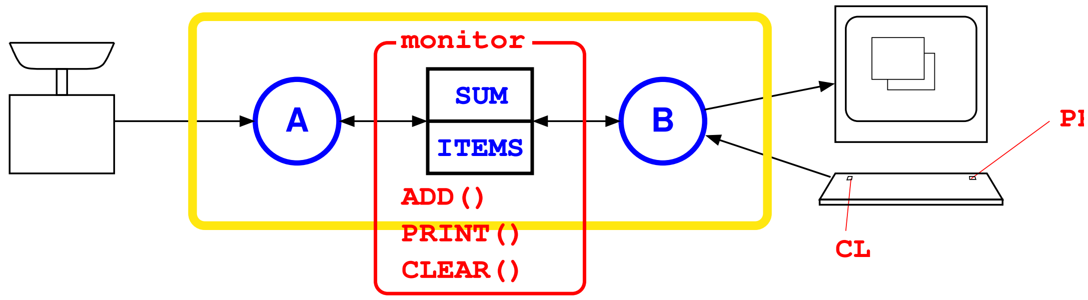
Course 02158

Monitors

Hans Henrik Løvengreen

DTU Compute

The Sharing Problem




```
process A =
  loop
    read w
    ADD(w)

  end loop
```

```
process B =
  loop
    read key
    case key
      PR : PRINT()

      CL : CLEAR()

  end loop
```

Sharing via Monitor

- **monitor** Balance

  var SUM, ITEMS : integer := 0;

  procedure ADD(w : integer)
  SUM := SUM + w
  ITEMS := ITEMS + 1

  procedure PRINT()
  if ITEMS $\neq$ 0 **then**
  print SUM/ITEMS

  procedure CLEAR()
  SUM := 0
  ITEMS := 0

  end

Monitors

- Hoare/Brinch Hansen 1973:

  Monitor = data abstraction + atomicity + synchronization

Data abstraction

- Given by class construct: Private data variables + operations

Atomicity

- By implicit mutual (or R/W) exclusion among operations
- By explicit use of critical sections in operations

Synchronization

- By explicit *condition queues* (*wait*, *signal*)
  - Many variations in sematics
- By use of *guards* (**when** B)

Condition Queues

- Explicit mechanism for condition synchronization within monitors
- A condition queue is associated with a monitor, e.g. by declaration

Basic queue operations

wait(c)	Leave monitor and enter c *atomically*	
signal(c)	Wake up a single process waiting on c if any	
signal_all(c)	Wake up all processes waiting on c	[SC only]

- Different semantics for signalling process:

SC	Signal-and-continue. Signaller continues in monitor
SW	Signal-and-wait. Woken process takes over monitor
Hoare	Signal-and-urgent-wait
	SW + signalling processes have priority to reenter

