Sistemas Operativos

2018-2019

Comunicación y Sincronización entre Procesos

Basado en: Sistemas Operativos
J. Carretero [et al.]

Contenido

- Procesos concurrentes
- Problemas clásicos
- Mecanismos C&S
  - Semáforos
  - Monitores: Mutex y Variables Condicionales
  - Memoria compartida
  - Señales
- Interbloqueos

Procesos concurrentes

- Modelos
  - Multiprogramación en un único procesador
  - Multiprocesador
  - Multicomputador (proceso distribuido)
- Razones
  - Compartir recursos físicos
  - Compartir recursos lógicos
  - Acelerar los cálculos
  - Modularidad
  - Comodidad

Contenido

- Procesos concurrentes
- Problemas clásicos
- Mecanismos C&S
  - Semáforos
  - Monitores: Mutex y Variables Condicionales
  - Memoria compartida
  - Señales
- Interbloqueos

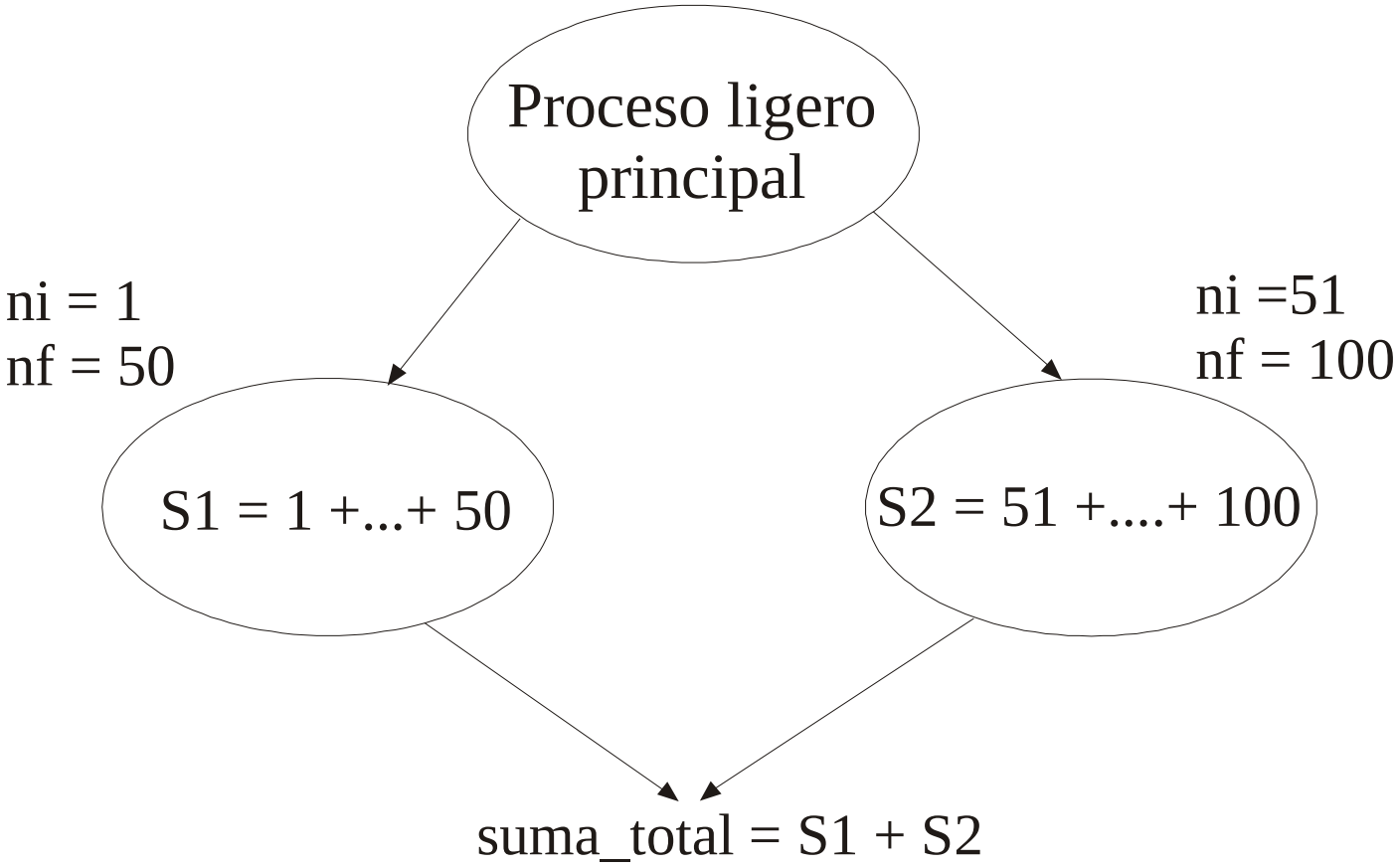
Problemas clásicos de comunicación y sincronización

- **El problema de la sección crítica**
- El problema del productor-consumidor
- El problema de los lectores-escritores
- Comunicación cliente-servidor
- Problema de los filósofos comensales

Problema de la sección crítica

- Supongamos un sistema compuesto por *n* hilos
- Cada uno tiene un fragmento de código que accede/modifica un recurso compartido:
  - *Sección crítica*
- Queremos que sólo uno de los hilos en cada instante pueda ejecutar su sección crítica

Ejemplo 1


Proceso ligero principal
ni = 1
nf = 50
ni =51
nf = 100
S1 = 1 +...+ 50
S2 = 51 +....+ 100
suma_total = S1 + S2

Ejemplo 1

- Calcula la suma de los *N* primeros números utilizando procesos ligeros.

```
int suma_total = 0; // Var compartida
void suma_parcial(int ni, int nf) {
    int j = 0;
    int suma_parcial = 0; // Var. privada
    for (j = ni; j <= nf; j++)
        suma_parcial = suma_parcial + j;
    suma_total = suma_total + suma_parcial;
    pthread_exit(0);
 }
```

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

Ejemplo 1

- Posible codificación en ensamblador para el cálculo de *suma_total*:

```
suma_total = suma_total + suma_parcial;
```

```
LDR R1, suma_total          #R1=0 (la 1ª vez)
LDR R2, suma_parcial        #R2=1275
ADD R1,R1,R2                #R1=1275
STR R1, suma_total          #suma_total=1275
```

Ejemplo 1

- Posible situación de conflicto:

```
   LDR  R1, suma_total      #R1=0
   LDR  R2, suma_parcial    #R2=1275

############## Cambio de contexto ##############
   LDR  R1, suma_total      #R1=0
   LDR  R2, suma_parcial    #R2=3775
   ADD   R1, R1, R2         #R1=3775
   STR R1, suma_total       #suma_total=3775

############## Cambio de contexto ##############
   ADD   R1, R1, R2          #R1=1275
   STR   R1, suma_total      #suma_total=1275
```

Ejemplo con sección crítica

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

```
void suma_parcial(int ni, int nf) {
        int j = 0;
        int suma_parcial = 0;
        for (j = ni; j <= nf; j++)
             suma_parcial = suma_parcial + j;

         <Entrada en la sección crítica>
        suma_total = suma_total + suma_parcial;
         <Salida de la sección crítica>
        pthread_exit(0);
}
```

Ejemplo 2

```
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.
- Solución: secciones críticas

Ejemplo 2 con sección crítica

```
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 sección crítica:
  - **Exclusión mutua**: sólo un proceso en la región crítica
  - **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
  - **Espera limitada**: ningún proceso debe esperar indefinidamente para entrar en su región crítica
- Hay que tener también en mente:
  - Un proceso no debe ver retrasado el acceso a su sección crítica cuando no hay ningún otro proceso usándola
  - 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

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 bloqueado

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: *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 y uso 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, ininterrumplible
- Intercambia dos valores (potencialmente, ambos en memoria)
  - En Intel, sólo uno de los dos (`src` o `dst`) pueden estar en memoria

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

```
//puede estar en registro
tmp = 1;
//Espera activa
while( tmp== 1)
    xchg(dirM, tmp);
Sección_crítica();
*dirM= 0;
```

Semántica y uso de LL/SC

Load Link

```
ll src
   rout← Mem [src]
```

Store Cond.

```
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

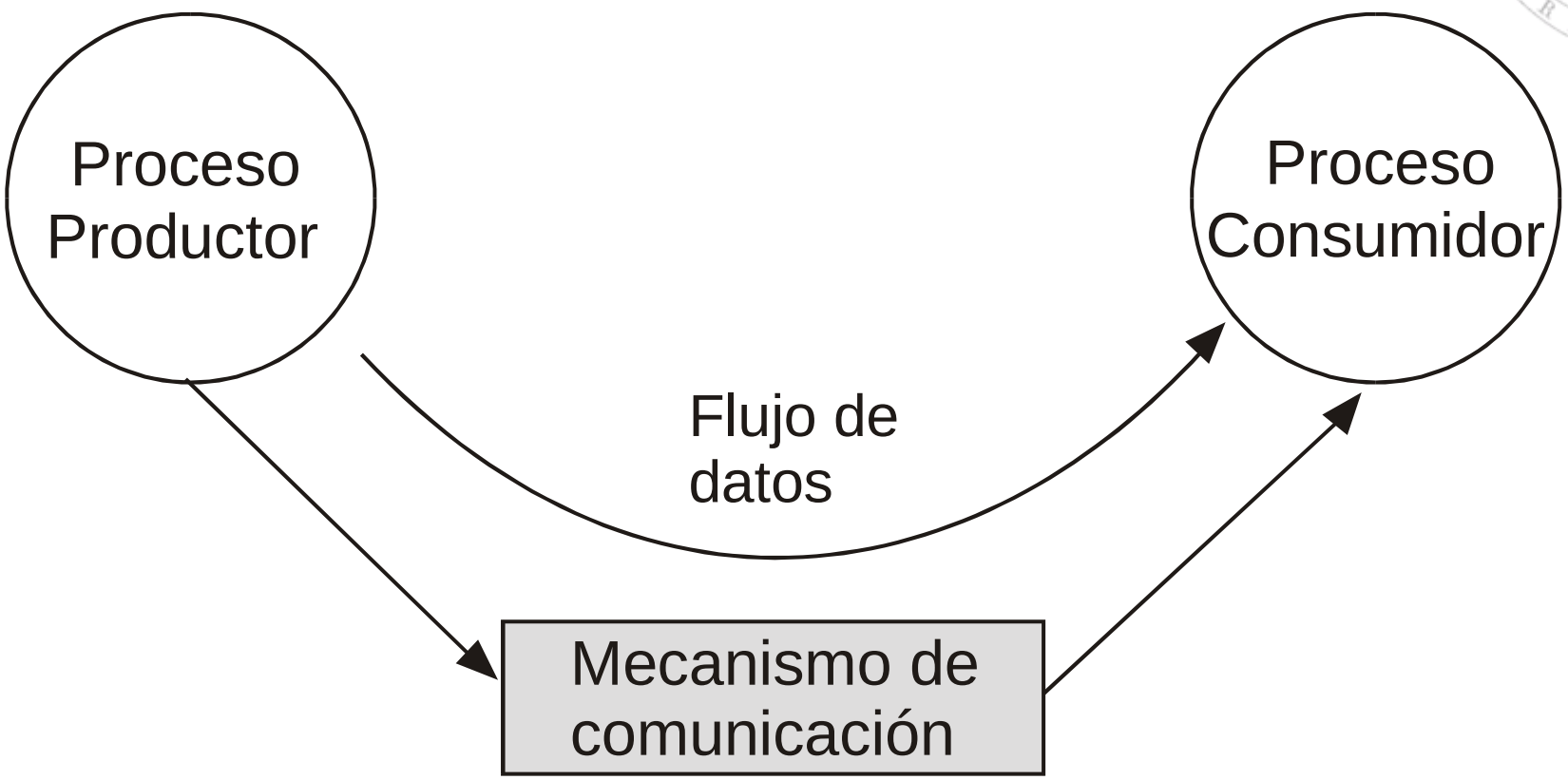
Solución al problema de la **Sección Crítica con LL/SC**

```
while (1) {
   while(ll(dirM)== 1);
   if (sc(dirM,1)==1) break;
   //si no, otra vez al Load-Link
}
Sección_crítica();
*dirM= 0;
```

Problemas clásicos de comunicación y sincronización

- El problema de la sección crítica
- **El problema del productor-consumidor**
- El problema de los lectores-escritores
- Comunicación cliente-servidor
- Problema de los filósofos comensales

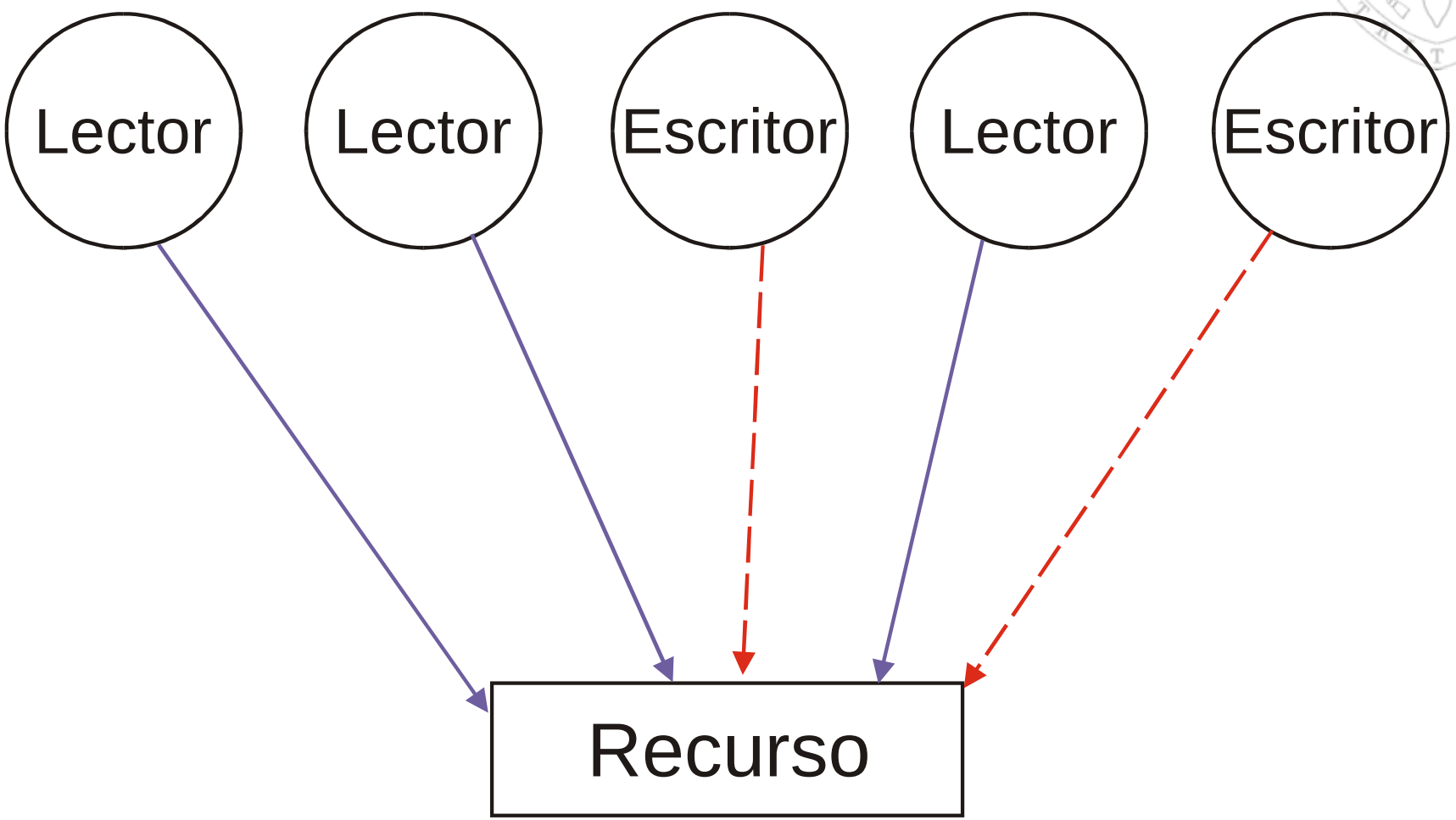
Problema del productor-consumidor



Problemas clásicos de comunicación y sincronización

- El problema de la sección crítica
- El problema del productor-consumidor
- **El problema de los lectores-escritores**
- Comunicación cliente-servidor
- Problema de los filósofos comensales

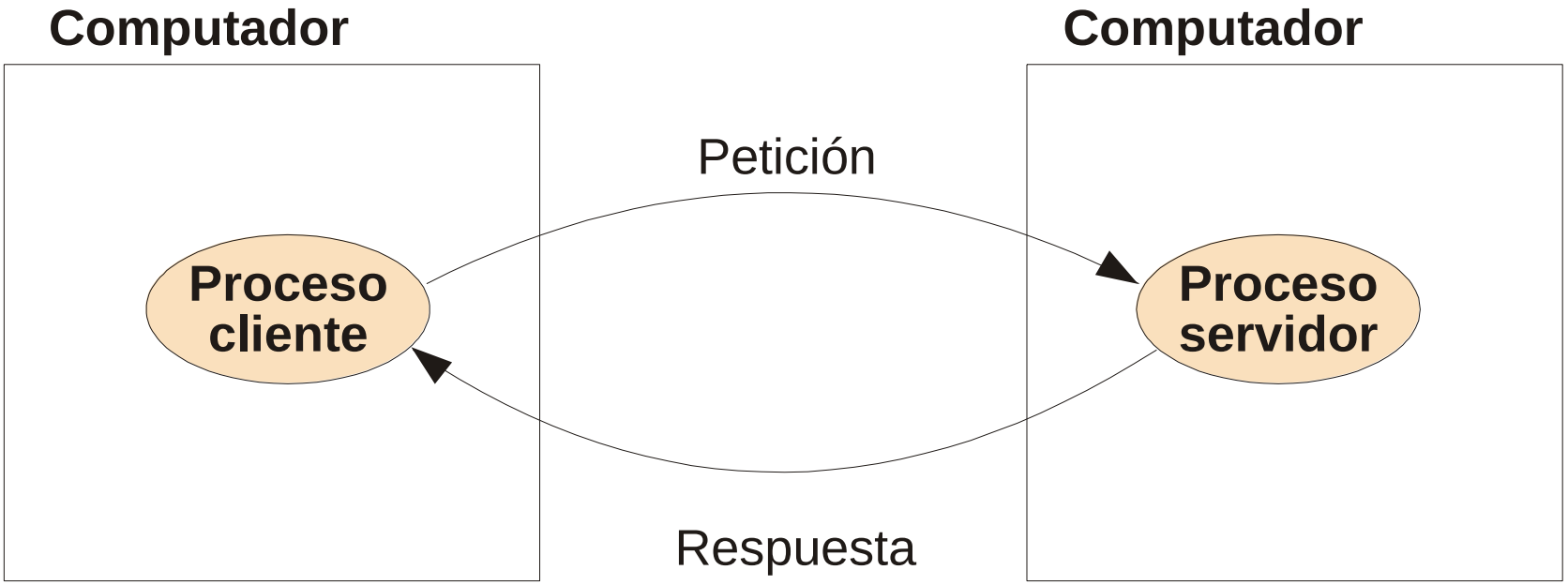
El problema de los lectores-escritores



Problemas clásicos de comunicación y sincronización

- El problema de la sección crítica
- El problema del productor-consumidor
- El problema de los lectores-escritores
- **Comunicación cliente-servidor**
- Problema de los filósofos comensales

Comunicación cliente-servidor

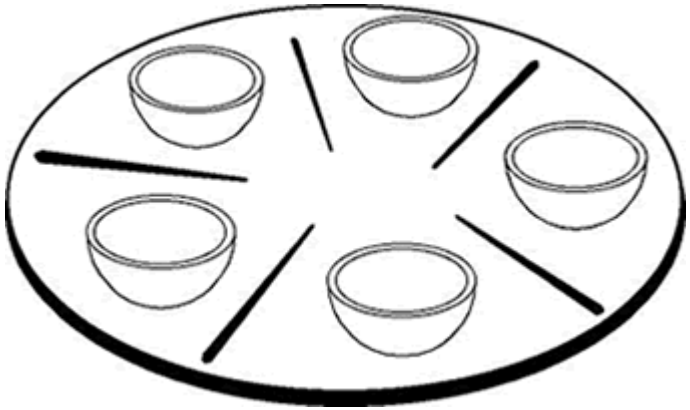


Problemas clásicos de comunicación y sincronización

- El problema de la sección crítica
- El problema del productor-consumidor
- El problema de los lectores-escritores
- Comunicación cliente-servidor
- **Problema de los filósofos comensales**

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…



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

- Procesos concurrentes
- Problemas clásicos
- Mecanismos C&S
  - Cerrojos y Variables Condicionales
  - Semáforos
  - Tuberías
  - Memoria compartida
- Interbloqueos

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
  - No estudiaremos cómo se implementan sino cómo se usan

Mecanismos de comunicación

- Archivos
- Tuberías (pipes, FIFOS)
  - No las estudiaremos
- Memoria compartida
  - Implícita: hilos
  - Explícita: necesidad de una API específica

Mecanismos de Sincronización

- Servicios del sistema operativo:
  - Señales: asincronas y no encolables (no las estudiaremos)
  - Tuberías (pipes, FIFOS) (no las estudiaremos)
  - Semáforos
  - Cerrojos y variables condicionales
- Las operaciones de sincronización deben ser **atómicas**

Cerrojos (mutex)

- Un cerrojo es un mecanismo de sincronización indicado para hilos.
  - Ideal para el problema de la sección crítica, pues garantiza exclusión mutua….
- Podemos pensar en un cerrojo como un objeto con 3 atributos y 2 métodos ***atómicos***

```
// Cerrojo abierto o cerrado
estado_t estado;

// Cola de hilos bloqueados
queue_t q;

//Hilo "propietario"
hilo_id owner;
```

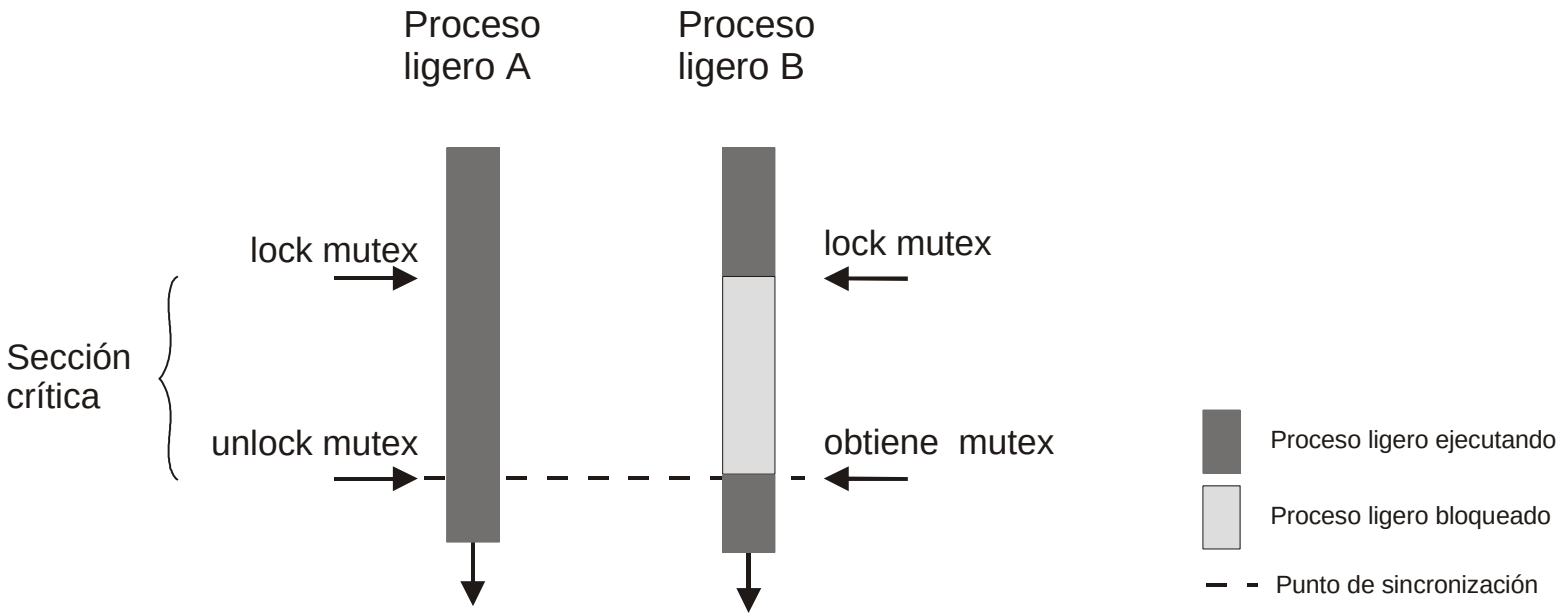
```
lock(m) {
   if (m->estado==cerrado) {
      queue_add(m->q, esteHilo);
     suspenderHilo;
    }
     m-> estado=cerrado;
     m->owner = esteHilo;

}
```

```
unlock(m) {
 if (m->owner==esteHilo) {
    m->estado=abierto;
    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 realizarla el proceso ligero que ejecutó `lock`



Servicios POSIX

- `int` **`pthread_mutex_init`**`(pthread_mutex_t *`**`mutex`**`, pthread_mutexattr_t *attr);`
  - Inicializa un mutex.
- `int` **`pthread_mutex_destroy`**`(pthread_mutex_t*`**`mutex`**`);`
  - Destruye un mutex.
- `int` **`pthread_mutex_lock`**`(pthread_mutex_t *`**`mutex`**`);`
  - Intenta obtener el `mutex`. Bloquea al proceso ligero si el `mutex` se encuentra adquirido por otro proceso ligero.
- `int` **`pthread_mutex_unlock`**`(pthread_mutex_t *`**`mutex`**`);`
  - Desbloquea el `mutex`.

Lectores-escritores con mutex

```
int dato = 5;                       /*recurso*/
int n_lect = 0;                     /*numero de lectores*/
pthread_mutex_t mutex;              /*controlar el acceso a dato*/
pthread_mutex_t m_lect;             /*controla la variable n_lect*/

main(int argc, char *argv[])  {
    pthread_t th1, th2, th3, th4;

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&m_lect, NULL);

    pthread_create(&th1, NULL, Lector, NULL);
    pthread_create(&th2, NULL, Escritor, NULL);
    pthread_create(&th3, NULL, Lector, NULL);
    pthread_create(&th4, NULL, Escritor, NULL);

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

    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&m_lect);

    exit(0);
}
```

Implementación incorrecta
¿ POR QUÉ?

Lectores-escritores con mutex (II)

```
/*codigo del lector */
void Lector(void) {
   while(1){
     pthread_mutex_lock(&m_lect);
     n_lect ++;
     if (n_lect == 1)
       pthread_mutex_lock(&mutex);
     pthread_mutex_unlock(&m_lect);

     /*leer*/
     printf("%d\n", dato);

     pthread_mutex_lock(&m_lect);
     n_lectores--;
     if (n_lectores == 0)
       pthread_mutex_unlock(&mutex);
     pthread_mutex_unlock(&m_lect);
   }
}
```

```
/*codigo del escritor */
void Escritor(void){
  while(1){
    pthread_mutex_lock(&mutex);

    /*modificar el recurso */
    dato = dato + 2;

    pthread_mutex_unlock(&mutex);

  }
}
```

Implementación incorrecta
¿ POR QUÉ?

Variables condicionales

- Variables de sincronización **asociadas** a un cerrojo
- Se usan entre `lock` y `unlock`
- Podemos pensar en una variable condicional como un objeto con un atributo (y un cerrojo asociado) y 3 métodos principales.

```
typedef struct var_cond {
  // Cola de hilos bloqueados
  queue_t vc_q;
} vc_t;
```

Hay una cola de espera ***adicional*** a la del cerrojo asociado

Semántica de funciones asociadas

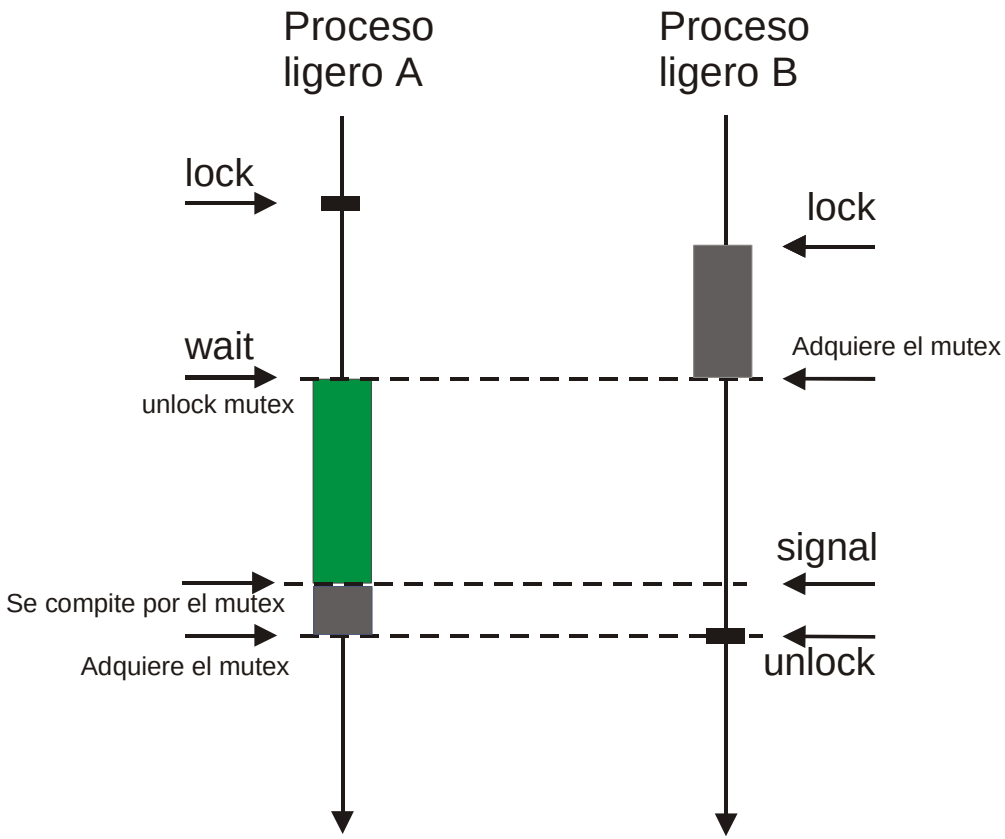
```
// El hilo que llama a esta función
// DEBE ser el propietario del cerrojo c
void cond_wait(lock_t c, vc_t varC ) {
    queue_add(varC->vc_q, esteHilo);
    unlock(c);
    park(); // suspender el hilo
    lock();
}
```

- **Siempre** que se llama a *cond_wait* el hilo se bloquea
- Antes de bloquearse libera el cerrojo para que otro hilo lo pueda adquirir
- Tras despertar del bloque, vuelve a **solicitar** el cerrojo
    - Puede implicar un nuevo bloqueo
    - Cuando hilo sale de *cond_wait*, **sigue en posesión del cerrojo**

```
// Despierta un hilo de la cola de espera
// dela Var. Cond
void cond_signal (vc_t varC ) {
  if (! isEmpty(varC->vc_q)
    unpark(queue_remove(varC->vc_q))
}

// Despierta a todos los hilos de la cola
// de espera
void cond_broadcast (vc_t varC ) {

  while (! isEmpty(varC->vc_q)
      unpark(queue_remove(varC->vc_q)
}
```

- Es **muy aconsejable** que el hilo que llama a estas funciones tenga el cerrojo asociado

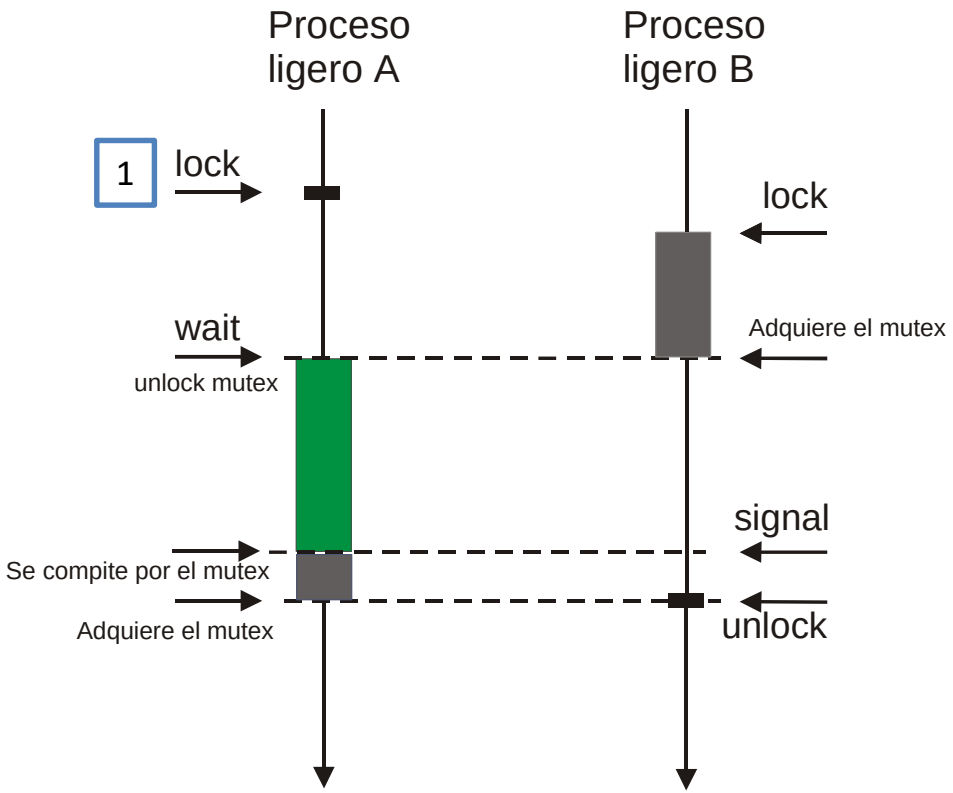
Variables condicionales (II)

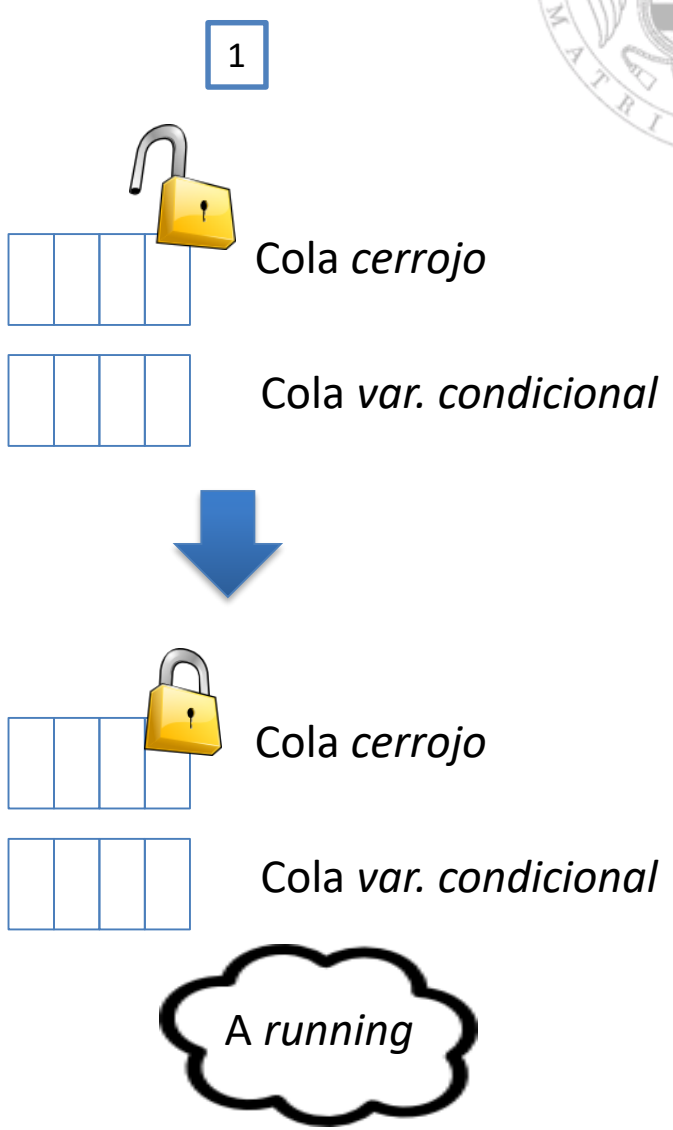


Proceso ligero bloqueado esperando unlock

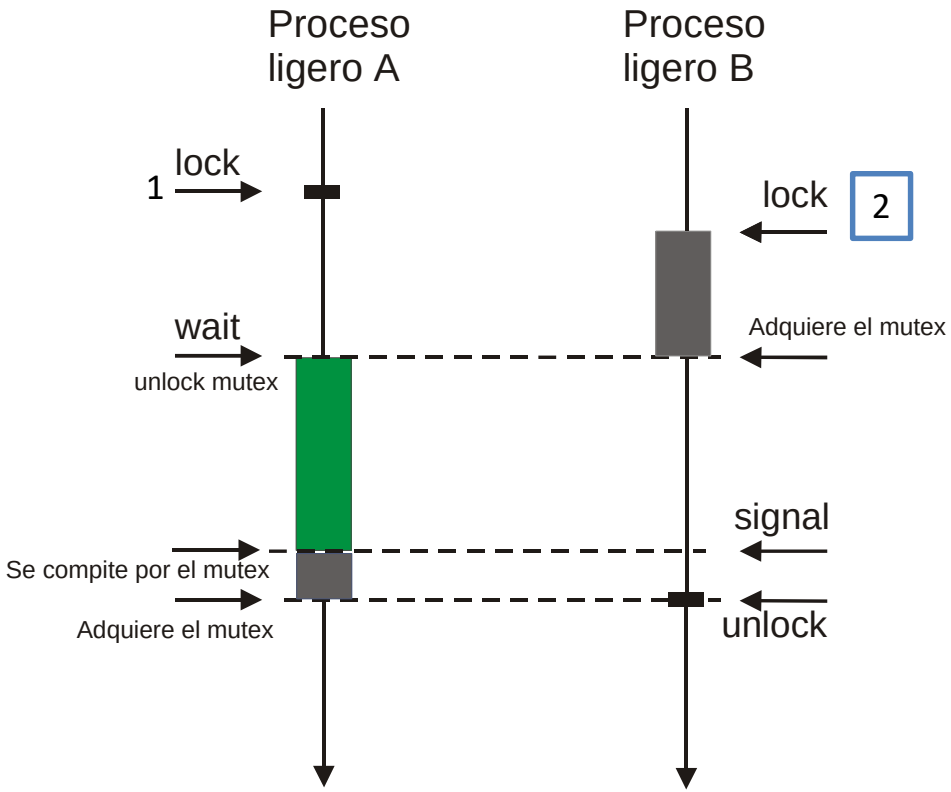
Proceso ligero bloqueado esperando signal

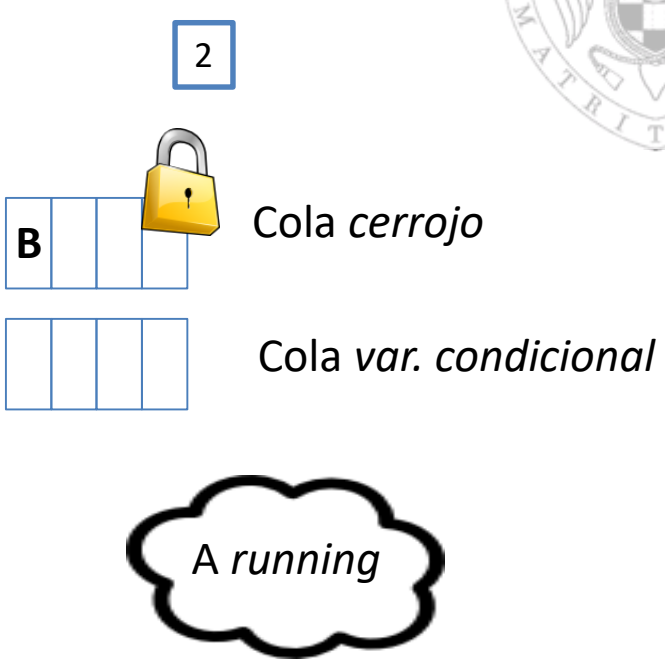
Uso de var. condicional



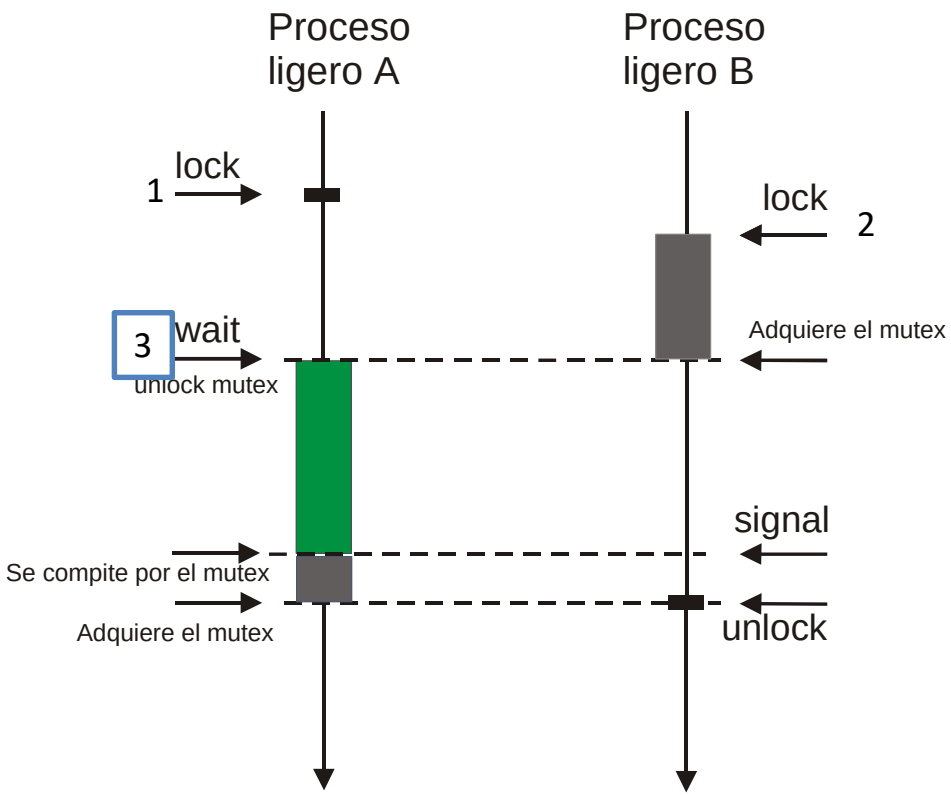


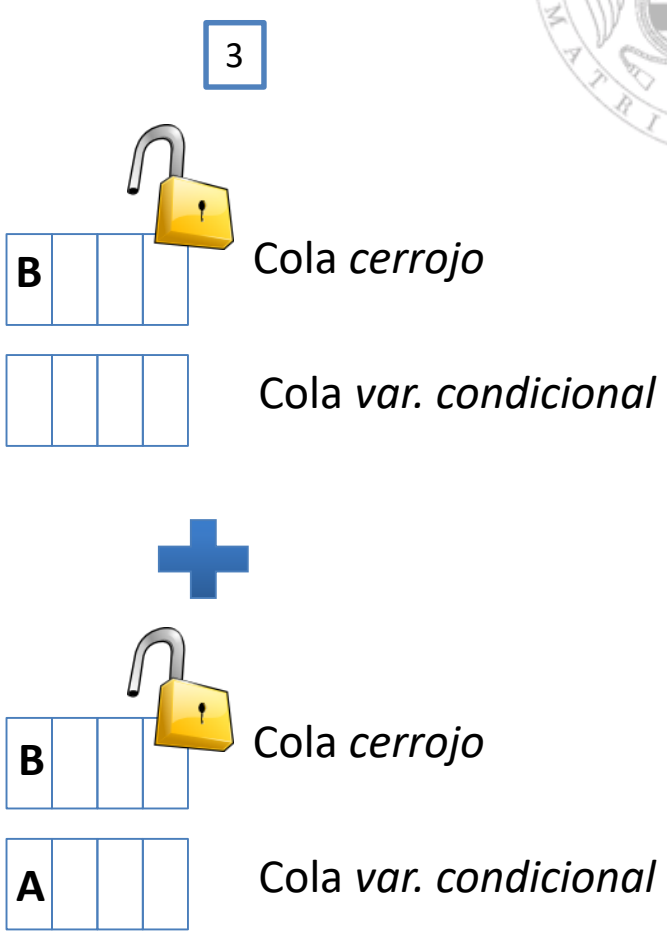
Uso de var. condicional



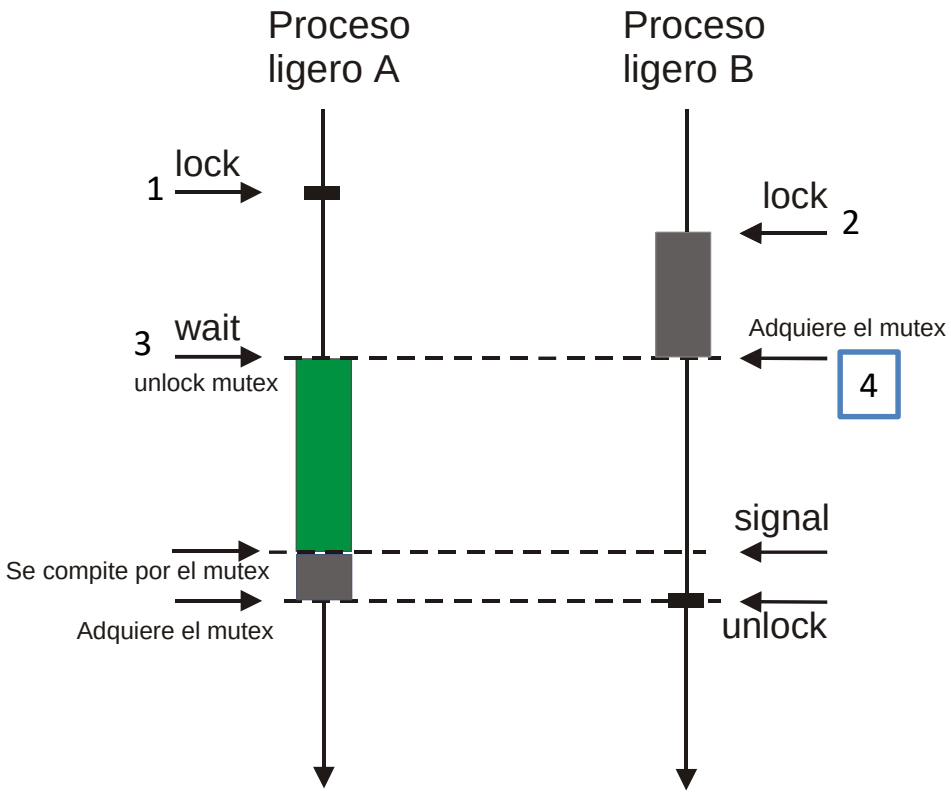


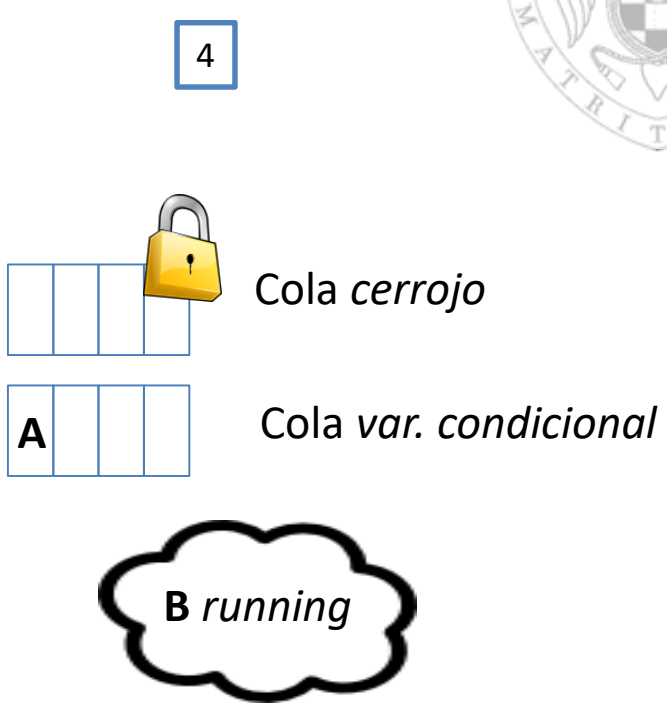
Uso de var. condicional



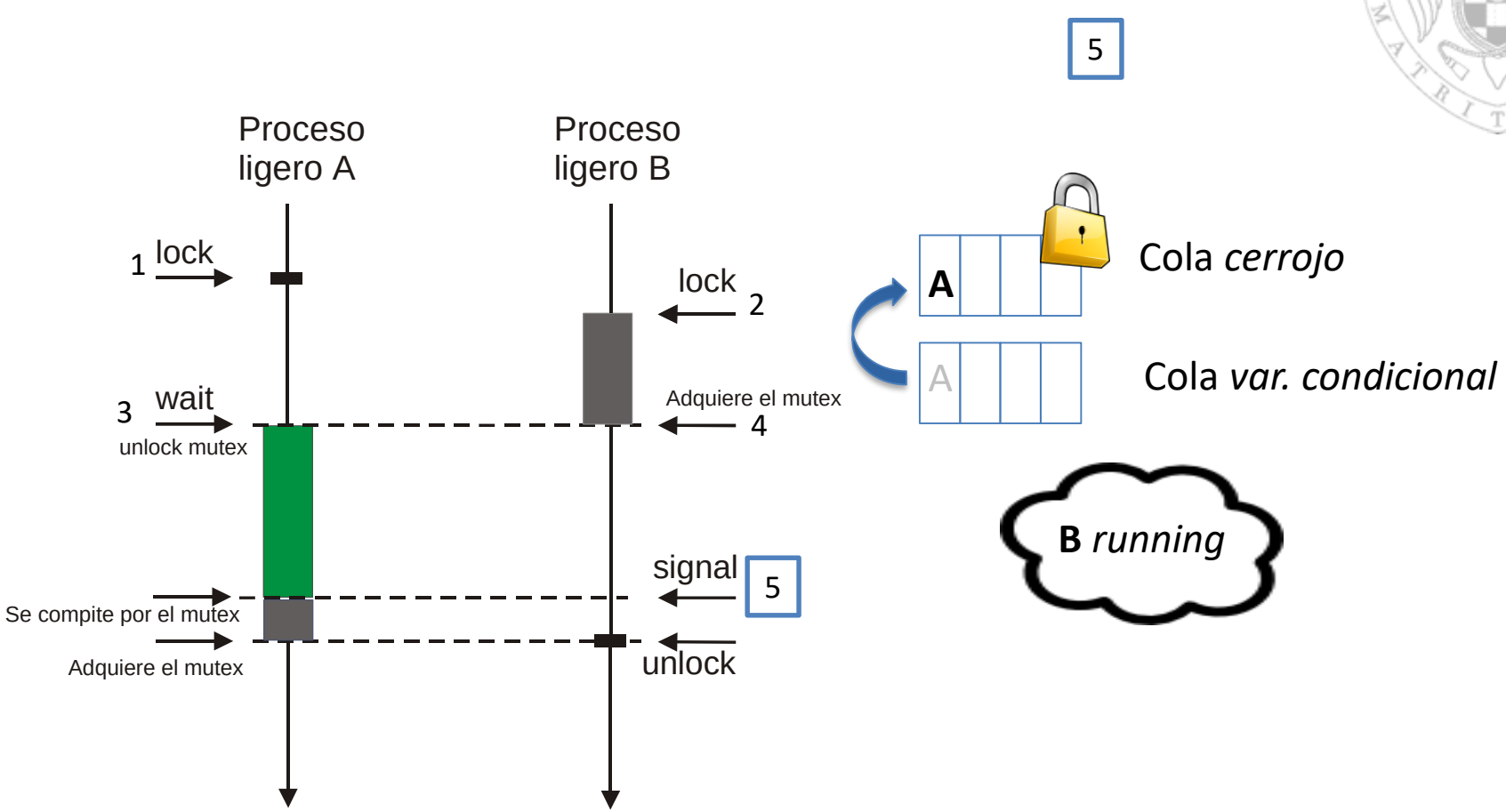


Uso de var. condicional

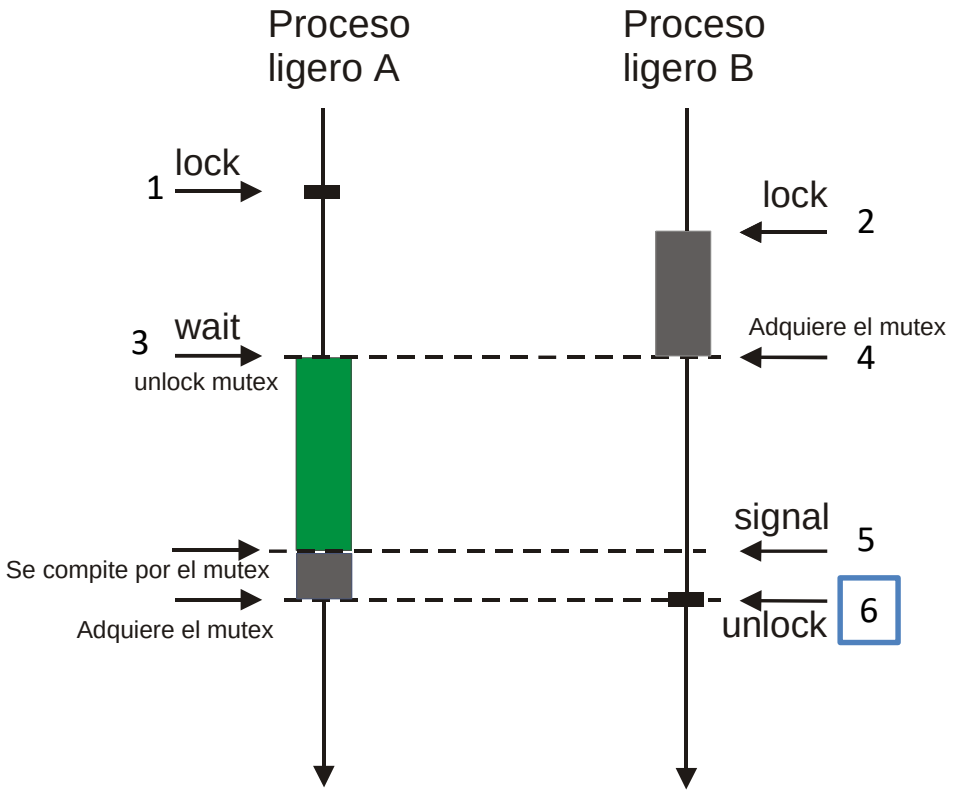


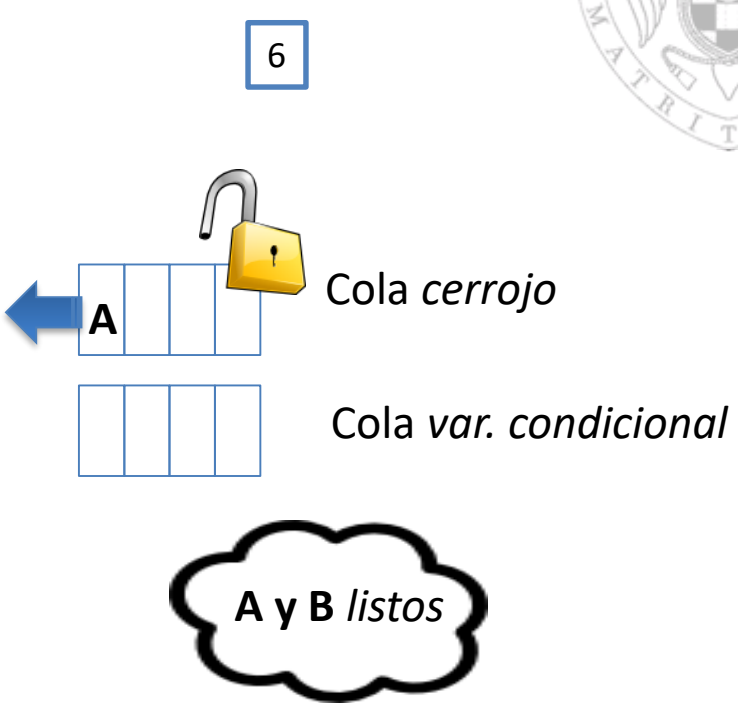


Uso de var. condicional



Uso de var. condicional





Uso de cerrojos / var. condicionales

- Proceso ligero A

```
lock(mutex);  /* acceso al recurso */
while (condición relacionada con el recurso == false)
     wait(condition, mutex);  /*bloqueo*/
<acciones deseadas que cumplen la condición>
unlock(mutex);
```

- Proceso ligero B

```
lock(mutex);  /* acceso al recurso */
<operaciones protegidas>
/*hemos podido afectar a otros procesos, desbloqueamos*/
signal(condition);
<más operaciones protegidas>
unlock(mutex);
```

- Importante utilizar `while`

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);`
  - Se reactivan uno o más de los procesos ligeros que están suspendidos en la variable condicional `cond`.
  - No tiene efecto si no hay ningún proceso ligero esperando (diferente a los semáforos).
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - Todos los threads suspendidos en la variable condicional `cond` se reactivan.
  - No tiene efecto si no hay ningún proceso ligero esperando.
- `int pthread_cond_wait(pthread_cond_t*cond, pthread_mutex_t*mutex);`
  - Suspende al proceso ligero hasta que otro proceso señaliza la variable condicional `cond`.
  - Automáticamente se libera el `mutex`. Cuando se despierta el proceso ligero vuelve a competir por el `mutex` y sólo continua con su ejecución cuando lo obtiene

Productor-consumidor con var. Cond.

```
#define MAX_BUFFER          1024      /* tamanio del buffer */
#define DATOS_A_PRODUCIR    100000    /* datos a producir */
pthread_mutex_t mutex;      /*mutex para buffer compartido*/
pthread_cond_t lleno;       /*controla el llenado del buffer*/
pthread_cond_t vacio;       /*controla el vaciado del buffer*/
int n_elementos;            /*numero de elementos en el buffer*/
int buffer[MAX_BUFFER];     /*buffer comun*/
main(int argc, char *argv[]){
    pthread_t th1, th2;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&lleno, NULL);
    pthread_cond_init(&vacio, NULL);
    pthread_create(&th1, NULL, Productor, NULL);
    pthread_create(&th2, NULL, Consumidor, NULL);
    pthread_join(th1, NULL);    pthread_join(th2, NULL);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&lleno);
    pthread_cond_destroy(&vacio);
    exit(0);
}
```

Productor-consumidor con var. Cond.

```
void Productor(void)  {   /* codigo del productor */
    int dato, i ,pos = 0;

    for(i=0; i < DATOS_A_PRODUCIR; i++ )  {
        dato = producir_dato();            /*producir dato*/
        pthread_mutex_lock(&mutex);        /*acceder al buffer*/
        while (n_elementos == MAX_BUFFER) /*si buffer lleno*/
            pthread_cond_wait(&lleno, &mutex); /*se bloquea*/
        buffer[pos] = dato;
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos ++;
        pthread_cond_signal(&vacio);   /*buffer no vacio*/
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

Productor-consumidor con var cond.

```
void Consumidor(void) {   /* codigo del sonsumidor */
    int dato, i ,pos = 0;

    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        pthread_mutex_lock(&mutex);     /* acceder al buffer */
        while (n_elementos == 0)        /* si buffer vacio */
            pthread_cond_wait(&vacio, &mutex); /* se bloquea */
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos --;
        pthread_cond_signal(&lleno);    /* buffer no lleno */
        pthread_mutex_unlock(&mutex);
        printf("Consume %d \n", dato);      /* consume dato */
    }
    pthread_exit(0);
}
```

Simplificación

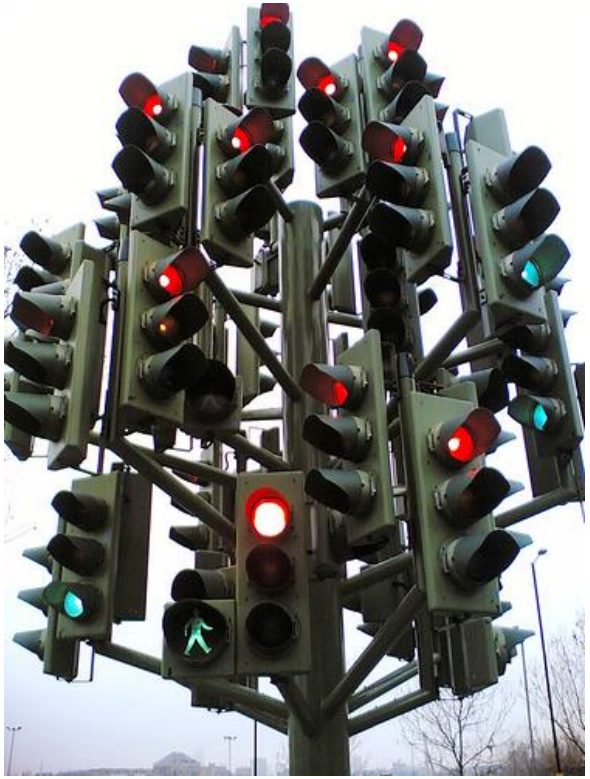
```
buffer[MAX_BUFFER];
indProd = 0, indCons = 0;
int n_elementos = 0;

mutex_t mutex;
cond_t evento;
```

```
void Productor () {
...
  mutex_lock(&mutex)
  while( n_elementos == MAX_BUFFER)
     cond_wait(&evento, &mutex);

  <inserta un elemento de la cola>

  cond_broadcast(&evento);
  mutex_unlock(&mutex);
...
}
```

```
void Consumidor () {
...
  mutex_lock(&mutex)
  while( !n_elementos )
     cond_wait(&evento, &mutex);

  <extrae un elemento de la cola>

  cond_broadcast(&evento);
  mutex_unlock(&mutex);
...
}
```

Semáforos (Dijkstra'65)

- Mecanismo de sincronización
- Misma máquina
- Objeto con un valor entero
- Dos operaciones **atómicas**
  - `wait`
  - `signal`



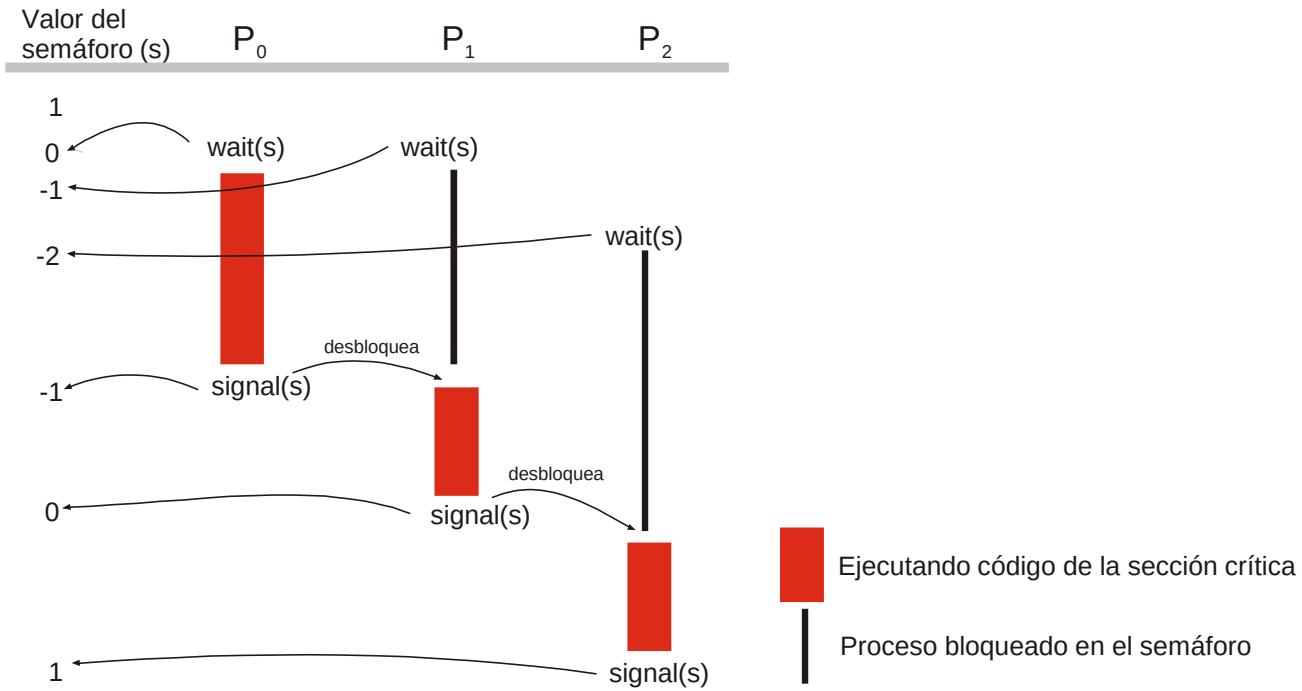
Operaciones sobre semáforos (semántica)

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

signal(s){
   s = s + 1;
   if (s <= 0){
      <Desbloquear a un proceso bloq. por wait>
   }
}
```

Secciones críticas con semáforos

```
wait(s); /* entrada en la seccion critica */
 Sección_crítica();
signal(s);  /* salida de la seccion critica */
```

- Ejemplo con valor inicial 1



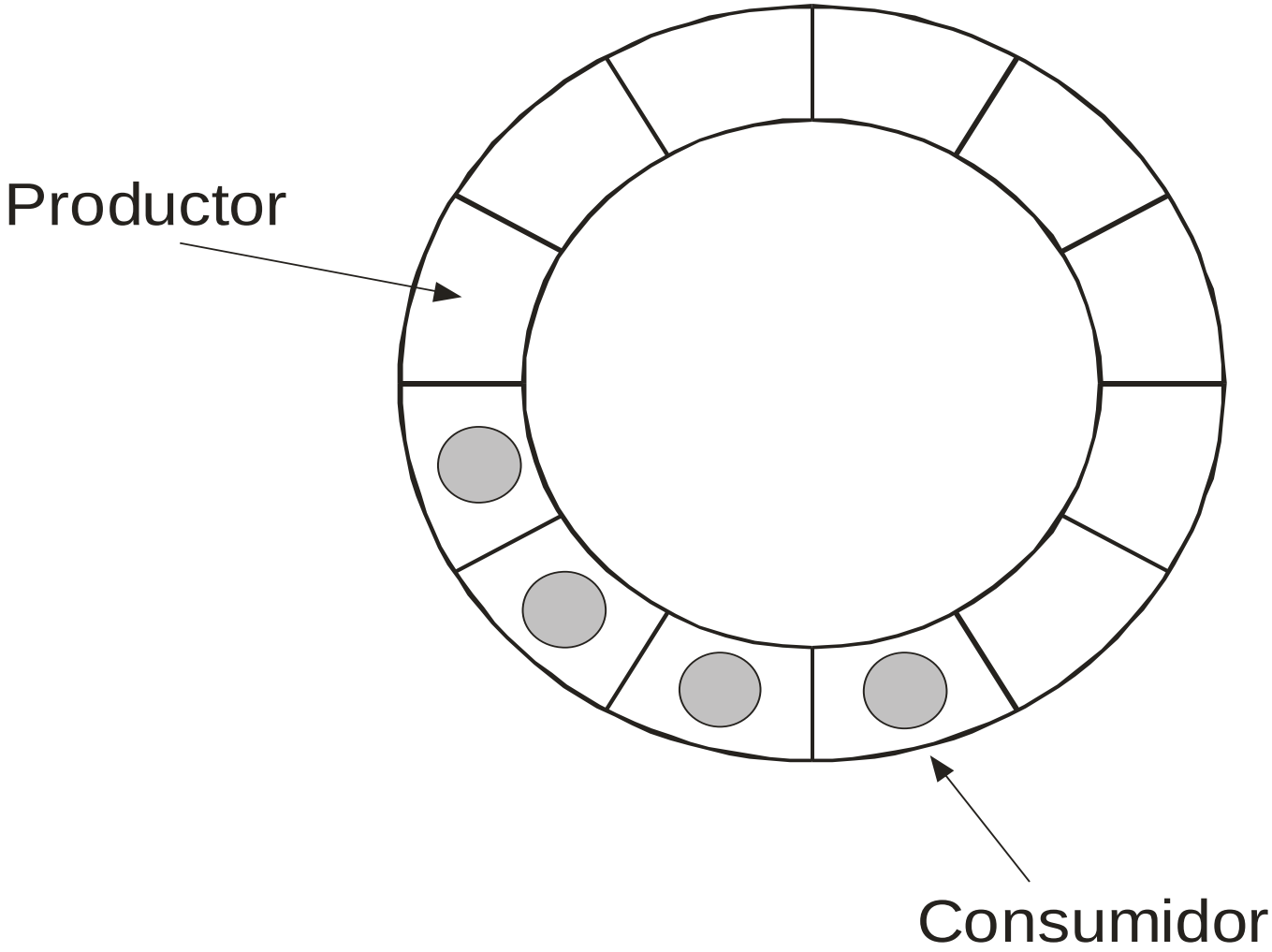
Semáforos POSIX

- `int sem_init(sem_t *sem, int shared, int val);`
  - Inicializa un semáforo sin nombre
  - `shared:` pensado para ser mapeado en memoria compartida
- `int sem_destroy(sem_t *sem);`
  - Destruye un semáforo sin nombre

- `sem_t*sem_open(char*name,int flag,mode_t mode,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

Productor-consumidor con semáforos (buffer acotado y circular)



