Lecture 4

Monitors

Summary – Semaphores

- Good news
  - Simple, efficient, expressive
    - Passing the Baton – any await statement
- Bad news
  - Low level, unstructured
    - omit a V: deadlock
    - omit a P: failure of mutex
  - Synchronisation code not linked to the data
    - Synchronisation code can be accessed anywere,
    - but good programming style helps!

Monitors

- A combination of data abstraction and mutual exclusion
  - invented by C.A.R. Hoare [1974]
- Widely used in concurrent programming languages and libraries
  - Java,
  - pthreads,
  - C#,
  - …

Key Features

- A collection of encapsulated procedures
  - a module or a class-like structure
- A single global lock to ensure mutex for all the operations in the monitor
  - Automatic mutex
- A special type of variables called condition variables which are used for condition synchronisation
  - Programmed conditional synchronisation

Aims

- Understand “classical” monitors
  - Examples
  - Standard Variations
  - Pseudo-monitor syntax (similar to the book)
- Understand Java monitors
  - Built-in
  - Library since Java 5

Counter – Pseudo-Syntax

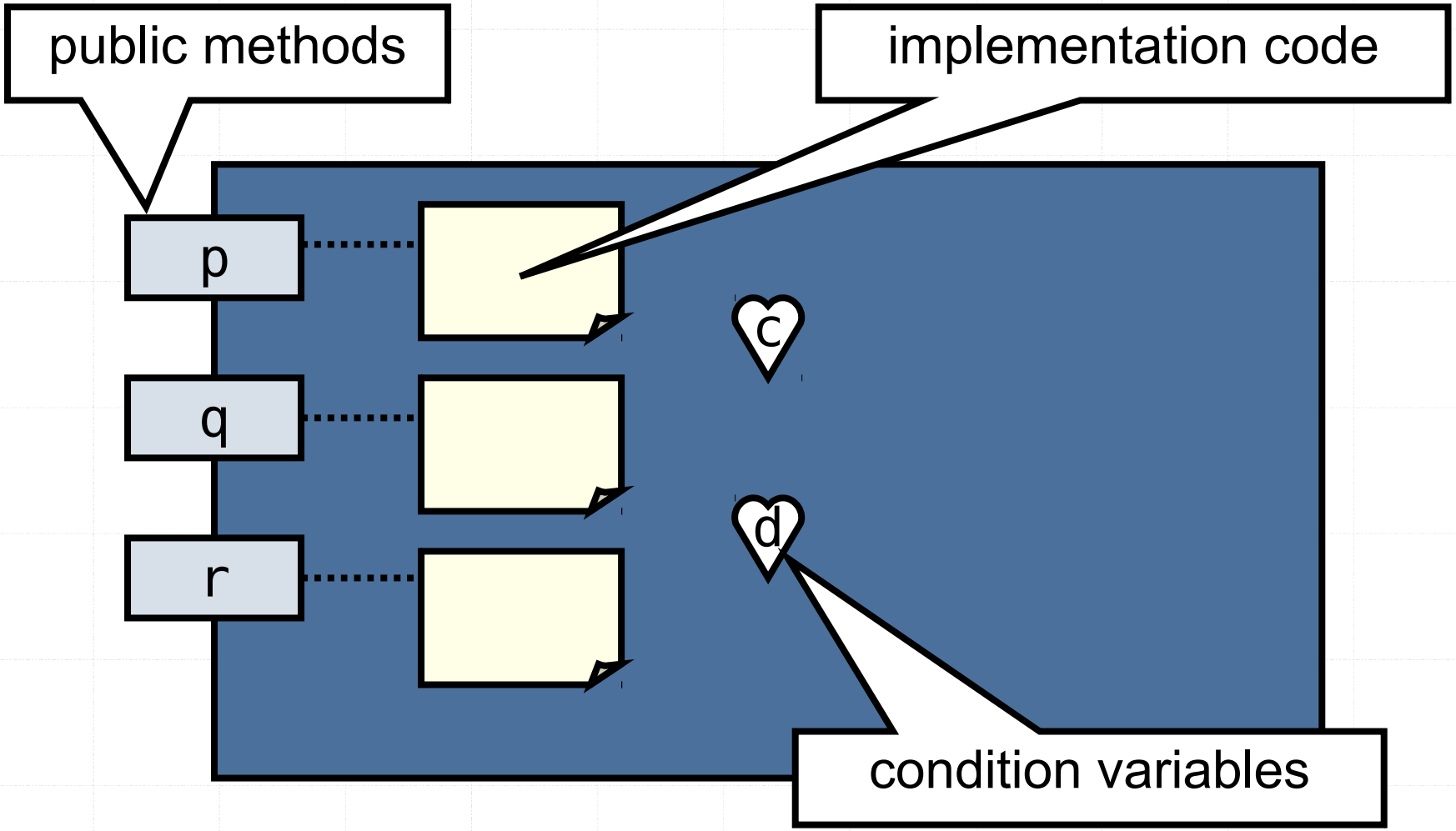
- class “becomes” monitor
  - Mutex for methods

```
monitor sharedCounter {

   private int counter = 0;

   public void increment() {
      counter++;
   }
}
```

