Sistemas Operativos II

Laura M. Castro Souto

Segundo Cuatrimestre
Curso 2000/2001

Índice general

1. Procesos en Unix **7**
1.1. Introducción . . . 7
1.1.1. Historia . . . 7
1.2. Procesos . . . 8
1.2.1. Modos de ejecución . . . 8
1.2.2. Estados de un proceso . . . 9
1.3. Creación y terminación de procesos . . . 11
1.3.1. Contexto de un proceso . . . 11
1.3.2. Variables de entorno . . . 13
1.3.3. Credenciales de los procesos . . . 13
1.3.4. Información del sistema . . . 14
1.3.5. Llamadas, interrupciones, excepciones . . . 15
1.3.6. Acceso a los recursos . . . 16
1.4. Planificación . . . 18
1.5. Gestión de Procesos . . . 24
1.6. Señales . . . 25
1.6.1. Implementación de las señales . . . 28
1.6.2. Envío de señales . . . 29
1.7. Comunicación entre procesos . . . 29
1.7.1. Recursos IPC: Pipes . . . 29
1.7.2. Semáforos . . . 29
1.7.3. Memoria Compartida . . . 29
1.7.4. Colas de Mensajes . . . 30
1.8. Transparencias y ejemplos del tema . . . 30

A. Sistemas de Ficheros **31**
A.1. Introducción . . . 31
A.2. Contabilidad del espacio libre . . . 32
A.3. Métodos de asignación . . . 33
A.3.1. Asignación contigua . . . 33
A.3.2. Asignación enlazada . . . 33
A.3.3. Asignación mediante índices . . . 34
A.4. Métodos de acceso . . . 34
A.4.1. Operaciones sobre ficheros . . . 34
A.5. Sistema de directorios . . . 35
A.6. Sistemas de Ficheros al uso . . . 36

Capítulo 1

Procesos en Unix

1.1. Introducción

1.1.1. Historia

En 1965, los laboratorios BELL llegaron a un acuerdo con la General Electric Company y Project MAC del MIT para desarrollar un nuevo sistema operativo llamado **Multics**. Sus objetivos eran, entre otros, proveer de acceso simultáneo a una gran comunidad de usuarios, junto con amplia capacidad de computación y almacenamiento de datos, así como una fácil compartición de los mismos en caso deseado. Mucha de la gente que más tarde tomaría parte en los primeros desarrollos del sistema **Unix** participó en este proyecto. A pesar de que pronto una primitiva versión de Multics se encontró corriendo en un GE 645 (en 1969), estaba lejos de lo que pretendía ser, y no se veía claro si se llegaría a alcanzar, de suerte que los laboratorios Bell abandonaron su participación en el proyecto.

En un intento de desarrollar su propio entorno de programación, miembros del Computer Science Research Center de Bell esbozaron un sistema de ficheros que más tarde evolucionaría a una de las primeras versiones del sistema de ficheros Unix. Eran Ken Thompson y Dennis Ritchie, entre otros. Implementaron su sistema en un PDP-7, incluyendo dicho sistema de ficheros, el subsistema de proceso y un pequeño conjunto de aplicaciones. El nuevo sistema se llamó **Unix** en contraste con *Multics*, apoyado por otro miembro del mismo departamento, Brian Kernigham.

A pesar de que esta primera versión de Unix parecía prometer, no se podría intuir su potencial hasta que se probase en un proyecto real. De este modo, proveyendo de un sistema de procesado de texto, fue instalado en un PDP-11 en 1971, en los propios laboratorios Bell. El sistema se caracterizaba por su reducido tamaño: 16 Kb para el sistema, 8 Kb para programas, un disco de 512 Kb y un límite de 64 Kb por fichero. Tras su primer éxito, Thompson decidió implementar un compilador Fortran, pero en lugar de ello creó el lenguaje B, influenciado por BCPL.

B era un lenguaje interpretado, así que Ritchie lo desarrolló hasta crear C, que permitía generación de código máquina, declaración de tipos y estructuras,... En 1973, el sistema operativo fue reescrito en C, algo inaudito en aquel entonces, pero que tendría una gran repercusión en la aceptación del mismo entre los usuarios. El número de instalaciones en Bell creció a 25 y se formó un Unix Systems Group para proveer soporte interno.

Al mismo tiempo, AT&T no podía comercializar productos informáticos debido a un

decreto firmado con el gobierno federal en 1956. No obstante, distribuyó el sistema Unix a universidades que lo adquirieron por motivos académicos. AT&T nunca lo comercializó ni soportó, pero aun así su popularidad creció rápidamente. En 1974, Thompson y Ritchie publicaron un artículo describiendo el sistema Unix en las Communications del ACM, dando un empuje aún mayor a su aceptación. En 1977, el número de estaciones con un sistema Unix ascendía a 500, de las cuales 125 estaban en universidades. Los sistemas Unix se popularizaron en las compañías telefónicas, proporcionando un buen entorno para el desarrollo de programas, servicios de red y en tiempo real. Se proporcionaron licencias tanto a universidades como a instituciones comerciales. El año 1977 fue también el año en que Unix se incorporó a un entorno de oficina (en Interactive Systems Corporation) y fue portado a una máquina no PDP, la Interdata 8/32.

Dada la creciente popularidad de los microprocesadores, otras compañías portaron Unix a nuevas máquinas, ya que su simplicidad y claridad invitaban a los desarrolladores a mejorarlo a su manera, dando lugar a diversas variantes del sistema base. Entre 1977 y 1982, los laboratorios Bell combinaron variantes de AT&T en un único sistema, que se conocería comercialmente como UNIX System III. Más tarde Bell le añadiría algunas características nuevas, dando lugar a UNIX System V. AT&T anunció soporte oficial para System V en enero de 1983. Paralelamente, gente de Berckely, en la Universidad de California, había desarrollado otra variante del sistema Unix, cuya más reciente versión se llamó 4.3 BSD, para máquinas VAX.

A principios de 1984, había sobre 100.000 instalaciones Unix en el mundo, corriendo en máquinas de un amplio rango de poder computacional, desde microprocesadores hasta mainframes. Ningún otro sistema operativo había logrado algo así.

1.2. Procesos

1.2.1. Modos de ejecución

En Unix hay **2 modos de ejecución**:

Modo usuario Un proceso que se ejecuta en este modo sólo puede acceder al *código*, *datos* y *pila* de usuario (**espacio de usuario**).

Modo kernel Un proceso ejecutándose en este modo puede acceder al espacio de usuario y también al **espacio del kernel** o **espacio del sistema**.

Son muchas las arquitecturas que proporcionan más de un modo de ejecución (Intel y AMD, por ejemplo, proporcionan 4 niveles o *anillos*).

Todos los procesos *ven* al kernel en el mismo lugar del espacio de direcciones virtuales.

Sólo hay tres ocasiones o maneras de que un proceso, que se ejecutará normalmente en modo usuario, pase a ejecutarse en modo kernel:

i) Al hacer una **llamada al sistema** (con una secuencia de pasos bien definida).

ii) Al producirse una **excepción**.

iii) Al ocurrir una **interrupción**.

Pregunta de examen.- Un proceso del root se ejecuta en modo kernel...

a) siempre.

b) nunca.

c) depende de la prioridad.

d) no existen procesos del root.

e) ninguna de las anteriores.

Respuesta.- La correcta es la *e*, ya que es como un proceso de cualquier otro usuario.

El **kernel** es la única parte del sistema operativo que interactúa con el hardware. Los procesos de usuario interactúan con el hardware a través del kernel, mediante una interfaz claramente definida: la interfaz de **llamadas al sistema**.

El kernel reside permanentemente en memoria, sus páginas de código no se intercambian (las de los procesos de usuario, obviamente, sí). Sus funciones son el control de ejecución de los procesos, la planificación de la CPU, la política de recursos y dispositivos (asignación, gestión, liberación de memoria, espacio en disco, impresora...), etc. Para ello mantiene una serie de estructuras tanto globales (del sistema) como específicas de cada proceso. Por ejemplo, mantiene una *tabla de ficheros abiertos* en el espacio del sistema y una *tabla de ficheros abiertos por el proceso* en el espacio de usuario de cada proceso.

En Unix, esta parte del espacio de direcciones del usuario que contiene información específica del proceso que el kernel necesita, se denomina **u_area**, y más adelante seguiremos hablando de ella.

Unix también soporta **multithread**, esto es, la posibilidad de que un proceso se esté ejecutando en varias partes a la vez, o bien que haya varios procesos ejecutándose en el mismo espacio de direcciones (diferencia con varios procesos comunes ejecutándose a la vez —*multitarea*—).

