Comunicación y Sincronizacion de procesos

Ingeniería del Software

EUITI – UPM

Índice

- Intro
  - Seccion critica
  - Productor consumidor
  - Lectores escritores
- Tuberias
- Semaforos
- Memoria compartida
- Mutex y variables condicionales
- Mensajes

Procesos concurrentes

- Modelos
  - Multiprogramacion, 1 CPU
  - Multiprocesador
  - Multicomputador (distribuido)
- Razones
  - Compartir recursos, logicos y fisicos
  - Uso eficiente CPU
  - Modularidad, comodidad

Problemas

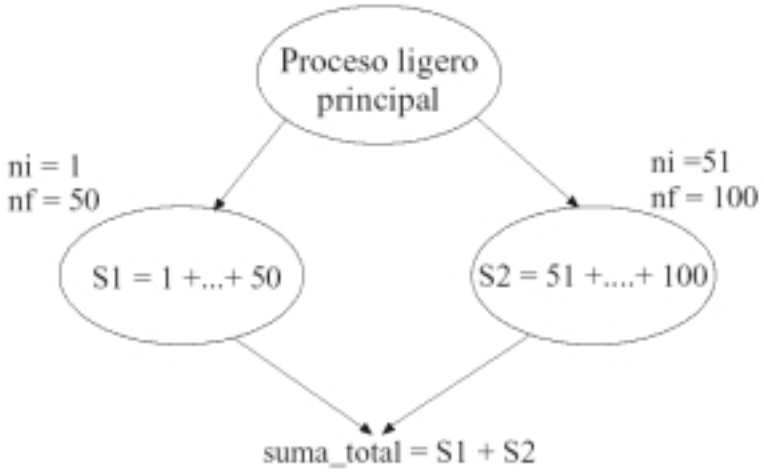
- El problema de la seccion critica
- El problema del productor-consumidor
- El problema de los lectores-escritores
- Comunicación cliente-servidor

Sección critica

- Sistema compuesto por n procesos
- Cada uno tiene un fragmento de código denominado “seccion critica” que cuando se esta ejecutando ningun otro proceso debe hacerlo.
- Ejecución de forma atomica

Ejemplo: suma

- Calcular la suma de N primeros números naturales con procesos ligeros (ej 100 con 2 threads)



Ejemplo

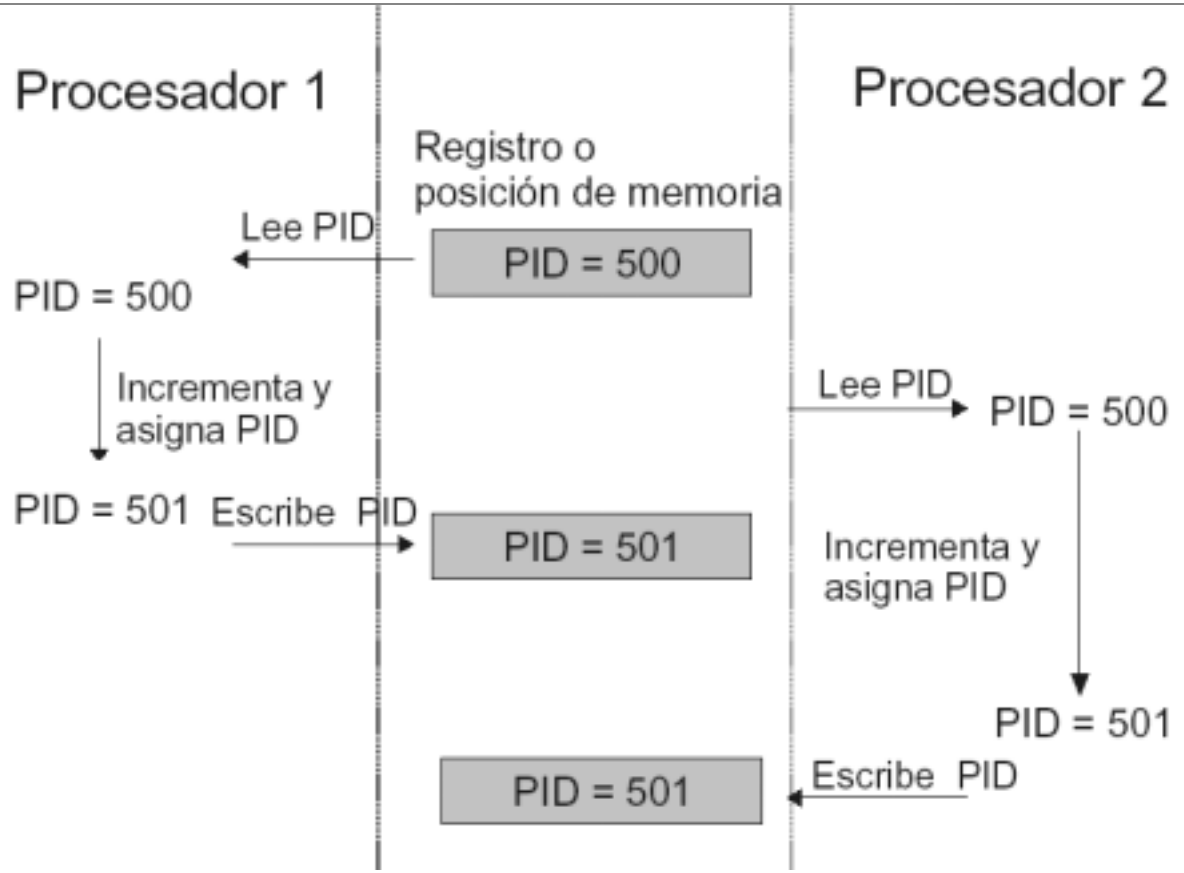
```
int suma_total=0;

void suma_parcial(int ni,int nf)
{
    int j;
    int suma=0;
    for(j=ni;j<=nf;j++)
          suma=suma+j;
    ENTRADA EN LA SECCION CRITICA
    suma_total=suma_total+suma;
    SALIDA DE LA SECCION CRITICA
}
```

