

Threads



Threads

- *Threads transcend Java!*
- Their role in programming?
 - to provide *intra-program concurrency*

intra = within

Why threads? Why ‘intra-program’ concurrency?

- A good (*but not very common*) reason:
 - you have a computationally heavy task that can be split into independent subtasks
 - you have a multiprocessor CPU
 - you would like to allocate each CPU to a subtask so that the overall task is finished more quickly
- threads make it **easy** to code such systems

‘Parallel’ (*not simply ‘concurrent’*)

Why threads? Why ‘intra-program’ concurrency?

- ***A more common reason:***
 - you need to handle data transfers on several separate ‘channels’
 - any channel can “block”
 - an input buffer is empty; you try to read, you block
 - an output buffer is full; you try to write, you block
 - you don’t want the overall processing to block because there are input data waiting on other channels or data that can be sent to ready output channels
 - you could program a polling technique but (wisely) would prefer not to poll
 - again, threads simplify coding

Example: Web browser fetching several pictures over net, displaying Java animation, and responding to user inputs: ‘channels’ - net connections, screen, mouse, keys, ...

Threads – analogy to “time share systems”

- Many similarities between threads and processes in a time share system.
- You should be familiar with time share systems from your work on Unix (and from CSCI212).
- Hence you know behaviours of typical process.

“time share system”

- You are all familiar with time-share system –
 - runs on machine with 1..4 CPUs
 - OS processes and user processes
 - many logged in users, each having several processes
 - at any one time
 - many processes blocked – waiting for user to input something, waiting for a disk transfer, ...
 - some processes suspended (eg a program being debugged, its process suspended while owner pokes around with debugger)
 - some on ‘ready queue’ waiting for a CPU
 - a few running (\leq number of CPUs that you have)

“time share system”

- Typical process on time share system
 - waits in ready queue
 - gets allocated a CPU
 - runs for a time
 - either blocks when asks for a data transfer that OS can't satisfy immediately (read keyed input when buffer empty; unbuffered read from disk; has filled all buffers for pending disk writes)
 - or uses up a “time-quantum”
 - clock interrupts
 - OS determines that current process has had its share of CPU attention
 - OS puts process back on ready queue allowing another to run

“time share system”

- Processes can communicate
 - lots of restrictions (parent - child, siblings, but not general intercommunication)
 - common mechanisms
 - “pipe” data buffer, one process writes to it, the other reads
 - “shared memory” quite literal, a block of memory that both processes can read/write
 - “signals” limited (normally relate to enforced process termination)

“time share system”

- Multiple processes on a time share system are really there to optimise use of CPU resource
 - keep CPU running; one process blocks, another is ready to run
- Sometimes can use multiple processes to achieve goals similar to those noted as motivating threads
 - parallel execution on a multiprocessor
 - separate i/o tasks allocated to different processes
- Processes are however
 - *expensive* (in terms of OS resources)
 - system dependent, process control & communications mechanisms vary a lot

“time share system” – on a micro scale

- A multi-threaded process is like a time share system on a micro scale
 - each thread is analogous to a process (in a process group using shared memory)
 - own program counter, register save area, stack
 - share access to heap, static data, code segments
 - process has an internal “thread scheduler” (analogous to OS's process scheduler)
 - thread descriptor table - status: running, ready, suspended, blocked
 - a priority mechanism for choosing next thread to run
 - *possibly* something equivalent to an OS's time-quantum
 - thread scheduler most often provided mainly by run-time library with limited OS support

“time share system” – on a micro scale

- Advantages of threads over concurrent processes:
 - much lower costs in terms of OS resources
 - thread descriptor tables much smaller than processor descriptor tables (all threads use same memory map etc)
 - time to switch threads much less than time to switch processes
 - start up cost of new thread << startup cost of process
 - much easier communication among threads
 - threads model can be defined by language or a standard API making threads relatively independent of platform

Support for threads

- POSIX standard threads API for C/C++ programs.
- Solaris threads for C/C++ (a nicer implementation)
- Threads packages for PC and Mac
- All are essentially run-time libraries that get linked to your code, some OS support

Support for threads

- Sun provides extensive OS support for threads on its multiprocessor machines
- Can specify number of actual CPUs that it would be nice to use
- Can specify how different threads should be distributed over CPUs
- All works on single CPU systems, but miss many of benefits

