Monitors & Condition Synchronization

INF2140 Parallel Programming: Lecture 5

Feb. 15, 2012

Monitors & condition Synchronization

Concepts: monitors:

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

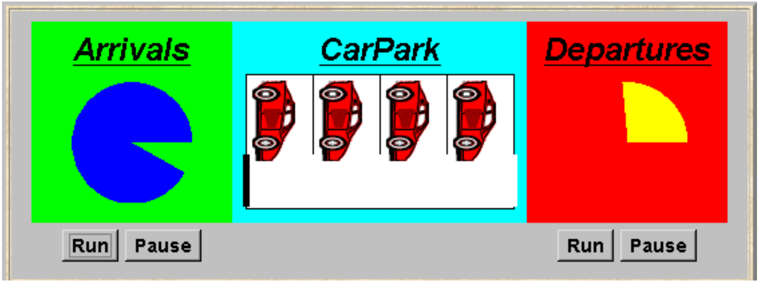
Models: guarded actions

Practice: private data and synchronized methods (exclusion).

- wait(), notify() and notifyAll() for condition synch.

single thread active in the monitor at a time

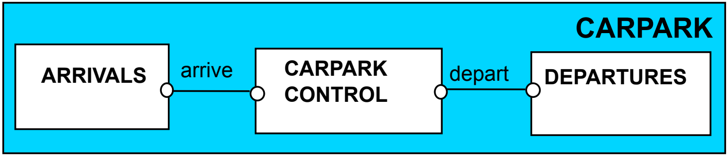
5.1 Condition synchronization



A controller is required for a carpark, which only permits cars to enter when the carpark is not full 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 process and interactions (structure).



carpark model

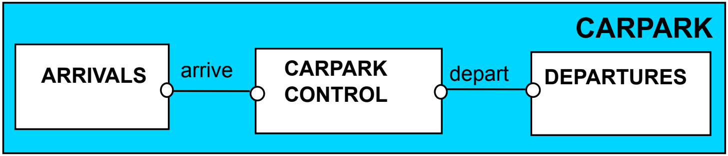
```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 | when(i<N) depart->SPACES[i+1]).

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

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

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

carpark program

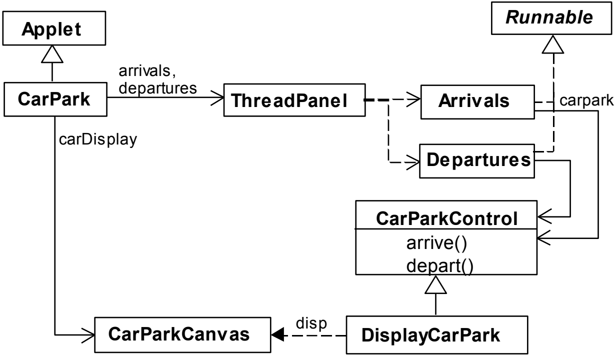
- *Model* - all entities are **processes** interacting by actions
- *Program* - need to identify **threads** and **monitors**
  - *thread* - **active** entity which initiates (output) actions
  - *monitor* - **passive** entity which responds to (input) actions.

For the carpark?



carpark program - class diagram

We omit DisplayThread and GraphicCanvas threads managed by ThreadPanel.



carpark program

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

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

carpark program - Arrivals and Departures threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;
  Arrivals(CarParkControl c) {carpark = c;}
  public void run() { try {
      while(true) {
        ThreadPanel.rotate(330);
        carpark.arrive();
        ThreadPanel.rotate(30); }
    } catch (InterruptedException e){}
  }
}
```

Similarly: `Departures` which call `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 excl. by synch. methods
- Condition synchronization?
- Block if full? (spaces==0)
- Block if empty? (spaces==N)

condition synchronization in Java

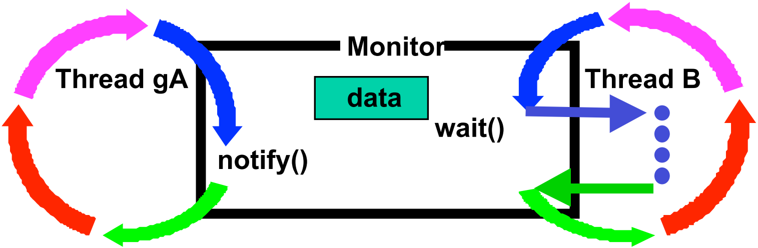
Java provides a thread **wait set** per monitor (actually per object) with the following methods:

- **public final void notify()**
  Wakes up a single thread waiting on this object's wait set.
- **public final void notifyAll()**
  Wakes up all threads that are waiting on this object's wait set.
- **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.

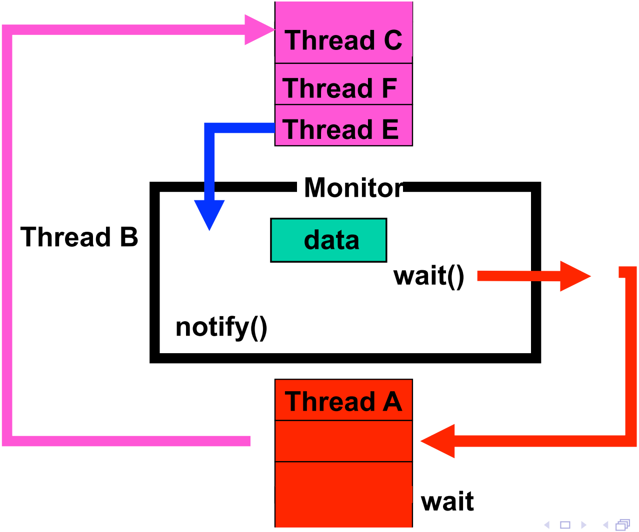
condition synchronization in Java

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.