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;
}
```

```
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;
}
```

Ejemplo 3


Procesador 1
Procesador 2
Registro o
posición de memoria
Lee PID
PID = 500
PID = 500
Incrementa y
asigna PID
Lee PID
PID = 500
PID = 501
Escribe PID
PID = 501
Incrementa y
asigna PID
PID = 501
PID = 501
Escribe PID

Sección critica

ENTRADA EN LA SECCION CRITICA

suma_total=suma_total+suma;

SALIDA DE LA SECCION CRITICA

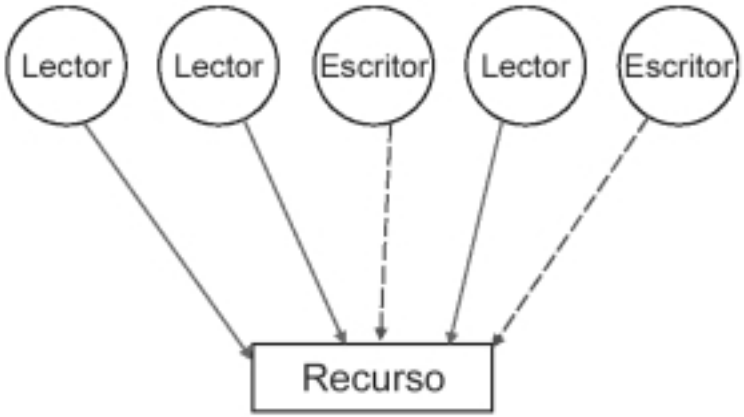
- Requisitos:
  - Exclusión mutua
  - Progreso, decisión sobre los procesos que quieren entrar, en tiempo finito
  - Espera limitada

Productor-consumidor

- Sincronizar:
  - Cuando el mecanismo se llene, el productor se debe bloquear
  - Cuando el mecanismo este vacío, el consumidor se debe bloquear

Lectores escritores

- Procesos escritores: exclusivamente 1 escritor
- Procesos lectores: pueden acceder simultáneamente.



Comunicación cliente-servidor

- Local:
  - Memoria compartida o archivos
- Remoto: Aplicación distribuida

Mecanismos

- Comunicación:
  - Archivos
  - Tuberias
  - Memoria compartida
  - Paso de mensajes
- Sincronización (operaciones atómicas)
  - Señales
  - Tuberías
  - Semáforos
  - Mutex y variables condicionales
  - Paso de mensajes

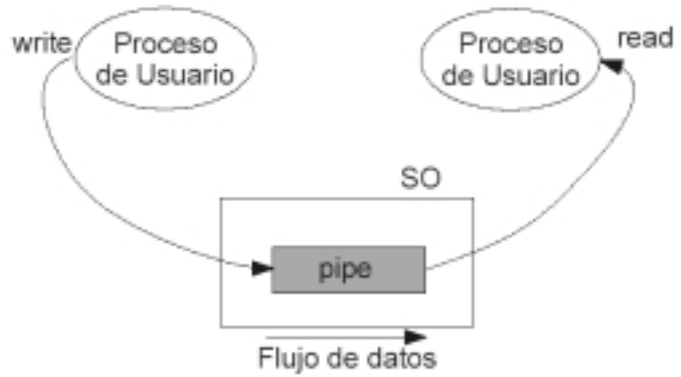
Comunicación con archivos

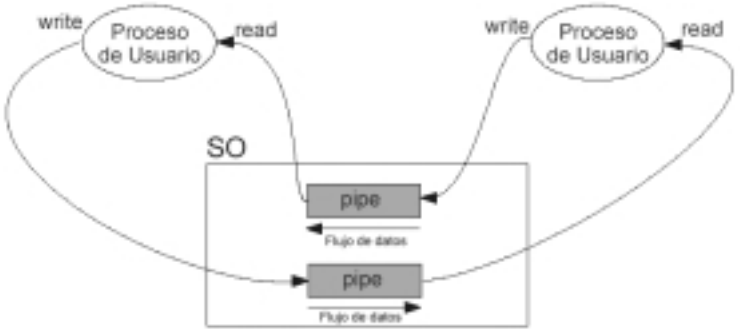
- Ventajas:
  - Numero ilimitado de procesos, siempre y cuando tengan permisos.
  - Los servidores de archivos ofrecen servicios fáciles de usar.
- Desventajas
  - Poco eficiente: disco lento
  - Necesita otro mecanismo de sincronizacion

