



# Sistemas Operativos

Grado en Ingeniería Informática  
2018-2019

## Procesos e hilos

Basado en:

Sistemas Operativos  
J. Carretero [et al.]



# Contenido

- Procesos
- Multitarea
- Información del proceso
- Formación y estados de un proceso
- Señales
- Hilos o *threads*

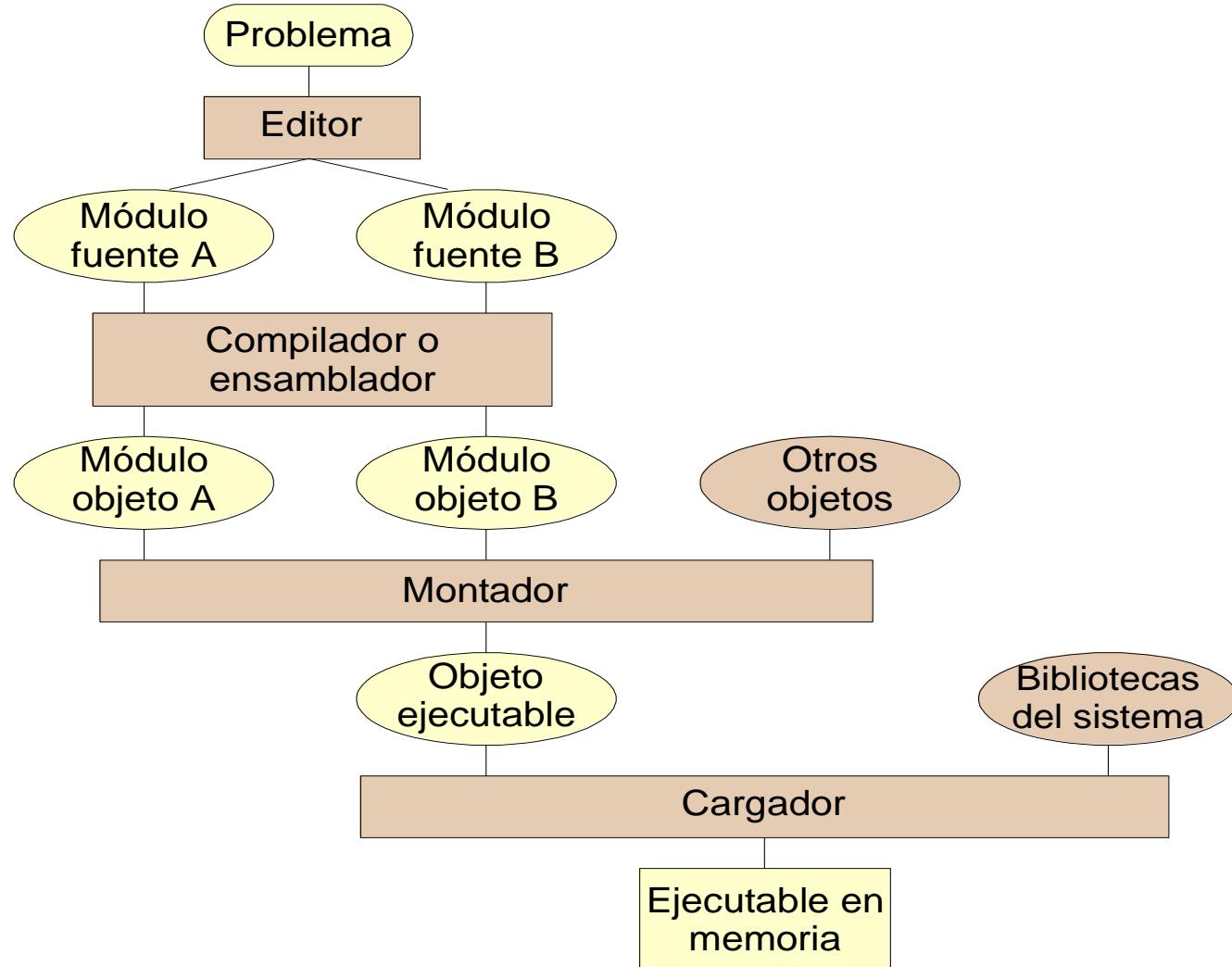


# Concepto de proceso

- Programa: fichero ejecutable en un dispositivo de almacenamiento permanente
- Proceso:
  - Programa en ejecución
  - Unidad de procesamiento gestionada por el SO
    - En realidad, la unidad *mínima* de procesamiento es el hilo (*thread*) ➔ Un proceso puede constar de uno o varios hilos
- Información del proceso:
  - Imagen de memoria: *core image*
  - Estado del procesador: registros del modelo de programación
  - Bloque de control del proceso **BCP**



# Preparación del código de un proceso





# Recuerda: comandos útiles

- gcc
  - Compilador C de GNU
  - Realiza todas las etapas
    - > `gcc -save-temps hello.c -o hello.o`
- ldd
  - Permite ver las librerías con las que hemos enlazado
- nm / objdump
  - Permiten visualizar partes de un ejecutable
  - Opciones típicas de objdump: -S, -t, -f, -h

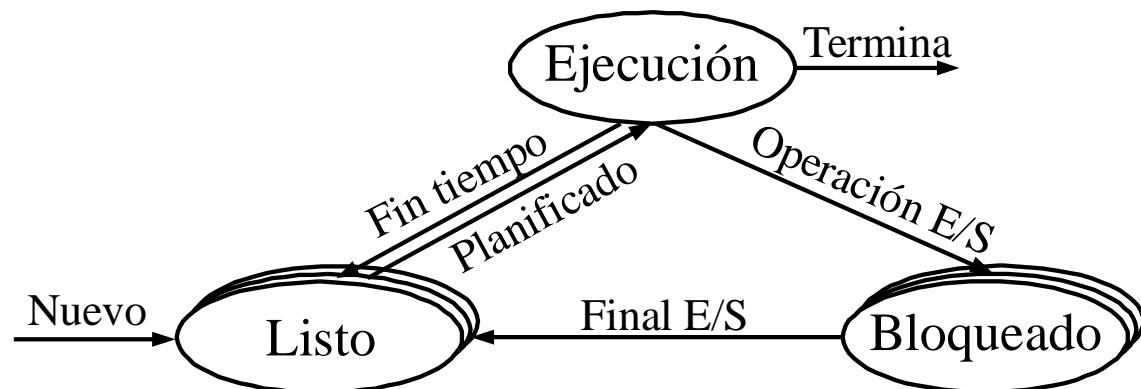


# Curiosidades Linux: /proc

- Directorio /proc contiene un directorio por cada proceso en ejecución
- Permite consultar información sobre el proceso:
  - Línea de comando
  - Mapa de memoria
  - Tabla de páginas
  - ...

# Estados básicos de un proceso

- En ejecución (uno por procesador/core)
- Bloqueado (en espera de completar E/S o por motivos de sincronización)
- Listo para ejecutar



- Planificador: Componente del SO que decide qué proceso se ejecuta en cada procesador y en qué instante
- Proceso nulo o *idle* (uno por cada procesador)



# Entorno del proceso

- Tabla *NOMBRE-VALOR* que se pasa al proceso en su creación
- Se establece:
  - Por defecto (heredados del proceso padre)
  - Mediante comandos del *shell* (*export NOMBRE=valor*)
  - Mediante funciones de la biblioteca estandar de “C” (*putenv*, *getenv*)
- Ejemplo:
  - PATH=/usr/bin:/home/pepe/bin
  - TERM=vt100
  - HOME=/home/pepe
  - PWD=/home/pepe/libros/primero
  - TIMEZONE=EDT

# Jerarquía de procesos (UNIX)

## ■ Familia de procesos

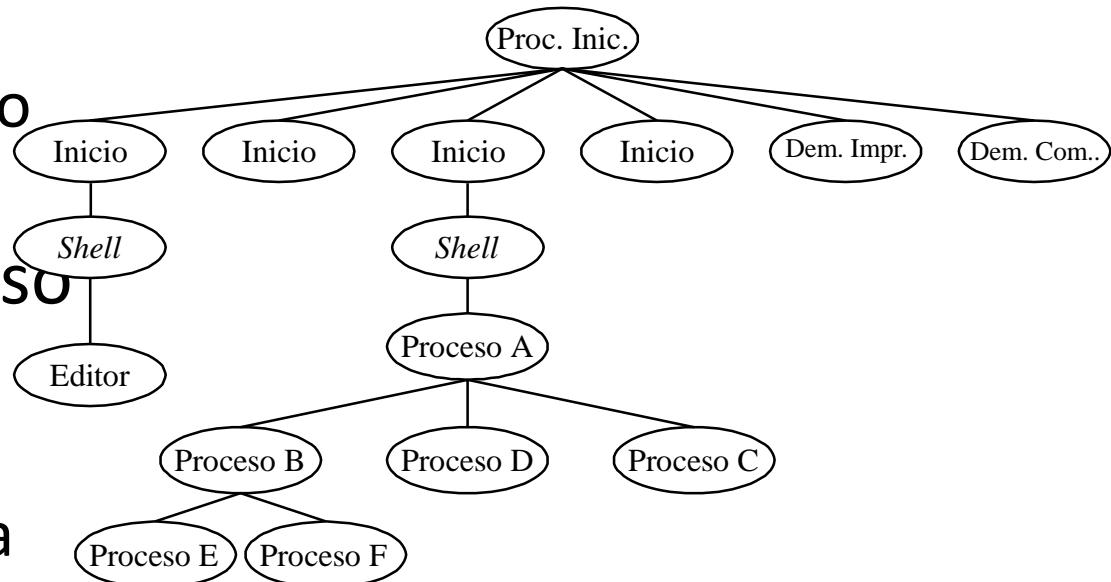
- Proceso hijo
- Proceso padre
- Proceso hermano
- Proceso abuelo

## ■ Vida de un proceso

- Crea
- Ejecuta
- Muere o termina

## ■ Ejecución del proceso

- Batch
- Interactivo





# Consulta procesos en ejecución

- ps
  - Permite ver la información de todos los procesos en ejecución
  - *man ps* para consultar las múltiples opciones
- top
  - Muestra los procesos en ejecución, refrescando la información periódicamente
  - Permite interaccionar con los procesos (enviar señales)

# Usuario



- Usuario: Persona autorizada a utilizar un sistema
  - Se identifica en la autenticación mediante:
    - Código de cuenta
    - Clave (*password*)
  - Internamente el SO le asigna el “uid” (*user identification*)
- Superusuario (*root*)
  - Tiene todos los derechos
  - Administra el sistema
- Grupo de usuarios
  - Los usuarios se organizan en grupos
    - Alumnos
    - Profesores
  - Todo usuario ha de pertenecer al menos a un grupo

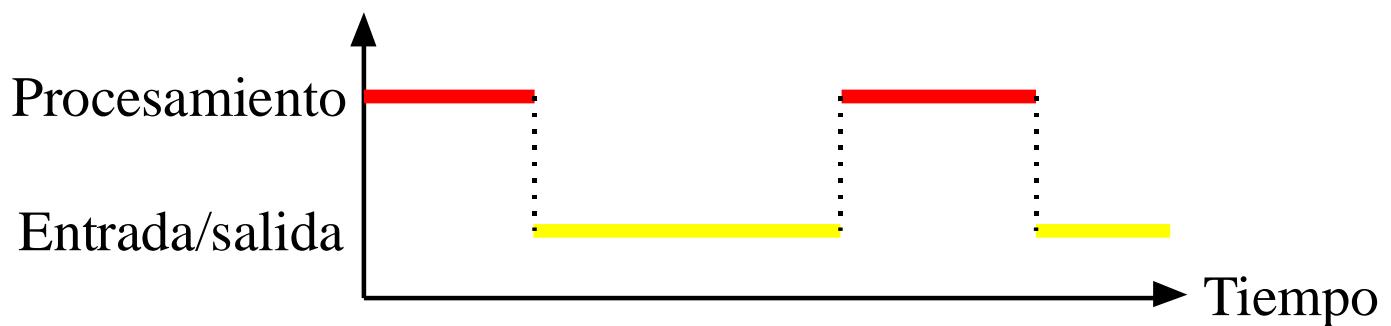


# Contenido

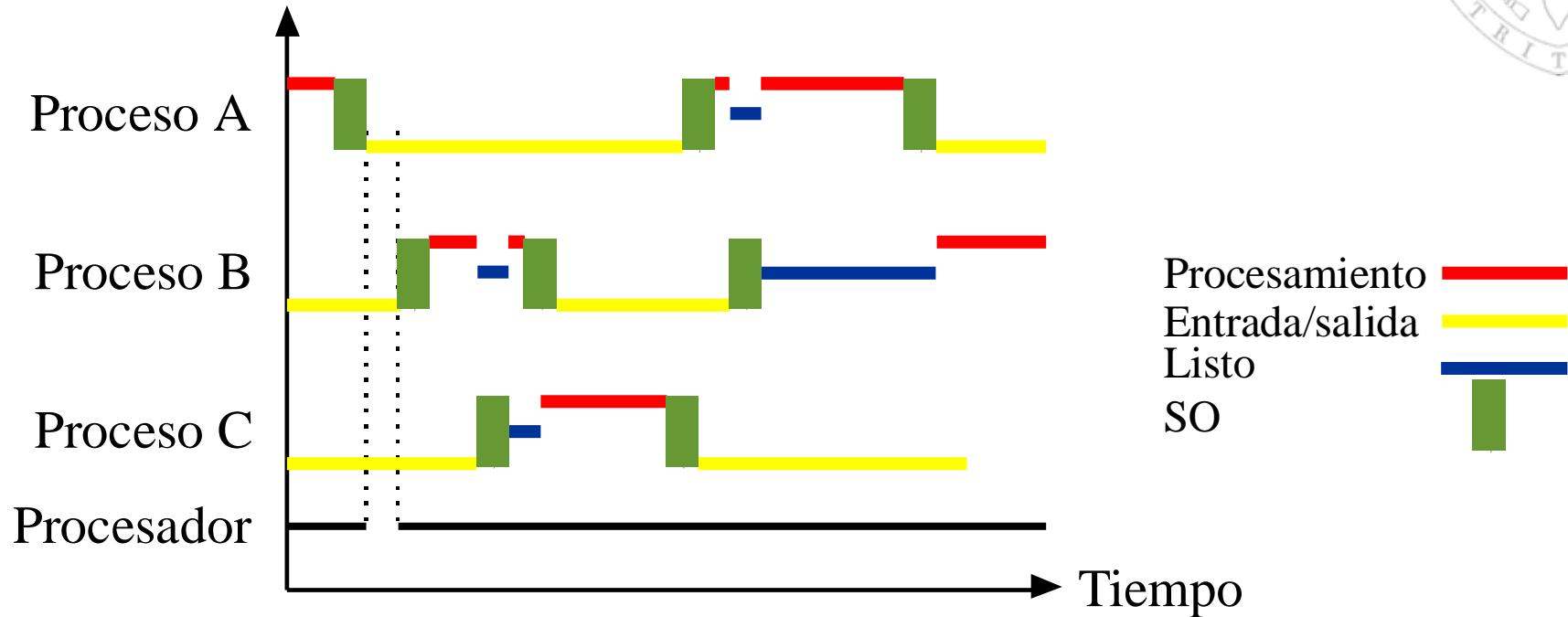
- Procesos
- Multitarea
- Información del proceso
- Formación y estados de un proceso
- Señales
- Hilos o *threads*

# Base de la multitarea

- Paralelismo real entre E/S y CPU (DMA)
- Alternancia en los procesos de fases de E/S y de procesamiento
- La memoria almacena varios procesos



# Ejecución en un sistema multitarea



- Proceso nulo o *idle*

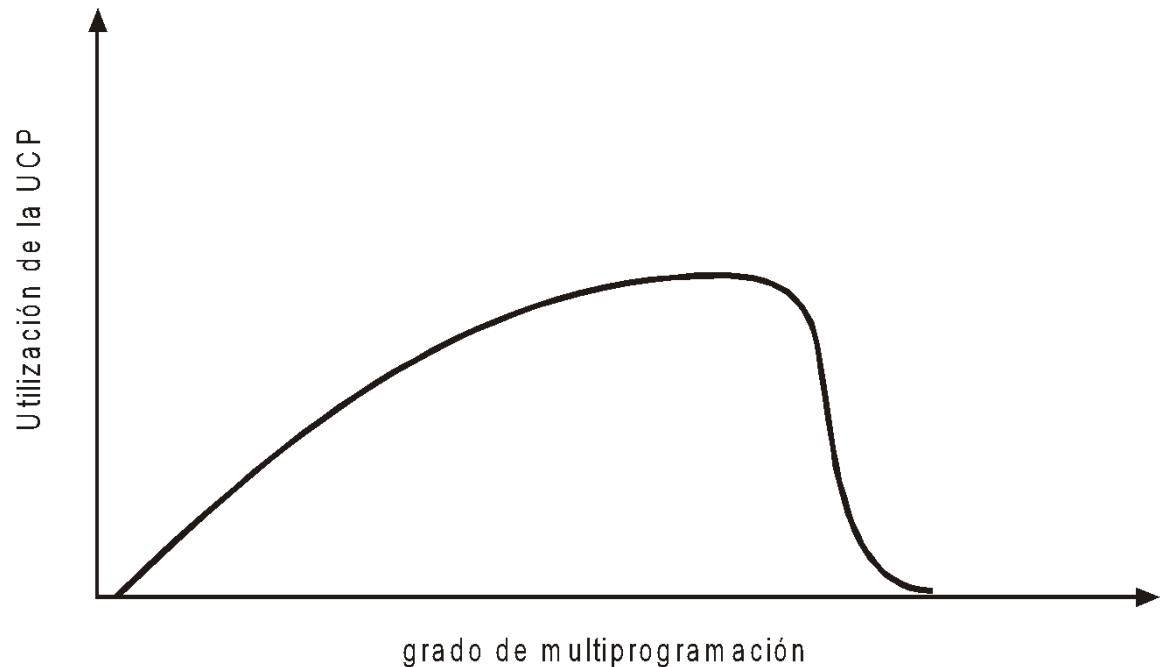
# Ventajas de la multitarea



- Facilita la programación, dividiendo los programas en procesos (modularidad)
- Permite el servicio interactivo simultáneo de varios usuarios de forma eficiente
- Aprovecha los tiempos que los procesos pasan esperando a que se completen sus operaciones de E/S
- Aumenta el uso de la CPU

# Grado de multiprogramación

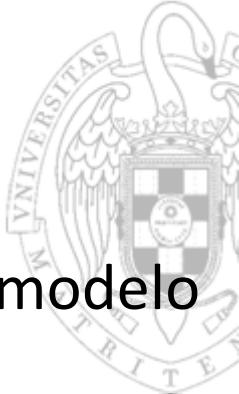
- Grado de multiprogramación: nº de procesos activos
- Para sistemas con memoria virtual:





# Contenido

- Procesos
- Multitarea
- Información del proceso
- Formación y estados de un proceso
- Señales
- Hilos o *threads*

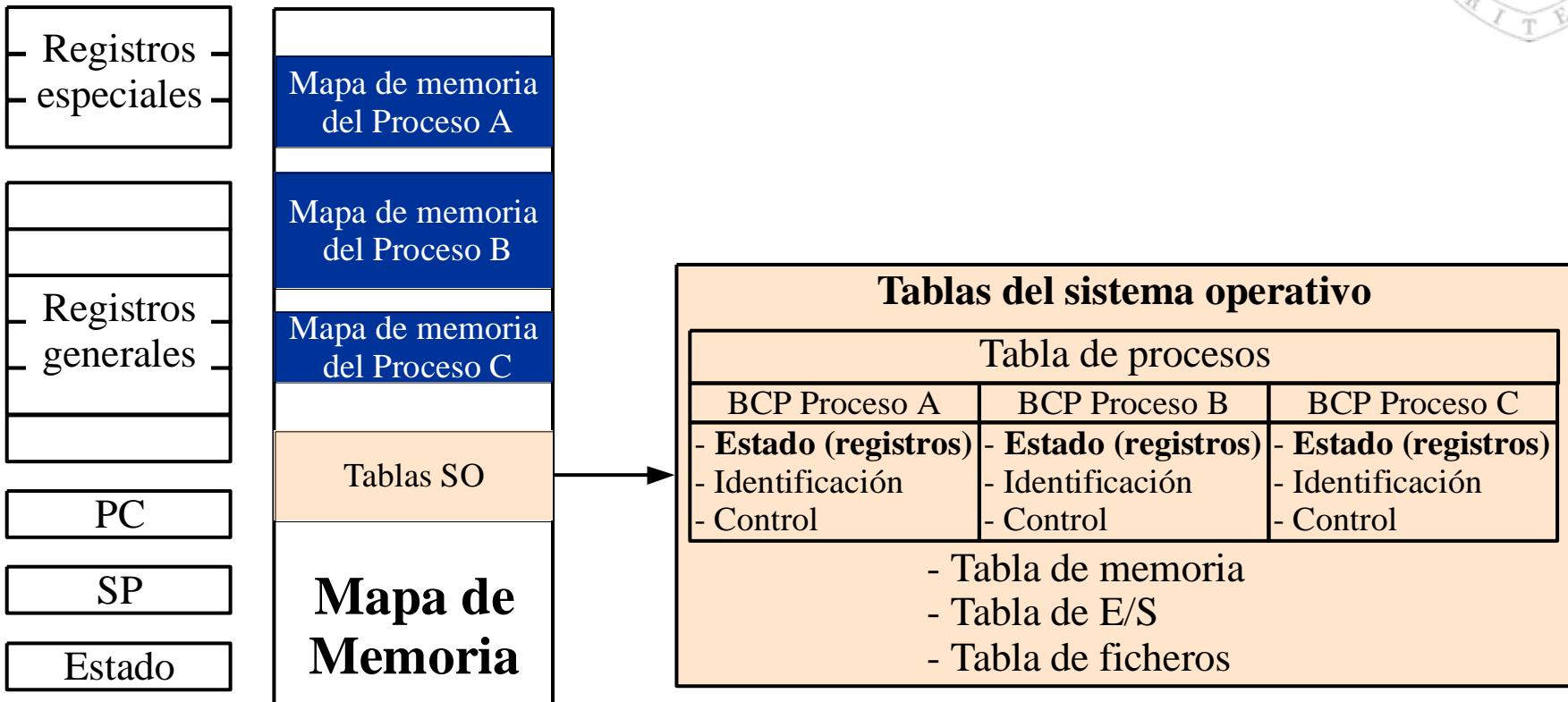


# Información de un proceso

- **Estado del procesador:** contenido de los registros del modelo de programación
- **Imagen de memoria:** contenido de los segmentos de memoria en los que reside el código y los datos del proceso
- **Bloque de control del proceso (BCP) o Descriptor de proceso**
  - Estado actual del proceso
  - Estado del procesador
    - Actualizado cuando proceso no se está ejecutando en la CPU
  - Identificadores *pid*, *uid*, etc.
  - Prioridad
  - Segmentos de memoria (espacio de direcciones)
  - Ficheros abiertos
  - Temporizadores
  - Señales
  - Semáforos
  - Puertos



# Información de un proceso (II)



# Estado del procesador



- Está formado por el contenido de todos sus registros:
  - Registros generales
  - Contador de programa
  - Puntero de pila
  - Registro de estado
  - Registros especiales
- Cuando un proceso está ejecutando su estado reside en los registros del computador.
- Cuando un proceso no ejecuta su estado reside en el BCP.

# Imagen de memoria



- La imagen de memoria está formada por el conjunto de regiones de memoria que un proceso está autorizado a utilizar
- Si un proceso genera una dirección que esta fuera del espacio de direcciones el HW genera una excepción que el SO captura
- La imagen de memoria, dependiendo del computador, puede estar referida a memoria virtual o memoria física



# Información del BCP

- Información de identificación:
  - PID del proceso, PID del padre (PPID)
  - ID de usuario y grupo reales (uid/gid reales)
  - ID de usuario y grupo efectivos (uid/gid efectivos)
- Estado del procesador
- Información de control del proceso:
  - Información de planificación y estado
  - Descripción de los segmentos de memoria del proceso
  - Recursos asignados (ficheros abiertos, ...)
  - Recursos de comunicación entre procesos
  - Punteros para estructurar los procesos en listas o colas

# Información del BCP II



- Información fuera del BCP
  - Conveniente por implementación (la consideramos del BCP)
  - Para compartirla
- La tabla de páginas se pone fuera
  - Describe la imagen de memoria del proceso
  - Tamaño variable
  - El BCP contiene el puntero a la tabla de páginas
  - La compartición de memoria requiere que sea externa al BCP
- Punteros de posición de los ficheros
  - Si se añaden a la tabla de ficheros abiertos (en el BCP) no se pueden compartir
  - Si se asocian al nodo-i se comparte siempre
  - Se ponen en una estructura común a los procesos y se asigna uno nuevo en cada servicio OPEN



# Tablas del sistema operativo

- **Tabla de procesos** (tabla de BCP)
- **Tabla de memoria**: información sobre el uso de la memoria.
- **Tabla de E/S**: guarda información asociada a los periféricos y a las operaciones de E/S
- **Tablas de fichero**: información sobre los ficheros abiertos.
- La información asociada a cada proceso en el BCP
- La decisión de incluir o no una información en el BCP se toma según dos criterios:
  - Eficiencia
  - Compartir información

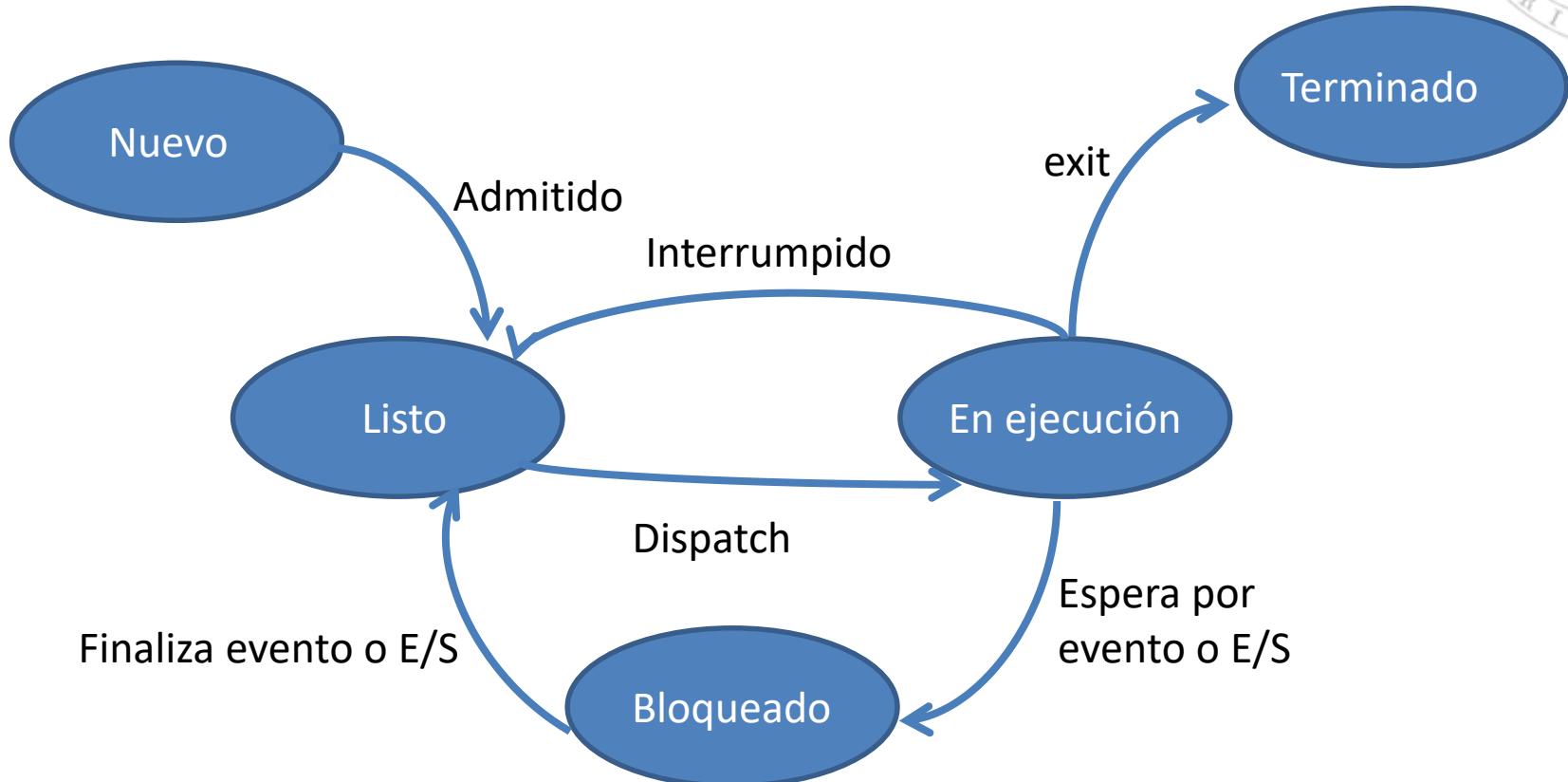


# Contenido

- Procesos
- Multitarea
- Información del proceso
- Formación y estados de un proceso
- Señales
- Hilos o *threads*



# Estados del proceso





# Cambio de modo del procesador

- Cuando se produce una interrupción o excepción mientras un proceso se ejecuta en modo usuario el procesador cambia de modo de ejecución (modo kernel)
- Al producirse la interrupción:
  - Se pasa a ejecutar la rutina de tratamiento de interrupción (RTI), bajo control del SO
  - Se salva el valor de los registros (estado del procesador) en la pila del kernel
- Al finalizar la RTI, si el proceso actual sigue “en ejecución”:
  - Restaura los registros del procesador
  - Termina con una instrucción RETI (retorno de interrupción)
    - Restituye el registro de estado (bit de nivel de ejecución)
    - Restituye el contador de programa (para el nuevo proceso).

