Semaphores

Dr. José Luis Zechinelli Martini

joseluis.zechinelli@udlap.mx

LDS-1101

Based on the course of Professor Carlo A. Furia, Chalmers University of Technology – University of Gothenburg

Agenda

- Semaphores:
  - Semaphores for permissions
  - Mutual exclusion for two processes
  - Weak vs. strong semaphores
  - Invariants
  - Binary semaphores
  - Using semaphores in Java

Semaphores

- A (general/counting) semaphore is a data structure with interface:

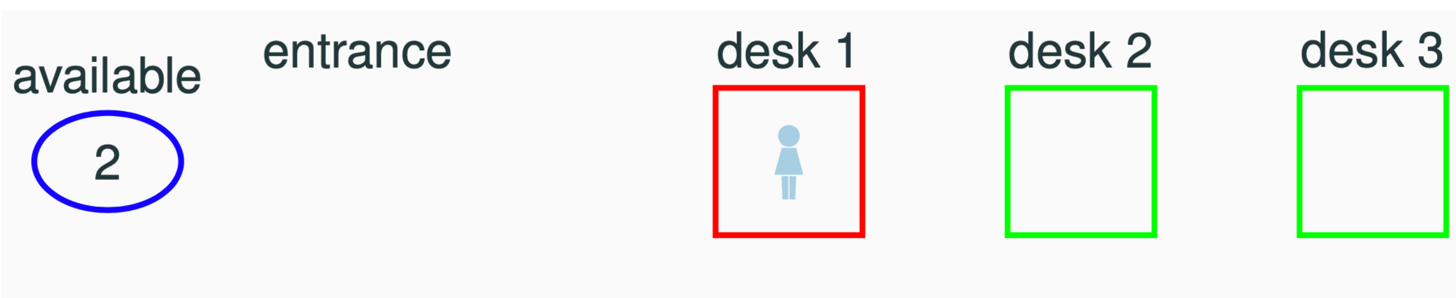
```
interface Semaphore {
    int count();      // current value of counter
    void up();        // increment counter
    void down();      // decrement counter
}
```

- Several threads share the same object **sem** of type Semaphore:
  - Initially count is set to a nonnegative value C, *i.e.*, the capacity
  - A call to sem.up() atomically increments count by one
  - A call to sem.down() waits until count is positive, and then atomically decrements count by one

Semaphores for permissions

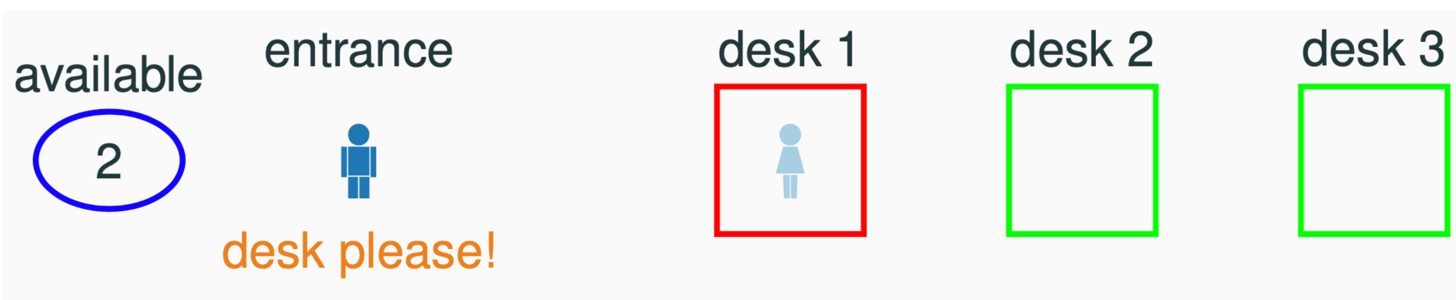
- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available