```
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;
}
```

Tuberias (pipes)

- Mecanismo del SO de comunicación y sincronización
- Pseudoarchivo mantenido por SO
  - Unidireccional ¿Bidireccional?
  - FIFO
  - Capacidad de almacenamiento de 4 Kb, implementadas como memoria compartida
  - Múltiples lectores y escritores
  - Misma maquina





Buffer circular

Tuberias

- Tipos
  - Sin nombre (pipes)
  - Con nombre (FIFOS)
- PIPES
  - Solo pueden utilizarse en procesos hijo (fork) del que creo el pipe. Los procesos hijo heredan el pipe.

    int pipe(int fildes[2]);

    Dos descriptores de archivo, de lectura y escritura

Lectura (pipe o FIFO)

- int read (int fd, char* buffer, int n)
- read(fildes[0],buffer,n)
  - Pipe vacio
    - Existen escritores ➔Bloquea el lector!!
    - No existen escritores ➔Fin de fichero (0), no bloqueante
  - Pipe con p bytes
    - Si p>= n devuelve n (los elimina)
    - Si p<n devuelve p (los elimina)
  - Operación atomica (cuidado con tamaños grandes)

Escritura (pipe o FIFO)

- int write (int fd, char* buffer,int n)
- write (fildes[1],buffer,n)
  - Si el pipe esta lleno, bloquea el lector
  - Si no hay lectores se recibe la señal SIGPIPE
  - Operación atomica (cuidado con tamaños grandes)

FIFOS

- int mkfifo(char* name,mode_t mode)

  0 si éxito, -1 en caso de error (errno)

  Solo lo llama un proceso
- El nombre es el de un pseudoarchivo, que se puede ver en la carpeta en la que se cree ls –al, con los permisos correspondientes (mode 0777)
- int open(char* fifo, int flag)

  flag=O_RDWR, O_RDONLY, O_WRONLY

  Devuelve un descriptor de archivo o -1 en caso de error. La llamada bloquea hasta que haya algun proceso en el otro extremo del FIFO (puede ser uno mismo)

Cierre del pipe o FIFO

- int close (int fd)

  0 si éxito, -1 si error

Eliminación del FIFO

- int unlink(char* fifo)
- Pospone la destrucción del FIFO hasta que todos los procesos que lo estén utilizando lo cierren con “close”
- Solo lo llama un proceso
- En el caso de tubería sin nombre, esta se elimina automáticamente cuando se cierra con “close” el ultimo descriptor asociado.

Resumen (FIFOS)

ESCRITOR

```
mkfifo("/home/mififo1", 0777);
int fd=open("/home/mififo1",O_WRONLY);
```

```
char buf[]="Hola"
write(fd,buf,sizeof(buf));
```

```
close(fd);
unlink ("/home/mififo1");
```

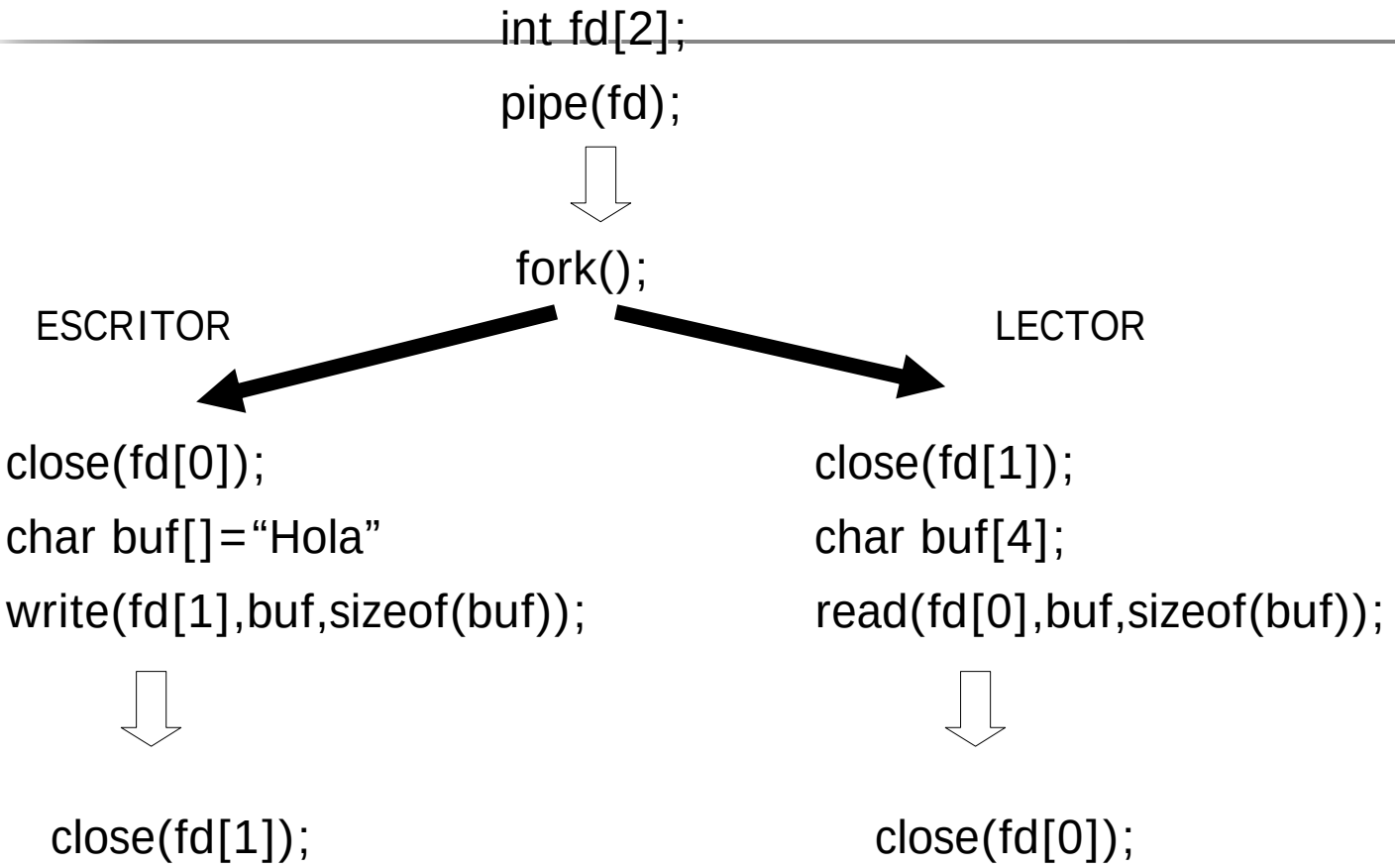
LECTOR

```
int fd=open("/home/mififo1",O_RDONLY);
```

```
char buf[4];
read(fd,buf,sizeof(buf));
```

```
close(fd);
```

Resumen (pipes)



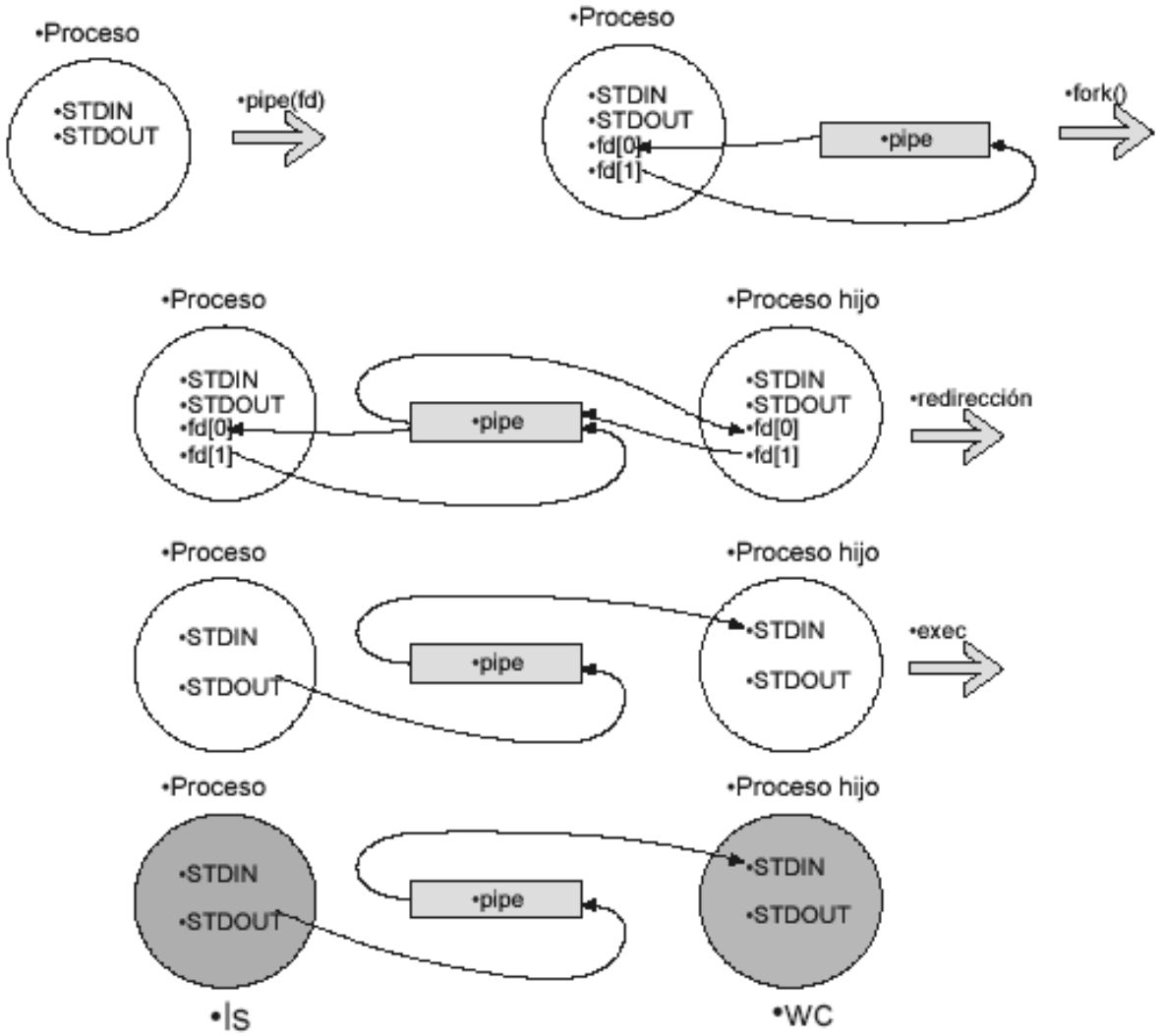
Usos

- Sección critica con tuberías
- Productor consumidor con tuberías
- Ejecución de mandatos con tuberías

  ls –al |more
- Comunicación con FIFOS

Ejecución de mandatos con tuberías

- ls |wc



Sincronización con señales

- Señales POSIX permiten:
  - Un proceso puede bloquearse mediante el servicio “pause” y esperar un señal de otro proceso enviada con “kill”
- Desventajas:
  - Las señales tienen comportamiento asincrono, puede recibir un señal en cualquier momento, aunque no la espere.
  - Las señales no se encolan, solo la ultima de un tipo, con lo que se pueden perder eventos de sincronizacion importantes
- No se suele usar para sincronizar procesos, exceptuando casos muy simples.

Semáforos

- Mecanismo de sincronización, en sistemas con memoria compartida
- Misma maquina
- Objeto con un valor entero, con un valor inicial no negativo
- Dos operaciones atómicas, que dependen del SO en particular
  - wait
  - signal

Operaciones sobre semaforos

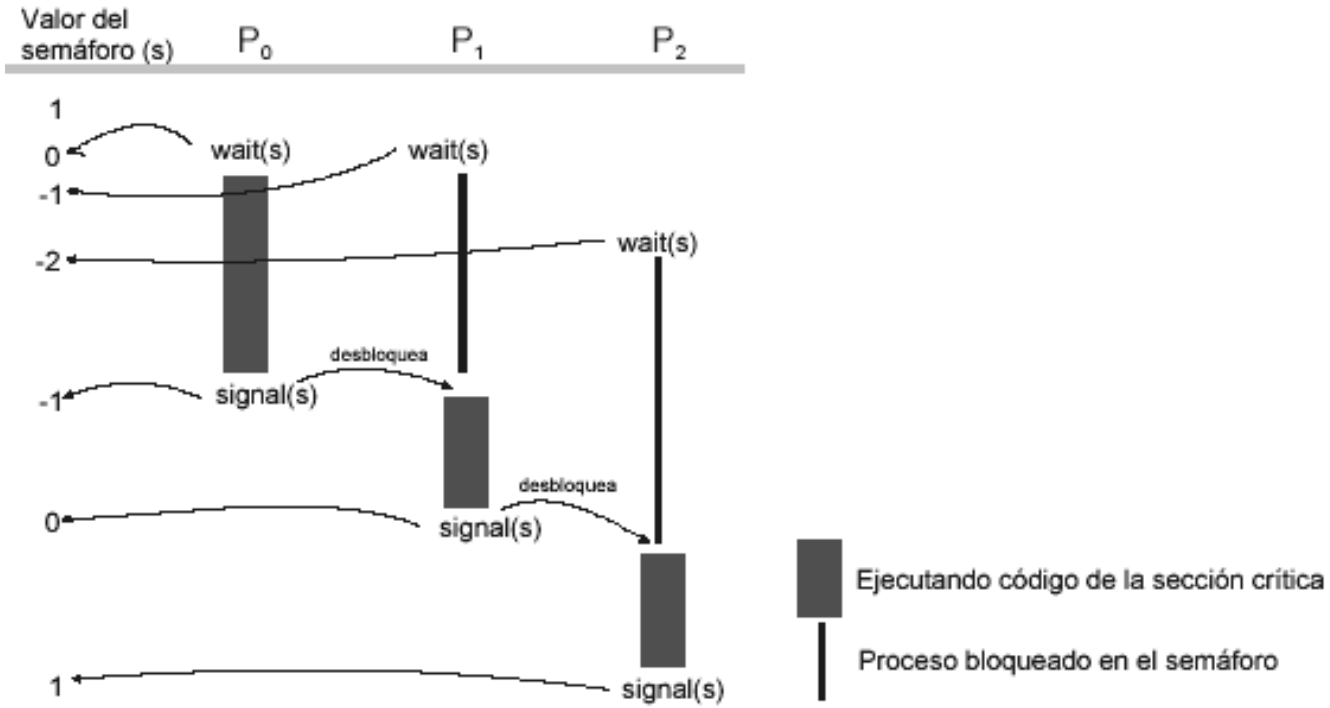
```
Semaforo s;
wait(s)
{
    s=s-1;
    if(s<0)
        //bloquear proceso
}
signal(s)
{
    s=s+1;
    if(s<=0)
        // desbloquear a un proceso bloqueado en la operación
        // wait
}
```

Seccion critica con semaforos

wait(s)

<seccion critica>

signal(s)

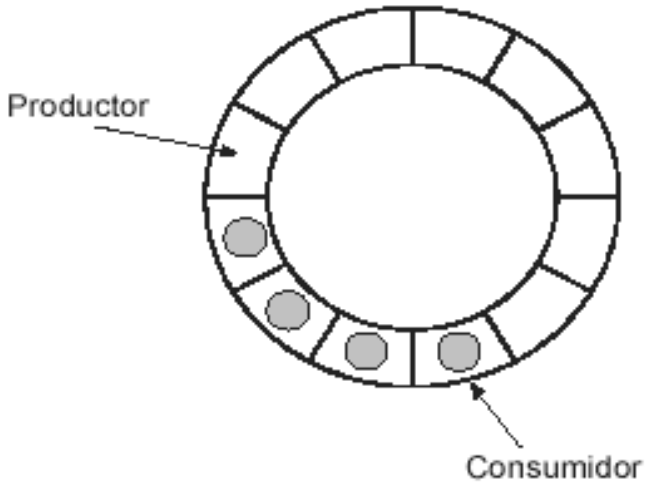


Semaforos POSIX

- Include
  - #include <semaphore.h>
- Creacion, inicializacion:
  - int sem_init (sem_t* s, int shared, int val)
  - sem_t* sem_open(char* name, int flag, mode_t m, int val)
  - sem_t* sem_open(char* name, int flag)
- Wait, signal:
  - int sem_wait(sem_t* s)
  - int sem_post(sem_t* s)
- Cierre, eliminacion:
  - int sem_close(sem_t* s)
  - int sem_destroy(sem_t* s)
  - int sem_unlink(sem_t* s)

Productor-consumidor con semaforos

- Cola circular de tamaño limitado
- El consumidor no puede sacar elementos cuando el buffer esta vacío
- El productor no puede colocar elementos cuando esta lleno
- El consumidor no debe sacar un elemento mientras el productor lo esta insertando
- 2 tipos de recursos: elementos y huecos, cada uno representado por un semáforo



Lectores y escritores con semaforos

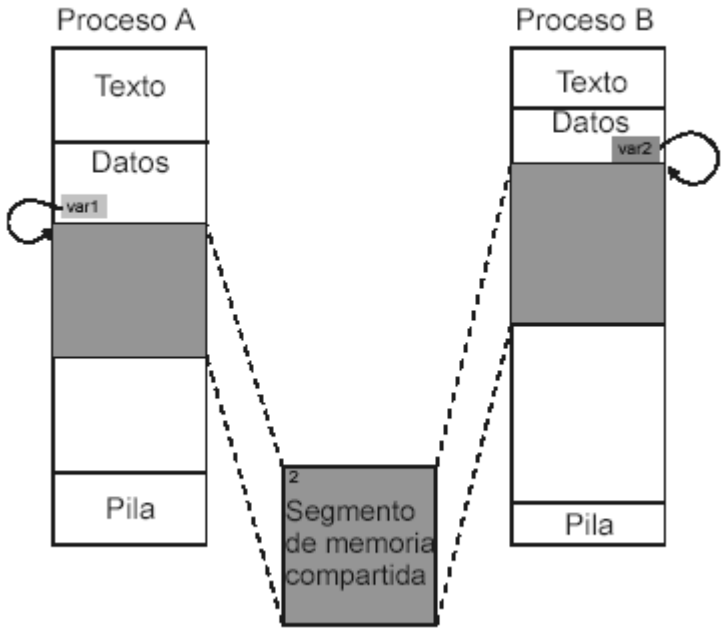
- Los lectores seran prioritarios
- Solo el primer lector debe esperar a que el recurso este libre.
- Solo el ultimo lector debe liberar el recurso
- 2 semaforos:
  - Acceso al recurso
  - Acceso al contador de lectores

Ejercicios

- Pares-Impares
- Suma en background

Memoria compartida

- Con procesos ligeros
  - Mismo espacio de direcciones
- Entre procesos:
  - V IPC
  - BSD mmap



Memoria compartida IPC