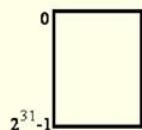
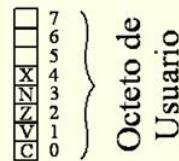


# Modo usuario y modo kernel

D0			
D1			
D2			
D3			
D4			
D5			
D6			
D7			

Registro de estado

A0			
A1			
A2			
A3			
A4			
A5			
A6			
A7			



Mapa de memoria

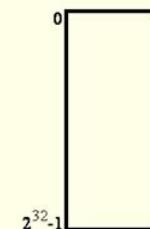


Juego de  
Instrucciones

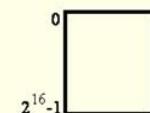
**Modo usuario**

D0			
D1			
D2			
D3			
D4			
D5			
D6			
D7			

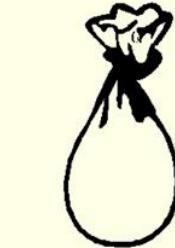
A0			
A1			
A2			
A3			
A4			
A5			
A6			
A7			



Mapa de memoria



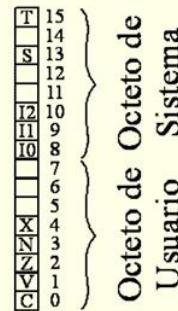
Mapa de  
E/S



Juego de  
Instrucciones

**Modo núcleo o modo kernel**

Registro de estado



Octeto de Sistema



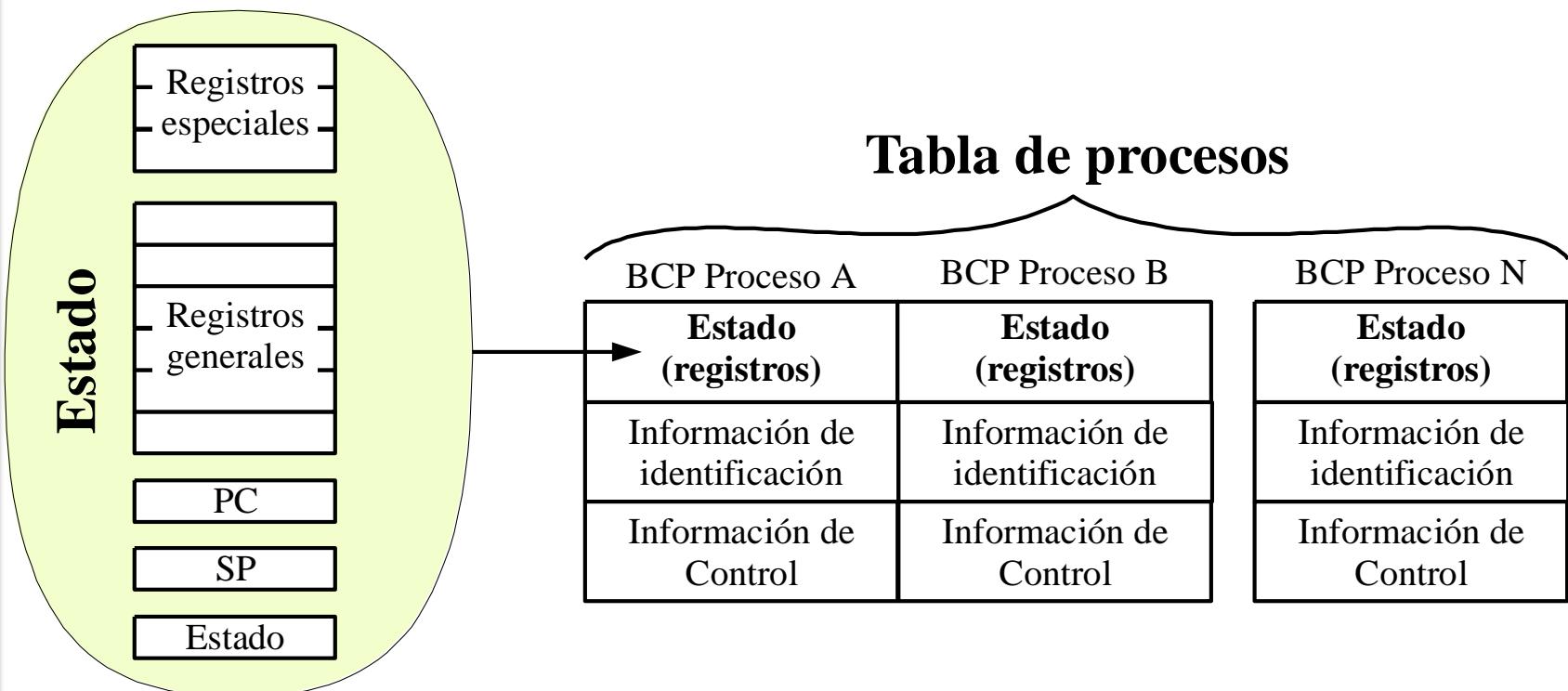
# Cambio de contexto

- El **cambio de contexto** es el conjunto de acciones que realiza el SO para cambiar el proceso que está actualmente en ejecución en una CPU
- Acciones (simplificación):
  1. Salvar el contexto del proceso saliente (registros del modelo de programación) en su BCP
  2. Cambiar el estado del proceso saliente (En ejecución -> Otro Estado)
  3. Intercambio de los espacios de direcciones
    - Segmentos o regiones de memoria que puede usar un proceso
    - En x86. GDT (Global descriptor Table)
    - En algunas arquitecturas → Invalidación de entradas de la TLB
  4. Cambiar el estado del proceso entrante, (Listo -> En ejecución)  
Restaurar su contexto (BCP -> registros) y volver a modo usuario
- Puede llegar a ser una operación bastante costosa
- *El cambio de modo de ejecución del procesador no siempre desencadena un cambio de contexto*

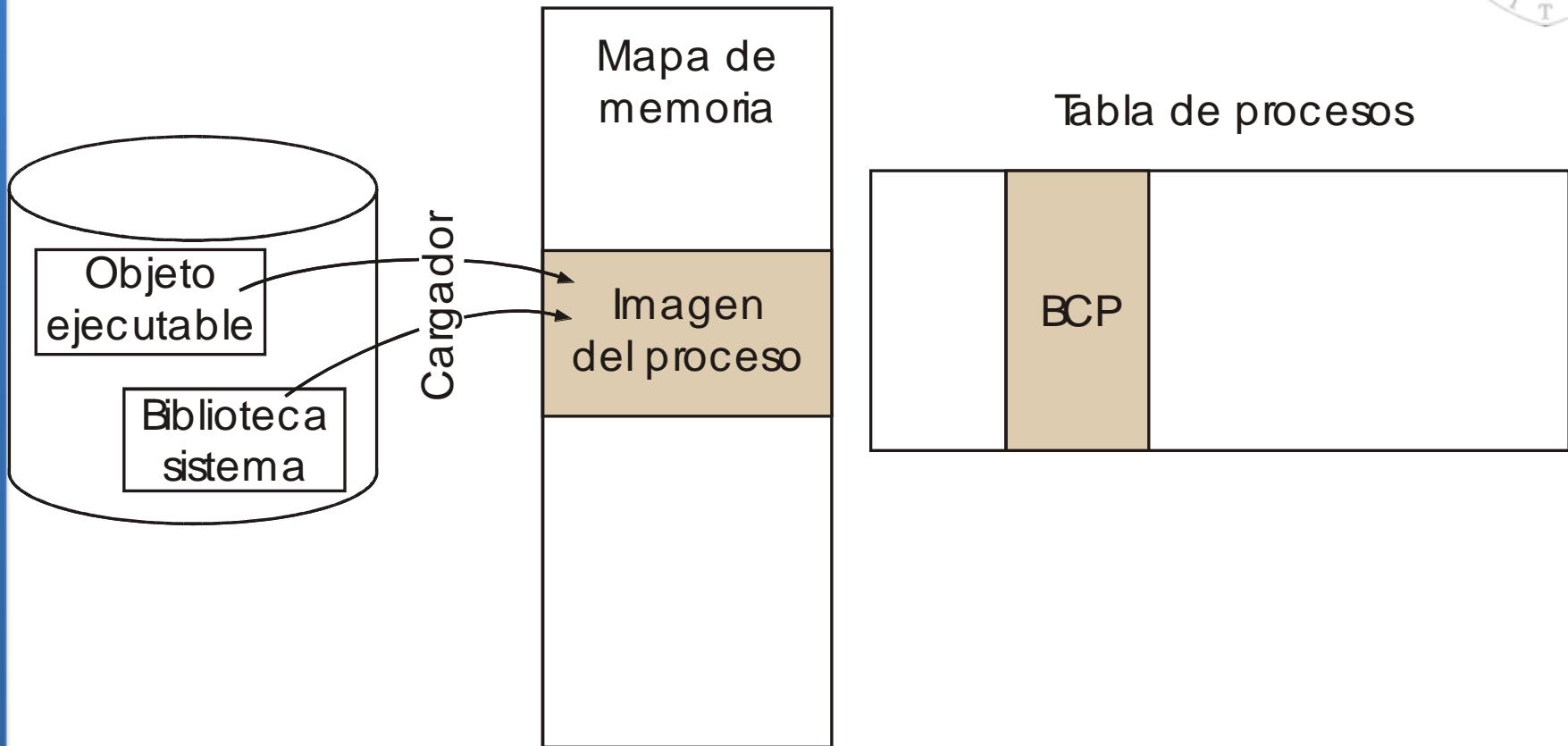


# Cambio de contexto

- Se salva el estado:



# Formación de un proceso



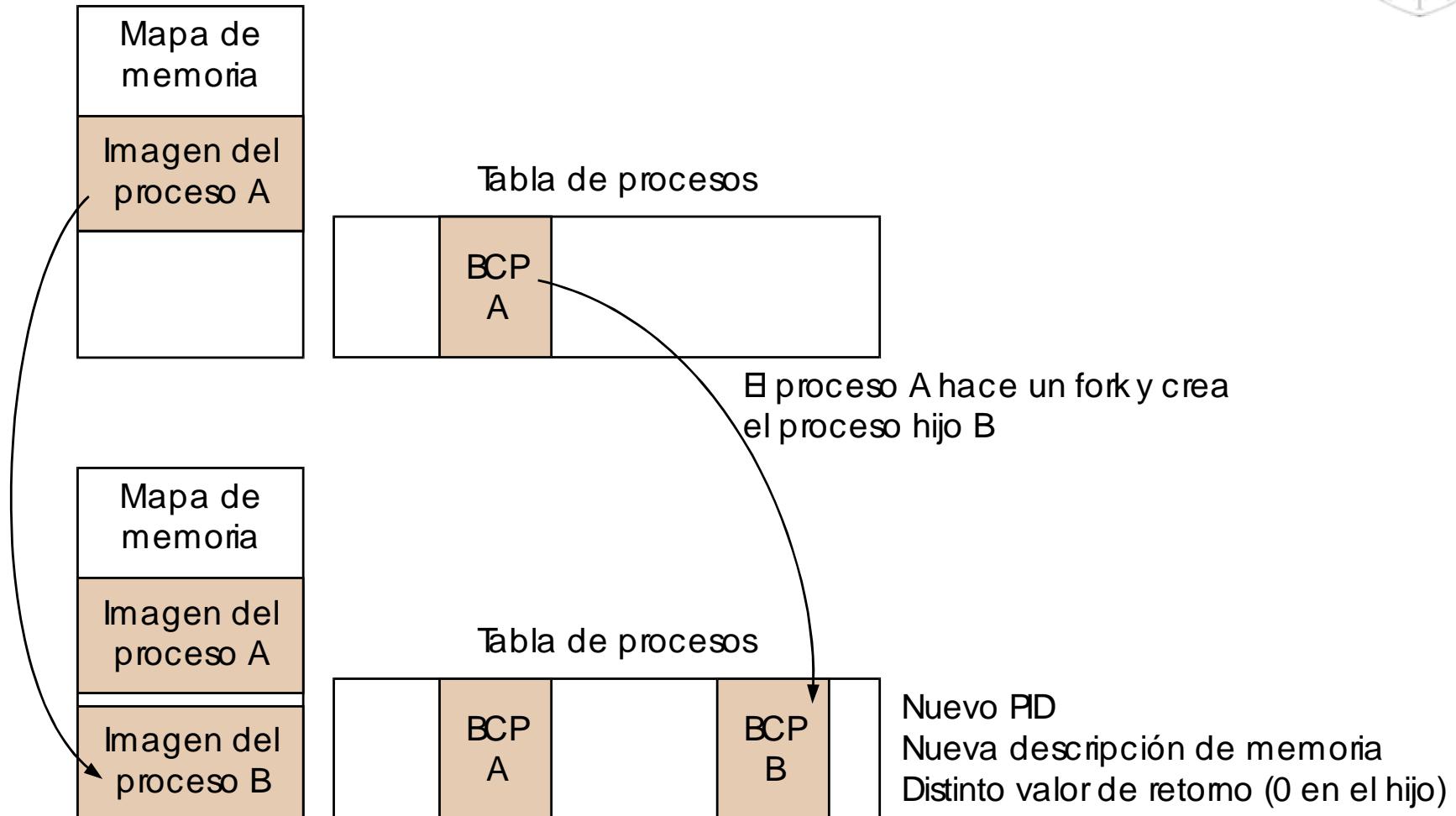


# Proceso vs. Ejecutable

- Si tenemos nuestro fichero binario ejecutable  $X$ , lo ejecutamos y sin esperar a que termine lo volvemos a ejecutar....
  - ¿Tendré uno o dos procesos?
  - Si tengo dos, ¿comparten todas las zonas de memoria?
  - Si uno abre un fichero, ¿el otro ya lo tiene abierto?

# Servicios POSIX: fork

- Crea un proceso clonando al padre





# fork. Crea un proceso

## ■ Servicio:

```
pid_t fork(void);
```

## ■ Devuelve:

- El identificador de proceso hijo al proceso padre y 0 al hijo
- -1 el caso de error

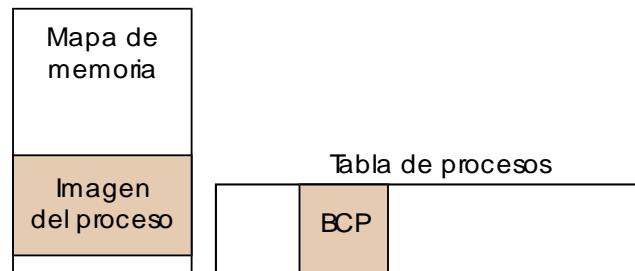
## ■ Descripción:

- Crea un proceso hijo que ejecuta el mismo programa que el padre
- Hereda los ficheros abiertos (se copian los descriptores).
- Las alarmas pendientes se desactivan.
- El proceso hijo sólo tiene un hilo.

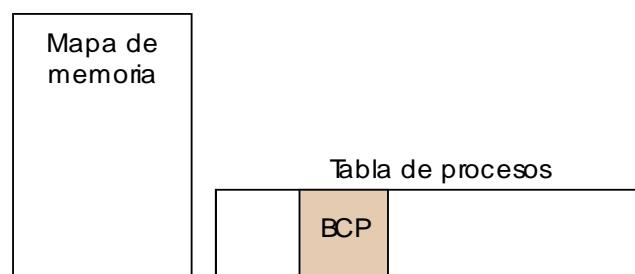
```
void main()
{
    pid_t pid;
    ...
    pid = fork();
    if (pid == 0) {
        /* proceso hijo */
        ...
    } else if (pid>0) {
        /* proceso padre */
        ...
    } else{
        /* error al crear */
        ...
    }
    ...
}
```

# Servicios POSIX: exec

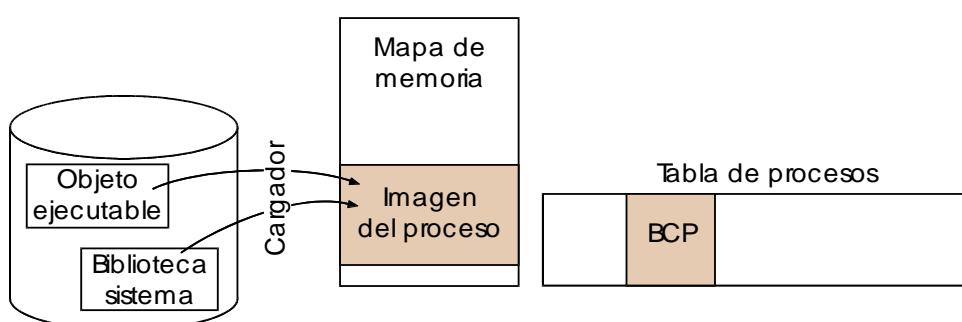
## ■ Cambia el programa de un proceso



El proceso hace un exec



Se borra la imagen de memoria  
Se borra la descripción de la memoria y registros  
Se conserva el PID



Se carga la nueva imagen  
Se pone PC en dirección de arranque  
Se conservan los fd



# exec. Cambio del programa de un proceso

## ■ Servicios:

```
int exec1 (const char *path, const char *arg, ...)  
int execlp(const char *file, const char *arg, ...)  
int execvp(const char *file, char *const argv[])
```

## ■ Argumentos:

- path, file: nombre del archivo ejecutable
- arg: argumentos

## ■ Descripción:

- Devuelve -1 en caso de error, en caso contrario **no** retorna.
- Cambia la imagen de memoria del proceso.
- El mismo proceso ejecuta otro programa.
- Los ficheros abiertos permanecen abiertos
- Las señales con la acción por defecto seguirán por defecto, las señales con manejador tomarán la acción por defecto.



# exit. Terminación de un proceso

## ■ Servicios:

```
int exit(int status);
```

## ■ Argumentos:

- Código de retorno al proceso padre

## ■ Descripción:

- Finaliza la ejecución del proceso.
- Se cierran todos los descriptores de ficheros abiertos.
- Se liberan todos los recursos del proceso



# wait. Espera la terminación de un proceso hijo

## ■ Servicios:

```
#include <sys/types.h>
pid_t wait(int *status);
```

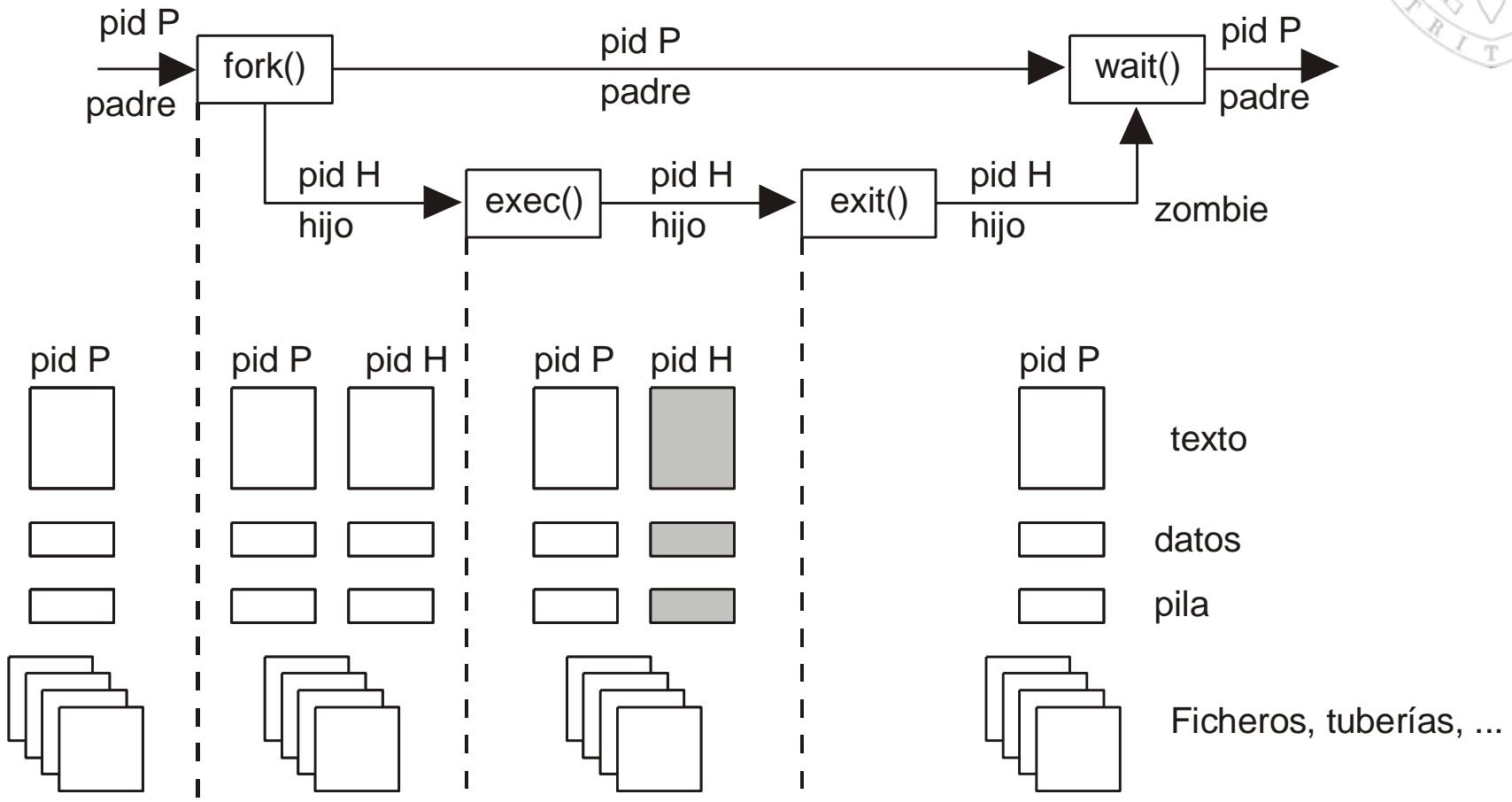
## ■ Argumentos:

- Devuelve el código de terminación del proceso hijo.

## ■ Descripción:

- Devuelve el identificador del proceso hijo o -1 en caso de error.
- Permite a un proceso padre esperar hasta que termine un proceso hijo. Devuelve el identificador del proceso hijo y el estado de terminación del mismo.

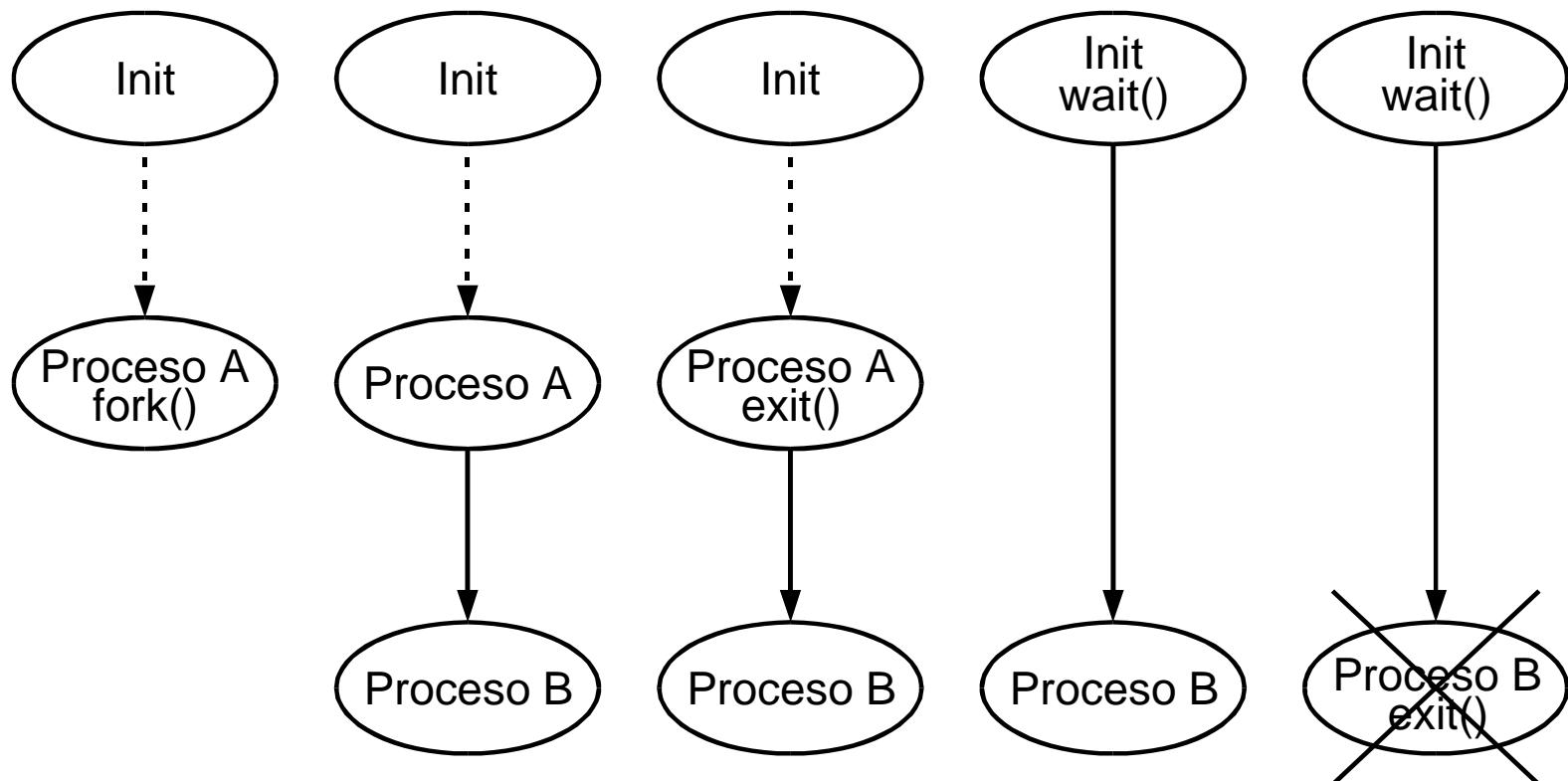
# Uso normal de los servicios



# Evolución de procesos I



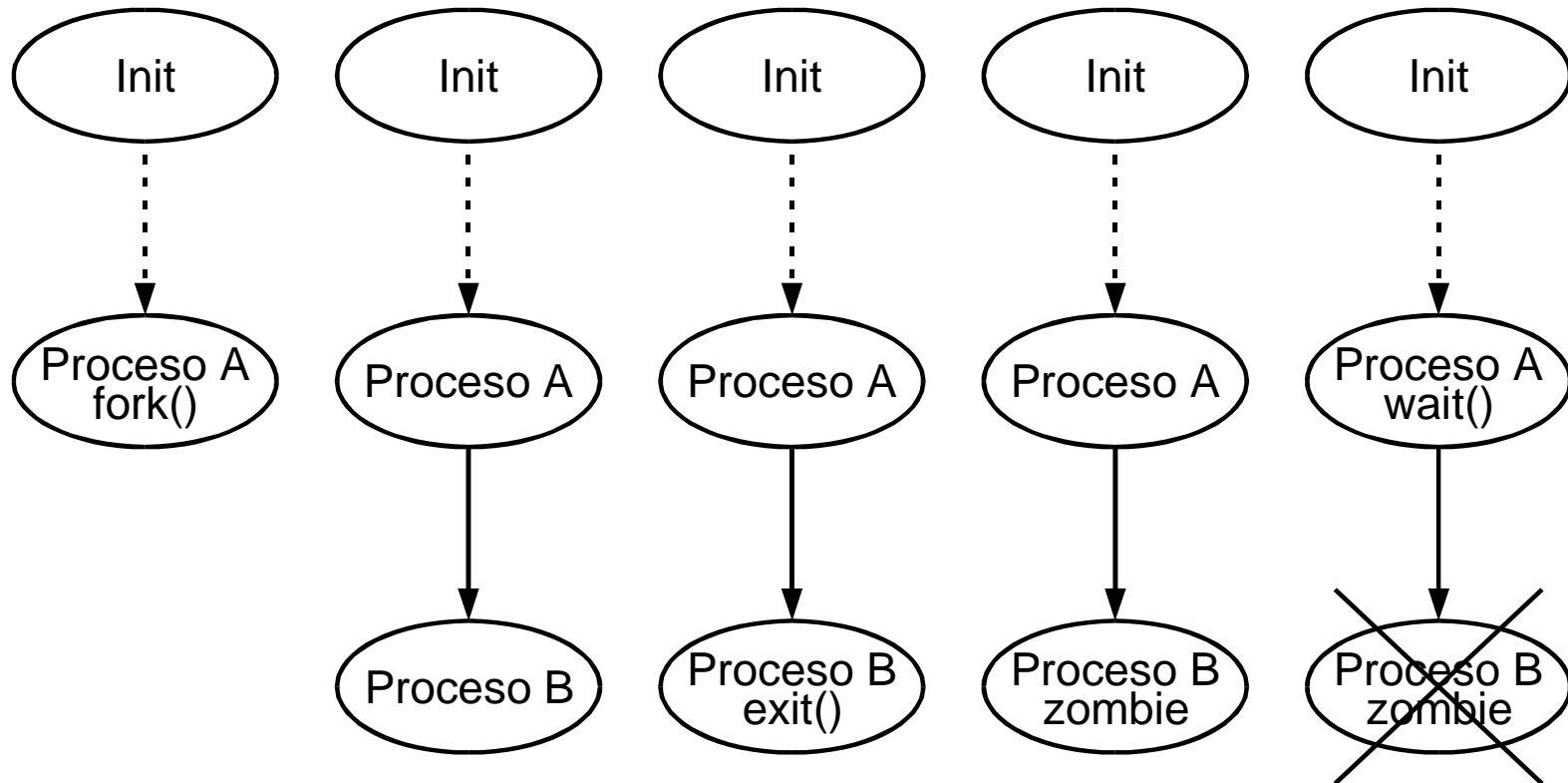
- El padre muere: *INIT* acepta los hijos