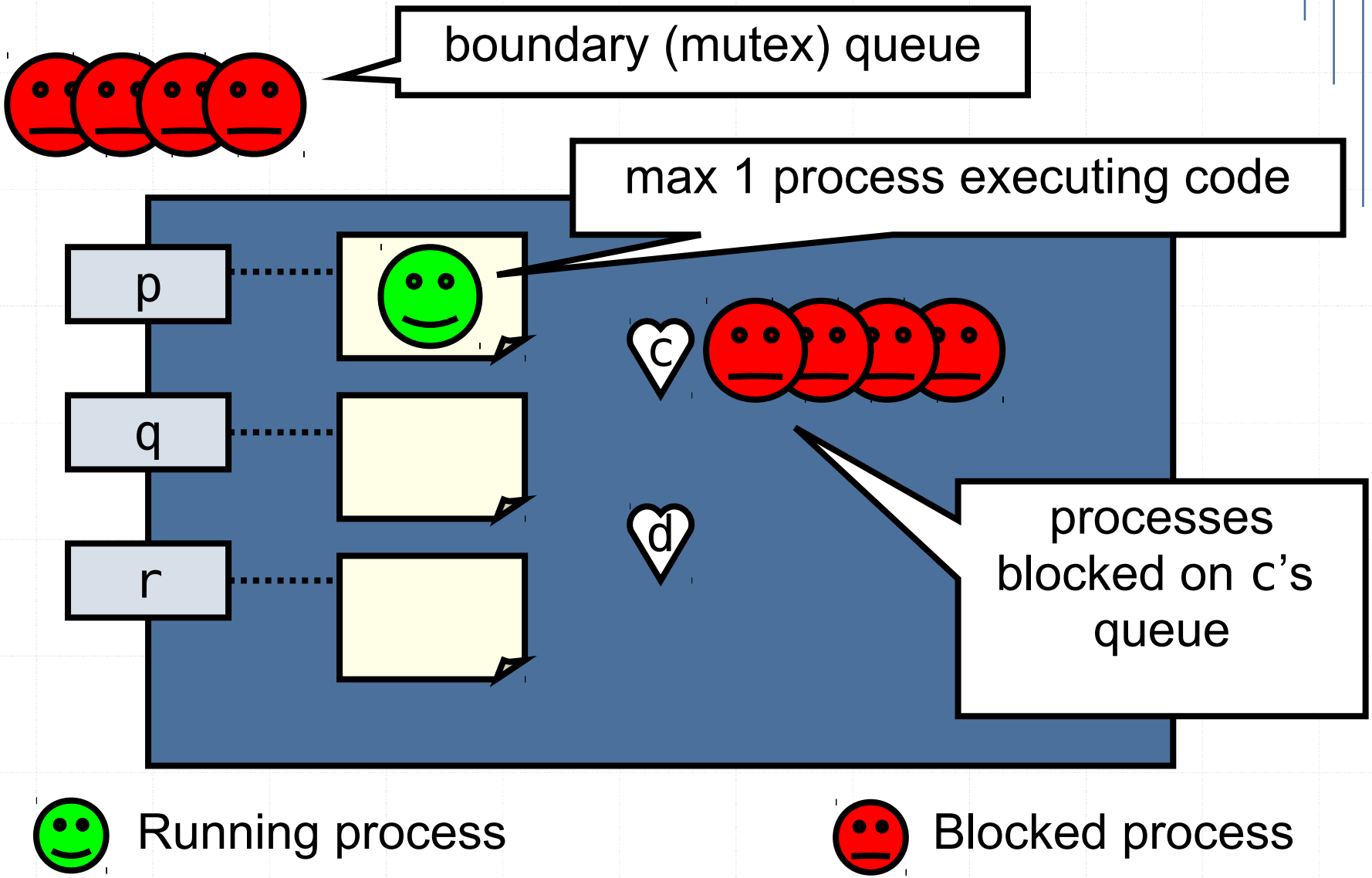
Condition Variables

- Global to the monitor
- Associated to each one is a queue for blocked processes
- Two operations:
  - `wait(c)`, and
  - `signal(c)`