Auxiliary Condition Queue Operations

- Given: **var** *c* : *condition*

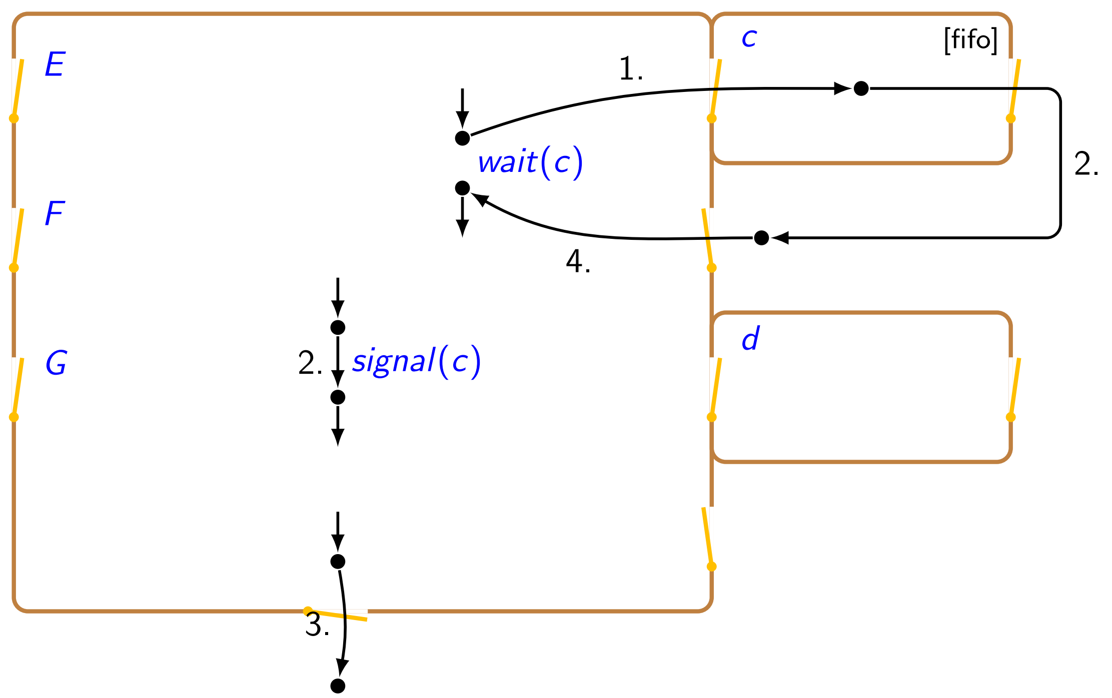
Queue size

- *empty*(c)   No processes are currently waiting on *c*
- Queue length can be maintained in explicit monitor variables.

Priority Queuing

- *wait*(c, *rank*)   Wait in *c* according to *rank* (ascending)
  minrank(c)   Return rank of first process waiting in *c*

Monitor Access — signal-and-continue



Semaphore Monitor

- **monitor** *Semaphore*

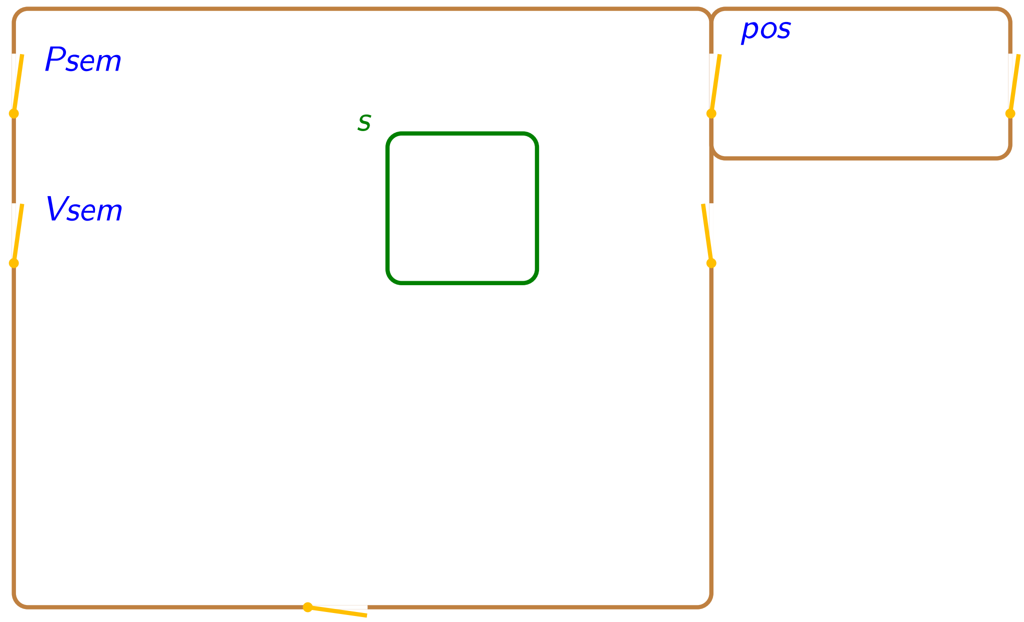
  var s : *integer* := 0; — Semaphore value
  pos : *condition*;

