Concurrent Programming
19530-V (WS01)

Lecture 6: Introduction to Monitors and Semaphores

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Abstracting Locking Details

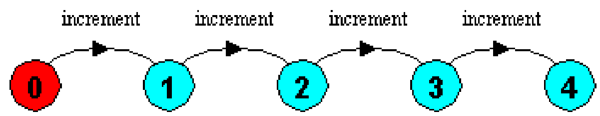
Recall our discussion of abstracting details

```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = (read[u]->VAR[u]
           |write[v:T]->VAR[v]).

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
             ->(when (x<N) write[x+1]
                 ->release->increment->INCREMENT
               )
             )+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```

We can hide the locking details of a shared resource by hiding its internal actions and only exposing the desired external actions (e.g., similar to public methods in an object)

Abstraction Leads to Simpler Model

<u>Minimized LTS for synchronized **COUNTER** process</u>



<u>A simpler process that also describes a synchronized counter</u>

```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment
                          ->COUNTER[v+1]).
```

*This process generates the same LTS as the previous **COUNTER** definition, thus they describe the same atomic increment behavior*

Benefits of Abstracted Model

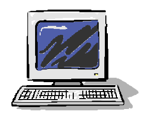
```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment
                          ->COUNTER[v+1]).
```

- Encapsulates state
  - The counter variable is no longer directly accessible
- Exposes only the allowable actions
  - In this case, the increment action
- Guarantees mutually exclusive access

- *This is the definition of a **monitor***

Monitor Concept

- A ***monitor*** is a high-level data abstraction mechanism for mutual exclusion
  - Monitors encapsulate state
  - Monitors provide operations to access and modify the state
    - These operations are the only means to modify the state
  - Monitors guarantee mutual exclusion among operations
    - Only one operation can execute at a time, thus the operation has exclusive access to the state
- ***Monitors sound very similar to what?***



Monitors as Java Classes

- Monitors are a data abstraction and classes in Java are also data abstractions
- It is possible to implement a monitor using a Java class by following two simple rules
  - All data members must be declared **`private`**
  - All methods that access data members must be declared **`synchronized`**
- Why is this high-level?
  - Because someone using the data encapsulated in the monitor does not need to worry about mutual exclusion issues at all

Monitor Example

```java
public class Counter {
   private int MAX = 5;
   private int count = 0;
   public Counter(int max)
      { MAX = max; }
   public synchronized void increment()
      { if (count < MAX) count++; }
   public synchronized void decrement()
      { if (count > 0) count--; }
   public synchronized int getCount()
      { return count; }
}
```

What are the semantics of this counter?

Counter Monitor Example

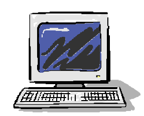
The counter in this example may ignore an increment or a decrement if the count is at the maximum or minimum, respectively.

How do we create a counter that does ***not*** ignore increments or decrements?

Naïve Monitor Solution

```
// Shared counter object
counter = new Counter(MAX);
...
// Try to make sure increment is not ignored
while (true) {
  if (counter.getCount() < MAX) {
    counter.increment();
    break;
  }
}
...
```

This fails because it is not atomic and even if it did work, it waste CPU cycles with busy waiting.



Condition Variable Concept

- Monitors are usually not used alone, but are combined with a low-level synchronization mechanism, called ***condition variables*** (also referred to as ***condition synchronization***)
- Condition variables
  - Support **`wait`** and **`notify`** operations, both can only be called from inside a monitor
    - This means that in order to use these operations, the caller must own the monitor lock!
  - When a process **`wait`**s on a condition variable, it gives up the lock and is suspended until another process performs a **`notify`** on the condition variable
  - Each condition variable has a ***waiting queue*** that can have any number of processes waiting on it

Condition Variables in Java

In Java, *every* object can be used as a condition variable

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

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

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

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

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

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

Condition Variables in FSP and Java

FSP