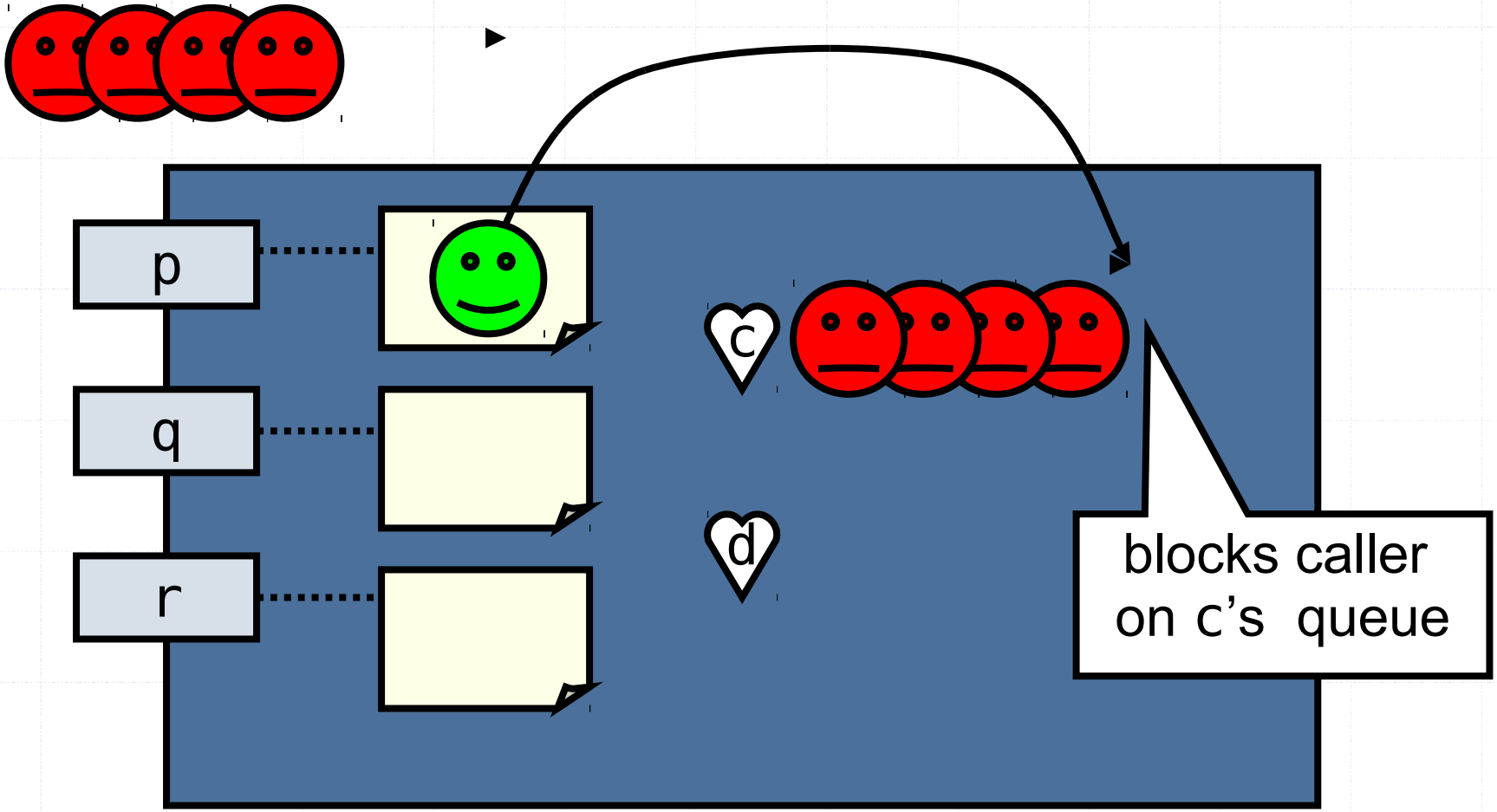
Monitors – Behaviour



A Typical Monitor State



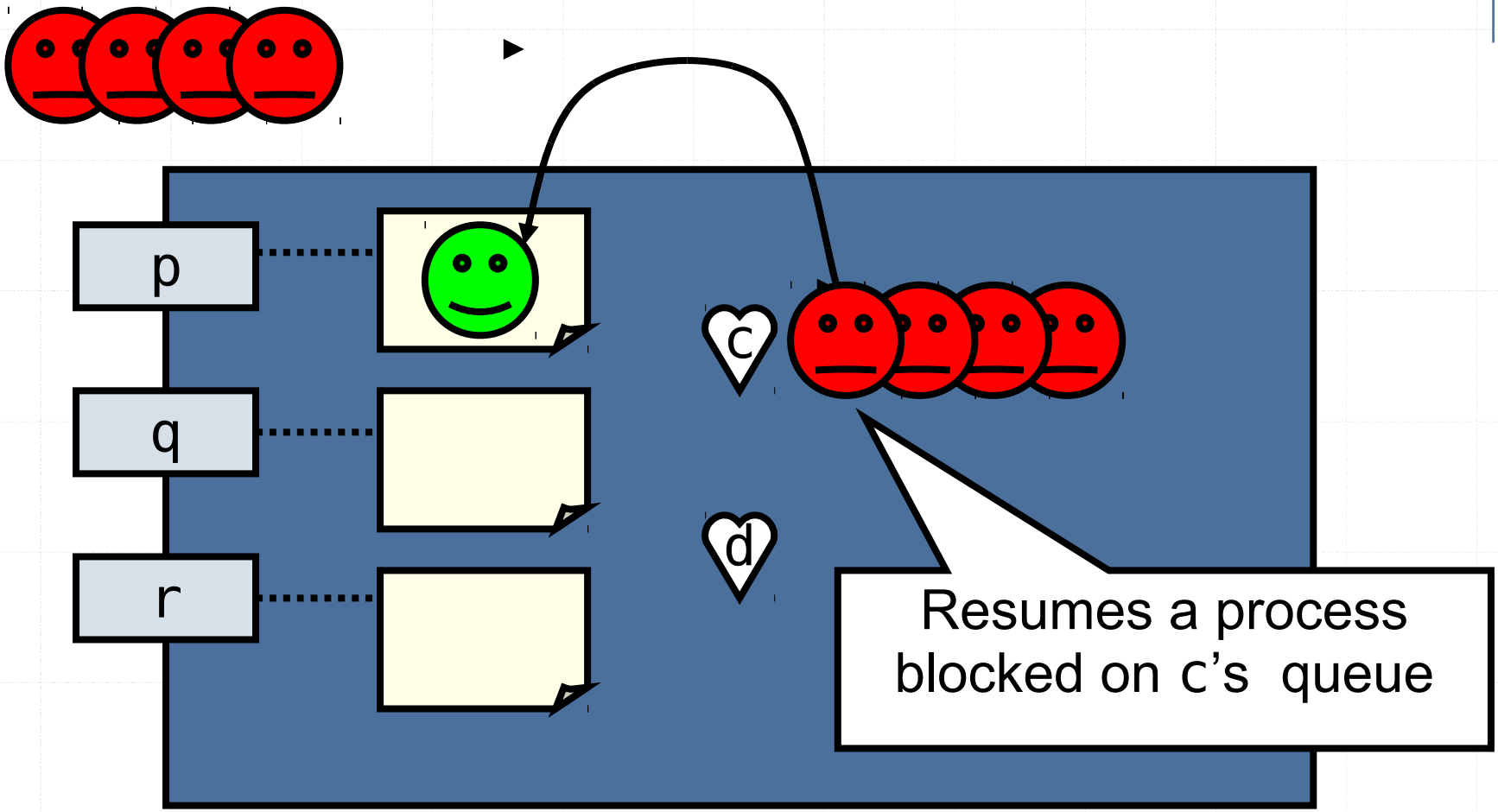
wait(c)



wait(c)

- A bit like the semaphore operation P
  - `wait(c)` blocks the executing process on c
- The blocked process must release the mutex lock on the monitor
- How is it different from P?

signal(c)



continues with the instruction following the `wait(c)`

signal(c)

- A bit like the semaphore operation V
  - `signal(c)` unblocks the first process blocked on c
- What happens with the mutex?
- How is it different from V?

