

# Sistemas Operativos

FACULTAD DE INFORMÁTICA  
UNIVERSIDAD COMPLUTENSE DE MADRID

## Módulo 3.3: Comunicación y Sincronización entre Procesos e Hilos

Juan Carlos Sáez

# Contenido

## 1 Introducción

- Problemas clásicos de comunicación y sincronización

## 2 Mecanismos de comunicación y sincronización

- Mutexes
- Semáforos
- Variables condición
- Memoria compartida entre procesos

## 3 Implementación de primitivas de sincronización

# Contenido

## 1 Introducción

- Problemas clásicos de comunicación y sincronización

## 2 Mecanismos de comunicación y sincronización

- Mutexes
- Semáforos
- Variables condición
- Memoria compartida entre procesos

## 3 Implementación de primitivas de sincronización

# Procesos concurrentes

- **Concurrencia:** ejecución simultánea o entrelazada de múltiples flujos de instrucciones de distintos procesos o hilos
- Modelos
  - Multiprogramación en un único procesador
  - Multiprocesador
  - Multicomputador (proceso distribuido)
- Razones
  - Compartir recursos físicos
  - Compartir recursos lógicos
  - Acelerar los cálculos
  - Modularidad

## Problemas clásicos de concurrencia

- El problema de la sección crítica
- El problema del productor-consumidor
- El problema de los lectores-escriptores
- Problema de los filósofos comensales

## Problema de la sección crítica

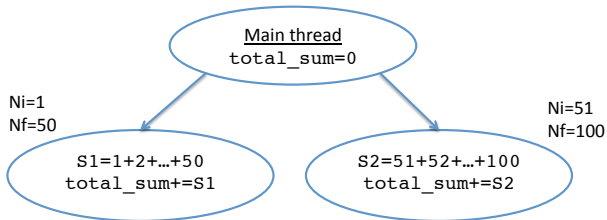
- Disponemos de un programa concurrente formado por  $n$  procesos/hilos
- Cada uno tiene un fragmento de código que accede/modifica un recurso compartido:
  - Sección crítica
- Queremos que sólo uno de los procesos/hilos en cada instante puede ejecutar su sección crítica

# Problema de la sección crítica: Ejemplo 1

```
int i;
int total_sum= 0;
for (i = 1; i <= 100; i++)
    total_sum+=i;
```



Parallelization with two threads



## Problema de la sección crítica: Ejemplo 1

- Calcular  $\sum_{i=1}^N i$  usando múltiples hilos

```
int total_sum = 0; // Var. compartida
```

```
void calculate_partial_sum(int ni, int nf) {  
    int j = 0;  
    int partial_sum = 0; // Var. privada  
    for (j = ni; j <= nf; j++)  
        partial_sum = partial_sum + j;  
    total_sum = total_sum + partial_sum;  
    pthread_exit(0);  
}
```

- Si varios procesos ejecutan concurrentemente este código se puede obtener un resultado incorrecto



## Problema de la sección crítica: Ejemplo 1

- Posible implementación de la sección crítica en ensamblador:

```
total_sum = total_sum + partial_sum;
```

```
LDR R1,total_sum      #R1=0 (la 1ª vez)
```

```
LDR R2,partial_sum    #R2=1275
```

```
ADD R1,R1,R2          #R1=1275
```

```
STR R1,total_sum      #total_sum=1275
```

# Problema de la sección crítica: Ejemplo 1

## ■ Posible situación de conflicto:

```
LDR R1,total_sum    #R1=0
LDR R2,partial_sum  #R2=1275
```

##### Cambio de Contexto #####

```
LDR R1,total_sum    #R1=0
LDR R2,partial_sum  #R2=3775
ADD R1,R1,R2        #R1=3775
STR R1,total_sum    #total_sum=3775
```

##### Cambio de Contexto #####

```
ADD R1,R1,R2        #R1=1275
STR R1,total_sum    #total_sum=1275
```

# Problema de la sección crítica: Ejemplo 1

## ■ Solución:

- Solicitar permiso para entrar en sección crítica
- Indicar la salida de sección crítica

```
int total_sum = 0; // Var. compartida

void calculate_partial_sum(int ni, int nf) {
    int j = 0;
    int partial_sum = 0; // Var. privada
    for (j = ni; j <= nf; j++)
        partial_sum = partial_sum + j;

    <Entrar a la sección crítica>
    total_sum = total_sum + partial_sum;
    <Salir de la sección crítica>
    pthread_exit(0);
}
```

## Problema de la sección crítica: Ejemplo 2

- Un banco almacena el saldo de las cuentas de cada cliente en un fichero por cuenta
  - Por cada ingreso en cuenta se debe actualizar el saldo almacenado en el fichero correspondiente

```
void ingresar(char *cuenta, int cantidad) {  
    int saldo, fd;  
    fd = open(cuenta, O_RDWR);  
    read(fd, &saldo, sizeof(int));  
    saldo = saldo + cantidad;  
    lseek(fd, 0, SEEK_SET);  
    write(fd, &saldo, sizeof(int));  
    close(fd);  
    return;  
}
```

- Si dos procesos ejecutan concurrentemente este código se puede perder algún ingreso

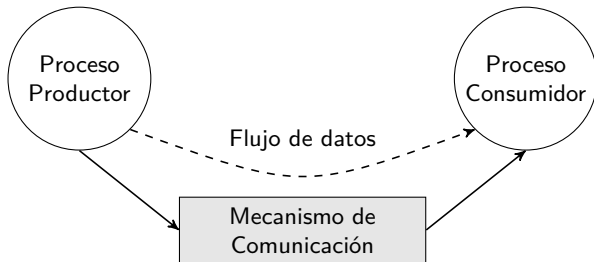
## Problema de la sección crítica: Ejemplo 2

```
void ingresar(char *cuenta, int cantidad) {  
    int saldo, fd;  
  
    fd = open(cuenta, O_RDWR);  
    <Entrada en la sección crítica>  
    read(fd, &saldo, sizeof(int));  
    saldo = saldo + cantidad;  
    lseek(fd, 0, SEEK_SET);  
    write(fd, &saldo, sizeof(int));  
    <Salida de la sección crítica>  
    close(fd);  
    return;  
}
```

## Solución al problema de la sección crítica

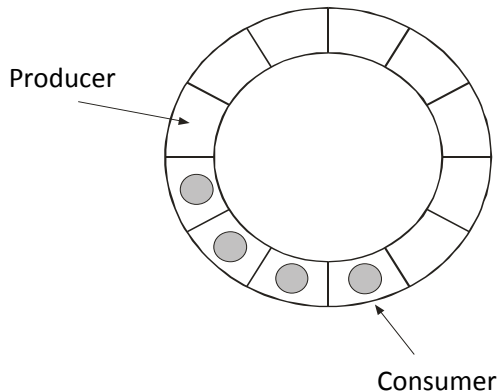
- Requisitos que debe ofrecer cualquier solución para resolver el problema de la SC:
  - **Exclusión mutua:** sólo un proceso en la sección crítica
  - **Eficiencia/Progreso:** Si ningún proceso está ejecutando dentro de la sección crítica, la decisión de qué proceso entra en la sección se hará sobre los procesos que desean entrar
    - Un proceso no debe ver retrasado el acceso a su sección crítica cuando no hay ningún otro proceso usándola
  - **Evitar inanición/Garantizar espera limitada:** ningún proceso debe esperar indefinidamente para entrar en su región crítica
- Hay que tener también en mente:
  - No deben hacerse suposiciones sobre las velocidades relativas de los procesos o sobre el número de procesos competidores
  - Un proceso permanece dentro de su sección crítica un tiempo finito

