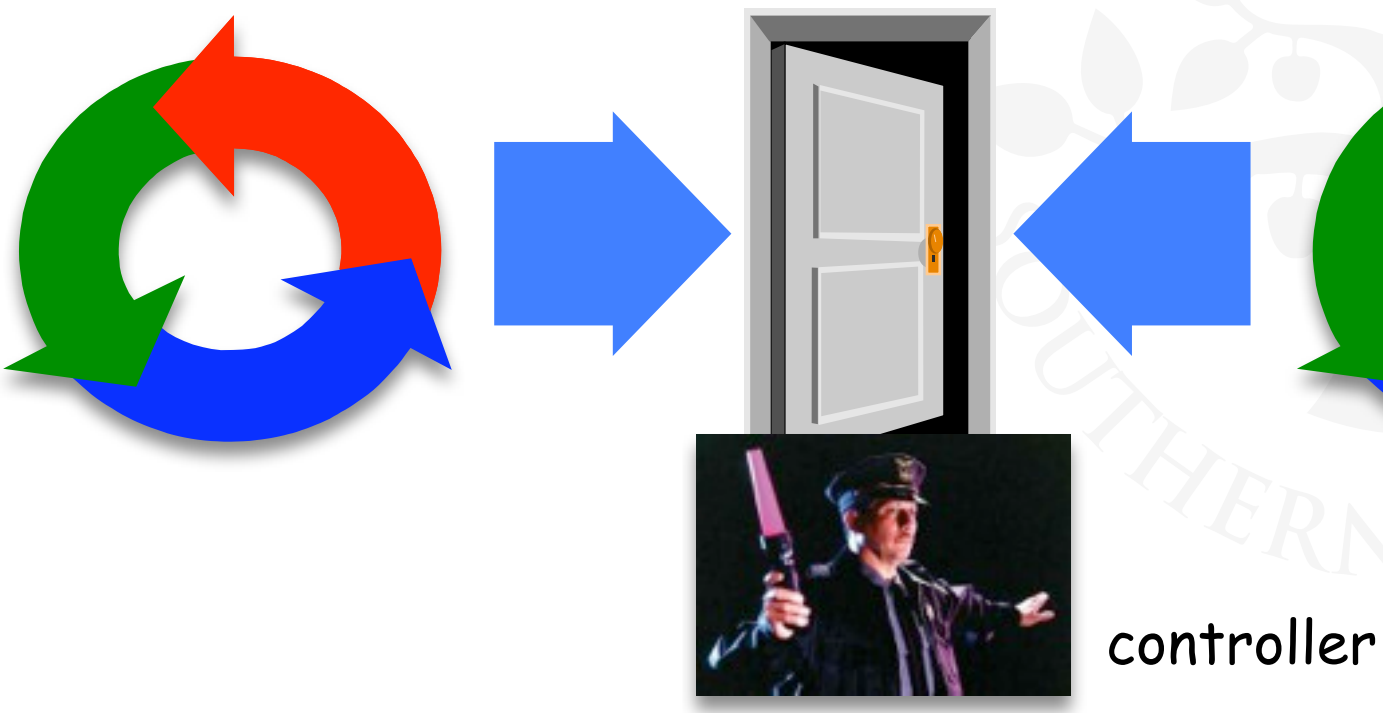
Chapter 5

University of Southern Denmark

Monitors & Condition Synchronisation



Monitors & Condition Synchronisation

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

encapsulated data + access procedures +

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

encapsulated data + access procedures +

mutual exclusion + condition synchronisation +

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

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

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Models: guarded actions

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Models: guarded actions

Practice: private data and synchronized methods (exclusion).

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Models: guarded actions

Practice: private data and synchronized methods (exclusion).
wait(), notify() and notifyAll() for condition synchronisation

Monitors & Condition Synchronisation

University of Southern Denmark

Concepts: monitors (and controllers):

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

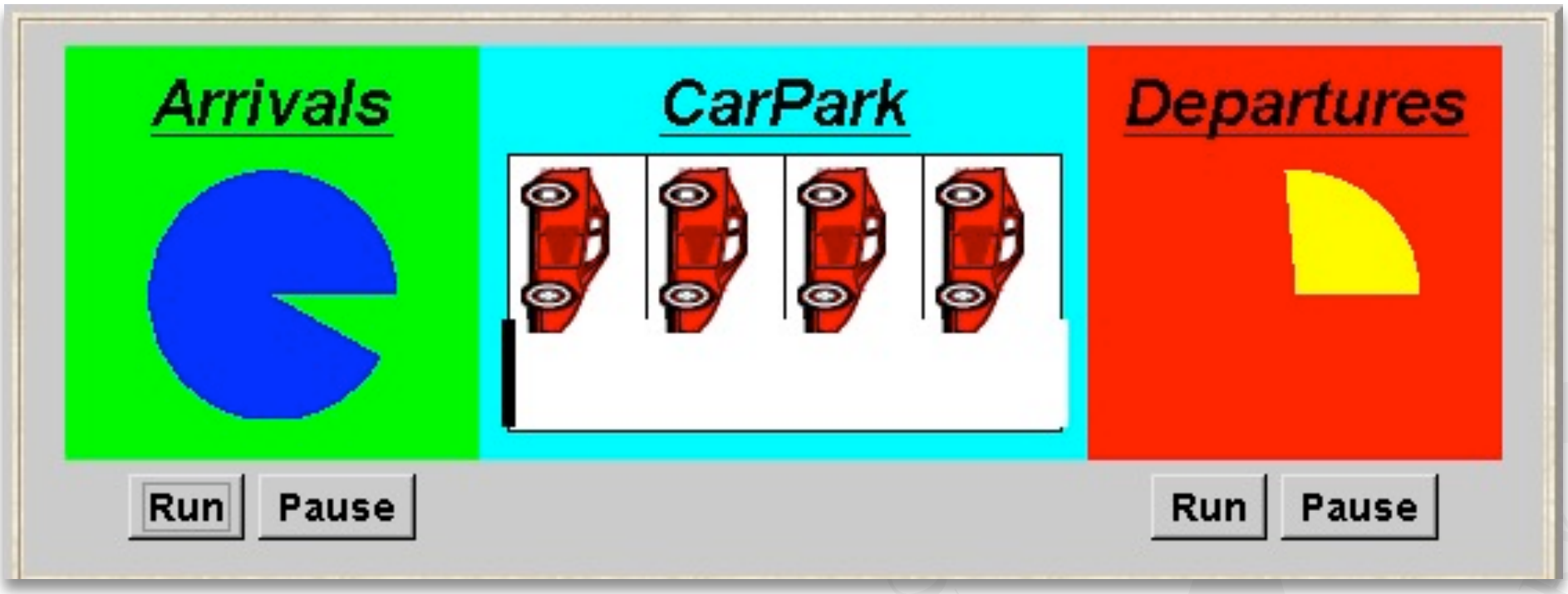
nested monitors ("nested monitor problem")

Models: guarded actions

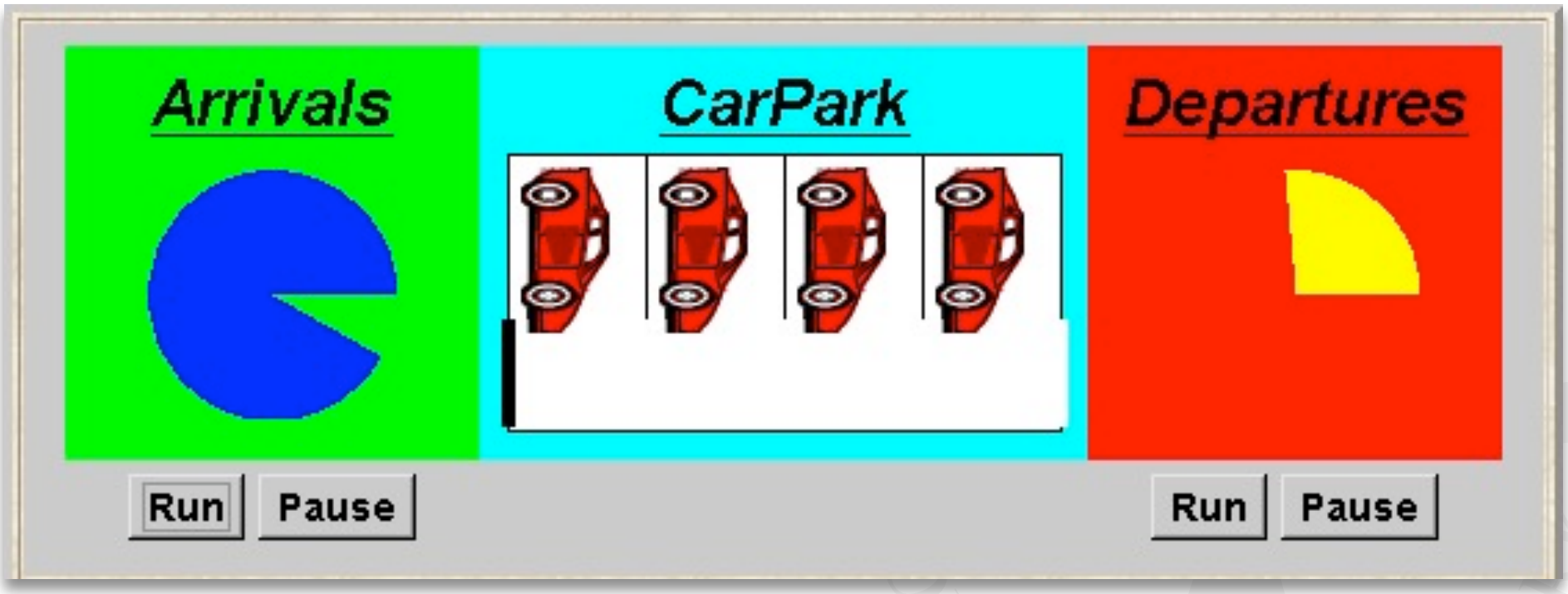
Practice: private data and synchronized methods (exclusion).
wait(), notify() and notifyAll() for condition synchronisation
single thread active in the monitor at a time

Condition Synchronisation

5.1 Condition Synchronisation (Car Park)

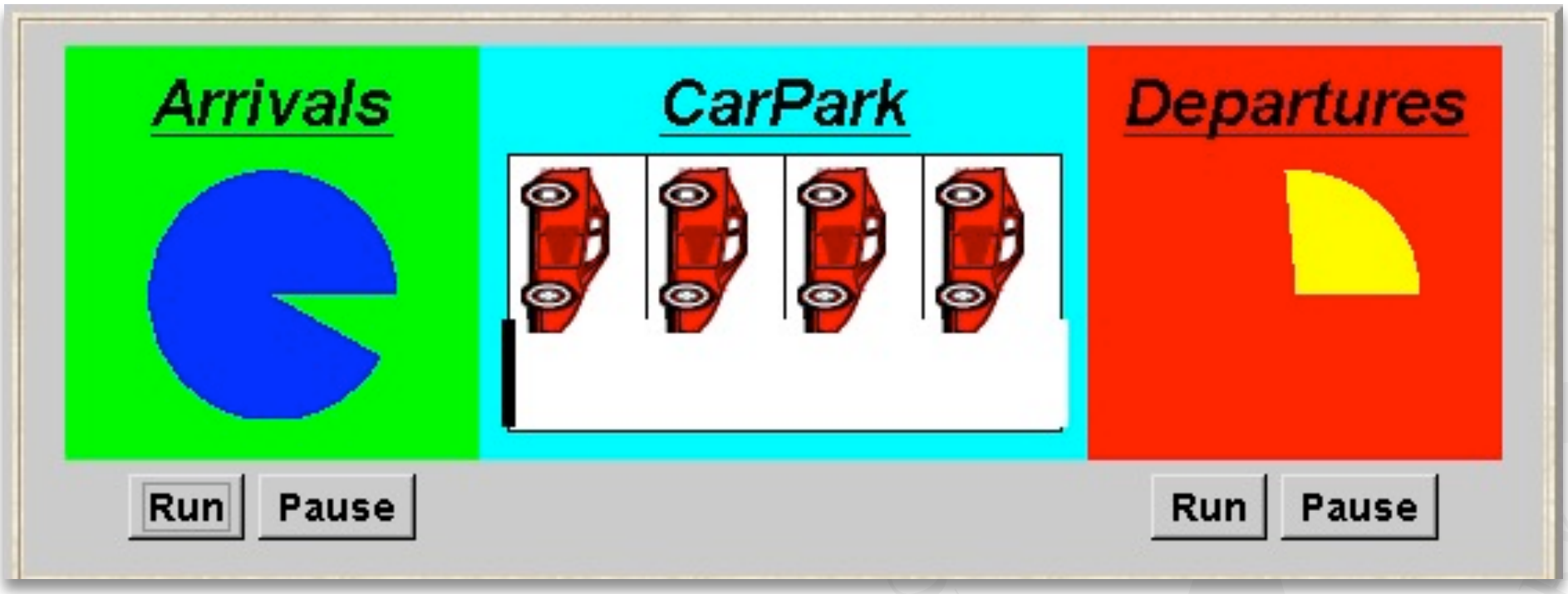


5.1 Condition Synchronisation (Car Park)



A controller is required to ensure:

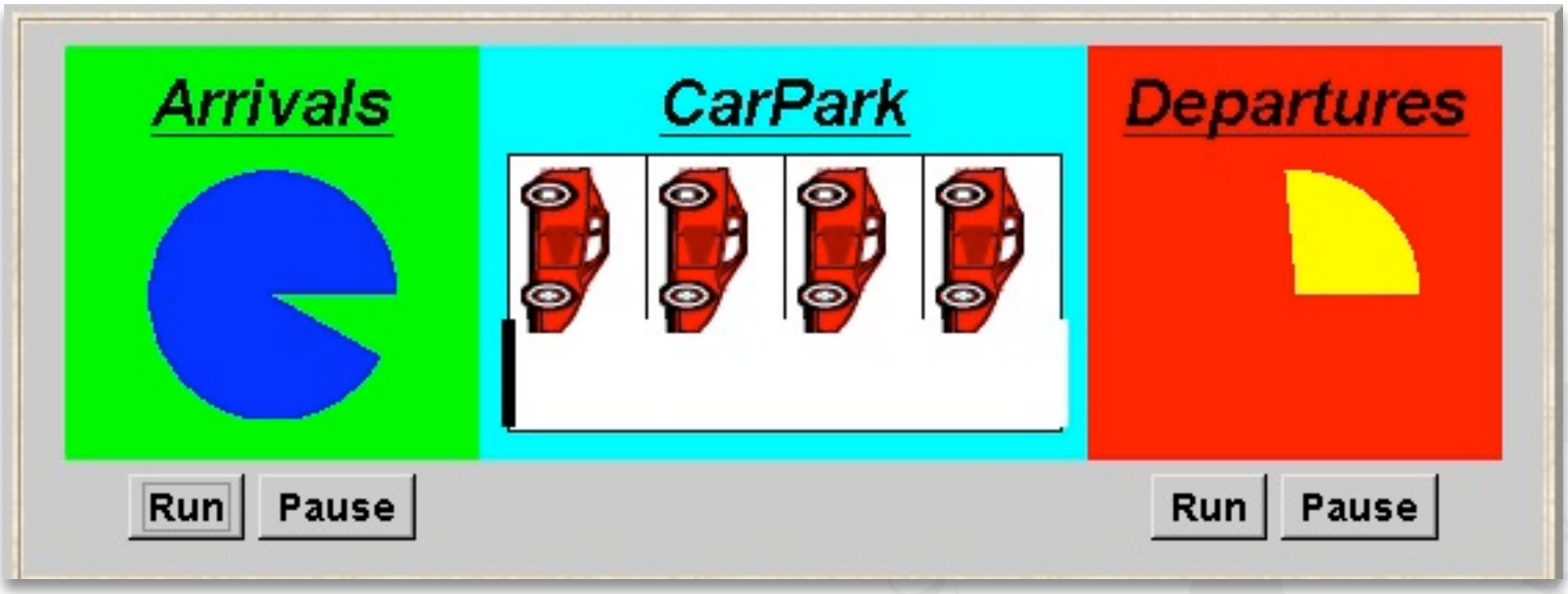
5.1 Condition Synchronisation (Car Park)



A controller is required to ensure:

- cars can only enter when not full

5.1 Condition Synchronisation (Car Park)

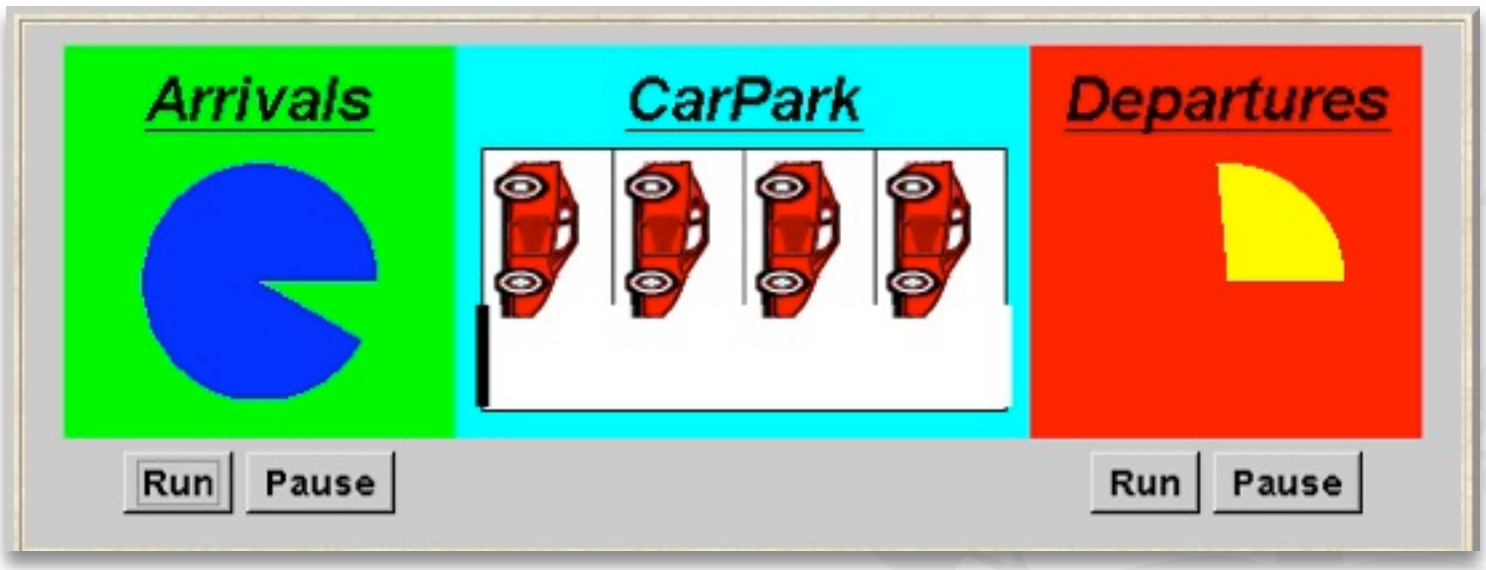


A controller is required to ensure:

- cars can only enter when not full
- cars can only leave when not empty

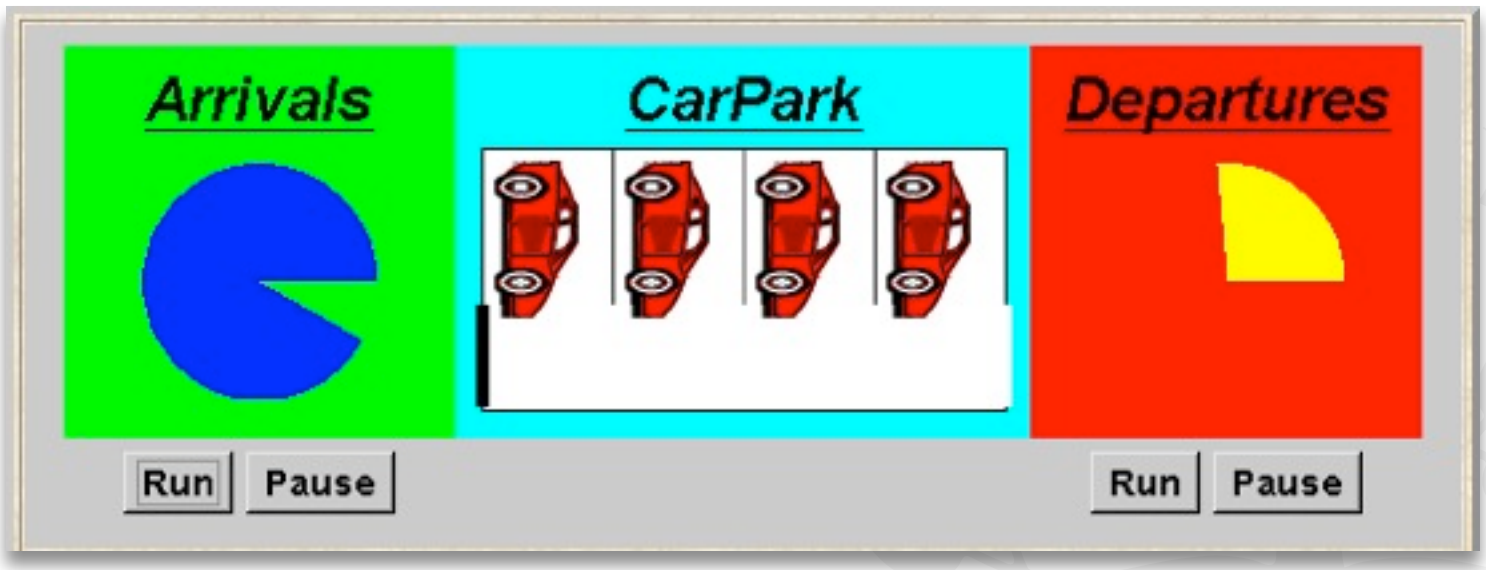
Car Park Model (Actions and Processes)

University of Southern Denmark



Car Park Model (Actions and Processes)