Productor-consumidor con semáforos (II)

```
#define MAX_BUF              1024    /*tamanio del buffer */
#define PROD               100000    /*datos a producir */
sem_t elementos;                     /*elementos en el buffer */
sem_t huecos;                        /* huecos en el buffer */
int buffer[MAX_BUF];                 /* buffer comun */
Int cons,prod = 0;                    /*posicion dentro del buffer*/

void main(void){
   pthread_t th1, th2;  /* identificadores de threads */
   /* inicializar los semaforos */
   sem_init(&elementos, 0, 0);   sem_init(&huecos, 0, MAX_BUFFER);

   /*crear los procesos ligeros */
   pthread_create(&th1, NULL, Productor, NULL);
   pthread_create(&th2, NULL, Consumidor, NULL);
   /*esperar su finalizacion */
   pthread_join(th1, NULL);   pthread_join(th2, NULL);

   sem_destroy(&huecos);   sem_destroy(&elementos);
   exit(0);
}
```

Productor-consumidor con semáforos (II)

```
void Productor(void){
/*dato a producir*/
   int dato;
   int i;

   for(i=0; i < PROD; i++ ){
      /*producir dato*/
      dato = producir_dato();
      /*un hueco menos*/
      sem_wait(&huecos);
      buffer[prod] = dato;
      prod = (prod + 1) % MAX_BUF;
      /*un elemento mas*/
      sem_post(&elementos);
  }
  pthread_exit(0);
}
```

```
void Consumidor(void){
/*dato a producir*/
   int dato;
   int i;

   for(i=0; i<PROD; i++ ){
      /*un elemento menos*/
      sem_wait(&elementos);
      dato = buffer[con];
      cons= (cons+ 1) % MAX_BUF;
      /*un hueco mas*/
      sem_post(&huecos);
      cosumir_dato(dato);
   }
   pthread_exit(0);
}
```

CUIDADO: problema de sección crítica SIN RESOLVER (si hay muchos productores y/o consumidores)

Lectores-escritores con semáforos

```
int dato = 5;          /* recurso */
int n_lectores = 0;    /* numero de lectores */
sem_t sem_lec;         /* controlar el acceso n_lectores */
sem_t cerrojo;         /* controlar el acceso a dato */
void main(void){
   pthread_t th1, th2, th3, th4;
   sem_init(&cerrojo, 0, 1);   sem_init(&sem_lec, 0, 1);

   pthread_create(&th1, NULL, Lector, NULL);
   pthread_create(&th2, NULL, Escritor, NULL);
   pthread_create(&th3, NULL, Lector, NULL);
   pthread_create(&th4, NULL, Escritor, NULL);

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

   /* cerrar todos los semaforos */
   sem_destroy(&cerrojo);     sem_destroy(&sem_lec);
   exit(0);
}
```

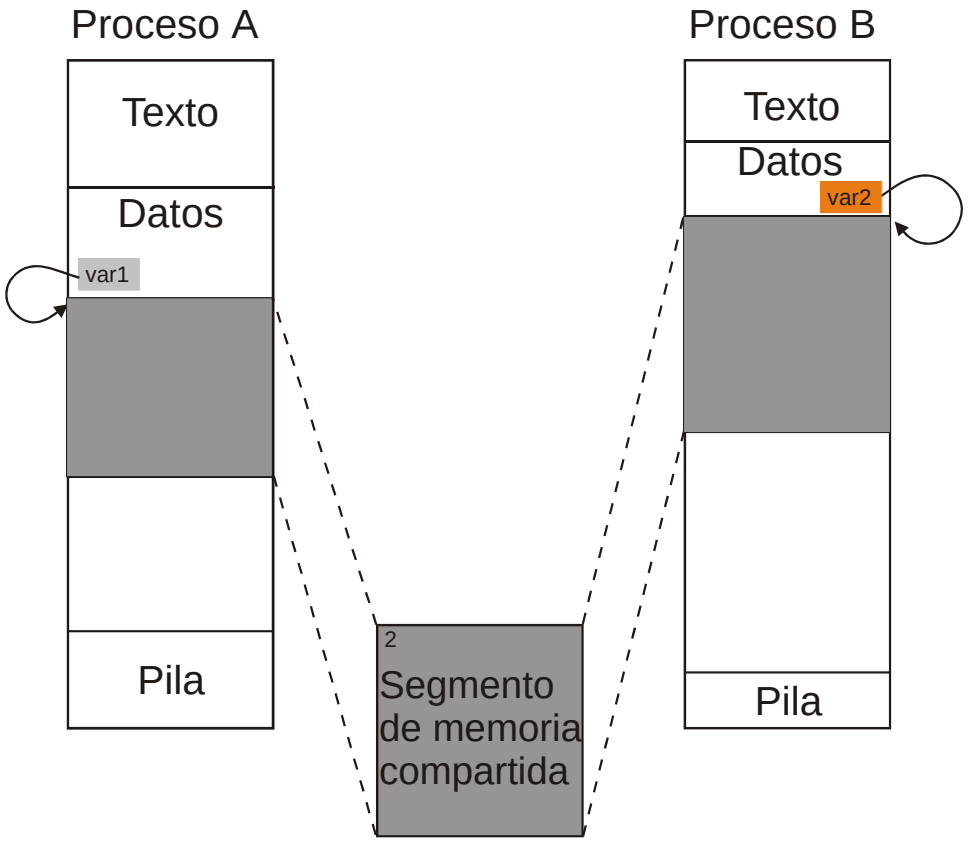
Lectores-escritores con semáforos (II)

```
void Lector(void) {
  While(1){
    sem_wait(&sem_lec);
    n_lectores = n_lectores + 1;
    if (n_lectores == 1)
      sem_wait(&cerrojo);
    sem_post(&sem_lec);

    /* leer dato */
    printf("%d\n", dato);

    sem_wait(&sem_lec);
    n_lectores = n_lectores - 1;
    if (n_lectores == 0)
      sem_post(&cerrojo);
    sem_post(&sem_lec);
  }
}
```

```
void Escritor(void) {
  while(1){
    sem_wait(&cerrojo);

    /* modificar el recurso */
    dato = dato + 2;

    sem_post(&cerrojo);

  }
}
```

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`)
  - Esta región de memoria puede ser compartida o privada:
    - `flags: MAP_SHARED ó MAP_PRIVATE`
  - También se puede declarar sin respaldo en disco:
    - `flags: MAP_ANONYMOUS` (compartir padre-hijo)
    - Empleando `shm_open` para obtener un descriptor
- `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.
- `int msync(void *addr, size_t len, int flags);`
  - Escribe cualquier dato (página) modificada en memoria en su correspondiente fichero de respaldo

Productor-consumidor 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    /* tamaño del buffer */
#define DATOS_A_PRODUCIR  100000    /* datos a producir */
sem_t *elementos;  /* elementos en el buffer */
sem_t *huecos;     /* huecos en el buffer */
void main(int argc, char *argv[]){
   int shd;
   int *buffer;    /* buffer comun */

    /* el productor crea el archivo a proyectar */
   shd = open("BUFFER", O_CREAT|O_WRONLY, 0700);
   ftruncate(shd, MAX_BUFFER * sizeof(int));

   /*proyectar el objeto de memoria compartida en el espacio
   de direcciones del productor*/
   buffer = (int*) mmap(NULL, MAX_BUFFER * sizeof(int),
                        PROT_WRITE, MAP_SHARED, shd, 0);
```

Código del productor (II)

```
    /* El productor crea los semaforos */
    elementos = sem_open("ELEMENTOS", O_CREAT, 0700, 0);
    huecos =    sem_open("HUECOS", O_CREAT, 0700, MAX_BUFFER);

    /*código de producción*/
    Productor(buffer);

    /* desproyectar el buffer compartido */
    munmap(buffer, MAX_BUFFER * sizeof(int));
    close(shd);          /*cerrar el objeto de memoria compartida*/
    unlink("BUFFER");  /* borrar el objeto de memoria */

    sem_close(elementos);
    sem_close(huecos);
    sem_unlink("ELEMENTOS");
    sem_unlink("HUECOS");
}
```

Código del consumidor

```
#define MAX_BUFFER          1024    /* tamanio del buffer */
#define DATOS_A_PRODUCIR  100000    /* datos a producir */

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

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

   /* el consumidor abre el archivo a proyectar */
   shd = open("BUFFER", O_RDONLY);

   /*proyectar el objeto de memoria compartida en el espacio de
   direcciones del productor*/
   buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int),
                         PROT_READ, MAP_SHARED, shd, 0);
```

Código del consumidor (II)

```
   /*El consumidor abre los semaforos*/
   elementos = sem_open("ELEMENTOS", 0);
   huecos    = sem_open("HUECOS", 0);

   /*proceso consumidor con buffer proyectado
   Consumidor(buffer);

   /*desproyectar el buffer compartido*/
   munmap(buffer,  MAX_BUFFER * sizeof(int));
   close(shd);   /* cerrar el objeto de memoria compartida */

   /*cerrar los semaforos*/
   sem_close(elementos);
   sem_close(huecos);
}
```

Función del productor

```
void Productor(int *buffer)   /* codigo del productor */
{
   int pos = 0;  /* posicion dentro del buffer */
   int dato;     /* dato a producir */
   int i;

   for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
      dato = producir_dato();  /* producir dato */
      sem_wait(huecos);        /* un hueco menos */
      buffer[pos] = dato;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(elementos);     /* un elemento mas */
   }
   return;
}
```

Función del consumidor

```
void Consumidor(char *buffer)  /* codigo del Consumidor */
{
   int pos = 0;
   int i, dato;

   for(i=0; i < DATOS_A_PRODUCIR; i++ )  {
      sem_wait(elementos);     /* un elemento menos */
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(huecos);        /* un hueco mas */
      printf("Consume %d \n", dato); /* cosumir dato */
   }
   return;
}
```

Resumen

- Hilos:
  - Memoria compartida (variables globales)
  - Mutex y variables condicionales
- Procesos emparentados (fork):
  - memoria compartida (mapeada)
  - Semáforos con o sin nombre
- Procesos no emparentados en la misma máquina:
  - Memoria compartida (regiones con nombre)
  - Semáforos con nombre