

# T2-Procesos



# Índice

---

- Conceptos relacionados con la Gestión de procesos
- Servicios básicos para gestionar procesos (basado en Linux)
- Comunicación entre procesos
- 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.
- Threads/Flujos
- 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 de 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 inicializados, los datos no inicializados 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 hacer un uso eficiente del sistema, el SO permite la existencia de más de proceso a la vez, compartiendo la máquina. Para gestionar la información de cada uno el SO representa internamente a los procesos con una estructura de datos llamada PCB (Process Control Block)
- Cada vez que ponemos un programa a ejecutar (aunque sea el mismo), se crea un nuevo proceso
  - Pueden haber limitaciones en el sistema

# Tareas del sistema operativo sobre procesos

---

- Crear procesos
  - **Asignar** recursos (¿cómo? ¿cuáles?), inicializar gestión (PCB)
  - Garantizar **protección** entre procesos
  - **Carga** del programa en memoria
- Eliminar procesos
  - **Liberar** recursos
- **Suspender/Resumir** procesos
- Proporcionar mecanismos de **sincronización**
- Proporcionar mecanismos de **comunicación**
  - Memoria compartida
  - Dispositivos especiales
  - Gestión de **signals**
- **Planificación** de procesos
  - Reparto del uso de la CPU
  - Qué proceso ocupa la CPU y durante cuánto tiempo
  - **Eficiencia** del sistema

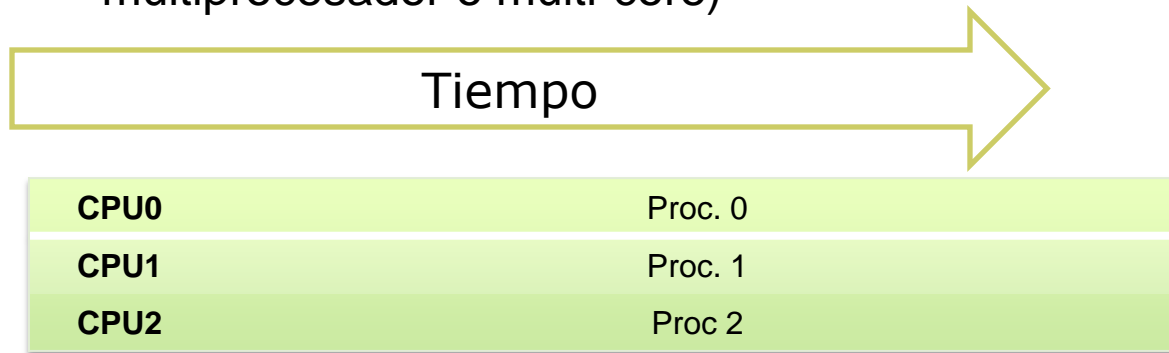
# Concurrencia y Paralelismo

---

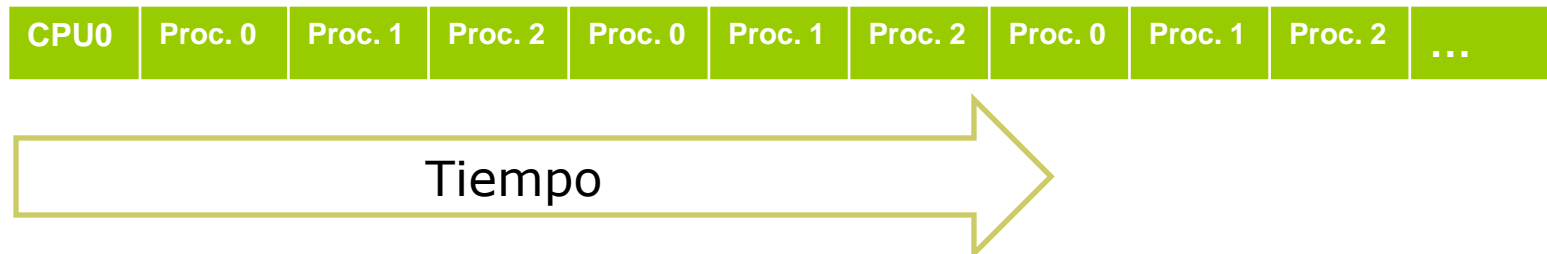
- En un sistema de propósito general, lo habitual es tener varios procesos a la vez, de forma que se aproveche al máximo los recursos de la máquina
- ¿Por qué nos puede interesar ejecutar múltiples procesos simultáneamente?
  - Si tenemos varios procesadores podemos ejecutar más procesos a la vez, o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

# Concurrencia y Paralelismo

- 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 y Paralelismo

---

- 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).



