SISTEMAS OPERATIVOS:

Lección 7:
Hilos y mecanismos de comunicación y sincronización

Jesús Carretero Pérez
Alejandro Calderón Mateos
José Daniel García Sánchez
Francisco Javier García Blas
José Manuel Pérez Lobato
María Gregoria Casares Andrés

ADVERTENCIA

- Este material es un simple guión de la clase: no son los apuntes de la asignatura.

- El conocimiento exclusivo de este material no garantiza que el alumno pueda alcanzar los objetivos de la asignatura.

- Se recomienda que el alumno utilice los materiales complementarios propuestos.

SISTEMAS OPERATIVOS:

Lección 7:
Hilos y mecanismos de comunicación y sincronización

Contenido

- Comunicación y sincronización.
- Semáforos.
- El problema de los lectores escritores.
  - Solución con semáforos.
- Mutex y variables condición.

Mecanismos de comunicación

- Los mecanismos de comunicación permiten la transferencia de información entre dos procesos.

- Archivos
- Tuberías (pipes, FIFOS)
- **Variables en memoria compartida**
- Paso de mensajes

Mecanismos de sincronización

- Los mecanismos de sincronización permiten forzar a un proceso a detener su ejecución hasta que ocurra un evento en otro proceso.

- Construcciones de los lenguajes concurrentes (procesos ligeros)
- Servicios del sistema operativo:
  - Señales (asincronismo)
  - Tuberías (pipes, FIFOS)
  - **Semáforos**
  - **Mutex y variables condicionales**
  - Paso de mensajes
- Las operaciones de sincronización deben ser **atómicas**

Semáforos

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

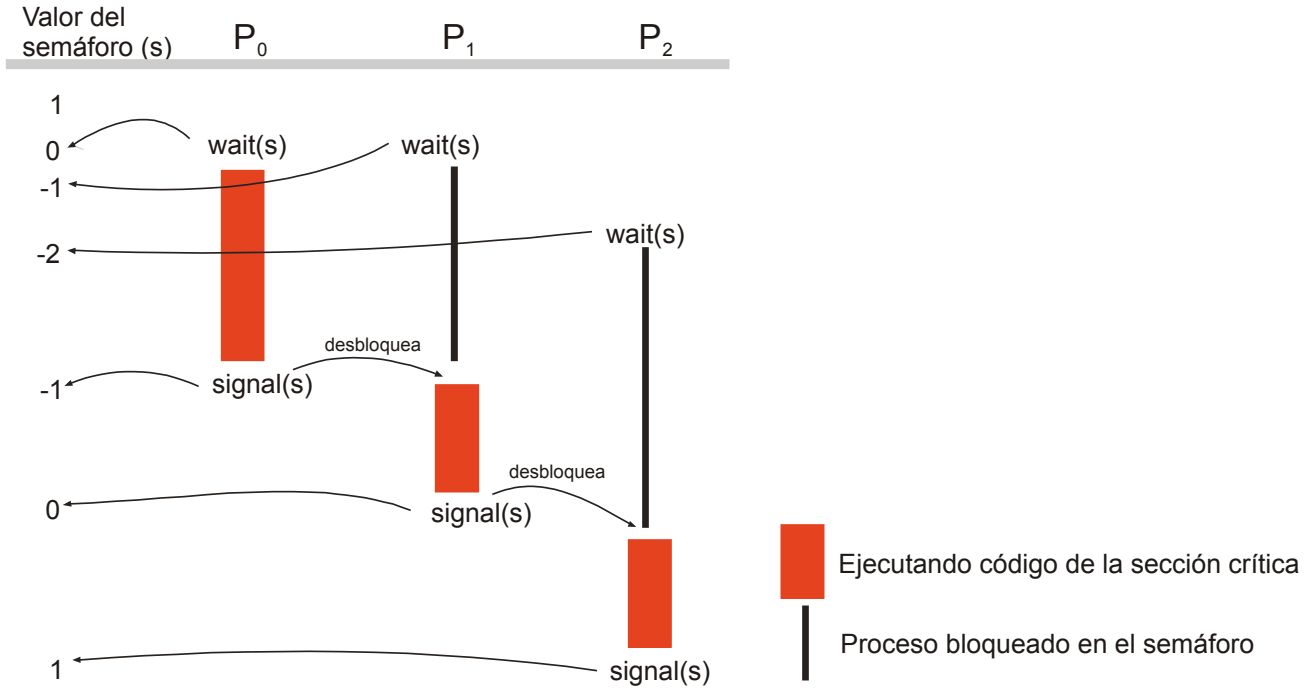
Operaciones sobre semáforos

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

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

Secciones críticas con semáforos

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

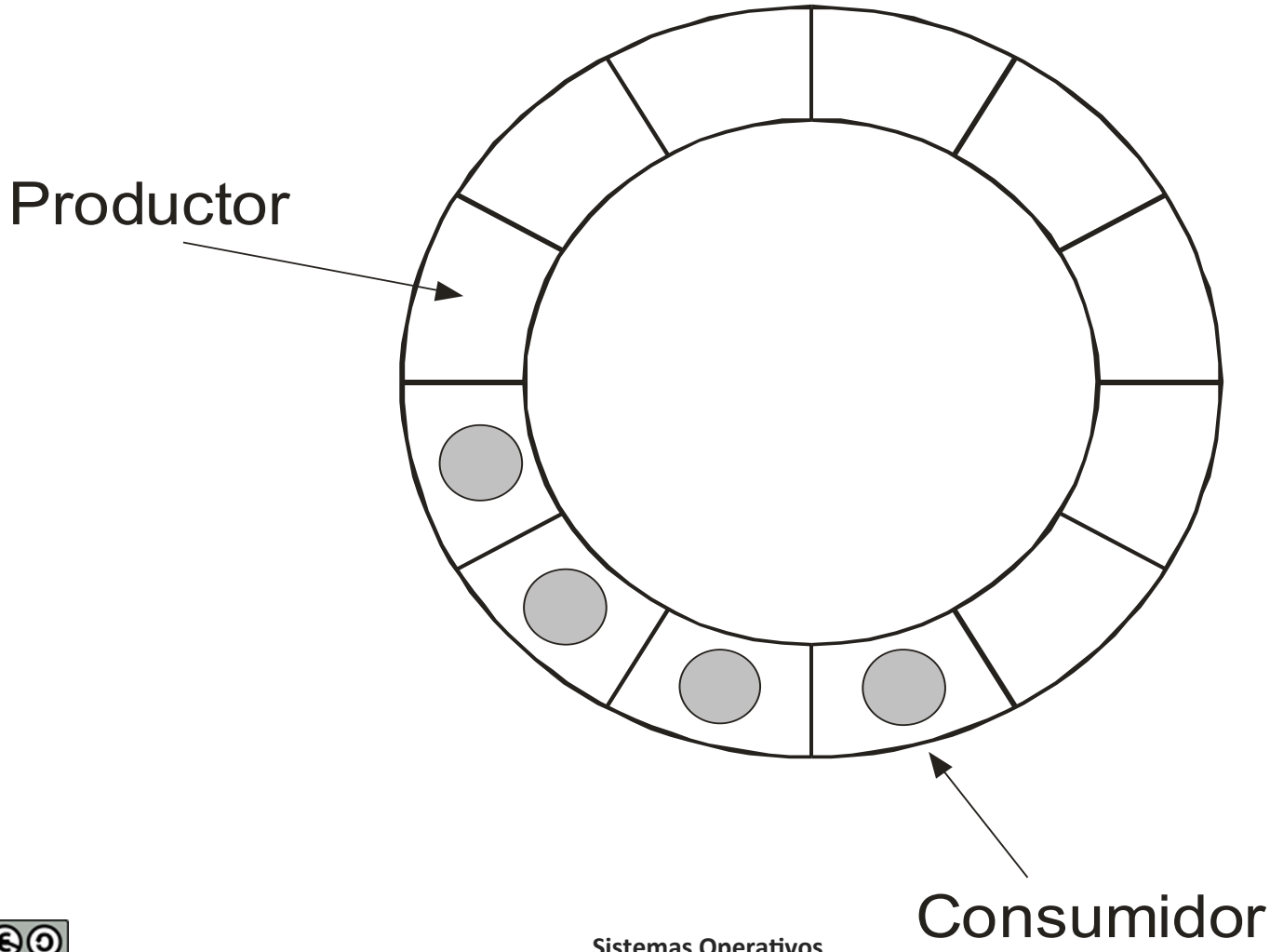
- El semáforo debe tener valor inicial 1



Semáforos POSIX

- `int sem_init(sem_t *sem, int shared, int val);`
  - Inicializa un semáforo sin nombre
- `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 (búfer acotado y circular)



Productor-consumidor con semáforos

```
#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 */
int buffer[MAX_BUFFER];             /* buffer comun */

void main(void)
{
   pthread_t th1, th2;  /* identificadores de threads */

   /* inicializar los semaforos */
   sem_init(&elementos, 0, 0);
   sem_init(&huecos, 0, MAX_BUFFER);
```

Productor-consumidor con semáforos

```
 /* 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: Hilo productor

```
void Productor(void)   /* 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 = i;            /* producir dato */
      sem_wait(&huecos);  /* un hueco menos */
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);  /* un elemento mas */
  }
  pthread_exit(0);
}
```

Productor-consumidor: Hilo consumidor

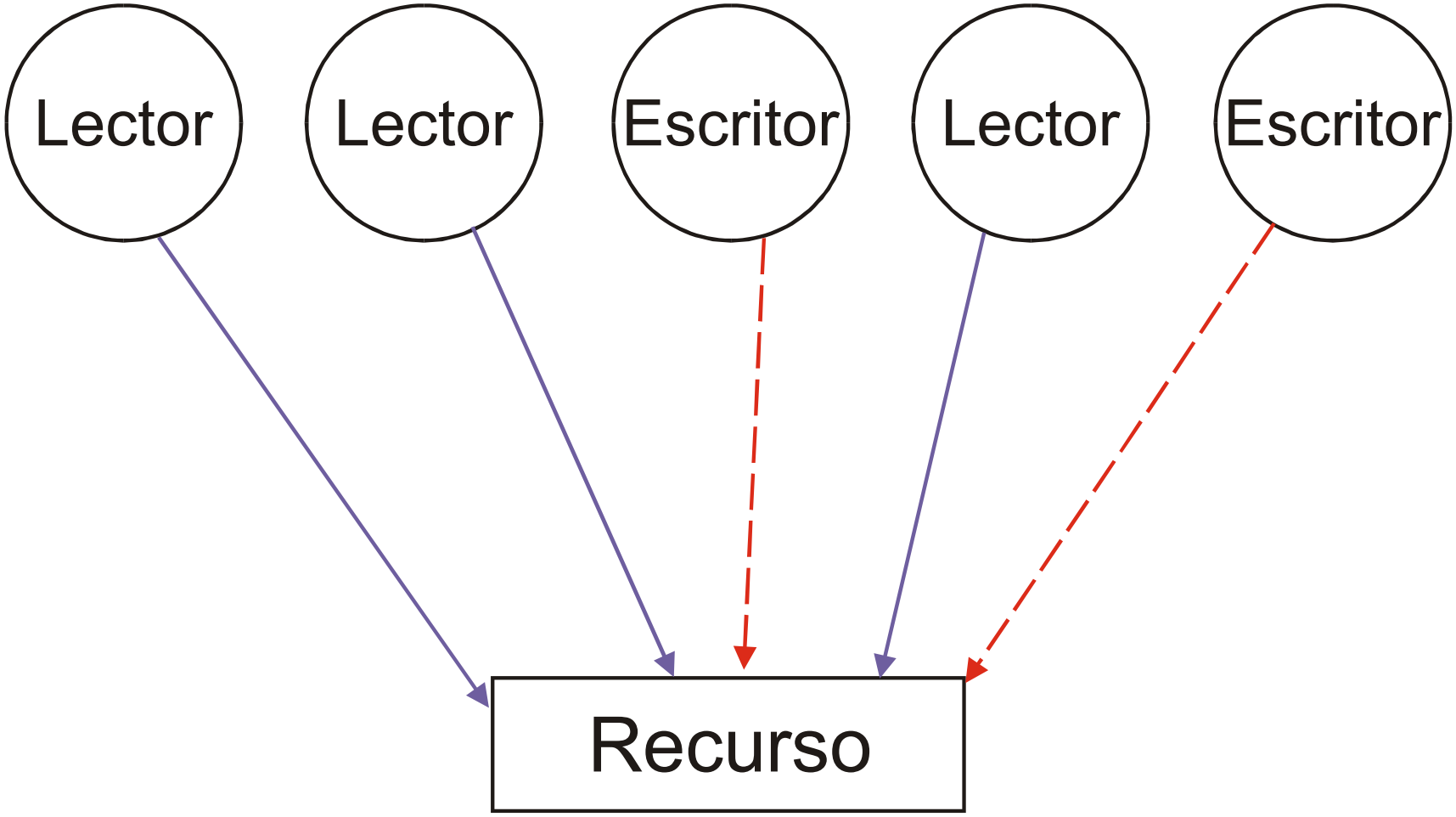
```
void Consumidor(void)   /* codigo del Consumidor */
{
    int pos = 0;
    int dato;
    int i;

    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 */
        /* cosumir dato */
    }
    pthread_exit(0);
}
```

Problema de los lectores-escritores

- Problema que se plantea cuando se tiene un área de almacenamiento compartida.
  - Múltiples procesos leen información.
  - Múltiples procesos escriben información.

- Condiciones:
  - Cualquier número de lectores pueden leer de la zona de datos concurrentemente.
  - Solamente un escritor puede modificar la información a la vez.
  - Durante una escritura ningún lector puede realizar una consulta.

El problema de los lectores-escritores



Diferencias con otros problemas

- Exclusión mutua:
  - En el caso de la exclusión mutua solamente se permitiría a un proceso acceder a la información.
  - No se permitiría concurrencia entre lectores.
- Productor consumidor:
  - En el productor/consumidor los dos procesos modifican la zona de datos compartida.

- Objetivos de restricciones adicionales:
  - Proporcionar una solución más eficiente.

Alternativas de gestión

- Los lectores tienen prioridad.
  - Si hay algún lector en la sección crítica otros lectores pueden entrar.
  - Un escritor solamente puede entrar en la sección crítica si no hay ningún proceso.
  - Problema: Inanición para escritores.
- Los escritores tienen prioridad.
  - Cuando un escritor desea acceder a la sección crítica no se admite la entrada de nuevos lectores.

Los lectores tienen prioridad

- Lector

`int nlect; semaforo lec=1; semaforo = escr=1;`

```
for(;;) {
  semWait(lec);
  nlect++;
  if (nlect==1)
    semWait(escr);
  semSignal(lec);

  realizar_lect();

  semWait(lec);
  nlect--;
  if (nlect==0)
    semSignal(escr);
  semSignal(lec);
```

- Escritor

```
for(;;) {
  semWait(escr);
  realizar_escr();
  semSignal(escr);
}
```

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 mutex;          /* controlar el acceso a dato */

void main(void) {
   pthread_t th1, th2, th3, th4;

   sem_init(&mutex, 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);
```

Lectores-escritores con semáforos

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

  /* cerrar todos los semaforos */
  sem_destroy(&mutex);
  sem_destroy(&sem_lec);

  exit(0);
}
```

Lectores-escritores: Hilo lector

```
void Lector(void) { /* codigo del lector */
  sem_wait(&sem_lec);
  n_lectores = n_lectores + 1;
  if (n_lectores == 1) sem_wait(&mutex);
  sem_post(&sem_lec);

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

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

Lectores-escritores: Hilo escritor

```
void Escritor(void) {    /* codigo del escritor */
   sem_wait(&mutex);
    dato = dato + 2;     /* modificar el recurso */
    sem_post(&mutex);

    pthread_exit(0);
}
```

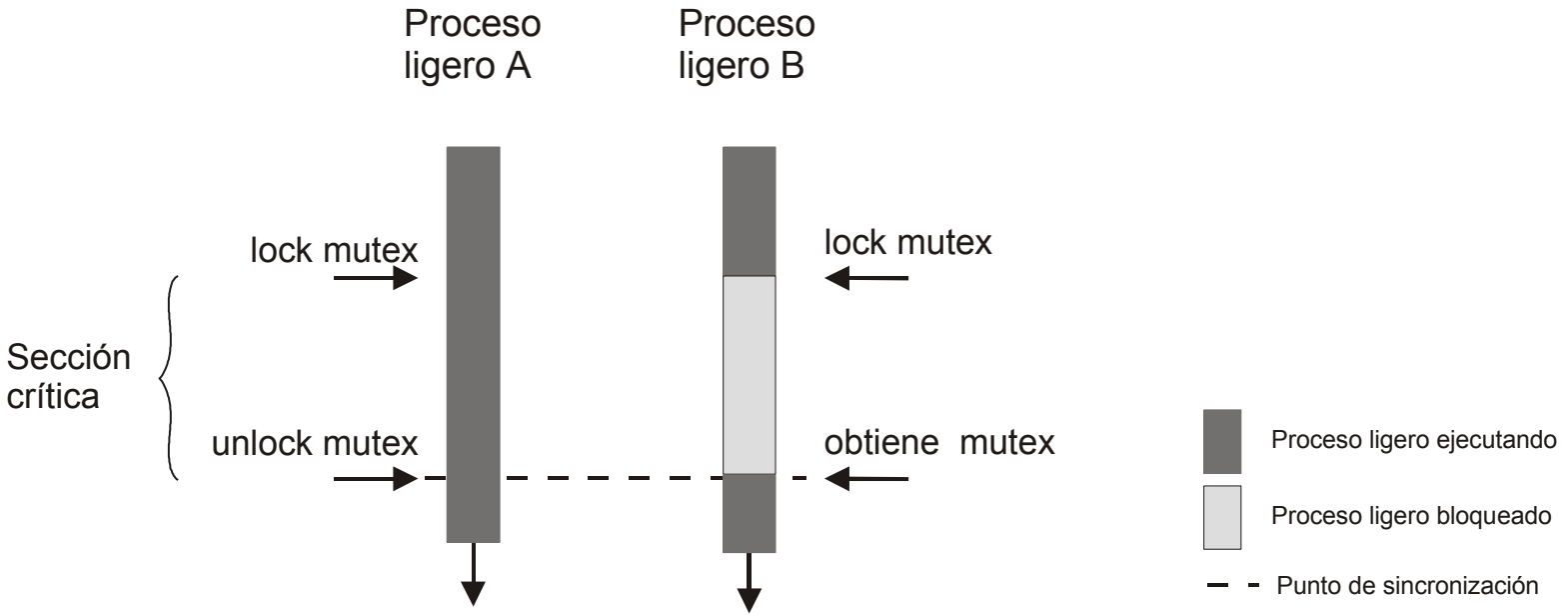
Mutex y variables condicionales

- Un mutex es un mecanismo de sincronización indicado para procesos ligeros.
- Es un semáforo binario con dos operaciones atómicas:
  - **lock(m)** Intenta bloquear el mutex, si el mutex ya está bloqueado el proceso se suspende.
  - **unlock(m)** Desbloquea el mutex, si existen procesos bloqueados en el mutex se desbloquea a uno.

Secciones críticas con mutex

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

- La operación `unlock` debe realizarla el proceso ligero que ejecutó `lock`

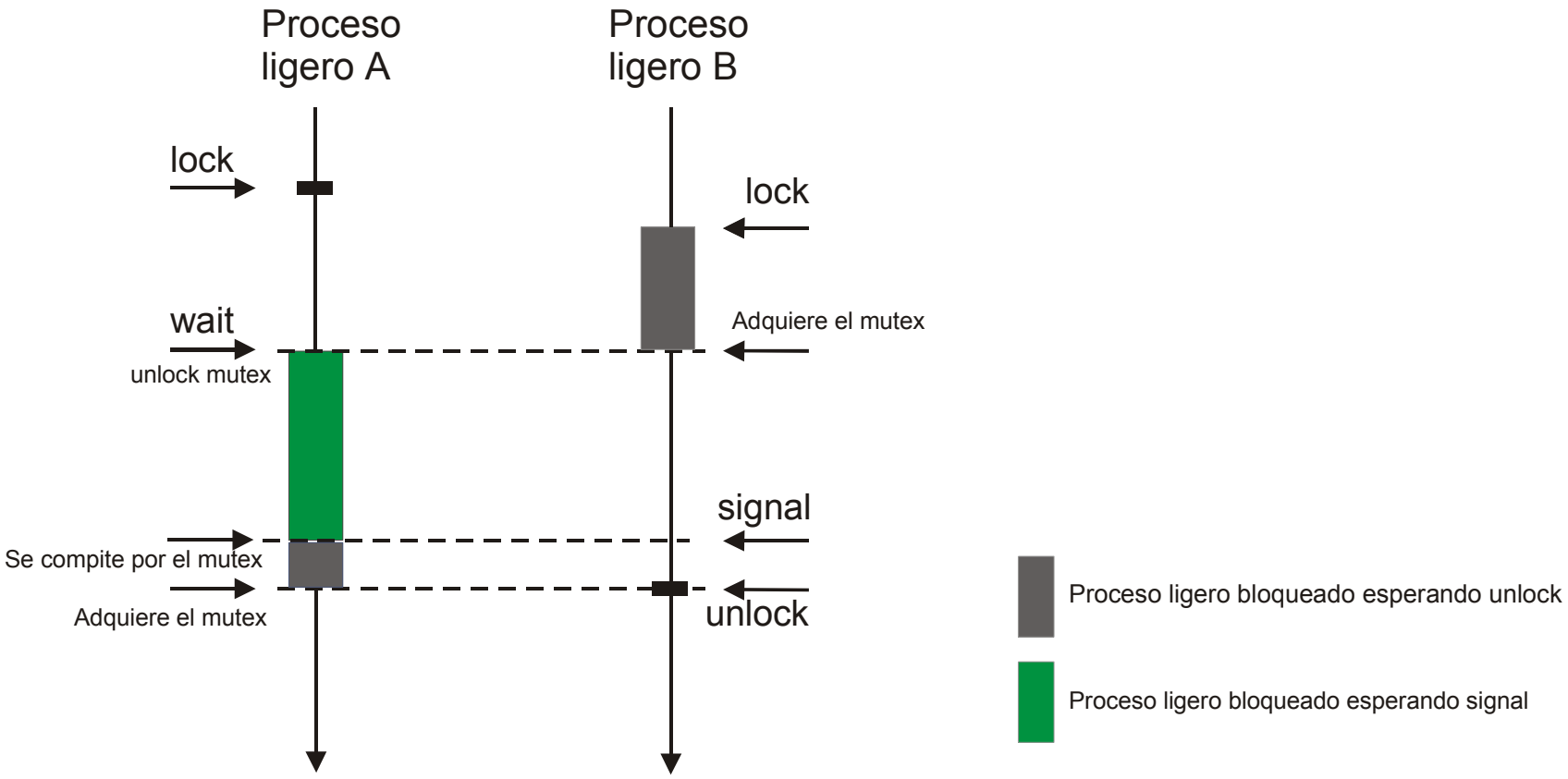


Variables condicionales

- Variables de sincronización asociadas a un mutex
- Dos operaciones atómicas:
  - `wait` Bloquea al proceso ligero que la ejecuta y le expulsa del mutex
  - `signal` Desbloquea a uno o varios procesos suspendidos en la variable condicional. El proceso que se despierta compite de nuevo por el mutex

- Conveniente ejecutarlas entre `lock` y `unlock`

Variables condicionales



Uso de mutex y variables condicionales

- Proceso ligero A

```
lock(mutex);  /* acceso al recurso */
comprobar las estructuras de datos;
while (recurso ocupado)
        wait(condition, mutex);
marcar el recurso como ocupado;
unlock(mutex);
```

- Proceso ligero B

```
lock(mutex);  /* acceso al recurso */
marcar el recurso como libre;
signal(condition, mutex);
unlock(mutex);
```

- Importante utilizar `while`

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`.

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

- Inicializa una variable condicional.

Servicios POSIX

`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`.

Productor consumidor con mutex

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

pthread_mutex_t mutex; /* mutex de acceso al buffer compartido */
pthread_cond_t no_lleno;   /* controla el llenado del buffer */
pthread_cond_t no_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(&no_lleno, NULL);
    pthread_cond_init(&no_vacio, NULL);
```

Productor-consumidor con mutex

```
    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(&no_lleno);
    pthread_cond_destroy(&no_vacio);

    exit(0);
}
```

Productor

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

Consumidor

```
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(&no_vacio, &mutex); /* se bloquea */
    dato = buffer[pos];
    pos = (pos + 1) % MAX_BUFFER;
    n_elementos --;
    pthread_cond_signal(&no_lleno);   /* buffer no lleno */
    pthread_mutex_unlock(&mutex);
    printf("Consume %d \n", dato);     /* consume dato */
  }
  pthread_exit(0);
}
```

Lectores-escritores con mutex

```
int dato = 5;                        /* recurso */
int n_lectores = 0;                  /* numero de lectores */
pthread_mutex_t mutex;               /* controlar el acceso a dato */
pthread_mutex_t mutex_lectores; /* controla acceso n_lectores */

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

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&no_lectores, 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);
```

Lectores-escritores con mutex

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

  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&no_lectores);

  exit(0);
}
```

Escritor

```
void Escritor(void) {   /* codigo del escritor */
   pthread_mutex_lock(&mutex);
   dato = dato + 2;     /* modificar el recurso */
   pthread_mutex_unlock(&mutex);
   pthread_exit(0);
}
```

Lector

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

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

   pthread_mutex_lock(&mutex_lectores);
   n_lectores--;
   if (n_lectores == 0) pthread_mutex_unlock(&mutex);
   pthread_mutex_unlock(&mutex_lectores);

   pthread_exit(0);
}
```

SISTEMAS OPERATIVOS:

Lección 7:
Hilos y mecanismos de comunicación y sincronización