

T2-Procesos



Índice

- Conceptos relacionados con la Gestión de procesos
- Servicios básicos para gestionar procesos (basado en Linux)
- Comunicación entre procesos
 - Signals Linux y Sincronización
- Gestión interna de los procesos
 - Datos: PCB
 - Estructuras de gestión: Listas, colas etc, relacionadas principalmente con el estado del proceso
 - Mecanismo de cambio de contexto. Concepto y pasos básicos
 - Planificación
- Relación entre las llamadas a sistema de gestión de procesos y la gestión interna del S.O.
- Protección y seguridad

Definición

Tareas del sistema operativo

Concurrencia y paralelismo

Estados de los procesos

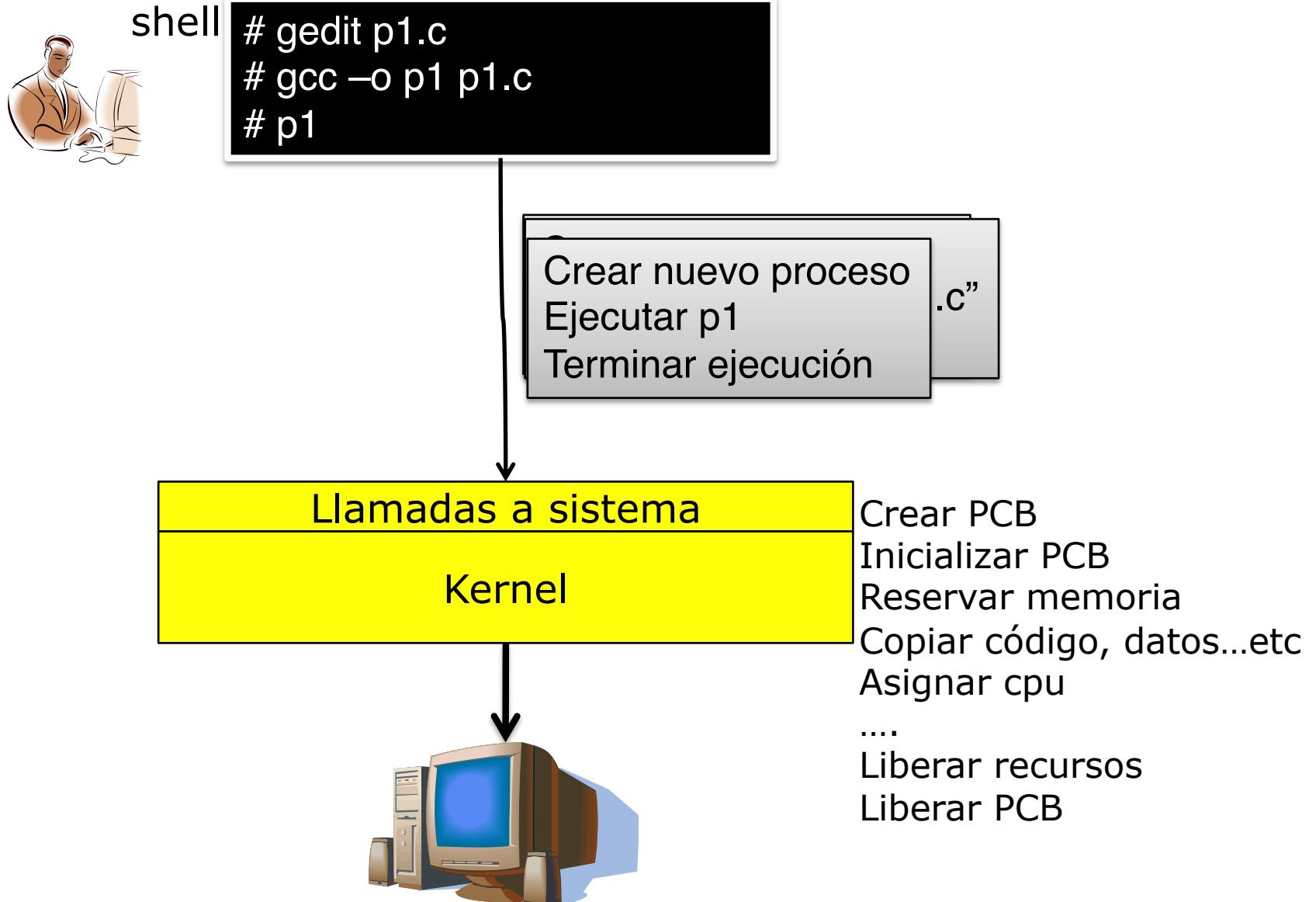
Propiedades de un proceso en Linux

CONCEPTOS

Concepto de proceso

- Un proceso es la representación del SO de un programa en ejecución.
- Un programa ejecutable básicamente es un código y una definición de datos, al ponerlo en ejecución necesitamos:
 - Asignarle memoria para el código, los datos y la pila
 - Inicializar los registros de la cpu para que se empiece a ejecutar
 - Ofrecer acceso a los dispositivos (ya que necesitan acceso en modo kernel)
 - Muchas más cosas que iremos viendo
- Para gestionar la información de un proceso, el sistema utiliza una estructura de datos llamada PCB (Process Control Block)
- Cada vez que ponemos un programa a ejecutar, se crea un nuevo proceso
 - Pueden haber limitaciones en el sistema

Procesos: ¿Como se hace?



Process Control Block (PCB)

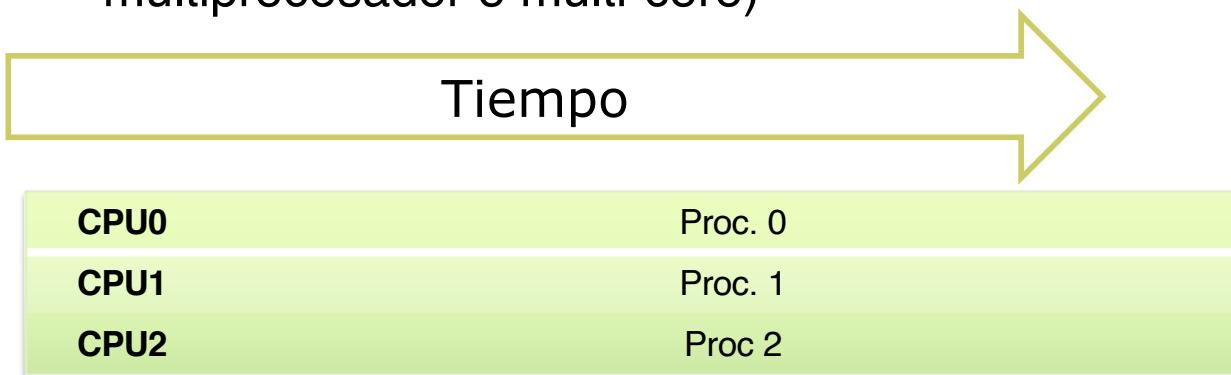
- Contiene la información que el sistema necesita para gestionar un proceso. Esta información depende del sistema y de la máquina. Se puede clasificar en estos 3 grupos
 - Espacio de direcciones
 - ▶ descripción de las regiones del proceso: código, datos, pila, ...
 - Contexto de ejecución
 - ▶ SW: PID, información para la planificación, información sobre el uso de dispositivos, estadísticas,...
 - ▶ HW: tabla de páginas, program counter, ...

Utilización eficiente de la CPU

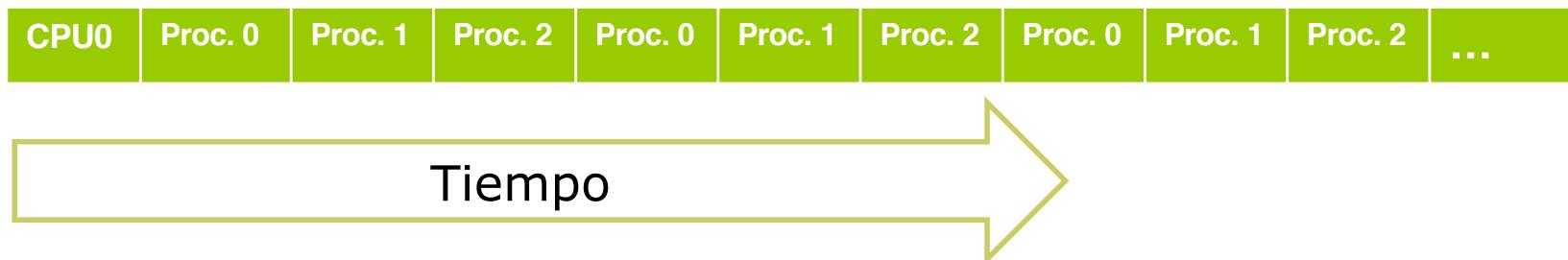
- En un sistema de propósito general, lo habitual es tener varios procesos a la vez, de forma que se aproveche al máximo los recursos de la máquina
- ¿Por qué nos puede interesar ejecutar múltiples procesos simultáneamente?
 - Si tenemos varios procesadores podemos ejecutar más procesos a la vez, o uno mismo usando varios procesadores
 - Aprovechar el tiempo de acceso a dispositivos (Entrada/Salida) de un proceso para que otros procesos usen la CPU
- Si el SO lo gestiona bien, consigue la ilusión de que la máquina tiene más recursos (CPUs) de los que tiene realmente

Concurrencia

- Concurrencia es la **capacidad** de ejecutar varios procesos de forma simultánea
 - Si realmente hay varios a la vez es paralelismo (arquitectura multiprocesador o multi-core)



- Si es el SO el que genera un paralelismo virtual mediante compartición de recursos se habla de concurrencia



Concurrencia

- Se dice que varios procesos son concurrentes cuando se tienen la capacidad de ejecutarse en paralelo si la arquitectura lo permite
- Se dice que varios procesos son secuenciales si, independientemente de la arquitectura, se ejecutarán uno después del otro (cuando termina uno empieza el siguiente). En este caso, es el programador el que fuerza que esto sea así mediante sincronizaciones.
 - ▶ Poniendo un waitpid entre un fork y el siguiente
 - ▶ Mediante signals (eventos)
- **Paralelismo** es cuando varios procesos concurrentes se ejecutan de forma simultánea:
 - Depende de la máquina
 - Depende del conjunto de procesos
 - Depende del SO

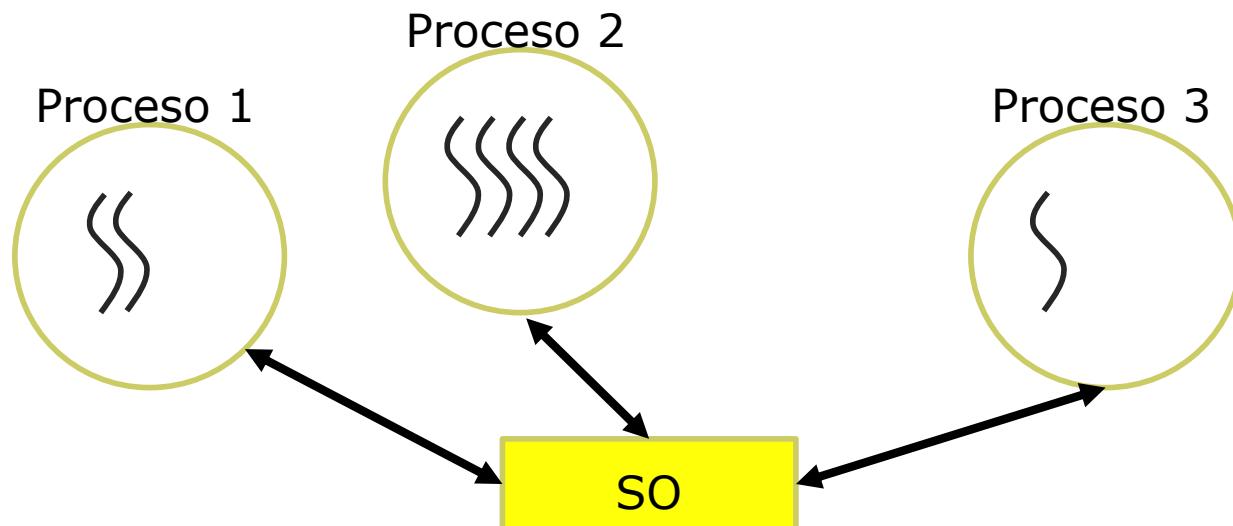
Hilos de Ejecución (Threads) – Qué son?

