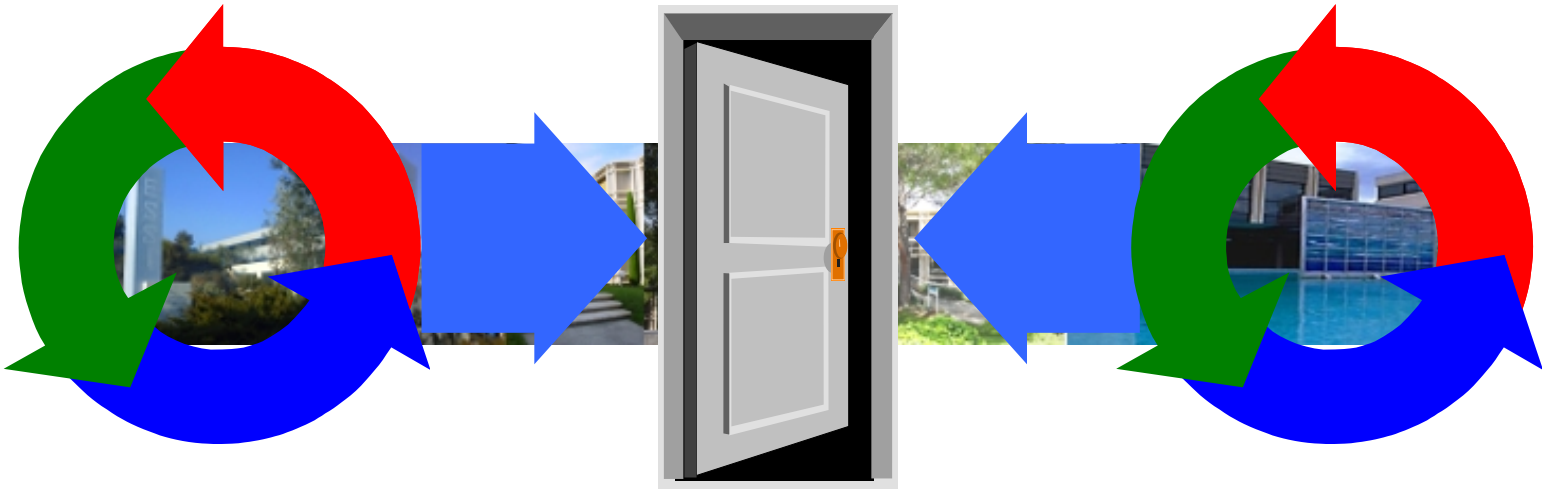
Moniteurs

riveill@unice.fr
http://www.i3s.unice.fr/~riveill



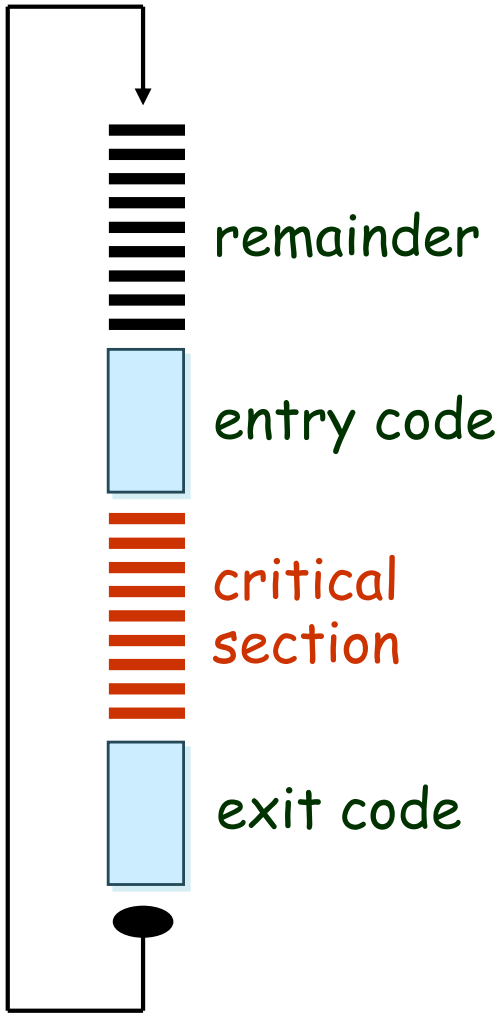
Université Nice Sophia Antipolis

Rappel du cours précédent
(mes fiches de révision)

The mutual exclusion problem

- Mutual Exclusion: No two processes are in their critical sections at the same time.
- Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
  - Property of safety
- Starvation-freedom: If a process is trying to enter its critical section, then this process must eventually enter its critical section.
  - Property of liveness

– No assumption time
– All process execute an equivalent algorithms



Semaphores

- Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore ***s*** is an 'integer variable' that can take only positive or null values.
- The only operations permitted on ***s*** are *up(s)* and *down(s)*. Blocked processes are held in a FIFO queue.

