

Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

Bloque Temático 2: Gestión de Procesos

Seminario Unidad Temática 6

Sincronización: Semáforos Posix

fSO

DISCA



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

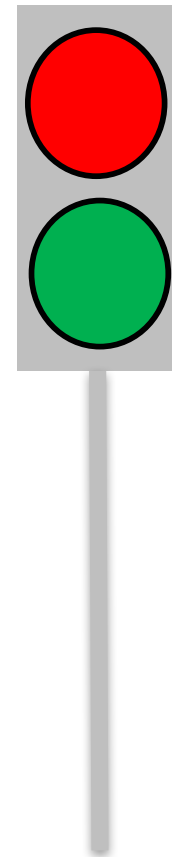
- **Objetivos:**

- Familiarizarse con el concepto de **sección crítica**
- Conocer los mecanismos de **sincronización** que ofrece el **Sistema Operativo**
- Identificar la sincronización de actividades como un mecanismo básico de los Sistemas Operativos

- **Bibliografía**

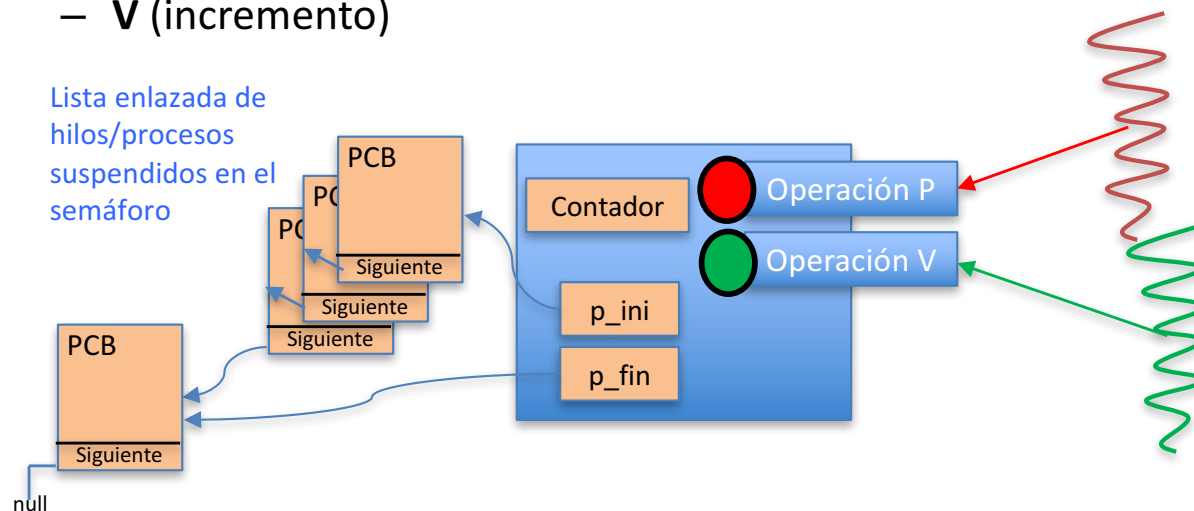
- “Fundamentos de sistemas operativos” Silberschatz 7ª Ed
- “Sistemas operativos: una visión aplicada” Carretero 2º Ed
- “UNIX Programación Práctica”, Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4

- **Contenido**
 - **Solución a nivel de Sistema Operativo**
 - Semáforos POSIX
 - Mutex POSIX
 - Ejercicios



- **Semáforo**

- Un semáforo puede conceptualmente entenderse, como un valor entero que admite operaciones especiales de incremento y decremento
 - El decremento **puede** suspender al proceso que lo invoca
 - El incremento **puede** despertar a otro proceso, previamente suspendido
- Es un tipo de dato que el Sistema Operativo pone a disposición de los procesos de usuario
 - Se declara como una variable de tipo “semáforo”, indicando su valor entero inicial
 - Posee dos operaciones de acceso, implementadas como llamadas al sistema
 - **P** (decremento)
 - **V** (incremento)



- **Semáforo: Operaciones de acceso**

- Declaración e inicialización

Semaphore S(N);

- Se declara el semáforo “S” con un valor inicial “N”
- “N” debe ser **mayor o igual que cero** (no puede ser negativo)

