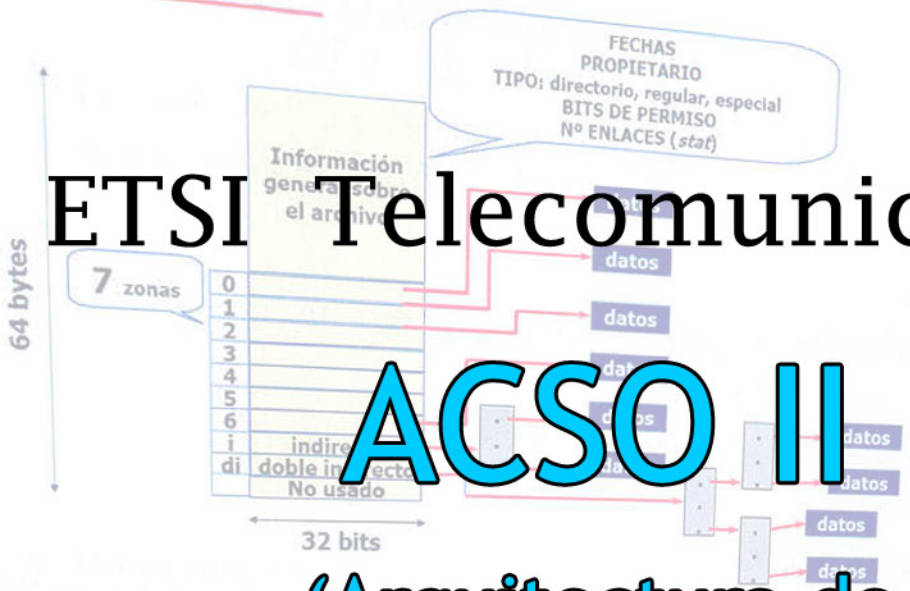
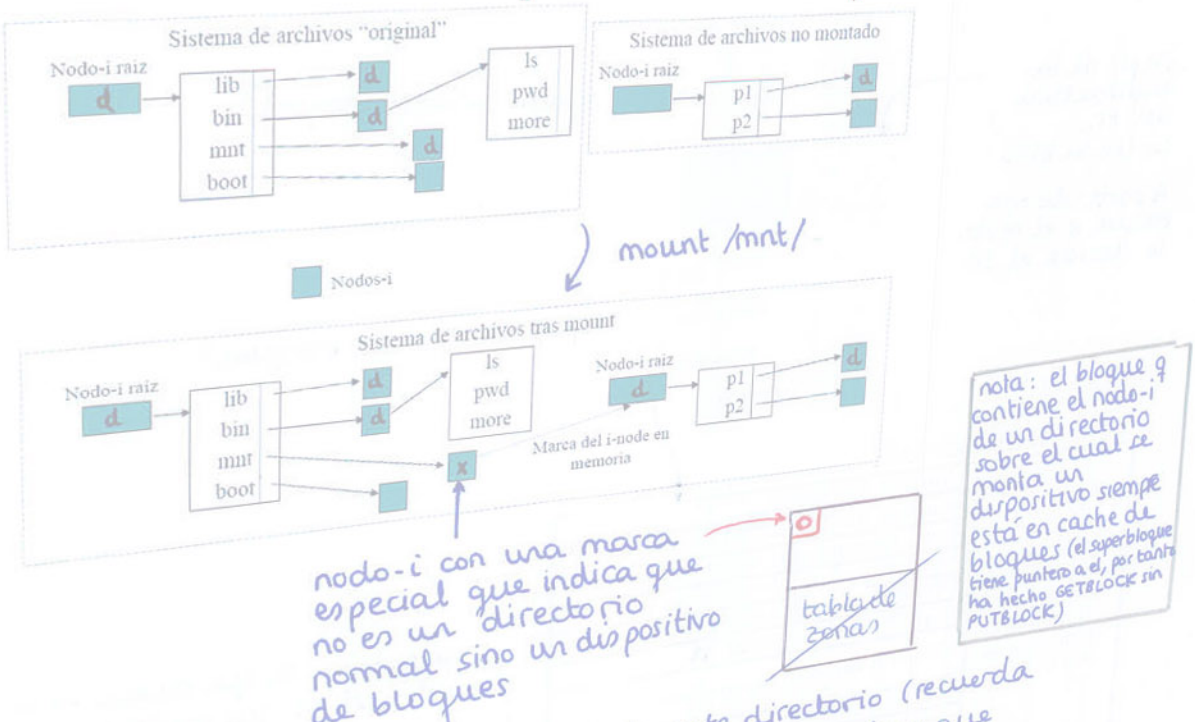
ETSI Telecomunicación

ACSO II

(Arquitectura de Computadores y Sistemas Operativos)





Implica que para hallar las zonas de este directorio (recuerda que un directorio TAMBIEN es un archivo con zonas) no hay que buscar en la tabla de zonas del nodo-i, sino que:
Habrá que buscar en cada superbloque de cada dispositivo hasta hallar aquel cuyo campo "puntero al nodo-i del directorio sobre el que se montó el dispositivo" coincida con el nodo-i; entonces para éste directorio habrá que utilizar el nodo-i apuntado por "puntero al nodo-i del directorio raiz del sistema de ficheros" del dispositivo. (que será un nodo-i que ya estará en un bloque de la cache de bloques al cual se hizo GETBLOCK cuando se montó el dispositivo; por tanto no necesitaremos ningún GETBLOCK adicional por un directorio sobre el cual hay un dispositivo montado.)

Apuntes de Pak

Arquitectura de Computadores y Sistemas Operativos II

Apuntes de Pak (Fco. J. Rodríguez Fortuño)
ETSI Telecomunicación. Universidad Politécnica de Valencia.
Segundo cuatrimestre de 4º curso
Curso 2006/2007

Referencias y agradecimientos

Estos apuntes están basados en las clases de ACSO II que me impartió el profesor Dr. D Julio Pons Terol así como en las transparencias de la asignatura (tanto de teoría como de prácticas) elaboradas por él y los demás profesores de la asignatura Doctores D. Miguel Ángel Mateo Pla y D. Pedro Pablo Cruz Alcázar.

Las figuras impresas que aparecen en estos apuntes están tomadas de dichas transparencias, así como el texto a máquina de los temas 2 (llamadas al sistema) y 4. Los exámenes que reproduzco en los apuntes son también de los ofrecidos por los profesores.

