits the maximum number of competing threads. Additional tasks are placed in a queue;

***Single-threaded pool*:** retains only one thread that executes a task at a time;

***Fork/join pool*:** a special thread pool that uses the Fork/Join framework to use the benefits of the processor that splits a more complex task into several smaller parts in a recursive way;

Synchronized methods

- **Synchronized (*sync*) methods** allow *mutual exclusion*.
- If **2 threads** tries to *access same attribute/method* (shared) the *sync* methods will *not allow a simultaneous* access.
- The **object** that contains the attribute/method being accessed by a *sync* method *will block other threads* to access the same attribute/method.
- If the *sync* method belongs to ***an instance of the class***, the associated ***lock*** will be activated (using **current-*this* object**).
- If the *sync* method is a ***static*** one, the ***lock*** will be associated to the ***class***.

- To understand the need for synchronization, let’s begin with a simple example that ***does not use it***—but should. The following program has three simple classes.

- The first one, ***Callme***, has a single method named ***call( )***. The ***call( )*** method takes a ***String*** parameter called **msg**.
- This method tries to print the **msg** string inside of square brackets.
- The interesting thing to notice is that after ***call( )*** prints the opening bracket and the **msg** string, next it calls ***Thread.sleep(1000)***, which pauses the current thread for one second.
- After that the closing bracket is print.

- The **constructor** of the next class, ***Caller***, takes a reference to an instance of the ***Callme*** class and a **String**, which are stored in ***target*** and ***msg***, respectively.
- The **constructor** also creates a new thread that will call this object's ***run( )*** method (***target.call(msg)***).
- The thread is started immediately. The ***run( )*** method of ***Caller*** calls the ***call( )*** method on the ***target*** instance of ***Callme***, passing in the ***msg*** string.
- Finally, the ***Synch*** class starts by creating a single instance of ***Callme***, and three instances of ***Caller***, each with a unique message string. The same instance of ***Callme*** is passed to each ***Caller***.

```
//This program is not synchronized.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);} catch(InterruptedException e) {
System.out.println("Interrupted"); }
System.out.println("]");
                    }
          }//class
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start( );
          }
public void run( ) {
target.call(msg);
                        }
            }//class
public class SynchMeth {
public static void main(String args[ ]) {
Callme target = new Callme( );
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join( );
ob2.t.join( );
ob3.t.join( ); } catch(InterruptedException e) {
System.out.println("Interrupted"); }
                        }//main
            }//class
```

- Here is the output produced by this program:

```
[Hello[Synchronized[World]
]
]
```

- As you can see, by calling ***sleep( )***, the ***call( )*** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings.
- In this program, ***nothing exists to stop*** *all three threads from calling the same method* **call( )***, on the same object (***target***), at the same time.*
- This is known as a ***race condition***, because the three threads are racing each other to complete the method.

- This example used ***sleep( )*** to make the effects repeatable and obvious. In most situations, a ***race condition*** is more subtle and less predictable, because you can't be sure when the *context switch* will occur.
- This can cause a program to run ***right one time*** and ***wrong the next***.
- To fix the preceding program, you must ***serialize*** **access** to ***call( )***.
- That is, you ***must restrict its access to only one thread*** at a time.
- To do this, you simply need to ***precede call( )***'s definition with the keyword ***synchronized***, as shown here:

```
class Callme {
synchronized void call(String msg) {
...}
```

- This prevents other threads from entering ***call( )*** while another thread is using it.
- After ***synchronized*** has been added to ***call( )***, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multi-threaded situation, you should use the ***synchronized*** keyword to guard the state from race conditions.
- Remember, **once** a thread enters any *synchronized* method on an instance, no other thread can enter any other *synchronized* method on the same instance.
- However, ***non-synchronized*** **methods** on that instance will continue to be callable.
- A similar example is next presented:

```
//synchronized method

class Parentheses {
//void display(String s)
synchronized void display(String s)
          {
                    System.out.print("("+s);
                    try{
                              Thread.sleep(1000);
                    }catch(InterruptedException e){

          System.out.println("Interrupted");
                    }
                    System.out.print(")");
          }//display
}//Par
```

```
class MyThread implements Runnable{
        String s1;
        Parentheses p1;
        Thread t;
        public MyThread(Parentheses p2, String s2){
                p1=p2;
                s1=s2;
                t=new Thread(this);
                t.start( ); }//cons
                        public void run( ){
                p1.display(s1); }
                                }//MyThread
```

```
Public class Demo{
        public static void main(String args[ ]){
                        Parentheses p3=new Parentheses( );
                MyThread name1=new MyThread(p3, "Bob");
                MyThread name2=new MyThread(p3, "Mary");
                        try{
                                name1.t.join( );
                                name2.t.join( );
                        }catch(InterruptedException e){
                                System.out.println("Interrupted");
                        }

        }
}//Demo
```

