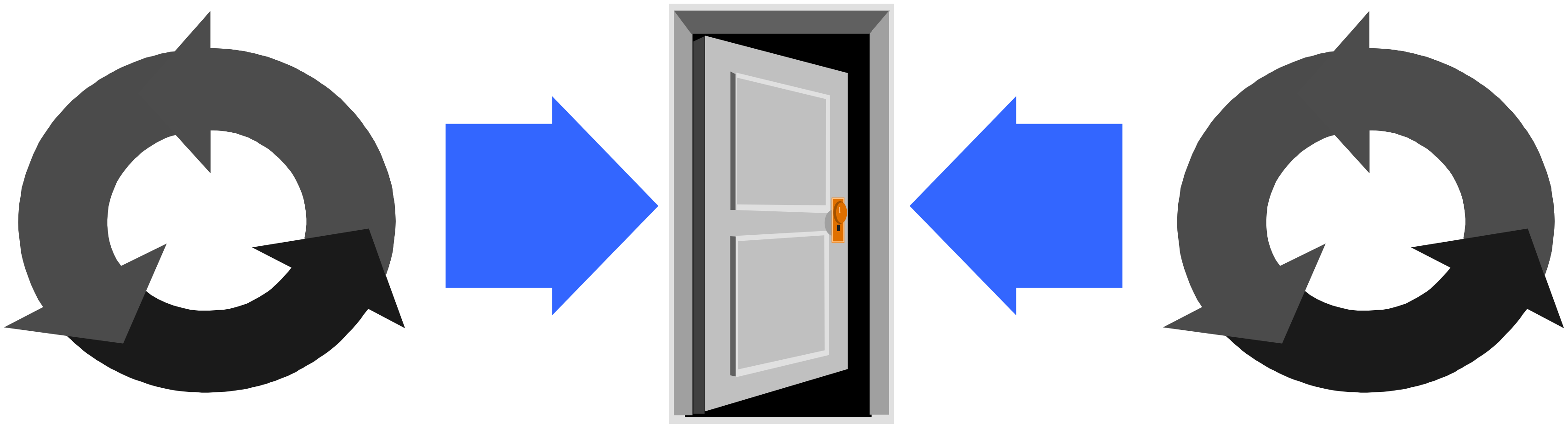
5 - Monitors & Condition Synchronization

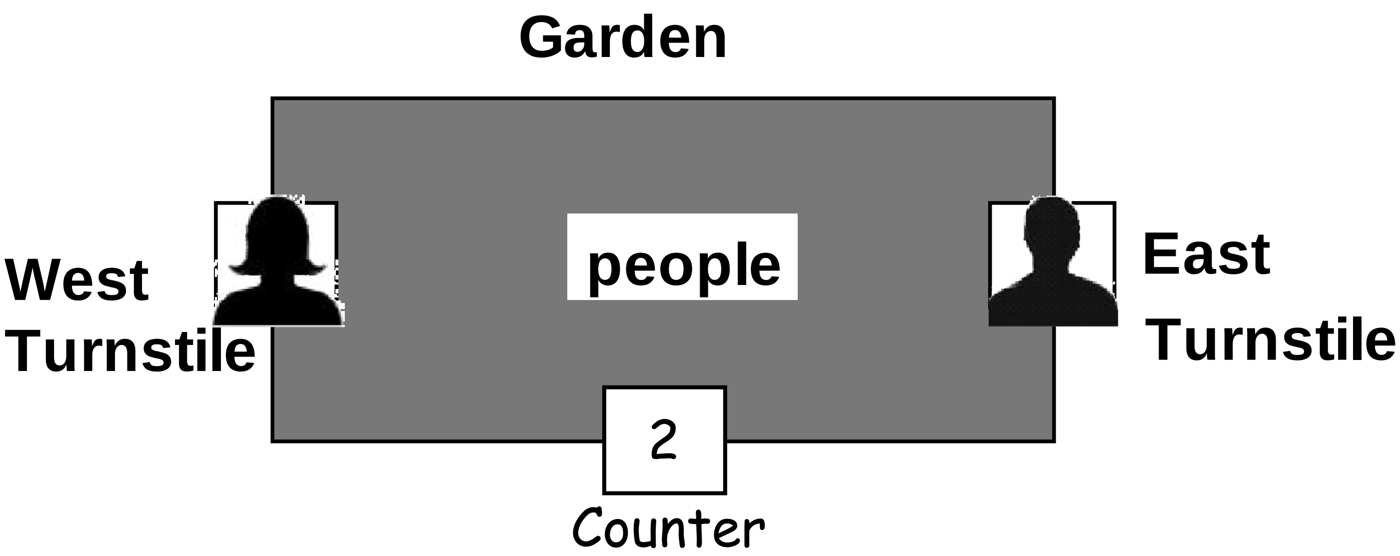


Credits for the slides:
Claus Braband
Jeff Magee & Jeff Kramer

Alexandre David
adavid@cs.aau.dk

Repetition - Interference (Ornamental Garden Problem)

People enter an ornamental garden through either of two turnstiles. Management wishes to know how many are in the garden at any time. (Nobody can exit).



Repetition - Running the Applet

After the East and West turnstile threads each have incremented the counter **20 times**, the garden people counter is **not the sum of the counts** displayed.

Repetition - Model Checking (reveals the error)

Ornamental Garden Model reveals the error:

```
||TESTGARDEN = ( GARDEN || TEST ).
```

- Use *LTSA* to perform an exhaustive search for **ERROR**:

```
Trace to property violation in TEST:
    go
    east.arrive
    east.value.read.0
    west.arrive
    west.value.read.0
    east.value.write.1
    west.value.write.1
    end
    display.value.read.1
    wrong
```

LTSA produces the shortest path to reach the **ERROR** state.