- Decremento

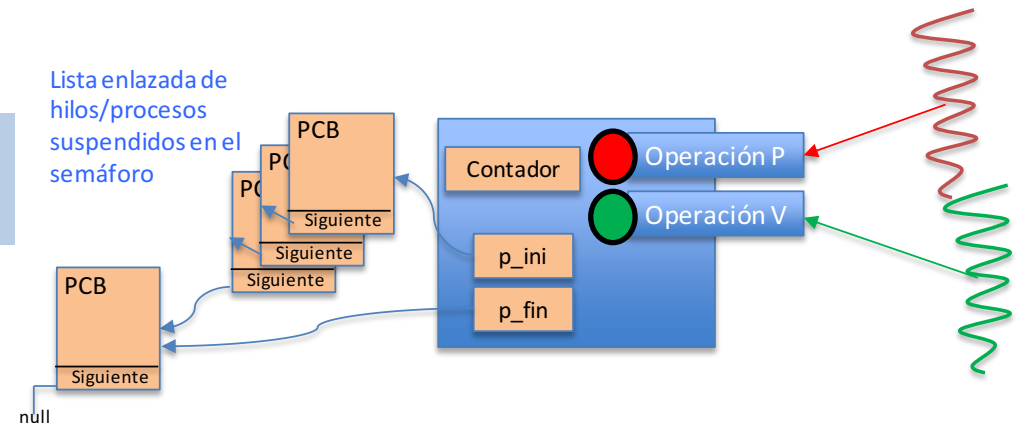
P(S);

S = S - 1;
si S < 0 entonces suspender(S);

- Incremento

V(S);

S = S + 1;
si S <= 0 entonces despertar(S);



Notas:

- El S.O. asegura que las operaciones **P** y **V** son **atómicas**.
- **suspender(S)**: suspende al proceso invocante en una cola asociada a “S”
- **despertar(S)**: extrae un proceso de la cola de “S” y lo despierta (pasa a preparado)

- **Semáforos: Solución al problema de la sección crítica**
 - Definimos un semáforo compartido, denominado “mutex”, iniciado a 1
Semaphore mutex(1);
 - Código de los N procesos/hilos:

```
void *hilo_i(void *p) {  
  
    while(1) {  
        P(mutex);  
  
        /* Sección crítica */  
  
        V(mutex);  
  
        /* Sección restante */  
  
    }  
}
```

Esta solución cumple:

- Exclusión mutua
- Progreso
- Espera limitada, si la cola del semáforo es FIFO

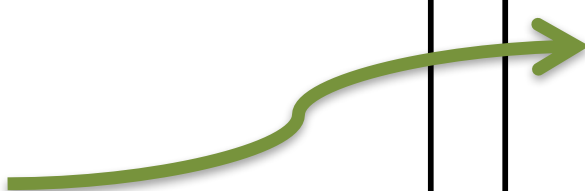
- **Semáforos:** útil para **sincronizar** procesos en general
- Ejemplo:
 - **Establecer** un cierto **orden o precedencia** en la ejecución de zonas de código de diferentes procesos/hilos
 - “hilo1” debe ejecutar la función “F1” antes que “hilo2” ejecute “F2”
 - Se define un semáforo compartido denominado “sinc” e inicializado a cero

Semaphore `sinc(0);`

```
void *hilo1(void *p)
{
    ...

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