  procedure *Psem*()
  while $s = 0$ **do** *wait*(*pos*);
  $s := s - 1$;

  procedure *Vsem*()
  $s := s + 1$;
  signal(*pos*);

  end

Monitor Access — semaphore



Bounded Buffer

- **monitor** *BoundedBuffer*

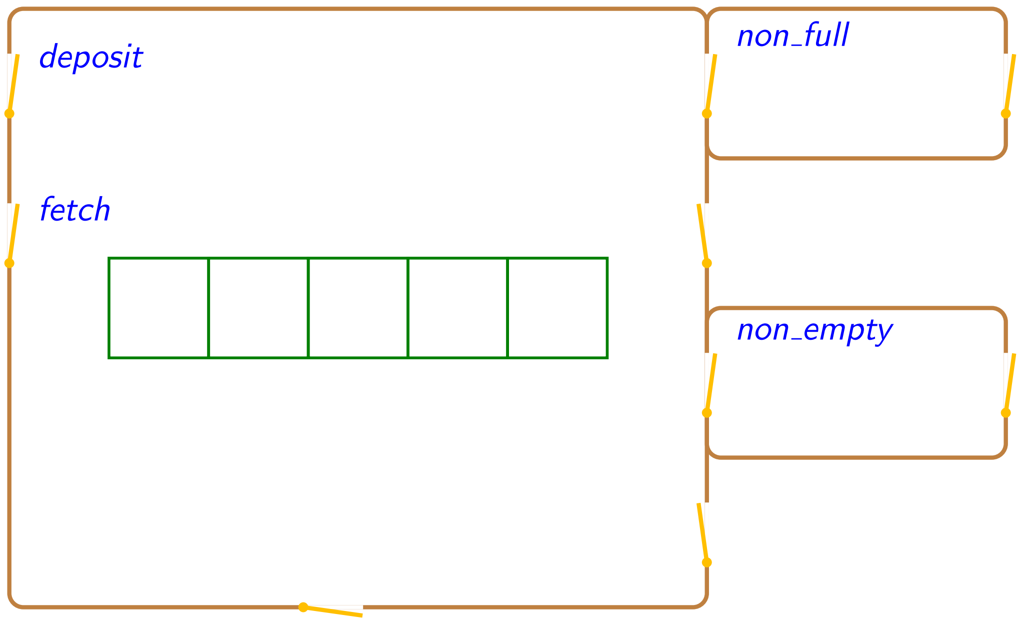
  var $buf[1..n] : T;$
  $in, out, count : integer := 0;$
  $not\_full, not\_empty : condition;$

  procedure $deposit(data : T)$
  while $count = n$ **do** $wait(not\_full);$
  $buf[in] := data;$
  $in = in + 1$ **mod** $n; \quad count + +;$
  $signal(not\_empty)$

  procedure $fetch($**var** $data : T)$
  while $count = 0$ **do** $wait(not\_empty);$
  $data := buf[out];$
  $out = out + 1$ **mod** $n; \quad count - -;$
  $signal(not\_full)$

  end

Monitor Access — bounded buffer



Monitor Invariants

Idea

- To express local safety properties ensured by the *atomicity* of monitor operations.

Definition

- A *monitor invariant* I is a predicate on the state of a monitor M that must be true whenever the monitor is free.
- A monitor is *free* when new processes may start executing monitor operations.

Consequences

- Any process can assume I at the start of a monitor operation.
- The invariant will put *constraints* on the order of operations.
- The constraints may express properties of the whole program.