# Evolución de procesos II



- *Zombie*: el hijo muere y el padre no hace *wait*





# Programa de ejemplo

```
#include <sys/types.h>
#include <stdio.h>
/* programa que ejecuta el mandato ls -l */
main() {
    pid_t pid;
    int status;
    pid = fork();
    if (pid == 0) { /* proceso hijo */
        execvp("ls","ls","-l",NULL);
        exit(-1);
    }
    else          /* proceso padre */
        while (pid != wait(&status));
    exit(0);
}
```



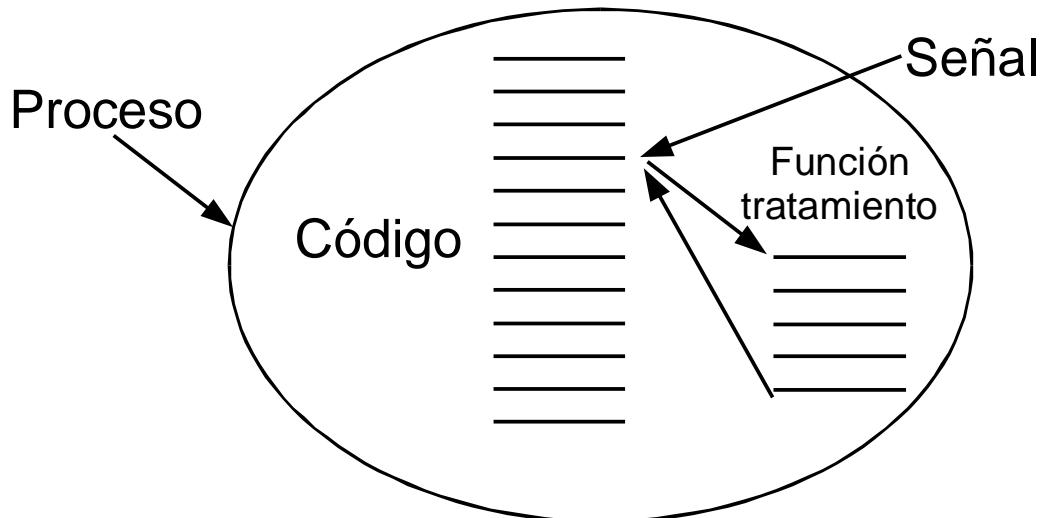
# Contenido

- Procesos
- Multitarea
- Información del proceso
- Formación y estados de un proceso
- Señales
- Hilos o *threads*

# Señales



- Las señales son interrupciones al proceso
- Envío o generación:
  - Proceso → Proceso (con mismo *uid*) con *kill*
  - SO → Proceso





# Señales II

- Hay muchos tipos de señales, según su origen
  - SIGILL instrucción ilegal
  - SIGALRM vence el temporizador
  - SIGKILL mata al proceso
- El SO las transmite al proceso
  - El proceso debe estar preparado para recibirla
    - Especificando un manejador de señal con *sigaction*
    - Enmascarando la señal con *sigprogmask*
  - Si no está preparado → acción por defecto
    - El proceso, en general, muere
    - Hay algunas señales que se ignoran o tienen otro efecto
- El servicio *pause* para el proceso hasta que recibe una señal

# Señales: servicios POSIX



- **int kill(pid\_t pid, int sig)**
  - Envía al proceso *pid* la señal *sig*
- **int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);**
  - Permite especificar la acción a realizar *act* como tratamiento de la señal *signum*. Permite almacenar la acción previa en *oldact*
- **int pause(void)**
  - Bloquea al proceso hasta la recepción de una señal.
- Ejemplo:

**\$ kill -SIGINT pid**

Envía al proceso con identificador *pid* la señal de interrupción (Ctrl C).

Si el proceso tiene un manejador para esa señal ejecutará el código del manejador.

En caso contrario, el proceso muere.

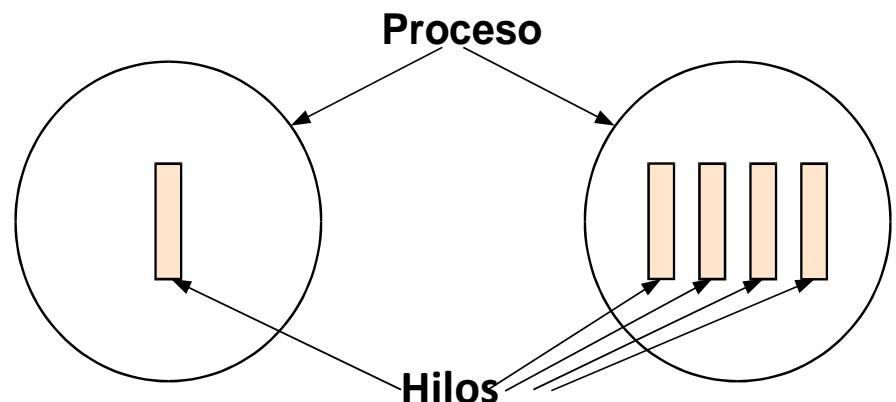


# Contenido

- Procesos
- Multitarea
- Información del proceso
- Formación y estados de un proceso
- Señales
- Hilos o *threads*

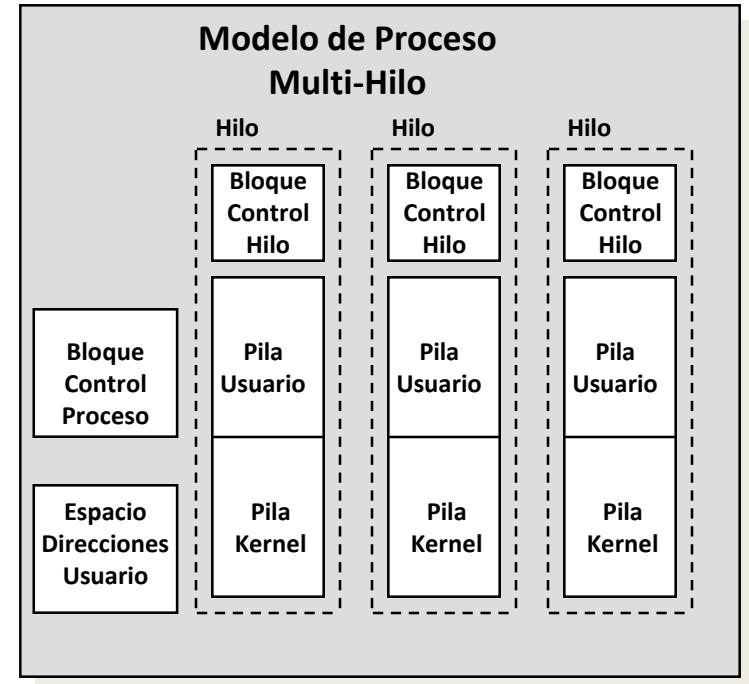
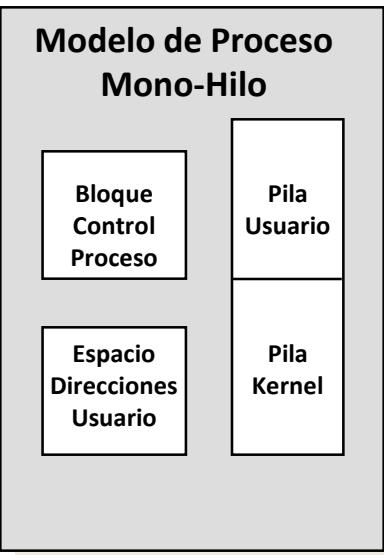
# Hilos o threads

- Por Hilo
  - Contador de programa, Registros
  - Pila
  - Estado (ejecutando, listo o bloqueado)
  - Bloque de control de *thread*
- Por proceso
  - Espacio de direcciones de memoria
  - Variables globales
  - Ficheros abiertos
  - Procesos hijos
  - Temporizadores
  - Señales y semáforos

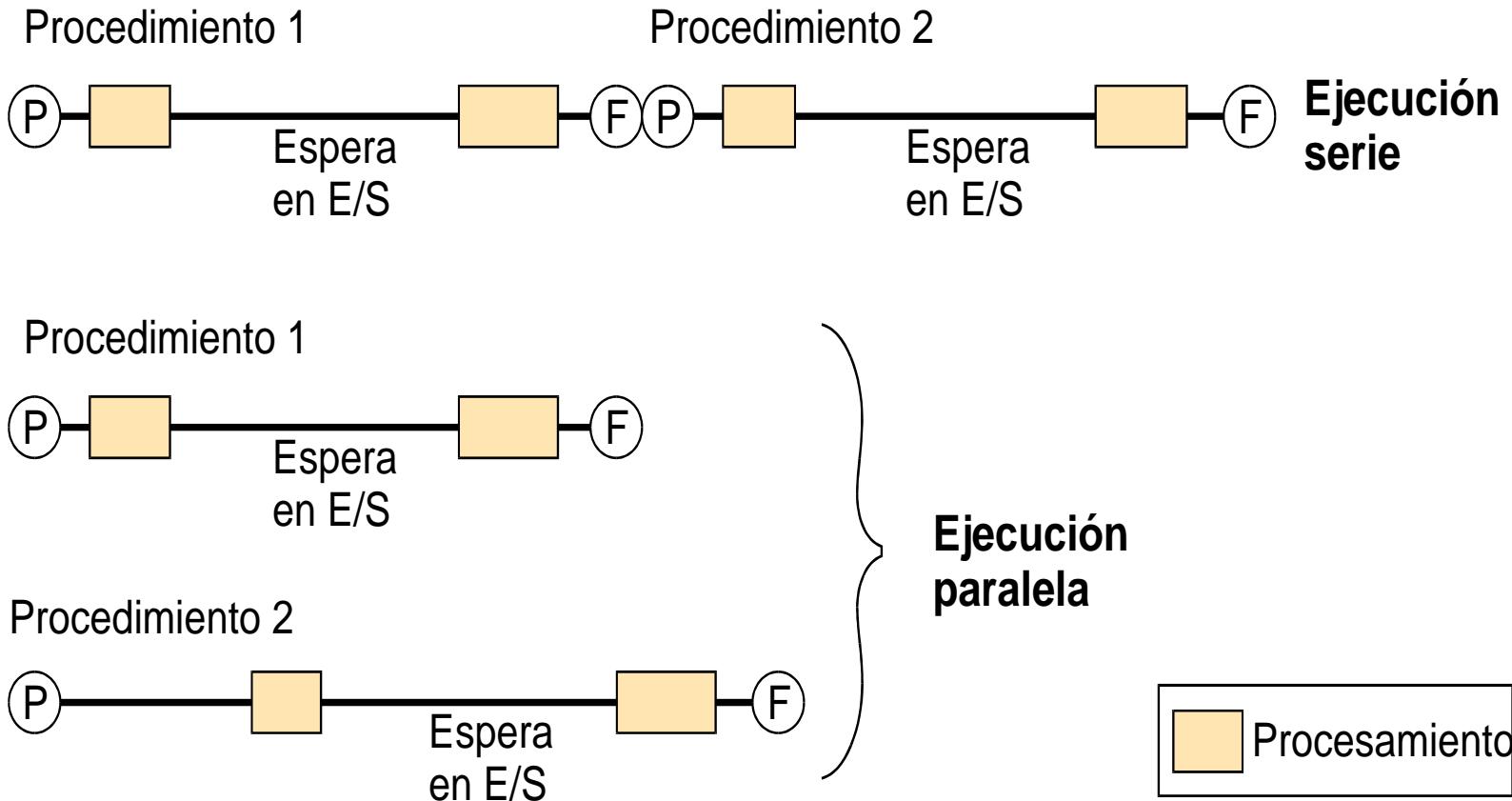




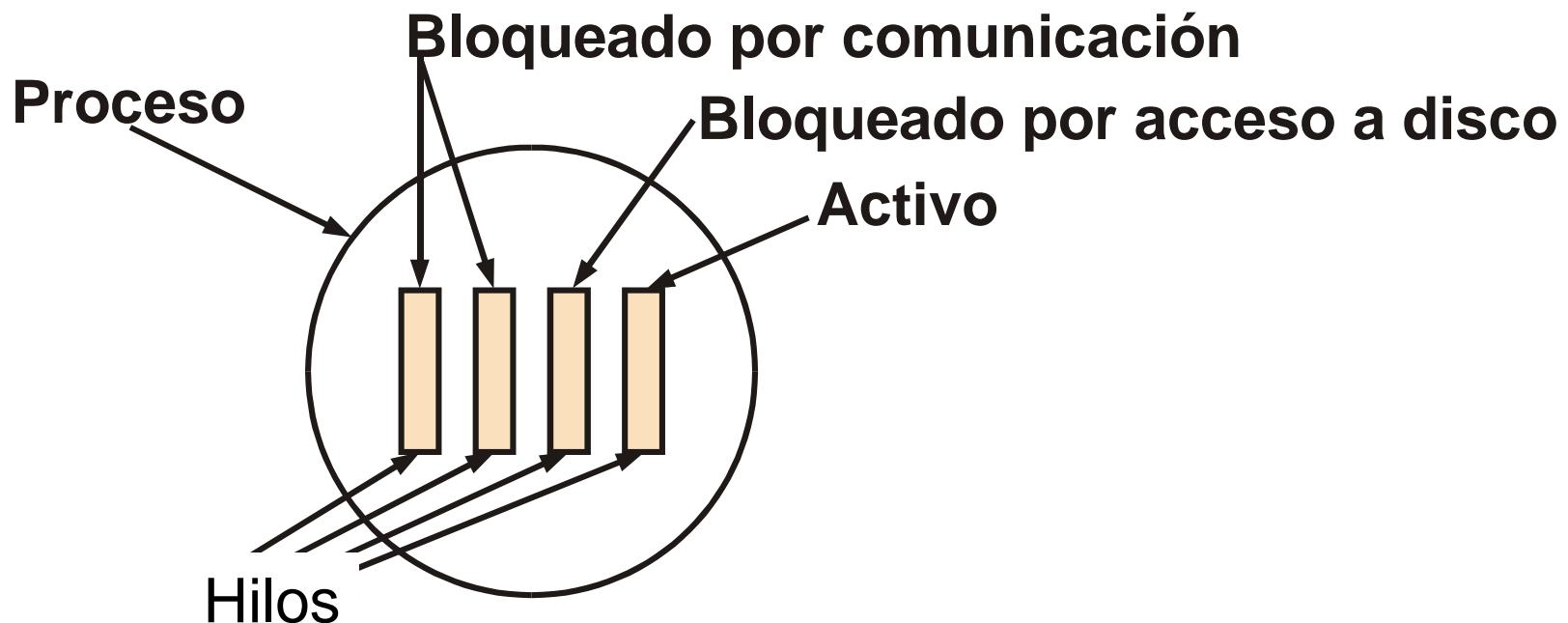
# Mono-hilo vs Multi-hilo



# Paralelización utilizando hilos



# Estados de un hilo





# Ventajas threads vs. procesos

- Tiempo de procesador para operaciones relacionadas con creación, destrucción, planificación y sincronización:
  - 10 hilos vs 100 proceso.
- El cambio de contexto entre hilos (de kernel) de un mismo proceso es menos costoso → No es necesario cambiar el espacio de direcciones “activo” de usuario
- Permiten compartir memoria entre ellos de forma fácil y eficiente
  - ¡¡Todos tienen el mismo espacio de direcciones!!

# Diseño con hilos



- Permite separación de tareas
- Paralelismo
  - Aumenta la velocidad de ejecución del trabajo
- Programación concurrente (memoria compartida)
  - Variables o estructuras de datos compartidas
  - Funciones reentrantes
    - Imaginar otra llamada al mismo código
  - Mecanismos de sincronización entre hilos (mutex, semáforos,...)
  - Variables globales
  - Simplicidad vs exclusión en el acceso

# Alternativas al diseño multihilo



- Proceso con un solo hilo
  - No hay paralelismo
    - Llamadas al sistema bloqueantes
  - Paralelismo gestionado por el programador
    - Llamadas al sistema no bloqueantes
- Múltiples procesos convencionales cooperando
  - Permite paralelismo
  - No comparten variables
  - Mayor sobrecarga de ejecución



# POSIX para la gestión de hilos

- **int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*func)(void \*), void \*arg)**
  - Crea un hilo que ejecuta "func" con argumento "arg" y atributos "attr".
  - Los atributos permiten especificar: tamaño de la pila, prioridad, política de planificación, etc.
  - Existen diversas llamadas para modificar los atributos.
- **int pthread\_join(pthread\_t thid, void \*\*value)**
  - Suspende la ejecución de un hilo hasta que termina el hilo con identificador "thid".
  - Devuelve el estado de terminación del hilo.
- **int pthread\_exit(void \*value)**
  - Permite a un hilo finalizar su ejecución, indicando el estado de terminación del mismo.
- **pthread\_t pthread\_self(void)**
  - Devuelve el identificador del thread que ejecuta la llamada.



# POSIX para la gestión hilos (II)

- **int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate)**
  - Establece el estado de terminación de un hilo.
  - Si *detachstate* = PTHREAD\_CREATE\_DETACHED el hilo liberará sus recursos cuando finalice su ejecución.
  - Si *detachstate* = PTHREAD\_CREATE\_JOINABLE no se liberarán los recursos, es necesario utilizar *pthread\_join()*.



# Programa de ejemplo

```
#include <stdio.h>
#include <pthread.h>
#define MAX_THREADS 10
void* func(void* arg)  {
    printf("Thread %d \n", pthread_self());
    pthread_exit(0);
}
int main(void)      {
    int j;
    pthread_attr_t attr;
    pthread_t thid[MAX_THREADS];
    pthread_attr_init(&attr);
    for(j = 0; j < MAX_THREADS; j++)
        pthread_create(&thid[j], &attr, func, NULL);
    for(j = 0; j < MAX_THREADS; j++)
        pthread_join(thid[j], NULL);
    return 0;
}
```

---

# Problemas de Sistemas Operativos

Dpto. ACyA, Facultad de Informática, UCM

## Modulo 3.1 – Procesos

---

### Problemas básicos

1.- Dado el siguiente programa, dibuje el esquema jerárquico de procesos creados:

```
void main(void){  
  
    int i;  
    for(i=0; i<3; i++)  
        fork();  
  
    ...  
}
```

2.- Considere el siguiente código:

```
int varGlobal;  
  
void main() {  
    int varLocal=3;  
    pid_t pid;  
  
    varGlobal=10;  
    printf("Soy el proceso original. Mi PID es %d\n", getpid());  
    fflush(NULL);  
  
    pid = fork();  
    if (pid == -1) {  
        perror("Error en fork()\n");  
        exit(-1);  
    }  
    if (pid == 0 ) {  
        // CODIGO DEL PROCESO HIJO  
        varGlobal = varGlobal + 5;  
        varLocal = varLocal + 5;  
    }  
    else {  
        // CODIGO DEL PADRE: pid contiene el pid del hijo  
        wait(NULL);  
        varGlobal = varGlobal + 10;  
        varLocal = varLocal + 10;  
    }  
    printf("Soy el proceso con PID %d. Mi padre es %d Global: %d Local %d\n",  
          getpid(), getppid(), varGlobal, varLocal );  
}
```

Asumiendo que el proceso original tiene PID=100 y es hijo del proceso `init` (PID=1), indica qué se mostrará por pantalla al ejecutar el código. ¿Es posible que los valores finales de las variables varíen de una ejecución a otra en función del orden de planificación? ¿Puede cambiar el orden en el que se muestran los diferentes mensajes por pantalla?

3.- Considere el siguiente código:

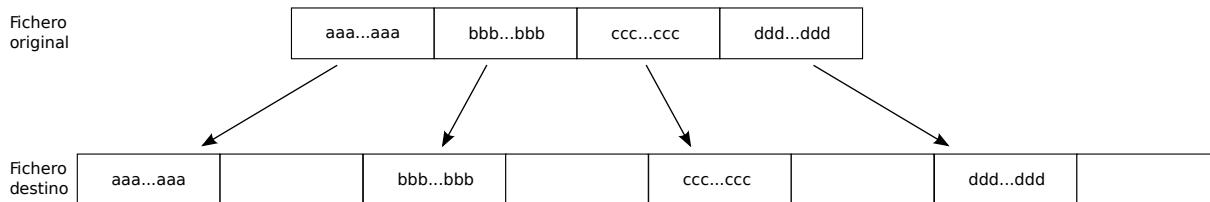
```

int a = 3;
void main() {
    int b=2;
    for (i=0;i<4;i++) {
        p=fork();
        if (p==0) {
            b++;
            execvp("comando" ...);
            a++;
        }
        else {
            wait();
            a++;
            b--;
        }
    }
    imprime(a,b);
}

```

- a) ¿Cuántos procesos se crean en total? (sin contar el padre original) ¿Cuántos coexisten en el sistema como máximo?
- b) ¿Qué se imprimirá al mostrar los valores de a y b?

4.- Un programador poco avezado pretende hacer una aplicación que realice copias intercaladas de ficheros en paralelo. El concepto de copia intercalada se ilustra en la figura: se copia el primer bloque íntegro, y se deja un bloque vacío. Se copia el segundo bloque del fichero origen al tercer bloque del destino, y así sucesivamente.



El código de su programa para la copia intercalada de un fichero de 4 bloques mediante 4 procesos, es el siguiente:

```

1 #define BLOCK 4096
2 char buf[BLOCK] = "xxxxxxxx...xxxxx";
4 void copia_bloque(int fdo, int fdd) {
5     read(fdo,buf,BLOCK);
6     write(fdd,buf,BLOCK);
7 }
9 void main() {
10 pid_t pid;
11 int fdo,fdd;
13 fdo = open("Origen",O_RDONLY);
14 fdd = open("Destino",O_RDWR|O_CREAT|
15             O_TRUNC,0666);
16 for (int i=0; i < 4; i++) {
17     lseek(fdo,i*BLOCK, SEEK_SET);
18     lseek(fdd,2*i*BLOCK,SEEK_SET);
20     pid = fork();
21     if (pid==0){
22         copia_bloque(fdo,fdd);
23         exit(0);
24     }
25 }
27 while (wait(NULL)!=-1) { };
29 read(fdd,buf,BLOCK);
30 lseek(fdd,0,SEEK_SET);
31 read(fdd,buf,BLOCK);
32 }

```

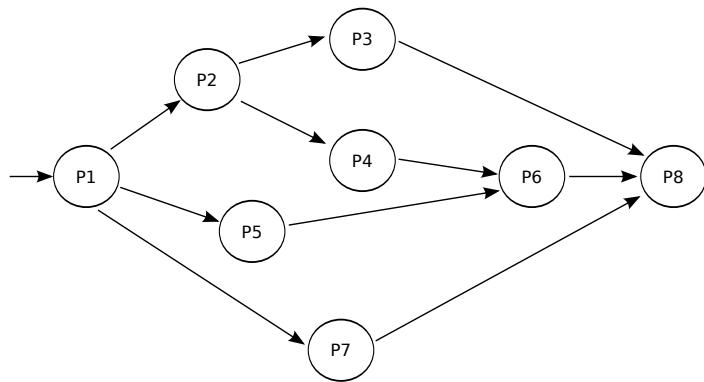
Responde a las siguientes preguntas, suponiendo que **la prioridad es para el proceso padre**, y después para cada uno de los hijos en el orden de creación.

- a) Indica el contenido del array `buf` inmediatamente después de la ejecución de las líneas 29 y 31. Justifica tu respuesta.
- b) Sea sistema de ficheros de tipo Linux (nodos-i), con 3 punteros directos y un indirecto simple, tamaño de bloque de 4KiB (4096 bytes) y 4bytes por puntero. Dibuja un posible

estado final de toda la información del sistema de ficheros relativa al fichero “Destino”. ¿Realiza el código la copia intercalada correctamente? Justifica tu respuesta.

- c) Escribe una versión correcta de la copia intercalada **paralela** que no haga suposiciones artificiales a cerca de la planificación.

5.- Use las llamadas `fork()`, `exec()`, `exit()` y `wait()` de UNIX para describir la sincronización de los ocho procesos cuyo grafo general de precedencia es el siguiente. Para ello escriba un función `main()` en la que se vayan creando los 8 procesos mediante llamadas a `fork()`, se ejecuten los binarios asociados a cada proceso (por ejemplo, `p1.out` para el proceso `p1`, etc.) y se respete la precedencia mostrada en la figura (por ejemplo, `p6` no puede comenzar hasta que no hayan acabado `p4` y `p5`).



6.- Considere el siguiente código:

```

int fd = -1;
char buf1[4] = "aaaa";
char buf2[4] = "bbbb";

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, h1, NULL);
    pthread_create(&tid2, NULL, h2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    close(fd);
}

void* h1() {
    fd = open("prueba", O_RDWR | O_CREAT | O_TRUNC,
              , 0666);
    write(fd, buf1, 4);
}

void* h2() {
    while (fd == -1) {};
    write(fd, buf2, 4);
}
  
```

Indique si cada una de las siguientes afirmaciones es cierta o falsa justificando la respuesta:

- El hilo 2 (función `h2`) nunca saldrá del bucle `while` inicial.
- La escritura del hilo 2 (función `h2`) producirá un error por no haber abierto antes el fichero.
- El contenido final del fichero prueba será o bien, `aaaa` o bien `bbbb`; ninguna otra alternativa es posible.
- La llamada a `close()` del programa principal no debería devolver `-1` (esto es, no debería producir ningún error).

7.- En un sistema monoprocesador los siguientes trabajos llegan a procesarse en los instantes indicados. ¿Cuáles son los tiempos de retorno (*turnaround*) y de espera para cada uno de ellos, los tiempos de retorno y de espera promedios, así como la productividad (*throughput*) del sistema, aplicando las diferentes estrategias de planificación listadas? Aplique los algoritmos de planificación sin expropiación y base las decisiones en la información de que se dispone en el momento de tomarlas.

Trabajo	Llegada	CPU	E/S	CPU	E/S
1	0	1	5	1	
2	1	3	1	1	1
3	0	5	4	1	
4	3	3	2	1	1

- a) FCFS: Primero en llegar, primero en pasar (sin expropiación)
- b) SJF: Primero el de menor tiempo de CPU siguiente
- c) RR: Prioridad circular con cuanto = 3
- d) RR: Prioridad circular con cuanto = 1

**8.-** Cinco trabajos de lotes, de *A* a *E*, llegan casi al mismo tiempo a un centro de cálculo dotado de un único procesador. La estimación de sus respectivos tiempos de ejecución es de 10, 6, 2, 4, y 8 minutos. Sus prioridades, determinadas externamente, son 3, 5, 2, 1 y 4, respectivamente, siendo 5 la prioridad superior. Para cada uno de los siguientes algoritmos de planificación determine el tiempo medio de retorno de los procesos. (Ignorar el tiempo de comutación de procesos).

- a) Por turnos (**round-robin**) con  $q \cong 0$ , es decir, empleando la política de compartición del procesador y midiendo los tiempos al final de los turnos completos.
- b) Por prioridad estricta.
- c) Por orden de llegada (FCFS)
- d) Por menor tiempo de ejecución (SJF)

**9.-** Considere un sistema monoprocesador con una política de planificación de procesos de 3 niveles con realimentación, de forma que los procesos que agotan el cuanto bajan de nivel y los que ceden la CPU antes de agotarlo son promocionados. Cada nivel usa a su vez una política de planificación circular **round robin** cuyos cuantos de tiempo son 2, 4 y 8, respectivamente. Al principio hay 3 procesos en la cola del nivel 1 (máxima prioridad). Los patrones de ejecución de los procesos son los siguientes (y se repiten indefinidamente):

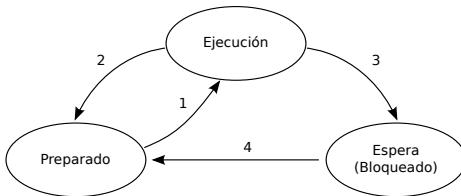
P1 (3-CPU,5-E/S)   P2 (8-CPU,5-E/S)   P3 (5-CPU,5-E/S)

Las colas de los otros dos niveles están vacías. Cuando acaba una operación de E/S, los procesos entran en la cola de mayor prioridad. Usando un diagrama de tiempos muestre:

- a) qué proceso está ejecutándose y
- b) qué procesos hay en cada nivel, durante las 30 primeras unidades de tiempo de ejecución. Calcule además la utilización de CPU y los tiempos de espera de cada proceso.

