



SISTEMAS OPERATIVOS SINCRONIZACIÓN Y COMUNICACIÓN ENTRE PROCESOS

Francisco Rosales
<frosal@fi.upm.es>

- **Concurrencia, compartición y coordinación**
- **Mecanismos de sincronización y modelos clásicos**
- **Mecanismos de comunicación local y remota**
- **Interbloqueos**

Siglas y definiciones:

PL	Proceso Ligero o <i>thread</i>
PP	Proceso Pesado
MC	Memoria Compartida
BD	Base de Datos
sii	Si y sólo si
<i>Thread</i>	Flujo de ejecución de un proceso pesado o ligero



CONCURRENCIA

Sucede a muchos niveles:

Concurrencia **hardware**

- Unidades funcionales del procesador
 - ◆ Superescalar
 - ◆ HyperThreading
- Procesador y dispositivos E/S
- Multiprocesador =
N procesadores
+ memoria compartida
 - ◆ Número de Cores
- Multicomputador =
N nodos de cómputo
+ red de interconexión

micro

macro

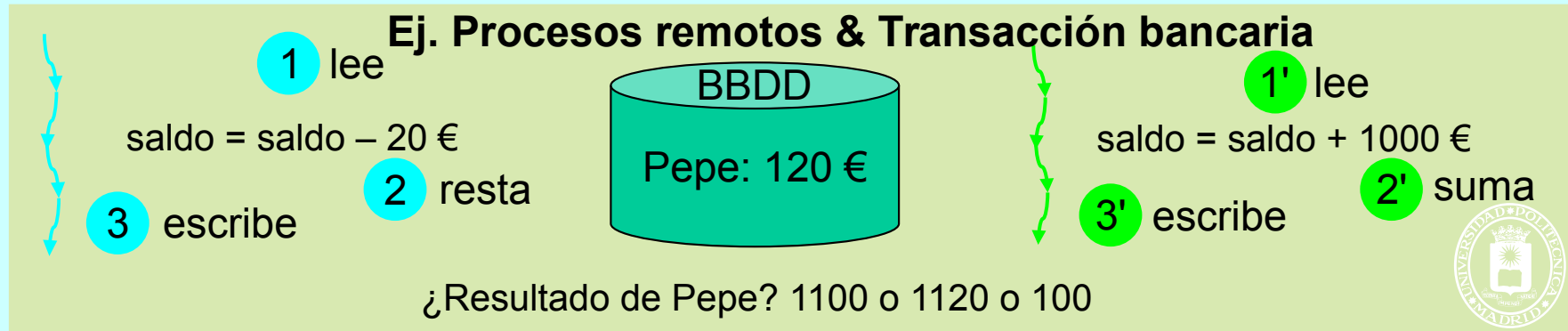
Concurrencia **software**

- Múltiples threads en un proceso (procesos ligeros)
- Multiprogramación de un procesador
- Aplicación paralela

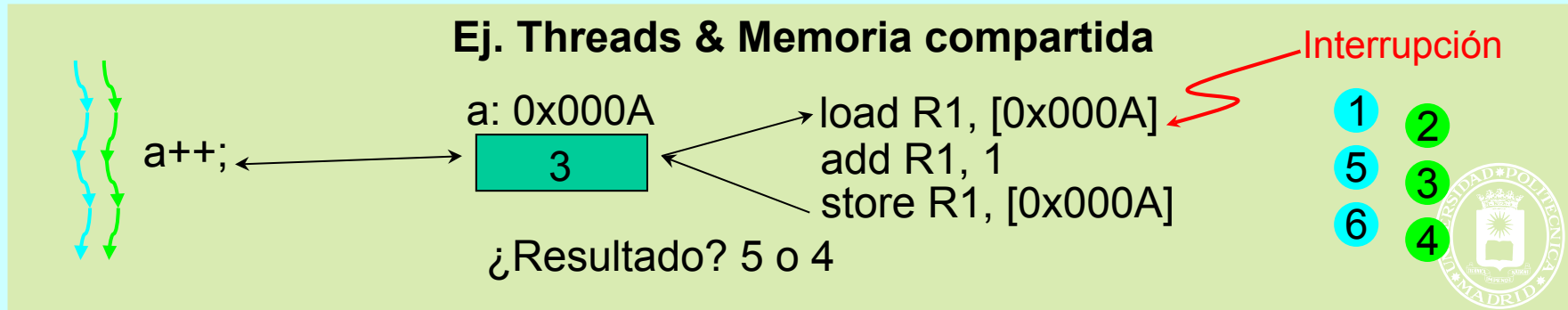
El acceso concurrente SIN coordinación a recursos compartidos puede NO producir los resultados esperados

Porque...

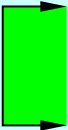
- NO se puede (ni debe) garantizar la velocidad relativa de los procesos concurrentes (== el orden en que ejecutan)

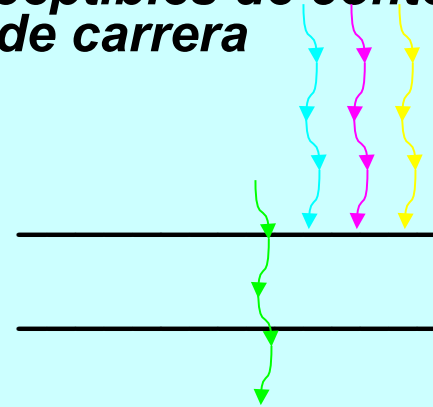


- La sentencia de alto nivel más simple requiere varias instrucciones máquina



Segmentos de código susceptibles de contener una condición de carrera

Ej. `int a; /* compartida */`
 `<<Entrada a sección crítica>>`
`a = a + 1;`
`<<Salida de sección crítica>>`



- Protegeremos la sección crítica con mecanismos de sincronización que deberán garantizar:
 - Exclusión mutua: sólo entra un thread al tiempo. Esto implica que los threads deben esperar a que esté libre la sección crítica
 - Progreso: un thread fuera no impide que otro entre
 - Espera acotada: si espero, entraré en tiempo finito
- Los mecanismos de sincronización detienen la ejecución produciendo contención. Al programar deberemos:
 - Retener los recursos el menor tiempo posible (menor contención posible)
 - No quedar bloqueados dentro de la sección crítica
 - En definitiva: coordinar el acceso a recursos compartidos

■ **Implícita:** la realiza el SO (gestor de recursos)

- Procesos independientes
- Compiten por recursos (pero lo ignoran)
- Control de acceso transparente y asociado al recurso

Ej. Procesos pesados (PPs) y... ¡¡cualquier recurso!! hw o sw

Ej. PPs y coutilización de fichero ⇒ el fichero no se corrompe pero ¡la información contenida puede no ser coherente!

■ **Explícita:** la realizan los procesos

- Procesos cooperantes
- Comparten recursos (son conscientes de ello)
- Coordinan el acceso al recurso, y para ello...
- Usan mecanismos de sincronización y/o comunicación proporcionados por: el SO, una biblioteca o el lenguaje

Ej. Procesos ligeros (PLs) y variables en memoria compartida

Ej. PPs y coutilización de fichero como BD ⇒ fichero OK e información contenida coherente

Ej. Procesos distribuidos colaborando para jugar al mus en red

***Permiten resolver la sección crítica
(== coordinar el acceso a un recurso compartido)***

- **Para procesos fuertemente acoplados (== que comparten memoria)**
 - **Semáforos (adecuado para PPs con MC)**
 - **Mutex y Condiciones (adecuado para PLs)**
 - **Otros propios de lenguajes concurrentes (Monitores)**

...se suelen poner a prueba aplicados a “modelos clásicos”

- **Para procesos independientes**
 - **Semáforos, sii son servicio del SO**
 - **Cerrosos sobre (regiones de) fichero**
- **La sincronización conlleva espera. En general, esta espera será pasiva (sin consumir ciclos de procesador)**

- **Definir la información que se comparte**
 - Buffer circular
 - Registros con campos de tamaño fijo
 - Registros con campos de tamaño variable
 - ...
- **Definir el soporte de la información compartida**
 - Variables comunes (threads) . Atomicidad: load, store, test & set
 - Región de memoria . Atomicidad: load, store, test & set
 - Fichero. Atomicidad: operación E/S
- **Definir la información de control (que también se comparte) y las condiciones de acceso.**
 - Contadores, etc.
- **Seleccionar los mecanismos de control**
 - Tipo de mecanismo (semáforo, mutex y cond., cerrojo, etc.)
 - Instancias de esos mecanismos