- Example: Hot Desks



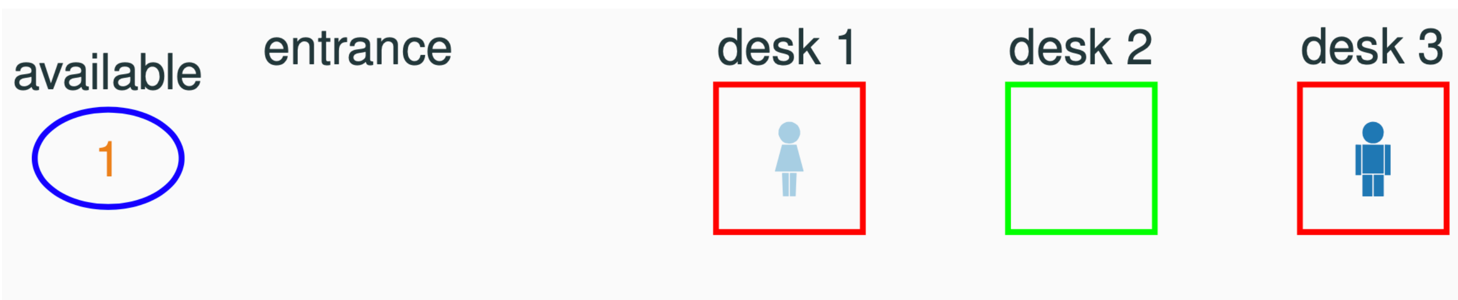
Semaphores for permissions

- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



Semaphores for permissions

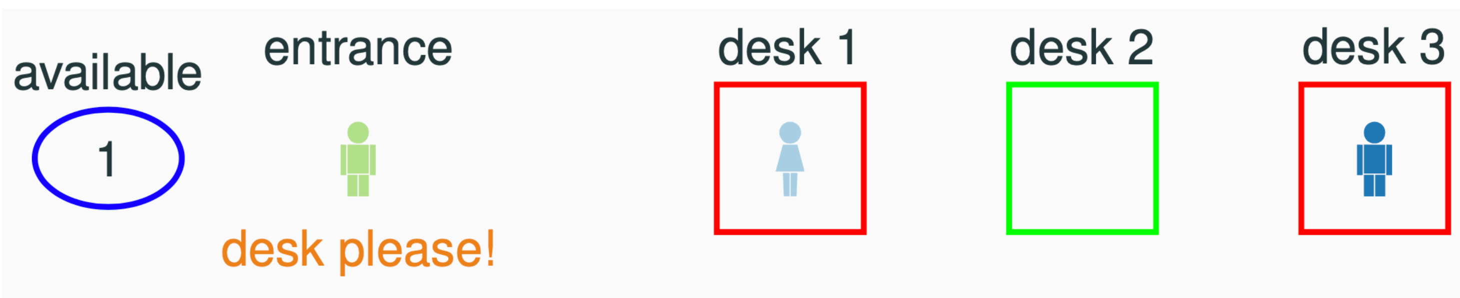
- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



Semaphores for permissions

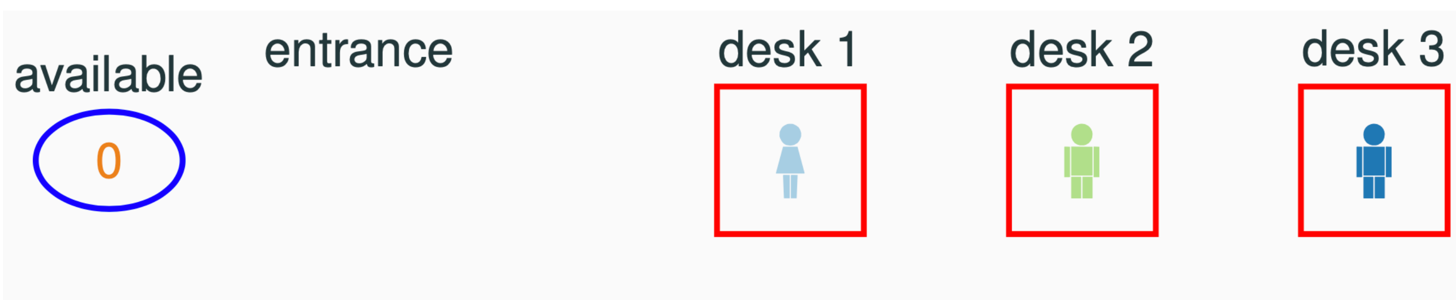
- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available

