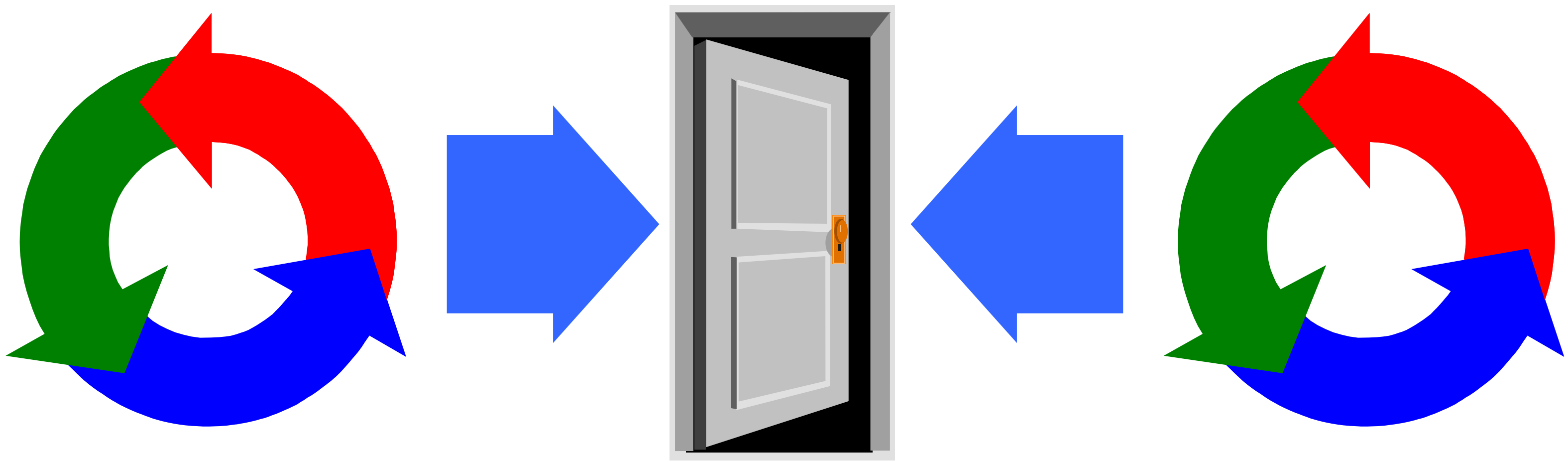
Lecture 7

- Administration

Chapter 5

Monitors & Condition Synchronization



monitors & condition synchronization

Concepts: monitors:

encapsulated data + access procedures
mutual exclusion + condition synchronization
single access procedure active in the monitor
nested monitors

Models: guarded actions

Practice: private data and synchronized methods (exclusion).
wait(), notify() and notifyAll() for condition synch.
single thread active in the monitor at a time

http://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

Example of Notify()

P1 locks and puts and unlocks
P2 locks, tries to put, waits&unlock
P3 tries to put, waits&unlock
C1 locks
C2 blocked
C3 blocked
C1 calls notify(), unlocks
P2 is awakened, tries to put
C2 locks, tries to get, waits&unlock
C3 locks, tries to get, waits&unlock
P2 locks, and puts, calls notify(), unlocks
P3 is awakened, gets lock, waits again

