SISTEMAS OPERATIVOS: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Hilos y mecanismos de comunicación y sincronización

A recordar…

Alejandro Calderón Mateos

Antes de clase | Clase | Después de clase

Preparar los pre-requisitos.

Estudiar el material asociado a la **bibliografía**:
las transparencias solo no son suficiente.
Preguntar dudas (especialmente tras estudio).

Ejercitar las competencias:

- Realizar todos los **ejercicios**.
- Realizar los **cuadernos de prácticas** y las **prácticas** de forma progresiva.

Lecturas recomendadas

Alejandro Calderón Mateos

Base

1. Carretero 2020:
   1. Cap. 6
2. Carretero 2007:
   1. Cap. 6.1 y 6.2

Recomendada

1. Tanenbaum 2006:
   1. (es) Cap. 5
   2. (en) Cap. 5
2. Stallings 2005:
   1. 5.1, 5.2 y 5.3
3. Silberschatz 2006:
   1. 6.1, 6.2, 6.5 y 6.6

Contenidos

Alejandro Calderón Mateos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escritores
- Mecanismos de sincronización de *threads* (II)
  - Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.
- Caso estudio: desarrollo de servidores concurrentes

Contenidos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escritores
- Mecanismos de sincronización de *threads* (II)
  - Semáforos
    - **Llamadas al sistema para semáforos.**
    - Problemas clásicos de concurrencia.
  - Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.
- Caso estudio: desarrollo de servidores concurrentes

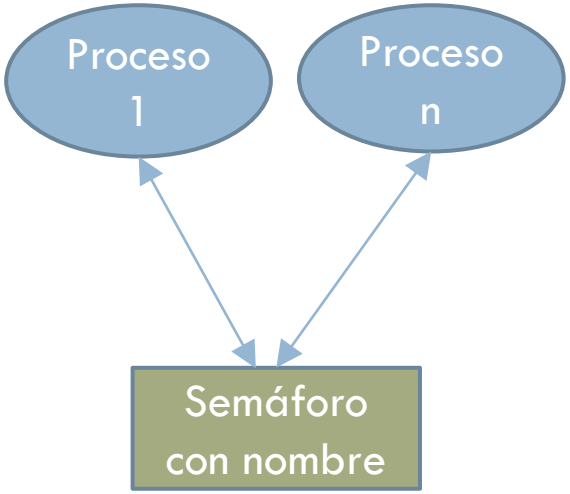
Semáforos POSIX

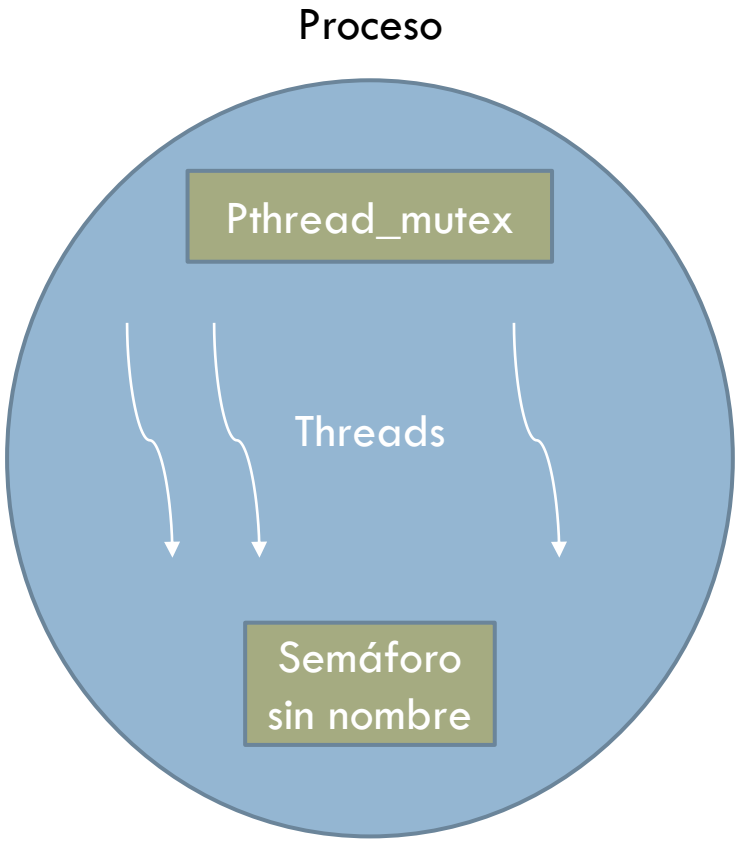
- Mecanismo de sincronización para procesos y/o threads en la misma máquina.

```
#include <semaphore.h>
```