## Problema del productor-consumidor



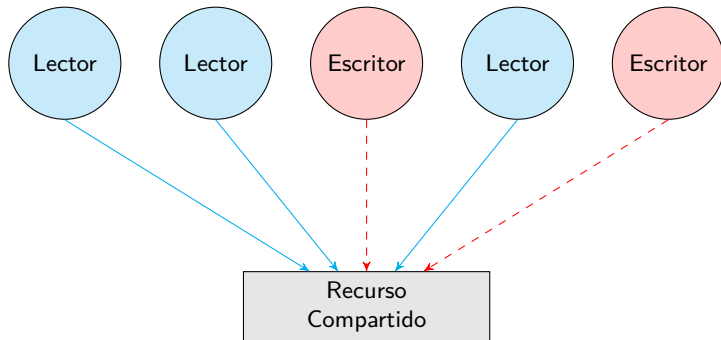
- El consumidor se quedará bloqueado cuando no haya datos que leer
- El productor se bloqueará al intentar enviar datos si no hay espacio libre en el mecanismo de comunicación para almacenarlos

## Productor-consumidor: buffer circular





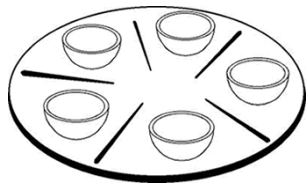
## Problema de los lectores-escritores



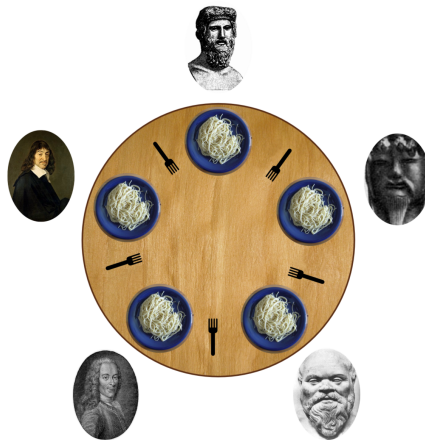
- Los lectores pueden ejecutar su sección crítica (SC) simultáneamente
- Si hay un escritor en su SC no puede haber ningún otro proceso (ni lector, ni escritor) ejecutando su SC

## Filósofos comensales (Dijkstra'65)

- Cinco filósofos sentados en una mesa piensan y comen arroz:
  - Emplean un tiempo finito en comer y pensar
  - Necesitan 2 palillos para comer, que se cogen de uno en uno
  - Ningún filósofo debe morir de hambre (evitar inanición e interbloqueo)
  
- Comportamiento de cada filósofo (bucle infinito):
  - 1 Pensar...
  - 2 Coger un palillo, coger el otro.
  - 3 comer
  - 4 soltar un palillo y luego el otro. Ir a 1.



# Filósofos comensales (Dijkstra'65)



# Filósofos comensales (Dijkstra'65)

## Soluciones:

- Turno rotativo:
  - Desperdicia recursos
- Un camarero arbitra el uso de los palillos
  - Necesitamos un supervisor
- Numerar los palillos, coger siempre el menor, luego y el mayor y soltarlos en orden inverso:
  - Penalizamos al último filósofo
- Si no puedo coger el segundo palillo, suelto el primero
  - ¿Y si mis vecinos comen alternativamente?

# Contenido

## 1 Introducción

- Problemas clásicos de comunicación y sincronización

## 2 Mecanismos de comunicación y sincronización

- Mutexes
- Semáforos
- Variables condición
- Memoria compartida entre procesos

## 3 Implementación de primitivas de sincronización

## Mecanismos C&S

- Todos los problemas clásicos tienen en común:
  - Necesitan compartir información
    - Que todos puedan conocer el valor de una variable...
  - Necesitan sincronizar su ejecución
    - Que un proceso espere a otro...
- Estudiaremos qué mecanismos suelen ofrecer los sistemas operativos para este fin
  - Resolveremos problemas clásicos para ilustrar cómo se usa cada mecanismo

# Mecanismos de comunicación

- Archivos
- Tuberías (pipes, FIFOs)
  - No las estudiaremos
- Memoria compartida
  - 1 Implícita: hilos
    - Hilos de un mismo proceso se comunican mediante variables o estructuras de datos globales
  - 2 Explícita: necesidad de una API específica
    - Hilos de distintos procesos no comparten memoria
    - Para establecer comunicación → Crear regiones de memoria compartida

# Mecanismos de Sincronización

- Servicios del sistema operativo:
  - Cerrojos, Locks o Mutexes
  - Semáforos
  - Variables condición
  - Señales: asíncronas y no encolables
  - Tuberías (pipes, FIFOS) (no las estudiaremos)
- Las operaciones de sincronización deben ser **atómicas**



## Cerrojo (mutex)

- Mecanismo ideal para resolver el problema de la sección crítica
- Podemos ver un cerrojo como un objeto con **dos estados**:
  - abierto ó cerrado
- Soporta **dos operaciones atómicas**:

- *Adquirir el cerrojo (lock)*

```
lock() {  
    while(estado!=abierto)  
        .. espera ..
```

```
    estado=cerrado;
```

```
}
```

- *Liberar el cerrojo (unlock)*

```
unlock() { estado=abierto; }
```

- lo ejecuta el hilo que lo cerró con lock() antes (propietario)



# Tipos de cerrojos

```
lock() {  
    while(estado!=abierto)  
        .. espera ..  
  
    estado=cerrado;  
}
```

## 3 tipos de cerrojo o mutex

- 1 Bloqueante:** El hilo se va a dormir cuando durante la espera
  - Estado  $\Rightarrow$  Bloqueado
  - `unlock()` *despierta* a un hilo  $\Rightarrow$  estado Listo para ejecutar
- 2 De espera activa:** El hilo consume ciclos de CPU mientras comprueba la condición en repetidas ocasiones:
  - También llamado *spin lock*
  - Objetivo: reducir el número de cambios de contexto
- 3 Adaptativo:** El hilo realiza espera activa durante un tiempo y luego se bloquea

## Cerrojo (mutex)

- Un cerrojo o mutex es un mecanismo de sincronización indicado para hilos
  - Ideal para el problema de la sección crítica, pues garantiza exclusión mutua (*MUTual EXclusion*)...
- Cerrojo: “objeto” con 3 atributos y 2 métodos atómicos

```
/* abierto/cerrado */
state_t state;

/* cola de hilos
   bloqueados */
queue_t q;

/* Hilo propietario */
thread_id owner;
```

```
lock(m) {
    if (m->estado==cerrado) {
        queue_add(m->q, esteHilo);
        suspenderHilo();
        queue_del(m->q, esteHilo);
    }

    m->estado = cerrado;
    m->owner = esteHilo;
}
```

