Data Races & Race Conditions

- A *data race* occurs when two concurrent threads access a shared variable
  - at least one access is a write and
  - the threads use no explicit mechanism to prevent the accesses from being simultaneous

- A *race condition* occurs when a program's correctness unexpectedly depends on the ordering of events

Race Condition

- If you run a program with a race condition, will you always get an unexpected result?
  - No! It depends on the scheduler
  - ...i.e., which JVM you're running
  - ...and on the other threads/processes/etc that are running on the same CPU
- Schedule-driven problems are hard to reproduce

Atomicity

- One way to prevent undesirable schedules is to ensure that the code in the two threads is *atomic*
  - Operations A and B are **atomic** with respect to each other if, from the perspective of the thread executing A, when another thread executes B, either all of B has executed or none of it has.
  - An **atomic operation** is one that is atomic with respect to all operations, including itself, that operate on the same state.

Locks

- Commonly used to enforce atomicity
  - Descends from semaphore construct in OS research & design
- Only one thread can hold a lock
  - Other threads block until they can acquire it
  - The operation of acquiring a lock is atomic
    - Cannot have a race on lock operations themselves!
- In Java every Object has (can act as) a lock
  - Called an *intrinsic lock*

Synchronized Statement

- **synchronized (obj) {** *statements* **}**
  - Acquires (*locks*) the **obj** intrinsic lock before executing statements in block
  - Releases (*unlocks*) the lock when the statement block completes, whether due to a break, return, exception, etc.

More on Locks

- Intrinsic locks are reentrant
  - The thread can reacquire the same lock many times
  - Lock is released when object unlocked the corresponding number of times
- No way to *attempt* to acquire an intrinsic lock
  - Either succeeds, or blocks the thread
  - Java 1.5 java.util.concurrent.locks package added separate locks with more operations (will discuss these later in the semester)

Reentrant Locking

- Consider following code used to do atomic updating of a bounded counter

```
public synchronized boolean isMaxed() {
  return (value == upperBound);
}

public synchronized void inc () {
  if (!isMaxed()) ++inc;
}
```

  - Without reentrant locking, every call to `inc()` would block forever!

Synchronization Style

- Internal sync. (class is thread-safe)
  - Have a stateful object synchronize itself (e.g., with synchronized methods). Robust to threaded callers
  - E.g., class Math.Random

- External sync. (class is thread-compatible)
  - Have callers perform synchronization before calling the object
  - If they don’t, behavior may be unpredictable

Quiz 5

```
public class Set extends Thread {
  static List lst = new ArrayList();
  String s;
  void add(String s) {
    synchronized (lst) { lst.add(s); }}

  boolean check(String s) {
    synchronized (lst) {
        return lst.contains(s);
    }
  }
  public void run() {
    if (!check(s)) add(s);
  }
  public void main(String args[]) {
    Worker thread1 = new Worker("hello");
    Worker thread2 = new Worker("hello");
    Worker thread3 = new Worker("goodbye");
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Is it possible to have
`lst ={ "hello", "goodbye", "hello"}`

A. Yes   B. No

Answer: Yes

- There are no data races
  - All accesses are synchronized
- There is a race condition
  - Race condition caused by a violation of atomicity.
  - We expect the output to be { “hello”, “goodbye” }
  - But in fact it could also be { “hello”, “hello”, “goodbye” }

Compound Actions

- This is an example of a compound action
  - A sequence of operations that need to be atomic
- Common examples
  - Read-modify-write
  - Check-then-act

Thread-Compatible class fixed

```
public class Worker extends Thread {
  static List lst = new ArrayList();
  String s;
  public void run() {
    synchronized (lst) {
       if (!lst.contains(s))
         lst.add(s);
    }
  }

  public void main(String args[]) {
    Worker thread1 = new Worekr (“hello”);
    Worker thread2 = new Worker(“hello”);
    Worker thread3 = new Worker(“goodbye”);
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Both contains() and add() are now guarded by a single synchronized block, making them atomic

Aspects of Synchronization

- Ordering
  - Ensuring that you aren't surprised by the order in which statements are executed

```
T1:
x = 5;
y = 6;
```

```
T2:
x = 0;
if (y == 6)
  System.out.println(x);
```

output: 0

What Will This Program Print?

```
public class NoVisibility{
  private static boolean ready;
  private static int number;
  private static class ReaderThread extends Thread {
    public void run() {
      while (!ready){
           Thread.yield();
      }
      System.out.println(number);
    }
  }
    public static void main(String[] args) {
      new ReaderThread().start();
      number = 42;
      ready = true;
   }
}
```

Possible output

a) 42
b) 0
c) Runs forever

Non-atomic 64-bit Operations

Java Language Specification (JSL-17.7)

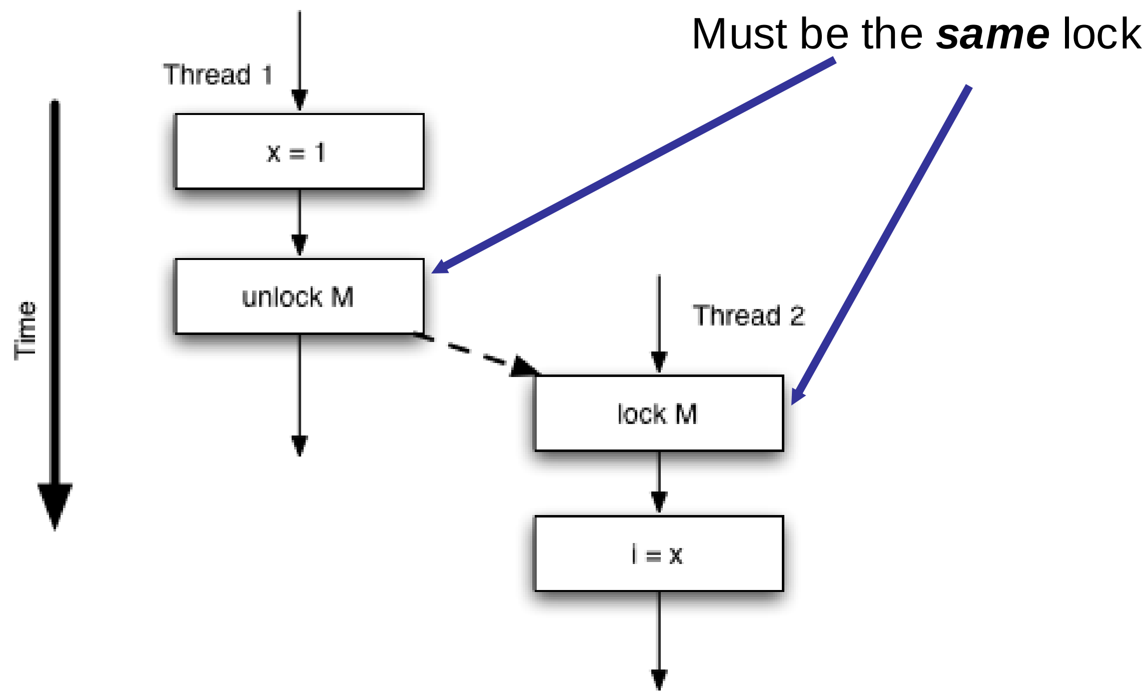
For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Long integer 64 bit