- Example: Hot Desks



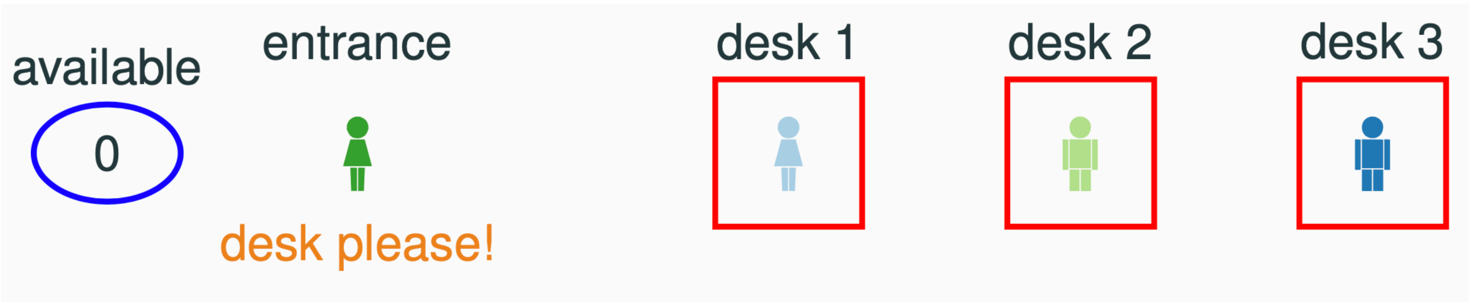
Semaphores for permissions

- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



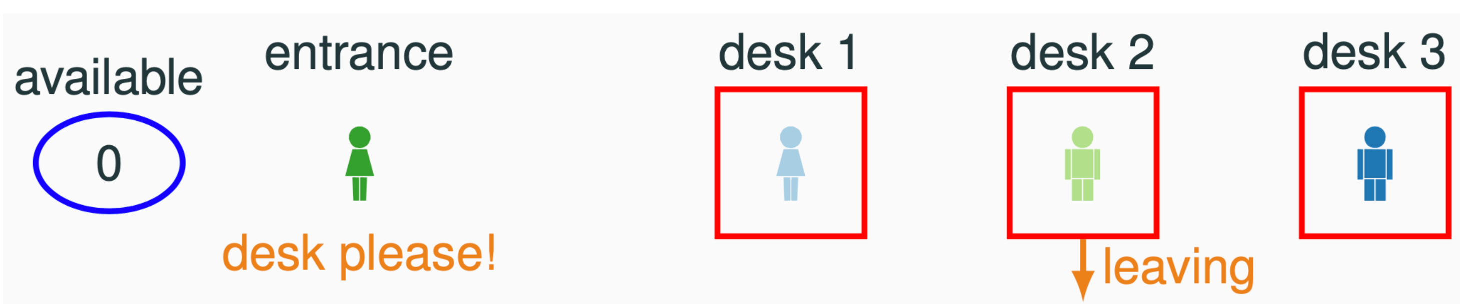
Semaphores for permissions

- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



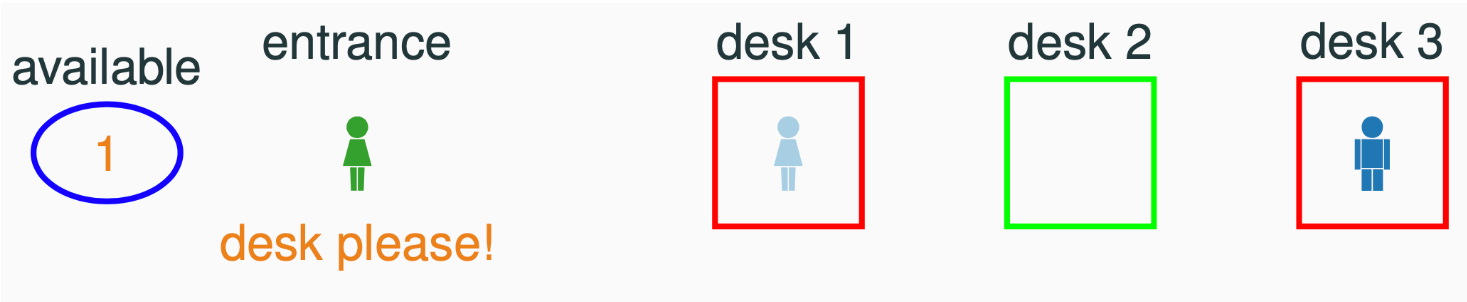
Semaphores for permissions

- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



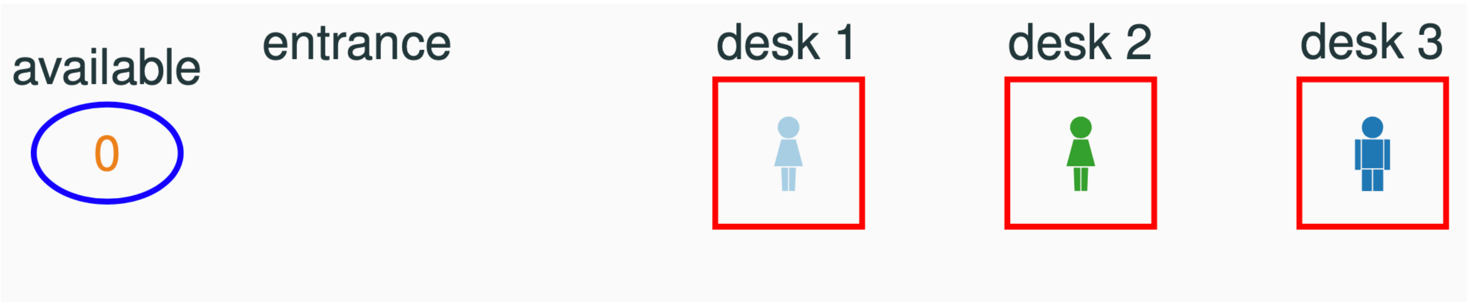
Semaphores for permissions

- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



Semaphores for permissions

- A semaphore is often used to regulate access permits to a finite number of resources:
  - The capacity C is the number of initially available resources
  - Up (also called signal) releases a resource, which becomes available
  - Down (also called wait) acquires a resource if it is available
- Example: Hot Desks



Agenda

- Semaphores:
  - ✓ Semaphores for permissions
  - Mutual exclusion for two processes
  - Weak vs. strong semaphores
  - Invariants
  - Binary semaphores
  - Using semaphores in Java

Mutual exclusion for two processes with semaphores

- With semaphores the entry/exit protocols are trivial:
  - Initialize semaphore to 1
  - Entry protocol: call sem.down()
  - Exit protocol: call sem.up()

```
Semaphore sem = new Semaphore(1);
```

thread t	thread u
`int cnt;`	`int cnt;`
`sem.down();`	`sem.down();`
`  cnt = counter;`	`  cnt = counter;`
`  counter = cnt + 1;`	`  counter = cnt + 1;`
`sem.up();`	`sem.up();`

- The implementation of the Semaphore interface guarantees mutual exclusion, deadlock freedom, and starvation freedom