## Problemas adicionales

**10.-** Podemos describir gran parte de la gestión del procesador en términos de diagramas de transiciones de estado como éste:



- a) ¿Qué 'evento' causa cada una de las transiciones marcadas?
- b) Si consideramos todos los procesos del sistema, podemos observar que una transición de estado por parte de un proceso podría hacer que otro proceso efectuara una transición también. ¿Bajo qué circunstancias podría la transición 3 de un proceso provocar la transición 1 inmediata de otro proceso?
- c) ¿Bajo qué circunstancias, si las hay, podrían ocurrir las siguientes transiciones causa-efecto?
  - a.-  $2 \rightarrow 1$  (por el hecho de que ocurra una transición tipo 2, hay una transición 1)
  - b.-  $3 \rightarrow 2$
  - c.-  $4 \rightarrow 1$
- d) ¿Bajo qué circunstancias, si las hay, las transiciones 1, 2, 3 y 4 NO producirían ninguna otra transición inmediata?

**11.-** Considere un sistema monoprocesador con una planificación MLF (multinivel con realimentación) donde el número de niveles es  $n = 10$ , y el intervalo de planificación para el nivel  $i$  es  $T_i = 2i \cdot q$ , siendo  $q$  es el valor del intervalo básico de planificación o **quanto**. En nuestro sistema sólo hay tres procesos, se encuentran inicialmente en la cola de máxima prioridad T1 y sus tiempos respectivos hasta la siguiente petición de E/S son 3, 8, y 5 quantos. Cuando un proceso alcanza su petición de E/S permanece suspendido (sin competir por la CPU) durante 5 unidades de tiempo, tras las cuales vuelve a entrar en la cola de máxima prioridad. Los tiempos de CPU requeridos hasta la petición de E/S siguiente son de nuevo 3, 8, y 5. Usando un diagrama de tiempos, muestre

- a) Qué proceso estará ejecutándose y qué procesos estarán en qué cola durante cada una de las primeras 30 unidades de tiempo de ejecución.
- b) Cuáles son los valores de las siguientes medidas de rendimiento: productividad, tiempos retorno (**turnaround**) individual y promedio, y tiempos de espera individual y promedio.

**12.-** Repita el ejercicio anterior pero suponiendo que  $T_i = 2 \cdot q$  (es decir, constante), para todos los niveles de prioridad.

**13.-** Se tiene un proceso productor y otro consumidor. El proceso consumidor debe crear al proceso productor y asegurarse de que está vivo. El proceso consumidor llama a una función `leer_entero()`, que devuelve un valor entero, generado por el productor, y lo guarda en un archivo (`datos`). Si el valor leído es 0, es señal de que el productor ha muerto por algún motivo, por lo que deberá crear otro proceso productor. El proceso consumidor debe leer un valor cada 10 segundos, cuando haya leído 100 valores, deberá matar al productor y acabar la ejecución.

Suponiendo que tanto el proceso productor como la función `leer_entero()` están ya escritos, se pide programar el proceso consumidor en C, resaltando claramente las llamadas al sistema. Opcionalmente se puede hacer en pseudo-código (manteniendo al menos la nomenclatura y los parámetros para las llamadas al sistema).

**14.-** Sea un sistema operativo para entornos monoprocesador que sigue el modelo cliente-servidor en el que existe un proceso servidor **SF** (Sistema de Ficheros) y un proceso **CD** (Controlador de Disco). Las prioridades de los procesos **SF** y **CD** son mayores que la de los procesos de usuario. En un determinado instante la cola de **PREPARADOS** de este sistema contiene dos procesos de usuario **A** y **B**, en dicho orden. Las características de ejecución de **A** y **B** son las siguientes: Se

Proceso A: 160 ms CPU, 50 ms E/S de disco, 50 ms CPU  
Proceso B: 20 ms CPU, 50 ms E/S de disco, 60 ms CPU

pide construir un diagrama de tiempos donde se muestre, a partir del instante en que aparecen los procesos **A** y **B** en el sistema, los estados de estos dos procesos (**EJECUCIÓN**; **PREPARADO**; **BLOQUEADO**). Las operaciones de E/S de disco conllevan una petición por parte del usuario al **SF**, que a su vez realizará una petición al proceso **CD**. Cuando los datos solicitados al **CD** estén listos, éste avisará al proceso invocante (**SF**) que a su vez notificará al proceso de usuario. Se supone que el tiempo de ejecución de los procesos **CD** y **SF** es despreciable. El quanto de planificación aplicado a los procesos de usuario es de 100 ms.



# Sistemas Operativos

Grado en Ingeniería Informática  
2018-2019

## Planificación de procesos e hilos

Basado en:

Sistemas Operativos  
J. Carretero [et al.]



# Contenido

- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP



# Planificación de procesos/hilos

- Objetivos:
  - Optimizar uso de las CPUs
  - Minimizar tiempo de espera
  - Ofrecer reparto equitativo (justicia)
  - Proporcionar grados de urgencia (prioridades)
- Tipos de planificación
  - No expropiativa: el proceso conserva la CPU hasta que se bloquea, la cede expresamente o termina su ejecución.
  - Expropiativa: el SO expulsa al proceso de la CPU
    - Exige un reloj que interrumpe periódicamente
- Colas de procesos/hilos (*run queues*)

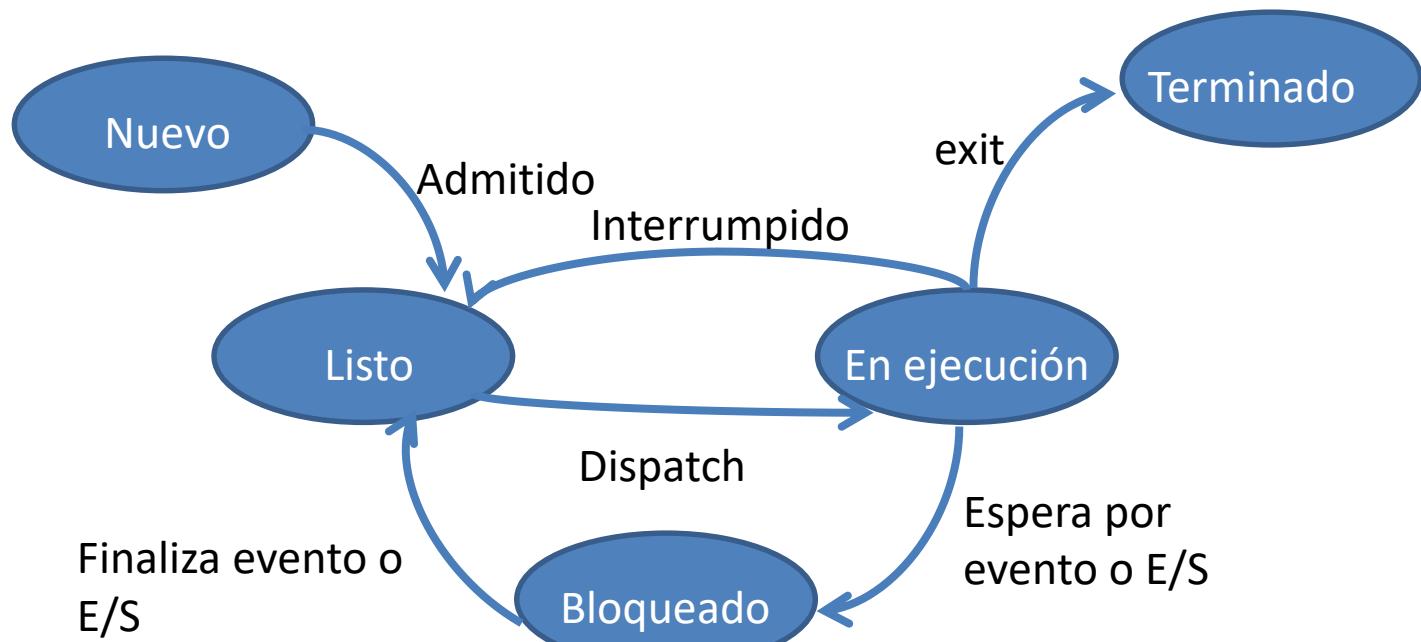


# Métricas de un planificador

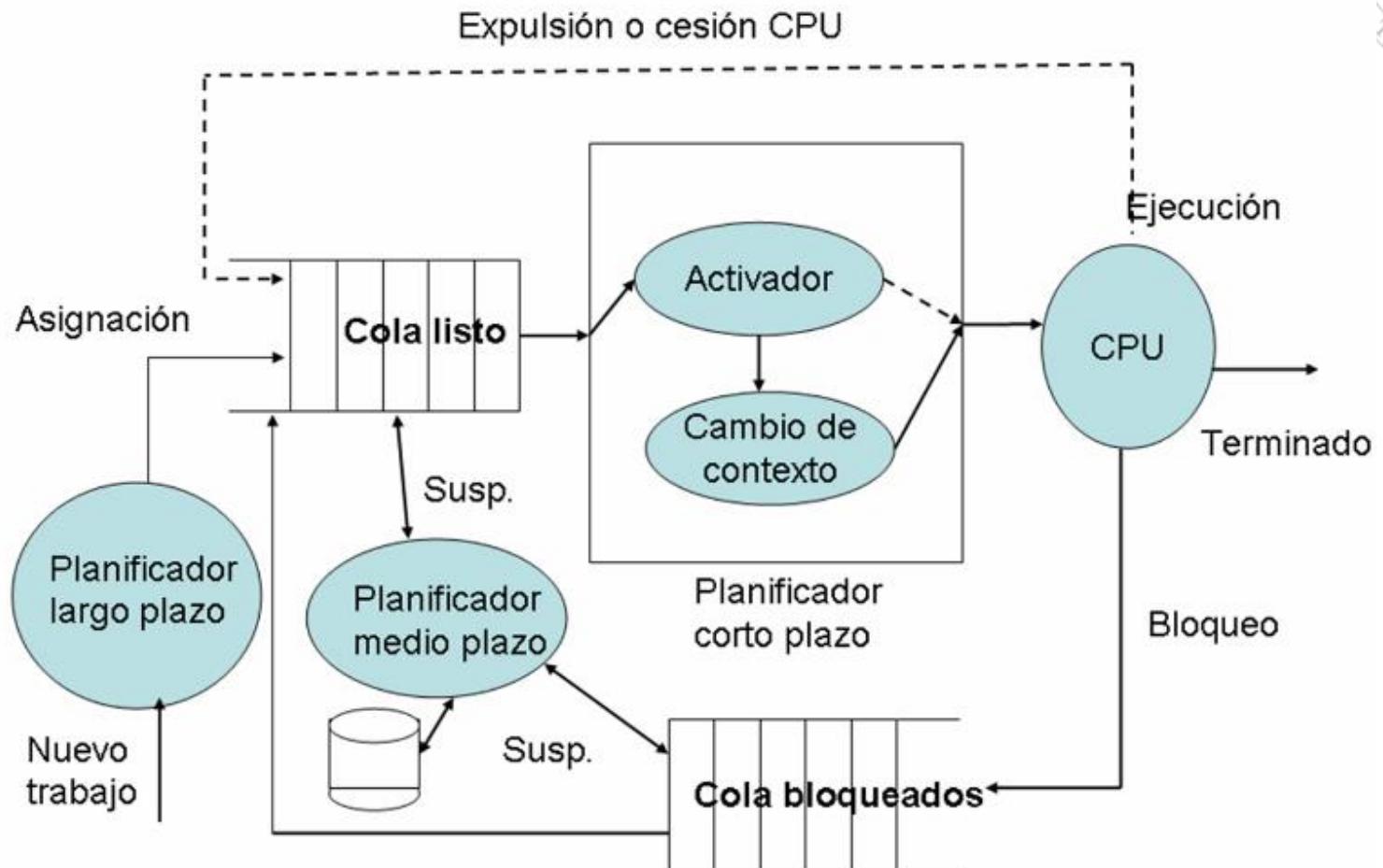
- Parámetros por entidad (proceso o *thread*)
  - Tiempo de ejecución: creación - terminación
  - Tiempo de espera: tiempo total listo y sin CPU
  - Tiempo de respuesta: creación – 1<sup>er</sup> uso de CPU
- Parámetros globales
  - Porcentaje de utilización del procesador
  - Productividad: número de trabajos completados por unidad de tiempo

# Tipos de planificadores

- Planificador: selecciona el proceso
  - A largo plazo: añadir procesos a ejecutar (batch)
  - A medio plazo: añadir procesos a RAM
  - A corto plazo: qué proceso tiene la CPU
  - Planificación de E/S
- Activador o Dispatcher: da control al proceso (cambio de contexto)



# Estructuras del planificador





# Puntos de activación del planificador

- Periódicamente (interrupción del temporizador de la CPU)
- Como resultado del procesamiento de alguna interrupción generada por otros dispositivos de E/S
- El proceso en ejecución causa una excepción que lo bloquea (fallo de página) o fuerza su terminación (violación de segmento)
- Cuando el proceso en ejecución termina
- El proceso realiza una llamada bloqueante
- Cesión voluntaria del procesador
  - `sched_yield()`
- Se desbloquea un proceso más “importante” que el actual
  - Cambios de prioridad en el sistema de procesos



# Contenido

- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP



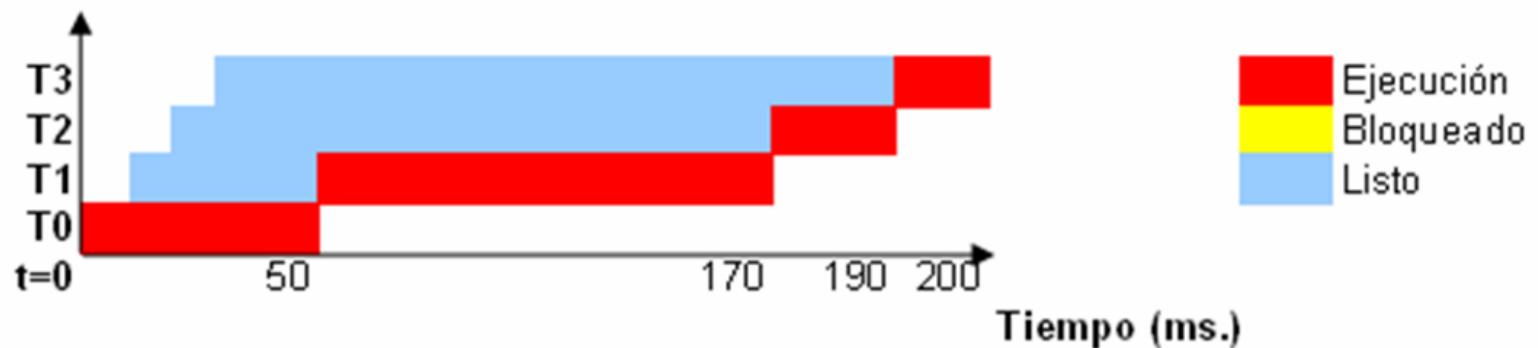
# Algoritmos no expropiativos

- El planificador no quita la CPU al proceso una vez que está en ejecución, a no ser esta la ceda voluntariamente, termine o se bloquee por E/S
- Algoritmos
  - Primero en llegar primero en ejecutar FCFS (First-Come First-Served)
  - Primero el trabajo más corto SJF (*Shortest Job First*)
  - Planificación basada en prioridades

# Primero en llegar primero en ejecutar FCFS (I)

- Muy sencillo y óptimo en uso de CPU

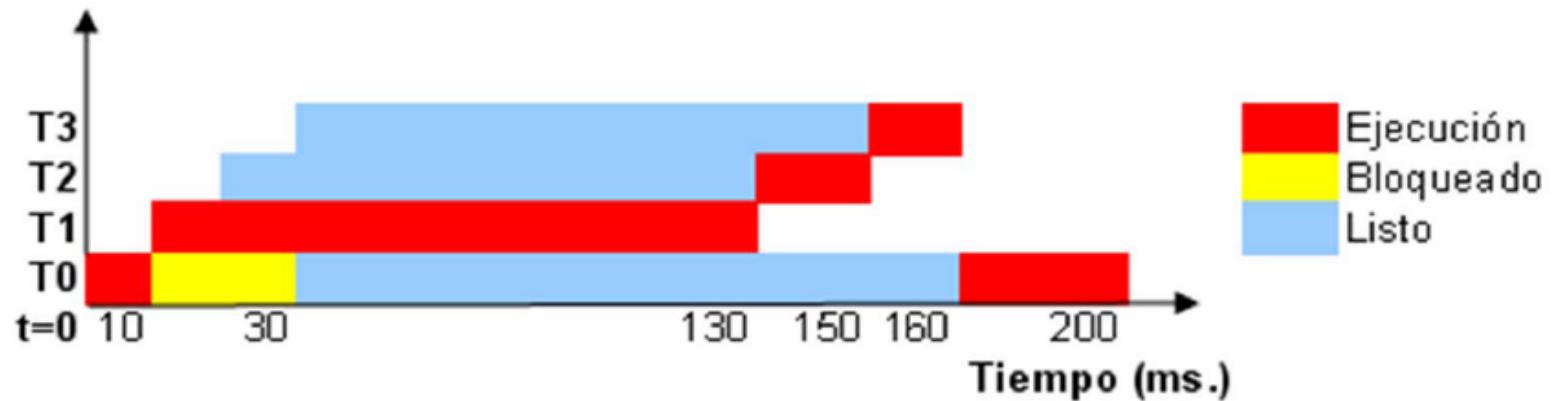
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



# Primero en llegar primero en ejecutar FCFS (II)

- Programas con E/S son encolados al final
- Programas largos afectan al sistema

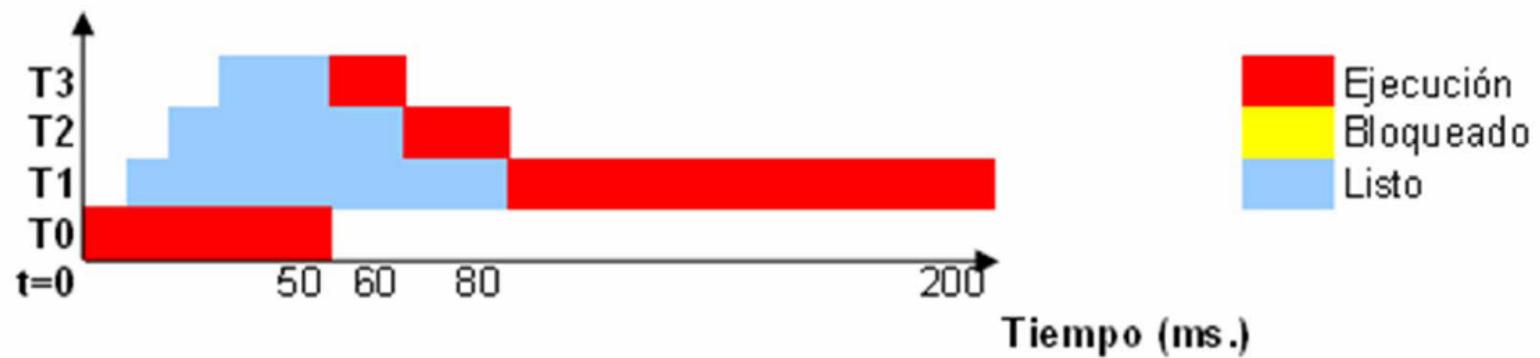
Proceso o <i>thread</i>	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



# Primero el trabajo más corto SJF

- Bueno para programas interactivos
- Necesita conocer el perfil de las tareas
- Problemas de inanición

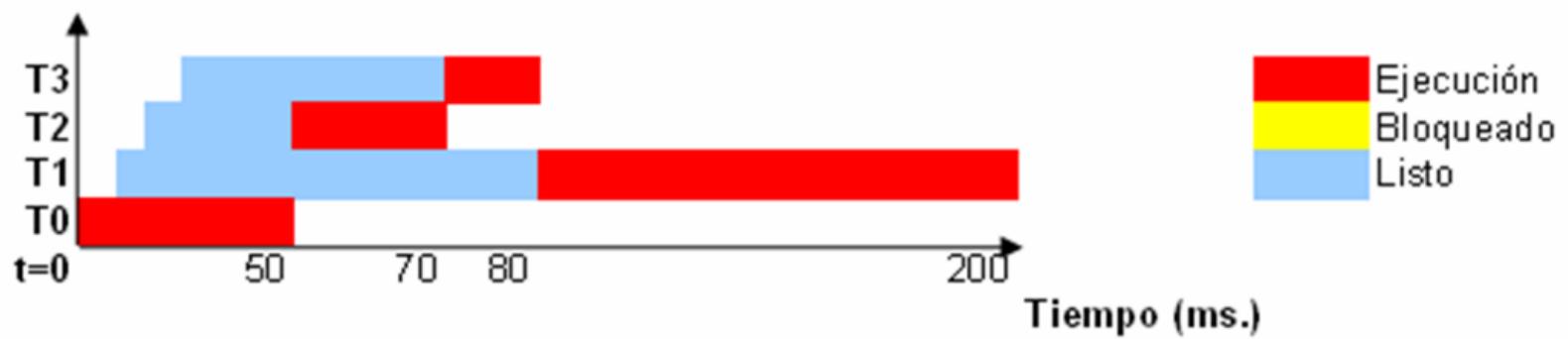
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



# Prioridades

- Bueno para sistemas con grados de urgencia
- Problema de inanición:
  - Aumento de la prioridad con la edad

Proceso o <i>thread</i>	Instante de llegada	Tiempo de procesador (ms)	Prioridad
T0	0	50	4
T1	10	120	3
T2	20	20	1
T3	30	10	2





# Algoritmos expropiativos

- No expropiativos no son adecuados para SSOO de propósito general
  - Mezcla de trabajos interactivos y trabajos intensivos en CPU
- Algoritmos expropiativos
  - *Round Robin* (turno rotatorio)
  - Primero el de menor tiempo restante (SRTF)
    - Shortest Remaining Time First
  - Prioridad expulsiva
  - Colas multinivel



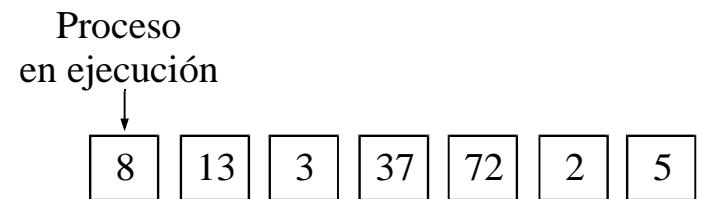
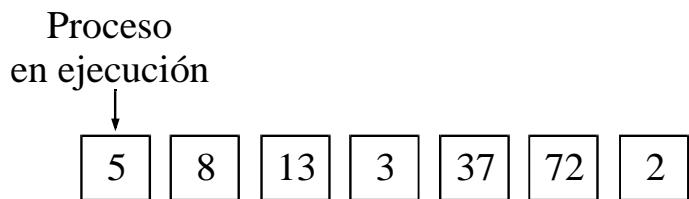
# Algoritmos expropiativos

- La planificación se realiza dividiendo el tiempo de CPU en rodajas llamadas *quanto* o *timeslice*
- El planificador se activa periódicamente
  - El temporizador de la CPU se configura para generar interrupciones cada *tick* ( $\sim ms$ )
  - El *timeslice* se expresa en *ticks* (Dependiente de SO)
  - Cada *tick*, el planificador realiza *CPU accounting*
    - Ej: incrementar el contador de ticks que el hilo ha usado desde que entró a la CPU
  - Cuando el planificador lo considera pertinente, cambia el proceso/hilo que hay ejecutando por otro



# Round Robin (I)

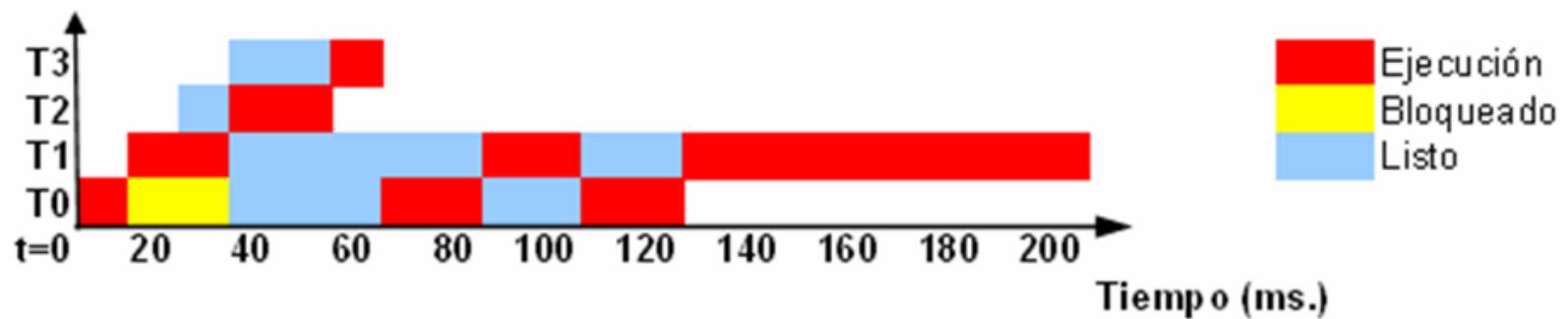
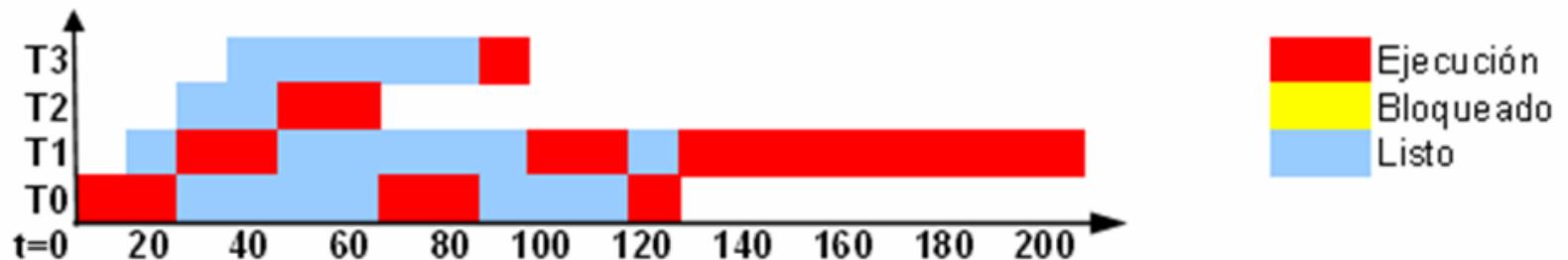
- FCFS + timeslice
- Asignación de procesador rotatoria
- Uso en sistemas de tiempo compartido
  - Equitativo (mejor hacerlo por *uid* y no por proceso)
- Se asigna un tiempo máximo de procesador que el proceso puede consumir sin ser expropiado (*timeslice*)





# Round Robin (II)

Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10

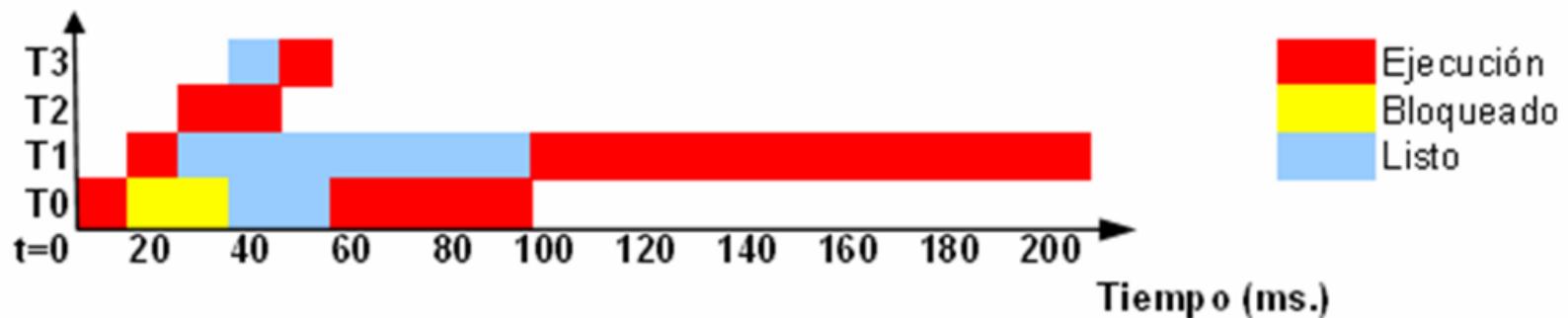


# Primero el de menor tiempo restante SRTF

## SJF + *timeslice variable*

- Bueno para programas interactivos
- Necesita conocer el perfil de las tareas
- Problemas de inanición

