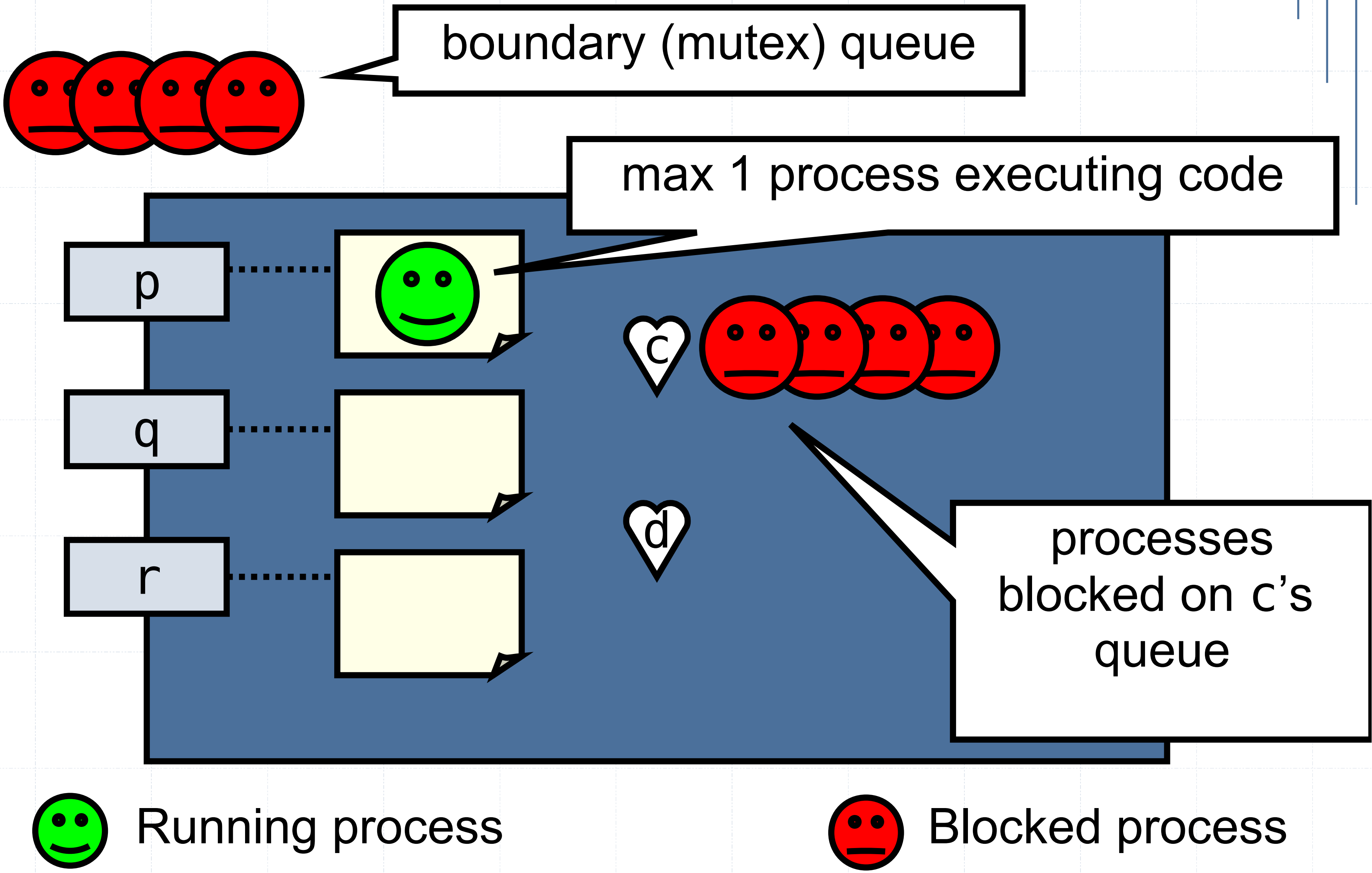
Lecture 5

Monitors

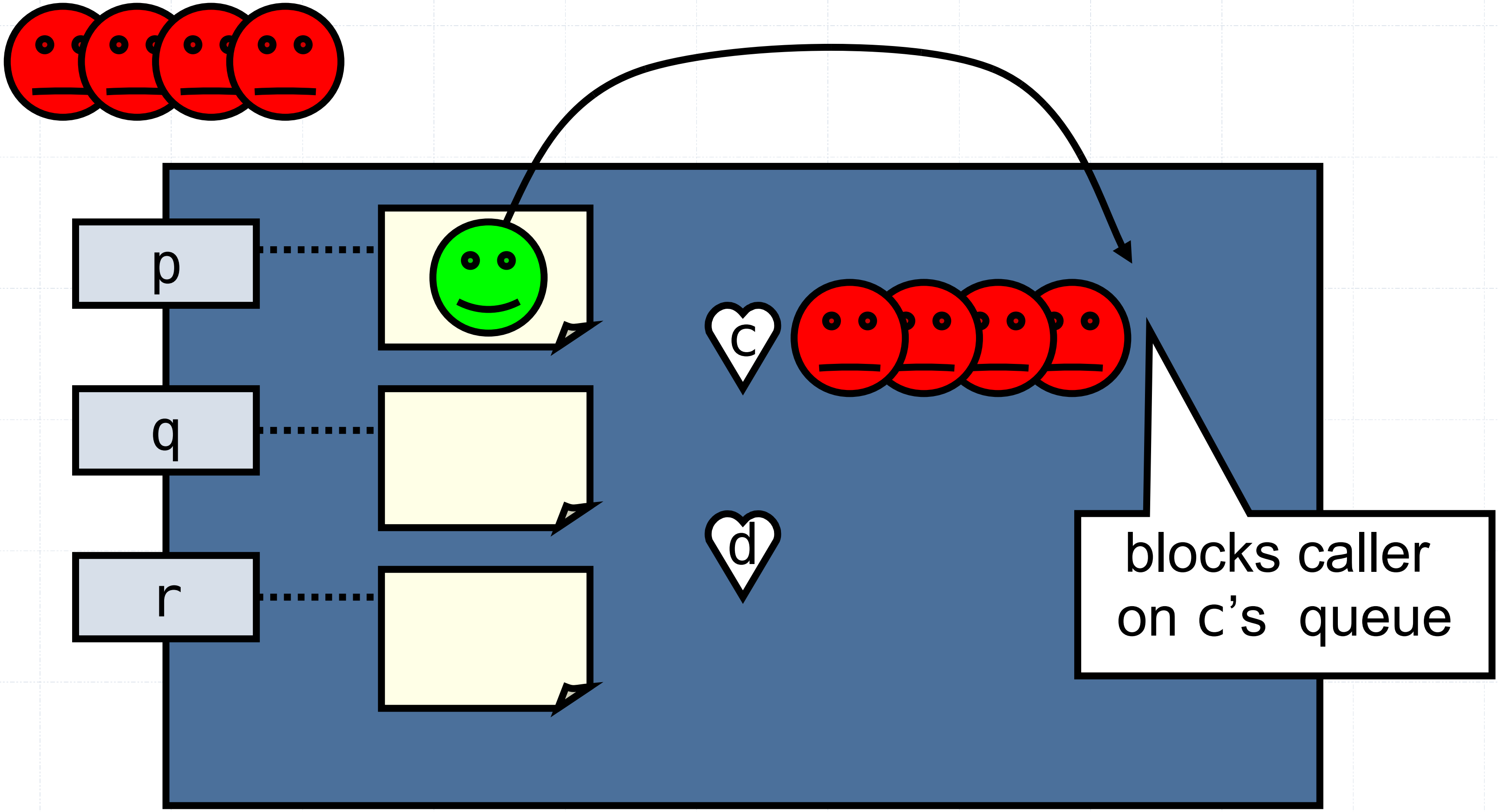
Monitors

- Summary: Last time
  - A combination of data abstraction and mutual exclusion
    - Automatic mutex
    - Programmed conditional synchronisation
  - Widely used in concurrent programming languages and libraries
    - Java, pthreads, C#, …
- Today
  - More monitor synchronisation
  - Problem solving using monitors in Java