Result synchronized display method:

```
(Bob)(Mary)
```

Result not synchronized display method:

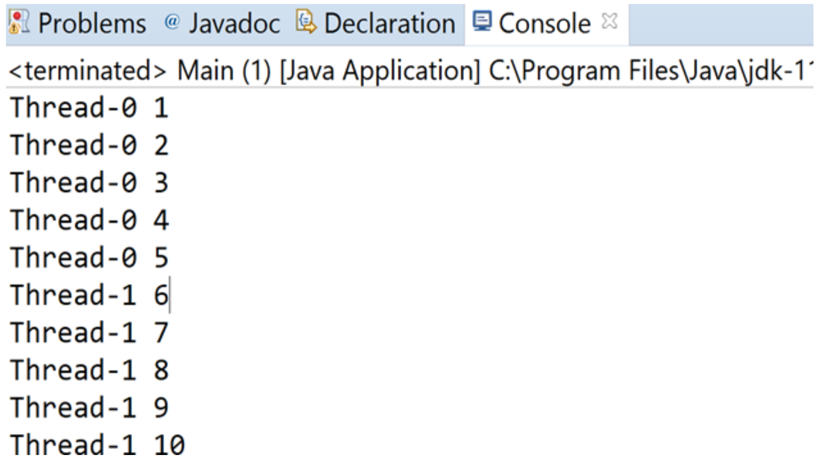
```
(Bob(Mary))
```

Example synchronized counter:

```
class SynchronizedCounter{
        private int nr=0;
synchronized public void increment( ){
                nr++;
System.out.println(Thread.currentThread( ).getName( )+" "+nr);
try{
Thread.sleep(300);
        }catch (Exception e){System.out.println(e);}
                }
}//SynchronizedCounter
```

```
class IncThread extends Thread{
            SynchronizedCounter sc;
public IncThread(SynchronizedCounter ob){
            sc=ob;  }
public void run( ){
for(int i=0; i<5; i++)sc.increment( );
                        }
}//IncThread

public class SyncCounterTest{
public static void main(String args[ ]){
            SynchronizedCounter ob=new    SynchronizedCounter( );
            IncThread f1=new IncThread(ob);
            IncThread f2=new IncThread(ob);
            f1.start( );
            f2.start( );
            }
}//TestS
```



```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-1
Thread-0 1
Thread-0 2
Thread-0 3
Thread-0 4
Thread-0 5
Thread-1 6
Thread-1 7
Thread-1 8
Thread-1 9
Thread-1 10
```

Synchronized Instruction (block)

- Not always we use *sync* methods; there are **unknown classes** *provided by others* where only the objects are available
- In this case a **synchronized instruction** will be used considering a ***synchronized block*** *with objects and methods* that must be synchronized
- The **calling** of *sync* methods from the *sync block* will be realized *after obtaining the **lock*** by the thread

- This is the general form of the ***synchronized*** statement:

```
synchronized(object) {
// statements to be synchronized
}
```

- Here, *object* is a reference to the object being synchronized.
- A ***synchronized block*** *ensures* that *a call to a method* that is a member of ***object*** occurs only after the current thread has successfully *entered object*’s ***lock***.
- Here is an ***alternative version*** of the first synchronized example, using a *synchronized block* within the ***run( )*** method:

```
//This program uses a synchronized block.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);} catch (InterruptedException e) {
System.out.println("Interrupted");}
System.out.println("]");
                    }
          }//class
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start( );
                    }
// synchronize calls to call( )
public void run( ) {
synchronized(target) { // synchronized block
target.call(msg);}
                    }//run
          }//class
```
public class SyncBlock{
```
public static void main(String args[ ]) {
Callme target = new Callme( );
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join( );
ob2.t.join( );
ob3.t.join( );
} catch(InterruptedException e) {
System.out.println("Interrupted");}
                    }//main
          }//class
```

[Hello]
[World]
[Synchronized]

- Here, the ***call( )*** method is not modified by ***synchronized***.
- Instead, the ***synchronized*** statement is used inside **Caller**’s ***run( )*** method imposing that the *target* object will be once accessed by only a thread.
- This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.
- The next example uses *synchronized* instruction for the second example:

```
//SI: synchronized instruction example
class Parentheses {
        void display(String s)
        {System.out.print("("+s);
                try{
                        Thread.sleep(1000);
                }catch(InterruptedException e){
                        System.out.println("Interrupted");}
                System.out.print(")");
        }//display
}//Par
```

```
class MyThread implements Runnable{
        String s1;
        Parentheses p1;
        Thread t;
        public MyThread(Parentheses p2, String s2){
                p1=p2;
                s1=s2;
                t=new Thread(this);
                t.start( );
        }//cons
        public void run( ){
                synchronized(p1){
                p1.display(s1);
                }
        }
}//MyThread
```

```
public class Demo1{
        public static void main(String args[ ]){
                Parentheses p3=new Parentheses();
                MyThread name1=new MyThread(p3, "Bob");
                MyThread name2=new MyThread(p3, "Mary");
                        try{
                                name1.t.join( );
                                name2.t.join( );
                        }catch(InterruptedException e){
                                System.out.println("Interrupted");
                        }

        }
}//Demo1
```