Proceso o <i>thread</i>	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10

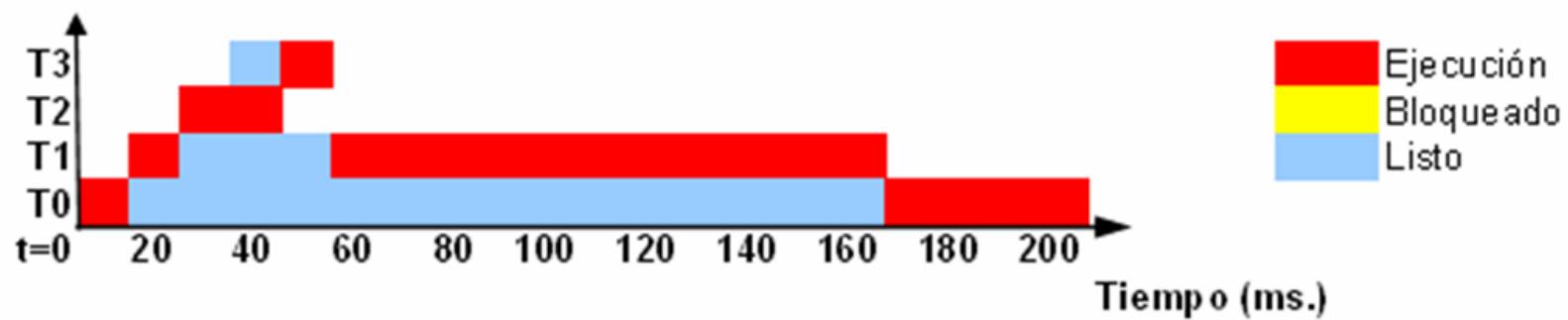




# Experiencia basado en prioridades

- Bueno para sistemas con grados de urgencia
- Problema de inanición:
  - Aumento de la prioridad con la edad

Proceso o thread	Instante de llegada	Tiempo de procesador (ms)	Prioridad
T0	0	50	4
T1	10	120	3
T2	20	20	1
T3	30	10	2

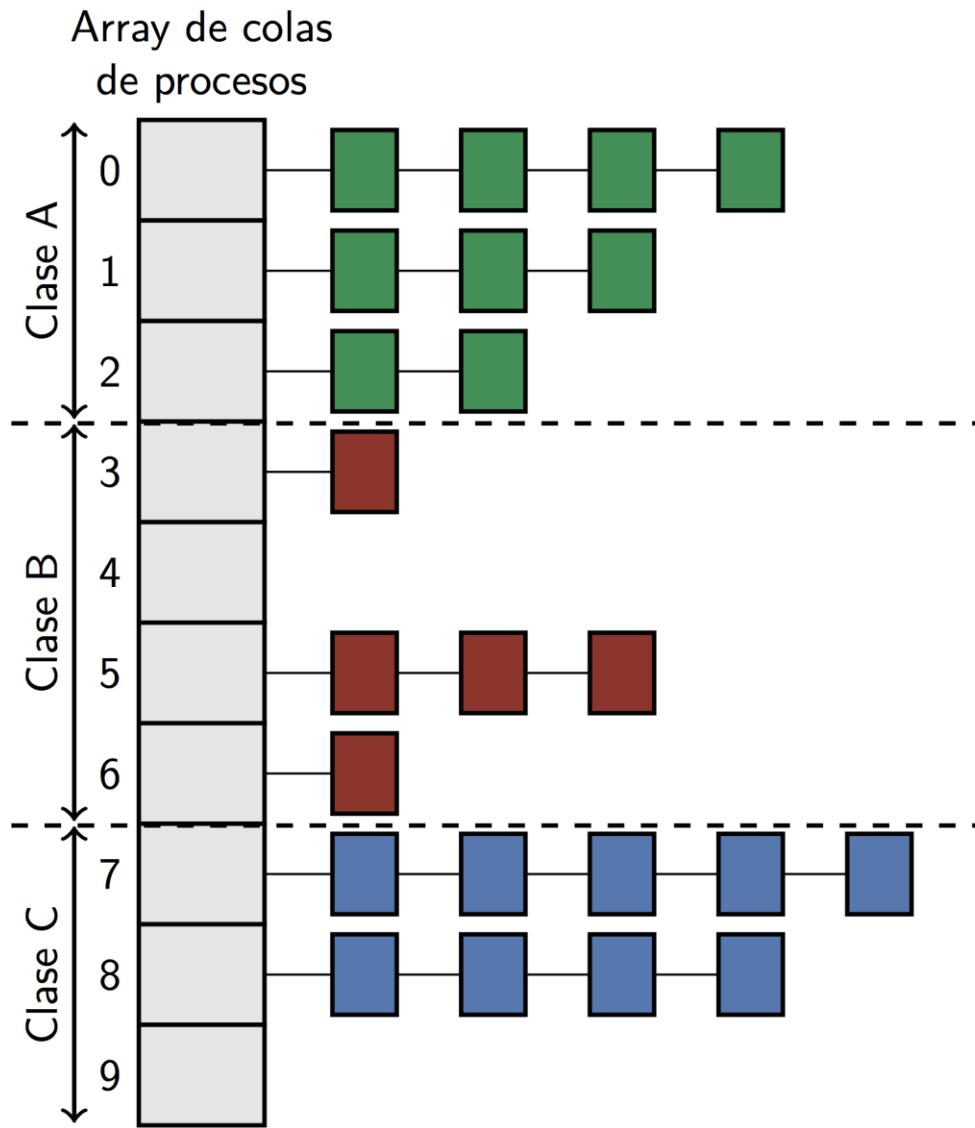


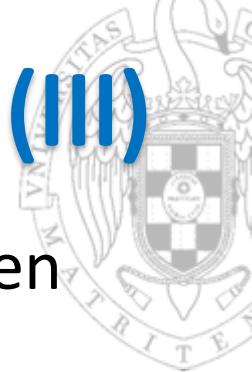


# Planificación con colas multinivel (I)

- Objetivo: dar soporte a distintas clases de procesos
- En el sistema existen  $k$  niveles de prioridad, cada uno con una cola de procesos asociada
  - El SO gestiona realmente un array de colas de procesos
  - En cada nivel de prioridad puede haber un *timeslice* diferente
- Los niveles de prioridad se agrupan en rangos para dar servicio a distintos tipos de procesos/hilos
  - Tiempo real
  - Hilos de sistema
  - Interactivos
  - *Batch*
  - ...

# Planificación con colas multinivel (II)





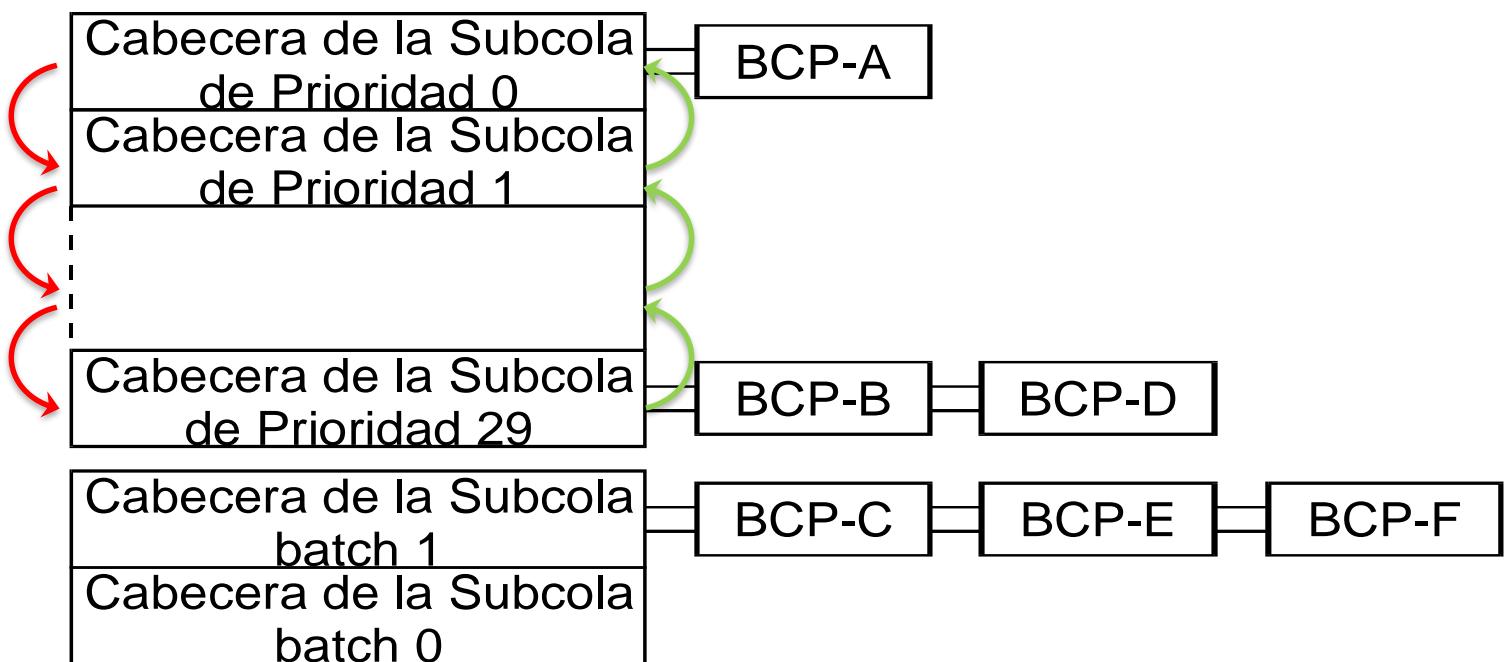
# Planificación con colas multinivel (III)

- Las colas dentro de un rango de prioridades pueden gestionarse de dos formas:
  - **Sin realimentación** (procesos con prioridad fija)
    - Proceso en la misma cola durante toda su vida
  - **Con realimentación** (procesos con prioridad dinámica)
    - Los procesos pueden cambiar de nivel
      - El cambio de nivel sólo se produce dentro del rango de prioridades gestionado por el “planificador local”
    - Necesario definir *política de cambio de nivel*
      - Ejemplo: Política para favorecer a procesos interactivos
        - Si proceso agota su timeslice, baja de nivel
        - Si proceso no agota su timeslice (p. ej., bloqueo E/S), sube de nivel

# Colas de procesos

*quanto  
agotado*

*quanto no  
agotado*





# Contenido

- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP



# Planificador Linux (v3.14)

- 140 niveles de prioridad (0 → más prioridad)
  - 100 para procesos *real-time* (Deadline, RR y FIFO)
  - 40 para procesos *normales* (CFS)
- Objetivos CFS (Completely Fair Scheduler)
  - Aproximar planificación completamente justa
    - No se asigna un *timeslice* por prioridad sino un *porcentaje* del procesador
  - Proporcionar buenos tiempos de respuesta
    - Entornos interactivos (GUIs)



# Idea planificador CFS

- Repartir el tiempo de CPU entre hilos de forma similar a lo que ocurriría si pudieran ejecutarse simultáneamente
- Si hay 4 hilos en el sistema ejecutándose durante 10ms de tiempo, lo *justo* sería que cada uno se hubiese ejecutado 2.5ms en ese período
  - ¿Y si los hilos tienen diferentes pesos?



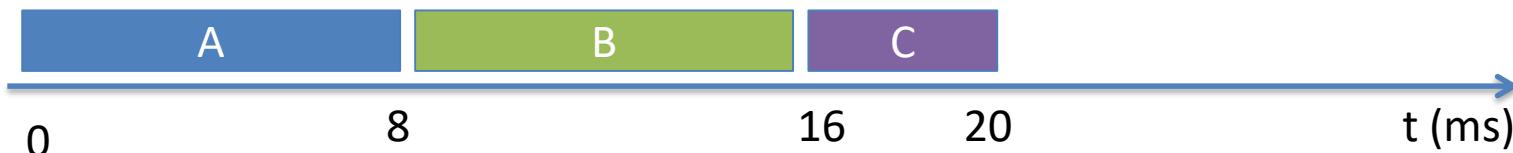
# Ejemplo planificación CFS

- Consideremos estas 3 tareas (CPU bound) durante 20ms

Tarea	Peso
A	4
B	4
C	2

- Tarea A debe ejecutarse un 40% del tiempo total (8ms)

$$T_{CPU}(P) = \text{sched\_period} \times \frac{\text{peso}(P)}{\sum_{i=1}^n \text{peso}(i)} \rightarrow T_A = 20\text{ms} * (4/10) = 8\text{ms}$$





# Parámetros CFS

- *sysctl\_sched\_latency* (*sched\_period*)
  - Período durante el cual cada tarea activa se debería planificar al menos una vez
  - $\text{timeslice}_{\text{proc. } n} \sim \text{sched period} * \text{peso de proc. } n / \sum \text{(pesos proc. activos)}$
- Peso de una tarea proporcional a su prioridad (*nice*)
  - Existe una tabla de conversión prioridad <-> peso
  - Una diferencia de 1 en dos valores de *nice* supone un cambio de 10% en el reparto de CPU
- *sysctl\_sched\_min\_granularity*
  - Tiempo mínimo que podrá ejecutar cualquier tarea antes de ser expropiado

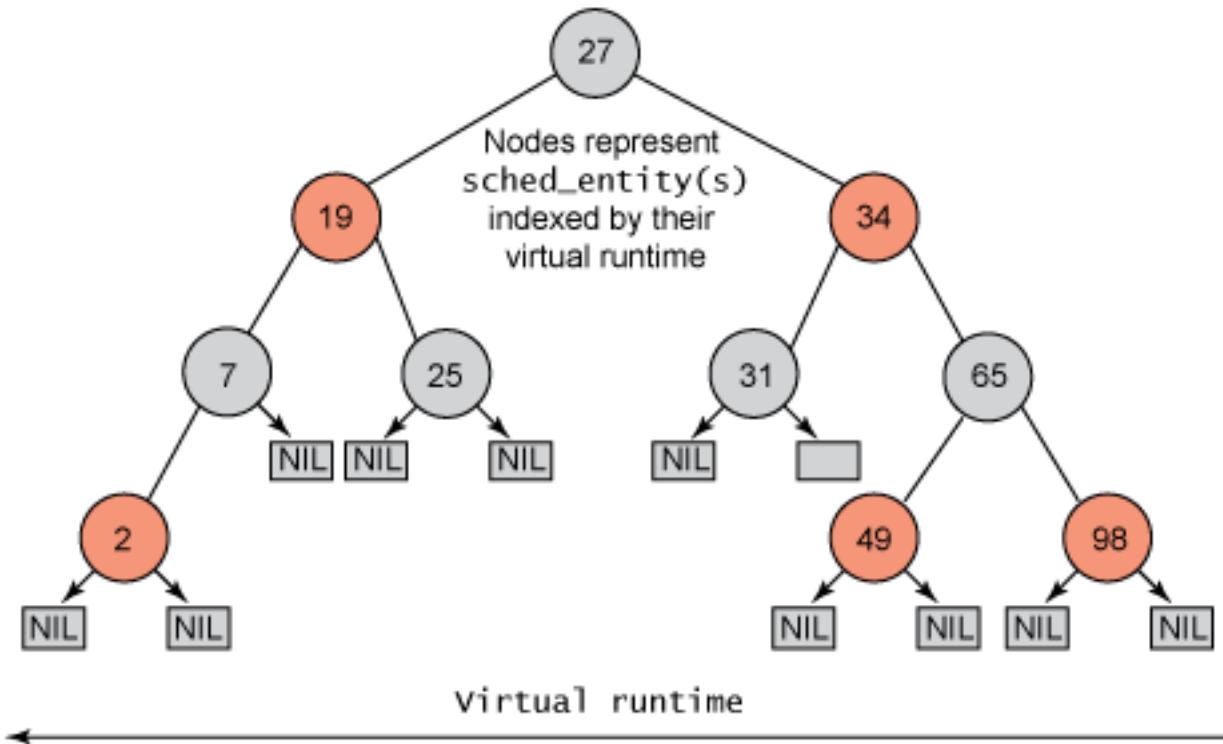


# CFS: siguiente tarea para ejecutar

- El hilo actual puede abandonar la CPU por...
  - Haber agotado su *slice*
  - Abandonar voluntariamente (E/S, *yield...*)
  - Otro hilo se *merece* más la CPU
- Siguiente tarea: la de menor *virtual runtime*
  - No hay colas de prioridades
  - El tiempo de CPU virtual transcurre más rápidamente para procesos de menor prioridad y más lentamente para los de mayor prioridad
  - Justicia: que todos los procesos reciban el mismo *vruntime*

# CFS

- Cola de ejecución-> Red-black tree
  - Se ejecuta el proceso de *más a la izquierda*
  - Árbol balanceado: operaciones  $O(\log N)$





# Interacción con el planificador

- *nice* permite ejecutar un comando con una prioridad entre -19 y 20
  - Recuerda: 40 niveles de prioridad
  - man nice ( o, mejor, info nice)
- *renice* permite alterar la prioridad de un proceso en ejecución
- *sysctl* permite consultar/modificar parámetros del kernel
  - sysctl -A | grep sched



# Comprobación CFS

- Creamos dos procesos *CPU bound* y forzamos a que se ejecuten en el core 0

```
> taskset -c 0 dd if=/dev/zero of=/dev/null &  
> taskset -c 0 dd if=/dev/zero of=/dev/null &
```

- Hacemos *top* para comprobar el reparto de CPU
  - Deberían estar al 50%
- Bajamos la prioridad a uno de ellos

```
> su  
> renice 1 <pid>
```

- Hacemos *top* nuevamente



# Contenido

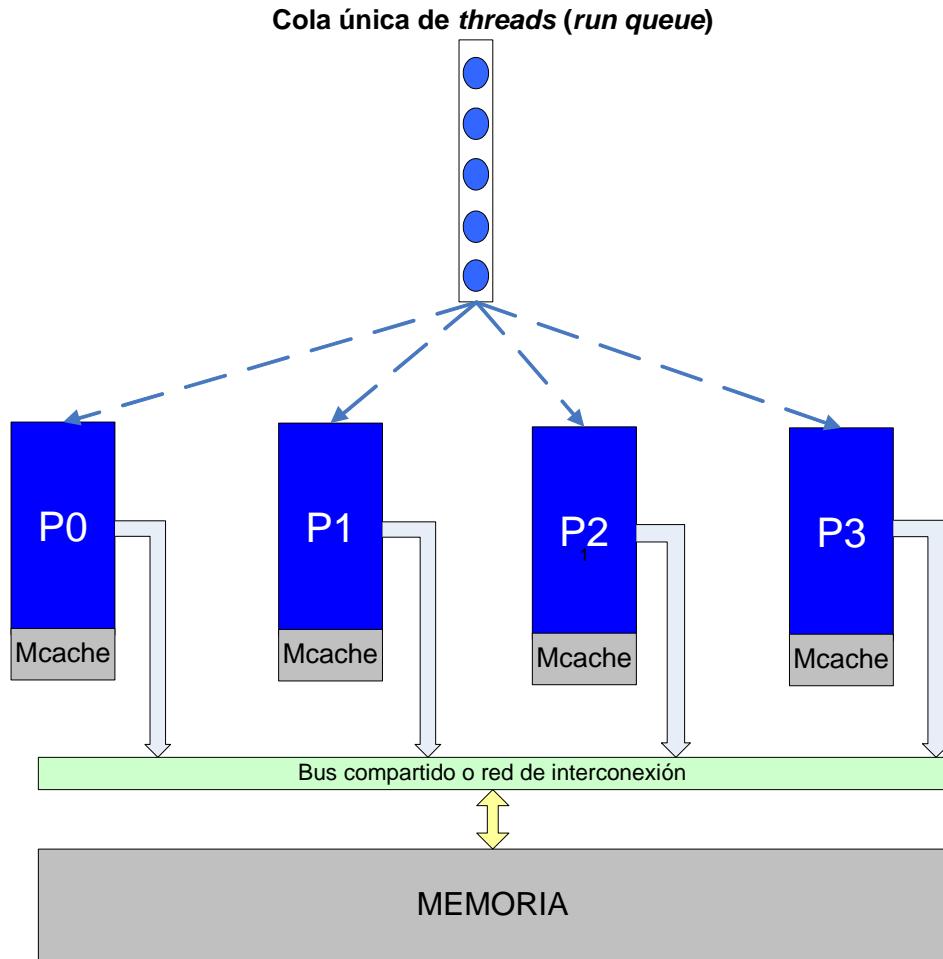
- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP



# Planificación SMP

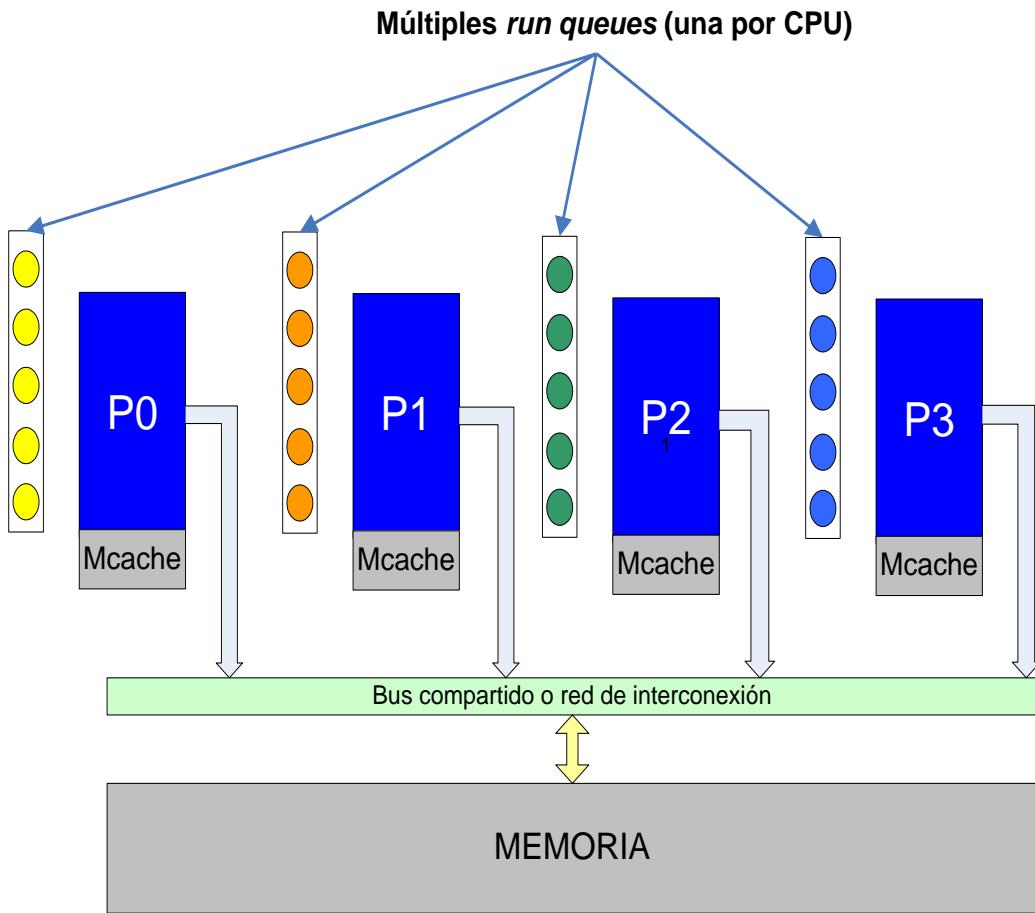
- Garantizar un equilibrio de carga
  - Que no haya un *procesador* ocioso y otros con mucha carga de trabajo
- Tener en cuenta la afinidad de procesos y procesadores
  - Importante al replanificar un proceso (evitar migrar)
- Tener en cuenta la compartición de datos entre procesos/hilos si hay varios nodos de memoria (NUMA)
  - Si dos hilos comparten memoria, probablemente sea bueno que comparten todo lo posible su nivel de jerarquía

# Planificación SMP (Linux v2.4)



- Una única *run queue* para todos los procesadores
- Bueno para el *load balancing*
  - Todos los procesadores tienen, potencialmente el mismo trabajo
- Malo para la *afinidad*
  - El proceso A se ejecutó en la CPU1 y luego se le envía CPU2 → migración
  - ¿Qué supone una *migración*?
- Problemas de escalabilidad

# Planificación SMP (Linux v2.6.x+)

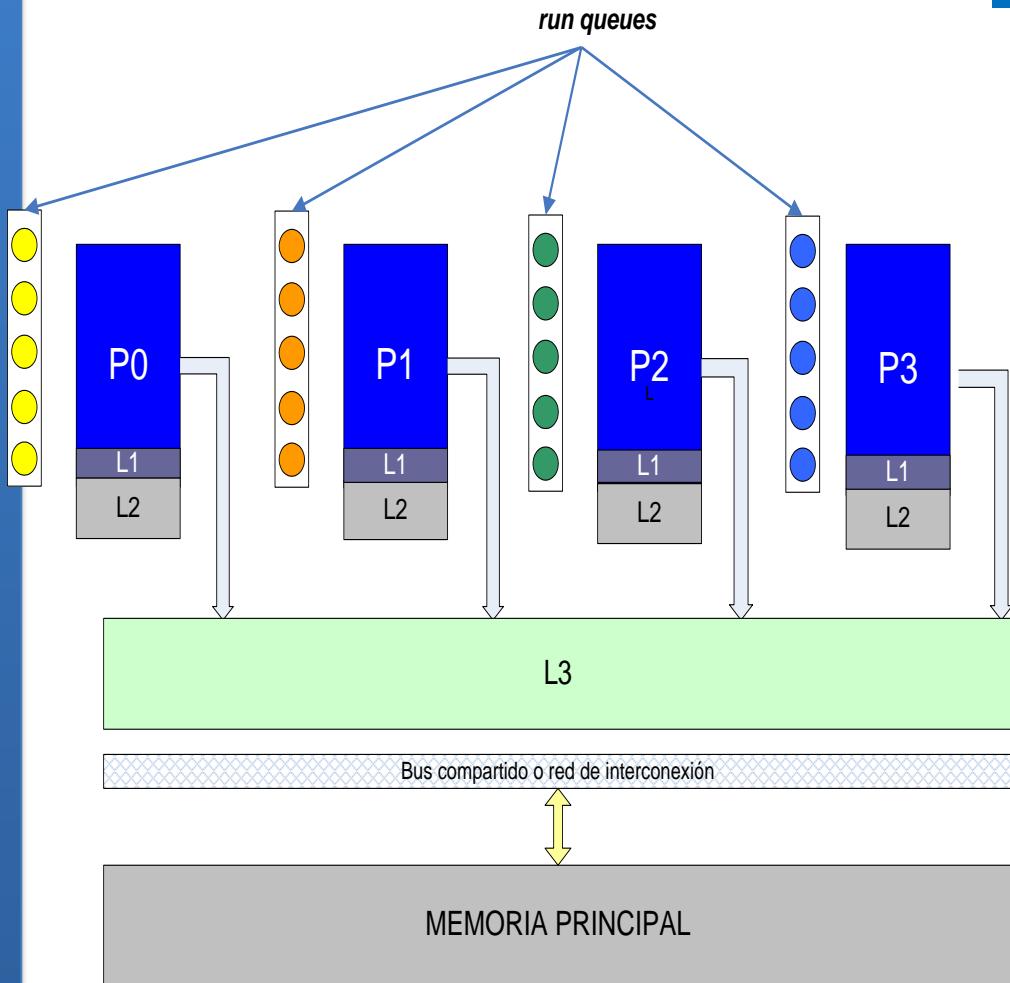


- Una cola de ejecución por procesador
- Mayor escalabilidad
- Períódicamente (o bajo demanda) se ejecuta el equilibrador de carga
  - Considera qué procesos pueden/deben migrarse
  - Tiene en cuenta la afinidad
- El usuario puede especificar la afinidad hilo-CPUs permitidas

➔ *Este modelo es el que utilizan la mayor parte de SSOO actuales de propósito general (Linux, Solaris, FreeBSD o MS Windows)*



# Planificación en multicore y SMT



- El SO ve cada core como un procesador independiente, pero no lo es
  - Algún nivel de cache compartido entre cores
  - *Cores pueden tener SMT (ej: Hyperthreading)* dos CPUs lógicas para el SO comparten todo el *datapath* (registros, unidades funcionales, colas...)



# Sistemas Operativos

2018-2019

## Comunicación y Sincronización entre Procesos

Basado en:

Sistemas Operativos  
J. Carretero [et al.]