Stating and Proving Monitor Invariants

- A monitor invariant I may be stated in terms of
  - The monitor variables
  - Added history variables
  - The state of condition queues
  - The state of reentry/signalling queues

Notation

- $\mathit{waiting}(c)$ Number of processes waiting on c
  $\mathit{woken}(c)$ Number of processes awakened from c
- Refinement, e.g.:

  $\mathit{waiting}_{op}(c)$ Number of processes waiting on c from operation op

Inductive Proof

- I holds in the initial monitor state
- I is preserved by any *stretch of activity*

Semaphore Monitor

- **monitor** *Semaphore*

  var s : *integer* := 0; — Semaphore value
  pos : *condition*;
  v, p : *integer* := 0; — History variables

  procedure *Psem*()
  while $s = 0$ **do** *wait*(pos);
  $s := s - 1$;
  $p := p + 1$

  procedure *Vsem*()
  $s := s + 1$;
  signal(pos);
  $v := v + 1$

  end
- Semaphore invariant: $p + s = v \wedge s \geq 0$
- Pre-liveness: $waiting(pos) > 0 \Rightarrow s \leq woken(pos)$

Semaphore Monitor (FIFO)

- **monitor** *FIFOsemaphore*

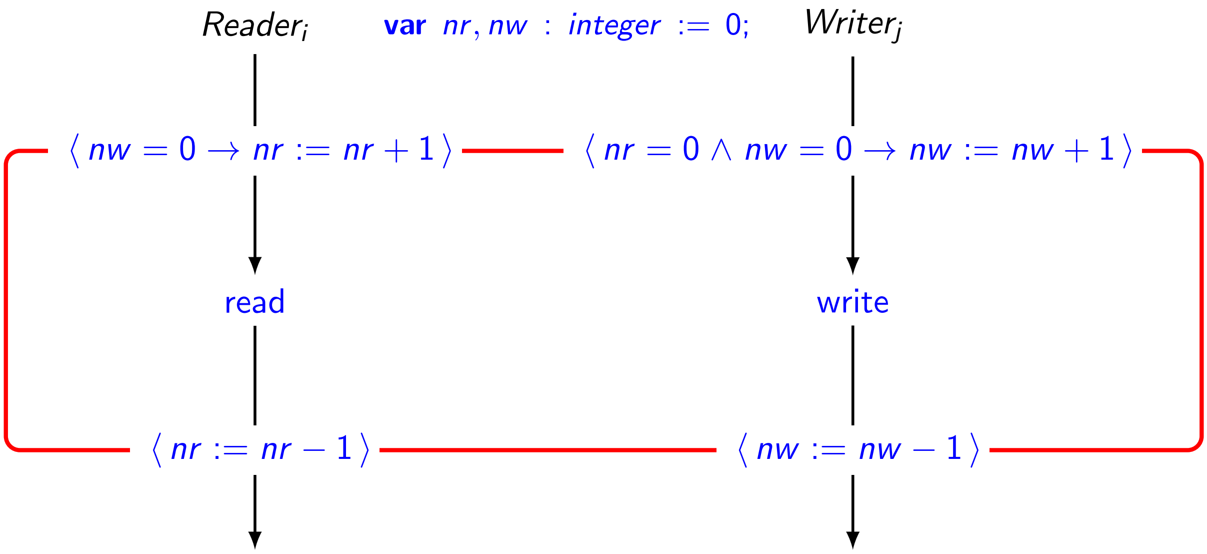
  var s : *integer* := 0; — Semaphore value
  pos : *condition*;
  v, p : *integer* := 0; — History variables

  procedure *Psem*()
  if $s = 0$ **then** *wait*(pos)
  else $s := s - 1$;
  $p := p + 1$

  procedure *Vsem*()
  if *empty*(pos) **then** $s := s + 1$
  else *signal*(pos);
  $v := v + 1$

