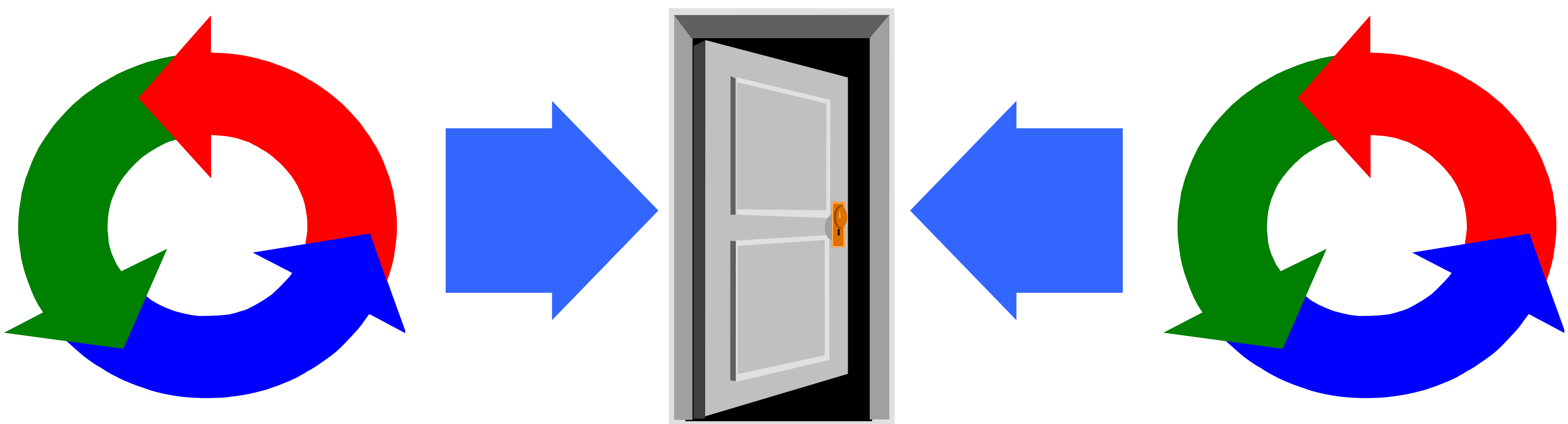
## Chapter 5

# Monitors & Condition Synchronization



# Monitors & Condition Synchronization

**Concepts**: monitors:

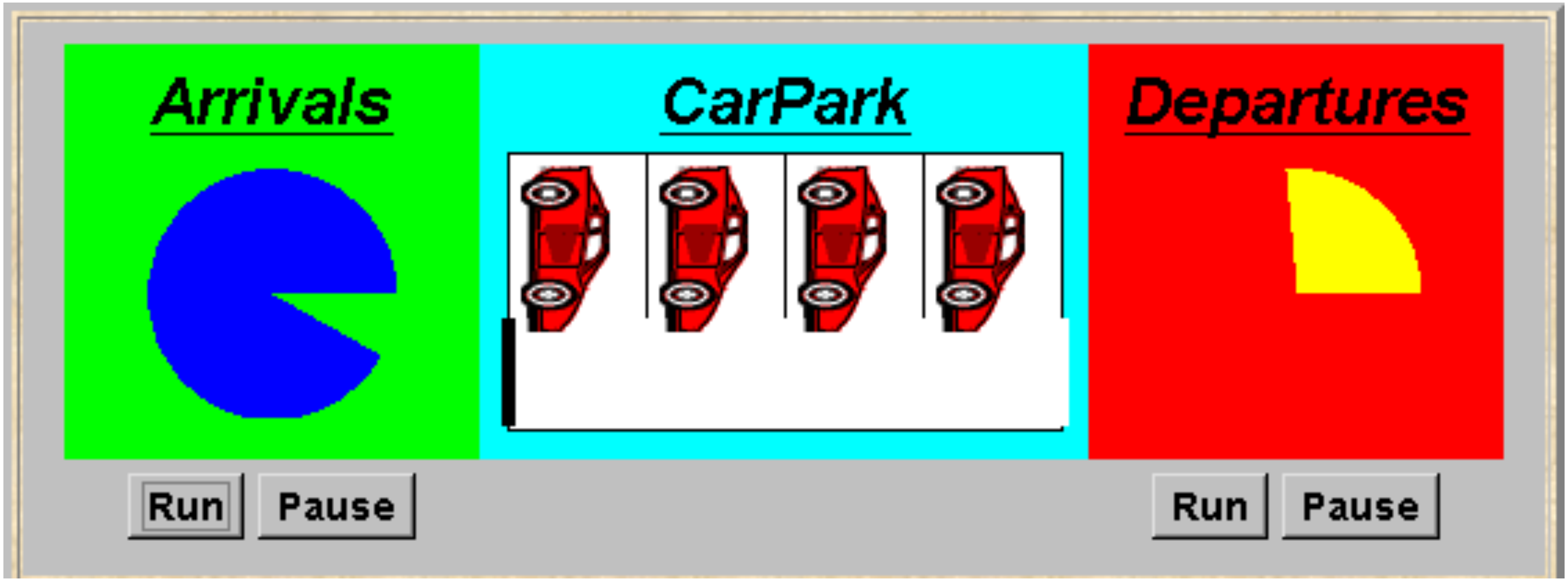
encapsulated data + access procedures
mutual exclusion + condition synchronization
single access procedure active in the monitor

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

# 5.1 Condition Synchronization




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

## Carpark Model

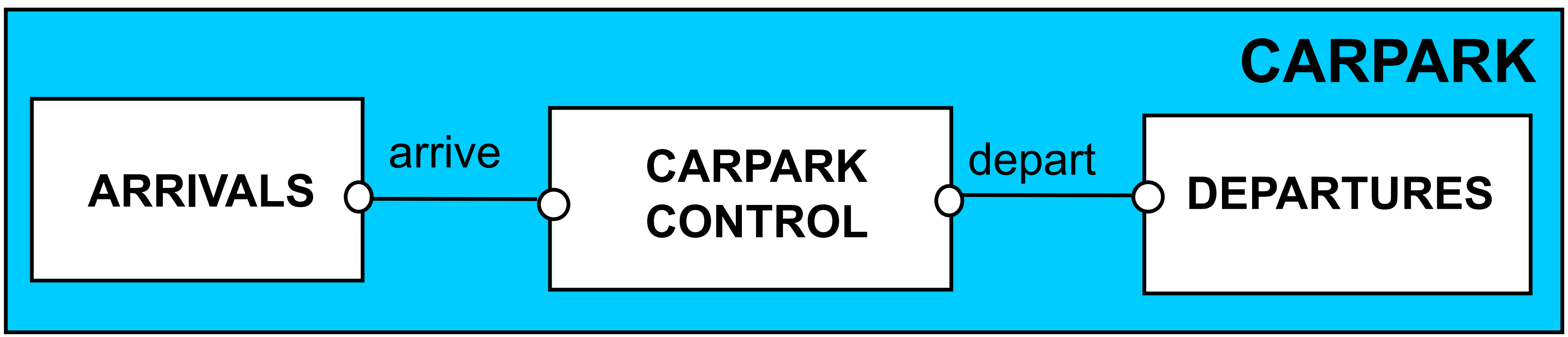
- Events or actions of interest?