- Si entramos más en detalle en el concepto de Proceso...
 - representación del SO de un programa en ejecución

...podemos afirmar que un **Proceso** es la unidad de asignación de recursos de un programa en ejecución (memoria, dispositivos E/S, threads)
- Entre los recursos está el/los **hilo/s de ejecución (thread)** de un proceso
 - Se trata de la instancia/flujo de ejecución de un proceso y es la mínima unidad de planificación del SO (asignación de tiempo de CPU)
 - ▶ Cada parte del código que se puede ejecutar de forma independiente se le puede asociar un thread
 - Tiene asociado el contexto necesario para seguir el flujo de ejecución de las instrucciones
 - ▶ Identificador (Thread ID: TID)
 - ▶ Puntero a Pila (Stack Pointer)
 - ▶ Puntero a siguiente instrucción a ejecutar (Program Counter),
 - ▶ Registros (Register File)
 - ▶ Variable errno
 - Los threads **comparten** los recursos del proceso (PCB, memoria, dispositivos E/S)

Hilos de Ejecución (Threads)

- Un proceso tiene un thread al inicio de la ejecución
- Un proceso puede tener varios threads
 - Ej.: videojuegos actuales de altas prestaciones tienen **>50 threads**; Firefox/Chrome tienen **>80 threads**
- En la siguiente figura: Proceso1 tiene 2 threads; el Proceso2 tiene 4 threads; el Proceso3 tiene 1 thread
- La gestión de procesos con varios threads dependerá del soporte del SO
 - **User Level Threads vs Kernel Level Threads**



Hilos de Ejecución (Threads) – Para qué?

- Cuando y para qué se usan...
 - Explotar paralelismo (del código y de recursos hardware)
 - Encapsular tareas (programación modular)
 - Eficiencia en la E/S (Threads específicos para E/S)
 - Pipelining de solicitudes de servicio (para mantener QoS de servicios)
- Ventajas
 - Los threads tienen menor coste al crear/terminar y al cambiar de contexto (dentro del mismo proceso) que los procesos
 - Al compartir memoria entre threads de un mismo proceso, pueden intercambiar información sin llamadas al sistema
- Desventajas
 - Difícil de programar y *debuggar* debido a la memoria compartida
 - ▶ Problemas de sincronización y exclusión mutua
 - Ejecuciones incoherentes, resultados erróneos, bloqueos, etc.

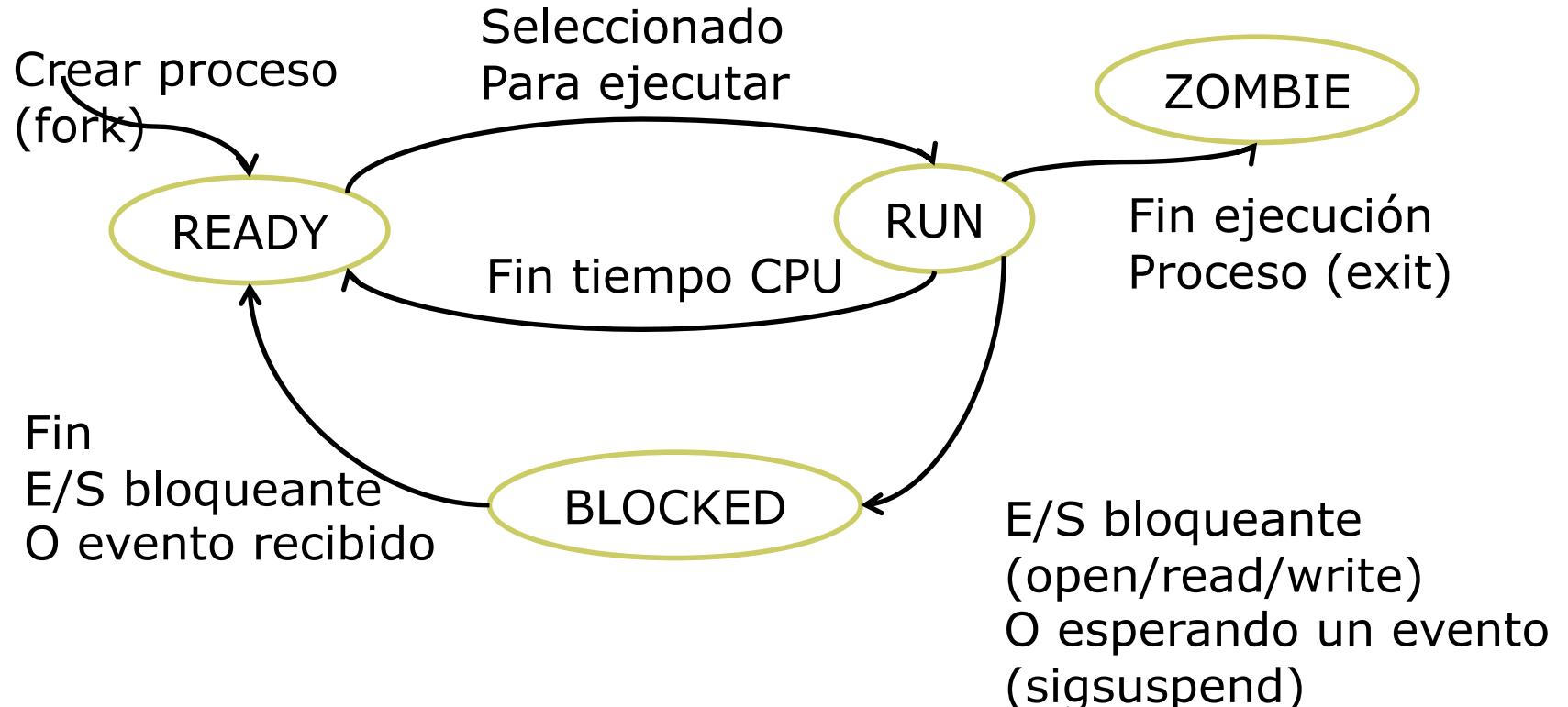
Estados de un proceso

- No es normal que un proceso esté todo el tiempo utilizando la CPU, durante periodos de su ejecución podemos encontrarlo:
 - Esperando datos de algún dispositivo : teclado, disco, red, etc
 - Esperando información de otros procesos
- Para aprovechar el HW tenemos:
 - Sistemas multiprogramados: múltiples procesos activos. Cada proceso tiene su información en su propio PCB.
 - El SO sólo asigna CPU a aquellos procesos que están utilizándola.
- El SO tiene “clasificados” los procesos en función de “que están haciendo”, normalmente a esto se llama el “**ESTADO**” del proceso
- El **estado** suele gestionarse o con un campo en el PCB o teniendo diferentes listas o colas con los procesos en un estado concreto.

Estados de un proceso

- Cada SO define un grafo de estados, indicando que eventos generan transiciones entre estados.
- El grafo define que transiciones son posibles y como se pasa de uno a otro
- El grafo de estados, muy simplificado, podría ser:
 - **run**: El proceso tiene asignada una CPU y está ejecutándose
 - **ready**: El proceso está preparado para ejecutarse pero está esperando que se le asigne una CPU
 - **blocked**: El proceso no tiene/consume CPU, está *bloqueado* esperando un que finalice una entrada/salida de datos o la llegada de un evento
 - **zombie**: El proceso ha terminado su ejecución pero aún no ha desaparecido de las estructuras de datos del kernel
- ▶ Linux

Ejemplo diagrama de estados



Los estados y las transiciones entre ellos dependen del sistema.
Este diagrama es solo un ejemplo

Ejemplo estados de un proceso

- Objetivo: Hay que entender la relación entre las características del SO y el diagrama de estados que tienen los procesos
 - Si el sistema es multiprogramado → READY, RUN
 - Si el sistema permite e/s bloqueante → BLOCKED
 - Etc
- SO con soporte para procesos con multiples threads
 - Estructuras para diferenciar threads del mismo proceso y gestionar estados de ejecución, entre otras cosas
 - ▶ P.Ej.: Light Weight Process (LWP) en SOs basados en Linux/UNIX

Linux: Propiedades de un proceso

- Un proceso incluye, no sólo el programa que ejecuta, sino toda la información necesaria para diferenciar una ejecución del programa de otra.
 - **Toda esta información se almacena en el kernel, en el PCB.**
- En Linux, por ejemplo, las propiedades de un proceso se agrupan en tres:
la identidad, el entorno, y el contexto.
- **Identidad**
 - Define quién es (identificador, propietario, grupo) y qué puede hacer el proceso (recursos a los que puede acceder)
- **Entorno**
 - Parámetros (argv en un programa en C) y variables de entorno (HOME, PATH, USERNAME, etc)
- **Contexto**
 - Toda la información que define el estado del proceso, todos sus recursos que usa y que ha usado durante su ejecución.

Linux: Propiedades de un proceso (2)

- La **IDENTIDAD** del proceso define quien es y por lo tanto determina que puede hacer
 - **Process ID (PID)**
 - ▶ Es un identificador **ÚNICO** para el proceso. Se utiliza para identificar un proceso dentro del sistema. En llamadas a sistema identifica al proceso al cual queremos enviar un evento, modificar, etc
 - ▶ El kernel genera uno nuevo para cada proceso que se crea
 - **Credenciales**
 - ▶ Cada proceso está asociado con un usuario (**userID**) y uno o más grupos (**groupID**). Estas credenciales determinan los derechos del proceso a acceder a los recursos del sistema y ficheros.

-
- Creación
 - Mutación (carga de un ejecutable nuevo)
 - Finalización
 - Espera

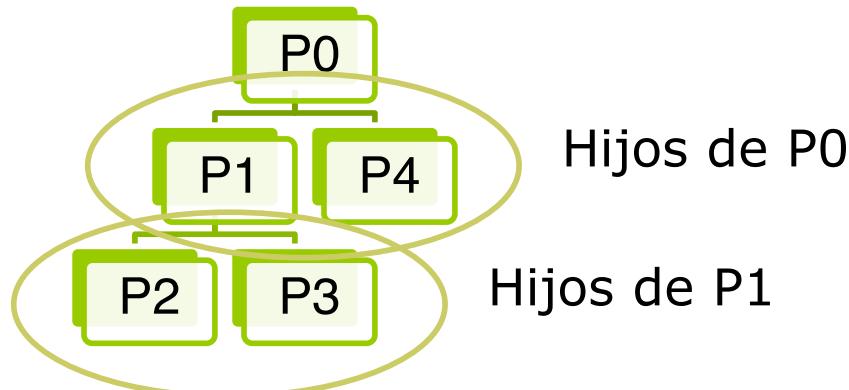
SERVICIOS BÁSICOS PARA GESTIONAR PROCESOS

Servicios y funcionalidad

- El sistema nos ofrece como usuarios un conjunto de funciones (llamadas a sistema) para gestionar procesos
 - Crear /Planificar/Eliminar procesos
 - Bloquear/Desbloquear procesos
 - Proporcionar mecanismos de sincronización
 - Proporcionar mecanismos de comunicación
 - ▶ Memoria compartida
 - ▶ Dispositivos especiales
 - ▶ Gestión de signals
- En este curso **NO** entraremos en detalle en llamadas al sistema vinculadas a los threads

Creación de procesos

- Cuando un proceso crea otro, se establece una relación jerárquica que se denomina **padre-hijo**. A su vez, el proceso hijo (y el padre) podrían crear otros procesos generándose un **árbol de procesos**.



- Los procesos se identifican en el sistema mediante un *process identifier* (PID)
- El SO decide aspectos como por ejemplo:
 - **Recursos**: El proceso hijo, ¿comparte los recursos del padre?
 - **Planificación**: El proceso hijo, ¿se ejecuta antes que el padre?
 - **Espacio de direcciones**. ¿Qué código ejecuta el proceso hijo? ¿El mismo? ¿Otro?