# Estados de un proceso

---

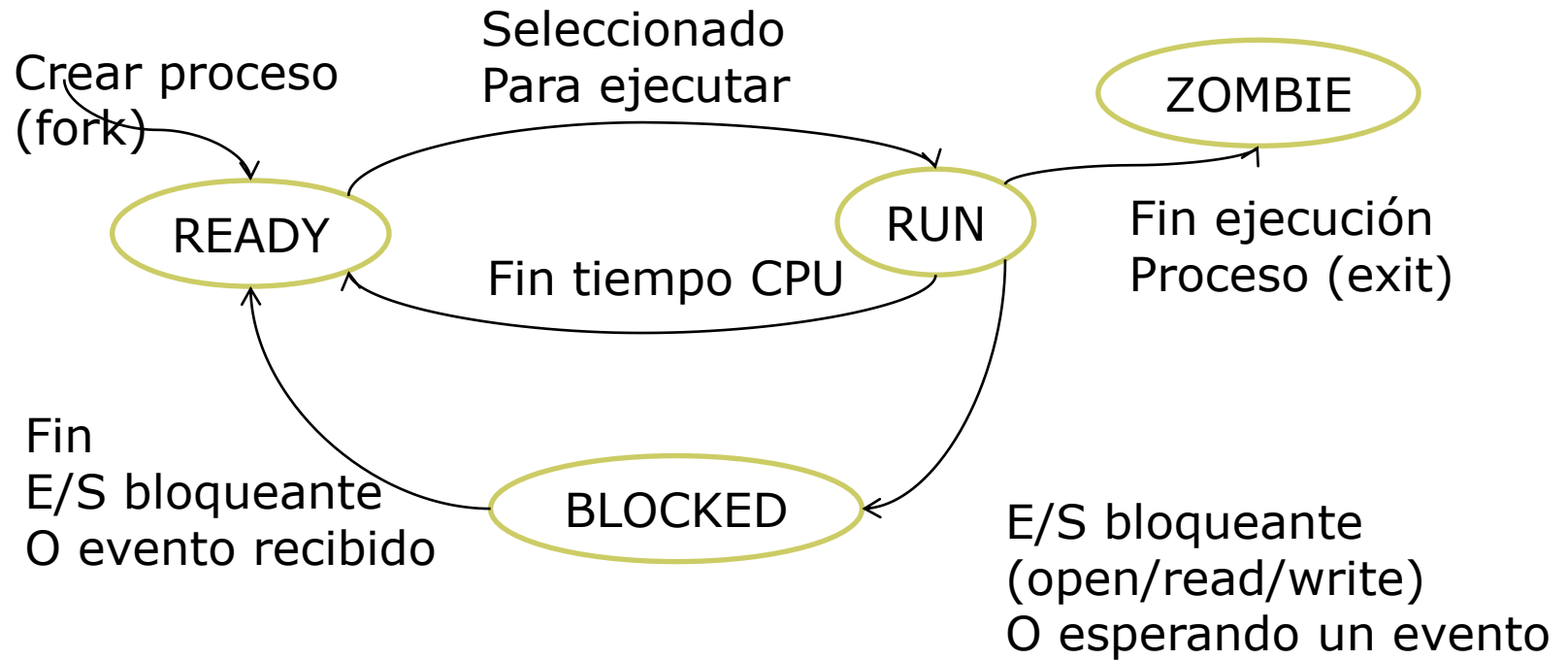
- Durante la vida de un proceso, su “estado” de ejecución cambia
  - Se crea (inicialización de estructuras de datos, carga en memoria, etc)
  - El SO lo pone a ejecutar (le asigna CPU)
  - El SO puede decidir poner otro proceso en la CPU
    - ▶ Cuando accede a dispositivos de lento acceso
    - ▶ Cuando está esperando un evento
    - ▶ Cuando pasa un cierto tiempo
    - ▶ Cuando termina (en este caso siempre cambiaría a otro proceso)
    - ▶ Otras situaciones dependiendo del sistema
- El SO define un grafo de estados, indicando que eventos generan transiciones entre estados

# Estados de un proceso (2)

---

- Los estados (el número y el nombre) en los que puede estar un proceso dependen del sistema y se representan mediante un grafo que define los estados y las transiciones entre ellos: que transiciones son posible 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 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 ya que su padre no ha hecho un waitpid.
    - ▶ Linux

## Ejemplo diagrama de estados (3)



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

# 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
- **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.

# Linux: Propiedades de un proceso (3)

---

- El **ENTORNO** de un proceso se hereda del proceso que lo crea (su *padre*). El entorno se compone de dos listas (*null-terminated vectors*):
  - La lista de **parámetros** (o argumentos). Son los parámetros pasados por línea de comando cuando se ejecutó el programa. El primero de la lista (argv[0]) es el nombre del ejecutable.
  - La lista de **variables de entorno**. Cada elemento son pares con el formato NOMBRE=VALOR.
  - La existencia de estas variables y el hecho que se hereden es una forma de pasar información de padres → hijos.
    - ▶ Las variables se pueden modificar
  - Además, algunas variables de entorno son interpretadas por el SO, ofreciendo una forma fácil de configurar el sistema en función del proceso, en lugar de tener una única configuración global
    - ▶ PATH → lista de directorios donde buscar ejecutables
    - ▶ HOME → Directorio base del proceso

# Linux: Propiedades de un proceso (4)

---

- El **CONTEXTO** es la información que define el estado del proceso en cada momento (no solo el estado puntual sino información detallada). Es como una foto de su ejecución. La información que define el contexto está relacionada con su ejecución
  - **Información de planificación.** Es la información que el planificador necesita para poder suspender/resumir el proceso.
    - ▶ Contenido de registros
    - ▶ Mapeo de memoria
  - **Información de contabilidad de recursos (accounting).** Información sobre los recursos que está utilizando y los que ha consumido durante su ejecución (Hay límites en la utilización de recursos)
  - **Información de E/S.** Es un vector de punteros relacionada con la Gestión de dispositivos.



# Linux: Propiedades de un proceso (5)

---

## ■ El **CONTEXTO...**

- La **tabla de gestión de signals** (signal-handler table). Los procesos pueden mandarse eventos (signals) y recibir determinados eventos del kernel. Cada proceso tiene una tabla que define qué hacer para cada evento.
- **Información sobre la memoria virtual**. Describe el contenido de su espacio de direcciones.

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

## **SERVICIOS BÁSICOS PARA GESTIONAR PROCESOS**

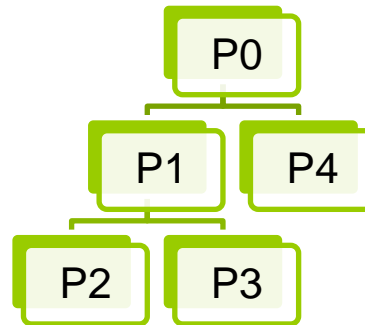
# Servicios y funcionalidad

---

- El sistema nos ofrece como usuarios un conjunto de funciones (llamadas a sistema) para gestionar procesos
  - Crear /Gestionar/Eliminar procesos
  - Suspende/Resumir procesos
  - Proporcionar mecanismos de sincronización
  - Proporcionar mecanismos de comunicación
    - ▶ Memoria compartida
    - ▶ Dispositivos especiales
    - ▶ Gestión de signals

# 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)

---

- Recursos
  - El padre y el hijo comparten todos los recursos
  - El hijo comparte parte de los recursos del padre
  - **El padre y el hijo no comparten nada (UNIX)**
- 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**
  - 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 (execvp)
Terminar proceso	exit
Esperar a proceso hijo	wait/waitpid
Devuelve el PID del proceso	getpid
Devuelve el PID del padre del proceso	getppid

- Hay varias variantes de la llamada exec, por simplicidad usaremos execvp
- Mas info: **man fork, man execvp, man exit, man waitpid...**

# Crear proceso: fork en UNIX



- Un proceso crea un proceso nuevo. Se crea una relación jerárquica padre-hijo
- El padre y el hijo se ejecutan de forma **concurrente**
  - El hijo inicia la ejecución en el punto en el que estaba el padre en el momento de la creación
  - Valor de retorno del fork es diferente:
    - ▶ Padre recibe el PID del hijo
    - ▶ Hijo recibe un 0.
- El hijo es un **duplicado** del padre (recibe una copia privada), hereda (recibe una copia de....):
  - **El espacio de direcciones (código, datos, pila, etc).**
  - 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
  - Contadores internos de utilización (Accounting)
  - Alarmas y signals pendientes

```
int fork();
```



# Ejemplos con fork (1)

- ¿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 (2)

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

```
int id1, id2, ret;

id1 = getpid();
ret = fork();
id2 = getpid();
printf("Valor de id1: %d; valor de ret: %d; valor de id2: %d\n", id1,
ret, id2);
```



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

```
int id1,ret;

id1 = getpid(); /* getpid devuelve el pid del proceso que la ejecuta */
ret = fork();
id1 = getpid();
printf("Valor de id1: %d; valor de ret: %d", id1, ret);
```



# Ejemplo: fork/exit (examen)



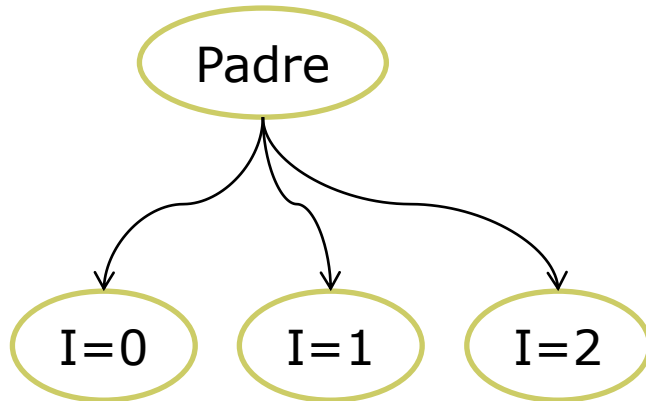
```
void main()
{
    ...
    printf("Mi PID es el %d\n", getpid());
    for (i = 0; i < 3; i++) {
        pid = fork();
        if (pid == 0)
            hacerTarea();
    }
    while (1);
}

void hacerTarea()
{
    printf("Mi PID es %d y el de mi padre %d\n", getpid(), getppid());
    exit(0); ← Ahora, probad de quitar esta instrucción
}
```

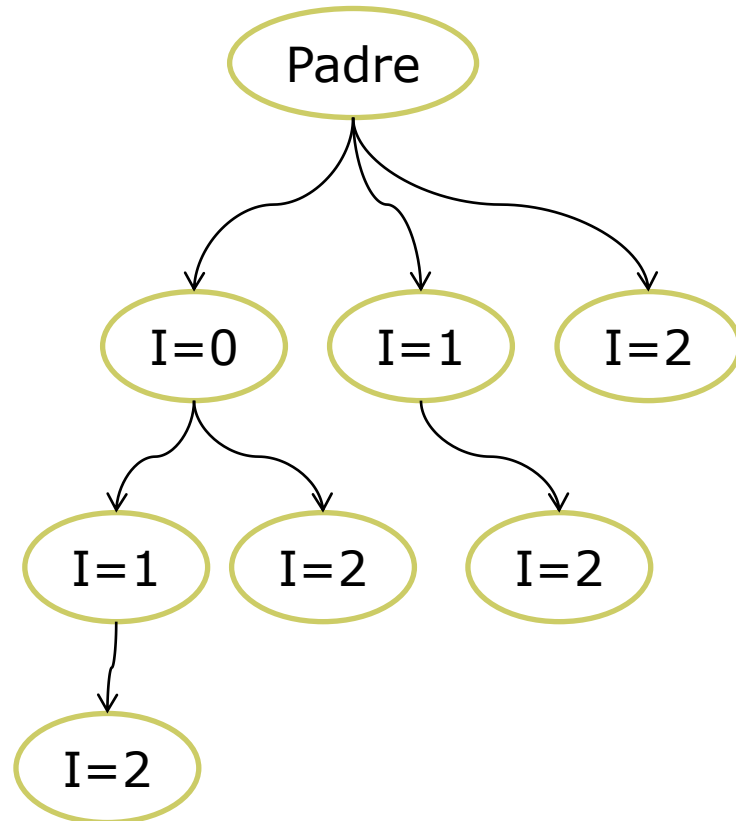
Podéis encontrar el código completo en: *Nprocesos.c* y *NprocesosExit.c*

# Árbol de procesos (examen)

Con exit

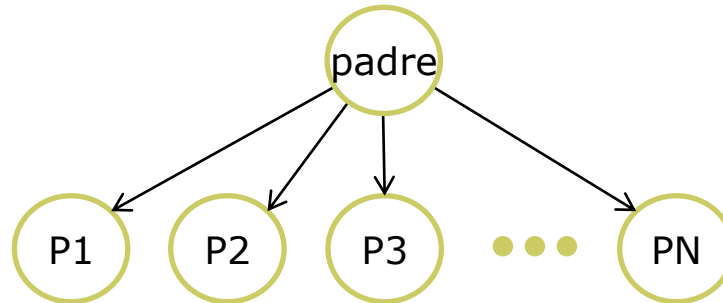


Sin exit

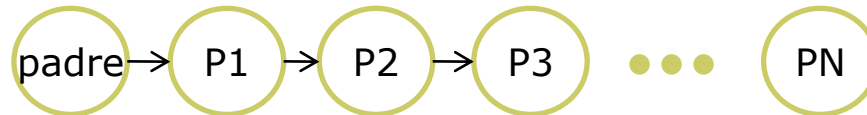


# Otros ejemplos con fork

- 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**.
- `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
  - **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

# 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

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];  
...}
```

# 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?

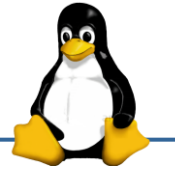
```
...
pid = fork();
if (pid == 0) {
    exec1p("/bin/ls", "ls", "-l", (char *)NULL);
}
// A partir de aquí, este código sólo lo ejecuta el padre
...
```



- Hace falta poner un *exit*?



# Terminación de procesos



- Cuando un proceso quiere finalizar su ejecución , liberar sus recursos y liberar las estructuras de kernel reservadas para él, se ejecuta la llamada a sistema `exit`.
  - El hijo puede enviar información de finalización (exit code) al padre mediante la llamada a sistema `exit` y el padre la recoge mediante `wait/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
  - Si un proceso muere sin liberar los PCB's de sus hijos el proceso `init` del sistema los libera

# Ejemplo: fork/waitpid



## ■ Código 1:

```
for (i = 0; i < 3; i++) {  
    pid = fork();  
    if (pid == 0) hacerTarea();  
    waitpid(-1, NULL, 0);  
}
```



## ■ Código 2:

```
for (i = 0; i < 3; i++) {  
    pid = fork();  
    if (pid == 0) hacerTarea();  
}  
while ((waitpid(-1, NULL, 0) > 0);
```

## ■ Preguntas:

- ¿Cuál es concurrente?
- ¿Cual es secuencial?
- Dibuja el árbol de procesos en cada caso

# Ejemplo: gestión del código de exit



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

...
pid = fork();
if (pid == 0) hacerTarea();

...
pid = waitpid(-1, &exit_code, 0); // o también pid = wait(&exit_code);

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

# 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] );
    while ((pid = waitpid(-1, &exit_code, 0) > 0)
        trataExitCode(pid, exit_code);
    exit(0);
}

void lanzaCmd(char *cmd)
{
    ...
    pid = fork();
    if (pid == 0)
        execlp(cmd, cmd, (char *)NULL);
}

void trataExitCode(...)
    ...
```

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

---

-Linux:Signals

# COMUNICACIÓN ENTRE PROCESOS

# Comunicación entre procesos

---

- Los procesos pueden ser independientes o cooperar entre si
- ¿Porqué 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)

# Linux: Signals



- Notificaciones que indican que ha sucedido un determinado evento
  - Cada tipo de evento tiene un signal asociado
    - ▶ Ejemplos: SIGALRM, SIGSEGV, SIGINT, SIGKILL, SIGSTOP, ...
    - ▶ El signal es un número, pero existen constantes definidas para usarlas en los programas o en línea de comandos
  - Están predefinidos por el SO
  - Hay 2 no definidos para que el usuario lo use como quiera
- Quién los envía
  - El SO para informar a los procesos
  - Procesos del mismo usuario: para comunicarse y sincronizarse entre ellos
    - ▶ llamada a sistema kill y comando kill
- Usos que les daremos principalmente
  - Control del tiempo (alarmas)
  - Algunos signals están relacionados con el control de procesos (parar/continuar)
  - Sincronizar procesos, los signals notificarán sucesos.
- La gestión de signals es por proceso
  - Cada proceso tiene una tabla (1 entrada por signal), donde se indica que acción realizar cuando se reciba el evento y tiene un bitmap de eventos pendientes (1 bit por signal)
  - Se guarda en el PCB



# Linux: Signals (2)

---

- Todos los signals tienen una acción definida en la tabla de programación de signals que puede ser (una de estas tres):
  - Ignorarlo (excepto 2: SIGKILL y SIGSTOP). (opción SIGIGN)
  - Ejecuta la acción por defecto del sistema (opción SIGDFL)
  - Ejecutar una función proporcionada por el proceso (excepto 2: SIGKILL y SIGSTOP). A asociar una función del proceso al signal es lo que se conoce como “capturar” el signal
- Modificar el tratamiento asociado: reprogramar el signal
- El kernel tiene 4 posibles **ACCIONES POR DEFECTO** asociadas a los signals que no están marcados para ignorar y no están “capturados” por el proceso.
  - **Terminar**, La acción por defecto es terminar el proceso.
  - **Ignorar**, La acción por defecto es ignorar la señal.
  - **Core**, La acción por defecto es terminar el proceso y realizar un volcado de memoria.
  - **Stop**, La acción por defecto es detener el proceso.

# Linux: Signals (3)

---

Servicio	Llamada sistema
Reprogramar un signal	signal
Enviar un signal	kill
Esperar un evento	pause
Programar el envío automático del signal SIGALRM (alarma)	alarm

- Servicios (Hemos seleccionado un conjunto mínimo)
- **Más info: `man signal`, `man kill`, `man 2 kill`, `man alarm`**
- **Fichero con signals: `/usr/include/bits/signum.h`**

# Ejemplo: Programación básica de signal



```
void main()
{
    ...
    signal(SIGALRM, f_alarma);
    alarm(1);

    while(1) {
        printf("Estoy haciendo cierta tarea\n");
    }
}

void f_alarma()
{
    printf("TIEMPO!\n");
    alarm(1);    // si se quiere un signal cada segundo
                 // hay que reprogramarlo
}
```

*Podéis encontrar el código completo en: `signal_basico.c`*

# Linux: Signals (4)

---

Nombre	Default	Evento
<b>SIGCHLD</b>	IGNORAR	Un proceso hijo ha terminado o ha sido parado
<b>SIGCONT</b>		Continúa si estaba parado
<b>SIGSTOP</b>	STOP	Parar proceso
<b>SIGTERM</b>	TERMINAR	Interrumpido desde el teclado (CtrlC)
<b>SIGALRM</b>	TERMINAR	El contador definido por la llamada alarm ha terminado
<b>SIGKILL</b>	TERMINAR	Terminar el proceso
<b>SIGSEGV</b>	CORE	Referencia inválida a memoria
<b>SIGUSR1</b>	TERMINAR	Definido por el usuario (proceso)
<b>SIGUSR2</b>	TERMINAR	Definido por el usuario (proceso)

# Linux: signals (5)

---

- El PCB de un proceso contiene información con:
  - Una tabla de acciones asociados a los signals
  - Un vector de bits con los signals pendientes de procesar ( 1 bit por signal)
  - Una máscara para indicar que eventos hay que tratar
- Relación con fork y exec
  - FORK: **Proceso nuevo**
    - ▶ El hijo hereda la tabla de acciones asociadas a los signals del proceso padre
    - ▶ Los eventos son enviados a procesos concretos (PID's), el hijo es un proceso nuevo → La lista de eventos pendientes se borra
  - 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

# Ejemplo: reprogramación de signals (1)



```
void main()
{
    ...
    signal(SIGALRM, f_alarma);
    for(i = 0; i < 10; i++) {
        alarm(2);
        pause();
        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: reprogramación de signals (2)