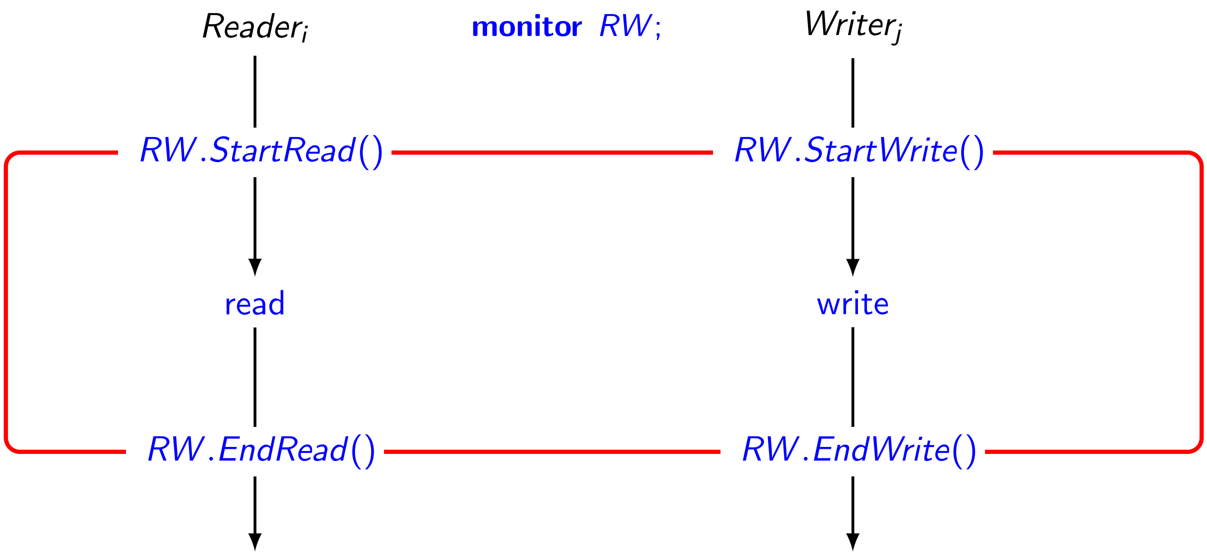
  end
- Semaphore invariant: $p + woken(pos) + s = v \wedge s \geq 0$
- Pre-liveness: $waiting(pos) > 0 \Rightarrow s = 0$

Reader/Writer with Coarse-grained Atomic Actions



- R/W invariant: $nw \leq 1 \wedge (nr = 0 \vee nw = 0)$

Reader/Writer with Coarse-grained Atomic Actions



- R/W invariant: $nw \leq 1 \wedge (nr = 0 \vee rw = 0)$ [inside monitor]

Reader/Writer Monitor — Preliminary: All woken

- **monitor** *RW*

```
var nr, nw : integer := 0;
    ok_rd, ok_wr : condition;

procedure StartRead(){
  while nw > 0 do wait(ok_rd);
  nr++;
  signal_all(ok_rd);
  signal_all(ok_wr)
}

procedure StartWrite(){
  while nr > 0 ∨ nw > 0 do wait(ok_wr);
  nw++;
  signal_all(ok_rd);
  signal_all(ok_wr)
}

procedure EndRead(){
  nr--;
  signal_all(ok_rd);
  signal_all(ok_wr)
}

procedure EndWrite(){
  nw--;
  signal_all(ok_rd);
  signal_all(ok_wr)
}

end
```

Reader/Writer Monitor — Optimized

- **monitor** *RW*

```
var nr, nw : integer := 0;
    ok_rd, ok_wr : condition;

procedure StartRead(){
  while nw > 0 do wait(ok_rd);
  nr++;
}

procedure StartWrite(){
  while nr > 0 ∨ nw > 0 do wait(ok_wr);
  nw++;
}

procedure EndRead(){
  nr--;
  if nr = 0 then signal(ok_wr)
}

procedure EndWrite(){
  nw--;
  signal_all(ok_rd);
  signal(ok_wr)
}

end
```

Timer Monitor — covering condition

- **monitor** *Timer*

```
var tod : integer := 0;
    check : condition;

procedure delay(interval : integer)
  var waketime : integer;
  waketime := tod + interval;
  while waketime > tod do
    wait(check);

procedure tick()
  tod := tod + 1;
  signal_all(check)

end
```

