

Procesos

Juan José Costa and Alejandro Pajuelo

Escola Tècnica Superior de Telecomunicacions de Barcelona
(ETSETB)

Universitat Politècnica de Catalunya (UPC)

2021-2022Q2

Índice

- Conceptos relacionados con la Gestión de procesos
- Servicios básicos para gestionar procesos (basado en Linux)
- 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.

Definición

Tareas del sistema operativo

Concurrencia y paralelismo

Estados de los procesos

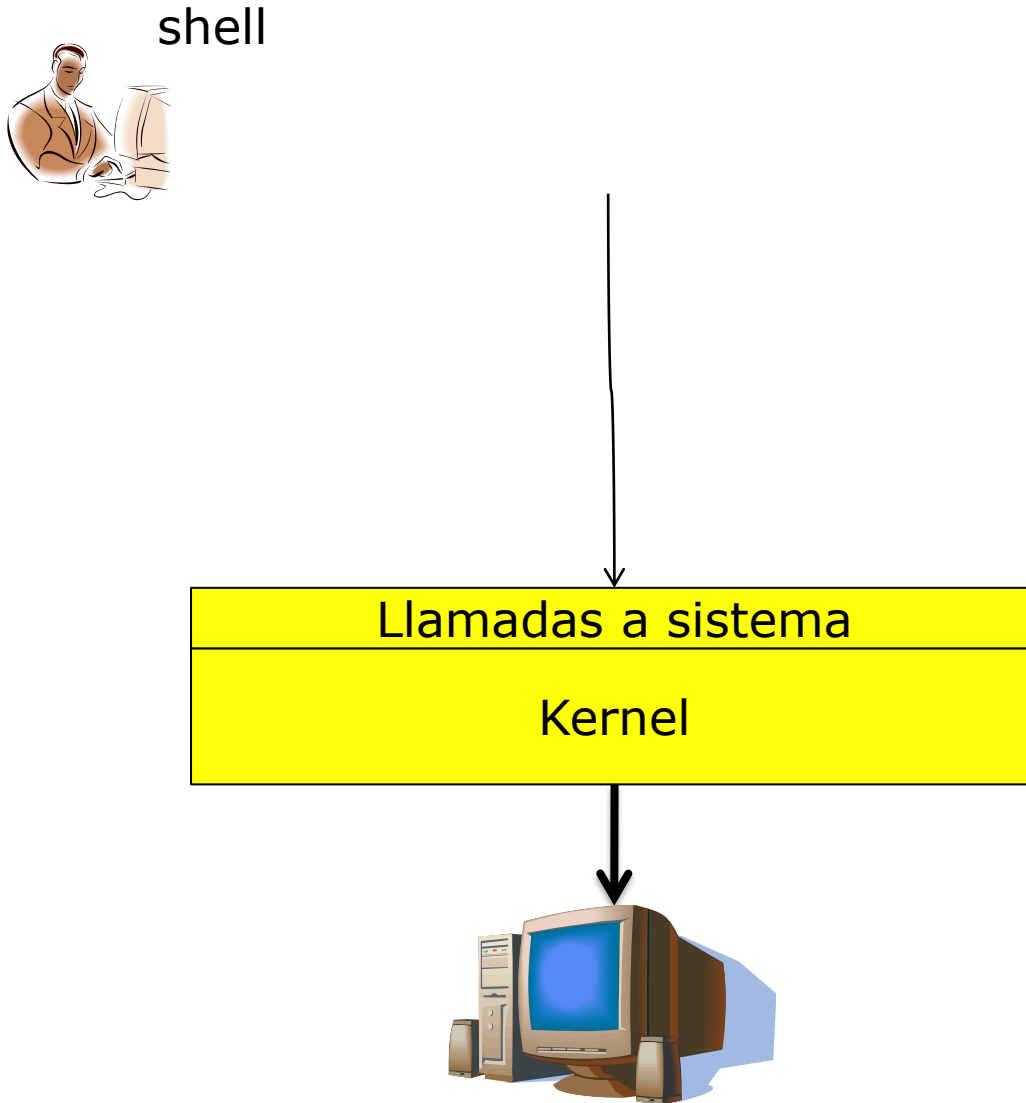
Propiedades de un proceso en Linux

CONCEPTOS

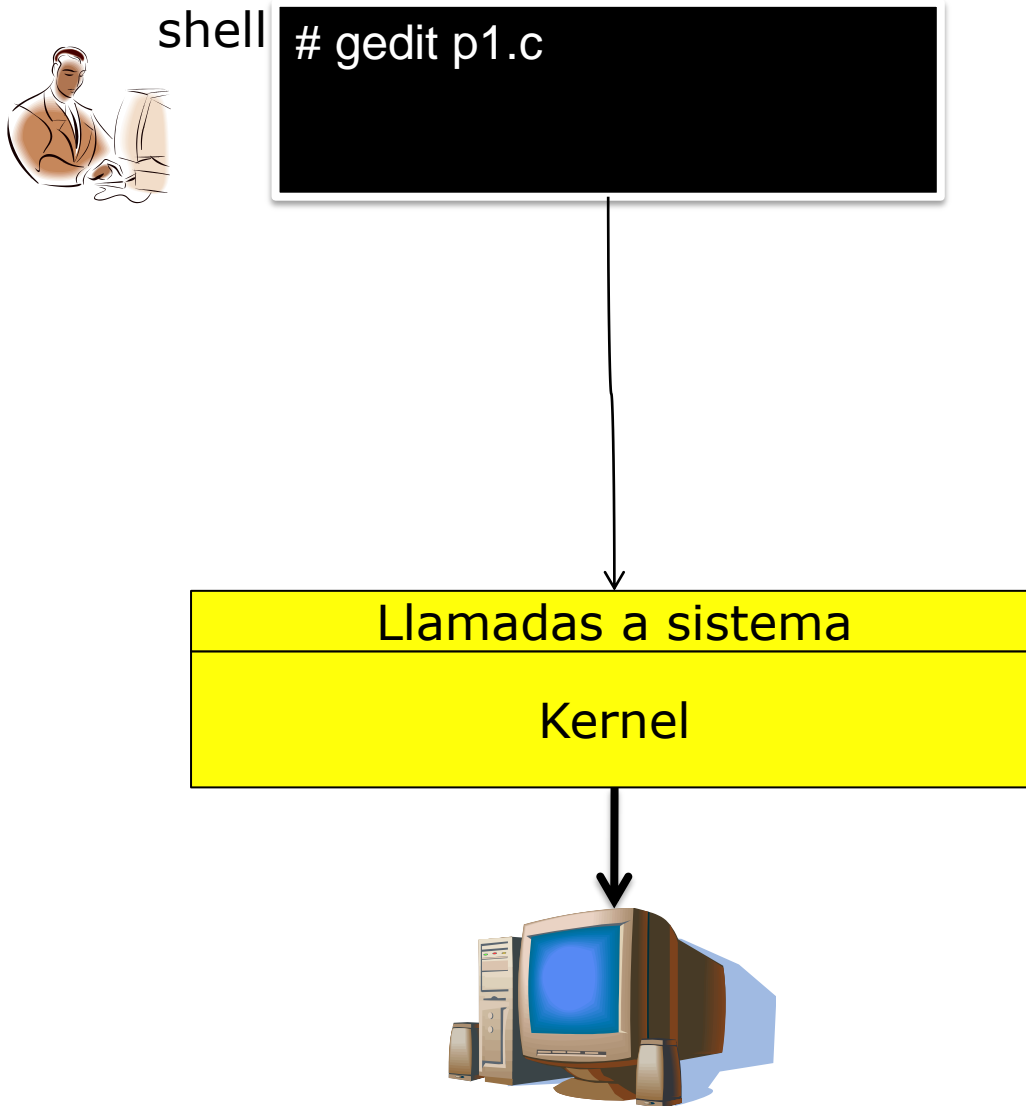
Concepto de proceso

- 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

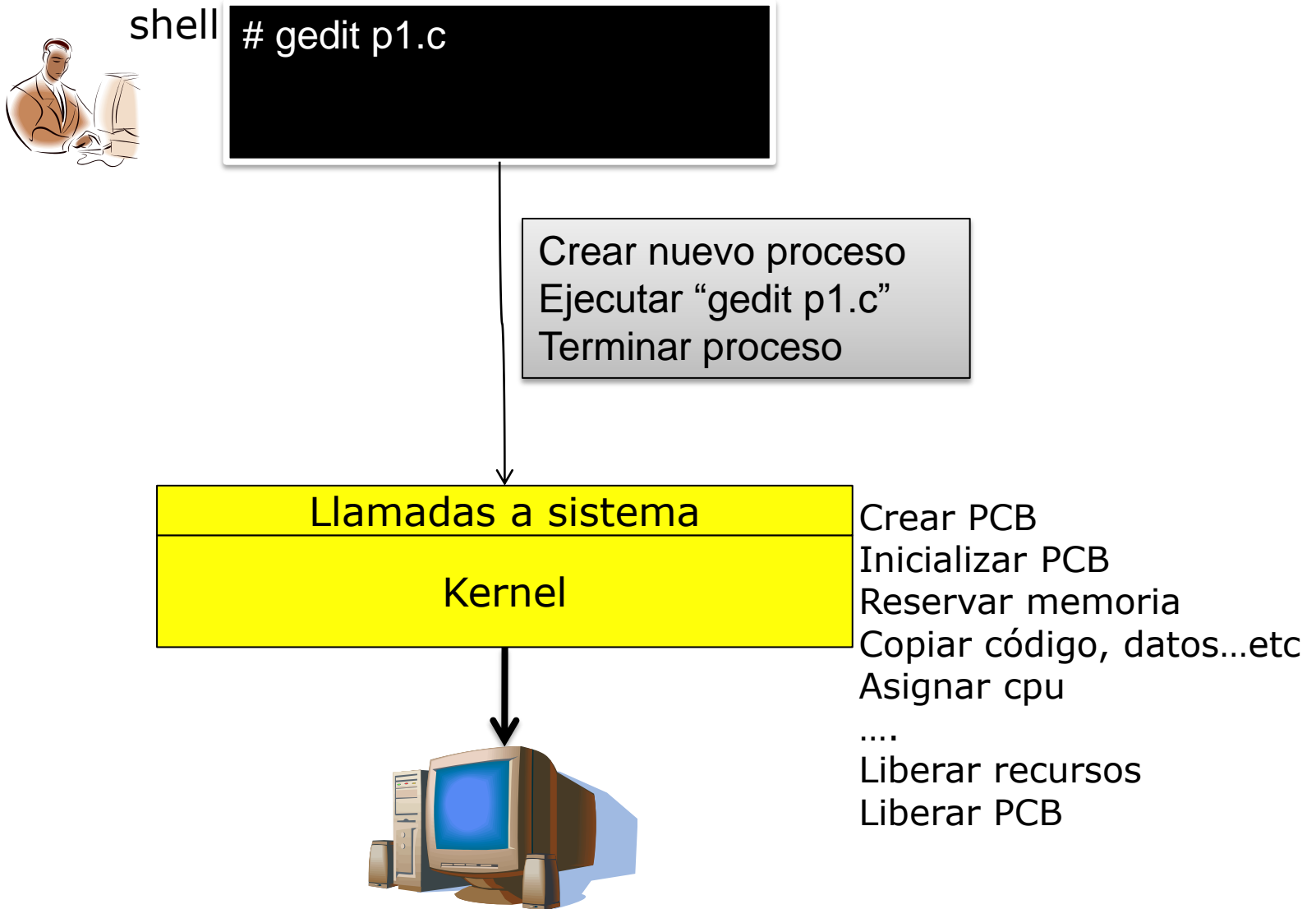
Procesos: ¿Como se hace?



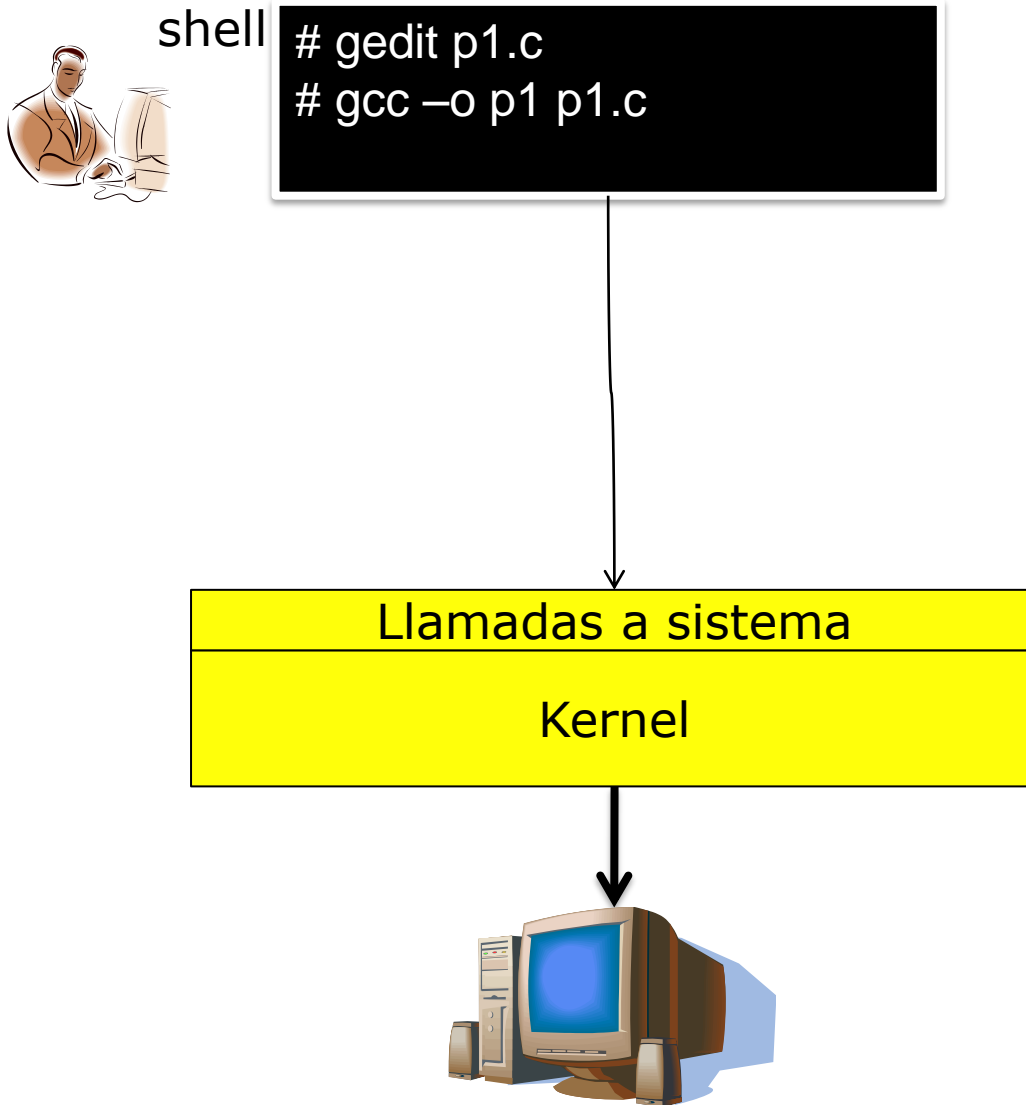
Procesos: ¿Como se hace?



Procesos: ¿Como se hace?



Procesos: ¿Como se hace?



Procesos: ¿Como se hace?