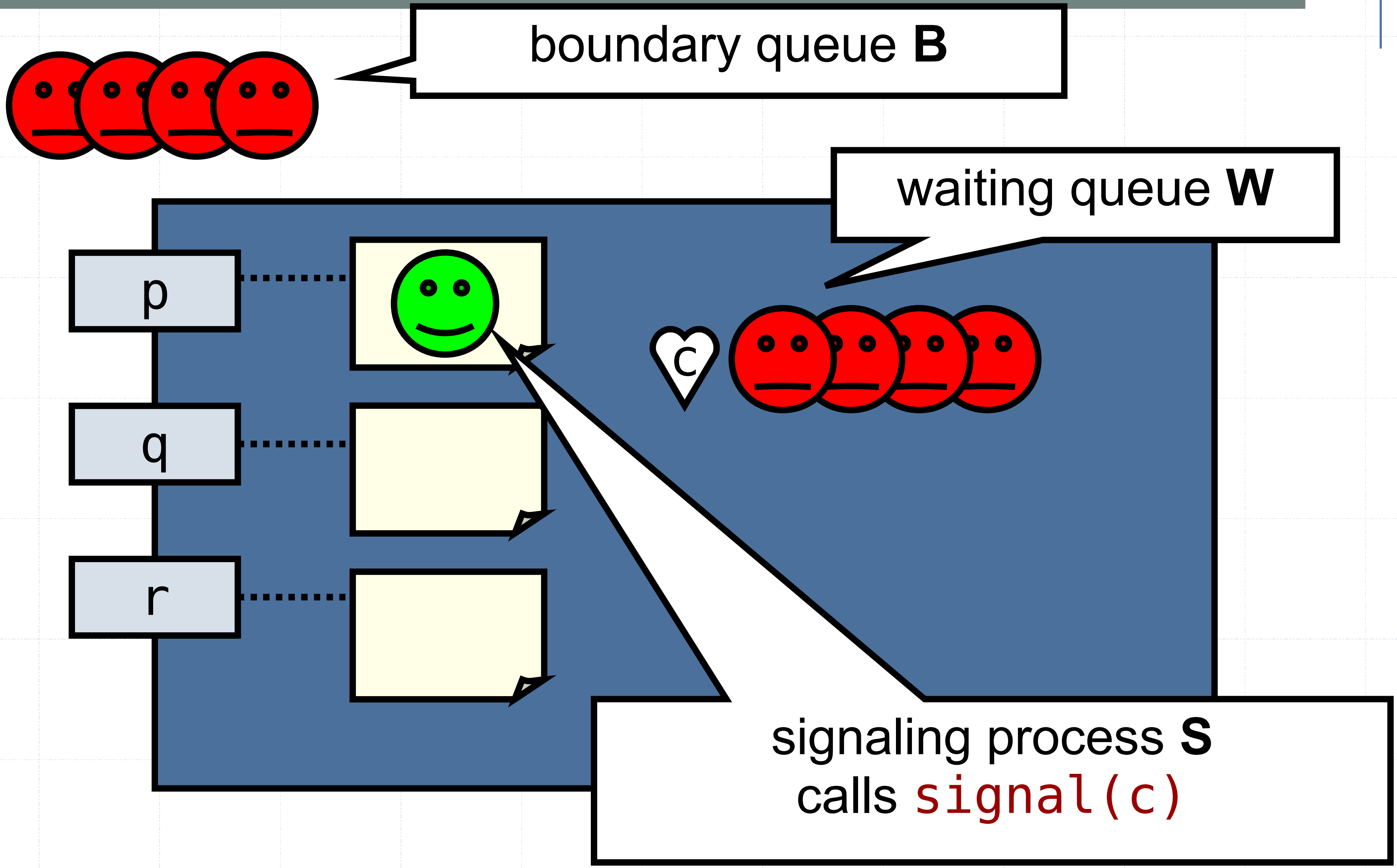
A Typical Monitor State



wait(c)

