Outline

- ❑ Monitors
  - Monitors in Java
- ❑ Barrier synchronization
- ❑ The sleeping barber problem
- ❑ Readers and Writers
- ❑ One-way tunnel

Monitors - higher-level synchronization *(Hoare, Hansen, 1974-5)*

- Semaphores and event-counters are low-level and error-prone
- **Monitors** are a programming-language construct
- Mutual exclusion constructs generated by the compiler. Internal data structures are invisible. ***Only one process is active*** in a monitor ***at any given time -*** *high level mutual exclusion*
- Monitors support <u>*condition variables*</u> for thread cooperation.
- Monitor disadvantages:
  - May be less efficient than lower-level synchronization
  - Not available from all programming languages

Monitors

Only one monitor procedure active at any given time

```
monitor example
    integer i;
    condition c;

    procedure p1( );
    .
    .
    .
    end;

    procedure p2( );
    .
    .
    .
    end;
    end monitor:
```

Slide taken from a presentation by Gadi Taubenfeld from IDC

Monitors: Condition variables

- Monitors guarantee "automatic" mutual exclusion
- Condition variables enable other types of synchronization
- Condition variables support two operations: *wait* and *signal*
  - **Signaling has no effect if there are no waiting threads!**
- The monitor provides *queuing* for *waiting* procedures
- When one operation *waits* and another *signals* there are two ways to proceed:
  - The signaled operation will execute first: signaling operation immediately followed by *block()* or *exit_monitor* (**Hoare** semantics)
  - The signaling operation is allowed to proceed

type *monitor-name* **= monitor**
variable declarations

procedure entry *P1* (…);
begin … end;

procedure entry *P2* (…);
begin … end;

.
.
.

procedure entry *Pn* (…);
begin … end;

begin
initialization code
end

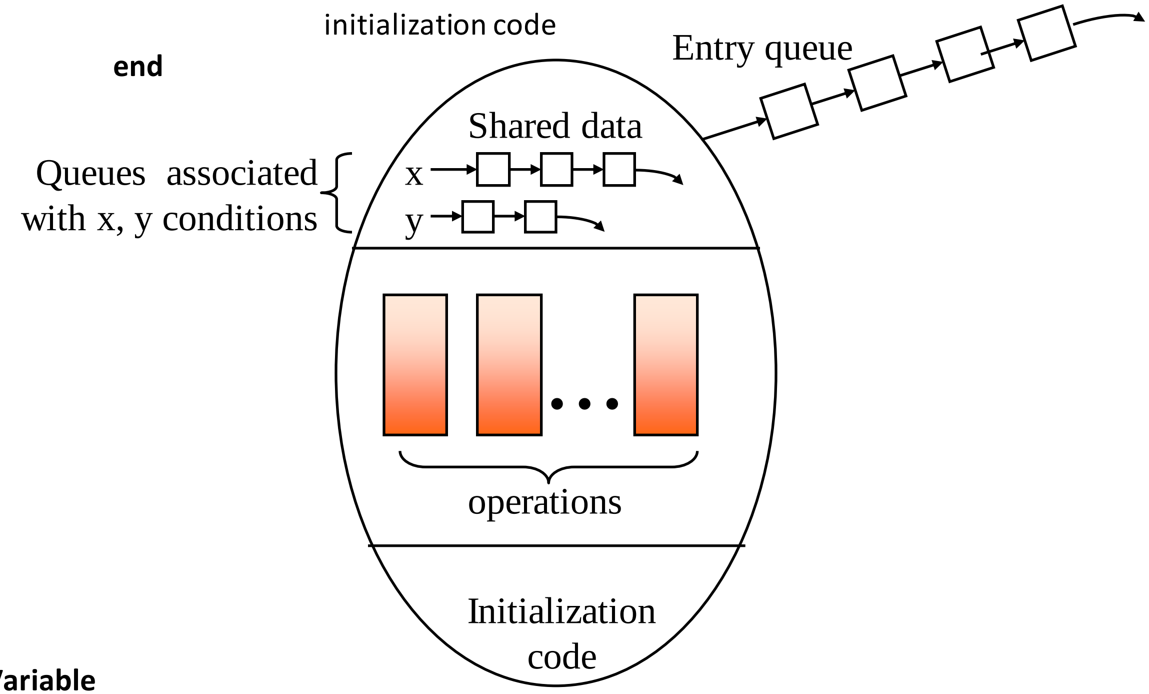


Figure 6.20 Monitor with Condition Variable

Bounded Buffer Producer/Consumer with Monitors

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
```

```
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

This code only works if a signaled thread is the next to enter the monitor (Hoare)

Any problem with this code?