There is no one to call notify() to wake up P3 or C2 or C3

Java Monitors

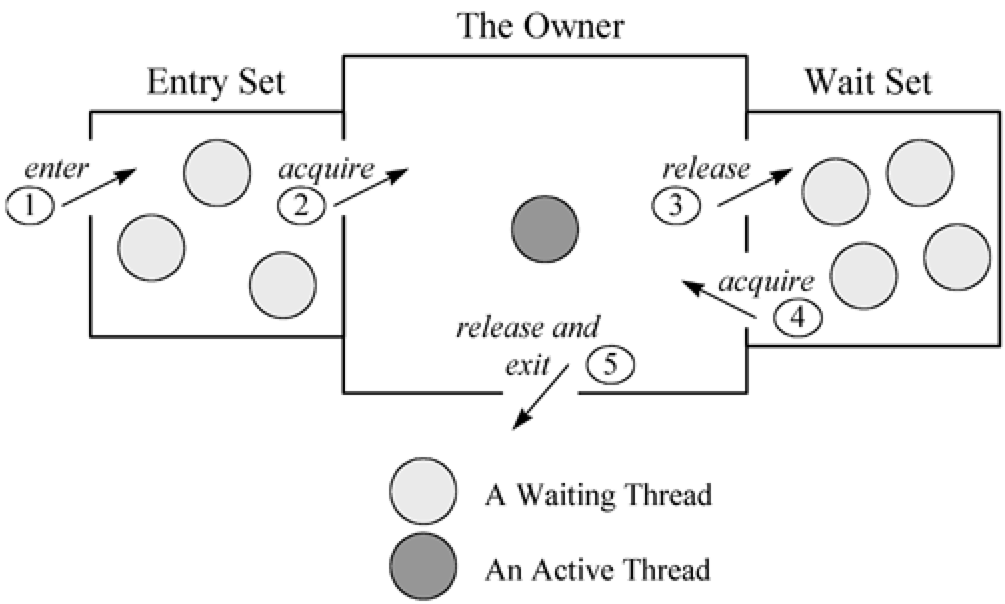
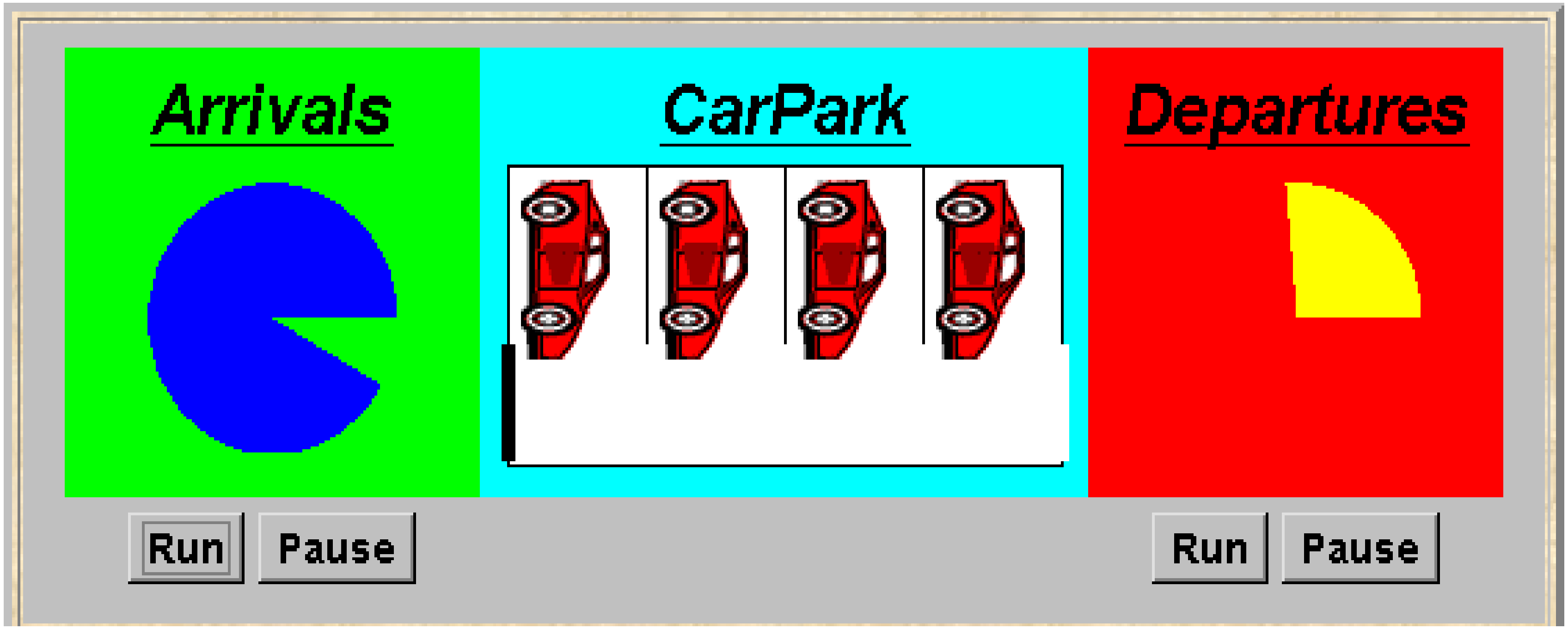


Figure 20-1. A Java monitor.

