Monitors

CS511

Review

- We've seen that semaphores are an efficient tool to solve synchronization problems
- However, they have some drawbacks
  1. They are low-level constructs
     - It is easy to forget an acquire or release
  2. They are not related to the data
     - They can appear in any part of the code

Monitors

- Combines ADTs and mutual exclusion
  - Proposed by Tony Hoare:

Monitors: An Operating System Structuring Concept (Communications of the ACM, 17(10), 549-557, 1974).

- Adopted in many modern PLs
  - Java
  - C#
  - Python
  - Ruby

Main Ingredients

- A set of operations encapsulated in modules
- A unique lock that ensures mutual exclusion to all operations in the monitor
- Special variables called condition variables, that are used to program conditional synchronization

Counter Example

- Construct a counter with two operations:
  - inc()
  - dec()
- No two threads should be able to simultaneously modify the value of the counter
  - Think of a solution using semaphores
  - A solution using monitors

Counter using Semaphores

```
1  class Counter {
2    private int c = 0;
3    private Semaphore mutex = new Semaphore(1);
4
5    public void inc() {
6        mutex.acquire();
7        c++;
8        mutex.release();
9    }
10   public void dec() {
11       mutex.acquire();
12       c--;
13       mutex.release();
14   }
15 }
```

Counter using Monitors

```
1  class Counter {
2
3    private int counter = 0;
4
5    public synchronized void inc() {
6      counter++;
7    }
8
9    public synchronized void dec() {
10     counter--;
11   }
12
13 }
```

- Each object has its own lock called intrinsic or monitor lock
- It also has its own wait-set for this lock (more on this later)
- This code is both Groovy and Java code

Counter as Monitor in Groovy – Continued

```
1  Counter c = new Counter()
2
3  P = Thread.start {
4      10.times {
5          c.inc()
6      }
7  }
8
9  Q = Thread.start {
10     10.times {
11         c.inc()
12     }
13 }
14
15 P.join()
16 Q.join()
17 println c.counter
```

Condition Variables

- Apart from the lock, there are condition variables associated to the monitor
  - Built-in: called a wait set and associated to the intrinsic lock
  - User-declared
- They have
  1. Three operations:
     - `Cond.wait()/await()`
     - `Cond.notify()/signal()`
     - `Cond.notifyAll()/signalAll()`
  2. A set of blocked processes.

Condition Variables

`Cond.wait()/await()`

- Always blocks the process and places it in the waiting set of the variable `Cond`.
- When it blocks, it releases the mutex on the monitor.

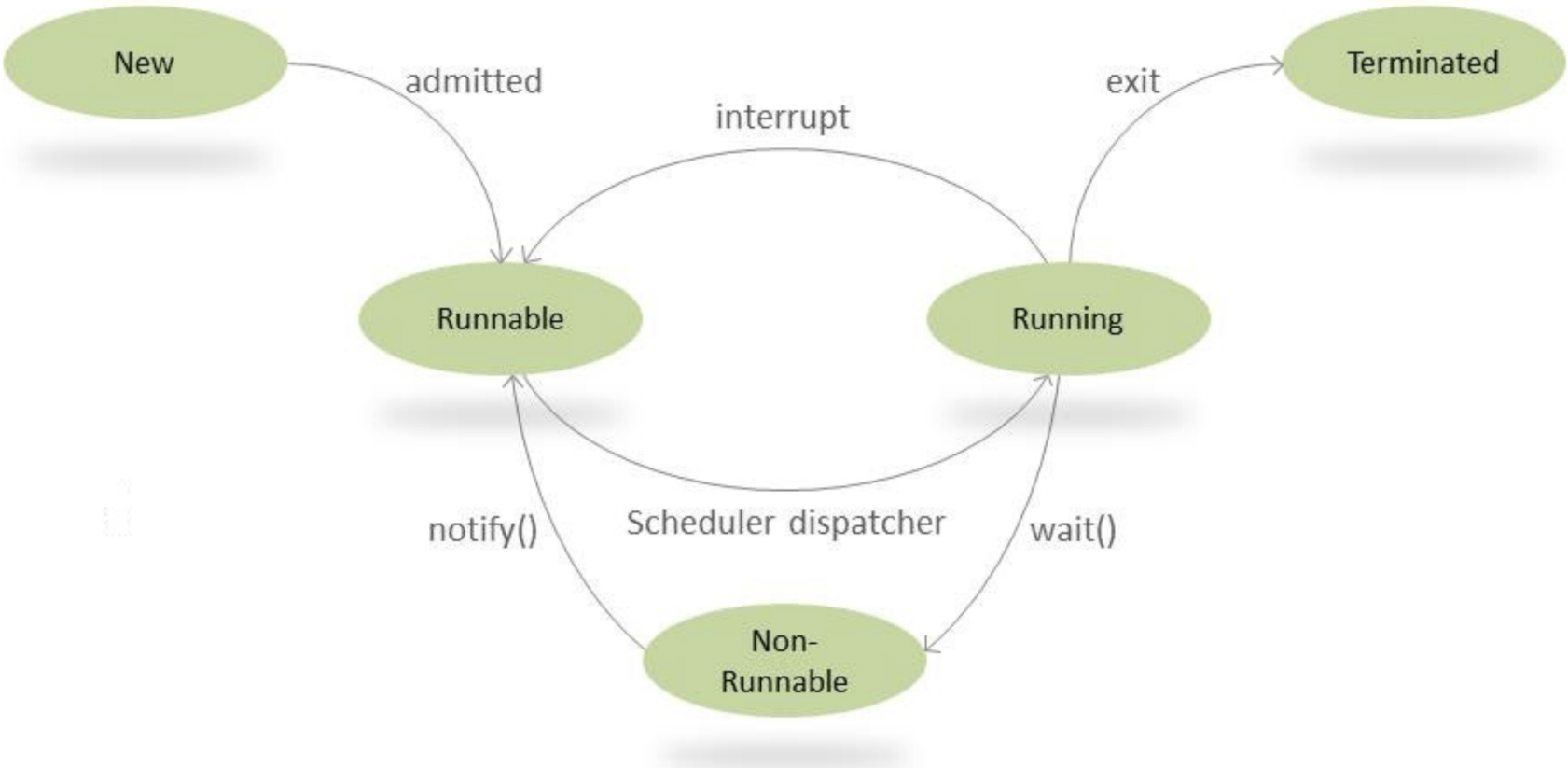
`Cond.notify()/signal()`

- Unblocks the first process in the waiting set of the variable `Cond` and sets it to the RUNNABLE state
- If there are no processes in the waiting set, it has no effect.