Slide taken from a presentation by Gadi Taubenfeld from IDC

Issues of non-Hoare semantics

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
```

```
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

1. The buffer is full, k producers (for some k>1) are waiting on the full condition variable. Now, N consumers enter the monitor one after the other, but only the first sends a signal (since count==N-1 holds for it). Therefore only a single producer is released and all others are not. The corresponding problem can occur on the empty semaphore.

Slide taken from a presentation by Gadi Taubenfeld from IDC

Issues of non-Hoare semantics *(cont'd)*

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
```

```
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

2) The buffer is full, a single producer p1 sleeps on the full condition variable. A consumer executes and makes p1 ready but then another producer, p2, enters the monitor and fills the buffer. Now p1 continues its execution and adds another item to an already full buffer.

Slide taken from a presentation by Gadi Taubenfeld from IDC

Monitors - some comments

- Condition variables ***do not accumulate signals*** for later use
- ***wait()*** must come before ***signal()*** in order to be signaled
- N*o race conditions*, because monitors have mutual exclusion
- More complex to implement – but done by compiler
- ***Implementation issues:***
  - ***How to interpret nested monitors?***
  - ***How to define wait, priority scheduling, timeouts, aborts ?***
  - ***How to Handle all exception conditions ?***
  - ***How to interact with process creation and destruction ?***

Implementing Monitors with Semaphores – take 1

```
semaphore mutex=1; /*control access to monitor*/
semaphore c /*represents condition variable c */

void  enter_monitor(void)       {
      down(mutex);  /*only one-at-a-time*/
                                 }
void  leave(void)       {
      up(mutex); /*allow other processes in*/
                        }
void  leave_with_signal(semaphore  c)  /* leave with signaling c*/
{     up(c)  /*release the condition variable, mutex not released */
}

void wait(semaphore c)  /* block on a condition  c */

{   up(mutex); /*allow other processes*/
    down (c);    /*block on the condition variable*/
}
```

Any problem with this code? May deadlock.

Implementing Monitors with Semaphores - Correct

```
Semaphore mutex = 1; /* control access to monitor */
Cond c; /*  c = {count; semaphore}   */
void  enter_monitor(void)          {
      down(mutex); /*  only one-at-a-time */
                                   }
void  leave(void)        {
      up(mutex); /*  allow other processes in */
                         }
void  leave_with_signal(cond  c) {  /*  cond c is a struct */
      if(c.count == 0) up(mutex);     /*  no waiting, just leave.. */
      else {c.count--;
            up(c.s)}
}
void  wait(cond  c)      { /*  block on a condition  */
      c.count++;           /*  count waiting processes */
      up(mutex);           /*  allow other processes  */
      down(c.s);           /*  block on the condition   */
}
```

Outline

- Monitors
  - Monitors in Java
- Barrier synchronization
- The sleeping barber problem
- Readers and writers
- One-way tunnel

Monitors in Java

- Originally, no condition variables (actually, only a single implicit one)
- Procedures are designated as ***synchronized***
- Synchronization operations:
  - ***Wait***
  - ***Notify***
  - ***Notifyall***

Producer-consumer in Java (cont'd)

```
Class ProducerConsumer {
    Producer prod = new Producer();
    Consumer cons = new Consumer();
    BundedBuffer bb = new BoundedBuffer();
    Public static void main(String[] args) {
        prod.start();
        cons.start();
}}
```

Producer-consumer in Java

```
→ Class Producer extends Thread {
→   void run() {
→     while(true) {
→       int item = produceItem();
→       BoundedBuffer.insert(item);
}}}

→ Class Consumer extends Thread {
→   int item
→   void run() {
→     while(true) {
→       item = BoundedBuffer.extract();
→       consume(item);
}}}
```