5.1 Condition synchronization



A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark. Car arrival and departure are simulated by separate threads.

carpark model

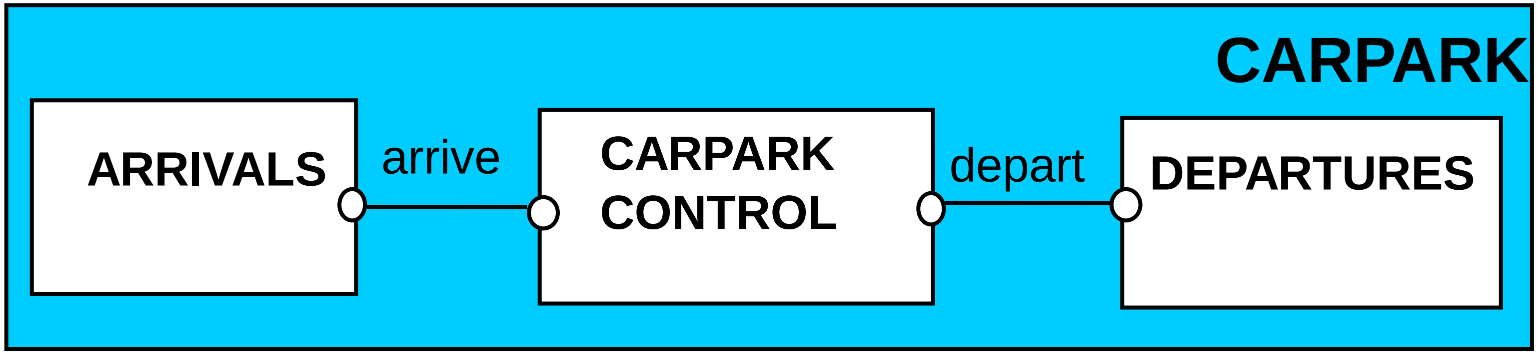
- Events or actions of interest?

  arrive and depart

- Identify processes.

  arrivals, departures and carpark control

- Define each process and interactions (structure).



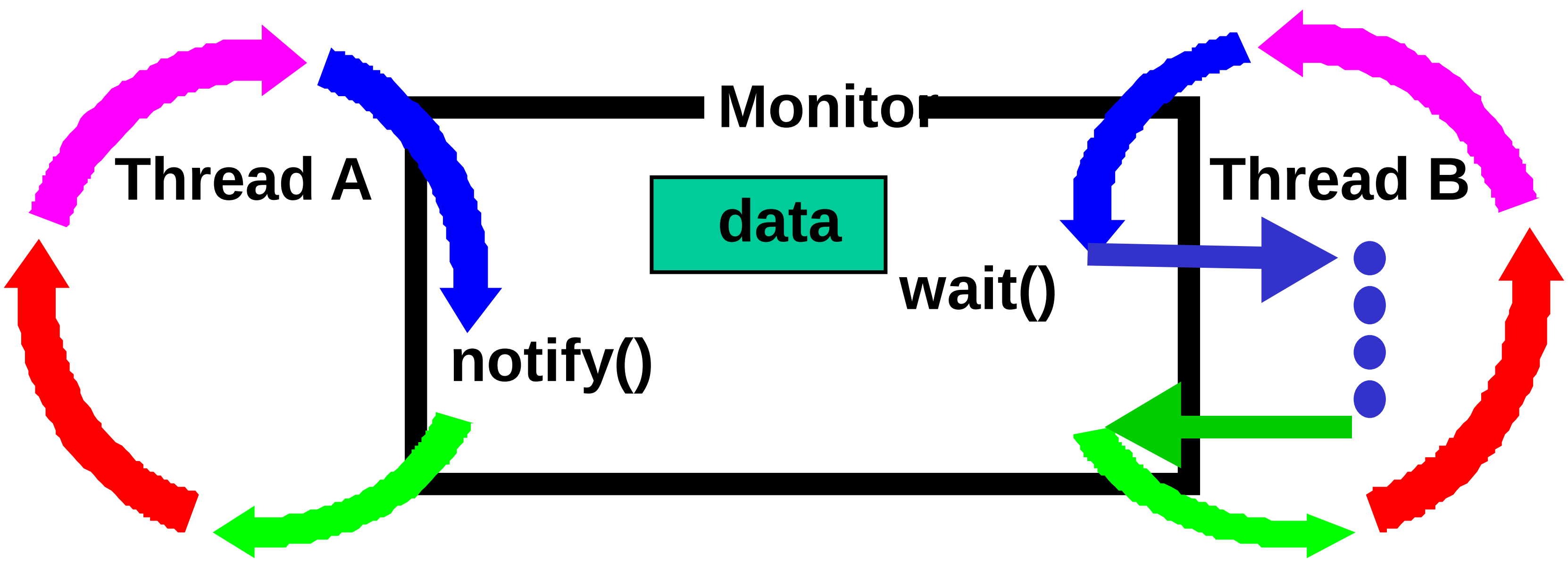
carpark model

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK =
      (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

condition synchronization in Java

We refer to a thread ***entering*** a monitor when it acquires the mutual exclusion lock associated with the monitor and ***exiting*** the monitor when it releases the lock.