  arrive and depart

- Identify processes.

  arrivals, departures and carpark control

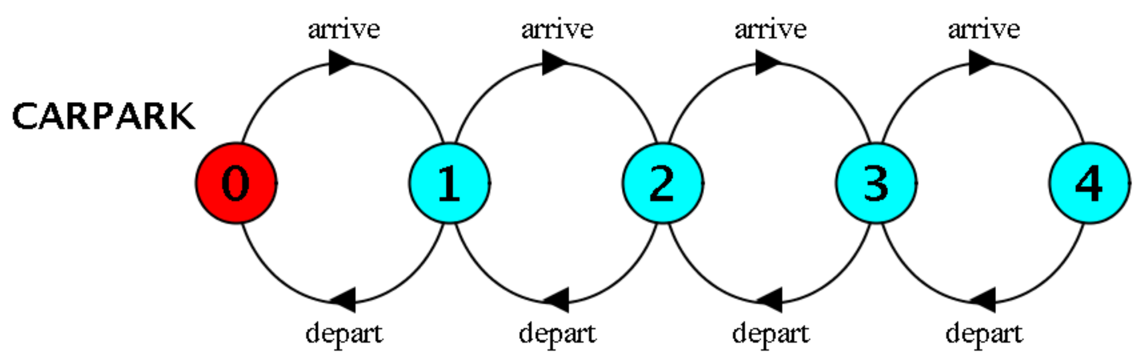
- Define each process and interactions (structure).



# Carpark Model (FSP)

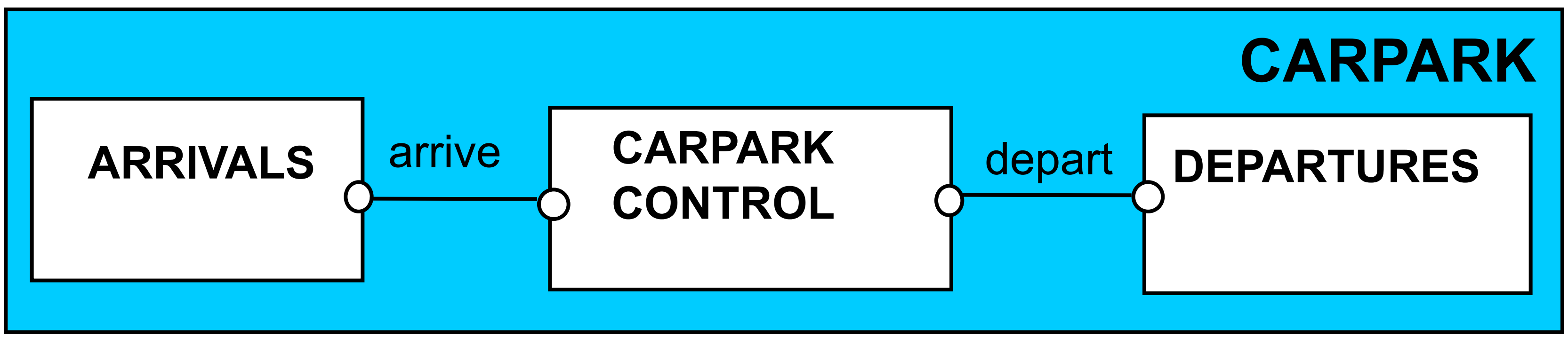
```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

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

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

Guarded actions are used to control **arrive** and **depart**.

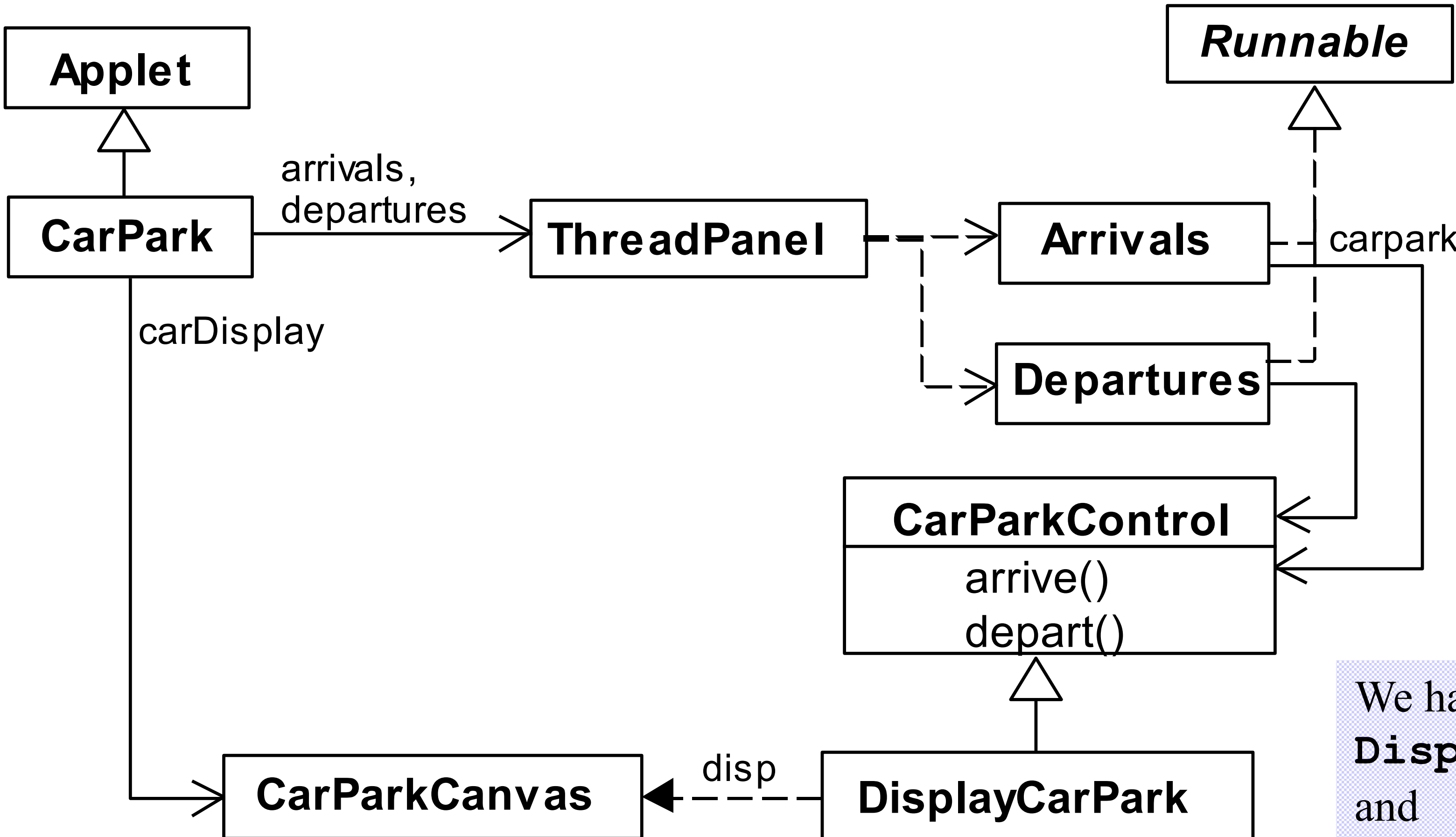


## Carpark Program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and monitors
  - **thread** - **active** entity which **initiates actions**—arrive, depart
  - **monitor** - **passive** entity which **responds** to **actions**.



# Carpark Program – Class Diagram




We have omitted **DisplayThread** and **GraphicCanvas** threads managed by **ThreadPanel**.

# Carpark Program

- `Arrivals` and `Departures` implement `Runnable`
- `CarParkControl` provides the **control--condition synchronization**.
- Instances of these are created by the `start()` method of the `CarPark` applet :