`Cond.notifyAll()/signalAll()`

- Unblocks all the processes in the waiting set of the variable `Cond` and sets them to the RUNNABLE state
- If there are no processes in the waiting set, it has no effect.

Condition Variables and Process States[1]



[1] Source: https://www.baeldung.com/java-wait-notify

Example: Buffer of Size 1 in Groovy

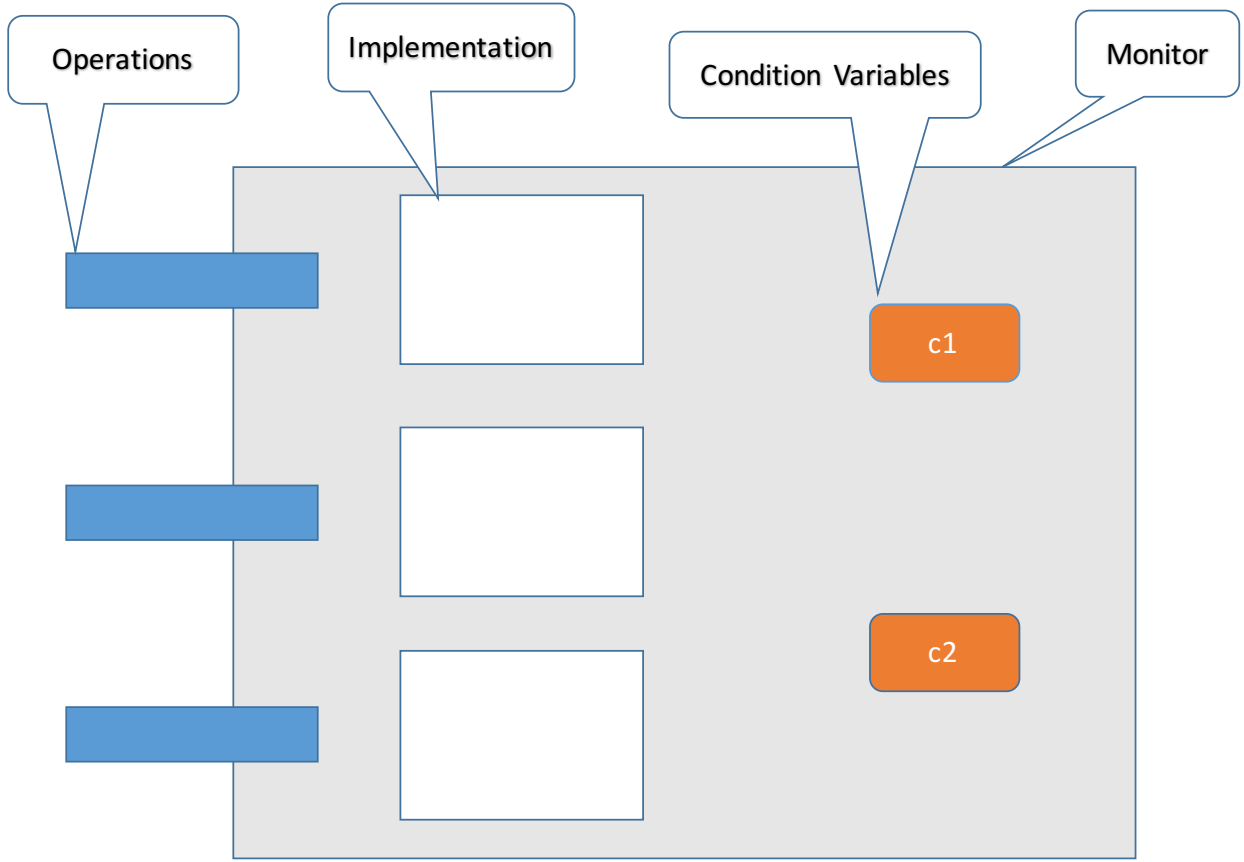
```
1  class Buffer {
2    Object buffer = null; // shared buffer
3
4    synchronized Object consume() {
5      while (buffer == null)
6        wait(); // wait on object's wait-set
7      Object aux = buffer;
8      buffer = null;
9      notifyAll(); // signal on object's wait-set
10     return aux;
11   }
12
13   synchronized void produce(Object o) {
14    while (buffer != null)
15       wait(); // wait on object's wait-set
16     buffer = o;
17     notifyAll(); // signal on object's wait-set
18   }
19 }
```

- `wait`, `notify` and `notifyAll` must be called from synchronized methods or else an `IllegalMonitorStateException` is raised

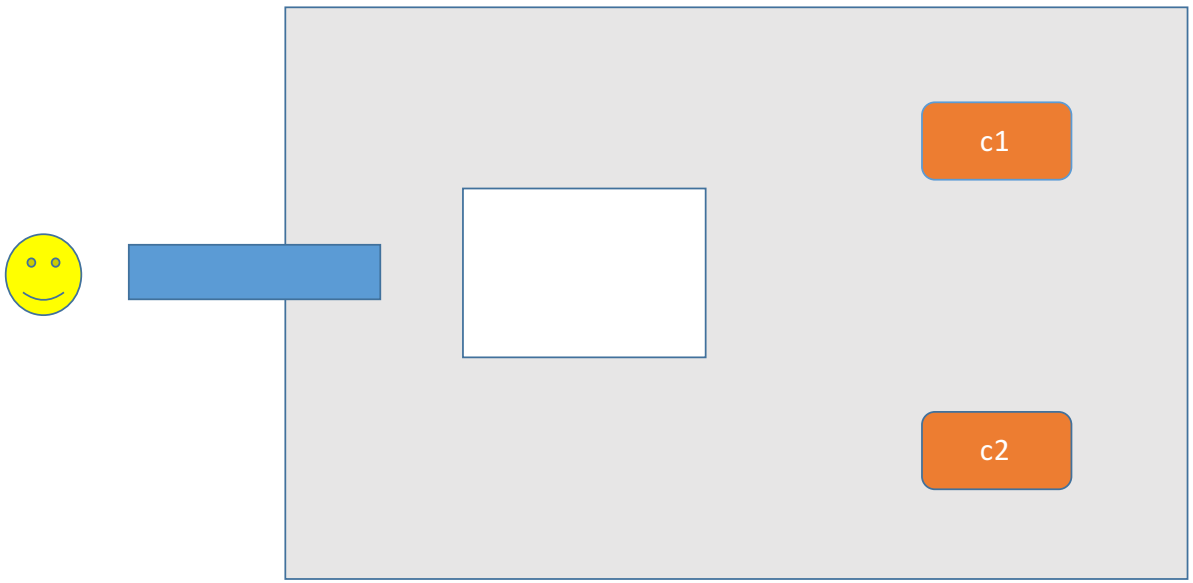
Example: Buffer of Size 1 in Groovy (cont)

```
1  Buffer b =new Buffer()
2
3  20.times {
4     int id=it
5     Thread.start {
6       println (id+": consumer "+ b.consume())
7     }
8  }
9
10 20.times {
11     int id=it
12     Thread.start {
13       b.produce(id)
14       println (id+": producer ")
15   }
16 }
```