Wait() - causes the thread to exit the monitor,
permitting other threads to enter the monitor.



Java Monitors

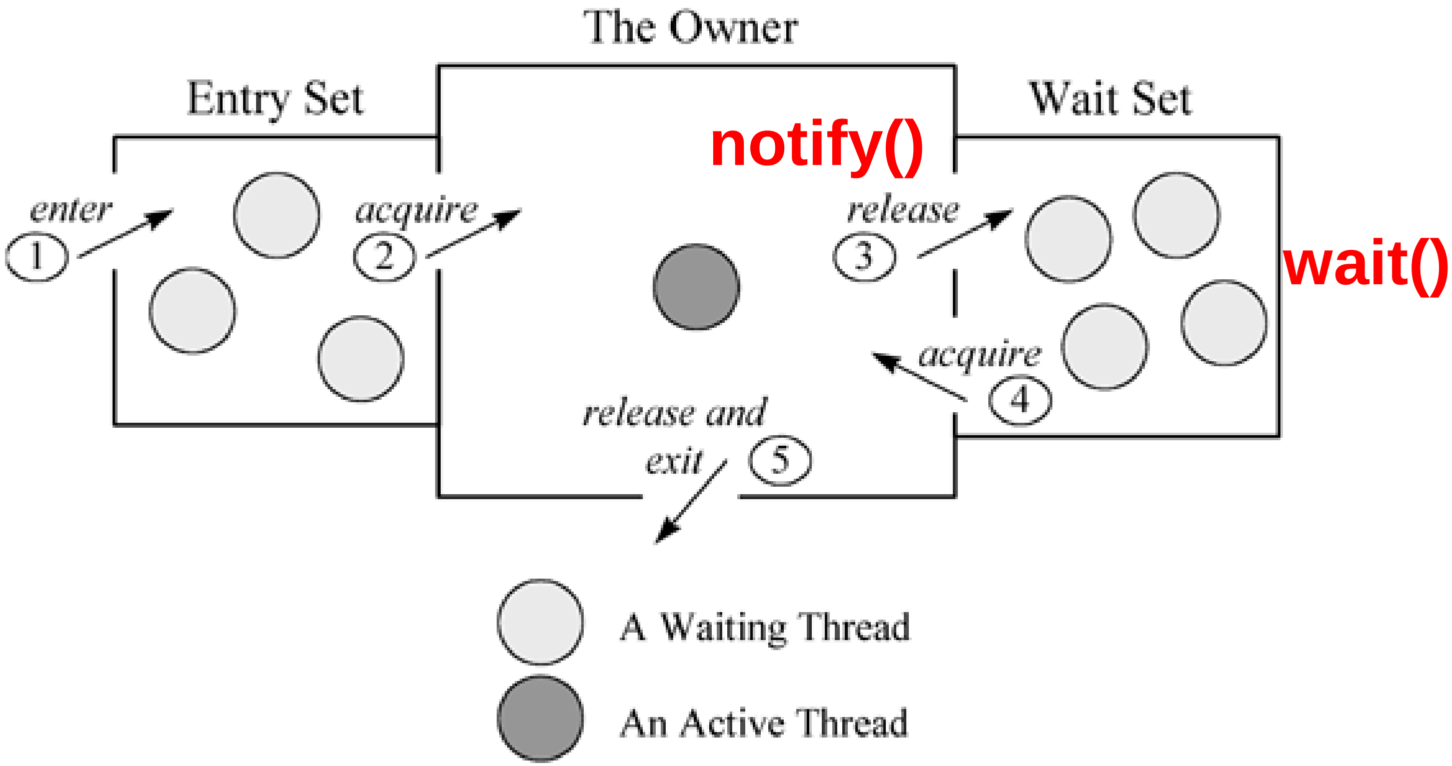


Figure 20-1. A Java monitor.

condition synchronization in Java

```
FSP:    when cond act -> NEWSTAT
```

```
Java:  public synchronized void act()
                throws InterruptedException
       {
        while (!cond) wait();
          // modify monitor data
        notifyAll()
       }
```

The **while** loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when it re-enters the monitor.

notifyall() is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

Semaphores

Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore *s* is an integer variable that can take only non-negative values.