32 bit	**32 bit**
F F F F F F F F	0 0 0 0 0 0 0 0

For safe reads, writes of these variables, need synchronization

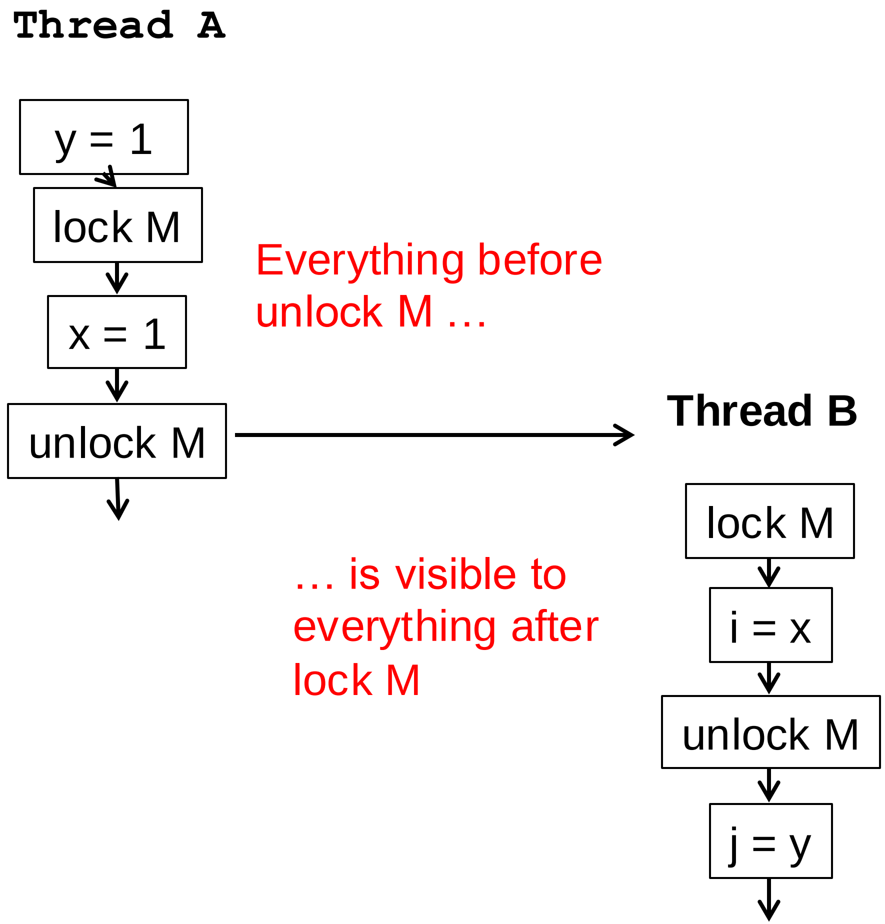
When Are Actions Visible?



Forcing Visibility of Actions

- All writes from thread that holds lock M are visible to next thread that acquires lock M
  - Must be the same lock
- When accesses are unsynchronized you get no guarantees
- One effect of synchronization is to enforce **visibility**

Locking and Visibility



Happens-Before

- "Happens before" is a partial order describing program events, invented by Leslie Lamport.

- Let A and B represent operations performed by a multithreaded process. If A ***happens-before*** B, then the memory effects of A effectively become visible to the thread performing B before B is performed.

Rules for Happens-Before

- Program order rule:
  - Each action in a thread happens-before every action in that thread that comes later in the program order.

- The program order rule guarantees that, *within individual threads*, reordering optimizations introduced by the compiler cannot produce different results from what would have happened if the program had been executed in sequentially.

```
x = 1;
y = 2
z = x + y
```

Happens-Before (x = 1; → y = 2)

Rules for Happens-Before

- Monitor lock rule:
  - An unlock on a monitor lock happens before every subsequent lock on that same monitor lock.

Thread A

```
Lock M1
 x = 100;
…

Unlock M1
```

Happens-Before

Thread B

```
Lock M1
 y = x;
…

Unlock M1
```

100 is visible

Rules for Happens-Before

- Volatile variable rule:
  - A write to a volatile field happens before every subsequent read of that same field.

Thread A

```
volatile boolean ready = false;
void update(){
  data = 100;
  count++;
  ready = true;
}
```

flushes to memory

Thread B

```
void consume(){
  while(!ready){
     //busy wait
  }
  send(data);
  count--;
  ready = false;
}
```

Data and count are visible to Thread B.

Rules for Happens-Before

- Thread start rule:
  - A call to Thread.start() on a thread happens before every action in the started thread.

Thread A

```
Thread t1 = new Thread()
 x = 100;
 …
 t1.start()
```

Happens-Before

Thread t1

```
public void run(){
  y = x;
  …

}
```

100 is visible

Rules for Happens-Before

- Thread termination rule:
  - Any action in a thread happens before another thread detects that thread has terminated, either by successfully return from Thread.join() or by Thread.isAlive returning false.

Thread A

```
Thread t1 = new Thread()
t1.start();
…



t1.join();
y=x
```

Thread t1

```
public void run(){
   x = 100;
   …

}
```

Happens-Before

100 is visible

Rules for Happens-Before

- Interruption rule:
  - A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt (either by having InterruptedException thrown, or invoking isInterrupted or interrupted).

```
Thread t1 = new Thread()
t1.start();
x = 100;
 …
t1.interrupt();
```

Happens-Before

Thread t1

```
public void run() {
  try {
    ...
    } catch (InterruptedException e) {
       y = x;
    }
  }
}
```

100 is visible

Rules for Happens-Before

- Finalizer rule:
  - The end of a constructor for an object happens-before the start of the finalizer for that object.

Thread A

```
public Frame(){
  data = 100;
  count++;
  ready = true;
}
```

Happens-Before

Thread B

```
public void finalize() {
  cleanup(data);
}
```

Rules for Happens-Before

- Transitivity:
  - If A happens-before B, and B happens-before C, then A happens-before C.

$$A < B \text{ and } B < C \;\rightarrow\; A < C$$

Example

Initially x == 0

	Thread 1:	Thread 2:	
①	x = 1	y = x;	②
④	y = 2;	③	

- R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)
- read(T2,x,0) does not happen-before write(T1,x,1)
- Both x==1 and x==0 are visible

Example

- So **y** can end up being {0,1,2}

R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)

R2 == write(T1,x,1); read(T2,x,1); write(T2,y,1); write(T1,y,2)

R3 == read(T2,x,0); write(T1,x,1); write(T2,y,0); write(T1,y,2)

R4 == write(T1,x,1); read(T2,x,1); write(T1,y,2); write(T2,y,1)

R5 == read(T2,x,0); write(T1,x,1); write(T1,y,2); write(T2,y,0)

Data Races

- The happens-before relation allows us to formally define data races
- A data race takes place when there are two events in trace R that
  - access the same memory location
  - at least one is a write
  - they are unordered according to happens-before

Data Race

Initially `x = 0`

Thread 1:
```
x = 1;
y = 2;
```

Thread 2:
```
y = x;
```

- R1 == write(T1,x,1); read(T2,x,0); write(T2,y,0); write(T1,y,2)
- Happens-before
  - write(T1,x,1) <: write(T1,y,2) and read(T2,x,0) <: write(T2,y,0)
- Data races between
  - write(T1,x,1) and read(T2,x,0)
  - write(T1,y,2) and write(T2,y,0)

Using Volatile

- A one-writer/many-reader value
  - Simple control flags:
    - volatile boolean done = false;

- Keeping track of a “recent value” of something

Limitations

- Incrementing a volatile field is not atomic
  - In general, writes to a volatile field that depend on the previous value of that field don't work
- A volatile reference to an object isn't the same as having the fields of that object be volatile
  - No way to make elements of an array volatile
- Can't keep two volatile fields in sync

Example

```
class Test {
  static int i = 0, j = 0;
  static void one() { i++; j++; }
  static void two() {System.out.print("i=" + i + " j=" + j);}
}
```

- Thread A calls Test.one() repeatedly
- Thread B calls Test.two() repeatedly
- Can the printed value of `j` ever be greater than that of `i`?
  - Yes. This is completely unsynchronized.