Java and threads

- In many ways more limited than facilities provided by libraries like Solaris threads package.
- Integrated into language and run time model
 - all objects automatically get synchronisation locks for use by multithreaded programs
 - thread control via language statements rather than function calls
 - slightly more secure
 - slightly more concise

Java and threads

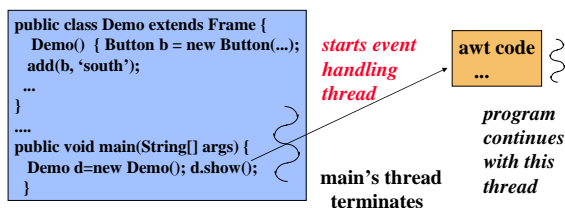
- Why build threads into the language?
- Probably because of intended application of Java in network intensive systems and particularly to support Applets
 - most Java programs will have potential for concurrency,
 - will have GUI interfaces which should always respond to user
 - will be transferring data on networks (frequent i/o blocking)
 - JVM really needs to be thread aware to support multiple Applets running

Java and threads

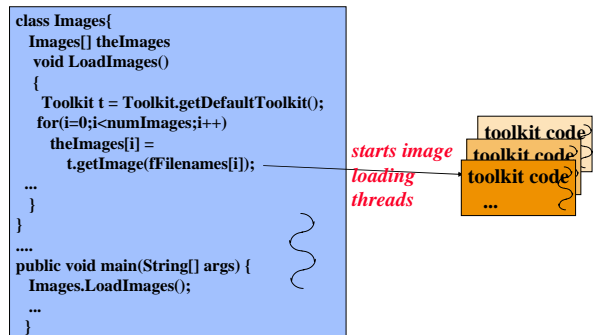
- Language support
 - those locks available with every object
 - Thread class in java.lang package
 - basic thread operations
 - start, suspend, resume, yield, kill
 - priorities
 - concept of thread groups
 - Runnable interface
 - “synchronized” qualifier for methods
 - also, synchronized blocks within methods
 - wait() and notify() operations defined at Object

You've used threads ...

- “Behind the scenes”, Java has already been putting multiple threads into your programs:



Threads working away behind scenes ...



When might you use threads explicitly?

- **Compute task**
 - action button “Calculate”; data object listens for event; when gets event, does 5 minutes of number crunching;
 - your Applet/application would freeze, no response to further prodding with mouse until the 5 minutes of calculations completed
- SO –
 - at least, *start a separate thread for the calculations*
 - possibly also display something like a progress bar

When might you use threads explicitly?

- Your problem involves a simulation with a large number of ‘active’ objects each running a separate agenda.
- Coding may be more natural if each active object has its own thread§
 - “Doom” game: thread with each non-player character
(*npc.run() { forever do { move(), sleep a bit } }*)

– *I said more natural, not intrinsically better, not more efficient; just easier to think about*

§ alternative is single thread structure like

forever do { for each activeItem in List do activeItem.move(); maybe sleep }

When might you use threads explicitly?

- You are handling multiple concurrent i/o channels.
 - as a reader (client)
 - eg downloading several images
 - as a server
 - support connections from many concurrent clients

java.lang.Thread

- Java.lang offers two ways of using threads
 - “**thread object**” + “**runnable object**”
 - thread object represents active locus of control, something that can be suspended, resumed, ... (virtual CPU in ‘process’ analogy)
 - runnable object represents particular activity (program in ‘process’ analogy)
 - “**instance of subclass of thread**”
 - two concepts are folded together

java.lang.thread Thread and Runnable

- run() function is code with agenda for active object;
- **class Activity implements Runnable**
 - create **Activity** object
 - create standard **Thread** object, giving it the **Activity**
 - tell **Thread** object to start
 - run() defined in class Activity, implementing specification defined in interface Runnable
- **class Activity extends Thread**
 - create **Activity** object, tell it to run
 - run() defined in class Activity, replacing version inherited from class Thread

java.lang.thread Thread and Runnable

- Textbooks rather inconsistent on usage.
- *Generally, you should only inherit from a class if you want to adapt or extend its behaviour.*
- If you simply want a standard Thread (with no changes to methods like start(), stop()) then use a standard Thread that works with a Runnable.
- Only implement as subclass of Thread if changing more than just the run() method.