The only operations permitted on *s* are *up(s)* and *down(s)*. Blocked processes are held in a FIFO queue.

```
down(s): if s >0 then
            decrement s
         else
            block execution of the
         calling process

up(s):      if processes blocked on s
            then
                  awaken one of them
            else
                  increment s
```

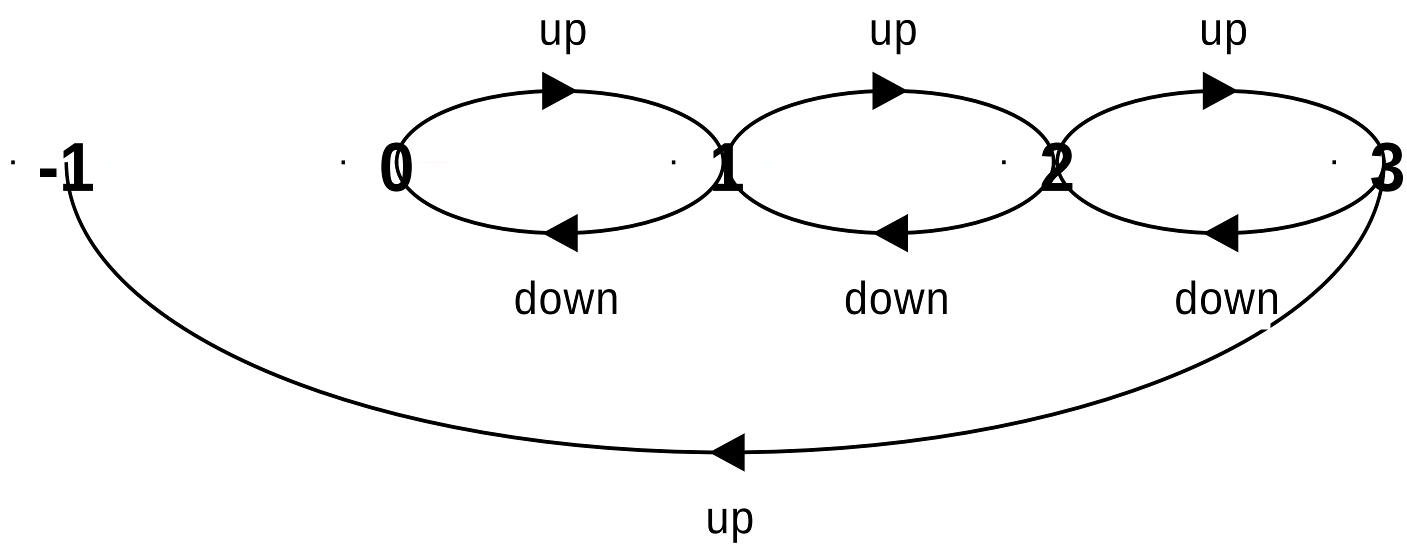
modeling semaphores

To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. N is the initial value.

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]
                        ),
SEMA[Max+1]    = ERROR.
```

LTS?

modeling semaphores



Action down is only accepted when value v of the semaphore is greater than 0.

Action up is not guarded.

Trace to a violation:

up → up → up → up

semaphore demo - model

Three processes **p[1..3]** use a shared semaphore **mutex** to ensure mutually exclusive access (action **critical**) to some resource.

```
LOOP = (
     mutex.down->critical->mutex.up->LOOP
     ).

||SEMADEMO = (p[1..3]:LOOP
             ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