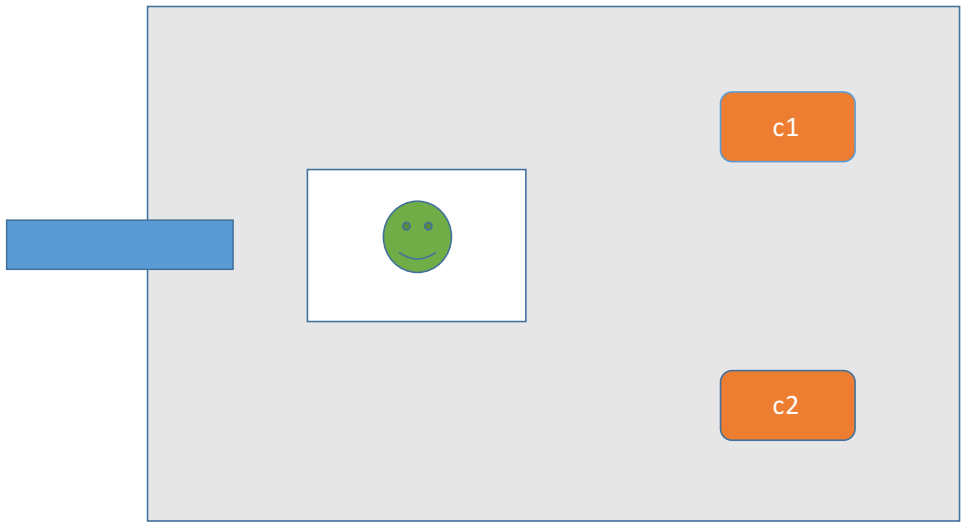
Explaining Monitors Graphically



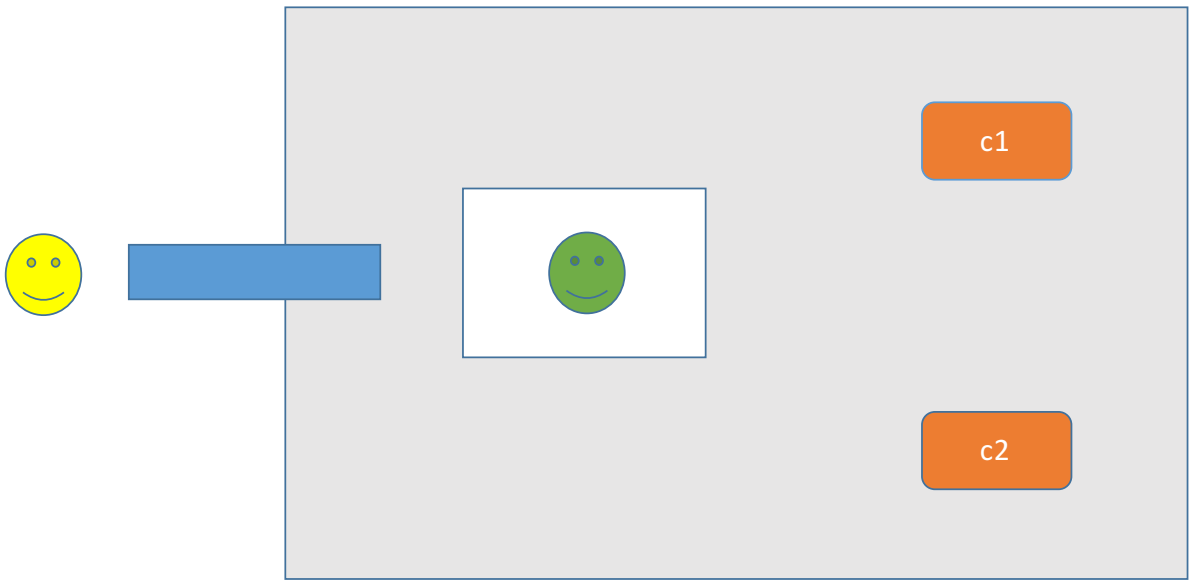
Typical Behavior



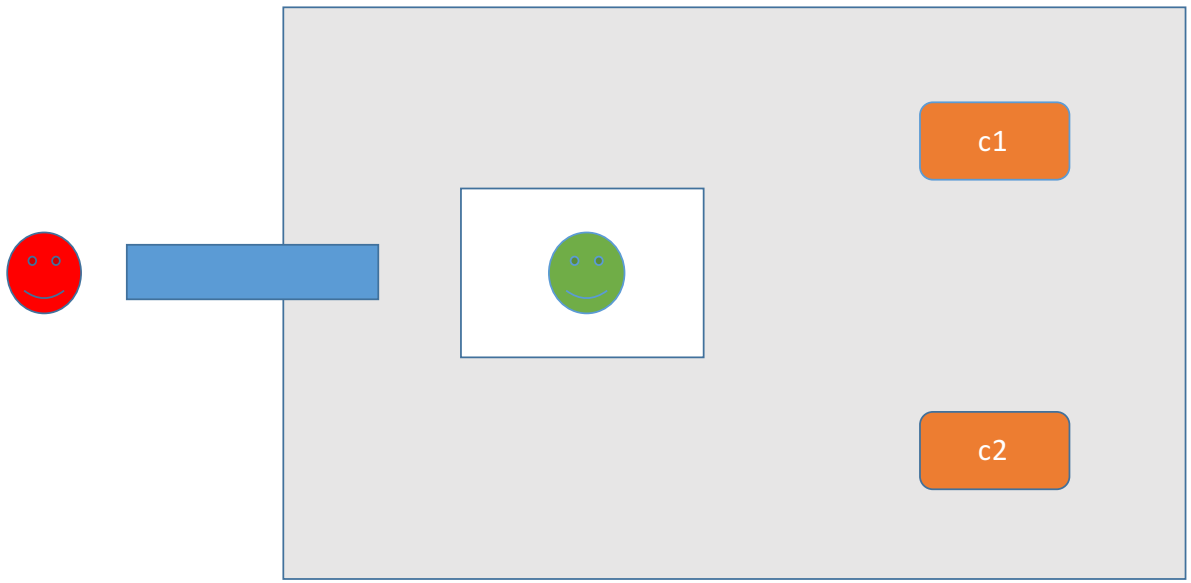
Typical Behavior



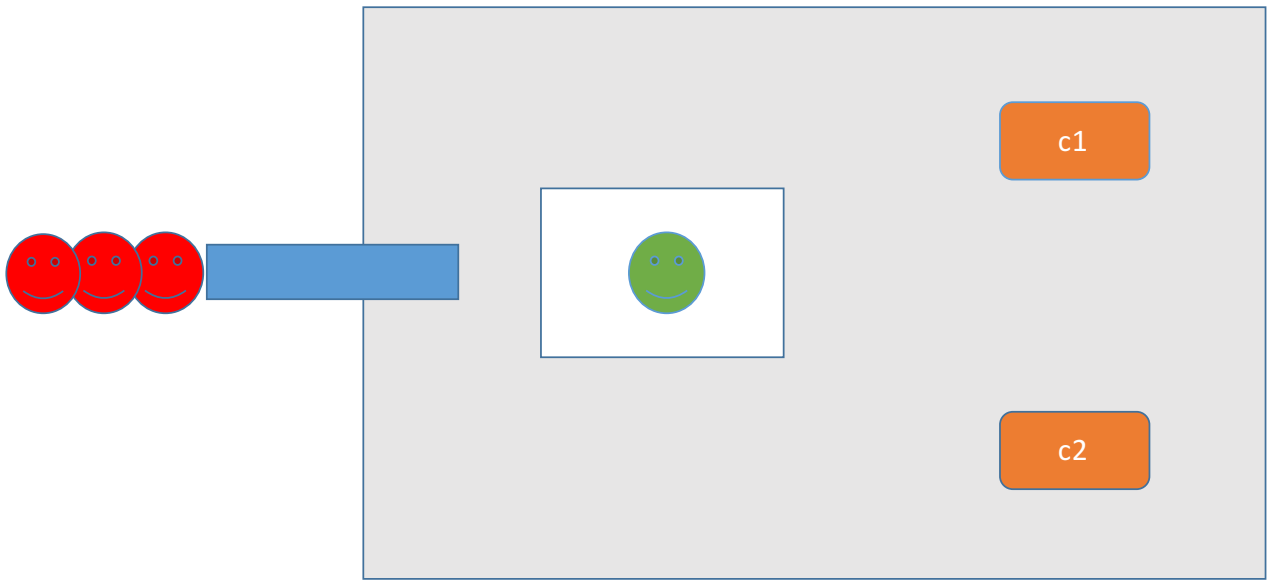
Typical Behavior