Runnable

- Simply an interface
- ```
interface Runnable {
 public abstract void run();
}
```
- As an interface, can be folded into any other class (no restrictions on number of interfaces that a class implements).

## Thread

- Thread
  - owns
    - maybe a reference to a Runnable
    - data like a “name”, threadgroup identifier, priority
    - *and lots of stuff in thread descriptor table that you can't get at*
  - does (or has done to it!)
    - start(), stop(), run(), resume(), sleep(), join(), destroy(), interrupt()
    - set/getName(), set/getPriority(), setDaemon()
    - isInterrupted(), isAlive()

## Thread

- Thread is concrete class
  - class X can extend Thread,
    - but as no multiple inheritance, cannot extend Thread and extend some other class as well
- 
- *Daemon thread*
    - *handles background activities,*
    - *no termination in its code (forever ...)*
    - *JVM will stop if only daemon threads left*

## Thread

- Priority
  - value between Thread.MIN\_PRIORITY and Thread.MAX\_PRIORITY
  - thread scheduler (in runtime library) gets to run when new thread created, a thread gets blocked, a thread yields, ...
    - picks highest priority thread to run next
    - arbitrary selection among processes of equal priority (no fairness guarantee)
  - thread package not required to support anything like “time quantum” of typical OS

## Life for a Thread

- construct
  - thread object created, maybe linked to a runnable, priority & daemon status same as creating thread
- changes to priority, daemon status
- start()
  - this is where Thread object comes “alive”
    - current (calling) thread continues
    - new thread entry in runtime tables
  - start calls run()

## Life for a Thread

- run()
  - execute agenda of attached runnable
  - if run() returns, Thread terminates, no longer alive; (neither Thread nor Runnable objects discarded
    - can be accessed to get at data used or created when running)

## Life for a Thread

- Within run()
  - may voluntarily relinquish CPU, getting put back on ready queue (either immediately or after some sleep time)
  - may be suspended
    - either by requesting suspension, or by another thread requesting that a target thread be suspended (a thread cannot arbitrarily suspend another, there are access restrictions, eg are they in same threadgroup)
  - may be stopped by another thread (again, access controls apply)
  - may get blocked (asking for i/o)
  - ...

## Life for a Thread

- Within run()
  - ...
  - may get blocked (asking for i/o)
  - may get blocked asking for access to an object locked by some other thread
  - may get blocked by saying it wants to wait until some other thread terminates
  - may get blocked by saying it wants to wait until some other thread notifies it that a synchronisation variable has been changed

## suspend() & resume() “deprecated”

- Operations “suspend” and “resume” quite useful in simple thread demonstration programs
  - Control thread can suspend worker thread
  - Control thread can later resume worker thread
- Operations were part of original java.lang.Thread class
- Now “deprecated” –
  - Oxford dictionary: “*express wish against or disapproval*”
  - Sun Java: “*don't use this function in any new program*”

## suspend() & resume() “deprecated”

- Advantage: Easy to use functions providing simple control of threads
- Disadvantage: Easy to cause major problems in complex programs; e.g. :
  - Worker thread claims resource and locks it
  - Control thread tells worker to suspend
  - Control tries to access locked resource
  - **Deadlock**

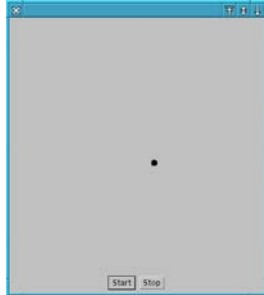
## stop() deprecated

- Similarly, “stop” operation (which allowed control thread to terminate execution of worker thread) is now deprecated.
- Problem:
  - Worker thread working
    - Updating some important data structure (for which may have lock)
    - Update not yet complete
  - Control thread tells worker to stop
  - Worker thread releases any locks it held, then terminates
  - Control thread tries to used data structure, gets confused by information from incomplete update

## Workarounds for deprecated operations

- Basically
  - Have boolean “flag” variables (“oktocontinue”, “quitnow”) associated with Runnable object
  - Functions that allow these to be set by a control thread
  - Main run loop in runnable object should check these at appropriate points

## Trivial thread example



## Trivial thread example

- This example based on that of Horstmann (Volume 2)
- “Ball” object has (actually *is*) a thread; run() method is
  - erase, move, redraw
  - sleep
- Button controls to suspend & resume this thread
- Overall program has main thread living in awt event handling; this thread deals with operation of “start” & “stop” buttons and window closing.