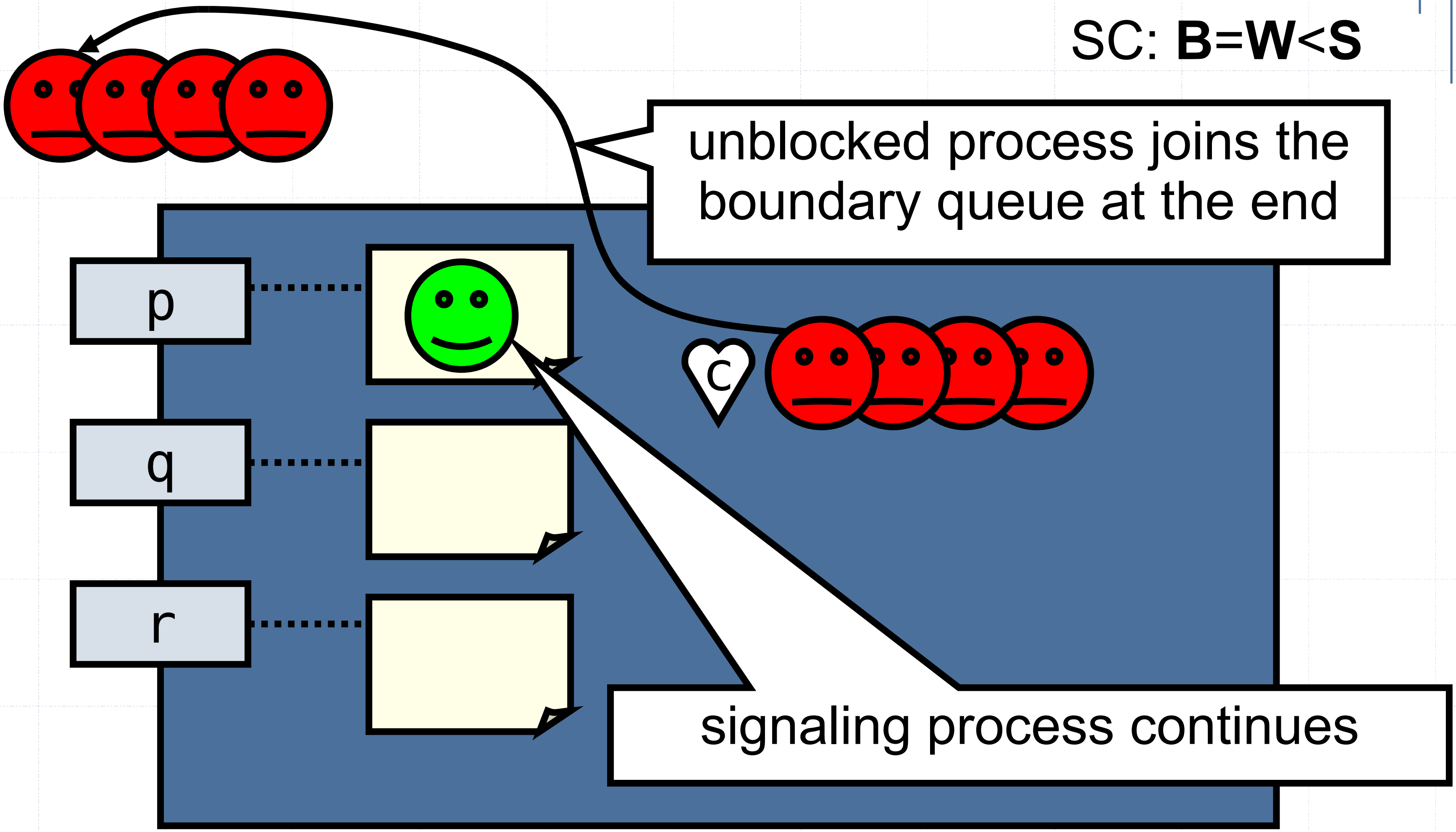


Signal and What Happens Next?



Signal and Continue

SC: **B**=**W**<**S**



Java and Monitors

- The essence of a monitor is the combination of
  - data abstraction
    - class
  - mutual exclusion
    - synchronized
  - condition variables
    - default: implicit, one per object
  - operations for blocking and unblocking on condition variables
    - included in Object

Barrier Monitor

- Simple but not 100% reliable solution
  - Spurious wakeup possible

```
public synchronized void await()
 throws InterruptedException {
    arrived++;
    if (arrived < N)
       wait();
    else {
       notifyAll();
       arrived = 0;
    }
}
```

Barrier Monitor Zeroth Attempt

```
public synchronized void await()
 throws InterruptedException {
    arrived++;
    if (arrived < N) {
       do
          wait();
       while (arrived < N);
    } else {
       notifyAll();
       arrived = 0;
    }
}
```

Barrier Monitor

```
public class CyclicBarrier {

   private int arrived = 0;
   private int N;

   private Map<Integer,Integer> flag =
      new HashMap<Integer,Integer>();
   private int turn = 0;

   public CyclicBarrier(int N) {
      this.N = N;
      flag.put(turn, 0);
   }
//next slide
```

Barrier Monitor – First Attempt

```
public synchronized void await() throws IE {
   arrived++;
   if (arrived < N)
      while (flag.get(turn) == 0)
         wait();
   else {
      flag.put(turn, N-1);
      turn++;
      flag.put(turn, 0);
      notifyAll();
      arrived = 0;
}}
```

Barrier Monitor – First Attempt

```
public synchronized void await() throws IE {
   arrived++;
   if (arrived < N)
      while (flag.get(turn) == 0)
         wait();
   else {
      flag.put(turn, N-1);
      turn++;
      flag.put(turn, 0);
      notifyAll();
      arrived = 0;
}}
```

Is this really my turn?

