

Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

Part 2: Process management

Seminar 6

Synchronization: POSIX Semaphores

fSO

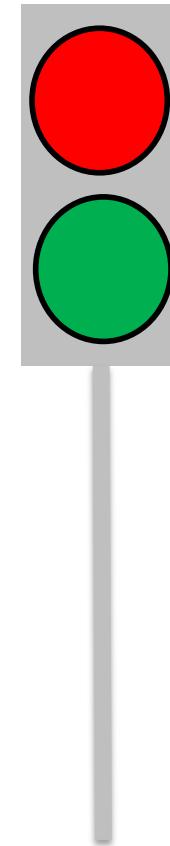
DISCA



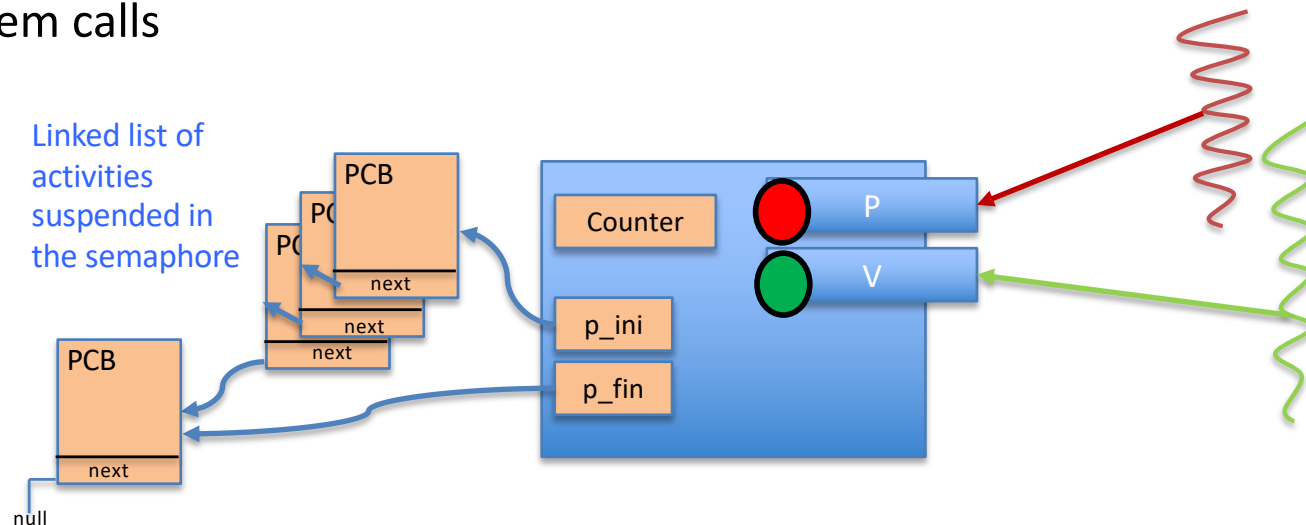
UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

- **Goals:**
 - To get use to deal with **critical section** problems
 - To know the **synchronization** mechanisms **offered by the OS**
 - To use **semaphores** and **mutexes** to solve critical section synchronization
- **Bibliography**
 - Silberschatz 8th Ed, chapter 6
 - Robbins, chapters 13, 14

- **OS level solutions**
- POSIX semaphores
- POSIX mutexes
- Exercises



- Semaphore
 - Can be seen as an integer that admits an increment by 1 and a decrement by 1 operations performed by an activity (process or thread)
 - The decrement operation can suspend the activity
 - The increment operation can awake another activity previously suspended
 - It is a synchronization object offered by the OS to user activities
 - It is declared as type “semaphore” specifying its initial value
 - The increment (V) and the decrement (P) operations are implemented as system calls



Introduced by E.W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. **Two operation: up and down. Later it proposes P and V** *Proberen* (try) and *Verhogen* (raise, make higher).