Repetition - Interference and Mutual Exclusion

- Interference (Java):

```
x = x + 1;   ||   x = x + 1;
```

- Mutual exclusion (Java):

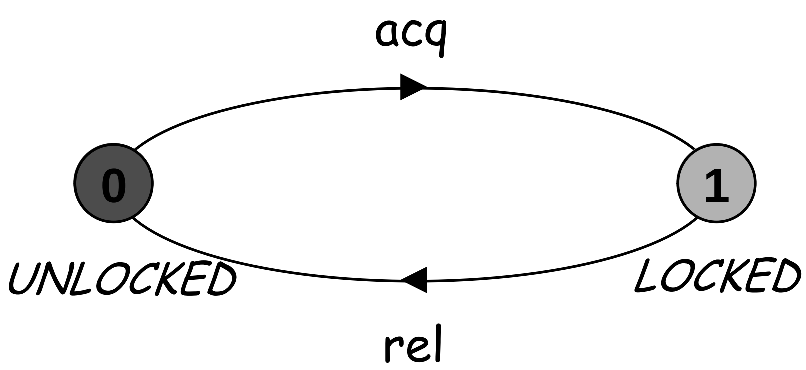
```
synchronized (obj) {
    x = x + 1;
}
```

||

```
synchronized (obj) {
    x = x + 1;
}
```

- Modelling mutual exclusion (FSP):

```
LOCK = (acq -> rel -> LOCK).
```



Monitors & Condition Synchronization

Concepts: monitors:

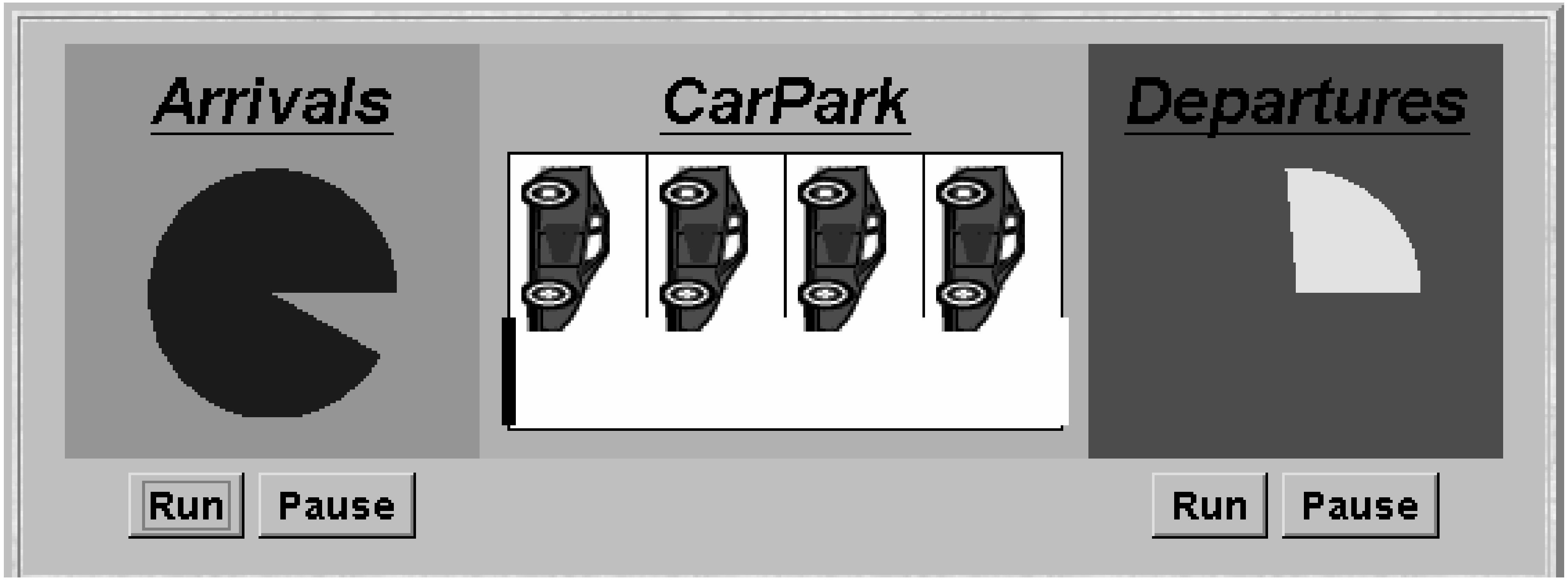
encapsulated data + access procedures
mutual exclusion + condition synchronization
single access procedure active in the monitor

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

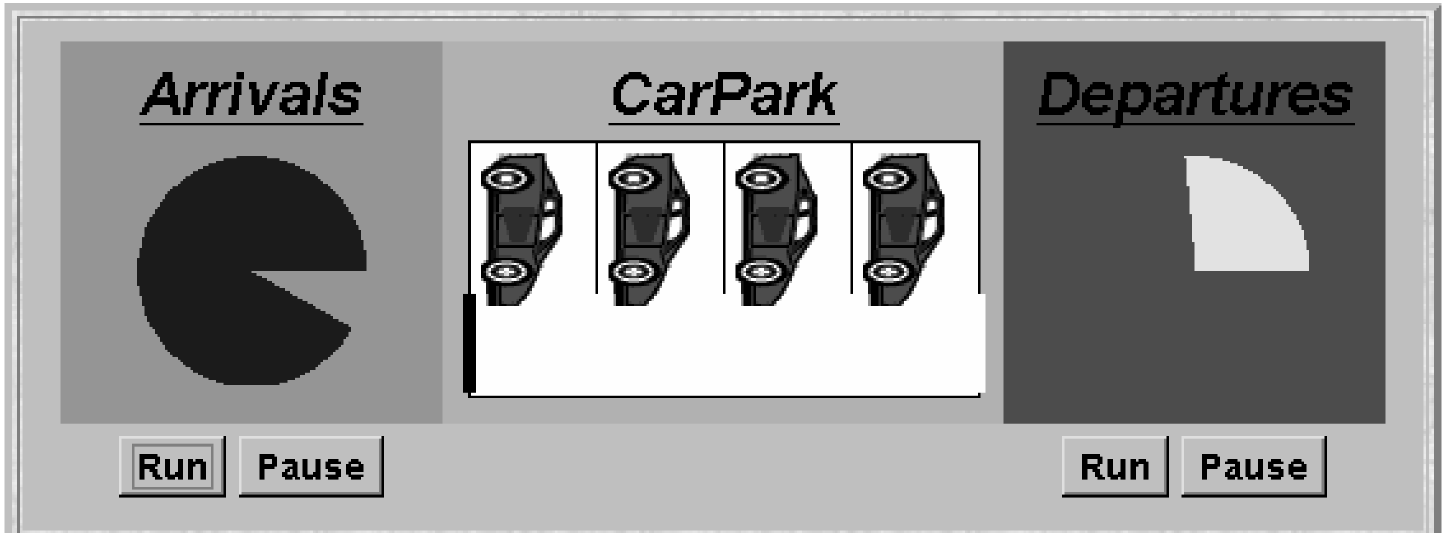
5.1 Condition Synchronization (Car Park)