```
void *hilo2(void *p)
{
    ...
    P(sinc);
    F2;
    ...
}
```



- **Semáforo:** herramienta que permite controlar el número de procesos simultáneos, que accede a un punto del código
 - Sea un código que ejecutan concurrentemente muchos hilos
 - Hay suficientes recursos para que 5 hilos puedan acceder a un determinado lugar del código, donde se invoca a la función “F”
 - Definimos un semáforo compartido “max_5”, inicializado a **cinco**

Semaphore max_5(5);

.....

pthread_t th1, th2, th3, th4, th5;

pthread_attr_t attr;

pthread_attr_init(&attr);

pthread_create(&th1, &attr, hilo_i NULL);

pthread_create(&th2, &attr, hilo_i NULL);

pthread_create(&th3, &attr, hilo_i NULL);

pthread_create(&th4, &attr, hilo_i NULL);

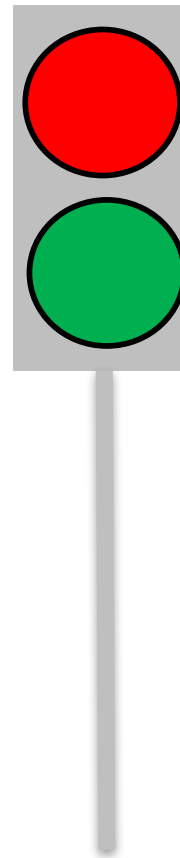
.....

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


- **El semáforo es un mecanismo esencial del SO que permite:**
 - implementar colas de procesos suspendidos a la espera de poder usar recursos
 - sincronización de actividades
- **Semáforo : útil como “contador de recursos”**
 - Se inicializa el contador del semáforo S al número de recursos a contabilizar
 - Cada vez que un hilo **necesita un recurso** realiza una **operación $P(S)$** , decrementando el contador de S , si no hay recursos ($S \leq 0$) el hilo se suspende
 - Cuando un hilo **termina de utilizar un recurso**, realiza una **operación $V(S)$** , incrementando el contador de S y si hay algún hilo suspendido en S ($S < 0$) a la espera del recurso, la operación V lo activa
 - Si el contador del semáforo es positivo, su valor absoluto indica la cantidad de recursos disponibles **si $S > 0$ entonces $|S| = \text{cantidad de recursos disponibles}$**
 - Si el contador del semáforo es negativo, su valor absoluto indica la cantidad de procesos suspendidos en la cola del semáforo **si $S < 0$ entonces $|S| = \text{numero de procesos suspendidos}$**
 - Cuando el contador del semáforo es cero, ambas afirmaciones anteriores son aplicables **si $S = 0$ entonces \rightarrow no hay procesos suspendidos
no hay recursos disponibles**

- **Contenido**

- Solución a nivel de Sistema Operativo
- **Semáforos POSIX**
- Mutex POSIX
- Ejercicios



- POSIX.1b introdujo el tipo de variable semáforo `sem_t`


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

- Los **semáforos se pueden compartir** entre procesos y pueden ser accedidos por parte de todos los hilos del proceso.
- Los **semáforos se heredan padre-hijo** igual que los descriptores de fichero.


- Las operaciones que soporta son:

```
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);
```

Operación
P(sem)



Operación
V(sem)



- **Productor Consumidor: Version1.0**

- Definimos un semáforo “mutex” inicializado a **uno** → **sem_init(&mutex,0,1);**

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

    while(1) {
        item = producir();

        sem_wait(& mutex);

        while (contador == N)
            /*bucle vacio*/ ;
        buffer[entrada] = item;
        entrada = (entrada + 1) % N;
        contador = contador + 1;

        sem_post(&mutex);
    }
}
```

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

    while(1) {
        sem_wait(&mutex);

        while (contador == 0)
            /*bucle vacio*/ ;
        item = buffer[salida];
        salida = (salida + 1) % N;
        contador = contador - 1;

        sem_post(&mutex);

        consumir(item);
    }
}
```

- Esta solución **no funciona**, porque un productor o consumidor que haya cerrado “mutex” y se quede en el bucle “while”, deja a todos los demás hilos bloqueados!!

- Productor Consumidor: Versión 2.0

```
#include <semaphore.h>
sem_t mutex, items, huecos;
```

```
void *func_prod(void *p) {
    int item;
    while(1) {
        item = producir();
        sem_wait(&huecos);
        sem_wait(&mutex);

        buffer[entrada] = item;
        entrada = (entrada + 1) % N;
        contador = contador + 1;

        sem_post(&mutex);
        sem_post(&items);
    }
}
```

```
void *func_cons(void *p) {
    int item;
    while(1) {
        sem_wait(&items);
        sem_wait(&mutex);

        item = buffer[salida];
        salida = (salida + 1) % N;
        contador = contador - 1;