University of Southern Denmark

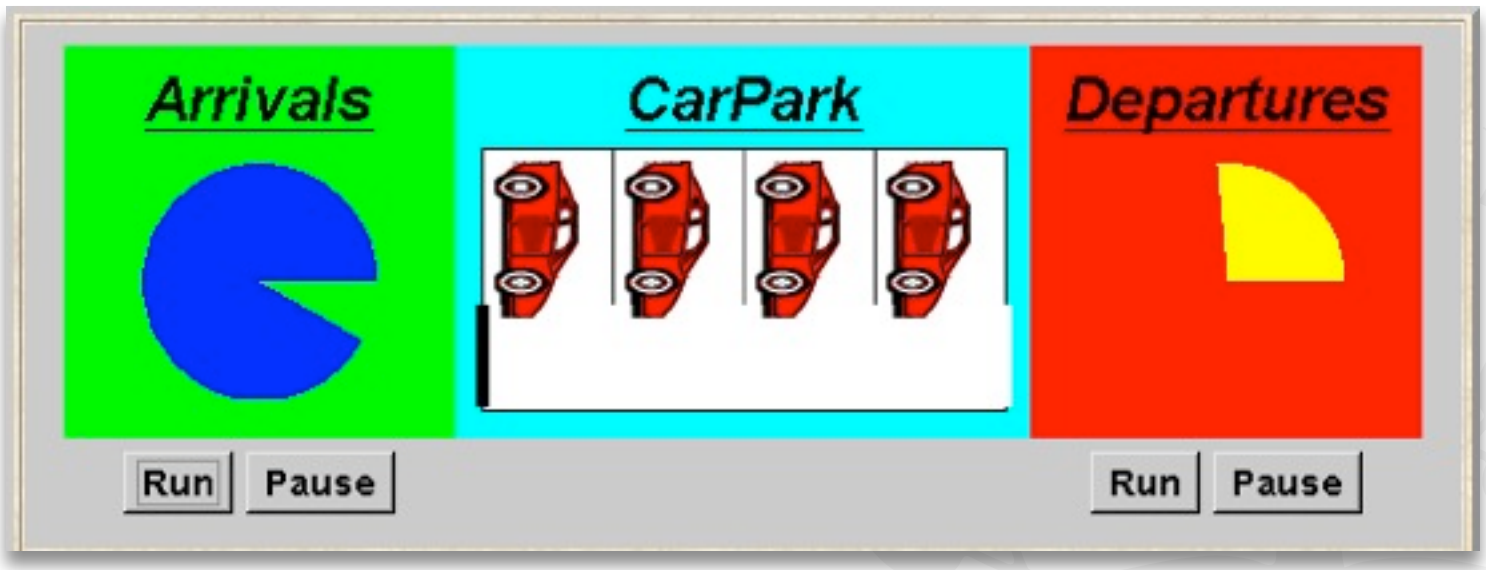


- Actions of interest:

- Processes:

Car Park Model (Actions and Processes)

University of Southern Denmark

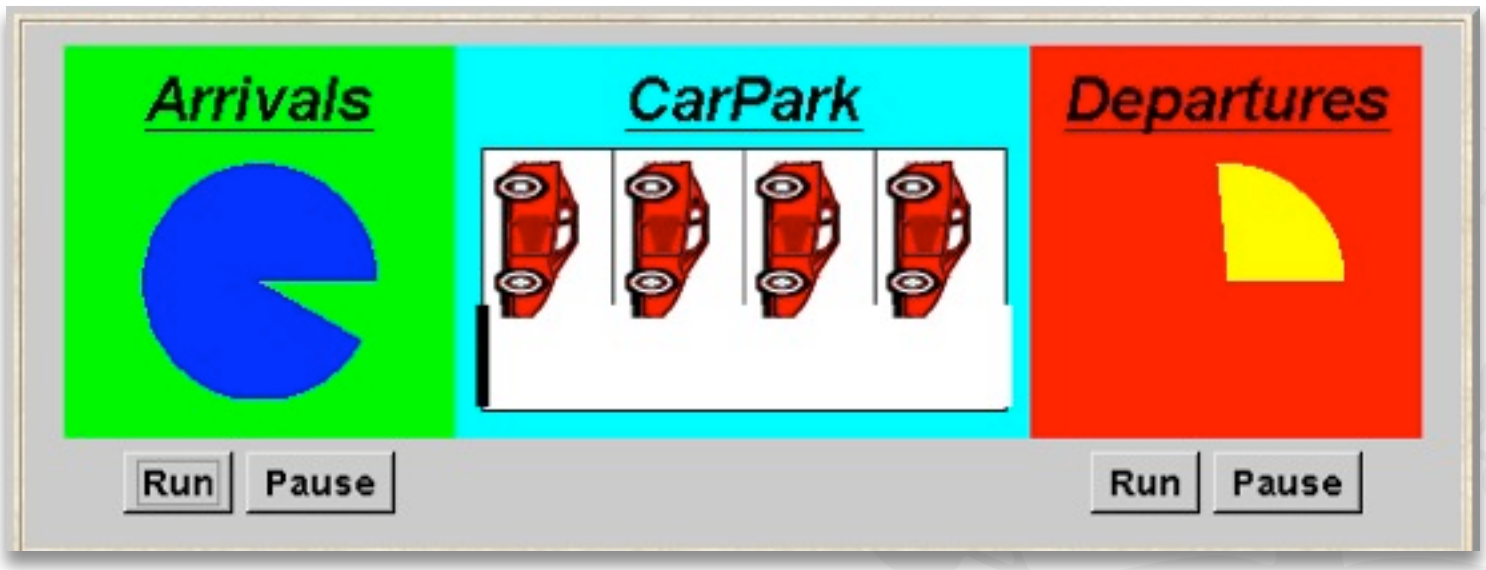


- Actions of interest:
  - `arrive`

- Processes:

Car Park Model (Actions and Processes)

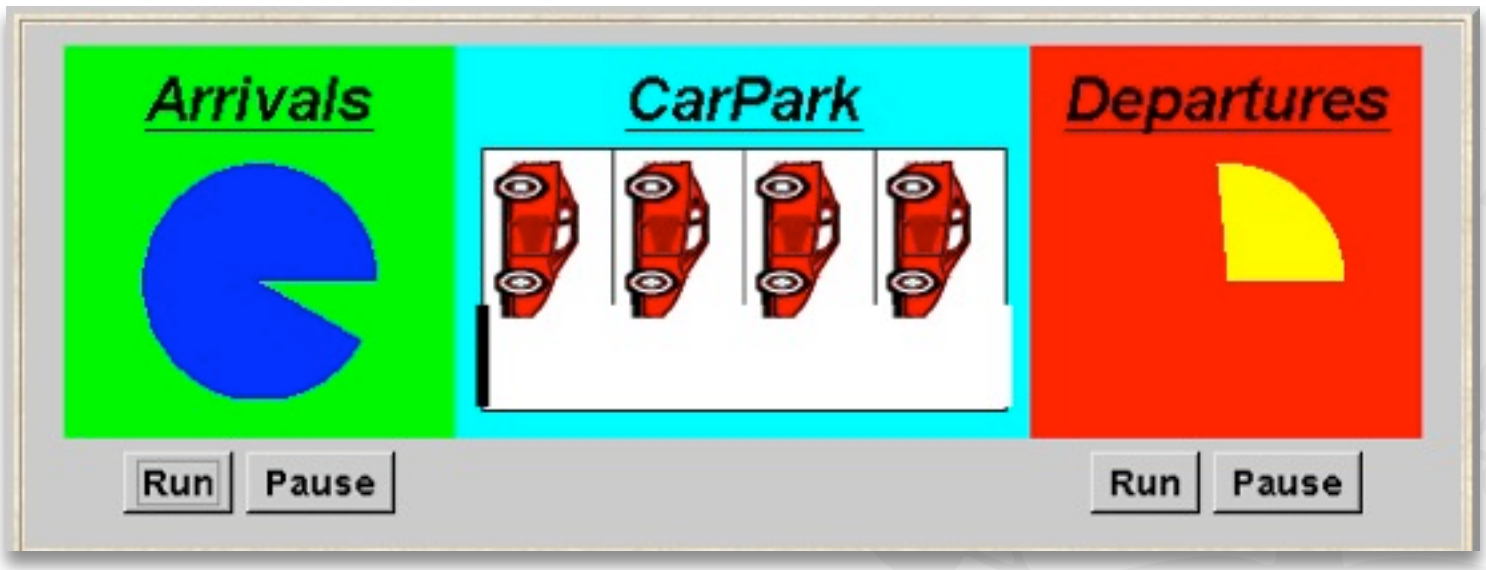
University of Southern Denmark



- Actions of interest:
  - arrive
  - depart

- Processes:

Car Park Model (Actions and Processes)

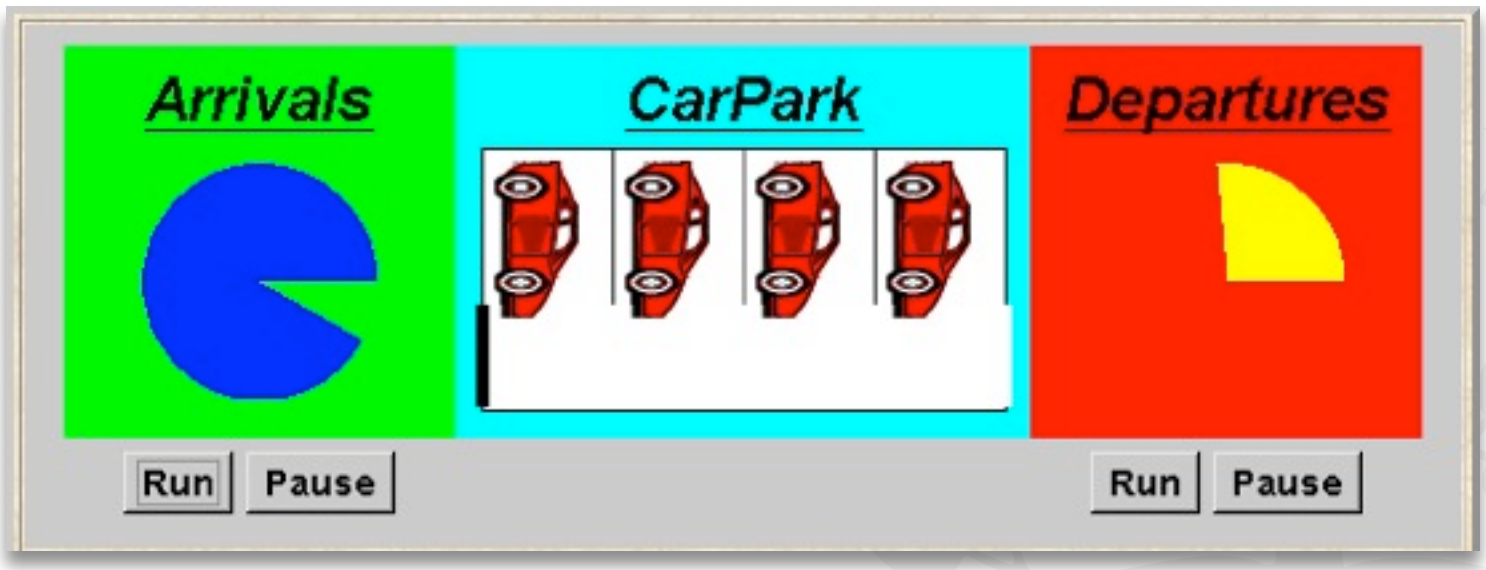


- Actions of interest:
  - `arrive`
  - `depart`

- Processes:
  - Arrivals

Car Park Model (Actions and Processes)

University of Southern Denmark

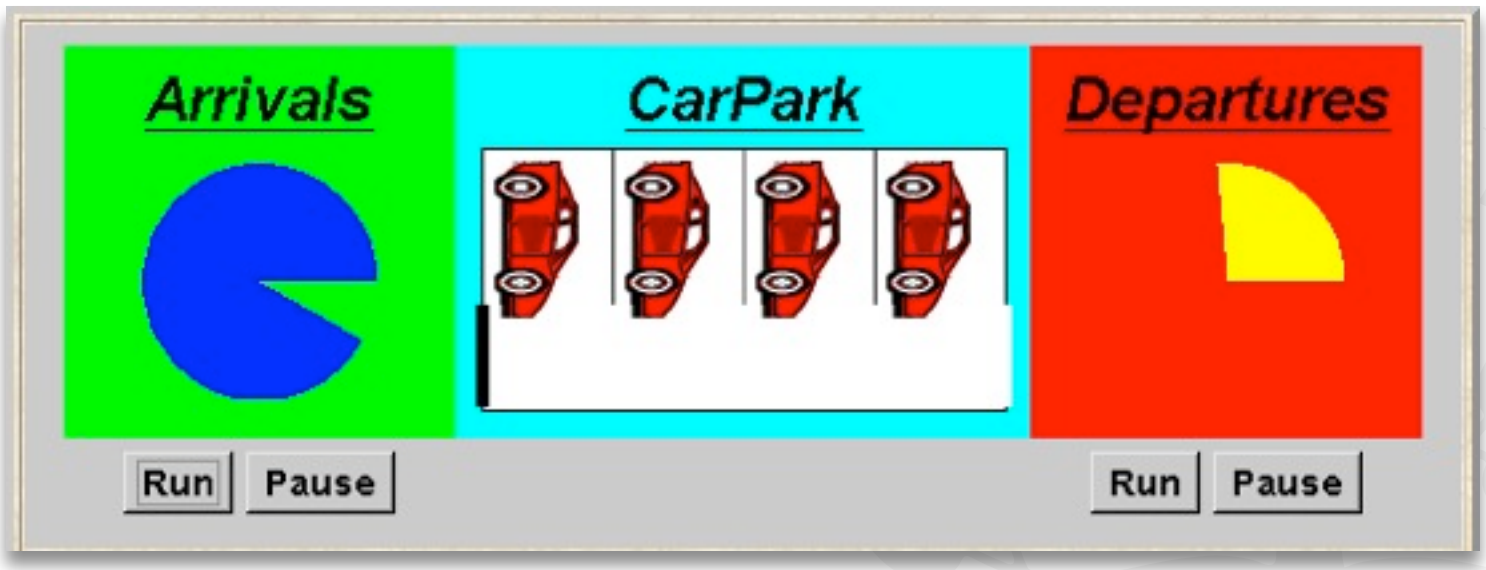


- Actions of interest:
  - `arrive`
  - `depart`

- Processes:
  - Arrivals
  - Departures

Car Park Model (Actions and Processes)

University of Southern Denmark



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

- Processes:
  - Arrivals
  - Departures
  - Carpark (Control)

Car Park Model (Structure Diagram)

University of Southern Denmark

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

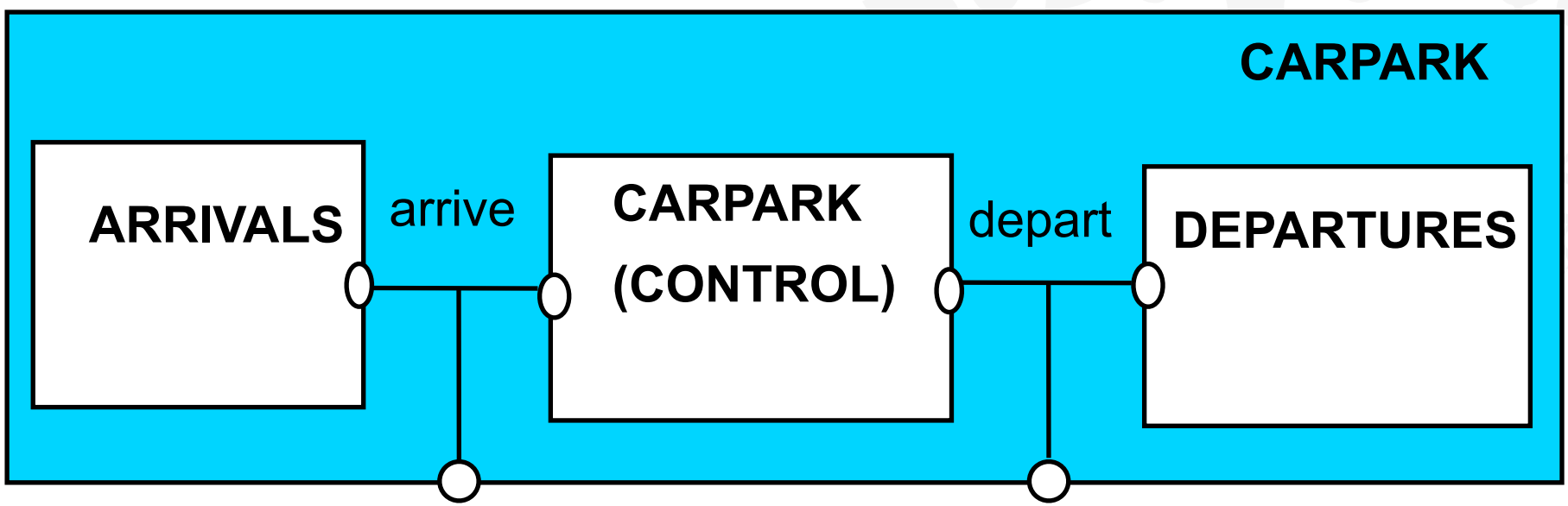
- Identify processes:
  - Arrivals
  - Departures
  - Carpark (Control)

Car Park Model (Structure Diagram)

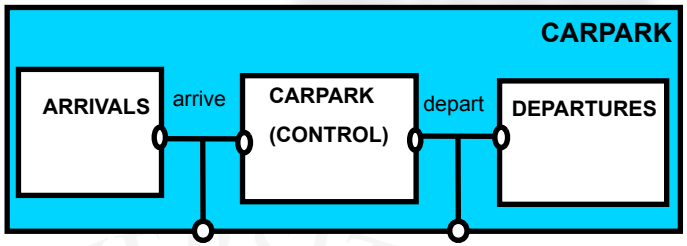
University of Southern Denmark

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

- Identify processes:
  - Arrivals
  - Departures
  - Carpark (Control)

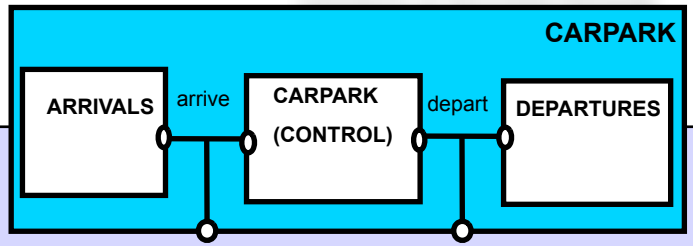


Car Park Model (FSP)



Car Park Model (FSP)

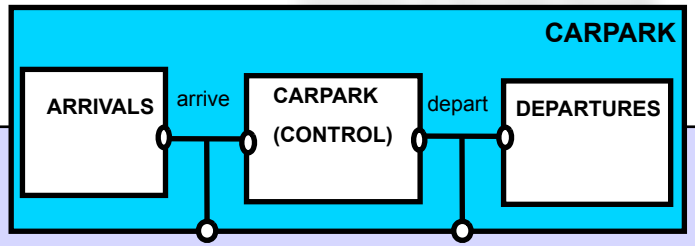
University of Southern Denmark




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

DEPARTURES = (depart -> DEPARTURES).
```

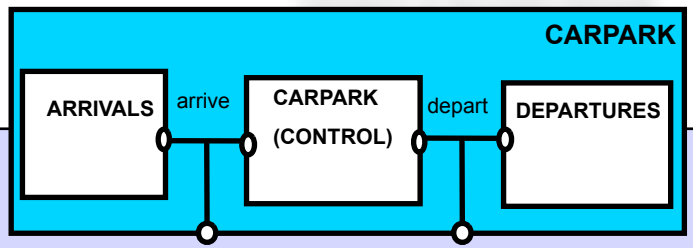
Car Park Model (FSP)

University of Southern Denmark



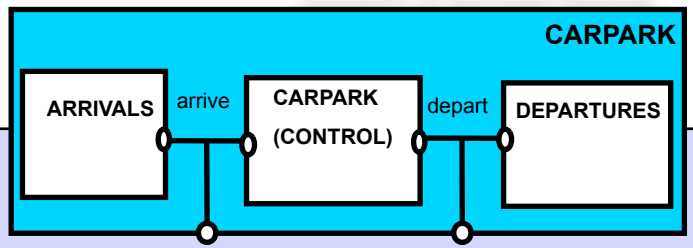

```
ARRIVALS = (arrive -> ARRIVALS).

DEPARTURES = (depart -> DEPARTURES).

CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when (spaces>0)          arrive -> SPACES[spaces-1]
      |when (spaces<CAPACITY) depart -> SPACES[spaces+1]).
```

Car Park Model (FSP)




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

DEPARTURES = (depart -> DEPARTURES).

CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when (spaces>0)          arrive -> SPACES[spaces-1]
      |when (spaces<CAPACITY) depart -> SPACES[spaces+1]).

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

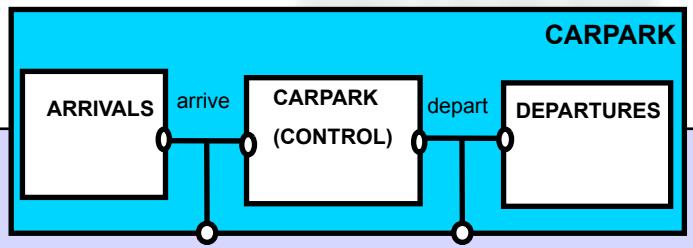
Car Park Model (FSP)




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

DEPARTURES = (depart -> DEPARTURES).

CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when (spaces>0)          arrive -> SPACES[spaces-1]
      |when (spaces<CAPACITY) depart -> SPACES[spaces+1]).

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

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

Car Park Model (FSP)

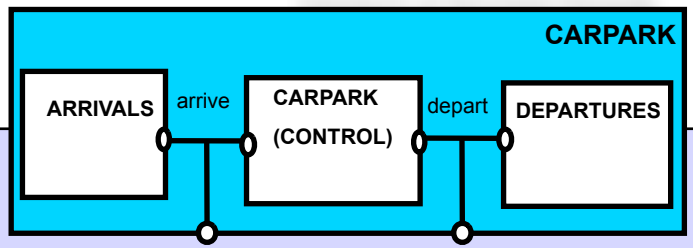



```
ARRIVALS = (arrive -> ARRIVALS).

DEPARTURES = (depart -> DEPARTURES).

CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when (spaces>0)          arrive -> SPACES[spaces-1]
      |when (spaces<CAPACITY) depart -> SPACES[spaces+1]).

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

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

LTS?

Car Park Model (FSP)




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

DEPARTURES = (depart -> DEPARTURES).

CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when (spaces>0)          arrive -> SPACES[spaces-1]
      |when (spaces<CAPACITY)  depart -> SPACES[spaces+1]).

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

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

LTS?

What if we **remove ARRIVALS and DEPARTURES?**

Car Park Program

Car Park Program

University of Southern Denmark

- Model:
  - all entities are processes interacting via **shared actions**

Car Park Program

University of Southern Denmark

- Model:
  - all entities are processes interacting via **shared actions**
- Implementation:

  we need to identify threads and monitors:
  - thread - **active** entity which initiates (output) actions
  - monitor - **passive** entity which responds to (input) actions.

