

Tema 6. Control de Procesos



.

Contenidos

1. Motivación.
2. Contexto de ejecución. Flujos de ejecución. Procesos e hilos.
3. Sistemas multiprogramados y multihilo.
4. Estados y transiciones.
5. Cambio de contexto.
6. Planificación de procesos.
7. Llamadas al sistema relacionadas con el control de procesos.
8. Forma general de trabajo de un sistema operativo multiprogramado

Bibliografía:

[Capítulos 4 y 5] F.M. Márquez: UNIX. Programación Avanzada 3ª Edición. Rama, 2004.

[Capítulos 3 y 5.3] C. Rodríguez, I. Alegria, J. González, A. Lafuente: *Descripción Funcional de los Sistemas Operativos*. Síntesis, 1994

[Capítulos 9 y 10] W. Stallings: Sistemas Operativos. 5º Ed. Pearson Prentice-Hall, 2005.

[Capítulo 2.1, 2.3, 6 y 7] G. Nutt: Sistemas Operativos. 3º Ed. Pearson Addison Wesley, 2004

[Capítulos 4,5 y 6] A. Silberschartz: Operating Systems Concepts. Sixth, John Wiley & Son, 2003

6.1 Motivación

Diferencia de velocidad entre los dispositivos de E/S y el Procesador.

a) E/S Asíncrona

- ✓ Ordenación especial de las instrucciones.
- ✓ Incrementa la complejidad de la programación.
- ✓ Necesita sincronización posterior.

b) Buffering

Utilizar varios buffers donde se van a realizar las lectura/escrituras de manera transparente al programador.

Doble Buffer:

- Probabilidad de mucha E/S, lleno o vacío
- Ocupar memoria

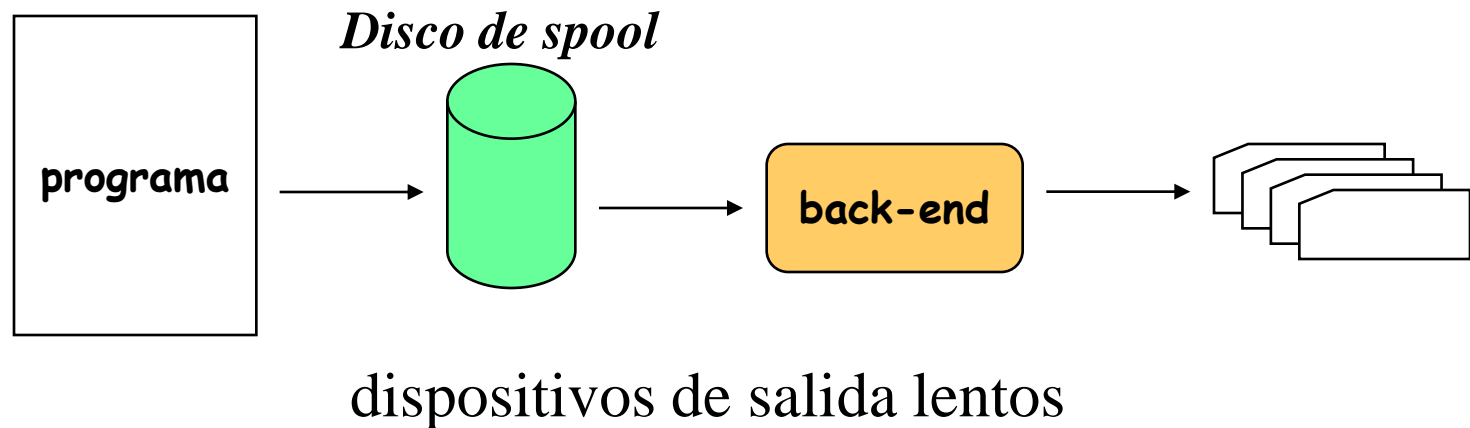
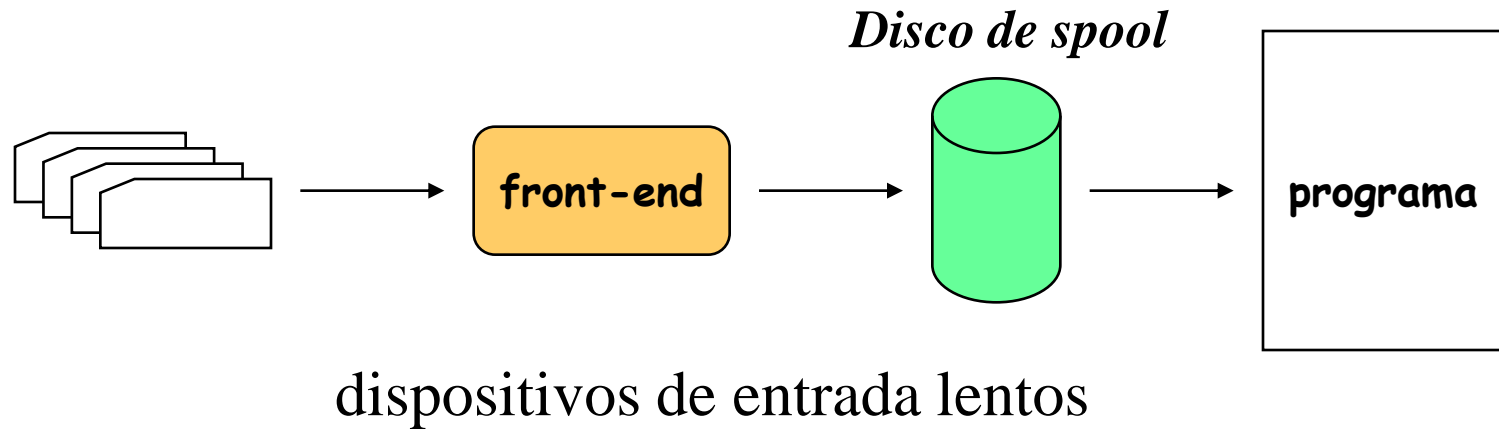
c) Spooling

Cuando se utilizan dispositivos de E/S lentos y no de forma interactiva.