```
when (cond) act -> NEWSTAT
```

Java

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

The `while` loop in Java is necessary to re-test the wait condition to ensure that it is indeed satisfied when the thread re-enters the monitor.

`notifyAll()` is used to awaken other threads that may be waiting on the object instance's condition variable wait queue.

Blocked and Waiting Threads in Java

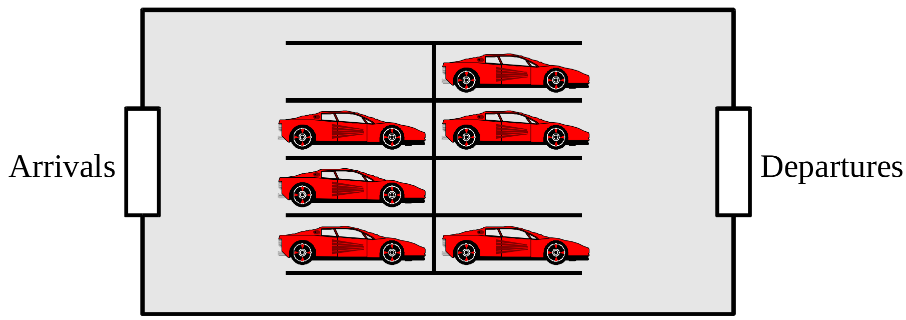
- If a thread is unable to enter a `synchronized` method because another threads owns the object's lock, then this thread is said to be blocked
  - Blocking and unblocking of threads is performed transparently, we do not worry about this
- If a thread owns an object's lock and calls `wait()` on that object, then that thread is said to be waiting on the object's wait queue
  - Adding and removing threads from the wait queue is specifically handled by the program using a combination of `wait()`/`notify()`/`notifyAll()` calls

Condition Variables in FSP and Java

Counter that does not ignore increments and decrements

```
public class StrictCounter extends Counter {
  ...
  public synchronized void increment() {
    while (getCount() >= MAX) { try { wait(); }
      catch (InterruptedException ex) { } }
    super.increment();
    notifyAll();
  }
  public synchronized void decrement() {
    while (getCount() <= 0) { try { wait(); }
      catch (InterruptedException ex) { } }
    super.decrement();
    notifyAll();
  }
}
```

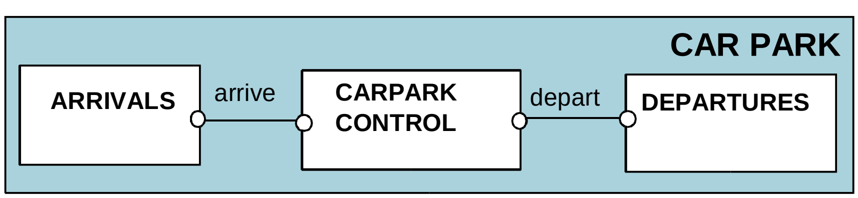
Car Park Example



A controller is required for a car park, which only permits cars to enter when the car park is not full and does not permit cars to leave when there are no cars in the car park.

Modeling the Car Park

- Actions of interest
  - **`arrive`** and **`depart`**
- Processes of interest
  - **`ARRIVALS`**, **`DEPARTURES`**, and **`CARPARKCONTROL`**
- Define processes and interactions (structure)



Car Park Model

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

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

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

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

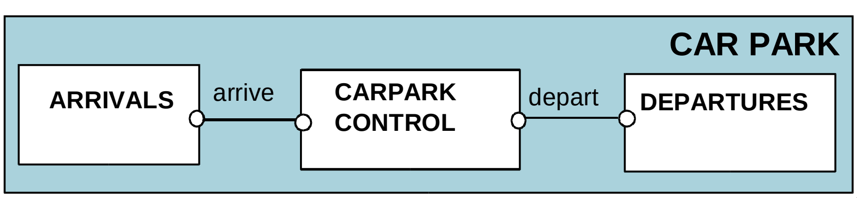
Car Park Program

In the FSP model all entities are processes interacting by actions

For the program we need to identify threads and monitors

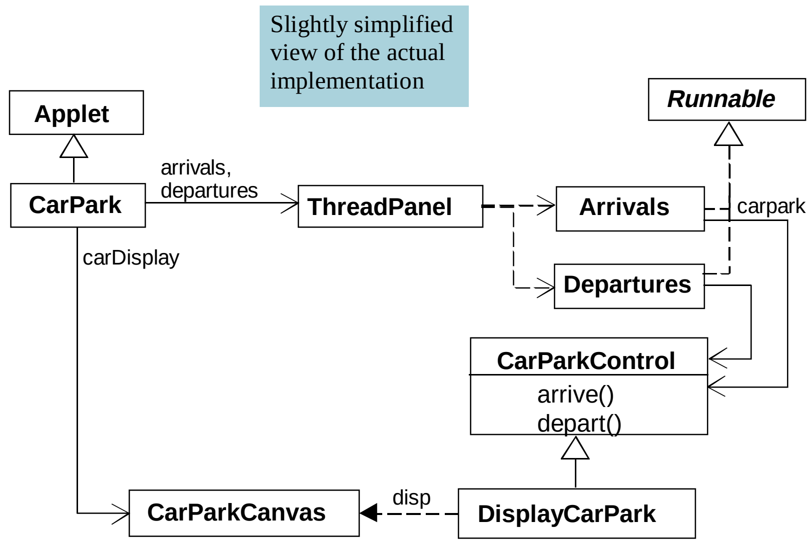
- ***Thread*** - active entity which initiates actions
- ***Monitor*** - passive entity which responds to actions

This is easy in the case of the car park...



Process *Monitor* *Process*

Car Park Class Diagram



Car Park Program

`Arrivals` and **`Departures`** implement **`Runnable`**, **`CarParkControl`** provides the control (condition synchronization).

Instances of these are created by the **`start()`** method of the **`CarPark`** applet

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

Car Park Arrival Thread