- Initialization, P and V specification

- Declaration and initialization

Semaphore S(N);

- It declares a semaphore “S” with “N” as initial value
- “N” has to be greater or equal to zero

- DOWN (decrement)

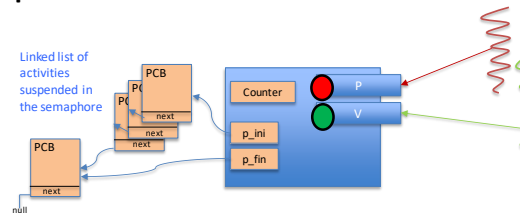
P(S);

```
S = S - 1;
if S < 0 then suspend(S);
```

- UP (increment)

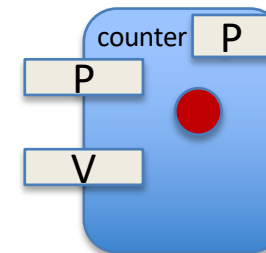
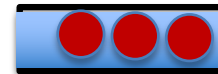
V(S);

```
S = S + 1;
if S <= 0 then awake(S);
```



Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**.

entry queue



Notes:

- P and V are guaranteed to be atomic by the OS
- suspend(S) suspends the calling activity in a queue associated to S
- awake(S) extracts an activity from the S queue and awakes it (it goes to the scheduler ready queue)

- Solving the critical section problem
 - We define as a global variable the semaphore “sem”, initialized to 1
- Semaphore sem(1);**
- Code for N threads/processes

```
void *thread_i(void *p) {  
  
    while(1) {  
        P(sem);  
  
        /* Critical section */  
  
        V(sem);  
  
        /* Remaining section */  
  
    }  
}
```

It complies with:

- Mutual exclusion
- Progress
- Limited waiting, if the semaphore queue is scheduled with FCFS policy

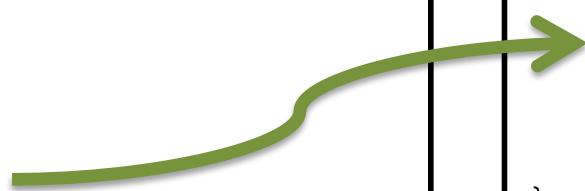
- Another synchronization use of semaphores:
Establishing an execution order
 - We want “thread1” to execute function “F1” before “thread2” executes function “F2”
 - We define a shared semaphore between “thread1” and “thread2” named “sync”, **initialized to 0**

Semaphore **sync(0);**

```
void *thread1(void *p)
{
    ...

    F1;
    V(sync);
    ...
}
```

```
void *thread2(void *p)
{
    ...
    P(sync);
    F2;
    ...
}
```



- Another synchronization use of semaphores: Limiting the number of activities that can pass through a certain point in the code simultaneously
 - We have a function executed by many threads
 - We want a maximum of 5 threads call function “F” located in a certain place inside the function
 - We define a shared semaphore “max_5”, **initialized to 5**

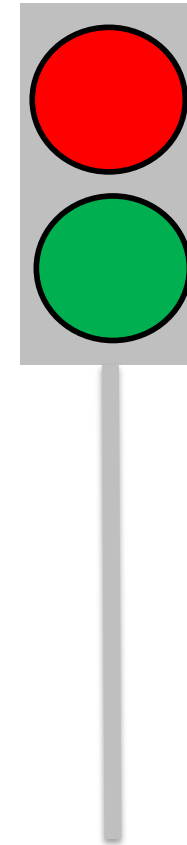
Semaphore max_5(5);

```
.....  
pthread_t th1, th2, th3, th4, th5;  
pthread_attr_t attr;  
  
pthread_attr_init(&attr);  
pthread_create(&th1, &attr, thread_i NULL);  
pthread_create(&th2, &attr, thread_i NULL);  
pthread_create(&th3, &attr, thread_i NULL);  
pthread_create(&th4, &attr, thread_i NULL);  
.....
```

```
void *thread_i(void *p) {  
  
    ...  
    P(max_5);  
    F;  
    V(max_5);  
    ...  
}
```


- **Basically a semaphore is a “resource counter”**
 - When an activity tries to use a resource, it decrements the counter (*P* operation) and if there are not available resources then the activity is suspended
 - When an activity finishes using a resource then it increments the counter (*V* operation). If there is any suspended activity waiting for the resource, then the *V* operation awakes it.
 - If the counter is greater than 0 then its value indicates the number of available resources
if $S > 0$ then $|S|$ = number of available resources
 - When its value is less than zero its absolute value indicates the number of suspended activities in the semaphore queue
if $S < 0$ then $|S|$ = number of suspended processes
 - When the counter is zero both of the former sentences are true
**if $S = 0$ then \rightarrow there are no suspended processes
there are no available resources**
- With semaphores the OS manages suspended activity queues, waiting for available resources
- Activity synchronization is one of the essential mechanisms that an OS has to provide