Result:

(Bob)(Mary)

Syncronized block example:

```
class SynchronizedThread extends Thread{
          //lock object
public Object lock;
private int nr=0;

public void run( ){
doSomething( );
           }
public void doSomething( ){
synchronized(lock){
for(int i = 0; i < 10; i++){
          nr++;
System.out.println("Thread " +  this.getName( )+" "+ nr + " is running ");     }
                              }//lock
                    }
}// class
```

```
public class SynchronizedThreadExample{
public static void main(String[ ] args){
//create a lock object
        SynchronizedThreadExample ex = new
SynchronizedThreadExample( );
//create first thread
        SynchronizedThread thread1 = new SynchronizedThread( );
        thread1.lock = ex;
//create second thread
        SynchronizedThread thread2 = new SynchronizedThread( );
        thread2.lock = ex;
//start both threads
        thread1.start( );
        thread2.start( );
                        }//main
        }
```

Producer/consumer as communication with threads

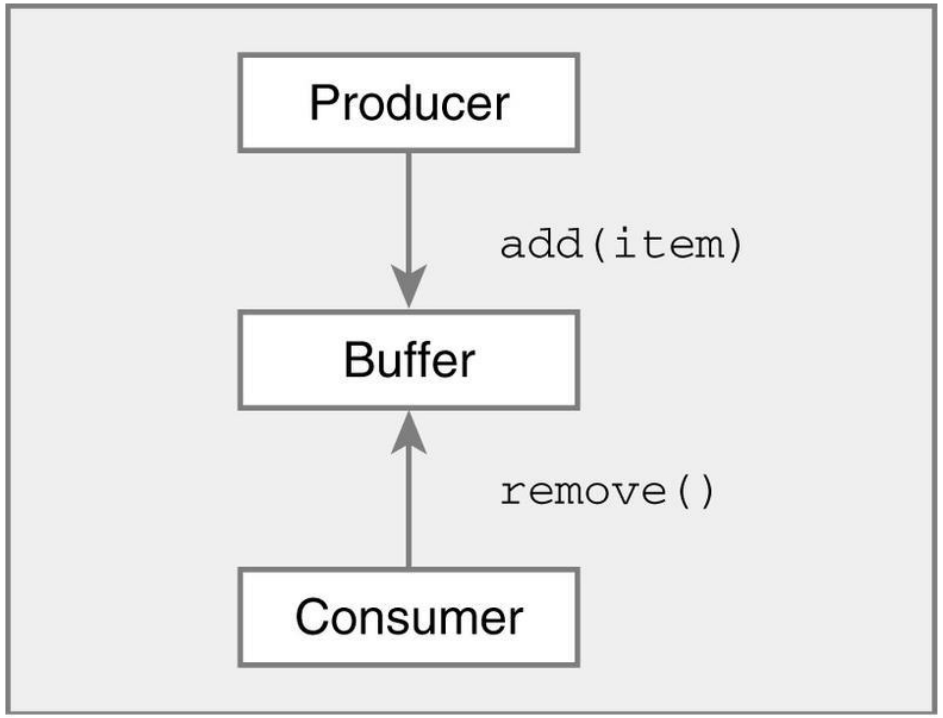
- The **producer** will provide some data that the **consumer** wants to use.
- It is possible that *the activity* to produce and to consume to be *different*.
- If the producer is faster, than will override some data
- The methods *wait( )*, and *notify( )*, *notifyAll( )* are used from the ***Object*** class (not ***Thread*** class) to manage the process.
- This mechanism will consider *mutual exclusion* with *co-operationn* with an *entry-set* owner and *wait-set*, specific to ***monitors***.

Buffer

Transfers items from ***producers*** to ***consumers***

Very useful in multithreaded programs

Synchronization needed to prevent multiple consumers removing same item



Buffer usage

- ***Producer*** thread
  - calls *buffer.add(o)*
  - adds *o* to the buffer

- ***Consumer*** thread
  - calls *buffer.remove( )*
  - if object in buffer, removes and returns it
    otherwise, waits until object is available to remove

Example: Buffer Implementation – raw, non generic

```
public class Buffer {
    private LinkedList objects = new LinkedList( );
    public synchronized add( Object x ) {
        objects.add(x);
    }
    public synchronized Object remove( ) {
        while (objects.isEmpty( )) {
            ;  // waits for more objects to be added
        }
        return objects.removeFirst( );//from LinkedList
    }
} // if empty buffer, remove( ) holds lock and waits
  //   prevents add( ) from working ⇒ deadlock
```