A controller is required to ensure:

- cars can only enter when not full
- cars can only leave when not empty (duh!)

Car Park Model (Actions and Processes)



- **Actions of interest:**
  - `arrive`
  - `depart`

- **Identify processes:**
  - Arrivals
  - Departures
  - Control

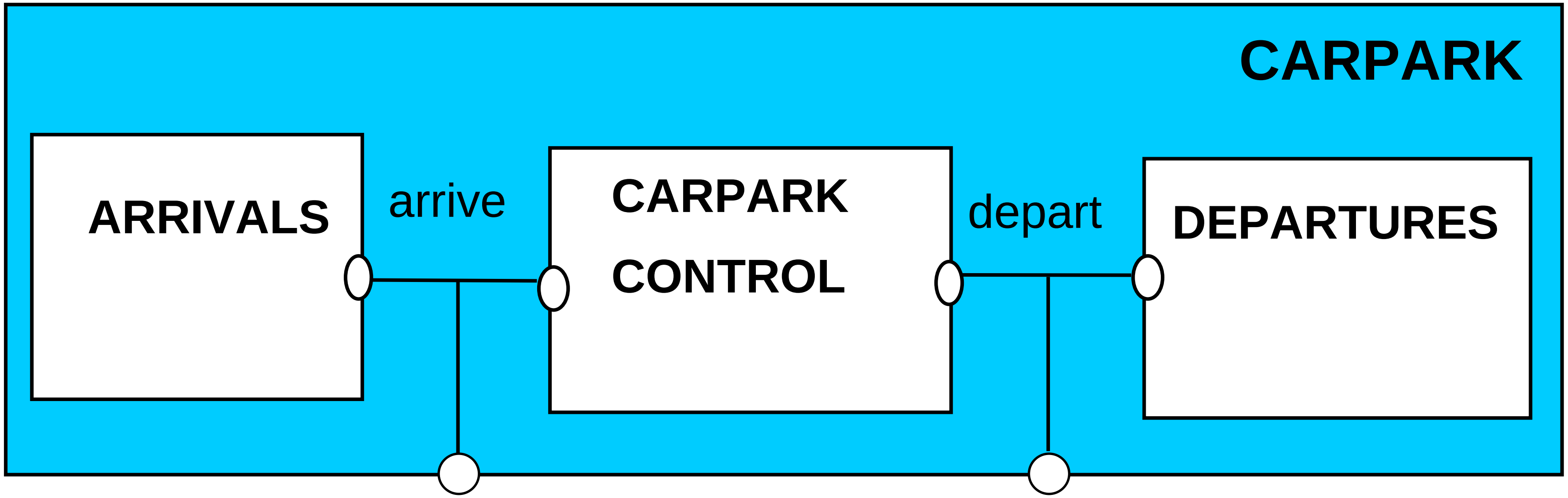
  (Arrivals and Departures: *env*)

Car Park Model (Structure Diagram)

♦ Actions of interest:
- **`arrive`**
- **`depart`**

♦ Identify processes:
- Arrivals
- Departures
- Control

(Arrivals and Departures: *env*)