Car Park Program

University of Southern Denmark

- Model:
  - all entities are processes interacting via **shared actions**
- Implementation:

  we 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:
- Departures:
- Control:

Car Park Program

University of Southern Denmark

- Model:
  - all entities are processes interacting via **shared actions**
- Implementation:

  we 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:
- Control:

Car Park Program

University of Southern Denmark

- Model:
  - all entities are processes interacting via **shared actions**
- Implementation:

  we 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:

Car Park Program

University of Southern Denmark

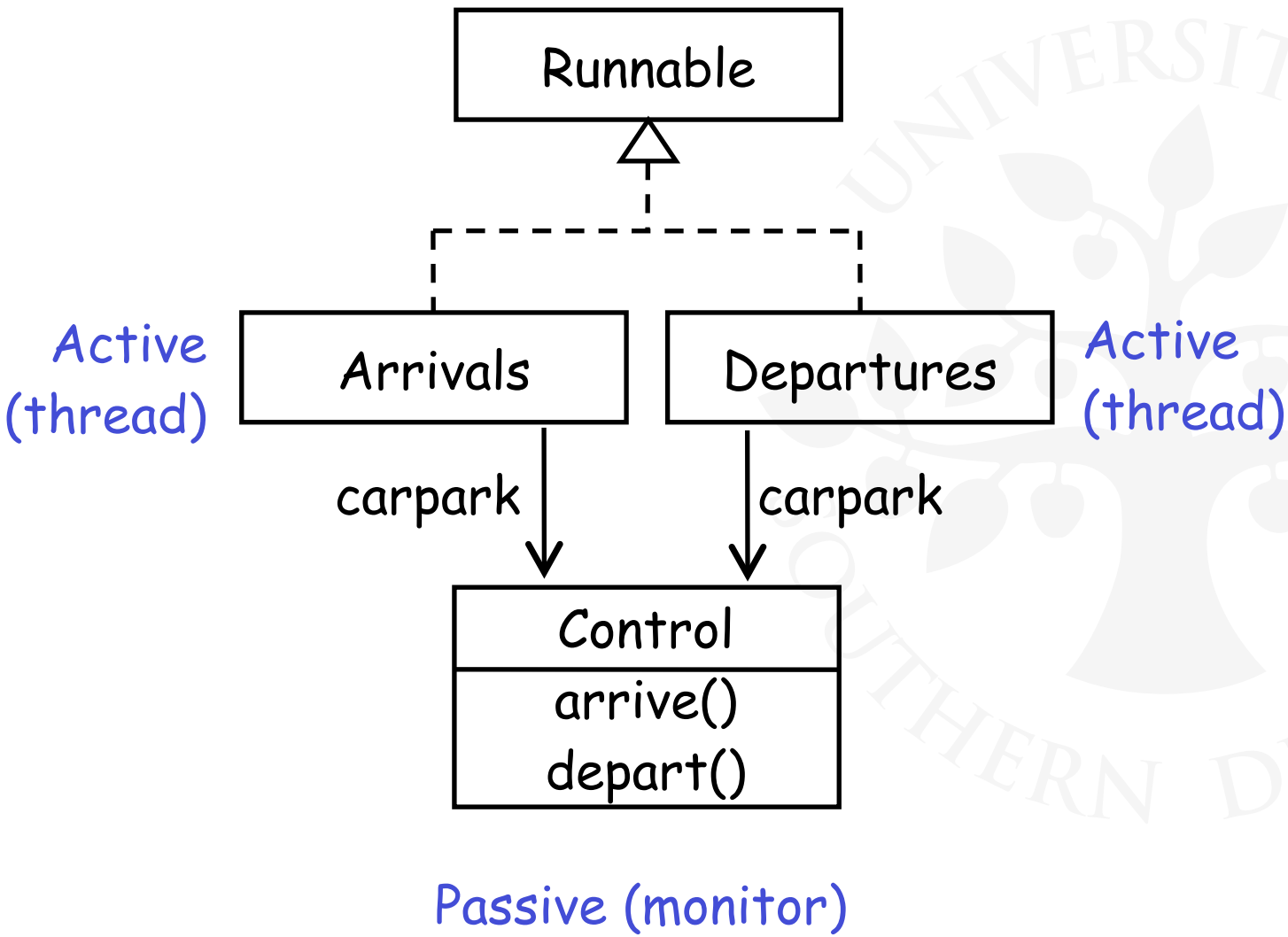
- Model:
  - all entities are processes interacting via **shared actions**
- Implementation:

  we 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)

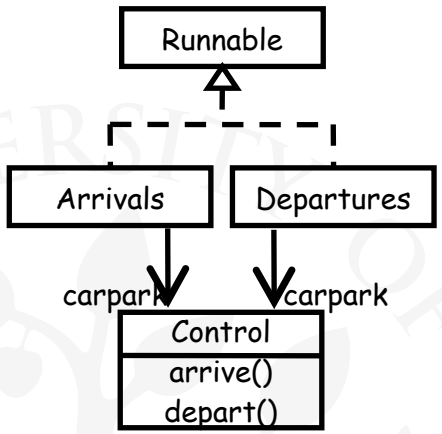


Car Park Program - `Main`

University of Southern Denmark

The main() method creates:

- **Control** monitor
- **Arrivals** thread
- **Departures** thread

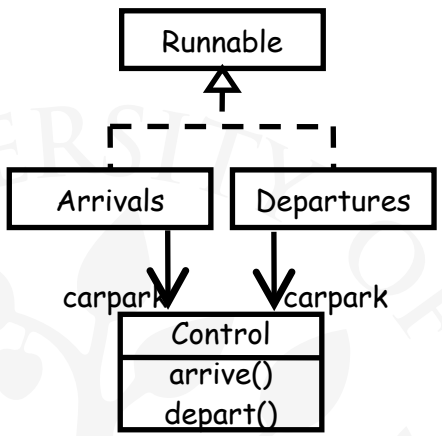


Car Park Program - `Main`

University of Southern Denmark

The main() method creates:

- **Control** monitor
- **Arrivals** thread
- **Departures** thread



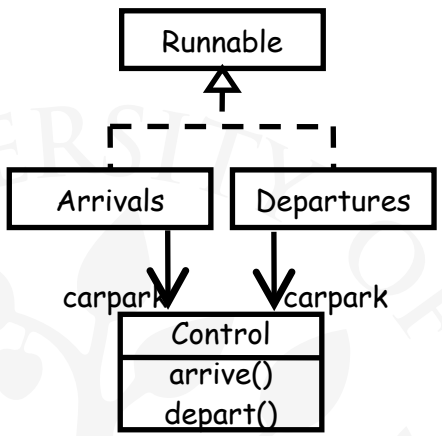

```
public static void main(String[] args) {
    Control c = new Control(CAPACITY);
    arrivals = new Thread(new Arrivals(c));
    departures = new Thread(new Departures(c));
    arrivals.start();
    departures.start();
}
```

Car Park Program - `Main`

University of Southern Denmark

The main() method creates:

- **Control** monitor
- **Arrivals** thread
- **Departures** thread




```
public static void main(String[] args) {
    Control c = new Control(CAPACITY);
    arrivals = new Thread(new Arrivals(c));
    departures = new Thread(new Departures(c));
    arrivals.start();
    departures.start();
}
```

The **Control** is shared by the **Arrivals** and **Departures** threads

Car Park Program - `Arrivals`

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

```
class Arrivals implements Runnable {
    Control carpark;

    Arrivals(Control c) { carpark = c; }

    public void run() {
        try {
            while(true) {
                Thread.sleep(...);
                carpark.arrive();
            }
        } catch (InterruptedException _) {}
    }
}
```

... similar for Departures (calling *carpark.depart()*)

Car Park Program - Arrivals

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

```
class Arrivals implements Runnable {
    Control carpark;

    Arrivals(Control c) { carpark = c; }

    public void run() {
        try {
            while(true) {
                Thread.sleep(...);
                carpark.arrive();
            }
        } catch (InterruptedException
    }
}
```

Would like to somehow block Arrivals thread here…

… similar for Departures (calling *carpark.depart()*)

Car Park Program - Arrivals

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

```
class Arrivals implements Runnable {
    Control carpark;

    Arrivals(Control c) { carpark = c; }

    public void run() {
        try {
            while(true) {
                Thread.sleep(...);
                carpark.arrive();
            }
        } catch (InterruptedException e) {}
    }
}
```

Would like to somehow block Arrivals thread here…

… similar for Departures (calling *carpark.depart()*)

Where should we do the "blocking"?

Car Park Program - Arrivals

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

```
class Arrivals implements Runnable {
    Control carpark;

    Arrivals(Control c) { carpark = c; }

    public void run() {
        try {
            while(true) {
                Thread.sleep(...);
                carpark.arrive();
            }
        } catch (InterruptedException e) {}
    }
}
```

Would like to somehow block Arrivals thread here…

… similar for Departures (calling *carpark.depart()*)

Where should we do the "blocking"?

How do we implement the Carpark **Controller**'s control?

Control Monitor

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when(spaces>0)         arrive -> SPACES[spaces-1]
      |when(spaces<CAPACITY) depart -> SPACES[spaces+1]).
```

```
class Control {
                static final int CAPACITY;
                int spaces;

    Control(int n) {
        CAPACITY = spaces = n;
    }

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

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

Control Monitor

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when(spaces>0)         arrive -> SPACES[spaces-1]
      |when(spaces<CAPACITY)  depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    Control(int n) {
        CAPACITY = spaces = n;
    }

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

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

Encapsulation

~ protected

Control Monitor

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when(spaces>0)         arrive -> SPACES[spaces-1]
      |when(spaces<CAPACITY) depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    Control(int n) {
        CAPACITY = spaces = n;
    }

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

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

Encapsulation
~ protected

Mutual exclusion ~
synchronized

Control Monitor

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when(spaces>0)         arrive -> SPACES[spaces-1]
      |when(spaces<CAPACITY)  depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    Control(int n) {
        CAPACITY = spaces = n;
    }

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

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

Encapsulation
~ protected

Mutual exclusion ~ synchronized

Condition synchronisation:

Control Monitor

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when(spaces>0)         arrive -> SPACES[spaces-1]
      |when(spaces<CAPACITY)  depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    Control(int n) {
        CAPACITY = spaces = n;
    }

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

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

Encapsulation
~ protected

Mutual exclusion ~ synchronized

Condition synchronisation:

Block, if full?
¬(spaces>0)

Control Monitor

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
      (when(spaces>0)         arrive -> SPACES[spaces-1]
      |when(spaces<CAPACITY)  depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    Control(int n) {
        CAPACITY = spaces = n;
    }

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

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

Encapsulation
~ protected

Mutual exclusion ~ synchronized

Condition synchronisation:

Block, if full?
¬(spaces>0)

Block, if empty?
¬(spaces < CAPACITY)

Condition Synchronisation in Java

University of Southern Denmark

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

Condition Synchronisation in Java

University of Southern Denmark

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

`Object` has the following methods:

Condition Synchronisation in Java

University of Southern Denmark

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

`Object` has the following methods:

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

Condition Synchronisation in Java

University of Southern Denmark

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

`Object` has the following methods:

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

Waits to be notified ;
Releases the synchronisation lock associated with the object.

When notified, the thread must reacquire the synchronisation lock.

Condition Synchronisation in Java

University of Southern Denmark

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

`Object` has the following methods:

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

Waits to be notified ;
Releases the synchronisation lock associated with the object.

When notified, the thread must reacquire the synchronisation lock.

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

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

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

Condition Synchronisation in Java (enter/exit)

Condition Synchronisation 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.

Condition Synchronisation in Java (enter/exit)

University of Southern Denmark

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

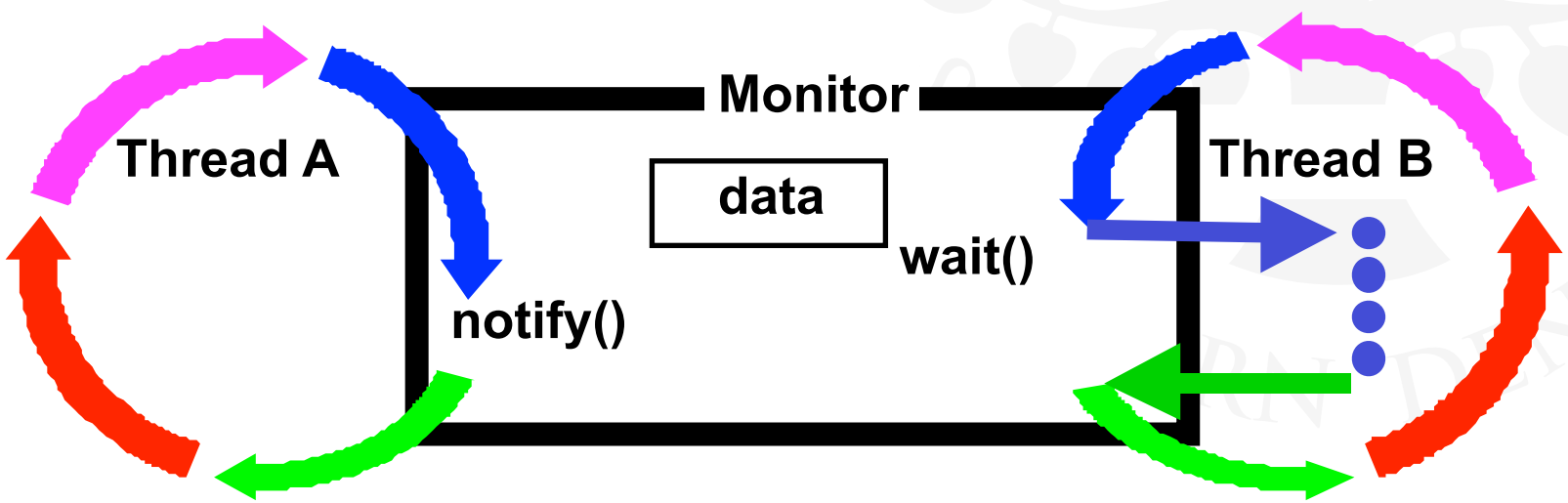
Condition Synchronisation in Java (enter/exit)

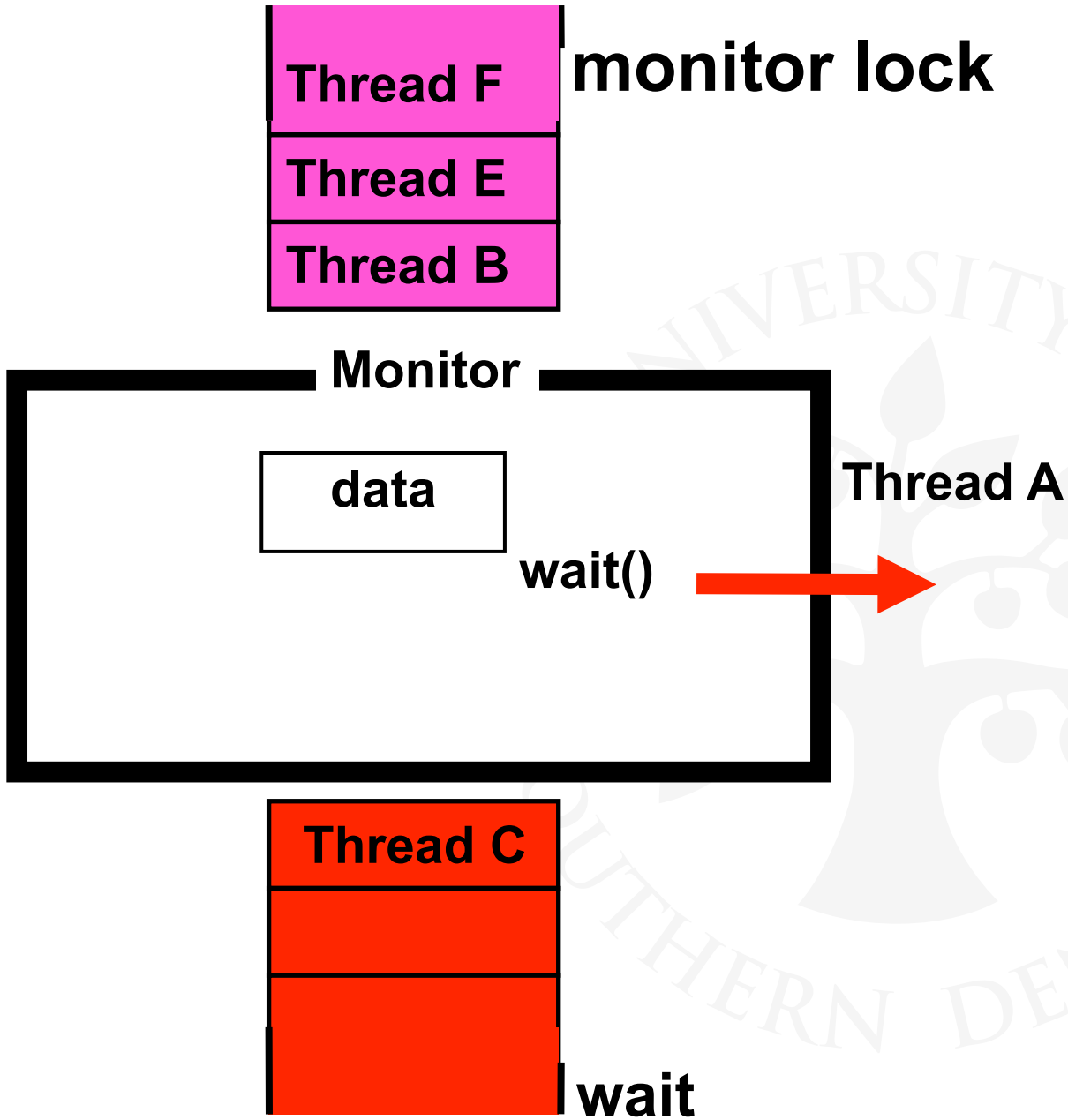
University of Southern Denmark

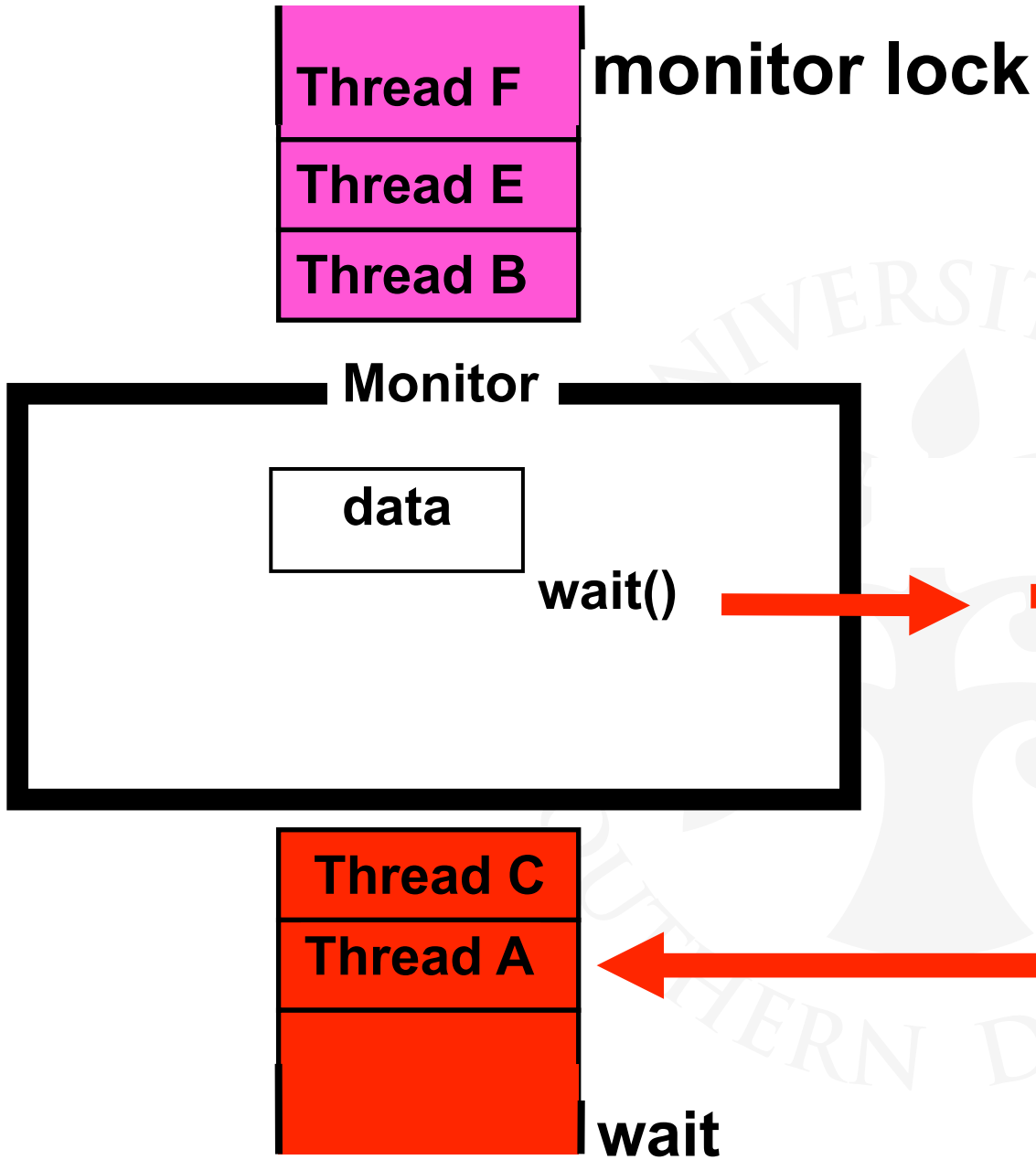
A thread:

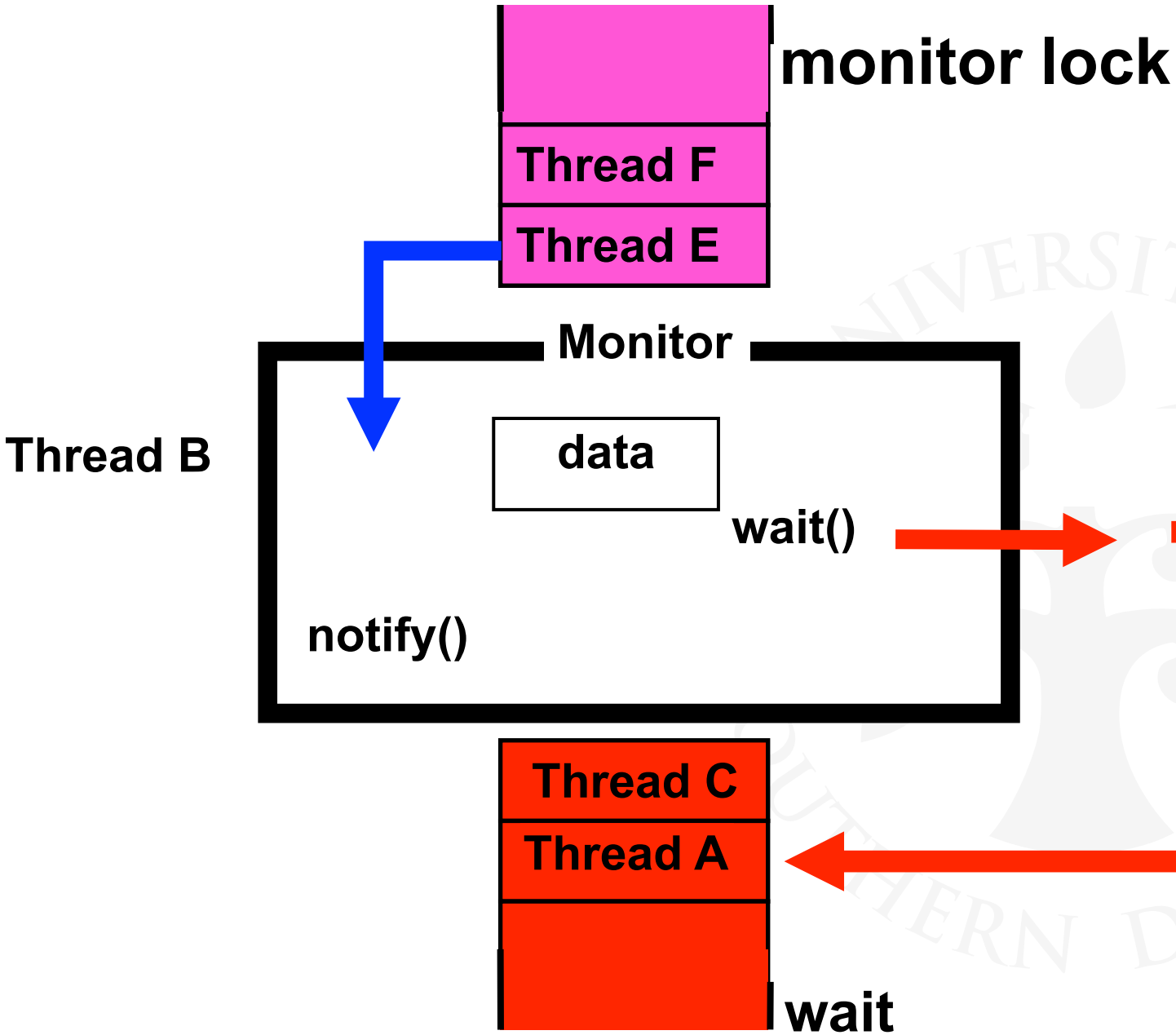
- **Enters** a monitor when a thread acquires the lock associated with the monitor;
- **Exits** a monitor when it releases the lock.