Eliminating Deadlock

```
public class Buffer {
    private Object [ ] objects;
    private int numberObjects = 0;

    public synchronized add( Object x ) {
            objects.add(x);
        }
    }

    public Object remove( ) {
        while (true) {  // waits for more objects to be added
            synchronize(this) {
                if (!objects.isEmpty( )) {
                    return objects.removeFirst( ); }
            }
        }
    } // if empty buffer, remove( ) gives up lock for a moment
```

- To avoid *pooling*, Java includes an elegant inter-process communication mechanism via the ***wait( )***, ***notify( )***, and ***notifyAll( )*** methods.
- These methods are implemented as ***final*** methods in ***Object*** class, so all classes have them.
- All these methods ***can be called only*** from within a ***synchronized*** context.
- Although conceptually advanced from a computer science perspective, the ***rules for using*** these methods are quite simple:

- ***wait( )*** tells the calling thread to give up the monitor and *go to sleep* until some *other thread* enters the same monitor and calls ***notify( )***.

- ***notify( )*** *wakes up the first thread* that called ***wait( )*** on the same object.

- ***notifyAll( )*** *wakes up all the threads* that called ***wait( )*** on the same object.

The highest priority thread will run first.

- The *wait( )* method of an object will impose that the thread from where is realized the calling process to enter in a **blocked** state, passing the monitor.
- When another thread will call *notify( )* or *notifyAll( )* for the same object in a *synchronized method* (from the object monitor) the thread will be reactivated.

- We have different formats for the *wait( )* method with milils/nano seconds
- Sun/Oracle recommends that the calling of *wait( )* to do not be realized in a loop that will verify the waiting condition of the thread.
- A ***“spurious wakeup” (false wakeup)*** it is possible to appear, the tread to wake up for no reason when *notify( )* will wake up a thread, but this process is *not controlled* if more threads are in the same monitor

- ***wait( )***
  - Invoked on object
  - must already hold lock on that object
  - gives up lock on that object
  - goes into a *wait* state
- ***notifyAll( )***
  - Invoked on object
  - must already hold lock on that object
  - all threads waiting on that object are woken up
    - but they all gave up their lock when they performed *wait( )*
    - will have to regain lock before then can run
    - thread performing *notify( )* holds lock at the moment

wait( ), notify( ), notifyAll( ) **methods prototypes**

- *public final void wait( ) throws IllegalMonitorStateException, InterruptedException;*

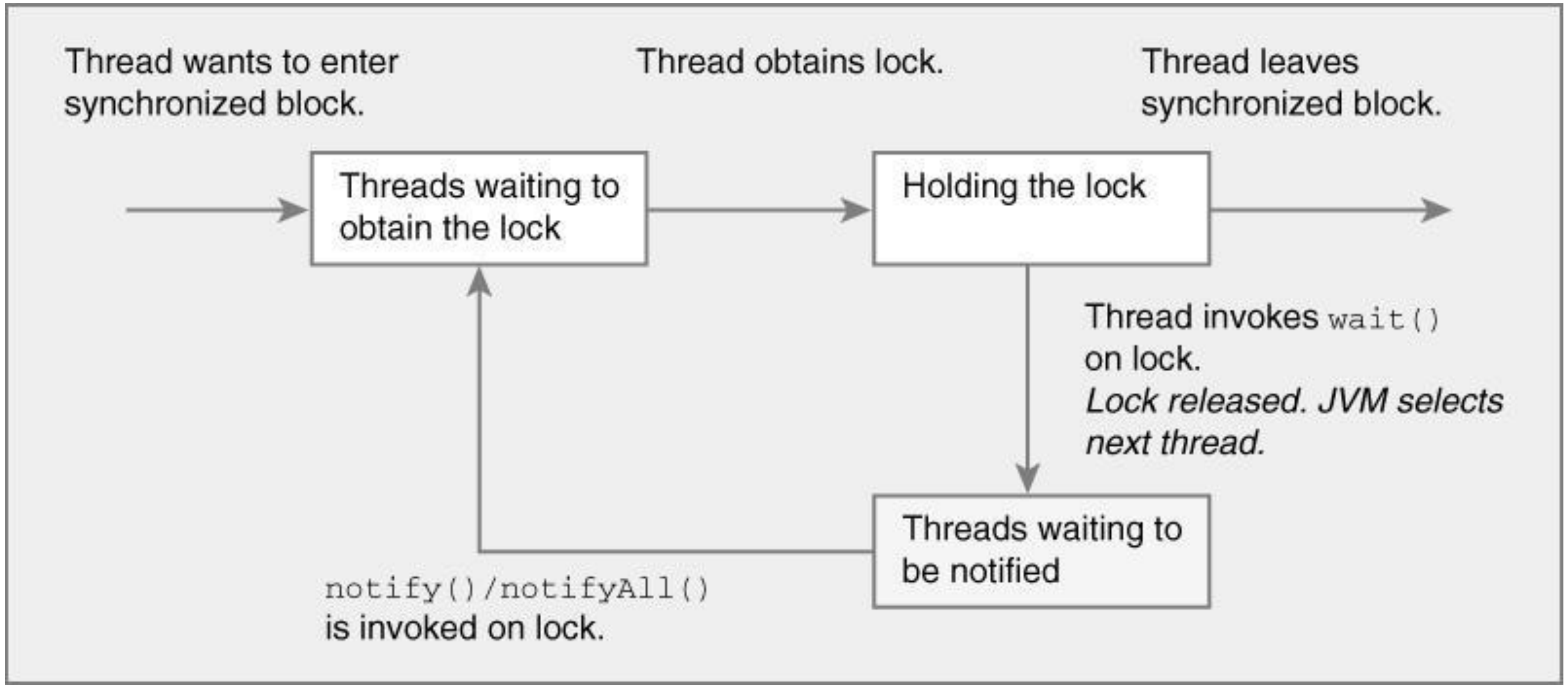
- *public final void wait(long millis) throws IllegalMonitorStateException, InterruptedException;*