```
key_t mi_key=ftok("/bin/ls",13);
int shmid=shmget(mi_key,NUM_DATOS,0x1ff|IPC_CREAT);
char* punt=shmat(shmid,0,0x1ff);

shmdt(punt);
shmctl(shmid,IPC_RMID,NULL);
```

Memoria compartida BSD

#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);

prot=PROT_WRITE

flags=MAP_SHARED

int munmap(void *start, size_t length);

Abrir archivo:

int open(“ARCHIVO”,O_CREATE | O_RDWR, 0777);

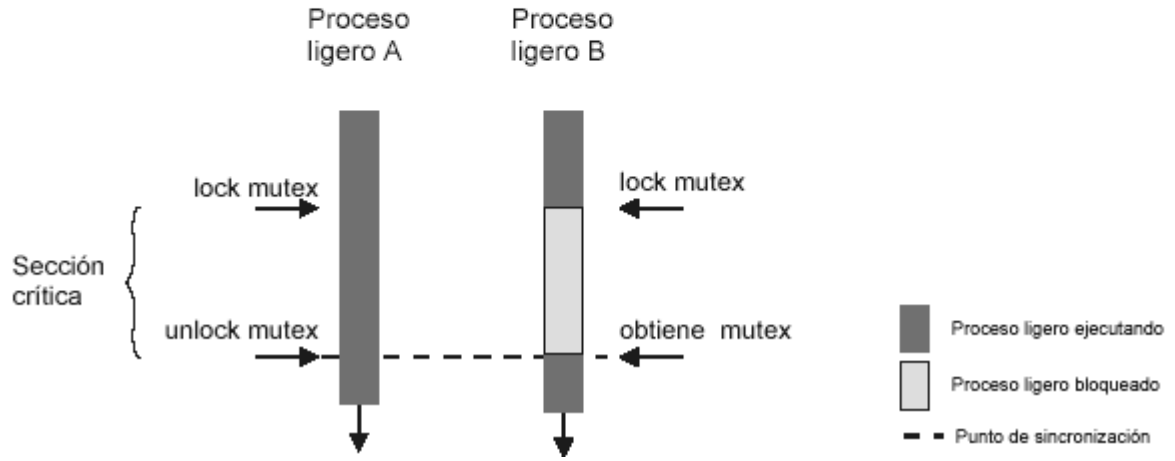
Mutex y variables condicionales

- Mutex es un mecanismo de comunicación para procesos ligeros
- Semaforo binario
  - Mutex m
  - lock(m)
  - unlock(m)

Seccion critica con mutex

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

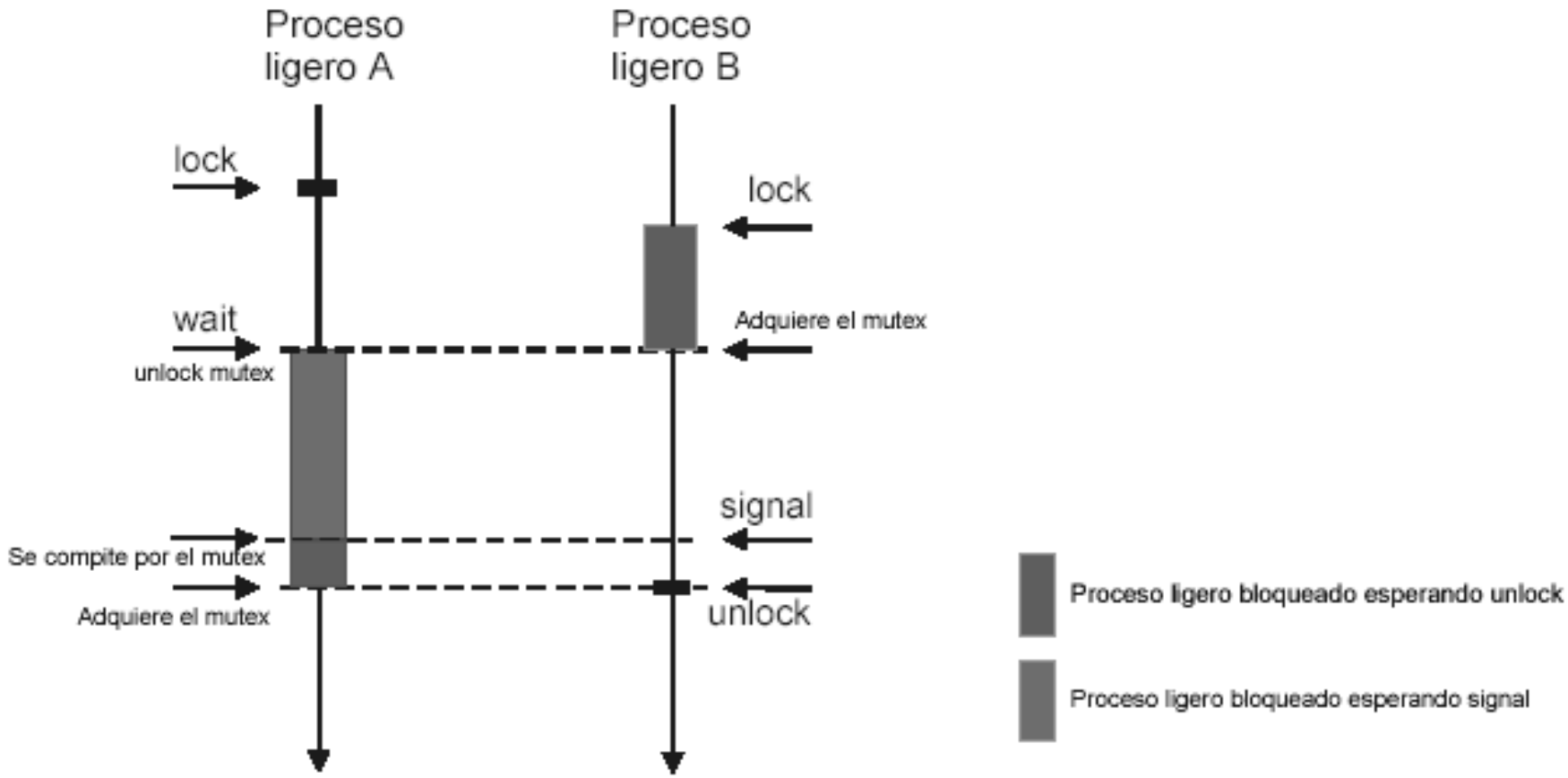
- La operación `unlock` debe realizarla el proceso ligero que ejecutó `lock`



Variables condicionales

- Variables de sincronizacion asociadas a un mutex que sirven para bloquear un proceso hasta que ocurra algun suceso.
- Conveniente ejecutarlas entre lock() y unlock()
- Dos operaciones atomicas
  - c_wait: bloquea al proceso que ejecuta la llamada y le expulsa del mutex, permitiendo que otro proceso lo adquiera.
  - c_signal: desbloquea a uno o varios procesos suspendidos en una variable condicional. El proceso que despierta compite de nuevo por el mutex.

Variables condicionales



Mutex POSIX

- Mutex
  - Inicializacion, destruccion:
    - int pthread_mutex_init (pthread_mutex_t* mutex,
      pthread_mutexattr_t* attr)
    - int pthread_mutex_destroy (pthread_mutex_t* mutex)
- Bloqueo, desbloqueo
    - int pthread_mutex_lock(pthread_mutex_t* mutex)