Creación de procesos: opciones(Cont)

- Planificación
 - **El padre y el hijo se ejecutan concurrentemente (UNIX)**
 - El padre espera hasta que el hijo termina (se sincroniza)
- Espacio de direcciones (rango de memoria válido)
 - **El hijo es un duplicado del padre (UNIX), pero cada uno tiene su propia memoria física. Además, padre e hijo, en el momento de la creación, tienen el mismo contexto de ejecución (los registros de la CPU valen lo mismo)**
 - El hijo ejecuta un código diferente
- UNIX
 - **fork** system call. Crea un nuevo proceso. El hijo es un clon del padre
 - **exec** system call. Reemplaza (muta) el espacio de direcciones del proceso con un nuevo programa. El proceso es el mismo.



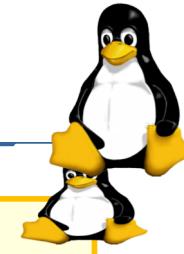
Servicios básicos (UNIX)

Servicio	Llamada a sistema
Crear proceso	fork
Cambiar ejecutable=Mutar proceso	exec (execlp)
Terminar proceso	exit
Esperar a proceso hijo (bloqueante)	wait/waitpid
Devuelve el PID del proceso	getpid
Devuelve el PID del padre del proceso	getppid



- Una llamada a sistema bloqueante es aquella que puede bloquear al proceso, es decir, forzar que deje el estado RUN (abandone la CPU) y pase a un estado en que no puede ejecutarse (WAITING, BLOCKED,, depende del sistema)

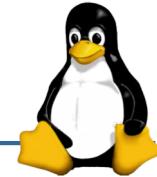
Crear proceso: fork en UNIX



```
int fork();
```

- Un proceso crea un proceso nuevo. Se crea una relación jerárquica padre-hijo
- El padre y el hijo se ejecutan de forma **concurrente**
- La memoria del hijo se inicializa con una copia de la memoria del padre
 - Código/Datos/Pila
- El hijo inicia la ejecución en el punto en el que estaba el padre en el momento de la creación
 - Program Counter hijo= Program Counter padre
- Valor de retorno del fork es diferente (es la forma de diferenciarlos en el código):
 - ▶ Padre recibe el PID del hijo
 - ▶ Hijo recibe un 0.

Creación de procesos y herencia



- El hijo HEREDA algunos aspectos del padre y otros no.
- **HEREDA** (recibe una copia privada de....)
 - El espacio de direcciones lógico (código, datos, pila, etc).
 - ▶ La memoria física es nueva, y contiene una copia de la del padre (en el tema 3 veremos optimizaciones en este punto)
 - La tabla de programación de signals
 - Los dispositivos virtuales
 - El usuario /grupo (credenciales)
 - Variables de entorno
- **NO HEREDA** (sino que se inicializa con los valores correspondientes)
 - PID, PPID (PID de su padre)
 - Contadores internos de utilización (Accounting)
 - Alarmas y signals pendientes (son propias del proceso)

Terminar ejecución/Esperar que termine



- Un proceso puede acabar su ejecución **voluntaria** (exit) o **involuntariamente (signals)**
- Cuando un proceso quiere finalizar su ejecución (**voluntariamente**), liberar sus recursos y liberar las estructuras de kernel reservadas para él, se ejecuta la llamada a sistema exit.
- Si queremos sincronizar el padre con la finalización del hijo, podemos usar waitpid: El proceso espera (si es necesario se bloquea el proceso) a que termine un hijo cualquiera o uno concreto
 - ▶ waitpid(-1,NULL,0) → Esperar (con bloqueo si es necesario) a un hijo cualquiera
 - ▶ waitpid(pid_hijo,NULL,0)→ Esperar (con bloqueo si es necesario) a un hijo con pid=pid_hijo
- El hijo puede enviar información de finalización (exit code) al padre mediante la llamada a sistema exit y el padre la recoge mediante wait o waitpid
 - ▶ El SO hace de intermediario, la almacena hasta que el padre la consulta
 - ▶ Mientras el padre no la consulta, el PCB no se libera y el proceso se queda en estado ZOMBIE (defunct)
 - Conviene hacer wait/waitpid de los procesos que creamos para liberar los recursos ocupados del kernel
- Si un proceso muere sin liberar los PCB's de sus hijos el proceso init del sistema los libera

```
void exit(int);
pid_t waitpid(pid_t pid, int *status, int options);
```



pid_t waitpid(pid_t pid, int *status, int options);

Parámetro pid== -1 Estado hijos al hacer waitpid	options==0	options==WNOHANG
Algún hijo zombie	No se bloquea Trata la muerte de un zombie (no está especificado cual) Devuelve el pid del hijo tratado	Idem que options==0
Todos los hijos vivos	Se bloquea hasta que uno acaba (cualquiera) Trata la muerte del que haya acabado Devuelve el pid del hijo tratado	No se bloquea Devuelve 0
Parámetro pid==pidh Estado de pidh al hacer waitpid	options==0	options==WNOHANG
Zombie	No se bloquea Trata la muerte del hijo Devuelve pidh	Idem que flags=0
Vivo	Se bloquea hasta que acaba Trata la muerte del hijo Devuelve pidh	No se bloquea Devuelve 0



Mutación de ejecutable: exec en UNIX

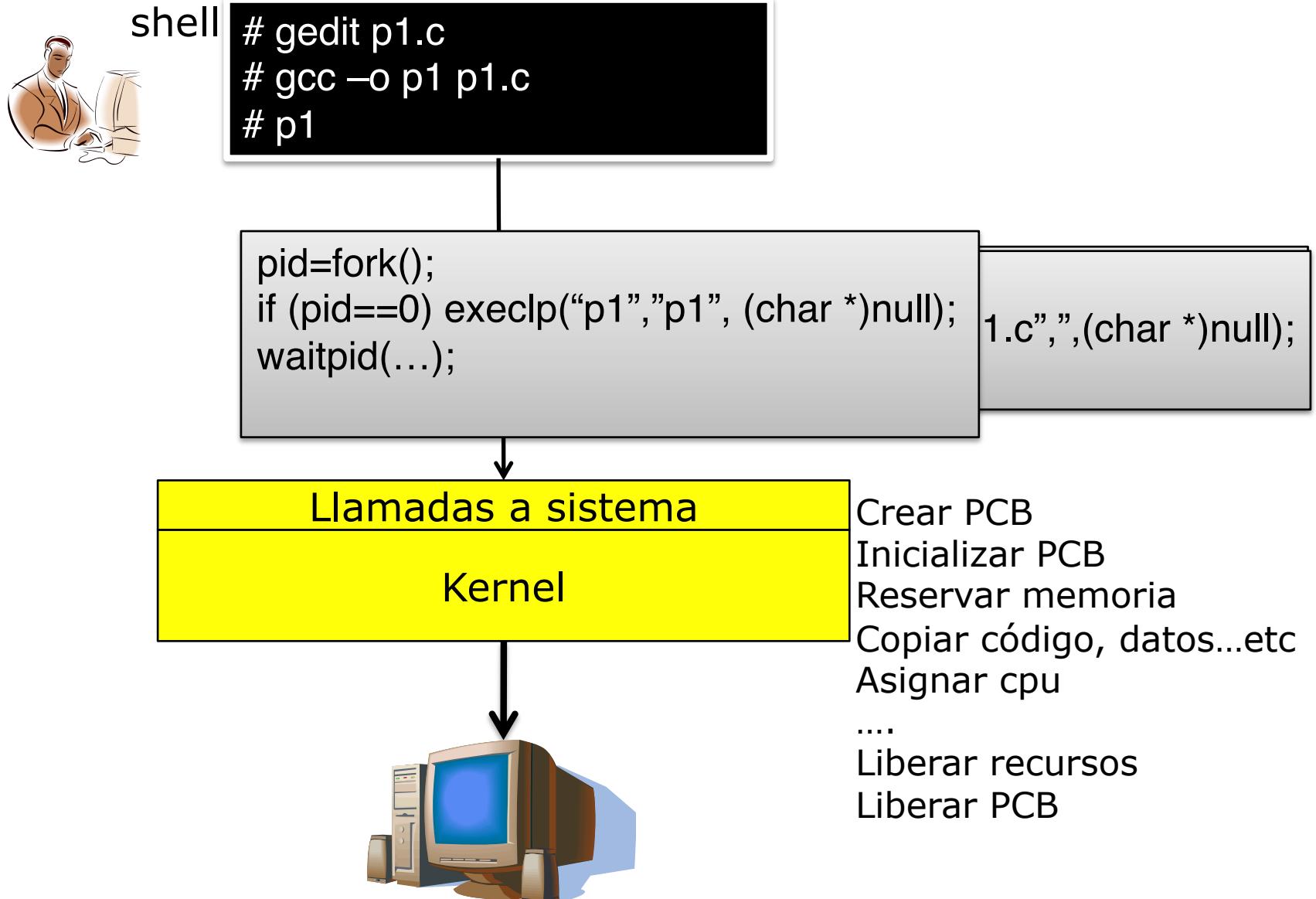
- Al hacer fork, el espacio de direcciones es el mismo. Si queremos ejecutar otro código, el proceso debe MUTAR (Cambiar el binario de un proceso)
- execlp: Un proceso cambia (muta) su propio ejecutable por otro ejecutable (pero el proceso es el mismo)
 - Todo el contenido del espacio de direcciones cambia, código, datos, pila, etc.
 - ▶ Se reinicia el contador de programa a la primera instrucción (main)
 - Se mantiene todo lo relacionado con la identidad del proceso
 - ▶ Contadores de uso internos , signals pendientes, etc
 - Se modifican aspectos relacionados con el ejecutable o el espacio de direcciones
 - ▶ Se define por defecto la tabla de programación de signals

```
int execlp(const char *file, const char *arg, ...);
```



EJEMPLOS GESTIÓN PROCESOS

Procesos: Ahora ya sabemos como se hace



Creación procesos

Caso 1: queremos que hagan líneas de código diferente

1. int ret=fork();
2. if (ret==0) {
3. // estas líneas solo las ejecuta el hijo, tenemos 2 procesos
4. }else if (ret<0){
5. // En este caso ha fallado el fork, solo hay 1 proceso
6. }else{
7. // estas líneas solo las ejecuta el padre, tenemos 2 procesos
8. }
9. // estas líneas las ejecutan los dos

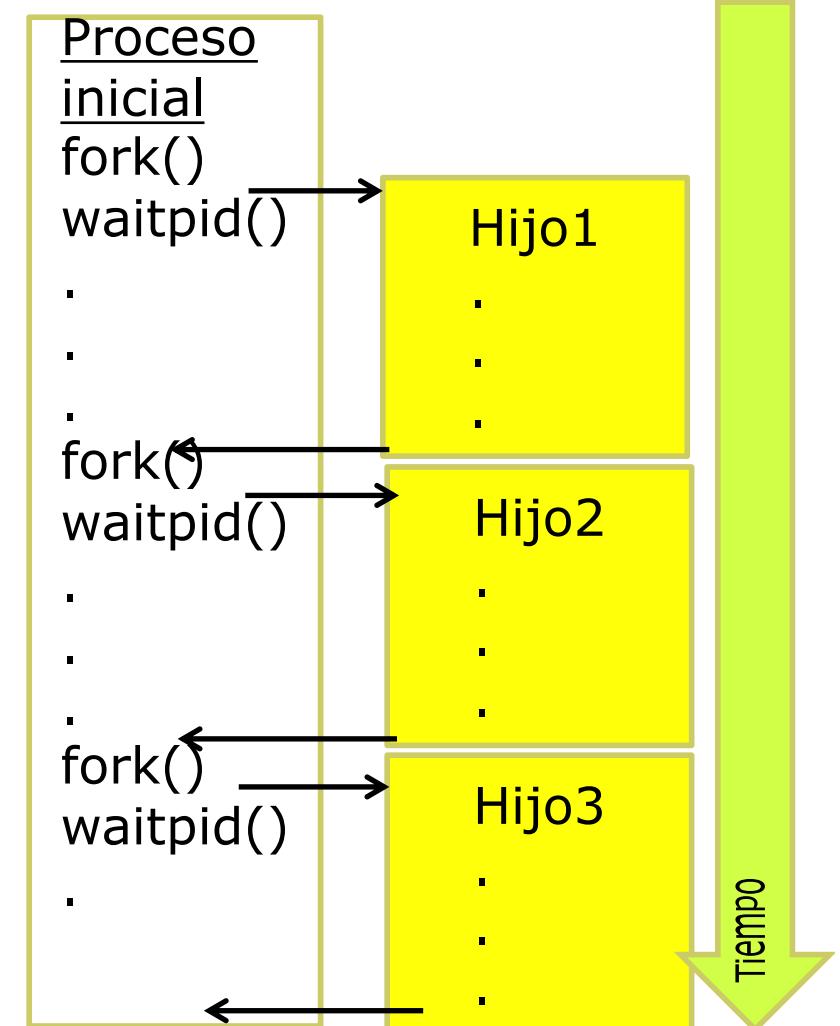
Caso 2: queremos que hagan lo mismo

1. fork();
2. // Aquí, si no hay error, hay 2 procesos

Esquema Secuencial

Secuencial: Forzamos que el padre espere a que termine un hijo antes de crear el siguiente.

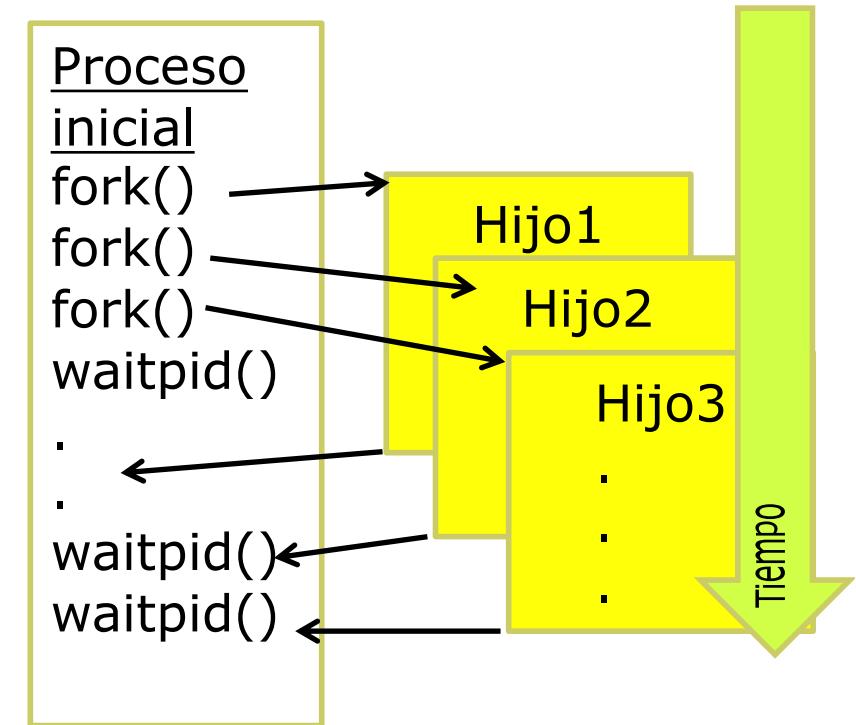
```
1. #define num_procs 2
2. int i,ret;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // estas líneas solo las ejecuta el
7.         // hijo
8.         codigohijo();
9.         exit(0); //
10.    }
11.    waitpid(-1,NULL,0);
12.}
```



Esquema Concurrente

Concurrente; Primero creamos todos los procesos, que se ejecutan concurrentemente, y después esperamos que acaben..

```
1. #define num_procs 2
2. int ret,i;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // estas líneas solo las ejecuta el
7.         // hijo
8.         codigohijo();
9.         exit(0); //
10.    }
11. }
12. while( waitpid(-1,NULL,0)>0);
```



Ejemplo con fork

- ¿Qué hará este código?

```
1. int id;  
2. char buffer[128];  
3. id=fork();  
4. sprintf(buffer,"fork devuelve %d\n",id);  
5. write(1,buffer,strlen(buffer));
```



- ¿Qué hará si el fork funciona?
- ¿Qué hará si el fork falla?
- PROBADLO!!

Ejemplos con fork

- ¿Cuántos procesos se crean en este fragmento de código?

```
...  
fork();  
fork();  
fork();
```



- ¿Y en este otro fragmento?

```
...  
for (i = 0; i < 10; i++)  
    fork();
```



- ¿Qué árbol de procesos se genera?

Ejemplos con fork

- Si el pid del proceso padre vale 10 y el del proceso hijo vale 11

```
int id1, id2, ret;
char buffer[128];
id1 = getpid();
ret = fork();
id2 = getpid();
sprintf(buffer,"Valor de id1: %d; valor de ret: %d; valor de id2: %d\n",
        id1, ret, id2);
write(1,buffer,strlen(buffer));
```



- ¿Qué mensajes veremos en pantalla?
- ¿Y ahora?

```
int id1,ret;
char buffer[128];
id1 = getpid(); /* getpid devuelve el pid del proceso que la ejecuta */
ret = fork();
id1 = getpid();
sprintf(buffer,"Valor de id1: %d; valor de ret: %d", id1, ret);
write(1,buffer,strlen(buffer));
```



Ejemplo: fork/exit (examen)



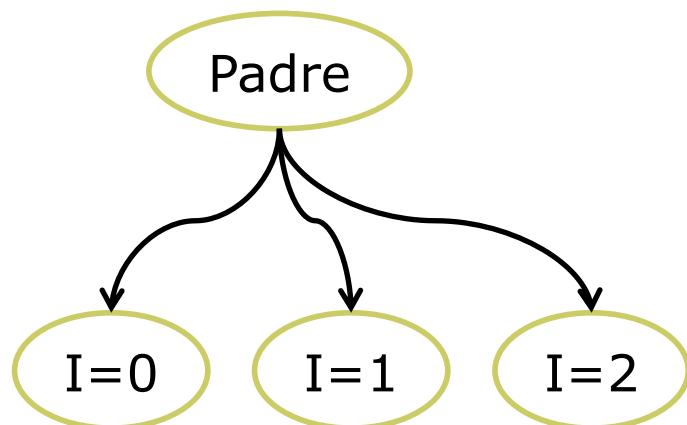
```
void main()
{
    char buffer[128];
    ...
    sprintf(buffer, "Mi PID es el %d\n", getpid());
    write(1,buffer,strlen(buffer));
    for (i = 0; i < 3; i++) {
        ret = fork();
        if (ret == 0)
            hacerTarea();
    }
    while (1);
}

void hacerTarea()
{
    char buffer[128];
    sprintf("Mi PID es %d y el de mi padre %d\n", getpid(), getppid());
    write(1,buffer,strlen(buffer));
    exit(0); ← Ahora, probad a quitar esta instrucción,  
es muy diferente!!!
}
```

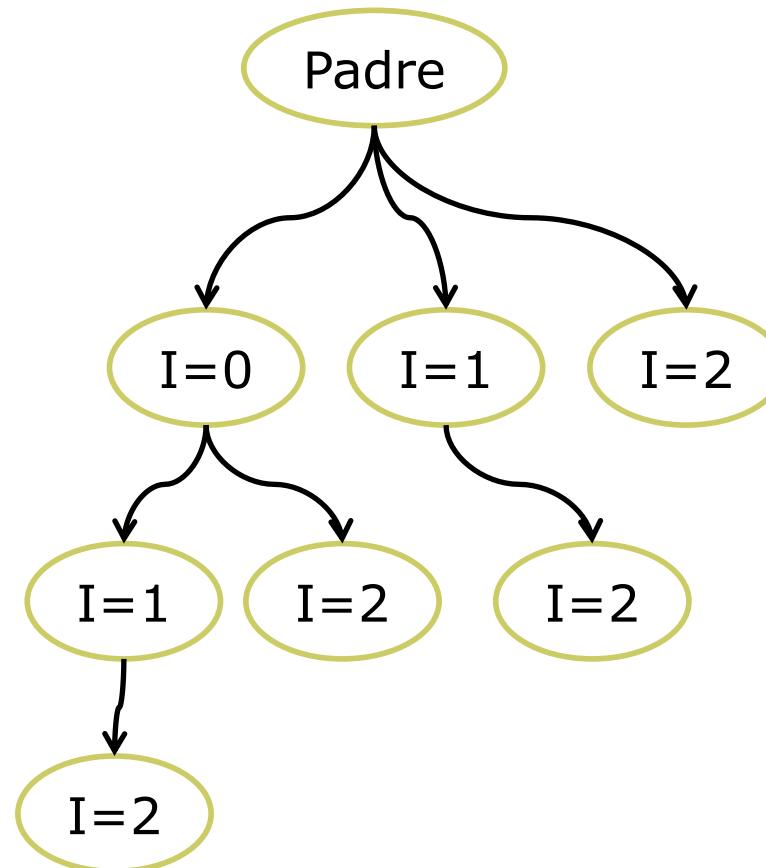
Podéis encontrar el código completo en: Nprocesos.c y NprocesosExit.c

Árbol de procesos (examen)

Con exit

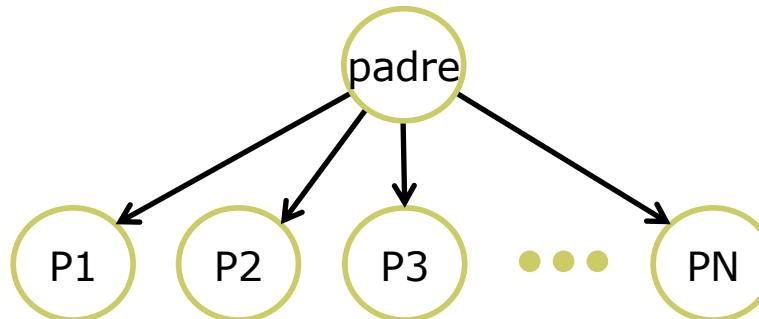


Sin exit

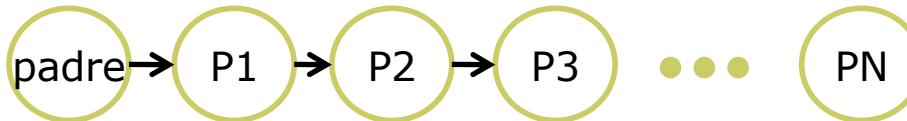


Otros ejemplos con fork

- Escribid un programa que cree N procesos según el siguiente árbol de procesos:



- Modificad el código anterior para que los cree según este otro árbol de procesos:



Ejemplo fork+exec

```
fork();
execp("/bin/procB","procB", (char *) 0);
while(...)
```

progA

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

progB

P1

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

P2

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

Ejemplo fork+exec

```
int pid;  
pid=fork();  
if (pid==0) execvp("/bin/procB","procB", (char *) 0);  
while(...)
```

progA

```
main(...){  
int A[100],B[100];  
...  
for(i=0;i<10;i++) A[i]=A[i]+B[i];  
...}
```

progB

P1

```
int pid;  
pid=fork();  
if (pid==0) execvp(.....);  
while(...)
```

P2

```
main(...){  
int A[100],B[100];  
...  
for(i=0;i<10;i++) A[i]=A[i]+B[i];  
...}
```