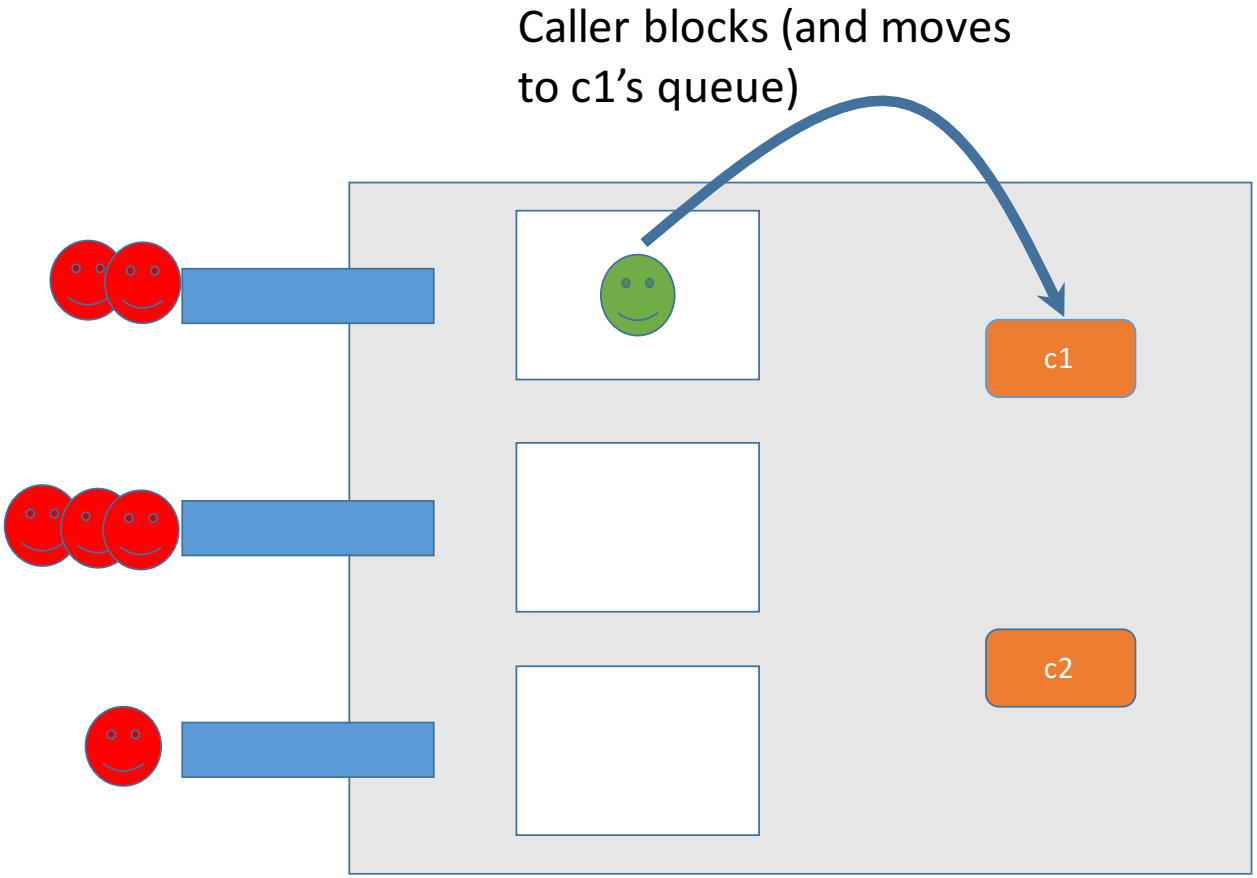
Typical Behavior



Typical Behavior

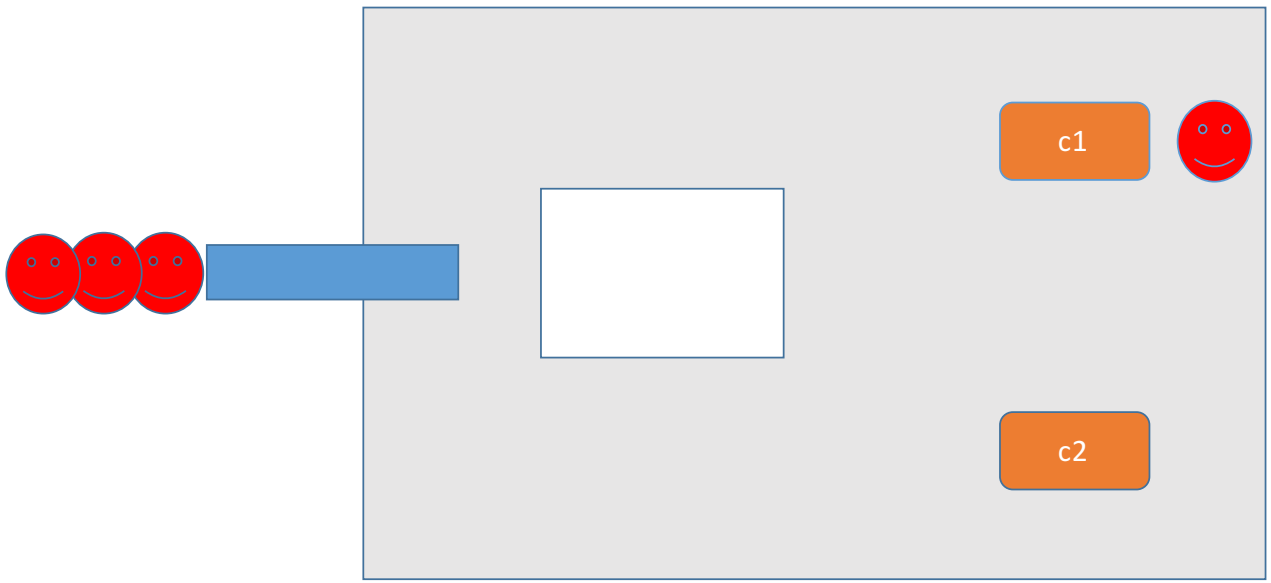


Wait



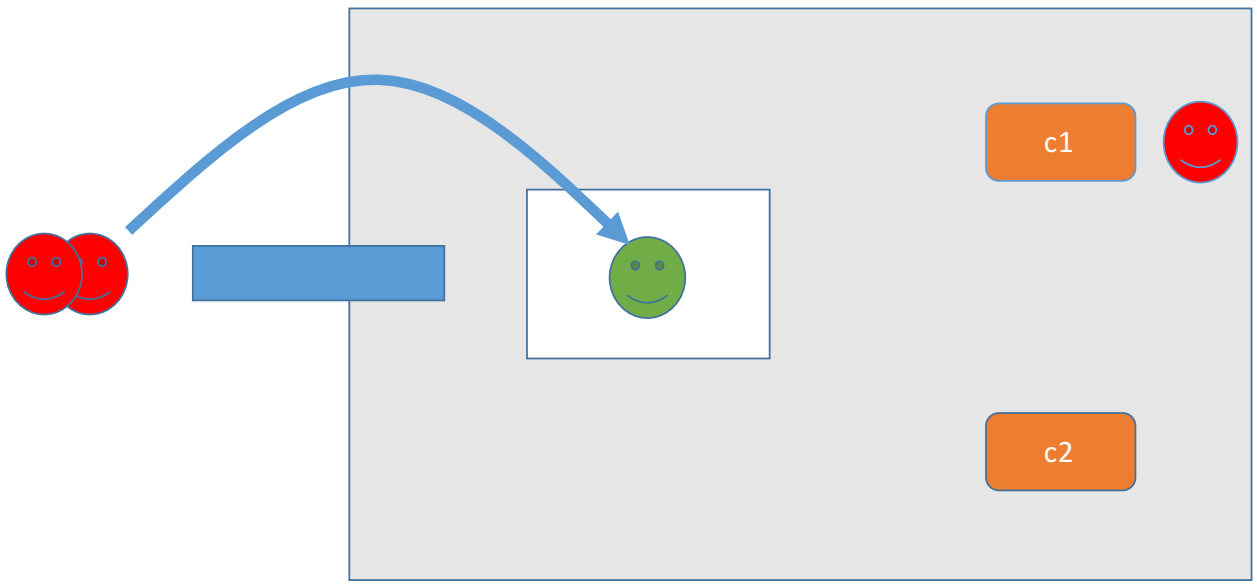
- Blocks process currently executing and associates it to variable's waiting set