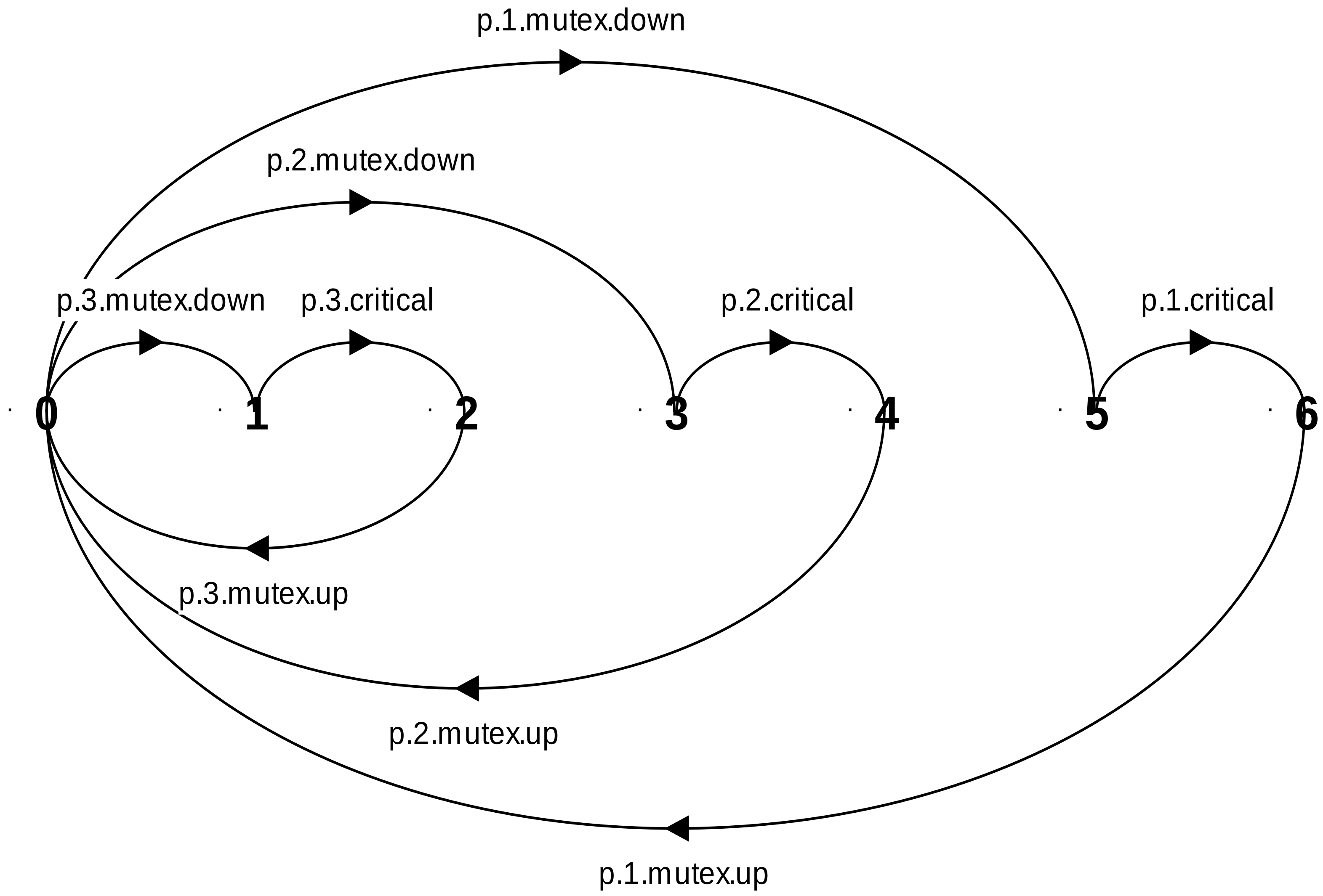
For mutual exclusion, the semaphore initial value is 1. ***Why?***

Is the **ERROR** *state reachable for* **SEMADEMO***?*

Is a binary semaphore sufficient (i.e. **Max=1***) ?*

LTS?