Barrier Monitor – Second Attempt

```
public synchronized void await() throws IE {
    arrived++;
    if (arrived < N) {
        int myTurn = turn;
        while (flag.get(myTurn) == 0)
            wait();
    }
    else {
        flag.put(turn, N-1);
        turn++;
        flag.put(turn, 0);
        notifyAll();
        arrived = 0;
}}
```

Barrier Monitor – Second Attempt

```
public synchronized void await() throws IE {
    arrived++;
    if (arrived < N) {
        int myTurn = turn;
        while (flag.get(myTurn) == 0)
            wait();
    }
    else {
        flag.put(turn, N-1);
        turn++;
        flag.put(turn, 0);
        notifyAll();
        arrived = 0;
}}
```

Memory problem?

Barrier Monitor – Final Attempt

```
public synchronized void await() throws IE {
   arrived++;
   if (arrived < N) {
      int myTurn = turn;
      while (flag.get(myTurn) == 0)
         wait();
      if (flag.put(myTurn,
                   flag.get(myTurn)-1) == 1)
         flag.remove(myTurn);
   }
   else {
      ...
}}
```

Java 5 and Monitors

- The essence of a monitor is the combination of
  - data abstraction
    - class
  - mutual exclusion
    - explicit locking
    - package `java.util.concurrent.locks`
  - condition variables
    - unlimited
  - operations for blocking and unblocking on condition variables

Readers/Writers Problem

- Another classic synchronisation problem
- Two kinds of processes share access to a "database"
  - Readers examine the contents
  - Multiple readers allowed concurrently
  - Writers examine and modify
  - A writer must have mutex
- Invariant
  - $\Box((nr==0 \vee nw==0) \wedge nw<=1)$

Readers/Writers Monitor

- Database is globally accessible
  - Cannot be internal to monitor (critical section!)
- Encapsulate only the access protocol

```
public interface ReadersWriters {
   public void startRead()
      throws  InterruptedException;
   public void endRead();
   public void startWrite()
      throws InterruptedException;
   public void endWrite();
}
```

Readers/Writers Monitor

- Start with an easier non-fair solution

```
public class RWController implements RW {
   private final Lock lock =
      new ReentrantLock();
   private final Condition okToRead  =
      lock.newCondition();
   private final Condition okToWrite =
      lock.newCondition();

   private int nr = 0;
   private int nw = 0;
//next slides
```

Notation – Macro

```
public … method(…) throws … {
   lock.lock();
   Thread ct = Thread.currentThread();
   try {
      //Normal main code here
   }
   finally {
      lock.unlock();
}}
```

```
SYNC … method(…) throws … {
   //Normal main code here
}
```

Readers/Writers Reading

- Signal a writer after all readers left

```
SYNC void startRead() throws IE {
   while (nw > 0)
      okToRead.await();
   nr++;
}

SYNC void endRead() {
   nr--;
   if (nr == 0)
      okToWrite.signal();
}
```

Readers/Writers Reading

- Signal a writer after all readers left

```
SYNC void startRead() throws IE {
    while (nw > 0)
        okToRead.await();
    nr++;
}

SYNC void endRead() {
    nr--;
    if (nr == 0)
        okToWrite.signal();
}
```

nr==nw==0

Readers/Writers Writing

- On leave: signal a writer and all readers

```
SYNC void startWrite() throws IE {
   while (nr > 0 || nw > 0)
      okToWrite.await();
   nw++;
}

SYNC void endWrite() {
   nw--;
   okToWrite.signal();
   okToRead.signalAll();
}
```