Ejemplo: exec

- Cuando en el *shell* ejecutamos el siguiente comando:

```
% ls -l
```

1. Se crea un nuevo proceso (*fork*)
2. El nuevo proceso cambia la imagen, y ejecuta el programa ls (*exec*)

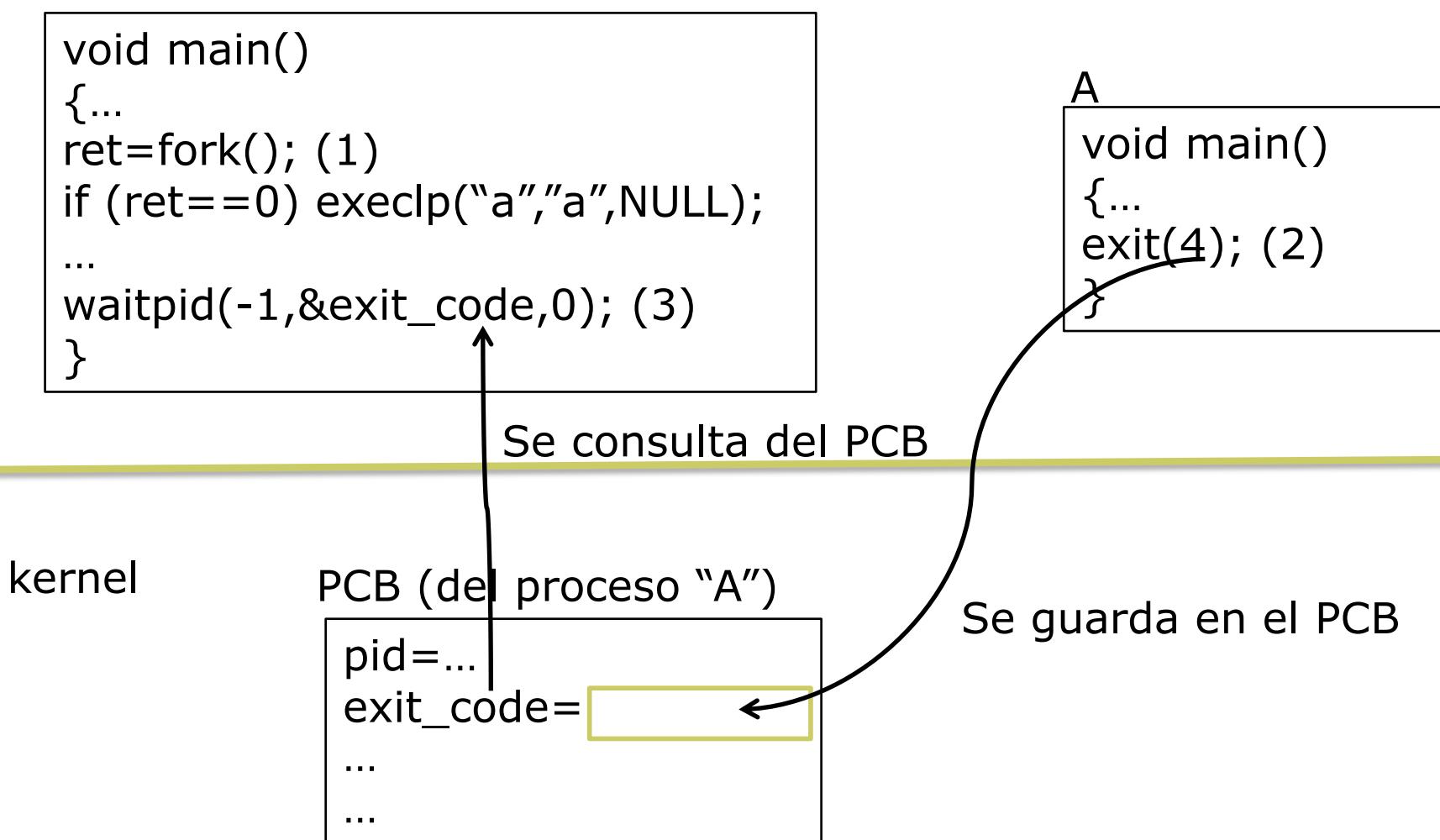
- Como se implementa esto?

```
...
ret = fork();
if (ret == 0) {
    execp("/bin/ls", "ls", "-l", (char *)NULL);
}
// A partir de aquí, este código sólo lo ejecutaría el padre
...
```



- ¿Hace falta poner un *exit* detrás del *execp*?
- ¿Qué pasa si el *execp* falla?

Terminación de procesos. exit



Sin embargo, `exit_code` no vale 4!!! Hay que procesar el resultado

Ejemplo: fork/exec/waitpid



```
// Usage: plauncher cmd [[cmd2] ... [cmdN]]  
  
void main(int argc, char *argv[])
{
    ...
    num_cmd = argc-1;
    for (i = 0; i < num_cmd; i++)
        lanzaCmd( argv[i+1] );
    // waitpid format
    // ret: pid del proceso que termina o -1
    // arg1== -1 → espera a un proceso hijo cualquiera
    // arg2 exit_code → variable donde el kernel nos copiara el valor de
    // finalización
    // argc3==0 → BLOQUEANTE
    while ((pid = waitpid(-1, &exit_code, 0) > 0)
        trataExitCode(pid, exit_code);
    exit(0);
}  
  
void lanzaCmd(char *cmd)
{
    ...
    ret = fork();
    if (ret == 0)
        execvp(cmd, cmd, (char *)NULL);
}
  
void trataExitCode(int pid, int exit_code) //next slide
...
```

Examinad los ficheros completos: *plauncher.c* y *Nplauncher.c*

trataExitCode



```
#include <sys/wait.h>

// PROGRAMADLA para los LABORATORIOS
void trataExitCode(int pid,int exit_code)
{
int exit_code,statcode,signcode;
char buffer[128];

if (WIFEXITED(exit_code)) {
    statcode = WEXITSTATUS(exit_code);
    sprintf(buffer,"El proceso %d termina con exit code %d\n", pid,
statcode);
    write(1,buffer,strlen(buffer));
}
else {
    signcode = WTERMSIG(exit_code);
    sprintf(buffer,"El proceso %d termina por el signal %d\n", pid,
signcode);
    write(1,buffer,strlen(buffer));
}
```

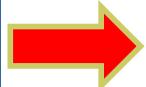
-Linux:Signals

COMUNICACIÓN ENTRE PROCESOS

Comunicación entre procesos

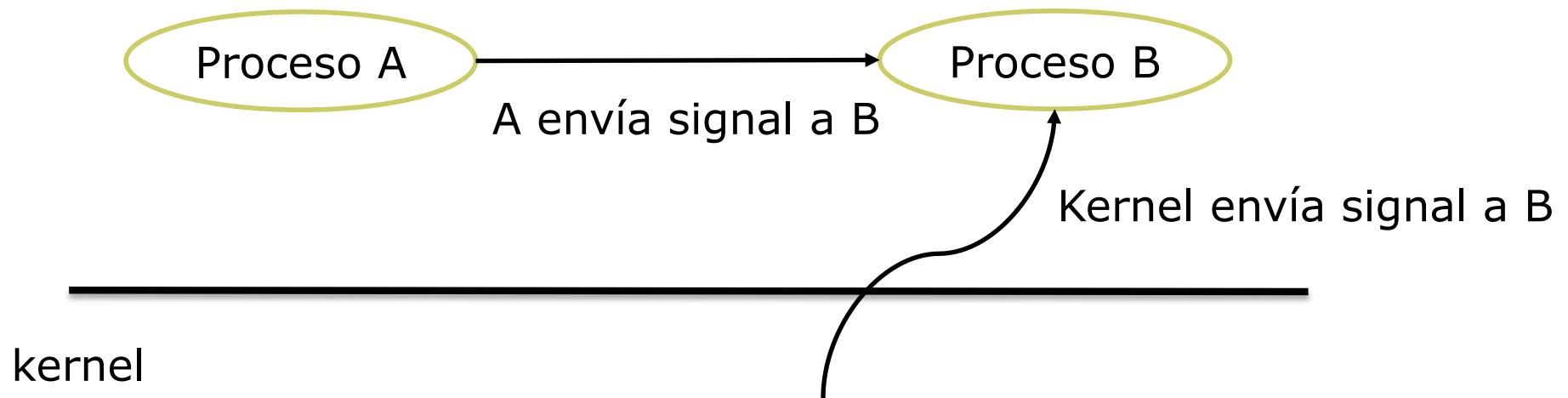
- Los procesos pueden ser independientes o cooperar entre si
- ¿Por qué puede ser útil que varios procesos cooperen?
 - Para compartir información
 - Para acelerar la computación que realizan
 - Por modularidad
- Para poder cooperar, los procesos necesitan comunicarse
 - Interprocess communication (IPC) = Comunicación entre procesos
- Para comunicar datos hay 2 modelos principalmente
 - Memoria compartida (Shared memory)
 - ▶ Los procesos utilizan variables que pueden leer/escribir
 - Paso de mensajes (Message passing)
 - ▶ Los procesos utilizan funciones para enviar/recibir datos

Comunicación entre procesos en Linux

- 
- Signals – Eventos enviados por otros procesos (del mismo usuario) o por el kernel para indicar determinadas condiciones (Tema 2)
 - Pipes – Dispositivo que permite comunicar dos procesos que se ejecutan en la misma máquina. Los primeros datos que se envían son los primeros que se reciben. La idea principal es conectar la salida de un programa con la entrada de otro. Utilizado principalmente por la shell (Tema 4)
 - FIFOs – Funciona con pipes que tienen un nombre en el sistema de ficheros. Se ofrecen como pipes con nombre. (Tema 4)
 - Sockets – Dispositivo que permite comunicar dos procesos a través de la red
 - Message queues – Sistema de comunicación indirecta
 - Semaphores - Contadores que permiten controlar el acceso a recursos compartidos. Se utilizan para prevenir el acceso de más de un proceso a un recurso compartido (por ejemplo memoria)
 - Shared memory – Memoria accesible por más de un proceso a la vez (Tema 3)

Signals: idea

- Signals: notificaciones que puede recibir un proceso para informarle de que ha sucedido un evento
- Los puede mandar el kernel o otros procesos del mismo usuario





Tipos de signals y tratamientos (I)

- Cada posible evento tiene un *signal* asociado
 - Los eventos y los signals asociados están predefinidos por el kernel
 - ▶ El signal es un número, pero existen constantes definidas para usarlas en los programas o en línea de comandos
- Hay dos signals que no están asociados a ningún evento para que el programador los use como quiera → SIGUSR1 y SIGUSR2
- **Cada proceso** tiene un **tratamiento** asociado a **cada signal**
 - Tratamientos por defecto
 - El proceso puede **capturar** (modificar el tratamiento asociado) todos los tipos de signals **excepto** SIGKILL y SIGSTOP

Tipos de signals y tratamientos(2)

■ Algunos signals

Nombre	Acción Defecto	Evento
SIGCHLD	IGNORAR	Un proceso hijo ha terminado o ha sido parado
SIGCONT		Continua si estaba parado
SIGSTOP	STOP	Parar proceso
SIGINT	TERMINAR	Interrumpido desde el teclado (CtrC)
SIGALRM	TERMINAR	El contador definido por la llamada alarm ha terminado
SIGKILL	TERMINAR	Terminar el proceso
SIGSEGV	CORE	Referencia inválida a memoria
SIGUSR1	TERMINAR	Definido por el usuario (proceso)
SIGUSR2	TERMINAR	Definido por el usuario (proceso)

■ Usos que les daremos principalmente

- Sincronización de procesos
- Control del tiempo (alarmas)

Tipos de signals y tratamientos(3)

- El tratamiento de un signal funciona como una interrupción provocada por software:
 - Al recibir un signal el proceso interrumpe la ejecución del código, pasa a ejecutar el tratamiento que ese tipo de signal tenga asociado y al acabar (si sobrevive) continúa donde estaba
- Los procesos pueden **bloquear/desbloquear** la recepción de **cada signal excepto SIGKILL y SIGSTOP** (tampoco se pueden bloquear los signals SIGFPE, SIGILL y SIGSEGV si son provocados por una excepción)
 - Cuando un proceso bloquea un signal, si se le envía ese signal el proceso no lo recibe y el sistema lo marca como pendiente de tratar
 - ▶ bitmap asociado al proceso, sólo recuerda un signal de cada tipo
 - Cuando un proceso desbloquea un signal recibirá y tratará el signal pendiente de ese tipo

Linux: Interfaz relacionada con signals

Servicio	Llamada sistema
Enviar un signal concreto	kill
Capturar/reprogramar un signal concreto	sigaction
Bloquear/desbloquear signals	sigprocmask
Esperar HASTA que llega un evento cualquiera (BLOQUEANTE)	sigsuspend
Programar el envío automático del signal SIGALRM (alarma)	alarm

- **Fichero con signals: /usr/include/bits/signum.h**
- Hay varios interfaces de gestión de signals incompatibles y con diferentes problemas, Linux implementa el interfaz POSIX

Interfaz: Enviar / Capturar signals

■ Para enviar:

```
int kill(int pid, int signum)
```



- signum → SIGUSR1, SIGUSR2, etc

- Requerimiento: conocer el PID del proceso destino

■ Para capturar un SIGNAL y ejecutar una función cuando llegue:

```
int sigaction(int signum, struct sigaction *tratamiento,  
struct sigaction *tratamiento_antiguo)
```



- signum → SIGUSR1, SIGUSR2, etc
- tratamiento → struct sigaction que describe qué hacer al recibir el signal
- tratamiento_antiguo → struct sigaction que describe qué se hacía hasta ahora. Este parámetro puede ser NULL si no interesa obtener el tratamiento antiguo

A envía un signal a B

- El proceso A envía (en algún momento) un signal a B y B ejecuta una acción al recibirlo

Proceso A

.....

```
Kill( pid, evento);
```

.....