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

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

Mise en œuvre d'une section critique

- Solution 1
  - Utilisation d'un verrou
  - En Java : **`synchronized`** ou class Lock
- Solution 2
  - Utilisation d'un mutex
  - Rappel : un mutex est un sémaphore initialisé à 1
  - En Java : class Semaphore
- Solution 3
  - Utilisation d'un algorithme par attente active
  - Plus efficace en général mais plus difficile à utiliser

  → attention au respect des hypothèses : lectures et écritures atomique
  - En Java : **`volatile`** (ne fonctionne pas sur les tableaux)

Propriétés

- A **safety** property asserts that **nothing** bad happens.
  - In complex systems, it is usually better to specify safety properties by stating directly what **is** require

```
property MUTEX = (enter -> exit -> MUTEX).
```

- A **liveness** property asserts that something good *eventually* happens.
  - A progress property asserts that it is *always* the case that an action is *eventually* executed.

```
// enter always executed
progress L = {enter}
// enter or exit always executed
progress M = {enter, exit}
```

Tout ce qui est dans les transparents précédents est supposé être connu

Monitor

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

Moniteur versus sémaphore

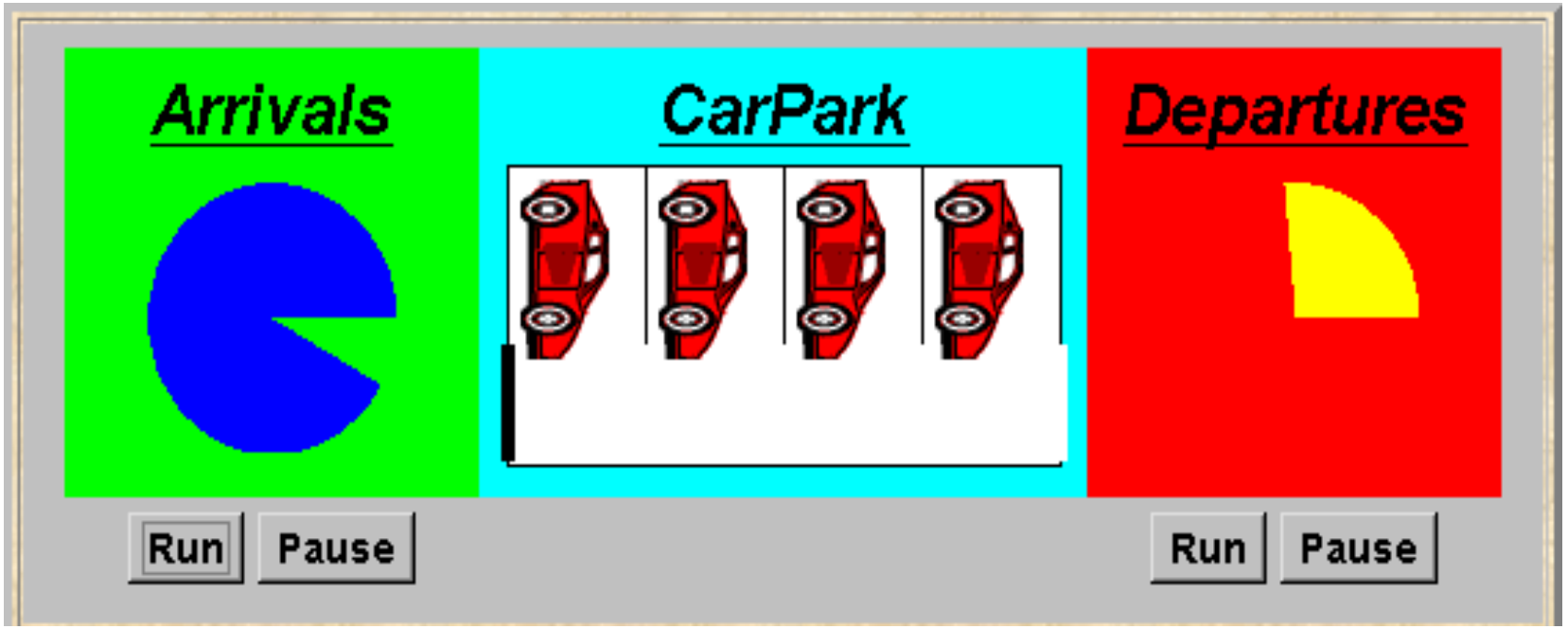
- Deux concepts différents pour un même objectif
  - Faire coopérer des processus
  - Sémaphore
    - Inventé par Edsger Dijkstra en 1962
    - Implémenté dans de très nombreux systèmes d'exploitation

    → plutôt système d'exploitation / moins structurants
  - Moniteur
    - Définie par C. A. R. Hoare en 1974
    - Implémenté par Per Brinch Hansen en 1975 dans une version concurrente du langage Pascal

    → plutôt langage / plus structurants
- **En général, on n'utilise pas les 2 concepts en même temps**
  - **Les effets d'un outil sur l'autre ne sont généralement pas spécifiés**

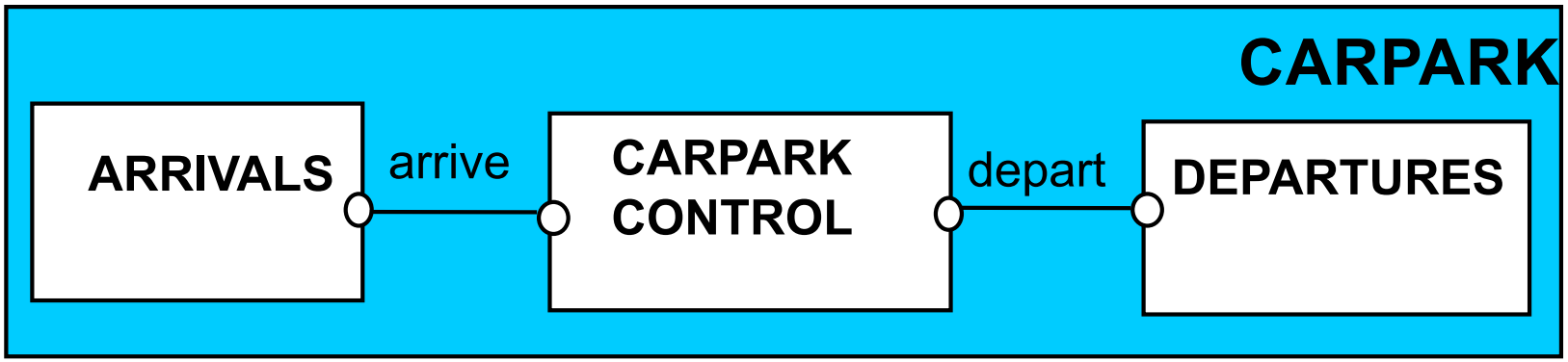
Condition synchronization

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



carpark model

- Events or actions of interest?
  - arrive and depart
- Identify processes
  - arrivals, departures and carpark control
- Define each interactions (structure).