```
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){}
  }
}
```

`Departures` works similarly, except it calls **`depart()`**

How do we implement **`CarParkControl`**?

Car Park Control Monitor

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

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

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

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

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

Summary: Model to Monitor

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

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

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

Semaphores

Semaphores (Dijkstram 1968) are widely used for dealing with inter-process synchronization in operating systems. A semaphore *s* is an integer variable that can hold only non-negative values.

The only operations permitted on ***s*** are *up(s)* (**V** = vrijgeven = release) and *down(s)* (**P** = passeren = pass). Blocked processes are held in a FIFO queue.

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

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

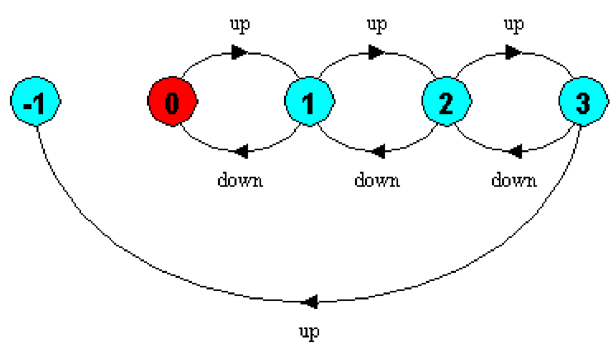
Modeling Semaphores

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

```
const Max = 3
range Int = 0..Max
SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]),
SEMA[Max+1]    = ERROR.
```

LTS?

Modeling Semaphores



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

Action **up** is not guarded.

Trace to a violation

up → **up** → **up** → **up**

Semaphore Example

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

```
LOOP = (mutex.down->critical->mutex.up->LOOP).
||SEMADEMO = (p[1..3]:LOOP
             ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