semaphore demo - model



semaphores in Java

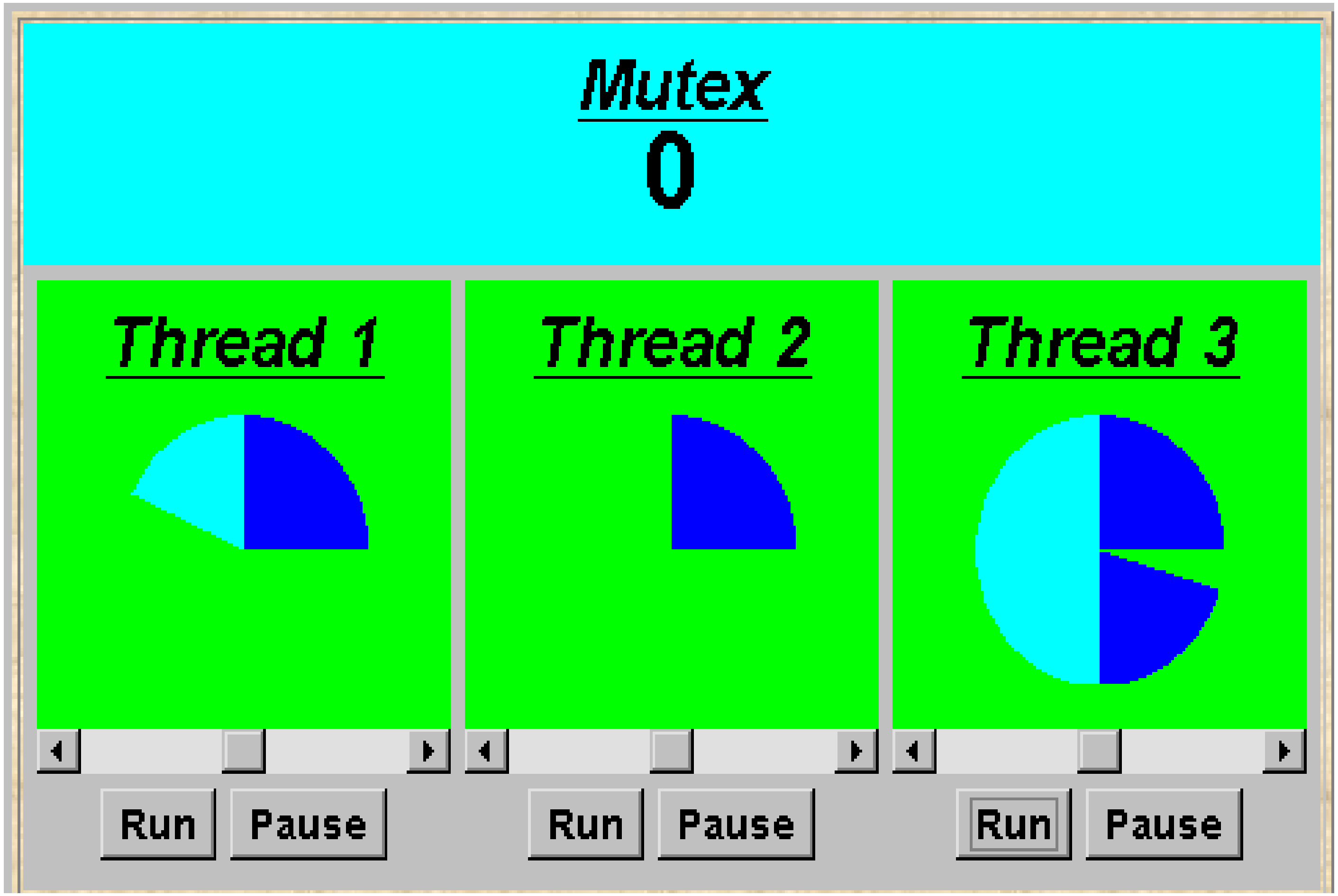
Semaphores are passive objects, therefore implemented as **monitors**.