Java Monitors

- Follows standard SC monitor notion, but uses explicit mutual exclusion

Object region

- Every object has an associated critical region (lock)
- Critical section of o-region:

$$\textbf{synchronized}\ (o)\{\ldots\}$$

- A method declared **synchronized** becomes a critical section
- In a monitor class, every method should be synchronized

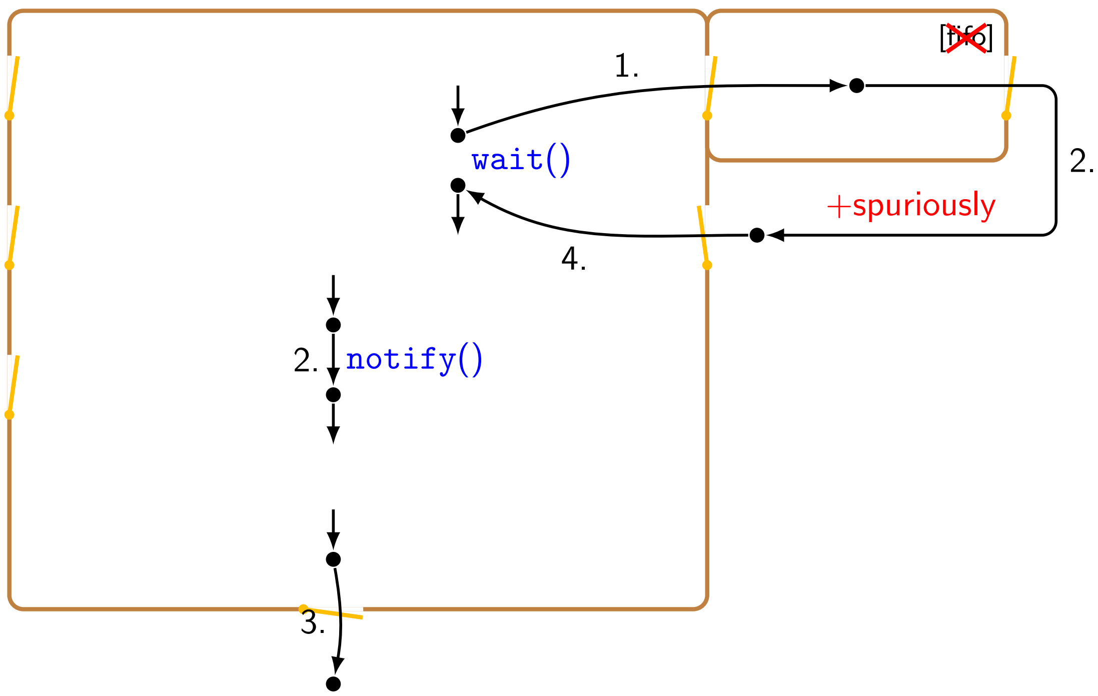
Condition queue

- Every object region has a **single** condition queue with operations:

`wait()`	Wait in the (anonymous) condition queue
`notify()`	Wake up one thread on the condition queue
`notifyAll()`	Wake up all threads on the condition queue

- Condition queue is *unordered*
- May suffer from spurious wakeups — and they really do!

Monitor Access — Java



Sharing Monitor — Java

```
public class Balance {

  private int sum = 0, items = 0;

  public void ADD(int v) {
    synchronized (this) {
      sum = sum + v;
      items++;
    }
  }

  public void PRINT() {
    synchronized (this) {
      if (items != 0) System.out.println(sum/items);
    }
  }
  ⋮
}
```

Sharing Monitor — Java

```
public class Balance {

  private int sum = 0, items = 0;

  public synchronized void ADD(int v) {
    sum = sum + v;
    items++;
  }

  public synchronized void PRINT() {
    if (items != 0) System.out.println(sum/items);
  }

  public synchronized void CLEAR() {
    sum = 0;
    items = 0;
  }

}
```