Example

```
class Test {
  static int i = 0, j = 0;
  static synchronized void one() { i++; j++; }
  static synchronized void two() {
       System.out.println("i=" + i + " j=" + j);
  }
}
```

- How about now?
  - No. `i` and `j` are updated and read in apparent textual order

Example

```
class Test {
  static volatile int i = 0, j = 0;
  static void one() { i++; j++; }
  static void two(){
       System.out.print("i=" + i + " j=" + j);
  }
}
```

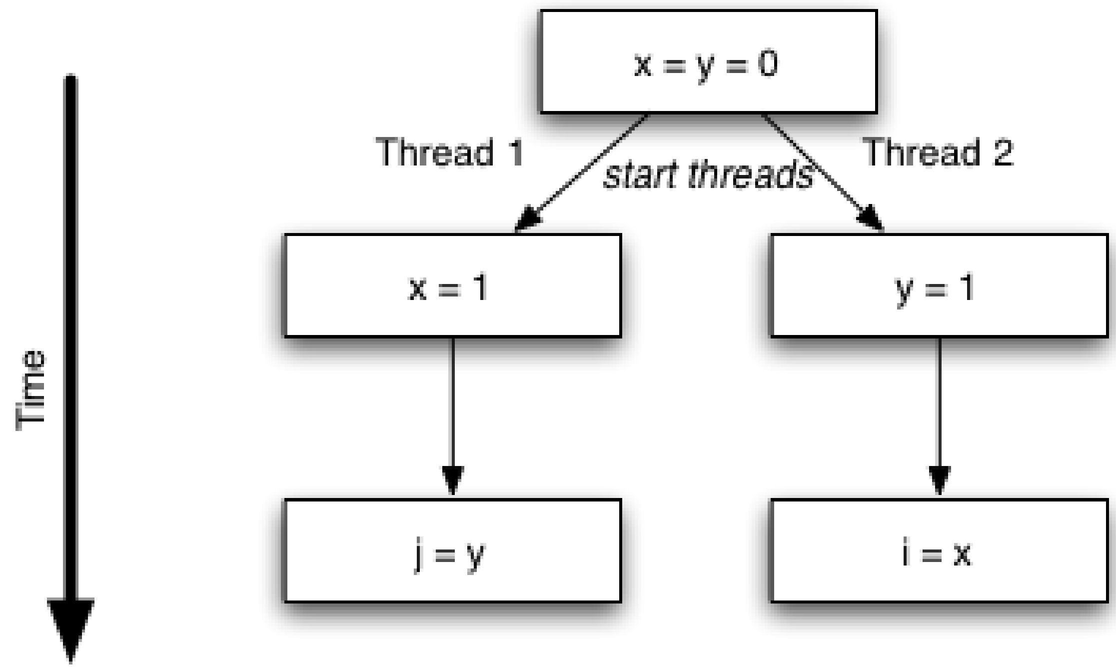
- How about now?

Example

```
class Test {
  static volatile int i = 0, j = 0;
  static void one() { i++; j++; }
  static void two(){   ①     ③ ③ ③
      System.out.print("i=" + i + " j=" + j);
  }                           ②        ④
}
```

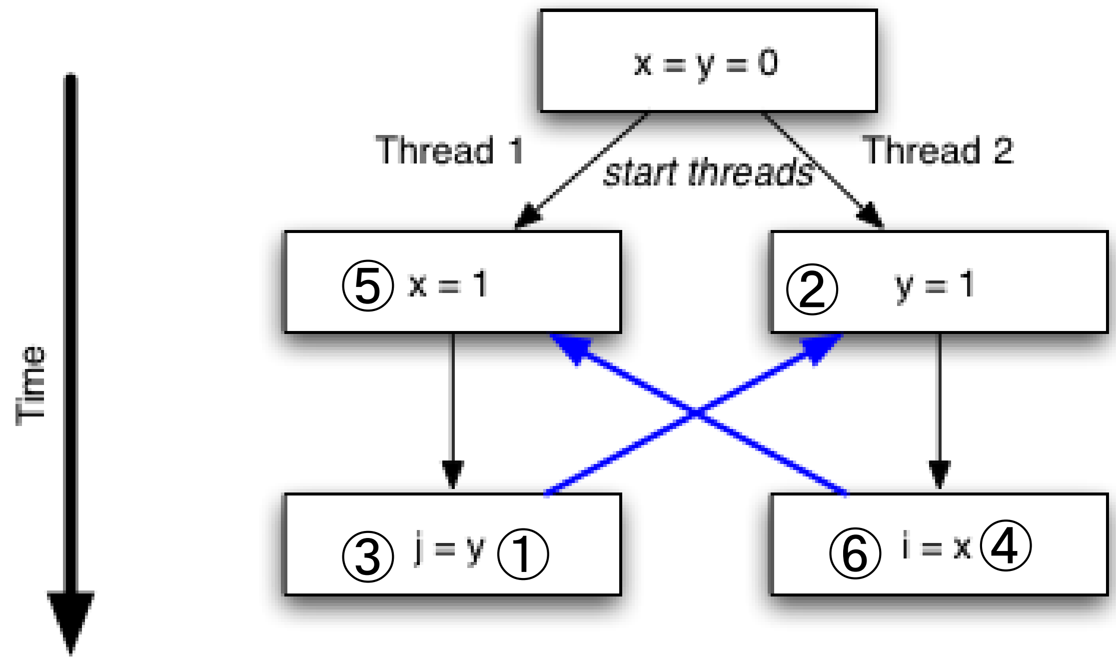
- How about now?
  - `j always >= i-1`, but could be a lot bigger
  - e.g., `one()` could be called many times between the time `two()` access the value of `i` and then accesses the value of `j`.

Quiz Time



- Can this result in i=0 and j=0?

Doesn’t Seem Possible...



- But this can happen!

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor systems, values not synchronized in global memory

Synchronization: Deadlocks

Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*
  - A set of threads is *deadlocked* if each thread is waiting for an event that only another thread in the set (including itself) can cause.

Synchronization not a Panacea

Two threads can block on locks held by the other; this is called *deadlock.*

```
Object A = new Object();
Object B = new Object();
```

```
T1.run() {
  synchronized (A) {
    …
    synchronized (B) {
      …
    }
  }
}
```

```
T2.run() {
  synchronized (B) {
    …
    synchronized (A) {
      …
    }
  }
}
```

Synchronization not a Panacea

Two threads can block on locks held by the other; this is called *deadlock.*

```
Object A = new Object();
Object B = new Object();
```

```
T1.run() {
  synchronized (A) {
    ...
    synchronized (B) {
      ...
    }
  }
}
```

```
T2.run() {
  synchronized (B) {
    ...
    synchronized (A) {
      ...
    }
  }
}
```

Context switch

Deadlock

- Easy to write code that deadlocks
  - Thread 1 holds lock on **A**
  - Thread 2 holds lock on **B**
  - Thread 1 is blocked trying to acquire lock on **B**
  - Thread 2 is blocked trying to acquire lock on **A**
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - Other than by the fact that nothing is happening

Solution

- A program will be free of lock-ordering deadlocks
  - if all threads acquire the locks they need in a fixed global order.
    - For example:
      - Always acquire left lock before right lock
      - Requires a global analysis of your program's locking behavior
  - If all threads lock all the required resource at once
    - All required resources are available
    - Not efficient, difficult to get all the resources

Example

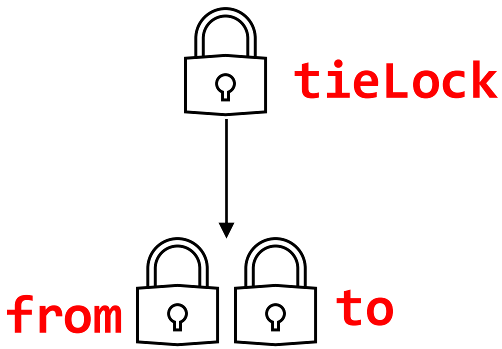
```
public static void transferMoney(
        Account fromAcc, Account toAcc, int amount){
         synchronized (fromAcc) {
             synchronized (toAcc) {
                   fromAcc.debit(amount);
                   toAcc.credit(amount);
             }
         }
}
```

A: TransferMoney(checking, saving, 10);
B: TransferMoney(saving, checking, 20);

No control over the lock order. Depends on the caller

Possible Solution

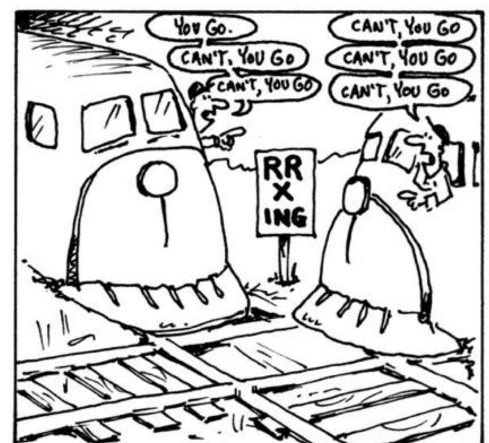
```
 int fromHash = System.identityHashCode(fromAcct);
 int toHash = System.identityHashCode(toAcct);
 if (fromHash < toHash) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                    new Helper().transfer();
                }
            }
}else if (fromHash > toHash) {
            synchronized (toAcct) {
               synchronized (fromAcct) {
                     new Helper().transfer();
               }
            }
} else {
      synchronized (tieLock) {
          synchronized (fromAcct) {
                synchronized (toAcct) {…
}
```



The Deadlock problem

- There is a Kansas law still in existence which reads:

When two trains approach each other at a crossing, they shall both come to a full stop and neither shall start up until the other has gone.