(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  synchronized public void up() {
     ++value;
     notifyAll();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value== 0) wait();
    --value;
  }
}
```

Is it safe to use `notify()` *here rather than* `notifyAll()`*?*

SEMADEMO display



current semaphore value

thread 1 is executing critical actions.

thread 2 is blocked waiting.

thread 3 is executing non-critical actions.

SEMADEMO

What if we adjust the time that each thread spends in its ***critical section*** *?*

- large resource requirement - *more conflict?*

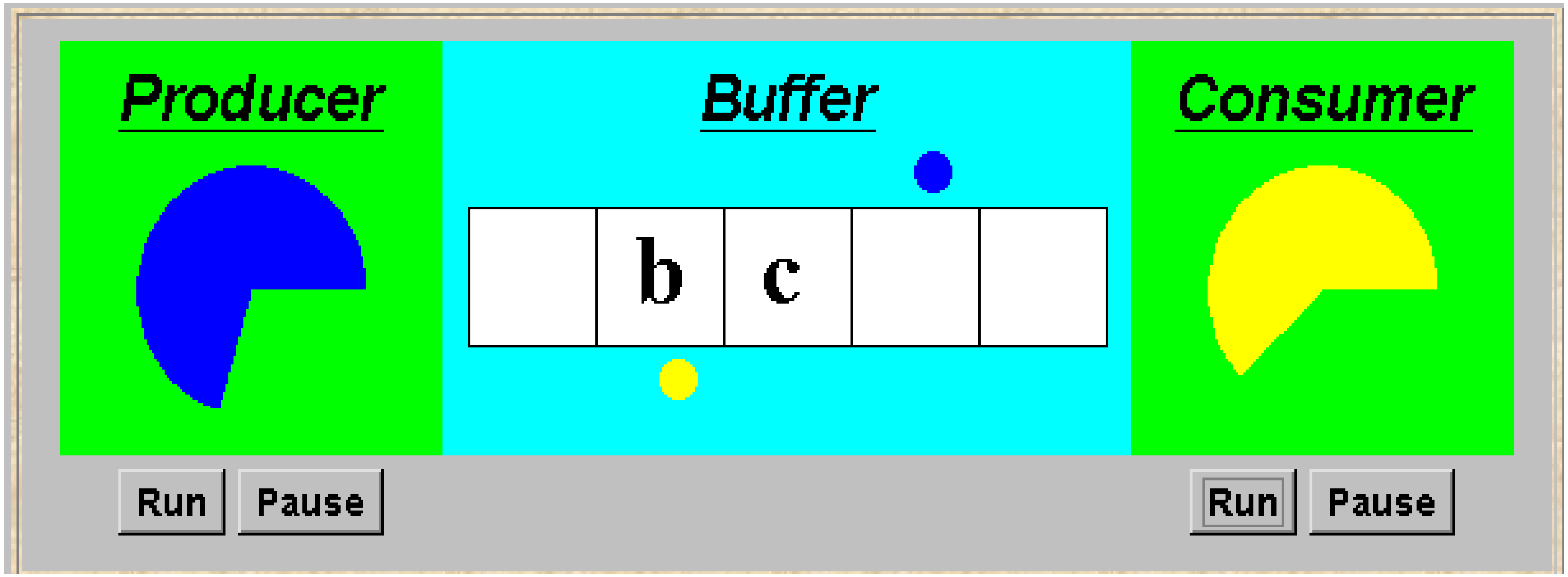
  (eg. more than 67% of a rotation)?

- small resource requirement - *no conflict?*

  (eg. less than 33% of a rotation)?

Hence the time a thread spends in its critical section should be kept as short as possible.

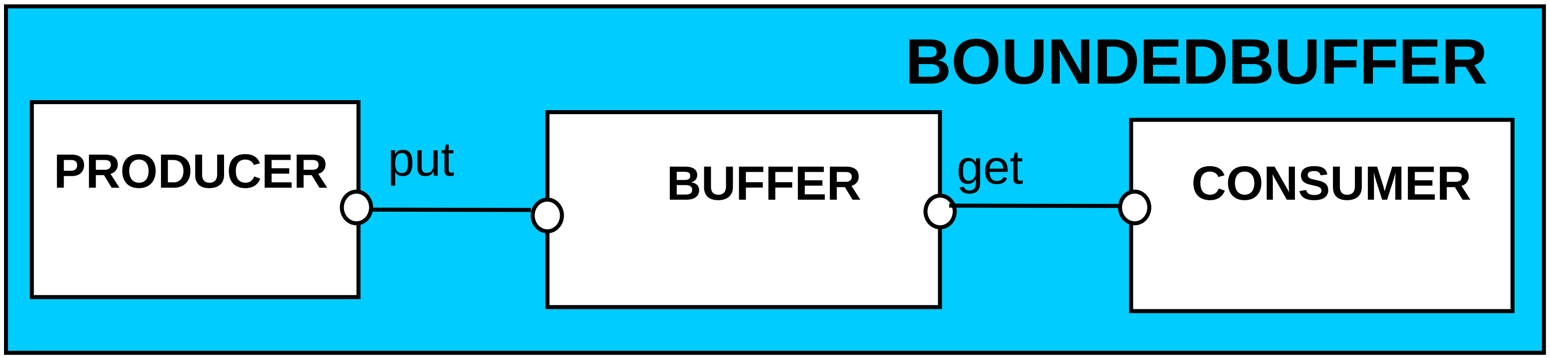
5.3 Bounded Buffer



A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a *producer* process and removed by a *consumer* process. It can be used to smooth out transfer rates between the *producer* and *consumer.*

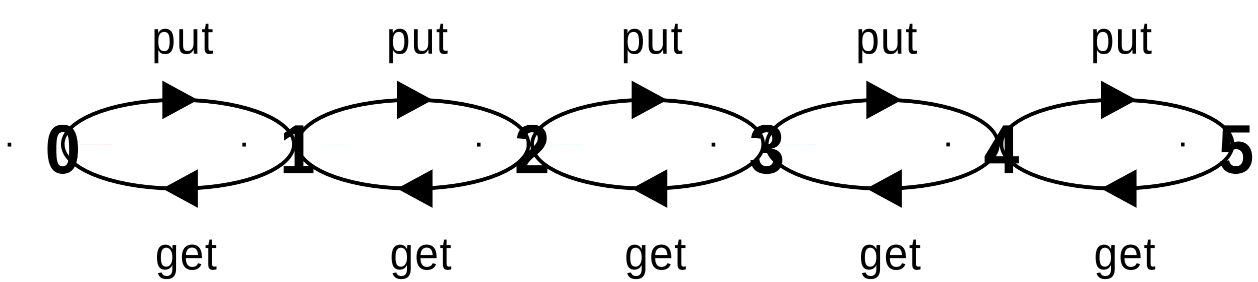
(see car park example)

bounded buffer



The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner.

LTS:



Data-independent model

bounded buffer

```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
      = (when (i<N) put->COUNT[i+1]
       |when (i>0) get->COUNT[i-1]
       ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER = (PRODUCER||BUFFER(5)||CONSUMER).
```