shell

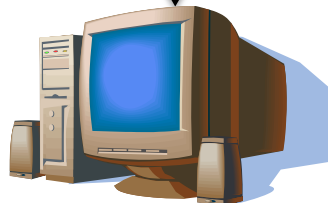
```
# gedit p1.c  
# gcc -o p1 p1.c
```

Crear nuevo proceso
Ejecutar "gcc -o p1 p1.c"
Terminar proceso

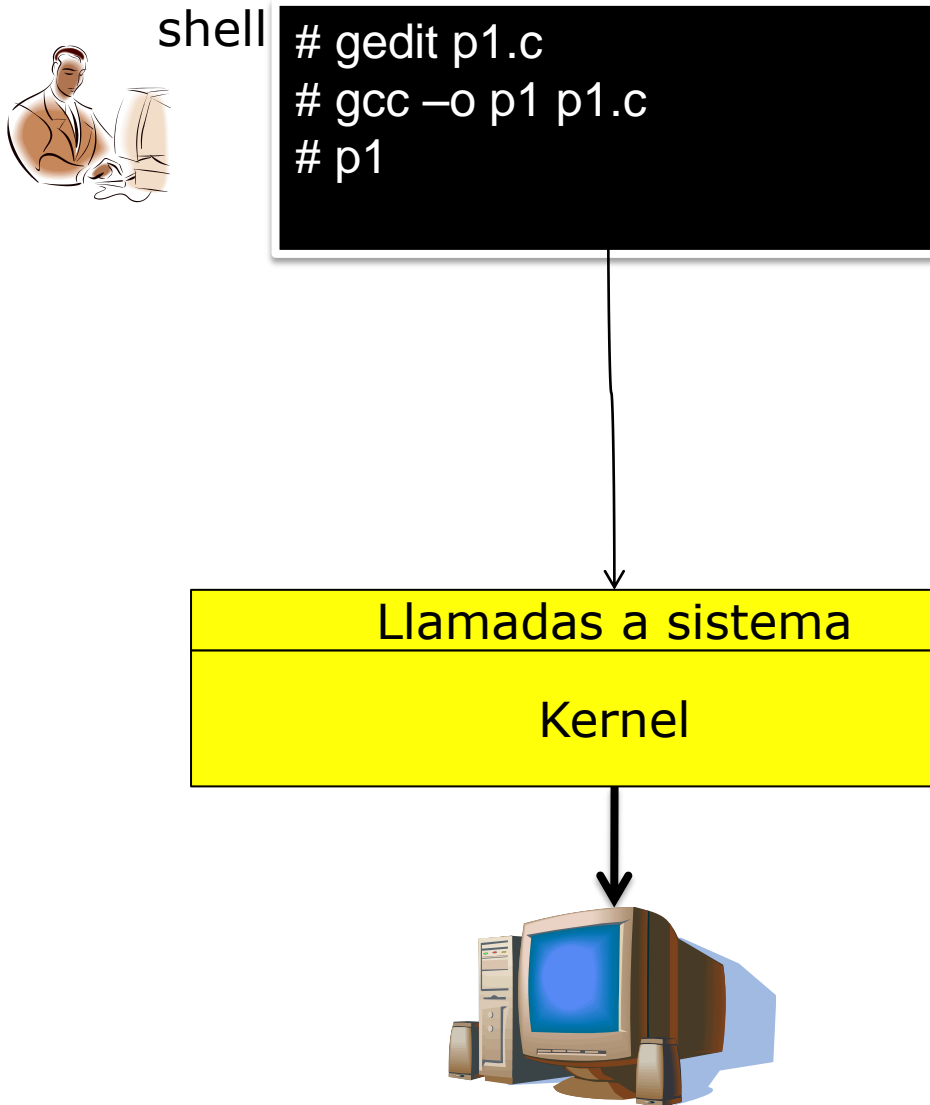
Llamadas a sistema

Kernel

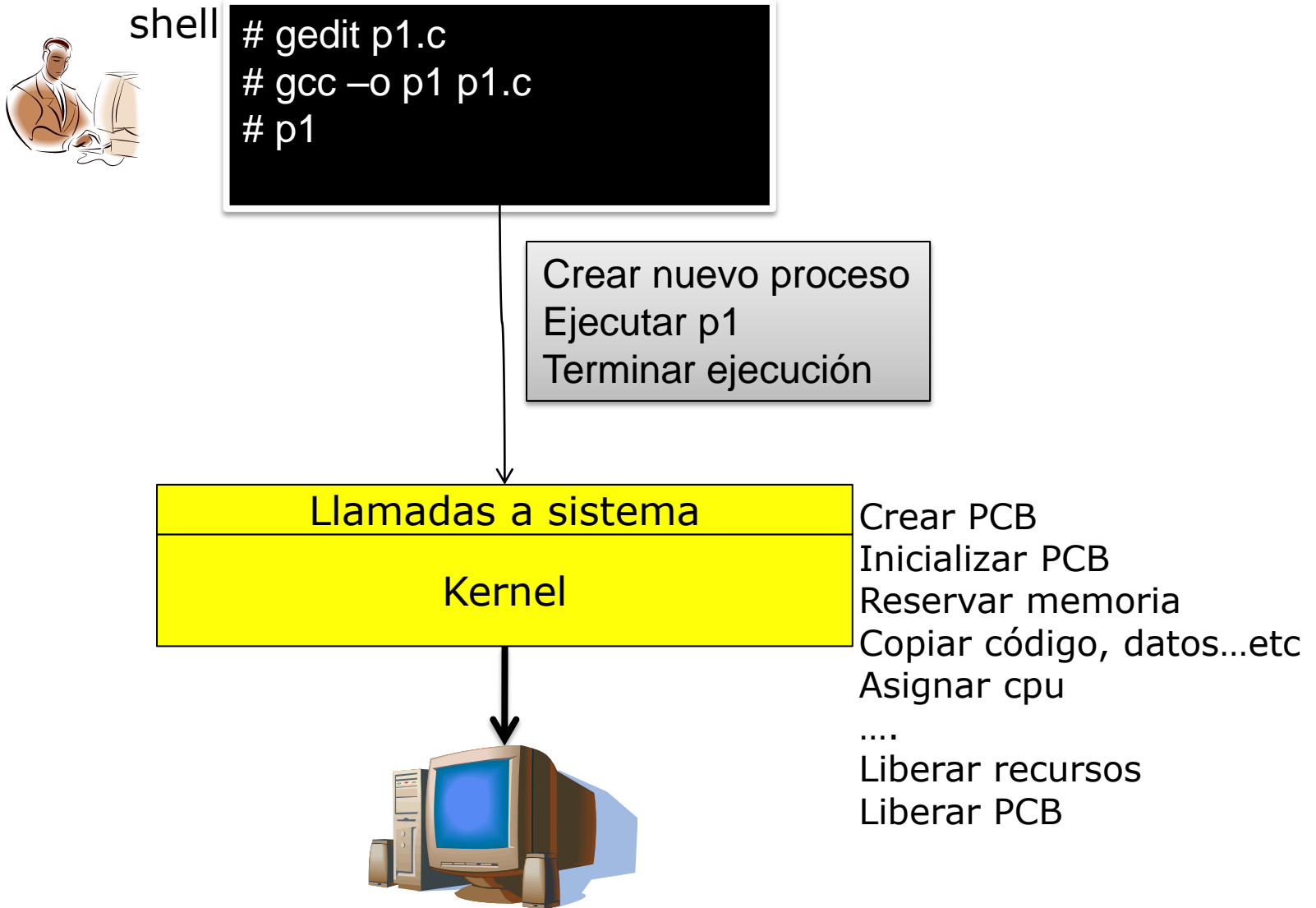
Crear PCB
Inicializar PCB
Reservar memoria
Copiar código, datos...etc
Asignar cpu
....
Liberar recursos
Liberar PCB



Procesos: ¿Como se hace?



Procesos: ¿Como se hace?



Procesos: ¿Como se hace?



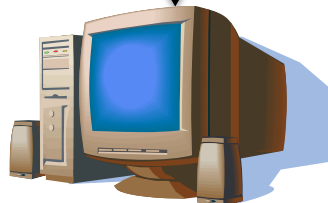
shell

```
# gedit p1.c  
# gcc -o p1 p1.c  
# p1
```



Llamadas a sistema

Kernel



Crear PCB
Inicializar PCB
Reservar memoria
Copiar código, datos...etc
Asignar cpu
....
Liberar recursos
Liberar PCB

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.

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 2 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, ...

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)

<https://elixir.bootlin.com/linux/v5.11/source/include/linux/sched.h#L657>

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/cuales 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

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

Servicios básicos (UNIX)

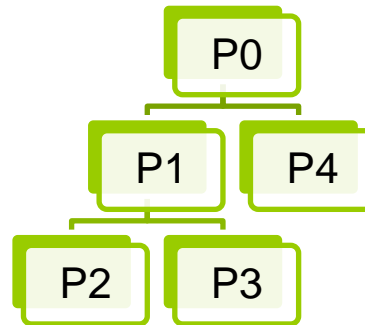


Servicio	Llamada a sistema
Crear proceso	fork
Cambiar ejecutable=Mutar proceso	exec (execlp)
Terminar proceso	exit
Esperar a proceso hijo (bloqueante)	wait/waitpid
Devuelve el PID del proceso	getpid
Devuelve el PID del padre del proceso	getppid

- 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)

Creación de procesos

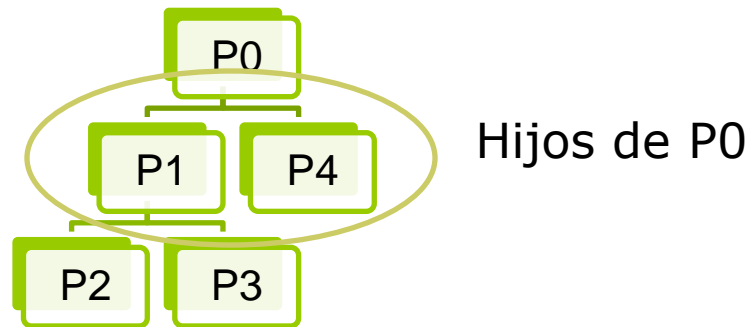
- Cuando un proceso crea otro, se establece una relación jerárquica que se denomina padre-hijo. A su vez, el proceso hijo (y el padre) podrían crear otros procesos generándose un **árbol de procesos**.



- 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

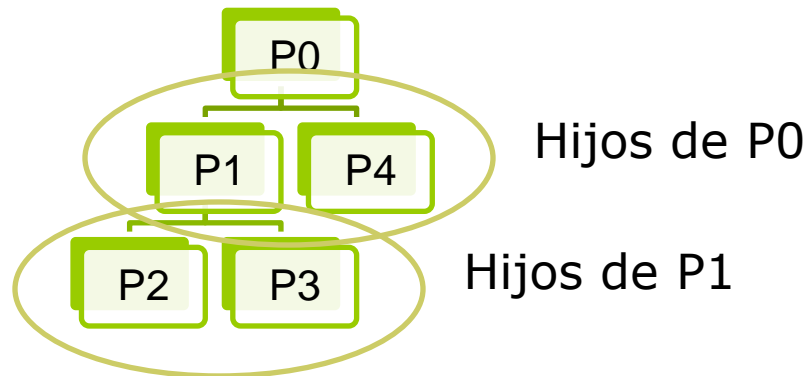
- Cuando un proceso crea otro, se establece una relación jerárquica que se denomina padre-hijo. A su vez, el proceso hijo (y el padre) podrían crear otros procesos generándose un **árbol de procesos**.



- 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

- 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**.

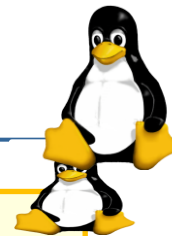


- Los procesos se identifican en el sistema mediante un *process identifier* (PID)
- El SO decide aspectos como por ejemplo:
 - **Recursos**: El proceso hijo, ¿comparte los recursos del padre?
 - **Planificación**: El proceso hijo, ¿se ejecuta antes que el padre?
 - **Espacio de direcciones**. ¿Qué código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

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



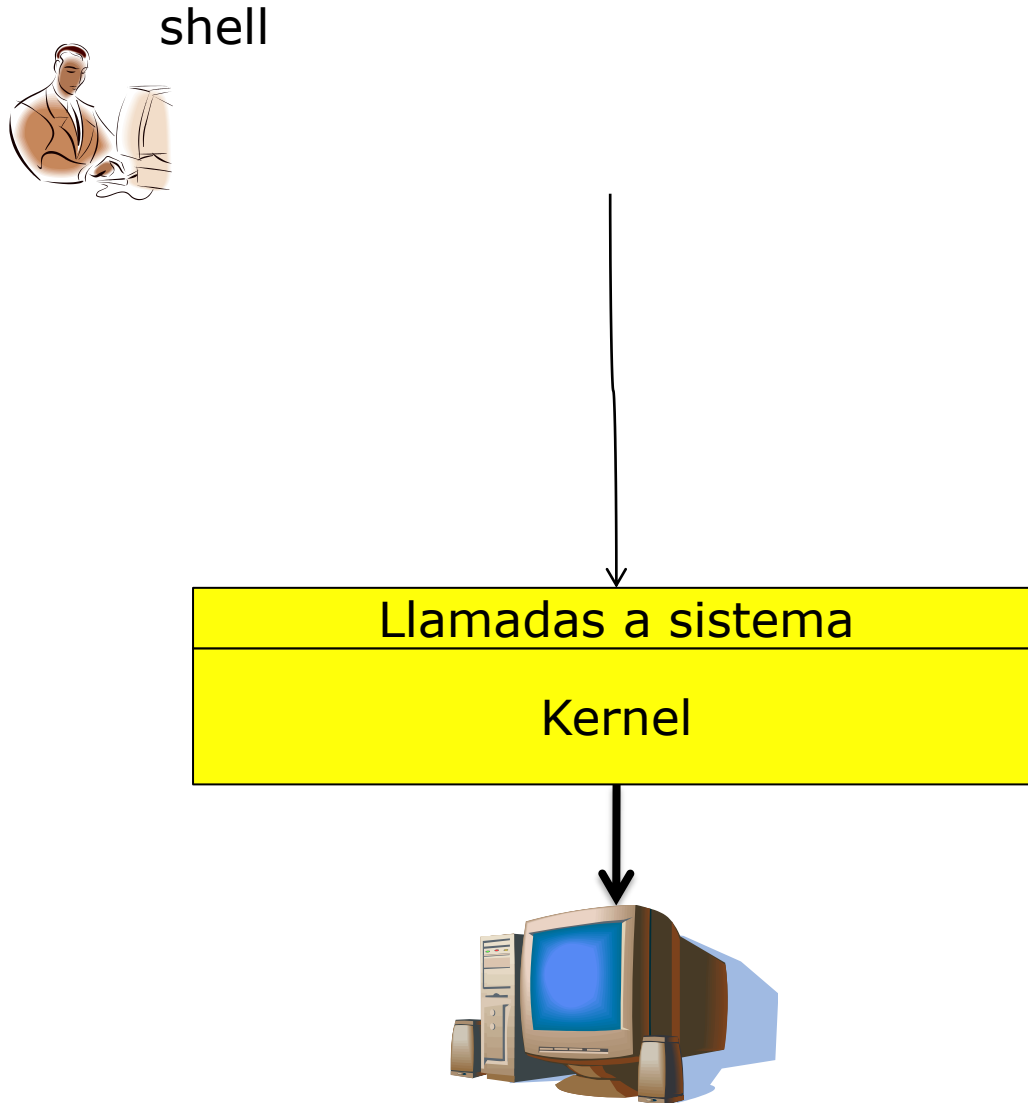
- Un proceso puede acabar su ejecución **voluntaria** (exit) o **involuntariamente (signals)**
- Cuando un proceso quiere finalizar su ejecución (**voluntariamente**), liberar sus recursos y liberar las estructuras de kernel reservadas para él, se ejecuta la llamada a sistema exit.
- Si queremos sincronizar 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);
```

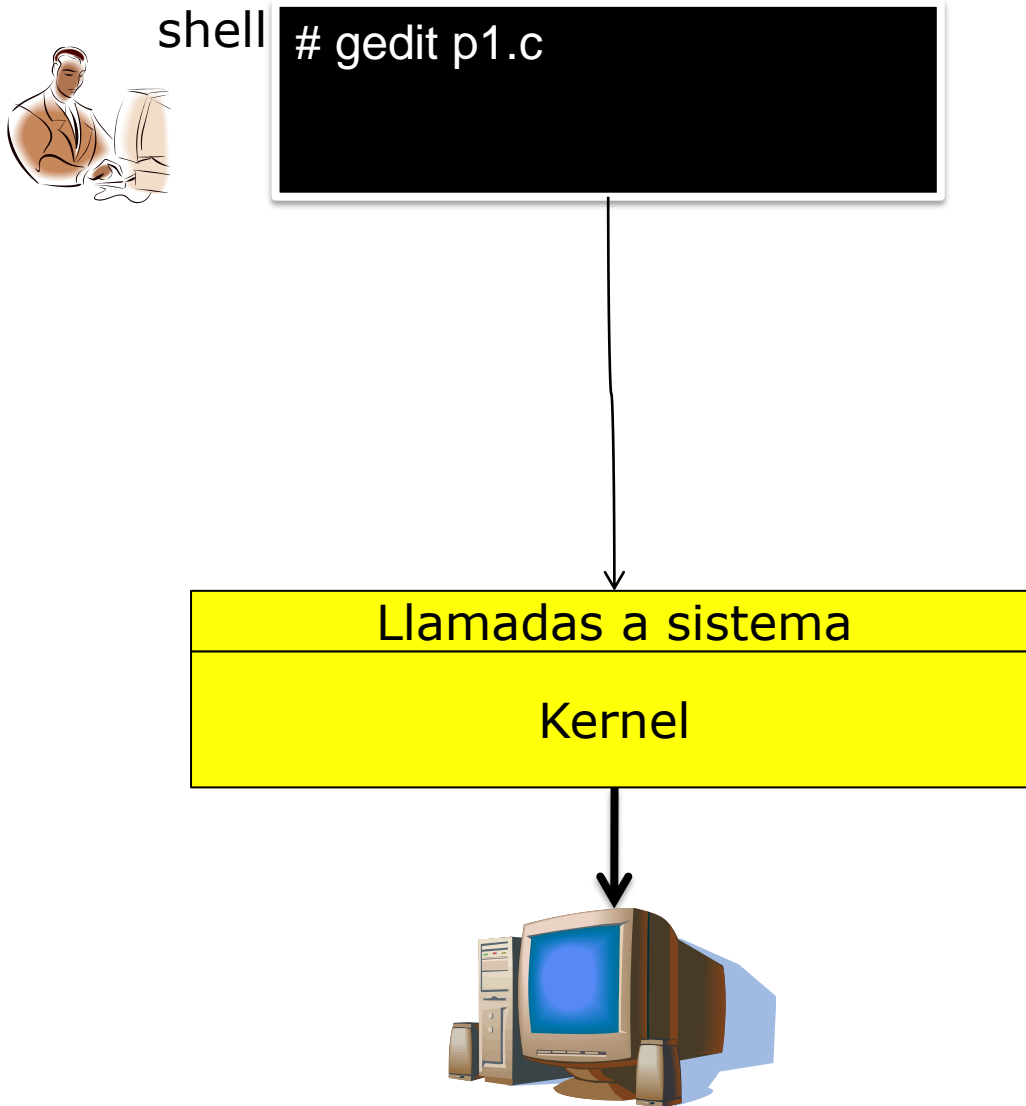


EJEMPLOS GESTIÓN PROCESOS

Procesos: Ahora ya sabemos como se hace



Procesos: Ahora ya sabemos como se hace



Procesos: Ahora ya sabemos como se hace



shell

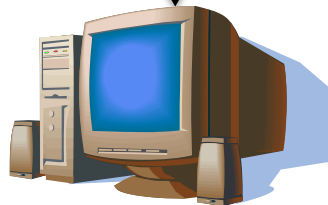
```
# gedit p1.c
```

```
pid=fork();  
if (pid==0) execlp("gedit","gedit","p1.c",(char *)null);  
waitpid(...);
```

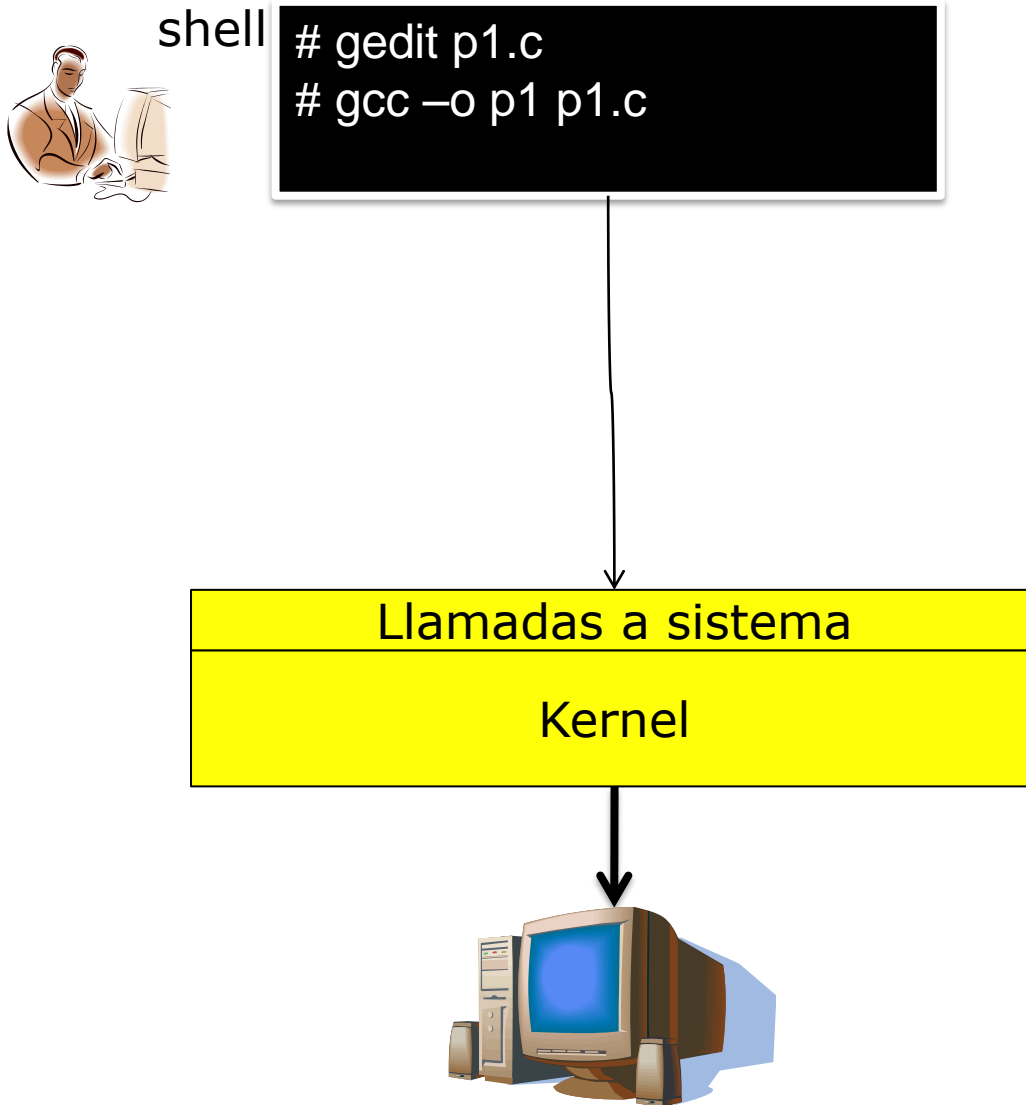
Llamadas a sistema

Kernel

Crear PCB
Inicializar PCB
Reservar memoria
Copiar código, datos...etc
Asignar cpu
....
Liberar recursos
Liberar PCB



Procesos: Ahora ya sabemos como se hace



Procesos: Ahora ya sabemos como se hace



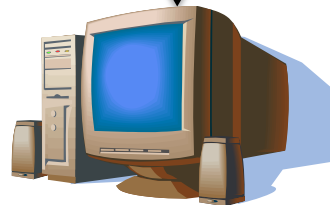
shell

```
# gedit p1.c  
# gcc -o p1 p1.c
```

```
pid=fork();  
if (pid==0) execlp("gcc","gcc","-o","p1","p1.c",",(char *)null);  
waitpid(...);
```

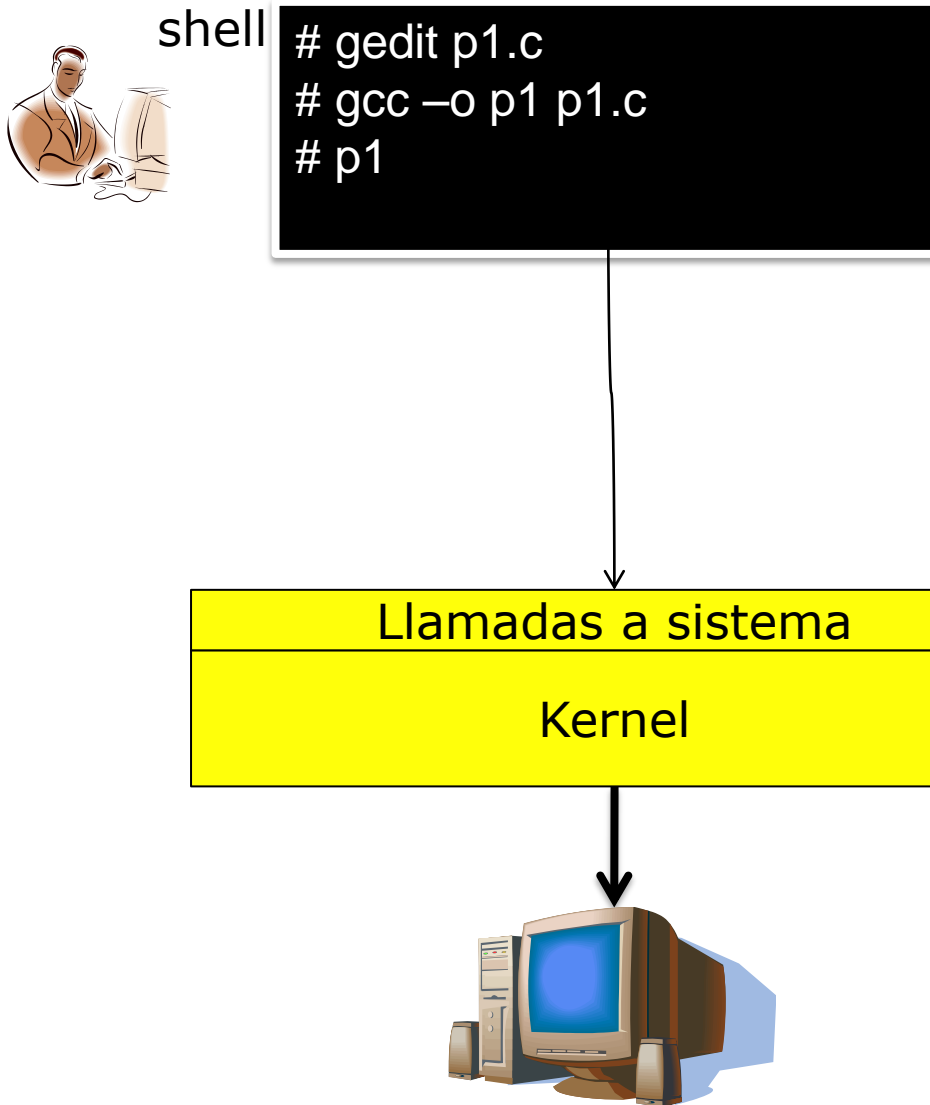
Llamadas a sistema

Kernel



Crear PCB
Inicializar PCB
Reservar memoria
Copiar código, datos...etc
Asignar cpu
....
Liberar recursos
Liberar PCB

Procesos: Ahora ya sabemos como se hace



Procesos: Ahora ya sabemos como se hace



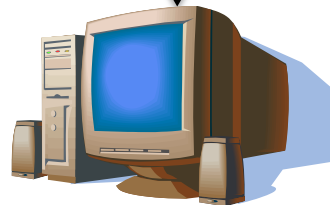
shell

```
# gedit p1.c  
# gcc -o p1 p1.c  
# p1
```

```
pid=fork();  
if (pid==0) execlp("p1","p1", (char *)null);  
waitpid(...);
```

Llamadas a sistema

Kernel



Crear PCB
Inicializar PCB
Reservar memoria
Copiar código, datos...etc
Asignar cpu
....
Liberar recursos
Liberar PCB

Procesos: Ahora ya sabemos como se hace

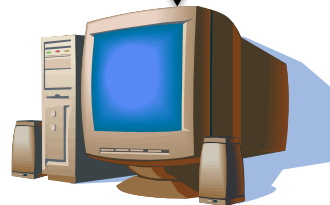


shell

```
# gedit p1.c  
# gcc -o p1 p1.c  
# p1
```

Llamadas a sistema

Kernel



Crear PCB
Inicializar PCB
Reservar memoria
Copiar código, datos...etc
Asignar cpu
....
Liberar recursos
Liberar PCB

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
```

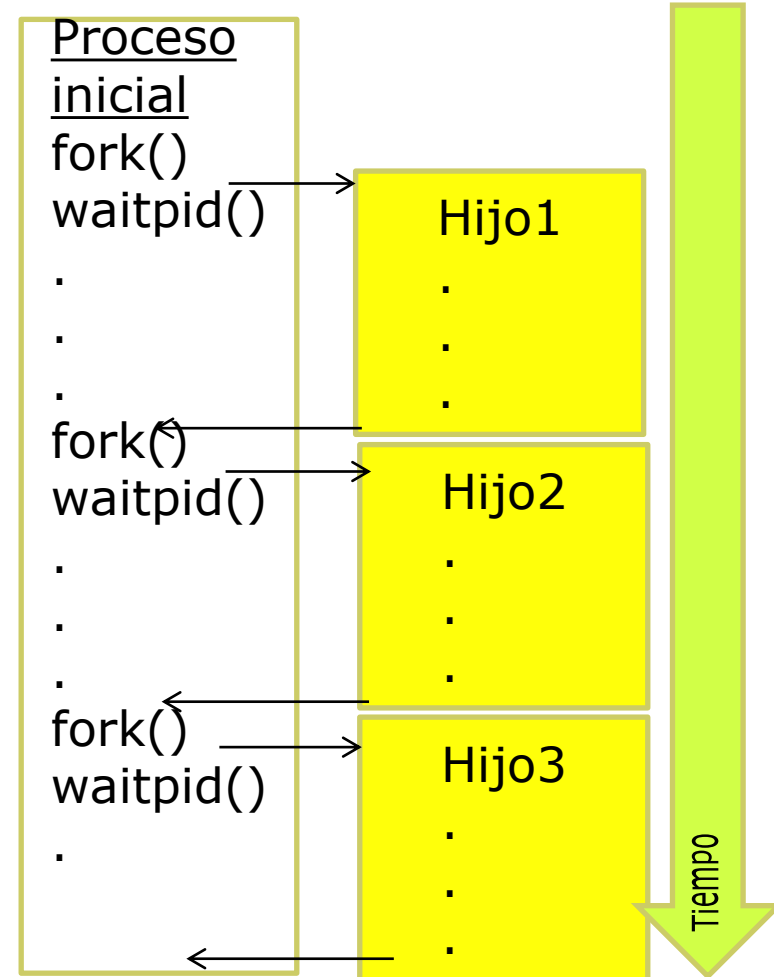
Caso 2: queremos que hagan lo mismo

```
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.

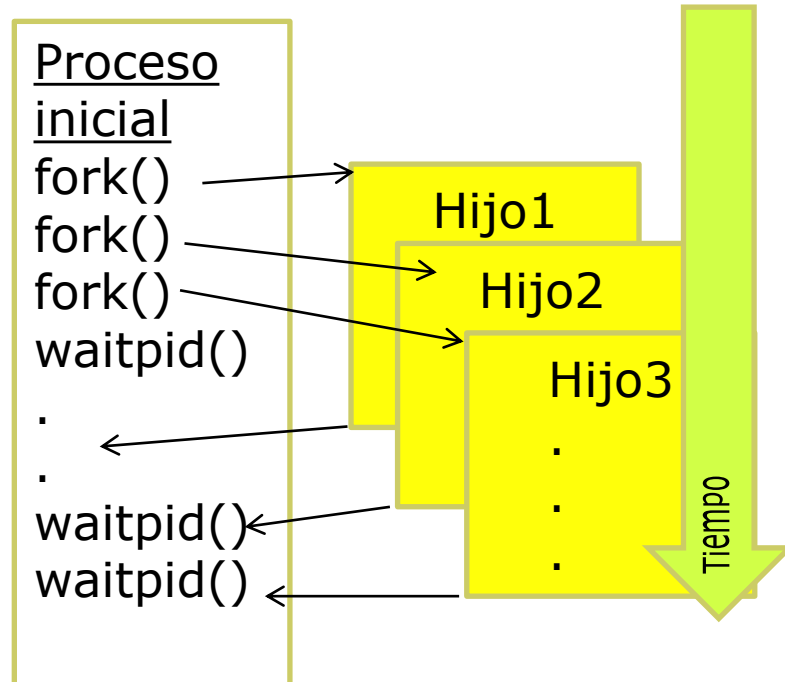
```
1. #define num_procs 2
2. int i,ret;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // estas líneas solo las ejecuta el
7.         // hijo
8.         codigohijo();
9.         exit(0); //
10.    }
11.    waitpid(-1,NULL,0);
12.}
```



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);
}
```

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

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);
}

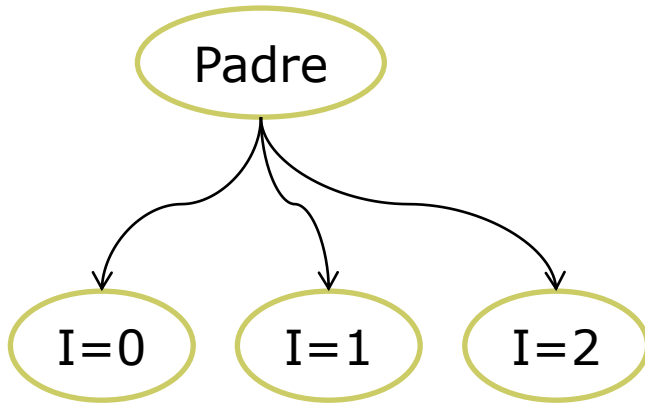
void hacerTarea()
{   char buffer[128];
    sprintf("Mi PID es %d y el de mi padre %d\n", getpid(), getppid());
    write(1,buffer,strlen(buffer));
    exit(0);
}
```

← **Ahora, probad a quitar esta instrucción, es muy diferente!!!**

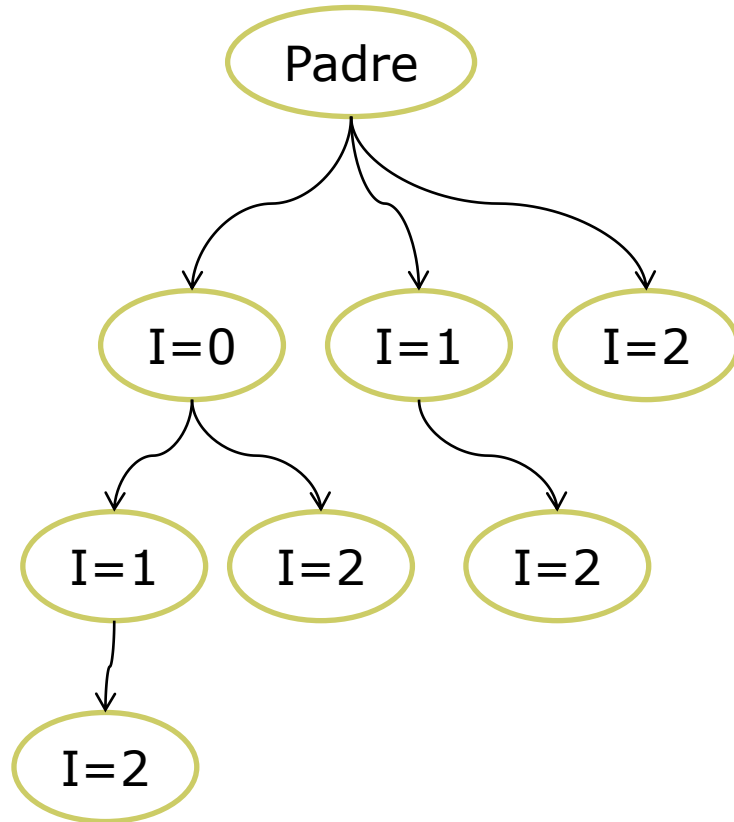
Podéis encontrar el código completo en: Nprocesos.c y NprocesosExit.c

Árbol de procesos (examen)

Con exit

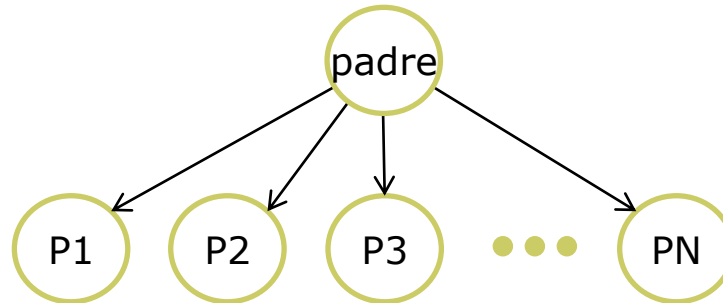


Sin exit

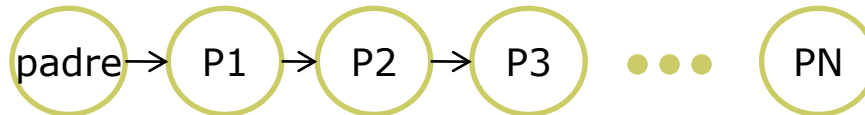


Otros ejemplos con fork

- 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:



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, ...);
```