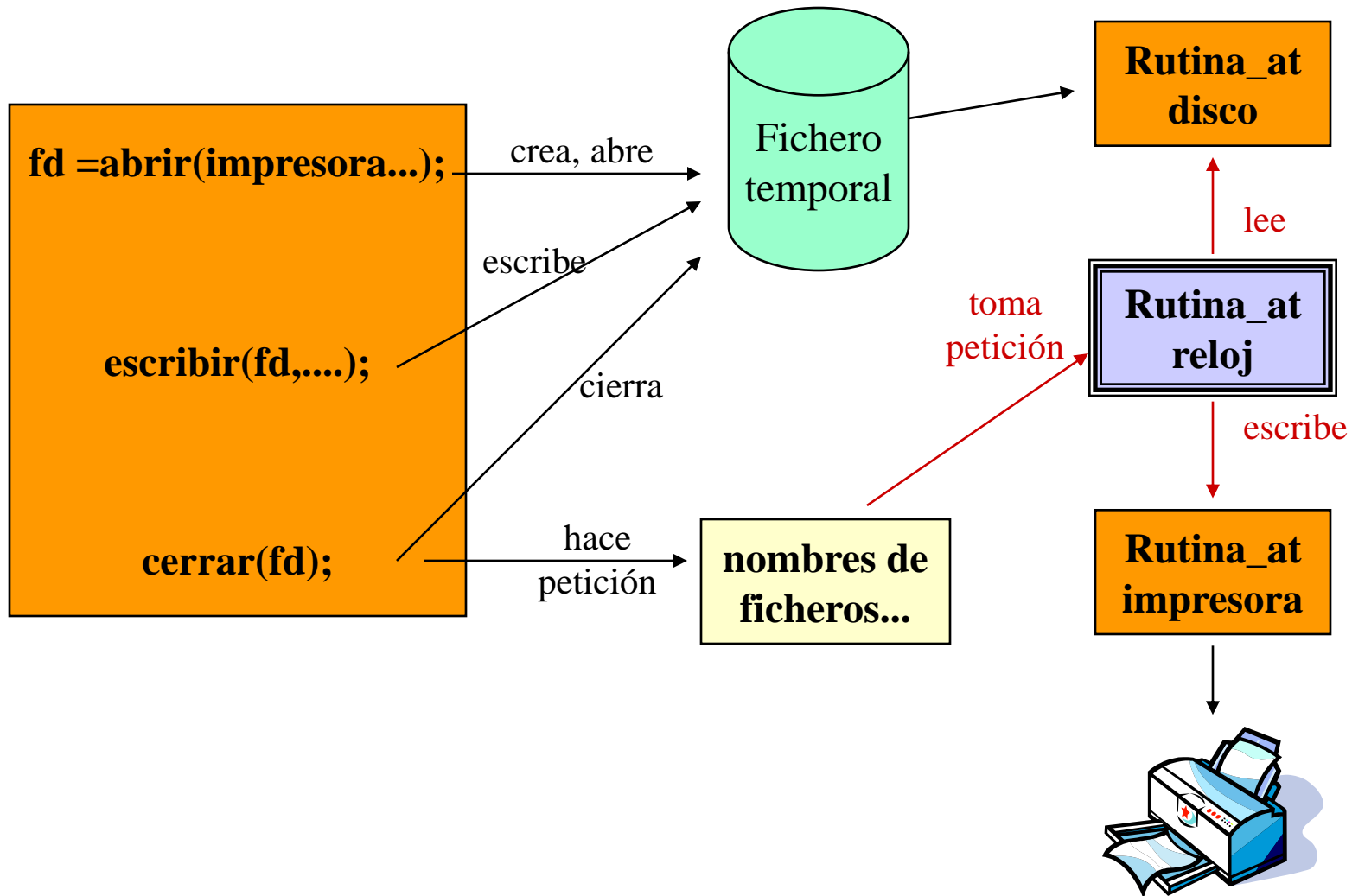
Se puede utilizar un dispositivo intermedio más rápido.

Suele haber un hardware dedicado

“Spooling”



“Spooling” de la impresora



Ventajas - Inconvenientes

👉 Ventajas

- ✓ Buffering: solapa E/S con su propia ejecución.
- ✓ Spooling: solapa E/S con la ejecución de otro trabajo.
- ✓ Spooling: es fácil obtener varias copias del trabajo.

👉 Inconvenientes

- ✓ Espacio en disco.
- ✓ Tiempo de tratamiento.

Se sigue sin resolver el problema:

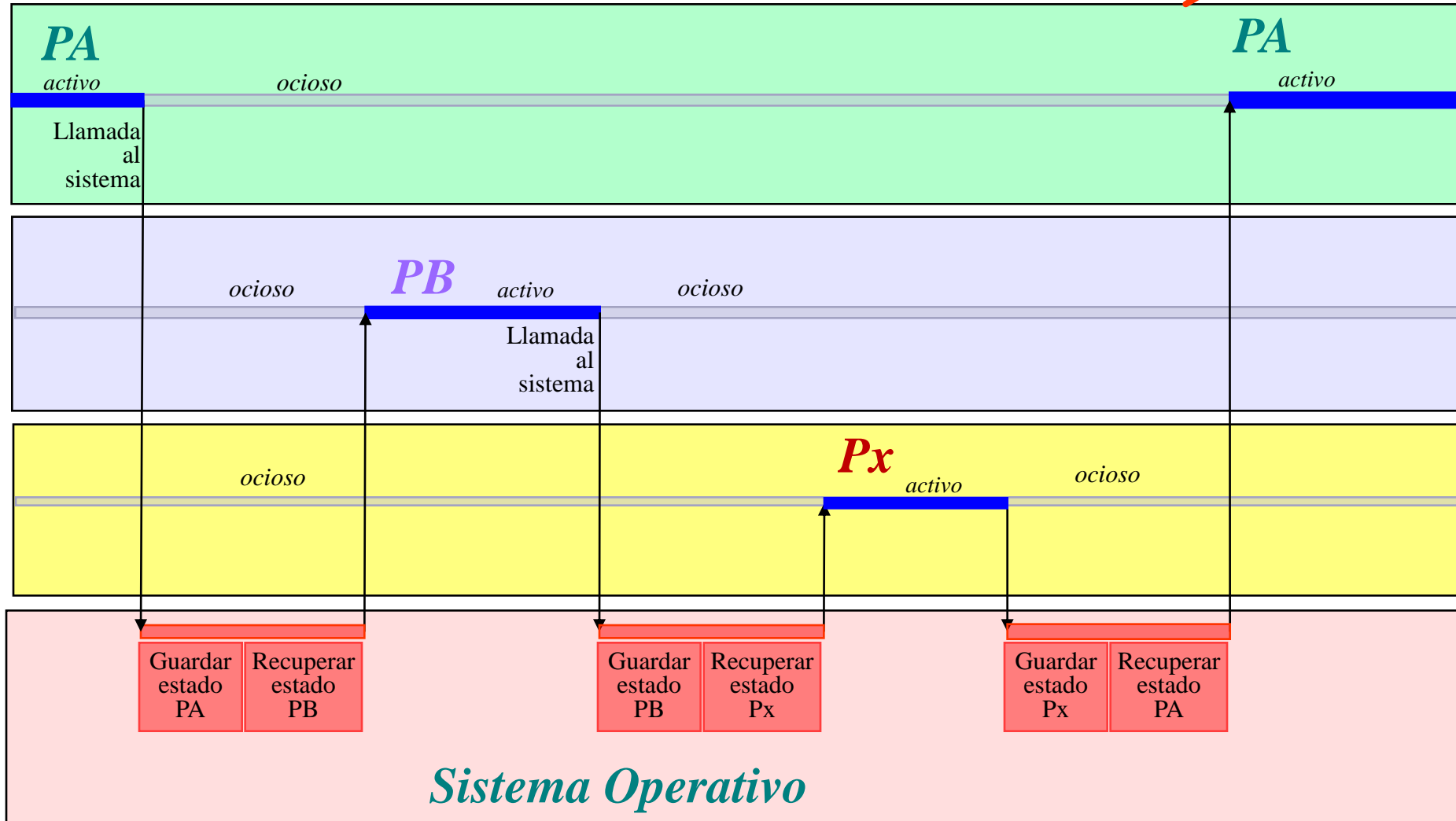
Un usuario, en general no tiene ocupada la CPU todo el tiempo.

❑ Posible Solución:

- Si tenemos suficiente espacio en memoria *cargamos varios programas.*
- Cuando un programa en ejecución necesita esperar por E/S, se le para y si hay otro que pueda ejecutarse, que lo haga.

Cambio de contexto

Tiempo →



6.2 Contexto de ejecución

Asegurar que los programas realizan la función para la que han sido codificados independientemente de si se ejecutan solos (*monoprogramación*) o concurrentemente con otros (*multiprogramación*).

Esto obliga al S.O. a mantener por cada programa la información necesaria para realizar el cambio de ejecución de un programa a otro: **cambio de contexto**

PROCESO = *Información de contexto* + programa(inst. y datos)

Información de contexto:

- Identificador
- Tabla de canales
- Estado de los recursos del procesador (*registros*,...)
- ...

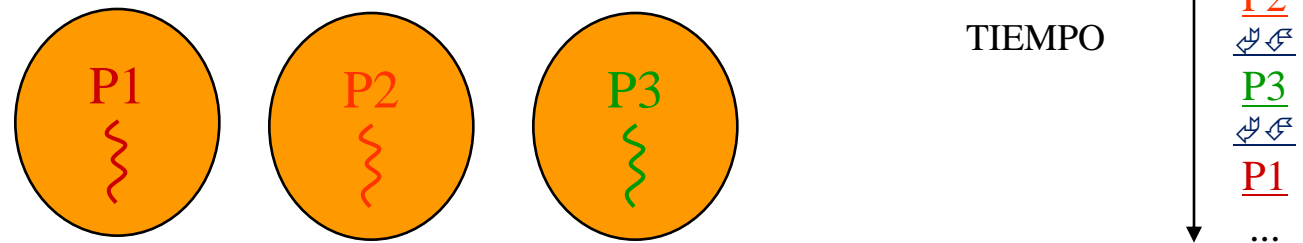
PROCESO: Instancia de un programa en ejecución.