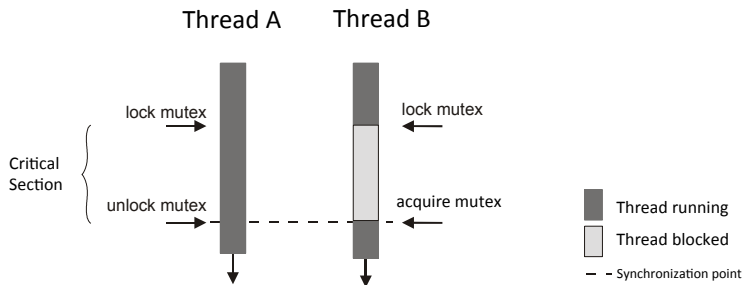
```
unlock(m) {
    if (m->owner==esteHilo) {
        m->estado=abierto;
        m->owner=NULL;
        if (m->q.notEmpty())
            despiertaUnHiloDeCola();
    }
    else
        error!!
}
```



## Secciones críticas con mutex

```
lock(m);  /* entrada en la seccion critica */
< seccion critica >
unlock(m); /* salida de la seccion critica */
```

- La operación `unlock()` debe invocarla el hilo que ejecutó `lock()` (propietario del mutex)



## Servicios POSIX: mutexes

- `int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);`
  - Inicializa un mutex.
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - Libera los recursos del mutex
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - Intenta adquirir el mutex. Bloquea al hilo si el mutex se encuentra adquirido por otro hilo.
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - Desbloquea el mutex. El hilo invocador debe ser el propietario del mutex.

# Semáforos (Dijkstra'65)

- Mecanismo de sincronización
- Misma máquina
- Objeto con 2 atributos
  - Cola de procesos/hilos bloqueados
  - Contador
    - se inicializa a un valor  $\geq 0$
- Dos operaciones atómicas
  - `wait()`
  - `signal()`



# Operaciones sobre semáforos

```
wait(s){
    s.c = s.c - 1;
    if(s.c < 0){
        <Bloquear al proceso>
    }
}

signal(s){
    s.c = s.c + 1;
    if (hay_procesos_bloqueados){
        <Desbloquear a un proceso bloqueado
        por wait>
    }
}
```

## Significado de $c$

- $c \geq 0 \rightarrow c$  es el número de veces que se puede invocar a `wait()` sin que ningún proceso invocador se bloquee
- $c \leq 0 \rightarrow |c|$  es el número de procesos bloqueados en el semáforo

# Uso de los semáforos

## ■ Usos típicos

### 1 Garantizar exclusión mutua

- Ejemplo: solución al problema de la sección crítica
- Crear semáforo global inicializado a 1
- Secciones críticas se encierran entre `wait()` y `signal()`

### 2 Imponer restricciones de sincronización asociadas a condiciones sobre números enteros

- Ejemplo: solución del problema productor/consumidor

### 3 Semáforo como cola de espera

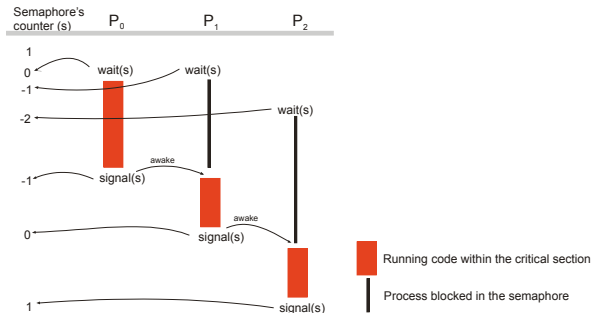
- Ejemplos: solución de problemas de este tema
- Crear semáforo  $S$  inicializado a 0 + variable entera que lleve la cuenta de número de procesos bloqueados en la cola del semáforo
- La variable entera estará protegida por un mutex u otro semáforo



# Secciones críticas con semáforos

- Inicializar semáforo global con  $c=1$

```
wait(s); /* entrada en la seccion critica */
<sección crítica>
signal(s); /* salida de la seccion critica */
```



# Semáforos POSIX: Clasificación

## ■ Dos tipos de semáforos POSIX:

### 1 Semáforos sin nombre

- Permiten sincronizar hilos o procesos (típicamente emparentados)
- Se crean/destruyen mediante `sem_init()/sem_destroy()`

### 2 Semáforos con nombre

- Suelen usarse para sincronizar procesos no emparentados
- Nombre del semáforo: ID global (cadena de caracteres)
  - Se crean/destruyen mediante `sem_open()/sem_unlink()`

- Las funciones `sem_wait()` y `sem_post()` implementan las operaciones *wait()* y *signal()* de ambos tipos de semáforos
  - Nos centraremos en el uso de los semáforos sin nombre

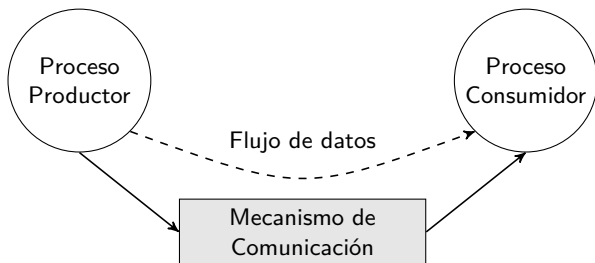
## Semáforos POSIX: API

- `int sem_init(sem_t *sem, int shared, unsigned int val);`
  - Inicializa un semáforo sin nombre
    - `shared`: 0 → hilos; 1 → procesos
    - si `shared==1`, el descriptor del semáforo (`sem_t`) debe almacenarse en una región de memoria compartida
- `int sem_destroy(sem_t *sem);`
  - Destruye un semáforo sin nombre
- `int sem_wait(sem_t *sem);`
  - Realiza la operación `wait()` sobre un semáforo
- `int sem_post(sem_t *sem);`
  - Realiza la operación `signal()` sobre un semáforo

## Semáforos POSIX: API

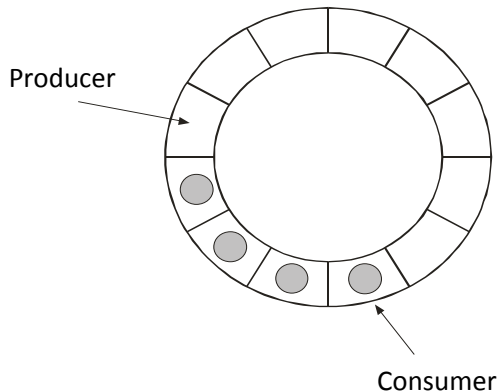
- `sem_t *sem_open(char* name,int flags,mode_t mode, unsigned int val);`
  - Abre (crea) un semáforo con nombre
- `int sem_close(sem_t *sem);`
  - Cierra un semáforo con nombre.
- `int sem_unlink(char *name);`
  - Borra un semáforo con nombre
- `int sem_wait(sem_t *sem);`
  - Realiza la operación *wait()* sobre un semáforo
- `int sem_post(sem_t *sem);`
  - Realiza la operación *signal()* sobre un semáforo

## Problema del productor-consumidor



- El consumidor se quedará bloqueado cuando no haya datos que leer
- El productor se bloqueará al intentar enviar datos si no hay espacio libre en el mecanismo de comunicación para almacenarlos

## Productor-consumidor: buffer circular



# Productor-consumidor con semáforos (I)

```
#define MAX_BUF          1024    /* buffer size */
#define PROD            100000   /* number of items to produce */
sem_t items;             /* number of items in the buffer */
sem_t gaps;              /* number of free gaps in the buffer */
int buffer[MAX_BUF];      /* shared buffer */

void main(void){
    pthread_t th1, th2; /* thread descriptors */
    /* semaphore initialization */
    sem_init(&items, 0, 0); sem_init(&gaps, 0, MAX_BUF);

    /* thread creation */
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    /* wait for thread completion */
    pthread_join(th1, NULL); pthread_join(th2, NULL);

    sem_destroy(&gaps); sem_destroy(&items);
    exit(0);
}
```