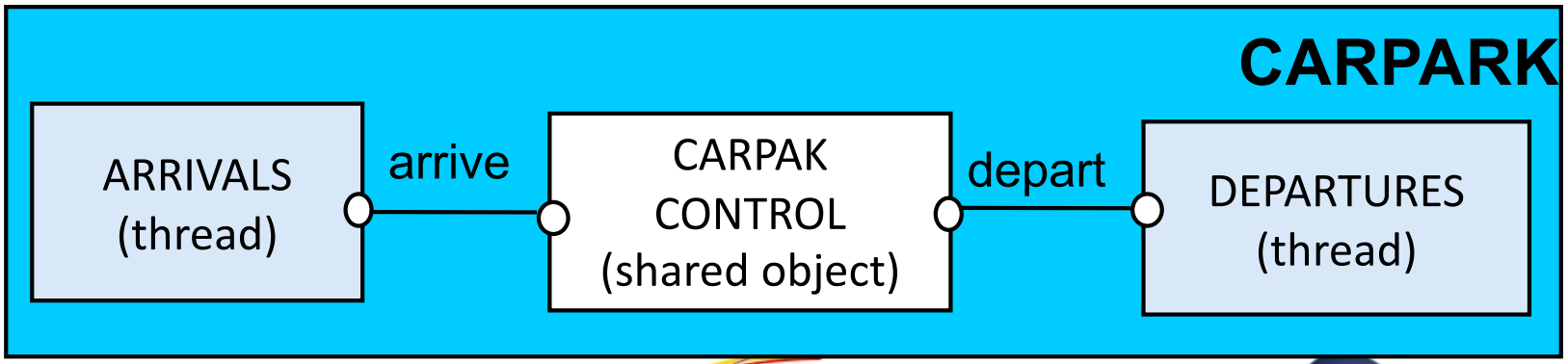
carpark model

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

```
const N = 4

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

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

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

carpark program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and shared object
  - Thread - **active** entity which initiates (output) actions
  - Shared object - **passive** entity which responds to (input) actions, need synchronisation
    - Shared between thread



Comment mettre en œuvre la synchronisation avec des sémaphores ?

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

- Sémaphore possède
  - 1 compteur
  - 1 primitive (down) qui bloque quand le compteur est à 0
  - 1 primitive (up) qui libère un processus ou incrémente le compteur

- On peut implémenter facilement