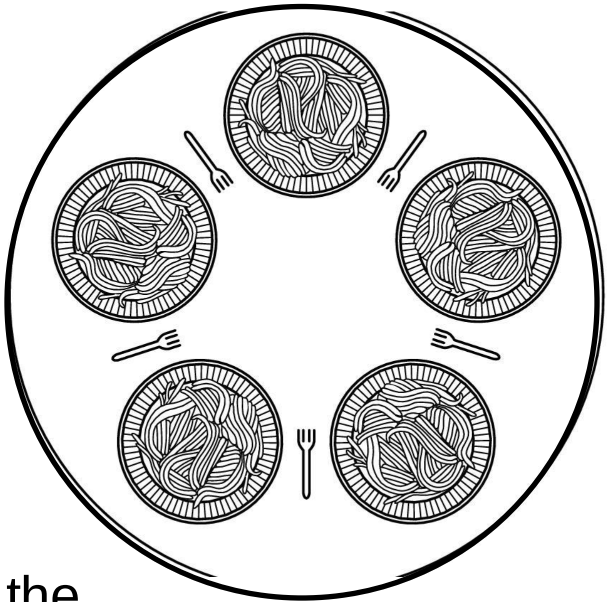


Open Calls

- Calling a method with no locks held is called open call.
- Classes rely on open calls are more well-behaved than classes that make calls with locks held.
- Use synchronized when it is necessary.
  - To achieve atomicity

Dining Philosopher's Problem

- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time



Formulated in 1965 by Dijkstra to capture the concept of multiple processes competing for limited resources

Rules of the Game

- The philosophers are very logical
  - They want to settle on a shared policy that all can apply concurrently
  - They are hungry: the policy should let everyone eat (eventually)
  - They are utterly dedicated to the proposition of equality: the policy should be totally fair

What can go wrong?

- Primarily, we worry about:
  - Deadlock: A policy that leaves all the philosophers “stuck”, so that nobody can do anything at all
  - Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
  - Livelock: A policy that makes them all do something endlessly without ever eating!

Solution 1

What is your solution?

Solution 1

- Learn to eat with one fork (sanitary solution)

Solution 1

- Learn to eat with one fork (sanitary solution)
- Buy more forks
- A philosopher can eat only if the neighbors are not eating
- Solutions are less interesting than the problem itself!

Solution 2 (flawed)

Each Philosopher:

```
while(true) {
    think();
    take_fork(i);    //left
    take_fork((i+1)%5);  //right
    eat();
    put_fork(i);
    put_fork((i+1)% 5);
}
```

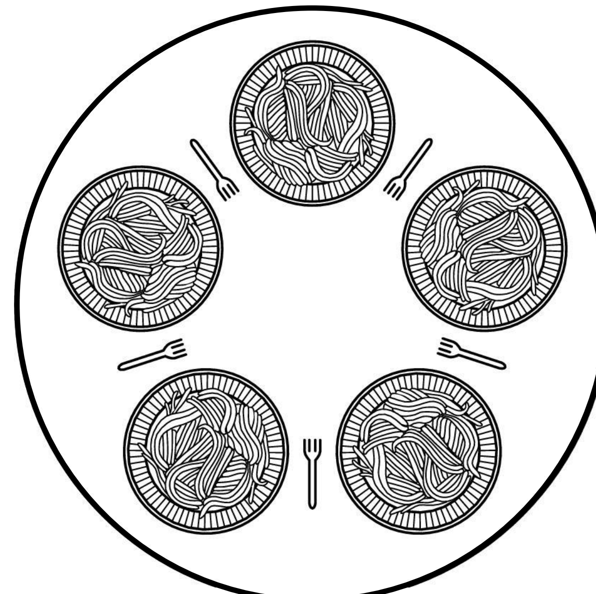
Solution in Java (flawed)

```
Philosopher[] philosophers=new Philosopher[5];
Object[]forks=new Object[5];

for(inti= 0; i < forks.length; i++) {
     forks[i] =newObject();
}

for(inti= 0; i < 5; i++) {
  Object lFork= forks[i];
  Object rFork= forks[(i + 1) % 5];
  philosophers[i] =new Philosopher(lFork, rFork);
  Thread t = new
       Thread(philosophers[i],"Philosopher "+(i+ 1));
  t.start();
}
```

Flawed solution?

Oops! Subject to deadlock if they all pick up their “right” fork simultaneously!

```
while(true) {
    think();
    take_fork(i);
        //all wait
    take_fork((i+1)%5);
    eat();
    put_fork(i);
    put_fork((i+1)% 5);
}
```



Solution 3

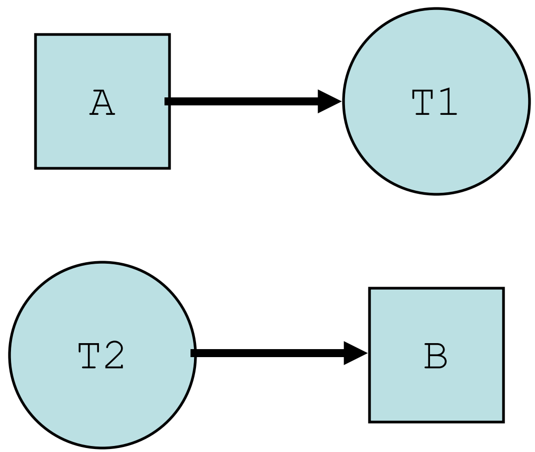
- Allow only 4 philosophers to sit simultaneously
- Asymmetric solution
  - Odd philosopher picks left fork followed by right
  - Even philosopher does vice versa
- Pass a token
- Allow philosopher to pick fork only if both available

Deadlock Conditions

- For deadlock to occur the following conditions must hold simultaneously

1. Mutual exclusion: a non-sharable resource exists
2. Hold and wait: processes already holding resources may request new resources held by other processes
3. No preemption: No resource can be forcibly removed from a process holding it
4. Circular wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds

Deadlock: Wait graphs

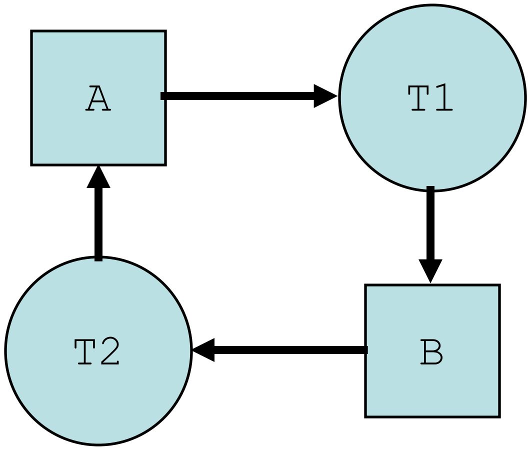


Thread T1 holds lock A

Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph

Wait graph example



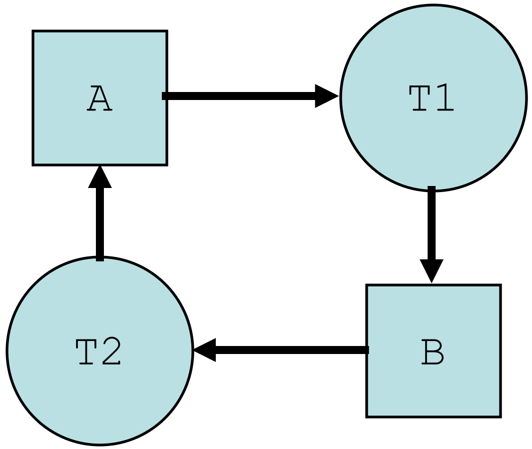
T1 holds lock on **A**

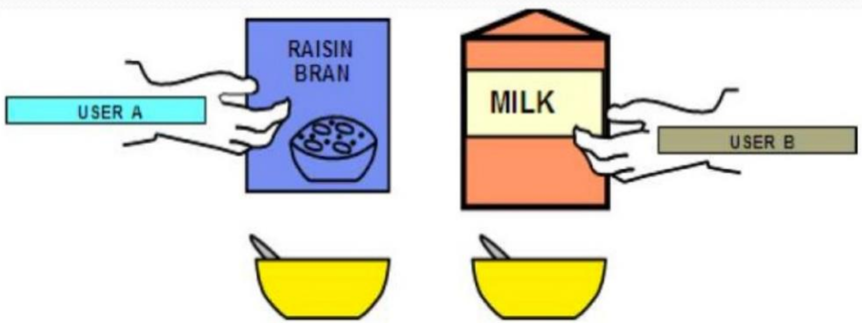
T2 holds lock on **B**

T1 is trying to acquire a lock on **B**

T2 is trying to acquire a lock on **A**

Wait graph example





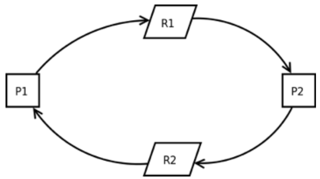
Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or effectively in parallel on a single processor
    - Assuming a running thread can be preempted at any time
- Threads can share data
  - Need to prevent interference
    - Synchronization, immutability, and other methods
  - Overuse use of synchronization can create deadlock
    - Violation of liveness