Weak vs. strong semaphores

- Every implementation of semaphores should guarantee the atomicity of the up and down operations, as well as deadlock freedom:
  - For threads only sharing one semaphore, deadlocks may still occur if there are other synchronization constraints

- Fairness is optional:
  - **Weak semaphore**: Threads waiting to perform down are scheduled no deterministically
  - **Strong semaphore**: Threads waiting to perform down are scheduled fairly in FIFO (First In First Out) order

Agenda

- Semaphores:
  - ✓ Semaphores for permissions
  - ✓ Mutual exclusion for two processes
  - ✓ Weak vs. strong semaphores
  - ☐ Invariants
  - ☐ Binary semaphores
  - ☐ Using semaphores in Java

Invariants

- An object's invariant is a property that always holds between calls to the object's methods:
  - The invariant holds initially (when the object is created)
  - Every method call starts in a state that satisfies the invariant
  - Every method call ends in a state that satisfies the invariant
- For example: A bank account that cannot be overdrawn has an invariant (balance >= 0)

```
class BankAccount {
    int balance = 0;
    void deposit(int amount)
        { if (amount > 0) balance += amount; }
    void withdraw(int amount)
        { if (amount > 0 && balance > amount) balance -= amount; }
}
```

Invariants in pseudo-code

- We occasionally annotate classes with invariants using the pseudo-code keyword invariant
- Note that invariant is not a valid Java keyword — that is why we highlight it in a different color — but we will use it whenever it helps make more explicit the behavior of classes

```
class BankAccount {
    int balance = 0;
    void deposit(int amount)
        { if (amount > 0) balance += amount; }
    void withdraw(int amount)
        { if (amount > 0 && balance > amount) balance -= amount; }
    invariant { balance >= 0; }   // not valid Java code
}
```

Invariants of semaphores

- A semaphore object with capacity C satisfies the invariant:

```
interface Semaphore {
    int count();
    void up();
    void down();
    invariant {
        count() >= 0;
        count() == C + #up - #down;
    }
}
```

Number of calls to down

Number of calls to up

- Invariants characterize the behavior of an object, and are very useful for proofs

Agenda

- Semaphores:
  - ✓ Semaphores for permissions
  - ✓ Mutual exclusion for two processes
  - ✓ Weak vs. strong semaphores
  - ✓ Invariants
  - ☐ Binary semaphores
  - ☐ Using semaphores in Java

Binary semaphores

- A semaphore with capacity 1 and operated such that count() is always at most 1 is called a binary semaphore:

```
interface BinarySemaphore extends Semaphore {
    invariant
    {  0<=count()<=1;
       count() == C + #up - #down;    }
}
```

- Mutual exclusion uses a binary semaphore:

```
Semaphore sem = new Semaphore(1); // shared by all threads
```

thread t

```
sem.down();
  // critical section
sem.up();
```

- If the semaphore is strong this guarantees starvation freedom

Binary semaphores vs. locks

- Binary semaphore are very similar to locks with one difference:
  - In a lock, only the thread that decrements the counter to 0 can increment it back to 1
  - In a semaphore, a thread may decrement the counter to 0 and then let another thread increment it to 1

- Thus (binary) semaphores support transferring of permissions

Agenda

- Semaphores:
  - ✓ Semaphores for permissions
  - ✓ Mutual exclusion for two processes
  - ✓ Weak vs. strong semaphores
  - ✓ Invariants
  - ✓ Binary semaphores
  - ☐ Using semaphores in Java

Using semaphores in Java

- Method acquire may throw an InterruptedException (catch or propagate):

```
package java.util.concurrent;

public class Semaphore {
    Semaphore(int permits);                 // initialize with capacity permits
    Semaphore(int permits, boolean fair);           // fair ⇔ fair semaphore
                // fair == true also called ‘strong’ semaphore
                // fair == false also called ‘weak’ semaphore
    void acquire();                         // corresponds to down
    void release();                         // corresponds to up
    int availablePermits();                 // corresponds to count
}
```