## Bouncing ball

- Horstmann has
  - Ball as subclass of Thread (while only changing run())
  - direct drawing to graphics object of Canvas (not using repaint...update mechanisms) ‡
  - both are somewhat dubious design decisions, OK for a toy but I would object if done this way for real program
- Drawing code illustrates use of XOR mode - a standard approach for simple graphics movement (draw, draw again - as XOR original drawing disappears)
- Also note disposal of Graphics objects (relates to problem on PCs)

‡Restrict direct drawing to canvas to situations where you are providing immediate feedback for mouse-based drawing actions

```

import java.awt.*;
import java.awt.event.*;
// Based on Horstmann's example
class Ball extends Thread { ... }
public class BouncingBall extends Frame implements
 ActionListener, WindowListener {
 public static void main(String[] argv) {
 BouncingBall bb = new BouncingBall();
 bb.show();
 }
 ...
}

```

```

class Ball extends Thread {
 public Ball(Canvas c) { box = c; }
 public void draw() { ... }
 public void move() { ... }
 public void run() {
 suspend();
 draw();
 for(;;) {
 move();
 try{ Thread.sleep(5); } catch (InterruptedException ie) {}
 }
 }
 private Canvas box;
 private static final int XSIZE = 10; private int x = 0; private int y = 0;
 private int dx = 2; private int dy = 2;
}

```

```

public void draw() {
 Graphics g = box.getGraphics();
 g.fillOval(x, y, XSIZE, YSIZE);
 g.dispose();
}
public void move() {
 if(!box.isVisible()) return;
 Graphics g = box.getGraphics();
 g.setXORMode(box.getBackground()); g.fillOval(x, y, XSIZE, YSIZE);
 x+=dx; y+=dy;
 Dimension d=box.getSize();
 if(x<0) { x = 0; dx = -dx; }
 if(x+XSIZE>=d.width) { x=d.width-XSIZE; dx=-dx; }
 if(y<0) { y=0; dy=-dy; }
 if(y+YSIZE>=d.height) { y=d.height-YSIZE; dy=-dy; }
 g.fillOval(x,y, XSIZE,YSIZE);
 g.dispose();
}

```

```

public class BouncingBall extends Frame implements ActionListener,
WindowListener {
public void windowClosing(WindowEvent e) { System.exit(0); }
...
public void windowDeactivated(WindowEvent e) { }
private void actionPerformed(ActionEvent evt) {
String s = evt.getActionCommand();
if(s.equals("Start")) DoStart(); else if(s.equals("Stop")) DoStop();
}
private void DoStart() { if(!running) { aBall.resume(); running = true; } }
private void DoStop() { if(running) { aBall.suspend(); running = false; } }
...
private boolean running = false;
private Ball aBall;
}

```

*Safety checks not really needed, it won't hurt a Thread if tell it to resume while it is running*

**suspend() & resume() now deprecated;**  
**use a synchronization object with wait() and notify()**

```

public BouncingBall() {
Canvas c = new Canvas(); c.setSize(400,400);
aBall = new Ball(c);
aBall.start(); // will immediately suspend until graphics ready
add(c,"Center");
Panel p = new Panel();
Button b = new Button("Start");
b.addActionListener(this);
p.add(b);
b = new Button("Stop");
b.addActionListener(this);
p.add(b);
add(p,"South");
pack();
addWindowListener(this);
}

```

### Synchronization of threads

- Sometimes have threads that really are independent, like Horstmann's later example with a dozen bouncing balls.
- Usually, threads interact when they access shared data.

### Questionable code fragment ...

|                                                                                                                                                                                                            |                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <pre> while(...) { ... // move to next image // in animation sequence // working cyclically img_num++; if(img_num &gt;= MaxImgs) img_num = 0; ... repaint(); } </pre> <p><b>run() method of object</b></p> | <pre> ... g.drawImage( theImages[img_num], 0, 0); ... </pre> <p><b>code executed by awt thread</b></p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|

### Questionable code fragment ...

- Run thread works cyclically through set of images, requesting repaint() after a change.
- paint() executed by main awt based thread.
- Works!
  - Except that on a very very few occasions, got an array bounds violation.