- OS level solutions
- **POSIX semaphores**
- POSIX mutexes
- Exercises



- POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero.
- Two operations can be performed on semaphores: increment the semaphore value by one (**sem_post**(3)); and decrement the semaphore value by one (**sem_wait**(3)). If the value of a semaphore is currently zero, then a **sem_wait**(3) operation will block until the value becomes greater than zero.
- **sem_init**(sem_t *sem, int pshared, unsigned int value); initialises the semaphore sem.
 - **sem** : Specifies the semaphore to be initialized.
 - **pshared** : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
 - **value** : Specifies the value to assign to the newly initialized semaphore.
- **Semaphore types:**
 - **Named semaphores** A named semaphore is identified by a name of the form /somename; Two processes can operate on the same named semaphore by passing the same name to **sem_open**(3). The **sem_open**(3) function creates a new named semaphore or opens an existing named semaphore.
 - **Unnamed semaphores (memory-based semaphores)** An unnamed semaphore does not have a name. Instead the semaphore is placed in a region of memory that is shared between multiple threads (a *thread-shared semaphore*) or processes (a *process-shared semaphore*). Global variable.

- POSIX.1b introduced the semaphore type `sem_t`

```
#include <semaphore.h>
sem_t sem;
```

- A semaphore can be used by all the threads inside a process and also can be shared between processes.
 - After a `fork()` call semaphores in the parent can be inherited by the child depending on “pshared” parameter value in “`sem_init`”. Not implemented. Value = 0.

- POSIX.1b semaphore operations are:

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- `int sem_destroy(sem_t *sem);`
- `int sem_wait(sem_t *sem);`
- `int sem_trywait(sem_t *sem);`
- `int sem_post(sem_t *sem);`
- `int sem_getvalue(sem_t *sem, int *sval);`

Operation
P(sem)

Operation
V(sem)

- Producer/Consumer **version 1**
 - We define a semaphore “sem” initialized to 1 → **sem_init(&sem,0,1);**

```
void *func_prod(void *p) {
    int item;

    while(1) {
        item = produce();

        sem_wait(&sem);

        while (counter == N)
            /*empty loop*/ ;
        buffer[input] = item;
        input = (input + 1) % N;
        counter = counter + 1;

        sem_post(&sem);
    }
}
```

```
void *func_cons(void *p) {
    int item;

    while(1) {
        sem_wait(&sem);

        while (counter == 0)
            /*empty loop*/ ;
        item = buffer[output];
        output = (output + 1) % N;
        counter = counter - 1;

        sem_post(&sem);

        consume(item);
    }
}
```

- If the producer enters the critical section being the buffer full it will get into the while loop forever, the same happens with the consumer when the buffer is empty

- Producer/Consumer **version 2**

```
#include <semaphore.h>
sem_t sem, empty, full;
```

```
void *func_prod(void *p) {
    int item;
    while(1) {
        item = produce();
        sem_wait(&empty);
        sem_wait(&sem);

        buffer[input] = item;
        input = (input + 1) % N;
        counter = counter + 1;

        sem_post(&sem);
        sem_post(&full);
    }
}
```

```
void *func_cons(void *p) {
    int item;
    while(1) {
        sem_wait(&full);
        sem_wait(&sem);

        item = buffer[output];
        output = (output + 1) % N;
        counter = counter - 1;

        sem_post(&sem);
        sem_post(&empty);
        consume(item);
    }
}
```

```
sem_init(&mutex, 0, 1);
sem_init(&empty, 0, N); // indicates the initial number of buffer items
sem_init(&full, 0, 0); // indicates the initial number of buffer available
...
```

- OS level solutions
- POSIX semaphores
- **POSIX mutexes**
- Exercises



- **Mutex**

- POSIX.1c defines the **mutex** object for thread synchronization
 - It can be seen as **semaphores that can only be initialized to 1**
 - It is used only to guaranty mutual exclusion
 - It works as a latch -> two operations: **lock** and **unlock**
 - A mutex has at every moment:
 - **State**: Open or closed
 - **Owner**: It is a thread that has executed a successful **lock** operation on it



- Mutex operation:
 - A mutex is created opened and without owner
 - When a thread calls to **lock**
 - If the mutex was open (without owner), the calling thread closes it and becomes its owner
 - If the mutex was closed, the calling thread is suspended
 - When an owner thread calls to **unlock**
 - The mutex gets open
 - If there were suspended threads on it, one of them is awaked, then the mutex gets closed and the awoken thread is the new owner