# Contenido

- Procesos concurrentes
- Problemas clásicos
- Mecanismos C&S
  - Semáforos
  - Monitores: Mutex y Variables Condicionales
  - Memoria compartida
  - Señales
- Interbloqueos

# Procesos concurrentes



- Modelos
  - Multiprogramación en un único procesador
  - Multiprocesador
  - Multicomputador (proceso distribuido)
- Razones
  - Compartir recursos físicos
  - Compartir recursos lógicos
  - Acelerar los cálculos
  - Modularidad
  - Comodidad



# Contenido

- Procesos concurrentes
- Problemas clásicos
- Mecanismos C&S
  - Semáforos
  - Monitores: Mutex y Variables Condicionales
  - Memoria compartida
  - Señales
- Interbloqueos

# Problemas clásicos de comunicación y sincronización



- El problema de la sección crítica
- El problema del productor-consumidor
- El problema de los lectores-escritores
- Comunicación cliente-servidor
- Problema de los filósofos comensales

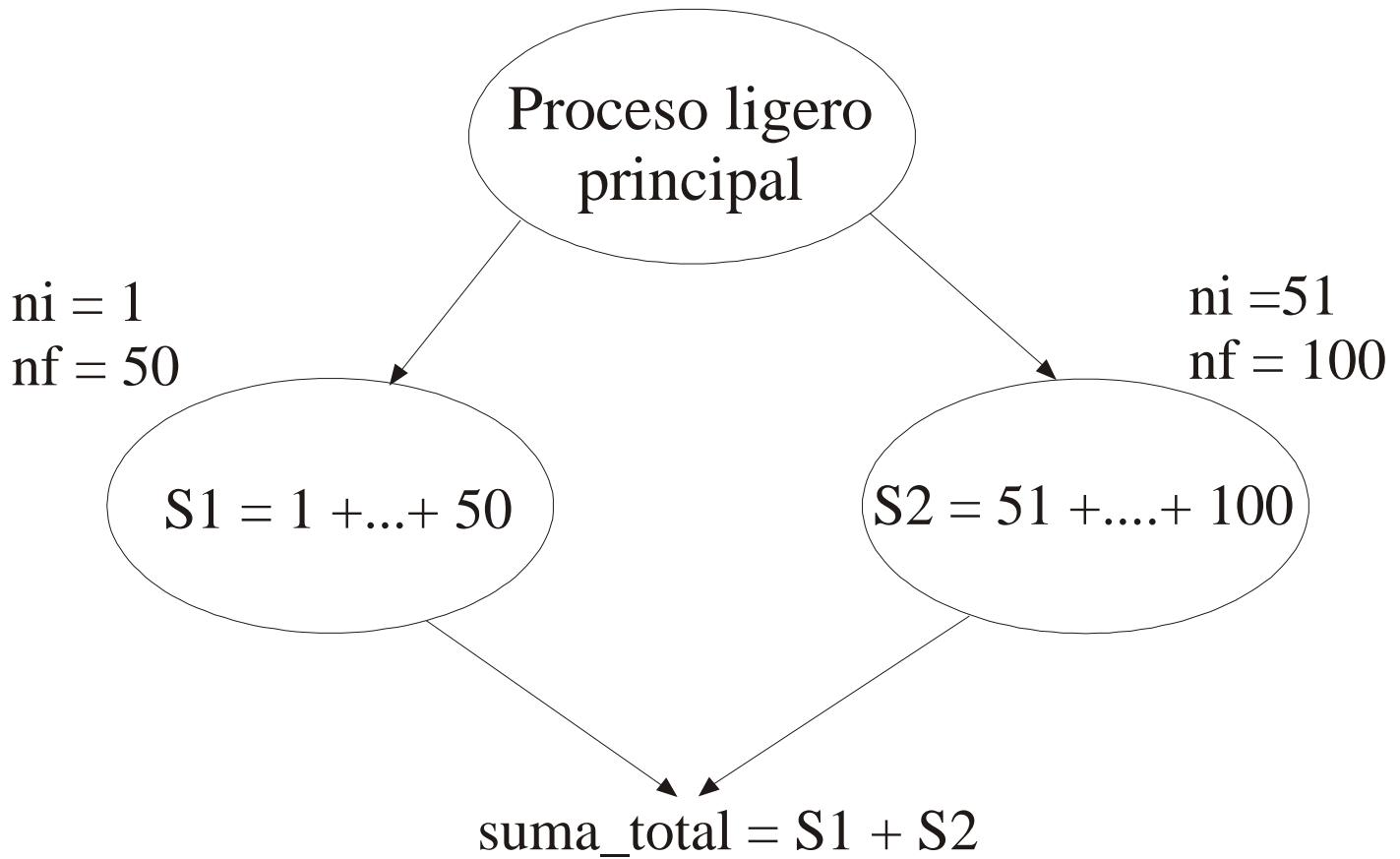


# Problema de la sección crítica

- Supongamos un sistema compuesto por  $n$  hilos
- Cada uno tiene un fragmento de código que accede/modifica un recurso compartido:
  - *Sección crítica*
- Queremos que sólo uno de los hilos en cada instante pueda ejecutar su sección crítica



# Ejemplo 1





# Ejemplo 1

- Calcula la suma de los  $N$  primeros números utilizando procesos ligeros.

```
int suma_total = 0; // Var compartida
void suma_parcial(int ni, int nf) {
    int j = 0;
    int suma_parcial = 0; // Var. privada
    for (j = ni; j <= nf; j++)
        suma_parcial = suma_parcial + j;
    suma_total = suma_total + suma_parcial;
    pthread_exit(0);
}
```

- Si varios hilos ejecutan concurrentemente este código se puede obtener un resultado incorrecto.



# Ejemplo 1

- Posible codificación en ensamblador para el cálculo de *suma\_total*:

```
suma_total = suma_total + suma_parcial;
```

LDR R1, suma_total	#R1=0 (la 1 <sup>a</sup> vez)
LDR R2, suma_parcial	#R2=1275
ADD R1,R1,R2	#R1=1275
STR R1, suma_total	#suma_total=1275



# Ejemplo 1

## ■ Posible situación de conflicto:

```
LDR R1, suma_total      #R1=0
LDR R2, suma_parcial    #R2=1275
```

##### Cambio de contexto #####

```
LDR R1, suma_total      #R1=0
LDR R2, suma_parcial    #R2=3775
ADD  R1, R1, R2         #R1=3775
STR R1, suma_total      #suma_total=3775
```

##### Cambio de contexto #####

```
ADD  R1, R1, R2         #R1=1275
STR R1, suma_total      #suma_total=1275
```

# Ejemplo con sección crítica



## ■ Solución:

- Solicitar permiso para entrar en sección crítica
- Indicar la salida de sección crítica

```
void suma_parcial(int ni, int nf) {  
    int j = 0;  
    int suma_parcial = 0;  
    for (j = ni; j <= nf; j++)  
        suma_parcial = suma_parcial + j;  
  
    <Entrada en la sección crítica>  
    suma_total = suma_total + suma_parcial;  
    <Salida de la sección crítica>  
    pthread_exit(0);  
}
```



# 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;  
}
```

- Si dos procesos ejecutan concurrentemente este código se puede perder algún ingreso.
- Solución: secciones críticas



# Ejemplo 2 con sección crítica

```
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;  
}
```



# Solución al problema de la sección crítica

- Requisitos que debe ofrecer cualquier solución para resolver el problema de la sección crítica:
  - **Exclusión mutua**: sólo un proceso en la región crítica
  - **Progreso**: Si ningún proceso está ejecutando dentro de la sección crítica, la decisión de qué proceso entra en la sección se hará sobre los procesos que desean entrar
  - **Espera limitada**: ningún proceso debe esperar indefinidamente para entrar en su región crítica
- Hay que tener también en mente:
  - Un proceso no debe ver retrasado el acceso a su sección crítica cuando no hay ningún otro proceso usándola
  - No deben hacerse suposiciones sobre las velocidades relativas de los procesos o sobre el número de procesos competidores
  - Un proceso permanece dentro de su sección crítica un tiempo finito

# Tipos de soluciones



- Espera activa
  - Sin soporte HW
    - Basadas en variables de control (Peterson 1981)
  - Con soporte HW
    - Test And Set (*TAS*), *XCHG*, *LL/SC*
- Sin espera activa
  - Uso de primitivas anteriores
  - El SO cambiará el estado del proceso bloqueado



# Instrucciones Máquina

- Se utiliza una instrucción máquina para actualizar una posición de memoria
- Puede aplicarse cualquier número de procesos:
  - Ciclo de memoria **RMW** (*read/modify/write*)
- No sufren injerencias por parte de otras instrucciones
- Puede aplicarse a múltiples secciones críticas
- Es simple y fácil de verificar



# Ejemplos de instrucciones

- Generales
  - Test and set (T&S)
  - Fetch and add (F&A)
  - Swap/Exchange
  - Compare and Swap (exchange)
  - Load link/ Store conditional (LL/SC)
- Intel (x86)
  - Muchas instrucciones pueden ser atómicas: *lock*
  - F&A → lock; xaddl eax, [mem\_dir];
  - XCHG → xchg eax, [mem\_dir\_lock]
  - CMPXCHG -> lock cmpxchg [dirMem], eax
- ARM (y otros)
  - LL/SC → LDREX y STREX



# Semántica y uso de Swap/Exchange

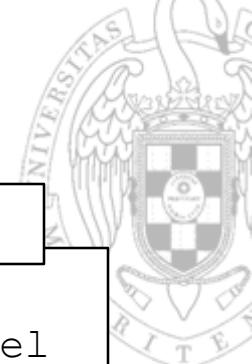
## Exchange

```
xchg src, dst
    rtmp ← Mem [src]
    Mem [src] ← Mem [dst]
    Mem [dst] ← rtmp
}
```

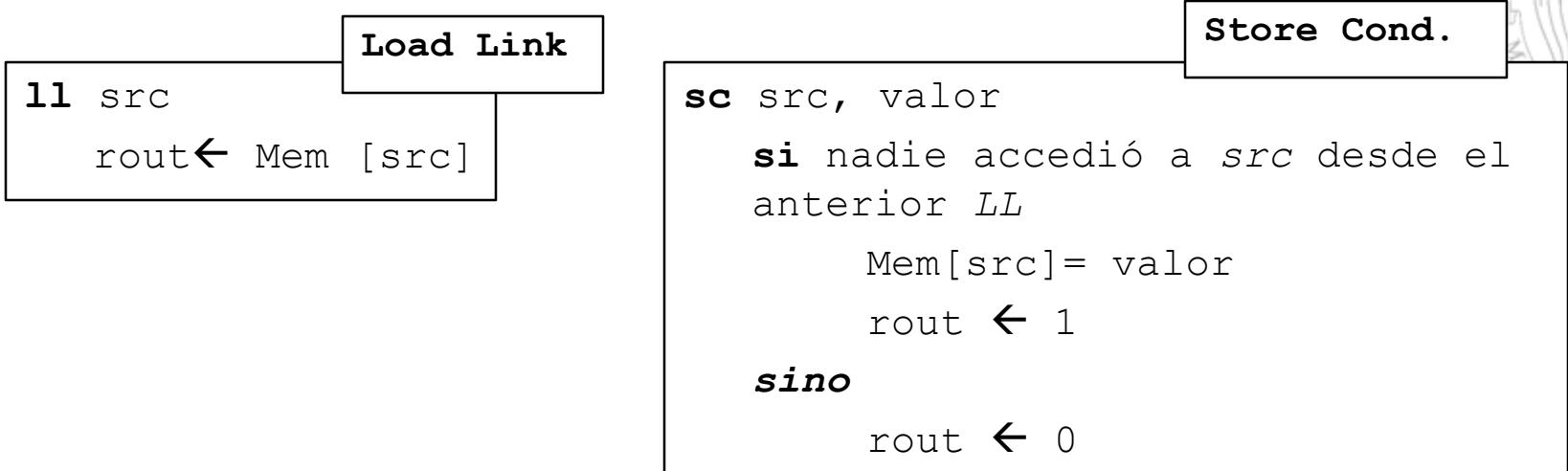
- Es UNA instrucción máquina (NO una función)
  - Es atómica, ininterrumpible
- Intercambia dos valores (potencialmente, ambos en memoria)
  - En Intel, sólo uno de los dos (`src` o `dst`) pueden estar en memoria

```
//puede estar en registro
tmp = 1;
//Espera activa
while( tmp== 1)
    xchg(dirM, tmp);
Sección_crítica();
*dirM= 0;
```

Solución al problema de la **Sección Crítica con XCHG**



# Semántica y uso de LL/SC



- Son DOS instrucciones máquina
  - Una siempre hace el *load*; la otra sólo hace *store* si no hubo escrituras a esa posición de memoria posteriores al LL

```

while (1) {
    while(ll(dirM) == 1);
    if (sc(dirM, 1)==1) break;
    //si no, otra vez al Load-Link
}
Sección_crítica();
*dirM= 0;
  
```

Solución al problema de la **Sección Crítica con LL/SC**

# Problemas clásicos de comunicación y sincronización



- El problema de la sección crítica
- **El problema del productor-consumidor**
- El problema de los lectores-escritores
- Comunicación cliente-servidor
- Problema de los filósofos comensales



# Problema del productor-consumidor



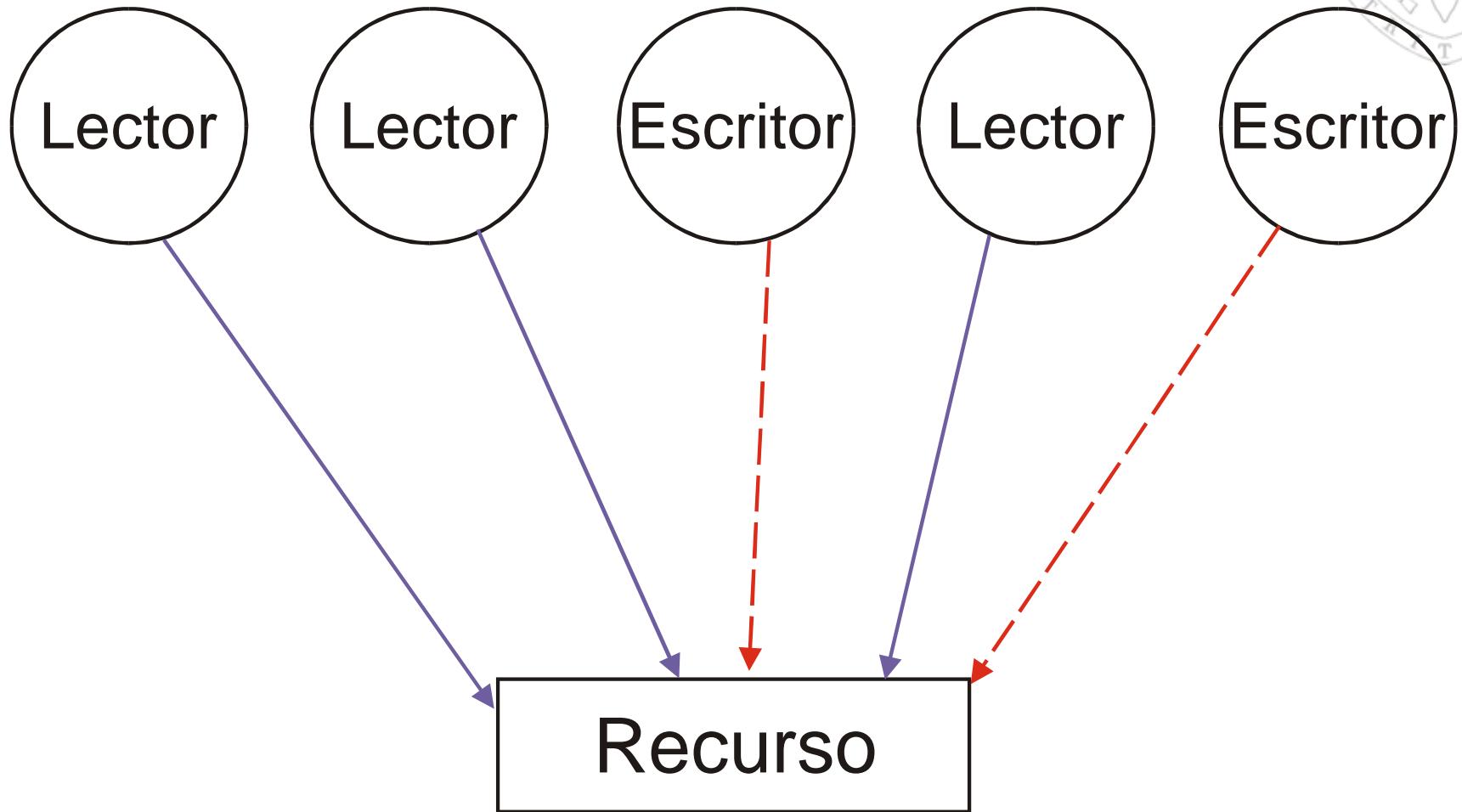
# Problemas clásicos de comunicación y sincronización



- El problema de la sección crítica
- El problema del productor-consumidor
- **El problema de los lectores-escritores**
- Comunicación cliente-servidor
- Problema de los filósofos comensales



# El problema de los lectores-escritores



# Problemas clásicos de comunicación y sincronización

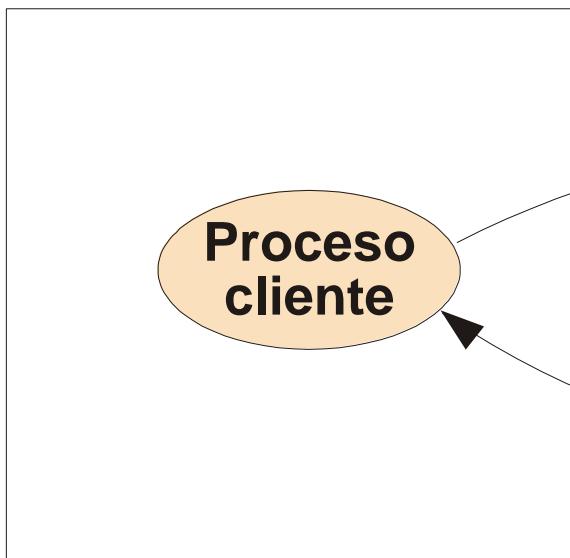


- El problema de la sección crítica
- El problema del productor-consumidor
- El problema de los lectores-escritores
- **Comunicación cliente-servidor**
- Problema de los filósofos comensales

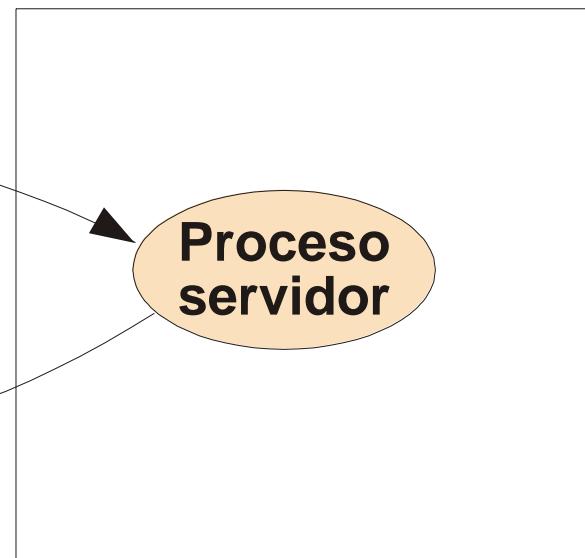


# Comunicación cliente-servidor

Computador



Computador



# Problemas clásicos de comunicación y sincronización

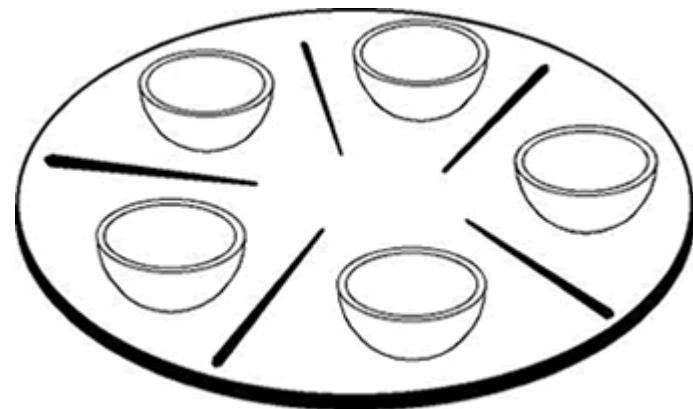


- El problema de la sección crítica
- El problema del productor-consumidor
- El problema de los lectores-escritores
- Comunicación cliente-servidor
- **Problema de los filósofos comensales**



# Filósofos comensales (Dijkstra'65)

- Cinco filósofos sentados en una mesa piensan y comen arroz:
  - Ningún filósofo debe morir de hambre (evitar bloqueo)
  - Necesitan 2 palillos para comer, que se cogen de uno en uno
  - Emplean un tiempo finito en comer y pensar
- Algoritmo:
  - Pensar...
  - Coger un palillo, coger el otro, comer, soltar un palillo y soltar el otro
  - Pensar...





# Filósofos comensales (Dijkstra'65)

- Soluciones:
  - Turno rotativo:
    - Desperdicia recursos
  - Un Camarero arbitra el uso de los palillos
    - Necesitamos un supervisor
  - Numerar los palillos, coger siempre el menor, luego y el mayor y soltarlos en orden inverso:
    - Penalizamos al último filósofo
  - Si no puedo coger el segundo palillo, suelto el primero
    - ¿Y si mis vecinos comen alternativamente?



# Contenido

- Procesos concurrentes
- Problemas clásicos
- Mecanismos C&S
  - Cerrojos y Variables Condicionales
  - Semáforos
  - Tuberías
  - Memoria compartida
- Interbloqueos



# Mecanismos C&S

- Todos los problemas clásicos tienen en común:
  - Necesitan **compartir** información
    - Que todos puedan conocer el valor de una variable...
  - Necesitan **sincronizar** su ejecución
    - Que un proceso espere a otro...
- Estudiaremos qué mecanismos suelen ofrecer los sistemas operativos para este fin
  - No estudiaremos cómo se implementan sino cómo se usan

# Mecanismos de comunicación



- Archivos
- Tuberías (pipes, FIFOs)
  - No las estudiaremos
- Memoria compartida
  - Implícita: hilos
  - Explícita: necesidad de una API específica



# Mecanismos de Sincronización

- Servicios del sistema operativo:
  - Señales: asincronas y no encolables (no las estudiaremos)
  - Tuberías (pipes, FIFOs) (no las estudiaremos)
  - Semáforos
  - Cerrojos y variables condicionales
- Las operaciones de sincronización deben ser **atómicas**



# Cerrojos (mutex)

- Un cerrojo es un mecanismo de sincronización indicado para hilos.
  - Ideal para el problema de la sección crítica, pues garantiza exclusión mutua....
- Podemos pensar en un cerrojo como un objeto con 3 atributos y 2 métodos **atómicos**

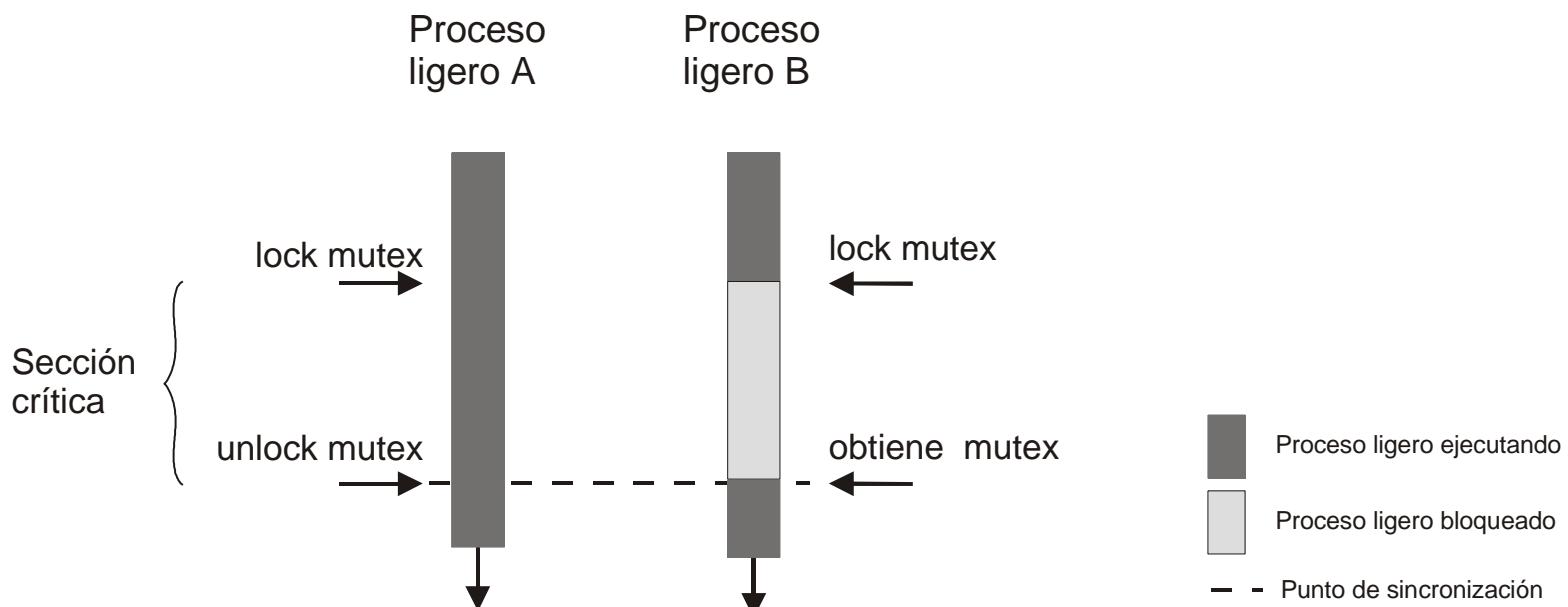
```
// Cerrojo abierto o cerrado  
estado_t estado;  
  
// Cola de hilos bloqueados  
queue_t q;  
  
//Hilo "propietario"  
hilo_id owner;
```

```
lock(m) {  
    if (m->estado==cerrado) {  
        queue_add(m->q, esteHilo);  
        suspenderHilo;  
    }  
    m->estado=cerrado;  
    m->owner = esteHilo;  
}  
  
unlock(m) {  
    if (m->owner==esteHilo) {  
        m->estado=abierto;  
        if (m->q.notEmpty ())  
            despiertaUnHiloDeCola();  
    }  
    else  
        error!!  
}
```

# Secciones críticas con mutex

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

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



# 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.



# Lectores-escritores con mutex

```
int dato = 5;                                /*recurso*/
int n_lect = 0;                               /*numero de lectores*/
pthread_mutex_t mutex;                        /*controlar el acceso a dato*/
pthread_mutex_t m_lect;                        /*controla la variable n_lect*/

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

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&m_lect, 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); Implementación incorrecta
                                                ¿POR QUÉ?

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

    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&m_lect);

    exit(0);
}
```



# Lectores-escritores con mutex (II)

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

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

        pthread_mutex_lock(&m_lect);
        n_lecadores--;
        if (n_lecadores == 0)
            pthread_mutex_unlock(&mutex);
        pthread_mutex_unlock(&m_lect);
    }
}
```

```
/*codigo del escritor */
void Escritor(void) {
    while(1){
        pthread_mutex_lock(&mutex);

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

        pthread_mutex_unlock(&mutex);
    }
}
```

Implementación incorrecta  
¿POR QUÉ?