- *public final void wait(long millis, int nanos) throws IllegalMonitorStateException, InterruptedException;*

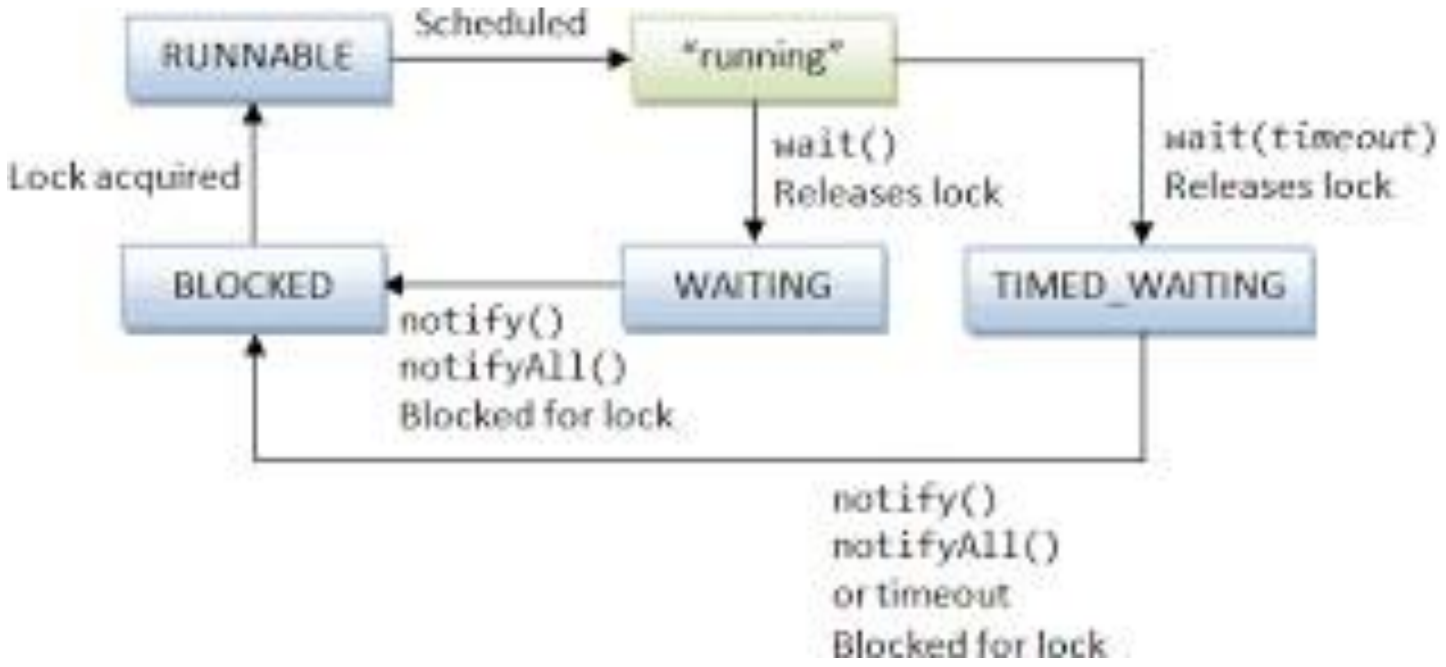
- *public final void notify( ) throws IllegalMonitorStateException;*

- *public final void notifyAll( ) throws IllegalMonitorStateException;*

State transitions using *wait( )* and *notify( )*



Thread wants to enter synchronized block.
Thread obtains lock.
Thread leaves synchronized block.
Threads waiting to obtain the lock
Holding the lock
Thread invokes wait() on lock.
Lock released. JVM selects next thread.
Threads waiting to be notified
notify()/notifyAll() is invoked on lock.