• Mutex operation

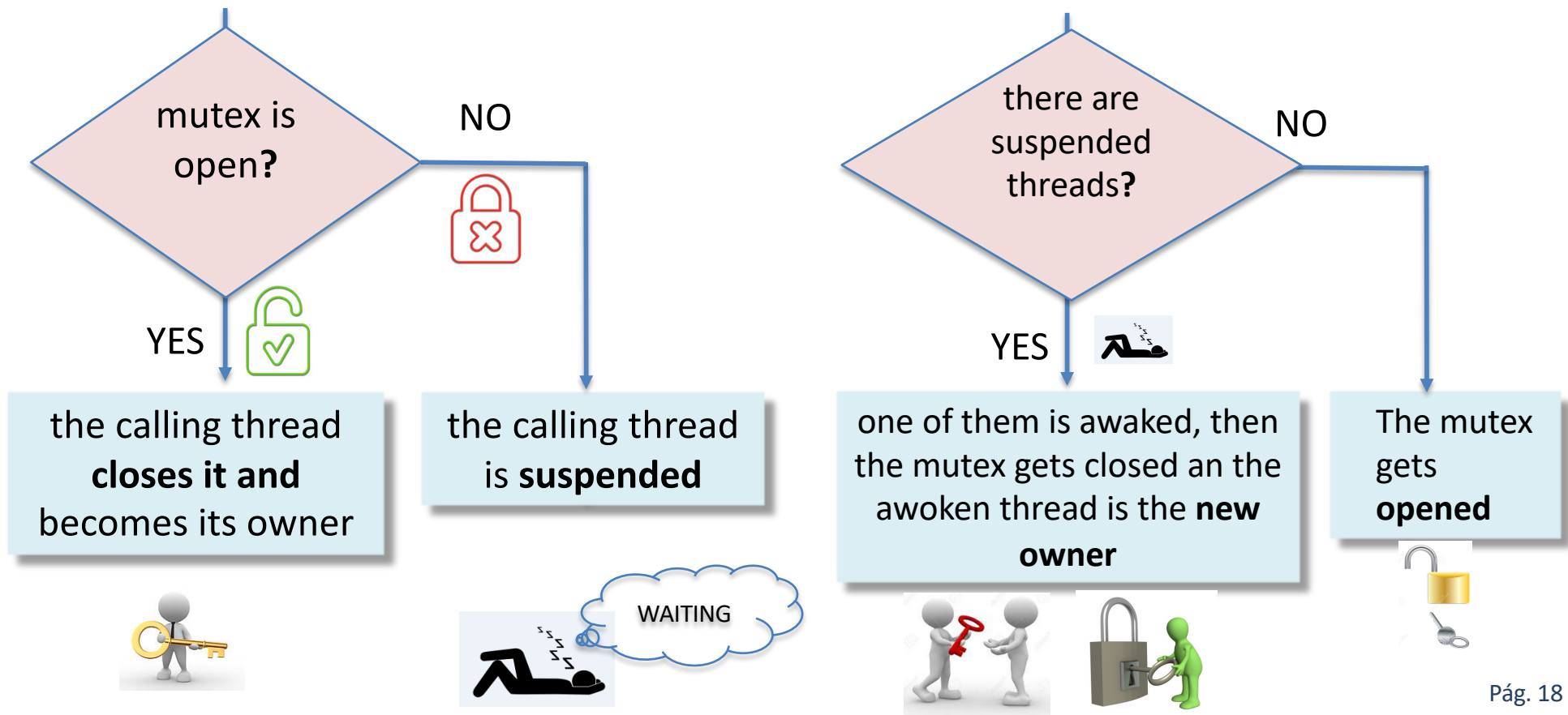
A mutex is created **opened** and without owner



When a thread **calls to lock** and



when the **owner** thread calls to **unlock** and ...