Grado de Multiprogramación

Flujos de Ejecución

Procesos - Hilos

Varios programas en memoria **alternando en el tiempo la ejecución** de su código, todos bajo el control del Sistema Operativo



*El cambio de contexto (↔) entre procesos puede ser costoso (CPU).
Otra posibilidad es el uso de hilos o Procesos ligeros*

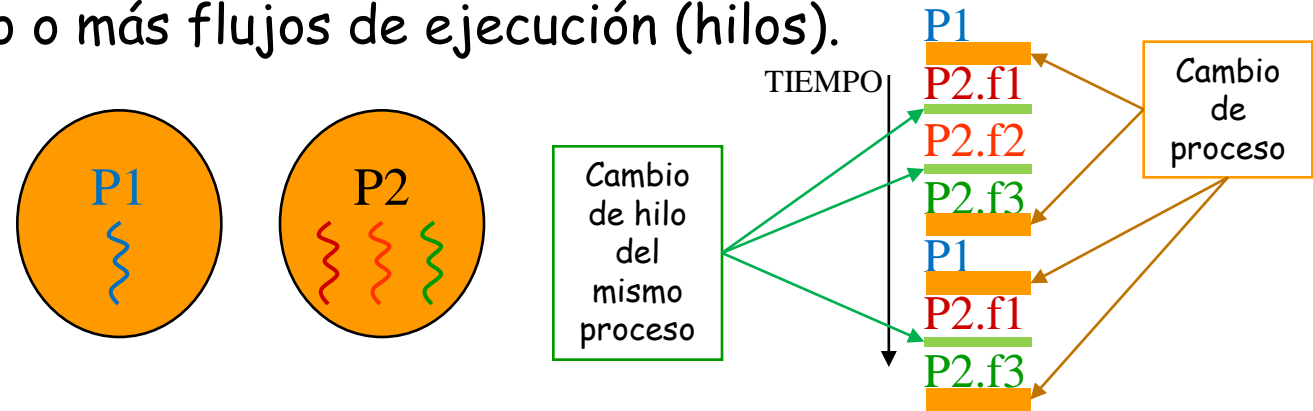
Un proceso (o tarea) puede tener varios flujos de ejecución, **alternando en el tiempo la ejecución** de cada flujo, todos bajo el control del Sistema Operativo(o aplicación)

El cambio de contexto entre hilos de una misma tarea será mucho menos costoso, además va a Permitir compartir ciertos recursos de forma mucho más cómoda.



6.3 Sistemas Multiprogramados y Multihilo

PROCESO: *Una instancia de un programa en ejecución.* Un proceso puede tener uno o más flujos de ejecución (hilos).

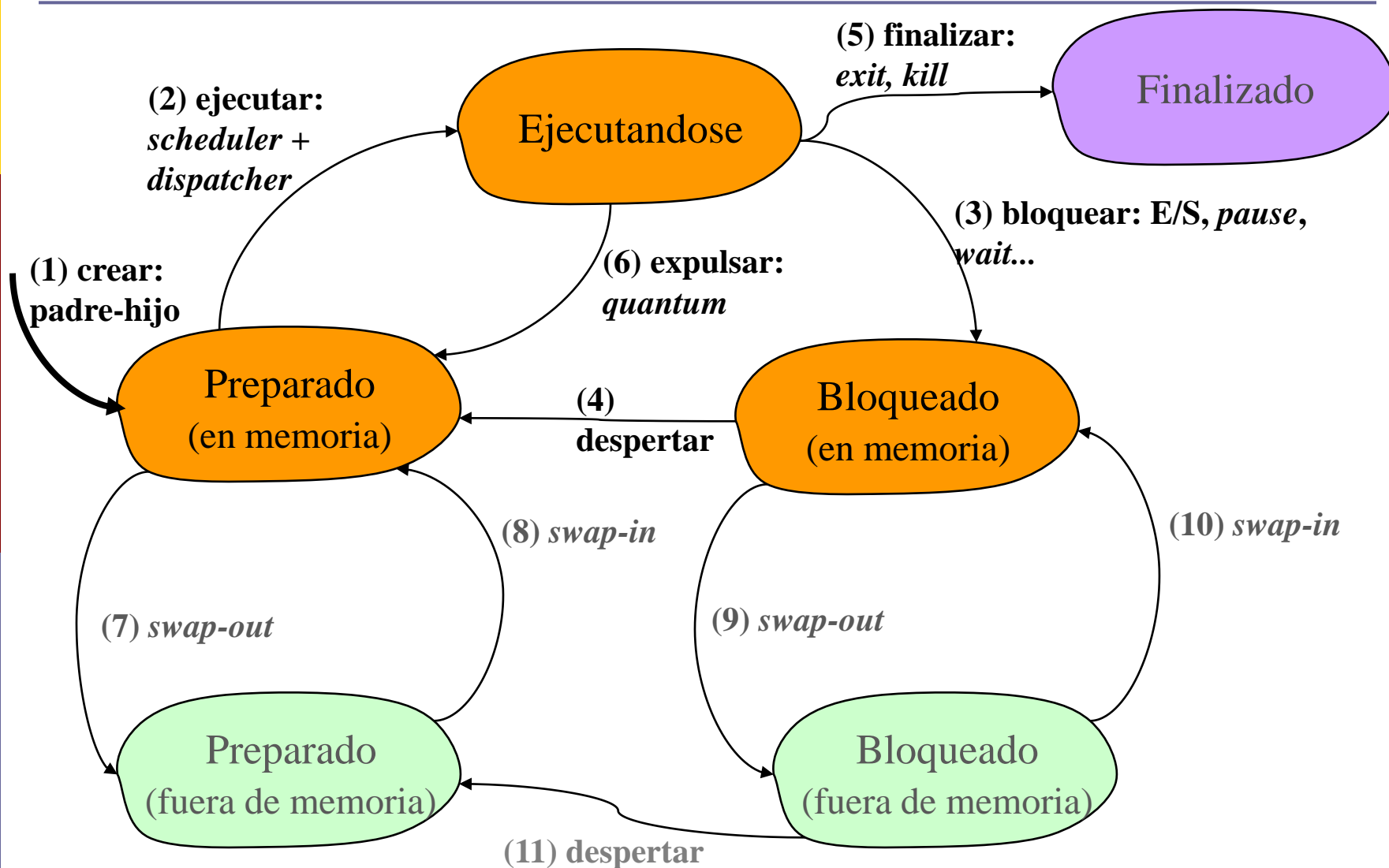


Sistemas Multihilo: Aquellos SO que proporcionan mecanismos de gestión de hilo. Esta gestión la realiza el propio sistema operativo (**Kernel Level Thread**)

En otros sistemas la gestión de los hilos la realizan las propias aplicaciones mediante el uso de una biblioteca de gestión de hilos. El SO no conoce la existencia de los hilos (**User Level Thread**)

Biblioteca Pthreads (Posix Threads): biblioteca estándar que simplifica el desarrollo de aplicaciones basadas en hilos. Además asegura la portabilidad de las aplicaciones a diferentes entornos.