```
public void start() {
  CarParkControl c =
      new DisplayCarPark(carDisplay,Places);
      arrivals.start(new Arrivals(c));
      departures.start(new Departures(c));
}
```

# Carpark Program - Arrivals and Departures threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;

  Arrivals(CarParkControl c) {carpark = c;}

  public void run() {
    try {
      while(true) {
        ThreadPanel.rotate(330);
        carpark.arrive();
        ThreadPanel.rotate(30);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly **Departures** which calls `carpark.depart()`.

# Carpark Program – `CarParkControl` Monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
     …  --spaces; …
     }

  synchronized void depart() {
     … ++spaces; …
     } }
```

*mutual exclusion by synch methods*

*(condition) synchronization?*

*block if full? (spaces==0)*

*block if empty? (spaces==N)*

Monitor:

- a synchronization construct allowing threads to have both
  - mutual exclusion
  - ability to wait for a certain condition to become true
- consists of a **mutex** (lock) object and **condition variables**.

# Condition Synchronization in Java

Java provides a thread **wait set** per monitor (actually per object) with the following methods:

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

Waits to be notified by another thread. The **waiting thread releases** the synchronization **lock** associated with the **monitor**. When notified, the thread must wait to **reacquire** the monitor before resuming execution.

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

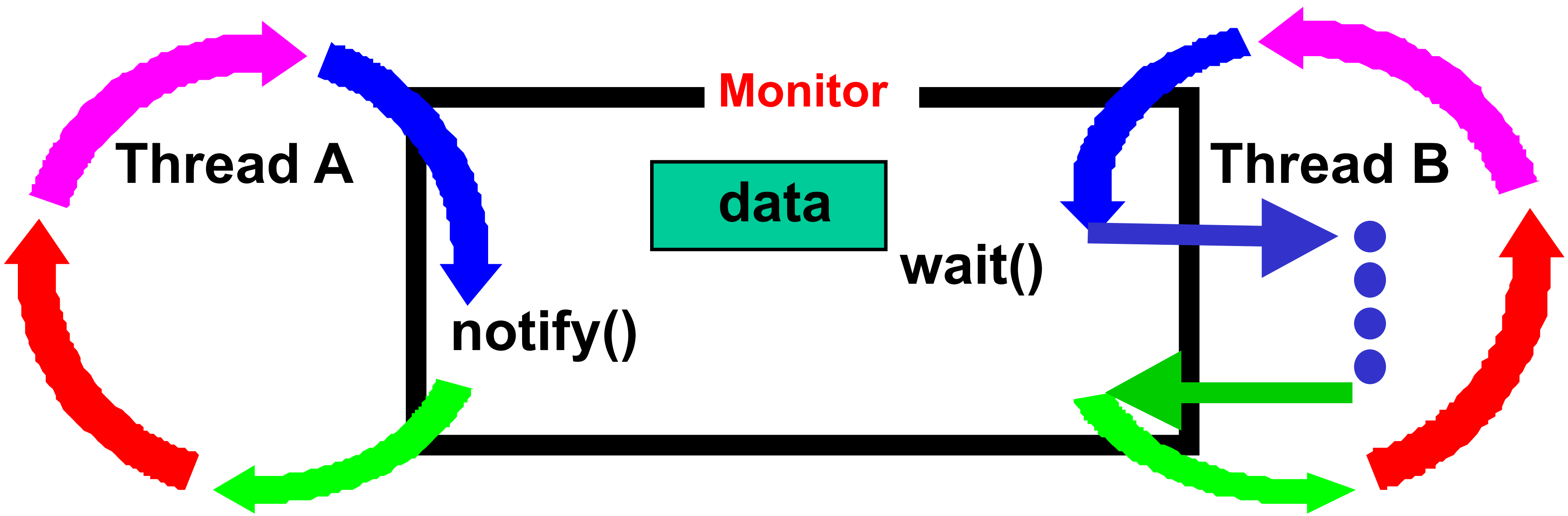
Wakes up a single thread that is waiting on this object's wait set.