Signal and Exit

- When a process is woken it resumes execution at the instruction after the wait call.
- What about mutex?
- Signal and exit monitors
  - The caller executing `signal(c)` terminates, and hands over the the mutex lock on the monitor to the unblocked process

One-slot Buffer

- Condition variable naming
  - Producers wait until not full
  - Consumers wait until not empty

```
monitor Buffer<E> {
   private Condition notFull;
   private Condition notEmpty;

   private boolean isEmpty = true;
   private E buf = null;
//next slide
```

One-slot Buffer

```
public void put(E e) {
   if (!isEmpty) wait(notFull);
   buf = e;
   isEmpty = false;
   signal(notEmpty);
}

public E get() {
   if (isEmpty) wait(notEmpty);
   E result = buf;
   isEmpty = true;
   signal(notFull);
   return result;
}
```

N-slot Buffer

- Flag `isEmpty` replaced with a counter

```
monitor Buffer<E> {
   private Condition notFull;
   private Condition notEmpty;

   private int count = 0;
   private int front = 0;
   private int rear = 0;
   private E buf[N] = (E[])new Object[N];
//next slide
```

N-slot Buffer

```
public void put(E e) {
   if (count == N) wait(notFull);
   buf[front] = e;
   front = (front+1)%N;
   count++;
   signal(notEmpty);   }

public E get() {
   if (count == 0) wait(notEmpty);
   E result = buf[rear];
   rear = (rear+1)%N;
   count--;
   signal(notFull);
   return result;   }
```

Buffer Shootout

- Semaphores vs Monitors

```
void put(E e) {
   P(empty);
   P(mutexP);
   buf[front] = e;
   front =
      (front+1)%N;
   V(mutexP);
   V(full);
}
```

```
void put(E e) {
   if (count == N)
      wait(notFull);
   buf[front] = e;
   front =
      (front+1)%N;
   count++;
   signal(notEmpty);
}
```

General 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?

Semaphore Monitor

- Monitors can easily implement semaphores

```
monitor Semaphore {

   private int sv;
   private Condition notZero;

   public Semaphore(int sv) {
      this.sv = sv;
   }
//next slide
```

Semaphore Monitor

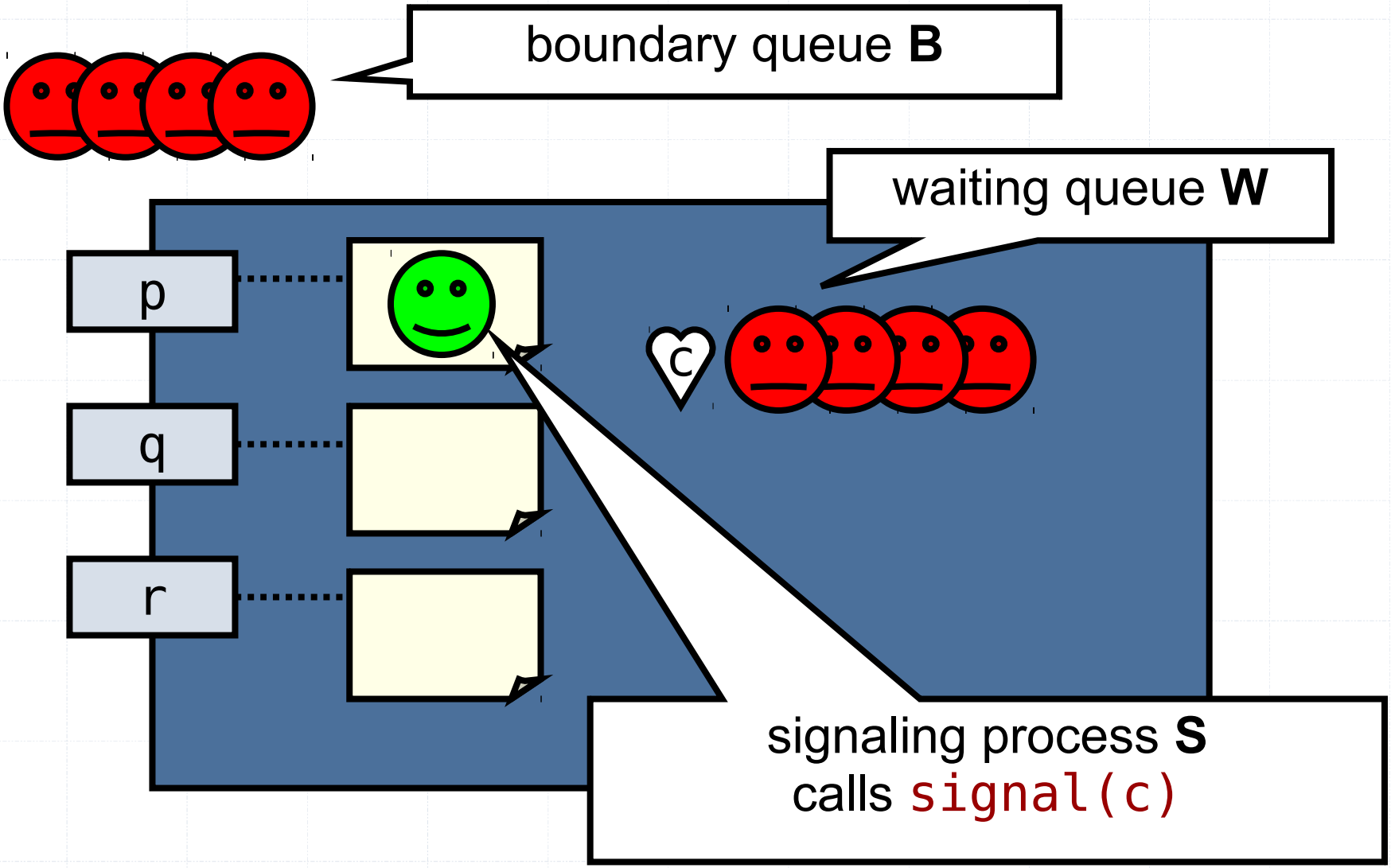
```
public void P() {
   if (sv == 0)
      wait(notZero);
   sv--;
}

public void V() {
   sv++;
   signal(notZero);
}
```