```
void main()
{
    ...
    signal(SIGALRM, f_alarma);
    signal(SIGCHLD, fin_hijo);
    for (i = 0; i < 10; i++) {
        alarm(2);
        esperar_alarma(); // OPCIONES
        alarma = 0;
        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);
}
```

*Podéis encontrar el código completo en: `cada_segundo_sigchld.c`*

- 
- 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 → PCB
  - **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

# 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

# Planificador

---

- Eventos que pueden generar que se ejecute el planificador (existen más, pero sólo veremos estos)
  1. Si el proceso realiza un acceso a un dispositivo bloqueante y el proceso pasa de estado run a estado blocked → Evento no preemptivo
  2. Si el proceso que está en ejecución termina → Evento no preemptivo
  3. Si el proceso lleva X tiempo ejecutándose → Evento preemptivo
- El S.O. puede ser preemptivo (apropiativo) o no preemptivo (no apropiativo)
  - No preemptivo: El sistema no le quita la cpu al proceso, él la “libera”. Sólo soporta eventos no preemptivos. (eventos tipo 1 y 2)
  - Preemptivo: El sistema le quita la cpu al proceso. Soporta eventos preemptivos (eventos tipo 3) y no preemptivos.
- Dependiendo del sistema, podrían haber otro eventos, como la llegada de procesos más prioritarios

# Caracterización de procesos

---

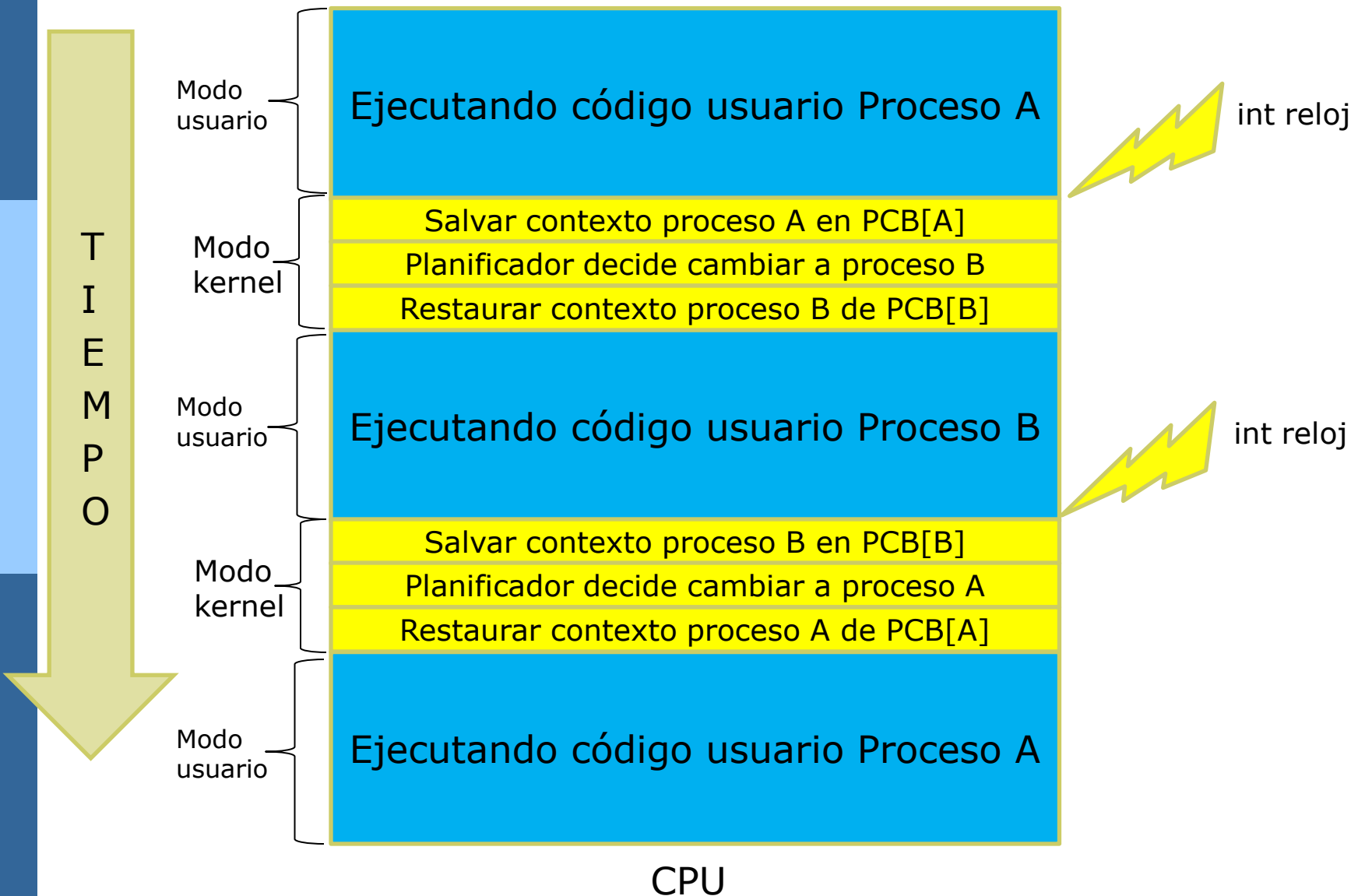
- Los procesos suelen presentar ráfagas de computación y ráfagas de acceso a dispositivos (E/S) que pueden bloquear al proceso
- En función de estas ráfagas, los procesos se consideran:
  - Procesos de cálculo: Consumen más tiempo haciendo cálculo que E/S
  - Procesos de E/S: Consumen más tiempo haciendo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

# 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
  - Tiempo de espera de un proceso- Tiempo que el proceso pasa en estado ready

# Representación de la política

---

- Para representar una planificación concreta con un conjunto de procesos se puede utilizar el Diagrama de Gantt
  - Diagrama horizontal (tiempo en el eje X) que muestra para cada instante que valor toma un cierto parámetro
    - ▶ Por ejemplo el estado de cada proceso o el proceso que está en la CPU en cada momento

---

Round Robin

# **POLÍTICAS DE PLANIFICACIÓN**

# Round Robin (RR)

---

- 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 por E/S
  2. Cuando termina el proceso
  3. Cuando termina el quantum
- 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.

# Ejemplo: Diagrama Gantt

- 4 procesos con ráfagas de cpu y e/s

	CPU	E/S	CPU	E/S	CPU
<b>P0</b>	3ms	2ms	5ms		
<b>P1</b>	6ms	1ms	5ms	1ms	5ms
<b>P2</b>	2ms	3ms	5ms		

- El orden de llegada es P0, P1 y P2
- Dibuja el Diagrama de gantt con una política RoundRobin Q=5
- Calcula el tiempo de espera medio
- Calcula el tiempo de ejecución medio

[illegible]

Tiempo espera:

P0:5ms

P1:14ms

P2:11ms

Tiempo medio espera=(5+14+11)/3=10ms

Tiempo total ejecución:

P0:15ms

P1:32ms

P2:21ms

Tiempo medio ejecución=  $(15+32+21)/3=22,6\text{ms}$



---

## **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 bloqued 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 operaciones privilegiadas
- 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.

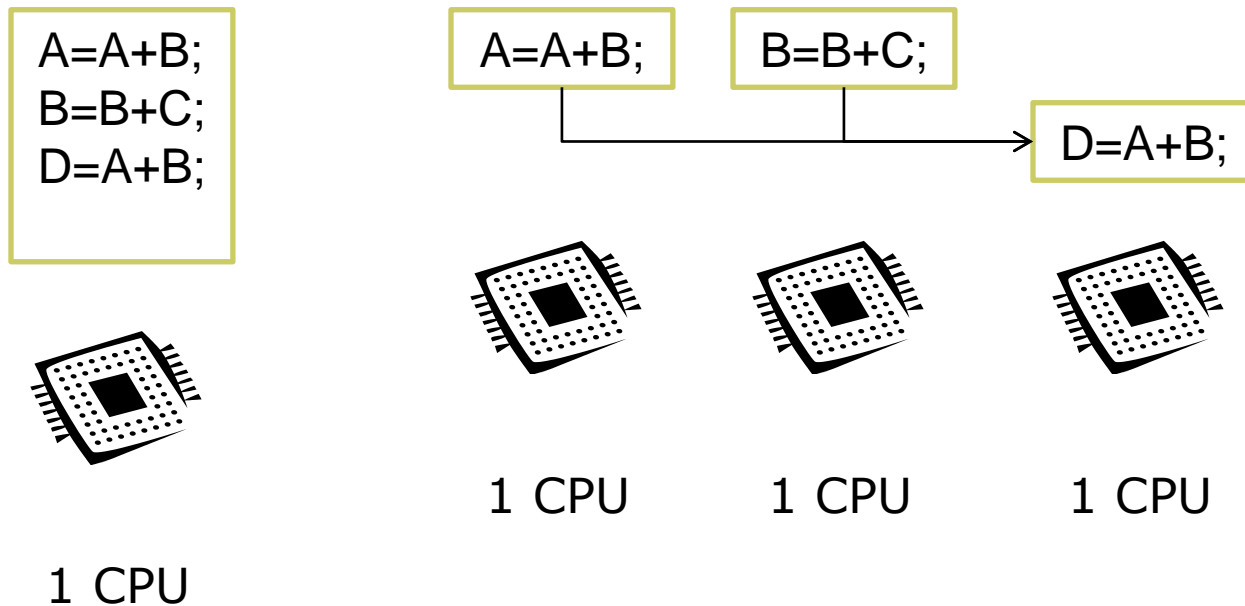
---

# PROCESOS MULTI-THREAD



# Flujos

- Un proceso puede tener partes de su código (tareas) que se pueden ejecutar en paralelo ya que no dependen unos de otro
  - Ejemplo:



# Flujos

---

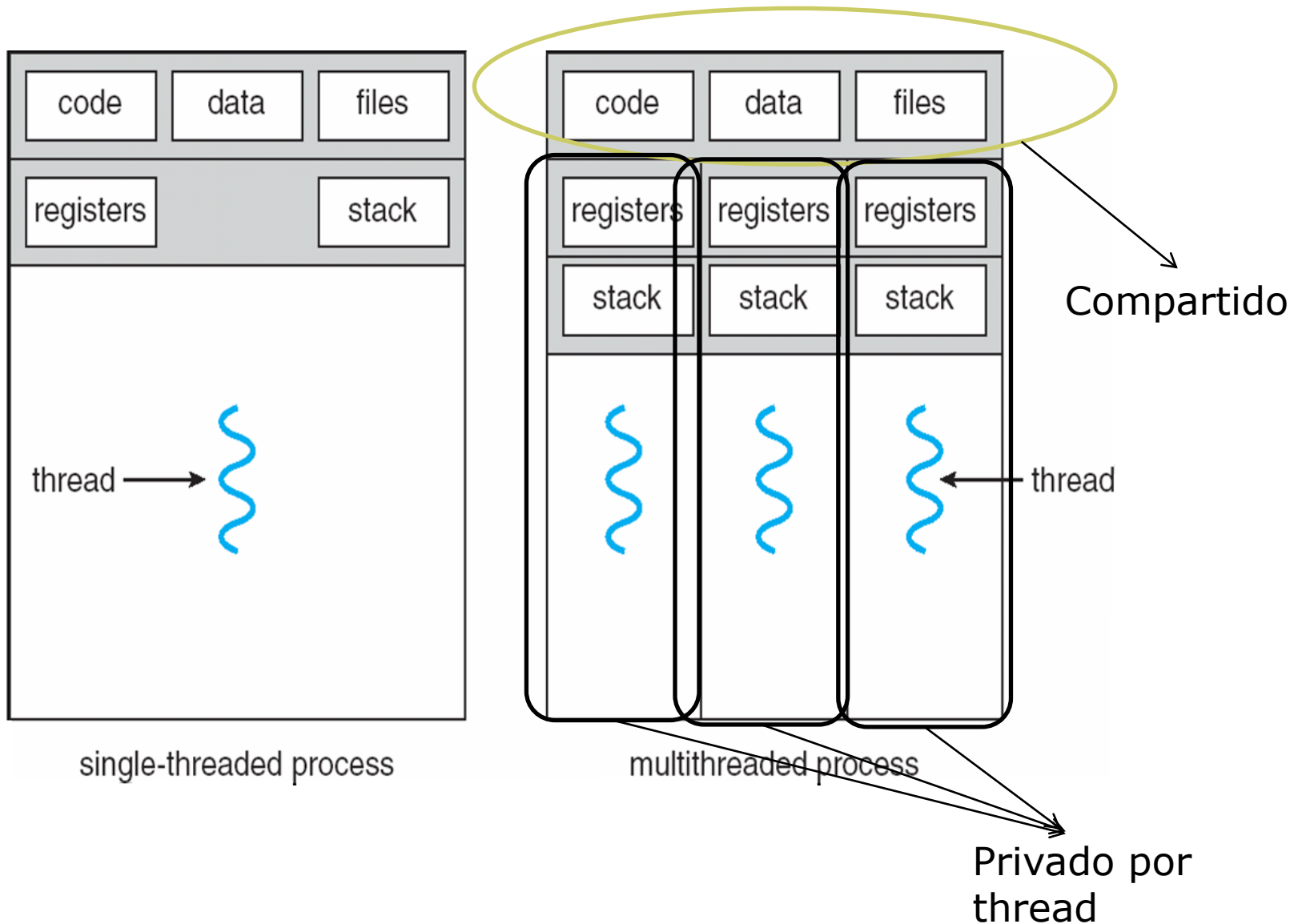
- Ejecución concurrente de diferentes secuencias de instrucciones del mismo código:
  - Permiten programar de forma modular (encapsular tareas)
  - Permiten explotar el paralelismo (multiprocesadores)
  - Programas que realizan Entrada/Salida
    - ▶ Usar procesos dedicados SOLO a la Entrada/Salida
  - Atención a varias peticiones de servicio (Servidores)
  
- Se puede conseguir con los programas que crean varios procesos
  - Para cada proceso réplica del espacio de direcciones y de todos los recursos asociados
  - ¿Es necesario?: todas las secuencias de instrucciones concurrentes podrían compartir la mayor parte de recursos
  - El SO sólo necesita mantener por separado:
    - ▶ Un identificador
    - ▶ Program Counter : indica la siguiente instrucción a ejecutar
    - ▶ Registros del procesador
    - ▶ Pila: para gestionar las llamadas a subrutinas que haga esa secuencia de instrucciones

# Flujos (2)

---

- 1 Flujo (o thread) es la unidad básica de utilización de CPU. También podríamos decir que es una CPU virtual.
  - Un proceso tiene un mínimo de 1 flujo de ejecución
  - Un proceso tradicional tiene un solo flujo de ejecución
- Los diferentes threads son partes del programa que pueden ejecutarse en paralelo. Habitualmente subrutinas.
- Todos los threads de un proceso comparten todos los recursos del proceso excepto el program counter , los registros
  - ▶ Aunque cada thread tiene su pila para uso privado todos pueden acceder a ella
  - ▶ También tienen su propia variable errno (1 por thread)
- Flujos de procesos distintos NO COMPARTEN RECURSOS (ni memoria, ni dispositivos)

# Flujos (3)



# Ventajas de utilizar flujos

---

- Algunos códigos puede programarse como múltiples procesos o múltiples threads. En ese caso, conviene tener en cuenta:
  - Es más rápido crear un thread que un proceso
  - Es más rápido terminar un thread que un proceso
    - No necesitamos duplicar todas las estructuras, espacio de memoria etc, sino que gran parte es compartido
  - Es más rápido cambiar de thread (dentro del mismo proceso) que cambiar de proceso