- Semáforos POSIX de dos tipos:
  - **Semáforos con nombre:**
    - Puede ser usado por distintos procesos que conozcan el nombre.
    - No requiere memoria compartida.
    - `sem_t *semaforo; // nombrados`
  - **Semáforos sin nombre:**
    - Pueden ser usados solo por el procesos que los crea (y sus threads) o por procesos que tengan una zona de memoria compartida.
    - `sem_t  semaforo; // no nombrado`

Semáforos POSIX. ¿Cuál usar?





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_trywait (sem_t *sem)`
  - Intenta hacer wait, pero si está bloqueado vuelve sin hacer nada y da -1
- `int sem_post(sem_t *sem);`
  - Realiza la operación `signal` sobre un semáforo.

Operaciones sobre semáforos

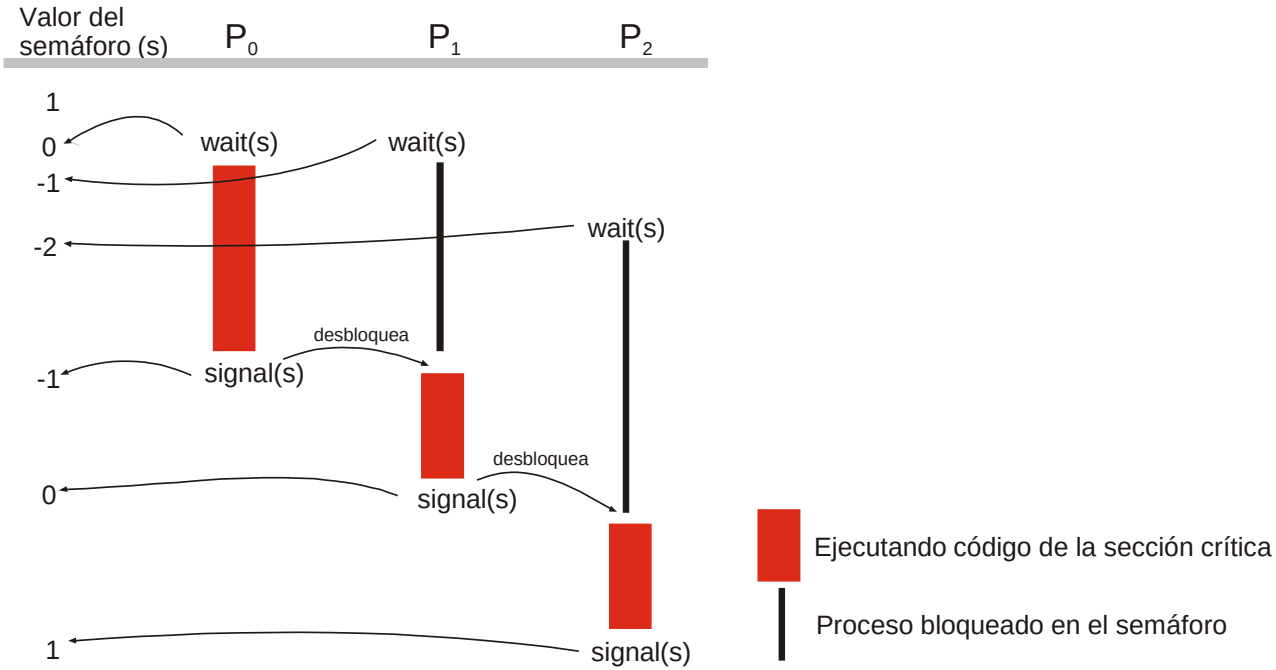

```
sem_wait(s) {
    s = s - 1;
    if (s <= 0) {
        <Bloquear
         al proceso>
    }
}
```

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

Secciones críticas con semáforos


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

- El semáforo debe tener valor inicial 1 o superior



Contenidos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escritores
- Mecanismos de sincronización de *threads* (II)
  - Semáforos
    - Llamadas al sistema para semáforos.
    - **Problemas clásicos de concurrencia.**
  - Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.
- Caso estudio: desarrollo de servidores concurrentes