Readers/Writers Writing

- On leave: signal a writer and all readers

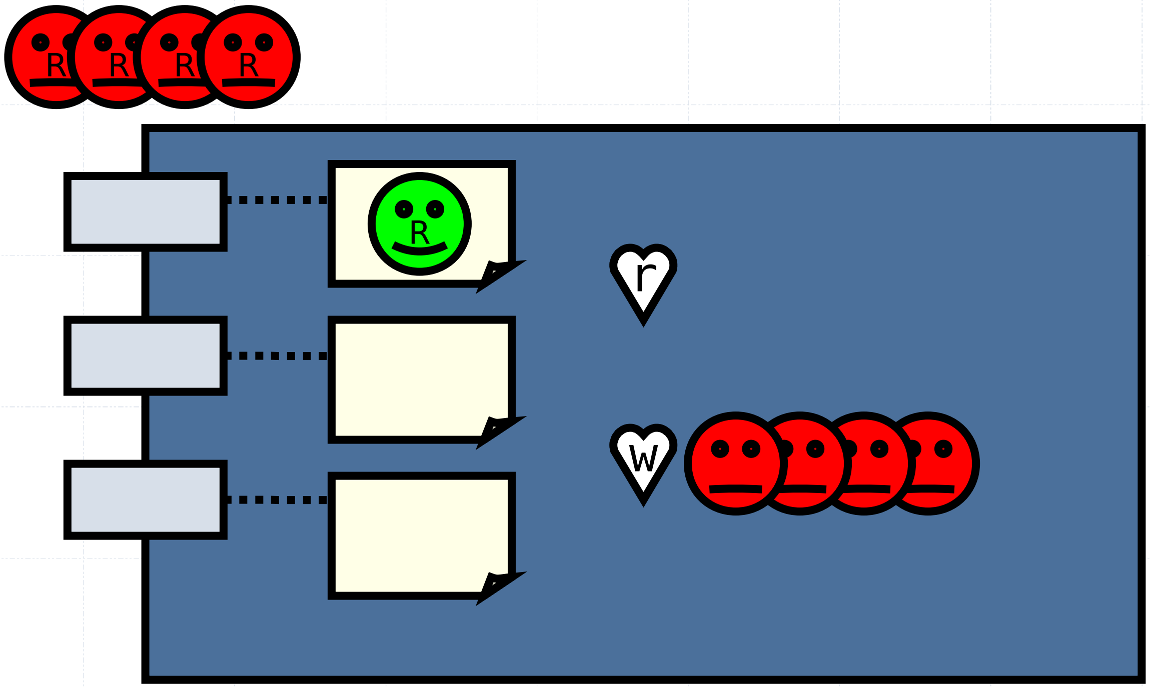
```
SYNC void startWrite() throws IE {
   while (nr > 0 || nw > 0)
      okToWrite.await();
   nw++;
}

SYNC void endWrite() {
   nw--;
   okToWrite.signal();
   okToRead.signalAll();
}
```

nr==nw==0

Analysis

- Starvation
  - Readers can continuously read
  - Waiting writers will not be woken



Fairness Considerations

- Suitable policy? For example:
  - No new readers when a writer is waiting
  - Change turns in some way
  - Strict order of arrival
- Performance
  - Fairness often requires more book-keeping
  - Depends highly on platform
  - Java
    - `signalAll()` might be inevitable for condition rechecking

Fair Readers/Writers

- No new readers when a writer is waiting
  - Avoids starvation
  - Spurious wakeup can spoil fairness
  - Quite efficient
  - Count delayed readers and writers
    - Similar to semaphore passing the baton solution

```
private int dr = 0;
private int dw = 0;
```

Readers/Writers Reading

- We only need to modify `startRead()`
  - One extra fairness condition
- `endRead()` is exactly the same

```
SYNC void startRead() throws IE {
    if (nw > 0 || dw >0) {
        dr++;
        do okToRead.await();
        while (nw > 0);
        dr--;
    }
    nr++;
}
```

Readers/Writers Writing

- One extra fairness condition

```
SYNC void startWrite() throws IE {
    if (nr > 0 || nw > 0 || dr > 0) {
        dw++;
        do okToWrite.await();
        while (nr > 0 || nw > 0);
        dw--;
    }
    nw++;
}
```

Readers/Writers Writing

- On leave:
  - Fairly signal all readers, or
  - One writer

```
SYNC void endWrite() {
    nw--;
    if (dr > 0)
        okToRead.signalAll();
    else
        okToWrite.signal();
}
```

Fair Readers/Writers

- Strict order of arrival
  - Fairest
  - Least efficient
    - at least in Java ⇒ spurious wakeup
  - Maintain queue of threads as they arrive
  - We also need to know their type

```
enum Type {Reader, Writer};
class Pair {…}
Queue<Pair> w = new ArrayDeque<Pair>();
```

Special Pairs

```
private class Pair {
   public Type type;
   public Thread thread;
   public Pair(Type type, Thread thread) {
      this.type = type; this.thread = thread;
   }
   public boolean equals(Object obj) {
      if (obj instanceof Pair)
         return thread.equals(
                   ((Pair)obj).thread);
      else
         return false;
   }
}
```

Readers/Writers Reading

```
SYNC void startRead() throws IE {
   w.add(new Pair(Type.Reader,ct);
   while (nw > 0 ||
          (!w.isEmpty() &&
           !ct.equals(w.peek().thread))) {
      okToRead.await();
   }
   w.poll();
   nr++;
   if (!w.isEmpty() &&
        w.peek().type == Type.Reader)
      okToRead.signalAll();
}
```