```
CARPARKCONTROL = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
```

→ Utilisation d'un sémaphore i, initialisé à N

Comment mettre en œuvre la synchronisation avec des sémaphores ?

- Pour l'autre action
  when(i<N) depart->SPACES[i+1]
  - il faut se ramener dans la même situation i.e.
    when (x > 0) depart -> SPACES[x-1]
- Possible, en introduisant une variable j tel que i+j = N

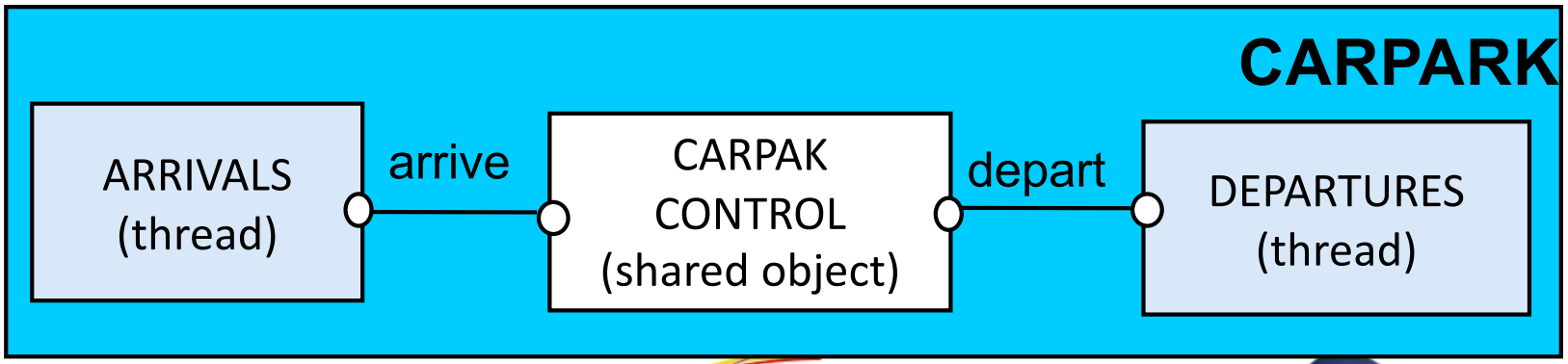
```
CARPARKCONTROL = SPACES[N][0],
SPACES[i:0..N][j:0..N] =
     (when(i>0) arrive->SPACES[i-1][j+1]
     |when (j>0) depart->SPACES[i+1][j-1]).
```

- Solution finale
  - 2 sémaphores : i = Semaphore(N) ; j = Semaphore(0)

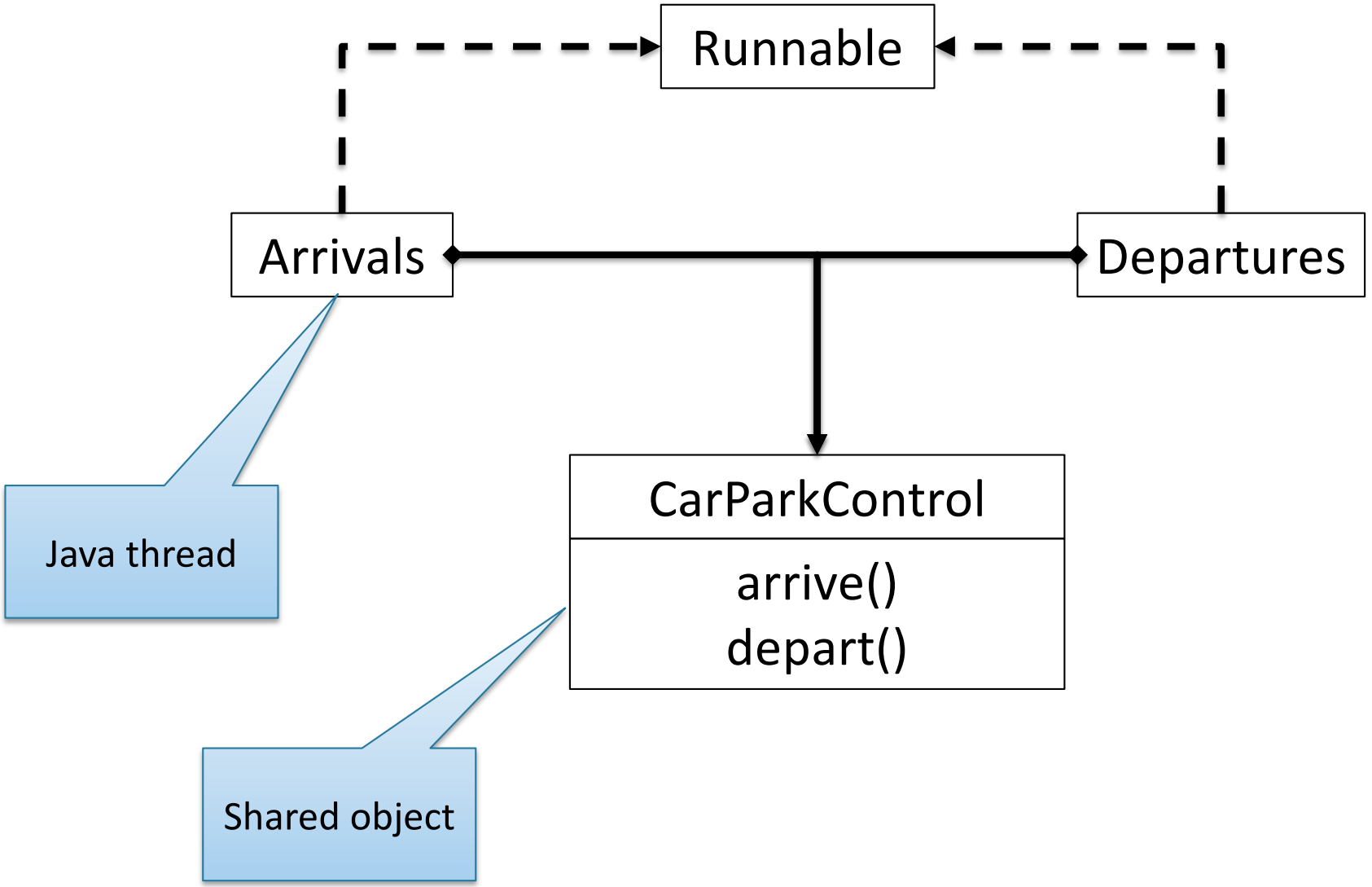
```
when(i>0) arrive->SPACES[i-1][j-1]
→ down(i); up(j); arrive