- **POSIX calls for mutexes**
 - Creation and destruction
 - `pthread_mutex_init`
 - `pthread_mutex_destroy`
 - Attribute initialization
 - `pthread_mutexattr_init`
 - `pthread_mutexattr_destroy`
 - Changing/Checking attribute values
 - Lock and unlock
 - `pthread_mutex_lock`
 - `pthread_mutex_trylock`
 - `pthread_mutex_unlock`

Equivalent
to `P(sem)`

Equivalent
to `V(sem)`

- **Example:** concurrent access to a shared variable by two threads
 - Main function code:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int V = 100;

// Threads code (next slide)

int main ( ) {
    pthread_t      thread1, thread2;
    pthread_attr_t  attributes;
    pthread_attr_init(&attributes);
    pthread_create(&thread1, &attributes, func_thread1, NULL);
    pthread_create(&thread2, &attributes, func_thread2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
}
```

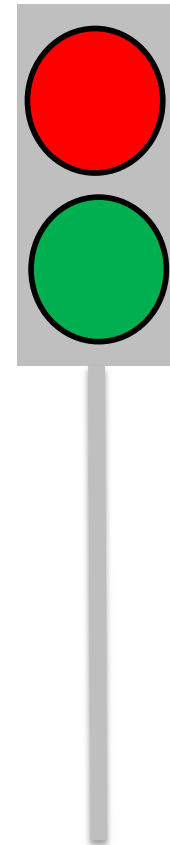
- Example (cont)
 - Thread functions:

```
void *func_thread1(void *p) {  
    int c;  
  
    for(c=0; c<1000; c++) {  
        pthread_mutex_lock(&m);  
  
        V = V + 1;  
  
        pthread_mutex_unlock(&m);  
    }  
    pthread_exit(0);  
}
```

```
void *func_thread2(void *p) {  
    int c;  
  
    for(c=0; c<1000; c++) {  
        pthread_mutex_lock(&m);  
  
        V = V - 1;  
  
        pthread_mutex_unlock(&m);  
    }  
    pthread_exit(0);  
}
```

- **Mutex** or Mutual Exclusion Object is used to give access to a resource to only one process at a time. The mutex object allows all the processes to use the same resource but at a time, only one process is allowed to use the resource. Mutex uses the lock-based technique to handle the critical section problem.
- **Semaphore** is an integer variable **S**, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of **S** i.e. **wait()** and **signal()**.
- There are two categories of semaphores i.e. **Counting semaphores** and **Binary semaphores**.
- **Difference between Mutex and Semaphore**
- Mutex uses a locking mechanism i.e. if a process wants to use a resource then it locks the resource, uses it and then release it. But on the other hand, semaphore uses a signalling mechanism where wait() and signal() methods are used to show if a process is releasing a resource or taking a resource.
- A mutex is an object but semaphore is an integer variable.
- A mutex object allows multiple process threads to access a single shared resource but only one at a time. On the other hand, semaphore allows multiple process threads to access the finite instance of the resource until available.
- In mutex, the lock can be acquired and released by the same process at a time. But the value of the semaphore variable can be modified by any process that needs some resource but only one process can change the value at a time.

- OS level solutions
- POSIX semaphores
- POSIX mutexes
- **Exercises**



- **Exercise S06.1** What possible values will take **x** as a result of the concurrent execution of the following threads?

```
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
sem_t s1,s2,s3;
int x;
```



<pre>void *func_thread1(void *a) { sem_wait(&s1); sem_wait(&s2); x=x+1; sem_post(&s3); sem_post(&s1); sem_post(&s2); } void *func_thread2(void *b) { sem_wait(&s2); sem_wait(&s1); sem_wait(&s3); x=10*x; sem_post(&s2); sem_post(&s1); }</pre>	<pre>int main() { pthread_t h1,h2 ; x = 1; sem_init(&s1,0,1); /*Inicializa a 1*/ sem_init(&s2,0,1); /*Inicializa a 1*/ sem_init(&s3,0,0); /*Inicializa a 0*/ pthread_create(&h1,NULL,func_thread1,NULL); pthread_create(&h2,NULL,func_thread2,NULL); pthread_join(h1,NULL); pthread_join(h2,NULL); }</pre>
--	---

Exercise S06.2 What possible values will take shared variables **x** and **y** at the end of the following concurrent threads. The initial values are: $x=1$, $y=4$, $S1=1$, $S2=0$ y $S3=1$.

Thread A

```
P(S2);  
P(S3);  
x = y * 2;  
y = y + 1;  
V(S3);
```

Thread B

```
P(S1);  
P(S3);  
x = x + 1;  
y = 8 + x;  
V(S2);  
V(S3);
```

Thread C

```
P(S1);  
P(S3);  
x = y + 2;  
y = x * 4;  
V(S3);  
V(S1);
```