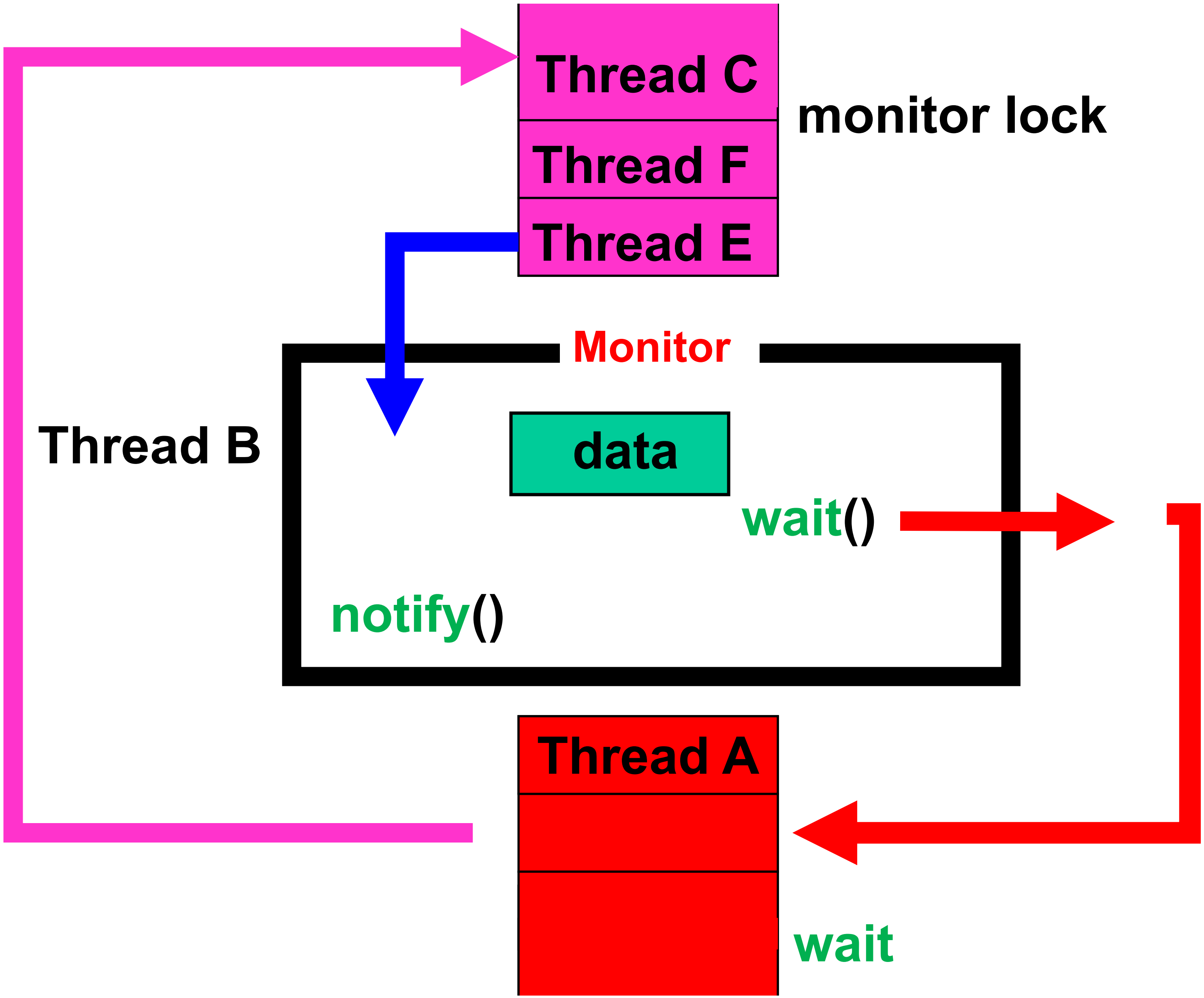
```
public final void notifyAll()
```

Wakes up all threads that are waiting on this object's wait set.

# Condition Synchronization in Java

- ***Entering*** a monitor: when a thread acquires the mutual exclusion **lock** associated with the monitor
- ***Exiting*** the monitor: when it releases the **lock**.
- **Wait()** - causes the thread to exit the monitor, permitting other threads to enter the monitor.




Thread C
monitor lock
Thread F
Thread E
Monitor
Thread B
data
wait()
notify()
Thread A
wait

## 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**.

# CarParkControl – Condition Synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notifyAll();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notifyAll();
  }
}
```

Note: **notify**() wakes up the first thread that called wait() on the same object.
**notifyAll**() wakes up all the threads that called wait() on the same object.

# Models of Monitors - Summary

**Active entities** (that initiate actions) are implemented as **threads**.
**Passive entities** (that respond to actions) are implemented as **monitors**.

Each **guarded** action in the model of a monitor is implemented as a **`synchronized`** method which uses a **while loop** and **`wait()`** to implement the guard.

Changes in the **state** of the monitor are signalled to waiting threads using **`notify()`** or **`notifyAll()`**.

## 5.2 Semaphores

- Semaphores: used for controlling access to a common resource by multiple processes in a concurrent system.
- Semaphore s: non-negative, integer variable.
- 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

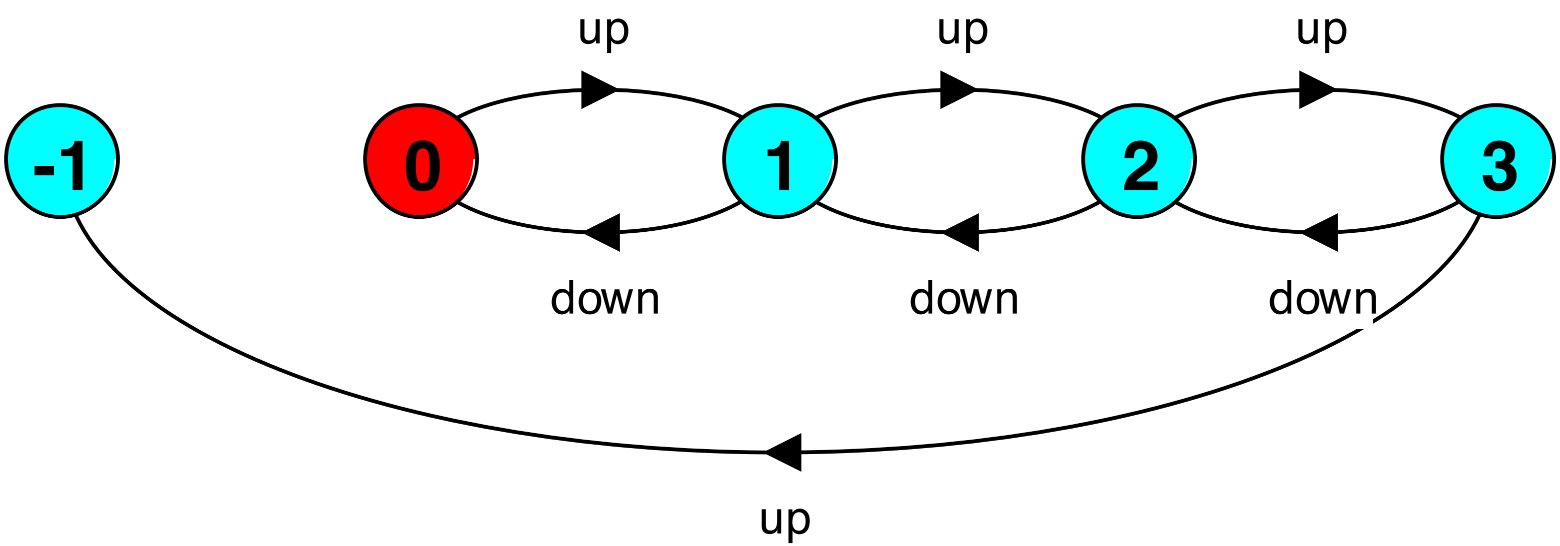
***up(s)*: if** processes are blocked on ***s*** **then**
awaken one of them
increment ***s***

## Modelling Semaphores

- Semaphores take a **finite** range of values. If the range is exceeded then ERROR.
- N with initial value = 0.