1.2.2. Estados de un proceso

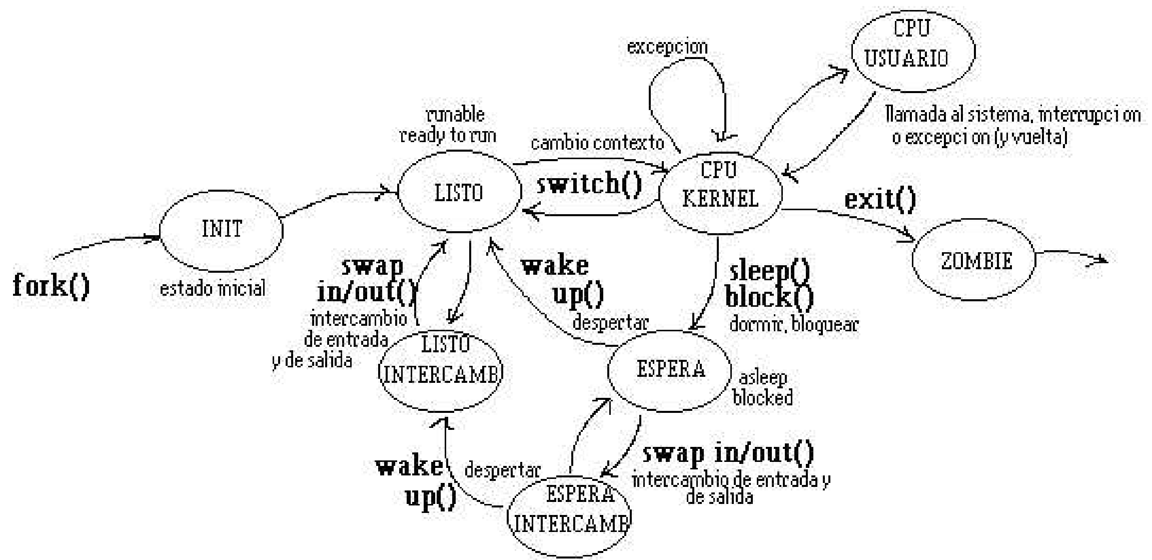
A lo largo de su *vida*, un proceso puede pasar por distintas etapas, que se ilustran muy bien el el gráfico 1.1 (página 10).

Para que un proceso pase a espera ha de ser necesariamente desde ejecución en modo kernel, pues si tiene que esperar será que solicitó algo, esto es, realizó la pertinente llamada al sistema.

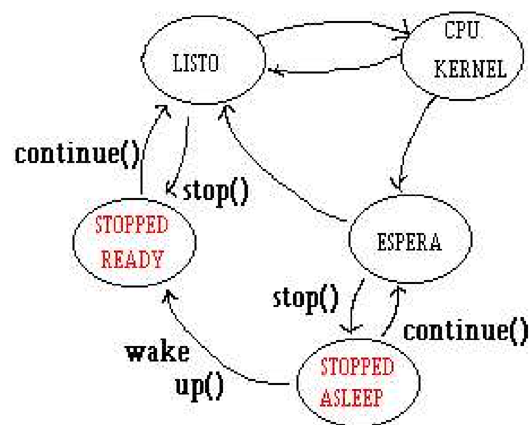
Asimismo, cuando un proceso que se está ejecutando pasa a "listo" también lo hará desde ejecución en modo kernel, ya que será debido a una interrupción (con gran probabilidad, el tic de reloj).

Además, desde System V R3 se añade un nuevo estado, **stopped**, que además de ser útil en depuradores, permite ejecutar varios procesos en una sola sesión.

Figura 1.1:



Grafo de transición de estados para los procesos en Unix.



1.3. Creación y terminación de procesos

La llamada al sistema en Unix para crear un proceso es `fork()`; en BSD es `vfork()`, que no es sino una variante (optimización) de la anterior. Ambas existen en System V, aunque la segunda no es más que un alias de la primera.

Todo proceso en Unix termina con la llamada al sistema `exit()`, que permite al proceso hijo devolver un valor al padre. Dicho valor permanece en la tabla del sistema hasta que es recogido por el padre. El estado en que queda el proceso hijo ya terminado hasta que el padre libera ese espacio en la tabla es lo que hemos llamado **zombie**. Ese valor se libera con la llamada `wait()`.

Tanto las llamadas al sistema que crean como las que terminan los procesos fuerzan al operativo a llevar a cabo una serie de tareas, como la reserva de espacio, su posterior liberación, el cerrado de ficheros, etc.
Ahora bien, si en principio la llamada `fork()` simplemente crea un *proceso idéntico* al que se estaba ejecutando, ¿cómo podemos ejecutar diferentes programas? La respuesta es: mediante la llamada `exec()`, que posibilita que un proceso ya creado ejecute otro proceso distinto.

1.3.1. Contexto de un proceso

En líneas generales, llamaremos **contexto de un proceso** a todo aquello que cambia cuando de estar ejecutándose un proceso se pasa a ejecutar otro.

Está formado típicamente por:

Espacio de direcciones Contiene código, datos, pila, regiones de asignación dinámica, librerías,...

Información de control Se agrupa en dos estructuras:

- **estructura proc** ubicada en el espacio del sistema
- **u_area** ubicada en el espacio del proceso, conteniendo la pila del kernel y mapas de direcciones

Variables de entorno

Credenciales

Contexto hardware Valor de los registros del micro.

Cada proceso tiene su *espacio de direcciones* ***virtual*** (ver figura 1.2, página 12).
En el espacio del kernel (espacio virtual donde los procesos lo ven), se guarda la **tabla de procesos** (*process table*, ver figura 1.3 en página 12)
Esta tabla contiene información sobre cada uno de los procesos que hay en el sistema; es en realidad un **array de estructuras proc**. Otra parte importante del espacio de usuario de un proceso es su **u_area**. Todos los procesos ven su **u_area** en la misma posición (dirección) del espacio virtual[1].

[1]De este modo el acceso a la u_area de un proceso es inmediata dentro del sistema operativo.

Figura 1.2:

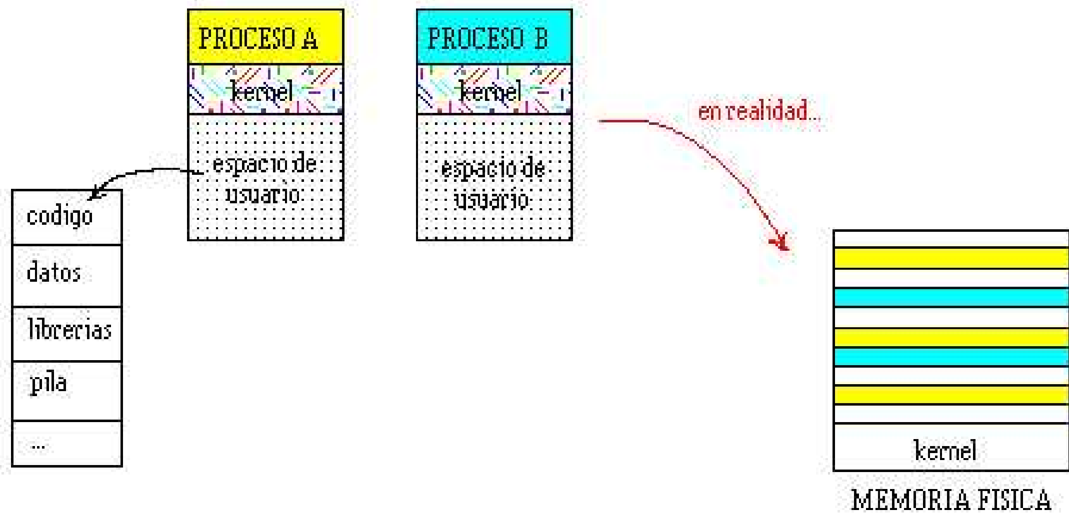
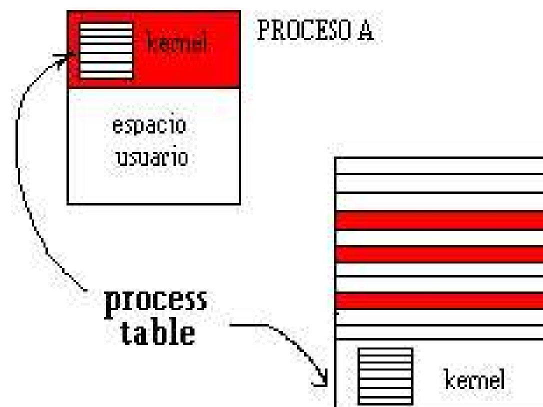


Figura 1.3:



Código y datos del kernel son compartidos por los procesos (por ello el kernel de Unix se denomina **reentrante**), esto es, varios procesos pueden estar ejecutando diferentes tareas (lecturas, escrituras... en general llamadas al sistema) en modo kernel. Debido a ello se requiere:

- protección de **acceso concurrente** a los datos del kernel
- su propia **pila del kernel** para cada proceso (sino un proceso no podría llamar al sistema a menos que todos hubiesen acabado ya sus respectivas llamadas)[2]

La pila del kernel de cada proceso está en su espacio de usuario pero no puede ser accedida en modo usuario.

1.3.2. Variables de entorno

Las **variables de entorno** son cadenas de la forma "`NOMBRE=valor`" que se usan para pasar información del entorno a los procesos.

Las variables de entorno son parte del espacio de direcciones de un proceso.

En el cshell, el comando

```
# setenv DISPLAY = . . .
```

cambia una variable de entorno del shell, claro que como los procesos son *hijos* del shell, la *heredarán*.

Un proceso sólo puede cambiar **sus** variables de entorno, aunque se pueden crear nuevas variables y ponerlas en memoria compartida. Las llamadas que permiten consultar y cambiar los valores de estas variables son `getenv()` y `putenv()`.