## Productor-consumidor con semáforos (II)

```
void Producer(void){
    int widx = 0; /* write index */
    int data; /* data to produce */
    int i;

    for(i=0; i < PROD; i++ ){
        /* produce data */
        data = generate_data();
        /* one gap less */
        sem_wait(&gaps);
        buffer[widx] = data;
        widx = (widx + 1) % MAX_BUF;
        /* added one item */
        sem_post(&items);
    }

    pthread_exit(0);
}
```

```
void Consumer(void){
    int ridx= 0; /* read index */
    int data; /* data to be consumed */
    int i;

    for(i=0; i<PROD; i++ ){
        /* an item will be removed */
        sem_wait(&items);
        data = buffer[ridx];
        ridx= (ridx+ 1) % MAX_BUF;
        /* one free gap more */
        sem_post(&gaps);
        do_something(data);
    }

    pthread_exit(0);
}
```



## N Productores - 1 consumidor con semáforos

```
#define MAX_BUF      1024    /* buffer size */
#define PROD        100000  /* number of items to produce */
#define N 2          /* Number of producers */
sem_t items;           /* number of items in the buffer */
sem_t gaps;            /* number of free gaps in the buffer */
sem_t producers; /* To enforce mutual exclusion among producers*/
int widx=0;           /* Shared write index */
int buffer[MAX_BUF];   /* shared buffer */

void main(void){
    int i;
    pthread_t thp[N], thc; /* thread descriptors */
    /* semaphore initialization */
    sem_init(&items, 0, 0); sem_init(&gaps, 0, MAX_BUF); sem_init(&producers, 0, 1);

    /* thread creation */
    for (i=0;i<N;i++) pthread_create(&thp[i], NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    /* wait for thread completion */
    for (i=0;i<N;i++) pthread_join(&thp[i], NULL);
    pthread_join(th2, NULL);

    sem_destroy(&gaps); sem_destroy(&items); sem_destroy(&producers);
    exit(0);
}
```

## N Productores - 1 consumidor con semáforos

```
void Producer(void){
    int data; /* data to produce */
    int i;

    for(i=0; i < PROD; i++){
        /* produce data */
        data = generate_data();
        /* one gap less */
        sem_wait(&gaps);
        sem_wait(&producers);
        /* Critical section */
        buffer[widx] = data;
        widx = (widx + 1) % MAX_BUF;
        sem_post(&producers);
        /* added one item */
        sem_post(&items);
    }

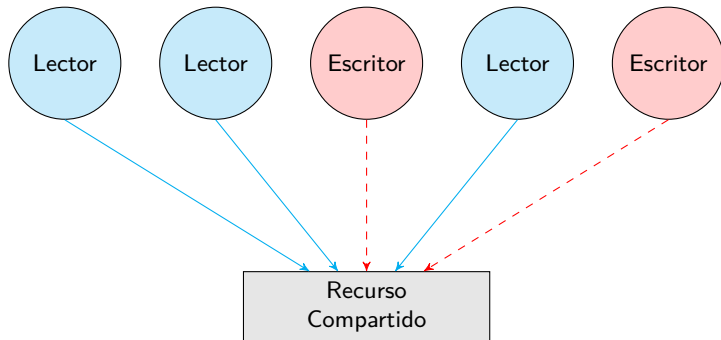
    pthread_exit(0);
}
```

```
void Consumer(void){
    int ridx= 0; /* read index */
    int data; /* data to be consumed */
    int i;

    for(i=0; i<PROD*N; i++){
        /* an item will be removed */
        sem_wait(&items);
        data = buffer[ridx];
        ridx= (ridx+ 1) % MAX_BUF;
        /* one free gap more */
        sem_post(&gaps);
        do_something(data);
    }

    pthread_exit(0);
}
```

## Problema de los lectores-escritores



- Los lectores pueden ejecutar su sección crítica (SC) simultáneamente
- Si hay un escritor en su SC no puede haber ningún otro proceso (ni lector, ni escritor) ejecutando su SC

# Lectores-escriptores con semáforos (I)

```
int data = 5;    /* shared resource */
int nr_readers = 0; /* number of readers */
sem_t sem_nreaders; /* control access to nr_readers */
sem_t sem_read_write; /* mutual exclusion between reader-writer and writer-writer */

void main(void){
    pthread_t th1, th2, th3, th4;
    sem_init(&sem_read_write, 0, 1); sem_init(&sem_nreaders, 0, 1);

    pthread_create(&th1, NULL, Reader, NULL);
    pthread_create(&th2, NULL, Writer, NULL);
    pthread_create(&th3, NULL, Reader, NULL);
    pthread_create(&th4, NULL, Writer, NULL);

    pthread_join(th1, NULL); pthread_join(th2, NULL);
    pthread_join(th3, NULL); pthread_join(th4, NULL);

    /* destroy semaphores */
    sem_destroy(&sem_read_write); sem_destroy(&sem_nreaders);
    exit(0);
}
```

## Lectores-escriptores con semáforos (II)

```
void Reader(void) {
    while(1){
        sem_wait(&sem_nreaders);
        nr_readers = nr_readers + 1;
        if (nr_readers == 1)
            sem_wait(&sem_read_write);
        sem_post(&sem_nreaders);

        /* read data */
        printf(" %d\n", data);

        sem_wait(&sem_nreaders);
        nr_readers = nr_readers - 1;
        if (nr_readers == 0)
            sem_post(&sem_read_write);
        sem_post(&sem_nreaders);
    }
}
```

```
void Writer(void) {
    while(1){
        sem_wait(&sem_read_write);

        /* modify the resource */
        data = data + 2;

        sem_post(&sem_read_write);
    }
}
```

## Variables condición

- Mecanismo de sincronización para hilos
- Cada variable condición tiene asociada una cola de espera y un mutex
  - El mutex se comparte habitualmente entre múltiples variables condición usadas en el programa concurrente
- Las variables condición soportan 3 operaciones:
  - 1 `cond_wait()`: el hilo invocador se bloquea en la cola de espera
  - 2 `cond_signal()`: se despierta a un solo hilo de los que estaban esperando en la cola (si hay alguno)
  - 3 `cond_broadcast()`: se despierta a todos los hilos que estaban esperando en la cola (si hay alguno)
- Estas operaciones deben invocarse en un fragmento de código entre `lock(mutex)` y `unlock(mutex)`

## Operaciones sobre variables condición (I)

```
void cond_wait(lock_t m, vc_t varC ) {  
    queue_add(varC->queue, curThread);  
    unlock(m);  
    park(); // el hilo se bloquea en la cola de espera  
    lock(m);  
}
```

- El hilo invocador debe ser el propietario del mutex
- `cond_wait()` **siempre** bloquea al hilo invocador
  - Antes de bloquearse en la cola, el hilo libera el cerrojo para que otro hilo pueda adquirirlo
- Cuando el hilo se despierta y sale la cola, debe readquirir el cerrojo (lock) de nuevo.
- Cuando la operación `cond_wait()` retorna, el hilo invocador es el propietario del mutex de nuevo