- Upon blocking frees the lock allowing the entry of other processes

Wait



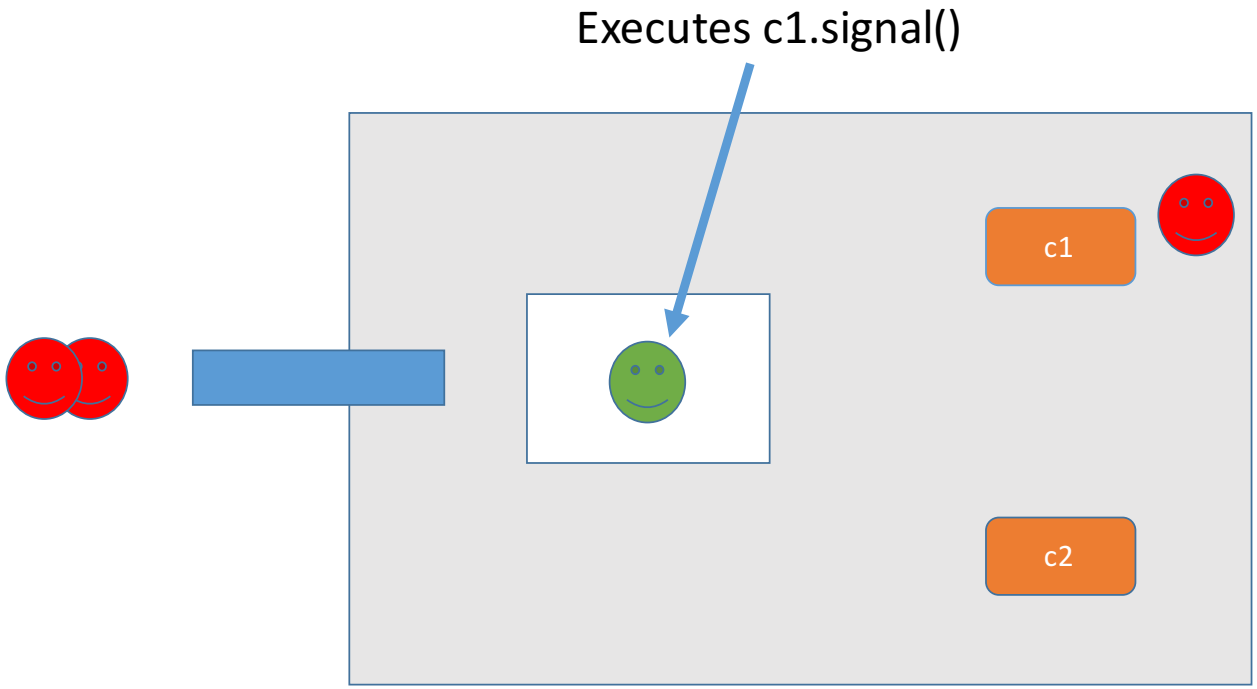
- Blocks process currently executing and associates it to variable's waiting set
- Upon blocking frees the lock allowing the entry of other processes

