# SISTEMAS OPERATIVOS: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

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.

  Al recibir una señal se desbloquea

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

  } Es en las que nos centramos. Las + recomendadas.

- Las operaciones de sincronización deben ser **atómicas**

# Semáforos POSIX



- Mecanismo de sincronización para procesos y/o threads en la misma máquina
- Semáforos POSIX de dos tipos:
  - **Semáforos con nombre**: puede ser usado por distintos procesos que conozcan el nombre. No requiere memoria compartida.
  - **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.

```
#include <semaphore.h>
sem_t *semaforo; //nombrados
sem_t semaforo; // no nombrado
```

sem_wait(s) Resta
sem_post(s) Suma

# Semáforos POSIX

Procesos distintos: Sem. con nombre
Mismo proceso: sem sin nombre ó pthread_mutex.
Non copia un sem. al hacer fork.



- `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.

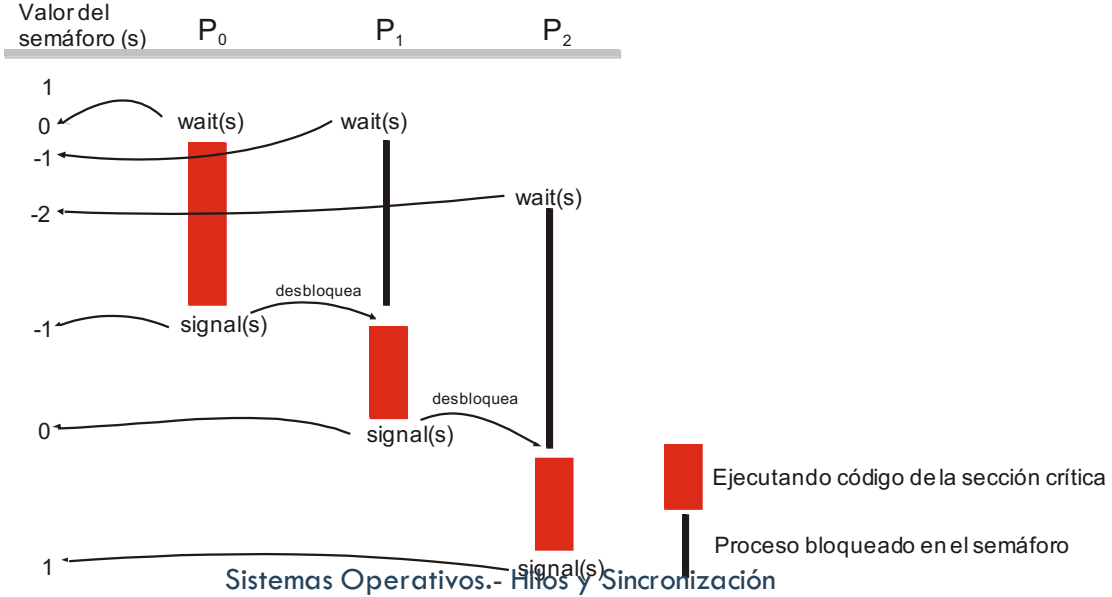
¿0
¿los que pueden acceder a la vez inicialmente } si lo pongo en 1 pueden pasar y con 0 esperan en sem_wait.
nombre
si creor o no O_CREAT
si lectra/escrit O_RDWR
valor de inicialización,
1. Crea (abre)
2. cierra, pero no borra
3. Borro, importante hacerlo.
Cuando es 0 espera
No lo usamos
Suma 1, para liberar y dejar pasar.

# 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



# 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>
    }
}
```

# Semáforos sin nombre.
# Ejemplo: Productor-consumidor.

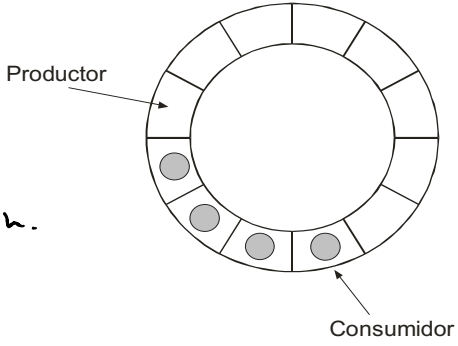


```
#define MAX_BUFFER          1024    /* tamanio del buffer */
#define DATOS_A_PRODUCIR  100000    /* datos a producir */

sem_t elementos;                    /* elementos en el buffer */
sem_t huecos;                       /* huecos en el buffer */
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);
}
```

Que ejecute

Espero que terminen.

Destruyo los semáforos

Fin programa.

# Semáforos sin nombre.

## 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);
}
```

Quita y pide hueco
Escribe
La siguiente posicion del cir circular
Añade un elemento.