# Variables condicionales

- Variables de sincronización **asociadas** a un cerrojo
- Se usan entre lock y unlock
- Podemos pensar en una variable condicional como un objeto con un atributo (y un cerrojo asociado) y 3 métodos principales.

```
typedef struct var_cond {  
    // Cola de hilos bloqueados  
    queue_t vc_q;  
} vc_t;
```



Hay una cola de espera **adicional** a la del cerrojo asociado



# Semántica de funciones asociadas

```
// El hilo que llama a esta función  
// DEBE ser el propietario del cerrojo c  
void cond_wait(lock_t c, vc_t varC ) {  
    queue_add(varC->vc_q, esteHilo);  
    unlock(c);  
    park(); // suspender el hilo  
    lock();  
}
```

- **Siempre** que se llama a *cond\_wait* el hilo se bloquea
- Antes de bloquearse libera el cerrojo para que otro hilo lo pueda adquirir
- Tras despertar del bloque, vuelve a **solicitar** el cerrojo
  - Puede implicar un nuevo bloqueo
  - Cuando hilo sale de *cond\_wait*, **sigue en posesión del cerrojo**

```
// Despierta un hilo de la cola de espera  
// dela Var. Cond  
void cond_signal (vc_t varC ) {  
    if (! isEmpty(varC->vc_q)  
        unpark(queue_remove(varC->vc_q))  
}
```

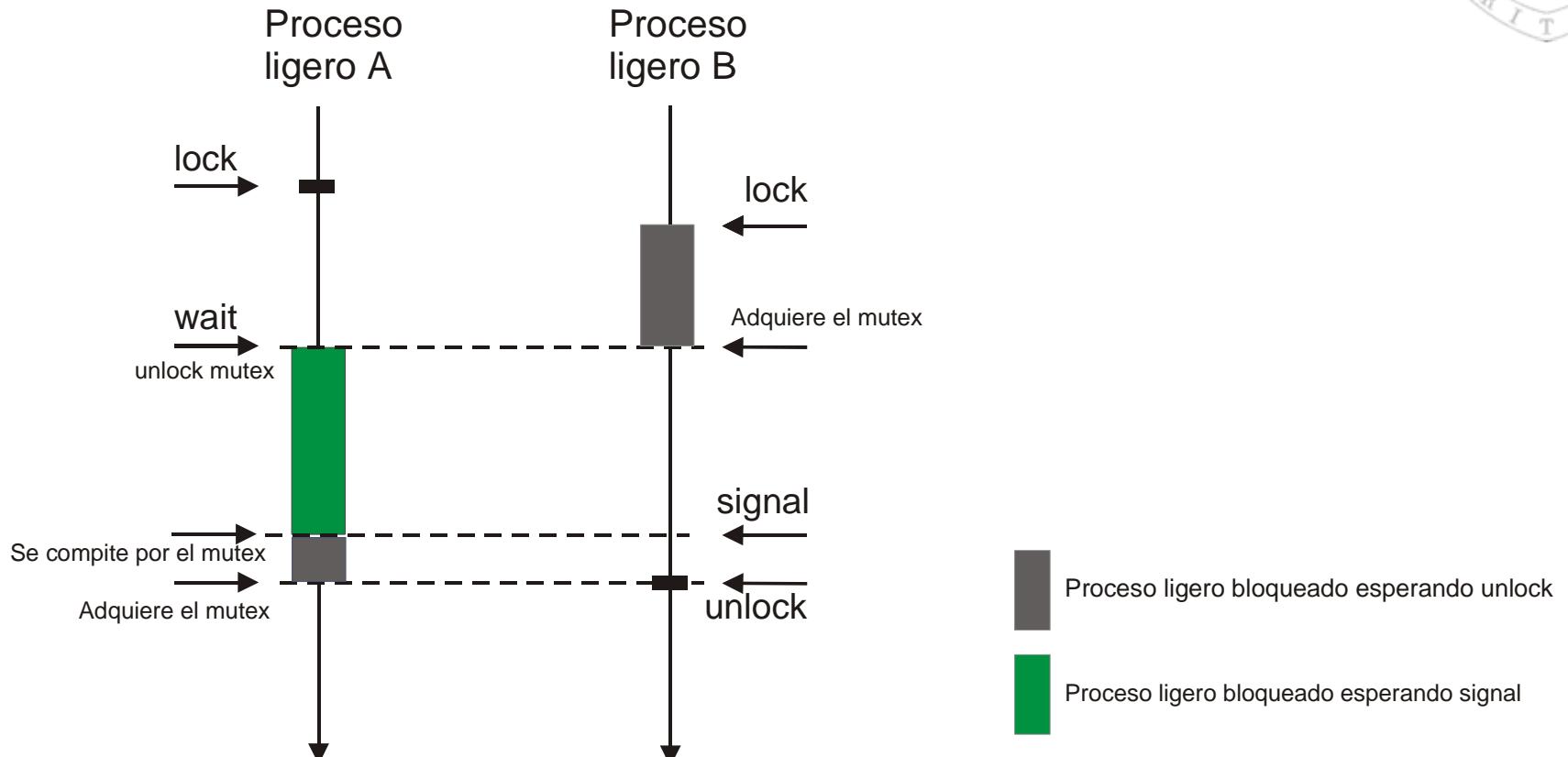
```
// Despierta a todos los hilos de la cola  
// de espera  
void cond_broadcast (vc_t varC ) {
```

```
    while (! isEmpty(varC->vc_q)  
        unpark(queue_remove(varC->vc_q))  
}
```

- Es **muy aconsejable** que el hilo que llama a estas funciones tenga el cerrojo asociado

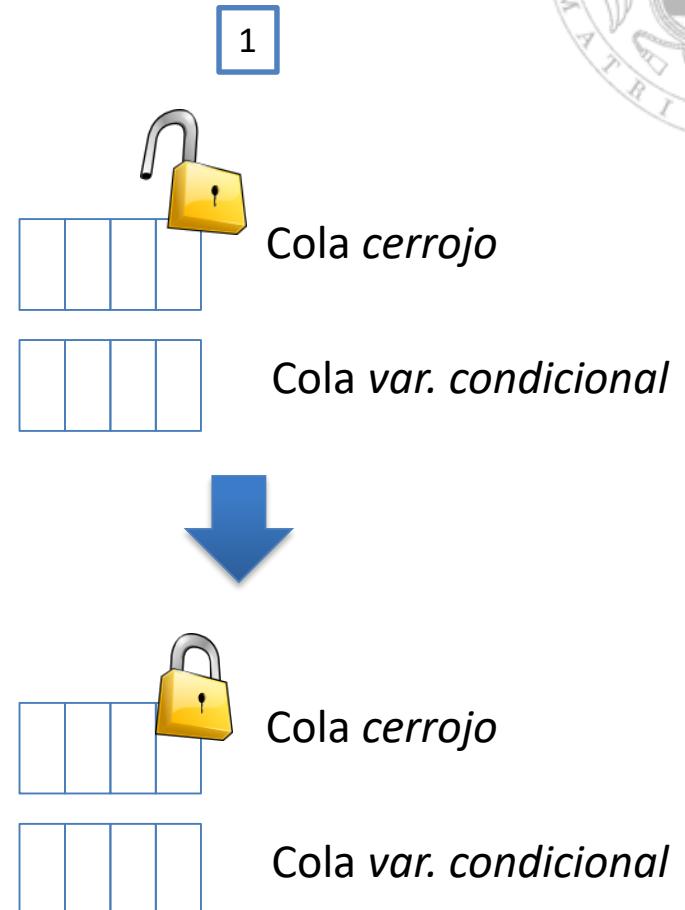
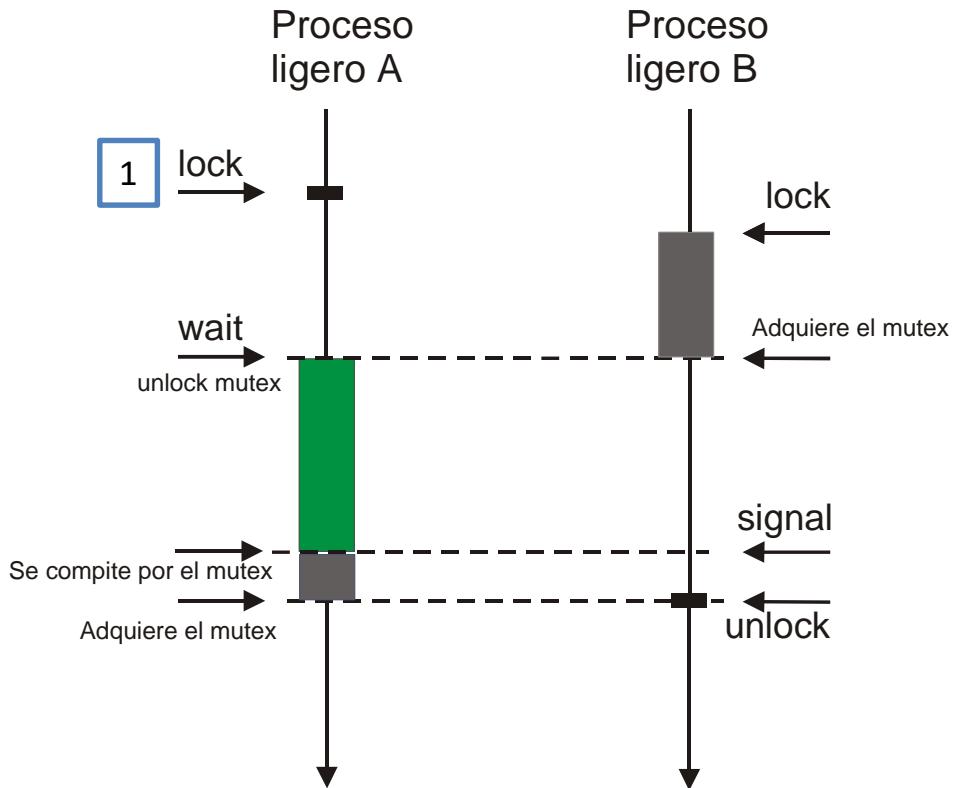


# Variables condicionales (II)



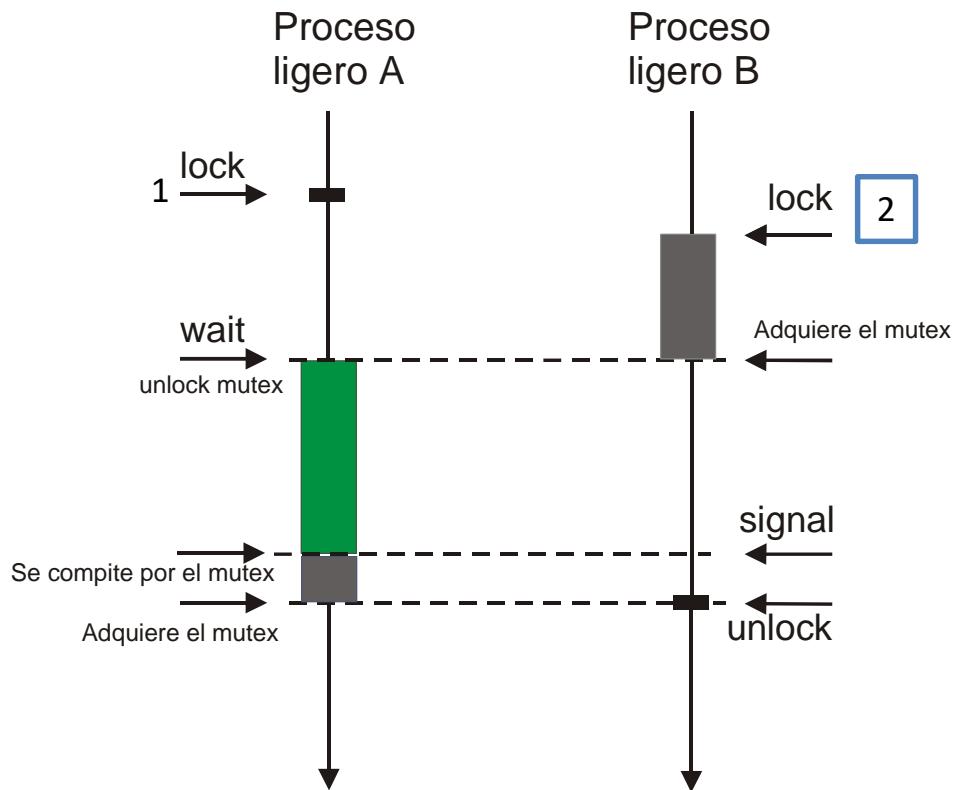


# Uso de var. condicional





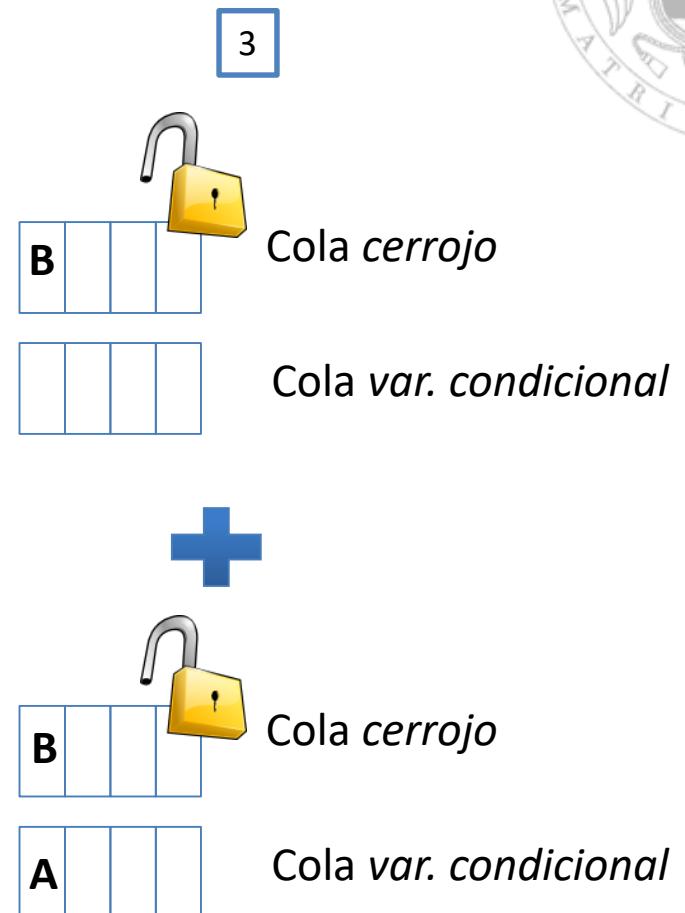
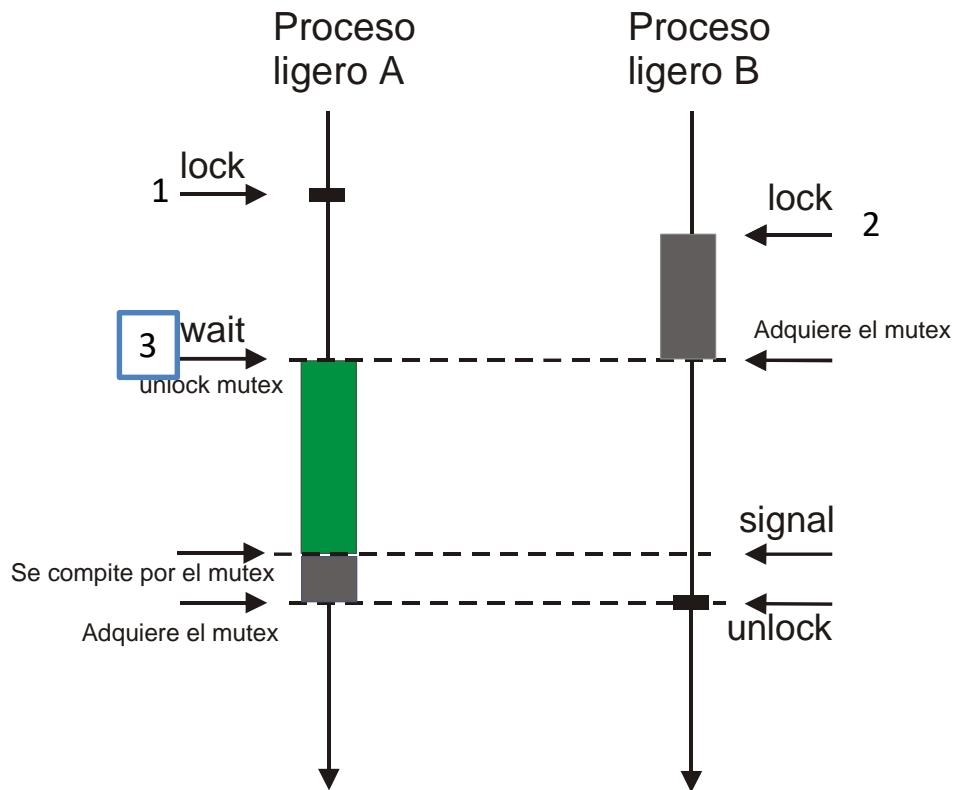
# Uso de var. condicional



A running

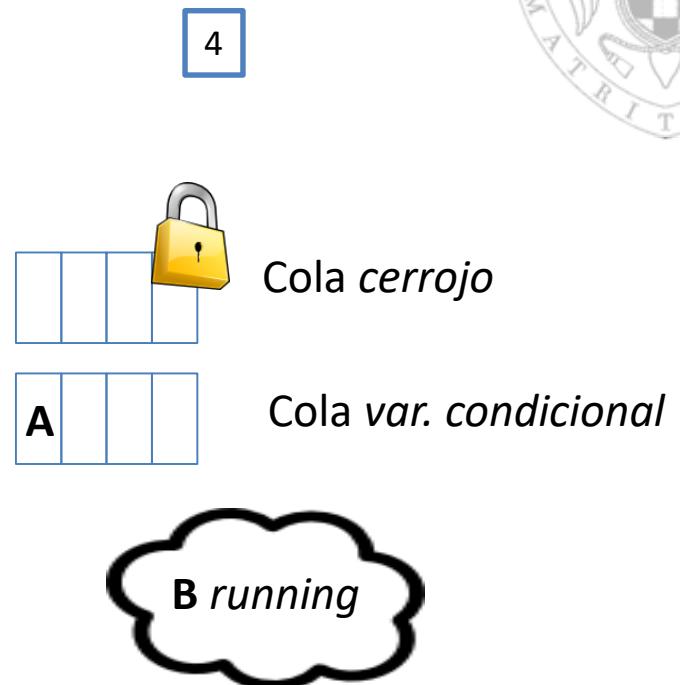
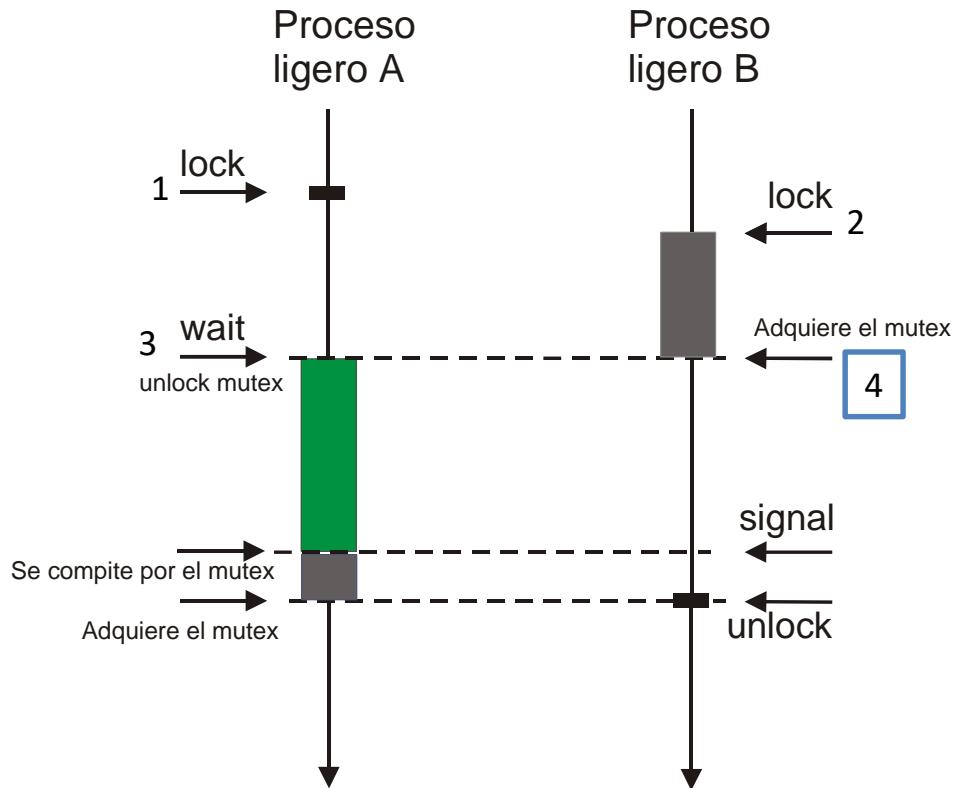


# Uso de var. condicional



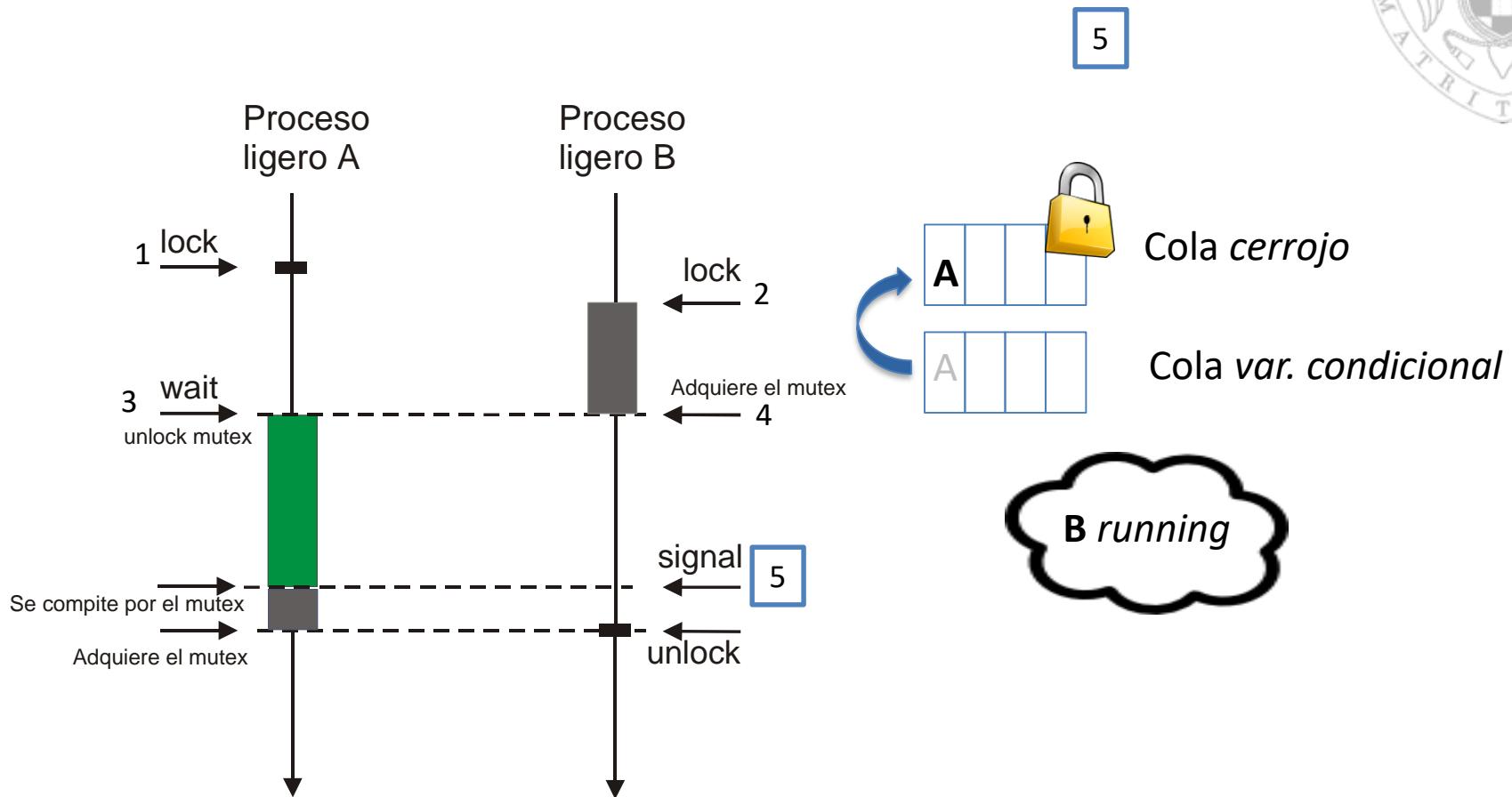


# Uso de var. condicional



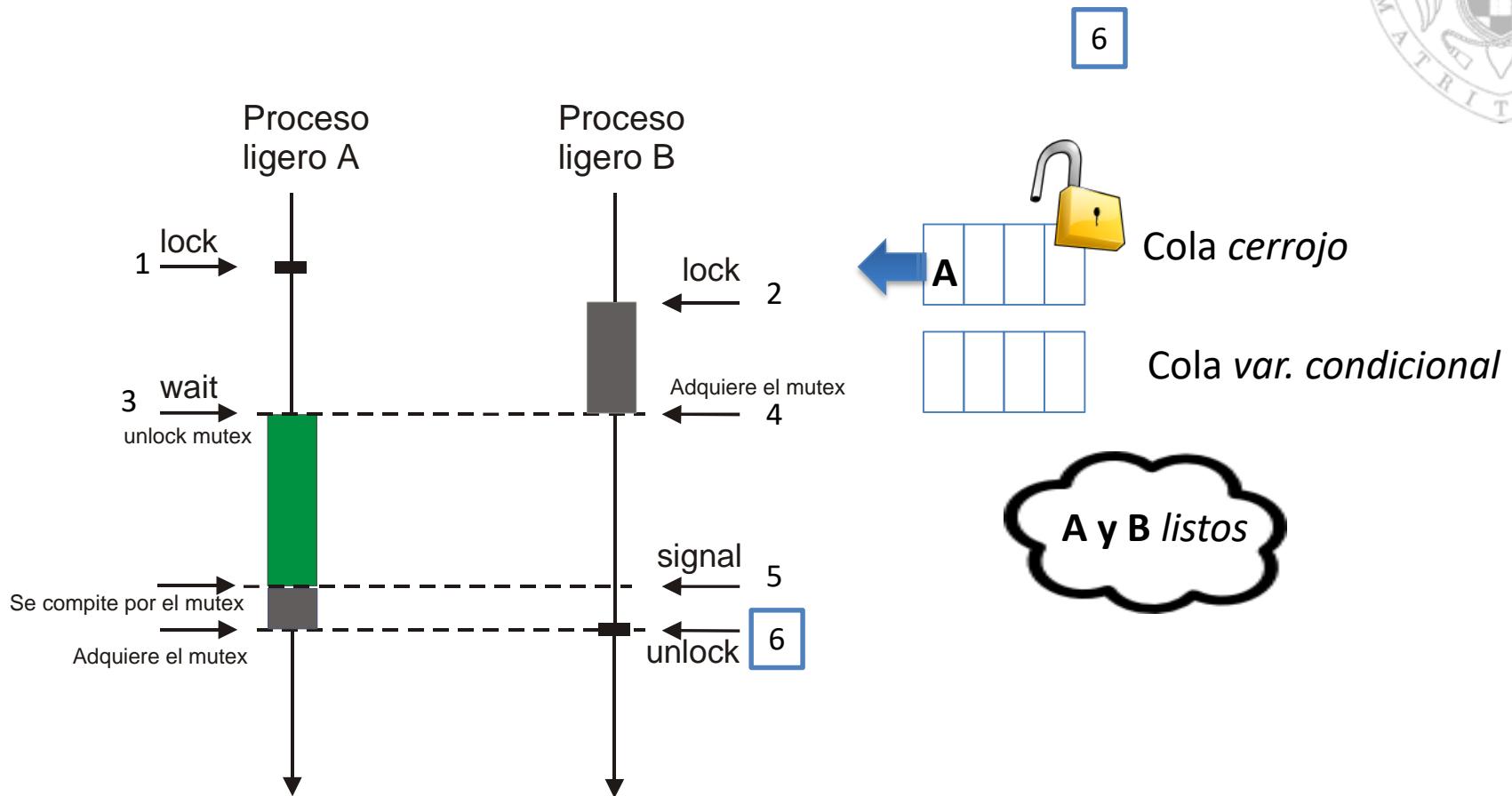


# Uso de var. condicional





# Uso de var. condicional





# Uso de cerrojos / var. condicionales

## ■ Proceso ligero A

```
lock(mutex); /* acceso al recurso */  
while (condición relacionada con el recurso == false)  
    wait(condition, mutex); /*bloqueo*/  
<acciones deseadas que cumplen la condición>  
unlock(mutex);
```

## ■ Proceso ligero B

```
lock(mutex); /* acceso al recurso */  
<operaciones protegidas>  
/*hemos podido afectar a otros procesos, desbloqueamos*/  
signal(condition);  
<más operaciones protegidas>  
unlock(mutex);
```

## ■ Importante utilizar while



# Servicios POSIX (II)

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);`
  - Inicializa una variable condicional.
- `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` y sólo continua con su ejecución cuando lo obtiene



# Productor-consumidor con var. Cond.

```
#define MAX_BUFFER          1024      /* tamano del buffer */
#define DATOS_A_PRODUCIR    100000    /* datos a producir */
pthread_mutex_t mutex;           /*mutex para buffer compartido*/
pthread_cond_t lleno;           /*controla el llenado del buffer*/
pthread_cond_t 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(&lleno, NULL);
    pthread_cond_init(&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(&lleno);
    pthread_cond_destroy(&vacio);
    exit(0);
}
```



# Productor-consumidor con var. Cond.

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

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



# Productor-consumidor con var cond.

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



# Simplificación

```
buffer[MAX_BUFFER];
indProd = 0, indCons = 0;
int n_elementos = 0;

mutex_t mutex;
cond_t evento;
```

```
void Productor () {
...
    mutex_lock(&mutex)
    while( n_elementos == MAX_BUFFER)
        cond_wait(&evento, &mutex);

    <inserta un elemento de la cola>

    cond_broadcast(&evento);
    mutex_unlock(&mutex);
...
}
```

```
void Consumidor () {
...
    mutex_lock(&mutex)
    while( !n_elementos )
        cond_wait(&evento, &mutex);

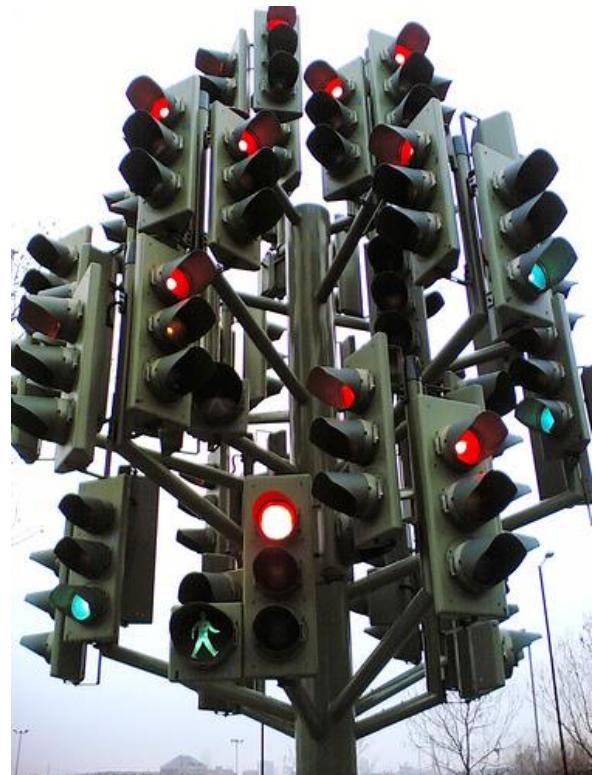
    <extrae un elemento de la cola>

    cond_broadcast(&evento);
    mutex_unlock(&mutex);
...
}
```