when (j>0) depart->SPACES[i+1][j-1]
→ down(j); up(i); depart
```

carpark program
Another implementation with monitor

- Model - all entities are processes interacting by actions
- Program - need to identify threads and shared object
  - thread - **active** entity which initiates (output) actions
  - monitor - **passive** entity which responds to (input) actions
    - Shared between thread



Autre approche en utilisant un objet moniteur



carpark program

- The active entities of the model
  - **Arrivals** and **Departures** (FSP model)
  - Are implemented in Java by a thread (implement **Runnable** interface)

- The passive entity of the model
  - **CarParkControl** (FSP model)
  - Provides the control (condition synchronization)
  - Are implemented in Java by a shared object

- All instances of these are created by the **main()** method

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

carpark program - `Arrivals` and `Departures` threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;

  Arrivals(CarParkControl c) {carpark = c;}

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

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

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

Carpark program - `CarParkControl` monitor

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

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

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

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

mutual exclusion by synchronized methods

condition synchronization?

block if full? (spaces==0)

block if empty? (spaces==N)

Université Nice Sophia Antipolis

i3S sophia antipolis

cnrs

Carpark program - `CarParkControl` monitor

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

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

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

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

mutual exclusion by synchronized methods

condition synchronization?

block if full? (spaces==0)

block if empty? (spaces==N)

Université Nice Sophia Antipolis

i3S sophia antipolis

cnrs

Hoare Monitor = Mutex + Condition variables

- Mutex
  - Ensure mutual exclusion execution of the procedures
- Condition synchronisation/Condition variables
  - Primitive to block or restart process execution
- There are mainly three main operations on condition variables:
  - **`wait c, m`**, where:
    - ***c*** is a condition variable
    - **m** is a mutex (lock)
  - **`signal c`**
  - **`broadcast c`**

Java monitor

- Mutex
  - Use `synchronized method`
- Condition synchronization
  - Java provides a thread **wait set** per monitor with the following methods:

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

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

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

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

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

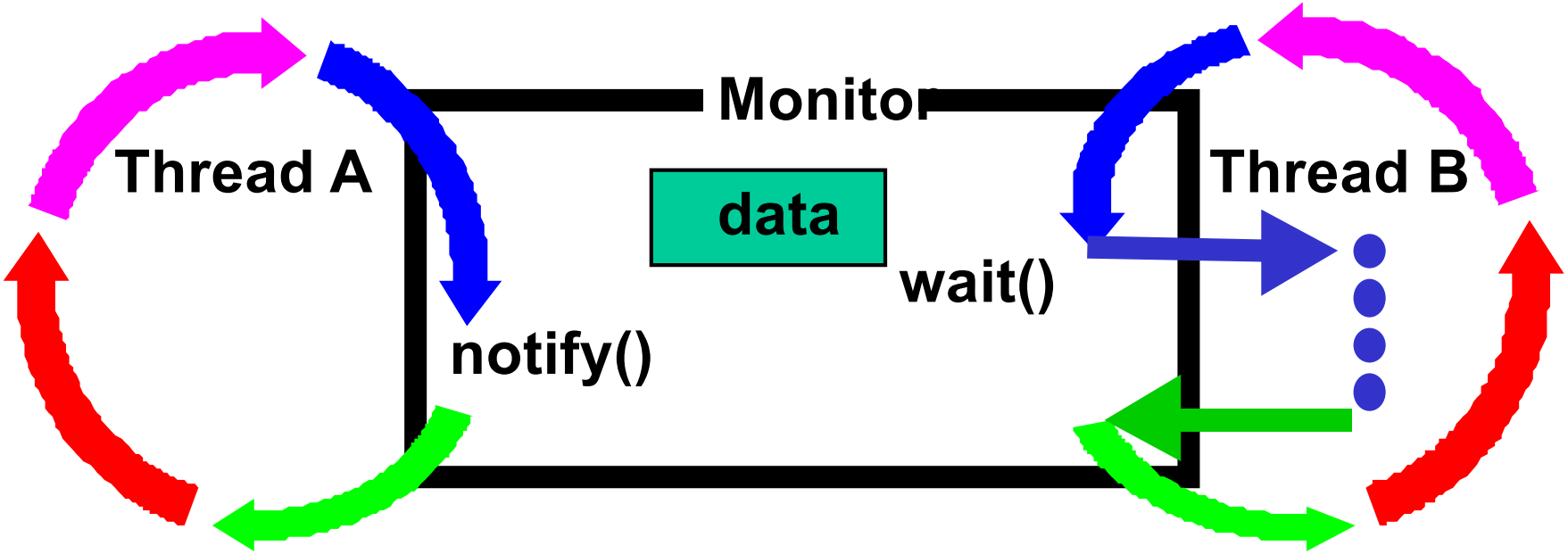
    - Wakes up all threads that are waiting on this object's wait set.

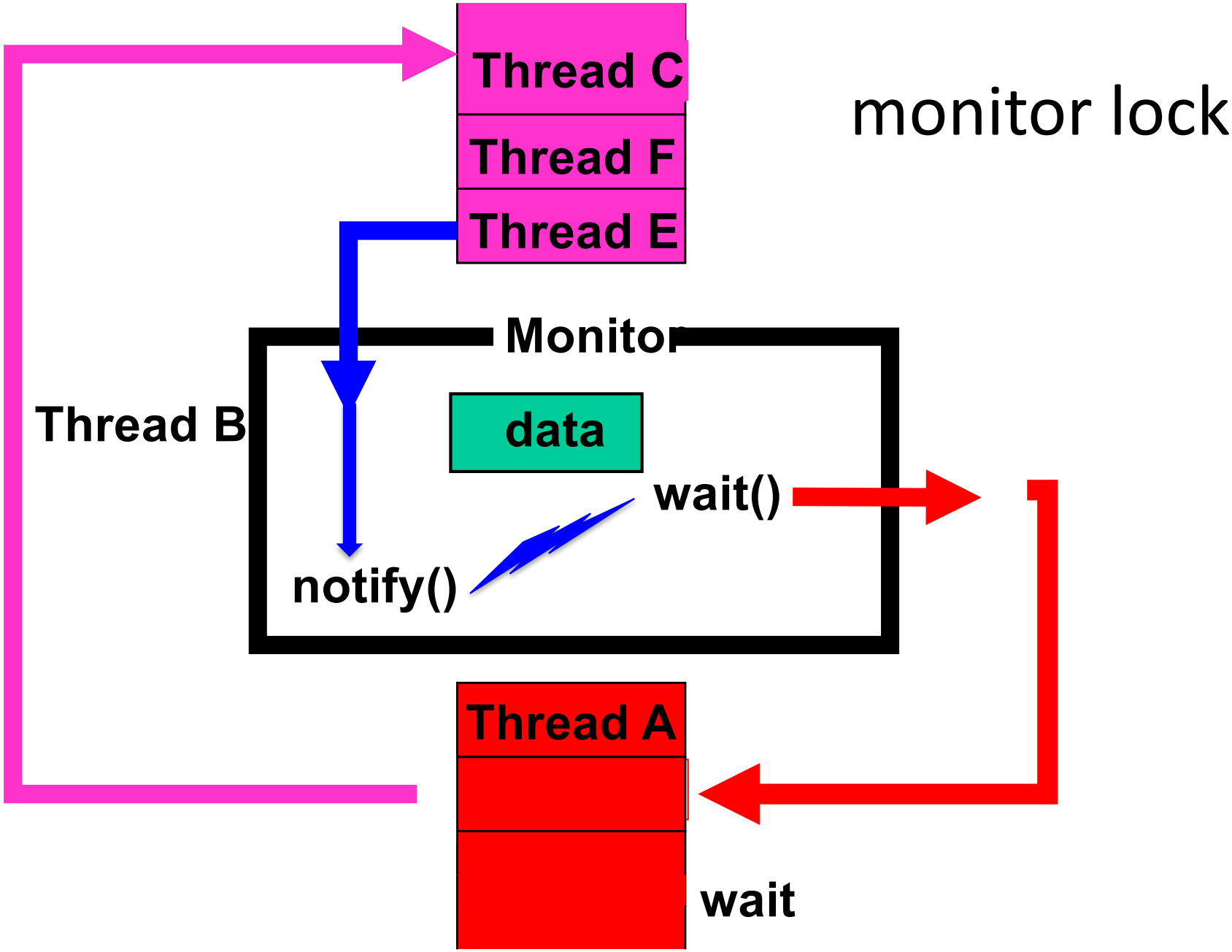
condition synchronization in Java

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

  Wait() - causes the thread to exit the monitor,

  permitting other threads to enter the monitor.




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

condition synchronization in Java

- **FSP:**

  when *cond act* -> NEWSTAT

- **Java:**

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

- The `while` loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when it re-enters the monitor.
- `notifyAll()` is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

CarParkControl - condition synchronization

FSP

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

Java