Cyclic wait

- For example… consider a deadlock
  - Each philosopher is holding one fork
  - … and each is waiting for a neighbor to release one fork
- We can represent this as a graph in which
  - Nodes represent philosophers
  - Edges represent waiting-for

Cyclic wait



- We can define a system to be in a deadlock state if
  - There exists ANY group of processes, such that
  - Each process in the group is waiting for some other process
  - And the wait-for graph has a cycle

- Doesn’t require that every process be stuck… even two is enough to say that the system as a whole contains a deadlock (“is deadlocked”)

Real World Deadlocks?

- Gridlock

Avoiding the Deadlock

- Never acquire more than one lock
  - Not always practical
- Ordering the locks
- Open calls

Diagnosing the Deadlock

- Timed Lock
  - trylock
  - When a timed lock attempt fails, releases all locks and wait again
- Database
  - When cycle detected, arbitrarily kills a transaction
- Java does not detect the deadlock
- Thread dumps: debugging

Starvation

Philosopher can go hungry in some situation even though they collaborate

Starvation

- Avoid the temptation to use thread priorities.

- They are platform dependent

Livelock

- Philosophers may have to think endlessly without ever eating!
- The issue is that processes may be active and yet are “actually” waiting for one-another in some sense
- Need to talk about whether or no processes make progress
- Once we do this, starvation can also be formalized

Livelock Example

```
int couner = 5
```

T1:

```
while(counter.get()<10) {
        counter.inc();
}
```

T2:

```
while(counter.get()>0) {
        counter.dec();
}
```

Livelock

- Carrier-sense multiple access with collision detection(CSMA/CD)

- Two polite friends

Publishing and Escape

- *Publishing* an object: making it available to parts of a program outside the scope in which it was created
  - Sometimes you want to
  - Other times you don't
- Object *escape*: unintended (or poorly considered) publishing
  - Source of many subtle errors
  - Problems can be especially tricky in presence of threads

Perils of Publishing (1/2)

Consider slight modification to Line class

```
public class BadLine {
  //@Invariant:  p1 and p2 must be different points
  private MutablePoint p1;
  private MutablePoint p2;
  // BadLine throws exception if points overlap
  BadLine(MutablePoint p1, MutablePoint p2) throws IllegalArgumentException{
            … error checking … }
  …
  MutablePoint getP1() { return p1; }
  MutablePoint getP2() { return p2; }
  …
}
```

Here is the MutablePoint class

```
public class MutablePoint {
  private double x;
  private double y;
  …
  public void setX(double newX) { x = newX; }
  public void setY(double newY) { y = newY; }
}
```

Obvious Forms of Publishing

- Assigning to a public field
  - Consider

```
public class ReallyBadLine {
  public MutablePoint p1;
  …
}
```

  - Really bad idea: don't do this (almost impossible to enforce correctness)
- Via getters (cf. BadLine)
  - Using getters is better than using public fields
  - Remember that once an inner object is obtained by alien code, an enclosing object loses control

Indirect Publishing (1/2)

- Publishing an object also publishes any objects accessible from that object
- Consider (from book)

```
class UnsafeStates {
  private String[] states = new String[] {“AK”, “AL”, …};

  public String[] getStates() {
    return states;
  }
}
```

  - `getStates()` publishes private field states, which can now be modified (probably not what is intended)
  - It also publishes all the `String` objects in the `states` array as well
- *Indirect publishing is the most common form of escape!*

Indirect Publishing (2/2)

- Nested classes can give rise to a subtle form of indirect publishing
  - Inner objects have a reference to outer, enclosing objects
  - This is stored in a hidden field `this$0`
  - There are means to access `this$0`
  - So: publishing an inner object indirectly publishes its enclosing object also

Outer / Inner Object Example

- Consider class Outer

```
public class Outer {
  private int a = 1;
  public void foo() {
          System.out.println ("Outer a = " + a);
  }
  public class Inner {
    private int b = a + 1;
    public void foo() {
     System.out.println ("Inner b = " + b);
    }
  }
}
```

Outer / Inner Object Example

- Now consider (credit to: http://stackoverflow.com/questions/763543/in-java-how-do-i-access-the-outer-class-when-im-not-in-the-inner-class)

```
import java.lang.reflect.Field;
public class OuterInnerTest {
  public static void main(String[] args) {
    Outer.Inner v = new Outer().new Inner();
    v.foo ();
    Field outerThis = v.getClass().getDeclaredField("this$0");
      Outer u = (Outer)outerThis.get(v);
      u.foo();
  }
}
```

- What gets printed?

  inner b = 2

  Outer a = 1

  - Outer object is available, even though it is not directly published

Multi-Threading and Escape

- Escape is especially problematic in the presence of threads
  - The usual issues of thread-safety are especially evident when an object escapes
  - There is also an issue with incompletely constructed objects being visible to other threads!
- Examples follow

Morals

- Object is only fully constructed when constructor terminates
- Don't let `this` escape during object construction!
  - Don't do it!
  - Book: object is *improperly constructed* when this is the case
- Related point
  - Don't start threads inside constructors
  - Reason: very easy to publish `this` to such threads

Overhead of Sharing Objects

- Sharing objects among threads imposes costs
  - Thread-safety must be implemented explicitly
  - This involves locking
  - Locking incurs run-time overhead, programming complexity

Thread Confinement

- One way to minimize complexity: don't share!
  - Of course, some sharing is needed
  - However, objects that are confined to a single thread are guaranteed to be thread-safe
  - Many graphical-user-interface (GUI) follow this paradigm
    - There is a single thread handling events
    - Applications put events into event queue
    - Handler repeatedly checks event queue, calls appropriate handler
    - Objects that only reside in handler need not be synchronized

Stack Confinement

- Local variables belong to a single thread, by definition
  - Local variables live on the stack
  - In Java, only the heap is shared
- Objects will be *stack confined* if they are:
  - Created in a thread
  - Assigned to a local variable in the thread
  - Never published

ThreadLocal

- Another mechanism for localizing objects in threads so that thread-safety is guaranteed
  - A ThreadLocal object can be seen as a container for other objects, e.g.
    - `ThreadLocal<List<Long>> idList;`
    - **`idList`** is a `ThreadLocal` object containing several **`List<Long>`** objects
  - Each thread accessing a ThreadLocal object is given its own variable pointing to a contained object

  E.g. any thread accessing `idList` is given its own local `List<Long>` variable by `idList`

ThreadLocal API

- Key methods for `ThreadLocal<T>`
  - **`public T get()`**
    Get instance of `T` associated with thread executing `get()`
  - **`public void set(T e)`**
    Change thread's instance of `T` to e
  - **`protected T initialValue()`**
    Define initial value associated with a thread (called when `get()` invoked first time, provided `set()` not called previously)
  - **`public void remove()`**
    Remove object associated with thread
- How to define **`initialValue()`**? Usually via anonymous inner classes

Immutability

- Synchronization incurs overhead
  - Locking reduces performance
  - Ensuring thread-safety makes code more complex
- How to reduce overhead?
  - Don’t share objects among threads if you don’t have to
  - Use *immutable* objects whenever you can!