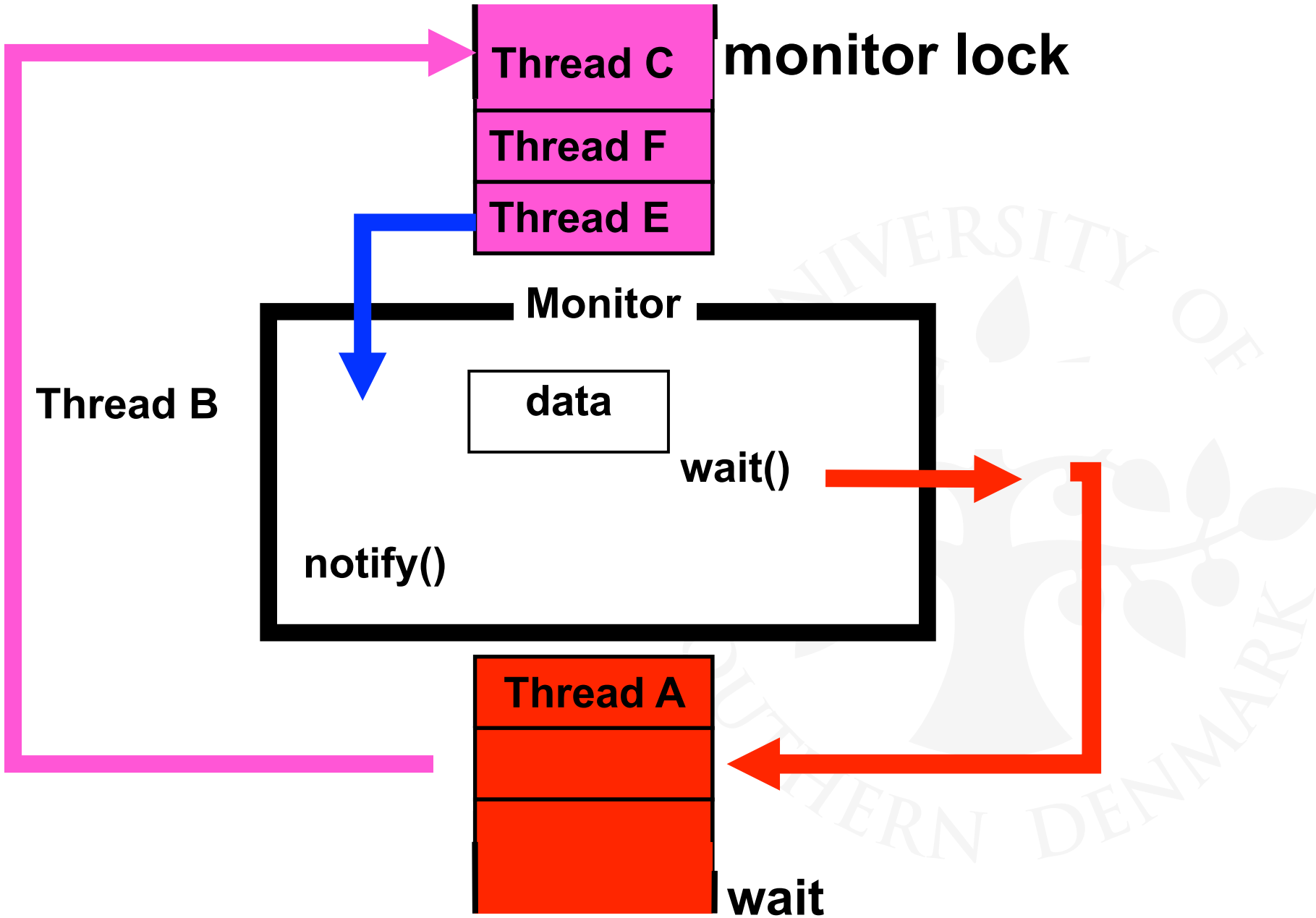
Wait() causes the thread to **exit** the monitor, permitting other threads to **enter** the monitor




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


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


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


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

Condition Synchronisation in FSP and Java

University of Southern Denmark

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

Condition Synchronisation in FSP and Java

University of Southern Denmark

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

```
synchronized void action() throws Int'Exc' {

}
```

Condition Synchronisation in FSP and Java

University of Southern Denmark

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

```
synchronized void action() throws Int'Exc' {
        if (!cond) wait();
```

Condition Synchronisation in FSP and Java

University of Southern Denmark

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

```
synchronized void action() throws Int'Exc' {
        if (!cond) wait();
    // modify monitor data
    notifyAll();
}
```

Condition Synchronisation in FSP and Java

University of Southern Denmark

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

```
synchronized void action() throws Int'Exc' {
    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.

Condition Synchronisation in FSP and Java

University of Southern Denmark

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

```
synchronized void action() throws Int'Exc' {
    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 Synchronisation


```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
              (when(spaces>0)          arrive -> SPACES[spaces-1]
              |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

CarParkControl - Condition Synchronisation


```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
              (when(spaces>0)          arrive -> SPACES[spaces-1]
              |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;
```

CarParkControl - Condition Synchronisation


```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
              (when(spaces>0)          arrive -> SPACES[spaces-1]
              |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    synchronized void arrive() throws Int'Exc' {
        while (!(spaces>0)) wait();
        --spaces;
        notifyAll();
    }
```

CarParkControl - Condition Synchronisation

University of Southern Denmark

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
              (when(spaces>0)          arrive -> SPACES[spaces-1]
              |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    synchronized void arrive() throws Int'Exc' {
        while (!(spaces>0)) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart() throws Int'Exc' {
        while (!(spaces<CAPACITY)) wait();
        ++spaces;
        notifyAll();
    }
}
```

CarParkControl - Condition Synchronisation

University of Southern Denmark

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
              (when(spaces>0)          arrive -> SPACES[spaces-1]
              |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

```
class Control {
    protected static final int CAPACITY;
    protected int spaces;

    synchronized void arrive() throws Int'Exc' {
        while (!(spaces>0)) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart() throws Int'Exc' {
        while (!(spaces<CAPACITY)) wait();
        ++spaces;
        notifyAll();
    }
}
```

Would it be sensible here to use `notify()` rather than `notifyAll()`?

More about Object.notify() and Object.notifyAll()

University of Southern Denmark

More about Object.notify() and Object.notifyAll()

University of Southern Denmark

notify() can be used instead of notifyAll() only when both of these conditions hold:

More about Object.notify() and Object.notifyAll()

University of Southern Denmark

notify() can be used instead of notifyAll() only when both of these conditions hold:

Uniform waiters. Only one condition predicate and each thread executes the same logic upon returning from wait(); and

More about Object.notify() and Object.notifyAll()

University of Southern Denmark

notify() can be used instead of notifyAll() only when both of these conditions hold:

Uniform waiters. Only one condition predicate and each thread executes the same logic upon returning from wait(); and

One-in, one-out. A notification enables at most one thread to proceed.

More about Object.notify() and Object.notifyAll()

University of Southern Denmark

notify() can be used instead of notifyAll() only when both of these conditions hold:

Uniform waiters. Only one condition predicate and each thread executes the same logic upon returning from wait(); and

One-in, one-out. A notification enables at most one thread to proceed.

Prevailing wisdom: use notifyAll() in preference to single notify() when you are not sure.

Models to Monitors - Guidelines

University of Southern Denmark

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

Models to Monitors - Guidelines

University of Southern Denmark

- **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.

Models to Monitors - Guidelines

University of Southern Denmark

- **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.

Models to Monitors - Guidelines

University of Southern Denmark

- **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 signalled to waiting threads using **notifyAll()** (or **notify()**).

Semaphores

5.2 Semaphores

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

5.2 Semaphores

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

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

5.2 Semaphores

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

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

s.down(): **when** (s>0) **do** decrement(s);

5.2 Semaphores

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

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

s.down(): **when** (s>0) **do** decrement(s);

s.up(): increment(s);

5.2 Semaphores

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

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

s.down():	**when** (s>0) **do** decrement(s);

s.up():	increment(s);

Usually implemented as blocking wait:

5.2 Semaphores

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

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

s.down(): **when** (s>0) **do** decrement(s);

s.up(): increment(s);

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);

5.2 Semaphores

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

Semaphore **s** : integer var that can take only non-negative 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

University of Southern Denmark

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

Modelling Semaphores

University of Southern Denmark

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

```
const Max = 3
```

Modelling Semaphores

University of Southern Denmark

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

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

Modelling Semaphores

University of Southern Denmark

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

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

SEMAPHORE(N=0) = SEMA[N], // N initial value
```

Modelling Semaphores

University of Southern Denmark

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

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

SEMAPHORE(N=0) = SEMA[N], // N initial value
SEMA[v:Int]    = (up->SEMA[v+1]
```

Modelling Semaphores

University of Southern Denmark

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

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

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

Modelling Semaphores

University of Southern Denmark

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

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

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

Modelling Semaphores

University of Southern Denmark

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

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

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

LTS?

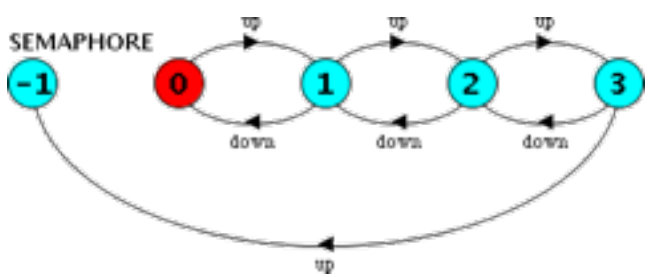
Modelling Semaphores

University of Southern Denmark

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

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

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

LTS?



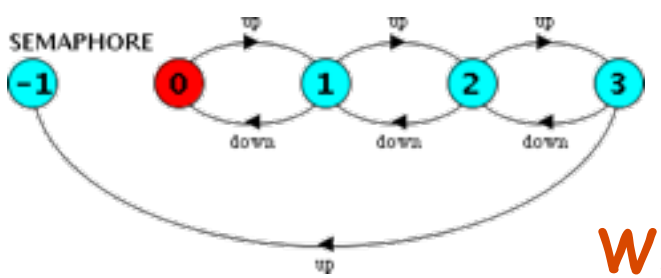
Modelling Semaphores

University of Southern Denmark

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

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

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

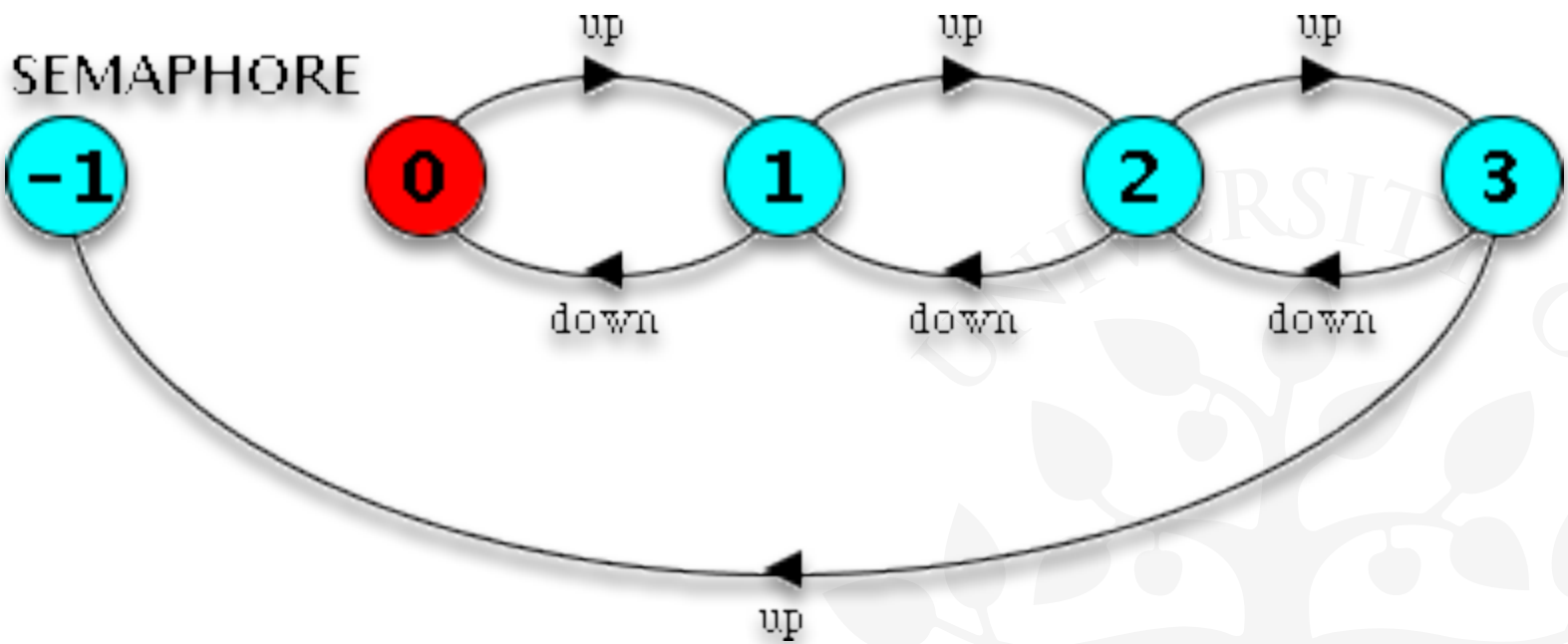
LTS?



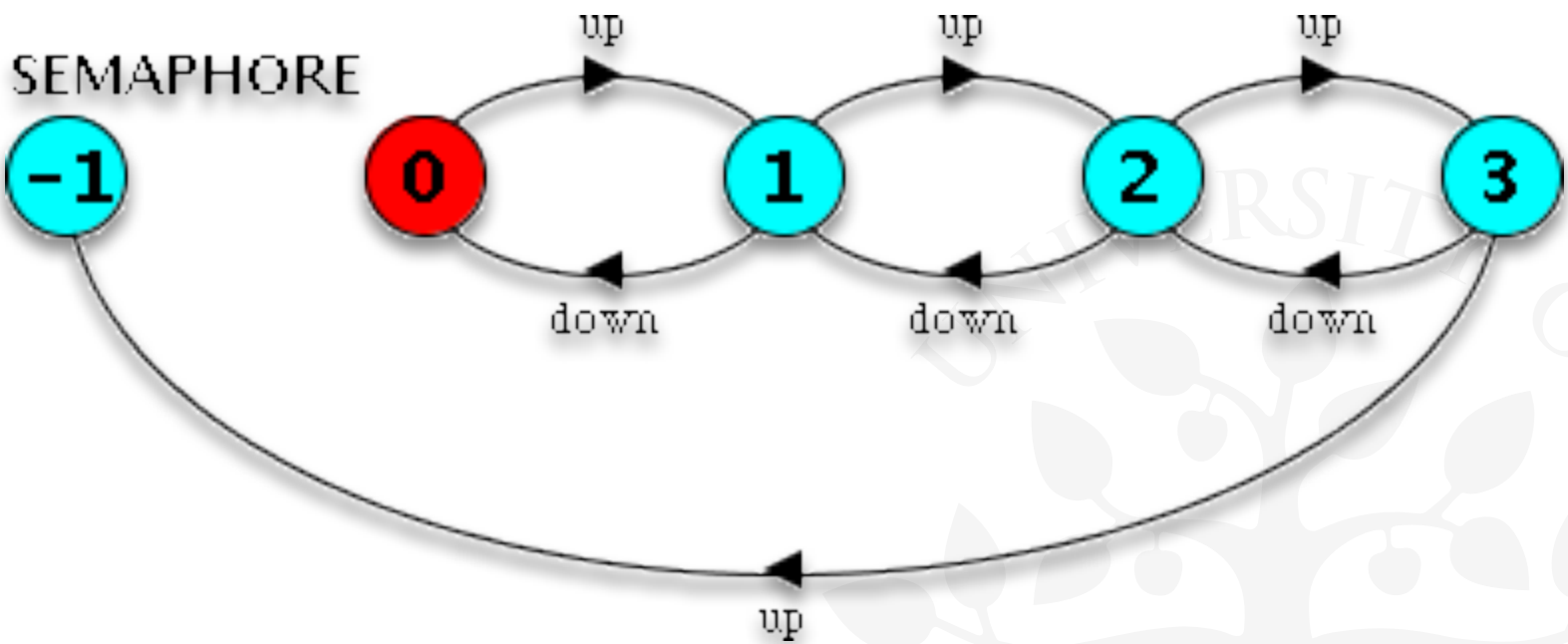
What if we omit the last line above?

Modelling Semaphores

University of Southern Denmark

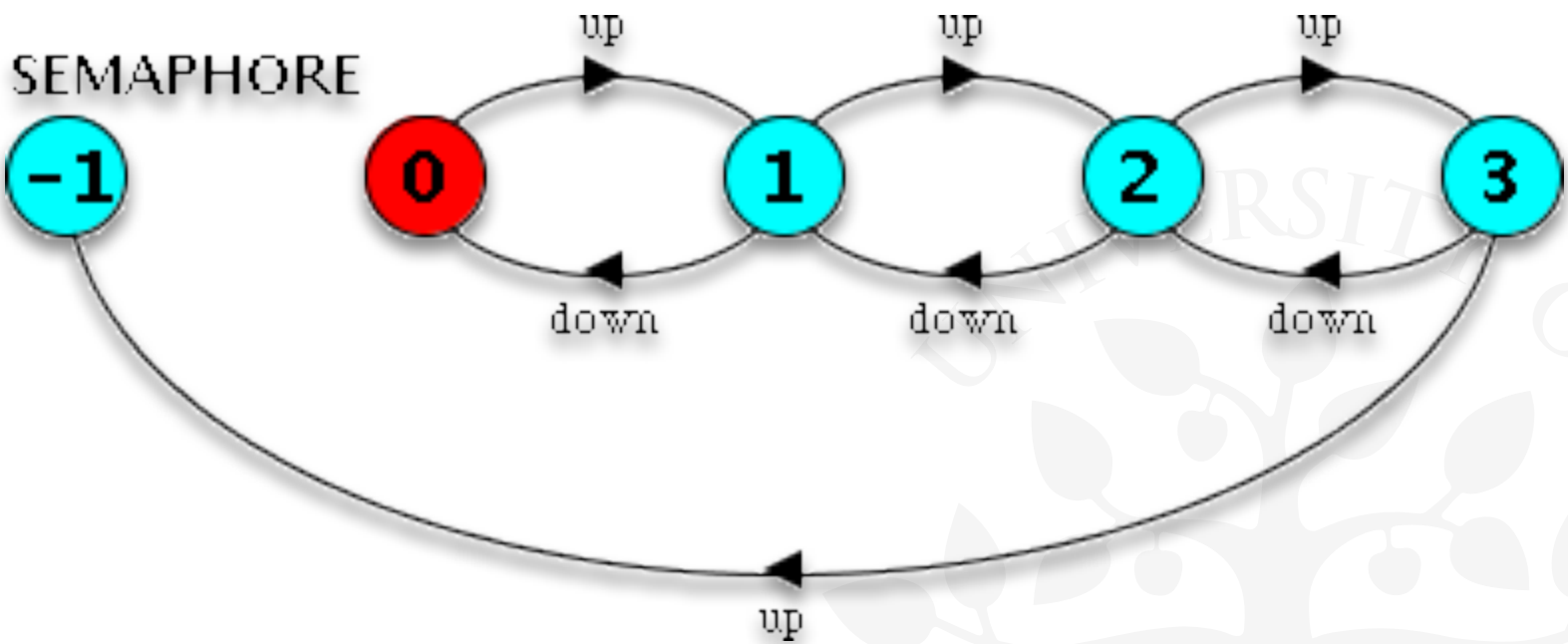


Modelling Semaphores



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

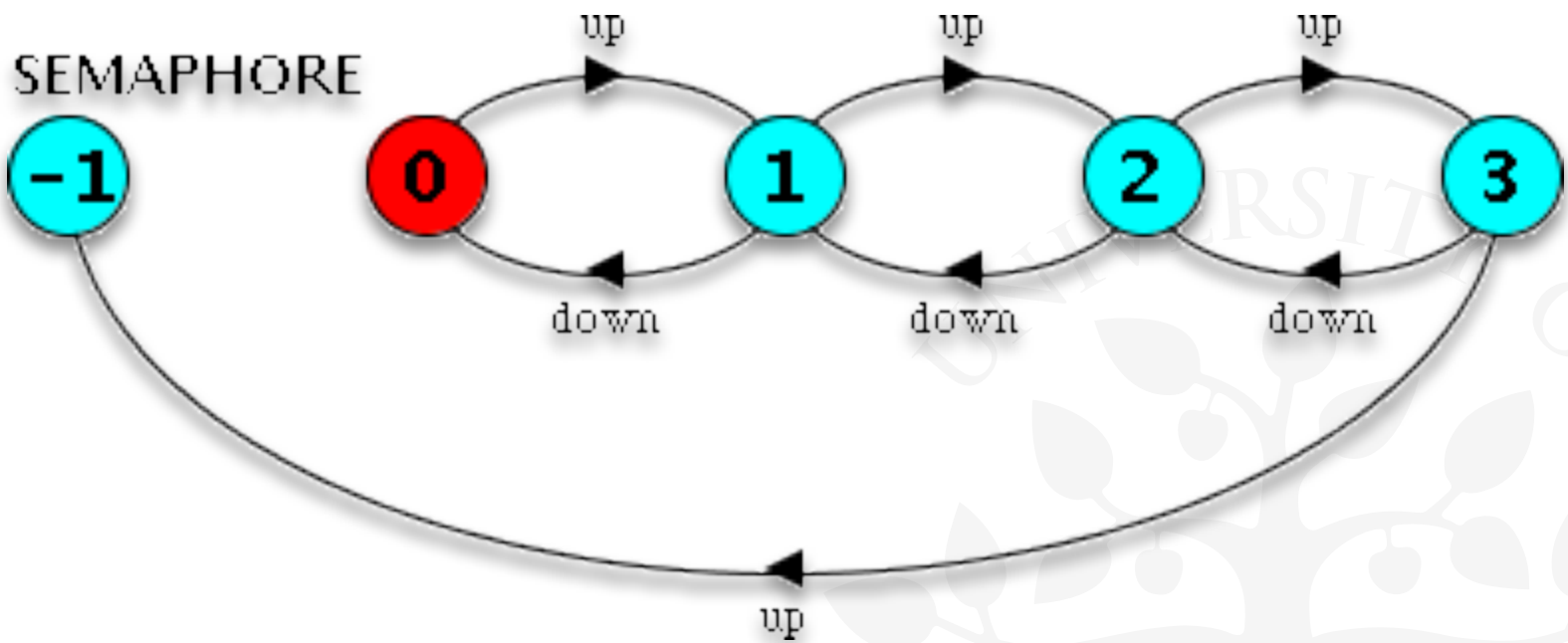
Modelling Semaphores



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

Action up is not guarded.