RUNNABLE
Scheduled
"running"
wait()
Releases lock
wait(timeout)
Releases lock
Lock acquired
BLOCKED
WAITING
TIMED_WAITING
notify()
notifyAll()
Blocked for lock
notify()
notifyAll()
or timeout
Blocked for lock

Eliminating Deadlock- Using *wait( )* and *notifyAll( )*

```
public class Buffer {
    private LinkedList objects = new LinkedList( );

    public synchronized add(Object x ) {
        objects.add(x);
        this.notifyAll( );
    }

    public synchronized Object remove( ) {
        while (objects.isEmpty( )) {
                this.wait( );
        }
        return objects.removeFirst( );
    }
}
```

- The program will not compile because:

- the *wait( )* method is declared to throw an *InterruptedException*
  - a checked exception

- You rarely have situations where a *wait( )* will throw an *InterruptedException*
  - but the compiler forces you to deal with it

```
public class Buffer {
    private LinkedList objects = new LinkedList( );

    public synchronized add(Object x ) {
        objects.add(x);
        this.notifyAll( );
    }

    public synchronized Object remove( ) {
        while (objects.isEmpty( )) {
                try {
                  this.wait( );
                } catch (InterruptedException e) { }
        }
        return objects.removeFirst( );
    }
}
```

```
//Producer-consumer complete example
class Queue{
        int exVal;//take from Publisher
        boolean busy=false;//value in Queue if true

        synchronized int get( ){
                if(!busy)
                        try{
                                wait( );//no value in que
                        }catch(InterruptedException e){
        System.out.println("Get:InterruptedException ");
                        }
                System.out.println("Get: "+exVal);
                        busy=false;
                        notify( );//to put other value
                        return exVal;
        }//get from queue
synchronized void put (int exVal){
                    if(busy)
                              try{
                                        wait( );//we have value
                              }catch(InterruptedException e){
          System.out.println("Put:InterruptedException ");
                              }
                              this.exVal=exVal;
                              busy=true;//available value in queue
                              System.out.println("Put: "+exVal);
                              notify( );//to take the value by get
          }//put in Queue

}//Queue
```

```
class Publisher implements Runnable{
        Queue q;
        Publisher(Queue q){
                this.q=q;
                new Thread(this, "Publisher").start( ); }
        public void run( ){
                for(int i=0;i<3;i++)
                        q.put(i);
        }
}// Publisher
```

```
class Consumer implements Runnable{
          Queue q;
          Consumer(Queue q){
                    this.q=q;
                    new Thread(this, "Consumer").start( );
          }
                    public void run( ){
                    for(int i=0;i<3;i++){
                              q.get( );  }
                                                            }
}//Consumer
public class DemoProdCons{
          public static void main(String args[ ]){
          Queue q=new Queue( );
          new Publisher(q);
          new Consumer(q);}
}//DemoPC
```

```
Put: 0
Get: 0
Put: 1
Get: 1
Put: 2
Get: 2
```

Oracle (Sun) Producer/Consumer example

```
//Oracle Producer-Consumer
public class ProducerConsumerTest {
    public static void main(String[ ] args) {
        CubbyHole c = new CubbyHole( );
        Producer p1 = new Producer(c, 1);//Producer1 puts using object c
        Consumer c1 = new Consumer(c, 1);//Consumer1 gets using object c
        p1.start( );
        c1.start( );
    }
}
```

```
class CubbyHole {
   private int contents;
   private boolean available = false;
   public synchronized int get( ) {
      while (available == false) {
         try {  wait( ); } catch (InterruptedException e) {  }
      }//while
      available = false; notifyAll( );//notify to put other value
      return contents; //we get the value
           }//get
  public synchronized void put(int value) {
      while (available == true) {
         try {
            wait( );  } catch (InterruptedException e) {  }
      }//while
      contents = value;//we put the value
      available = true; notifyAll( );//notify to get the value
   }//put
}//class
```

```
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
   public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    @Override
public void run( ) {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get( );
            System.out.println("Consumer #" + this.number
                                   + " got: " + value);
        }
    }//run
}//class
```

```
class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number; }
@Override
    public void run( ) {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random( ) * 100));
            } catch (InterruptedException e) { }
        }
    }//run
}//class
```

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

Old suspend/resume a thread

- Sometimes, **suspending execution** of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn’t want a clock, then its thread can be suspended.
- Whatever the case, suspending a thread is a simple matter. Once suspended, **restarting** the thread is also a simple matter.
- The **mechanism** to *suspend*, *stop*, and *resume* threads **differ between** Java new versions and earlier versions.
- Although you should use the Java new versions approach for all new code, you still need to understand how these operations were accomplished for earlier Java environments.