Immutable Objects

- Why do we need synchronization? To cope with changes to object state
  - If fields in a method are modified while a method executes, the invariants in the class spec might be temporarily invalidated
  - Without synchronization these invalid values are visible to threads with access to the object
- If object's don't change, then there is no need to synchronize!
  - If invariant holds when object is created, then they are guaranteed to remain true
  - *Immutable objects* have this property: once they are created, their state never changes

Mutability and Visibility

- Final fields change values once!
  - When a constructor is first called, fields are allocated and given default values
  - As the constructor executes, new values are computed and assigned to fields
- If a constructor publishes `this`, then another thread might see the value of a `final` field before it has been assigned to.

Immutability Redefined

- An object is *immutable* if
  - All its fields are `final`
  - Its state never changes after construction
  - It is *properly constructed*: `this` does not escape during construction
- If an object is immutable, then:
  - it is thread-safe
  - it may be safely accessed / published without synchronization!

Immutability and Visibility

- What guarantees visibility of assignments to final fields in immutable objects?
- Answer: the Java Memory Model
  - If an object's fields are all `final` …
  - … then the JMM says that all writes to these fields are immediately visible, as are all memory writes that happen before it
  - This is like behavior of volatile variables!
- This property is called *initialization safety*

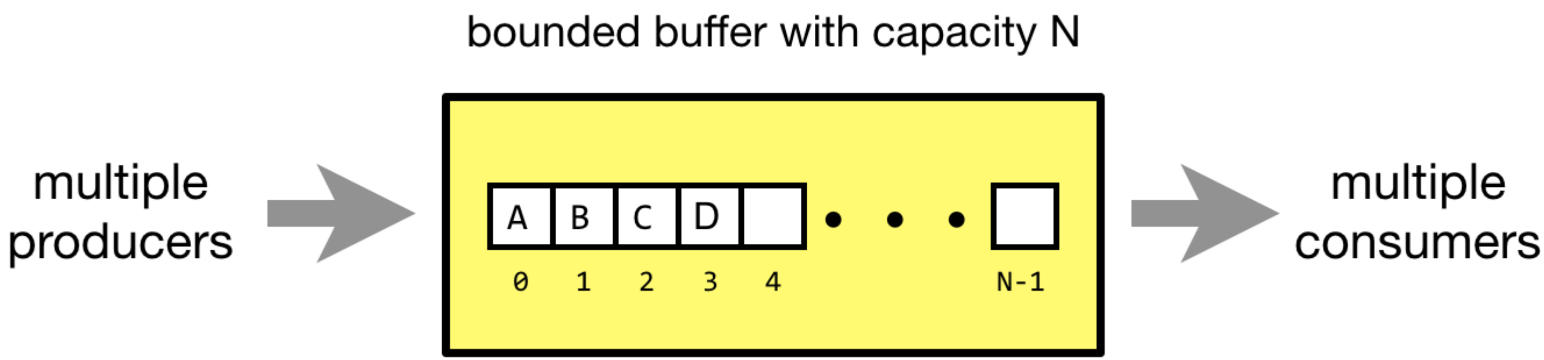
Guarded suspension

- **Guarded suspension** is a software design pattern for managing operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.

```
void stateDependentMethod(){
    synchronized(lock){
        while(!conditionPredicate())
            lock.wait();
        //now in a desired state
    }
}
```

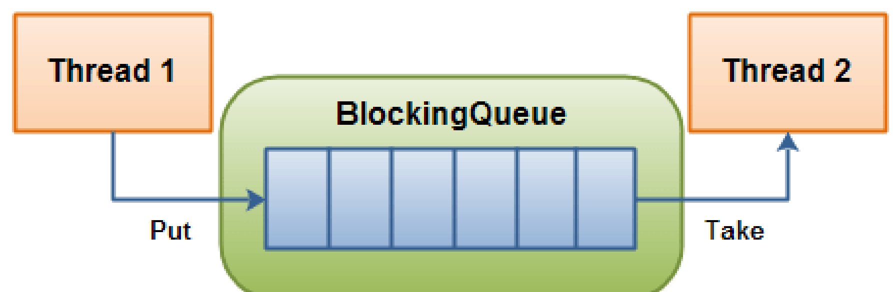
Guarded Suspension

- For bounded buffers in a multithreaded environment:
  - If the buffer is empty, a `take()` operation cannot complete
  - Another thread could deposit an element later, and a `take()` could succeed!



Guarded Suspension

- In guarded-suspension approaches to state-dependent actions, threads “go to sleep” until the actions they want to perform are possible
- Needed mechanisms
  - … for going to sleep (“suspend”)
  - … for waking up (“resume”)



Busy-Waiting

- An old-fashioned mechanism for suspend/resume
  - Use a while loop to test for enabled-ness of state-dependent action
  - When true: exit loop, perform action
  - E.g.

```
while (!enabled) ; // Suspend via spinning
// Resume
```

- Considerations
  - Consumes computing resources
  - Enabled-ness condition might become false after loop terminates, so synchronization should be used

wait() / notify() / notifyAll()

- A more modern mechanism in Java for suspending / resuming
  - To suspend, a thread performs a **`wait()`**
  - Other threads perform **`notify()`** / **`notifyAll()`** to enable resumption of suspended threads

wait() / notify() / notifyAll()

- Benefits
  - No consumption of cycles while suspended
  - Synchronization taken care of (we will see how in a moment)

- Dangers
  - A suspended thread is dependent on other threads to wake it up
  - If no other thread performs `notify()` / `notifyAll()`, then thread sleeps forever

Example: `BoundedBufferWait`

```
// Pre:  number of elements is below maxSize
// Post:  elt is added to end of elements, waiting threads notified
// Exception:  If number of elements is too high, suspend.
 public synchronized void put (Object elt) throws
                                 InterruptedException {
   while (elements.size() == maxSize) wait();
   elements.add(elt);
   notifyAll();
 }
```

- In **`put()`** / **`take()`** operations, **`wait()`** executed when state does not allow action
- When an operation succeeds, waiting threads notified

Example: `BoundedBufferWait`

- When a thread wakes up, it must check that condition it was waiting for holds!
  - This is why loop is used with `wait()` inside. You should do this always unless you have an ironclad argument for not needing a loop!
  - Just because a thread is resumed does not mean it is safe to proceed
- When a thread modifies the state of the object (e.g. by successfully adding an element) it must notify sleeping threads
- `InterruptedException`?
  - **`wait()`** is a blocking operation, meaning it could never terminate
  - Any thread can be interrupted (a topic for a later date) by another thread
  - This exception is raised in this case, because a blocked thread may need some cleanup

notify() / notifyAll()

- Consider **take()** operation in **BoundedBufferWait**

```
public synchronized Object take() throws InterruptedException {
        while (elements.size() == 0) wait();
        Object elt = elements.get(0);
         elements.remove(0);
         notifyAll();
         return elt;
}
```

- Doesn't this introduce a race condition?
  - notifyAll() called before return of element
  - Could this cause problems?
- Answer: no
  - notify() / notifyAll() do not release locks
  - So lock on buffer only released when **take()** operation terminates

Why `notifyAll()`?

- `put()` / `take()` use `notifyAll()` rather than `notify()`
  - It seems wasteful to wake everyone up!
  - Why not just wake up one thread?
- There is a reason!
  - Waiting threads are potentially concerned with different conditions
    - Putters are waiting for buffer not to be full
    - Takers are waiting for buffer not to be empty
  - If you use `notify()`, you only wake up one thread
  - If you wake up the wrong thread, you can wind up in a deadlock!