Producer-consumer in Java (cont'd)

```
Class BoundedBuffer {
    private int[] buffer = new int buff[N];
    int first = 0, last = 0;
    public synchronized void insert(int item) {
        while((last – first) == N)
            wait();
        buff[last % N] = item;
        notify();
        last++; }
```

What is the problem with this code?

```
public synchronized int extract(int item) {
        while(last == first)
            wait();
        int item = buff[first % N];
        first++;
        notify();
        return item;
 }}
```

The problem with the code in previous slide

- Assume a buffer of size 1
- The buffer is empty, consumers 1, 2 enter the monitor and wait
- A producer enters the monitor and fills it, performs a notify and exits. Consumer 1 is ready.
- The producer enters the monitor again and waits.
- Consumer 1 empties the buffer, performs a notify and exits.
- Consumer 2 gets the signal and has to wait again. DEADLOCK.

We must use notifyAll()!

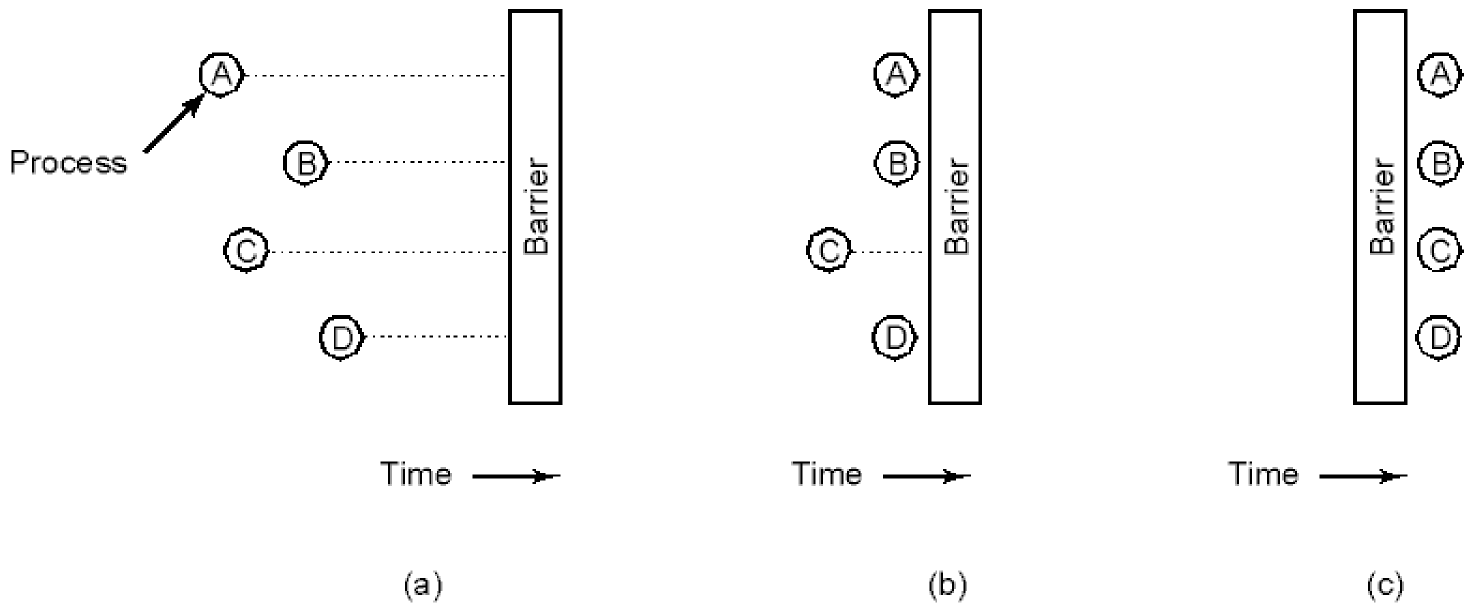
Monitors in Java: comments

- notify() does not have to be the last statement
- wait() adds the calling Thread to the queue of waiting threads
- a Thread performing notify() is not blocked - just moves one waiting Thread to state ready
- once the monitor is open, all queued ready Threads (including former waiting ones) are contesting for entry
- To ensure correctness, wait() operations must be part of a condition-checking loop