- **Prior to Java new versions**, a program used ***suspend( )*** and ***resume( )***, which are methods defined by **Thread**, to pause and restart the execution of a thread. They have the form shown below:

  final void suspend( )

  final void resume( )

- The following program demonstrates these methods:

```
//Using suspend( ) and resume( ).
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start( ); // Start the thread
}
// This is the entry point for thread.
public void run( ) {
try {
for(int i = 15; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(200);}} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
                    }
          }//class
```

```
public class SuspendResume {
@SuppressWarnings("deprecation")
public static void main(String args[ ]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try {Thread.sleep(1000);
ob1.t.suspend( );
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.t.resume( );
System.out.println("Resuming thread One");
ob2.t.suspend( );
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.t.resume( );
System.out.println("Resuming thread Two");} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");}
// wait for threads to finish
try {System.out.println("Waiting for threads to finish.");
ob1.t.join( );
ob2.t.join( );} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");}
System.out.println("Main thread exiting.");}//main
            }//class_SR
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
```

```
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

- While the ***suspend( )***, ***resume( )***, and ***stop( )*** methods defined by **Thread** seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why.
- The ***suspend( )*** method of the **Thread** class is deprecated in Java new versions. This was done because ***suspend( )*** can sometimes cause ***serious system failures***.
- Assume that a thread has ***obtained locks on critical data*** structures. If that thread is suspended at that point, those *locks* are not abandoned.
- Other threads that may be waiting for those ***resources can be deadlocked***.

- The ***resume( )*** method is also deprecated. It does not cause problems but cannot be used without the ***suspend( )*** method as its counterpart.

- The ***stop( )*** method of the **Thread** class, too, is deprecated in Java new versions. This was done because this method can sometimes cause ***serious system failures***.

- Assume that a thread is writing to a **critically important data** structure and has ***completed only part of*** its changes. If that thread is stopped at that point, that ***data*** structure might be ***left in a corrupted state***.

- The following example illustrates how the ***wait( )*** and ***notify( )*** methods that are inherited from ***Object*** class can be used to control the execution of a thread.
- This example is similar to the previous program example as effect. However, the deprecated method calls have been removed. Let us consider the operation of this program. The implemented mechanism is based on ***monitors*** that involves ***mutual exclusion*** (*synchronized* block) and ***communication*** (*wait-notify*).

- The **NewThread** class contains a ***boolean*** instance variable named ***suspendFlag***, which is used to control the execution of the thread. It is initialized to ***false*** by the constructor.
- The ***run( )*** method contains a ***synchronized*** statement block that checks ***suspendFlag***. If that variable is ***true***, the ***wait( )*** method is invoked to suspend the execution of the thread.
- The ***mysuspend( )*** method sets ***suspendFlag*** to ***true***.
- The ***myresume( )*** method sets ***suspendFlag*** to ***false*** and invokes ***notify( )*** to wake up the thread.
- Finally, the ***main( )*** method has been modified to invoke the ***mysuspend( )*** and ***myresume( )*** methods.

```
//Suspending and resuming a thread for Java wait-notify
class NewThread implements Runnable {
String name; // name of thread
Thread t;
boolean suspendFlag;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
suspendFlag = false;
t.start( ); }// Start the thread
// This is the entry point for thread.
public void run( ) {
                    try {
for(int i = 15; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(200);
synchronized(this) {
while(suspendFlag){
          wait( );    }        }
                      } } catch (InterruptedException e) {
          System.out.println(name + " interrupted.");}
System.out.println(name + " exiting.");
                              }//run method
```

```
void mysuspend( ) {
suspendFlag = true;}
synchronized void myresume( ) {
suspendFlag = false;
notify( );}
}//class NewThread

public class SuspendResumeWaitNotify{
public static void main(String args[ ]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
            try {Thread.sleep(1000);
ob1.mysuspend( );
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume( );
System.out.println("Resuming thread One");
ob2.mysuspend( );
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume( );
System.out.println("Resuming thread Two");
            } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");}
// wait for threads to finish
             try {
System.out.println("Waiting for threads to finish.");
ob1.t.join( );
ob2.t.join( );
             } catch (InterruptedException e) {
System.out.println("Main thread Interrupted");}
System.out.println("Main thread exiting.");
                          }//main
}//class SuspendResumeWN
```

The output from this program is like to that shown in the previous example (differences concerning the system execution).
Next example will present the same principle.

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 15
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
One: 10
Two: 10
Suspending thread One
Two: 9
Two: 8
Two: 7
Two: 6
Two: 5
Resuming thread One
Suspending thread Two
```

```
One: 9
One: 8
One: 7
One: 6
One: 5
Resuming thread Two
Waiting for threads to finish.
Two: 4
One: 4
One: 3
Two: 3
One: 2
Two: 2
Two: 1
One: 1
One exiting.
Two exiting.
Main thread exiting.
```