Fecha de última actualización: 26 Junio 2009

UT1. VISIÓN DEL PROGRAMADOR DE UN SO

Tema 1. Estructura del sistema operativo

1. Visión del Programador

Aplicaciones de 3 tipos:
- Scripts: No se utilizan servicios del SO, sino programas (o librerías)
- Independientes del SO: (por ejemplo Java). El programador utiliza un lenguaje que se interpreta para cada S.O.
- Binarias: llaman al SO para realizar tareas que no puede hacer directamente

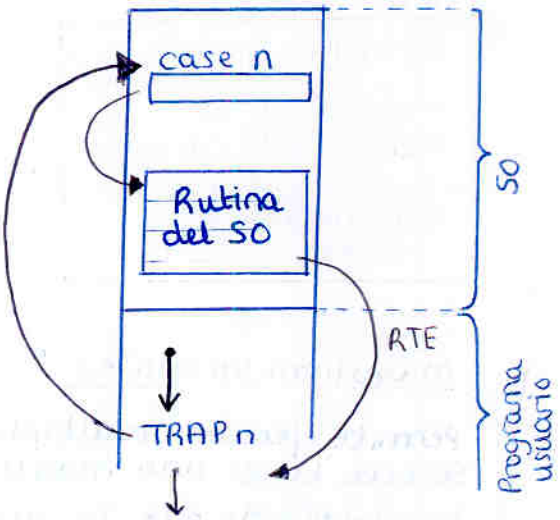
¿qué servicios ofrece el SO? ¿Cómo se accede a ellos?

2. Conceptos útiles

Hay dos alternativas para realizar las llamadas al sistema:

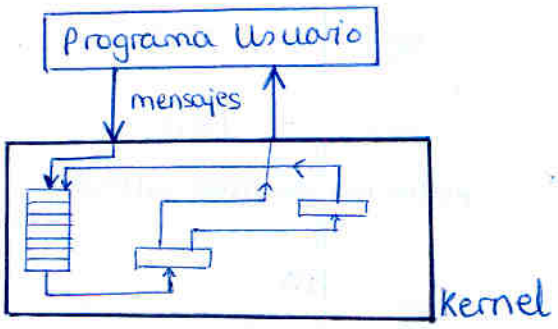
- TRAPS:

El programa de usuario está en modo usuario y el sistema operativo en modo supervisor. Cuándo el usuario quiere utilizar algún servicio del sistema operativo (que normalmente requiere modo supervisor) hace una llamada a TRAP con un parámetro n correspondiente al nº de llamada que se desea. En ese momento el usuario pierde el control y el programa salta a la rutina del S.O. (para el procesador es la rutina de atención a la interrupción TRAP, que se hace en modo supervisor) Ésta acaba su servicio y devuelve el control al programa de usuario donde estaba (RTE)



- Intercambio de mensajes:

El SO es un proceso en marcha que está esperando mensajes de los programas de usuario



3. Estructura del S.O.

Aspectos a tener en cuenta: necesidades de memoria para ejecución, tiempo de ejecución de las tareas, nº de procesos pertenecientes al SO, sistemas de protección, formas de comunicación entre procesos, formas de comunicación entre procero y SO.

Hay alternativas:

- Sistema Monolítico

El S.O. se compone de un único programa que realiza todas las tareas. Todas las estructuras de datos y algoritmos están disponibles desde cualquier línea del S.O.

Todo variables globales y puedo reutilizar código → más eficiente en ejecución

→ más difícil de mantener, construir y modificar

Kernel (núcleo) → nombre que recibe el sistema operativo

Linux utiliza ejecución monolítica pero con diseño modular

- sistema en capas

Aplicaciones
Servicios (llamadas al sistema)
Tareas E/S (drivers)
Administración Procesos

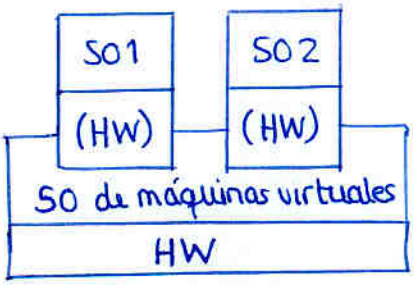
La capa n utiliza los servicios de la capa n-1 y ofrece otros a la capa n+1

- máquinas virtuales

Permite ejecutar múltiples SO's en una misma máquina, donde cada SO cree tener una máquina completa.

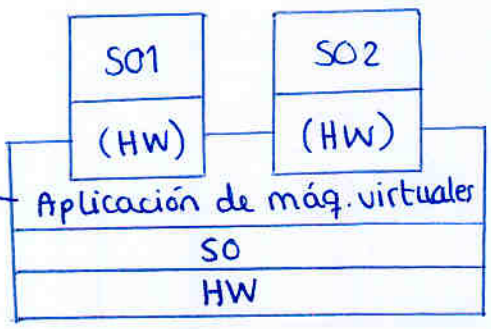
La gestión de ésto la puede hacer:

(a) Un propio SO diseñado para ello



(b) Una aplicación diseñada para ello corriendo sobre un SO

ejemplo VMWare



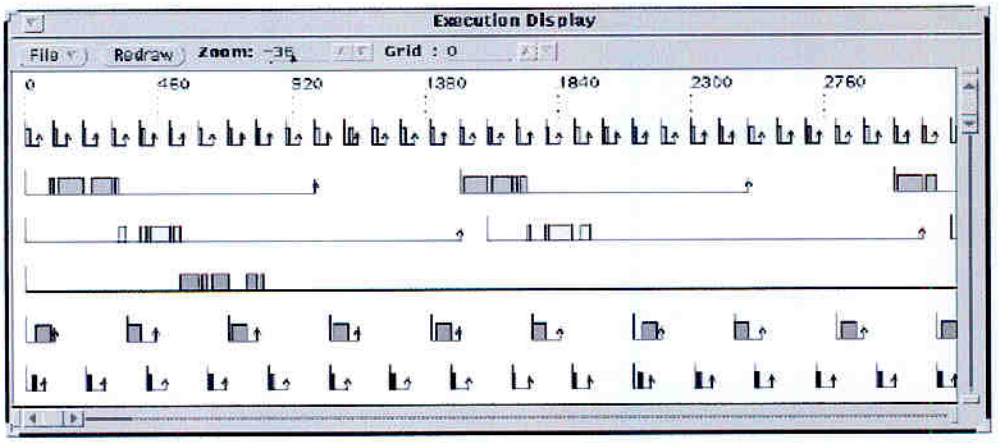
- Sistema cliente/servidor

- Kernel básico que permite la comunicación entre procesos por paso de mensajes.
- Todos los servicios del SO se realizan con procesos servidores

4. Sistemas de Tiempo Real

Definición: Un sistema de tiempo real es aquel sistema informático en el que la corrección del sistema no sólo depende de los resultados lógicos de los algoritmos, sino que también depende del momento en el que estos se producen.