### Race condition in access to shared data!

|                                                                                                                   |                                                                                                            |
|-------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <pre> img_num++;  if(img_num &gt;= MaxImgs) img_num = 0; ... </pre> <p><b>MaxImgs is 10<br/>img_num was 9</b></p> | <pre> ... g.drawImage( theImages[img_num], 0, 0); ... </pre> <p><b>array bounds exception occurred</b></p> |
|-------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|

## Valid states of objects

- When multiple threads (processes) use shared data, have to be careful that their operations don't overlap in ways that cause an object to be seen in an invalid state.
- In example, the two steps  

```
img_num++;
if(img_num >= MaxImgs) img_num = 0;
```
- should be "atomic" – should only see object in state where neither operation performed, or both performed.
- If execution of awt thread occurs between steps, it may see object in invalid state.

## Locks

- A thread should *lock* object(s) when it needs
  - to make several changes to an individual object,
  - or to make changes to more than one object
- Locks prevent other threads from using the same objects (another thread trying to access a locked object gets put into "blocked" list of thread scheduler).
- Locks are held until all changes have been made in a correct, consistent manner.

## Locks

- Every object (class instance) has an individual lock.
- *Each class has a lock that can be used if need controlled access to (static) class variables.*
- Each member function of a class can specify that appropriate lock must be acquired for the affected instance (or class for a static member function) before thread seeking to invoke function can proceed.

## Locks

- Can also have a method of one object acquire a lock for another object – this useful if want thread executing the method to perform a sequence of operations on the other object without interference from other threads.

## synchronized

- Basic structure  

```
synchronized (anyObject) {
 code involving that anyObject
}
```
- behaviour
  - block until anyObject is not locked
  - lock anyObject
  - do code involving anyObject
  - release lock

## Example

- Affected object is "BankAccount"
  - owns a Funds value
  - has getFunds() and setFunds() methods
- Active objects are ATMs; their run() methods are  

```
account.getFunds()
decide on +ve or -ve transfer and amount
compute newAmount
account.setFunds(newAmount)
```
- Active objects display what they are doing in labels on screen
- Sleep requests could allow interleaved execution (meant to represent activities that take a long time in a real program)

```

import java.awt.*;
import java.awt.event.*;
class BankAccount { ... }
class ATM implements Runnable { ... }
public class ThreadLocked extends Frame implements
 WindowListener {
 ...
 public static void main(String[] argv) {
 ThreadLocked t1 = new ThreadLocked();
 }
}

```

```

class BankAccount {
 public BankAccount (int initial)
 { fFunds = initial; }
 public int getFunds() { return fFunds; }
 public void setFunds(int newFunds)
 { fFunds = newFunds; }
 private int fFunds;
}

```

```

public class ThreadLocked extends Frame implements
 WindowListener { ...
 public ThreadLocked() {
 BankAccount ba = new BankAccount(1234);
 ATM anATM;
 Label l = new Label("ATM 1 ");
 add(l, "North");
 anATM = new ATM(l, ba);
 Thread t1 = new Thread(anATM);
 l = new Label("ATM 2 ");
 add(l, "South");
 anATM = new ATM(l, ba);
 Thread t2 = new Thread(anATM);
 pack();
 show();
 addWindowListener(this);
 t1.start();
 t2.start();
 }
}

```

```

class ATM implements Runnable {
 public ATM(Label l, BankAccount b) {
 fLabel = l;
 fAccount = b;
 }
 public void run() {
 for(;;) { ... }
 }
 Label fLabel;
 BankAccount fAccount;
}

```

```

public void run() {
 for(;;) { int sleeptime;
 sleeptime = 500 + (int) (Math.random()*2000);
 try { Thread.sleep(sleeptime); } catch (InterruptedException ie) {}
 synchronized(fAccount) {
 fLabel.setText("Accessing account");
 try { Thread.sleep(1000); } catch (InterruptedException ie) {}
 int amount = fAccount.getFunds();
 int transfer = 10 + (int) (Math.random()*200);
 if(Math.random() < 0.5) transfer = -transfer;
 String s = new String("Account $" + amount + ", transfer $" + transfer);
 fLabel.setText(s);
 amount = amount + transfer;
 sleeptime = 200 + (int) (Math.random()*5000);
 try { Thread.sleep(sleeptime); } catch (InterruptedException ie) {}
 fAccount.setFunds(amount);
 fLabel.setText("Finished with account");
 }
 }
}

```

*Leave out synchronized, and watch the threads overwrite each other's changes*

## Lock placed by “synchronized()”

- Note: the lock placed by `synchronized(fAccount)` only affects those threads that check it!
- If there was another piece of code somewhere that invoked methods of the same `BankAccount` object, then a thread executing this code could proceed despite the fact that the account is “locked”.