```
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.
```

## Modelling 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)).
```

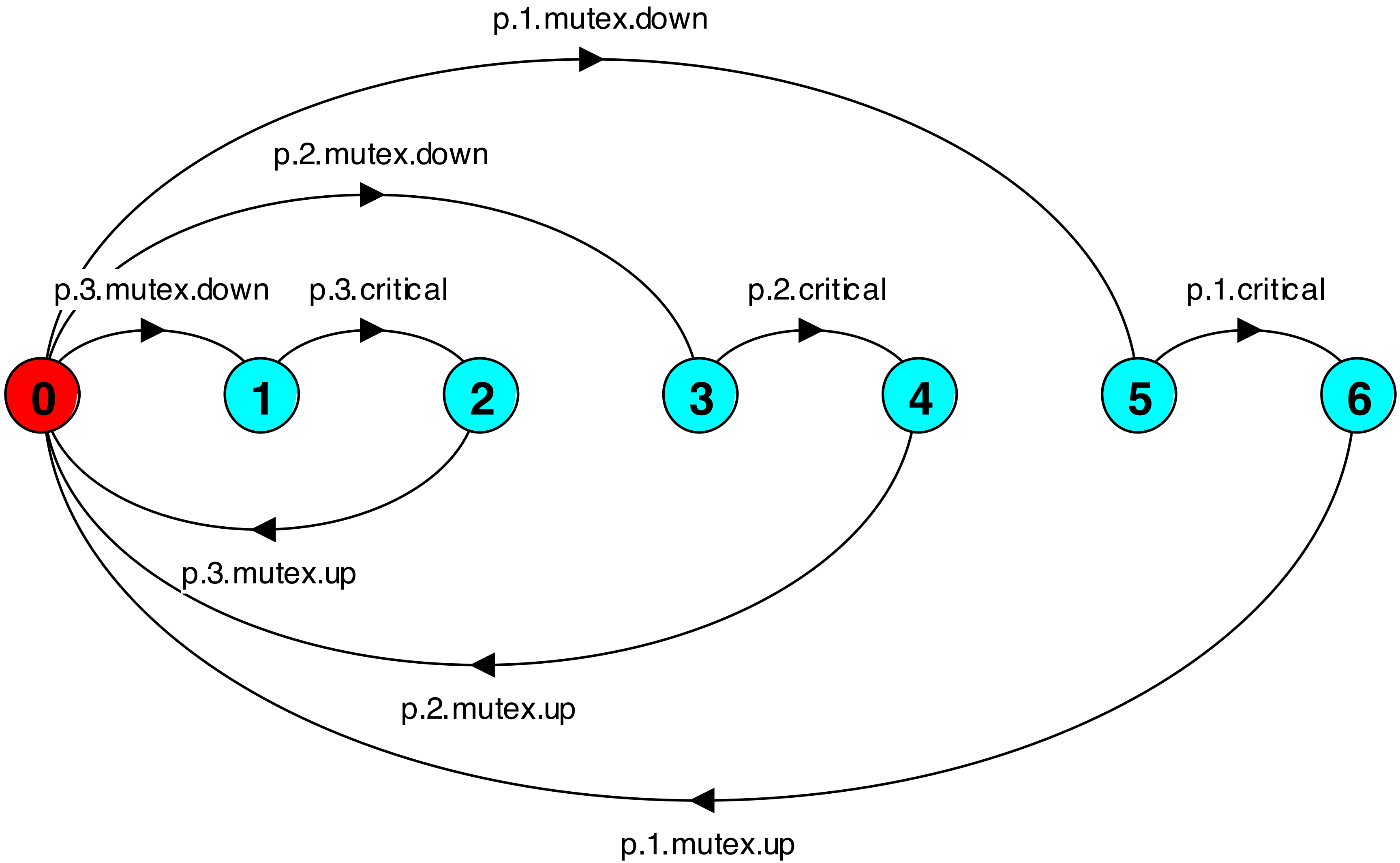
For mutual exclusion, the semaphore initial value is 1. ***Why?***

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

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

Note: semaphore can be counted, while mutex can only count to 1.