    - int pthread_mutex_unlock(pthread_mutex_t* mutex)

Variables condicionales POSIX

- Inicialización, destrucción
  - int pthread_cond_init (pthread_cond_t* cond,
    pthread_condattr_t* attr);
  - int pthread_cond_destroy (pthread_cond_t* cond);
- Signal, wait
  - int pthread_cond_signal (pthread_cond_t* cond)
    - Sin efecto si no hay procesos esperando en wait de la variable condicional “cond”
    - Desbloquea procesos en wait de la variable “cond”
  - int pthread_cond_broadcast (pthread_cond_t* cond)
  - int pthread_cond_wait(pthread_cond_t* cond,
    pthread_mutex_t* mutex);
    - Suspende al proceso hasta que otro hace un signal sobre “cond”
    - Libera el mutex.

Productor consumidor con mutex

```
Productor()
{
    int pos=0;
    for(num_datos)
    {
        //producir un dato
        lock(mutex);
        while(num_elem==TAM_BUFFER)
            c_wait(lleno,mutex);
        buffer[pos]=dato;
        pos=(pos+1)%TAM_BUFFER;
        num_elem++;
        if(num_elem==1)
            c_signal(vacio);
        unlock(mutex);
    }
}
```

```
Consumidor()
{
    int pos=0;
    for(num_datos)
    {
        lock(mutex);
        while(num_elem==0)
            c_wait(vacio,mutex);
        dato=buffer[pos];
        pos=(pos+1)%TAM_BUFFER;
        num_elem--;
        if(num_elem==TAM_BUFFER-1)
            c_signal(lleno);
        unlock(mutex);
    }
}
```

Lectores-escritores con mutex

```
Lector()
{
    lock(mutex_lectores);
    n_lectores++;
    if(n_lectores==1)
        lock(mutex_recurso);
    unlock(mutex_lectores);