## synchronized block example

- Problem in this case was that the calls
 

```
int amount = fAccount.getFunds(); ...
fAccount.setFunds(amount);
```
- really form a single operation.
- Hence the need for a lock that lasts across multiple calls.

## Locking out interfering threads in a single object

- Another common requirement is a mechanism to prevent two different threads from executing different (or the same) methods of an object from concurrently changing that object.
- Example:
  - program has indexable collection of data elements (Vector or similar)
  - can remove element at particular index, subsequent elements “move up one place” to fill gap
  - can add element at specified index, subsequent elements “moved down one place” to make gap

## Locking out interfering threads in a single object

- If two threads in same program tried simultaneously accessing the same Vector storage object
  - one doing a removeElementAt,
  - other doing an insertElementAt
 result would be chaos!

## Locking out interfering threads in a single object

```
void removeElementAt(int index) {
 synchronized(this) {
 ...
 // code that does remove
 ...
 }
}

void insertElementAt(
 Object obj, int index) {
 synchronized(this) {
 ...
 // code that does insert
 ...
 }
}
```

Bodies of two functions could start with a **synchronize** request that locks the object. One thread would get there first, eg remove thread, lock the Vector, do its remove. Other thread would have to wait.

## Locking out interfering threads in a single object

Er? Suppose we had code like the following:

```
void doSomething() {
 synchronized(this) {
 ...
 doSomethingElse();
 ...
 }
}

void doSomethingElse() {
 synchronized(this) {
 ...
 // code that does insert
 ...
 }
}
```

Won't it “deadlock”? !

No problems!

Synchronization lock is a form “recursive mutex”

## Recursive mutex

- Lock structure has record of current thread identifier.
- A second attempt to acquire lock will go ahead if the requesting thread is the same as the thread currently holding the lock.

```
lock
mutex_lock
lock_thread_id
count
```

This pseudo code for a “recursive mutex” is oversimplified!

```
get_lock
if(lock_thread_id
== current_thread_id()) { count++; return }
mutex_lock.Lock();
lock_thread_id=current_thread_id()
count=1
```

```
release_lock
count--;
if count>0 return;
lock_thread_id = none
mutex_lock.Unlock();
```

## “Syntactic sugar”

- Java provides an alternative mechanism for specifying the presence of such synchronization locks:

Vector:

```
public final synchronized void removeElementAt(int index) { ... }
public final synchronized void insertElementAt(...) { ... }
```

- Advantages:
  - making synchronization part of signature clarifies operations
  - less likely to make errors, synchronized code block is inserted by the compiler

## Java’s collections

- Newer ones, like LinkedList, don’t have synchronization on methods – your responsibility to lock linked list if there is chance of other threads trying to use it.
- Older ones, like Vector, are synchronized
- *Anomaly*
  - Synchronized keyword no longer appears in API specification for methods of Vector

## Java 1.5 Lock

- Java 1.5 has added a new Lock interface and some implementing classes
  - Use only if your algorithm requires more sophisticated locking mechanisms than the standard synchronized construct
  - (Lock allows for more complex patterns of lock acquisition and release – but increases the possibility of error. Only use if you are really confident about what you are doing.)

## Your code, shared objects

- **Either:**
  - if instances of class are used “transaction style” (several separate requests that must be handled atomically)
  - then all code using such objects must have **synchronized blocks** – claim object, use, release
- **Or:**
  - if requests to class instance are independent but concurrent requests could interfere
  - then make all mutative **member functions synchronized** (also any access functions that could retrieve inconsistent data)

## More sophisticated operations

wait() and notify()

## A small problem with locks ...

- As an example, consider a “bounded buffer”
  - putElement
  - getElement
- Member functions have potential to interfere
  - eg, you might have an elementCount;
    - ‘put’ does elementCount++;
    - ‘get’ does elementCount--;
    - although ++ and -- look atomic, they aren’t (load value onto stack, do an add/subtract 1 leaving result on stack, store from stack – with different concurrent threads doing same thing using different stacks)