  - Ya que los threads comparten memoria y ficheros, se puede intercambiar información sin llamar a rutinas de sistema
    - ▶ También puede ser un problema si se produce de forma involuntaria o no coordinada. El resultado de la ejecución puede ser incorrecta.
- Por todo esto, explotar la concurrencia (o paralelismo) mediante threads nos permite hacerlo normalmente a un grano más fino

# Linux: Procesos y threads

---

- Linux utiliza la misma representación interna para procesos y threads. Un thread es simplemente un nuevo proceso que comparte el espacio de direcciones con su padre
- Hay una nueva llamada a sistema, `clone`, más genérica que `fork`
  - `fork` crea un nuevo proceso con un nuevo contexto
  - `clone` crea un nuevo proceso, con una nueva identidad, pero que potencialmente puede compartir recursos con su padre, los parámetros de `clone` determinan que datos se comparten y que datos no, siendo mucho más flexible que `fork`.

# Pthreads

---

- Es la librería de gestión de threads más generalizada
- Puede estar implementada a nivel usuario o ofrecidos directamente por el kernel. Nos ofrece principalmente **PORTABILIDAD**
  - **P**ortable **O**perating **S**ystem **I**nterface
  - **POSIX** standard (IEEE 1003.1c) API para la creación y sincronización de threads
  - El API define el comportamiento de la librería, la implementación depende de cada caso
  - Está en todos los sistemas UNIX (Solaris, Linux, Mac OS X)
  - Windows proporciona un “Subsystem for UNIX-based Applications (SUA)” que ofrece una shell compatible con POSIX para ejecutar aplicaciones UNIX
  - Define la interfície de gestión de flujos
    - ▶ Creación, destrucción (pthread\_create, pthread\_exit)
    - ▶ Prioridades
    - ▶ Planificación
    - ▶ Sincronización (pthread\_mutex\_lock, pthread\_mutex\_unlock, ..)

---

# **SOPORTE A DIFERENTES ARQUITECTURAS/ENTORNOS**

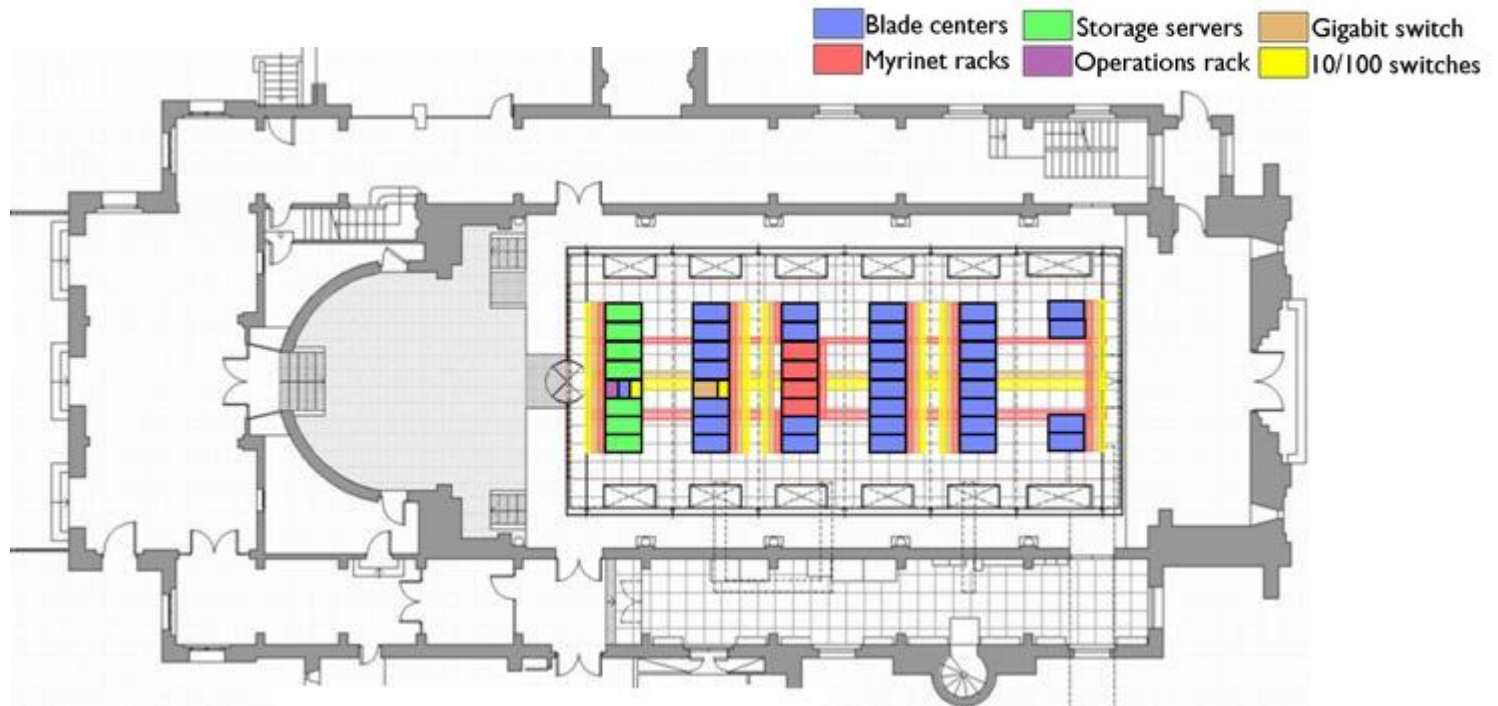


# Diferentes sistemas/entornos

---

- Dependiendo de la arquitectura tenemos diferentes configuraciones, políticas, etc
  - Ordenadores “personales”
  - Clusters de SMP's
  - **Supercomputadores**
  - Portátiles
  - Sistemas multimedia
  - Mviles (empotrados)
  - **Sistemas con políticas Real-time (incluidos multimedia)**

# Supercomputadores



- ❑ Barcelona Supercomputing Center (MareNostrum)
- ❑ Cada Blade tiene: 2 procesadores a 2.3GHz, ejecutando Linux.

# Supercomputadores

---

## ■ Requerimientos:

- Es un sistema de **High Performance** Computing (HPC)
- Muchos usuarios
- Muchos trabajos enviados a ejecutar
  - ▶ Ejecución interactiva : Para desarrollar código pero no es la misión principal de un Supercomputador
  - ▶ Ejecución Batch (la mayoría)
  - ▶ Aplicaciones paralelas (sinó para que 10K procesadores?). En estos entornos se llaman trabajos o jobs.