Es decir, las tareas definen unos plazos (intervalos de tiempo) para su ejecución.
(Los intervalos no tienen porqué ser pequeños, por ejemplo control de rumbo de un barco; por tanto no confundir sistema en tiempo real con sistema rápido)



Tipos de sistemas de tiempo real:

- Duros o estrictos: se debe ASEGURAR el cumplimiento de las especificaciones temporales ej: sistemas de control ej: airbag

- Blandos: Pueden no cumplirse siempre las restricciones temporales ej: sistemas de reproducción de contenidos multimedia

sistemas de tiempo real y SO

El SO se encarga de:
- El algoritmo de planificación
- Los mecanismos de comunicación entre tareas
- Gestionar las interrupciones
- Activar las tareas en cada uno de sus periodos

· Todas estas tareas influyen en el cumplimiento de las restricciones temporales.

· El objetivo de un SO en tiempo real es que éstas actividades se realicen con un coste temporal predecible y lo más rápidamente posible

Tema 2. Llamadas al sistema

1. Creación de Procesos

Los parámetros que habría que pasar en la creación de un proceso son demasiados (tablas, permisos, variables, ...) por tanto se decidió que para crear un proceso exista una única llamada sin parámetros que clone el proceso actual

```
a = fork();
```

Crea un proceso idéntico al proceso padre, copiando todas las variables y registros con una única diferencia: el valor de a
Tanto el padre como el hijo creen ser el proceso original, y sólo pueden diferenciarse comprobando el valor de a.

Para el padre: a devuelve el PID del hijo
Para el hijo: a devuelve 0

```
a=fork();
if a!=0
   /soy el padre
else
   /soy el hijo
end if
```

Para que el nuevo proceso se convierta en un proceso distinto al padre (parece lógico que queremos hacer esto, ya que es la única forma de ejecutar un proceso que queramos) se usa la orden exec (típicamente execve)
↓ indica cómo se le pasan los parámetros

execve(nombre_programa) ← ejemplo "/bin/ls"

NOTA: no busca el ejecutable en el conjunto de directorios contenidos en la variable de entorno PATH.

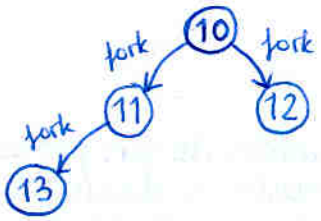
reemplaza el proceso actual por el nuevo programa (sustituye la memoria de instrucciones) manteniendo todo lo que no depende del código (mismo PID, hijos, protecciones, ...)

exit (codigo_de_terminacion)
↓ de 0 a 255
= 0 → Ok
≠ 0 → Error

- exit termina el proceso
- todos los compiladores ponen automáticamente un exit al final; el valor que le ponemos al return del main es el parámetro que el compilador le pone al exit.

yo = getpid(); ← devuelve mi PID
padre = getppid(); ← devuelve el PID de mi padre (recuerda: proceso INIT ← PID=1)

El que hace fork es el padre y el clon es el hijo



Si ahora (11) hace un exit, (13) se queda huérfano; pero en UNIX es obligado que todo proceso tenga padre
→ se le da un padre adoptivo; el (1) INIT

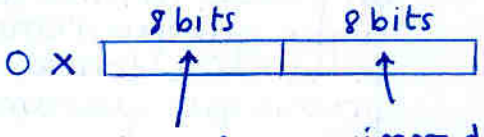
Cuando el 11 hace un exit con cierto código de retorno, ¿Cómo conoce 10 ese código? El padre estará haciendo sus cosas, y deberá hacer

```
n = wait(&status)
```

devuelve el PID del hijo muerto

a wait hay que pasarle un puntero a int para que pueda ser sobreescrito

wait bloquea al padre mientras espera a que muera alguno de sus hijos, cuando un hijo muere la variable entera status toma el siguiente valor



código de terminación si el hijo a muerto con un exit, 0 en otro caso

número de señal si el hijo a muerto por una señal, 0 en otro caso

si el hijo ha terminado correctamente, lógicamente ambos campos tendrán un cero

```
pid_hijo = wait(&status)
if status == 0
   / el hijo ha terminado bien
if status < 256
   / num_señal = status
if status > 255
   / cod_terminacion = status / 256
```

← Lo que debería tener un proceso tras el wait

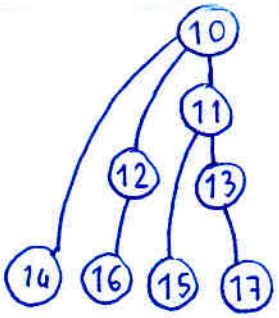
¿Y qué pasa si al morir el hijo el padre no estaba en un wait?
El hijo mantiene su programa en memoria (sin ejecutarse) junto a todas sus variables: proceso zombie
¿con qué utilidad? Útil para los postmortem debuggers
mientras el padre no haga wait (y reciba el status) mantenemos al zombie en memoria

¿Y si muere el padre antes que alguno de sus hijos?
El hijo "huérfano" es adoptado por el procero INIT (PID=1) el cual está continuamente en un bucle con wait para atender a sus hijos que mueran.