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



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

Ejemplo fork+exec

```
fork();  
execlp("/bin/progB", "progB", (char *) 0);  
while(...)
```

progA

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

progB

P1



```
fork();  
execlp("/bin/progB", .....  
while(...)
```


Ejemplo fork+exec

```
fork();  
execlp("/bin/progB", "progB", (char *) 0);  
while(...)
```

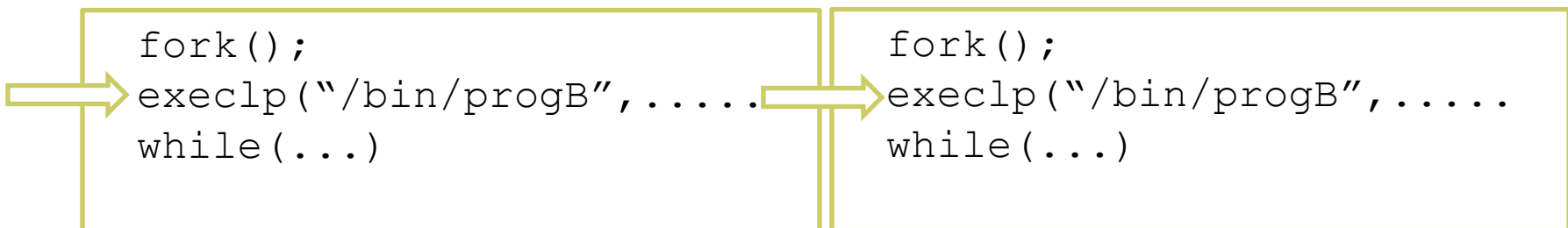
progA

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

progB

P1

P2



```
fork();  
execlp("/bin/progB", .....);  
while(...)
```

```
fork();  
execlp("/bin/progB", .....);  
while(...)
```

Ejemplo fork+exec


```
fork();  
execlp("/bin/progB", "progB", (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) execlp("/bin/progB","progB", (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) execlp(.....);  
while(...)
```

Ejemplo fork+exec

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


progA

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

progB

P1

P2



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

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

Ejemplo fork+exec

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

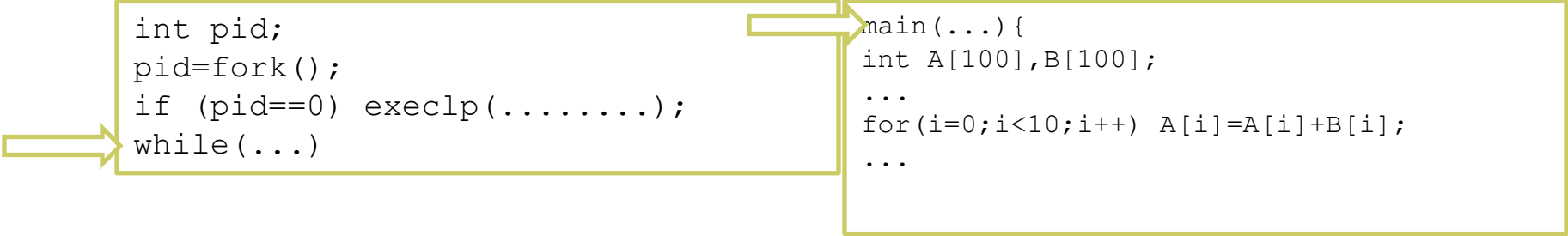
progA

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

progB

P1

P2



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

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

Terminación de procesos. exit

```
void main()  
{...  
ret=fork(); (1)  
if (ret==0) execlp("a","a",NULL);  
...  
waitpid(-1,&exit_code,0); (3)  
}
```

kernel

Terminación de procesos. exit

```
void main()  
{...  
ret=fork(); (1)  
if (ret==0) execlp("a","a",NULL);  
...  
waitpid(-1,&exit_code,0); (3)  
}
```

A

```
void main()  
{...  
exit(4); (2)  
}
```

kernel

Terminación de procesos. exit

```
void main()
{...
ret=fork(); (1)
if (ret==0) execlp("a","a",NULL);
...
waitpid(-1,&exit_code,0); (3)
}
```

A

```
void main()
{...
exit(4); (2)
}
```

kernel

PCB (del proceso "A")

```
pid=...
exit_code=
```

...

...

Terminación de procesos. exit

```
void main()  
{...  
ret=fork(); (1)  
if (ret==0) execlp("a","a",NULL);  
...  
waitpid(-1,&exit_code,0); (3)  
}
```

A

```
void main()  
{...  
exit(4); (2)  
}
```

kernel

PCB (del proceso "A")

pid=...

exit_code=

...

...

Se guarda en el PCB

Terminación de procesos. exit

```
void main()  
{...  
ret=fork(); (1)  
if (ret==0) execlp("a","a",NULL);  
...  
waitpid(-1,&exit_code,0); (3)  
}
```

A

```
void main()  
{...  
exit(4); (2)  
}
```

Se consulta del PCB

kernel

PCB (del proceso "A")

pid=...

exit_code=

...

...

Se guarda en el PCB

Terminación de procesos. exit

```
void main()  
{...  
ret=fork(); (1)  
if (ret==0) execlp("a","a",NULL);  
...  
waitpid(-1,&exit_code,0); (3)  
}
```

A

```
void main()  
{...  
exit(4); (2)  
}
```

Se consulta del PCB

kernel

PCB (del proceso "A")

pid=...

exit_code=

...

...

Se guarda en el PCB

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
    // arg3 == 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)
        execlp(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];

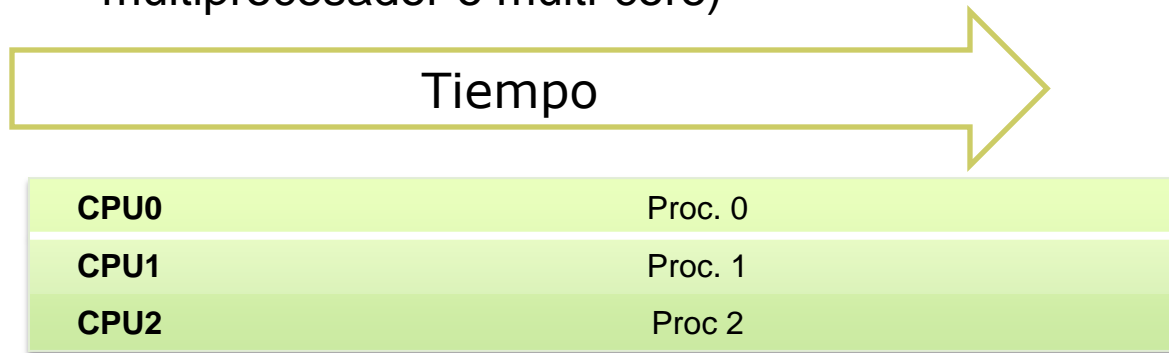
    if (WIFEXITED(exit_code)) {
        statcode = WEXITSTATUS(exit_code);
        sprintf(buffer,"El proceso %d termina con exit code %d\n", pid,
statcode);
        write(1,buffer,strlen(buffer));
    }
    else {
        signcode = WTERMSIG(exit_code);
        sprintf(buffer,"El proceso %d termina por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

Utilización eficiente de la CPU

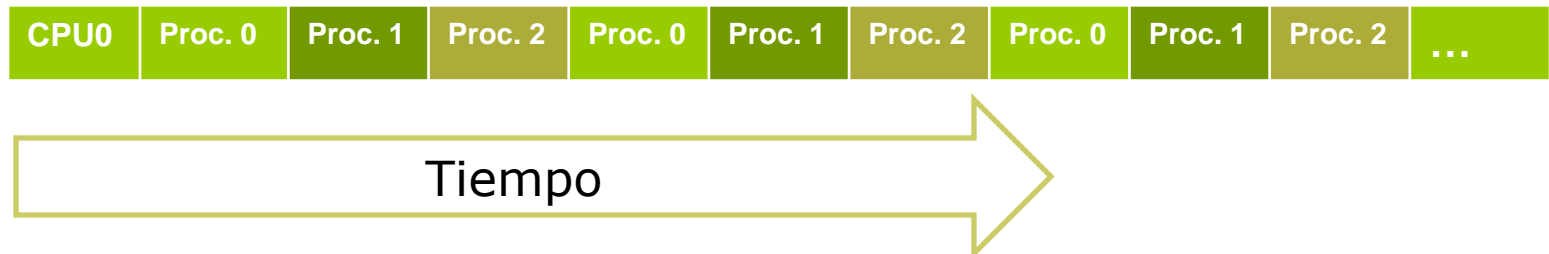
- En un sistema de propósito general, lo habitual es tener varios procesos a la vez, de forma que se aproveche al máximo los recursos de la máquina
- ¿Por qué nos puede interesar ejecutar múltiples procesos simultáneamente?
 - Si tenemos varios procesadores podemos ejecutar más procesos a la vez, o uno 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

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.