Car Park Model (FSP)

```
ARRIVALS = (arrive -> ARRIVALS).

DEPARTURES = (depart -> DEPARTURES).


CONTROL(N=4) = SPACES[N],

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


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

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

LTS?

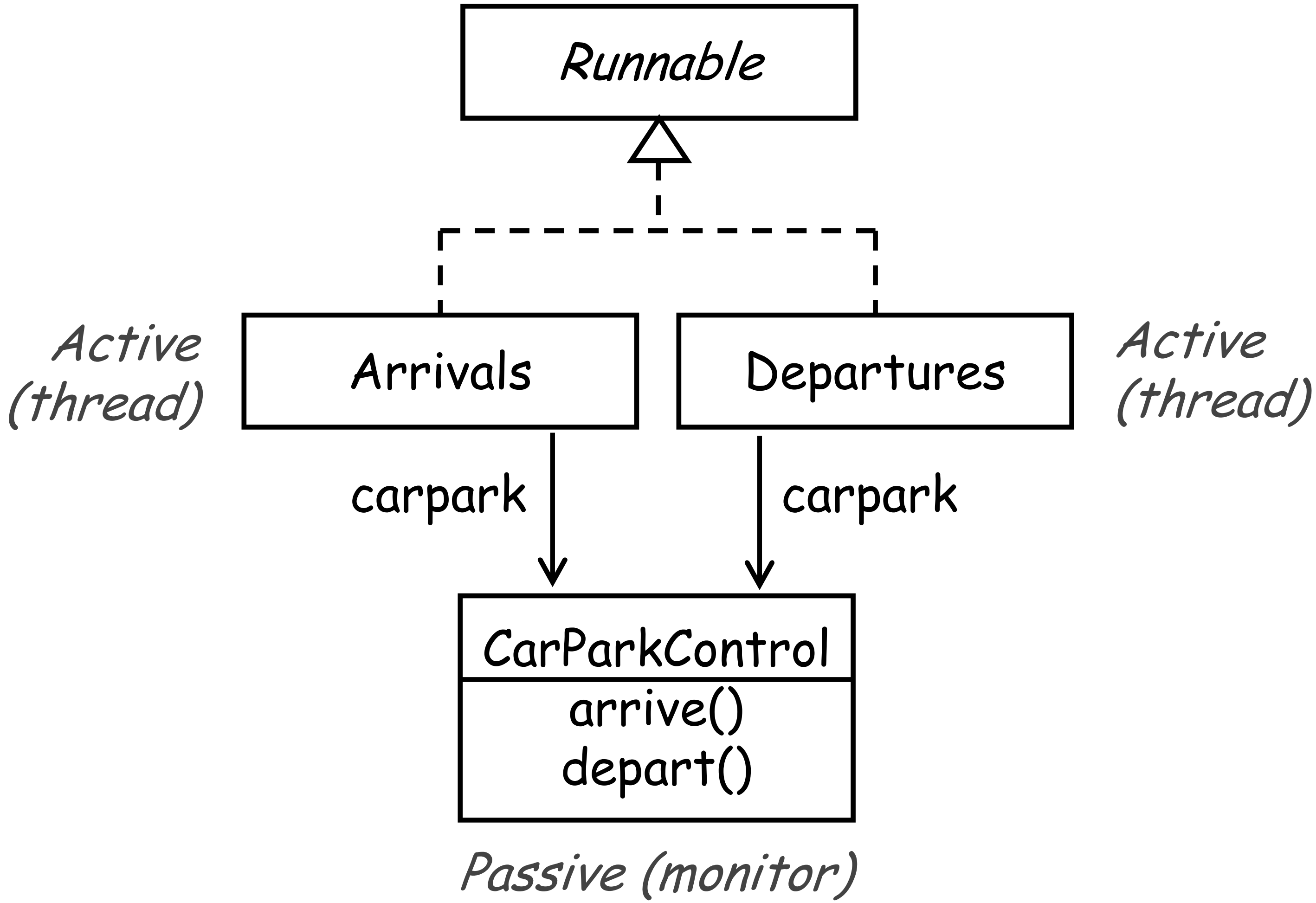
Car Park Program

- Model
  - - all entities are processes interacting via ***shared actions***
- Program - need to identify threads and monitors:
  - thread - **active** entity which initiates (output) actions
  - monitor - **passive** entity which responds to (input) actions.

For the carpark?

• Arrivals:	**active**	=>	thread
• Departures:	**active**	=>	thread
• Control:	**passive**	=>	monitor

Car Park Program (Interesting part of Class Diagram)



Car Park Program - `Applet::start()`

The Applet's start() method creates:

- **CarParkControl** monitor (with condition synchr.)
- **Arrival** thread
- **Departures** thread

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

The **CarParkControl** is *shared* by **Arrival** and **Departures** threads

Car Park 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 _) {}
    }
}
```

Similarly, **Departures** calls:

`carpark.depart()`

How do we implement the control of **CarParkControl**?

Car Park Program - `CarParkControl` Monitor

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

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

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

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

Encapsulation ~ **protected**

Mutual exclusion ~ **synchronized**

Condition synchronization?

Block if full? (spaces==0)

Block if empty? (spaces==N)

Condition Synchronization in Java

Java provides a **thread wait queue per object** (*not* per class).

`Object` has methods:

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

Waits to be *notified* (i.e. another thread invokes **notify**). Releases the synchronization lock associated with the obj.

When notified, the thread must reacquire the synchr. lock.

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

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

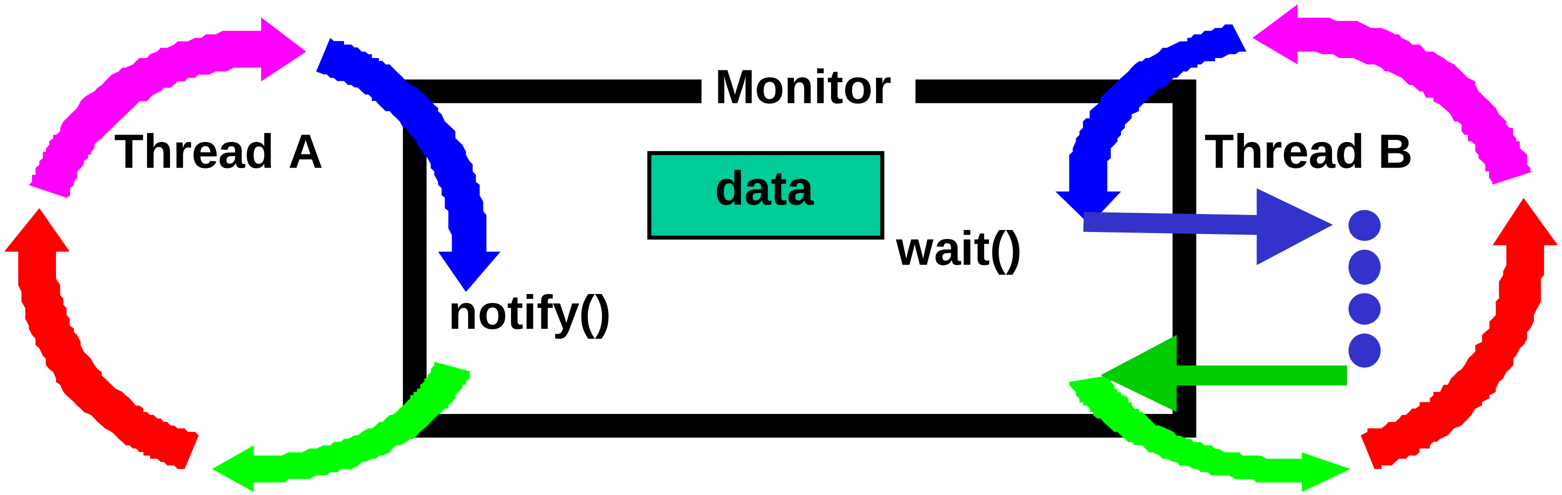
Wakes up (notifies) a thread waiting on the object's queue.