//Other Wait-Notify simple example

```
class MyThread implements Runnable{
        String name;
        Thread t;
        boolean suspend;
        MyThread( ){t=new Thread(this, "Thread");
                suspend=false;
                t.start( );}
                public void run( ){
try{for(int i=0;i<10;i++){System.out.println("Thread "+i);Thread.sleep(200);
        synchronized(this){while(suspend) wait( );}          }
        }catch(InterruptedException e){System.out.println("Thread 
interrupted");}
                System.out.println("Thread exiting"); }//run
        void suspendThread( ){suspend=true;      }//suspend
        synchronized void resumeThread( ){suspend=false;
                notify( ); }//resume
}//MyThread
```

```
public class DemoWaitNotify {
public static void main(String[ ] args) {
            MyThread t1=new MyThread( );
try{
            Thread.sleep(1000);
            t1.suspendThread( );
            System.out.println("Thread suspended");
            Thread.sleep(1000);
            t1.resumeThread( );
            System.out.println("Thread resumed");
}catch(InterruptedException e){
            }
try{
            t1.t.join( );
}catch(InterruptedException e){
            System.out.println("Main Thread interrupted");
            }
            }//main
}//DemoWN
```

```
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread suspended
Thread resumed
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread exiting
```

Deadlock

- When **two threads** have a **circular dependency** to a pair of synchronized objects
- A **thread enter** in the *monitor of object X* and **other thread** in the *monitor of object Y*
- If **the thread from X** try to *call a sync method from Y* will **block it**, and if **thread from Y** try to *call any sync method from X*, the thread will **wait to infinite**

- **Example:** Two writers may use same one paper and one ball pen to write something. One will take the paper, the other one the ball pen.

Avoiding Deadlock

- In general, want to be careful about performing any operations that might take a long time while holding a lock
- What could take a really long time?
    - getting another lock

- Particularly if you get deadlock

Deadlock Example 1

```
Thread1( ) {
    synchronized(a) {
        synchronized(b) {
            ...
        }
    }
}
```

```
Thread2( ) {
    synchronized(b) {
     synchronized(a) {
            ...
        }
    }
}
```

```
// Thread1 holds lock for a, waits for b
// Thread2 holds lock for b, waits for a
```

Deadlock Example 2

```
void moveMoney(Account a, Account b, int amount) {
    synchronized(a) {
        synchronized(b) {
            a.debit(amount);
            b.credit(amount);
        }
    }
}

Thread1( ) { moveMoney(a,b,10);  }
   // holds lock for a, waits for b

Thread2( ) { moveMoney(b,a,100);  }
   // holds lock for b, waits for a
```

Producer consumer it is another deadlock example if is not correctly managed.

Sometimes, you need to wait for another thread else to do something before you can do something

- To understand deadlock fully, it is useful to see it in action.
- The next example creates two classes, ***A*** and ***B***, with methods ***foo( )*** and ***bar( )*** respectively, which pause briefly before trying to call a method in the other class.
- The main class, named **Deadlock**, creates an ***A*** and a ***B*** instance, and then starts a second thread to set up the deadlock condition.
- The ***foo( )*** and ***bar( )*** methods use ***sleep( )*** as a way *to force the deadlock* condition to occur.

```
//An example of deadlock.
class A {
synchronized void foo(B b) {
String name = Thread.currentThread( ).getName( );
System.out.println(name + " entered A.foo");
try {
Thread.sleep(1000);
} catch(Exception e) {
System.out.println("A Interrupted");
}
System.out.println(name + " trying to call B.last( )");
b.last( );
}
synchronized void last( ) {
System.out.println("Inside A.last");
                    }
          }//class_A
```

```
class B {
synchronized void bar(A a) {
String name = Thread.currentThread( ).getName( );
System.out.println(name + " entered B.bar");
try {
Thread.sleep(1000);
} catch(Exception e) {
System.out.println("B Interrupted");
}
System.out.println(name + " trying to call A.last( )");
a.last( );
                                        }
synchronized void last( ) {
System.out.println("Inside A.last");
                                        }
          }//class_B
```

```
public class Deadlock implements Runnable {
A a = new A( );
B b = new B( );
Deadlock( ) {
Thread.currentThread( ).setName("MainThread");
Thread t = new Thread(this, "RacingThread");
t.start( );
a.foo(b); // get lock on a in this thread.
System.out.println("Back in main thread");
                    }
public void run( ) {
b.bar(a); // get lock on b in other thread.
System.out.println("Back in other thread");
                    }
public static void main(String args[ ]) {
new Deadlock( );     }
          }//class_D
```

When you run this program, you will see the output shown here:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last( )
RacingThread trying to call A.last( )
```

-Because the program has deadlocked, you need to press CTRL-C to end the program (depending on the OS).
-You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC .

You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**.
At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete.