

# Sistemas Operativos



Gestión de Procesos  
Comunicación y sincronización

# Agenda



- 1** Concurrencia
- 2** El problema de la Sección Crítica
- 3** Variables de Condición
- 4** Semáforos
- 5** Memoria Compartida
- 6** Interbloqueos - Deadlocks

# Agenda



- 1** Concurrencia
- 2 El problema de la Sección Crítica
- 3 Variables de Condición
- 4 Semáforos
- 5 Memoria Compartida
- 6 Interbloqueos - Deadlocks

# Tareas Concurrentes



Dos o más tareas (procesos o hilos) se dice que son concurrentes si los periodos de tiempo en los que ejecutan se solapan en el tiempo

- No ejecutan necesariamente al mismo tiempo (una por CPU)
- Se **reparten** el tiempo de CPU alternando su ejecución
- El orden en que ejecutan es indeterminado (planificador)

Utilidad de la concurrencia

- Compartir recursos físicos
- Compartir recursos lógicos
- Acelerar los cálculos
- Modularidad
- Comodidad

# Mecanismos de C&S



- Para que interaccionen necesitamos que las tareas puedan:
  - 1 **Compartir** información
    - Que varios hilos puedan consultar y/o modificar alguna(s) variable(s) de forma segura (sección crítica)
  - 2 **Sincronizar** su ejecución
    - Que un hilo pueda esperar a otro
    - Que un hilo pueda avisar a otro para que continúe
- Estudiaremos qué mecanismos de C&S ofrece habitualmente un SO
  - Estudiaremos cómo se usan
  - Veremos ejemplos de cómo se puede implementar la sección crítica

# Mecanismos de Comunicación



- Archivos
- Memoria compartida
  - Implícita para hilos de un **mismo proceso** (.data, .bss, heap)
  - Explícita para hilos de **distintos procesos**
    - Requiere de un API específico
- Tuberías (pipes, FIFOs)
  - No las estudiamos, aunque los usamos en los *shell scripts*
- Sockets: No los estudiamos
- Paso de Mensajes: No lo estudiamos

# Mecanismos de Sincronización



- Servicios del SO:
  - Cerrojos + Variables de Condición (Monitores)
  - Semáforos
  - Señales: asíncronas y no encolables (no las estudiaremos)
  - Tuberías (pipes, FIFOs) (no las estudiaremos)
  - Paso de Mensajes (no lo estudiamos)
- Las operaciones de sincronización deben ser **atómicas**

# Agenda



- 1 Concurrencia
- 2 El problema de la Sección Crítica**
- 3 Variables de Condición
- 4 Semáforos
- 5 Memoria Compartida
- 6 Interbloqueos - Deadlocks



# Suma de los N primeros naturales



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    unsigned long long int i, N, total = 0;
    char *endptr;

    if ((argc != 2) ||
        !(N = strtoll(argv[1], &endptr, 10)) ||
        (*endptr != '\0')){
        printf("Error, uso:\n\t%s unsigned_long_long\n", argv[0]);
        return(-1);
    }

    printf("Sumando de 1 a %llu (Gauss: %llu)... \n",N,
        (N * (N + 1))/2);
    for (i = 1; i <= N; i++)
        total += i;
    printf("Resultado = %llu\n", total);

    return 0;
}
```

# Ejecutando el programa



suma

```
> gcc -o suma suma.c
> for i in 10 1000 500000000; do time ./suma $i; done
Sumando de 1 a 10 (Gauss: 55)...
Resultado = 55

real    0m0.003s
user    0m0.002s
sys     0m0.001s
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500

real    0m0.002s
user    0m0.003s
sys     0m0.000s
Sumando de 1 a 500000000 (Gauss: 125000000250000000)...
Resultado = 125000000250000000

real    0m1.281s
user    0m1.276s
sys     0m0.004s
```

# Optimización: aprovechar varias CPUs



- Con el código anterior sólo una CPU estará ocupada en la suma, aunque el sistema tenga más de una
- Como la suma es conmutativa y asociativa, ¿no podemos hacer que varios hilos cooperen sumando cada uno un rango distinto?



# Suma Paralela

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define M 4
struct param {
    unsigned long long ni,nf;
};

unsigned long long total = 0;
void* suma(void* arg)
{
    struct param *p = (struct param*) arg;
    unsigned long long i;

    for (i = p->ni; i <= p->nf; i++)
        total += i;

    return NULL;
}
```



# Suma Paralela

```
void reparto(struct param [M], unsigned long long N);
int main(int argc, char* argv[]){
    int i;
    unsigned long long N;
    char *endptr;
    pthread_t hilos[M];
    struct param P[M] = {0};

    if ((argc != 2) || !(N = strtoll(argv[1], &endptr, 10)) ||
        (*endptr != '\0')){
        printf("Error, uso:\n\t%s unsigned_long_long\n", argv[0]);
        return(-1);
    }
    printf("Sumando de 1 a %llu (Gauss: %llu)... \n",N,
        (N * (N+1))/2);
    reparto(P, N);

    for (i=0; i<M; i++)    pthread_create(&hilos[i], NULL, suma, &P[i]);
    for (i=0; i<M; i++)    pthread_join(hilos[i], NULL);
    printf("Resultado = %llu\n", total);

    return 0;
}
```

# Suma Paralela



```
void reparto(struct param P[M], unsigned long long int N)
{
    unsigned long long i,j,n,resto;
    n = N / M;
    resto = N % M;
    j = 0;
    for (i=0; (i < M) && (j < N); i++){
        P[i].ni = j+1;
        P[i].nf = j+n;
        if (resto){
            P[i].nf++;
            resto--;
        }
        j = P[i].nf;
    }
}
```

# Ejecutando el programa



## Suma paralela

```
> gcc -o parsum parsum.c -lpthread
> for i in 10 1000 500000000; do time ./parsum $i; done
Sumando de 1 a 10 (Gauss: 55)...
Resultado = 55

real    0m0.004s
user    0m0.001s
sys     0m0.004s
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500

real    0m0.003s
user    0m0.000s
sys     0m0.003s
Sumando de 1 a 500000000 (Gauss: 125000000250000000)...
Resultado = 49372410099158516

real    0m1.320s
user    0m4.512s
sys     0m0.008s
```

# Ejecutando el programa



Suma paralela

```
> while true; do ./parsum 1000 ; sleep 1; done
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 406625
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
Resultado = 500500
^C
```





# ¿Qué está sucediendo?

Revisemos el código de los hilos:

```
unsigned long long total = 0;
void* suma(void* arg)
{
    struct param *p = (struct param*) arg;
    unsigned long long i;

    for (i = p->ni; i <= p->nf; i++)
        total += i;

    return NULL;
}
```

No es atómico!!!



```
load R1,=total
load R2, [R1]
load R3, [SP,#ioffset]
add  R2, R2, R3
str  R2, [R1]
```



## Secuencia posible con dos hilos

Supongamos que inicialmente *total* tiene el valor  $T$ , y que:

- El Hilo 1 quiere sumar  $x$  a *total*
- El Hilo 2 quiere sumar  $y$  a *total*

El resultado final debería ser:  $total = T + x + y$

```
load R1,=total
```

```
load R2, [R1]
```

```
load R3, [SP,#ioffset]
```

← Carga del valor  $T$

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

```
load R1,=total
```

```
load R2, [R1]
```

```
load R3, [SP,#ioffset]
```

```
add R2, R2, R3
```

```
str R2, [R1]
```

← Nueva carga del valor  $T$

←  $total = T + y$

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

```
add R2, R2, R3
```

```
str R2, [R1]
```

←  $total = T + x$

# Solución



El problema desaparece si podemos hacer que cuando un hilo comience a ejecutar la secuencia de instrucciones (`total += i;`):

```
load R1,=total
load R2, [R1]
load R3, [SP,#ioffset]
add  R2, R2, R3
str  R2, [R1]
```

ningún otro hilo pueda ejecutar parte de esa secuencia hasta que éste termine.

# Sección Crítica



Fragmento delimitado de código que cumple con los siguientes requisitos:

- **Exclusión mutua:** sólo un hilo en la región crítica
- **Progreso:** Si ningún hilo está ejecutando dentro de la sección crítica, la decisión de qué hilo entra en la sección se hará sobre los procesos que desean entrar
- **Espera limitada:** ningún proceso debe esperar indefinidamente para entrar en su región crítica

Lo que generalmente da lugar a la siguiente estructura de código:

<Solicitud de entrada en la sección crítica>

Código de la sección crítica

<Indicación de salida 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
  - Uso de primitivas anteriores
  - El SO cambiará el estado del proceso a bloqueado

# Instrucciones máquina



- Se utiliza una instrucción máquina para actualizar una posición de memoria
- Puede aplicarse a 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



# Ejemplo 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: lock
- F&A: `lock; xaddl eax, [mem_dir];`
- XCHG: `xchg eax, [mem_dir_lock]`
- CMPXCHG: `lock cmpxchg [dirMem], eax`

## ■ ARM y otros:

- LL/SC: `LDREX y STREX;`



# Semántica de Swap/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



# Sección Crítica con XCHG: spin-lock



```
typedef struct __lock_t {  
    int guard;  
} lock_t;
```

```
lock_init(lock_t *lc)  
{  
    lc->guard = 0;  
}
```

```
lock(lock_t *lc)  
{  
    int tmp = 1;  
    while (tmp == 1)  
        xchg(&lc->guard, &tmp);  
}  
  
unlock(lock_t *lc)  
{  
    lc->guard = 0;  
}
```

Uso de las macros para sección crítica:

```
lock(&lc);  
Sección_crítica();  
unlock(&lc);
```

# Semántica de LL/SC



// Load Link

```
LL src {  
    rout = Mem[src]  
}
```

// Store Conditional

```
SC src, valor {  
    if (nadie accedió a src  
        desde el anterior LL) {  
        Mem[src] = valor  
        rout = 1  
    } else  
        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

# Sección Crítica con LL/SC: spin-lock



```
typedef struct __lock_t {  
    int guard;  
} lock_t;
```

```
void lock_init(lock_t *lc)  
{  
    lc->guard = 0;  
}
```

```
void lock(lock_t *lc)  
{  
    while (LL(&lc->guard) == 1 ||  
           !SC(&lc->guard, 1));  
}
```

```
void unlock(lock_t *lc)  
{  
    lc->guard = 0;  
}
```

Uso de las macros para sección crítica:

```
lock(&lc);  
Sección_crítica();  
unlock(&lc);
```

# Spin-locks: problemas



Los spin-locks contruidos con las instrucciones atómicas:

- No garantizan la espera limitada
- Ineficientes:
  - Hilos ocupan todo un time slice ejecutando el bucle de espera
  - En un monoprocesador, es imposible que durante ese tiempo se libere el cerrojo
  - Se puede paliar un poco haciendo yields, pero seguiremos teniendo mucho cambio de contexto innecesario

Para solucionarlo necesitamos soporte del SO:

- establecer un orden de acceso que garantice una espera limitada
- eliminar spin-locks *largos* (mientras otro hilo esta en una sección crítica de usuario)

# SO: bloquear el hilo



Servicio del sistema operativo (Solaris):

- `park()`: bloquea al hilo actual
- `unpark(thread_id)`: desbloquea al hilo
- `setpark()`: marca al hilo como que está apunto de bloquearse
  - si entre `setpark` y `park` se hace una llamada a `unpark` para este hilo, la llamada a `park` no bloqueará el hilo.
  - solución para evitar una condición de carrera

# Cerrojos (Mutex)

```
typedef struct __mutex {  
    int locked; //state  
    int guard; //spin-lock  
    thread_id owner;  
    queue_t *q;  
} mutex_t;
```

```
mutex_init(mutex_t *m)  
{  
    m->locked = 0;  
    m->guard = 0;  
    queue_init(&m->q);  
}
```



# Cerros (Mutex)



```
typedef struct __mutex {
    int locked; //state
    int guard; //spin-lock
    thread_id owner;
    queue_t *q;
} mutex_t;

mutex_lock(mutex_t *m)
{
    int tmp = 1;
    while (tmp == 1)
        xchg(&m->guard, &tmp);

    if (!m->locked) {
        m->locked = 1;
        m->owner = getpid();
        m->guard = 0;
    } else {
        queue_add(m->q, getpid());
        setpark();
        m->guard = 0;
        park();
    }
}
```

```
mutex_init(mutex_t *m)
{
    m->locked = 0;
    m->guard = 0;
    queue_init(&m->q);
}

mutex_unlock(mutex_t *m)
{
    int tid, tmp = 1;
    while (tmp == 1)
        xchg(&m->guard, &tmp);

    if (queue_empty(m->q)) {
        m->locked = 0;
    } else {
        tid = queue_remove(m->q);
        m->owner = tid;
        unpark(tid);
    }
    m->guard = 0;
}
```



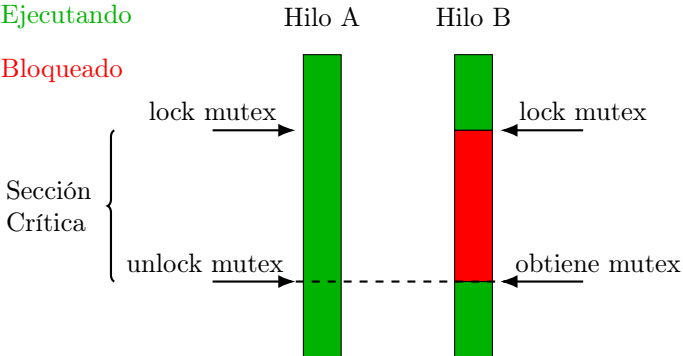
## Sección Crítica con Mutex

Ideales para implementar una sección crítica entre hilos de un mismo proceso

```
lock(m);  
<sección crítica>  
unlock(m);
```

Ejecutando

Bloqueado







# Mutex en POSIX

- Inicializar un mutex:

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        pthread_mutexattr_t *attr);
```

- Destruir un mutex:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Obtener el mutex o bloquear al hilo si el mutex lo tiene otro hilo:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Liberar el mutex:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Referencia: *Estándar POSIX, pthread*



## Suma paralela con mutex

```
pthread_mutex_t mtx;
unsigned long long total = 0;
void* suma(void* arg)
{
    struct param *p = (struct param*) arg;
    unsigned long long i, suma_local = 0;

    for (i = p->ni; i <= p->nf; i++)
        suma_local += i;

    pthread_mutex_lock(&mtx);
    total += suma_local;
    pthread_mutex_unlock(&mtx);

    return NULL;
}
```

Sección Crítica

# Resultados



## Suma paralela con mutex

```
> gcc -o parsum parsum.c -lpthread
> for i in 10 1000 500000000; do time ./parsum $i; done
Sumando de 1 a 10 (Gauss: 55)...
sum 1..10 = 55

real    0m0.004s
user    0m0.000s
sys 0m0.004s
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500

real    0m0.003s
user    0m0.000s
sys 0m0.004s
Sumando de 1 a 500000000 (Gauss: 125000000250000000)...
sum 1..500000000 = 125000000250000000

real    0m0.272s
user    0m1.072s
sys 0m0.000s
```

# Resultados



## Suma paralela con mutex

```
> while true; do ./parsum 1000; sleep 1; done
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
Sumando de 1 a 1000 (Gauss: 500500)...
sum 1..1000 = 500500
^C
```

## Ejercicio



Comparar los resultados de la solución propuesta con otra que no realice suma parcial, metiendo la sección crítica dentro del bucle.

- ¿Se nota alguna diferencia? Si es así, ¿a qué se debe?

# Agenda



- 1 Concurrencia
- 2 El problema de la Sección Crítica
- 3 Variables de Condición**
- 4 Semáforos
- 5 Memoria Compartida
- 6 Interbloqueos - Deadlocks

# Esperar a que a que otro haga algo



## Problema (tipo productor-consumidor)

- Un sistema utiliza dos hilos productores que van a producir N paquetes que un tercer hilo consumidor debe procesar.
- El sistema dispone de un buffer compartido de una posición, donde los productores pueden dejar el paquete, siempre que encuentren el buffer libre. Si el buffer está ocupado el productor debe esperar a que quede libre.
- El consumidor debe esperar a que el buffer tenga un paquete, cuando lo tenga lo podrá sacar y procesar, dejando libre el buffer para que alguno de los productores deje su siguiente paquete.

# Esquema de solución



```
pack_t *buffer = NULL;
```

```
void Producer(int n) {  
    pack_t *p;  
    while (n--) {  
        p = new_pack();  
  
        //Esperar buffer vacío  
  
        buffer = p;  
  
        //Avisar consumidor  
    }  
}
```

```
void Consumer(void) {  
    pack_t *p;  
    while (1) {  
        //Esperar buffer lleno  
  
        p = buffer;  
        buffer = NULL;  
  
        //Avisar productores  
  
        process_pack(p);  
    }  
}
```



# Variables de Condición



Mecanismo de sincronización que:

- Permite a un hilo bloquearse (esperar) si se cumple un predicado
  - operación `wait`
- Más de un hilo puede esperar en la misma variable de condición
- Cuando se produce un cambio en el predicado otros hilos pueden desbloquear a los hilos bloqueados en la variable de condición
  - `signal`: desbloquea a un hilo bloqueado en la variable de condición
  - `broadcast`: desbloquea a todos los hilos bloqueados en la variable de condición
- La comprobación del predicado debe hacerse en una sección crítica, creada con un `mutex`
  - el `mutex` es liberado antes de bloquearse en la variable de condición y vuelto a adquirir al desbloquearse de forma segura
  - en `posix threads` el `mutex` se pasa como argumento en la operación de `wait`
  - Los cambios en los predicados se harán en secciones críticas creadas con el mismo `mutex`



# Uso de Variables de Condición

- Hilo que espera hasta que se cumpla una condición:

```
lock(mutex);  
while (!<conditional expresion>)  
    wait(cond_var, mutex);  
<acciones restantes en sección crítica>  
unlock(mutex);  
<acciones deseadas fuera de sección crítica>
```

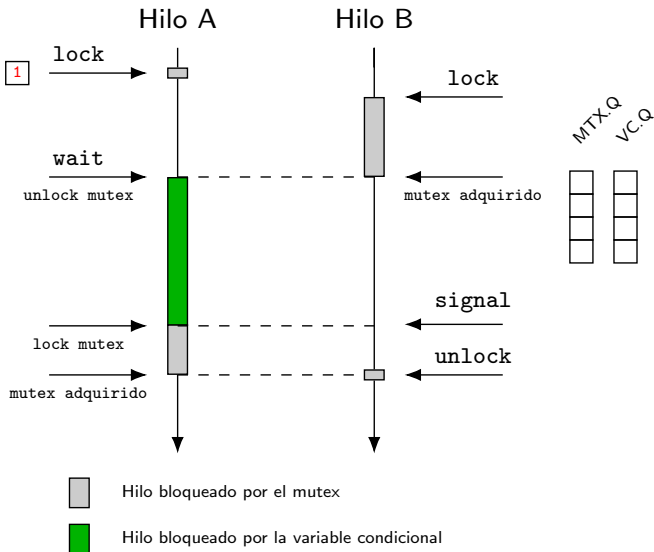
- Señalización desde otro hilo:

```
lock(mutex);  
<operaciones que ponen a true la expresión condicional>  
signal(cond_var);  
<otras operaciones protegidas>  
unlock(mutex);
```

- Es decir:

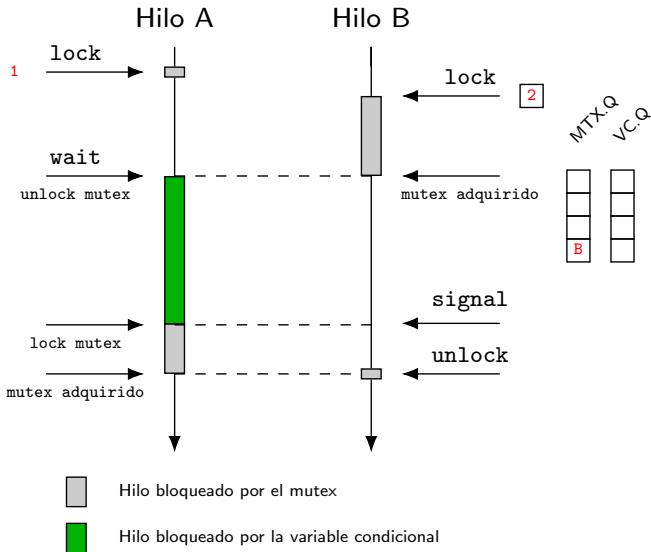
- wait: **siempre** en un bucle **while**
- wait: **siempre** con el mutex cogido
- signal: **mejor** con el mutex cogido

# Mutex + Variable de Condición (Monitor)

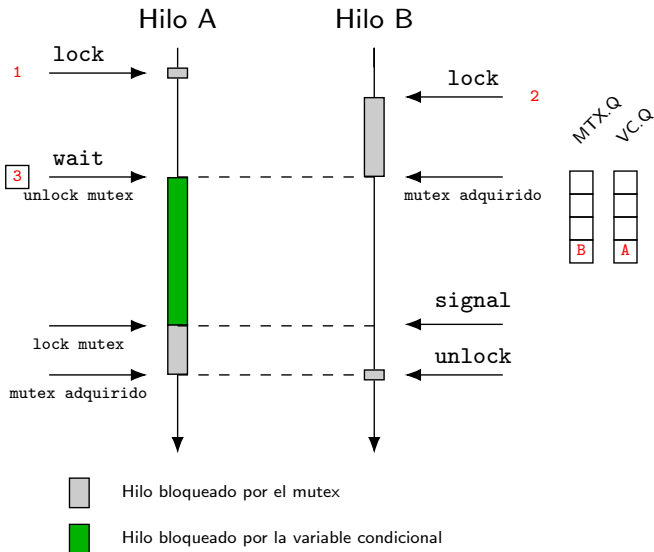




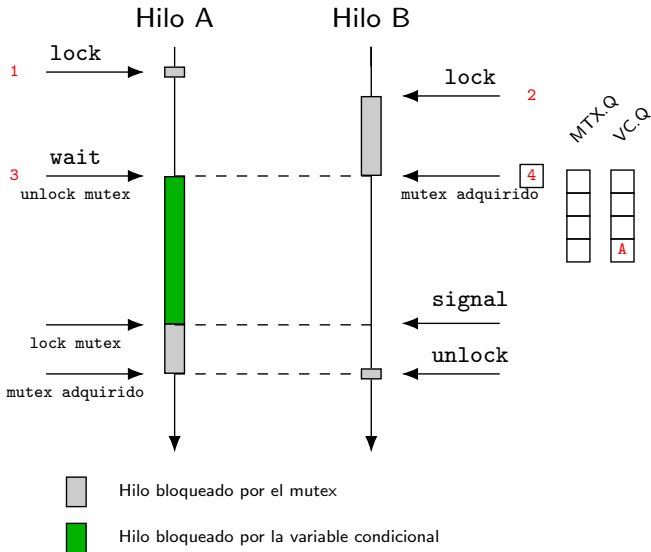
## Mutex + Variable de Condición (Monitor)



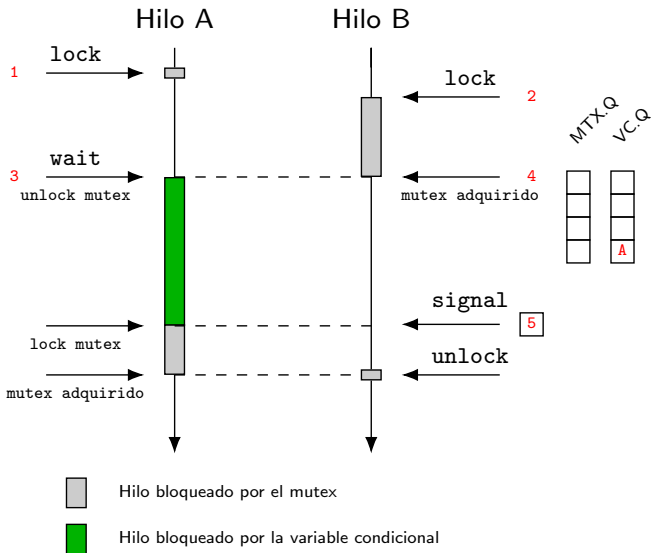
# Mutex + Variable de Condición (Monitor)



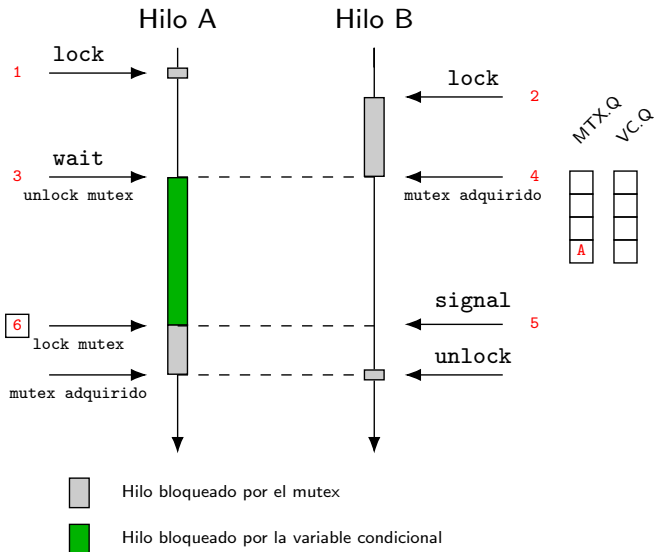
# Mutex + Variable de Condición (Monitor)



# Mutex + Variable de Condición (Monitor)

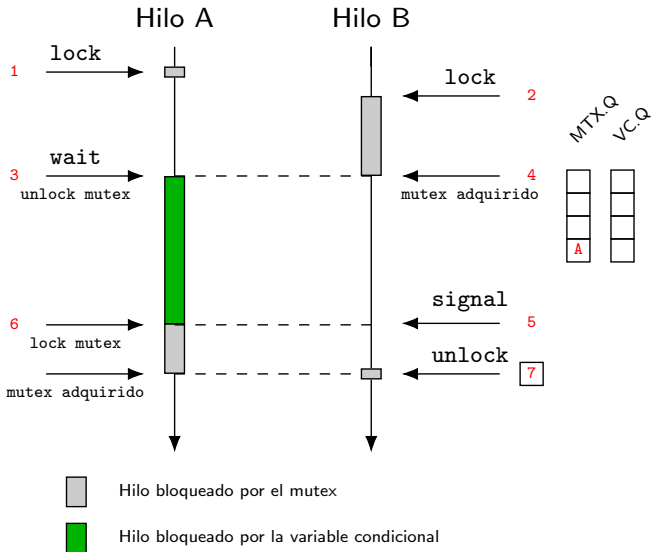


# Mutex + Variable de Condición (Monitor)

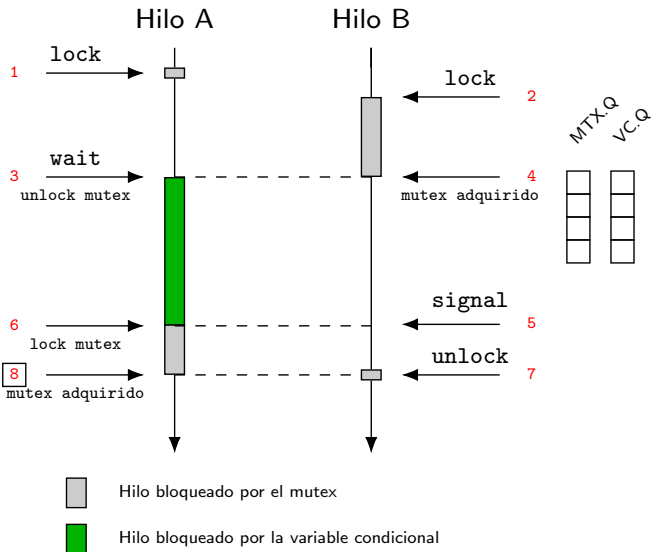




# Mutex + Variable de Condición (Monitor)



# Mutex + Variable de Condición (Monitor)



# Solución a nuestro problema



```
mutex_t m;  
cond_t empty, full;  
pack_t *buffer = NULL;
```

```
void Producer(int n) {  
    pack_t *p;  
    while (n-- > 0) {  
        p = new_pack();  
        mutex_lock(&m);  
        while (buffer != NULL)  
            cond_wait(&empty, &m);  
        buffer = p;  
        cond_signal(&full);  
        mutex_unlock(&m);  
    }  
}
```

```
void Consumer(void) {  
    pack_t *p;  
    while (1) {  
        mutex_lock(&m);  
        while (buffer == NULL)  
            cond_wait(&full, &m);  
        p = buffer;  
        buffer = NULL;  
        cond_signal(&empty);  
        mutex_unlock(&m);  
        process_pack(p);  
    }  
}
```



# POSIX: Variables de Condición

- Inicialización de variable condicional

```
int pthread_cond_init(pthread_cond_t*cond,  
pthread_condattr_t*attr);
```

- Destrucción de variable condicional

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Operación signal

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Operación broadcast

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Operación wait

```
int pthread_cond_wait(pthread_cond_t*cond, pthread_mutex_t*mutex);
```

# Insistimos, el while es imprescindible



```
pthread_cond_wait
```

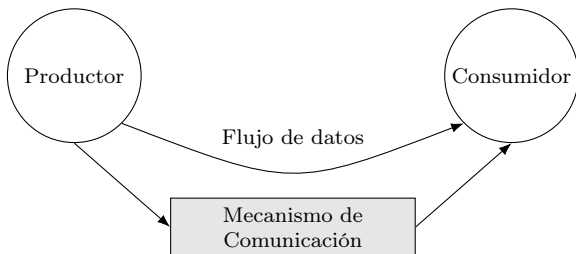
```
# man pthread_cond_wait
```

```
....
```

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_timedwait()` or `pthread_cond_wait()` functions may occur. Since the return from `pthread_cond_timedwait()` or `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

```
....
```

# Problema del Productor-Consumidor



- Varios hilos producen datos
- Varios hilos los consumen
- Buffer intermedio para desacoplar productores y consumidores
- Los Productores deben
  - esperar a que haya hueco libre
  - señalar que han añadido datos
- Los consumidores deben
  - esperar a que haya datos
  - señalar que han extraído datos (dejando hueco libre)

# Productor-Consumidor con Mutex y VC



```
#define MAX_BUFFER          1024      /* tamaño del buffer */
#define DATA_TO_PRODUCE    100000    /* datos a producir */

pthread_mutex_t mutex;      /*mutex para buffer compartido*/
pthread_cond_t full;        /*buffer tiene elementos*/
pthread_cond_t empty;       /*buffer tiene huecos (espacio libre)*/

int buffer[MAX_BUFFER];     /*buffer comun*/
int num_elem = 0;           /*numero de elementos en el buffer*/
int wr_pos = 0;             /*posición de escritura*/
int rd_pos = 0;             /*posición de lectura*/
```

# Productor-Consumidor con Mutex y VC



```
void* Producer(void *arg)
{
    int data, i;

    for(i=0; i < DATA_TO_PRODUCE; i++ ) {
        data = produce_data();          /* producir data */
        pthread_mutex_lock(&mutex);     /* acceder al buffer */
        while (num_elem == MAX_BUFFER) /*si buffer full */
            pthread_cond_wait(&empty, &mutex); /*se bloquea */
        buffer[wr_pos] = data;
        wr_pos = (wr_pos + 1) % MAX_BUFFER;
        num_elem++;
        pthread_cond_signal(&full);    /* buffer con elementos */
        pthread_mutex_unlock(&mutex);
    }
}
```



# Productor-Consumidor con Mutex y VC



```
void* Consumer(void *arg)
{
    int data, i;

    for(i=0; i < DATA_TO_PRODUCE; i++ ) {
        pthread_mutex_lock(&mutex);          /* acceder al buffer */
        while (num_elem == 0)                /* si buffer empty */
            pthread_cond_wait(&full, &mutex); /* se bloquea */
        data = buffer[rd_pos];
        rd_pos = (rd_pos + 1) % MAX_BUFFER;
        num_elem--;
        pthread_cond_signal(&empty);         /* huecos en buffer */
        pthread_mutex_unlock(&mutex);
        consume_data(data);
    }
}
```

# Producer-Consumidor Simplificado



```
buffer[MAX_BUFFER];  
indProd = 0, indCons = 0;  
int num_elem = 0;  
  
mutex_t mutex;  
cond_t event;
```

```
void Producer() {  
    ...  
    element = produce();  
    mutex_lock(&mutex)  
    while( num_elem == MAX_BUFFER)  
        cond_wait(&event, &mutex);  
    insert(element);  
    cond_broadcast(&event);  
    mutex_unlock(&mutex);  
    ...  
}
```

```
void Consumer() {  
    ...  
    mutex_lock(&mutex)  
    while( !num_elem )  
        cond_wait(&event, &mutex);  
    element = extract();  
    cond_broadcast(&event);  
    mutex_unlock(&mutex);  
    consume(element);  
    ...  
}
```

# Agenda



- 1 Concurrencia
- 2 El problema de la Sección Crítica
- 3 Variables de Condición
- 4 Semáforos**
- 5 Memoria Compartida
- 6 Interbloqueos - Deadlocks



# Semáforos (Dijkstra'65)

- Mecanismo de sincronización para hilos en una misma máquina
- Un atributo entero ( $s$ )
  - Invariante: si  $s < 0 \implies |s|$  es el número de hilos bloqueados en el semáforo
- Dos operaciones **atómicas**

```
wait(s) { //P
    s = s - 1;
    if (s < 0) {
        //Bloquear al proceso
    }
}
```

```
post(s) { //V
    s = s + 1;
    if (s <= 0) {
        //Desbloquear a un proceso bloqueado por wait
    }
}
```

# Semáforos POSIX



## ■ Semáforos anónimos

- Inicialización (shared memoria compartida)

```
int sem_init(sem_t *sem, int shared, int val);
```

- Destrucción

```
int sem_destroy(sem_t *sem);
```

## ■ Semáforos con nombre

- Creación o apertura de semáforo creado

```
sem_t* sem_open(char*name,int flag,mode_t mode,int val);
```

- Cierre

```
int sem_close(sem_t *sem);
```

- Borrado

```
int sem_unlink(char *name);
```

## ■ Ambos:

- Operación wait

```
int sem_wait(sem_t *sem);
```

- Operación signal

```
int sem_post(sem_t *sem);
```



# Semáforos usados como locks

```
sem_t m;  
sem_init(&m, 0, 1);  
  
sem_wait(&m);  
-- critical section --  
sem_post(&m);
```

Ojo, no es un mutex. Algunas diferencias relevantes son:

- Un mutex tiene un propietario, y sólo éste lo puede liberar
  - Cualquiera puede hacer un post a un semáforo, aunque no haya hecho wait en él
- Si hacemos más posts que waits, dejará de comportarse como un lock
- Un mutex puede inicializarse con el atributo recursivo (avanzado), los semáforos no (no tiene sentido)

# Semáforos usados como colas de espera



```
sem_t q;  
sem_init(&q, 0, 0);  
  
...  
if(!condition)  
    sem_wait(&q);
```

Ojo, no son variables de condición:

- Un signal a una variable de condición se pierde si ningún hilo está bloqueado en el momento del signal
  - Un post a un semáforo tiene efecto a futuro
  - Hay que tener mucho cuidado de no hacer posts de más
- No están asociados a ningún mutex, no liberan ningún recurso cuando un hilo queda bloqueado
  - Es fácil equivocarse y provocar interbloqueos (deadlocks)



# Producer-Consumidor: intento 1

```
sem_t holes, elements;  
sem_init(&holes, 0, N);  
sem_init(&elements, 0, 0);
```

```
void Producer(void) {  
    int data;  
    int i;  
  
    for (i = 0; i < PROD; i++) {  
        data = producir_data();  
        sem_wait(&holes);  
        buffer[prod] = data;  
        prod = (prod + 1) % MAX_BUF;  
        sem_post(&elements);  
    }  
}
```

```
void Consumer(void) {  
    int data;  
    int i;  
  
    for (i = 0; i < PROD; i++) {  
        sem_wait(&elements);  
        data = buffer[con];  
        cons = (cons + 1) % MAX_BUF;  
        sem_post(&holes);  
        cosumir_data(data);  
    }  
}
```

- ¿Qué pasa si hay varios productores o varios consumidores simultáneamente accediendo al buffer?



## Intento 2: accedemos en sección crítica



```
sem_t holes, elements, common;
sem_init(&holes, 0, N);
sem_init(&elements, 0, 0);
sem_init(&common, 0, 1);
```

```
void Producer(void) {
    int data;
    int i;

    for (i = 0; i < PROD; i++) {
        data = producir_data();
        sem_wait(&common);
        sem_wait(&holes);
        buffer[prod] = data;
        prod = (prod + 1) % MAX_BUF;
        sem_post(&elements);
        sem_post(&common);
    }
}
```

```
void Consumer(void) {
    int data;
    int i;

    for (i = 0; i < PROD; i++) {
        sem_wait(&common);
        sem_wait(&elements);
        data = buffer[con];
        cons = (cons + 1) % MAX_BUF;
        sem_post(&holes);
        sem_post(&common);
        cosumir_data(data);
    }
}
```

- Eh... pero si llega primero el consumidor, coge el *cerrojo* common y se bloquea en el semáforo de elementos... **no libera el cerrojo** (a diferencia del wait en una variable de condición)



## Solución: sección crítica lo último

```
sem_t holes, elements, common;
sem_init(&holes, 0, N);
sem_init(&elements, 0, 0);
sem_init(&common, 0, 1);
```

```
void Producer(void) {
    int data;
    int i;

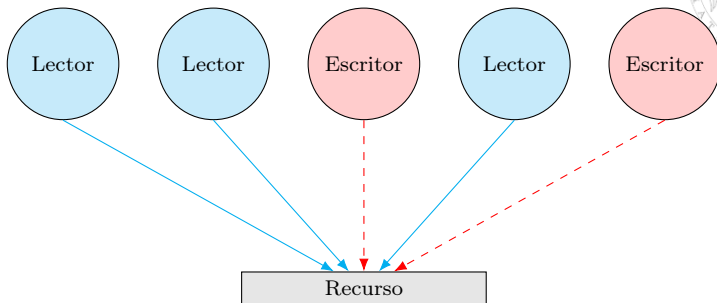
    for (i = 0; i < PROD; i++) {
        data = producir_data();
        sem_wait(&holes);
        sem_wait(&common);
        buffer[prod] = data;
        prod = (prod + 1) % MAX_BUF;
        sem_post(&common);
        sem_post(&elements);
    }
}
```

```
void Consumer(void) {
    int data;
    int i;

    for (i = 0; i < PROD; i++) {
        sem_wait(&elements);
        sem_wait(&common);
        data = buffer[con];
        cons = (cons + 1) % MAX_BUF;
        sem_post(&common);
        sem_post(&holes);
        consumir_data(data);
    }
}
```

- Cuidado: al bloquearse en un semáforo con un cerrojo cogido el cerrojo no se libera

## Otro clásico: Lectores y Escritores



- **Lectores:** acceden sólo para consultar, no modifican el recurso
- **Escritores:** acceden para modificar el recurso
- Los escritores deben acceder en exclusiva
- Pueden acceder varios lectores simultáneamente (siempre que no acceda ningún escritor)



# Lectores y Escritores con semáforos

Prioridad a lectores:

- Los escritores compiten entre sí por un lock
- Sólo el primer lector compite por este lock
- Otro lock para esperar a que entre el primer lector

```
sem_t swriters, sreaders;  
sem_init(&swriters, 0, 1);  
sem_init(&sreaders, 0, 1);
```

```
void Reader(void) {  
    while (1) {  
        sem_wait(&sreaders);  
        if (++num_readers == 1)  
            sem_wait(&swriters);  
        sem_post(&sreaders);  
  
        printf("%d\n", data);  
  
        sem_wait(&sreaders);  
        if (--num_readers == 0)  
            sem_post(&swriters);  
        sem_post(&sreaders);  
    }  
}
```

```
void Writer(void) {  
    while (1) {  
        sem_wait(&swriters);  
        data = data + 2;  
        sem_post(&swriters);  
    }  
}
```

```
//RETO: obtener una  
//implementación con  
//mutex y variables de  
//condición
```

# Agenda



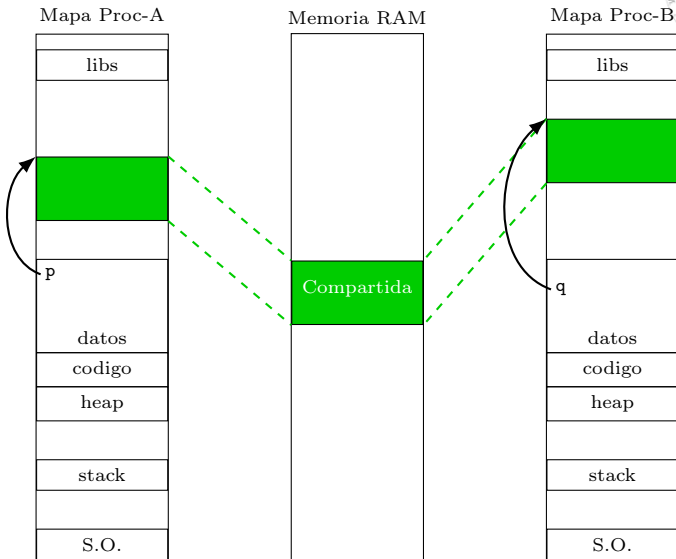
- 1 Concurrencia
- 2 El problema de la Sección Crítica
- 3 Variables de Condición
- 4 Semáforos
- 5 Memoria Compartida**
- 6 Interbloqueos - Deadlocks

# Memoria compartida en POSIX



- Entre hilos de un mismo proceso:
  - Implícita: regiones de datos y heap son compartidas (del proceso)
- Entre (hilos de) procesos *sin relación de parentesco*
  - Regiones marcadas como compartidas con soporte en el mismo fichero (`mmap`)
  - Regiones de memoria compartida con nombre (`shm_open`)
- Entre (hilos de) procesos padre e hijo
  - Regiones creadas por el padre antes del `fork` y marcadas como compartidas (`mmap`)
  - Regiones marcadas como compartidas con soporte en el mismo fichero (`mmap`)
  - Regiones de memoria compartida con nombre

# Memoria compartida entre procesos



# POSIX: mmap



```
void* mmap(void *addr, size_t len, int prot, int flags,  
           int fd, off_t offset);
```

Crea una región nueva en el mapa de memoria del proceso, y según los flags activos en la máscara flags:

- **MAP\_PRIVATE**: región privada, se copia como una región más (*copy on write*) y no se actualiza el fichero. No se especifica si los cambios en el fichero afectarán a la región privada.
- **MAP\_SHARED**: región compartida, los cambios son visibles en cualquier proceso que tenga mapeado el mismo fichero
- **MAP\_ANON**: memoria sin respaldo en fichero, inicializada a 0. El parámetro `fd` debe ponerse a -1 y `offset` a 0.

El parámetro `prot` fija los permisos de acceso (`PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`)



# POSIX: munmap y msync



```
int munmap(void *addr, size_t len);
```

- Elimina la región mapeada de len bytes a partir de la dirección addr del mapa del proceso, quedando la región como *no válida*
  - En una región con respaldo el SO escribirá (quizá no inmediatamente) los cambios en el fichero asociado
  - En caso contrario el SO deshecha los cambios

```
int msync(void *addr, size_t len, int flags);
```

- En una región con respaldo se actualiza el fichero:
  - MS\_SYNC: bloqueando el hilo hasta que se complete la operación
  - MS\_ASYNC: planificando la operación pero retornando inmediatamente (sin completarla)
  - MS\_INVALIDATE: invalida otros mapeos del mismo fichero

# POSIX: memoria compartida con nombre



```
int shm_open(const char *name, int oflag, mode_t mode);
```

- Crea una *objeto de memoria compartida* con nombre `name`, con los permisos especificados por `oflag` devolviendo su descriptor de fichero. Análoga a `open`.
- Suele después mapearse en memoria con `mmap`

```
int shm_unlink(const char *name);
```

- Análoga a `unlink`

# Prod-Cons con memoria compartida



## ■ 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 el archivo esté creado para abrirlo (`open`)
- Proyecta el archivo en su espacio de direcciones (`mmap`)
- Utiliza la zona de memoria compartida
- Cierra el archivo

# Productor



```
#define MAX_BUFFER          1024      /* tamaño del buffer */
#define DATA_TO_PRODUCE    100000    /* datos a producir */

sem_t *elements; /* elementos en el buffer */
sem_t *holes;     /* huecos en el buffer */

void Producer(int *buffer)
{
    int pos = 0;
    int dato;
    int i;

    for(i=0; i < DATA_TO_PRODUCE; i++ ) {
        dato = produce_data();
        sem_wait(holes);
        buffer[pos] = dato;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(elements);
    }
}
```

# Producer



```
int main(int argc, char *argv[])
{
    int shd, *buffer;

    shd = open("BUFFER", O_CREAT|O_RDWR, 0700);
    ftruncate(shd, MAX_BUFFER * sizeof(int));

    buffer = (int*) mmap(NULL, MAX_BUFFER * sizeof(int),
        PROT_WRITE, MAP_SHARED, shd, 0);
    elements = sem_open("/ELEMENTS", O_CREAT, 0700, 0);
    holes = sem_open("/HOLES", O_CREAT, 0700, MAX_BUFFER);
    Producer(buffer);

    munmap(buffer, MAX_BUFFER * sizeof(int));
    close(shd);
    unlink("BUFFER");
    sem_close(elements);
    sem_close(holes);
    sem_unlink("/ELEMENTS");
    sem_unlink("/HOLES");
    return 0;
}
```

# Consumidor



```
#define MAX_BUFFER          1024    /* tamaño del buffer */
#define DATA_TO_PRODUCE    100000  /* datos a producir */

sem_t *elements; /* elementos en el buffer */
sem_t *holes;    /* huecos en el buffer */

void Consumer(int *buffer)
{
    int pos = 0;
    int i, dato;

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

# Consumidor



```
int main(int argc, char *argv[]){
    int shd, *buffer;

    shd = open("BUFFER", O_RDONLY);
    buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int),
        PROT_READ, MAP_SHARED, shd, 0);
    elements = sem_open("/ELEMENTS", 0);
    holes    = sem_open("/HOLES", 0);