Modelling Semaphores

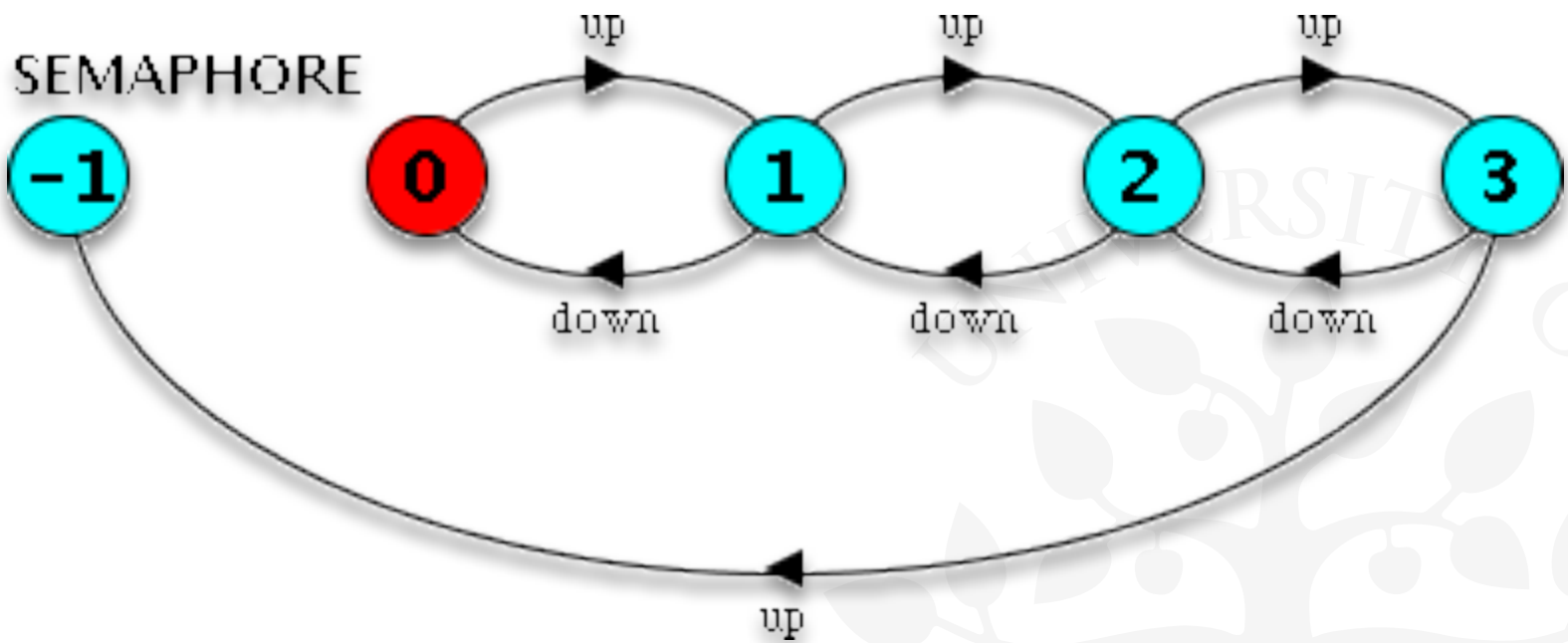


Action down is only accepted when value (v) of the semaphore is greater than 0.

Action up is not guarded.

Trace to a violation:

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

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

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

Semaphore Demo - Model

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

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).
```

Semaphore Demo - Model

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

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
```

Semaphore Demo - Model

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

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)).
```

Semaphore Demo - Model

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

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?**

Semaphore Demo - Model

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

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**?

Semaphore Demo - Model

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

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**) ?

Semaphore Demo - Model

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

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)).
```

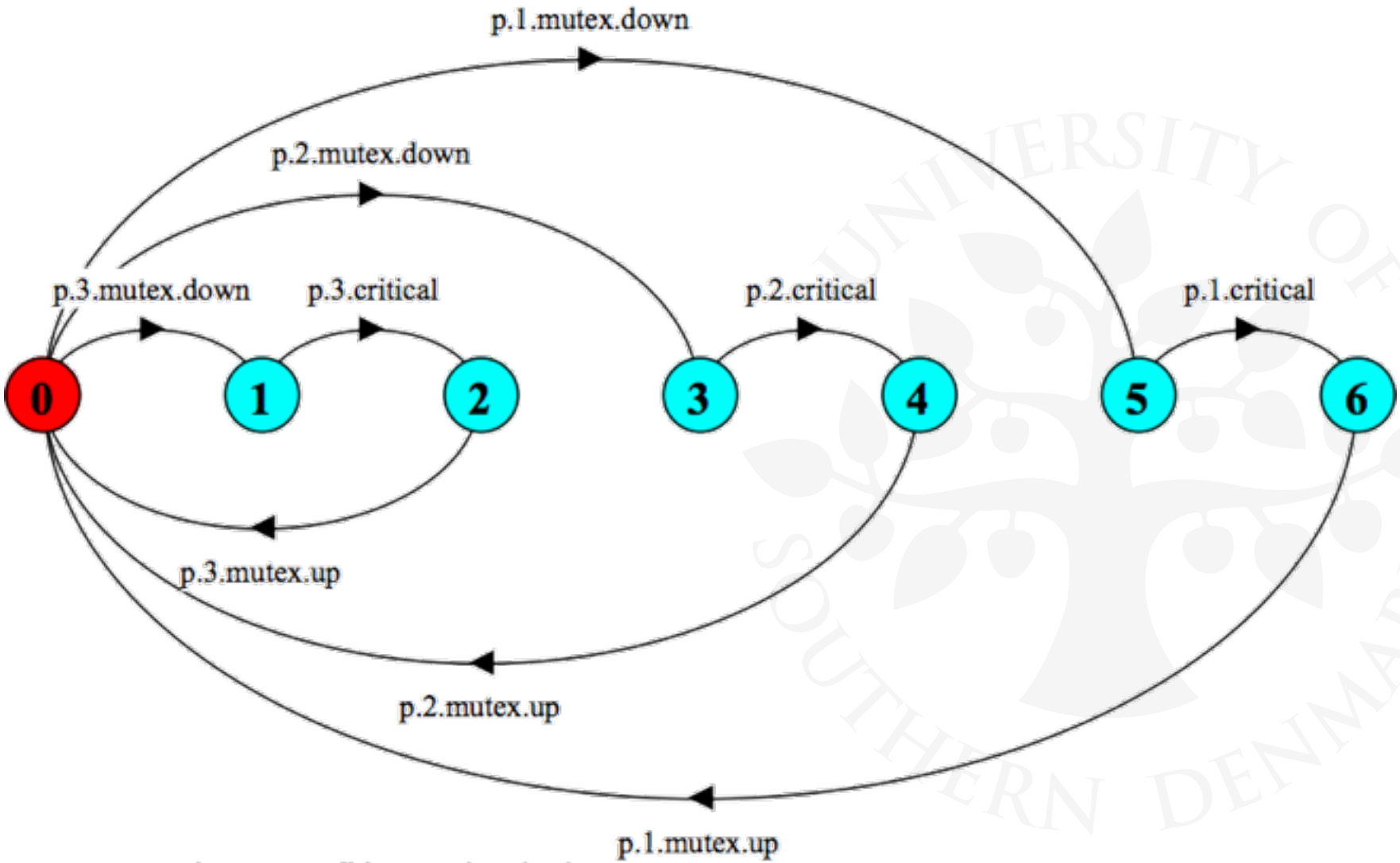
For mutual exclusion, the semaphore initial value is 1. Why?

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

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

LTS?

Semaphore Demo - Model



Semaphores in Java

University of Southern Denmark

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

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;
```

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

    synchronized public void down() throws Int'Exc' {
```

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

    synchronized public void down() throws Int'Exc' {
        while (!(value > 0)) wait();
```

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

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

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

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

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

    synchronized public void down() throws Int'Exc' {
        while (!(value > 0)) wait();
        --value;
        notifyAll();
    }

    synchronized public void up() {
```

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

    synchronized public void down() throws Int'Exc' {
        while (!(value > 0)) wait();
        --value;
        notifyAll();
    }

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

Semaphores in Java

University of Southern Denmark

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

```
public class Semaphore {
    protected int value;

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

    synchronized public void down() throws Int'Exc' {
        while (!(value > 0)) wait();
        --value;
        notifyAll();
    }

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

Do we need `notifyAll()` here?

Semaphores in Java

University of Southern Denmark

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

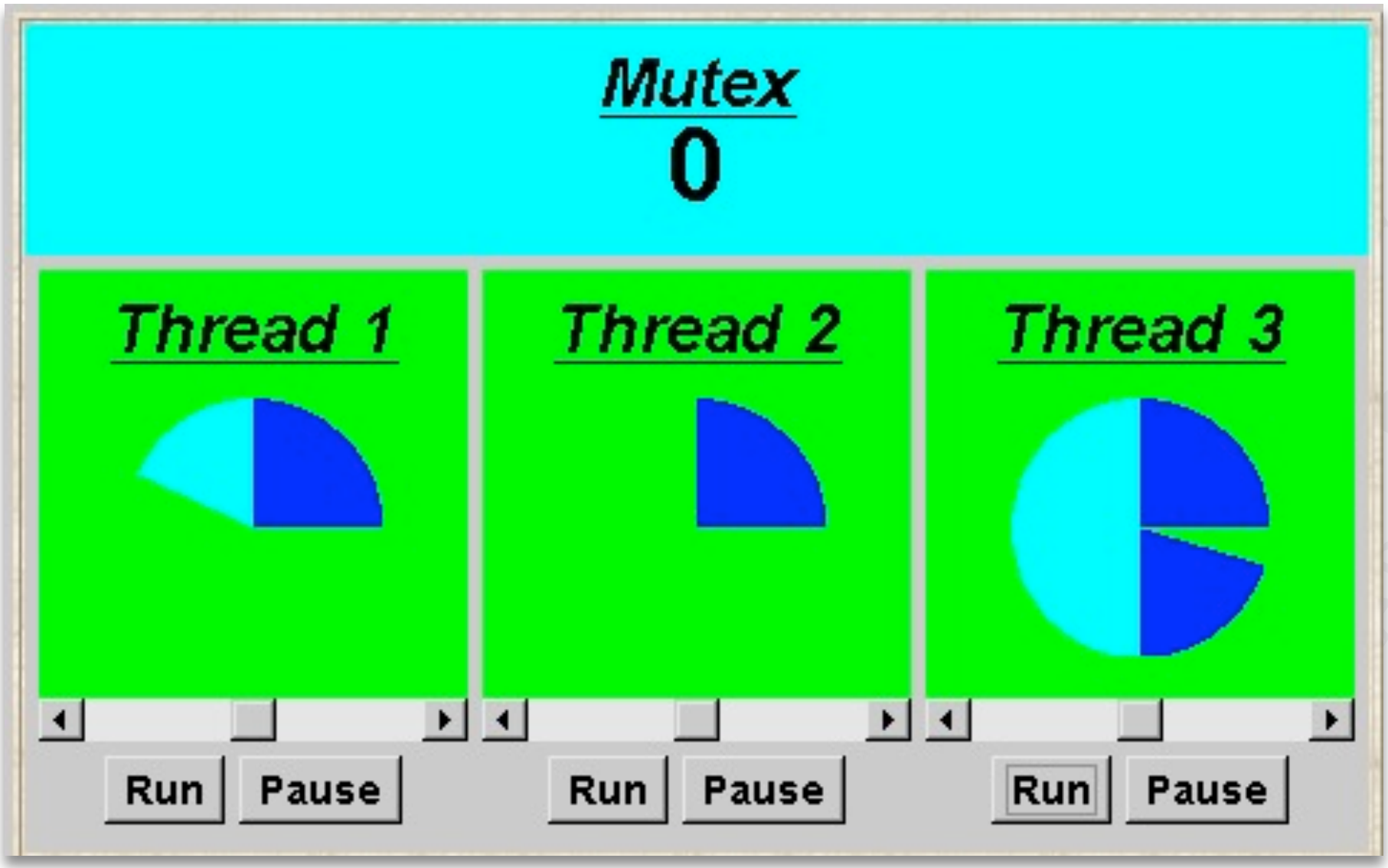
```
public class Semaphore {
    protected int value;

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

    synchronized public void down() throws Int'Exc' {
        while (!(value > 0)) wait();
        --value;
        notifyAll();
    }

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

Do we need `notifyAll()` here?

...what about here?

SEMADEMO Display

University of Southern Denmark



SEMADEMO Program - MutexLoop

University of Southern Denmark

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

```
class MutexLoop implements Runnable {
    Semaphore mutex; // shared semaphore

    MutexLoop (Semaphore sem) { mutex=sem; }

    public void run() {
        try {
            while(true)  {
                // non-critical actions
                mutex.down();              // acquire
                // critical actions
                mutex.up();                // release
            }
        } catch(InterruptedException _) {}
    }
}
```

SEMADEMO Program - MutexLoop

University of Southern Denmark

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

```
class MutexLoop implements Runnable {
    Semaphore mutex; // shared semaphore

    MutexLoop (Semaphore sem) { mutex=sem; }

    public void run() {
        try {
            while(true)  {
                // non-critical actions
                mutex.down();              // acquire
                // critical actions
                mutex.up();                // release
            }
        } catch(InterruptedException _) {}
    }
}
```

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

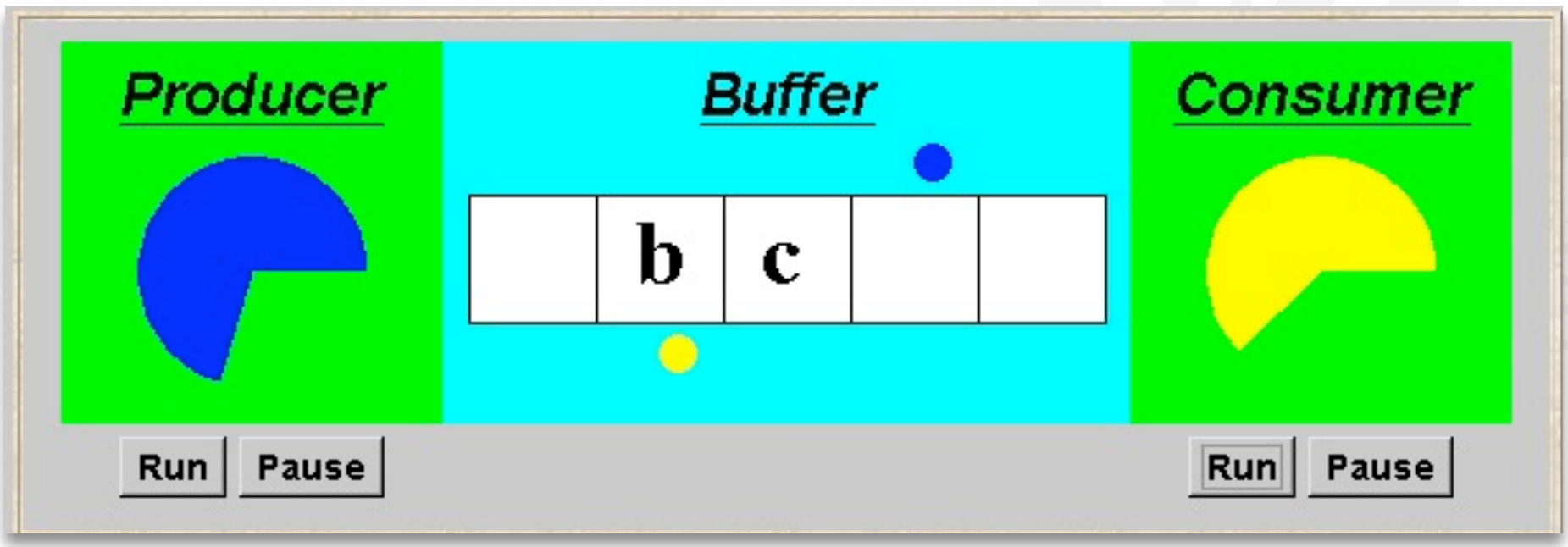
Producer / Consumer

5.3 Producer / Consumer

University of Southern Denmark

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:

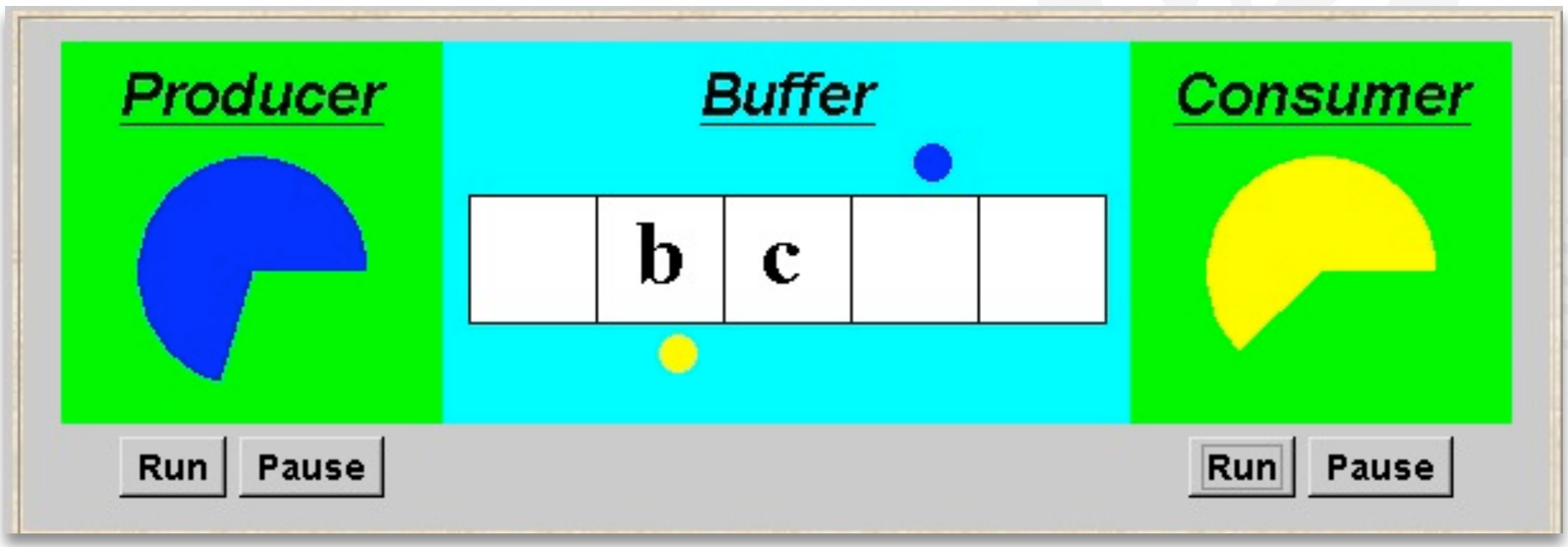


5.3 Producer / Consumer

University of Southern Denmark

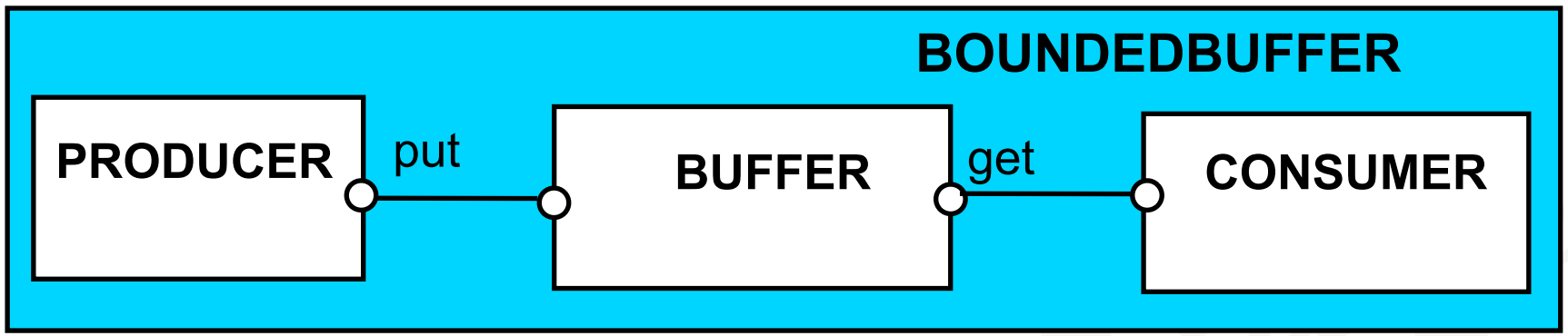
A **bounded buffer** consists of a fixed number of slots.

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

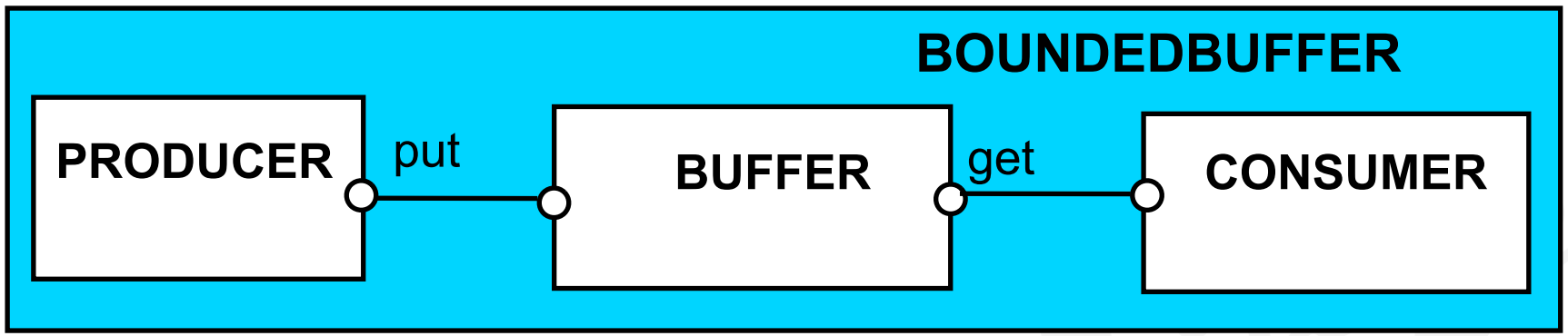


≈ Car Park Example!