Condition Synchronization in Java (enter/exit)

A thread:

- ***Enters*** a monitor when a thread acquires the lock associated with the monitor;
- ***Exits*** a monitor when it releases the lock.

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



Condition Synchronization in FSP and Java

```
FSP:    when(cond) action->NEWSTATE
```

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

The **while** loop is necessary to re-test 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, capacity;

    synchronized void arrive() throws Int'Exc' {
        while (spaces==0) wait();
        --spaces;
        notify();
    }

    synchronized void depart() throws Int'Exc' {
        while (spaces==capacity) wait();
        ++spaces;
        notify();
    }
}
```

Why is it sensible to use **notify()** *here rather than* **notifyAll()**?

Models to Monitors - Guidelines

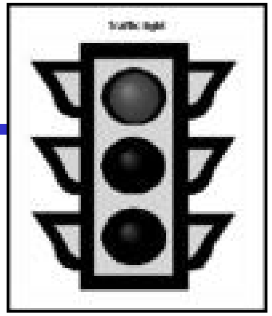
- **Active** entities (that initiate actions) are implemented as **threads**.
- **Passive** entities (that respond to actions) are implemented as **monitors**.

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.

The while loop condition is the negation of the model guard condition.

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

5.2 Semaphores



Semaphores are widely used for dealing with inter-process synchronization in operating systems.

Semaphore *s* : integer var that can take only non-neg. values.

s.down():	**when** s>0 **do** decrement(s);	*Aka. "P" ~ Passern*
s.up():	increment(s);	*Aka. "V" ~ Vrijgeven*

Usually implemented as *blocking wait:*

s.down(): **if** (s>0) **then** decrement(s);
else block execution of calling process

s.up(): **if** (processes blocked on s) **then** awake one of them
else increment(s);

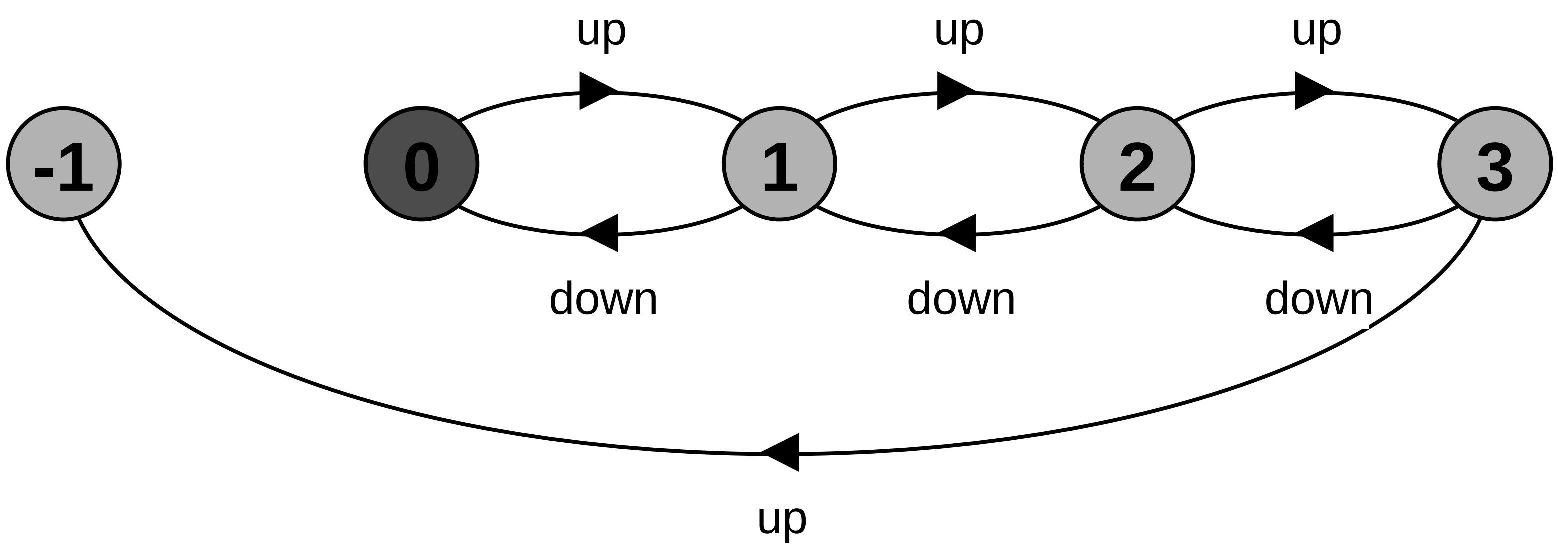
Modelling Semaphores

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?

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)).
```