■ Productor - consumidor

- Almacenamiento compartido típico: Buffer circular (huecos tamaño fijo)
- Variables de control típicas:
 - N° de huecos libres
 - Punteros posición

■ Lectores y escritores

- Almacenamiento compartido típico: Conjunto de registros
- Granularidad del control
 - Todo el almacenamiento \Rightarrow Contención
 - Cada registro individual \Rightarrow Complejidad
- Variables de control típicas:
 - N° de lectores
 - N° de escritores (solamente puede haber uno)



SEMÁFORO MECANISMO DE SINCRONIZACIÓN

Objeto compartido con:

- Campo entero contador de valor inicial dado, que significa:

sii > 0 nº de veces disponible (\approx nº plazas)

sii ≤ 0 nº de threads esperando disponibilidad

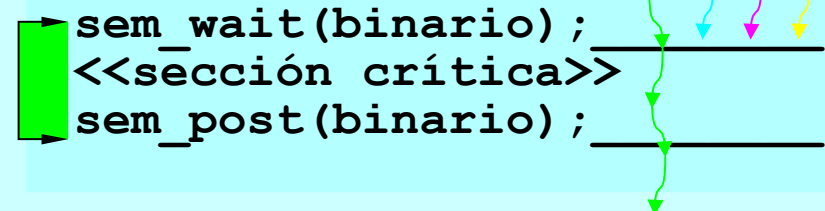
Información de control

- Dos acciones atómicas

```
sem_wait(s)    == down
{
    <<decrementar contador>>;
    if (<<contador>> < 0)
        <<esperar en s>>
}
```

```
sem_post(s)    == up
{
    <<incrementar contador>>
    if (<<contador>> ≤ 0)
        <<que continúe uno>>
}
```

- Semáforo binario == máximo valor del contador es 1
- La "sección crítica" con semáforo:



- SIN posesión ⇒ se puede hacer post sin hacer wait antes



```
int sem_init (sem_t *sem, int shared, int valini);
```

```
int sem_destroy (sem_t *sem);
```

- Inicializar y destruir un semáforo SIN nombre

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

```
int sem_close (sem_t *sem);
```

```
int sem_unlink (char *name);
```

- Abrir (o crear), cerrar y borrar un semáforo CON nombre

```
int sem_wait (sem_t *sem);
```

- Decrementar el contador del semáforo (competir por pasar)

```
int sem_post (sem_t *sem);
```

- Incrementar el contador del semáforo (salir)

El semáforo es un mecanismo diseñado originalmente para PPs. Para PLs pueden ser demasiado pesados.



MUTEX Y CONDICIONES MECANISMO DE SINCRONIZACIÓN

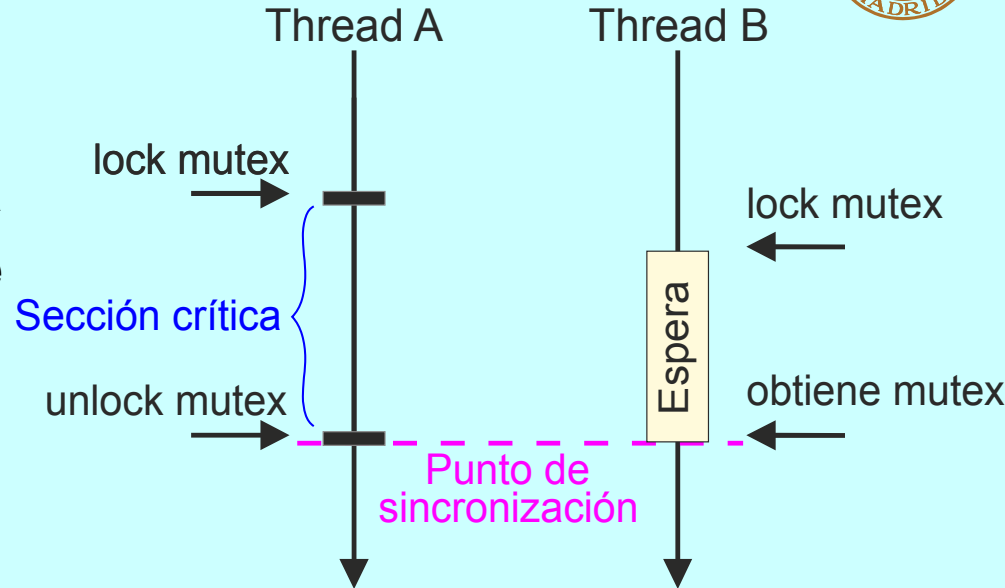
Mutual Exclusion

- == Semáforo binario **sin memoria**
 - Cerradura cerrada, llave puesta
 - Abro y cierro y me llevo la llave

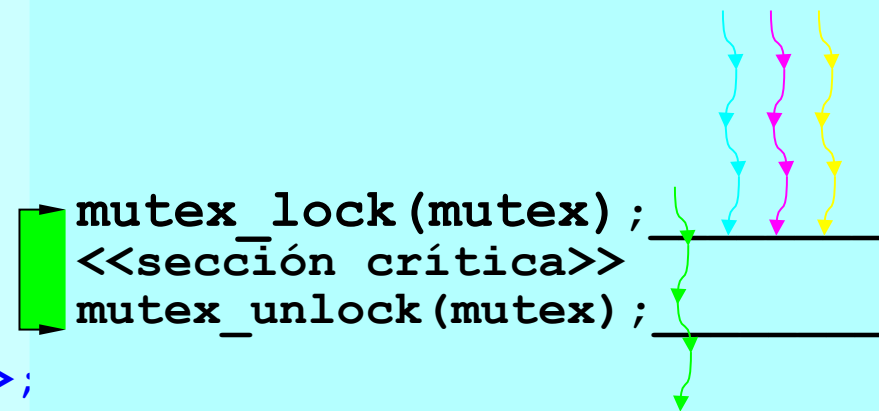
- Dos acciones atómicas

```
mutex_lock(m)
{
    if (<<no hay llave>>)
        <<esperar llave>>;
    <abrir cerrar y llevármela>>;
}

mutex_unlock(m)
{
    if (<<alguien espera>>)
        <<entregar llave>>;
    else
        <<devolver llave a cerradura>>;
}
```



- La "sección crítica" con mutex:



- CON posesión ⇒ no se puede hacer unlock sin haber hecho lock antes

- Permiten liberar un mutex sin salir de la sección crítica
- Tres acciones atómicas

```
condition_wait(c,m)
{
    mutex_unlock(m);
    <<esperar aviso>>;
    mutex_lock(m);
}
condition_signal(c)
{
    if (<<alguien espera>>)
        <<avisar que siga>>;
    else (<<se pierde>>)
}
condition_broadcast(c)
{
    while(<<alguien espera>>)
        <<avisar que siga>>;
}
```

A veces el thread que está (o posee) la sección crítica no puede continuar, porque "no se da" cierta condición que sólo podría cambiar otro thread desde dentro de la sección crítica.

- Es preciso pues:
 - Liberar **temporalmente** el mutex que protege la sección crítica...
 - ...mientras se espera a que la condición se dé.
- Todo ello:
 - Sin abandonar la sección crítica...
 - ...y de forma atómica.

La combinación mutex-condición muestra la lógica del problema.

!Es realmente cómoda de usar!