Producer / Consumer
- a Data-Independent Model

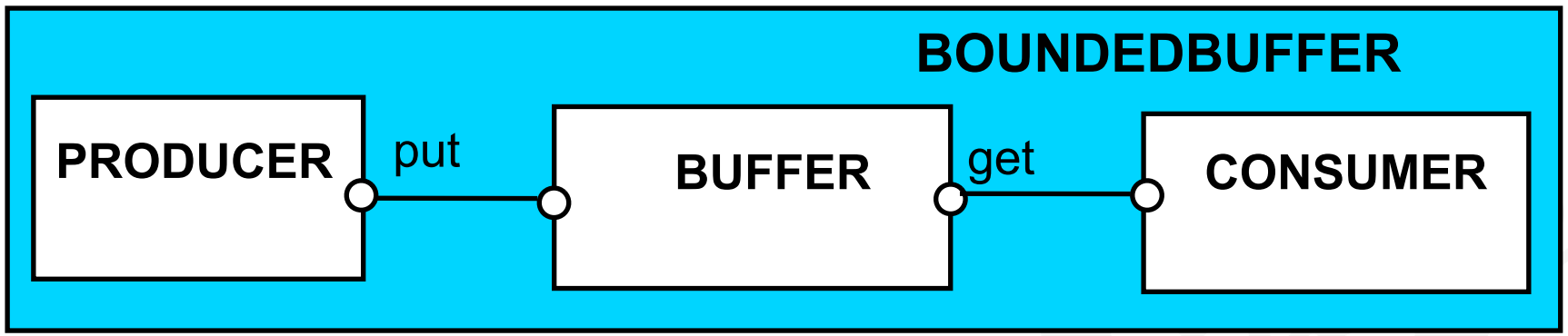


Producer / Consumer
- a Data-Independent Model



The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner (i.e., we abstract away the letters).

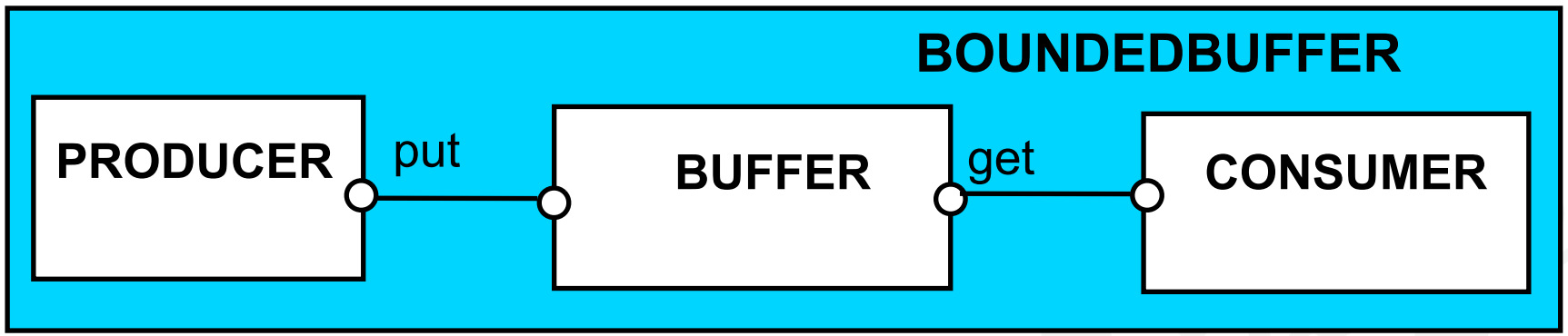
Producer / Consumer
- a Data-Independent Model



The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner (i.e., we abstract away the letters).

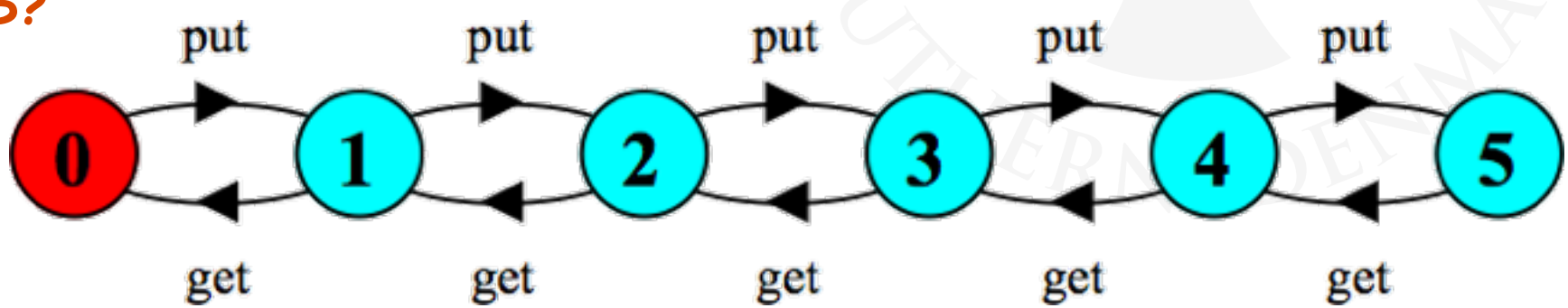
LTS?

Producer / Consumer
- a Data-Independent Model

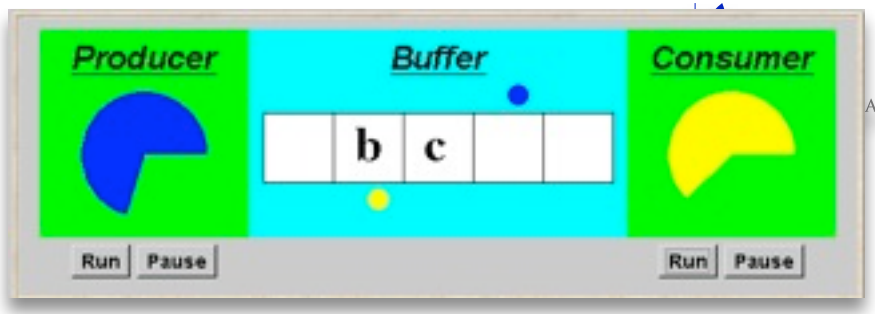


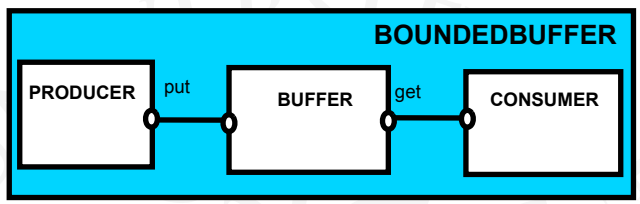
The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner (i.e., we abstract away the letters).

LTS?

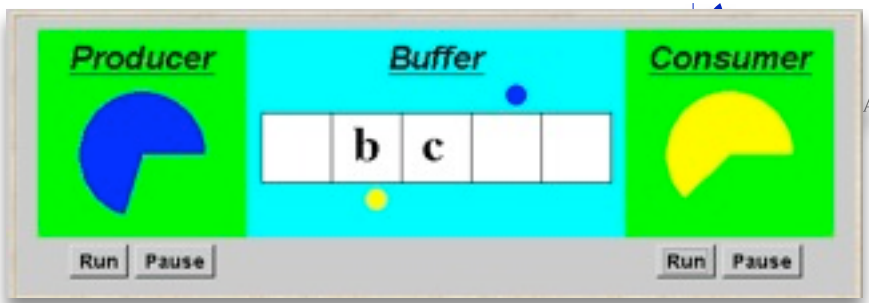


Producer / Consumer

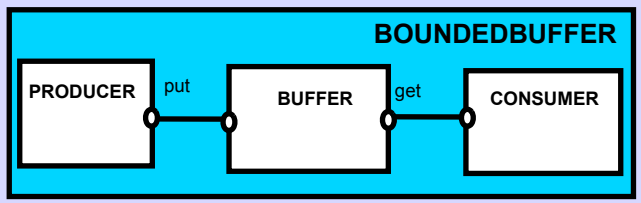




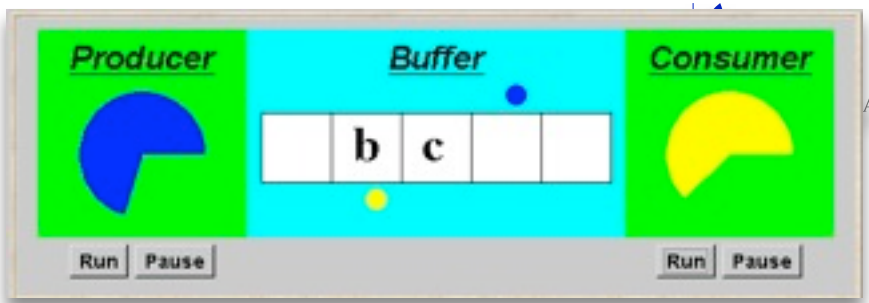
Producer / Consumer



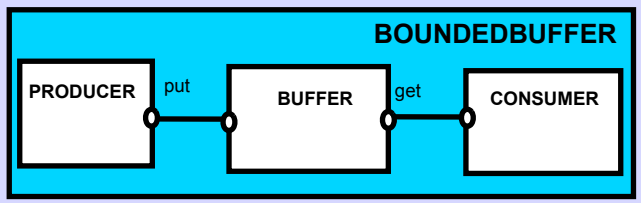

```
PRODUCER = (put->PRODUCER).

CONSUMER = (get->CONSUMER).
```



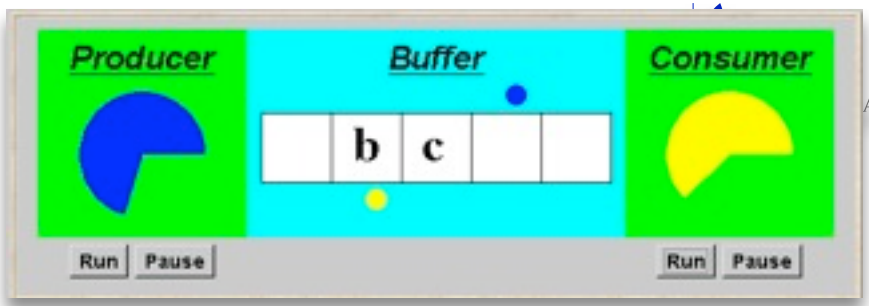
Producer / Consumer



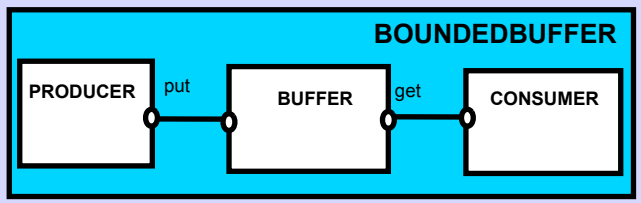

```
PRODUCER = (put->PRODUCER).

CONSUMER = (get->CONSUMER).

BUFFER(SIZE=5) = COUNT[0],
```



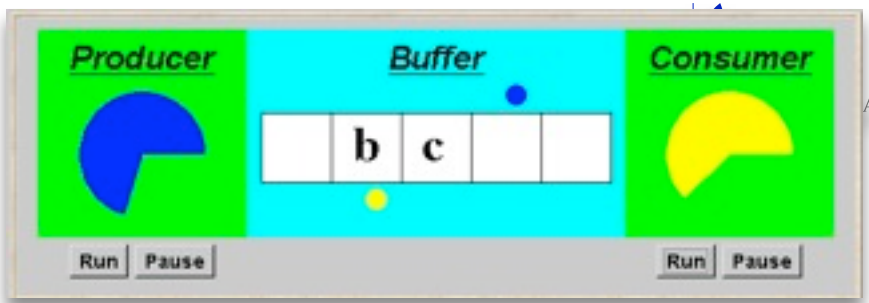
Producer / Consumer



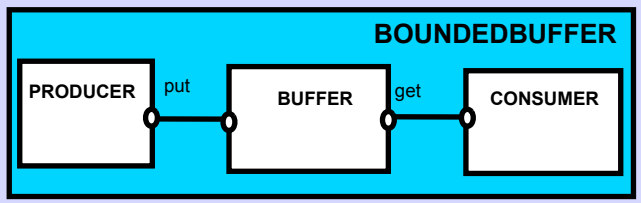

```
PRODUCER = (put->PRODUCER).

CONSUMER = (get->CONSUMER).

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



Producer / Consumer




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

CONSUMER = (get->CONSUMER).

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

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

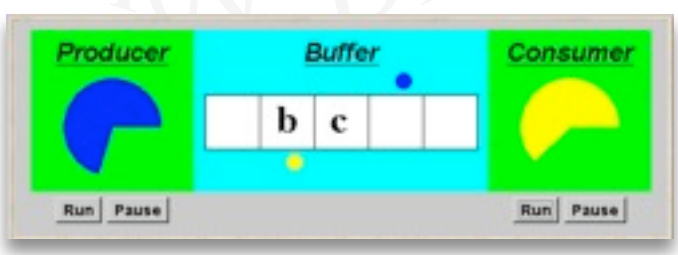


Bounded Buffer Program - Buffer Monitor

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```



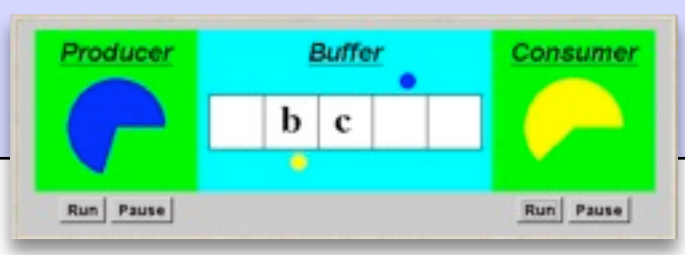
Bounded Buffer Program - Buffer Monitor

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
class BufferImpl<E> implements Buffer<E> {
    protected E[] queue;
    protected int in, out, count, SIZE;
    …
```



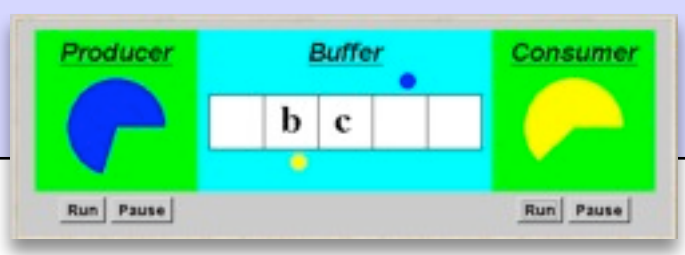
Bounded Buffer Program - Buffer Monitor

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
class BufferImpl<E> implements Buffer<E> {
    protected E[] queue;
    protected int in, out, count, SIZE;
    …
    synchronized void put(E o) throws Int'Exc' {
        while (!(count<SIZE)) wait();
```



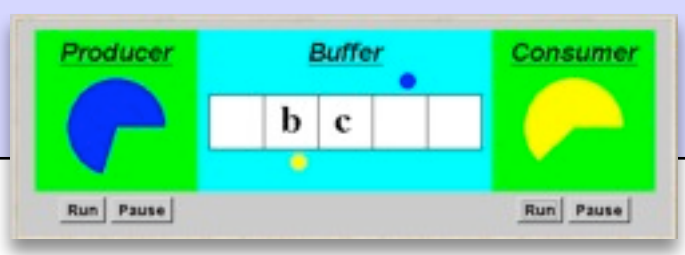
Bounded Buffer Program - Buffer Monitor

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
class BufferImpl<E> implements Buffer<E> {
    protected E[] queue;
    protected int in, out, count, SIZE;
    …
    synchronized void put(E o) throws Int'Exc' {
        while (!(count<SIZE)) wait();
        queue[in] = o;
        count++;
        in = (in+1) % SIZE;
```



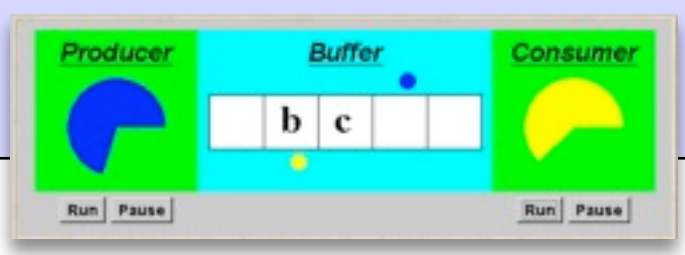
Bounded Buffer Program - Buffer Monitor

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
class BufferImpl<E> implements Buffer<E> {
    protected E[] queue;
    protected int in, out, count, SIZE;
    …
    synchronized void put(E o) throws Int'Exc' {
        while (!(count<SIZE)) wait();
        queue[in] = o;
        count++;
        in = (in+1) % SIZE;
        notifyAll();
    }
```



Bounded Buffer Program - Buffer Monitor

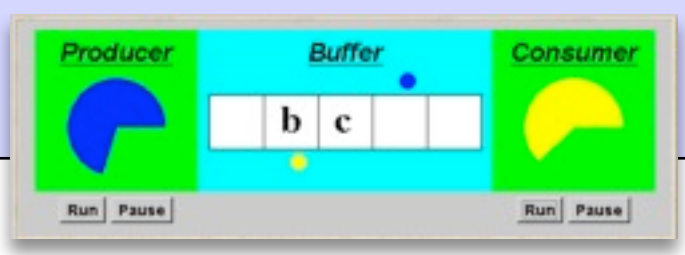
University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
class BufferImpl<E> implements Buffer<E> {
    protected E[] queue;
    protected int in, out, count, SIZE;
    …
    synchronized void put(E o) throws Int'Exc' {
        while (!(count<SIZE)) wait();
        queue[in] = o;
        count++;
        in = (in+1) % SIZE;
        notifyAll();
    }
```

Can we use notify()?



Bounded Buffer Program - Buffer Monitor

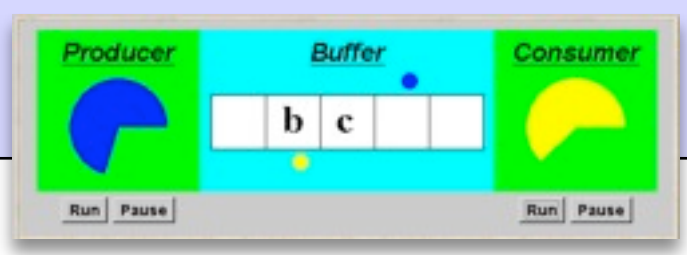
University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
class BufferImpl<E> implements Buffer<E> {
    protected E[] queue;
    protected int in, out, count, SIZE;
    …
    synchronized void put(E o) throws Int'Exc' {
        while (!(count<SIZE)) wait();
        queue[in] = o;
        count++;
        in = (in+1) % SIZE;
        notifyAll();
    }
```

if(count == 1)

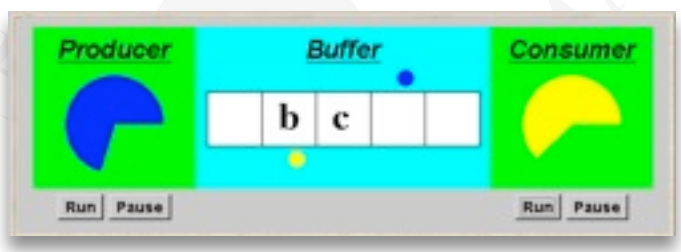


Similarly for get()

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```



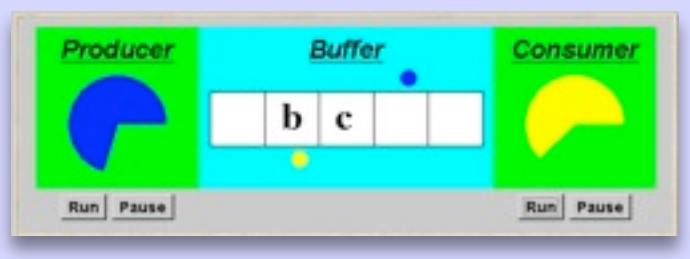
Similarly for get()

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
    …
    synchronized E get() throws Int'Exc' {
```



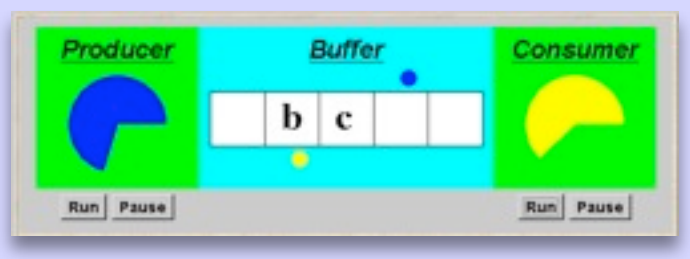
Similarly for get()

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
    …
    synchronized E get() throws Int'Exc' {
1.      while (!(count>0)) wait();
```

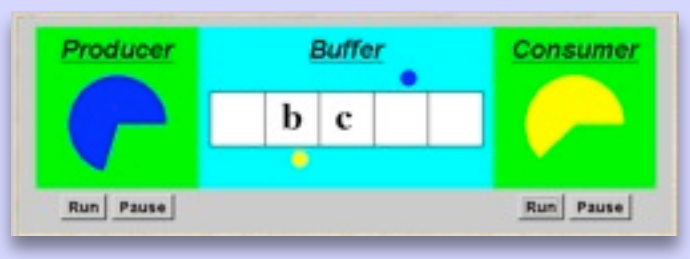


Similarly for get()


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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
    …
    synchronized E get() throws Int'Exc' {
1.      while (!(count>0)) wait();
2.      E obj = queue[out];
 < 2½.  queue[out] = null;              // WHY(?)
3.      count--;
4.      out = (out+1) % SIZE;
```



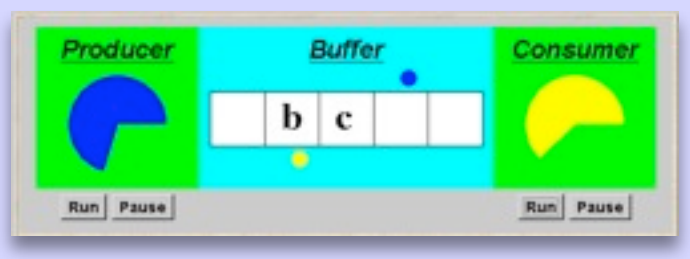
Similarly for get()

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
    …
    synchronized E get() throws Int'Exc' {
1.      while (!(count>0)) wait();
2.      E obj = queue[out];
 < 2½.  queue[out] = null;              // WHY(?)
3.      count--;
4.      out = (out+1) % SIZE;
5.      notifyAll();
```



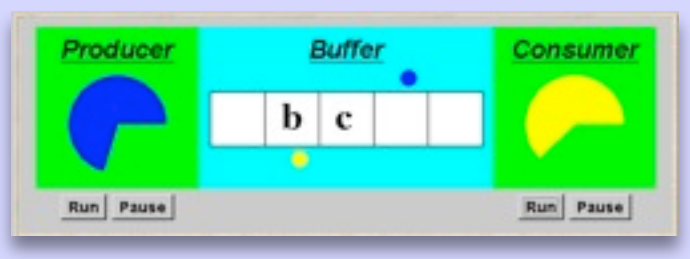
Similarly for get()

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
    …
    synchronized E get() throws Int'Exc' {
1.      while (!(count>0)) wait();
2.      E obj = queue[out];
 < 2½.  queue[out] = null;              // WHY(?)
3.      count--;
4.      out = (out+1) % SIZE;
5.      notifyAll();
6.      return obj;
    }
```



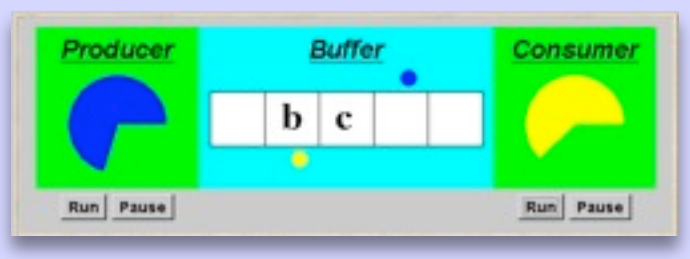
Similarly for get()