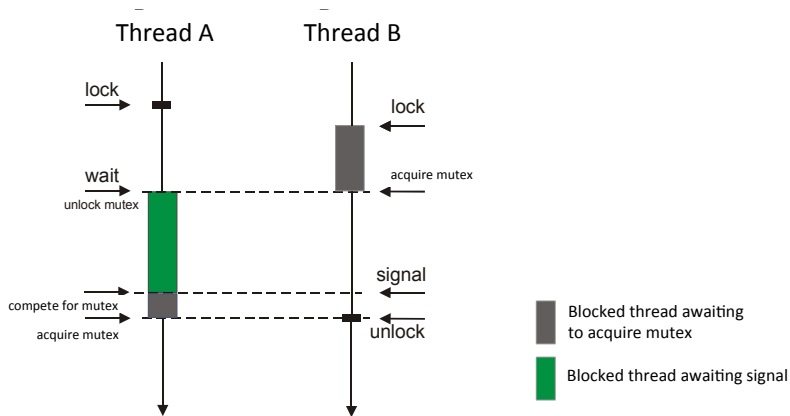
## Operaciones sobre variables condición (II)

```
/* Despierta a un solo hilo bloqueado en la cola */  
void cond_signal (vc_t varC ) {  
    if (!isEmpty(varC->queue))  
        unpark(queue_remove(varC->queue))  
}  
  
/* Despierta a todos los hilos bloqueados en la cola */  
void cond_broadcast (vc_t varC ) {  
    while (!isEmpty(varC->queue))  
        unpark(queue_remove(varC->queue))  
}
```

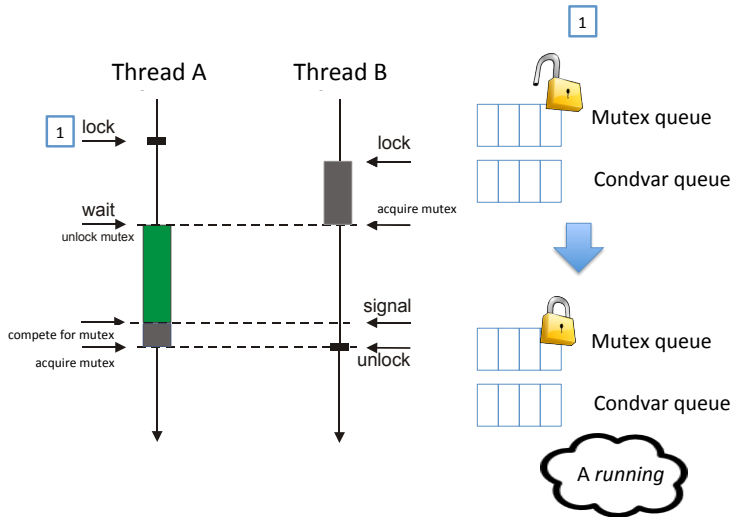
- Es aconsejable invocar estas operaciones en un fragmento de código encerrado entre `lock(mutex)` y `unlock(mutex)`



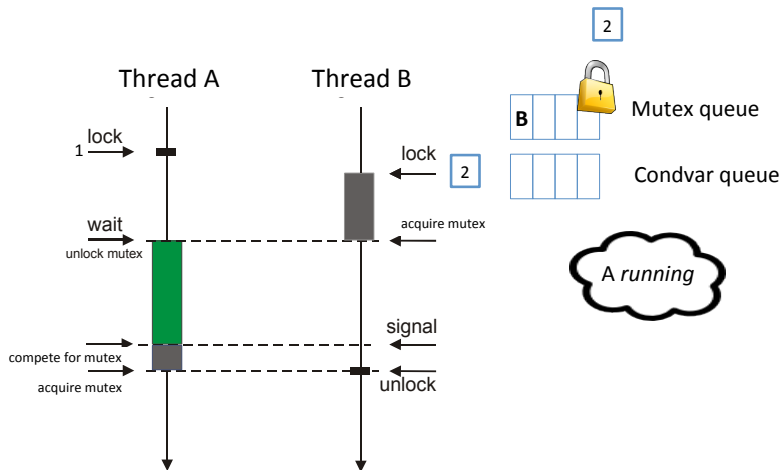
# Operaciones sobre variables condición (III)