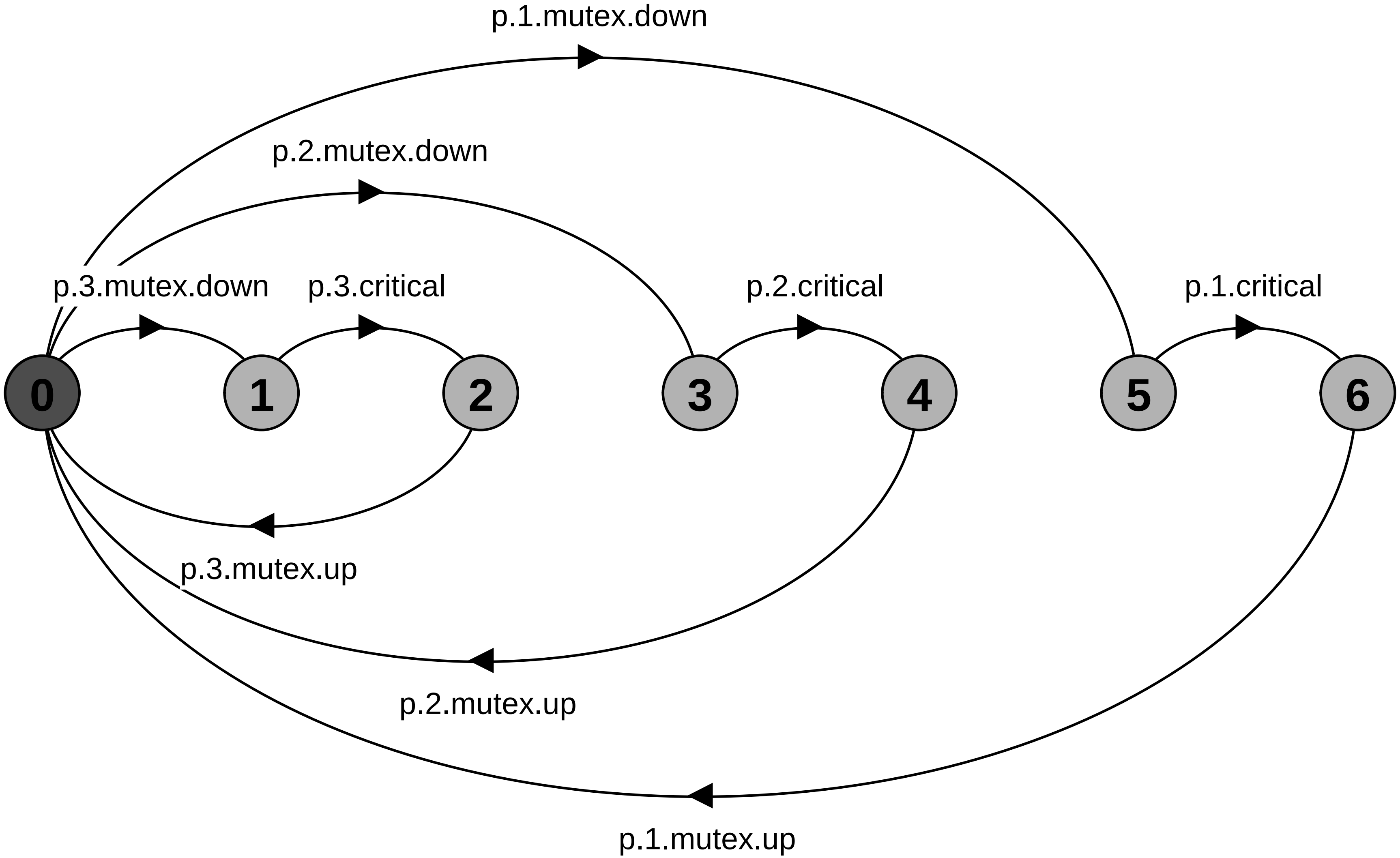
For mutual exclusion, the semaphore initial value is 1. *Why?*

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

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

LTS?