Productor-consumidor
Semáforos sin nombre

Alejandro Calderón Mateos

```
/* tamanio del buffer */
#define MAX_BUFFER          1024

sem_t elementos;           /* eltos. en el búfer */
sem_t huecos;              /* huecos en el búfer */
int buffer[MAX_BUFFER];    /* búfer común */
```

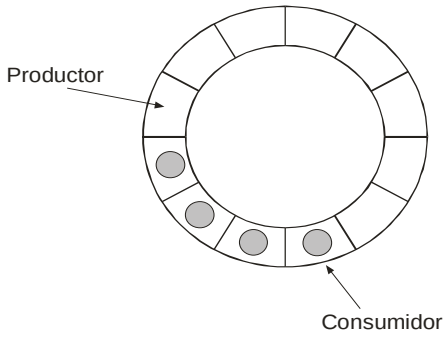
```
void Productor(void)
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      dato = i;
      sem_wait(&huecos);
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);
  }

  pthread_exit(0);
}
```

```
void Consumidor ( void )
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      sem_wait(&elementos);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);
      /* consumir dato */
   }

   pthread_exit(0);
}
```

Productor-consumidor
Semáforos sin nombre (1/4)

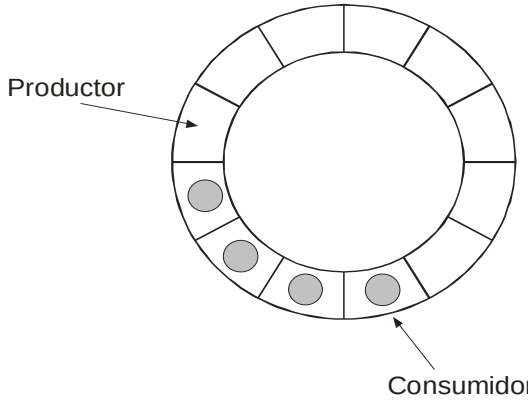

```
/* tamanio del buffer */
#define MAX_BUFFER          1024
/* datos a producir */
#define DATOS_A_PRODUCIR  100000

sem_t elementos; /* eltos. en el buffer */
sem_t huecos;      /* huecos en el buffer */
int buffer[MAX_BUFFER]; /* búfer común */

void main(void)
{
   pthread_t th1, th2;

   /* inicializar los semáforos */
   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);
}

void Productor(void)
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      dato = i;
      sem_wait(&huecos);
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);
   }

   pthread_exit(0);
}

void Consumidor ( void )
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      sem_wait(&elementos);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);
      /* consumir dato */
   }

   pthread_exit(0);
}
```

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

sem_t elementos; /* eltos. en el buffer */
sem_t huecos;      /* huecos en el buffer */
int buffer[MAX_BUFFER]; /* búfer común */

void main(void)
{
   pthread_t th1, th2;

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



Productor-consumidor
Semáforos sin nombre (2/4)


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

sem_t elementos; /* eltos. en el buffer */
sem_t huecos;       /* huecos en el buffer */
int buffer[MAX_BUFFER]; /* búfer común */

void main(void)
{
   pthread_t th1, th2;

   /* inicializar los semáforos */
   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);
}

void Productor(void)
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      dato = i;
      sem_wait(&huecos);
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);
   }

   pthread_exit(0);
}

void Consumidor ( void )
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      sem_wait(&elementos);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);
      /* consumir dato */
   }

   pthread_exit(0);
}
```




```
/* 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
Semáforos sin nombre (3/4)


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

sem_t elementos; /* eltos. en el buffer */
sem_t huecos;       /* huecos en el buffer */
int buffer[MAX_BUFFER]; /* búfer común */

void main(void)
{
   pthread_t th1, th2;

   /* inicializar los semáforos */
   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);
}
```

```
void Productor(void)
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      dato = i;
      sem_wait(&huecos);
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);
   }

   pthread_exit(0);
}
```

```
void Consumidor ( void )
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      sem_wait(&elementos);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);
      /* consumir dato */
   }

   pthread_exit(0);
}
```

```
void Productor(void)
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      dato = i;
      sem_wait(&huecos);
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);
   }

   pthread_exit(0);
}
```

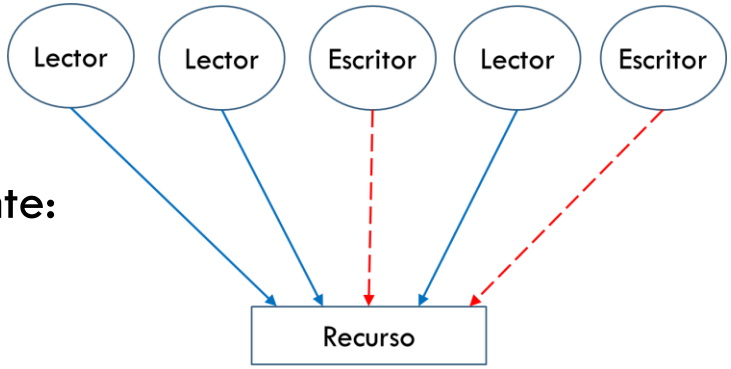
Productor-consumidor
Semáforos sin nombre (4/4)


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

sem_t elementos; /* eltos. en el buffer */
sem_t huecos;       /* huecos en el buffer */
int buffer[MAX_BUFFER]; /* búfer común */

void main(void)
{
   pthread_t th1, th2;

   /* inicializar los semáforos */
   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);
}

void Productor(void)
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      dato = i;
      sem_wait(&huecos);
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);
   }

   pthread_exit(0);
}
void Consumidor ( void )
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      sem_wait(&elementos);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);
      /* consumir dato */
   }

   pthread_exit(0);
}
```

```
void Consumidor ( void )
{
   int pos = 0;
   int dato;
   int i;

   for (i=0; i<DATOS_A_PRODUCIR; i++)
   {
      sem_wait(&elementos);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);
      /* consumir 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: posible varios lectores a la vez.
  - Solamente un escritor puede modificar la información a la vez.
  - Durante una escritura ningún lector puede leer.



Lectores y escritores

Semáforos sin nombre

Alejandro Calderón Mateos

```
int dato = 5;         /* recurso */
int n_lectores = 0;   /* num lectores */
sem_t sem_lec;        /* control el acceso n_lectores */
sem_t mutex;          /* controlar el acceso a dato */
```

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

  printf("%d\n", dato);

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

```
void Escritor(void)
{
    sem_wait(&mutex);
    dato = dato + 2;
    sem_post(&mutex);

    pthread_exit(0);
}
```

Lectores y escritores

Semáforos sin nombre

Alejandro Calderón Mateos

```
int dato = 5;         /* recurso */
int n_lectores = 0;   /* num lectores */
sem_t sem_lec;        /* control el acceso n_lectores */
sem_t mutex;          /* controlar el acceso a dato */

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

   sem_destroy(&mutex);
   sem_destroy(&sem_lec);

   return 0;
}
```

Lectores-escritores (lectores tienen prioridad)

Semáforos sin nombre

MEMORIA COMPARTIDA:

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

LECTOR:

```
for(;;) {
  sem_wait(lec);
  nlect++;
  if (nlect==1)
      sem_wait(escr);
  sem_signal(lec);

  realizar_lect();

  sem_wait(lec);
  nlect--;
  if (nlect==0)
      sem_signal(escr);
  sem_signal(lec);
}
```

ESCRITOR:

```
for(;;) {
  sem_wait(escr);
  realizar_escr();
  sem_signal(escr);
}
```

Lectores-escritores (escritores tienen prioridad)

Semáforos sin nombre

MEMORIA COMPARTIDA:

```
int nlect, nescr = 0;  semaphore lect, escr = 1;
semaphore x, y, z = 1;
```

LECTOR:

```
for(;;) {
    sem_wait(z);
    sem_wait(lect);
    sem_wait(x);
      nlect++;
      if (nlect==1)
          sem_wait(escr);
    sem_signal(x);
    sem_signal(lect);
    sem_signal(z);
      // doReading();
    sem_wait(x);
      nlect--;
      if (nlect==0)
          sem_signal(escr);
    sem_signal(x);
}
```

ESCRITOR:

```
for(;;) {
    sem_wait(y);
      nescr++;
      if (nescr==1)
          sem_wait(lect);
    sem_signal(y);
    sem_wait(escr);
      // doWriting();
    sem_signal(escr);
    sem_wait(y);
      nescr--;
      if (nescr==0)
          sem_signal(lect);
    sem_signal(y);
}
```

Semáforos con nombre
Nombrado

- Permiten sincronizar procesos distintos sin usar memoria compartida.
- El nombre de un semáforo es una cadena de caracteres (con las mismas restricciones de un nombre de fichero).
  - Si el nombre (ruta) es relativa, solo puede acceder al semáforo el proceso que lo crea y sus hijos.
  - Si el nombre es absoluto (comienza por "/") el semáforo puede ser compartido por cualquier proceso que sepa su nombre y tenga permisos.
- Mecanismo habitual para crear semáforos que comparten padres e hijos
  - Los "sin nombre" no valen -> los procesos NO comparten memoria.

Semáforos con nombre
Creación y uso

- Para crearlo:

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

  - Flag = O_CREAT lo crea.
  - Flag: O_CREAT | O_EXECL. Lo crea si no existe. -1 en caso de que exista.
  - Mode: permisos de acceso;
  - Val: valor incial del semáforo (>=0);

- Para usarlo:

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

  - Con flag 0. Si no existe devuelve -1.

- Importante:
  - Todos los procesos deben conocer “name” y usar el mismo.

Lectores y escritores
semáforos con nombre

Alejandro Calderón Mateos

```
int main ( int argc, char *argv[] )
{
    int i, n= 5; pid_t pid;

    /* Crea el semáforo nombrado */
    if ((mutex=sem_open("/tmp/sem_1", O_CREAT, 0644, 1))==(sem_t *)-1)
        { perror("No se puede crear el semaforo"); exit(1); }
    if((sem_lec=sem_open("/tmp/sem_2", O_CREAT, 0644, 1))==(sem_t *)-1)
        { perror("No se puede crear el semraáforo"); exit(1); }

    /* Crea los procesos */
    for (i = 1; i< atoi(argv[1]); ++i)
    {
      pid = fork();
      if (pid ==-1)
         { perror("No se puede crear el proceso"); exit(-1);}
      if (pid==0)
         { lector(getpid()); break; }
      else escritor(pid);
    }
    sem_close(mutex);
    sem_close(sem_lec);
    sem_unlink("/tmp/sem_1");
    sem_unlink("/tmp/sem_2");

    return 0;

}
```

Lectores y escritores
semáforos con nombre

Alejandro Calderón Mateos

```
int dato = 5;              /* recurso */
int n_lectores = 0;        /* num lectores */
sem_t *sem_lec;
sem_t *mutex;
```

```
void lector (int pid)
{
  sem_wait(sem_lec);
  n_lectores = n_lectores + 1;
  if (n_lectores == 1)
      sem_wait(mutex);
  sem_post(sem_lec);

  printf("lector %d  dato: %d\n",
         pid, dato);

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

```
void escritor (int pid)
{
    sem_wait(mutex);
    dato = dato + 2;

    printf("escritor %d  dato: %d\n",
           pid, dato);

    sem_post(mutex);
}
```

Contenidos

Alejandro Calderón Mateos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escritores
- Mecanismos de sincronización de *threads* (II)
  - Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - **Mutex y variables condición**
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.
- Caso estudio: desarrollo de servidores concurrentes