## A small problem with locks ...

- So, you synchronize

```
synchronized void
putElement(Object o)
{
 while(elementCount==max)
 sleep(100);
 store[put_ndx] = o;
 put_ndx++;
 if(put_ndx == max) put_ndx = 0;
 elementCount++;
}
```

```
synchronized Object getElement()
{
 while(elementCount==0)
 sleep(100);
 Object o = store[get_ndx];
 get_ndx++;
 if(get_ndx == max) get_ndx = 0;
 elementCount--;
 return o;
}
```

*(not an ideal way of coding, I don't particularly like that wait loop; but this code is to illustrate a different point!)*

## A small problem with locks ...

- Thread using getElement() starts first, claims lock, looks inside buffer, finds it empty, decides to hang around until someone fills buffer.
- Thread using putElement() arrives, tries to access buffer, finds it locked, so blocks until lock released.
- **System stalls.**

## wait & notify

- A thread may acquire a lock on an object, do some work, then find it cannot proceed until the object is changed by another thread.
- When/if it is to resume, it will still need an exclusive lock on the object while it does the rest of its work.
- But since it now has the lock, no other thread can get in and change the object.
- Obviously, need some hack to get around this.

## wait and notify

- A thread that owns a lock on an object can “wait” on that object.
  - the thread is blocked (waiting for this object to change)
  - lock on object is released
- Another thread that later changes the object (which it can get at because the object isn't locked anymore) can do a “notify” operation on the object (the thread has to own the lock on the object when it does the notify operation).
- The thread marked as blocked waiting for the object to change can resume, when it resumes it again has a lock on the object.

## Java's wait & notify

- wait() and notify() are methods of Object
  - wait() wait forever
  - wait(long), wait(long, int) wait with timeout
    - if timeout elapses, this thread will resume as soon as it can get the lock on the object
  - notify()
  - allows another thread waiting on object to resume; if several other threads are waiting, then one is selected randomly!
  - notifyall()
  - allows all threads waiting on object to resume, they will be in contention for object so will get chances in some arbitrary order

## wait

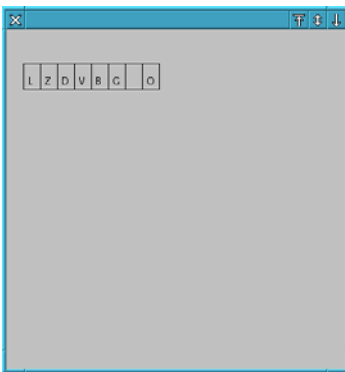
- Multiple threads waiting for same object?
  - does occur,
    - maybe waiting for object to be in different states
    - maybe waiting for object to be in same state (imagine several producers all putting data in same buffer, all waiting for there to be space in the buffer)
- wait() will typically be in a while clause –  
`while( ! condition_I_want_to_see) wait()`

## wait

- Having wait() in a while clause forces awakening thread to recheck condition; if object still not in desired state, thread blocks again.
- So notifyall() lets all threads check their conditions
  - if looking for different conditions, first satisfied thread will continue
  - in situation like several producers feeding one buffer, one will find it not full and proceed to fill it; others will then find buffer again full so wait
  - (thread has lock on return from wait(), is holding lock when rechecking condition in while, then either continuing with lock if happy with object state, or again doing a wait() so releasing its lock)

## Java --- thread interrupts

- Java allows a thread to “interrupt” another.
- A thread that is waiting may return from wait() because it has been interrupted rather than because it has received a notifyall or notify.
- wait() request will need a try ... catch ... relating to interrupt.