```
class CarParkControl {
    protected int spaces;
    protected int N;
    CarParkControl(int n) { N=spaces=n; }

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

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

Summary – Monitors

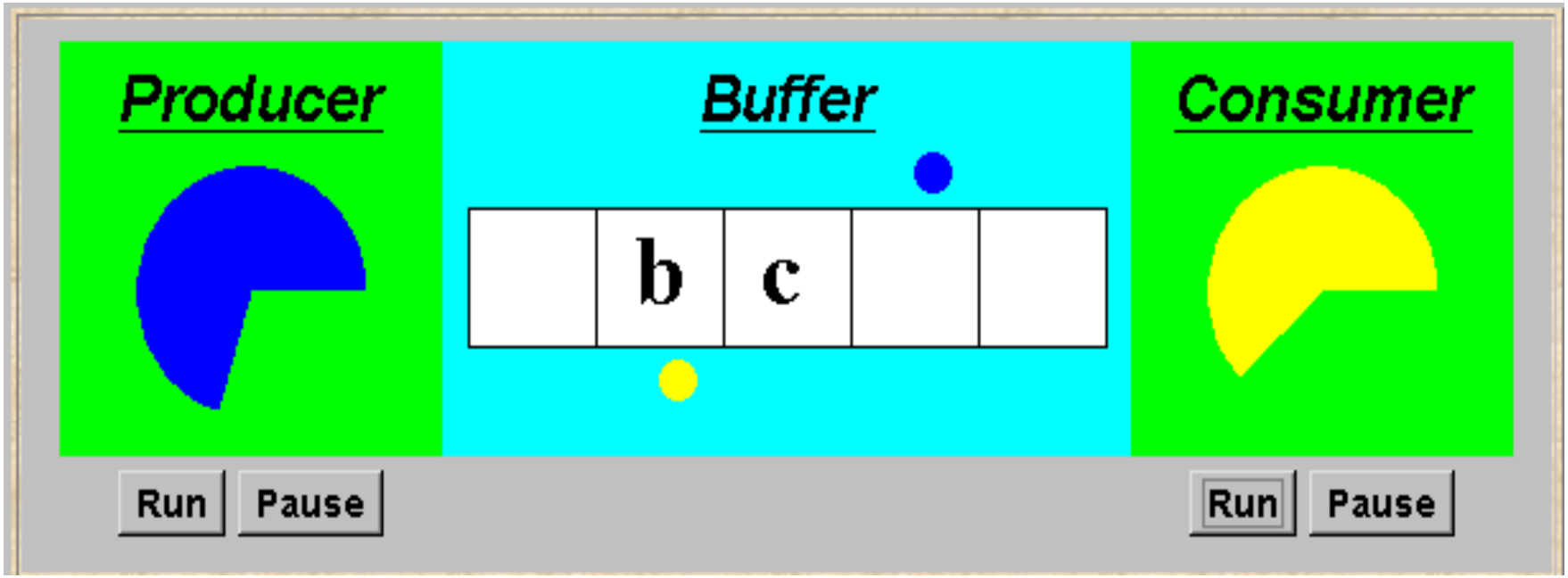
- Modelling
  - Active entities (that initiate actions) are implemented as threads.
  - Passive entities (that respond to actions) are implemented as monitors.
- Concurrency
  - Each guarded action in the model of a monitor is implemented as a `synchronized` method which uses a while loop and `wait()` 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 `notify()` or `notifyAll()`.
  - Rmq: Java implement only one variable condition

- if you want more variable conditions
  - package **java.util.concurrent.lock**
  - Declare one Lock (lock(), unlock()) + many conditions (signal(), await())
  - **Attention** : signal & await must necessarily be within an area protected by a lock