Proceso B

```
int main()
{
    struct sigaction trat,viejo_trat;
    /* código para inicializar trat */
    sigaction(evento, &trat, &viejo_trat);
    ...
}
```

Definición de struct sigaction

- struct sigaction: varios campos. Nos fijaremos sólo en 3:
 - sa_handler: puede tomar 3 valores
 - ▶ SIG_IGN: ignorar el signal al recibirlo
 - ▶ SIG_DFL: usar el tratamiento por defecto
 - ▶ función de usuario con una cabecera predefinida: void nombre_funcion(int s);
 - IMPORTANTE: la función la invoca el kernel. El parámetro se corresponde con el signal recibido (SIGUSR1, SIGUSR2, etc), así se puede asociar la misma función a varios signals y hacer un tratamiento diferenciado dentro de ella.
 - sa_mask: signals que se añaden a la máscara de signals que el proceso tiene bloqueados
 - ▶ Si la máscara está vacía sólo se añade el signal que se está capturando
 - ▶ Al salir del tratamiento se restaura la máscara que había antes de entrar
 - sa_flags: para configurar el comportamiento (si vale 0 se usa la configuración por defecto). Algunos flags:
 - ▶ SA_RESETHAND: después de tratar el signal se restaura el tratamiento por defecto del signal
 - ▶ SA_RESTART: si un proceso bloqueado en una llamada a sistema recibe el signal se reinicia la llamada que lo ha bloqueado

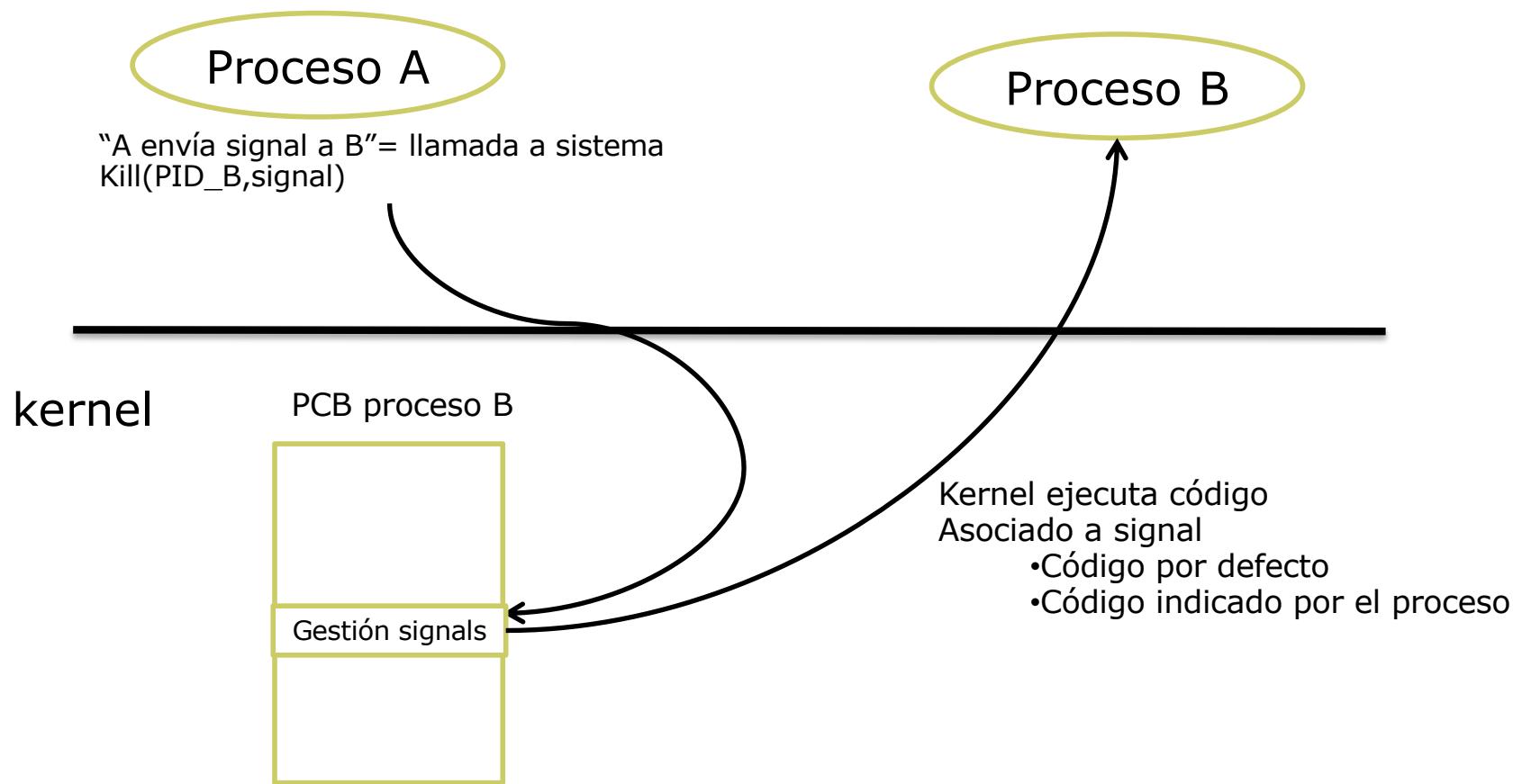


Estructuras de datos del kernel

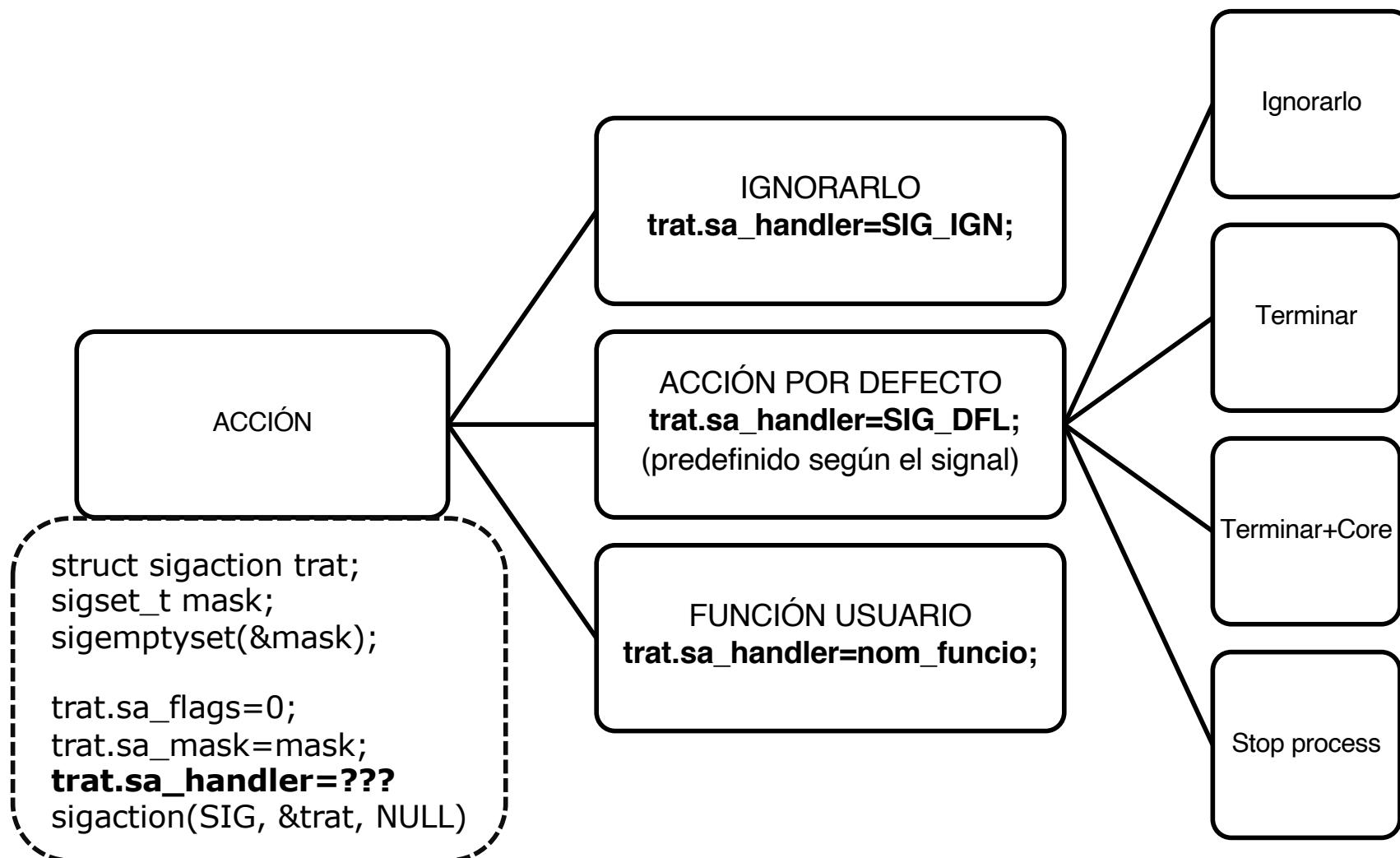
- La gestión de signals es por proceso, la información de gestión está en el PCB
 - Cada proceso tiene una **tabla de programación de signals** (1 entrada por signal),
 - ▶ Se indica que acción realizar cuando se reciba el evento
 - Un bitmap de **eventos pendientes** (1 bit por signal)
 - ▶ No es un contador, actúa como un booleano
 - Un único **temporizador** para la alarma
 - ▶ Si programamos 2 veces la alarma solo queda la última
 - Una **máscara de bits** para indicar qué signals hay que tratar

Signals: Envío y recepción

¿Qué sucede realmente?, el kernel ofrece el servicio de pasar la información.



Acciones posibles al recibir un signal



Donde SIG debe ser el nombre de un signal: SIGUSR1, SIGALRM, SIGUSR2, etc

Manipulación de máscaras de signals

- **sigemptyset:** inicializa una máscara sin signals

```
int sigemptyset(sigset_t *mask)
```



- **sigfillset:** inicializa una máscara con todos los signals

```
int sigfillset(sigset_t *mask)
```



- **sigaddset:** añade el signal a la máscara que se pasa como parámet

```
int sigaddset(sigset_t *mask, int signum)
```



- **sigdelset:** elimina el signal de la máscara que se pasa como parámet

```
int sigdelset(sigset_t *mask, int signum)
```



- **sigismember:** devuelve cierto si el signal está en la máscara

```
int sigismember(sigset_t *mask, int signum)
```



Ejemplo: capturar signals



```
void main()
{
    char buffer[128];
    struct sigaction trat;
    sigset(SIGINT, &trat);
    trat.sa_handler = f_sigint;
    trat.sa_flags = 0;
    trat.sa_mask = mask;

    if (sigemptyset(&mask) == -1)
        perror("Error en sigemptyset");
    if (sigaddset(SIGINT, &trat) == -1)
        perror("Error en sigaddset");

    while(1) {
        sprintf(buffer, "Estoy haciendo cierta tarea\n");
        write(1,buffer,strlen(buffer));
    }
}

void f_sigint(int s)
{
    char buffer[128];
    sprintf(buffer, "SIGINT RECIBIDO!\n");
    exit(0);
}
```

Podéis encontrar el código completo en: [signal_basico.c](#)

Bloquear/desbloquear signals

- El proceso puede controlar en qué momento quiere recibir los signals

```
int sigprocmask(int operacion, sigset_t *mascara, sigset_t  
*vieja_mascara)
```



- Operación puede ser:
 - ▶ SIG_BLOCK: **añadir** los signals que indica *mascara* a la máscara de signals bloqueados del proceso
 - ▶ SIG_UNBLOCK: **quitar** los signals que indica *mascara* a la máscara de signals bloqueados del proceso
 - ▶ SIG_SETMASK: hacer que la máscara de signals bloqueados del proceso pase a ser el parámetro *mascara*

Esperar un evento

■ Esperar (bloqueado) a que llegue un evento

```
int sigsuspend(sigset_t *mascara)
```



- Bloquea al proceso hasta que llega un evento cuyo tratamiento no sea SIG_IGN
- **Mientras** el proceso está bloqueado en el `sigsuspend` *mascara* será los signals que no se recibirán (signals bloqueados),
 - ▶ Así se puede controlar qué signal saca al proceso del bloqueo
- Al salir de `sigsuspend` automáticamente se restaura la mascara que había y se tratarán los signals pendientes que se estén desbloqueando

Sincronización: A envía un signal a B (1)

- El proceso A envía (en algún momento) un signal a B, B está esperando un evento y ejecuta una acción al recibirlo

Proceso A

```
....  
Kill( pid, evento);  
....
```

Proceso B

```
void funcion(int s)  
{  
...  
}  
int main()  
{  
sigaction(evento, &trat,NULL);  
....  
sigemptyset(&mask);  
sigsuspend(&mask);  
....  
}
```

¿Qué pasa si A envía el evento antes de que B llegue al sigsuspend?
¿Qué pasa si B recibe otro evento mientras está en el sigsuspend?

Sincronización: A envía un signal a B (2)

- El proceso A envía (en algún momento) un signal a B, B está esperando un evento y ejecuta una acción al recibirlo

Proceso A

.....

Kill(pid, evento);

....

- sigprocmask bloquea *evento*, así que si llega antes de que B llegue al sigsuspend no se le entrega
- Cuand B está en el sigsuspend él único evento que le puede desbloquear es el que se usa para la sincronización con A

Proceso B

```
void funcion(int s)
```

```
{
```

```
...
```

```
}
```

```
int main()
```

```
{
```

```
sigemptyset(&mask);
```

```
sigaddset(&mask,evento);
```

```
sigprocmask(SIG_SETMASK,&mask,NULL);
```

```
sigaction(evento, &trat,NULL);
```

```
....
```

```
sigfillset(&mask);
```

```
sigdelset(&mask,evento)
```

```
sigsuspend(&mask);
```

```
....
```

```
}
```

Alternativas en la sincronización de procesos

- Alternativas para “esperar” la recepción de un evento
 - 1. **Espera activa:** El proceso consume cpu para comprobar si ha llegado o no el evento. Normalmente comprobando el valor de una variable
 - Ejemplo: `while(!recibido);`
 - 2. **Bloqueo:** El proceso libera la cpu (se bloquea) y será el kernel quien le despierte a la recepción de un evento
 - Ejemplo: `sigsuspend`
- Si el tiempo de espera es corto se recomienda espera activa
 - No compensa la sobrecarga necesaria para ejecutar el bloqueo del proceso y el cambio de contexto
- Para tiempos de espera largos se recomienda bloqueo
 - Se aprovecha la CPU para que el resto de procesos (incluido el que estamos esperando) avancen con su ejecución

Control de tiempo: programar temporizado

- Programar un envío automático (lo envía el kernel) de signal SIGALRM
 - int alarm(num_secs);

```
ret=rem_time;
si (num_secs==0) {
    enviar_SIGALRM=OFF
}else{
    enviar_SIGALRM=ON
    rem_time=num_secs,
}
return ret;
```

Control de tiempo: Uso del temporizador

- El proceso programa un temporizador de 2 segundos y se bloquea hasta que pasa ese tiempo

```
void funcion(int s)
{
...
}
int main()
{
sigemptyset(&mask);
sigaddset(&mask, SIGALRM);
sigprocmask(SIG_SETMASK,&mask, NULL);
sigaction(SIGALRM, &trat, NULL);
....
sigfillset(&mask);
sigdelset(&mask,SIGALRM);
alarm(2);
sigsuspend(&mask);
....
}
```

El proceso estará bloqueado en el `sigsuspend`, cuando pasen 2 segundos recibirá el `SIGALRM`, se ejecutará la función y luego continuará donde estaba

Relación con fork y exec

■ FORK: Proceso nuevo

- El hijo hereda la tabla de acciones asociadas a los signals del proceso padre
- La máscara de signals bloqueados se hereda
- Los eventos son enviados a procesos concretos (PID's), el hijo es un proceso nuevo → La lista de eventos pendientes se borra (tampoco se heredan los temporizadores pendientes)

■ EXECLP: Mismo proceso, cambio de ejecutable

- La tabla de acciones asociadas a signals se pone por defecto ya que el código es diferente
- Los eventos son enviados a procesos concretos (PID's), el proceso no cambia → La lista de eventos pendientes se conserva
- La máscara de signals bloqueados se conserva

Ejemplo 1: gestión de 2 signals (1)



```
void main()
{
    sigemptyset(&mask1);
    sigaddset(&mask1, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask1, NULL);

    trat.sa_flags=0;
    trat.sa_handler = f_alarma;
    sigemptyset(&mask2);
    trat.sa_mask=mask2;
    sigaction(SIGALRM, &trat, NULL);

    sigfillset(&mask3);
    sigdelset(&mask3, SIGALRM);

    for(i = 0; i < 10; i++) {
        alarm(2);
        sigsuspend(&mask3);
        crea_ps();
    }
}
```

```
void f_alarma()
{
}
void crea_ps()
{
    pid = fork();
    if (pid == 0)
        execvp("ps", "ps",
               (char *)NULL);
}
```

Podéis encontrar el código completo en: *cada_segundo.c*

Ejemplo 2: espera activa vs bloqueo (1)



```
void main()
{
    configurar_esperar_alarma()
    trat.sa_flags = 0;
    trat.sa_handler=f_alarma;
    sigemptyset(&mask);
    trat.sa_mask=mask;
    sigaction(SIGALRM,&trat,NULL);
    trat.sa_handler=fin_hijo;
    sigaction(SIGCHLD,&trat,NULL);
    for (i = 0; i < 10; i++) {
        alarm(2);
        esperar_alarma(); // ¿Qué opciones tenemos?
        crea_ps();
    }
}
void f_alarma()
{
    alarma = 1;
}
void fin_hijo()
{
    while(waitpid(-1,NULL,WNOHANG) > 0);
}
```

```
void crea_ps()
{
    pid = fork();
    if (pid == 0)
        execlp("ps", "ps",
                (char *)NULL);
}
```

Ejemplo 2: espera activa vs bloqueo (2)

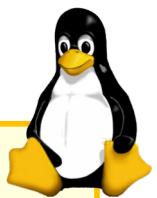
Opción 1: espera activa

```
void configurar_esperar_alarma() {  
    alarma = 0;  
}  
void esperar_alarma() {  
    while (alarma!=1);  
    alarma=0;  
}
```



Opción 2: bloqueo

```
void configurar_esperar_alarma() {  
    sigemptyset(&mask);  
    sigaddset(&mask, SIGALRM);  
    sigprocmask(SIG_BLOCK, &mask, NULL);  
}  
  
void esperar_alarma() {  
    sigfillset(&mask);  
    sigdelset(&mask, SIGALRM);  
    sigsuspend(&mask);  
}
```



-
- Datos
 - Estructuras de gestión
 - Políticas de planificación
 - Mecanismos

GESTIÓN INTERNA DE PROCESOS

Gestión interna

- Para gestionar los procesos necesitamos:
 - **Estructuras de datos**
 - ▶ para representar sus propiedades y recursos → PCB
 - ▶ para representar y gestionar threads → **depende del SO**
 - **Estructuras de gestión**, que organicen los PCB's en función de su estado o de necesidades de organización del sistema
 - ▶ Generalmente son **listas o colas**, pero pueden incluir estructuras más complejas como tablas de hash, árboles, etc.
 - ▶ Hay que tener en cuenta la eficiencia
 - ¿Son rápidas las inserciones/eliminaciones?
 - ¿Son rápidas las búsquedas?
 - ¿Cuál/cuáles serán los índices de búsqueda? ¿PID? ¿Usuario?
 - ▶ Hay que tener en cuenta la escalabilidad
 - ¿Cuántos procesos podemos tener activos en el sistema?
 - ¿Cuánta memoria necesitamos para las estructuras de gestión?
 - **Algoritmo/s de planificación**, que nos indique como gestionar estas estructuras
 - **Mecanismos** que apliquen las decisiones tomadas por el planificador

Datos: Process Control Block (PCB)

- Es la información asociada con cada proceso, depende del sistema, pero normalmente incluye, por cada proceso, aspectos como:
 - El identificador del proceso (PID)
 - Las credenciales: usuario, grupo
 - El estado : RUN, READY,...
 - Espacio para salvar los registros de la CPU
 - Datos para gestionar signals
 - Información sobre la planificación
 - Información de gestión de la memoria
 - Información sobre la gestión de la E/S
 - Información sobre los recursos consumidos (Accounting)

<http://lxr.linux.no/#linux-old+v2.4.31/include/linux/sched.h#L283>

Estructuras para organizar los procesos: Colas/listas de planificación

- El SO organiza los PCB's de los procesos en estructuras de gestión: vectores, listas, colas. Tablas de hash, árboles, en función de sus necesidades
- Los procesos en un mismo estado suelen organizarse en colas o listas que permiten mantener un orden
- Por ejemplo:
 - Cola de procesos – Incluye todos los procesos creados en el sistema
 - Cola de procesos listos para ejecutarse (ready) – Conjunto de procesos que están listos para ejecutarse y están esperando una CPU
 - ▶ En muchos sistemas, esto no es 1 cola sino varias ya que los procesos pueden estar agrupados por clases, por prioridades, etc
 - Colas de dispositivos– Conjunto de procesos que están esperando datos del algún dispositivo de E/S
 - El sistema mueve los procesos de una cola a otra según corresponda
 - ▶ Ej. Cuando termina una operación de E/S , el proceso se mueve de la cola del dispositivo a la cola de ready.

Planificación

- El algoritmo que decide cuando un proceso debe dejar la CPU, quien entra y durante cuanto tiempo, es lo que se conoce como **Política de planificación** (o scheduler)
- La planificación se ejecuta muchas veces (cada 10 ms, por ejemplo) → debe ser muy rápida
- El criterio que decide cuando se evalúa si hay que cambiar el proceso que está en la cpu (o no), que proceso ponemos, etc, se conoce como **Política de planificación**:
- Periódicamente (10 ms.) en la interrupción de reloj, para asegurar que ningún proceso monopoliza la CPU

Planificación

- Hay determinadas situaciones que provocan que se deba ejecutar la planificación del sistema
- Casos en los que el proceso que está RUN no puede continuar la ejecución → Hay que elegir otro → Eventos no preemptivos
 - Ejemplo: El proceso termina, El proceso se bloquea
- Casos en los que el proceso que está RUN podría continuar ejecutándose pero por criterios del sistema se decide pasarlo a estado READY y poner otro en estado RUN → La planificación elige otro pero es forzado → evento preemptivo
 - Estas situaciones dependen de la política, cada política considera algunos si y otros no)
 - Ejemplos: El proceso lleva X ms ejecutándose (RoundRobin), Creamos un proceso nuevo, se desbloquea un proceso,....