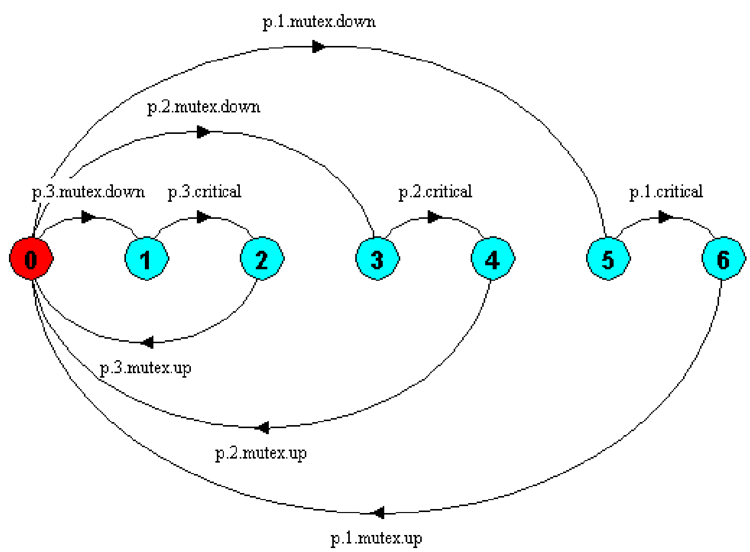
For mutual exclusion, the semaphore initial value is **1**. ***Why?***

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

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

LTS?

Semaphore Example



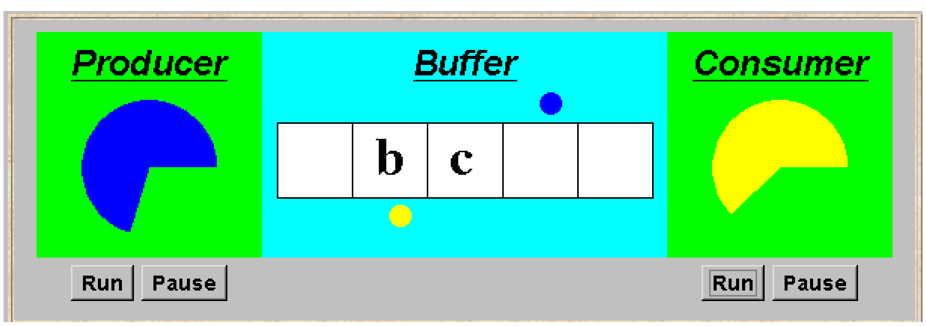
Semaphores in Java

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

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

```
public class Semaphore {
  private int value;

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

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

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

Bounded Buffer Example



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 Model

The behavior of **BOUNDEDBUFFER** is independent of the actual data values, and so can be modeled in a data-independent manner.

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

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

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

(see the Car Park example)

Bounded Buffer Monitor

```
public interface Buffer {…}

class BufferImpl implements Buffer {
     …

  public synchronized void put(Object o)
             throws InterruptedException {
    while (count == size) wait();
    buf[in] = o; ++count; in = (in+1) % size;
    notify();
  }

  public synchronized Object get()
             throws InterruptedException {
    while (count == 0) wait();
    Object o = buf[out];
    buf[out] = null; --count; out = (out+1) % size;
    notify();
    return (o);
  }
}
```

We create a separate buffer interface to permit alternative implementations.

Bounded Buffer Monitor

```
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(new Character(alphabet.charAt(ai)));
        ai = (ai+1) % alphabet.length();
        ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }
}
```

`Consumer` *is similar but calls* ***`buf.get()`****.*

Alternative Bounded Buffer

Suppose that, in place of using the **`count`** variable and condition synchronization directly, we instead use two semaphores **`full`** and **`empty`** to reflect the state of the buffer.

```
class SemaBuffer implements Buffer {
  …

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

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

Alternative Bounded Buffer

empty is decremented during the put() operation, which is blocked if empty is zero; full is decremented by the get() operation, which is blocked if full is zero.

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

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

Alternative Bounded Buffer 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?

Nested Monitor Problem

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*****.**

Nested Monitor Program Fix

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

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

Nested Monitor Model Fix

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

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

Does this behave as desired?