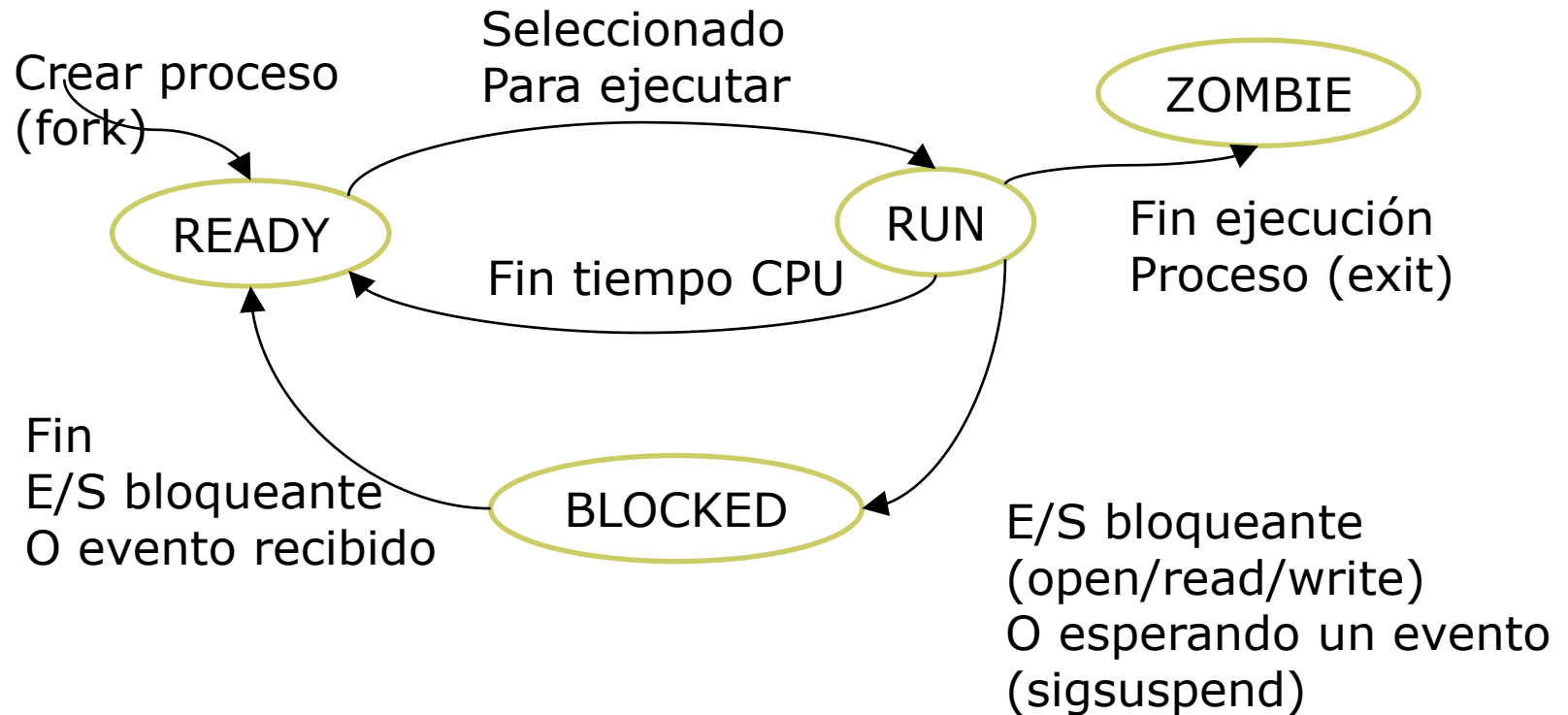
Estados de un proceso

- 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 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

Ejemplo diagrama de estados



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

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 **Planificador** (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.
 - Preemptiva: La política le quita la cpu al proceso. Soporta eventos preemptivos y no preemptivos.
- Si el SO aplica una política preemptiva el SO es preemptivo

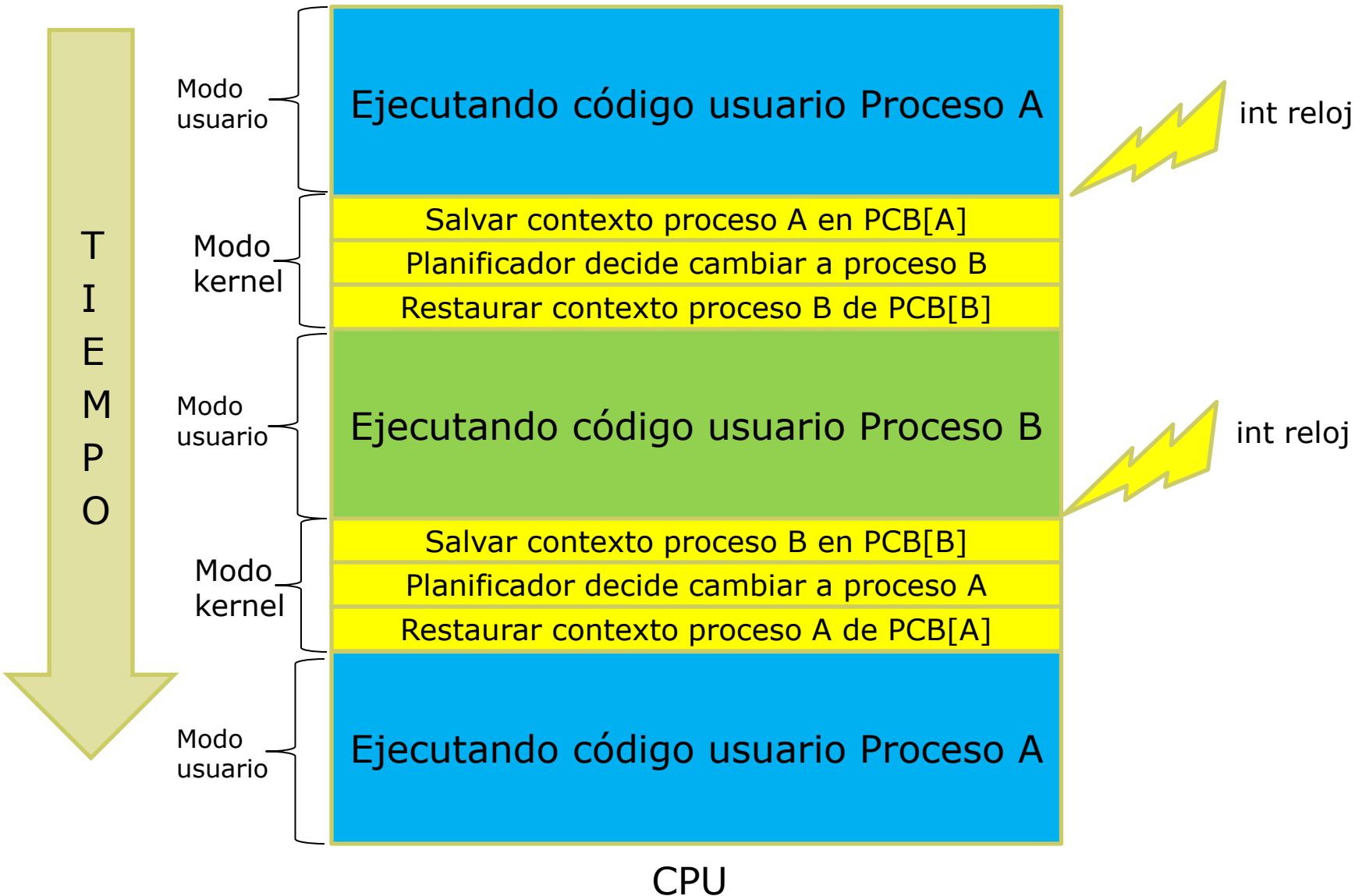
Caracterización de procesos

- 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



Prioridades

- Normalmente, todos los procesos tienen las mismas oportunidades para ejecutarse.
- Puede haber procesos más críticos que otros
- Añadir una característica al proceso que indique si es muy prioritario o no
- El planificador, por orden de prioridad, elegirá el proceso que se tiene que ejecutar a continuación:
 - Los más prioritarios serán los que se ejecuten antes
- La prioridad puede cambiarse a lo largo del tiempo
 - Para conseguir que los procesos menos prioritarios también se ejecuten
 - Envejecimiento / aging

Resumen

- Concepto de proceso
- Servicios relacionados
 - Creación → fork
 - Identificación → getpid
 - Destrucción → exit, waitpid
 - Mutación → exec
- Concurrencia/Paralelismo
- Ciclo de vida de un proceso
- Planificación
 - Planificador
 - Políticas planificación
 - Cambio de contexto
 - Prioridades

References

- [Arpaci18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau . 2018. “Operating Systems: Three Easy Pieces”. Available online <http://ostep.org/>
 - Chapters 3-6