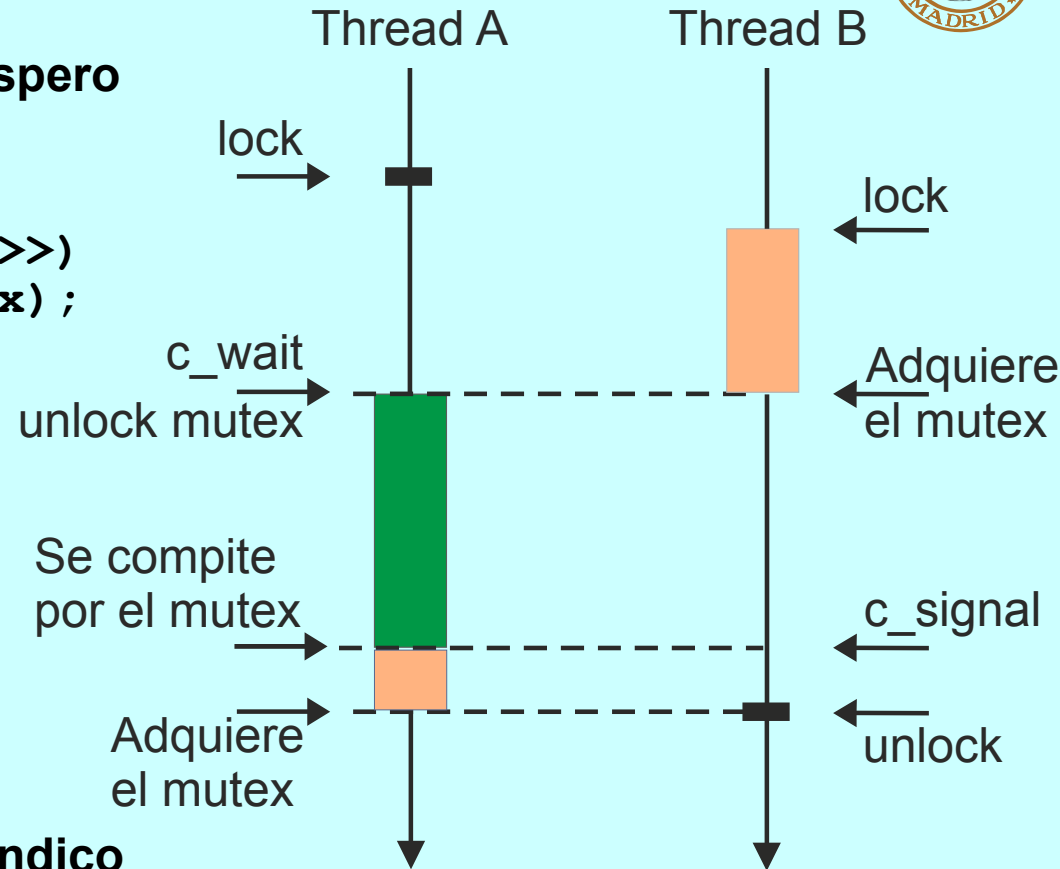
■ A: Mientras no pueda seguir, espero

```
mutex_lock(mutex);  
<<sección crítica...>>  
while (<<no puedo continuar>>)  
    condition_wait(cond, mutex);  
<<...sección crítica>>  
mutex_unlock(mutex);
```

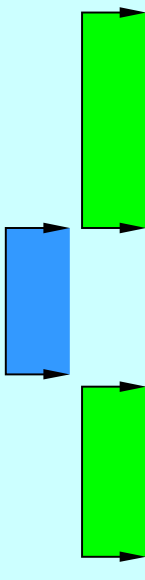
■ Fundamental utilizar **while**

■ B: Si alguien podría seguir, lo indico

```
mutex_lock(mutex);  
<<sección crítica...>>  
if (<<se podría continuar>>)  
    condition_signal(cond);  
<<...sección crítica>>  
mutex_unlock(mutex);
```



Solución general



```
mutex_lock(mutex);  
while (<<no puedo continuar (condición variables control)>>)  
    condition_wait(cond, mutex);  
<<modifica variables de control>>  
mutex_unlock(mutex);  
<<...sección crítica del problema>>  
mutex_lock(mutex);  
<<modifica variables de control>>  
condition_signal(cond);  
mutex_unlock(mutex);
```



```
int pthread_mutex_init (pthread_mutex_t *mutex,  
                        pthread_mutexattr_t *attr)
```

- Inicializar un mutex con atributos attr. Existen diversas llamadas para modificar los atributos, con NULL se especifican atributos por defecto

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- Borrar (destruir) un mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

- Competir por tomar el mutex

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

- Devolver el mutex (salir)

Los mutex y las condiciones son
un mecanismo idóneo para PLs



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

- Inicializar una variable condicional con atributos attr. Existen diversas llamadas para modificar los atributos (NULL == valores por defecto)

```
int pthread_cond_destroy (pthread_cond_t *cond) ;
```

- Borrar (destruir) una variable condicional

```
int pthread_cond_wait (pthread_cond_t *cond,  
                      pthread_mutex_t *mutex) ;
```

- Sin salir de la sección crítica, libera temporalmente el mutex que la protege, para esperar a que se "señale" la condición


```
int pthread_cond_signal (pthread_cond_t *cond) ;
```

- Indicar (señalar) un cambio que permitiría continuar a uno de los threads que esperan en la condición

```
int pthread_cond_broadcast (pthread_cond_t *cond) ;
```

- Indicar (señalar) un cambio que permitiría continuar a todos los threads que esperan en la condición




PC_PLs_mc.c

```
#define BUFF_SIZE      1024
#define TOTAL_DATOS    100000

int n_datos;                                /* Datos en el buffer */
int buffer[BUFF_SIZE];                      /* buffer circular compartido */
pthread_mutex_t mutex;                       /* Acceso a sección crítica */
pthread_cond_t no_lleno, no_vacio;           /* Condiciones de espera */

int main(void)
{
    pthread_t th1, th2;
    pthread_mutex_init(&mutex, NULL);        /* Situación inicial */
    pthread_cond_init(&no_lleno, NULL);
    pthread_cond_init(&no_vacio, NULL);
    pthread_create(&th1, NULL, Productor, NULL); /* Arranque */
    pthread_create(&th2, NULL, Consumidor, NULL);
    pthread_join(th1, NULL);                  /* Esperar terminación */
    pthread_join(th2, NULL);
    pthread_mutex_destroy(&mutex);            /* Destruir */
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    return 0;

}                                             /* continúa... */
```



```
void Productor(void)
```

```
{  
    int i, dato;  
    for(i=0; i < TOTAL_DATOS; i++) {  
        <<Producir el dato>>  
        pthread_mutex_lock(&mutex);  
        while(n_datos == BUFF_SIZE) //cond. espera  
            pthread_cond_wait(&no_lleno, &mutex);  
        buffer[i % BUFF_SIZE] = dato;  
        n_datos++;  
        pthread_cond_signal(&no_vacio);  
        pthread_mutex_unlock(&mutex);  
    }  
}
```

PC_PLS_mc.c

```
void Consumidor(void)
```

```
{  
    int i, dato;  
    for(i=0; i < TOTAL_DATOS; i++) {  
        pthread_mutex_lock(&mutex);  
        while(n_datos == 0) //condición espera  
            pthread_cond_wait(&no_vacio, &mutex);  
        dato = buffer[i % BUFF_SIZE];  
        n_datos--;  
        pthread_cond_signal(&no_lleno);  
        pthread_mutex_unlock(&mutex);  
        <<Consumir el dato>>  
    }  
}
```

PC_PLS_mc.c



```
pthread_mutex_t mutex; /* Control de acceso */
pthread_cond_t a_leer, a_escribir; /* Condiciones de espera */
int leyendo, escribiendo; /* Estado del acceso */
int main(void)
{
    pthread_t th1, th2, th3, th4;
    pthread_mutex_init(&mutex, NULL); /* Situación inicial */
    pthread_cond_init(&a_leer, NULL);
    pthread_cond_init(&a_escribir, NULL);
    pthread_create(&th1, NULL, Lector, NULL); /* Arranque */
    pthread_create(&th2, NULL, Escritor, NULL);
    pthread_create(&th3, NULL, Lector, NULL);
    pthread_create(&th4, NULL, Escritor, NULL);
    pthread_join(th1, NULL); /* Esperar terminación */
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    pthread_join(th4, NULL);
    pthread_mutex_destroy(&mutex); /* Destruir */
    pthread_cond_destroy(&a_leer);
    pthread_cond_destroy(&a_escribir);
    return 0;
} /* continúa... */
```



```
void Lector(void)
{
    pthread_mutex_lock(&mutex);
    while(escribiendo != 0) //condición espera
        pthread_cond_wait(&a_leer, &mutex);
    leyendo++;
    pthread_mutex_unlock(&mutex);
```

<<Lecturas simultáneas del recurso compartido>>

```
    pthread_mutex_lock(&mutex);
    leyendo--;
    if (leyendo == 0)
        pthread_cond_signal(&a_escribir);
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
}
```




LE_Pls_mc.c

```
void Escritor(void)
{
    pthread_mutex_lock(&mutex);
    while(leyendo !=0 || escribiendo !=0) //cond. espera
        pthread_cond_wait(&a_escribir, &mutex);
    escribiendo++;
    pthread_mutex_unlock(&mutex);
```

<<Acceso en exclusiva al recurso compartido>>

```
    pthread_mutex_lock(&mutex);
    escribiendo--;
    pthread_cond_signal(&a_escribir);
    pthread_cond_broadcast(&a_leer);
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
}
```



Asignatura:	Sistemas Operativos	Concurrencia
Lenguaje prog.	C sobre POSIX	Java
Procesos ligeros	pthread	Thread
Procesos pesados	si	no
Semáforos	sem_t	java.util.concurrent.Semaphore
-----	sem_wait	.acquire
-----	sem_post	.release
Mutex	pthread_mutex_t	java.util.concurrent.locks.Lock
-----	pthread_mutex_lock	.lock
-----	pthread_mutex_unlock	.unlock
Conditions	pthread_cond_t	
java.util.concurrent.locks.Condition		
-----	pthread_cond_wait	.await
-----	pthread_cond_signal	.signal
-----	pthread_cond_broadcast	.signalAll



CERROJOS

Servicio del SO que permite coordinar la coutilización de ficheros

- Para procesos independientes
- Permite establecer cerrojos sobre regiones de fichero

Cerrojos de tipo:

- o Compartido
 - No puede solapar con otro exclusivo
 - Sólo se puede (debe) hacer read de la región
- o Exclusivo:
 - No puede solapar con ningún otro
 - Se puede hacer read y write

Regiones relativas a:

- principio, posición actual o final de fichero

Modalidad:

- o Consultivo (advisory): sólo afecta los procesos que los usan (preferible)
- u Obligatorio (mandatory): afectan a procesos que no los usan (¡peligroso!)