Semaphore Monitor — Java

```
public class Semaphore {

    private int s = 0;

    public Semaphore(int s0) {
        if (s0 >= 0)  s = s0;  else  throw new Error(s0);
    }

    public synchronized void P() throws InterruptedException {
        while (s == 0) wait();
        s--;
    }

    public synchronized void V() {
        s++;
        notify();
    }
}
```

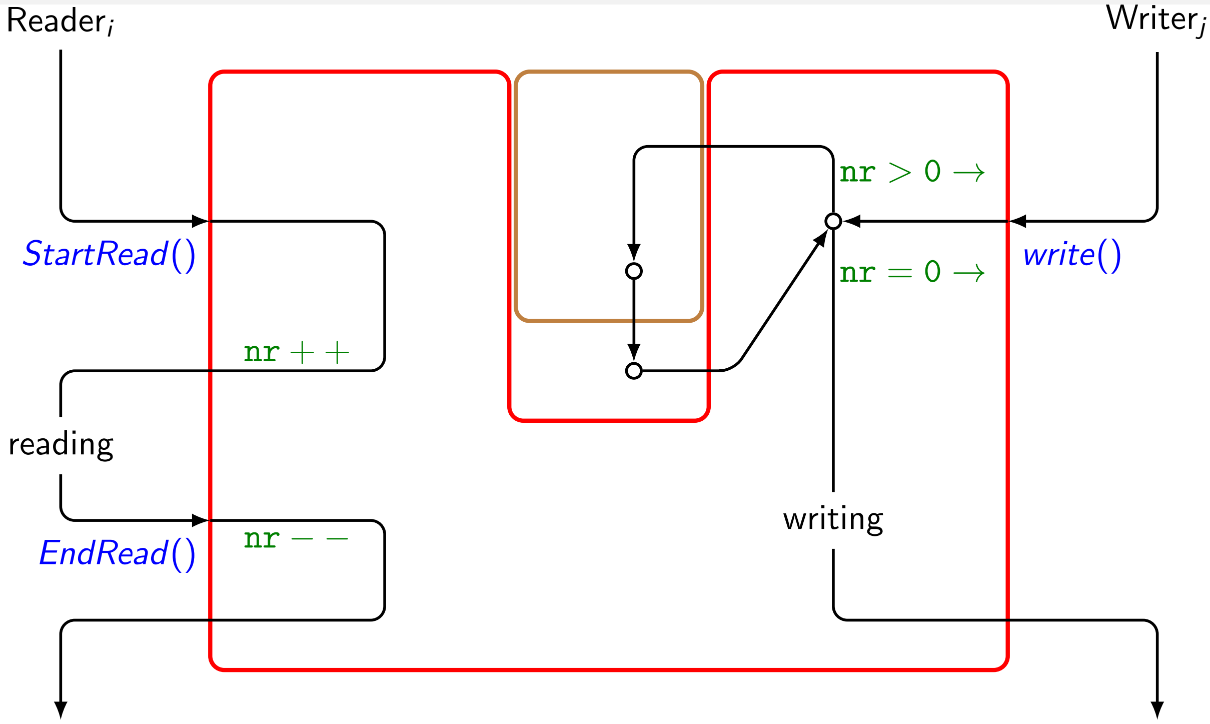
Reader/Writer Monitor — Java [Andrews] I

```
class ReaderWriter {

  Data data = new Data();
  int nr  = 0;

  private synchronized void StartRead() {
    nr++;
  }

  private synchronized void EndRead() {
    nr--;
    if (nr == 0) notify();
  }

  ⋮
```