Outline

- ❑ Monitors
  - Monitors in Java
- ❑ Barrier synchronization
- ❑ The sleeping barber problem
- ❑ Readers and writers
- ❑ One-way tunnel

Barriers



- ❑ Useful for computations that proceed in phases
- ❑ Use of a barrier:
  - (a) processes approaching a barrier
  - (b) all processes but one blocked at barrier
  - (c) last process arrives, all are let through

The fetch-and-increment instruction

Fetch-and-increment(*w*)
do **atomically**
  prev:=w
  w:=w+1
  return prev

A simple barrier using fetch-and-inc

```
shared integer counter=0
Barrier()
  counter := Fetch-and-increment(counter)
  if (counter = n)
    counter := 0
  else
    await (counter = 0)
```

Will this work ?

T_1: counter set to zero by nth process

T_2: nth process increments it again…

No waiting process has time to check that counter = 0

One shared atomic bit

```
shared integer counter=0, bit go
local local-go, local-counter
Barrier()
   local-go := go
   local-counter := Fetch-and-increment(counter)
   if (local-counter = n)
      counter := 0
      go := 1-go
   else
      await (local-go ≠ go)
```

All waiting processes are released by the atomic bit go !!

A barrier using Binary Semaphores

```
shared   atomic counter=0
         binary semaphore arrival=1 departure=0
Barrier()
  down(arrival)
  counter := counter + 1
  if(counter < n)
    up(arrival)
  else up(departure)
  down(departure)
  counter := counter – 1
  if(counter > 0)
    up(departure)
  else up(arrival)
```

Outline

- Monitors
  - Monitors in Java
- Barrier synchronization
- The sleeping barber problem
- Readers and writers
- One-way tunnel

The Sleeping Barber Problem

The sleeping barber problem (cont'd)

- Barber shop - *one* service provider; *many* customers
- A finite waiting queue
- One customer is served at a time
- Service provider, *barber*, sleeps when no customers are waiting
- Customer *leaves* if shop is full
- Customer *sleeps* while waiting in queue

The sleeping barber: implementation

```
#define   CHAIRS   5
semaphore customers = 0; // number of waiting customers
Semaphore barbers = 0; // number of available barbers: either 0 or 1
int  waiting = 0; // copy of customers for reading
Semaphore mutex = 1;   // mutex for accessing 'waiting'
void  barber(void)   {
     while(TRUE)     {
        down(customers); // block if no customers
        down(mutex); // access to 'waiting'
        waiting = waiting - 1;
        up(barbers); // barber is in..
        up(mutex);  // release 'waiting'
        cut_hair();      }
}
```

The sleeping barber: implementation (cont'd)

```
void  customer(void) {
      down(mutex);    // access to `waiting'
      if(waiting < CHAIRS)         {
        waiting = waiting + 1; // increment waiting
        up(customers); // wake up barber
        up(mutex); // release 'waiting'
        down(barbers); // go to sleep if barbers=0
        get_haircut();
        }
      else    {
        up(mutex);                                   /* shop full  .. leave */
      }}
```

Any problem with this code? Two customers on chair at once

The sleeping barber: correct synchronization

```
#define    CHAIRS    5
semaphore customers = 0; // number of waiting customers
semaphore barbers = 0; // number of available barbers: either 0 or 1
semaphore  mutex = 1;   // mutex for accessing ‘waiting’
semaphore  synch = 0;   // synchronizing the service operation
int  waiting = 0; // copy of customers for reading

void  barber(void)   {
     while(TRUE)      {
        down(customers); // block if no customers
        down(mutex); // access to ‘waiting’
        waiting = waiting - 1;
        up(barbers); // barber is in..
        up(mutex);  // release ‘waiting’
        cut_hair();
        down(synch)   //wait for customer to leave }
}
```

The sleeping barber: correct synchronization *(cont'd)*

```
void  customer(void) {
    down(mutex);    // access to `waiting'
    if(waiting < CHAIRS)         {
      waiting = waiting + 1; // increment waiting
      up(customers); // wake up barber
      up(mutex); // release 'waiting'
      down(barbers); // go to sleep if barbers=0
      get_haircut();
      up(sync); //synchronize service
      }
    else    {
      up(mutex);                                /* shop full  .. leave */
    }}
```