        sem_post(&mutex);
        sem_post(&huecos);
        consumir(item);
    }
}
```

```
sem_init(&mutex,0,1);
sem_init(&huecos,0,N); //indica el numero de huecos
sem_init(&items,0,0); //indica el número de ítems
...
```

- **Contenido**

- Solución a nivel de Sistema Operativo
- Semáforos POSIX
- **Mutex POSIX**
- Ejercicios



- **Mutex:** mecanismo de **sincronización entre hilos (threads)**
 - POSIX.1c define los objetos “mutex” para la sincronización de hilos
 - Son como semáforos que sólo pueden tomar **valor inicial 1**
 - Se utilizan sólo para garantizar **la exclusión mutua**.
 - Funcionan como un cerrojo: dos operaciones básicas **cierre y apertura**
 - Cada mutex posee en cada instante:
 - **estado** : Dos posibles estados internos, abierto y cerrado
 - **propietario**: Un hilo es el propietario del mutex si ha ejecutado sobre él una operación de cierre con éxito



- **Funcionamiento del mutex**

- Un **mutex se crea** inicialmente **abierto y sin propietario**
- Si un hilo invoca la operación de cierre sobre un mutex
 - Si el mutex estaba abierto (sin propietario), lo cierra y pasa a ser su propietario
 - Si el mutex ya estaba cerrado, el hilo invocante se suspende
- Cuando el hilo propietario del mutex invoca la operación de apertura
 - Se abre el mutex
 - Si existían hilos suspendidos en el mismo, se selecciona uno y se despierta, con lo que puede cerrar el mismo (y pasa a ser el nuevo propietario)

- **Funcionamiento del mutex**

Un **mutex** se crea inicialmente **abierto** y sin propietario



Si un hilo invoca una **operación de cierre** sobre un mutex y



mutex
está
abierto?

NO



SI



el mutex se **cierra** y el hilo pasa a ser su **propietario**



el hilo se **suspende** en la cola de mutex



esperando

Si un hilo **propietario** del mutex invoca una operación de **apertura** sobre un mutex y ...



hay hilos
suspendidos
en el mutex?

NO

SI



se despierta a un hilo, para que pueda cerrar el mutex y pasar a ser el **nuevo propietario**



se **abre** el mutex



- **LLamadas POSIX para gestión de mutex:**
 - **Creación y destrucción de mutex:**
 - pthread_mutex_init
 - pthread_mutex_destroy
 - **Inicialización de los atributos del mutex**
 - pthread_mutexattr_init
 - pthread_mutexattr_destroy
 - Modificación/Consulta de valores al atributo: compartición, protocolo, etc.
 - **Cierre y apertura de mutex**
 - pthread_mutex_lock
 - pthread_mutex_trylock
 - pthread_mutex_unlock

Operación
P (sem)

Operación
V (sem)

- **Ejemplo:** acceso concurrente a una variable por parte de dos hilos
 - Código del programa principal:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int V = 100;

// Código de los hilos (transparencia siguiente)

int main ( ) {
    pthread_t      hilo1, hilo2;
    pthread_attr_t  atributos;
    pthread_attr_init(&atributos);
    pthread_create(&hilo1, &atributos, fhilo1, NULL);
    pthread_create(&hilo2, &atributos, fhilo2, NULL);

    pthread_join(hilo1, NULL);
    pthread_join(hilo2, NULL);
}
```

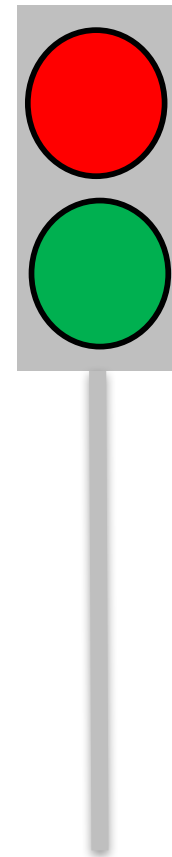
- Ejemplo
 - Código de los hilos:

```
void *f hilo1(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 *f hilo2(void *p) {  
    int c;  
  
    for(c=0; c<1000; c++) {  
        pthread_mutex_lock(&m);  
  
        V = V - 1;  
  
        pthread_mutex_unlock(&m);  
    }  
    pthread_exit(0);  
}
```

- **Contenido**

- Solución a nivel de Sistema Operativo
- Semáforos POSIX
- Mutex POSIX
- **Ejercicios**



- **Ejercicio S06.1:**

¿Cuáles son los posibles valores que tomará x como resultado de la ejecución concurrente de los siguientes hilos?

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



```
void *func_hilo1(void *a)
{
    sem_wait(&s1);
    sem_wait(&s2);
    x=x+1;
    sem_post(&s3);
    sem_post(&s1);
    sem_post(&s2);
}
void *func_hilo2(void *b)
{
    sem_wait(&s2);
    sem_wait(&s1);
    sem_wait(&s3);
    x=10*x;
    sem_post(&s2);
    sem_post(&s1);
}
```

```
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_hilo1,NULL);
    pthread_create(&h2,NULL,func_hilo2,NULL);
    pthread_join(h1,NULL);
    pthread_join(h2,NULL);
}
```

Ejercicio S06.2:

Comente qué valores posibles tendrían las variables x e y al finalizar la ejecución de los siguientes tres procesos concurrentes. Los valores iniciales son los siguientes: $x=1$, $y=4$, $S1=1$, $S2=0$ y $S3=1$.

Proceso A

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

Proceso B

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

Proceso C

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