6.4 Estados de un proceso y transiciones



Estados y transiciones de un proceso

(cont.I)

- Un proceso desde que inicia su ejecución hasta que acaba pasa por una serie de estados:

(1) Creación de un proceso

CAUSA: Llamada al sistema de ejecutar un programa

TAREAS: Asignación de un Identificador de proceso.

Carga del programa en memoria (opcional).

Inicialización de la información de Contexto (tabla de canales,...).

Proceso padre - Proceso hijo

Puede fallar, p.e. No existe el programa, no hay memoria suficiente,...

Estados y transiciones de un proceso (cont.II)

(2) Ejecución

CAUSA: El proceso que está en ejecución se bloquea, finaliza o se le acaba el *quantum*.

TAREAS: **SCHEDULER**: función del S.O. que elige el proceso. **Prioridades** estáticas o dinámicas
DISPATCHER: pone en ejecución el proceso seleccionado por el scheduler restaurando su información de contexto.

Proceso NULO.

Estados y transiciones de un proceso

(cont.III)

(3) Bloquearse

CAUSA: Realiza una llamada al sistema que lo bloquea:

- E/S
- Esperar un tiempo
- Sincronizar con otro proceso

...

TAREAS: El S.O. debe guardar la información de contexto del proceso.

Debe colocar otro proceso en ejecución (2).

Estados y transiciones de un proceso

(cont.IV)

(4) Despertar

CAUSA: Ejecución de una rutina **asíncrona** del S.O.
(rutina de atención a un dispositivo)
Final de una sincronización.

...

TAREAS: Poner en estado preparado al proceso que despierta.

Se puede dar la opción de pasar a ejecución. Para ello se pasa a estado preparado el proceso que está en ejecución (expulsión o preención a nivel de CPU) y se realiza de nuevo las funciones de scheduler y dispatcher.

Estados y transiciones de un proceso

(cont. V)

(5) Finalizar

CAUSA: Llamada al sistema de finalizar un programa, bien un proceso propio, `fin_programa` o matar a otro proceso

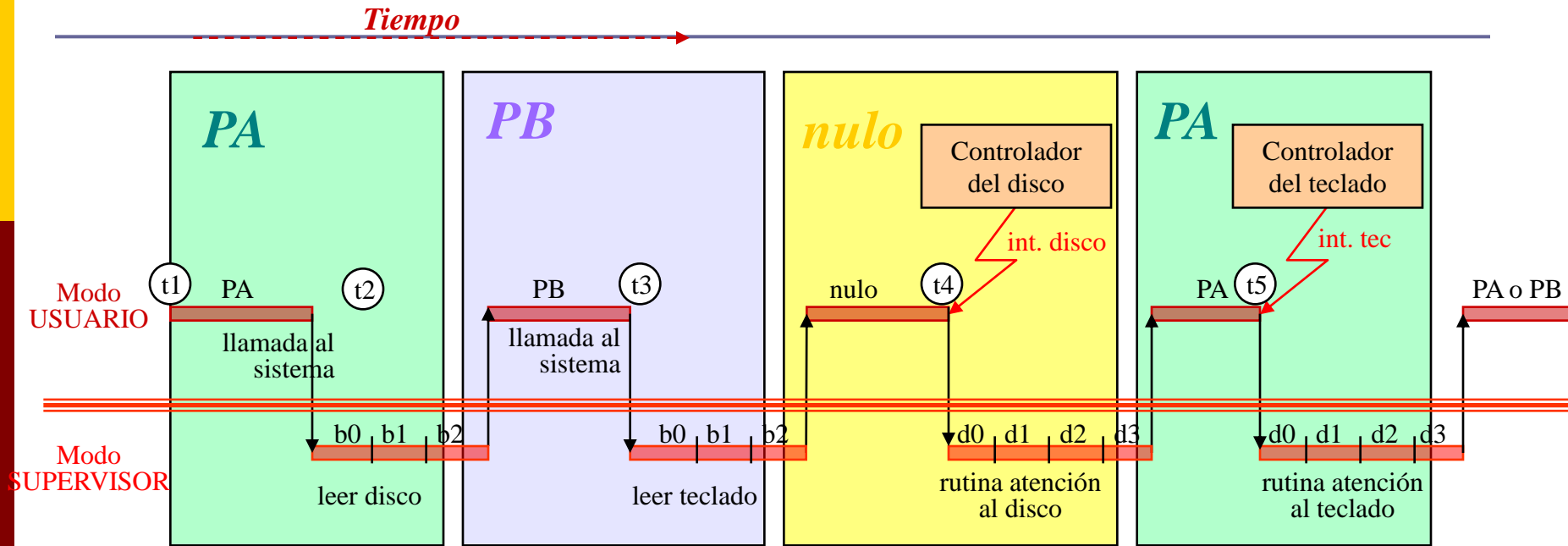
TAREAS: El S.O. debe liberar todos los recursos que utilizaba el proceso.

Pasar a estado finalizado.

Guardar el código de retorno para el padre.