Wait



- Blocks process currently executing and associates it to variable's waiting set
- Upon blocking frees the lock allowing the entry of other processes

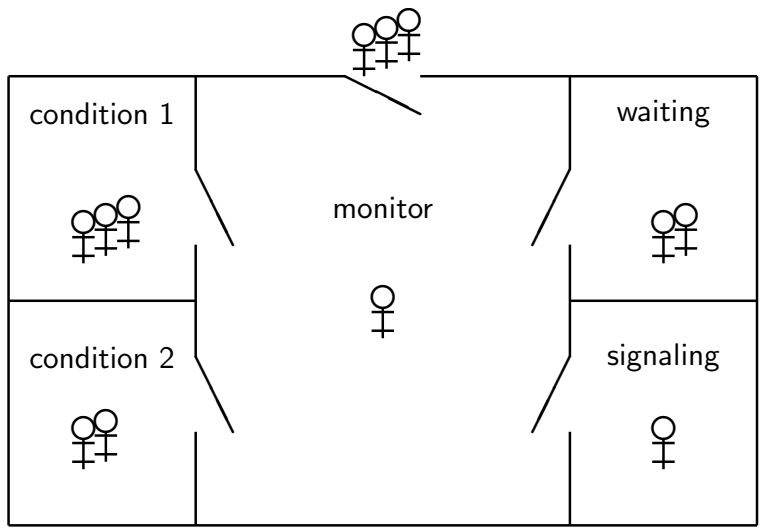
Signal



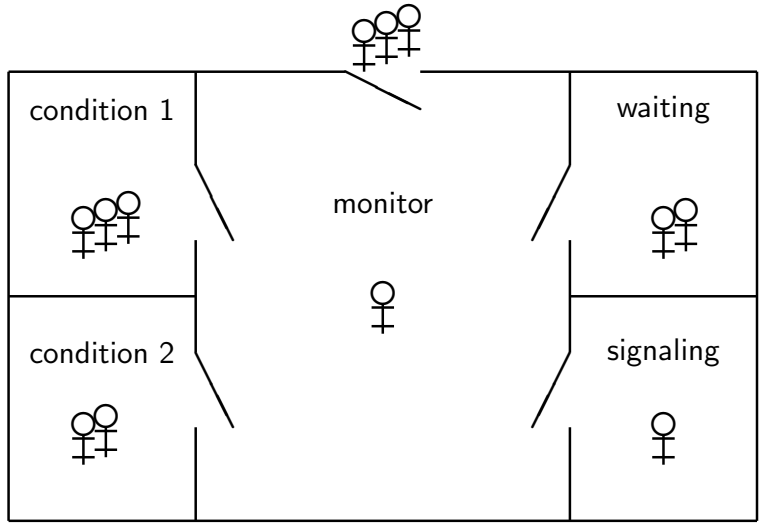
- Signalling process continues to execute after notifying on `c1`?
- Processes waiting in `c1`'s waiting-set start immediately running inside the monitor?
- What about the processes blocked on entry to the monitor?

Signal – States That a Process Can Be In

- Waiting to enter the monitor
- Executing within the monitor (only one)
- Blocked on condition variables
- Set of processes just released from waiting on a condition variable
- Set of processes that have just completed a `signal` operation



Notify



Two strategies:

- Signal and Urgent Wait: $E < S < W$ (classical monitors)
- Signal and Continue: $E = W < S$ (Java $\Leftarrow$ We adopt this)

where the letters denote the precedence of

- S: signalling processes
- W: waiting processes
- E: processes blocked on entry

Monitors

More Examples

Condition Variables

Visibility

Buffer of Size 1 in Groovy - Discussion

```
1  class Buffer {
2    Object buffer = null; // shared buffer
3
4    synchronized Object consume() {
5      while (buffer == null)
6        wait(); // wait on object's wait-set
7      Object aux = buffer;
8      buffer = null;
9      notifyAll(); // signal on object's wait-set
10     return aux;
11   }
12
13   synchronized void produce(Object o) {
14    while (buffer != null)
15       wait(); // wait on object's wait-set
16     buffer = o;
17     notifyAll(); // signal on object's wait-set
18   }
19 }
```

Buffer of Size 1 in Groovy - Discussion

```
1  class Buffer {
2    Object buffer = null; // shared buffer
3
4    synchronized Object consume() {
5      while (buffer == null)
6        wait(); // wait on object's wait-set
7      Object aux = buffer;
8      buffer = null;
9      notify(); // signal on object's wait-set
10     return aux;
11   }
12
13   synchronized void produce(Object o) {
14    while (buffer != null)
15       wait(); // wait on object's wait-set
16     buffer = o;
17     notify(); // signal on object's wait-set
18   }
19 }
```

- What goes wrong and why?