University of Southern Denmark

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

```
public interface Buffer<E> {
    public void put(E o)      throws InterruptedException;
    public E get()            throws InterruptedException;
}
```

```
    …
    synchronized E get() throws Int'Exc' {
1.      while (!(count>0)) wait();
2.      E obj = queue[out];
 < 2½.  queue[out] = null;              // WHY(?)
3.      count--;
4.      out = (out+1) % SIZE;
5.      notifyAll();
6.      return obj;
    }
```




```
if(count == queue.length-1)
```

Producer Process

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

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

    Producer(Buffer<Character> b) { buf = b; }

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

Producer Process

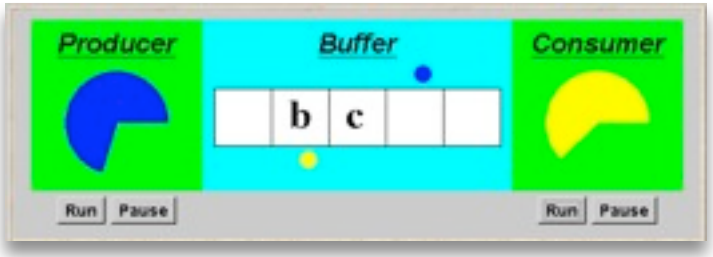
```
PRODUCER = (put->PRODUCER).
```

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

    Producer(Buffer<Character> b) { buf = b; }

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

Similar, **Consumer** calls **buf.get()**

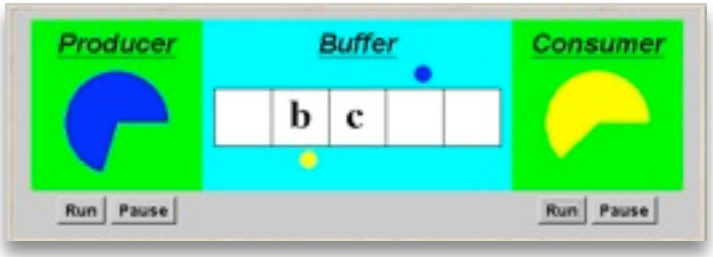
The Nested Monitor Problem

5.4 Nested Monitors (Semaphores)



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

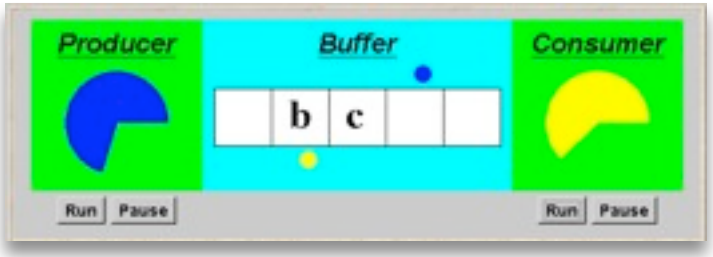
5.4 Nested Monitors (Semaphores)



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

```
class SemaBuffer implements Buffer {
    protected Object queue[];
    protected int in, out, count, SIZE;
    Semaphore empty; // block put appropriately
    Semaphore full;  // block get appropriately
```

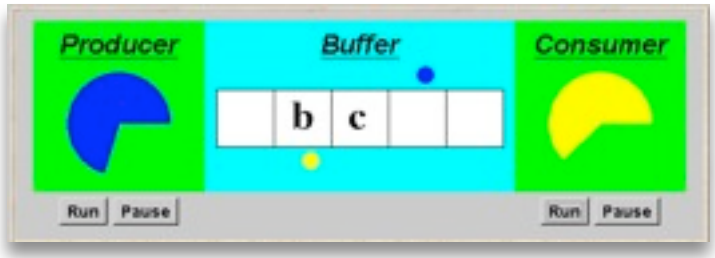
5.4 Nested Monitors (Semaphores)



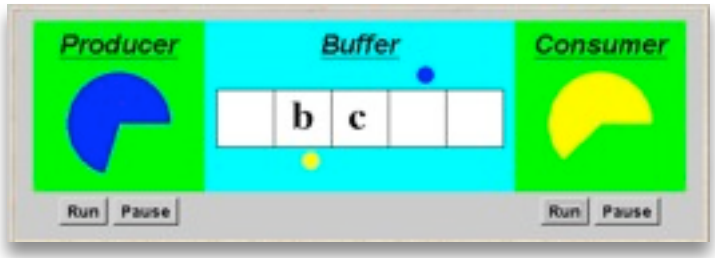
Suppose that, instead of using the **count** variable and condition synchronisation, we instead use 2 semaphores **full** and **empty** to reflect the state of the buffer:

```
class SemaBuffer implements Buffer {
    protected Object queue[];
    protected int in, out, count, SIZE;
    Semaphore empty; // block put appropriately
    Semaphore full;  // block get appropriately

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

Nested Monitors Java Program

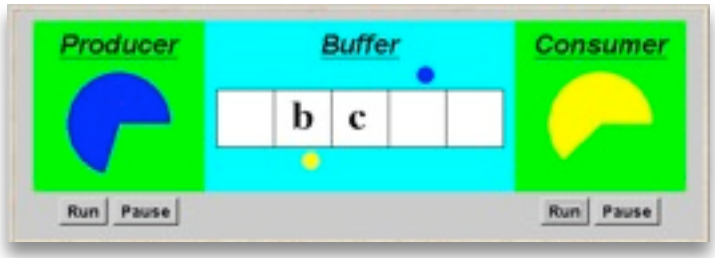


Nested Monitors Java Program




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

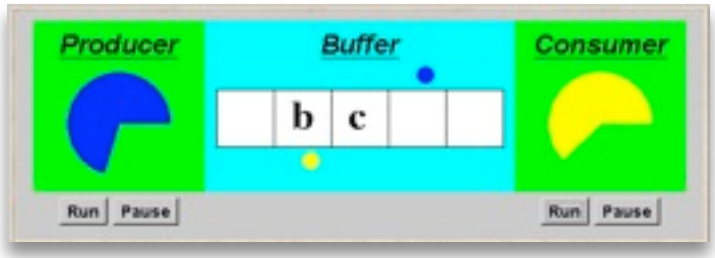
Nested Monitors Java Program




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

empty is decremented during a **put**, which is blocked if empty is zero, i.e., no spaces are left.

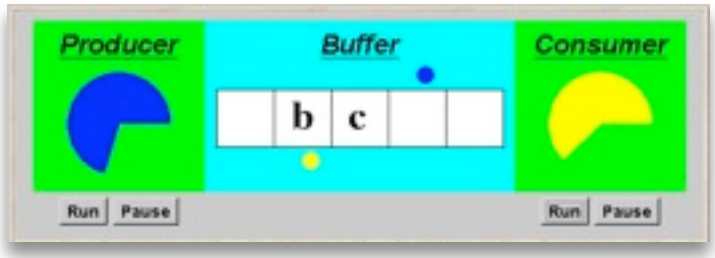
Nested Monitors Java Program




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

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

empty is decremented during a **put**, which is blocked if **empty** is zero, i.e., no spaces are left.

Nested Monitors Java Program

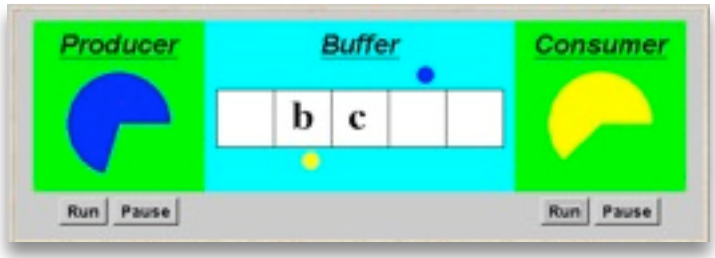



```
synchronized public void put(E o) throws Int'Exc' {
    empty.down();
    queue[in] = o;
    count++;
    in = (in+1) % SIZE;
    full.up();
}

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

empty is decremented during a **put**, which is blocked if **empty** is zero, i.e., no spaces are left.

full is decremented by a **get**, which is blocked if **full** is zero, i.e., if the buffer is empty.

Nested Monitors Java Program




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

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

empty is decremented during a **put**, which is blocked if **empty** is zero, i.e., no spaces are left.

full is decremented by a **get**, which is blocked if **full** is zero, i.e., if the buffer is empty.

Does this behave as desired?

Nested Monitors Model

Nested Monitors Model

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

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

CONSUMER = (get -> CONSUMER).

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

Nested Monitors Model

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

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

CONSUMER = (get -> CONSUMER).

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

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

Nested Monitors Model

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

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

CONSUMER = (get -> CONSUMER).

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

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

||BOUNDEDBUFFER =
     ( PRODUCER || BUFFER || CONSUMER
```

Nested Monitors Model

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

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

CONSUMER = (get -> CONSUMER).

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

Nested Monitors Model

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

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

CONSUMER = (get -> CONSUMER).

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

University of Southern Denmark

LTSA analysis predicts a DEADLOCK:

```
Composing
  potential DEADLOCK
...
Trace to DEADLOCK:
     get
```

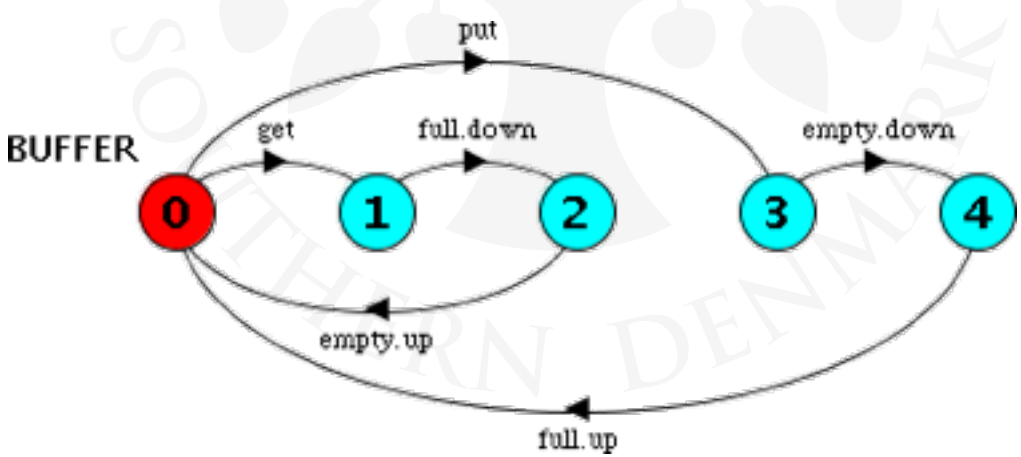
Nested Monitors

University of Southern Denmark

LTSA analysis predicts a DEADLOCK:

```
Composing
  potential DEADLOCK
...
Trace to DEADLOCK:
     get
```

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



Nested Monitors

University of Southern Denmark

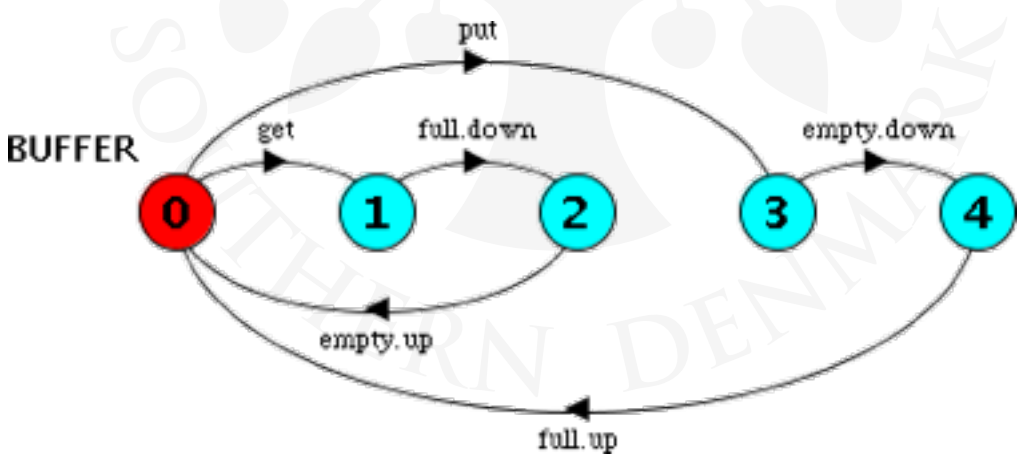
LTSA analysis predicts a **DEADLOCK**:

```
Composing
  potential DEADLOCK
...
Trace to DEADLOCK:
     get
```

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

Looking at BUFFER:

After **get** the next action is **full.down** (blocks).



Nested Monitors

University of Southern Denmark

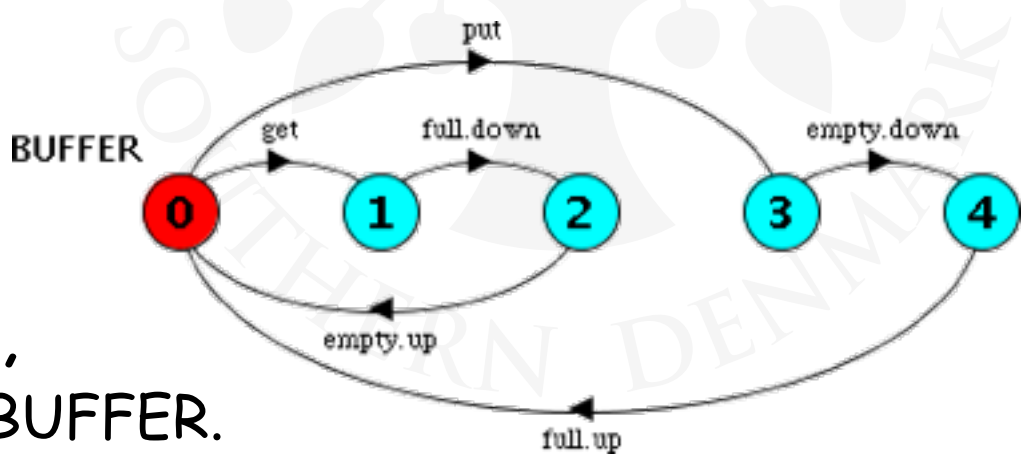
LTSA analysis predicts a **DEADLOCK**:

```
Composing
  potential DEADLOCK
...
Trace to DEADLOCK:
     get
```

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

Looking at BUFFER:
After **get** the next action is **full.down** (blocks).

We cannot do **put** (and unblock full), since we have the "semaphore" for BUFFER.



Nested Monitors

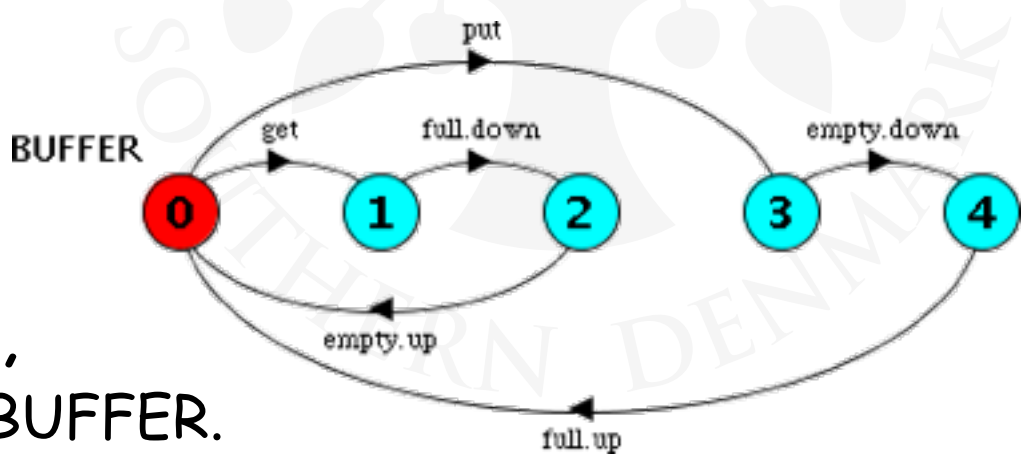
University of Southern Denmark

LTSA analysis predicts a **DEADLOCK**:

```
Composing
  potential DEADLOCK
...
Trace to DEADLOCK:
     get
```

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

Looking at BUFFER:
After **get** the next action is **full.down** (blocks).



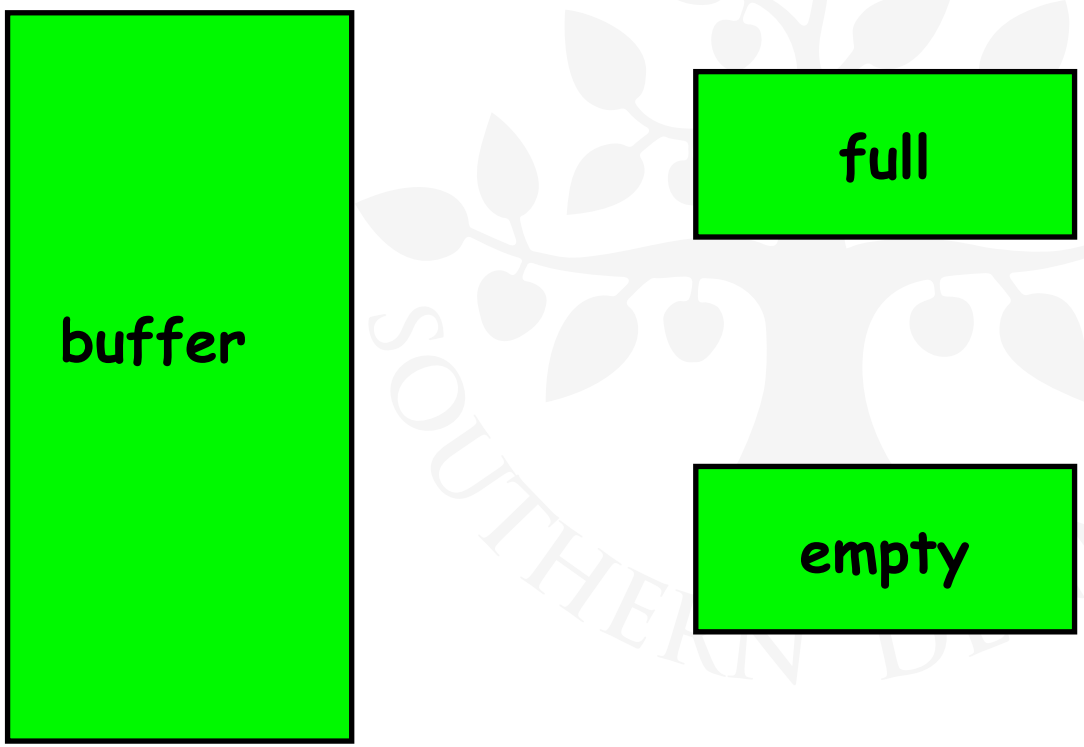
We cannot do **put** (and unblock full), since we have the "semaphore" for BUFFER.

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

Nested Monitor Problem

University of Southern Denmark

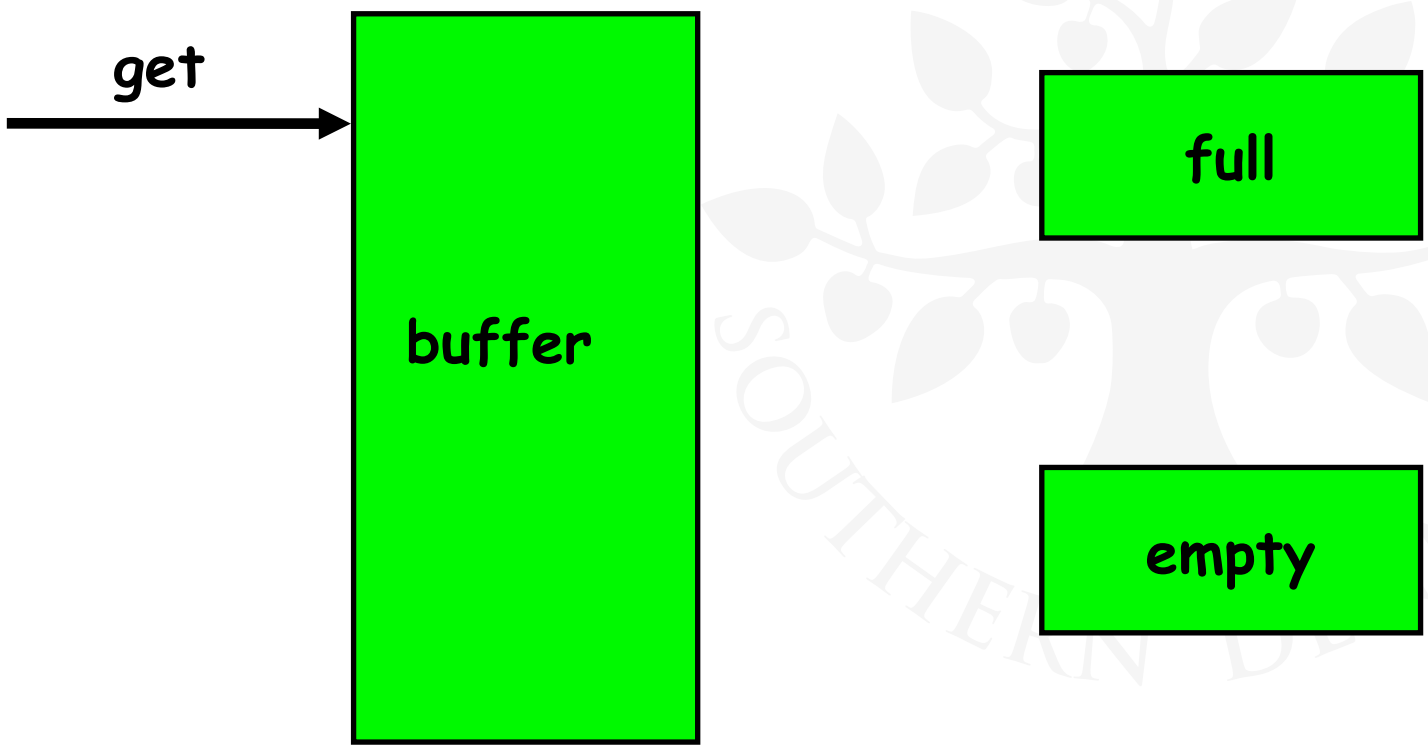
```
synchronized public E get()
                    throws InterruptedException{
     full.down(); // if no items, block!
     ...
   }
```



Nested Monitor Problem

University of Southern Denmark

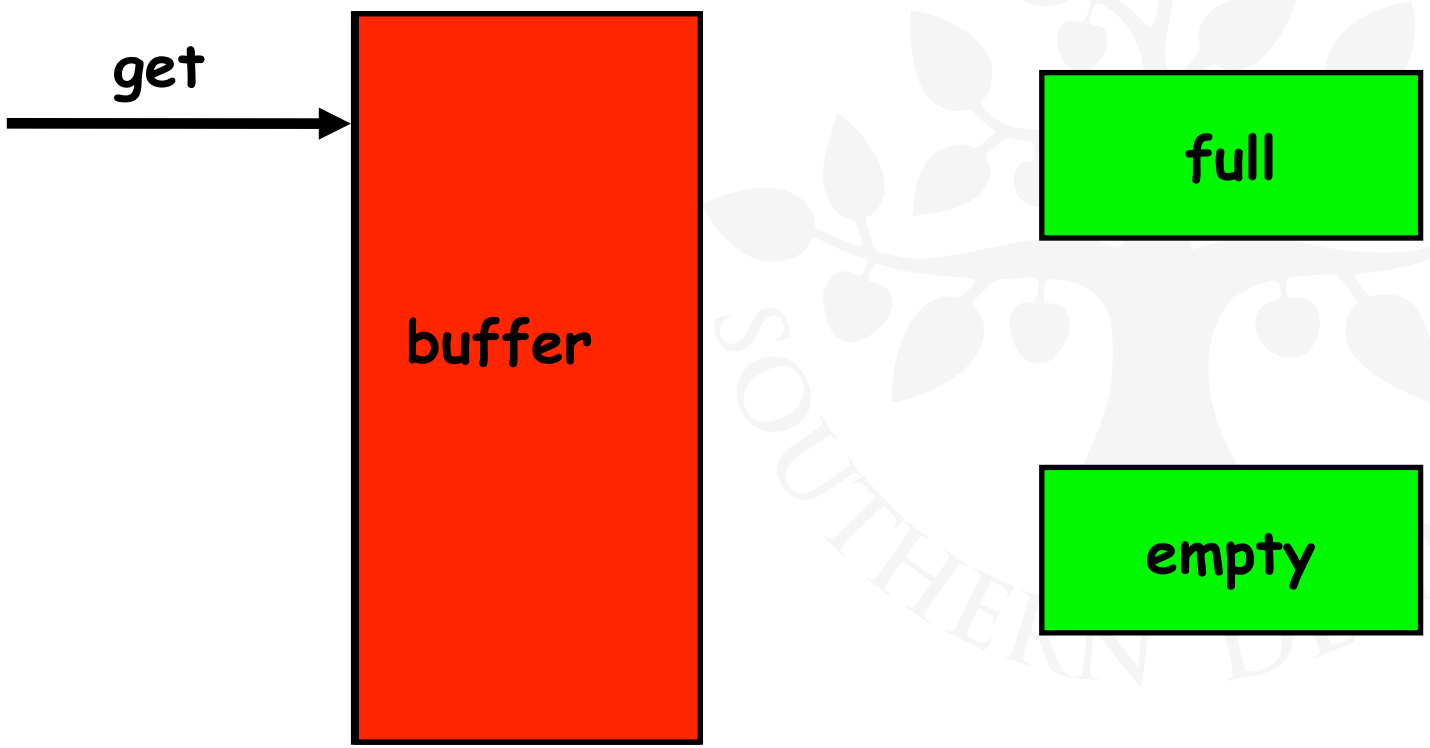
```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitor Problem