# Semaphore Demo - Model



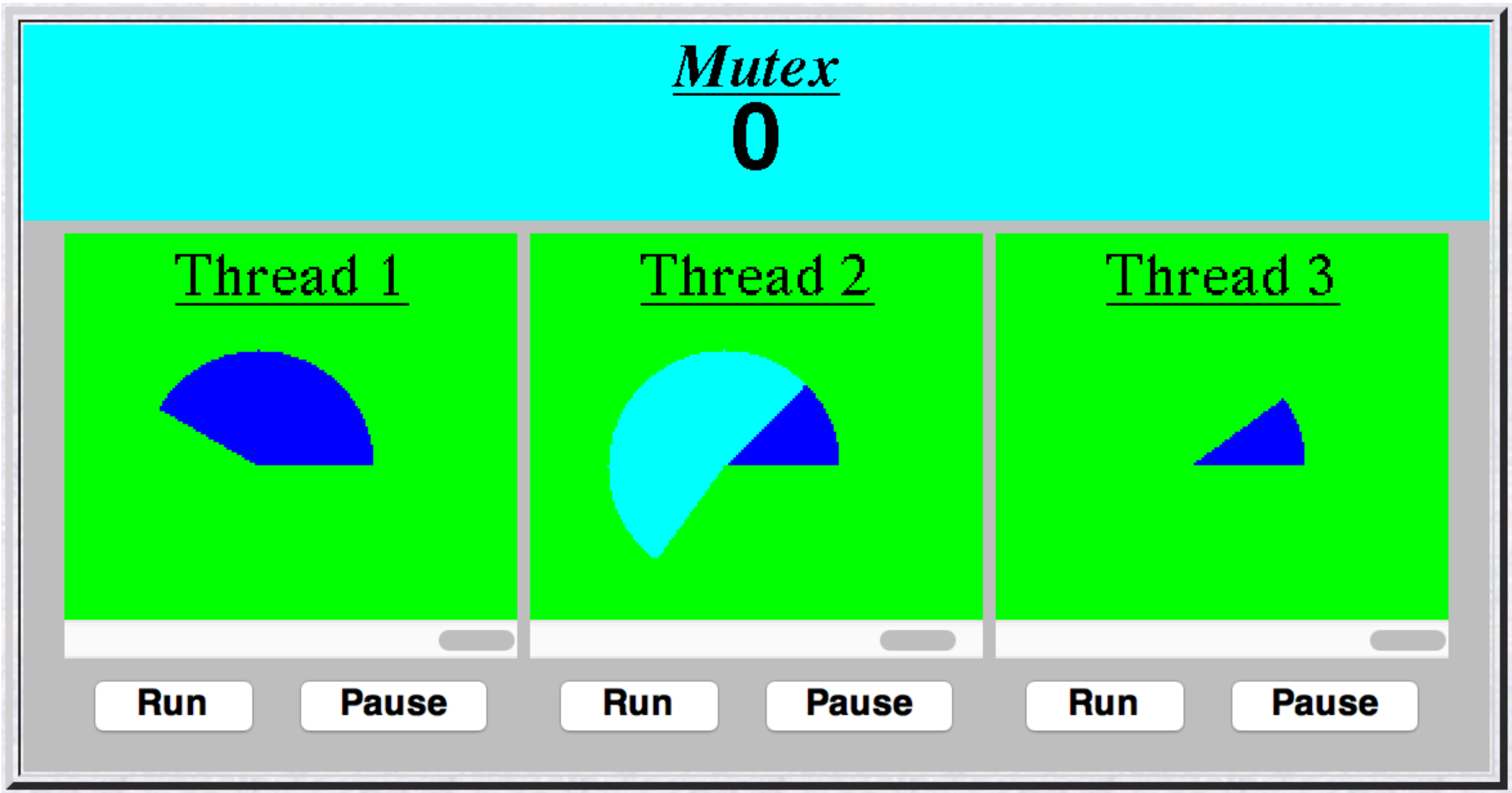
# Semaphores in Java

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

```
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 2 is executing critical actions.

Threads 1, 3 are blocked waiting.

## SEMADEMO

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

- large resource requirement - *more conflict*?

  (e.g., more than 67% of a rotation)?

- small resource requirement - *no conflict*?

  (e.g., less than 33% of a rotation)?

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

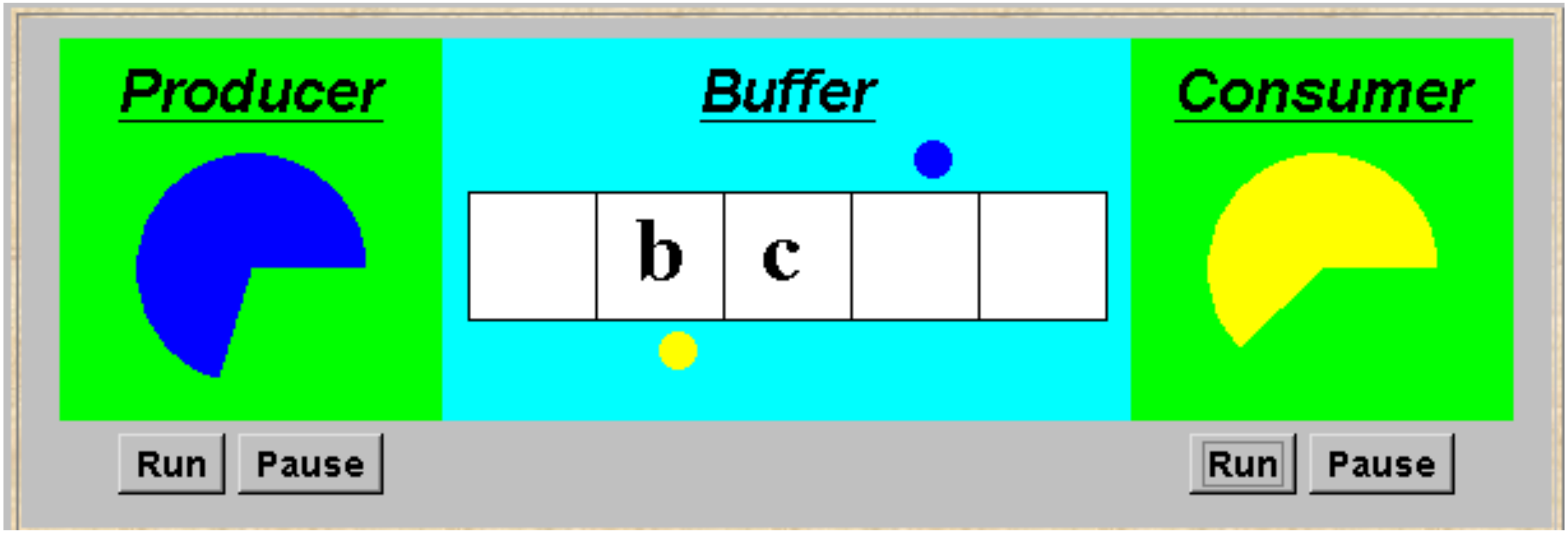
# SEMADEMO program - MutexLoop

```
class MutexLoop implements Runnable {
  Semaphore mutex;

  MutexLoop (Semaphore sema) {mutex=sema;}

  public void run() {
    try {
      while(true)  {
        while(!ThreadPanel.rotate());
        mutex.down();               // get mutual exclusion
        while(ThreadPanel.rotate()); // critical actions
        mutex.up();                 // release mutual exclusion
      }
    } catch(InterruptedException e){}
  }
}
```

Threads and semaphore are created by the applet `start()` method.

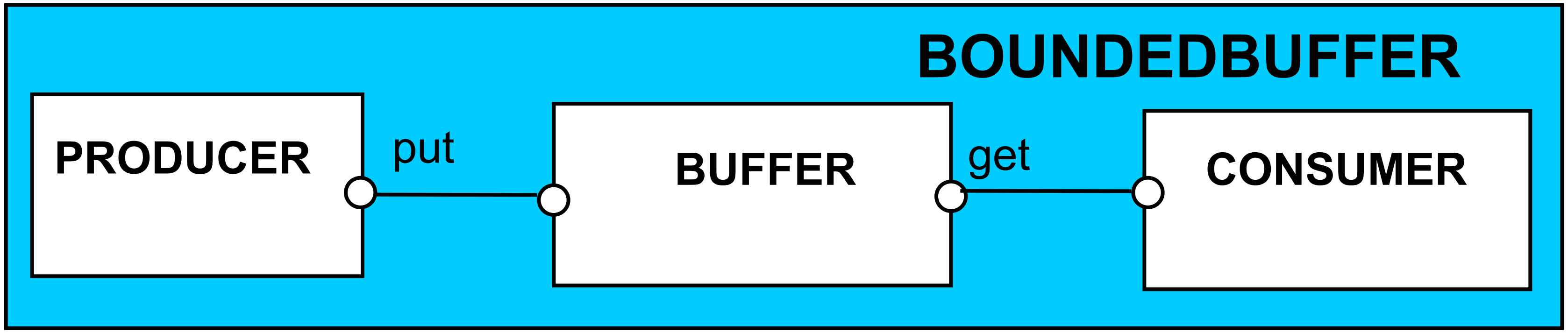
`ThreadPanel.rotate()` returns `false` while executing non-critical actions (blue color) and `true` otherwise.

# 5.3 Bounded Buffer



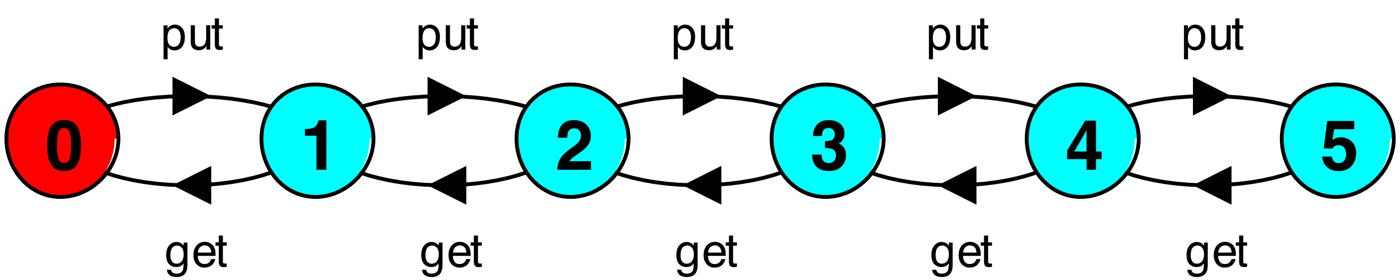

- A bounded buffer consists of a fixed number of slots.
- Items are put into the buffer by a *producer* process and removed by a *consumer* process.

# Bounded Buffer - a Data-Independent Model




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

LTS:



# Bounded Buffer - a Data-Independent Model

```
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).
```

# Bounded Buffer Program – Buffer Monitor