- Quite nice, but …

Signaling Disciplines

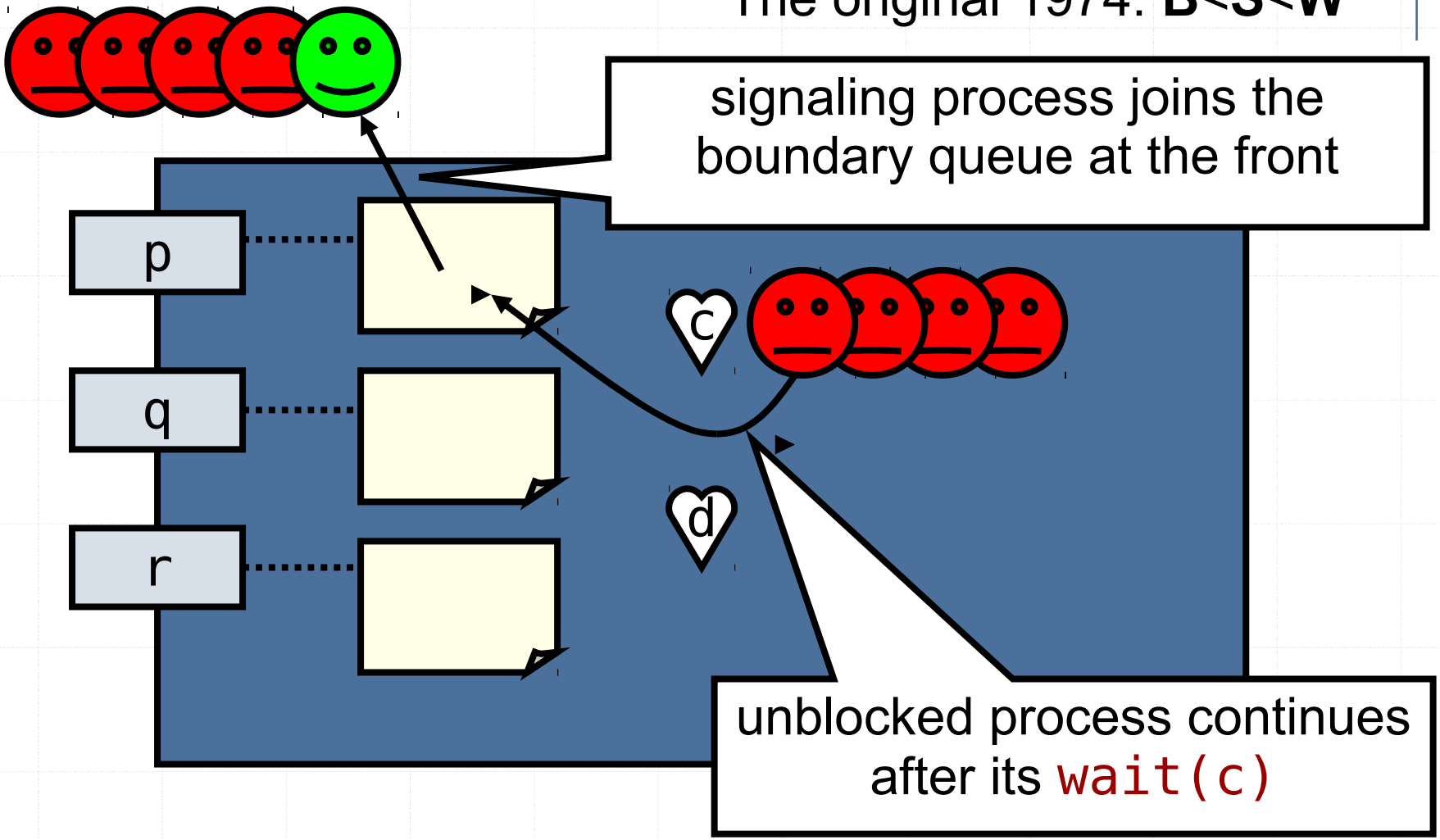
- So far we have looked at the Signal and Exit version of monitors
  - A signal is at the end of the method
  - Mutex is handed over to any woken process
- Other possibilities
  - What if the signal is not at the end of the procedure?
  - What is the scheduling can be different?
  - Several possible semantics exist
    - Ben-Ari: 13 in total but most are no sense

Signal and What Happens Next?