Outline

- Monitors
  - Monitors in Java
- Barrier synchronization
- The sleeping barber problem
- Readers and writers
- One-way tunnel

The readers and writers problem

- Motivation: database access
- Two groups of processes: readers, writers
- Multiple readers may access database simultaneously
- A writing process needs exclusive database access

Readers and Writers: 1st algorithm

```
Int rc = 0 // # of reading processes
semaphore mutex = 1; // controls access to rc
semaphore db = 1; // controls database access
```

```
void reader(void){
  while(TRUE){
    down(mutex);
      rc = rc + 1;
      if(rc == 1)
        down(db);
    up(mutex);
    read_data_base();
    down(mutex);
    rc = rc - 1;
    if(rc == 0)
      up(db);
    up(mutex); }
}
```

```
void writer(void){
  while(TRUE){
    down(db);
      write_data_base()
    up(db)
}
```

Who is more likely to run: readers or writers?

Comments on 1st algorithm

- *No reader is kept waiting*, unless a writer has already obtained the db semaphore
- *Writer processes may starve* - if readers keep coming in and hold the semaphore db
- An alternative version of the readers-writers problem requires that *no writer is kept waiting once it is "ready"* - when a writer is waiting, no new reader can start reading

Readers and Writers: writers' priority

Int rc, wc = 0 // # of reading/writing processes
semaphore Rmutex, Wmutex = 1; // controls readers/writers access to rc/wc
semaphore Rdb, Wdb = 1; // controls readers/writers database access

```
void reader(void){
  while(TRUE){
    down(Rdb);
      down(Rmutex)
        rc = rc + 1;
        if(rc == 1)
          down(Wdb);
      up(Rmutex);
    up(Rdb)
    read_data_base();
    down(Rmutex);
      rc = rc - 1;
      if(rc == 0)
        up(Wdb);
    up(Rmutex); }
}
```

```
void writer(void){
  while(TRUE){
    down(Wmutex);
      wc = wc + 1
      if (wc == 1)
        down (Rdb)
    up(Wmutex)
    down(Wdb)
      write_data_base()
    up(Wdb)
    down(Wmutex)
      wc=wc-1
      if (wc == 0)
        up(Rdb)
    up(Wmutex)
```

Comments on 2nd algorithm

- *When readers are holding Wdb*, the first *writer* to arrive grabs *Rdb*
- *All Readers* arriving later are blocked on *Rdb*
- all *writers* arriving later are blocked on *Wdb*
- only the last *writer* to leave *Wdb* releases *Rdb* – readers can wait…
- If a *writer* and a few *readers* are waiting on *Rdb*, the *writer* may still have to wait for these *readers*. If *Rdb* is unfair, the *writer* may again starve

Readers and Writers: improved writers' priority

Int rc, wc = 0 // # of reading/writing processes
→ semaphore Rmutex, Wmutex, <u>Mutex2</u> = 1;
semaphore Rdb, Wdb = 1;

```
void reader(void){
  while(TRUE){
    down(Mutex2)          <-
      down(Rdb);
        down(Rmutex)
          rc = rc + 1;
          if(rc == 1)
            down(Wdb);
        up(Rmutex);
      up(Rdb)
    up(Mutex2)            <-
    read_data_base();
    down(Rmutex);
    rc = rc - 1;
    if(rc == 0)
      up(Wdb);
    up(Rmutex); }
}
```

```
void writer(void){
  while(TRUE){
    down(Wmutex);
    wc = wc + 1
    if (wc == 1)
      down (Rdb)
    up(Wmutex)
    down(Wdb)
    write_data_base()
    up(Wdb)
    down(Wmutex)
    wc=wc-1
    if (wc == 0)
      up(Rdb)
    up(Wmutex)
```

Improved writers' priority

- After the first writer performs *down(Rdb),* the first reader that enters is blocked after *down(Mutex2)* and before *up(Mutex2)*
- Thus no other readers can block on *Rdb*
- This guarantees that the writer has to wait for at most a single reader
- Irrespective of the fairness of the *Rdb* semaphore’s queue