```
public interface Buffer <E> {…}

class BufferImpl <E> implements Buffer <E> {
     …
 public synchronized void put(E o)
            throws InterruptedException {
    while (count == maxSize) wait();
    buf[in] = o; ++count; in=(in+1)%maxSize;
    notifyAll();
  }
  public synchronized E get()
            throws InterruptedException {
    while (count == 0) wait();
    E o = buf[out];
    buf[out]=null; --count; out=(out+1)%maxSize;
    notifyAll();
    return (o);
    }
}
```

# Bounded Buffer Program – Producer Process

```
class Producer implements Runnable {
  Buffer buf;
  String alphabet= "abcdefghijklmnopqrstuvwxyz";

  Producer(Buffer b) {buf = b;}

  public void run() {
    try {
      int ai = 0;
      while(true) {
        ThreadPanel.rotate(12);
        buf.put(alphabet.charAt(ai));
        ai=(ai+1) % alphabet.length();
        ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }
}
```

`Producer` calls `buf.put()`.

`Consumer` calls `buf.get()`.

## 5.4 Nested Monitors

Instead of using ***count*** variable and ***condition synchronization*** directly, use two semaphores ***full*** and ***empty*** to reflect the **state** of the buffer.

```
class SemaBuffer <E> implements Buffer <E> {
  …

  Semaphore full;    // the number of items in Buffer
  Semaphore empty;  // the number of available spaces in Buffer

  SemaBuffer(int size) {
    this.size = size; buf =(E[])new Object[size];
    full = new Semaphore(0);
    empty= new Semaphore(maxSize); // buffer maxSize
  }
…
}
```

# Bounded Buffer Program – Buffer Monitor

```
public interface Buffer <E> {…}

class BufferImpl <E> implements Buffer <E> {
     …
 public synchronized void put(E o)
             throws InterruptedException {
    while (count == maxSize) wait();
    buf[in] = o; ++count; in=(in+1)%maxSize;
    notifyAll();
  }
  public synchronized E get()
             throws InterruptedException {
    while (count == 0) wait();
    E o = buf[out];
    buf[out]=null; --count; out=(out+1)%maxSize;
    notifyAll();
    return (o);
    }
}
```

***condition variable: count***

# Nested Monitors - Bounded Buffer Program

```
synchronized public void put(E o)
              throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)% maxSize;
   full.up();
 }

 synchronized public E get()
              throws InterruptedException{
   full.down();
   E o =buf[out]; buf[out]=null;
   --count; out=(out+1)% maxSize;
   empty.up();
   return (o);
 }
```

***Does this behave as desired?***

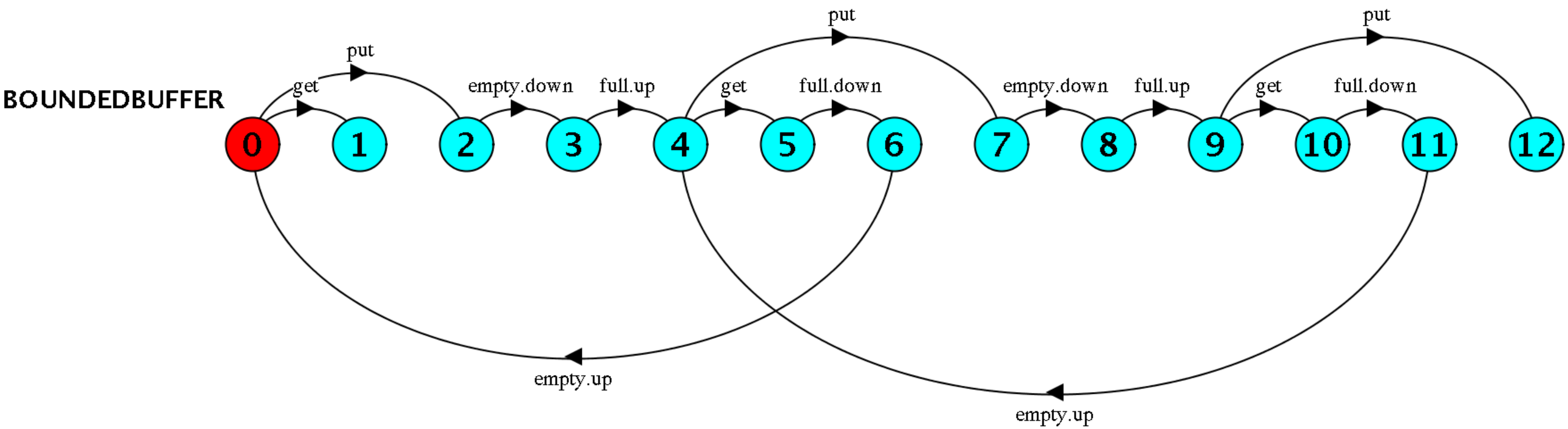
- *empty* is decreased in **put()**; it is blocked if *empty* is zero;
- *full* is decreased in **get()**; it is blocked if *full* is zero.

# Nested Monitors - Bounded Buffer Program

```
const Max = 2
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.
BUFFER =  (put -> empty.down -> full.up ->BUFFER
          |get -> full.down -> empty.up ->BUFFER).

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

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  || empty:SEMAPHORE(2)
                  || full:SEMAPHORE(0)
                  ).
```

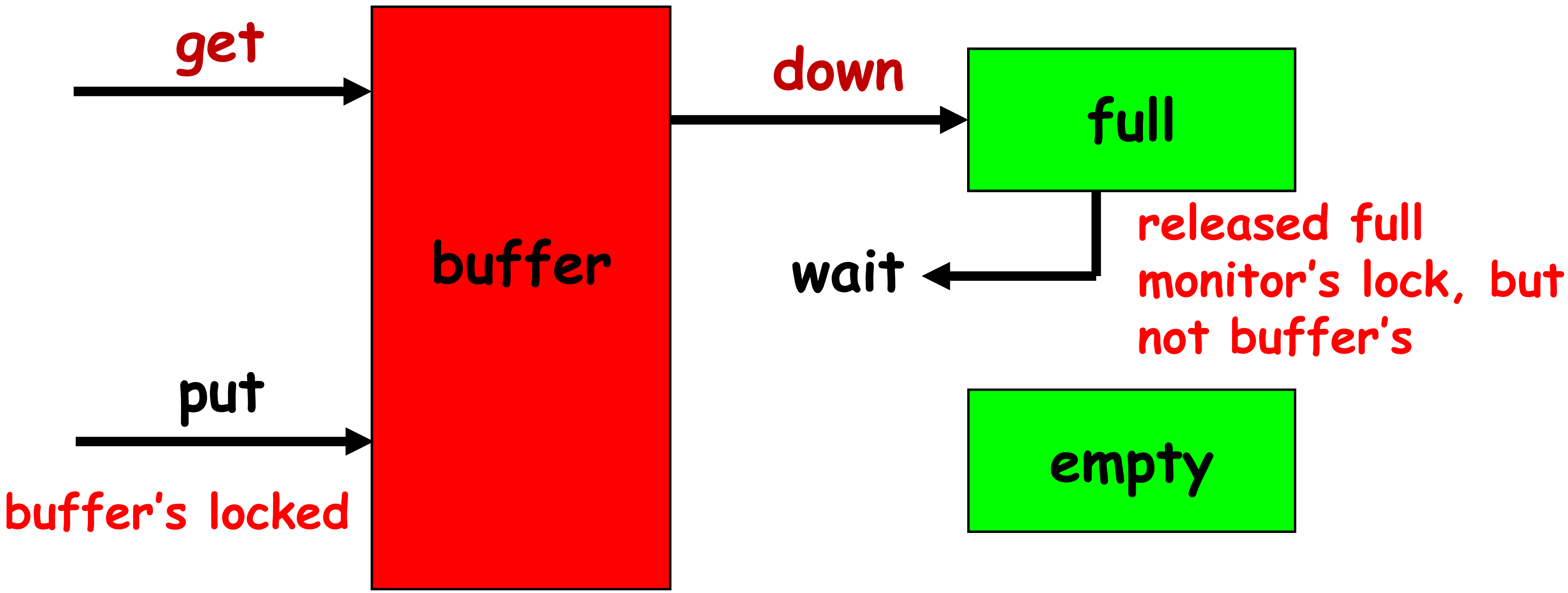
***Does this behave as desired?***

# Nested Monitors -  Bounded Buffer Program



# Nested Monitors - Bounded Buffer Program