		Solapando con otro ya concedido de tipo:		
		Exclusivo	Compartido	Ninguno
El cerrojo pedido es de tipo:	Exclusivo	NO	NO	SI
	Compartido	NO	SI	SI

¿Se puede conceder?

Solapando con otro ya concedido de tipo:

Exclusivo

Compartido

Ninguno

El cerrojo pedido es de tipo:

Exclusivo

NO

NO

SI

Compartido

NO

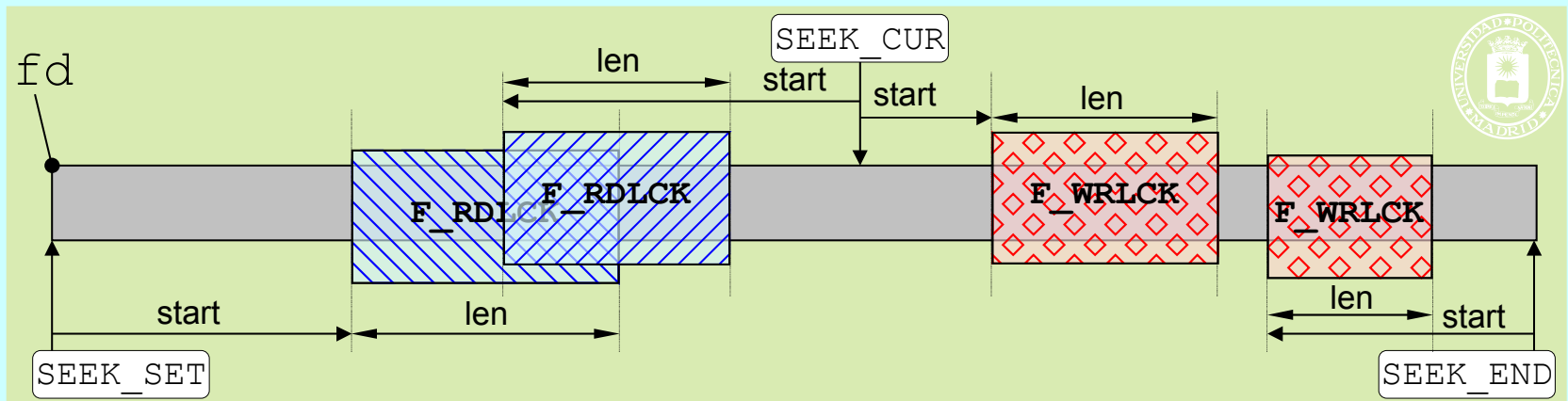
SI

SI

```
int fcntl(int fd, int cmd, struct flock *flockptr);
```

- **cmd:** F_GETLK, (comprueba si existe un cerrojo)
F_SETLK (establece cerrojo; no bloqueante, asíncrono)
F_SETLKW (establece cerrojo; bloqueante, síncrono)
- El cerrojo se establece sobre la región y con la modalidad indicada en:

```
struct flock {
    short l_type;      /* F_RDLCK (compartido), F_WRLCK
                       (exclusivo), F_UNLCK (elimina) */
    short l_whence;    /* SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;     /* Desfase relativo a l_whence */
    off_t l_len;       /* Tamaño. 0 == hasta EOF */
    pid_t l_pid;       /* Sólo F_GETLK, primer pid con lock*/
}
```



Lector

L_PPs_cf.c

```
int main(void)
{
    int fd, val, cnt;
    struct flock fl;
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0; Cierra todo el fichero
    fl.l_pid = getpid();
    fd = open("BD", O_RDONLY);
    for (cnt = 0; cnt < 10; cnt++)
    {
        fl.l_type = F_RDLCK; Compartido
        fcntl(fd, F_SETLKW, &fl); Sinc.
        lseek(fd, 0, SEEK_SET);
        read(fd, &val, sizeof(int));
        printf("%d\n", val);
        /*Lecturas simultáneas
        del recurso compartido*/
        fl.l_type = F_UNLCK;
        fcntl(fd, F_SETLK, &fl);
    }
    return 0;
}
```

Escritor

E_PPs_cf.c

```
int main(void)
{
    int fd, val, cnt;
    struct flock fl;
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0; Cierra todo el fichero
    fl.l_pid = getpid();
    fd = open("BD", O_RDWR);
    for (cnt = 0; cnt < 10; cnt++)
    {
        fl.l_type = F_WRLCK; Exclusivo
        fcntl(fd, F_SETLKW, &fl); Sinc.
        lseek(fd, 0, SEEK_SET);
        read(fd, &val, sizeof(int));
        val++; /*Acceso exclusivo*/
        lseek(fd, 0, SEEK_SET);
        write(fd, &val, sizeof(int));
        fl.l_type = F_UNLCK;
        fcntl(fd, F_SETLK, &fl);
    }
    return 0;
}
```

■ Cerrojos fcntl

- Los cerrojos se asocian al proceso
- Si el proceso cierra un descriptor cualquiera del fichero (que puede tener más de uno) se pierden los cerrojos
- Los cerrojos no se heredan
- No funcionan en sistemas de ficheros en red, como NFS

■ A título informativo indicamos que Linux permite cerrojos obligatorios

➤ En el sistema de ficheros

- Al montar el sistema de ficheros hay que habilitarlos:
`mount -o mand`

➤ En el fichero

- Hay que deshabilitar los permisos de grupo y habilitar los permisos: `set-group-ID (SGID)`



MECANISMOS DE COMUNICACIÓN

Mecanismo de comunicación = Recurso compartido + Mecanismo de sincronización

- **Para procesos fuertemente acoplados (== que comparten memoria)**
 - Memoria compartida + Semáforos (para PPs)
 - Imagen de memoria única + Mutex y Condiciones (para PLs)
 - Etc. ¡Ya los hemos visto!

- **Para procesos pesados**
 - PIPEs y FIFOs
 - Colas de mensajes
 - Fichero + cualquier mecanismo de sincronización

- **Para procesos remotos (o locales)**
 - Sockets

- **Dirección:** Identifica al receptor o al emisor (remite)
- **Tipo**
 - Sin nombre: Se hereda el identificador del creador del mecanismo
 - Nombre textual o simbólico (/home/jfelipe/mififo, www.fi.upm.es)
 - Nombre físico (fd = 5, IP = 138.100.8.100 TCP = 80)
 - Estructura en árbol
- **Ámbito**
 - Procesos emparentados (creador y sus descendientes)
 - Local: Misma máquina
 - Remoto: Máquinas distintas
- **Servidor de nombres**
 - Convierte el nombre textual en físico
 - Local: Servidor de nombres del servidor de ficheros
 - Remoto: Servidor DNS de Internet

Al crearlo (o abrirlo)