    - Cada nodo es un SMP de 4 CPUS → Mucha comunicación por red → Altos requerimientos de red.
- El objetivo del sistema es:
  - ▶ Maximizar el uso de la máquina
  - ▶ Ejecutar las aplicaciones lo más rápido posible
  - ▶ Como no son interactivas, el tiempo de respuesta no es lo importante

# Configuración

---

- Varios niveles de planificación y software de gestión muy avanzado
  - Se reservan algunos nodos para el trabajo interactivo. El resto de la máquina se gestiona con un gestor de colas:
    - ▶ MOAB : Software de gestión de colas. Gestiona diferentes colas a las que se asignan recursos (máximo tiempo de ejecución, máximo número de procesos), etc. Implementa una política que es una variante de **FCFS**. Los jobs piden directamente lo que necesitan y se les asigna lo que piden. Es un software de sistema pero está por encima del sistema operativo.
    - ▶ SLURM: Gestor de recursos. Crea los procesos y se encarga de monitorizarlos. Es un software de sistema pero está por encima del sistema operativo.
    - ▶ Linux: en cada nodo hay una instancia de Linux normal.
  - Se intenta minimizar los cambios de contexto para que las aplicaciones vayan lo más rápido posible

# Sistemas Real Time

---

- ¿Que es Real Time?
  - Un sistema Real –Time garantiza que los resultados se producen en un plazo determinado
  - Que se cumplan o no los plazos puede ser más o menos grave dependiendo del sistema
    - ▶ Real-Time críticos: Resultados catastróficos si no se cumplen los plazos. Entornos industriales.
  - Entornos Hard/Softt real-time. El sistema garantiza los plazos o simplemente prioriza las tareas real-time
- Características de estos sistemas, típicamente son para sistemas dedicados, con requerimientos muy claros de tiempo, se suelen implementar en dispositivos pequeños, como embeded.
- Que ofrece un kernel real-time
  - No suele servir para entornos típicos de PC de sobremesa
  - Suelen ser muy específicos
  - No requieren interacción con el usuario
  - Requieren información detallada de tiempos de los procesos
  - No están pensados para interactuar con los dispositivos que hay en los PC's

# Sistemas real-time

---

- Las tareas (procesos) están perfectamente descritas mediante:
  - Un periodo: son repetitivas
  - Un plazo(deadline): han de ejecutarse antes de ...
  - Un tiempo de ejecución
- Las diferentes políticas son más o menos seguras y intentan ( o garantizan) los plazos.
- El conjunto de tareas suele ser conocido “a priori”
- Los sistemas calculan si los plazos puede o no ser garantizados dados un conjunto de tareas
- El API de Pthreads ofrece funciones para gestionar threads real-time
  - Ofrece dos clases de threads
    - ▶ SCHED\_FIFO: Se sigue una política FCFS. No hay cambios de contexto para threads con la misma prioridad
    - ▶ SCHED\_RR . Como la anterior pero con RoundRobin para threads de la misma prioridad