Cuando el padre de un proceso realice esperar hijo, obtendrá el código de retorno de un hijo, eliminado la información del proceso hijo. Si tiene hijos y ninguno ha finalizado, el padre se bloqueará hasta que un hijo termine.

Ejemplo de Transición de Estado



Estado Inicial:
 PA : Ejecución
 PB, Nulo : Preparado

t2 PA realiza una llamada al sistema
 b0: Programar los controladores
 b1: Bloquear al proceso PA
 b2: scheduler y dispatcher


t3 PB realiza una llamada al sistema
 b0: Programar los controladores
 b1: Bloquear al proceso PB
 b2: scheduler y dispatcher

t4 Interrupción del Controlador de Disco
 Rutina de Atención:
 d0: Comprobar resultados
 d1: Desbloquear al proceso que solicitó la operación

Si política de EXPULSIÓN
 d2: Expulsar al proceso en ejecución (a Preparado)
 d3: scheduler y dispatcher

t5 Interrupción del Controlador del Teclado
 Rutina de Atención:
 (idem a t4)

Diagramas de Transición de Estado

	Ocurre	Ejecución	Preparado	Bloqueado	Finalizado
	... Leer(Disco)	PA	PB Nulo		
	... Leer(Tecl)	PB	Nulo	PA(Disco)	
	... Rut_Aten_Disco()	Nulo		PA(Disco) PB(Tecl)	
	... Rut_Aten_Tecl()	PA	Nulo	PB(Tecl)	
	...	PB	PA Nulo		

6.7 Llamadas para control de procesos

Tarea 6.1 Llamadas para la identificación de procesos.

Necesitamos construir una utilidad (**quiensoy**) que nos muestre la información relativa al proceso que se está ejecutando. En concreto queremos que nos muestre en la salida estandar:

- la identificación del proceso
 - la identificación del proceso padre
 - la identificación del usuario que ha creado el proceso
- a) Busca con la ayuda del man las funciones del sistema necesarias para identificación del proceso y del proceso padre. (5 min)
- b) Comenta/Discute su idoneidad y acuerda con ellos cuales serán las más idoneas (5 min)
- c) programa, compila y prueba la utilidad **quiensoy** (20 min)

Identificación de procesos

int getpid();

Devuelve el identificador (pid) del proceso

int getppid();

Devuelve el identificador (pid) del proceso padre

int getuid();

// int geteuid(); //ident. efectivo

Devuelve el identificador del usuario, “dueño” del proceso (uid)

int getgid();

Devuelve el identificador del grupo del proceso (uid)

Llamadas para control de procesos

Tarea 6.2 Llamadas para la creación de procesos.

Necesitamos construir una utilidad (**mihijo**) que cree un proceso hijo y a continuación tanto el proceso padre como el proceso hijo nos muestren su información de identificación. En concreto queremos que nos muestre en la salida estandar:

- Si se trata del proceso "PADRE" o el proceso "HIJO"
- la identificación del proceso
- la identificación del su proceso padre
- la identificación del usuario que ha creado el proceso

Además el proceso padre debe escribir el identificador del proceso hijo recién creado

- a) Busca con la ayuda del man las funciones del sistema necesarias para creación de un proceso. (10 min)
- b) Busca en internet ejemplos de código de creación de procesos en linux y comenta/discute/acuerda con tus compañeros de grupo su idoneidad (10 min)
- c) programa, compila y prueba la utilidad **mihijo**(15 min)
- d) Dibuja el diagrama de procesos de la solución (5 min)
- e) Modifica el programa inicial para que el proceso HIJO realice una espera de 1 segundo antes de finalizar. Compila y prueba la nueva utilidad. (10 min)
- f) Modifica el programa inicial para que el proceso HIJO realice una espera de 1 segundo antes de finalizar. Compila y prueba la nueva utilidad. Comenta con los compañeros los resultados de las dos adaptaciones. (10 min)

Llamadas para control de procesos

Tarea 6.3 Sincronización con un proceso hijo.

Necesitamos adaptar la utilidad previa (**mihijo**) y construir una nueva utilidad (**esperoamihijo**) para que el proceso padre, antes de finalizar, se espere a que termine el proceso hijo recién creado. El resto de la funcionalidad se debe mantener.

- a) Busca con la ayuda del man las funciones del sistema adicionales necesarias. (5 min)
- b) Comenta/discute/acuerda con tus compañeros de grupo su idoneidad y uso (10 min)
- c) programa, compila y prueba la utilidad **esperoamihijo** (15 min)
- d) Dibuja el diagrama de procesos de la solución (5 min)

Llamadas para control de procesos

Tarea 6.4 Ejecución de un programa en un nuevo proceso.

En el tema previo construimos una utilidad para realizar la carga de un programa que se le pasaba como argumento (tarea 5.4). Queremos adaptar esa utilidad utilizando la función de creación de procesos. Crea una nueva utilidad (***ejecutaprograma***) intentando reutilizar el código de los programas realizados en las tareas 5.4 y 6.3

- a) Busca con la ayuda del man las funciones del sistema adicionales relacionadas con la carga de programas. (10 min)
- b) Comenta/discute/ACUERDA con tus compañeros de grupo su idoneidad y uso (15 min)
- c) programa, compila y prueba la utilidad ***ejecutaprograma*** (25 min)
- d) Dibuja el diagrama de procesos de la solución (5 min)

Finalizar procesos

void exit(int estado);

Final del proceso “controlado”

Unix guarda el código de finalización (**estado**) hasta que el proceso padre ejecute **wait()**

int wait(int *estado);

Bloquea al proceso que lo llama hasta que uno de sus hijos finalice

Si no tiene hijos devuelve -1 sin bloquear al proceso que llama

Devuelve el identificador del proceso hijo finalizado

estado es el código de retorno devuelto por el proceso hijo

```
pid_t    wait    (int *status)
```