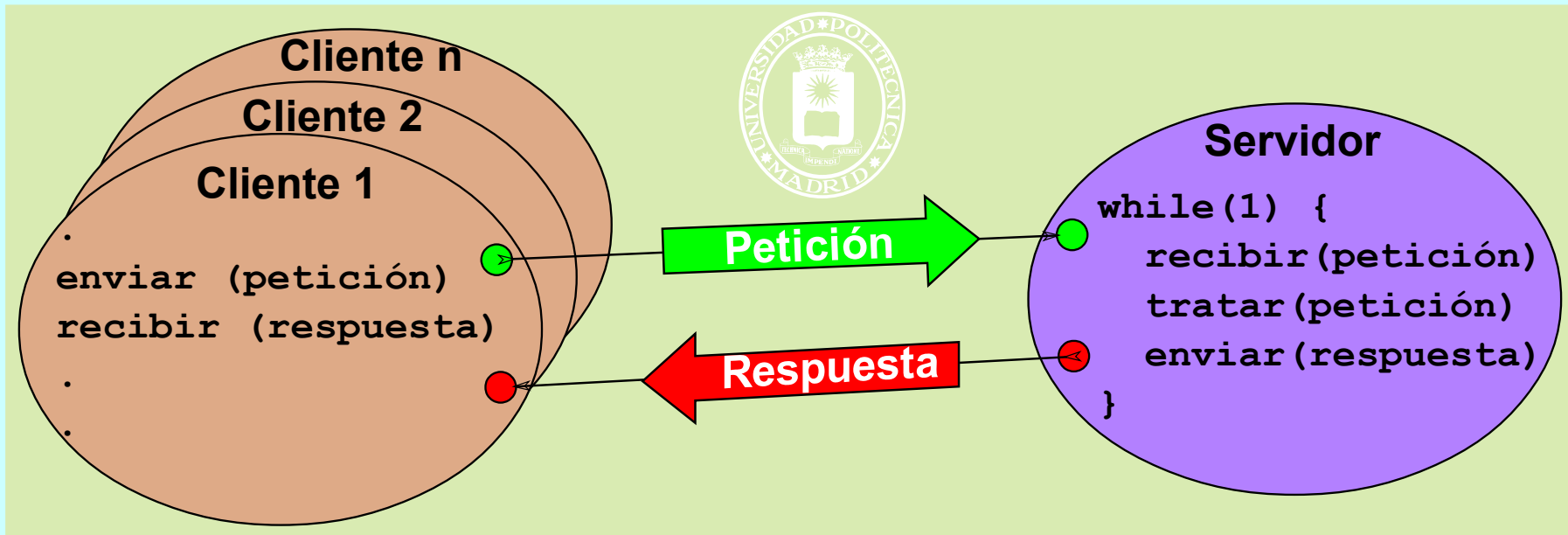
- Según la forma de nombrarlo
 - Sin nombre (Ej. PIPE)
 - Con nombre local (Ej. FIFO)
 - Con nombre de red (Ej. *Socket*)
- Según el identificador devuelto
 - Descriptor de fichero
(sii es servicio ofrecido por el SO)
 - Identificador propio
(sii es ofrecido por biblioteca)

Al usarlo (a través de su ID)

- Según el flujo de datos
 - Unidireccional
 - Bidireccional
- Según capacidad de memoria
 - Sin *buffering*
 - Con *buffering*
- Según bloquee o no
 - Síncrono (bloqueante)
 - Asíncrono (no bloqueante)

Modela acceso a gestor (local o remoto) de recurso

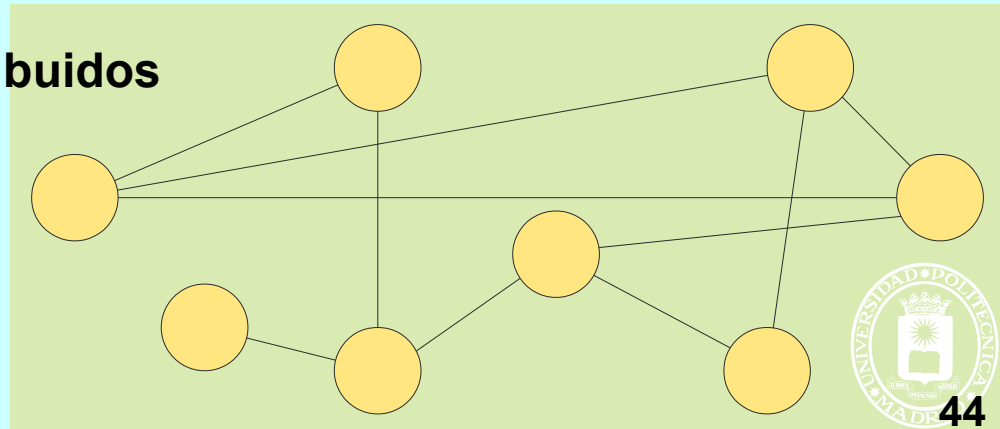
- Protocolo Petición-Respuesta para acceso a gestor de recurso
- Servidor == proveedor de servicio == gestor del recurso
- Por extensión, también se dice servidor a la máquina donde éste reside
- Cliente: Accede al proveedor siguiendo el protocolo
- Comunicación N (clientes) a 1 (servidor) o N a M



- El modelo Cliente-Servidor clásico es centralizado
- En el modelo P2P (“entre pares” o “entre iguales”):
 - Cada miembro es, a la vez, cliente y servidor
 - La información o recurso está distribuido entre los miembros
 - Red no estructurada: cada miembro tiene lo que quiere
 - Red estructurada: la información o recurso está repartido según determinados criterios
 - La gestión del directorio y del enrutamiento puede estar:
 - Centralizado (Napster y Audiogalaxy)
 - Totalmente distribuido (Ares Galaxy, Gnutella, Freenet y Kademia)
 - Híbrido (Bittorrent, eDonkey2000 y Direct Connect)

■ Características del P2P

- | | |
|-------------------|-----------------------|
| ➤ Escalable | ➤ Costes distribuidos |
| ➤ Robusto | ➤ Anonimato |
| ➤ Descentralizado | ➤ Seguridad |

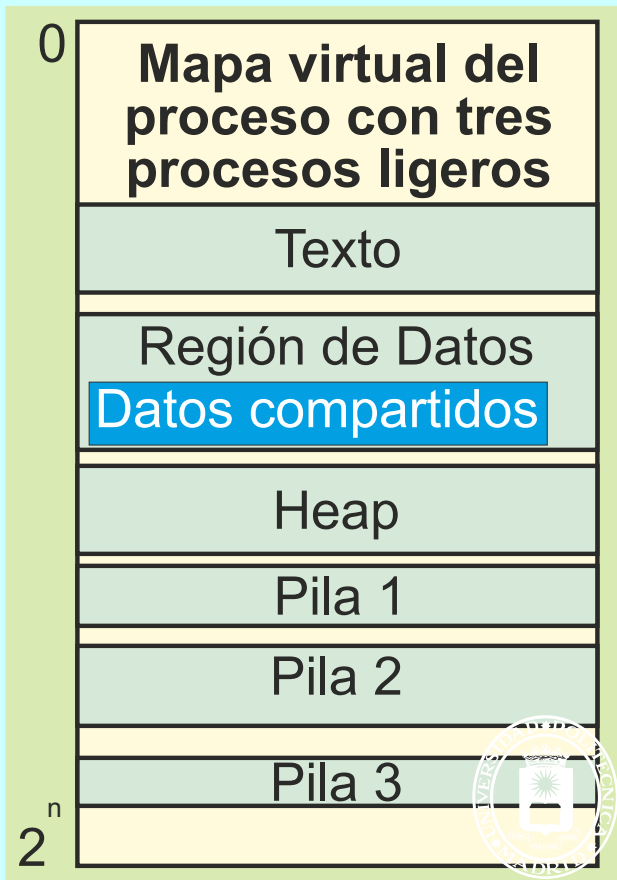




MECANISMOS DE COMUNICACIÓN LOCAL

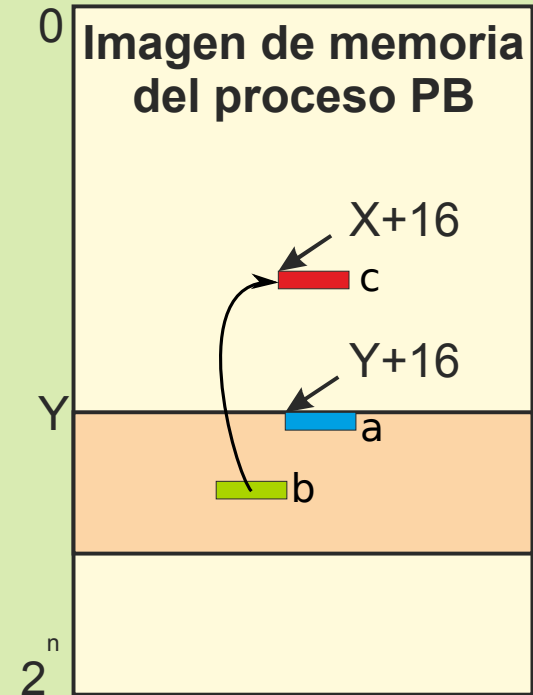
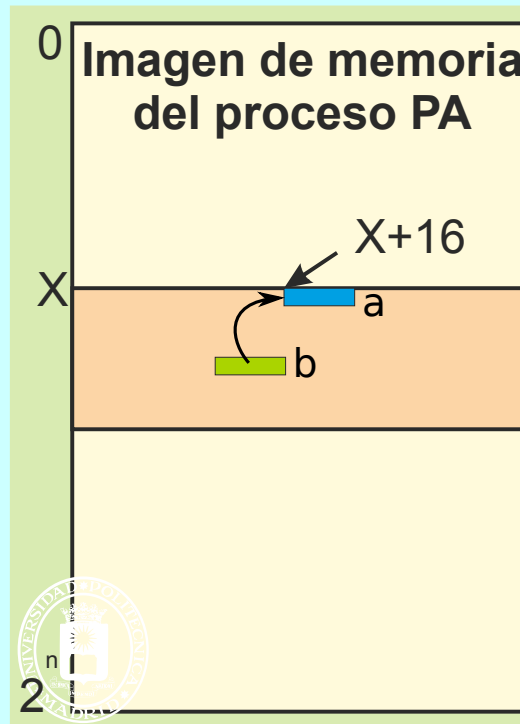
■ Procesos ligeros