Readers/Writers Reading

- Signal all writers after all readers left
  - Since there are no more readers there must be a waiting writer
  - Wake up all to find the one waiting longest

```
SYNC void endRead() {
   nr--;
   if (nr == 0)
      okToWrite.signalAll();
}
```

Readers/Writers Writing

```
SYNC void startWrite() throws IE {
   w.add(new Pair(Type.Writer,ct));
   while (nr > 0 || nw > 0 ||
          (!w.isEmpty() &&
           !ct.equals(w.peek().thread))) {
      okToWrite.await();
   }
   w.poll();
   nw++;
}
```

Readers/Writers Writing

- On leave: wake up only the appropriate process type to find the first

```
SYNC void endWrite() {
   nw--;
   if (!w.isEmpty() &&
       w.peek().type == Type.Reader)
      okToRead.signalAll();
   else
      okToWrite.signalAll();
}
```

Java 5 classes

- Java 5 contains several useful classes under java.util.concurrent.
- There are classes for Readers/Writers locks and Cyclic Barrier
- If you need these kinds of synchronization when programming, use the libraries as much as possible instead of writing your own code!

Resource Allocation – Single

- A controller controls access to copies of some resource
- Clients make requests to take (acquire) or return (release) one resource
  - A request should only succeed if there is a resource available,
  - Otherwise the request must block

Resource Allocator – PtC

```
monitor ResourceAllocator<E> {
   private Condition free;
   private int avail = N;
   private Queue<E> units = …
   public E acquire() {
      if (avail == 0) wait(free);
      else avail--;
      return units.remove();
   }
   public void release(E e) {
      units.add(e);
      if (empty(free)) avail++;
      else signal(free);
}}
```

Resource Allocation – Java

```
public class ResourceAllocator<E> {
   private Queue<E> units = …;

   public synchronized E allocate()
    throws InterruptedException {
      while (units.size() == 0)
         wait();
      return units.remove();
   }

   public synchronized void release(E e) {
      units.add(e);
      notify();
}}
```

Resource Allocation – Multiple

- Clients requiring multiple resources should not ask for resources one at a time
  - Why would this be bad?
- A controller controls access to copies of some resource
- Clients make requests to take or return *any* number of the resources
  - A request should only succeed if there are sufficiently many resources available,
  - Otherwise the request must block

Resource Allocation – Multiple

```
public class ResourceAllocator<E> {
   private Queue<E> units = …;

   public sync Set<E> allocate(int n)
    throws InterruptedException {
      while (units.size() < n)
         wait();
      return take(n);
   }

   public sync void release(Set<e> ret) {
      units.addAll(ret);
      notifyAll();
}}
```

Synchronisation Shootout

- Semaphores vs Monitors
  - Semaphores
    - Efficient
    - Expressive: any synchronisation (await-statement)
    - Easy to implement
  - Monitors
    - Can monitors implement semaphores?
      - Important theoretical question
      - An illustrative example, but not normal practice
      - Implementing a low-level language construct in a high-level language is not normally a good idea
    - Can semaphores implement monitors?

Implementing Monitors

```
public class MonitorImpl {

    private Semaphore e =
        new Semaphore(1, true);
    private Map<Thread,Semaphore> semMap =
        new HashMap<Thread,Semaphore>();

    protected void lock();
    protected void unlock();

    protected CV newCV();
    protected void wait(CV cv) throws IE;
    protected void signal(CV cv);
}
```

Monitor Entry

- Binary semaphore (lock) for entry

```
protected void lock() {
    e.acquireUninterruptibly();
}

protected void unlock() {
    e.release();
}
```

Condition Variables

- A condition variable is a queue of threads

```
public interface CV extends Queue<Thread> {
}
```

```
private class CVImpl
        extends ArrayDeque<Thread>
        implements CV {}

protected CV newCV() {
   return new CVImpl();
}
```

wait(cv)

- Private semaphore for blocking

```
protected void wait(CV cv) throws IE {
   cv.add(Thread.currentThread());
   Semaphore wait = new Semaphore(0);
   semMap.put(Thread.currentThread(), wait);
   e.release();
   try {
      wait.acquire();
   } finally {
      e.acquireUninterruptibly();
}}
```

signal(cv)

- Signal the first waiting thread on its private semaphore

```
protected void signal(CV cv) {
   if (cv.size() > 0) {
      Thread waiter = cv.remove();
      Semaphore wait = semMap.get(waiter);
      wait.release();
   }
}
```