```
synchronized public Object get()
                  throws InterruptedException{
    full.down(); // if there is no item in Buffer
    ...
  }
```



## Nested Monitors - Bounded Buffer Program

*LTSA* analysis predicts a possible **DEADLOCK**:

```
Composing
 potential DEADLOCK
States Composed: 28 Transitions: 32 in 60ms
Trace to DEADLOCK:
   get
```

The `Consumer` tries to `get` an item, but the buffer is empty. It blocks and releases the lock on the semaphore `full`. The `Producer` tries to `put` an item into the buffer, but also blocks.

This situation is known as the ***nested monitor problem***.

## Nested Monitors – Revised Bounded Buffer Program

The deadlock can be removed by ensuring that the **monitor lock** for the **buffer** is not **acquired** until ***after*** semaphores are decreased.

```
synchronized public E get() throws InterruptedException {
    full.down();
    E o =buf[out]; buf[out]=null; --count; out=(out+1)%size;
    empty.up();
    return (o);
  }   // original

 public E get() throws InterruptedException {
    full.down();
     synchronized(this) {
      E o =buf[out]; buf[out]=null; --count; out=(out+1)%size;}
    empty.up();
    return (o);
  } // new
```

# Nested Monitors – Revised Bounded Buffer Program

The deadlock can be removed by ensuring that the **monitor lock** for the **buffer** is not **acquired** until ***after*** semaphores are decreased.

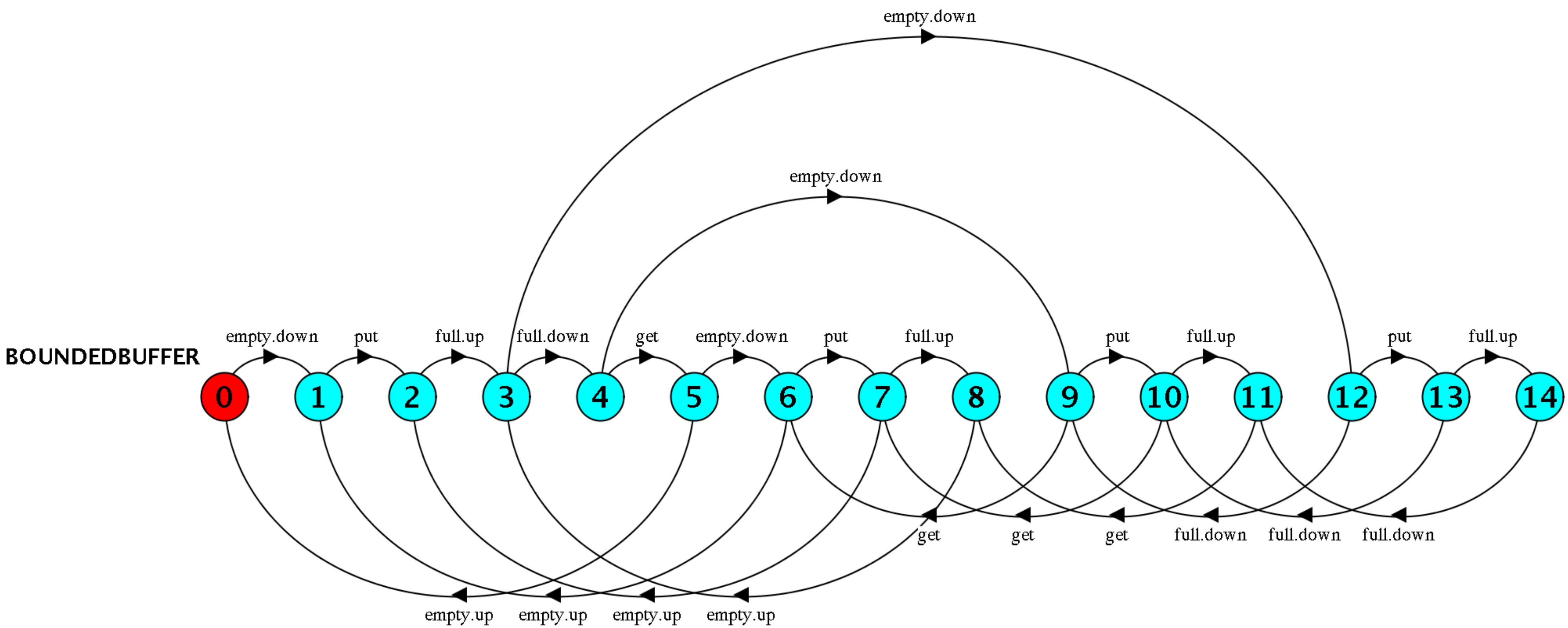
```
synchronized public void put(E o) throws InterruptedException{
   empty.down();
   buf[in] = o;  ++count; in=(in+1)%size;
   full.up();
 }  // original


 public void put(E o) throws InterruptedException {
    empty.down();
    synchronized(this) {
      buf[in] = o; ++count; in=(in+1)%size; }
    full.up();
  } // new
```

# Nested Monitors – Revised Bounded Buffer Program

```
// original
BUFFER = (put -> empty.down ->full.up ->BUFFER
         |get -> full.down ->empty.up ->BUFFER).
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

// New
BUFFER =  (put -> BUFFER
          |get -> BUFFER ).
PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).
```

The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are ***outside*** the monitor—get and put.

# Nested Monitors – Revised Bounded Buffer Program



# Summary

- Concepts
  - monitors: encapsulated data + access procedures
    mutual exclusion + condition synchronization
  - nested monitors
- Model
  - guarded actions
- Practice
  - private data and synchronized methods in Java
  - wait(), notify() and notifyAll() for condition synchronization
  - single thread active in the monitor at a time