Producer-Consumer Simulation

```
void take(){
  synchronized(buffer){
    while(buffer.empty) {
            buffer.wait();
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```

Consumer

Producer

Buffer

Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          Print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```

Output:

c1.A

c2.A

Buffer

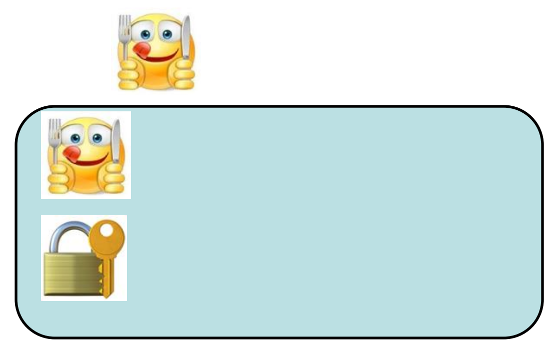
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
   while(buffer.empty) {
         print “B”
         buffer.wait();
         Print “C”
   }
   buffer.full = false;
   buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Buffer

Output:
c1.A
c2.A

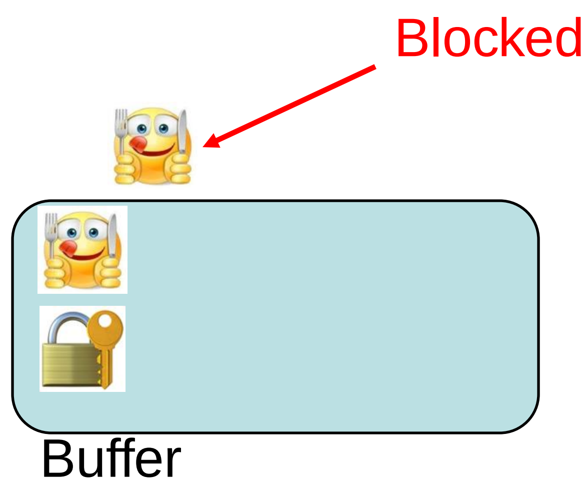
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          Print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Output:
c1.A
c2.A

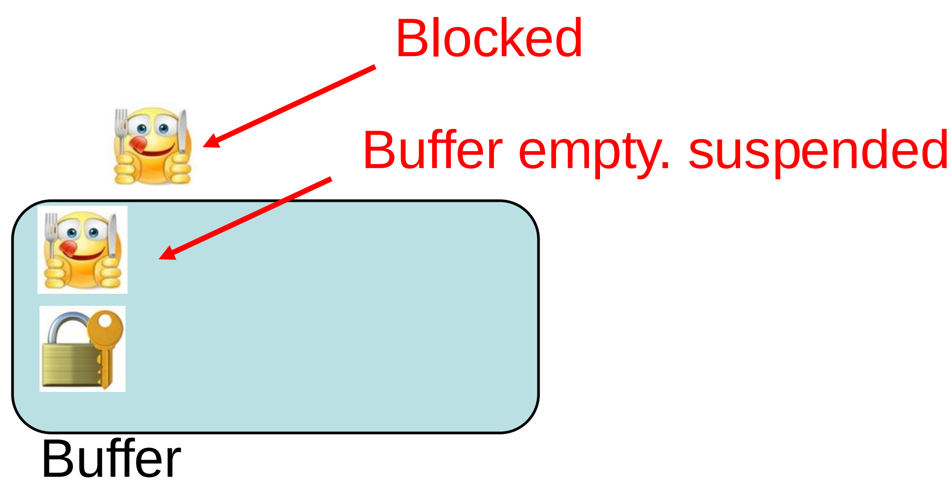
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          Print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Output:
c1.A
c2.A
c1.B

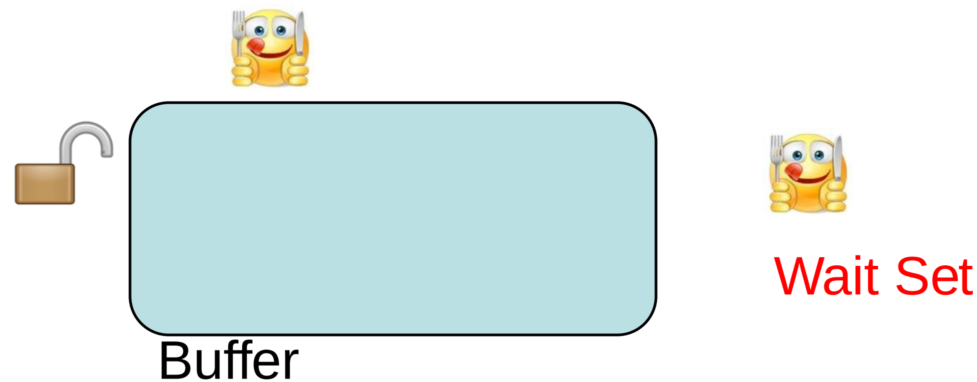
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
  ───→    buffer.wait();
          Print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Output:
c1.A
c2.A
c1.B

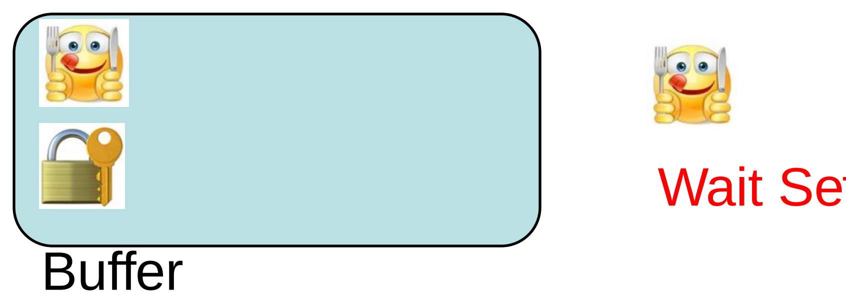
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          Print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Output:
c1.A
c2.A
c1.B

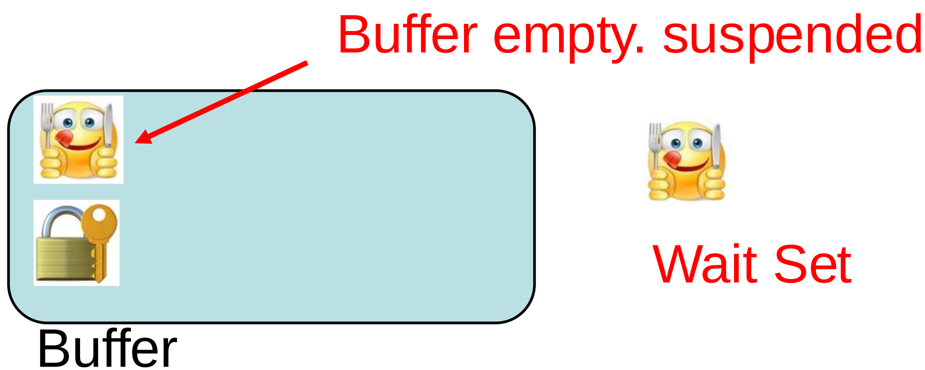
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
         print “B”
  -->    buffer.wait();
         print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Output:
c1.A
c2.A
c1.B
c2.B

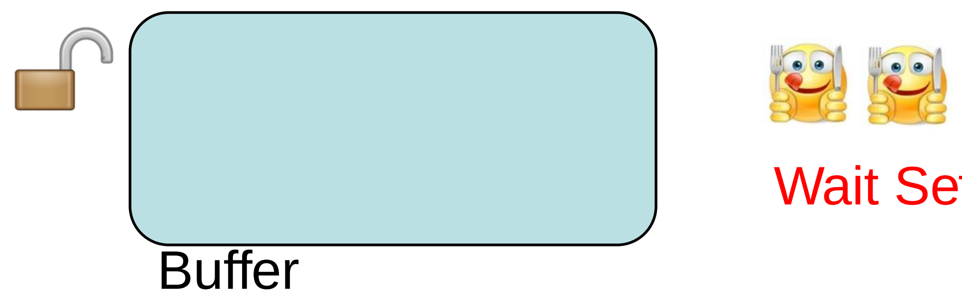
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();
```



Output:
c1.A
c2.A
c1.B
c2.B

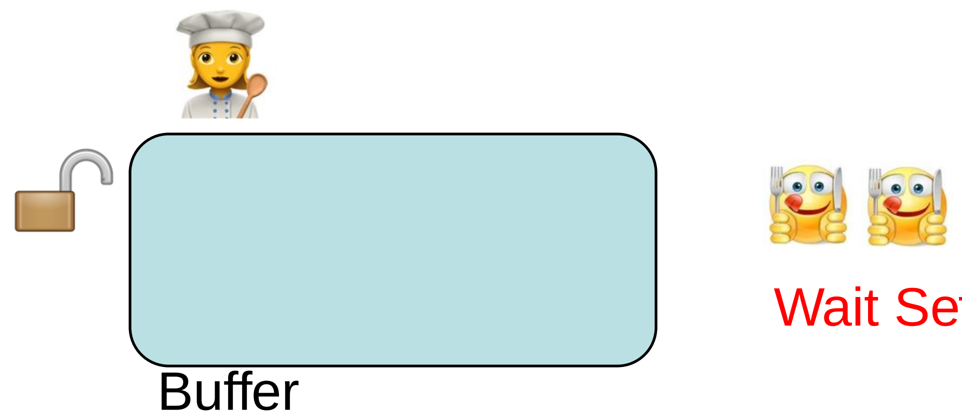
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
  ────▶   buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
c2.start();