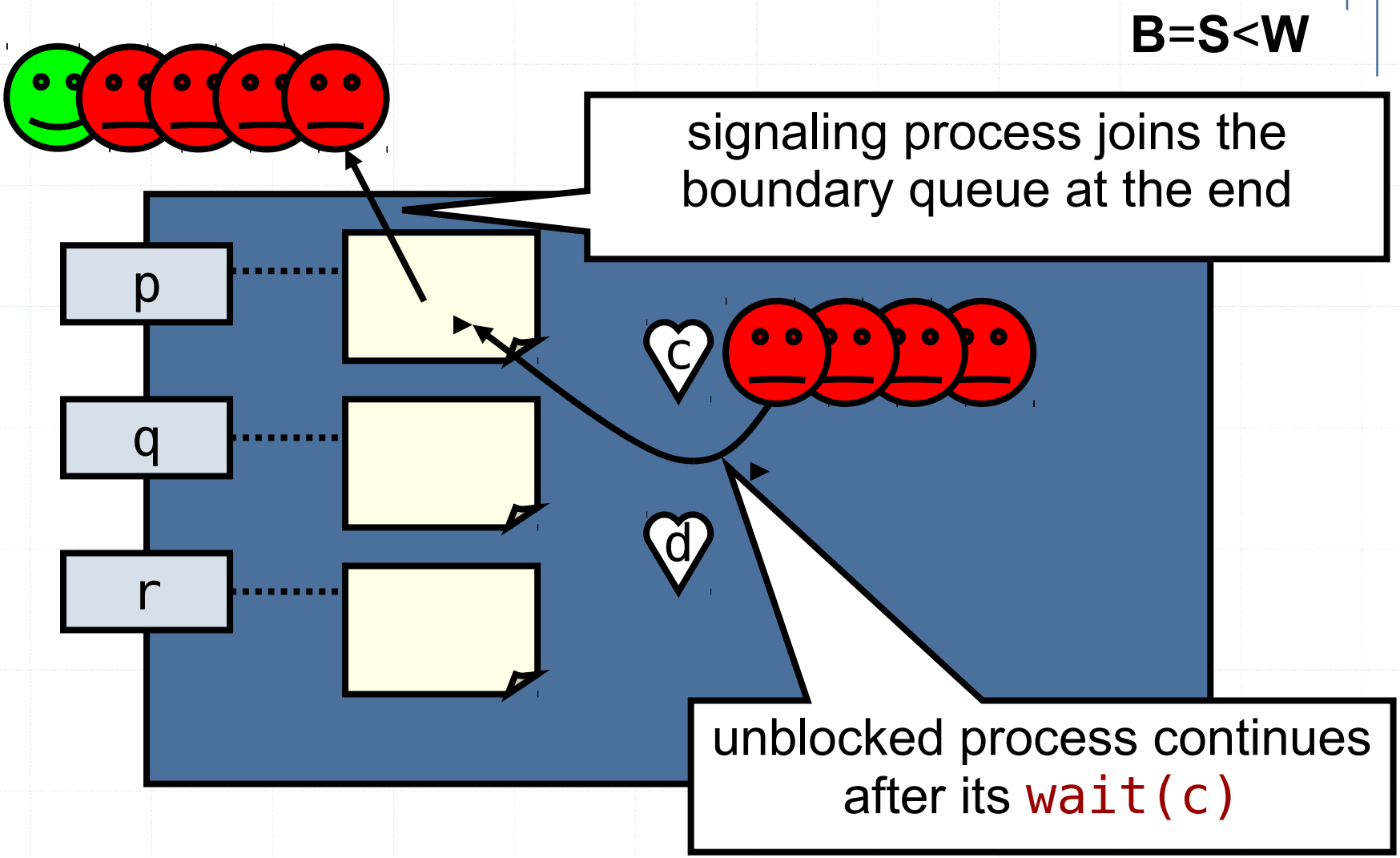


Signal and Urgent Wait

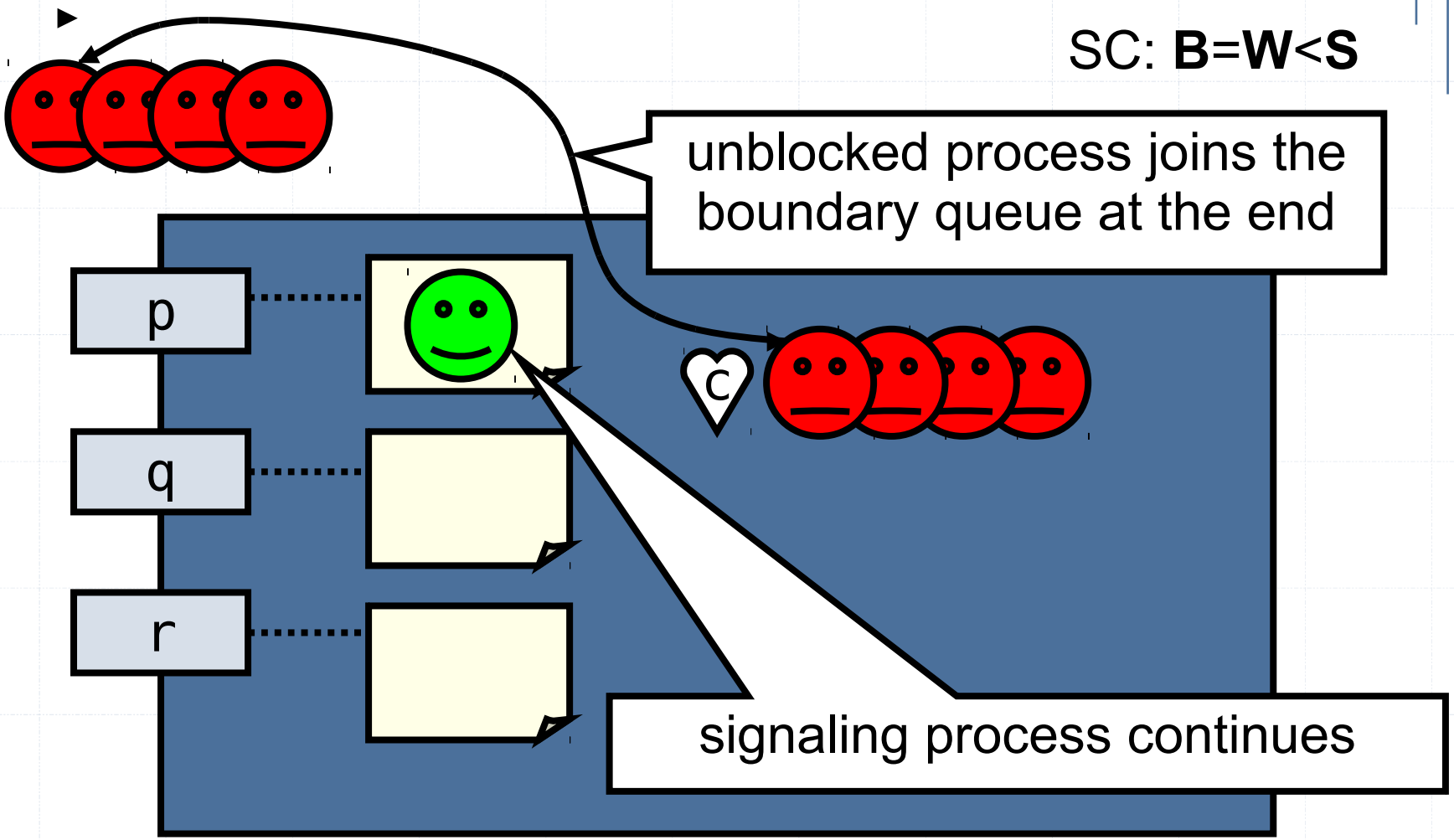
The original 1974: **B<S<W**



Signal and Wait

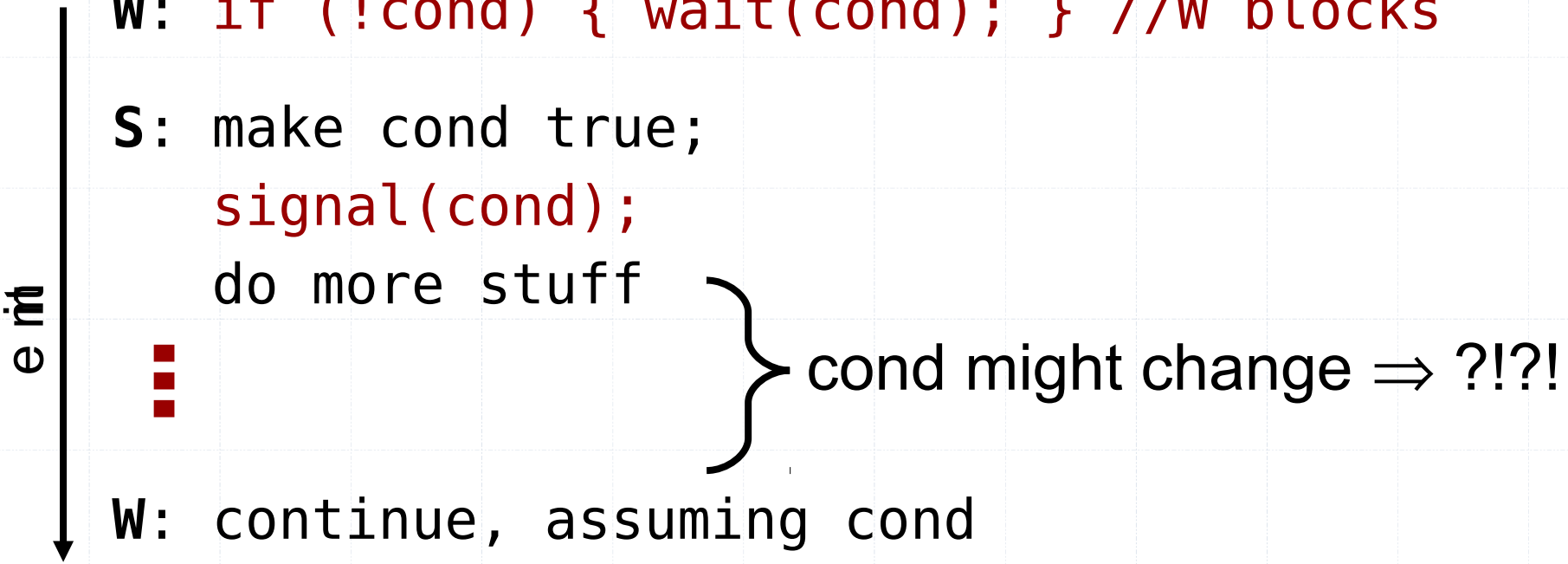


Signal and Continue



Signal and Continue

- Signaler **S** continues while
- Unblocked process **W** joins the boundary
- Typical pattern so far:



Signal and Continue

- We need to adapt our programming style
  - Use “passing the condition” technique, or
  - New pattern

```
public void P() {
   while (sv == 0)
      wait(notZero);
   sv--;
}
```

  - and take a great care about starvation

Signal and Continue

- Perhaps less intuitive to use, but
- Preferred signaling discipline today
  - Compatible with priority-based scheduling
  - Has simpler formal semantics
  - Widely used
    - UNIX
    - pthreads
    - Java
  - Some possible advantages: broadcast signal
    - signalAll operation

SC – Semaphore Monitor

```
public void P() {
   while (sv == 0)
      wait(notZero);
   sv--;
}

public void V() {
   sv++;
   signal(notZero);
}
```

- Quite nice, but …

SC – Semaphore Monitor

- Not a fair semaphore: `signal(notZero)` might be “stolen” by a process on the boundary queue
- A fair semaphore possible by “passing the condition”
  - signaler in effect passes the information that `sv` value is positive to the signalee
- Use the `empty(cv)` primitive to test whether a queue is empty

Semaphore Monitor

- Fair semaphore for all signaling disciplines
- Passing the condition
  - Monitor invariant is important
  - $\Box(\neg$ `empty(notZero)` $\Rightarrow$ `sv==0`$)$

```
public P() {
   if (sv == 0)
      wait(notZero);
   else
      sv--;
}
```

```
public V() {
   if (empty(notZero))
      sv++;
   else
      signal(notZero);
}
```

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

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

Java wait Operations

```
public final void wait()
      throws InterruptedException
```

- Blocks on the object's condition variable
- The waiting thread releases the synchronization lock associated with the object
- Note: "condition variable" is not standard Java terminology! Simply "condition" is used.