Problema: diagrama de procesos, valor de las variables y mensajes en pantalla

```
a = 1;
b = 2;
c = fork();
if (fork()==0){b= getpid();}
else {a= a+1}
printf("soy %d y 'a' vale %d \n", getpid(), a);
c = fork();
```

PID	10	11	12	13	14	15	16	17
a	~~1~~ 2	~~1~~ 2	1	1	2	2	1	1
b	2	2	~~2~~ 12	~~2~~ 13	2	2	12	13
c	~~11~~ 14	~~0~~ 15	~~11~~ 16	~~0~~ 17	0	0	0	0
fork	12	13	0	0				



soy 10 y 'a' vale 2
soy 11 y 'a' vale 2
soy 12 y 'a' vale 1
soy 13 y 'a' vale 1

2. Parámetros y entorno

La definición estándar en C de main es:

```
int main ( int argc , char * * argv , char * * envp )
```

- `argc`: número de argumentos; no es algo que se "PASA" sino que lo calcula el propio programa antes de main.
- `char * * argv`: puntero al 1er elem de un array de char "cadena" → puntero al 1er elemento de un array de cadenas
- `argv`: parámetros
- `char * * envp`: variables de entorno

ejemplo

```
$ export A=7
$ ls -l f1 f2   ⟶   argv[0] = "ls"
                    argv[1] = "-l"
                    argv[2] = "f1"
                    argv[3] = "f2"
                    argv[4] = NULL

                    envp[0] = "HOME = /home/..."
                    envp[1] = "PATH= /bin; /usr/bin; ..."
                    :
                    envp[18] = "A=7"
```

nota: `char * * argv ≡ char * argv[]` (vector de cadenas)

ejemplo de como pasar argumentos:

```
int main(int argc, char *argv[], char **envp){
    char * argv2[10];
    argv2[0] = "pepe";
    argv2[1] = "leches";
    argv2[2] = NULL;
    execve ("./hola", argv2, envp)
```

ejemplo: ¿qué hace bash?

```
printf($);
leer_linea(); → argv, nombre_fichero,...
a = fork();
if (a== 0) {
    execve(nombre_fich, argv, envp);
    printf("error: no he podido usar execve con la orden \n");
    exit(1);
}else {
    wait(&status);
}
```

Existe una variable del sistema

```
extern char * * environ;
```

que inicialmente copia el contenido de envp y se puede modificar fácilmente con funciones de librería

```
setenv(...);
getenv("HOME");
```

} existe para facilitar el uso del entorno

otras funciones de librería para ejecutar programas son:

```
execl ("/bin/ls", "ls", "-l", -0 );
execle ("/bin/ls", "ls", "-l", 0,
        "HOME=~", "PATH= ...", ...)
```

permite pasar argumentos y entorno uno a uno.

3. Señales

Las señales son un mecanismo de comunicación asíncrona entre procesos; cuando el S.O. tiene una señal para un proceso pueden ocurrir 4 cosas:

- el proceso ha decidido ignorar la señal
- el proceso ha definido ejecutar una determinada función ante la llegada de la señal
- el proceso no ha definido acción para esa señal y por tanto se hará la acción por defecto (que en muchos casos es que el proceso finalice)
- la señal está enmascarada (no usamos ésto en ACSO II)

Cada señal tiene asociado un número; algunas señales son las siguientes (ver: man 7 signal ó kill -l)

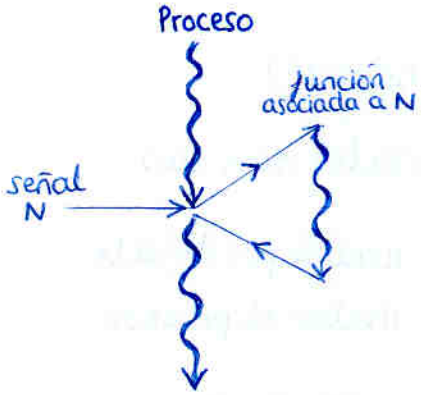
Señal	nº de señal	Comentario	Acción por defecto
SIGINT	2	Interrupción de teclado (CONTROL+C)	Matar el proceso
SIGKILL	9	Matar	Matar el proceso No puede ser capturada ni ignorada
SIGTERM	15	Terminar el proceso	Matar el proceso
SIGALRM	14	Señal de alarma	Matar el proceso
SIGSTOP	19	Parar el proceso	Parar el proceso No puede ser capturada ni ignorada
SIGCONT	18	Continuar si parado	
SIGTSTP	20	Petición de teclado para parar el proceso	Parar el proceso

Las acciones de un proceso ante señales se definen como entradas en una tabla; cada proceso tiene asociado su tabla

nº señal	función a ejecutar
1	← si no hay entrada hacemos acción por defecto
2	SIGIG ← constante que significa ignorar señal
3	
4	funcion_pepito
5	
⋮	

- Envío de señales: **kill** (int pid_destino, int num_signal)
  sólo funciona entre procesos del mismo usuario (salvo el root que puede a todos)

- Definición de qué hacer ante una señal:
  Para escribir una nueva entrada en la tabla usamos: **signal** (num_signal, funcion)
  (funcion: el nombre de la función, tal cual)



  · devuelve la función que tenía anteriormente en la tabla si tenía alguna (puntero a la función)
  · cuando llegue la señal "num_signal", nuestro proceso saltará a la función "funcion"
  además borrará de la tabla la función, de forma que la proxima señal "num_signal" que llegue hará su acción por defecto a menos que se haya hecho una nueva llamada a signal
  (si queremos que la señal siempre ejecute la misma función, una solución es escribir la orden signal (num_signal, funcion) dentro de la propia función para que se autoactualice en la tabla)

nota: usar la función signal tiene el problema de que no está definida en un estándar y su comportamiento podría para ciertos SO ser diferente (ligeramente) al descrito en clase: (Linux utiliza por defecto la definición BSD, matizaciones bien descritas en man 2 signal)
como en clase hemos estudiado la definición System V que usa MINIX, usaremos la función sysv_signal() para que el comportamiento sea el que suponemos en teoría.