- Todo su espacio de memoria está compartido:
 - Variables globales, variables estáticas, etc.
- Cada uno "usa" su propia pila



■ Procesos pesados con memoria compartida

- Regiones compartidas (mmap)
- Declaración independiente de variables proyectadas en MC



Mecanismo del SO que da resuelto el esquema de comunicación "Productores-Consumidores" entre procesos pesados emparentados

- Sin nombre (sólo visible por quien lo crea y procesos derivados de éste)
- Unidireccional
- Con buffering

```
int pipe (int fds[2]) ;
```

- Devuelve dos descriptores de fichero (uno por extremo del pipe)

```
read (fds[0], datos, n) ;
```

- Si vacío \Rightarrow se bloquea el lector
- Si con p bytes \Rightarrow
 - Si $p \geq n$ devuelve n
 - Si $p < n$ devuelve p
- Si vacío y no hay escritores \Rightarrow devuelve 0 (indicando EOF)

```
write (fds[1], datos, n) ;
```

- Si lleno \Rightarrow se bloquea el escritor
- Si no hay lectores \Rightarrow el escritor recibe la señal SIGPIPE

Las lecturas y escrituras pequeñas son atómicas (no se mezclan)

*Mecanismo del SO
para comunicar
procesos pesados no remotos*

- Se crean como FIFO
- Se usan como fichero
- Se comportan como PIPE

```
int mkfifo (char *name,  
            mode_t mode) ;
```

- Crea un FIFO con nombre y permisos iniciales dados

```
int unlink(char *name) ;
```

- Elimina un FIFO

```
int open (char *name,  
          int flags) ;
```

- Abrir para lectura, escritura o ambas
- Bloquea hasta que se abran los dos extremos

```
read (fd_in,datos,n) ;
```

```
write (fd_out,datos,n) ;
```

- Semántica PIPE

Crear FIFO

m_k_FIFO.c

```
int main(int argc, char *argv[])
{
    if (mkfifo(argv[1], 0666) == 0)
        return 0;
    perror(argv[0]);
    return 1;
}
```

Crea el FIFO

Productor

P_PPs_FIFO.c

```
int main(int argc, char *argv[])
{
    int fd;
    char ch;
    fd = creat(argv[1], 0666);
    if (fd == -1) {
        perror(argv[0]);
        return 1;
    }
    while(read(0, &ch, 1) == 1)
        write(fd, &ch, 1);
    return 0;
}
```

Abre el FIFO para escritura

Consumidor

C_PPs_FIFO.c

```
int main(int argc, char *argv[])
{
    int fd;
    char ch;
    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror(argv[0]);
        return 1;
    }
    while(read(fd, &ch, 1) == 1)
        write(1, &ch, 1);
    return 0;
}
```

Abre el FIFO para lectura

Ej. Ejecución de un mandato equivalente a: `ls -l | sort`
a través de un FIFO

```
$ mkfifo FIFO
$ ls -l FIFO
prw-r--r-- 1 frosal frosal      0 Apr 28 21:11 FIFO
$ sort < FIFO &
[135]
$ ls -l > FIFO
-rw-r----- 1 frosal frosal      0 Apr 29 00:08 Archivo
brw-rw---- 2 frosal disk    3,65 Sep 27  2000 Bloques
crw----- 2 root  root      4,1 Apr  8 13:02 Caracteres
drwxr-x--- 2 frosal frosal 4096 Apr 29 00:08 Directorio
lrwxrwxrwx 1 frosal frosal      7 Apr 29 00:10 Enlace -> Archivo
prw-r--r-- 1 frosal frosal      0 Apr 28 21:11 FIFO
srwxrwxrwx 2 root  gdm        0 Apr  8 13:02 Socket
$
```

¿Qué pasa con el FIFO del ejemplo?



MECANISMO DE COMUNICACIÓN REMOTA *SOCKET*



- Hay arquitecturas de computador que representan los números enteros de manera distinta: Little-endian vs. Big-endian
- Para transmitir enteros, hay que convertirlos a/de formato independiente

htonl()

htons()

- Convierten enteros largos y cortos de formato del host a formato de red

ntohl()

ntohs()

- Convierten enteros largos y cortos de formato del red a formato del host
- Por ejemplo, el campo `sockaddr_in.sin_port` (número de puerto)
- debe estar en formato red
- La comunicación de datos de otros tipos básicos (Ej. coma flotante) o estructuras exige otros métodos de conversión que no son estándar
- Por simplicidad, muchos protocolos usan texto como formato independiente y transmiten otros datos sin formato específico en binario (8 bits)

Mecanismo del SO para comunicación dentro del mismo dominio

- Con nombre (dirección)
- Bidireccional
- Con buffering
- Bloqueante o no

```
int socket (int dominio,  
            int tipo, int protocolo);
```

- Crea un **socket** (sin dirección) del dominio, tipo y protocolo dados y devuelve descriptor asociado **sd**
- **Dominio** == familia de direcciones
 - **AF_UNIX**: **intra-máquina**
(Dir. = nombre de fichero)
 - **AF_INET**: **entre máquinas**
(Dir. = dirección IP + nº de puerto)
 - Mismos servicios para todo dominio, pero diferente tipo de direcciones

■ Tipo

➤ Stream (**SOCK_STREAM**)

- Orientado a flujo de datos
- CON conexión
- Fiable: asegura entrega y orden
- [≈ Conversación telefónica]

➤ Datagrama (**SOCK_DGRAM**)

- Orientado a mensajes
- SIN conexión
- No fiable: pérdida y desorden
- [≈ Correspondencia postal]

■ Protocolo

➤ Mensajes y reglas de intercambio entre comunicantes

➤ En **AF_INET** (== Internet) existen dos protocolos de transporte:

- **IPPROTO_TCP**, para stream
- **IPPROTO_UDP**, para datagrama

- Las direcciones dependen del dominio, pero los servicios no (uso de cast)
 - `struct sockaddr`
estructura genérica de dirección
 - `struct sockaddr_in`
estructura específica `AF_INET`
 - Debe iniciarse a 0 (`bzero`)
 - `sin_family`: `AF_INET`
 - `sin_addr`: dirección del host (32 bits) (4 octetos [0..255])
 - `sin_port`: número de puerto (16 bits) (1024 reservados)
- Para los usuarios son texto
"138.100.8.100" ó "laurel.datsi.fi.upm.es"
- Una transmisión IP está caracterizada por cinco parámetros:
 - Protocolo (UDP o TCP)
 - Dirección host + puerto origen
 - Dirección host + puerto destino
- Cada socket debe estar asociado a:
 - una dirección local única
 - y una dirección remota sii:
 - está conectado
 - o para cada mensaje

`int inet_pton(int af, const char *src, void *dst);`
Conversión a binario (formato red IPv4 o IPv6) desde decimal-punto

`struct hostent *gethostbyname (char *str);`
Conversión a binario (formato red) desde dominio-punto



```
int socket (int dominio, int tipo, int protocolo);
```

- Crea un socket (sin dirección) del dominio, tipo y protocolo dados

```
int bind (int sd, struct sockaddr *dir, int tam);
```

- Asociar a una dirección local

```
int connect (int sd, struct sockaddr *dir, int tam);
```

- Asociar a una dirección remota (cliente)

```
int listen (int sd, int backlog);
```

- Prepara para aceptar conexiones (servidor)

```
int accept (int sd, struct sockaddr *dir, int *tam);
```

