Monitors, Condition Synchronization and Semaphores

CS 2SD3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Abstract Monitors and Condition Synchronization

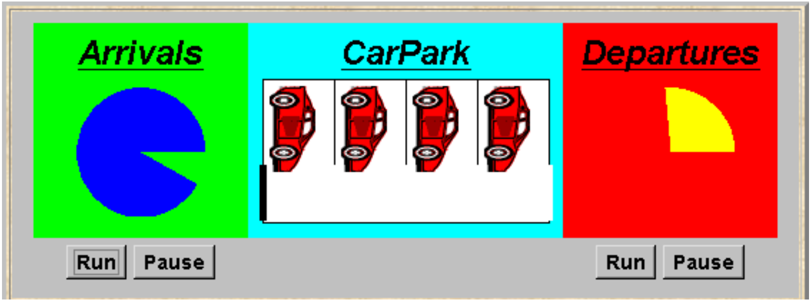
Concepts: **monitors**:

encapsulated data + access procedures
mutual exclusion + condition synchronization
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

Condition Synchronization: Carpark Model



A controller is required for a carpark, which only permits cars to enter when the carpark is not full and permits cars to leave when there it is not empty. Car arrival and departure are simulated by separate threads.

Carpark Model

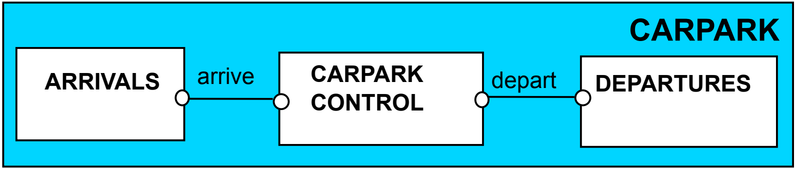
- Events or actions of interest?

  arrive and **depart**

- Identify processes.

  arrivals, departures and carpark control

- Define each process and interactions (structure).



```
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).
```

$SPACES[i:0..N]$ expands to:

$SPACES[0] = (depart \rightarrow SPACES[1]$
$SPACES[1] = (arrive \rightarrow SPACES[0] \mid depart \rightarrow SPACES[2])$
$SPACES[2] = (arrive \rightarrow SPACES[1] \mid depart \rightarrow SPACES[3])$
$SPACES[3] = (arrive \rightarrow SPACES[2] \mid depart \rightarrow SPACES[4])$
$SPACES[4] = (arrive \rightarrow SPACES[3]$

- Blue expansion is used when LTS is produced! This is also and abstract **monitor**
- Guarded actions are used to control *arrive* and *depart*

- A process is **active** if it initiates (output) actions.
- A process is **passive** if it responds to (input) actions. *Abstract monitors* (use *Guarded Actions*) are passive processes.

Semaphores

- Semaphores are widely used for dealing with inter-process synchronization in operating systems.
  In fact, anywhere where there is a restriction, at some point of processing, to having only one operating entity, like for instance *one processor*.
- Semaphore s is an integer variable that can take only non-negative values.
- The only semaphore operations are $down(s)$ ($wait(s)$, $V(s)$) and $up(s)$ ($signal(s)$, $P(s)$).
- In the model presented in the textbook, blocked processes are held in FIFO queue. In standard theoretical model the are held in a set (and choice releasing is non-deterministic). In general any protocol for releasing is allowed.

***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

- *Binary Semaphores*

down(s): **if** $s = 1$ **then** $s = 0$
else *block execution of the calling process*
up(s): **if** *process blocked on s* **then** *awaken one of them*
else $s = 1$

Modeling Semaphores with FSP

- Since the semantics of FSP is via LTS, we can only model semaphores that take a finite range of values.
- If this range is exceeded then we regard this as an **error**.
- This may not be true in other models!
- 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.
```

Modeling Semaphores with FSP

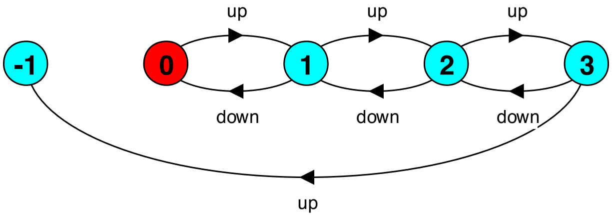
```
const Max = 3
range Int = 0..Max

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

It expands to:

$SEMA[0] = (up \rightarrow SEMA[1])$
$SEMA[1] = (up \rightarrow SEMA[2] \mid down \rightarrow SEMA[0])$
$SEMA[2] = (up \rightarrow SEMA[3] \mid down \rightarrow SEMA[1])$
$SEMA[3] = (up \rightarrow SEMA[4] \mid down \rightarrow SEMA[2])$
$SEMA[4] = ERROR$

- Using *ERROR* is questionable!
- What not: $SEMA[3] = (down \rightarrow SEMA[2])$!!

$$
\begin{aligned}
SEMA[0] &= (up \rightarrow SEMA[1]) \\
SEMA[1] &= (up \rightarrow SEMA[2] \mid down \rightarrow SEMA[0]) \\
SEMA[2] &= (up \rightarrow SEMA[3] \mid down \rightarrow SEMA[1]) \\
SEMA[3] &= (up \rightarrow SEMA[4] \mid down \rightarrow SEMA[2]) \\
SEMA[4] &= ERROR
\end{aligned}
$$



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

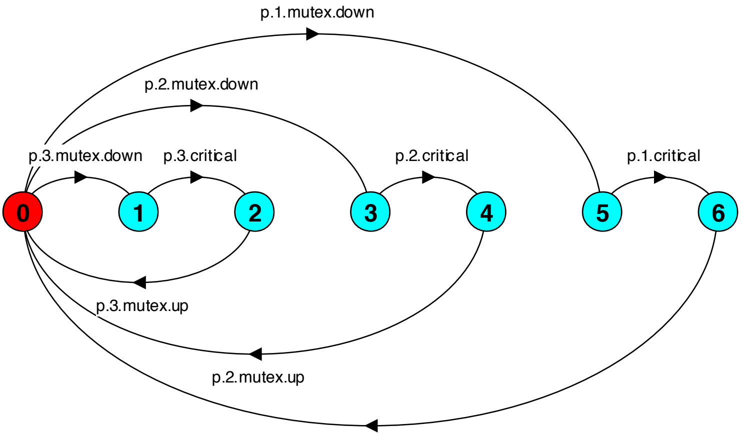
Mutex and Binary Semaphores

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.
- Binary semaphores are sufficient in this case, actually mutual exclusion provided main motivation to Edsgar Dijkstra for invention of semaphores in 1958 (actually he implemented what was first used for rail tracks in 19 century).

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

||SEMADEMO = (p[1..3]:LOOP
             ||{p[1..3]}::mutex:SEMAPHORE(1)).
```



Semaphores in Java (in Textbook)

- In Textbook approach (presumably for didactic reasons), semaphores are passive objects, therefore implemented as Java monitors.
- In practice, semaphores are a low-level mechanism often used for implementing the higher-level monitor, or other, constructs.
- Semaphores are often implemented in Assembler or even by hardware.