1.3.3. Credenciales de los procesos

Todos estamos familiarizados con los permisos que tienen los ficheros; para saber cuándo se han de aplicar, lo que hace el sistema es lo siguiente:

1. Si la credencial del usuario coincide con la credencial del propietario, se aplican los permisos de propietario.
2. Si no, si coinciden las credenciales de grupo, se aplican los permisos de grupo.
3. Si no, se aplica el tercer grupo de permisos.

Las credenciales de usuario y de grupo se guardan en el fichero `/etc/passwd`, que tiene una entrada por usuario:

```
infzzz00:XXXX:UID:GID:nombre_completo:
         directorio_inicio:programa_que_ejecuta_al_entrar
```

El directorio de inicio suele ser el *home* y el programa que se ejecuta al entrar, un *shell*.

En realidad no se tienen sólo un par de credenciales (usuario, grupo), sino que se tienen un par de pares: **(real,efectiva)::(usuario,grupo)**.

[2]Pregunta de examen.

Las **credenciales efectivas** son las que se usan para regular el acceso a los ficheros. Las **credenciales reales**, para determinar de quién puede un usuario recibir señales. Un proceso puede enviar señales a otro si la credencial real (o efectiva) del que *envía* es igual a la real del que *recibe*.

Las llamadas que cambian las credenciales son `setuid()` y `setgid()`, aunque con `exec()` también puede hacerse. El conjunto de llamadas disponibles son:

`setuid()` (cambia la credencial real o la efectiva dependiendo de quién la invoque)

`seteuid()`

`setruid()`

`setgid()`

`setegid()`

`setrgid()`

`exec()`

Un usuario normal no puede ponerse una credencial distinta de la suya, sólo puede volver a ponerse la propia; el *root* sí puede, en cambio, hacerlo. De hecho, el proceso `login` es un proceso del root que al entrar se pone la *uid* y la *gid* del usuario que entra.

Procesos como estos pueden ocurrir cuando el bit que marca los permisos de ejecución tiene una `s`:

```
rwx__x__x - > rws__x__x
              rwx__s__x
```

En el primer caso el fichero podría ejecutarlo cualquiera (por los permisos), y quien lo hiciese adoptaría la credencial efectiva del propietario del ejecutable. En el segundo caso, el fichero lo ejecutaría también cualquiera (por los permisos), y quien lo hiciese adoptaría la credencial efectiva de grupo del grupo del propietario del fichero.

No se permiten "ficheros setuid" en dispositivos ni por red.

1.3.4. Información del sistema

En la asignatura de Sistemas Operativos I estudiamos que la información de control, de mapas, de páginas... el sistema la guardaba en una estructura que contenía todo este tipo de datos sobre un proceso, que denominábamos **PCB**. En Unix, el PCB se "reparte" en dos estructuras:

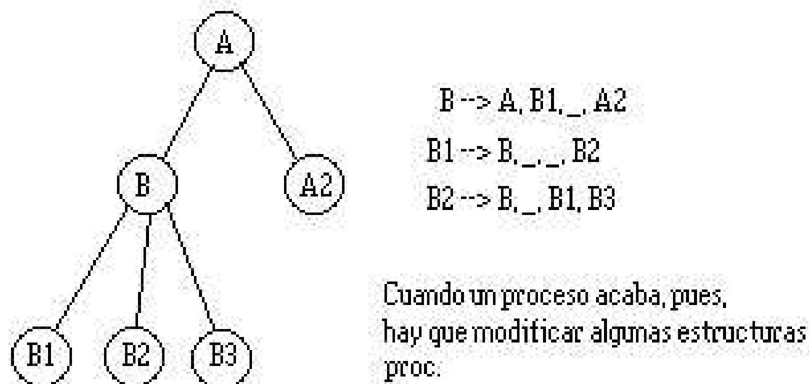
- **u_area**
- **proc**

Esta división está motivada porque hay información de los procesos que el sistema necesita siempre y otra que sólo requiere cuando están en ejecución. Por tanto, es lógico que la primera se guarde en el espacio del kernel (siendo, pues, visible para todos) y la restante en el espacio de usuario.

La tabla de procesos, de la que ya hemos hablado, es, como dijimos, un array de estructuras proc con una estructura por proceso. ¿Qué puede ocurrir si dicho array se llena? En sistemas en los que esta estructura es estática (como BSD o System V R3), si esto sucede simplemente ya no se pueden crear más procesos. En sistemas con asignación dinámica para las tablas del sistema (como System V R4), esta situación no podría darse.

En los sistemas Unix los procesos están relacionados mediante una estructura jerárquica (en forma de árbol) gracias a la llamada `fork()`. Como podemos ver en las transparencias del tema, no se guarda información de todos los hijos de un proceso (¡por eficiencia! no podemos saber cuántos hijos tendrá un proceso).

Figura 1.4:



1.3.5. Llamadas, interrupciones, excepciones

Como ya hemos mencionado con anterioridad, **llamadas al sistema**, **excepciones** e **interrupciones** son las únicas maneras de pasar de modo de ejecución usuario a modo kernel.

Llamadas al sistema

Las *llamadas al sistema* se ejecutan en modo kernel con credenciales (contexto) del proceso que hace la llamada.

Las funciones que usamos normalmente para hacer las llamadas al sistema se denominan **funciones envoltorio** (*wrapping functions*): `open`, `read`,... El modus operandi de estas funciones es común a todas ellas:

♭ ponen un número que representa la función en cuestión en la pila[3]

♭ ejecutan una instrucción especial dependiente de la arquitectura[4] que cambia el modo de ejecución a modo kernel y transfiere el control a una rutina del kernel que suele denominarse `syscall`. `Syscall` copia los parámetros de la pila de usuario en la **u_area** (sabe el número de parámetros por el número de intrucción). Mediante el número de función accede a un array indexado por número de servicio y obtiene la dirección de la función del kernel que realiza el servicio deseado (dicho array se denomina `sysent[]`).

Interrupciones

Las *interrupciones* no se ejecutan para un proceso particular, se ejecutan para el sistema, para un contexto del sistema; son siempre algo ajeno al proceso, algo externo (tic de reloj, final de una lectura en disco…).

Las interrupciones más comunes son las de dispositivo (ya que hay muchos dispositivos: teclado, disco, cdrom, tarjeta de red, de sonido,…y por tanto muchos tipos de interrupciones). Llegan en cualquier momento, cuando uno de los dispositivos reclama atención. ¿Qué ocurre entonces? Una interrupción siempre reclama atención prioritaria. Podemos observar el diagrama de las transparencias.

Aunque en general se intenta atender las interrupciones en el momento en que llegan, no siempre es posible, ya que puede coincidir que llegue una en el instante en que el sistema está tratando otra. Para discernir qué hacer en un caso así se asignan prioridades a las interrupciones, mediante el **ipl** (*interruption priority level*), y se trata siempre a la que corresponda siguiendo este criterio.

Excepciones

Las *excepciones*, ya por último, son provocadas por el propio proceso, por algo derivado de su funcionamiento interno (intento de división por cero, intento de acceso a una zona de memoria ajena, intento de ejecución de instrucciones inexistentes…). Para tratarlas, se pasa a una ejecución en modo kernel, pero con las credenciales del proceso causante.

1.3.6. Acceso a los recursos

Se mencionó con anterioridad la necesidad de proteger los datos y el código del kernel, ya que son compartidos por todos los procesos. La protección del código reside en que es “de sólo lectura”, pero los datos no se protegen con semáforos, ni monitores, sino simplemente con un **flag**, y algún otro tipo de recursos compartidos del sistema ni con eso. ¿Cómo puede ser entonces que todo funcione bien? La razón es sencilla: porque un proceso que se ejecuta en modo kernel *no es apropiable*, no se le puede apropiar la CPU.

Pero, si el modo kernel no es apropiable, ¿se puede decir en realidad que el kernel de Unix es reentrante? ¿Pueden de veras dos procesos ejecutar código del kernel a la vez? La respuesta es afirmativa, ya que un proceso ejecutándose en modo kernel puede, por

[3] Consultando la página de manual de `syscall(s)` y los *includes* asociados, podemos conocer dichos números.

[4] `trap` en MOTOROLA, `int` en INTEL, `ta` en SPARC…son interrupciones denominadas *interrupciones software*.

ejemplo, pasar a espera. En un caso así es el propio proceso el que abandona la CPU, no se le expropia. La estrategia consiste en que el proceso, antes de pasar a espera, *marca* el/los recurso(s) que estuviera usando como ocupado(s) con un simple flag, para prevenir al sistema de estados inconsistentes. Cualquier proceso ejecutándose en modo kernel, antes de acceder a un recurso cualquiera, comprobará el estado del mismo.

Cabe comentar que esto es válido en sistemas Unix tradicionales (esto es, sistemas monoprocesador y sin threads), ya que si, por ejemplo, tuviésemos más de un procesador, podrían dos procesos ejecutarse en modo kernel y acceder libremente a los recursos.

La función que pone un proceso en espera es la función del kernel `sleep()`, que llama a su vez al cambio de contexto para que se pueda ejecutar otro proceso que esté listo.