Thread p1 = new Producer("Producer",buffer);
p1.start();
```



Output:
c1.A
c2.A
c1.B
c2.B

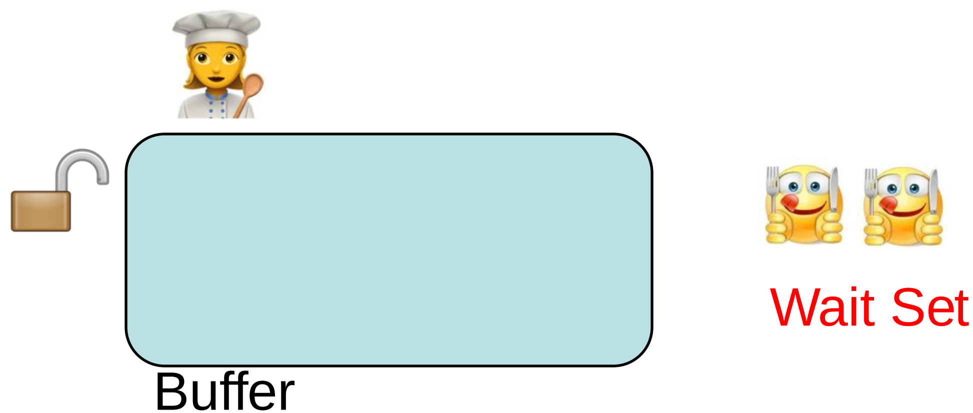
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
→ synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```



Output:
c1.A
c2.A
c1.B
c2.B

Producer-Consumer Simulation

Consumer:

```
void take(){
  print "A"
  synchronized(buffer){
    while(buffer.empty) {
          print "B"
          buffer.wait();
          print "C"
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```

Output:
c1.A
c2.A
c1.B
c2.B

Buffer

Wait Set

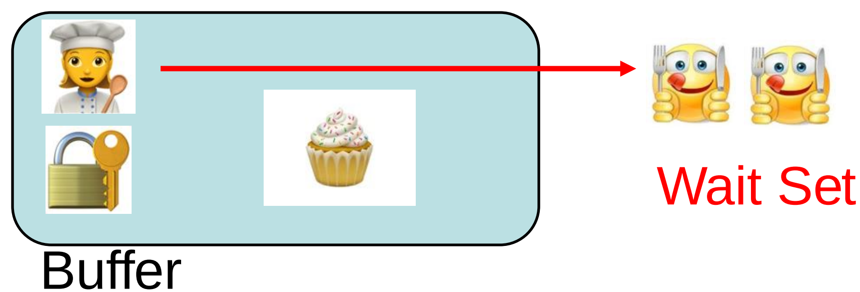
Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
→   buffer.notify();
  }
}
```



Output:
c1.A
c2.A
c1.B
c2.B

Producer-Consumer Simulation

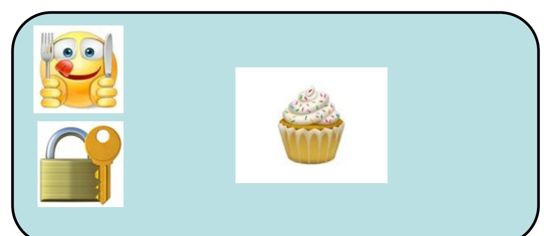
Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
        print “B”
        buffer.wait();
  ───→  print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
        buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```

Output:
c1.A
c2.A
c1.B
c2.B
c1.C


Buffer


Wait Set

Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```

Output:
c1.A
c2.A
c1.B
c1.B
c1.C

Producer-Consumer Simulation

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```

Output:
c1.A
c2.A
c1.B
c1.B
c1.C

Producer-Consumer Simulation

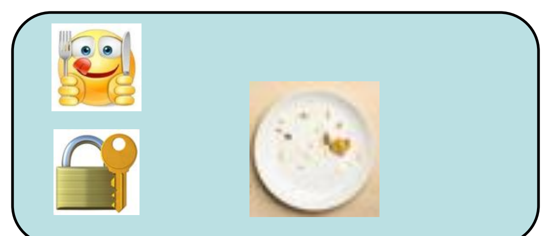
Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```

Output:
c1.A
c2.A
c1.B
c1.B
c1.C
c1.C
c2.C



Buffer

Producer-Consumer Simulation

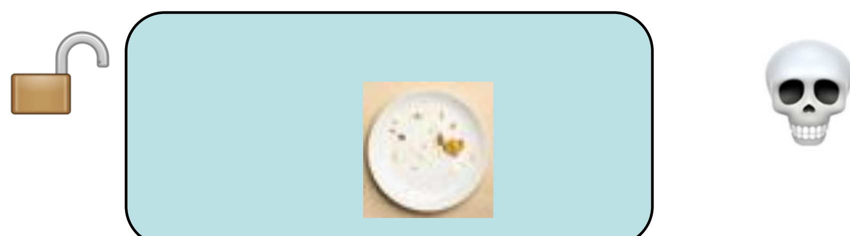
Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Output:
c1.A
c2.A
c1.B
c1.B
c1.C
c2.C
c2.B

Producer:

```
void put(){
  synchronized(buffer){
    while(buffer.full) {
          buffer.wait();
    }
    buffer.full = true;
    buffer.notify();
  }
}
```



Buffer

Quiz 1

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
C1.sleep(100);
c2.start();
```

What is the output?

A. ABC
B. AA
C. AAB
D. AABC
E. AABB
F. ABAB

Quiz 1

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
         print “B”
         buffer.wait();
         print “C”
    }
    buffer.full = false;
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer("Consumer 1",buffer);
Thread c2 = new Consumer("Consumer 2",buffer);
c1.start();
C1.sleep(100);
c2.start();
```

What is the output?

A. ABC
B. AA
C. AAB
D. AABC
E. AABB
F. ABAB

Quiz 2

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
        print “B”
        buffer.wait();
        print “C”
    }
    buffer.full = false;
    print “D”
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer(buffer);
Thread p1 = new Producer(buffer);
c1.start();
sleep(1000);
p1.start();
```

What is the output?

A. AB
B. ABC
C. ABCBC
D. ABCD
E. ABD

Quiz 2

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
         print “B”
         buffer.wait();
         print “C”
    }
    buffer.full = false;
    print “D”
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer(buffer);
Thread p1 = new Producer(buffer);
c1.start();
sleep(1000);
p1.start();
```

What is the output?

A. AB
B. ABC
C. ABCBC
D. ABCD
E. ABD

Quiz 3

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
          print “B”
          buffer.wait();
          print “C”
    }
    buffer.full = false;
    print “D”
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer(buffer);
Thread p1 = new Producer(buffer);
p1.start();
sleep(1000);
c1.start();
```

What is the output?

A. ABCD
B. AD
C. ABCCD
D. A

Quiz 3

Consumer:

```
void take(){
  print ”A”
  synchronized(buffer){
    while(buffer.empty) {
         print “B”
         buffer.wait();
         print “C”
    }
    buffer.full = false;
    print “D”
    buffer.notify();
  }
}
```

Main:

```
Buffer buffer = new Buffer();
Thread c1 = new Consumer(buffer);
Thread p1 = new Producer(buffer);
p1.start();
sleep(1000);
c1.start();
```

Producer runs first.
Buffer is not empty

What is the output?

A. ABCD
B. AD
C. ABCCD
D. A

When To Use `notify()`

- Only use `notify()` if
  - Every thread in wait-set is guaranteed to be waiting on same condition
  - Condition is guaranteed to be true when thread executing `notify()` surrenders its lock on object
- Otherwise: use `notifyAll()`

Timed Waiting

- Problem with `wait():` unbounded waiting
  - You do not know how long a thread might wait before being able to continue
  - In some applications this leads to unacceptable performance variability
- Variant: `wait(long millis)`
  - Wait for at least specified # of milliseconds
  - At time-out, exit wait-set
  - How do you tell if exit from wait-set is due to notification or timeout?
    - You don't
    - You have to check this yourself
- Intermediate between balking, guarded suspension