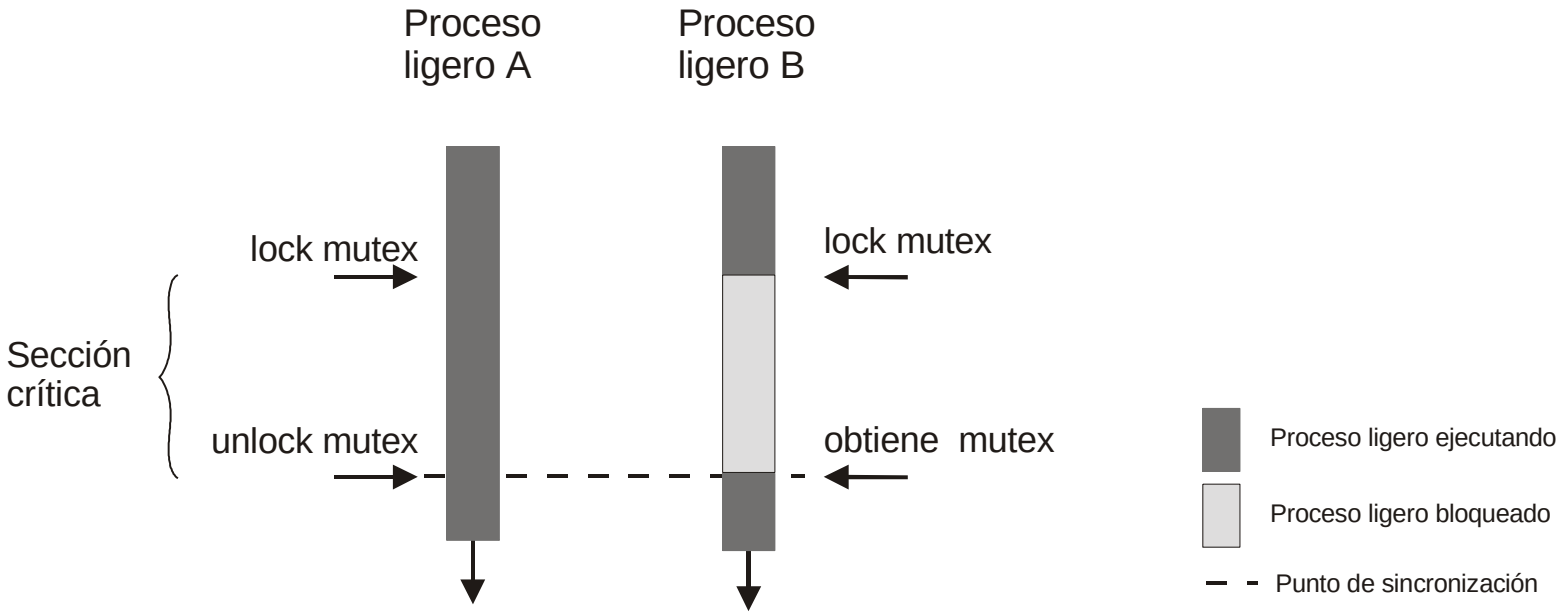
Mutex y variables condicionales

- 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 */
```

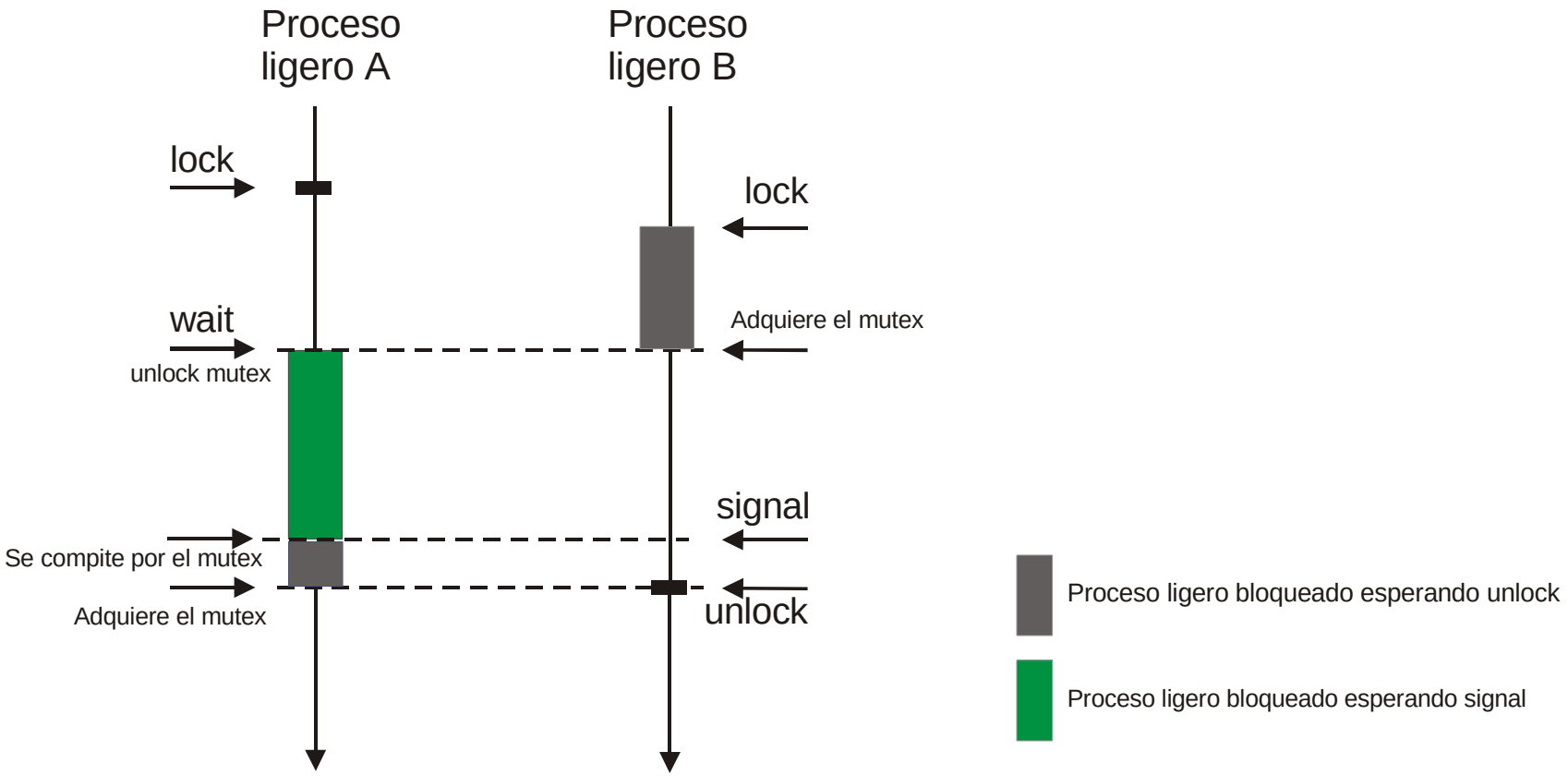
- La operación `unlock` debe realizarla el proceso ligero que ejecutó `lock`



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

Alejandro Calderón Mateos

- 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`

Contenidos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escritores
- Mecanismos de sincronización de *threads* (II)
  - Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - Mutex y variables condición
    - **Llamadas al sistema para mutex.**
    - Problemas clásicos de concurrencia.
- Caso estudio: desarrollo de servidores concurrentes

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

Alejandro Calderón Mateos