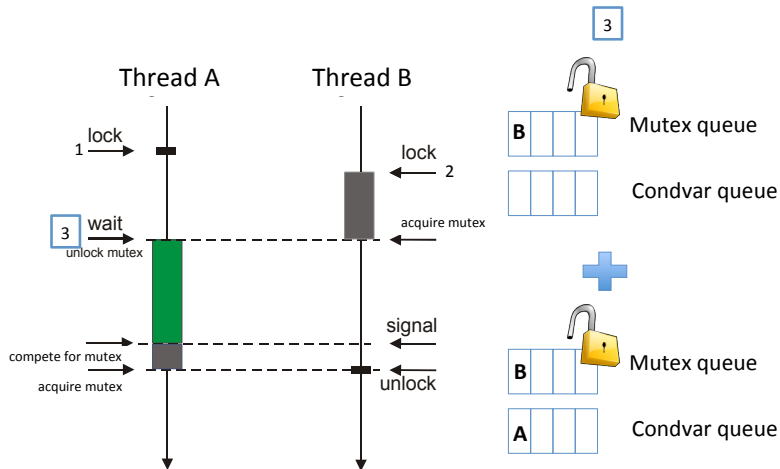
# Uso de variables condición



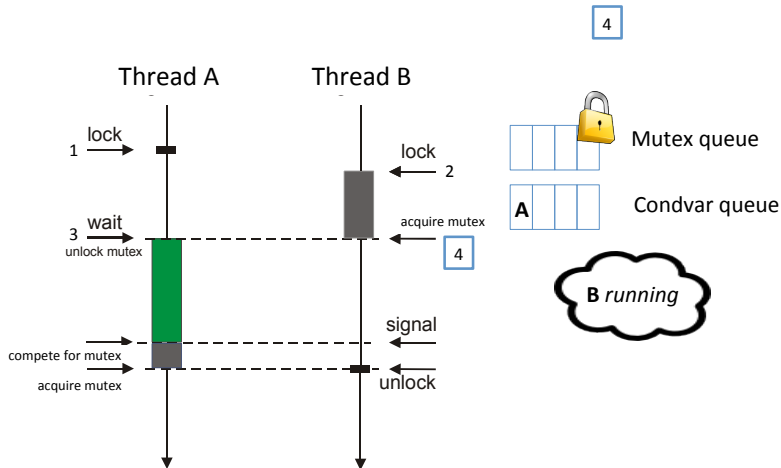
# Uso de variables condición



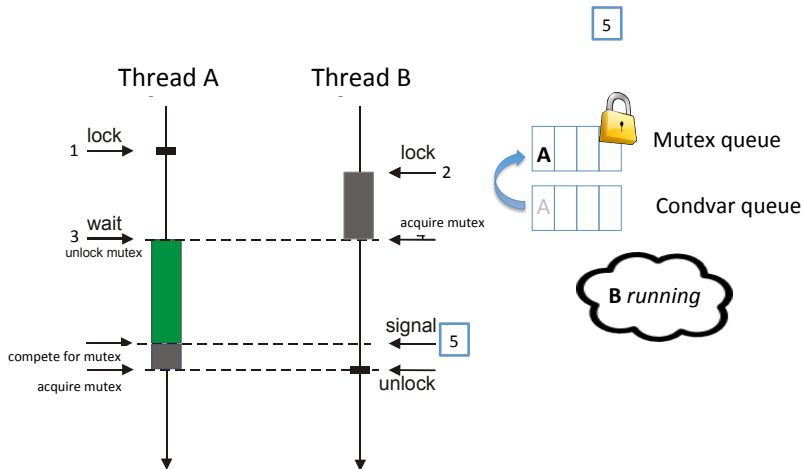
# Uso de variables condición



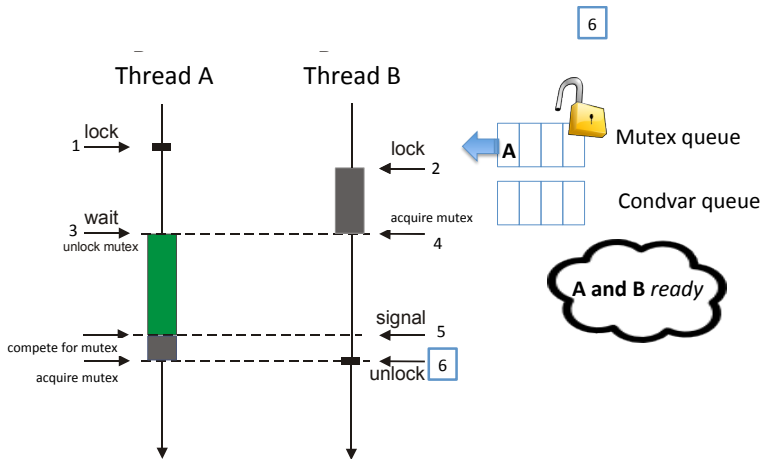
# Uso de variables condición



# Uso de variables condición



# Uso de variables condición



# Uso de cerrojos / var. condicionales

## ■ Hilo A

```
lock(mutex); /* entrar a la SC */
<operaciones sobre los recursos compartidos (exclusion mutua)>
while (condición relacionada con el recurso == false)
    cond_wait(condition, mutex); /* bloqueo */
<acciones deseadas que cumplen la condición>
unlock(mutex); /* salir de la SC */
```

## ■ Hilo B

```
lock(mutex); /* entrar a la SC */
<operaciones sobre los recursos compartidos (exclusion mutua)>
/* Como hemos podido afectar a la condición asociada al recurso, despertamos a otro hilo */
cond_signal(condition, mutex);
<más operaciones protegidas>
unlock(mutex); /* salir de la SC */
```

*Debemos usar `while` para re-evaluar la condición*



# Productor consumidor con var. condición (I)

- Por simplicidad, asumiremos que el buffer circular se da ya implementado como un tipo abstracto de datos (`cbuffer_t`)
  - Operaciones sobre `cbuffer_t`
    - `boolean is_empty(cbuffer_t cb);`
    - `boolean is_full(cbuffer_t cb);`
    - `void add(cbuffer_t cb, item_t data);`
    - `item_t remove(cbuffer_t cb);`
- Restricciones de uso de las operaciones
  - 1 Implementación no es *thread-safe*
    - No podemos invocar las operaciones simultáneamente desde múltiples threads
    - Se precisa de un cerrojo para serializar acceso al buffer
  - 2 `add()` no puede invocarse si buffer lleno
  - 3 `remove()` no puede invocarse si buffer vacío

# Productor consumidor con var. condición (II)

## Variables globales

```
cbuffer_t b; /* shared ring buffer (max capacity: N items) */
mutex m; /* Mutual exclusion when accessing buffer */
condvar prod, cons; /* To block producer/consumer, respectively */
```

```
void producer() {
    item_t data;

    while (true) {
        data=produce();
        lock(m);

        add(b,data);

        unlock(m);
        delay(...);
    }
}
```

```
void consumer() {
    item_t data;

    while (true) {
        lock(m);

        data=remove(b);

        unlock(m);
        do_something(data);
    }
}
```

# Productor consumidor con var. condición (II)

## Variables globales

```
cbuffer_t b; /* shared ring buffer (max capacity: N items) */
mutex m; /* Mutual exclusion when accessing buffer */
condvar prod, cons; /* To block producer/consumer, respectively */
```

```
void producer() {
    item_t data;

    while (true) {
        data=produce();
        lock(m);

        while (is_full(b))
            cond_wait(prod,m);

        add(b,data);

        unlock(m);
        delay(...);
    }
}
```

```
void consumer() {
    item_t data;

    while (true) {
        lock(m);

        while (is_empty(b))
            cond_wait(cons,m);

        data=remove(b);

        unlock(m);
        do_something(data);
    }
}
```

# Productor consumidor con var. condición (II)

## Variables globales

```
cbuffer_t b; /* shared ring buffer (max capacity: N items) */
mutex m; /* Mutual exclusion when accessing buffer */
condvar prod, cons; /* To block producer/consumer, respectively */
```

```
void producer() {
    item_t data;

    while (true) {
        data=produce();
        lock(m);

        while (is_full(b))
            cond_wait(prod,m);

        add(b,data);

        cond_signal(cons);

        unlock(m);
        delay(...);
    }
}
```

```
void consumer() {
    item_t data;

    while (true) {
        lock(m);

        while (is_empty(b))
            cond_wait(cons,m);

        data=remove(b);

        cond_signal(prod);

        unlock(m);
        do_something(data);
    }
}
```

## Servicios POSIX (II)

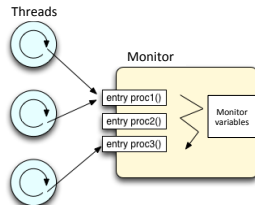
- `int pthread_cond_init(pthread_cond_t* cond, pthread_condattr_t* attr);`
  - Inicializa una variable condicional
- `int pthread_cond_destroy(pthread_cond_t *cond);`
  - Destruye un variable condicional
- `int pthread_cond_signal(pthread_cond_t *cond);`
  - Despierta a un hilo de aquellos que esperan en la cola de la variable condición
  - No tiene efecto si no hay ningún hilo esperando
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - Despierta a todos los hilos que esperan en la cola de la variable condición
- `int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);`
  - Bloquea al hilo invocador hasta que se despierta más adelante cuando otro hilo invoca `pthread_cond_signal()` o `pthread_cond_broadcast()`

# Monitores

- Monitores: uso sistemático de cerrojos y variables condición
  - Se usan cerrojos para conseguir la exclusión mutua
  - Y variables condición para restringir la ejecución en base a condiciones que dependen del estado de las variables compartidas entre hilos
- Ventajas monitores vs. semáforos o mútexes + var. condición:
  - Código más legible y fácil de depurar
  - Monitores se adaptan mejor a la programación orientada a objetos (p.ej., Java)
- Los monitores no son recursos de sincronización que ofrece el SO sino un mecanismo debemos construir a medida para nuestro programa concurrente

# Monitores

- Un monitor es un objeto que consta de 3 elementos:
  - 1 Procedimientos de monitor (métodos)
  - 2 Variables o estructuras de datos compartidas entre procedimientos (atributos del monitor)
  - 3 Un mutex + N variables condición
    - En algunas implementaciones de monitores el mutex es implícito
- Los procedimientos del monitor (métodos) se pueden invocar simultáneamente por varios hilos
  - Los problemas de concurrencia se resuelven en la implementación de los procedimientos