Monitor locking by wait and notify



condition synchronization in Java

FSP : **when cond act -> NEWSTAT**

Java:

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

- The **while** loop is necessary to retest **cond** to ensure that **cond** is indeed satisfied when it re-enters the monitor.
- **notifyAll()** is necessary to wake other threads waiting to enter the monitor – now that the monitor has been changed.

CarParkControl – condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;
  CarParkControl(int n){capacity =spaces =n;}
  synchronized void arrive()
    throws InterruptedException {
    while (spaces==0) wait();
    --spaces; notifyAll(); }
  synchronized void depart()
    throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces; notifyAll(); }
}
```

- Is it safe to use *notify()* here, rather than *notifyAll()*?

models to monitors - summary

Active entities (initiating actions) are implemented as **threads**.

Passive entities (responding 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()**.

5.2 Semaphores

- Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore **s** is like an integer variable that can take only non-negative 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** decrement s **else** block execution of the calling process

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

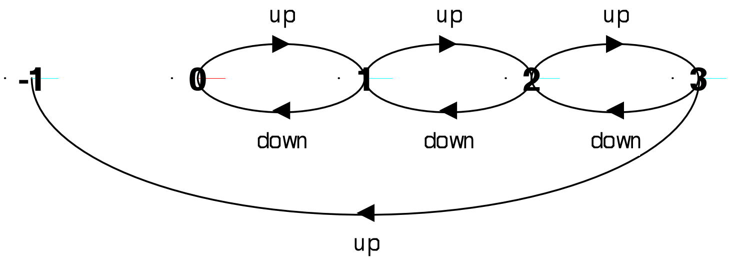
modelling semaphores

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

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

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

LTS?

modelling semaphores



- Action down is only accepted when value v of the semaphore is greater than 0.
- Action up is not guarded.
- Trace to a violation:

```
up -> up -> up -> up
```

semaphore demo - model

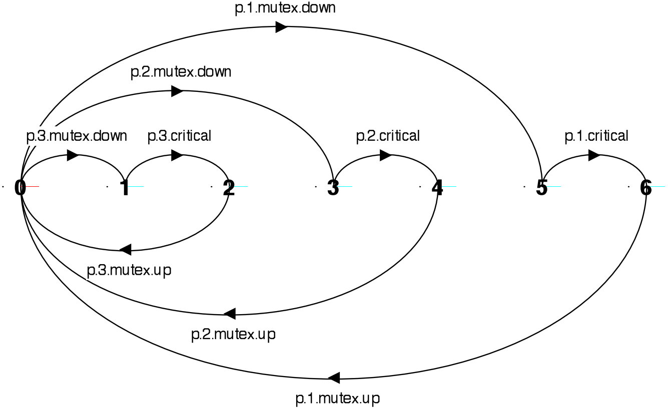
Three processes **p[1..3]** use a shared semaphore **mutex** to ensure mutually exclusive access (action **critical**) to some resource.

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

- For mutual exclusion, the semaphore initial value is 1. **Why?**
- Is the **ERROR** state reachable for SEMADEMO?
- Is a **binary** semaphore sufficient (i.e. Max=1) ?

LTS?

semaphore demo - model



semaphore demo - testing the model

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP).
||SEMADEMO = (p[1..3]:  LOOP
            || {p[1..3]}::mutex:SEMAPHORE(2)
            || {p[1..3]}::TEST ).
TEST = ({mutex.down,mutex.up} -> TEST
      | critical -> (critical -> ERROR
                    | {mutex.down,mutex.up} -> TEST)).
```

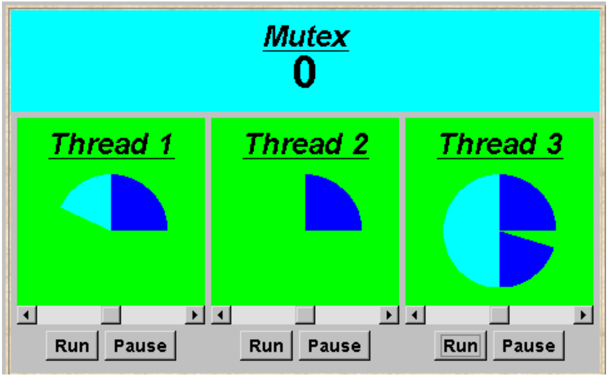
Try with SEMAPHORE(1) and SEMAPHORE(2).

semaphores in Java

```java
public class Semaphore {
  private int value;

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

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

- 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.)
- Is it safe to use notify() here rather than notifyAll()?

SEMADEMO display



5.5 Monitor Invariants

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

- CarParkControl Invariant: 0 <= space
- Semaphore) Invariant: 0 <= value

Invariants can be helpful in understanding a monitor, and also in reasoning about correctness of monitors using a logical proof-based approach. In this course we use a model-based approach amenable to mechanical checking.

Summary

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