Planificador

- Las políticas de planificación son preemptivas (apropiativas) o no preemptivas (no apropiativas)
 - No preemptiva: La política no le quita la cpu al proceso, él la “libera”. Sólo soporta eventos no preemptivos. (eventos tipo 1 y 2)
 - Preemptiva: La política le quita la cpu al proceso. Soporta eventos preemptivos (eventos tipo 3) y no preemptivos.
- Si el SO aplica una política preemptiva el SO es preemptivo

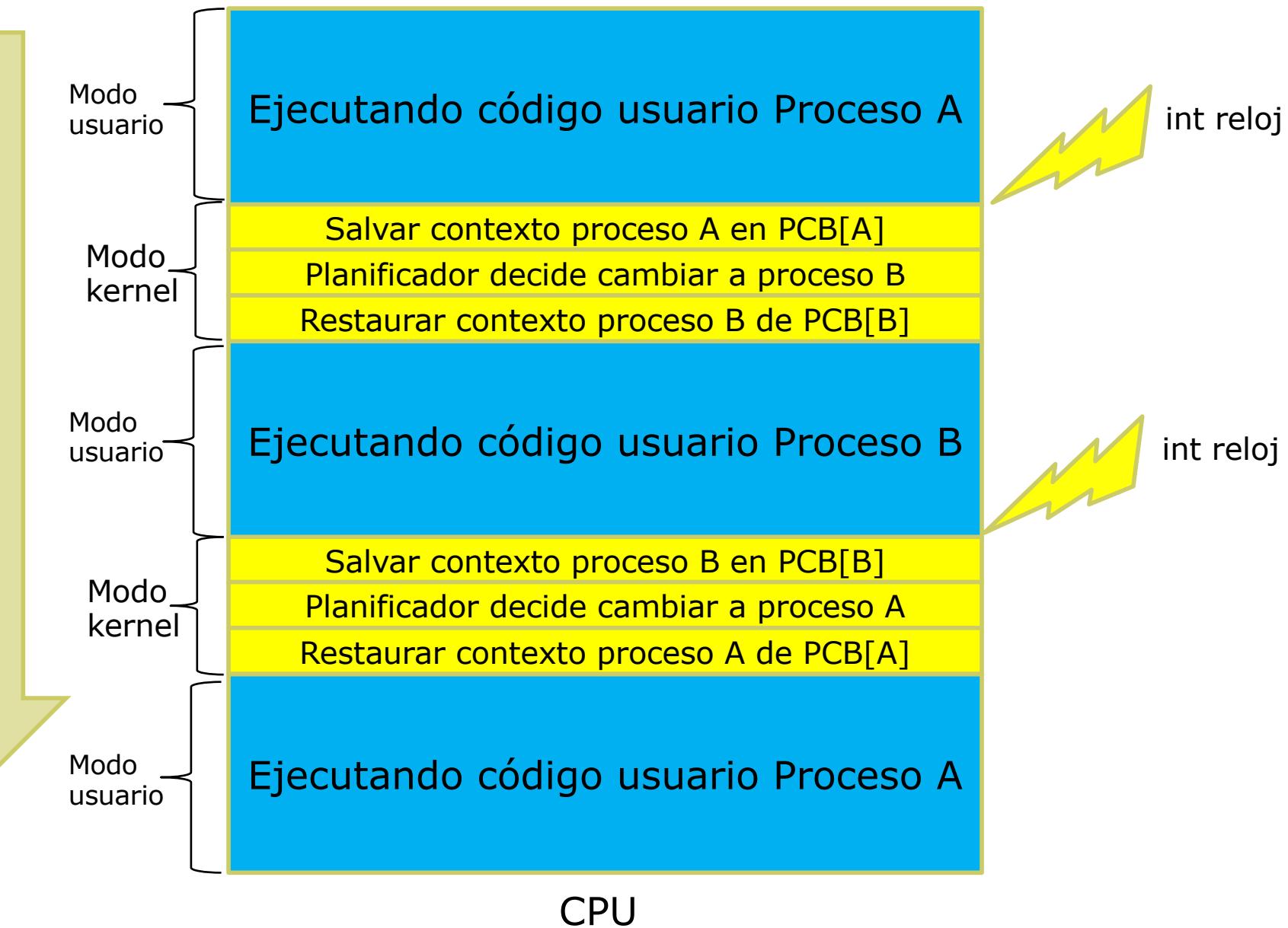
Caracterización de procesos

- Los procesos suelen presentar ráfagas de computación y ráfagas de acceso a dispositivos (E/S) que pueden bloquear al proceso
- En función de estas ráfagas, los procesos se consideran:
 - Procesos de cálculo: Consumen más tiempo haciendo cálculo que E/S
 - Procesos de E/S: Consumen más tiempo haciendo entrada/salida de datos que cálculo

Mecanismos utilizados por el planificador

- Cuando un proceso deja la CPU y se pone otro proceso se ejecuta un cambio de contexto (de un contexto a otro)
- Cambios de contexto(Context Switch)
 - El sistema tiene que salvar el estado del proceso que deja la cpu y restaurar el estado del proceso que pasa a ejecutarse
 - ▶ El contexto del proceso se suele salvar en los datos de kernel que representan el proceso (PCB). Hay espacio para guardar esta información
 - **El cambio de contexto no es tiempo útil de la aplicación, así que ha de ser rápido.** A veces el hardware ofrece soporte para hacerlo más rápido
 - ▶ Por ejemplo para salvar todos los registros o restaurarlos de golpe
 - Diferencias entre cambio de contexto entre threads
 - ▶ Mismo proceso vs distintos procesos

Mecanismo cambio contexto



Objetivos/Métricas de la planificación

- Las políticas de planificación pueden tener objetivos diferentes según el sistema para el cual estén diseñados, el tipo de usuarios que vayan a usarlos, el tipo de aplicación, etc. Sin embargo, estos son los criterios que suelen considerar (pueden haber otros) que determinan el comportamiento de una política
 - Tiempo total de ejecución de un proceso (Turnaround time)- Tiempo total desde que el proceso llega al sistema hasta que termina
 - ▶ Incluye el tiempo que está en todos los estados
 - ▶ Depende del propio proceso y de la cantidad de procesos que haya en la máquina
 - Tiempo de espera de un proceso- Tiempo que el proceso pasa en estado ready

Round Robin (RR)

- El sistema tiene organizados los procesos en función de su estado
- Los procesos están encolados por orden de llegada
- Cada proceso recibe la CPU durante un periodo de tiempo (time quantum), típicamente 10 ó 100 miliseg.
 - El planificador utiliza la interrupción de reloj para asegurarse que ningún proceso monopoliza la CPU

Round Robin (RR)

- Eventos que activan la política Round Robin:
 1. Cuando el proceso se bloquea (no preemptivo)
 2. Cuando termina el proceso (no preemptivo)
 3. Cuando termina el quantum (preemptivo)
- Es una política **apropiativa** o preemptiva
- Cuando se produce uno de estos eventos, el proceso que está run deja la la cpu y se selecciona el siguiente de la cola de ready.
 - Si el evento es 1, el proceso se añade a la cola de bloqueados hasta que termina el acceso al dispositivo
 - Si el evento es el 2, el proceso pasaría a zombie en el caso de linux o simplemente terminaría
 - Si el evento es el 3, el proceso se añade al final de la cola de ready

Round Robin (RR)

■ Rendimiento de la política

- Si hay N procesos en la cola de ready, y el quantum es de Q milisegundos, cada proceso recibe $1/n$ partes del tiempo de CPU en bloques de Q milisegundos como máximo.
 - ▶ Ningún proceso espera más de $(N-1)Q$ milisegundos.
- La política se comporta diferente en función del quantum
 - ▶ q muy grande \Rightarrow se comporta como en orden secuencial. Los procesos recibirían la CPU hasta que se bloquearan
 - ▶ q pequeño $\Rightarrow q$ tiene que ser grande comparado con el coste del cambio de contexto. De otra forma hay demasiado overhead.

Completely Fair Scheduling

- Algoritmo usado en las versiones actuales de Linux
- Métrica objetivo: tiempo de uso de CPU de todos los procesos tiene que ser equivalente
 - Round Robin penaliza a los procesos intensivos en E/S
- Tiempo máximo de uso consecutivo de CPU (*~quantum*) es variable
 - Teóricamente, tiempo consumido de CPU para cada proceso debería ser el resultado de dividir el tiempo que lleva en ejecución entre el número de procesos que compiten por la CPU
 - A cada proceso se le asigna la CPU hasta que se bloquee, acabe o su tiempo de CPU alcance el teórico que debería tener
- Prioridad → distancia al tiempo teórico de CPU (cuanto más lejos esté más prioritario)
- Crea grupos de procesos (criterio configurable por el administrador de la máquina) y permite contabilizar el uso de CPU por grupo
 - Objetivo: impedir que un usuario que ejecuta muchos procesos acapare la máquina

RELACIÓN ENTRE LAS LLAMADAS A SISTEMA DE GESTIÓN DE PROCESOS Y LA GESTIÓN INTERNA DEL S.O. (DATOS,ALGORITMOS, ETC).

¿Qué hace el kernel cuando se ejecuta un...?

■ fork

- Se busca un PCB libre y se reserva
- Se inicializan los datos nuevos (PID, etc)
- Se aplica la política de gestión de memoria (Tema 3)
 - ▶ P.ej.: reservar memoria y copiar contenido del espacio de direcciones del padre al hijo
- Se actualizan las estructuras de gestión de E/S (Tema 4 y 5)
- En el caso de Round Robin: Se añade el proceso a la cola de ready

■ exec

- Se substituye el espacio de direcciones por el código/datos/pila del nuevo ejecutable
- Se inicializan los datos del PCB correspondientes : tabla de signals, contexto, etc
- Se actualizan el contexto actual del proceso: variables de entorno, argv, registros, etc

¿Qué hace el kernel cuando se ejecuta un...?

■ exit

- Se liberan todos los recursos del proceso: memoria, dispositivos “en uso”, etc
- En Linux: se guarda el estado de finalización en el PCB y se elimina de la cola de ready (de forma que no podrá ejecutar más)
- Se aplica la política de planificación

■ waitpid

- Se busca el proceso en la lista de PCB's para conseguir su estado de finalización
- Si el proceso que buscamos estaba zombie, el PCB se libera y se devuelve el estado de finalización a su padre.
- Si no estaba zombie, el proceso padre se elimina pasa de estado run a blocked hasta que el proceso hijo termine.
 - ▶ Se aplicaría la política de planificación

PROTECCIÓN Y SEGURIDAD

Protección y Seguridad

- La protección se considera un problema Interno al sistema y la Seguridad se refiere principalmente a ataques externos

Protección UNIX

- Los usuarios se identifican mediante username y password (userID)
- Los usuarios pertenecen a grupos (groupID)
 - Para ficheros
 - ▶ Protección asociada a: Lectura/Escritura/Ejecución (rwx)
 - Comando ls para consultar, chmod para modificar
 - ▶ Se asocian a los niveles de: Propietario, Grupo, Resto de usuarios
 - A nivel proceso: Los procesos tienen un usuario que determina los derechos
- La excepción es ROOT. Puede acceder a cualquier objeto y puede ejecutar programas de administración de la máquina
- También se ofrece un mecanismo para que un usuario pueda ejecutar un programa con los privilegios de otro usuario (mecanismo de *setuid*)
 - Permite, por ejemplo, que un usuario pueda modificarse su password aun cuando el fichero pertenece a root.

Seguridad

- La seguridad ha de considerarse a cuatro niveles:
- Físico
 - Las máquinas y los terminales de acceso deben encontrarse en un habitaciones/edificios seguros.
- Humano
 - Es importante controlar a quien se concede el acceso a los sistemas y concienciar a los usuarios de no facilitar que otras personas puedan acceder a sus cuentas de usuario
- Sistema Operativo
 - Evitar que un proceso(s) sature el sistema
 - Asegurar que determinados servicios están siempre funcionando
 - Asegurar que determinados puertos de acceso no están operativos
 - Controlar que los procesos no puedan acceder fuera de su propio espacio de direcciones
- Red
 - La mayoría de datos hoy en día se mueven por la red. Este componente de los sistemas es normalmente el más atacado.