Semaphore Demo - Model



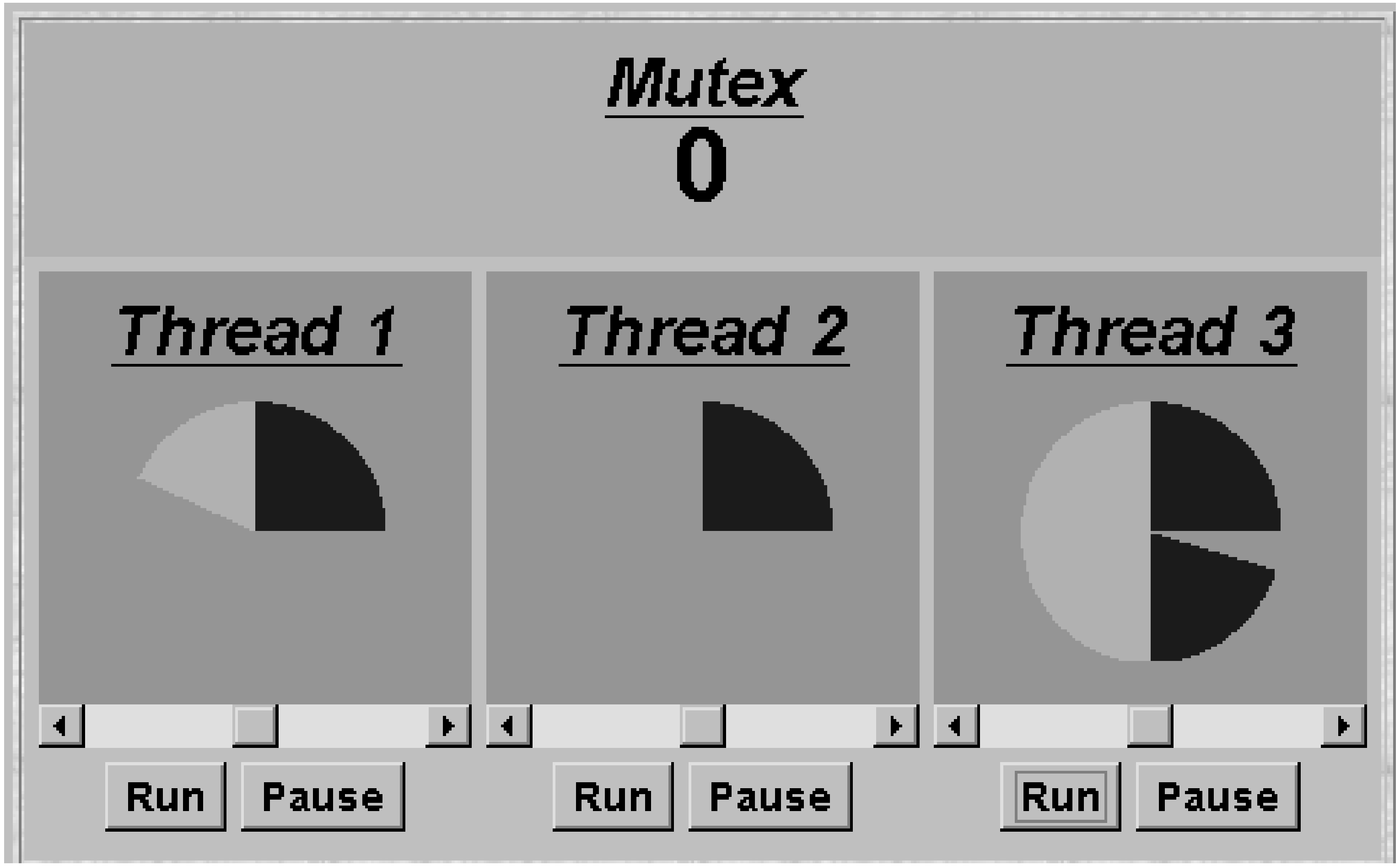
Semaphores in Java

Semaphores: passive objects => implemented as **monitors**.

```
public class Semaphore {
    private int value;

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

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

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

In practice,
semaphore is a ***low-level*** *mechanism often used in implementing* ***higher-level*** *monitor constructs.*

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.

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();                   // acquire
                while(ThreadPanel.rotate()); // critical
                mutex.up();                     // release
            }
        } catch(InterruptedException _) {}
    }
}
```

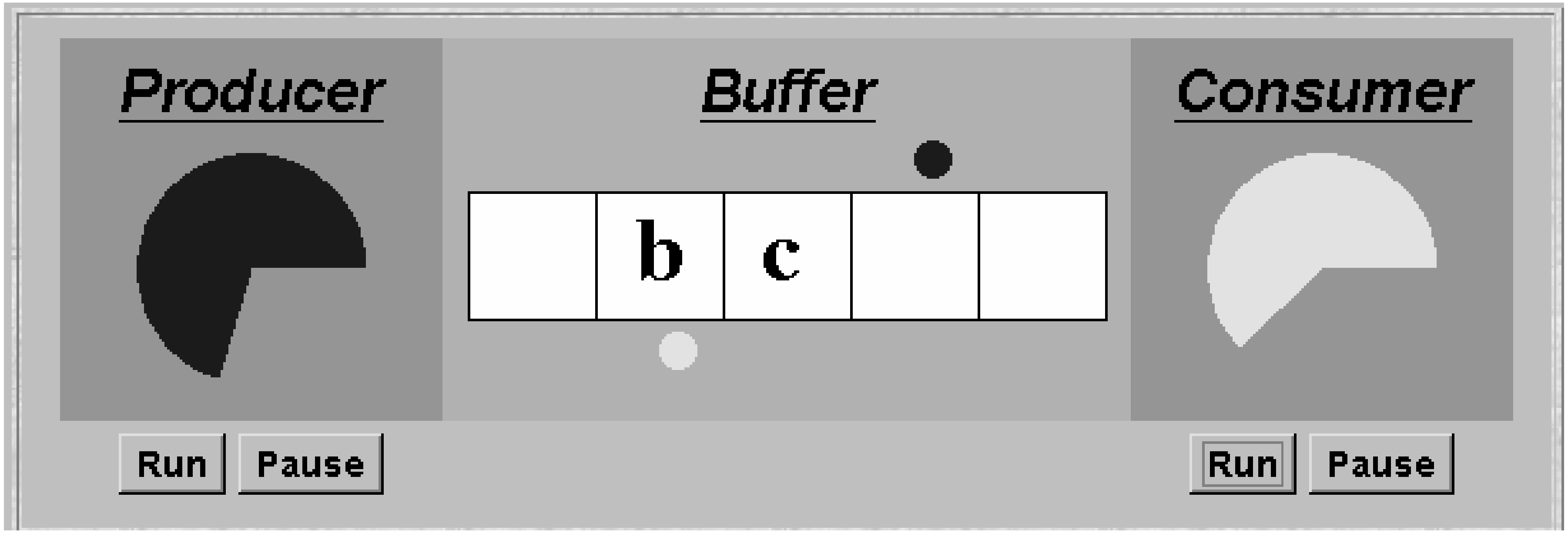
Threads and semaphore are created by the applet **`start()`** method.

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

5.3 Bounded Buffer

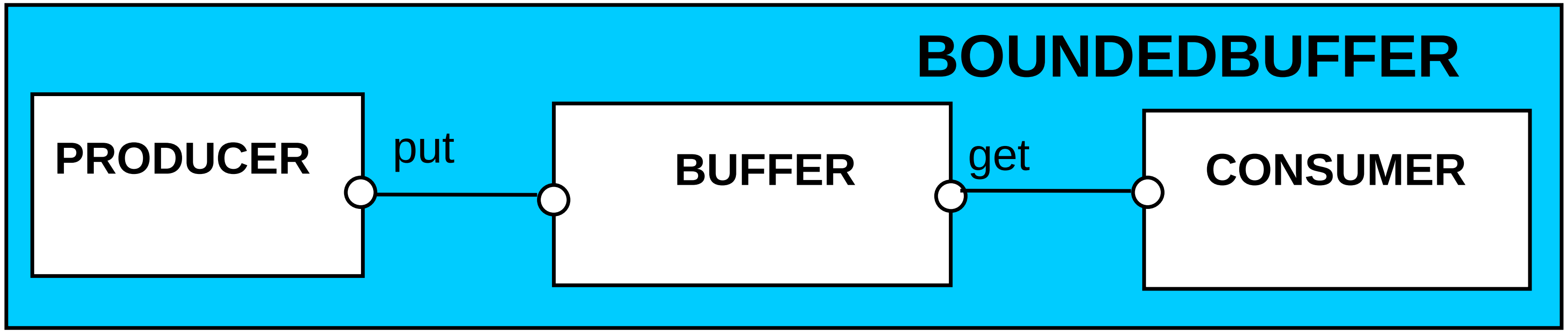
A **bounded buffer** consists of a fixed number of slots.