- pause() El programa se bloquea hasta que le llega una señal cuando llega la atiende y luego continua la ejecución

nota: pause() no modifica las acciones asociadas a las señales, a diferencia de sleep(), el cual en algunas versiones de algunos sistemas operativos sí las modifica (por ej: usando alarmas)

- resto = alarm (tiempo)
  ↓ planifica el envío a mi mismo de la señal SIGALRM dentro de tiempo segundos
  ↓ devuelve cuantos segundos faltaban para la alarm previa (el último alarm SOBREESCRIBE a los anteriores)

Lista de señales

(man 7 signal)

Señal	Valor	Acción	Comentario
SIGHUP	1	A	Cuelgue detectado en la terminal de control o muerte del proceso de control
SIGINT	2	A	Interrupción procedente del teclado
SIGQUIT	3	C	Terminación procedente del teclado
SIGILL	4	C	Instrucción ilegal
SIGABRT	6	C	Señal de aborto procedente de abort(3)
SIGFPE	8	C	Excepción de coma flotante
SIGKILL	9	AEF	Señal de matar
SIGSEGV	11	C	Referencia invÃ¡lida a memoria
SIGPIPE	13	A	Tubería rota: escritura sin lectores
SIGALRM	14	A	Señal de alarma de alarm(2)
SIGTERM	15	A	Señal de terminación
SIGUSR1	30,10,16	A	Señal definida por usuario 1
SIGUSR2	31,12,17	A	Señal definida por usuario 2
SIGCHLD	20,17,18	B	Proceso hijo terminado o parado
SIGCONT	19,18,25		Continuar si estaba parado
SIGSTOP	17,19,23	DEF	Parar proceso
SIGTSTP	18,20,24	D	Parada escrita en la tty
SIGTTIN	21,21,26	D	E. de la tty para un proc. de fondo
SIGTTOU	22,22,27	D	S. a la tty para un proc. de fondo

A La acción por omisión es terminar el proceso.

B La acción por omisión es no hacer caso de la señal.

C La acción por omisión es terminar el proceso y hacer un volcado de memoria.

D La acción por omisión es parar el proceso.

E La señal no puede ser capturada.

F La señal no puede ser pasada por alto.

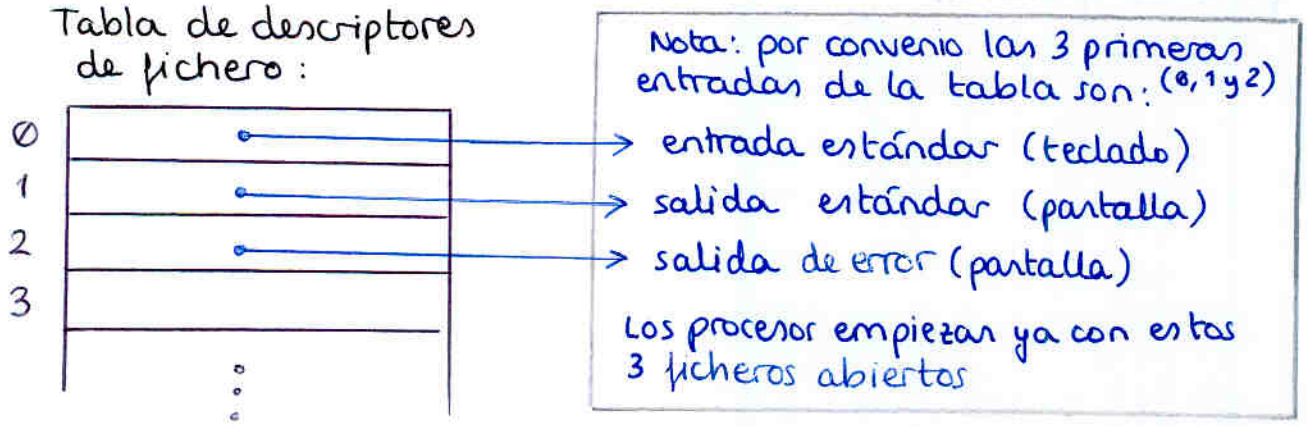
4. Ficheros

Cada fichero tiene un directorio y un nombre para identificarlo

/usr/pak/home/hola.c

Ésta 'operación' requiere varios accesos al disco buscando por las tablas (cada directorio es una tabla).
Para no tener que repetir éste proceso cada vez que queramos leer o escribir en el archivo, lo que tenemos es una función a la cual pasándole la ruta y el nombre del archivo devuelve un número, a partir de ese momento puedo acceder al fichero con ese número.

Cada proceso tiene localmente una tabla donde la posición asociada a cada número (descriptor de fichero) contiene un "puntero al fichero"




```
int fd; // descriptor de fichero
fd = open ("nombre_fichero", flags [, ini_perm])
                                   ↑ opcional
```

↑ asigna el fichero a la primera posición libre que haya en la tabla y devuelve el número de índice (el descriptor de fichero).

```
num_bytes_leidos = read (fd, buffer_destino, num_bytes)
```

↑ Lee num_bytes bytes desde el fichero asociado a fd y los pone en buffer. Devuelve el número real de bytes leidos (ej: por si se acababa el fichero antes de num_bytes) (devuelve -1 si error)

```
num_bytes_escritos = write (fd, buffer_origen, num_bytes)
```

↑ Escribe num_bytes del buffer al fichero asociado a fd

```
ok = close (fd)
```

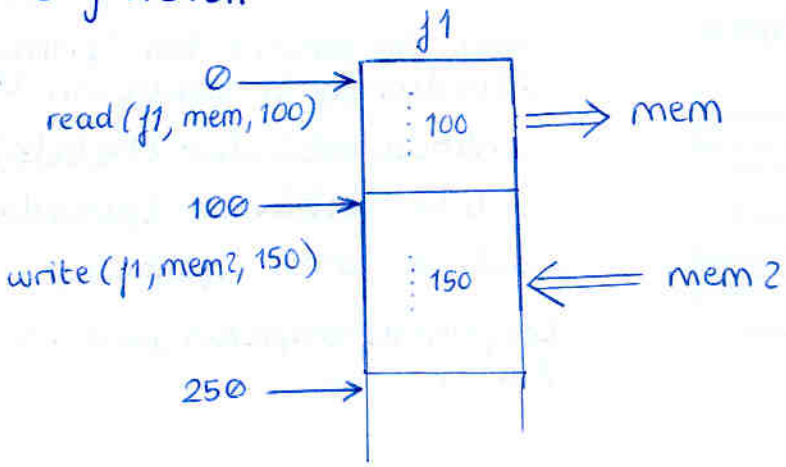
↑ Libera la entrada de la tabla ok=0 → correcto, ok=-1 → error

ejemplo: redirección de la salida estándar por parte del bash

```
$ ls > f1  ⟶   a = fork();
               if (a==0)
                   close(1); ← libera la salida estándar
                   open("f1") ← asocia fichero f1 a la salida estándar (es el primer hueco libre de la tabla)
                   execve("/bin/ls", —)
                   ↑ la tabla de descriptores de fichero se mantiene
```

posicionamiento en el fichero (lseek)

Si no se usa lseek, read y write es secuencial con un único puntero para leer y escribir



Usando lseek puede cambiarme de posición en un fichero.
Tiene 3 modalidades:
- posición absoluta
- posición relativa a la actual
- posición relativa al final

devuelve la posición en que ha quedado (si usamos la relativa al final con 0, nos devuelve el tamaño del fichero)

Nota acerca del final de fichero:
Si yo hago read de 100 bytes pero al fichero sólo le quedan 65 bytes para acabar; read no bloqueará al programa esperando 35 bytes más, sino que simplemente leerá los 65 bytes almacenándolos en buffer destino, y devolverá num_bytes_leidos = 65.
Esto es diferente, como veremos, a las tuberías, donde el SO no puede saber cuando se han acabado los datos de la tubería (a no ser que el origen la cierre) y por tanto read se quedaría bloqueado esperando hasta tener los 100 bytes.