```
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`.

Contenidos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escritores
- Mecanismos de sincronización de *threads* (II)
  - Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - Mutex y variables condición
    - Llamadas al sistema para mutex.
    - **Problemas clásicos de concurrencia.**
- Caso estudio: desarrollo de servidores concurrentes

Productor-consumidor con mutex

Alejandro Calderón Mateos

```
int 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);

    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);
    return 0;
}
```

Productor-consumidor con mutex


```
#define MAX_BUFFER        1024    /* tamaño del búfer */
#define DATOS_A_PRODUCIR  100000  /* datos a producir */

pthread_mutex_t mutex;            /* mutex de acceso al búfer compartido */
pthread_cond_t no_lleno;          /* controla el llenado del búfer */
pthread_cond_t no_vacio;          /* controla el vaciado del búfer */
int n_elementos;                  /* número de elementos en el búfer */
int buffer[MAX_BUFFER];           /* búfer común */
```

```
void Productor ( void )
{
  int dato, i ,pos = 0;

  for(i=0; i<DATOS_A_PRODUCIR; i++)
  {
    dato = i;
    pthread_mutex_lock(&mutex);
    while (n_elementos == MAX_BUFFER)
           pthread_cond_wait(&no_lleno,
                             &mutex);
    buffer[pos] = i;
    pos = (pos + 1) % MAX_BUFFER;
    n_elementos ++;
    pthread_cond_signal(&no_vacio);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

```
void Consumidor(void)
{
  int dato, i ,pos = 0;

  for(i=0; i<DATOS_A_PRODUCIR; i++)
  {
    pthread_mutex_lock(&mutex);
    while (n_elementos == 0)
           pthread_cond_wait(&no_vacio,
                             &mutex);
    dato = buffer[pos];
    pos = (pos + 1) % MAX_BUFFER;
    n_elementos --;
    pthread_cond_signal(&no_lleno);
    pthread_mutex_unlock(&mutex);
    printf("Consume %d \n", dato);
  }
  pthread_exit(0);
}
```

Lectores-escritores con mutex

Alejandro Calderón Mateos

```
int 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);
    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);
    return 0;
}
```

Lectores-escritores con mutex

Alejandro Calderón Mateos

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

```
void Lector(void)
{
  pthread_mutex_lock(&mutex_lectores);
  n_lectores++;
  if (n_lectores == 1)
      pthread_mutex_lock(&mutex);
  pthread_mutex_unlock(&mutex_lectores);

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

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

  pthread_exit(0);
}
```

```
void Escritor (void)
{
   pthread_mutex_lock(&mutex);

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

   pthread_mutex_unlock(&mutex);

   pthread_exit(0);
}
```

SISTEMAS OPERATIVOS: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Hilos y mecanismos de comunicación y sincronización