# Semáforos (Dijkstra'65)



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



# Operaciones sobre semáforos (semántica)



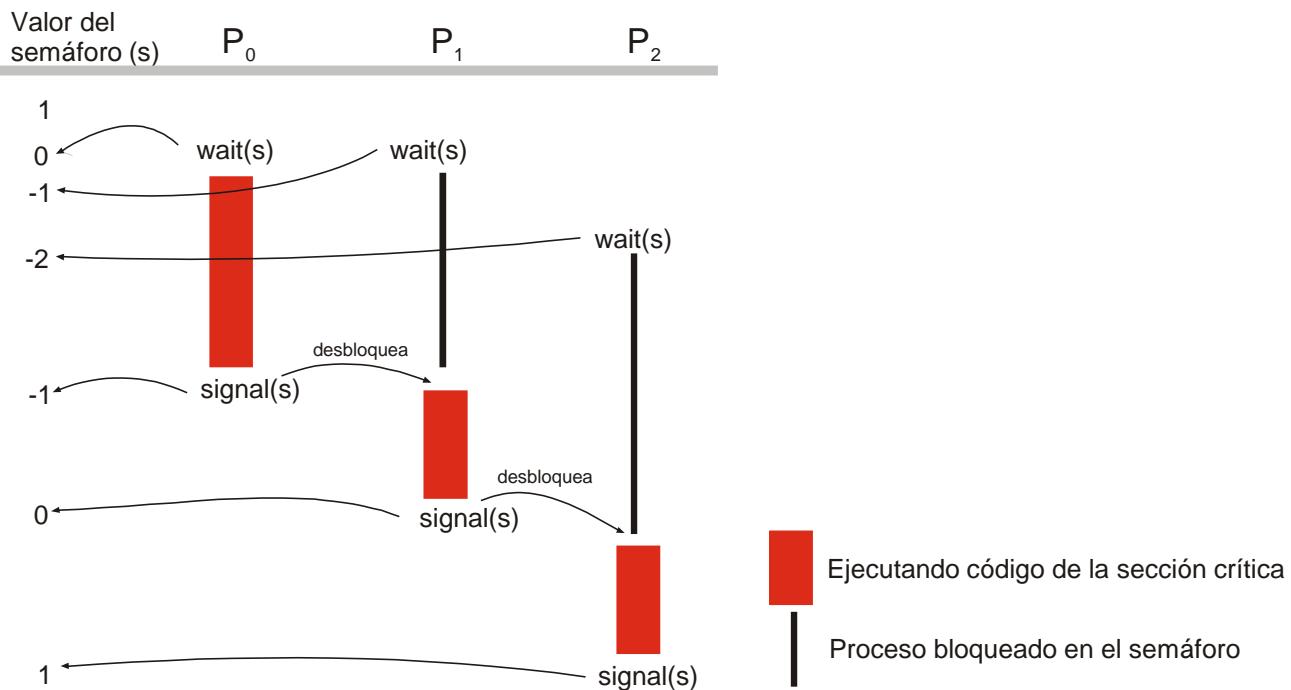
```
wait(s) {  
    s = s - 1;  
    if (s < 0) {  
        <Bloquear al proceso>  
    }  
}  
  
signal(s) {  
    s = s + 1;  
    if (s <= 0) {  
        <Desbloquear a un proceso bloq. por wait>  
    }  
}
```



# Secciones críticas con semáforos

```
wait(s); /* entrada en la sección critica */  
Sección_critica();  
signal(s); /* salida de la sección critica */
```

## Ejemplo con valor inicial 1

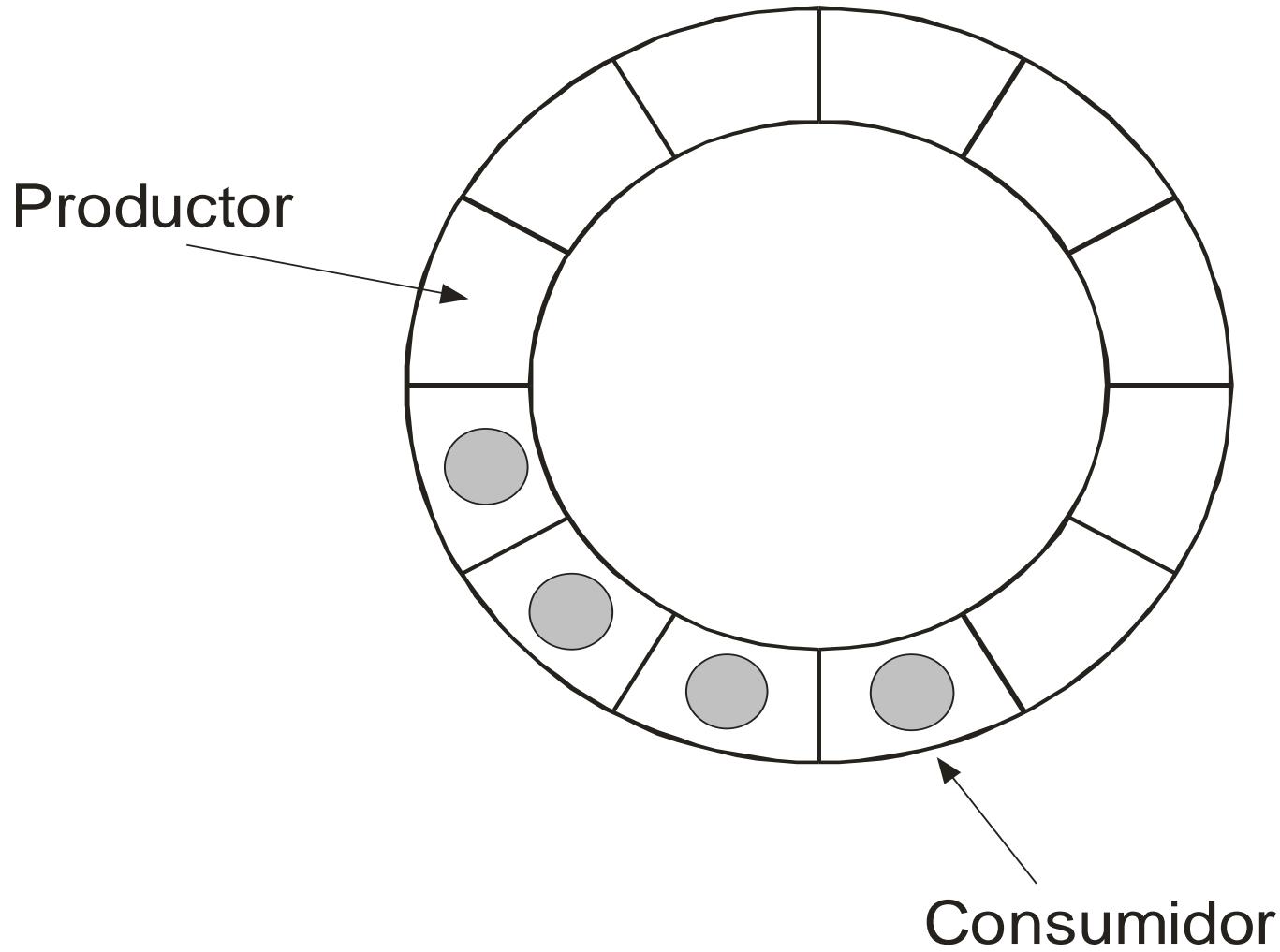




# Semáforos POSIX

- `int sem_init(sem_t *sem, int shared, int val);`
  - Inicializa un semáforo sin nombre
  - shared: pensado para ser mapeado en memoria compartida
- `int sem_destroy(sem_t *sem);`
  - Destruye un semáforo sin nombre
- `sem_t*sem_open(char*name, int flag, mode_t mode, int val);`
  - Abre (crea) un semáforo con nombre
- `int sem_close(sem_t *sem);`
  - Cierra un semáforo con nombre.
- `int sem_unlink(char *name);`
  - Borra un semáforo con nombre
- `int sem_wait(sem_t *sem);`
  - Realiza la operación wait sobre un semáforo
- `int sem_post(sem_t *sem);`
  - Realiza la operación signal sobre un semáforo

# Productor-consumidor con semáforos (buffer acotado y circular)





# Productor-consumidor con semáforos (II)

```
#define MAX_BUF          1024      /*tamanio del buffer */
#define PROD             100000    /*datos a producir */
sem_t elementos;
sem_t huecos;
int buffer[MAX_BUF];
Int cons,prod = 0;                /*buffer comun*/
                                         /*posicion dentro del buffer*/\n\nvoid main(void){
    pthread_t th1, th2; /* identificadores de threads */
    /* inicializar los semaforos */
    sem_init(&elementos, 0, 0);    sem_init(&huecos, 0, MAX_BUFFER);\n\n    /*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);\n\n    sem_destroy(&huecos);    sem_destroy(&elementos);
    exit(0);
}
```



# Productor-consumidor con semáforos (II)

```
void Productor(void){  
/*dato a producir*/  
    int dato;  
    int i;  
  
    for(i=0; i < PROD; i++ ){  
        /*producir dato*/  
        dato = producir_dato();  
        /*un hueco menos*/  
        sem_wait(&huecos);  
        buffer[prod] = dato;  
        prod = (prod + 1) % MAX_BUF;  
        /*un elemento mas*/  
        sem_post(&elementos);  
    }  
    pthread_exit(0);  
}
```

```
void Consumidor(void){  
/*dato a producir*/  
    int dato;  
    int i;  
  
    for(i=0; i<PROD; i++ ){  
        /*un elemento menos*/  
        sem_wait(&elementos);  
        dato = buffer[con];  
        cons= (cons+ 1) % MAX_BUF;  
        /*un hueco mas*/  
        sem_post(&huecos);  
        cosumir_dato(dato);  
    }  
    pthread_exit(0);  
}
```

**CUIDADO: problema de sección crítica SIN  
RESOLVER (si hay muchos productores y/o  
consumidores)**



# Lectores-escritores con semáforos

```
int dato = 5;          /* recurso */
int n_lectores = 0; /* numero de lectores */
sem_t sem_lec;      /* controlar el acceso n_lectores */
sem_t cerrojo;      /* controlar el acceso a dato */
void main(void) {
    pthread_t th1, th2, th3, th4;
    sem_init(&cerrojo, 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);

    /* cerrar todos los semafotos */
    sem_destroy(&cerrojo);    sem_destroy(&sem_lec);
    exit(0);
}
```



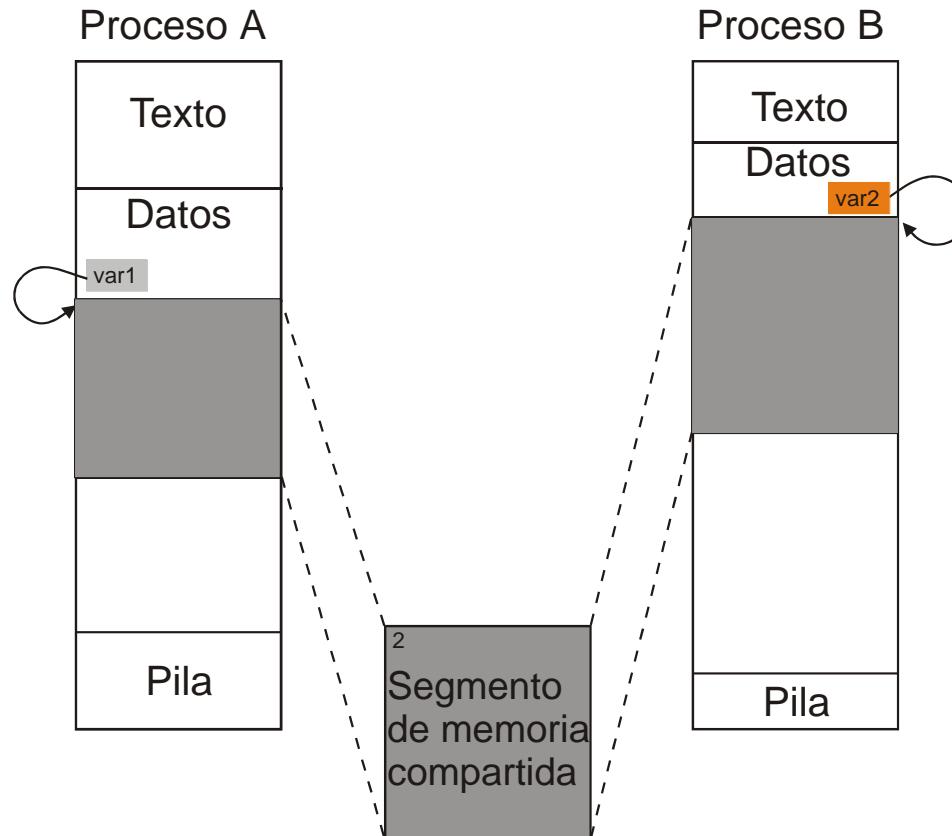
# Lectores-escritores con semáforos (II)

```
void Lector(void) {  
    While(1){  
        sem_wait(&sem_lec);  
        n_lectores = n_lectores + 1;  
        if (n_lectores == 1)  
            sem_wait(&cerrojo);  
        sem_post(&sem_lec);  
  
        /* leer dato */  
        printf("%d\n", dato);  
  
        sem_wait(&sem_lec);  
        n_lectores = n_lectores - 1;  
        if (n_lectores == 0)  
            sem_post(&cerrojo);  
        sem_post(&sem_lec);  
    }  
}
```

```
void Escritor(void) {  
    while(1){  
        sem_wait(&cerrojo);  
  
        /* modificar el recurso */  
        dato = dato + 2;  
  
        sem_post(&cerrojo);  
    }  
}
```

# Memoria compartida (entre procesos)

- Declaración independiente de variables dentro de los procesos que apuntan a la misma región de memoria “real”





# Memoria compartida POSIX

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
  - Ubica (mapea) una porción del fichero especificado por el descriptor fd en memoria, devolviendo un puntero a esa región (addr)
  - Esta región de memoria puede ser compartida o privada:
    - flags: MAP\_SHARED ó MAP\_PRIVATE
  - También se puede declarar sin respaldo en disco:
    - flags: MAP\_ANONYMOUS (compartir padre-hijo)
    - Empleando `shm_open` para obtener un descriptor
- `int munmap(void *addr, size_t length);`
  - Actualiza el fichero de respaldo de la región de memoria y borra las ubicaciones para el rango de direcciones especificado.
- `int msync(void *addr, size_t len, int flags);`
  - Escribe cualquier dato (página) modificada en memoria en su correspondiente fichero de respaldo



## ■ Productor:

- Crea los semáforos con nombre (`sem_open`)
- Crea un archivo (`open`)
- Le asigna espacio (`ftruncate`)
- Proyecta el archivo en su espacio de direcciones (`mmap`)
- Utiliza la zona de memoria compartida
- Desprojeta la zona de memoria compartida (`munmap`)
- Cierra y borra el archivo

## ■ Consumidor:

- Abre los semáforos (`sem_open`)
- Debe esperar a que archivo esté creado para abrirlo (`open`)
- Proyecta el archivo en su espacio de direcciones (`mmap`)
- Utiliza la zona de memoria compartida
- Cierra el archivo



# Código del productor

```
#define MAX_BUFFER          1024      /* tamaño del buffer */
#define DATOS_A_PRODUCIR   100000    /* datos a producir */
sem_t *elementos;    /* elementos en el buffer */
sem_t *huecos;        /* huecos en el buffer */

void main(int argc, char *argv[]) {
    int shd;
    int *buffer;      /* buffer comun */

    /* el productor crea el archivo a proyectar */
    shd = open("BUFFER", O_CREAT|O_WRONLY, 0700);
    ftruncate(shd, MAX_BUFFER * sizeof(int));

    /*proyectar el objeto de memoria compartida en el espacio
    de direcciones del productor*/
    buffer = (int*) mmap(NULL, MAX_BUFFER * sizeof(int),
                         PROT_WRITE, MAP_SHARED, shd, 0);
```



# Código del productor (II)

```
/* El productor crea los semaforos */
elementos = sem_open("ELEMENTOS", O_CREAT, 0700, 0);
huecos =      sem_open("HUECOS", O_CREAT, 0700, MAX_BUFFER);

/* código de producción*/
Productor(buffer) {

    /* desproyectar el buffer compartido */
    munmap(buffer, MAX_BUFFER * sizeof(int));
    close(shd);           /* cerrar el objeto de memoria compartida */
    unlink("BUFFER");   /* borrar el objeto de memoria */

    sem_close(elementos);
    sem_close(huecos);
    sem_unlink("ELEMENTOS");
    sem_unlink("HUECOS");
}
```



# Código del consumidor

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

sem_t *elementos;    /* elementos en el buffer */
sem_t *huecos;       /* huecos en el buffer */

void main(int argc, char *argv[]) {
    int shd;
    int *buffer;      /* buffer comun */

    /* el consumidor abre el archivo a proyectar */
    shd = open("BUFFER", O_RDONLY);

    /*proyectar el objeto de memoria compartida en el espacio de
    direcciones del productor*/
    buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int),
                         PROT_READ, MAP_SHARED, shd, 0);
```



# Código del consumidor (II)

```
/*El consumidor abre los semáforos*/  
elementos = sem_open("ELEMENTOS", 0);  
huecos     = sem_open("HUECOS", 0);  
  
/*proceso consumidor con buffer proyectado  
Consumidor(buffer);  
  
/*desproyectar el buffer compartido*/  
munmap(buffer, MAX_BUFFER * sizeof(int));  
close(shd); /* cerrar el objeto de memoria compartida */  
  
/*cerrar los semáforos*/  
sem_close(elementos);  
sem_close(huecos);  
}
```



# Función del productor

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

# Función del consumidor



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

    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 */
        printf("Consumo %d \n", dato); /* consumir dato */
    }
    return;
}
```



# Resumen

- Hilos:
  - Memoria compartida (variables globales)
  - Mutex y variables condicionales
- Procesos emparentados (fork):
  - memoria compartida (mapeada)
  - Semáforos con o sin nombre
- Procesos no emparentados en la misma máquina:
  - Memoria compartida (regiones con nombre)
  - Semáforos con nombre

---

# Problemas de Sistemas Operativos

Dpto. ACyA, Facultad de Informática, UCM

Modulo 3.3 – Comunicación y Sincronización

---

## Problemas básicos

1.- Estudie el siguiente código que trata de resolver, únicamente por software y sin mediación del sistema operativo, el problema de la exclusión mutua entre dos hilos:

```
turno = 0

Hilo 0
while(TRUE){
    //Espera ocupada
    while(turno!=0);
    sección_crítica_H1();
    turno = 1;
    otro_código_H1()
}

Hilo 1
while(TRUE){
    //Espera ocupada
    while(turno!=1);
    sección_crítica_H2();
    turno = 0;
    otro_código_H2()
}
```

¿Resuelve correctamente el problema de la sección crítica garantizando exclusión, progreso y espera limitada?. Justifique su respuesta

2.- Escriba un programa que cree tres hilos que se comunicarán entre ellos. El hilo 1 genera los 1000 primeros números pares y el hilo 2 los 1000 números impares. El hilo 3 irá leyendo esos números e imprimiéndolos en pantalla. Se debe garantizar que los números escritos por pantalla estén en orden: 1,2,3,4,5... Implementar dicha sincronización mediante:

- a) Cerrojos y variables condicionales.
- b) Semáforos.

3.- Implementar la funcionalidad de un semáforo general a partir de cerrojos y variables condicionales. Para ello, defina un tipo de datos llamado **sem\_t** (un struct en C con los campos necesarios), y las funciones **wait()** y **signal()** de acuerdo a la semántica estudiada en clase.

4.- **El Problema de los Fumadores [Suhas Patil]:** considerar un sistema con tres procesos fumadores y un proceso agente. Continuamente cada fumador se prepara un cigarrillo y lo fuma. Para hacer un cigarrillo se necesitan tres ingredientes: **tabaco**, **papel** y **cerillas**. Uno de los procesos tiene infinito **papel**, otro **tabaco** y el tercero **cerillas**. El agente tiene provisión infinita de los tres ingredientes.

El agente coloca dos ingredientes distintos y de forma aleatoria en la mesa. El fumador que tiene el ingrediente que falta puede recoger los ingredientes y señalizar al agente para indicarle que coloque otros dos ingredientes en la mesa, después fumará (tardando un tiempo indeterminado pero finito). La operación se repite indefinidamente.

Escribir un programa que sincronice al agente y los fumadores mediante semáforos o mutexes y variables condicionales. Asúmase que el agente no tiene forma de consultar los ingredientes que posee cada fumador.

5.- **Una tribu de salvajes** se sirven comida de un caldero con  $M$  raciones de estofado de misionero. Cuando un salvaje desea comer, se sirve una ración del caldero a menos que esté vacío. Si está vacío deberá avisar al cocinero para que reponga otras  $M$  raciones de estofado, y entonces se podrá servir su ración. Un cocinero y número arbitrario de salvajes se comportan del siguiente modo:

```

//Cocinero:
while (True){
    putServingsInPot()
}

//Salvajes:
while (True){
    getServingFromPot()
    eat()
}

```

Sobre este código realice los siguiente:

- 1) Añada el código necesario para garantizar la correcta sincronización, usando semáforos POSIX y variables de tipo entero, booleano...
- 2) Añada el código necesario para garantizar la correcta sincronización, usando cerrojos y variables condicionales

Se deben cumplir las siguientes restricciones:

- a) Los salvajes no pueden invocar `getServingFromPot()` si el caldero está vacío
- b) El cocinero sólo puede invocar `putServingsInPot()` si el caldero está vacío

**6.-** Simular con hilos el comportamiento de una gasolinera con 2 surtidores de pago directo con tarjeta. Cuando un cliente coge un surtidor puede servirse el combustible deseado (con una llamada a la función `void ServirCombustible( int surtidor, int dinero )`). Una vez servido el combustible dejará libre el surtidor para que otro cliente pueda usarlo. Para que funcione correctamente el servicio deben satisfacerse las siguientes restricciones:

- a) Los clientes deben acceder al servicio en orden de llegada, pudiendo escoger cualquiera de los surtidores libres.
- b) No puede haber más de un cliente simultáneamente en un mismo surtidor.
- c) Mientras un cliente se está sirviendo el combustible en un surtidor, cualquier otro cliente debe poder utilizar el otro surtidor si está libre.

Implementar el código de la función principal del hilo cliente: `void cliente(int dinero)`. El argumento de entrada de dicha función indica cuánto dinero invertirá ese cliente en el combustible.

```

// < declaración de variables globales >
void cliente(int dinero) {
    // < declaración de variables locales >
    // Comprobar que se cumplen los requisitos para repostar
    ServirCombustible(surtidor,dinero);
    // Acciones de salida
}

```

**7.-** Un **monitor** lo podemos entender como un objeto cuyos métodos son ejecutados con exclusión mutua, por lo que un hilo/proceso que invoque un método del monitor se puede bloquear a la espera de un evento generado por otro hilo/proceso a través del mismo monitor. Dicha exclusión mutua y espera selectiva se puede implementar mediante cerrojos y variables condicionales.

Resolver el problema de la **cena de los filósofos** codificando un monitor (usando cerrojos y variables condicionales) basándose en el siguiente ejemplo de filósofo situado en la posición i-ésima de la mesa. El monitor debe incluir la implementación de `cogerPalillosMonitor()` y `dejarPalillosMonitor()` así como la declaración de variables (privadas al monitor) que sean necesarias.

```

Filosofo(int i){
    while(1){
        pensar();

        //Solicitamos al monitor la necesidad de coger los palillos
        cogerPalillosMonitor(i);
        comer();

        //Solicitud al monitor que queremos dejar los palillos
        dejarPalillosMonitor(i);
    }
}

```

## Problemas adicionales

8.- El algoritmo de Peterson es una solución software correcta para el problema de la sección crítica. A continuación se muestra el algoritmo de Peterson para dos procesos/hilos. Justifique que, efectivamente, cumple los tres requisitos de cualquier solución correcta del problema de la sección crítica.

```
turno = 0
interesado = {false, false}

Hilo 0
interesado[0] = true
turno = 1
//no hace nada, espera ocupada
while(interesado[1]&&turno==1);
sección_crítica();
//fin de la sección crítica
interesado[0] = false
sección_no_critica();

Hilo 1
interesado[1] = true
turno = 0
//no hace nada, espera ocupada
while(interesado[0]&&turno==0);
sección_crítica();
//fin de la sección crítica
interesado[1] = false
sección_no_critica();
```

9.- El algoritmo de la panadería de Lamport es un algoritmo de computación creado por el científico en computación Dr Leslie Lamport, para implementar la exclusión mutua de N procesos o hilos de ejecución sin necesidad de ningún soporte HW específico. Se inspira en el funcionamiento normal de cualquier comercio con un tendero y múltiples clientes (ej. una panadería). En este escenario, un cliente que llega a la panadería coge un número con su turno y espera pacientemente a ser atendido. Sin embargo, obtener el número para el turno no es trivial en un computador debido a la posibilidad de que varios procesos reciban el mismo. El siguiente pseudo-código (fuente: wikipedia) ilustra la solución de Lamport a dicho problema:

```
// Variables globales
Num[N] = {0, 0, 0, ..., 0};
Eligiendo[N] = {falso, falso, falso, ..., falso};

//Código del hilo i-ésimo
Hilo(i) {
    loop {
        //Calcula el número de turno
        Eligiendo[i] = cierto;
        Num[i] = 1 + max(Num[1],..., Nun[N]);
        Eligiendo[i] = falso;

        //Compara con todos los hilos
        for j in 1..N {
            //Si el hilo j está calculando su número, espera a que termine
            while( Eligiendo[j] ) {yield()}

            while( Num[j]!=0 && (Num[j]<Num[i] || (Num[j]==Num[i] && i<j)) ) {yield()}

        }
        // Sección crítica
        ...
        // Fin de sección crítica

        Número[i] = 0;

        // Código restante
    }
}
```

Discutir la validez de la solución de Lamport (seguridad, interbloqueo, inanición).

10.- Codificar el problema de los **lectores/escritores** de forma que sean los escritores los que tengan prioridad de acceso (conocido como “Segundo Problema de los Lectores/Escritores”). Emplear como método de comunicación/sincronización únicamente semáforos.

**11.-** En el problema de los **lectores/escritores**, si se prioriza a un colectivo, ya sea lectores o escritores, es posible que los procesos priorizados nieguen el acceso al recurso compartido al otro colectivo y que, por lo tanto, se produzca inanición (*starvation*). Codificar una solución para el problema de los lectores/escritores en la cual se otorgue acceso en función del orden de solicitud (conocido como “Tercer Problema de los Lectores/Escritores”). Emplear para ello semáforos.

**12.- El Barbero Dormilón:** una barbería está compuesta por una sala de espera, con  $n$  sillas, y la sala del barbero, que tiene un sillón para el que está siendo atendido. Las condiciones de atención a los clientes son las siguientes:

- a) Si no hay ningún cliente, el barbero se va a dormir
- b) Si entra un cliente en la barbería y todas las sillas están ocupadas, el cliente abandona la barbería
- c) Si hay sitio y el barbero está ocupado, se sienta en una silla libre
- d) Si el barbero estaba dormido, el cliente le despierta
- e) Una vez que el cliente va a ser atendido, el barbero invocará `cortarPelo()` y el cliente `recibirCortePelo()`

Escriba un programa que coordine al barbero y a los clientes utilizando mutex y semáforos POSIX para la sincronización entre procesos.

**13.- El problema de la montaña rusa [Andrews's Concurrent Programming]:** suponga que hay un hilo por cada pasajero, haciendo un total de  $n$ , y un hilo para el coche. Los pasajeros están constantemente esperando y subiéndose a la montaña rusa, que puede llevar  $C$  pasajeros ( $C < n$ ). El coche sólo puede comenzar un viaje cuando está lleno. Se deben cumplir las siguientes restricciones:

- a) Los pasajeros deben invocar `board()` y `unboard()`
- b) El coche debe invocar `load()`, `run()` y `unload()`
- c) Los pasajeros no pueden hacer `board()` hasta que el coche haya invocado `load()`
- d) El coche no puede partir (`run()`) hasta que no haya  $C$  pasajeros en él.
- e) Los pasajeros no se pueden bajar hasta que el coche invoque `unload()`

A partir de las especificaciones realice los siguiente:

- 1) Escriba el código necesario usando semáforos POSIX generales (y variables enteras, booleanas...)
- 2) Escriba el código necesario usando cerrojos y variables condicionales (y variables enteras, booleanas...)

**14.- El problema del sushi bar [Kenneth Reek]:** supóngase un restaurante japonés con 5 asientos, si un cliente llega cuando hay un asiento libre puede sentarse inmediatamente. Sin embargo, si un cliente llega y encuentra los 5 asientos ocupados, supondrá que todos los clientes están cenando juntos y esperará hasta que todos ellos se levanten antes de tomar asiento. Además, los clientes son atendidos por orden. Codifique un programa que se comporte como un cliente según las especificaciones anteriores usando semáforos generales.

**15.- El problema del cuidado de niños [Max Hailperin]:** en cierta guardería se debe de cumplir que por cada tres niños debe de estar presente al menos un adulto. Codificar (con semáforos generales) el código correspondiente a los adultos de manera que se cumpla esta restricción y suponiendo que el número de niños se mantiene constante.

Amplíe la solución anterior para que tenga en cuenta la posible variación de la cantidad de niños. Codifique el hilo correspondiente a los niños.