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

int kill(int pid, int SIGKILL);

Se manda la señal **SIGKILL** al proceso **pid** para que finalice

Llamadas para control de tiempos

- `void unsigned int sleep (unsigned int segundos);` (POSIX)
 - Unix bloquea el proceso durante ese tiempo o hasta que llegue una señal
- `void pause ();`
 - Unix bloquea el proceso hasta que llegue una señal
- `unsigned alarm (unsigned segundos);`
 - Unix manda la señal **SIGALARM** al proceso que ha hecho la llamada, pasados los **segundos**
- `int signal (int señal, void fun())` /* señales */
 - Relaciona la función **fun** con la señal **señal**
- `unsigned long time(0);`
 - /* el tiempo en unix desde el 1 de enero de 1970 */
- `char * ctime (unsigned long t_unix);`
 - /* función de librería para obtener la fecha en string */

Control de Tiempos

```
void esperar_tiempo (unsigned seg)
{
    signal (SIGALARM,fnula);
    alarm (seg);
    pause ();
}

void fnula()
{
    return;
}
```

```
void esperar_tiempo (unsigned seg)
{
    sleep (seg);
}
```

Otras llamadas al sistema

- `int getpid (); /* retorna el pid propio del proceso */`
- `int nice (int valor);`
 - Vale para modificar las prioridades de los procesos
 - Siempre es posible bajar la prioridad de los procesos, sin embargo, para subirla es necesario ser *root* o administrador

Llamadas para control de procesos y tiempos

Tarea 6.5 Creación de un programa que controle el tiempo de ejecución de un proceso hijo.

Queremos dotar a nuestro sistema de una utilidad (**EjecMaxTime**) que nos permita ejecutar programas pero controlando el tiempo de ejecución. Si el tiempo de ejecución del programa supera un tiempo límite proporcionado, queremos abortarlo. Como ayuda, puedes utilizar el programa reloj.

Esta tarea se realizará en grupos de 2 o tres alumnos.

- a) Define la interfaz de usuario de **EjecMaxTime** en un formato similar al *man* de linux (15 min)
- b) Realiza el diseño de la solución. Comienza por la descripción de los procesos necesarios, su relación y una descripción de la funcionalidad. (20 min)
- c) implementa, compila y prueba la utilidad **EjecMaxTime** (20 min)

Reloj.c

```
/* reloj.c */
#include "tiempo.h"
main (int argc, char *argv[])
{
    esperar_tiempo(atoi(argv[1]));
}
```

Intérprete de comando básico (*lanzador1*)

Tarea 6.6 Creación de un intérprete de comandos básico.

Nos gustaría crear nuestro propio intérprete de comandos. Para ello vamos a comenzar construyendo uno propio, pero con una funcionalidad limitada.

Esta primera versión que llamaremos **lanzador1** queremos que tenga únicamente la funcionalidad de ejecutar un programa en modo "RUN". Ejecutar un *programaX* en modo RUN quiere decir que una vez lanzado el *programaX*, el **lanzador1** debe esperar a que el *programaX* finalice antes de continuar con la tarea de lanzar otro programa. El **lanzador1** termina cuando se detecte fin de fichero en la entrada estándar.

El **lanzador1** debe proporcionar las siguientes características propias de los interpretes de comandos:

Cada vez que el **lanzador1** esté en condiciones de ejecutar un nuevo comando, debe mostrar un prompt al usuario. En este caso el prompt será "Lanzador1>"

El usuario, una vez que observa el prompt puede introducir un comando. Para ello escribirá en una sola línea el nombre de programa a ejecutar seguido de sus argumentos, todos ellos separados por al menos un blanco o tabulador.

Para realizar el análisis de la línea de comandos dispones de la función ya programada **ExtraeArgs()**. Esta función se encuentra en el fichero **extrae.h**

Para implementar **lanzador1.c** te recomendamos utilizar como base el programa *copia.c*.

Esta tarea se realizará en grupos de 2 o tres alumnos.

a) implementa, compila y prueba la utilidad **lanzador1** (30 min)

Intérprete de comando básico (*lanzador2*)

Tarea 6.7 Mejora del lanzador añadiendo el modo de ejecución RUN o SPAWN.

Vamos a mejorar nuestro lanzador1 proporcionándole la posibilidad de ejecutar un comando en modo "RUN" o en modo "SPAWN". Para ello vamos a modificar la sintaxis de la línea de comandos:

El usuario, una vez que observa el prompt puede introducir un comando. Para ello escribirá en una sola línea el modo de ejecución "R" o "S", el nombre de programa a ejecutar seguido de sus argumentos , todos ellos separados por al menos un blanco o tabulador

Ejecutar un *programaX* en modo RUN quiere decir que una vez lanzado el *programaX* , el **lanzador2** debe esperar a que el *programaX* finalice antes de continuar con la tarea de lanzar otro programa. Ejecutar un *programaX* en modo SPAWN quiere decir que el **lanzador2** puede pasar a realizar la siguiente tarea sin esperar a que finalice el *programaX*.

El **lanzador2** termina cuando se detecte fin de fichero en la entrada estándar o cuando el usuario introduzca el comando exit.

En este caso el prompt será "Lanzador2>"

Para implementar **lanzador2.c** te DEBES utilizar como base el programa *lanzador1.c*.

Ejemplo de ejecución de un usuario.:

```
Lanzador2>      R  cp  f1  f2
```

```
Lanzador2>      S  ls
```

Esta tarea se realizará en grupos de 2 o tres alumnos.

a) implementa, compila y prueba la utilidad **lanzador2** (20 min)

Intérprete de comando básico (*lanzador3*)

Tarea 6.8 Cambio del lanzador añadiendo que se muestre el pid de cada proceso hijo al término de su ejecución.