Creación de un fichero (creat u open)

Nota histórica: inicialmente la llamada open no podía crear un fichero, por eso se utilizó la orden creat; pero con el tiempo fueron modificando la orden open, de forma que ahora ya lo hace todo:

Modos de open
- → Lectura
- → Escritura
- → L/E
- → Creación

Por tanto actualmente creat es un alias de open

```
creat("f1", 0 777)
```
- "f1": nombre del fichero a crear
- 0 777: permisos iniciales (la 0 indica octal)

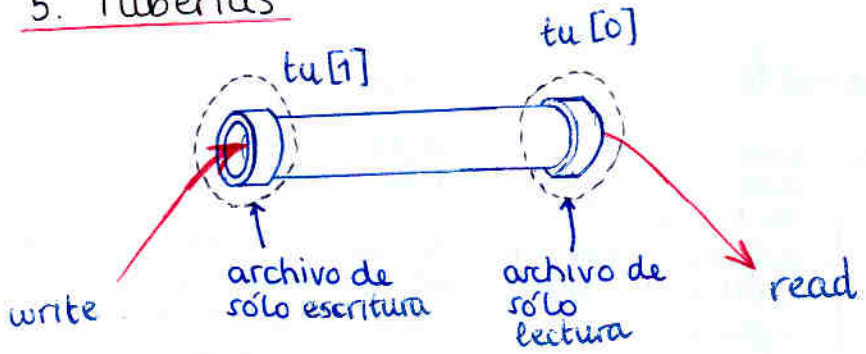
El SO tiene una variable umask que enmascara los permisos que se le intentan dar a un fichero (hace AND con la umask negada)

0 777
0 007
→ 0 770

Propiedades de un fichero

stat y fstat son funciones para obtener datos de un fichero (fechas, permisos, ...)

5. Tuberías



Se crea con la llamada `pipe(vector)`

vector de dos enteros en el cual se devuelven los dos descriptores de fichero
- 0 — salida
- 1 — entrada

} igual que en la tabla de descript. de ficheros
1 — stdout
0 — stdin

truco para recordar
0: stdin
1: stdout

tal que se hace
close(0)
dup(tu[0])
close(1)
dup(tu[1])

ejemplo:

```
int tu[2]
pipe(tu)
```

para leer: `read(tu[0], —)`
para escribir: `write(tu[1], —)`

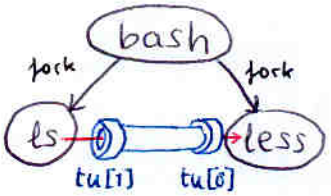
mientras el fichero de entrada a la tubería (tu[1]) siga abierto (i.e. presente en la tabla de descriptores de fichero) en uno o más procesos, el SO no tiene más remedio que suponer que aún no se ha acabado de escribir en la tubería. Por tanto si un proceso hace read a la tubería de 100 bytes, se le devolverá los 100 bytes que hayan en la tubería, y si no hay tantos, el proceso se quedará 'bloqueado' hasta que se metan suficientes datos o hasta que todos los procesos que escribían en la tubería cierren el fichero de escritura, en cuyo caso el read se comportaría como si hubiera encontrado un fin de fichero (es decir, leerá hasta donde pueda y devolverá el número de bytes leídos).

Por eso es importante que todo proceso que no vaya a usar más la tubería la cierre.

Para intercambiar datos padre e hijo:

- Padre hace PIPE antes del fork (y obtiene tu[0] y tu[1])
- Padre hace fork (y el hijo tendría tu[0] y tu[1])
- Ya puedo hacer write y read

ejemplo: $ ls | less




```
pipe(tu)
a = fork();
if (a==0) { % el hijo que será ls
    close(1)
    dup(tu[1])
    close(tu[1])
    close(tu[0])
    exec(ls);
} else {
    b = fork();
    if (b==0) {
        close(0);
        dup(tu[0])
        close(tu[0])
        close(tu[1])
        exec(less)
    } else {
        close(tu[0])
        close(tu[1])
        wait();
        wait();
    }
}
```

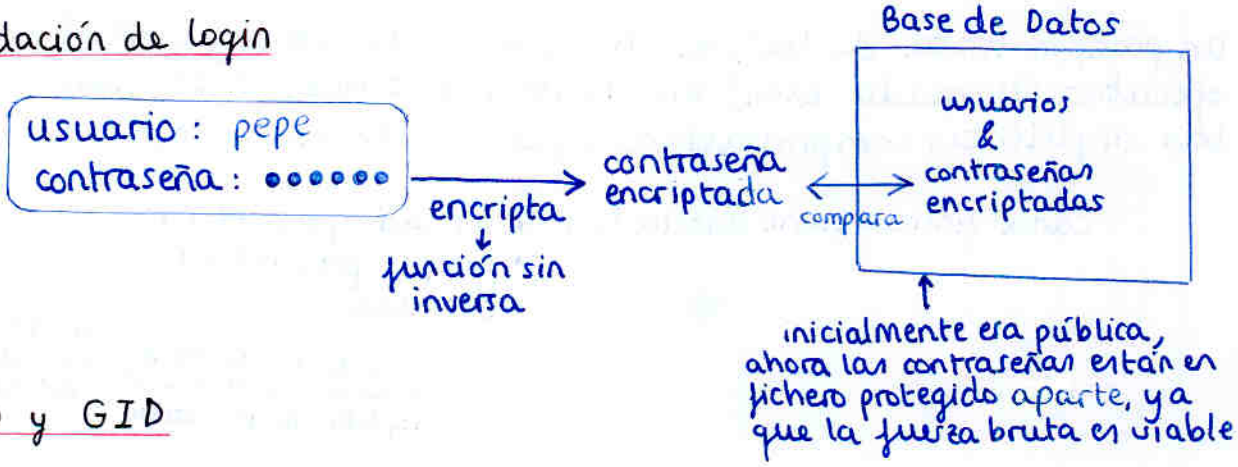
- close(1) cierra la salida estándar, dejando un hueco en la posición 1 de la tabla de descriptores de fichero.
- dup(tu[1]) DUPLICA el descriptor de fichero tu[1] en el primer lugar libre que haya en la tabla que sabemos que será el 1, por tanto ya hemos asociado lo que normalmente es la salida estándar con la entrada de la tubería
- se cierran tu[1] y tu[0] para evitar problemas como el del principio de la página (además no los vamos a usar)