Un problema con que nos podemos encontrar en este ámbito es el conocido como **problema de la inversión de prioridades**, que consiste en que un proceso con poca prioridad que continuamente está ejecutando llamadas al sistema (y por tanto ejecutándose en modo kernel, no apropiativo) puede mantener en espera a otros procesos de mayor prioridad. Para solucinar esto se establecen en el código del kernel *puntos de apropiación*. Algunas arquitecturas, como Solaris, optan por permitir la apropiación en cualquier momento y utilizar estructuras protegidas por semáforos.

```
while (recurso ocupado)
  sleep(hasta que el recurso quede libre);
marcar recurso como ocupado;
usar recurso;
. . .
marcar recurso como libre;
despertar procesos en espera por el recurso;
```

Cuadro 1.1: Algoritmo de acceso a un recurso.

Para evitar que los procesos queden en espera permanentemente tras solicitar un recurso que resulta estar ocupado, aquél que lo tenía en propiedad ha de despertarlos[5] además de liberarlo. El proceso puede saber que hay otros esperando mediante un flag al efecto, por ejemplo, y sabe cuáles de los procesos en espera esperaban por el recurso que él acaba de liberar gracias al campo *canal de espera* de la estructura proc correspondiente a cada uno de ellos. El canal de espera está en la estructura proc y no en la u_area porque el proceso en ejecución cuando se libera un recurso es el proceso que lo libera, que es quien tiene que despertar a los demás. En ese momento, sólo es accesible su u_area propia, y las estructuras proc de todos los demás[6].

El fallo de espera limitada es teóricamente posible en el algoritmo que acabamos de ver, pero en la práctica no se da porque los recursos suelen permanecer ocupados un breve período de tiempo.

[5]Despertar es pasar de espera a listo.
[6]Pregunta de examen.

1.4. Planificación

En Unix la CPU es compartida por todos los procesos mediante una **planificación por prioridades apropiativa**, donde:

- las prioridades son **dinámicas**, evolucionan con el tiempo, se recalculan: se disminuye la prioridad de los procesos en CPU y se aumenta la de los procesos que están en espera[7]
- entre procesos de igual prioridad se planifica por **round robin**[8]

En Unix, pues, el proceso de mayor prioridad se ejecuta siempre, salvo que el que esté en CPU esté en modo kernel, que no se le puede apropiar. La prioridad se almacena, lógicamente, en la estructura proc, puesto que es necesario que esté disponible en todo momento. En esta estructura hay 4 campos relacionados con la prioridad: `p_usrpri` (prioridad en modo usuario), `p_pri` (prioridad del proceso; cuando se ejecuta en modo usuario debe ser igual a la anterior), `p_cpu` (medida del tiempo de CPU que lleva el proceso) y `p_nice` (factor "nice").

Un proceso en modo kernel que pasa a espera, vuelve de la espera con un valor de prioridad especial que depende del motivo por el que pasó a tal estado. Este valor especial se denomina **kernel priority** o **sleep priority**, y es siempre mayor que cualquier prioridad en modo usuario[9], con lo que se consigue que un proceso que venga de una espera pase siempre antes a CPU que cualquier otro en modo usuario.

Para calcular la prioridad de los procesos el sistema analiza los campos `p_pri`, `p_cpu` y `p_nice`; un número menor indica mayor prioridad. El uso de CPU (`p_cpu`) se actualiza con cada tic de reloj, las prioridades se recalculan cada segundo de acuerdo con:

```
System V Release 2
------------------

p_usrpri = PUSER + p_cpu / 2 + p_nice
p_cpu = p_cpu / 2


BSD
---

p_usrpri = PUSER + p_cpu / 4 + 2 * p_nice
p_cpu = ( 2 * carga_media / (2 * carga_media + 1) ) * p_cpu
```

Cuadro 1.2: Fórmulas para el recálculo de prioridades en sistemas Unix.

[7]Son necesarias ambas cosas, ya que si sólo se actuase en un sentido llegaría un momento en que todos los procesos tendrían la mínima/máxima prioridad.

[8]Se consideran prioridades iguales con una diferencia de 4 unidades; el *quanto* suele ser de 100 ms.

[9]Las únicas prioridades que se recalculan son las prioridades en modo usuario.

PUSER es una *prioridad base* que se suma para que la prioridad de cualquier proceso sea siempre mayor que las prioridades en modo kernel[10].

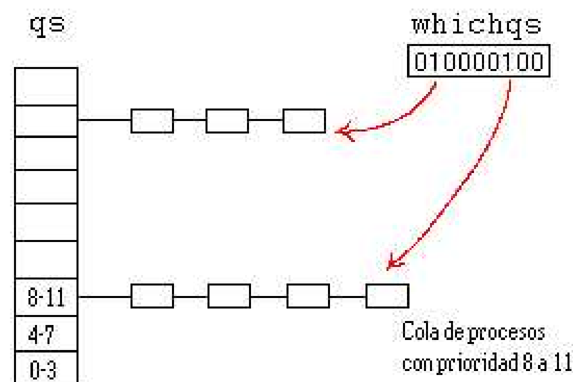
El rango del campo p_nice va entre 0 y 40, por defecto vale 20. Es el único sobre el que nosotros podemos actuar, mediante la llamada al sistema `$ nice <valor>`, a la que se le pasa el incremento (<valor>) que se le quiere dar sobre el actual y devuelve el exceso sobre 20. El root es el único que puede invocarla con incrementos negativos, pero tampoco puede hacer que la prioridad de un proceso llegue a ser menor que la de un proceso que viene de espera (prioridad kernel), debido al factor PUSER que interviene en los cálculos.

Repasemos, pues, el funcionamiento de la planificación: hemos dicho que el proceso de mayor prioridad se ejecuta siempre, salvo que el que esté en CPU esté en modo kernel, ya que en ese caso no se le puede apropiar. Si un proceso se está ejecutando en modo usuario y pasa a listo uno que estaba en espera, le apropiará siempre la CPU. Si está en ejecución un proceso en modo kernel y ocurre lo anterior, puesto que el modo kernel no es apropiable, el cambio de contexto se hará cuando vaya a pasar a modo usuario, momento en el que se comprobará la presencia de procesos listos mediante el chequeo de un *flag*, el flag runrun. Si dicho flag está puesto, se llamará a la función de cambio de contexto swtch, que buscará en el array de estructuras proc del sistema el proceso listo de mayor prioridad (según el motivo por el que se fue a espera).

Cuando un proceso está en modo kernel y llega una interrupción, el que se atienda o no (aun en modo kernel) depende del *ipl* (normalmente los procesos suelen tener ipl=0). Si dicha interrupción saca algún proceso de la espera (por ser, por ejemplo, una interrupción causada por el fin de una lectura de disco. . .) no se apropiará la CPU, pero se marcará el flag *runrun*. El tiempo de servicio de una interrupción se carga siempre en el quanto del proceso en CPU al que interrumpió.

La implementación de este mecanismo se basa en colas[11]:

Figura 1.5:



El sistema mantiene un array de colas (por ejemplo 32) al que suele llamar qs y un entero de 32 bits denominado whichqs donde un bit a 1 indica que la correspondiente cola *no* está vacía. Cada cola agrupa a los procesos con una determinada prioridad.

[10]Ver transparencias y un ejemplo de funcionamiento del mecanismo de recálculo de prioridades al final del tema.

[11]Es así, en concreto, en 4.3. BSD.

Los procesos que cambian de prioridad son movidos de una cola a otra. El cambio de contexto es, gracias a estos elementos, sencillo de realizar: se busca el primer bit no nulo de `whichqs` y se elige el primer proceso de la cola correspondiente. Si en la cola solo hay un proceso, se ejecutará él hasta que se recalculen prioridades y resulte cambiado de cola o por causa de una interrupción aparezca otro proceso en una cola inferior (**cambio de contexto involuntario**[12]). Cuando hay varios, una rutina del sistema llamada `RoundRobin()` se encarga de cambiarlos entre sí cada 100 ms. Otra rutina también del sistema, `schedcpu()`, es la que recalcula las prioridades de todos y reorganiza las colas cada segundo.

Este modelo de planificación, que es el usado por sistemas como SVR2, SVR3 y BSD, tiene pese a todo una serie de inconvenientes, que ya hemos dejado entrever; sobre todo, que sólo con `nice` podemos influir en la prioridad de los procesos. Llamadas como `getpriority(what, which)`, `setpriority(what, which, value)` en realidad actúan sobre dicho campo y no sobre la prioridad total. Hacerlo es posible en FreeBSD (variante de 4.4 BSD), donde aparecen *dos nuevos tipos de procesos*: procesos **realtime** y procesos **idle**, y la llamada `rtprio()`, que modifica su prioridad absoluta.

- Los procesos **realtime** son procesos que se ejecutan en *tiempo real*, siempre antes que cualquier otro. Tienen siempre la misma prioridad, cuyo valor puede además ser menor que una prioridad kernel, por lo que solo "compiten" entre ellos.

- Los procesos **idle** (*ociosos*) se ejecutan si no hay ningún otro proceso en el sistema, tienen la prioridad más baja de todos, de modo que también compiten sólo entre sí.

Se añaden, pues, en FreeBSD (y NetBSD) dos extremos, además de los procesos normales y su comportamiento, que venimos estudiando a lo largo de toda esta sección.

Para controlar la prioridad de estos nuevos tipos de procesos se tiene la llamada `rtprio(function, process_pid, struct rtprio *p)`, donde el parámetro `function` puede ser `RTP_LOOKUP` o `RTP_SET`. La estructura `rtprio` consta de dos enteros cortos (`u_short`), uno que indica si el proceso es *realtime* o *idle* y otro que guarda el valor de la prioridad.

Este esquema es el que usa y sigue utilizando BSD; aunque intenta solucionar la "cojera" del `nice`, es poca flexibilidad la que añade. Además, hemos dicho que se recalculan las prioridades de todos los procesos (en modo usuario) cada segundo, lo cual puede ser muy lento si en el sistema hay muchos procesos, y lo que es más grave, no garantiza la *latencia* de una aplicación (el tiempo que hay que esperar desde que está lista para ejecutarse hasta que realmente se ejecuta).

Planificación en SVR4

La planificación en System V Release 4 es un poco más compleja de lo que hemos visto hasta ahora. Se habla de *clases* de procesos (lo que permite añadir más). Existen una serie de rutinas que dependen de la clase a la que pertenezca un determinado proceso y otras

[12] El cambio de contexto, involuntario o no, se realiza siempre en modo kernel mediante la ejecución de la función `swtch()`, que es llamada desde rutinas como `sleep()` o `exit()`.

que no. Por todo ello es, como veremos, mucho más flexible; se separa la planificación en sí de los mecanismos con los que se hace, permitiendo que sean totalmente definibles.

Las rutinas comunes a todas las clases son las de manipulación de colas, cambio de contexto y apropiación. En cambio, las funciones de recálculo de prioridades y herencia son dependientes de la clase a la que pertenezcan los procesos.

En SVR4 hay dos *clases predefinidas*: **clase de tiempo real** (***RealTime***) y **clase de tiempo compartido** (***TimeShared***). En la estructura proc de cada proceso se tiene:

`p_cid`, un identificador de la clase a la que pertenece el proceso

`p_clfuncs[]`, un array de punteros a las funciones que implementan los recálculos de prioridades para esa clase

`p_cldata`, un puntero a datos necesarios para el recálculo de prioridades de esa clase

En este sistema números pequeños son prioridades más pequeñas (a la inversa de como veníamos viendo hasta ahora):

Rango	*Clase de procesos*
0-59	Prioridades en tiempo compartido (TS)
60-99	Prioridades kernel
100-159	Prioridades en tiempo real (RT)

Cuadro 1.3: Prioridades en System V Release 4.

Con la aparición de los procesos RT ya no se da el problema de inversión de prioridades. Además, las prioridades de los procesos TS no se recalculan todas, sino sólo las de los que están en CPU.

La implementación de los mecanismos es la misma (colas de procesos ordenadas por prioridad y un entero cuyos bits marcan el estado de cada una).

¿Qué ocurre si aparece un RT con mayor prioridad y hay uno en modo kernel ejecutándose? Existen unos **puntos de apropiación** (***preemption points***) en el código de las llamadas al sistema donde el *estado del kernel es consistente*, y en los que el proceso se detiene y comprueba si hay RTs esperando (listos). En caso afirmativo, abandona la CPU y se ejecuta el RT que corresponda. La existencia de un proceso listo de mayor prioridad se conoce por un flag, el flag `kprunrun`.

Cuando un proceso se crea, hereda la clase de su proceso padre (se heredan prioridades,...). Las interrupciones se atienden, durante la ejecución de un proceso RT, dependiendo como siempre del *ipl* del proceso.

Las prioridades de los procesos RT son *estáticas*. Se pueden cambiar mediante una llamada al sistema, pero no se recalculan, por eso se ejecutan siempre que estén listos y no haya otro RT de prioridad más grande. Un RT que vuelve de una espera no tiene una prioridad kernel, mantiene *su* prioridad. Si hay varios RT con igual prioridad, se ejecutan en round robin. Su quanto, que depende de su prioridad, también es fijo, aunque se puede cambiar asimismo mediante una llamada al sistema. Los procesos RT sólo son accesibles al superusuario.

Los procesos TS son los procesos "normales". Su quanto depende también de la prioridad (a mayor prioridad, menor quanto y viceversa, ya que un proceso con menor prioridad es más difícil que obtenga la CPU, de modo que cuando se le otorga, se le da más tiempo), prioridad que *sí* se recalcula dinámicamente, pero no todas, sino sólo la del proceso en CPU (lo cual es una ventaja con respecto a SVR2, SVR3 y BSD, que recalculaban la de todos) de la siguiente manera:

♭ si el proceso llega a agotar su quanto, se le disminuye la prioridad (pero la próxima vez tendrá un quanto mayor)

♭ si no llega a agotar el quanto (porque pasa a espera,...) se le aumenta la prioridad

La estructura proc de los TS tiene los siguientes miembros en `p_cldata`:

`ts_timeleft`, tiempo que le queda del quanto

`ts_cpupri`, parte de la prioridad que regula el sistema (la que se recalcula)

`ts_upri`, parte de la prioridad que pone el usuario (que puede modificarse mediante la llamada al sistema `priocntl`)

`ts_umdpri = ts_cpupri + ts_upri`, prioridad total real del proceso (rango 0-59)

`ts_dispwait`, segundos transcurridos desde que se le asignó la CPU, desde que empezó su quanto

El sistema usa una tabla para recalcular las prioridades, donde recoge las prioridades, los quantos asociados, un campo `time quantum expired` que marca lo que se le disminuye la prioridad a un proceso si agota su quanto, un campo `sleep return` que marca lo que se le aumenta la prioridad si no lo agota, un campo `longwait` que contiene el tiempo que ha de pasar en espera para que además se le aumente la prioridad en el contenido que señala un último campo `maxwait`[13].

La llamada que permite variar la prioridad de los procesos, `priocntl`, admite cuatro parámetros:

```
priocntl(idtype_t idtype, id_t id, int cmd, /* args */);
```

donde `idtype` especifica un proceso, un grupo de procesos, los procesos de un grupo, el thread de un proceso,... es decir, sobre *qué* quiere aplicarse; `id` es el identificador concreto, esto es, sobre *quién* quiere hacerse la llamada; `cmd` especifica qué se se quiere hacer, lo que puede requerir algunos parámetros más (cuarto argumento).

Los posibles valores del entero `cmd` son:

`PC_GETCID`, devuelve el id de clase

`PC_GETCLINFO`, devuelve información de la clase

`PC_SETPARAMS`, establece los parámetros de la estructura proc

[13]Ver un ejemplo en la sección de transparencias y ejemplos al final del tema.

PC_GETPARAMS, devuelve los parámetros de la estructura proc

En estos dos últimos casos, se puede especificar como parámetros:

```
typedef struct pcparams { id_t pc_cid;
                          int pc_dlparams[PC_CLPRMSIZE];
                        } pc_parms_t;


typedef struct tsparams { pi_t ts_uprilim;
                          pi_t ts_upri;
                        } tsparms_t;
```

Cuadro 1.4: Valores para idtype y pid correspondiente a especificar en id.

idtype	Significado	id correpondiente
P_LWID	Modificar la prioridad de un thread	pid de la hebra
P_PID	Modificar la prioridad de un proceso	pid del proceso
P_PPID	Modificar la prioridad de todos los hijos de un proceso	pid del proceso padre
P_SID	Modificar la prioridad de todos los procesos de una sesión	pid del proceso líder de la sesión
P_SGID	Modificar la prioridad de un grupo de procesos	pid del proceso líder del grupo
P_CID	Modificar la prioridad de todos los procesos de una clase	pid de la clase
P_UID	Modificar la prioridad de los procesos de un usuario	pid del usuario
P_GID	Modificar la prioridad de los procesos de un grupo de usuarios	pid del grupo de usuarios
P_ALL	Modificar la prioridad de todos los procesos del sistema	pid del proceso líder del sistema

14

Planificación en Linux

En Linux, además del valor nice y las llamadas al sistema getpriority() y setpriority() que actúan sobre él, se tienen las funciones sched_setscheduler sched_getscheduler, sched_setparams y sched_getparams. Entre los argumentos que manejan se hallan los que controlan los criterios (*policy*) de prioridad:

[14] Ver un ejemplo de su utilización en la sección del final del tema.

`SCHED_FIFO` (el proceso no abandona la CPU hasta que acabe o pase a espera)

`SCHED_RR` (el proceso se ejecuta en round robin con los de su misma prioridad)

`SCHED_OTHER` (el proceso actúa normalemente, con prioridad por defecto 0)

Los parámetros `SCHED_FIFO` y `SCHED_RR` sirven para *emular* los procesos RT de SVR4. Las prioridades estáticas varían entre 0-99.

1.5. Gestión de Procesos

Como ya comentamos en la sección 1.3 (página 11), existen dos formas de crear un proceso: las llamadas `fork()` y `vfork()`.

La llamada al sistema `fork()` hace una copia exacta del padre, copia que incluye datos, pila,... La única diferencia entre los dos procesos existentes a partir de que se efectúa, es el valor que devuelve: `0` al hijo y el `pid` del hijo al padre. Por lo demás, son dos procesos iguales, aunque independientes (cada uno maneja sus propias variables)[15].

El problema de la llamada `fork()` es que es *poco óptima*, sobre todo por la mencionada copia que se realiza de datos, pila, etc. Existen dos optimizaciones que atacan este punto flaco e intentan subsanarlo:

Optimización con `copy on write`

Esta mejora consiste en no duplicar datos y pila del padre, sino *compartirlos* con él. De este modo, lo único que tiene que hacer `fork()` es actualizar las páginas del hijo que se crea para que apunten a las del padre.

Debido a la compartición de datos y pila, el sistema ha de marcar las mencionadas páginas como de *sólo lectura* para ambos. En caso de que alguno de ellos quiera modificarlas, es entonces cuando se realiza la copia de la página de datos/pila que corresponda[16]. De esta manera se ahorra en memoria y se gana en tiempo.

Optimización de `vfork()`

Como también se mencionó en su momento, la llamada `vfork()` (existente en sistemas BSD), es una optimización de `fork()`, que consiste en que `vfork()` no reserva espacio de intercambio, ni asigna mapas de traslación de direcciones, ni realiza una copia de los datos y pila del padre, ni inicializa un nuevo contexto hardware para el hijo,... Simplemente, toma *prestado* el espacio de direcciones del padre (datos, código, pila, tablas, etc). De esta manera se logra gran rapidez en la llamada, porque no se busca ni se asigna ni se actualiza nada. Entre tanto, el padre queda en espera hasta que el hijo acaba (con `exit()`) o hace un `exec`.

De hecho, la motivación de esta mejora nace de que la mayoría de las veces en que un programa hace un `fork()` éste va seguido de un `exec` (¡y sólo debe usarse en estos casos!), ya que en situaciones así es una gran pérdida de tiempo el reservar y copiar todo

[15] Ver transparencias del final del tema.

[16] Lo que ocurre es que, cuando padre o hijo intentan escribir en alguna de las páginas compartidas, se produce una excepción, que el SO captura, identifica y resuelve de la manera mencionada.

para inmediatamente después "tirarlo" y reemplazarlo.

En sistemas como Linux, que no tienen llamada `vfork()` propiamente dicha, existe una función *alias* de `fork()` por compatibilidad con código fuente.

Mediante llamadas de la familia `exec`[17] hacemos que un proceso *ya creado* ejecute un programa, siendo *reemplazado* por él tanto su código, como sus datos, su pila,. . . [18]

1.6. Señales

Las **señales** son usadas por el kernel y los propios procesos para comunicarse entre sí. Cuando un proceso recibe una señal pueden pasar tres cosas:

- que se realice la acción por defecto asociada a dicha señal
- que se tenga un *manejador* para controlarla
- que se haga caso omiso de ella

Cualquiera de ellas tiene lugar no en el mismo instante en que la señal es recibida, sino en el momento en el que el proceso obtenga la CPU en *modo usuario*. Esto es, si está en espera, se aguardará a que obtenga la CPU, y si está ejecutándose en modo kernel, la(s) acción(es) pertinente(s) se realizará(n) en el momento en que vuelva a modo usuario (por seguridad). En realidad, hay *esperas interrumpibles* y *no interrumpibles*; en las primeras, se saca al proceso de la espera, se atiende la señal y se termina, devolviendo -1 y poniendo en `errno` que el proceso fue interrumpido por una señal.

Señales en SVR2

En SVR2 cada vez que llega una señal, el kernel "anota" en la estructura proc del proceso al que va dirigida que ha recibido una señal (activa un bit que lo indica). Cuando el proceso va a volver a modo usuario, el sistema consulta un array de punteros ubicado en la u_area, denominado `u_signal[]`, para obtener información sobre la acción a realizar por causa de dicha señal: `SIGIGN`, `SIGDFL`, `función_definida_por_usuario` (comúnmente denominada *manejador* o *handler*).

Si la acción a realizar encontrada en el array `u_signal[]` es la ejecución de un manejador, el kernel reestablece la entrada a la acción por defecto antes de ejecutarlo[19]. Esto hace que, salvo que se incluya un código como el de la tabla 1.6 (página 26), el envío de dos señales consecutivas a este proceso puede no ser "capturado" y, consecuentemente, puede provocar la finalización del proceso con el segundo `Ctrol+C`. ¿Qué ocurre, pues, si se está ejecutando un manejador y llega otra señal igual? Pues se ejecutaría de nuevo (si ha dado tiempo a restablecerlo), o se tomaría la acción por defecto.

[17] `execl`, `execv`, `execle`, `execve`, `execlp` y `execvp`.

[18] Ver transparencias del final del tema.

[19] Nótese que señalamos *si la acción a realizar es un manejador*, en otro caso (`SIGIGN`), no se modifica.

```
#include <signal.h>

void manejaControlC() {
  printf(``Proceso interminable\n'');
}

main() {
  signal(SIGINT, manejaControlC);
  while (1);
}
```

Cuadro 1.5: Ejemplo de función manejadora de señales.

```
void manejaControlC() {
  signal(SIGINT, manejaControlC);
  printf(``Proceso interminable\n'');
}
```

Cuadro 1.6: Ejemplo de función manejadora de señales (II).

La ejecución del manejador se hace de la forma habitual: se crea una capa de contexto en la pila y se guardan en ella parámetros de la función, dirección de vuelta, etc.

El motivo por el que el sistema sustituye la entrada de `u_signal[]` por la acción por defecto, es con vistas a evitar un posible desbordamiento de la pila en caso de que llegasen muchas señales del mismo tipo en un muy corto intervalo de tiempo[20].

Por estas razones, las señales en SVR2 se denominan **señales no fiables** (*not reliable*), y debido a ello su implementación cambió en SVR3.

Señales en SVR3

En SVR3 se permitió que los manejadores fueran *permanentes*, y se añadió la posibilidad de *enmascarar* las señales, con la finalidad de evitar un posible desbordamiento de pila: cuando una señal llega y es atendida, ejecutándose un manejador permanente, mientras éste se encuentre "activo", dicha señal queda "enmascarada", no se atiende hasta que el manejador acabe (el *handler* de una señal se ejecuta con esa señal "desactivada").

Existen funciones que permiten enmascarar y desenmascarar explícitamente una señal (`sighold()`, `sigrelse()`), así como dejar a un proceso en espera aguardando la llegada

[20]Puesto que es sólo un bit en la estructura proc el que indica la llegada de señales, no puede controlarse si llegan varias, de modo que el orden de atención depende normalmente de la implementación.

de una señal concreta (`sigpause()`). La función `signal()` es sustituída por `sigset()`, que funciona de la misma manera.

En SVR3 las señales son, pues, **señales fiables**.

No obstante, SVR3 aún no permite el tratamiento de señales más que de una en una. Remitimos a la parte de transaparencias del tema para la revisión de las funciones de manejo de señales tanto en los dos apartados anteriores como en los dos venideros (System V, BSD).

Señales en BSD

En BSD aparece el concepto de **máscara de señales**: cada proceso tiene asociado un entero cuyos bits marcan qué señales están enmascaradas y cuáles no. Para darle un valor a dicha máscara se usa la función `sigsetmask()`; con `sigblock()` se añaden señales a la máscara y con `sigmask()` se hace algo similar al `sighold()` de SVR3, enmascarar una señal concreta.

Para establecer un manejador para una señal existe la función `sigvec()`, entre cuyos argumentos se pasa además una máscara que se asocia al manejador durante su ejecución. Así pues, la máscara *efectiva* para esa señal mientras su manejador se esté ejecutando es una unión entre la propia señal para la cual el manejador se instala, las que se asocian al mismo al establecerlo y las que el proceso tiene ya enmascaradas. Se puede emular el comportamiento de `signal()` con `sigvec()`, pero no viceversa.

La ejecución de los manejadores de señales tiene lugar normalmente en la pila de usuario, como cualquier otra función. Sin embargo, `sigstack()` nos permite establecer una pila alternativa para la ejecución de las señales.

Ya hemos comentado que un proceso en espera que recibe una señal actúa de forma diferente según su espera sea interrumpible o no interrumpible. En el primer supuesto el proceso salía de espera y terminaba devolviendo -1. En BSD las llamadas al sistema se reinician automáticamente, así que esto no sucedería. Con `siginterrupt()` se puede especificar el comportamiento ante la llegada de una señal a un proceso en espera (si se interrumpe o no, si las llamadas al sistema se reinician o no...).

Señales en SVR4

En System V Release 4 el funcionamiento es similar al de BSD. Se unifican las funciones de manejo de máscaras a una sola, `sigprocmask(how, inset, outset)`. El parámetro `how` puede valer `SIGBLOCK` (añadir a la máscara), `SIGUNBLOCK` (restar de la máscara) o `SIGSETMASK` (establecer máscara); `inset` son las señales que se quieren añadir/restar/establecer como enmascaradas y en `outset` se recoge el estado de la máscara después de realizar los cambios.

La otra diferencia significativa está en la función de asignación de manejador, llamada ahora `sigaction(sig, insa, ousa)`, y que permite controlar cómo se establece. Por defecto, los manejadores son persistentes y fiables. Cada estructura `struct sigaction` (`insa`, `ousa`) tiene 3 campos:

`sa_handler`, la función manejador

`sa_mask`, señales que permanecen enmascaradas mientras se ejecuta el manejador (a parte de la propia señal asociada y las que están ya enmascaradas para el proceso)

sa_flags, que controlan la ejecución del handler, de los cuales los más usuales son:

- SA_RESETHAND (hace que el manejador no sea por defecto)
- SA_NODEFER (manejador no fiable, no se ejecuta con su señal enmascarada)
- SA_SIGONSTACK (ejecución en pila alternativa, siempre que se haya definido alguna)
- SA_SIGACTION (hace el manejador más completo, recibe más cosas).

```
struct sigaction s;

sigemptyset(s.sa_mask);
sigaddset(s.sa_mask, SIGSEGV);
s.sa_handler=manejador;
s.sa_flags=0;
sigaction(SIGINT, &s, NULL);
```

Cuadro 1.7: Ejemplo del uso de sigaction().

1.6.1. Implementación de las señales

En SVR4 la información relativa a las señales se mantiene en diferentes campos de la u_area y de la estructura proc de cada proceso:

u_area

- u_signal[], array que contiene las direcciones de los manejadores
- u_sigmask[], array de señales enmascaradas para cada manejador
- u_sigaltstack, dirección de la pila alternativa
- u_sigonstack, máscara de las señales que se ejecutan en la pila alternativa
- u_oldsig, máscara de las señales que exhiben el comportamiento antiguo (el de SVR2, con signal(): no permanente y no fiable)

proc

- p_cursig, señal que se está atendiendo (32 bits)
- p_sig, señales que han llegado
- p_hold, señales enmascaradas que han llegado
- p_ignore, señales ignoradas

1.6.2. Envío de señales

La función utilizada para enviar señales a los procesos es la función `kill(pid, signal)`. Su funcionamiento es simple: antes de nada, se comprueban las credenciales, ya que si la credencial real o efectiva del proceso que envía la señal no coincide con la real del que la va a recibir se le hará caso omiso. En caso de que se deba atender el envío, lo siguiente que se hace es, mediante el `pid`, que es el del proceso al que se quiere enviar la señal `signal`, se acude a su estructura proc (de fácil acceso gracias a la relación *hash* entre pid's y estructuras proc) y se mira en `p_ignore` si el proceso está ignorando la señal. Si no es así, si pone a 1 el bit correspondiente a la señal `signal` en la máscara `p_sig`.

Más tarde, cuando el proceso vaya a volver a modo usuario (o si ya lo está), examinará la información anterior y utilizará la de su u_area: en caso de que no esté enmascarada esa señal para él (`p_hold`), ejecutará el manejador correspondiente (`u_signal`) bien en la pila de usuario o bien en la pila alternativa (`u_sigonstack`), siguiendo el procedimiento habitual: guardado de la dirección de vuelta, creación de una capa de contexto, ejecución del manejador, etc.

1.7. Comunicación entre procesos

1.7.1. Recursos IPC: Pipes

La función `int pipe(int fd[2])` es una llamada que permite abrir dos ficheros (`fd[0]`, `fd[1]`) que son "invisibles" a la vez. La peculiaridad es que lo que se escribe en uno se lee con el otro y viceversa. Además, a medida que se leen, los datos desaparecen y sólo se leen bytes.

Con `pipe()`, pues, se crea un **canal de comunicación** entre procesos. Otras formas son: *semáforos*, *memoria compartida* y *colas de mensajes*, que veremos a continuación.

Algo común a los recursos IPC es su condición de **externos** a los procesos, no pertenecen a un proceso en concreto. Es un proceso el que lo crea, sí, pero éste puede acabar y el recurso permanecerá ahí hasta que la máquina sea reiniciada, por ejemplo. Todo recurso se identifica unívocamente por un número (*key*, de 32 bits).

1.7.2. Semáforos

Las funciones disponibles para su manejo son:

```
int semget(key_t key, int nsems, int semflg);
int semop(int semid, struct sembuf *sops, size_t nsops);
int semctl(int semid, int semnum, int cmd,...);
```

No nos detendremos en su descripción, dado su estudio en la primera parte de la presente asignatura.

1.7.3. Memoria Compartida

De nuevo, las funciones de las que podemos hacer uso son:

```
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(char *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Para utilizar una zona de memoria compartida, los pasos a seguir son:

1) Crear el bloque de memoria a compartir por los procesos. El tamaño de ese bloque puede tener un límite impuesto por hardware y/o por el kernel, que establece límites en el tamaño de las regiones. Esto se realiza mediante la llamada `shmget()`.

2) Hacer que, por medio de las tablas de páginas, esa zona sea vista en determinado lugar del espacio de direcciones del proceso A, proceso B... para poder de este modo acceder a ella. Es de lo que se ocupa la llamada al sistema `shmat()`.

Ejemplo:

```
int *p, id;

id = shmget (key, ...);
p = (int *) shmat (id, 0, ...);
```

el 0 hace que sea el kernel el que busque la dirección de memoria en el espacio de direcciones del proceso donde "colocar" (ver) la zona de memoria compartida. Una vez hecho esto, puede accederse mediante `p[0]`, `p[1]`,... o bien `*p`, `*(p+1)`...

Para *dejar de usar* el bloque de memoria compartida se emplea `shmdt()`. Con `shmctl()` se elimina. Se observan los permisos de la zona, que son establecidos por el proceso que la crea.

1.7.4. Colas de Mensajes

```
int msgget(key_t key, int msgflg);
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
int msgrcv(int msgid, void *msgp, size_t msgsz, long msglyp, int msgflg);
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

1.8. Transparencias y ejemplos del tema

Apéndice A

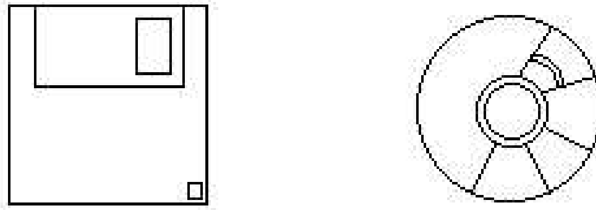
Sistemas de Ficheros

A.1. Introducción

Aunque a lo largo de la historia se han utilizado diversos soportes (tarjetas, cintas,...), los sistemas de ficheros actuales son en su totalidad **sistemas de ficheros en disco**.

Llamamos **sector** a la unidad mínima que se puede leer y/o escribir en un disco. Cada sector puede referenciarse mediante tres coordenadas: *cara*, *pista* y *sector*.

Figura A.1:

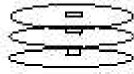


Un disquete de alta densidad tiene 2 caras, 80 pistas y 18 sectores por pista, esto es, un total de 2880 sectores.

Sin embargo, los sistemas operativos no trabajan a tan bajo nivel, sino que lo hacen con **bloques**, que no son más que grupos de sectores (1,2,4,8,...). Por ejemplo, en un disquete se pueden considerar 1439 bloques de 1 Kb (2 sectores).
Esto es así para evitar movimientos a las cabezas lectoras/escritoras:

```
bloque 0 - > cara 0 pista 0 sectores 1,2
bloque 1 - > cara 0 pista 0 sectores 3,4
. . .
bloque 8 - > cara 0 pista 0 sectores 17,18
bloque 9 - > cara 1 pista 0 sectores 1,2
```

Figura A.2:



En un disco duro se suele llamar **cilindro** a todas las pistas numeradas de igual modo que giran a la vez en distintos discos.

Es el **manejador de dispositivo** del sistema operativo el que se ocupa de acceder físicamente.

A.2. Contabilidad del espacio libre

Llevar la **contabilidad del espacio libre** en un disco significa saber en cada momento cuántos bloques están sin asignar.

Una cuestión importante se refiere a *dónde* ha de residir dicha contabilidad. La respuesta está clara: **en el propio dispositivo**, pues démonos cuenta si no de los problemas de portabilidad que ello supondría.

Una de las primeras estrategias que se nos ocurrirían para llevar esta contabilidad del espacio libre sería la de poner a los bloques libres una marca especial que indicaría que están sin asignar, pero esto es algo inviable, ya que obligaría a leer todo el disco cada vez que se quisiera conocer el espacio libre en el mismo, además de conllevar otro tipo de cuestiones menores, como la decisión del valor que indicaría esa no-asignación, por ejemplo.

Una primera solución (la más utilizada) es usar un **mapa de bits**. Cada bit representa un bloque. Un bit a 1 supondría bloque libre y un bit a 0 supondría bloque ocupado (o viceversa). En este mapa la búsqueda de espacio libre es rápida, aunque tiene el problema de que ocupa bastante espacio, pues hay que guardar dicho mapa de bits. El espacio que ocupa es, lógicamente, proporcional al tamaño del disco al que se refiere, así como al tamaño del bloque (proporción inversa en este último caso). Por ejemplo, para un disco de 200 Mb con tamaño de bloque de 1 Kb se necesitarían:

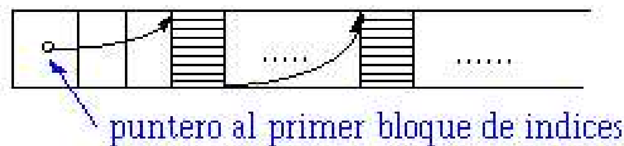
$$\frac{204800}{8} = 25 \text{ Kb } = 25 \text{ bloques}$$

para el mapa de bits.