    Consumer(buffer);

    munmap(buffer, MAX_BUFFER * sizeof(int));
    close(shd);
    sem_close(elements);
    sem_close(holes);
    return 0;
}
```

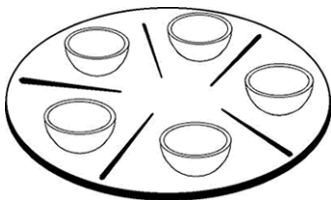
# Agenda



- 1 Concurrencia
- 2 El problema de la Sección Crítica
- 3 Variables de Condición
- 4 Semáforos
- 5 Memoria Compartida
- 6 Interbloqueos - Deadlocks**



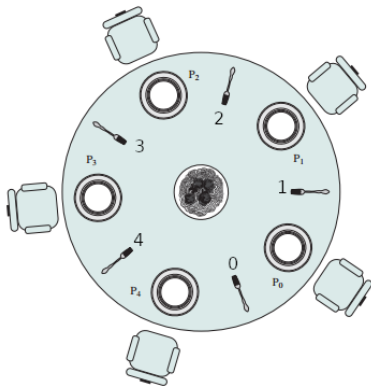
# Filósofos Comensales (Dijkstra'65)



- Cinco filósofos sentados en una mesa piensan y comen arroz:
  - Ningún filósofo debe morir de hambre (evitar bloqueo)
  - Necesitan 2 palillos para comer, que se cogen de uno en uno
  - Emplean un tiempo finito en comer y pensar
- Algoritmo:
  - Pensar...
  - Coger un palillo, coger el otro, comer, soltar un palillo y soltar el otro
  - Pensar...
- Problema: imaginemos que todos los comensales han cogido el palillo de su izquierda... ¿Quién come?



# Primer intento: interbloqueo (deadlock)



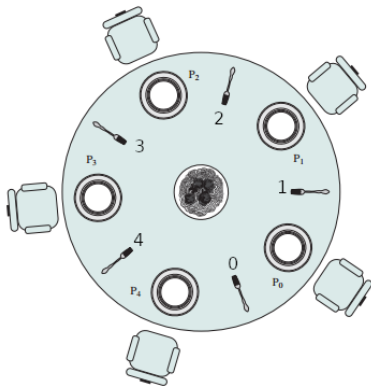
```
#define N 5
```

```
void philosopher(int i)
{
    while (1) {
        think();
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork((i + 1) % N);
        put_fork(i);
    }
}
```

Si todos cogen el de su izquierda → Deadlock



# Turno rotativo

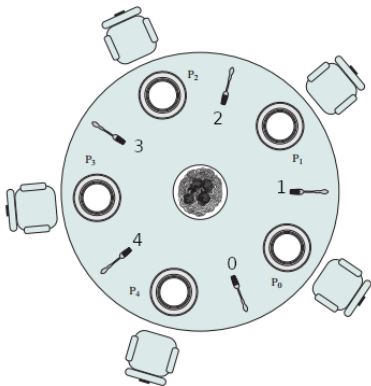


```
#define N 5
sem_t turn[N] = {1,0};

void philosopher(int i)
{
    while (1) {
        think();
        sem_wait(&turn[i]);
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork((i + 1) % N);
        put_fork(i);
        post(&turn[(i+1) % N]);
    }
}
```

Sólo puede comer un filósofo en cada *turno*, los demás tienen que esperar aunque el que tiene el *turno* no quiera comer

# Camarero



```
#define N 5
mutex_t waiter;

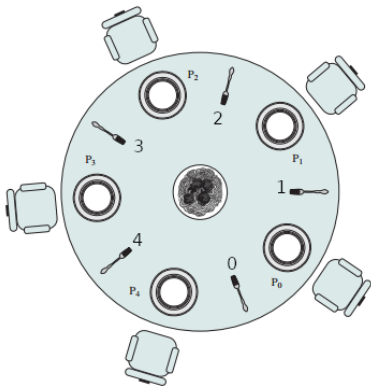
void philosopher(int i)
{
    int L = i;
    int R = (i + 1) % N;

    while (1) {
        think();
        lock(waiter);
        take_fork(R);
        take_fork(L);
        eat();
        put_fork(R);
        put_fork(L);
        unlock(waiter);
    }
}
```

Sólo puede comer un filósofo cada vez, aunque no hay un orden establecido



# Coger los dos o ninguno



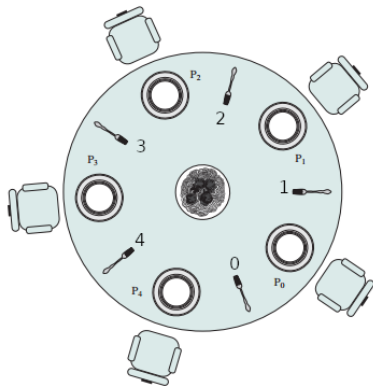
```
#define N 5
```

```
void philosopher(int i)
{
    int L = i;
    int R = (i + 1) % N;

    while (1) {
        think();
        while(1) {
            take_fork(L);
            if (try_take_fork(R))
                break;
            put_fork(L);
            sleep(T);
        }
        eat();
        put_fork(R);
        put_fork(L);
    }
}
```

Sólo hay deadlock si todos los filósofos ejecutan sincronamente haciendo exactamente lo mismo (muy poco probable si T es aleatorio)

# Solución de Tanenbaum



- Antes de comer un filósofo debe hacer un wait en su semáforo
- Antes de hacer este wait se entra en una sección crítica para ver si se puede comer
- Para poder comer los de su lado no deben estar comiendo
- Se usan estados para indicar si un filósofo está pensando, quiere comer o está comiendo
- Si los de al lado no están comiendo, se hace un post al semáforo para no quedarse bloqueado
- Puede codificarse algo similar con mutex y variables de condición

# Solución de Tanenbaum



```
#define N 5
#define LEFT(i) ((i+N-1) % N)
#define RIGHT(i) ((i+1) % N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2
mutex_t mtx;
int state[N];
sem_t s[N] = 0;

void test(int i)
{
    if (state[i] == HUNGRY
        && state[LEFT(i)] != EATING
        && state[RIGHT(i)] != EATING) {
        state[i] = EATING;
        post(s[i]);
    }
}

void philosopher(int i)
{
    while (1) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

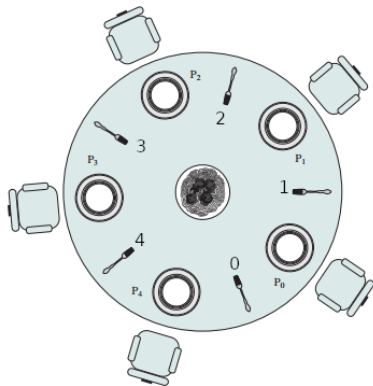
void take_forks(int i)
{
    lock(mtx);
    state[i] = HUNGRY;
    test(i);
    unlock(mtx);
    wait(s[i]);
}

void put_forks(int i)
{
    lock(mtx);
    state[i] = THINKING;
    test(LEFT(i));
    test(RIGHT(i));
    unlock(mtx);
}
```

Puede haber inanición si dos filósofos comen de forma alterna (ejemplo: P0 y P3, P4 tiene inanición)



# Orden global de recursos



```
#define N 5
```

```
void philosopher(int i)
{
    int first  = min(i, (i+1) % N);
    int second = max(i, (i+1) % N);

    while (1) {
        think();
        take_fork(first);
        take_fork(second);
        eat();
        put_fork(second);
        put_fork(first);
    }
}
```

- Solución simple, permite a varios filósofos comer simultáneamente.
- Si la presión sobre los recursos es grande se termina secuencializando.
- Utilizada extensivamente en el kernel de Linux.