Items are put into the buffer by a ***producer*** process and removed by a ***consumer*** process:



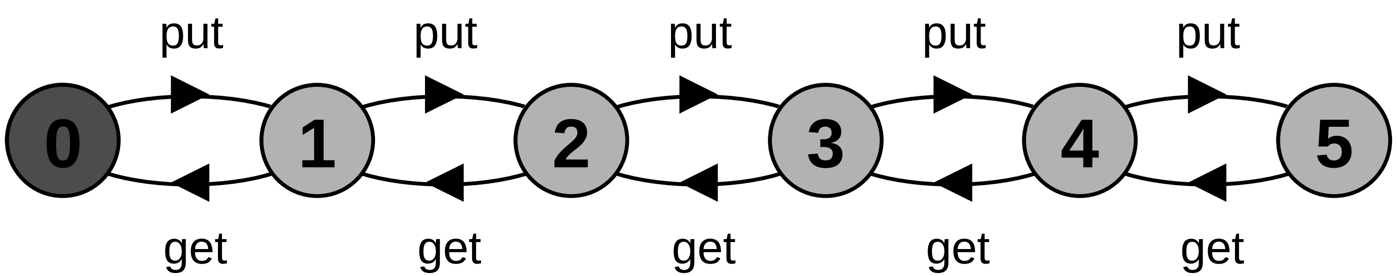
≈ Car Park Example!

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

```
PRODUCER = (put->PRODUCER).

CONSUMER = (get->CONSUMER).

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

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

Bounded Buffer Program - Buffer Monitor

```
public interface Buffer {
    public void put(Object o) throws Interrupted
    public Object get()       throws Interrupted
}
```

```
class BufferImpl implements Buffer {
    protected Object[] buf;
    protected int in, out, count, size;
    …
    synchronized void put(Object o) throws Int'Exc' {
        while (count==size) wait();
        buf[in] = o;
        count++;
        in = (in+1) % size;
        notifyAll();
    }
}
```

We separate the interface to permit an alternative implementation later.

Similarly for get()

```
     synchronized Object get() throws Int'Exc' {
1.       while (count==0) wait();
2.       Object obj = buf[out];
3.       buf[out] = null;
4.       count--;
5.       out = (out+1) % size;
6.       notifyAll();
7.       return obj;
     }
```

- What happens if we move **notifyAll()** up earlier (e.g. after line 1)?
- What is the point of line 3?

Bounded Buffer Program - Producer Process

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

    Producer(Buffer b) { buf = b; }

    public void run() {
        try {
            int i = 0;
            while(true) {
                ThreadPanel.rotate(12);
                buf.put(new Character(alpha.charAt(i)));
                i=(i+1) % alpha.length();
                ThreadPanel.rotate(348);
            }
        } catch (InterruptedException _) {}
    }
}
```

Similarly **Consumer** which calls **buf.get()**

5.4 Nested Monitors

Suppose that, instead of using the ***count*** variable and condition synchronization, we instead use 2 semaphores ***full*** and ***empty*** to reflect the state of the buffer:

```
class SemaBuffer implements Buffer {
    protected Object buf[];
    protected int in, out, count, size;

    Semaphore full;   //counts number of items
    Semaphore empty;  //counts number of spaces

    SemaBuffer(int s) {
        size  = s; in = out = count = 0;
        buf   = new Object[size];
        full  = new Semaphore(0);
        empty = new Semaphore(size);
    }
}
```

Nested Monitors - Bounded Buffer Program

```
synchronized public void put(Object o) throws Int'Exc' {
    empty.down();
    buf[in] = o;
    count++;
    in = (in+1) % size;
    full.up();
}

synchronized public Object get() throws Int'Exc' {
    full.down();
    Object o = buf[out];
    buf[out] = null;
    count--;
    out = (out+1) % size;
    empty.up();
    return o;
}
```

empty is decremented during a **put**, which is blocked if *empty* is zero.

full is decremented by a **get**, which is blocked if *full* is zero.

Does this behave as desired?

Nested Monitors - Bounded Buffer Model

```
PRODUCER = (put -> PRODUCER).

CONSUMER = (get -> CONSUMER).

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

BUFFER = (put -> empty.down -> full.up  -> BUFFER
         |get -> full.down  -> empty.up -> BUFFER).

||BOUNDEDBUFFER =
     ( PRODUCER || BUFFER || CONSUMER
               || empty:SEMAPHORE(5)
               || full:SEMAPHORE(0)      ).
```

Does this behave as desired?

Nested Monitors - Bounded Buffer Model

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** a character, but the buffer is empty. It blocks and releases the lock on the semaphore **full**. The **Producer** tries to **put** a character into the buffer, but also blocks. ***Why?***

Nested Monitors - Bounded Buffer Model

LTSA analysis predicts a possible **DEADLOCK**:

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

1) Consumer calls SemaBuffer.**get()**, acquiring a lock on the buffer
   synchronized public Object get()

2) Semaphore.**down()** acquires another lock on the Semaphore
   synchronized public void down()

3) Semaphore.**down()** releases *only its own* lock using **wait()**

4) Producer calls SemaBuffer.put(), blocking on the buffer
   synchronized public void put(Object)

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

Nested Monitors - Revised Bounded Buffer Program

The only way to avoid it in Java is by ***careful design***. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until ***after*** semaphores are decremented.

```
public void put(Object o) throws Int'Exc' {
    empty.down();
    synchronized (this) {
        buf[in] = o;
        count++;
        in = (in+1) % size;
    }
    full.up();
  }
```

Nested Monitors - Revised Bounded Buffer Model

```
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 .

Does this behave as desired?

Minimized LTS?

5.5 Monitor invariants

An **invariant** for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor i.e. on thread **entry** to and **exit** from a monitor .

INV(CarParkControl):		$0 \leq spaces \leq N$
INV(Semaphore):		$0 \leq value$
INV(Buffer):		$0 \leq count \leq size$
	and	$0 \leq in < size$
	and	$0 \leq out < size$
	and	$in = (out + count) \% size$

Like normal invariants, but must also hold when lock is released (wait)!

Summary

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