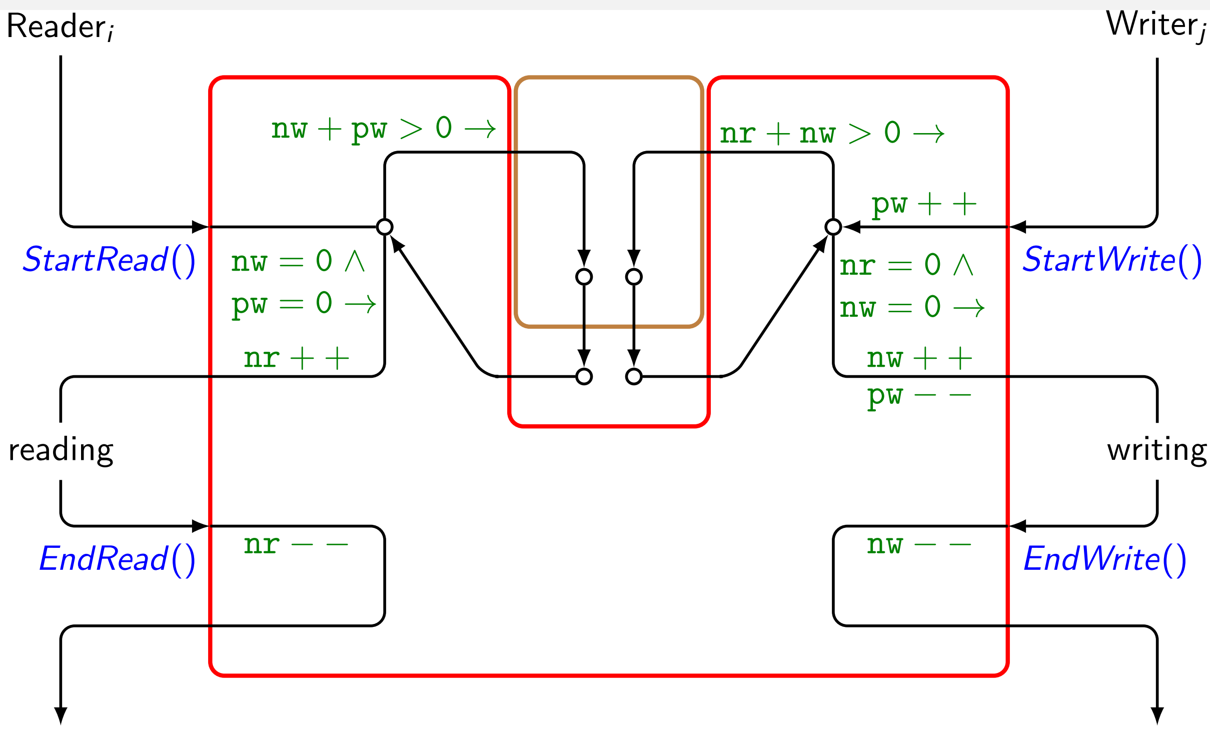
Reader/Writer Monitor — Java [Andrews] II

```
class ReaderWriter {

  ⋮

  public int read() {
    StartRead();
    int res = data.read();
    EndRead();
    return res;
  }

  public synchronized void write(int x) {
    while (nr > 0 )
      try {wait();} catch (InterruptedException e) {}
    data.write(x);
  }

}
```

Java Reader/Writer Control — Andrews



Java Reader/Writer Control — Writer priority



Reader/Writer Monitor — Java I

```
class ReaderWriter {

  int pw = 0;                          // pending writers
  int nr = 0;                          // active readers
  int nw = 0;                          // active writers

  public synchronized void StartRead() {
    while (nw + pw > 0)
      try {wait();} catch (InterruptedException e) {}
    nr++;
  }

  public synchronized void EndRead() {
    nr--;
    if (nr == 0)
      notifyAll();
  }
```

Reader/Writer Monitor — Java II

```
class ReaderWriter {

  ⋮

  public synchronized void StartWrite() {
    pw++;
    while (nr > 0 || nw > 0)
      try {wait();} catch (InterruptedException e) {}
    pw--;
    nw++;
  }

  public synchronized void EndWrite() {
    nw--;
    notifyAll();
  }

}
```