## Monitores: Implementación (I)

- Se dice que un hilo “entra al monitor” cuando invoca alguno de sus procedimientos y “sale” cuando finaliza la ejecución del procedimiento
- Restricciones de implementación:
  - 1 Sólo puede haber un hilo activo (no bloqueado) ejecutándose en el monitor (dentro del cuerpo de cualquiera de sus procedimientos)
    - El hilo que ejecuta código dentro del monitor accede a las variables del mismo (atributos) en exclusión mutua
  - 2 Puede haber varios hilos bloqueados dentro del monitor
    - Los hilos se bloquean en las colas de espera asociadas a las variables condición del monitor



## Monitores: Implementación (II)

- En la implementación en “C” con POSIX Threads de los monitores, las restricciones anteriores se garantizan de la siguiente forma:
  - Procedimiento del monitor es una función en C con un único punto de salida
  - El cuerpo del procedimiento de monitor se encierra entre `lock()` y `unlock()` del mutex del monitor

```
procedimiento_de_monitor() {  
    pthread_mutex_lock(&mutex); /* entrar al monitor */  
    <Cuerpo del procedimiento>  
    pthread_mutex_unlock(&mutex); /* salir del monitor */  
}
```

- Todo hilo que va a bloquearse en una variable condición del monitor libera implícitamente el mutex al invocar `cond_wait()`

## Uso de los monitores

- Los hilos ejecutan los procedimientos del monitor para dos cosas:
  - 1 Modificar/acceder a las variables del monitor de manera segura
  - 2 Entrar/salir de sección crítica

```
Procedimiento_de_entrada()  
<Sección crítica>  
Procedimiento_de_salida()
```

## Producer-consumidor con monitor

```
#define MAX_BUFFER 1024 /* buffer size */
#define DATA_TO_PRODUCE 10000
pthread_mutex_t mutex; /* monitor's mutex */
pthread_cond_t c_full; /* to block the producer */
pthread_cond_t c_empty; /* to block the consumer */
int nr_items; /* # of items in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */
int ridx, widx; /* R/W positions in the buffer */

void init_monitor(void); /* Initialize monitor */
void destroy_monitor(void); /* Free up monitor resources */
/* Monitor procedures */
void produce(int item); /* Insert item into the buffer */
int consume(void); /* Extract an item from the buffer */
```

## Producer-consumidor con monitor

```
void Producer(void) {  
    int i,item;  
    for (i=0;i<DATA_TO_PRODUCE;i++){  
        item=... generate an item ...  
        produce(item);  
    }  
    pthread_exit(0);  
}
```

```
void Consumer(void) {  
    int i,item;  
    for (i=0;i<DATA_TO_PRODUCE;i++){  
        item=consume();  
        ... Do something with item ...  
    }  
    pthread_exit(0);  
}
```

```
int main(int argc, char *argv[]){  
    pthread_t th1, th2;  
    init_monitor();  
    pthread_create(&th1, NULL, Producer, NULL);  
    pthread_create(&th2, NULL, Consumer, NULL);  
    pthread_join(th1, NULL); pthread_join(th2, NULL);  
    destroy_monitor();  
    return 0;  
}
```

## Productor-consumidor con monitor

```
void init_monitor(void)
{
    pthread_cond_init(&c_full, NULL);
    pthread_cond_init(&c_empty, NULL);
    pthread_mutex_init(&mutex, NULL);
    ridx=widx=nr_items=0;
}

void destroy_monitor(void)
{
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&c_full);
    pthread_cond_destroy(&c_empty);
}
```

## Producer-consumidor con monitor

```
void produce(int item) {
    /* "enter the monitor" */
    pthread_mutex_lock(&mutex);

    /* block while buffer full */
    while (nr_items == MAX_BUFFER)
        pthread_cond_wait(&c_full,&mutex);

    buffer[widx] = item;
    widx= (widx+ 1) % MAX_BUFFER;
    nr_items++;

    /* buffer is not empty */
    pthread_cond_signal(&c_empty);

    /* "exit the monitor" */
    pthread_mutex_unlock(&mutex);
}
```

```
int consume(void) {
    int item;
    /* "enter the monitor" */
    pthread_mutex_lock(&mutex);

    /* block while buffer empty */
    while (nr_items == 0)
        pthread_cond_wait(&c_empty,&mutex);

    item = buffer[ridx];
    ridx= (ridx + 1) % MAX_BUFFER;
    nr_items--;

    /* buffer is not full */
    pthread_cond_signal(&c_full);

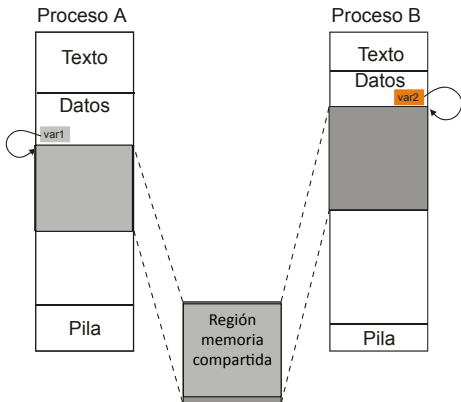
    /* "exit the monitor" */
    pthread_mutex_unlock(&mutex);
    return item;
}
```

# Mecanismos de comunicación

- Archivos
- Tuberías (pipes, FIFOs)
  - No las estudiaremos
- Memoria compartida
  - 1 Implícita: hilos
    - Hilos de un mismo proceso se comunican mediante variables o estructuras de datos globales
  - 2 Explícita: necesidad de una API específica
    - Hilos de distintos procesos no comparten memoria
    - Para establecer comunicación → Crear regiones de memoria compartida

## Memoria compartida (entre procesos)

- Declaración independiente de variables dentro de los procesos que apuntan a la misma región de memoria “real”





## Memoria compartida POSIX

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
  - Ubica (mapea) una porción del fichero especificado por el descriptor `fd` en memoria, devolviendo un puntero a esa región (`addr`)
    - Si parámetro `addr` es `NULL`, el SO escoge la dirección virtual automáticamente
  - Tres opciones para el parámetro `fd` (descriptor de fichero)
    - 1 `fd=-1`: para creación de zona de memoria anónima  
flags: `MAP_ANONYMOUS` → compartir padre-hijo
    - 2 `fd` obtenido con `open()` : fichero proyectado en memoria (Memoria compartida con respaldo en disco)
    - 3 `fd` obtenido con `shm_open()`: región de memoria compartida con nombre
  - La región de memoria puede ser compartida o privada:
    - flags: `MAP_SHARED` or `MAP_PRIVATE`
- `int munmap(void *addr, size_t length);`
  - Actualiza el fichero de respaldo de la región de memoria y borra las ubicaciones para el rango de direcciones especificado

## Memoria compartida POSIX

- `int msync(void *addr, size_t len, int flags);`
  - Escribe cualquier dato (página) modificada en memoria en su correspondiente fichero de respaldo

## Resumen

- Threads:
  - Memoria compartida (variables globales)
  - Mutexes, variables condicionales, semáforos o monitores
- Procesos emparentados (`fork()`):
  - Tuberías y memoria compartida (mapeada)
  - Semáforos sin nombre (mapeados en memoria)
- Procesos no emparentados en la misma máquina:
  - Tuberías con nombre
  - Semáforos con nombre