- Aceptación de una conexión (servidor)

```
int send (int sd, char *mem, int tam, int flags);
```

```
int recv (int sd, char *mem, int tam, int flags);
```

- Transmisión para conectados (también read y write)

```
int sendto (int sd, char *mem, int tam, int flags,  
            struct sockaddr *dir, int len);
```

```
int recvfrom (int sd, char *mem, int tam, int flags,  
              struct sockaddr *dir, int *len);
```

- Transmisión para NO conectados

Cliente

(Inicia la conversación)

socket

- Adquirir buzón UDP

bind

- Asignarle una dirección libre

sendto

- Enviar carta con remite: ¿Hola?

recvfrom

- Recoger respuesta

close

- Eliminar buzón

Servidor SIN estado

(Recibe mensajes y los atiende)

socket

- Adquirir buzón UDP

bind

- Asignarle dirección bien conocida

recvfrom

- Recoger carta
- Tomar dirección del remitente

sendto

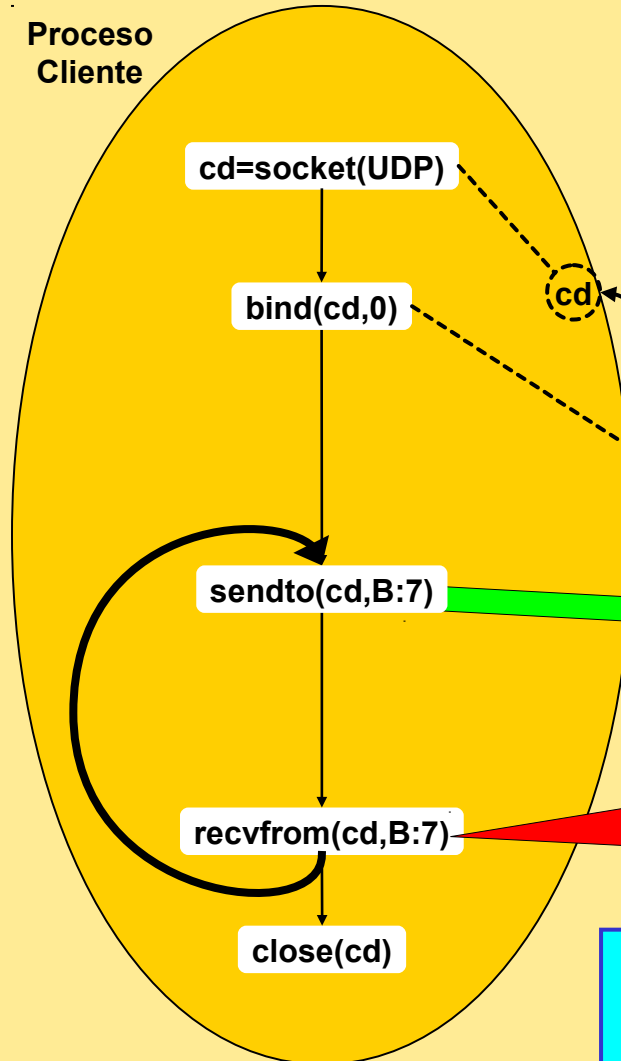
- Enviar al remitente: ¿Dígame?

close

- Eventualmente, eliminar buzón

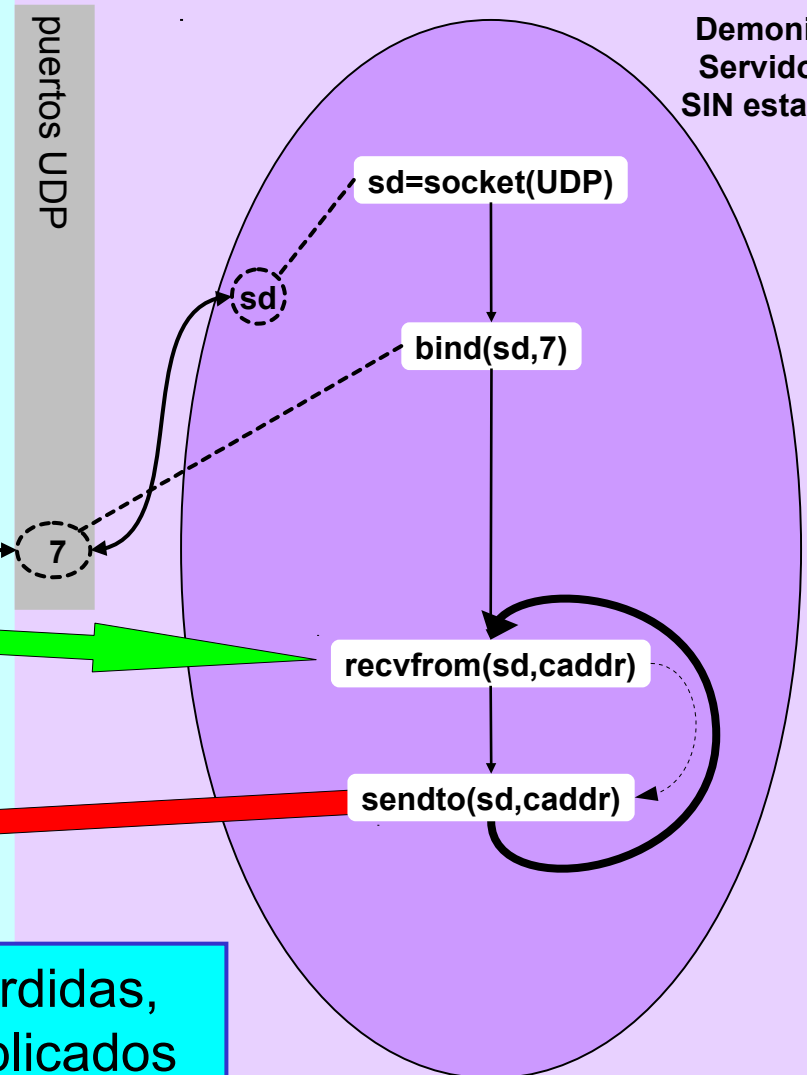
Máquina Cliente A

Proceso Cliente



Máquina Servidor B

Demonio Servidor SIN estado



NO fiable: pérdidas, desorden, duplicados

```
int main(int argc, char * argv[])
{
    int cd, size ;
    struct hostent * hp;
    struct sockaddr_in s_ain, c_ain;
    unsigned char byte;

    hp = gethostbyname(argv[1]); /* por ejemplo, otilio.fi.upm.es */
    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    memcpy (&(s_ain.sin_addr), hp->h_addr, hp->h_length);
    s_ain.sin_port = htons(7); /* echo port */

    cd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    bzero((char *)&c_ain, sizeof(c_ain));
    c_ain.sin_family = AF_INET;
    bind(cd, (struct sockaddr *)&c_ain, sizeof(s_ain));
    size = sizeof(c_ain);
    while(read( 0, &byte, 1) == 1) {
        sendto(cd, &byte, 1, 0, (struct sockaddr *)&s_ain, size);
        recvfrom(cd, &byte, 1, 0, (struct sockaddr *)&s_ain, &size);
        write(1, &byte, 1);
    }
    close(cd);
    return 0;
}
```

O se hace el
bind explícito o el
sendto lo hará
implícitamente

Si se pierde un byte
se "cuelga" el cliente



```
int main(void)
{
    int sd, size;
    unsigned char byte;
    struct sockaddr_in s_ain, c_ain;

    sd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    s_ain.sin_addr.s_addr = INADDR_ANY; /*Cualquier origen*/
    s_ain.sin_port = htons(7); /* echo server */
    bind(sd, (struct sockaddr *)&s_ain, sizeof(s_ain));
    size = sizeof(c_ain);

    while (1) {
        recvfrom(sd, &byte, 1, 0, (struct sockaddr *)&c_ain, &size);
        sendto(sd, &byte, 1, 0, (struct sockaddr *)&c_ain, size);
    }
}
```



Cliente *(Inicia la conversación)*

socket

- Adquirir teléfono

bind

- Contratar línea (nº de teléfono)

connect

- Descolgar
- Marcar

- Esperar establecimiento llamada

Centralita Servidor *(Recibe llamada y redirige)*

socket

- Adquirir centralita

bind

- Contratar línea (nº de teléfono)

listen

- Dimensionar centralita

accept

- Esperar establecimiento llamada
- Redirigir a teléfono de servicio

Servidor dedicado *(Atiende la conversación)*

send & recv

- Saludo:

¿Hola?

send & recv

- Diálogo:

Quisiera
¡Gracias!

close

- Colgar

send & recv

- Saludo:

¿Qué desea?

send & recv

- Diálogo:

¡Cómo no
Aquí tiene

close

- Colgar

CLIENTE-SERVIDOR TCP (SERVIDOR CONCURRENTE)

© Latsi UPM 2016



Máquina Cliente A

Proceso Cliente

cd=socket(TCP)

bind(cd,1025)

connect(cd,B:7)

send(cd)

recv(cd)

close(cd)

puertos TCP

puertos TCP

Máquina Servidora B

Demonio Servidor

sd=socket(TCP)

bind(sd,7)

listen(sd,5)

cd=accept(sd)

close(cd)

fork()

Servidor dedicado

close(sd)

recv(cd)

send(cd)

close(cd)

Conexión:
TCP:A:1025:B:7

Fiable: asegura entrega y orden



T C P _ c . c

```
int main(int argc, char * argv[])
{
    int cd;
    struct hostent * hp;
    struct sockaddr_in s_ain;
    unsigned char byte;

    hp = gethostbyname(argv[1]);
    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    memcpy (&(s_ain.sin_addr), hp->h_addr, hp->h_length); /* IP */
    s_ain.sin_port = htons(7); /* echo port */

    cd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    connect(cd, (struct sockaddr *)&s_ain, sizeof(s_ain));

    while(read( 0, &byte, 1) == 1) {
        send(cd, &byte, 1, 0); /* Bloqueante */
        recv(cd, &byte, 1, 0); /* Bloqueante */
        write(1, &byte, 1);
    }
    close(cd);
    return 0;
}
```

```
int main(void)
{
    int sd, cd, size;
    unsigned char byte;
    struct sockaddr_in s_ain, c_ain;

    sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    s_ain.sin_addr.s_addr = INADDR_ANY; /*Cualquier origen*/
    s_ain.sin_port = htons(7); /* echo port */

    bind(sd, (struct sockaddr *)&s_ain, sizeof(s_ain));

    listen(sd, 5);    /* 5 = tamaño cola */

    /* continúa... */
}
```



```
while(1) {
    size = sizeof(c_ain);
    cd = accept(sd, (struct sockaddr *)&c_ain, &size);
    switch(fork()) {
        case -1:
            perror("echo server");
            return 1;
        case 0:
            close(sd);
            while(recv(cd, &byte, 1, 0) == 1) /*Bloqueante*/
                send(cd, &byte, 1, 0);      /*Bloqueante*/
            close(cd);
            return 0;
        default:
            close(cd);
    } /* switch */
} /* while */
} /* main */
```

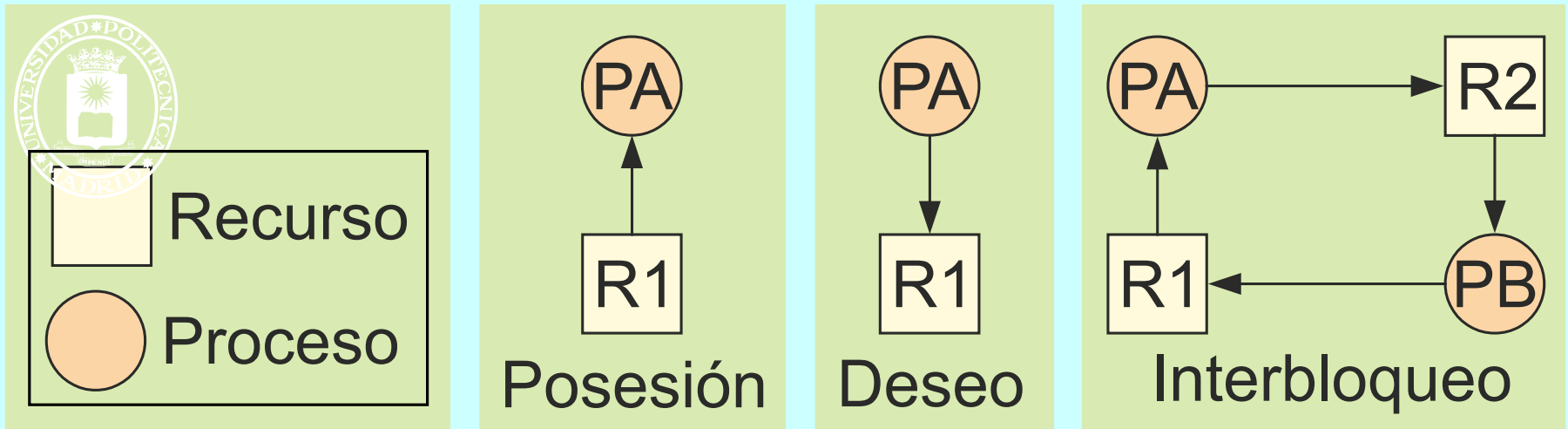


INTERBLOQUEOS

Un conjunto de procesos está en interbloqueo cuando cada proceso está bloqueado en espera de un recurso (o evento) que está asignado a (o sólo puede producir) un proceso (también bloqueado) del conjunto

- En grafo de asignación de recursos se visualizan como bucles

Ej. PA tiene R1 y necesita R2
PB tiene R2 y necesita R1



■ Ej. Si S y Q son semáforos binarios

P0	P1
<code>sem_wait(S)</code>	<code>sem_wait(Q)</code>
<code>sem_wait(Q)</code>	<code>sem_wait(S) ← !!</code>
<code>...</code>	<code>...</code>
<code>sem_post(S)</code>	<code>sem_post(Q)</code>
<code>sem_post(Q)</code>	<code>sem_post(S)</code>

■ Ej. PIPE y lectura

```
...  
pipe(pp) ;  
read(pp[0], ...) ; ← !!  
/* NO termina */  
/* !Quedan escritores! */  
...
```

■ Condiciones para el interbloqueo

- Exclusión mutua:
Un único usuario por recurso
- Mantener y esperar:
Espero sin liberar lo que tengo
- Sin expulsión:
No se quita un recurso ya asignado
- Espera circular:
Puede darse esta situación (bucle)

■ ¿Cómo tratar el interbloqueo?

- No hacer nada: Alg. del avestruz
- Prevenirlo: detectarlo y notificarlo
- Impedirlo: eliminar alguna de las condiciones necesarias
- Evitarlo: asignando con cuidado los recursos (Alg. banquero, etc.)



- **La concurrencia sucede a muchos niveles**
- **Implica compartición de recursos**
- **Condición de carrera == No se puede garantizar la velocidad relativa de los procesos concurrentes ← pueden suceder resultados incorrectos**
- **Sección crítica == trozos de código susceptibles de condición de carrera**
- **Para coordinar el acceso usamos:**
 - **Mecanismos de sincronización**
 - **Mecanismos de comunicación**
- **Mecanismos de sincronización:**
 - **Acciones atómicas sobre base de instrucciones máquina atómicas**
 - **Modelos clásicos: modelan arquitecturas clave en concurrencia**
- **Para procesos fuertemente acoplados (== que comparten memoria):**
 - **Semáforos == contador**
 - **Mutex == cerrojo**
 - **Condición: liberar temporalmente mutex para esperar indicación**
- **Para procesos independientes**
 - **Cerrojos sobre regiones de fichero**
- **Transacciones: procesos independientes y distribuidos**



- **Mecanismos de comunicación**
== **Recurso compartido + Mecanismo de sincronización**
 - **Con o sin nombre, descriptor de fichero o identificador propio, uni o bidireccional, con o sin *buffer*, bloqueante o no**
- **Para procesos locales:**
 - **PIPE == mecanismo para N productores y M consumidores**
 - **FIFO == crear con `mkfifo`, usar como fichero, semántica de PIPE**
- **Para procesos remotos:**
 - **Sockets: dominio, tipo, protocolo**
 - **Datagrama == correspondencia postal**
 - ***Stream* == conversación telefónica**
 - **Servicios genéricos PERO direcciones específicas del dominio**
 - **Formato de red**
- **Interbloqueos**
 - **Condiciones necesarias y tratamientos posibles**



...adecuados *normalmente* para

		HW	PLs <i>threads</i>	PPs Parientes	PPs Locales	PPs Remotos
...usados comunmente para	Sincronizar	tsl	mutex y condición		semáforo con nombre cerrojo sobre fichero	- Desacoplamiento + transacciones más cerrojos
	Avisar		semáforo sin nombre	señales		
	Almacenar		una imagen de memoria	memoria compartida	fichero proyectado	
	Comunicar		fichero			
		+ Alto nivel -		PIPE	FIFO socket UNIX datagrama o stream	socket INET datagrama o stream