el padre crea otro hijo y el nuevo hijo hace lo mismo pero con la entrada estándar

el padre cierra la tubería ya que ya no la necesita y se queda esperando a la muerte de sus hijos

6. Protecciones

- Validación de login



- UID y GID

Una vez validado, el usuario pasa a tener un número UID
Los usuarios pueden agruparse en grupos con número GID

Un usuario puede ser de varios grupos, pero cara al sistema sólo un grupo cada vez (aunque puede cambiarlo)

Entonces el SO crea un proceso inicial para el usuario (ej: shell o entorno gráfico) al cual le asigna el UID y GID.
A partir de entonces todas los nuevos programas que lance el usuario serán hijos de éste, y por tanto "heredarán" el UID y GID

- UID y GID reales y efectivos

En realidad cada proceso en el sistema tiene dos UID y dos GID:

UID real, GID real, UID efectivo, GID efectivo

UID real, GID real: número y grupo de la persona que ha hecho login en el sistema.
Nunca se cambia (con la única excepción del root cuando tras el login de un usuario le crea el primer proceso)

UID efectivo, GID efectivo: Inicialmente se copia del real, pero se puede cambiar bajo ciertas condiciones
Éste es el que se usa para comprobar permisos
getuid
getgid

- Cambio del UID y GID efectivo:

setuid (123)
setgid (13)

Podemos hacer ésto si se cumple cualquiera de las siguientes condiciones:

- soy el root
- el usuario al que quiero cambiar coincide con el real
- el usuario al que quiero cambiar ya lo he sido anteriormente (cada proceso guarda un historial)

· Comprobación de permisos

Un proceso trata de leer, escribir (llamadas read y write) o ejecutar (llamada exec) un fichero. Entonces el SO hace las siguientes comprobaciones para autorizarlo o no:

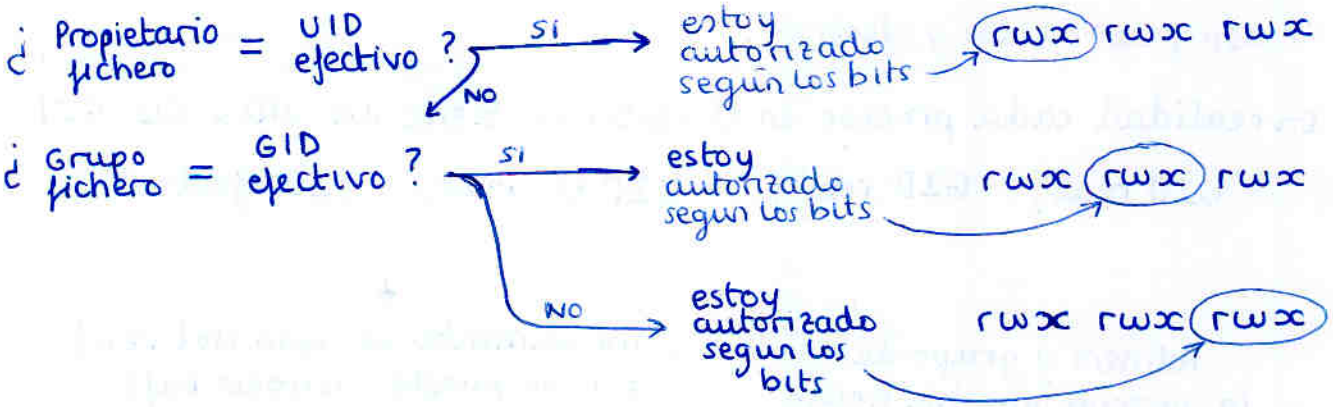
Cada fichero tiene asociado:
- usuario propietario
- grupo propietario
- permisos

rwx rwx rwx
propiet. grupo todos

r: read
w: write
x: execute

- bits SETUID y SETGID

El proceso que intenta leer, escribir, o modificar tiene (entre otras muchas cosas) asociado:
- UID real
- UID efectivo
- GID real
- GID efectivo



· Bits SETUID y SETGID

si el proceso quería ejecutar un fichero, y el SO ha comprobado que tiene permisos, el SO hace otras dos comprobaciones

si SETUID = 1 → al nuevo proceso hijo (correspondiente al fichero que queríamos ejecutar) se le asignará como UID efectivo el UID del propietario del fichero

si SETGID = 1 → al nuevo proceso hijo se le asignará como GID efectivo el GID del propietario del fichero

else → al nuevo proceso hijo se le asignan los GID y UID del proceso que lo ejecutaba (i.e. el padre) lo cual es lo que por defecto ya ha hecho fork antes de exec.

¿Qué sentido tiene eso?
Es como si cada vez que alguien tiene que ejecutar algo 'programado' por mi, tiene que "llamarme" para que vaya yo y lo haga, ya que soy el único que sabe hacerlo

ejemplo: base de datos a la cual sólo tiene permiso de acceso el creador, usuario "basedatos", pero para que otros usuarios puedan hacer uso de ella creó un programa para ello cuyo propietario es "basedatos" y con SETUID = 1.
De ésta forma cuando alguien quiera usar el programa, lo hará en nombre de "basedatos" y por tanto el programa podrá acceder a la base de datos.
La ventaja es que el programa sólo hace lo que el creador quiera y nada más; y es a través de ese seguro programa la única forma que tienen otros usuarios de acceder a la base de datos.
Una vez que el usuario cierre el programa, vuelve al "padre" cuyo UID sigue siendo el suyo propio.