## P&C con memoria compartida y semáforos

### ■ Productor:

- Crea los semáforos con nombre (`sem_open`)
- Crea un archivo (`open`)
- Le asigna espacio (`ftruncate`)
- Proyecta el archivo en su espacio de direcciones (`mmap`)
- Utiliza la zona de memoria compartida
- Desproyecta la zona de memoria compartida (`munmap`)
- Cierra y borra el archivo

### ■ Consumidor:

- Abre los semáforos (`sem_open`)
- Debe esperar a que archivo esté creado para abrirlo (`open`)
- Proyecta el archivo en su espacio de direcciones (`mmap`)
- Utiliza la zona de memoria compartida
- Cierra el archivo

## Código del productor

```
#define MAX_BUFFER      1024    /* buffer size */
#define DATA_TO_PRODUCE 100000 /* # elements to produce */
sem_t *elements; /* # of elements in the buffer */
sem_t *gaps;     /* # of free gaps in the buffer */

void main(int argc, char *argv[]){
    int shd;
    int *buffer; /* shared buffer */

    /* the producer creates the file */
    shd = open("BUFFER", O_CREAT|O_WRONLY, 0700);
    ftruncate(shd, MAX_BUFFER * sizeof(int));

    /* Maps the file into the process address space */
    buffer = (int*) mmap(NULL, MAX_BUFFER * sizeof(int),
        PROT_WRITE, MAP_SHARED, shd, 0);
```

## Código del productor (II)

```
/* The producer creates the semaphores */
elements = sem_open("ELEMENTS", O_CREAT, 0700, 0);
gaps = sem_open("GAPS", O_CREAT, 0700, MAX_BUFFER);

/* core producer's code */
Producer(buffer);

/* Unmap shared buffer */
munmap(buffer, MAX_BUFFER * sizeof(int));
close(shd); /* close the shared memory region */
unlink("BUFFER"); /* delete the shared memory region */

sem_close(elements);
sem_close(gaps);
sem_unlink("ELEMENTS");
sem_unlink("GAPS");
}
```

## Código del consumidor

```
#define MAX_BUFFER      1024    /* buffer size */
#define DATA_TO_PRODUCE 100000 /* # elements to produce */
sem_t *elements; /* # of elements in the buffer */
sem_t *gaps;     /* # of free gaps in the buffer */

void main(int argc, char *argv[]){
    int shd;
    int *buffer; /* shared buffer */

    /* the consumer opens the file */
    shd = open("BUFFER", O_RDONLY);

    /* Maps the file into the process address space */
    buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int),
                          PROT_READ, MAP_SHARED, shd, 0);
```

## Código del consumidor (II)

```
/* Consumer opens semaphores */
elementos = sem_open("ELEMENTS", 0);
huecos    = sem_open("GAPS", 0);

/* consumer's core processing */
Consumer(buffer);

/* unmap shared buffer */
munmap(buffer, MAX_BUFFER * sizeof(int));
close(shd); /* close shared memory object */

/* close semaphores */
sem_close(elements);
sem_close(gaps);
}
```



## Función del productor

```
void Producer(int *buffer)
{
    int pos = 0; /* write index */
    int item;    /* data to produce */
    int i;

    for(i=0; i < DATA_TO_PRODUCE; i++ ) {
        item = produce_item();
        sem_wait(gaps);
        buffer[pos] = item;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(elements);
    }
}
```

## Función del consumidor

```
void Consumer(char *buffer)
{
    int pos = 0; /* read index */
    int i, item;

    for(i=0; i < DATA_TO_PRODUCE; i++ ) {
        sem_wait(elements);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(gaps);
        printf("Received %d\n", item);
    }
}
```

# Contenido

## 1 Introducción

- Problemas clásicos de comunicación y sincronización

## 2 Mecanismos de comunicación y sincronización

- Mutexes
- Semáforos
- Variables condición
- Memoria compartida entre procesos

## 3 Implementación de primitivas de sincronización

## Implementación de lock()/unlock()

- El SO asegura que lock() y unlock() son operaciones **atómicas**
  - La implementación de estas operaciones requiere resolver un problema de sección crítica (variables compartidas)

```
lock(m) {  
    if (m->estado==cerrado) {  
        queue_add(m->q, esteHilo);  
        suspenderHilo();  
        queue_del(m->q,esteHilo);  
    }  
  
    m->estado = cerrado;  
    m->owner = esteHilo;  
}
```

```
unlock(m) {  
    if (m->owner==esteHilo) {  
        m->estado=abierto;  
        m->owner=NULL;  
        if (m->q.notEmpty())  
            despiertaUnHiloDeCola();  
    }  
    else  
        error!!  
}
```

# Soluciones al problema de la sección crítica

## Tipos de soluciones

- Espera activa
  - Sin soporte HW
    - Basadas en variables de control (Peterson 1981)
  - Con soporte HW
    - Test And Set (TAS), XCHG, LL/SC
- Sin espera activa
  - Soporte del SO
  - El SO bloquea el proceso/hilo

## Instrucciones máquina

- Se utiliza una instrucción máquina para actualizar una posición de memoria
- Puede aplicarse cualquier número de procesos:
  - Ciclo de memoria **RMW** (*read/modify/write*)
- No sufren injerencias por parte de otras instrucciones
- Puede aplicarse a múltiples secciones críticas
- Es simple y fácil de verificar

## Ejemplos de instrucciones

### Generales

- Test and set (T&S)
- Fetch and add (F&A)
- Swap/Exchange
- Compare and Swap (exchange)
- Load link/ Store conditional (LL/SC)

### Intel (x86)

- Muchas instrucciones pueden ser atómicas
- F&A lock; `xaddl eax, [mem_addr];`
- `XCHG xchg eax, [mem_addr]`
- `CMPXCHG -> lock cmpxchg [mem_addr], eax`

### ARM (y otros)

- LL/SC LDREX y STREX

## Semántica de Swap/Exchange

### Exchange

```
xchg src, dst
```

```
-----
```

```
rtmp <- Mem [src]
Mem [src] <- Mem [dst]
Mem [dst] <- rtmp
```

- Es una instrucción máquina (NO una función)
  - Es atómica, ininterrumpible
- Intercambia dos valores (potencialmente, ambos en memoria)
  - En Intel, sólo uno de los dos (src o dst) pueden estar en memoria



## Uso de Swap/Exchange

- Solución al problema de la **Sección Crítica con XCHG**

### Implementación

```
/* it may be stored in a register */
tmp = 1;
/* Busy wait */
while( tmp== 1)
    xchg(MemAddr, tmp);

Critical_section();

*MemAddr= 0;
```

# Semántica de LL/SC

## Load Link

```
ll src
```

```
-----
```

```
    rout <- Mem [src]
```

## Store Conditional

```
sc src, valor
```

```
-----
```

```
    si nadie accedió a src desde el anterior LL
```

```
        Mem[src]= valor
```

```
        rout <- 1
```

```
    sino
```

```
        rout <- 0
```

- Son DOS instrucciones máquina
  - Una siempre hace el load; la otra sólo hace store si no hubo escrituras a esa posición de memoria posteriores al LL

# Uso de LL/SC

- Solución al problema de la **Sección Crítica con LL/SC**

## Implementación

```
while (1) {  
    while(ll(MemAddr) == 1);  
    if (sc(MemAddr, 1) == 1) break;  
    /* otherwise, execute Load-Link again */  
}  
Critical_section();  
*MemAddr = 0;
```