# Semáforos sin nombre.
# 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);
}
```

Espera a que haya para leer

lee

siguiente posición

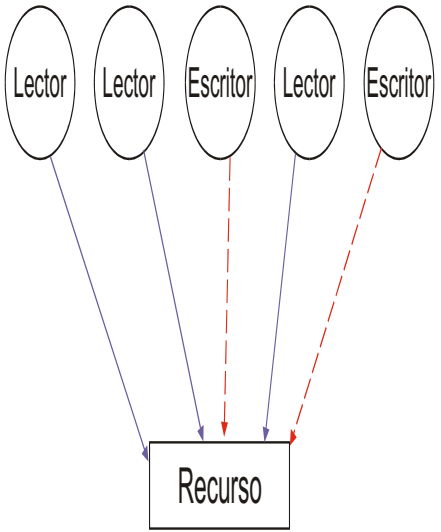
Añade un hueco

# 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.



# 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



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

**Lector**

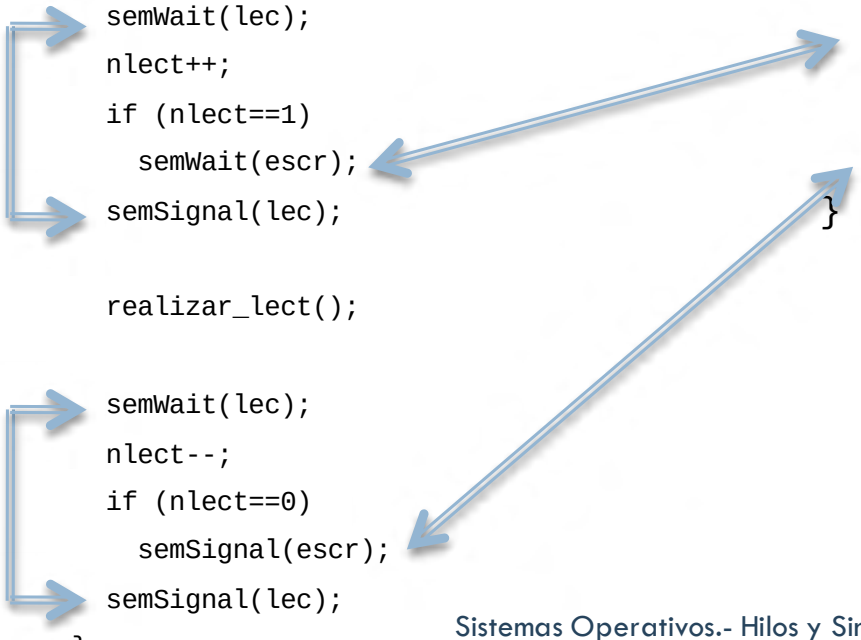
```
for(;;) {
  semWait(lec);
  nlect++;
  if (nlect==1)
    semWait(escr);
  semSignal(lec);

  realizar_lect();

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

**Escritor**

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



# Semáforos sin nombre. Lectores-escritores con semáforos



```
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 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);
```

2 lectores y 2 escritores

```
   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(&mutex);
   sem_destroy(&sem_lec);
   exit(0);
}
```

# Semáforos sin nombre. Lectores-escritores:Hilo lector y escritor



```
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);
}
```

bloqua escritor mientras haya lectores

Desbloquea la escritura al no haber lectores.

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

    pthread_exit(0);
}
```

# 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.

# Semáforos con nombre: Lectores - Escritores



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


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);}
    else if(pid==0){  /child
       lector(getpid());    break;
     }
     escritor(pid); /* parent */
  }
    sem_close(mutex);  sem_close(sem_lec);
    sem_unlink("/tmp/sem_1");
    sem_unlink("/tmp/sem_2");
}
```

# Semáforos con nombre: Procesos lectores y escritores



```
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);  /* leer 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;     /* modificar el recurso */
   printf("escritor %d  dato: %d\n", pid, dato);  /* leer dato */
   sem_post(mutex);
}
```

# Mutex y variables condicionales



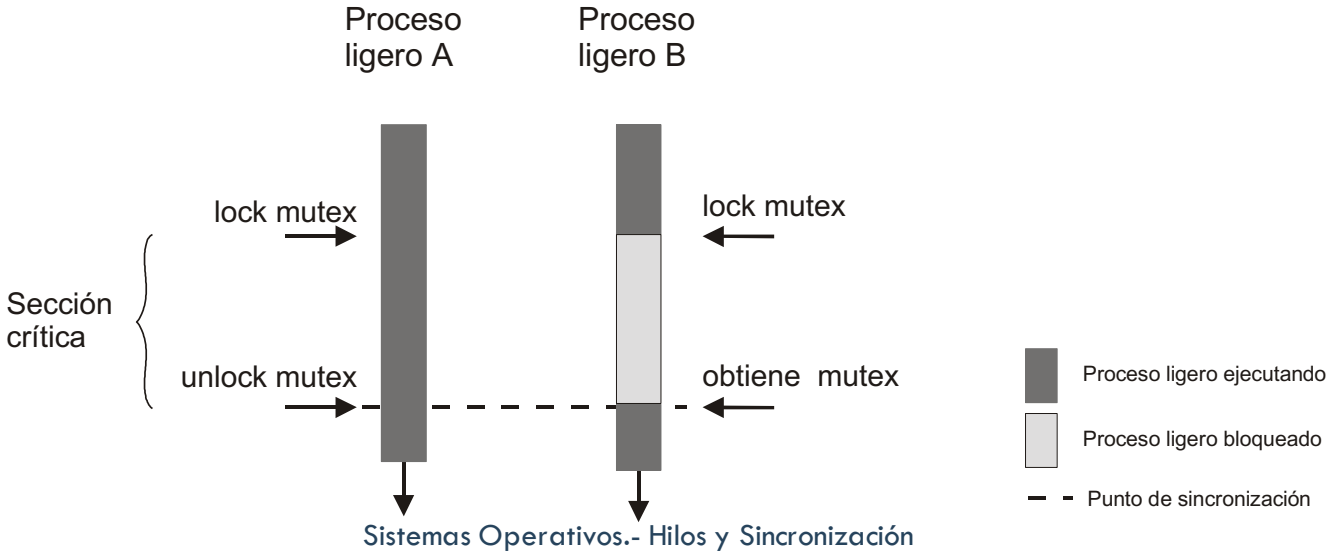
- Un mutex es un mecanismo de sincronización indicado para procesos ligeros.
- Es un semáforo binario con dos operaciones atómicas:
  - **lock(m)** Intenta bloquear el mutex, si el mutex ya está bloqueado el proceso se suspende.
  - **unlock(m)** Desbloquea el mutex, si existen procesos bloqueados en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