Nesteed monitor
or nesteed semaphore
or nesteed Lock

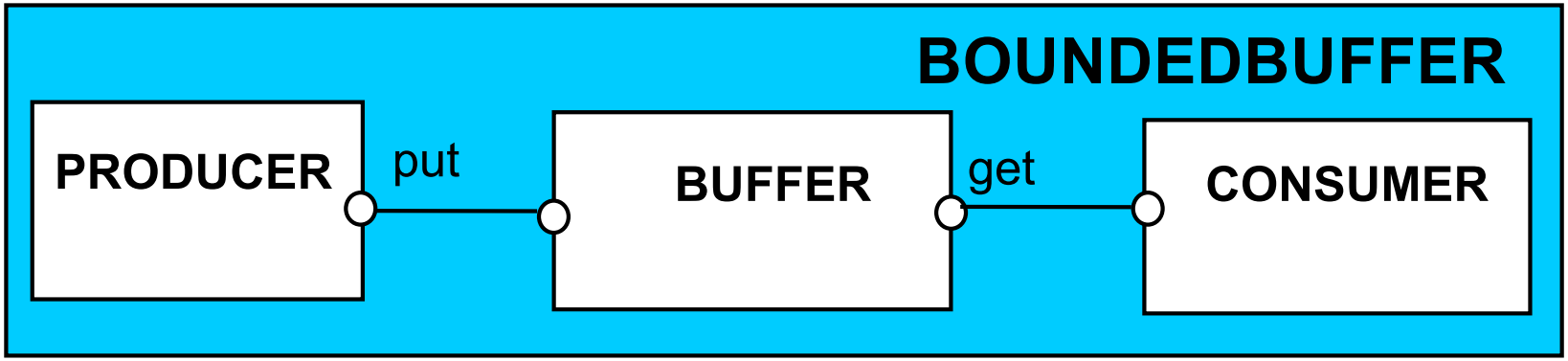
Bounded Buffer

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

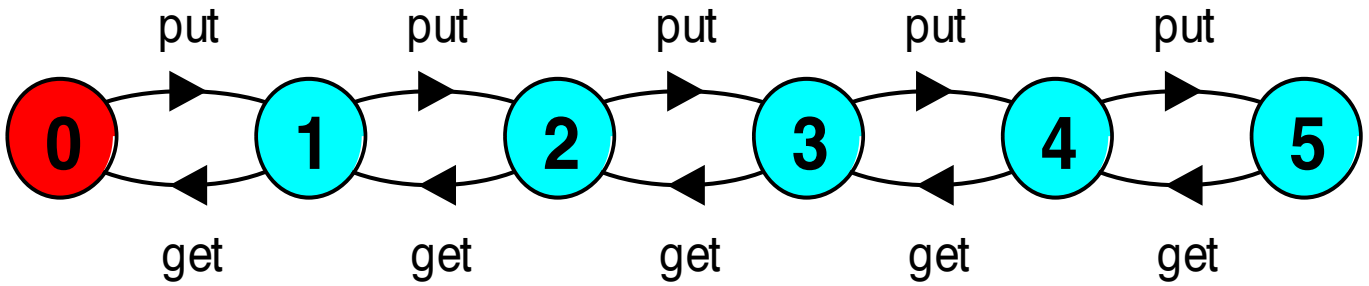


bounded buffer - a data-independent model

- 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

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

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

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

bounded buffer program - producer process

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

  Producer(Buffer b) {buf = b;}

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

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

bounded buffer program
Solution 1 : buffer monitor

```
public interface Buffer <E> {…}

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

}
```

bounded buffer program
Solution 2 : buffer semaphore

```
class BufferSemaphore <E> implements Buffer <E> {
    Semaphore full;   //counts number of items
    Semaphore empty; //counts number of spaces
    SemaBuffer(int size) {
        this.size = size; buf =(E[])new Object[size];
        full = new Semaphore(0);
        empty= new Semaphore(size);
    }
    public synchronized void put(E o) throws InterruptedException {
        empty.down();
        buf[in] = o;
        ++count; in=(in+1)% size;
        full.up();
    }
    public synchronized E get() throws InterruptedException {
        full.down();
        E o = buf[out]; buf[out]=null;
        --count; out=(out+1)% size;
        empty.up();
        return (o);
    }
}
```