Monitors vs Semaphores

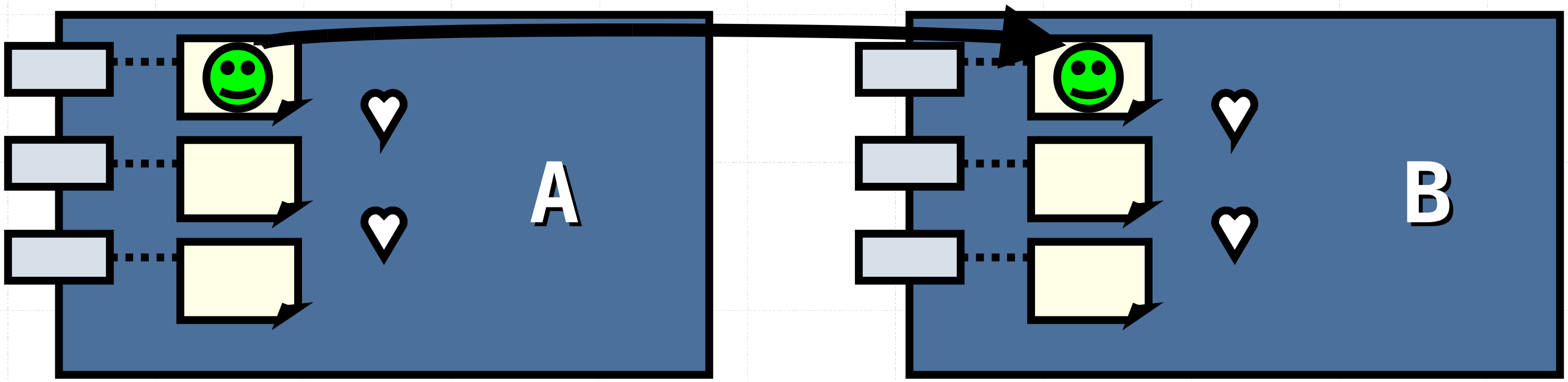
- Semaphores can implement monitors
- Monitors can implement semaphores

$\Downarrow$

- The same expressive power
  - Can implement any await statement
  - Important theoretical result

Nested Monitor Calls

- What happens if monitor **A** calls monitor **B**?



- Four approaches
  - Ban nested calls
  - Release **A**'s lock when entering **B**
    - More concurrency

Nested Monitor Calls

- Four approaches – continued:
  - Maintain lock on **A** while in **B**;
    `wait(…)` in **B** releases both locks
  - Maintain lock on **A** while in **B**;
    return to **A** on leaving **B**
    - `wait(…)` in **B** releases only **B**'s lock
    - less concurrency
    - can lead to deadlock
    - easier to reason about safety properties
      - Ordering access

The Java Case – Recursion

- First a special case
  - What if **A** and **B** are the same object?
  - Reentrant lock – can be re-locked safely

```
public class Reentrant {
   public synchronized void a() {
      b();
      System.out.println(TN+" in a()");
   }
   public synchronized void b() {
      System.out.println(TN+" in b()");
   }
...
```

The Java Case

- This works in a similar way across multiple objects
  - Threads collect the locks as they go
  - If they already have the lock on the object then they proceed
- A `wait(…)` operation releases the lock for the current object only. Other locks are still held.
- Note: this means that you will block while holding locks – a good chance to deadlock!

The Java Case

- The same rules apply to reentrant locks in package `java.util.concurrent.locks`
- Note: collecting locks means that you will block while holding locks
  - A good chance to deadlock!
  - Programming discipline helps
    - Remember the dining philosophers?
    - Ordering access/calls can help avoiding circular waiting

GUI Frameworks

- AWT
  - Attempted to be thread-safe
  - Result: deadlocks possible
- Swing
  - Abandons thread-safety in general
  - One main event-dispatching thread runs all Swing activity
  - Some thread-safe methods are provided
    - For example: `repaint()`

Swing

- Thread-safe Swing
  - Operations modifying Swing components must run in the event-dispatching thread

```
SwingUtilities.invokeLater(Runnable doRun)
```

```
SwingUtilities.invokeAndWait(Runnable doRun)
   throws InterruptedException,
          InvocationTargetException
```

Summary – Java

- Monitor based
  - Signal and Continue semantics
- Native Java
  - `synchronized` methods
  - One implicit condition variable
- Java 5
  - Fully fledged monitors
  - But more explicit programming

Next Time

- Shared-memory programming
  - Only for the insane programmer?
- Message passing
  - AKA Shared-nothing concurrency
  - First look at the possibilities