Vamos a cambiar nuestro lanzador2 haciendo que se muestre el pid de cada proceso hijo al término de su ejecución.

En este caso el prompt será "Lanzador3>"

Para implementar **lanzador3.c** te DEBES utilizar como base el programa *lanzador2.c*.

Esta tarea se realizará en grupos de 2 o tres alumnos.

a) implementa, compila utilidad **lanzador3** (15 min)

b) Prueba el correcto funcionamiento de la utilidad **lanzador3** al menos con la siguiente secuencia de comandos (10 min)

```
Lanzador3> R ls
```

```
Lanzador3> S pesado2
```

```
Lanzador3> S ls -al
```

```
Lanzador3> R pesado2
```


Intérprete de comando básico (*lanzador4*)

Tarea 6.9 Cambio del lanzador proporcionando un tiempo máximo de ejecución a cada comando.

Vamos a cambiar nuestro lanzador3 proporcionando un tiempo máximo de ejecución a cada comando. Si el programa rebasa el tiempo máximo, se abortará su ejecución y se escribirá el mensaje correspondiente. El formato de la línea de comandos en este caso será : R o S para indicar el modo de ejecución RUN o SPAWN, el tiempo máximo de ejecución del programa T= xx y a continuación el programa con todos los argumentos que necesite.

Por ejemplo.:

```
Lanzador4> R T= 3 cp f1 f2
```

```
Lanzador4> S T= 4 ls
```

En este caso el prompt será "Lanzador4>"

Para implementar **lanzador4.c** te DEBES utilizar como base el programa *lanzador3.c*.

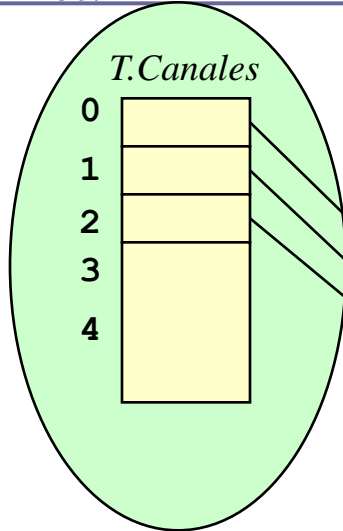
Debes probar este lanzador al menos con los programas pesado y pesado2.

Esta tarea se realizará en grupos de 2 o tres alumnos.

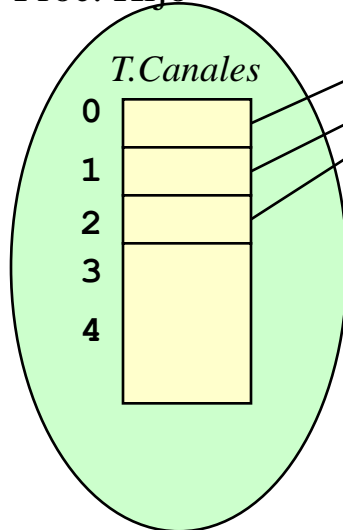
a) implementa, compila y prueba la utilidad **lanzador4** (30 min)

Mecanismo de herencia de la tabla de canales con `fork()`

Proc. A



Proc. Hijo



*Tabla de
Ficheros
Abiertos*

Modo
Índice
C-uso
Inode
Permisos
.....

*Tabla de
Inodes*

Tipo
Permisos
Propietario
Grupo
Fechas
Nºenlaces
Tamaño
C-uso
Bloques Datos
.....

Llamadas para control de procesos

Tarea 6.10 Creación de una función para ejecutar programas.

En la realización de la tarea previa habrás notado que linux no ofrece una función para ejecutar un programa. Sin embargo nos ayudaría de forma notoria el disponer de un función que realice la ejecución de programas. Otra característica que nos interesaría en esa función sería que nos permita realizar la "redirección" los canales estándar a otros ficheros o dispositivos.

Esta tarea se realizará en grupos de 2 o tres alumnos.

- a) Busca en la bibliografía o en internet ejemplos de cómo se realiza la redirección de los canales estandar. (10 min)
- b) Implementa la función `ejecutar_programa` (20 min)

`int ejecutar_programa (char *nom, char *c0, char *c1, char *c2, char *argv[]);`

Donde

- c) Pruebala en una nueva versión del lanzador1. (modo sin redirección) (10 min)
- d) Tarea OPCIONAL e INDIVIDUAL : (40 min)

Pruebala en una nueva versión del lanzador2. (modo con redirección)

LanzadorX> R ls -al > listado .dat

LanzadorX> R cat < listado .dat

LanzadorX> R cat < listado .dat > f2.dat

Diferencias con los sistemas monoprogramados

- **Gestión de Procesos**
 - Nuevas necesidades: sincronización, comunicación...
- **Gestión de Memoria :**
 - NECESIDAD de que el programa esté NO ENTERO
 - Enlace Dinámico
 - Si no queremos que la memoria se convierta en un problema insalvable a la hora de ejecutar programas concurrentemente.
 - Posibilidad de compartir código por varios procesos: código REENTRANTE.
- **Gestión de E/S :**
 - Recursos :
 - Compartibles (Disco)
 - Serialmente Utilizables (Impresora)

Diferencias con los sistemas monoprogramados (Cont.)

➤ **Llamadas al Sistema :**

- Control de procesos y tiempos
- Sincronización

➤ **Interprete de Comandos :**

- Posibilidad de lanzar tareas de fondo (SPAWN, BACKGROUND, EN PARALELO)
- Lugares de almacenamiento de información para comunicación de procesos (pipes, buzones,...) ls | more
- Posibilidad de lanzar un IC en paralelo para ejecutar ficheros de comandos.

6.8 Forma general de un sistema operativo