Java signal Operations

```
public final void notify()
```

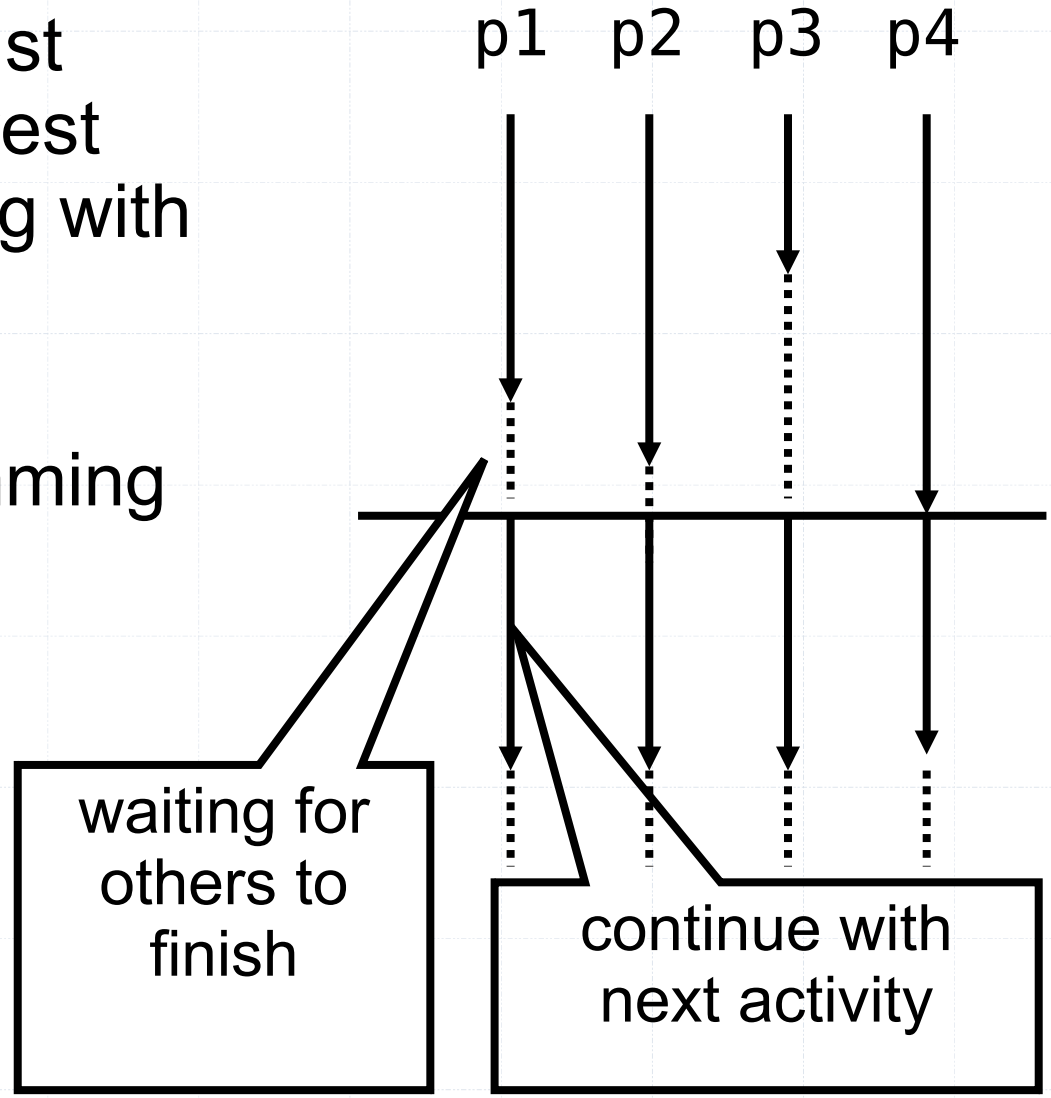
- Wakes up a single thread that is waiting on this object's queue

```
public final void notifyAll()
```

- Wakes up all threads that are waiting on this object's queue

Barrier Synchronisation

- N processes must wait for the slowest before continuing with the next activity
- Widely used in parallel programming



Barrier Monitor

```
public class CyclicBarrier {

   private int arrived = 0;
   private int N;

   public CyclicBarrier(int N) {
      this.N = N;
   }
//next slide
```

Barrier Monitor

- Simple but not 100% reliable solution

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

Java – Passing the Condition

- Cannot be used directly!
- Both `wait()` and `condition.await()`
  - Spurious wakeup is permitted
    - not notified,
    - not interrupted,
    - no timing out
  - Applications must guard against it
  - Always have waiting inside a while loop

```
while (condition)
   wait();
```

Java Semaphore Monitor – Unfair

```
public class Semaphore {
      private int sv;
   public Semaphore(int sv) { this.sv = sv; }
   public synchronized void P() throws IE {
      while (sv == 0)
         wait();
      sv--;
   }
   public synchronized void V() {
      sv++;
      notify();
   }
}
```

Java Semaphore Monitor – Almost-fair

- Keep a local queue of waiting processes to guarantee fair wakeup

```
public class Semaphore {
   private int sv;
   private Queue<Thread> w =
      new ArrayDeque<Thread>();

   public Semaphore(int sv) {
      this.sv = sv;
   }
//next slides
```

Java Semaphore Monitor – Almost-fair

- Signal all waiting threads to make sure that the intended one gets the signal
  - Use `notifyAll()`

```
public synchronized void V() {
    sv++;
    notifyAll();
}
```

Java Semaphore Monitor – Almost-fair

```
public synchronized void P()
 throws InterruptedException {
   Thread ct = Thread.currentThread();
   w.add(ct);
   while (sv==0 ||
           !ct.equals(w.peek())) {
      wait();
   }
   w.poll();
   sv--;
}
```

Java Semaphore Monitor – Analysis

- Fairness
  - Wakeup uses local queue
  - Entry into synchronized methods?
    - Java does not specify fairness for the boundary queue
    - Sun's JVM is said to be fair
    - There is apparently at least one know JVM that is using LIFO for the boundary queue
- True fairness
  - Enter Java 5

Fair Semaphore – Java 5 Locks

- Package `java.util.concurrent.locks`

```
public ReallyFairSemaphore {
      private int sv;
      private final Lock lock =
         new ReentrantLock(true);
      private final Condition notZero =
         lock.newCondition();
      private Queue<Thread> w =
         new ArrayDeque<Thread>();
   public ReallyFairSemaphore(int sv) {
      this.sv = sv;
   }  //next slides
```

Fair Semaphore – Java 5 Locks

```
public void V() {
    lock.lock();
    try {
        sv++;
        notZero.signalAll();
    }
    finally {
        lock.unlock();
    }
}
```

Fair Semaphore – Java 5 Locks

```
public void P() throws InterruptedException {
   lock.lock();
   try {
      Thread ct = Thread.currentThread();
      w.add(ct);
      while (sv==0 ||
               !ct.equals(w.peek())) {
         notZero.await();
      }
      w.poll();
      sv--;
   } finally {   lock.unlock();   }}
```

Thread.interrupt()

- Two possible effects
  - Internal Thread flag is set, or
  - Causes blocked threads to wake up and raise the InterruptedException
- Will immediately wake the thread if it tries to block/sleep
- Difficult to use safely as a programming primitive
  - Can leave objects in hard-to-predict states
- Nevertheless, very useful for final thread termination if threads can be in blocked state

Stopping a Process – Java

- Final thread termination

```
public void run() {
   try {
      while (!interrupted())
         //Do some work here
   } catch (InterruptedException e) {}
}

public void shutdown() {
   interrupt();
}
```

Summary – Monitors

- Allow better structured programming
- As expressive as semaphores
- Various monitor signaling semantics
- Practical side: Java monitors
  - Expressive, though complex
- More classic problems
  - barrier sync
- Next time
  - More Java monitors