Monitor that Defines a Semaphore

```
1  class Semaphore {
2
3    private int permissions;
4
5    Semaphore(int n) {
6      this.permissions = n;
7    }
8
9    synchronized void acquire() {
10     while (permissions == 0)
11       wait();
12     permissions--;
13   }
14
15   synchronized void release() {
16     permissions++;
17     notifyAll();
18   }
19
20 }
```

Monitor that Defines a Semaphore

```
1  c=0;
2  Semaphore mutex = new Semaphore(1);
3
4  P = Thread.start {
5              20.times{
6                  mutex.acquire();
7                  c++;
8                  mutex.release();
9              }
10         }
11
12 Q = Thread.start {
13             20.times {
14                 mutex.acquire();
15                 c++;
16                 mutex.release();
17       }
18     }
19
20 P.join();
21 Q.join();
22 println c;
```

Signal and Continue

- Must re-check the condition since may have gained entry long after it was notified

```
synchronized void acquire() {
    while (permissions == 0)
        nonZero.wait();
    permissions--;
}
```

Another reason for re-checking the condition:

- Spurious wakeups: "Implementations are permitted, although not encouraged, to perform "spurious wake-ups", that is, to remove threads from wait sets and thus enable resumption without explicit instructions to do so."[2]

[2] JLS for Java SE 17 (page 740).
https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf

Signal and Continue

```
1  class Semaphore {
2    private int permissions;
3
4    Semaphore(int n) {
5      this.permissions = n;
6    }
7
8    synchronized void acquire() {
9      while (permissions == 0)
10       wait();
11     permissions--;
12   }
13
14   synchronized void release() {
15     permissions++;
16     notifyAll();
17   }
18
19 }
```

- Is it fair?
- See *Specific Notification for Java Thread Synchronization* by Tom Cargill, 1996.[3]

[3] www.dre.vanderbilt.edu/%7Eschmidt/PDF/specific-notification.pdf

Monitors

More Examples

Condition Variables

Visibility

Buffer of Size 1 in Groovy - Discussion

```
1  class Buffer {
2    Object buffer = null; // shared buffer
3
4    synchronized Object consume() {
5      while (buffer == null)
6        wait(); // wait on object's wait-set
7      Object aux = buffer;
8      buffer = null;
9      notifyAll(); // signal on object's wait-set
10     return aux;
11   }
12
13   synchronized void produce(Object o) {
14    while (buffer != null)
15       wait(); // wait on object's wait-set
16     buffer = o;
17     notifyAll(); // signal on object's wait-set
18   }
19 }
```

- Inefficient
- Much more efficient (and clearer) to have multiple wait-sets

Explicit Locks

- Alternative to using intrinsic lock of an object
- Convenient for modeling condition variables
- Example:
  - Declare lock and associate condition variables to it

```
final Lock lock = new ReentrantLock();
final Condition empty = lock.newCondition();
final Condition full = lock.newCondition();
```

  - Replace `synchronized` with

```
Lock l = new ReentrantLock();
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Condition Variables

```
1  import java.util.concurrent.locks.*;
2
3  class Buffer {
4      Object buffer = null; // shared buffer
5      final Lock lock = new ReentrantLock();
6      final Condition empty = lock.newCondition();
7      final Condition full = lock.newCondition();
8
9      Object consume() {
10         lock.lock();
11         try {
12             while (buffer == null)
13                 full.await();
14             Object aux = buffer;
15             buffer = null;
16             empty.signal();
17             return aux;
18         } finally {
19             lock.unlock();
20         }
21     }
22
23     // continues in next slide
24 }
```

Condition Variables

```
1       void produce(Object o) {
2           lock.lock();
3           try {
4               while (buffer != null)
5                   empty.await();
6               buffer = o;
7               full.signal();
8           } finally {
9               lock.unlock();
10          }
11      }
12  }
```

Condition Variables

```
1  Buffer b =new Buffer()
2
3  20.times {
4      int id=it
5      Thread.start {
6          println "consumer "+ b.consume();
7      }
8  }
9
10 20.times {
11     int id=it
12     Thread.start {
13         b.produce(id);
14     }
15 }
```

Buffer of Size n

```
1  import java.util.concurrent.locks.*;
2  class PC {
3      private Integer[] data;
4      private int begin = 0;
5      private int end = 0;
6      private final int N;
7      final Lock lock = new ReentrantLock();
8      final Condition notEmpty = lock.newCondition();
9      final Condition notFull = lock.newCondition();
10
11     public PC(int size) {
12         this.N = size;
13         data = new Integer[N];
14     }
15
16     public void produce(Integer o) {
17         lock.lock();
18         try {
19             while (isFull()) {
20                 notFull.await();
21             }
22             data[begin] = o;
23             begin = (begin+1) % N;
24             notEmpty.signal()
25         } finally {
26             lock.unlock();
```

Buffer of Size *n*

```
1
2       public Integer consume() {
3           lock.lock();
4           try {
5               while (isEmpty()) {
6                   notEmpty.await();
7               }
8               Integer result = data[end];
9               end = (end+1) % N
10              notFull.signal();
11              return result;
12          } finally {
13              lock.unlock();
14          }
15      }
16
17      private boolean isEmpty() { return begin == end; }
18      private boolean isFull()  { return ((begin+1)%N) == end; }
19  }
```

Readers/Writers

```
1  class ReadersWriters {
2    ...
3
4    public synchronized void read() {
5      ...
6    }
7
8    public synchronized void write() {
9      ...
10   }
11
12 }
```

What is the problem with this setting?

Readers/Writers

```
1  import java.util.concurrent.locks.*;
2
3  class ReadersWriters {
4      Integer readers = 0;
5      Integer writers = 0;
6      final Lock lock = new ReentrantLock();
7      final Condition okToRead = lock.newCondition();
8      final Condition okToWrite = lock.newCondition();
9
10     public void startRead() {
11         lock.lock();
12         try {
13             while (writers != 0) {
14                 okToRead.await();
15             };
16             readers = readers + 1;
17         } finally {
18             lock.unlock();
19         }
20     }
21   // continues
```

Readers/Writers

```
1       public  void endRead() {
2           lock.lock();
3           try {
4               readers = readers - 1;
5               if (readers==0) {
6                   okToWrite.signal();
7               }
8           } finally {
9               lock.unlock();
10          }
11      }
12    // continues
```

Readers/Writers

```
1       public void startWrite() {
2           lock.lock();
3           try {
4               while (writers != 0 || readers != 0) {
5                   okToWrite.await();
6               }
7               writers = writers + 1;
8           } finally {
9               lock.unlock();
10          }
11        }
12
13      public  void endWrite() {
14          lock.lock();
15          try {
16              writers = writers - 1;
17              okToWrite.signal();
18              okToRead.signalAll();
19          } finally {
20              lock.unlock();
21          }
22      }
23  }
```

Readers/Writers