## Example

### *Bounded buffer with wait & notify*

- Circular buffer data structure
  - array of characters, ‘get’ index, ‘put’ index, count
- Consumer
  - runs at constant speed, fetch character from buffer
- Producer
  - runs intermittently
- *(Simple model of something like a printer controller, print mechanism runs at constant rate, computer sends characters in bursts)*

```
import java.awt.*;
import java.awt.event.*;
class BoundedBuffer { ... }
class MyCanvas extends Canvas { ... }
class Producer implements Runnable { ... }
class Consumer implements Runnable { ... }
public class buffer extends Frame implements WindowListener {
 ...
 public buffer() { ... }
 public static void main(String[] argv) {
 buffer b = new buffer();
 }
}
```

```
public class buffer extends Frame implements WindowListener {
 public void windowClosing(WindowEvent e) { System.exit(0); }
 ...
 public buffer() {
 BoundedBuffer bb = new BoundedBuffer(8);
 Canvas c = new MyCanvas(bb); bb.setCanvas(c); c.setSize(400,400);
 add(c,"Center"); pack(); show(); addWindowListener(this);
 Producer p = new Producer(bb);
 Consumer s = new Consumer(bb);
 Thread pThread = new Thread(p);
 Thread sThread = new Thread(s);
 sThread.start();
 pThread.start();
 }
}
```

```

class Producer implements Runnable {
 // Producer runs at random speed, adding a character about
 // every 1.2 seconds, (assuming there is space to add a character)
 public Producer(BoundedBuffer b) { fBuffer = b; }
 public void run() {
 for(;;) {
 int sleeptime = (int) (Math.random() * 2400);
 try { Thread.sleep(sleeptime); }
 catch (InterruptedException ie) {}
 char achar = (char) ((int)'A' + (int)(Math.random()*26));
 fBuffer.put(achar);
 }
 }
 private BoundedBuffer fBuffer;
}

```

```

class Consumer implements Runnable {
 // Consumer runs at constant speed, removing a character
 // every 1.5 seconds, (assuming there is a character to remove)
 public Consumer(BoundedBuffer b) { fBuffer = b; }
 public void run() {
 for(;;) {
 try { Thread.sleep(1500); }
 catch (InterruptedException ie) {}
 char discard = fBuffer.get();
 }
 }
 private BoundedBuffer fBuffer;
}

```

```

class BoundedBuffer {
 public BoundedBuffer(int size) { ... }
 void setCanvas(Canvas c) { fCanvas = c; }
 synchronized char get() { ... }
 synchronized void put(char ch) { ... }
 synchronized void paint(Graphics g) { ... }
 private Canvas fCanvas;
 private char[] fArray;
 private int fSize;
 private int fCount;
 private int fGetNdx;
 private int fPutNdx;
}

```

```

public BoundedBuffer(int size) {
 fArray = new char[size];
 for(int i=0;i<size;i++) fArray[i] = ' ';
 fSize = size;
 fCount = 0; fGetNdx = 0; fPutNdx = 0;
}
synchronized void paint(Graphics g) {
 int x0 = 20;
 int y0 = 40;
 int width = 20;
 int height = 30;
 for(int i=0; i < fSize; i++) {
 g.drawRect(x0, y0, width, height);
 g.drawChars(fArray, i, 1, x0 + 5, y0 + 25);
 x0 += width;
 }
}

```

*Wouldn't really matter if  
this unsynchronized*

```

synchronized char get() {
 while(fCount == 0) {
 try { wait(); }
 catch (InterruptedException ie) {}
 }
 char res = fArray[fGetNdx];
 fArray[fGetNdx] = ' ';
 fGetNdx++;
 if(fGetNdx==fSize) fGetNdx=0;
 fCount--;
 fCanvas.repaint();
 notifyAll();
 return res;
}

```

```

synchronized void put(char ch) {
 while(fCount == fSize) {
 try { wait(); }
 catch (InterruptedException ie) {}
 }
 fArray[fPutNdx] = ch;
 fPutNdx++;
 if(fPutNdx==fSize) fPutNdx=0;
 fCount++;
 fCanvas.repaint();
 notifyAll();
}

```

## Java 1.5 addition

- Java 1.5:
  - java.util.concurrent package has a number of classes
  - its ArrayBlockingQueue is a kind of real world version of this little bounded buffer class.
- Look at the concurrency library before committing yourself to writing code using wait(), notify() etc : what you need might already be there.

## join

- Another standard threads primitive is “join”
- Main thread launches thread to handle a subtask
- ...
- Main thread now needs results from that subtask; subtask thread may/may not have completed.
- Main thread does “join” with subtask.

## join

- Common situation –
  - main thread starts several threads,
    - each of these threads handles a subtask,
    - solution to whole problem is combination of partial results from subtasks
- Main thread can join each subtask in turn (doesn't really matter if they don't finish in sequence waited for)
- Solaris threads gives you option of a single join that can wait for all.