Readers-writers with Monitors

```
Monitor reader_writer{
    int             numberOfReaders = 0;
    boolean         writing = FALSE;
    condition       okToRead, okToWrite;
public:
    procedure startRead() {
        if(writing || (notEmpty(okToRead.queue)))  okToRead.wait;
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal;
            };
    procedure finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if(numberOfReaders == 0) okToWrite.signal;
            };
```

Readers-writers with Monitors *(cont'd)*

```
procedure startWrite() {
  if((numberOfReaders != 0) || writing) okToWrite.wait;
      writing = TRUE
  };

procedure finishWrite() {
  writing = FALSE;
  if(notEmpty(okToWrite.queue))
      okToWrite.signal
  else
      okToRead.signal;
                  };
}
```

Behavior of Readers & Writers Monitor

- Waiting Writers receive the db from leaving writers
- Or from leaving (last) Readers
- A leaving (last) Reader does not have to worry about signaling the next Reader
- Signal has the standard semantics
- All waiting Readers enter before a waiting Writer, when a Reader enters

Readers-writers with Monitors (counting)

```
Monitor reader_writer{
    boolean         writing = FALSE;
    condition       okToRead, okToWrite;
    int             numberOfReaders = 0, waitingWrite=0
public:
    procedure startRead() {
            if(writing|| (waitingWrite>0))
                okToRead.wait;
            numberOfReaders = numberOfReaders + 1;
            okToRead.signal;
                };
    procedure finishRead() {
            numberOfReaders = numberOfReaders - 1;
            if(numberOfReaders == 0) okToWrite.signal;
                };
```

Readers-writers with Monitors (counting)

```
procedure startWrite() {
  if((numberOfReaders != 0) || writing)
        waitingWrite++;
        okToWrite.wait;
        waitingWrite--;
   writing = TRUE
  };

procedure finishWrite() {
  writing = FALSE;
  if(waitingWrite>0))
      okToWrite.signal
  else
      okToRead.signal;
                    };
}
```

Monitor keeps writers' priority

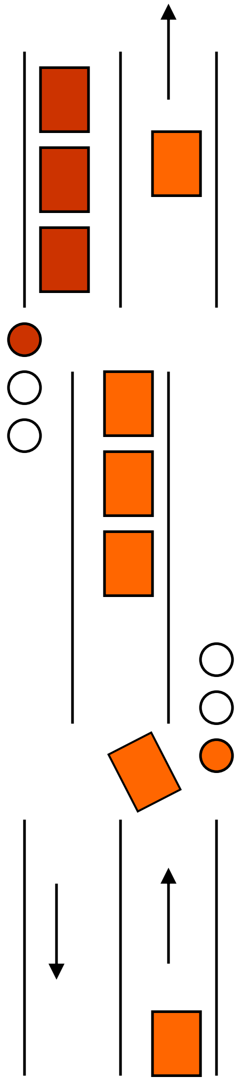
- *When there are waiting Writers*, one of them will have a chance to enter before any *new* Readers
- First line of *startRead()*
- *After the exit of a (last) Writer, all waiting Readers can enter before the next Writer can enter*
- This is guaranteed in the last line of *startRead()* – each entering Reader opens the door to the next one

Outline

- ❑ Monitors
  - Monitors in Java
- ❑ Barrier synchronization
- ❑ The sleeping barber problem
- ❑ Readers and writers
- ❑ One-way tunnel

The one-way tunnel problem

- One-way tunnel
- Allows any number of processes in the same direction
- If there is traffic in the opposite direction – have to wait
- A special case of readers/writers



One-way tunnel - solution

```
int count[2];
Semaphore mutex = 1, busy = 1;
Semaphore waiting[2] = {1,1};
```

```
void arrive(int   direction) {
      down(waiting[direction]);
        down(mutex);
        count[direction] += 1;
        if(count[direction] == 1)
          up(mutex);
          down(busy)
        else
         up(mutex);
      up(waiting[direction]);
}
```

```
void leave(int   direction) {
      down(mutex);
        count[direction] -= 1;
        if(count[direction] == 0)
              up(busy)}
      up(mutex);
}
```