```
1  ReadersWriters r = new ReadersWriters()
2
3  10.times {
4      int id = it;
5      Thread.start {
6          r.startRead();
7          println "$id: start reading...";
8          r.endRead();
9          println "$id: done reading...";
10     }
11 }
12
13 10.times {
14     int id = it;
15     Thread.start {
16         r.startWrite();
17         println "$id: start writing..."
18         r.endWrite();
19         println "$id: done writing..."
20     }
21   }
```

Assessment

- Upholds the readers-writers invariant but gives priority to readers over writers:
  - new readers can enter the monitor without waiting as long as a reader is active
  - waiting writers have to wait until the last reader calls endRead and signals OKtoWrite
  - as long as readers keep arriving and waiting to enter the monitor, the waiting writers will never execute

RW - Fair on Writers

```
1  class ReadersWriters {
2        ...
3        Integer waitingWriters = 0;
4        ...
5      public void startRead() {
6          lock.lock();
7          try {
8              while (writers != 0 || waitingWriters>0) {
9                  oKtoRead.await();
10             };
11             readers = readers + 1;
12         } finally {
13             lock.unlock();
14         }
15     }
16
17     public void startWrite() {
18     lock.lock();
19     try {
20         while (writers != 0 || readers != 0) {
21             waitingWriters++;
22             oKtoWrite.await();
23             waitingWriters--;
24         }
25         writers = writers + 1;
26     } finally {
```

Dining Philosophers

```
1  class ForkMonitor {
2      final int N
3      List<Integer> forks = []  // forks available to each phil
4      final Lock lock = new ReentrantLock()
5      final List<Condition> okToEat = []
6
7      ForkMonitor(int n) {
8          this.N = n;
9          N.times { forks.add(2) }
10         N.times { okToEat.add(lock.newCondition()) }
11     }
```

`forks[i]` is number of forks available to philosopher *i*

Dining Philosophers

```
1     private void updateForkCount(Integer i,Integer delta) {
2         lock.lock();
3         try {
4             forks[(i+1) % N] = forks[(i+1) % N] + delta;
5             if (i-1>0) {
6                 forks[i-1] = forks[i-1] + delta
7             } else { // i-1==0
8                 forks[N-1] = forks[N-1] + delta
9             }
10        } finally {
11            lock.unlock();
12        }
13    }
14
15    public void takeForks(int i) {
16        lock.lock();
17        try {
18            while (forks[i] != 2) {
19                okToEat[i].await();
20            }
21            updateForkCount(i,-1);
22        } finally {
23            lock.unlock();
24        }
25    }
```

Dining Philosophers

```
1       public void releaseForks(int i) {
2           lock.lock();
3           try {
4               updateForkCount(i,1);
5               if (forks[i+1] == 2) {
6                   okToEat[i+1].signal();
7               }
8               if (forks[i-1] == 2) {
9                   okToEat[i-1].signal();
10              }
11          } finally {
12              lock.unlock();
13          }
14      }
15  }
```

Dining Philosophers

```
1  ForkMonitor dp = new ForkMonitor(5)
2
3  5.times {
4      int id = it
5      Thread.start {
6          while (true) {
7              dp.takeForks(id)
8              println "$id grabbed forks"
9              dp.releaseForks(id)
10             println "$id released forks"
11         }
12     }
13 }
```

Monitors

More Examples

Condition Variables

Visibility

Visibility

- Whether a thread can see the modifications of other threads
- Visibility is subtle because the compiler may
  - Reorder operations
  - Cache values in registers
- `synchronization`, used for atomicity, helps with visibility too:
  - All changes made in one synchronized method or block are visible with respect to other synchronized methods and blocks employing the same lock.

Volatile Variables

```
1 int n=0;

2 Thread.start { // P
3   int local1, local2;
4   n = some expression;
5   computation not using n;
6   local1 = (n+5)*7;
7   local2 = n+5;
8   n = local1 *  local2;
9 }
```

```
2 Thread.start { // Q
3   int local;
4   local = n+6;
5
6
7
8
9 }
```

The instruction in Q may be interleaved at any place during the execution of the instructions in P

Volatile Variables

An optimizing compiler could translate statements in thread P as:

```
1 tempReg1 = some expression
2 computation not using n
3 tempReg2 = tempReg1 + 5
4 local2 = tempReg2
5 local1 = tempReg2 * 7
6 n = local1 * local2
```

```
1 n = some expression;
2 computation not using n;
3 local1 = (n+5)*7;
4 local2 = n+5;
5 n = local1 *  local2;
6 .
```

- No assignment to n in the first statement. Original statements p3 and p4 are executed out of order
- Without concurrency, the translated code would be correct
- With concurrency and interleaving, the translated code may no longer be correct
- Specifying a variable as volatile instructs the compiler to load and store the value of the variable at each use, rather than to optimize away these loads and stores.

Example 1[4]

```
1  class SharedVariable {
2     private static int  sharedVariable = 0;
3     public static void main(String[] args) throws InterruptedExcept
4
5     new Thread(new Runnable() {
6        @Override
7        public void run() {
8           try { Thread.sleep(100); }
9           catch (InterruptedException e) {
10                  e.printStackTrace();
11          }
12          sharedVariable = 1;
13       }}).start();
14
15    for(int i=0;i<1000;i++) {
16       for(;;) {
17          if(sharedVariable == 1) { break; }
18       }
19    }
20    System.out.println("SharedVariable : " + sharedVariable);
21    }
22 }
```

Try this code as is (loops due to compiler optimization), then add the qualified volatile to the declaration of `sharedVariable` and run

Example 2

```
1  class NoVisibility {
2      private static boolean ready;
3      private static int number;
4
5      private static class ReaderThread extends Thread {
6          public void run() {
7              while (!ready)
8                  Thread.yield();
9              System.out.println(number);
10         }
11     }
12
13     public static void main(String[] args) {
14         new ReaderThread().start();
15         number = 42;
16         ready = true;
17     }
18 }
```

- `java.lang.Thread.yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute
- What is the output?

Example 2

```
1  class NoVisibility {
2      private static boolean ready;
3      private static int number;
4
5      private static class ReaderThread extends Thread {
6          public void run() {
7              while (!ready)
8                  Thread.yield();
9              System.out.println(number);
10         }
11     }
12
13     public static void main(String[] args) {
14         new ReaderThread().start();
15         number = 42;
16         ready = true;
17     }
18 }
```

- `java.lang.Thread.yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute
- What is the output? Could loop forever or print 0!