Una segunda solución es mantener una **lista enlazada de bloques libres**. El sistema operativo guardaría entonces un puntero al primer bloque libre y éstos se enlazarían consecutivamente entre sí. La ventaja de este método es que no ocupa espacio a mayores (sólo el del puntero al primer bloque libre), pero arrastra el problema de que encontrar mucho espacio libre es muy lento.

Por último, una tercera solución consistiría en una **lista enlazada de bloques de índices**, lo que también es poco costoso a nivel de espacio y no tan excesivamente lento a la hora de buscar espacio libre.

Figura A.3:



Cada entrada del bloque de índices contiene una dirección de un bloque libre.
En caso necesario, la última entrada es la dirección de otro bloque de índices.

A.3. Métodos de asignación

Los **métodos de asignación** rigen cómo se asigna el espacio de disco a los ficheros. Es la **entrada de directorio** de un fichero la que nos indicará después en qué zonas del disco ha sido almacenado.

Hay diferentes formas en que los sistemas operativos asignan el espacio libre a los ficheros:

- **asignación contigua**
- **asignación enlazada**
- **asignación mediante índices**

A.3.1. Asignación contigua

Esta política de asignación mantiene *siempre juntos* todos los datos de un mismo fichero. Es un método sencillo y fácil de implementar. En la entrada de directorio del fichero sólo tiene que figurar la dirección del primer bloque asignado al mismo.

Soporta acceso directo, por lo que operaciones como `seek` son fáciles de construir. El problema que presenta se percibe cuando el disco se va llenando y los huecos libres son cada vez más pequeños. Hay una alta *fragmentación externa*.

A.3.2. Asignación enlazada

En este caso un fichero se guarda en disco como una *lista enlazada de bloques*.

No se tiene fragmentación externa, pero no soporta acceso directo (sólo secuencial), ya que no se pueden poner todas las direcciones de sus bloques en su entrada de directorio.

Para solucionar esto surge la tercera variante.

A.3.3. Asignación mediante índices

En la entrada de directorio del fichero se mantiene la dirección de un *bloque de índices*, un bloque que contiene las direcciones de los bloques asignados al fichero en el disco.

No hay fragmentación externa y soporta acceso directo. El único inconveniente es el espacio consumido por el bloque de índices.

Hoy en día todos los sistemas operativos trabajan con alguna variante de este tercer método.

A.4. Métodos de acceso

Los **métodos de acceso** soportados por un sistema operativo dependen fundamentalmente del método de asignación que utiliza. Pueden clasificarse:

- **Métodos de acceso secuenciales**
  - `read`
  - `write`
- **Métodos de acceso directo**
  - `read`
  - `write`
  - `seek`

Veremos cómo manejar los ficheros teniendo en cuenta esta primera clasificación.

A.4.1. Operaciones sobre ficheros

Las operaciones básicas que necesitamos (queremos y esperamos) poder realizar sobre un fichero son:

- **creación**
- **borrado**
- **lectura**
- **escritura**
- **posicionado**[1]

Creación de un fichero

El proceso consiste sencillamente en crear la entrada de directorio, inicializarla, buscar espacio libre y asignarlo.

Borrado de un fichero

Se busca en su entrada de directorio el espacio ocupado por el fichero, se marca como libre y por último se borra dicha entrada de directorio.

[1] En caso de que el ACCESO DIRECTO esté soportado.

Leer de un fichero

Lo primero que debe hacerse es leer su entrada de directorio para encontrar la dirección de disco del fichero, y posteriormente leer de disco.

Para evitar tener que leer la entrada de directorio y buscar la dirección física cada vez, los sistemas operativos implementan la llamada `abrir fichero`, que viene a ser como comunicar la intención de usar el fichero. Lo que se hace es colocar en una tabla del sistema (**tabla de ficheros abiertos**) toda la información relevante relativa al fichero.

Consecuentemente, se tiene la operación `cerrar fichero`, que comunica al sistema operativo que no se va a usar más el fichero, con lo que éste borra su entrada de la tabla de ficheros abiertos (que suele tener un tamaño fijo).

Escribir en un fichero

Es totalmente análogo a la lectura.

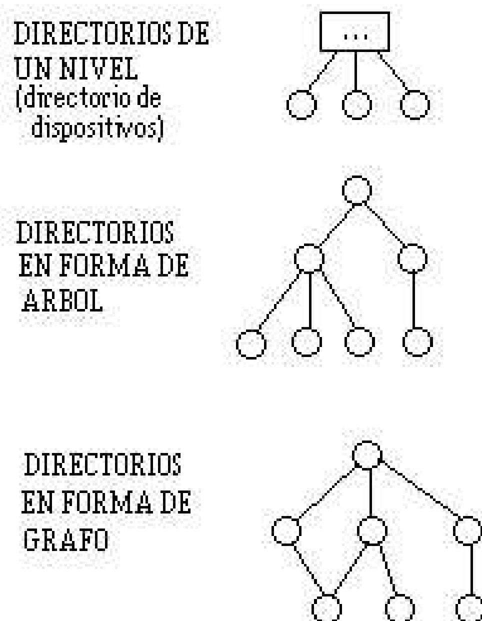
A.5. Sistema de directorios

En su concepto más elemental, un **directorio** es una tabla (antiguamente se denominaba **directorio de dispositivo**) que informa de qué archivos hay y dónde están situados.

A efectos prácticos, un **directorio** es un fichero que contiene los nombres de los archivos incluidos en él, así como otra información útil al sistema operativo. Cada línea en ese fichero, referente a un archivo diferente, se denomina **entrada de directorio** del archivo.

Podemos hablar de:

Figura A.4:



En el caso de los **directorios en forma de árbol** el sistema operativo ha de manejar los conecptos de:

√ directorio actual / directorio raíz

√ cambiar directorio actual / cambiar directorio raíz (del proceso)

√ ruta de búsqueda (dónde buscar ejecutables)

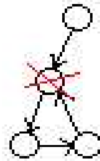
Un **directorio en forma de grafo** puede ser:

∗ acíclico

∗ general (permite ciclos)

Este último caso puede presentar el problema de que al borrar no se desasigne espacio:

Figura A.5:



Esto sólo puede solucionarse mirando todo el disco.

A.6. Sistemas de Ficheros al uso

IBM desarrolló el sistema de ficheros **hpfs**. NT se basa en el sistema de ficheros **ntfs**.
El sistema de ficheros de **OS/2** divide el disco en zonas llamadas *bandas*. La primera banda del disco contiene el número de bandas y sus tamaños, y cuenta con una *banda central de directorios*.

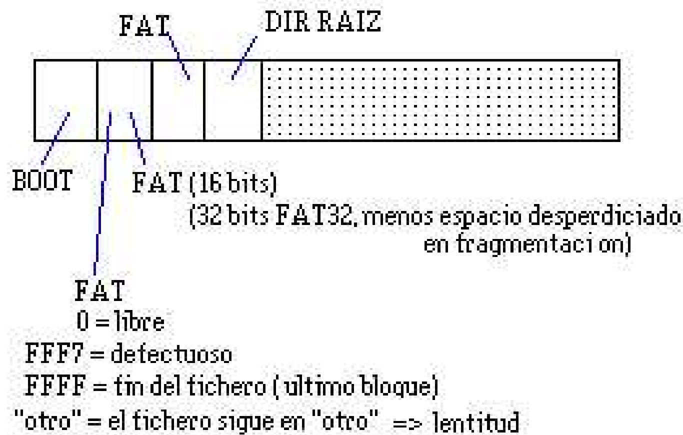
Figura A.6:

Es un sistema con poca fragmentación interna, pues los ficheros son listas enlazadas de sectores. En su entrada de directorio está el nombre del fichero y un puntero al primer sector (**fnode**).

De este modo, si el mapa de bits sufre algún desperfecto puede recuperarse recorriendo todos los **fnodes**.

Otro sistema de ficheros muy popular es el **FAT**. Este sistema divide el disco en *zonas*. En el primer sector, llamado de **boot**, se encuentra el código de arranque. A continuación se guardan dos copias (que han de ser exactamente iguales salvo error) de FAT, donde se lleva contabilidad del espacio libre, de ficheros,...

Figura A.7:



Los bloques de datos se denominan **clústers**. Si la FAT resulta dañada, se pierde todo. Las entradas de directorio son de 32 bits: 8+3 para el nombre (si el nombre es largo, se asignan dos entradas), 1 para atributo, 4 para fecha, 4 para tamaño, 2-4 para la dirección del primer bloque de datos del fichero.

Bibliografía

[1] Bach, M. J. *The design of the Unix Operating System*. Prentice Hall.

[2] Márquez García, F. *Unix programación avanzada*. Ra-ma.

[3] Robbins, Kay A. y Robbins, Steven. *Unix Programación Práctica. Guía para la Concurrencia, la Comunicación y los Multihilos*. Prentice Hall.