    //Consulta recurso compartido

    lock(mutex_lectores);
    n_lectores--;
    if(n_lectores==0)
        unlock(mutex_recurso);
    unlock(mutex_lectores);
}
```

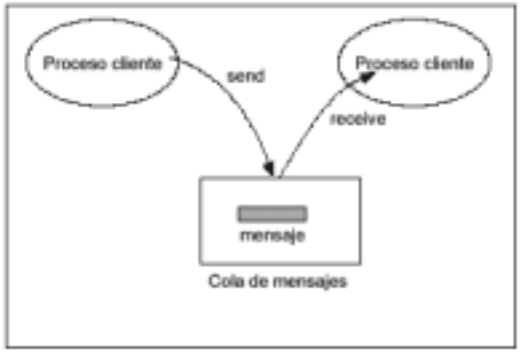
```
Escritor()
{
    lock(mutex_recurso);

    //Consulta recurso compartido

    unlock(mutex_recurso);
}
```

Mensajes

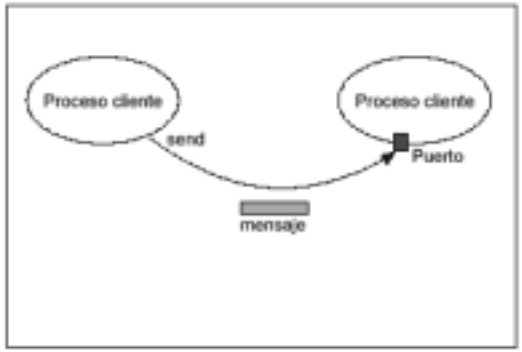
- Paso de mensajes permite resolver:
  - Distintas maquinas (aunque tb misma maquina)
- Primitivas
  - send(destino, mensaje)
  - receive(origen, mensaje)
- Caracterisiticas de los mensajes
  - Tamaño
  - Flujo de datos (uni o bidireccional)
  - Nombrado
    - Directa
      - send(P, mensaje)
      - receive(Q, mensaje) (Q=ANY)
    - Indirecta
  - Exclusion mutua
  - Sincronizacion
  - Almacenamiento

Mensajes

- Nombrado indirecto
  - Colas de mensajes
    - Puede haber multiples emisores y receptores
  - Puertos
    - Asociado a un proceso (receptor)
  - send(P, mensaje)
  - receive(P, mensaje)



Comunicación con colas de mensajes



Comunicación con puertos

Mensajes

- Sincronizacion
  - Sincrona: el emisor debe estar ejecutando “send” y el receptor “receive” simultaneamente
  - 3 combinaciones
    - Envio y recepcion bloqueante: Sincrona=“cita”
    - Envio no bloqueante, recepcion bloqueante: combinacion mas utilizada.
    - Envio y recepcion no bloqueante: totalmente asincrona, ningun proceso tiene que esperar al otro

Mensajes

- Almacenamiento
  - Sin capacidad: la comunicación debe ser sincrona
  - Con capacidad: espacio limitado para almacenar mensajes (cuando se satura, el emisor debe bloquearse)

Seccion critica con mensajes

```
//crear la cola
send(cola, testigo); //insertar un testigo

receive(cola, testigo);
//seccion critica
send(cola, testigo);
```

Colas mensajes POSIX

mqd_t mq_open(char* name, int flag, mode_t mode, struct
mq_attr* attr);
mqd_t mq_open(char* name, int flag);

Flag= O_CREAT
O_RDONLY, O_RDWR, O_WRONLY
O_NONBLOCK
Mode = 0777
mq_atrr
- mq_msgsize (tamaño maximo de un mensaje)
- mq_maxmsg (maximo numero de mensajes)

mq_close(mqd_t mqd)
mq_unlink(char* name)

Colas de mensajes POSIX

int mq_send (mqd_t q,char* msg, size_t len, int prio);

Si O_NONBLOCK y cola llena, devuelve -1

int mq_receive (mqd_t q,char* msg, size_t len, int* prio);

int mq_setattr(mqd_t q , struct mq_attr* qattr , struct mq_attr* old_qattr);

int mq_getattr(mqd_t q, struct mq_attr* qattr);

Seccion critica con colas de mensajes

```
void main(void)
{
    mqd_t mutex;
    struct mq_attr attr;
    char c;
    attr.mq_maxmsg=1;
    atrr.mq_msgsize=1;

    mutex=mq_open("MUTEX",O_CREAT|ORDWR,0777,&attr);
    mq_send(mutex,&c,1,0);
    if(fork()==0) //hijo
    {
        while(1)
        {
            mq_receive(mutex,&c,1,0);
            //seccion critica
            mq_send(mutex,&c,1,0);
        }
    }
    else //padre
    {
        while(1)
        {
            mq_receive(mutex,&c,1,0);
            //seccion critica
            mq_send(mutex,&c,1,0);
        }
    }
}
```

Productor consumidor con mensajes

```
#define MAX_BUFFER 100
#define NUM_DATOS 10000
mqd_t almacen;

void main(void)
{
	struct mq_attr attr;
	attr.mq_maxmsg=MAX_BUFFER;
	atrr.mq_msgsize=sizeof(int);

	mutex=mq_open(“ALMACEN”,O_CREAT|ORDWR,0777,&attr);
	if(fork()==0) //hijo
	{
		Productor();
	}
	else //padre
	{
		Consumidor();
	}
}
```

Productor con mensajes

```
void Productor(void)
{
   int dato,i;
   for(i=0;i<NUM_DATOS,ì++)
   {
        dato=i;//producir dato
        mq_send(almacen,&dato,sizeof(int));
   }
}
```

Consumidor con mensajes

```
void Consumidor(void)
{
   int dato,i;
   for(i=0;i<NUM_DATOS,ì++)
   {
        mq_receive(almacen, &dato,sizeof(int));
        //usar dato
   }
}
```