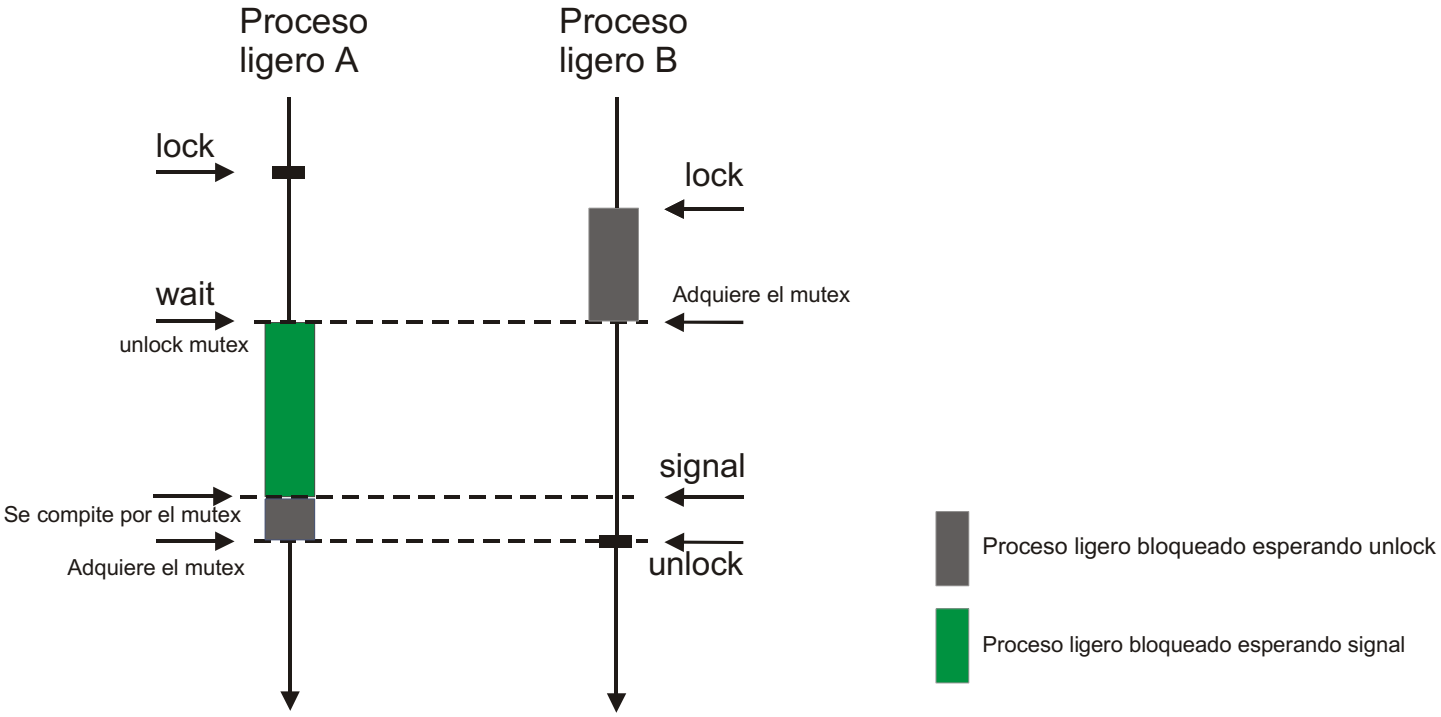
# Variables condicionales



- 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);
```

si no se cumple condicion se bloquea.

- Proceso ligero B

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

- Importante utilizar `while`

# Servicios POSIX

Solo tienen valor 1 o 0, no son mutex para acceder semáforos.

Solo threads, no procesos



```
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);
}
```

Crea los hilos

# 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);
}
```

Bloquea mientras lousa

Lo si esta lleno lo bloquea.

→ 1 + escrito

Hay algo

# 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 --; 1- que leer.
    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: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Mecanismos de comunicación y sincronización