University of Southern Denmark

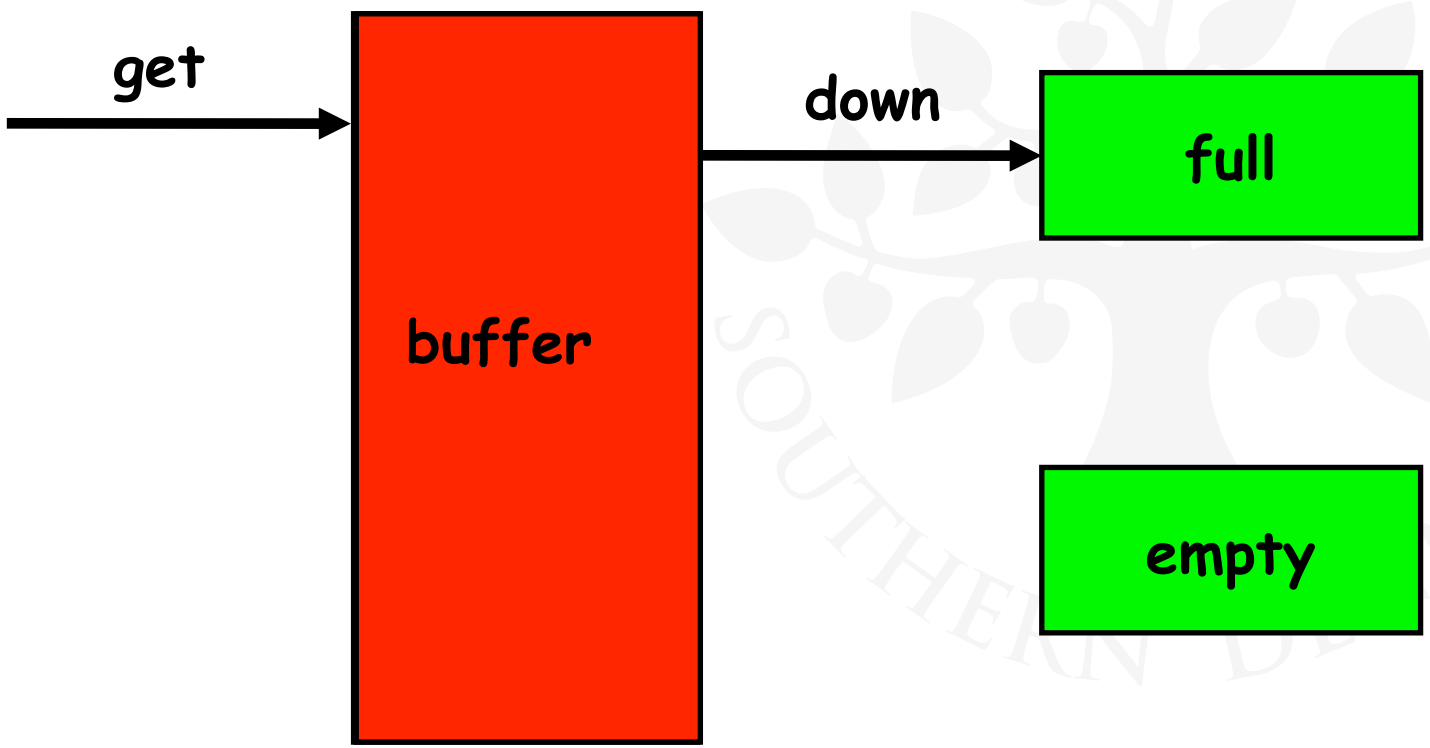
```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitor Problem

University of Southern Denmark

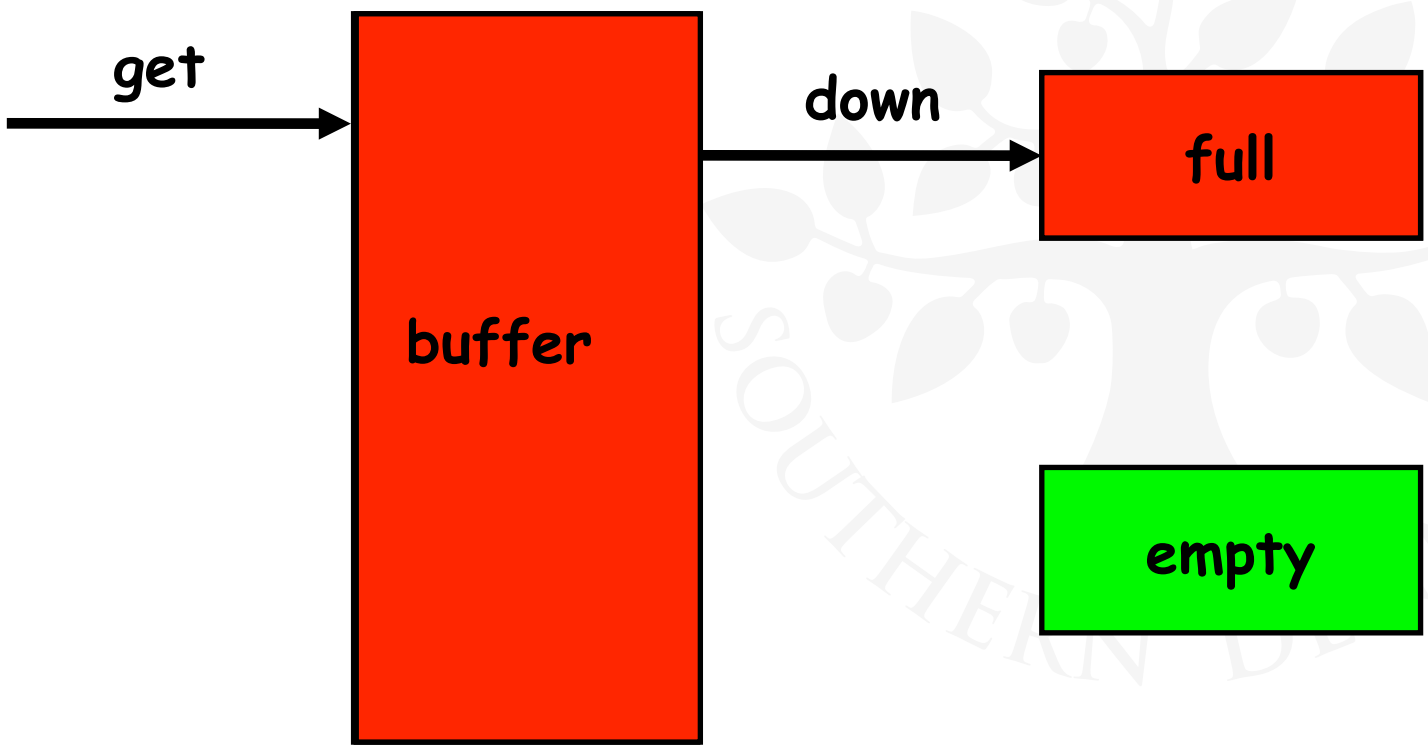
```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitor Problem

University of Southern Denmark

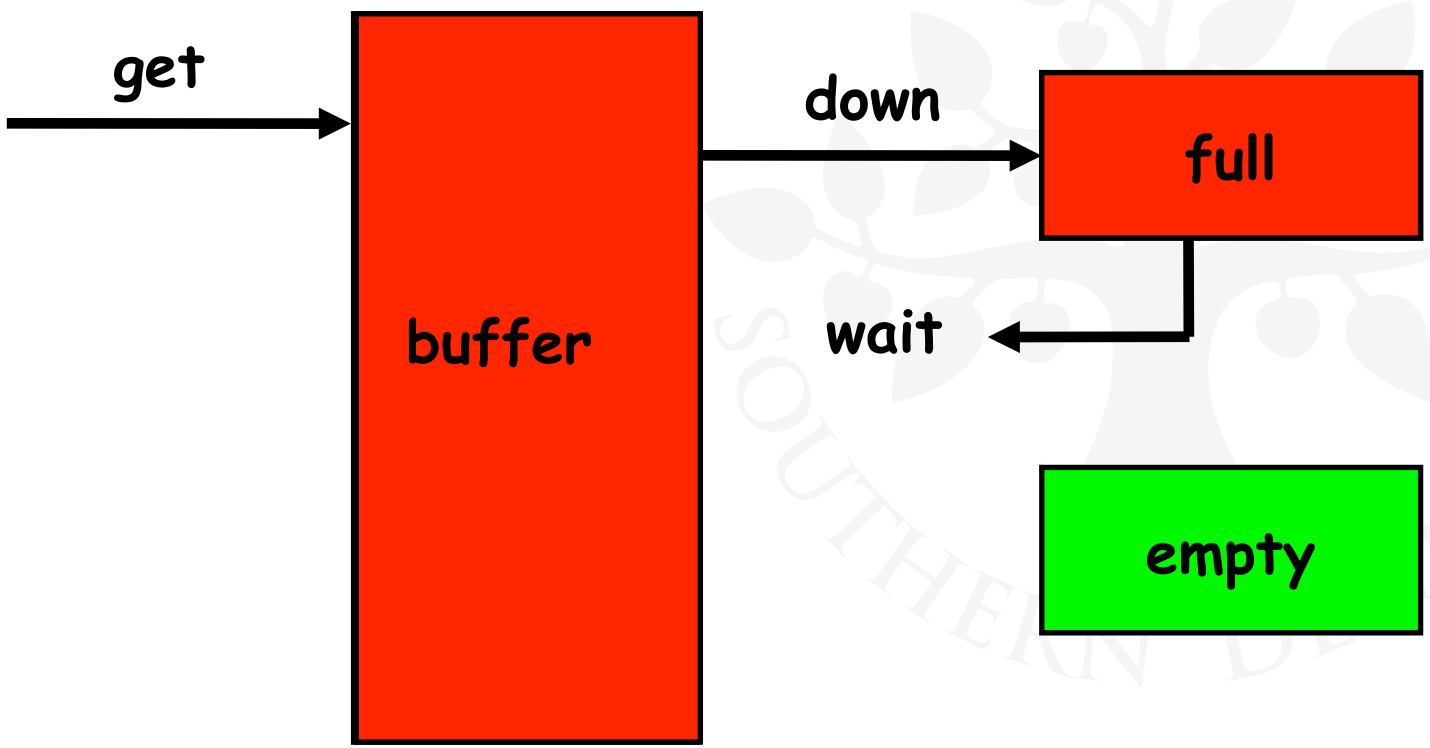
```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitor Problem

University of Southern Denmark

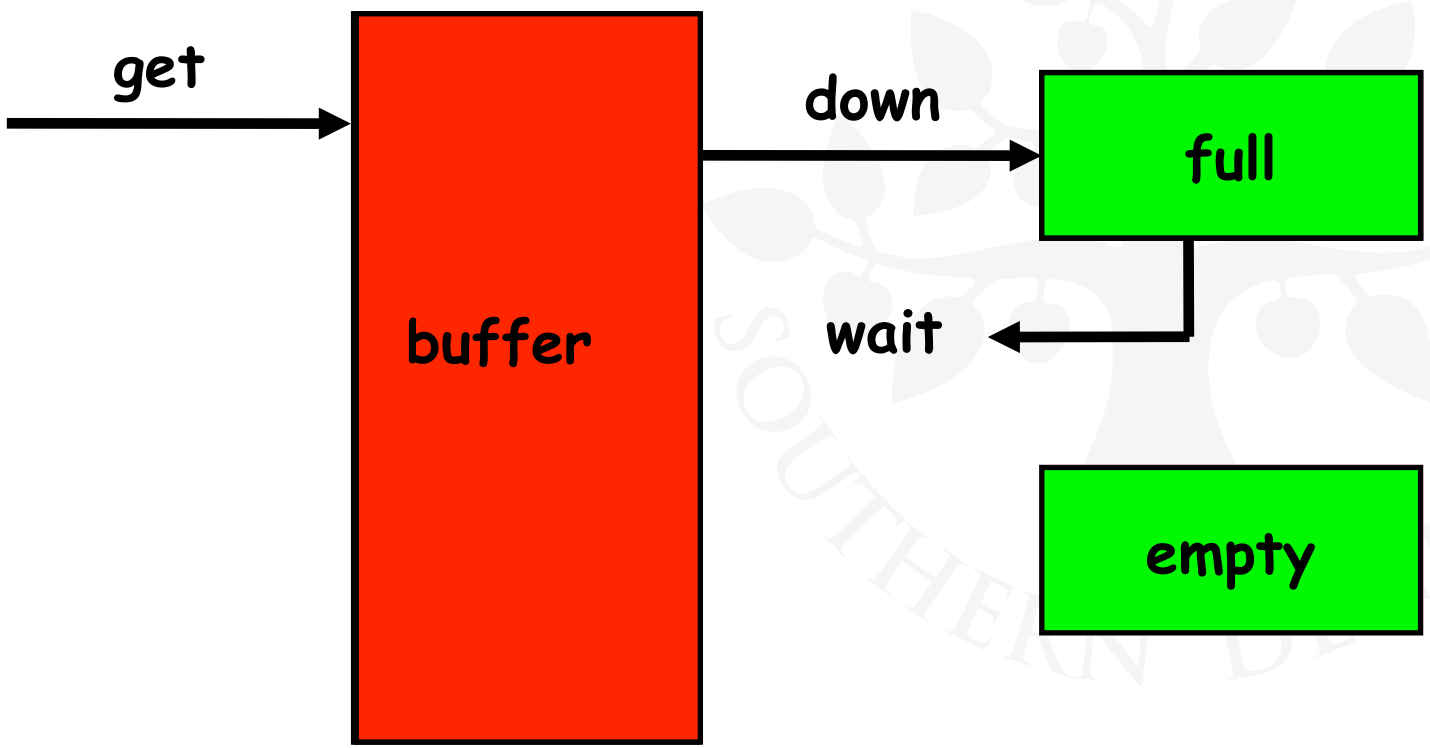
```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitor Problem

University of Southern Denmark

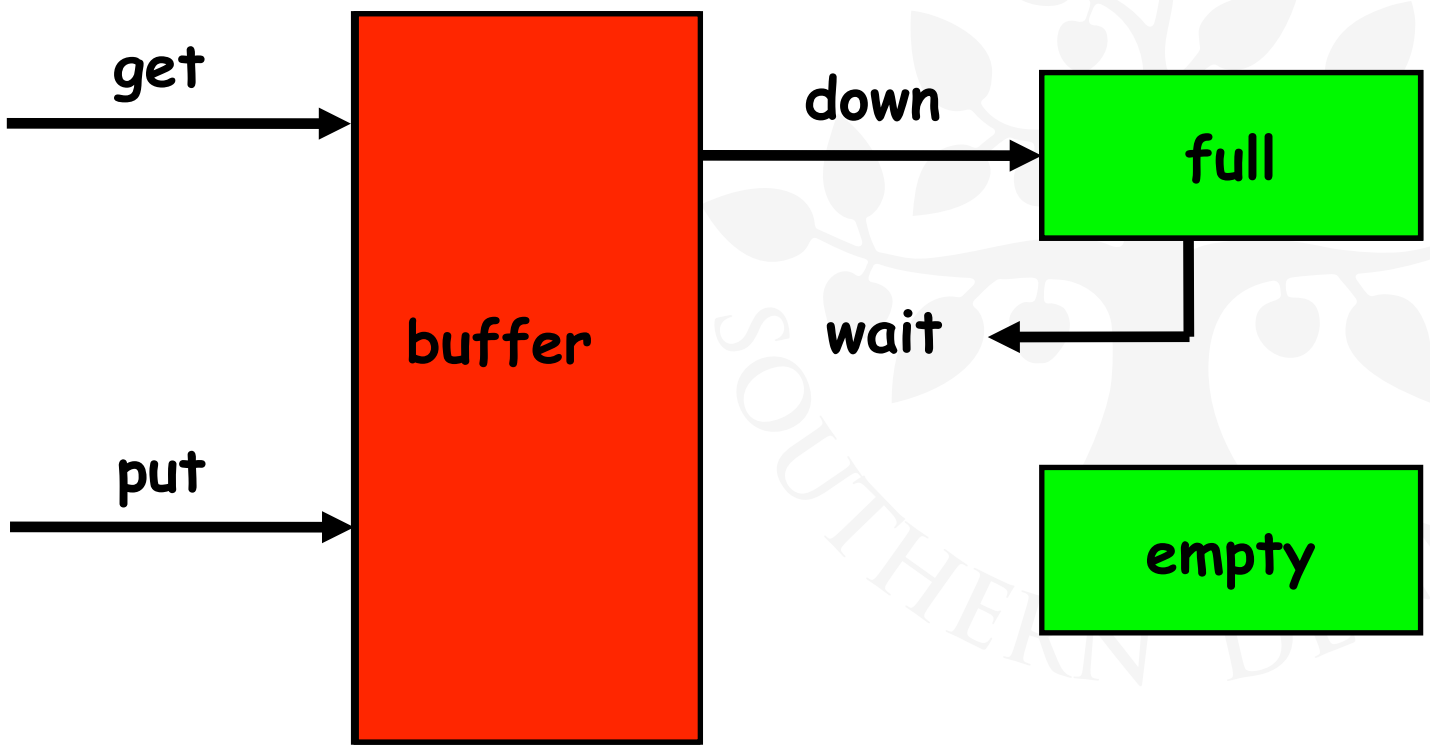
```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitor Problem

University of Southern Denmark

```
synchronized public E get()
                    throws InterruptedException{
    full.down(); // if no items, block!
    ...
}
```



Nested Monitors - Revised Bounded Buffer Program

University of Southern Denmark

The only way to avoid it in Java is by **careful design** :

Nested Monitors - Revised Bounded Buffer Program

University of Southern Denmark

The only way to avoid it in Java is by **careful design** :

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

Nested Monitors - Revised Bounded Buffer Program

University of Southern Denmark

The only way to avoid it in Java is by **careful design** :

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

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

Nested Monitors - Revised Bounded Buffer Program

University of Southern Denmark

The only way to avoid it in Java is by **careful design** :

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

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

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.

Nested Monitors
- Revised Bounded Buffer Model

University of Southern Denmark

The semaphore actions have been moved **outside** the monitor,
i.e., conceptually, to the producer and consumer:

Nested Monitors
- Revised Bounded Buffer Model

University of Southern Denmark

The semaphore actions have been moved **outside** the monitor,
i.e., conceptually, to the producer and consumer:

```
BUFFER =   (put -> BUFFER
           |get -> BUFFER).

PRODUCER = (empty.down -> put -> full.up  -> PRODUCER).
CONSUMER = (full.down  -> get -> empty.up -> CONSUMER).
```

Nested Monitors
- Revised Bounded Buffer Model

University of Southern Denmark

The semaphore actions have been moved **outside** the monitor,
i.e., conceptually, to the producer and consumer:

```
BUFFER =   (put -> BUFFER
           |get -> BUFFER).

PRODUCER = (empty.down -> put -> full.up  -> PRODUCER).
CONSUMER = (full.down  -> get -> empty.up -> CONSUMER).
```

Does this behave as desired?

Nested Monitors
- Revised Bounded Buffer Model

University of Southern Denmark

The semaphore actions have been moved **outside** the monitor,
i.e., conceptually, to the producer and consumer:

```
BUFFER =   (put -> BUFFER
           |get -> BUFFER).

PRODUCER = (empty.down -> put -> full.up  -> PRODUCER).
CONSUMER = (full.down  -> get -> empty.up -> CONSUMER).
```

Does this behave as desired?

```
No deadlocks/errors
```

5.5 Monitor invariants

University of Southern Denmark

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 .

5.5 Monitor invariants

University of Southern Denmark

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): $0 \leq spaces \leq CAPACITY$

5.5 Monitor invariants

University of Southern Denmark

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):	$0 \leq spaces \leq CAPACITY$
INV(Semaphore):	$0 \leq value$

5.5 Monitor invariants

University of Southern Denmark

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):	$0 \leq spaces \leq CAPACITY$
INV(Semaphore):	$0 \leq value$
INV(Buffer):	$0 \leq count \leq SIZE$

5.5 Monitor invariants

University of Southern Denmark

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):	$0 \leq spaces \leq CAPACITY$
INV(Semaphore):	$0 \leq value$
INV(Buffer):	$0 \leq count \leq SIZE$ and $0 \leq in < SIZE$

5.5 Monitor invariants

University of Southern Denmark

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):	$0 \leq spaces \leq CAPACITY$
INV(Semaphore):	$0 \leq value$
INV(Buffer):	$0 \leq count \leq SIZE$ and $0 \leq in < SIZE$ and $0 \leq out < SIZE$

5.5 Monitor invariants

University of Southern Denmark

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

5.5 Monitor invariants

University of Southern Denmark

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):	$0 \leq spaces \leq CAPACITY$
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

Summary

University of Southern Denmark

Concepts: monitors (and controllers):

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

Summary

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Summary

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Models: guarded actions

Summary

University of Southern Denmark

Concepts: monitors (and controllers):

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

nested monitors ("nested monitor problem")

Models: guarded actions

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