nested synchronization - bounded buffer semaphore model

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

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

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

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

                      )@{put,get}.
```

Does this behave as desired?

Université Nice Sophia Antipolis

i3S sophia antipolis

nested monitors - bounded buffer model

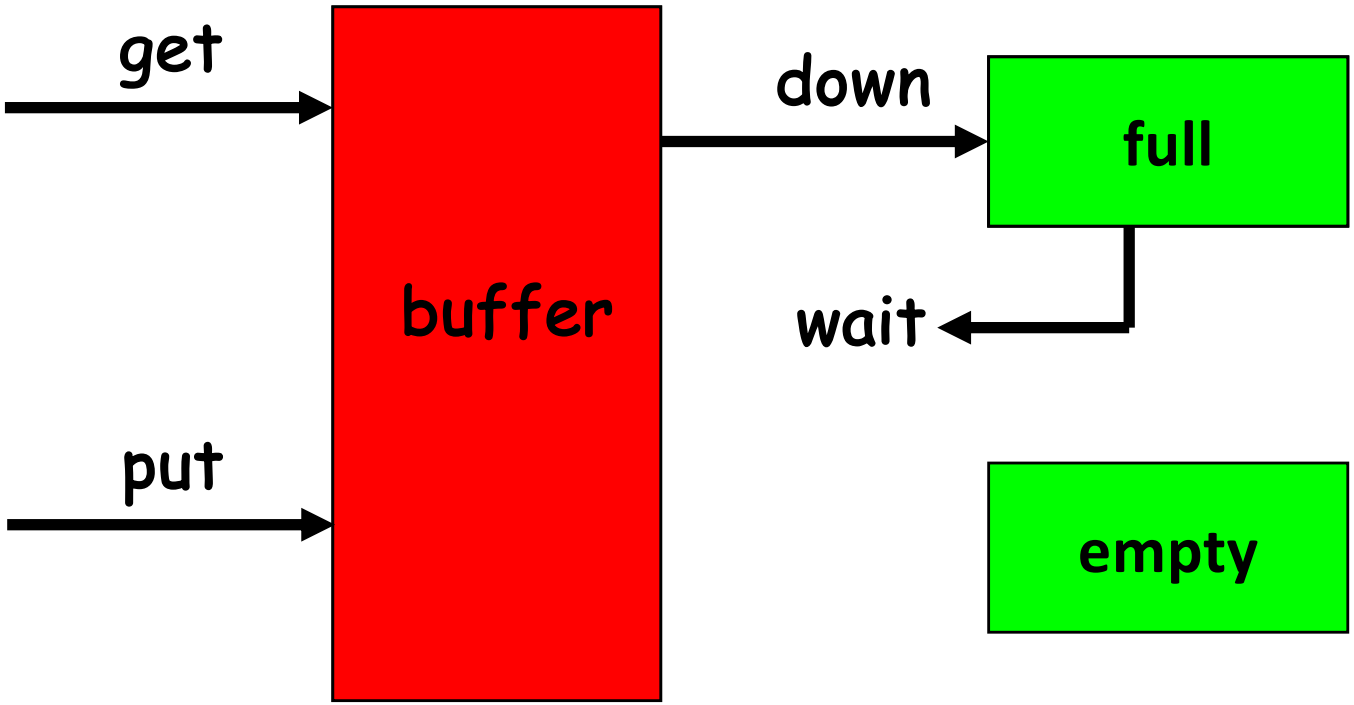
- *LTSA* analysis predicts a possible **DEADLOCK**:

```
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?***
- This situation is known as the *nested monitor problem*.
  - And occur when a monitor call another monitor

nested monitors - bounded buffer model

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



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

nested monitors - revised bounded buffer model

Incorrect implementation:

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

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

Correct implementation:

```
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

- 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?***

  Compose & minimise

  Composition:

  DEFAULT = CONSUMER || PRODUCER || BUFFER

  State Space: 3 * 3 * 1 = 2 ** 4

  Composing...

  -- States: 9 Transitions: 18 Memory used: 2899K

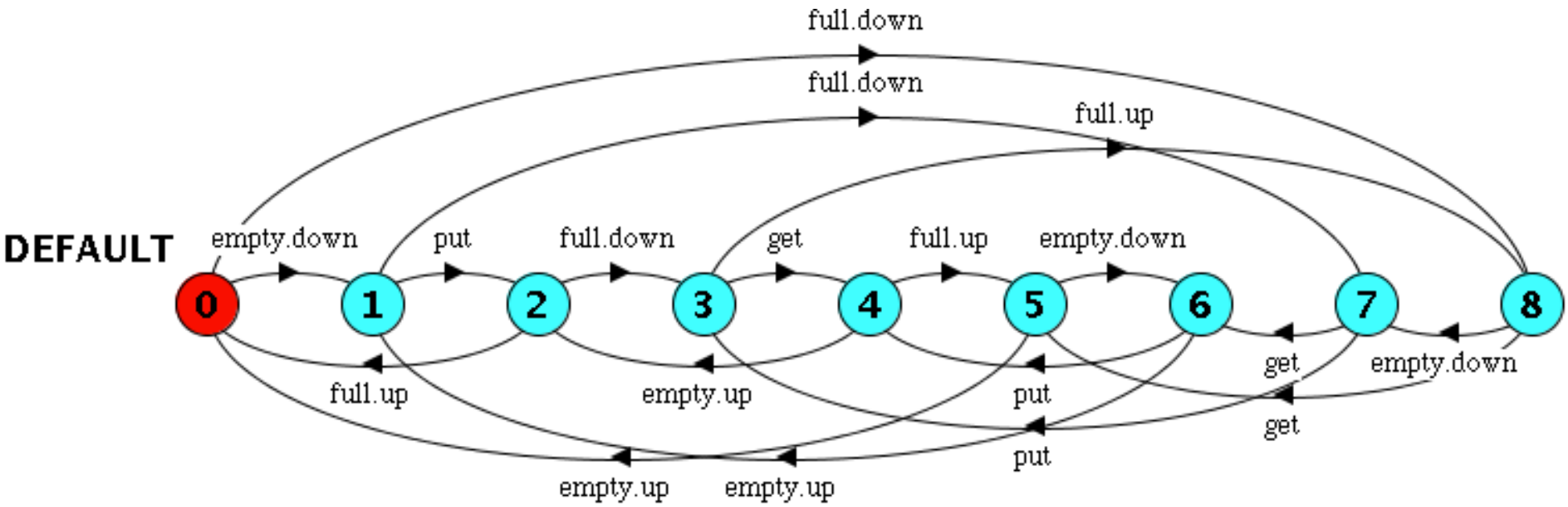
  Composed in 36ms

  DEFAULT minimising..

  Minimised States: 9 in 0ms

  No deadlock detected

nested monitors - revised bounded buffer model



Summary

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

Q&A

http://www.i3s.unice.fr/~riveill