Cuidado: si soy root y me ausento del PC 30 segundos, la maniobra más rápida y efectiva que alguien malintencionado puede hacer es activar SETUID al bash (el propietario de bash es root), de forma que en otro momento podrá ejecutar bash como root.

Por eso el SO busca programas con SETUID = 1 y da la alerta de nuevos programas con ese bit.
Los programas con SETUID están, digamos, registrados.

Ver el estado de los bits SETUID y SETGID:

ls -l

rwx rwx rwx
↓ ↓
rws rws ← setGID

setUID ↗ en lugar de 'x' aparece 's'

obviamente setUID y setGID no tienen sentido cuando no hay permiso de ejecución (aparece un guión)

Permisos de directorios

Los directorios tambien tienen permisos rwx rwx rwx (entrada de ls!)

Ésta vez 'r' y 'w' hacen referencia a leer y escribir en el nodo-i del directorio (ej: listar contenido o borrar archivo que contiene).
'x' hace referencia a 'abrir' el directorio

Nota: si tienes permiso 'w' en un directorio, puedes crear archivos, cambias nombres e incluso borrar ficheros aunque no tengas permiso sobre los ficheros. (i)

Llamada al sistema mount

mount (especial, nombre , opciones)

requiere ser superusuario (root) (parece lógico)

- especial: fichero del dispositivo, ej /dev/disco5
- nombre: directorio al que queremos asociar, ej /discos/usr1
- opciones: sólo lectura, ...

En UNIX toda la entrada salida se hace con ficheros (open, read, write) cuando se trata de impresora, escaner, webcam, raton, ... son ficheros "especiales" que se crean con mknod

Ésa es la llamada que hacen los "drivers" para crear el archivo si es un dispositivo de bloques, ese archivo especial hay que montarlo con mount

El comando mount (no confundir con la llamada al sistema mount) hace uso de la llamada al sistema mount para montar dispositivos. Ya que para ello el proceso mount debe ser root; es un ejemplo claro de la utilidad del bit setuid.

Ya que mount es propiedad de root y tiene setuid = 1, un usuario que tenga permisos para ejecutarlo se "convertirá" durante la ejecución en root efectivo, pudiendo por tanto usar la llamada mount (pero sólo a través de ese programa)

Nota: puede haber montaje automático configurable mediante un fichero (ej: montar automaticamente al meter un CD)

TEMA 2: Llamadas al sistema

1 Introducción

Llamada al sistema: Función que permite acceder a los servicios del Sistema Operativo
Ejemplo:

`cuenta=read(fich,buffer,nbytes)`

Leer `nbytes` de `fich` y depositarlos en `buffer`, devuelve el número de bytes que realmente se han leído.

POSIX:

Portable Operating System Interface. Norma Internacional 9945-1.
La X se suele emplear cuando un sistema operativo está basado en UNIX.
Intenta especificar un interfaz de programación estándar para sistemas operativos. Especifica el interfaz de una serie de procedimientos, no su implementación.
POSIX.1 : Programación general.
POSIX.4 : Extensiones de tiempo real.

En este tema se estudiarán las llamadas al sistema que ofrece MINIX, que siguen el estándar POSIX. Este hecho hace que sean compatibles con cualquier sistema que cumpla con el estándar POSIX, la mayor parte de las llamadas al sistema de LINUX son compatibles con este estándar. Además se incluyen algunas funciones que aunque no pertenecen al estándar POSIX

2 Clasificación

Vamos a estudiar las llamadas al sistema ordenadas en los siguientes grupos:

1. Administración de Procesos
2. Señales
3. Administración de Archivos
4. Sistema de archivos y directorios
5. Protección
6. Administración del tiempo

2.1 *Administración de Procesos*

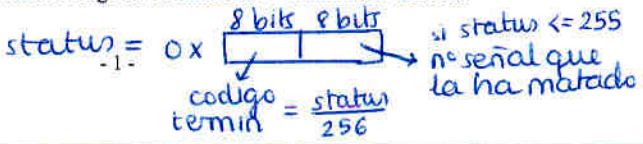
fork, exit, wait, getpid, getppid, exec

fork

Única llamada al sistema que permite crear nuevos procesos.
Sintaxis: `ident=fork()`
La llamada fork() crea un nuevo proceso (proceso hijo) que es idéntico al proceso que la llama (padre) y lanza la ejecución del proceso nuevo en el mismo punto donde se realiza la llamada. El valor que devuelve fork() es distinto para el padre y para el hijo. Al proceso padre le devuelve el identificador del hijo mientras que al hijo le devuelve el valor cero. (-1 si error)

exit

Termina la ejecución del proceso.
Sintaxis: `exit(codterm)`
El parámetro *codterm* indica el código de terminación: 0 – terminación correcta.



wait

Espera la finalización de un proceso hijo ó recepción de una señal.
Es necesario tener hijos para quedarse en espera.
No es del estándar POSIX, la versión POSIX es waitpid.
Sintaxis: `ident=wait(&status)`
ident = –1 si error ó señal ó ident = pid del hijo
en status se devuelve la razón de la terminación del hijo (el valor está basado en el código de exit o número de señal)
Conceptos relacionados: llamada exit, llamada waitpid y gestión de señales.

getpid

Devuelve el identificador del proceso. Cada proceso tiene asignado un identificador de proceso único y mayor que cero (el pid).
Sintaxis: `ident=getpid()`

getppid

Devuelve el identificador del proceso padre.
Sintaxis: `ident=getppid()`

exec

Cambia el código del proceso por el del programa que se especifique. NO CREA PROCESO NUEVO. Si no se produce ningún error NO VUELVE.
Sintaxis: `n=execve(nomfich,argv,envp)`
n = –1 si error.
nomfich nombre del fichero con el programa
argv array de argumentos (parámetros)
envp array de variables de entorno.
Conceptos relacionados: parámetros de la función *main* de los programas en C.

2.2 Señales

signal, kill, pause, alarm, sigretwn

Lista de algunas señales (`kill -l` o `man 7 signal`):

Señal	Valor	Acción	Comentario
SIGHUP	1	A	"Hangup" (acción de colgar detectada en el terminal en control) o muerte del proceso en control
SIGINT	2	A	Interrupción procedente del teclado (^C)
SIGQUIT	3	A	Terminación procedente del teclado
SIGILL	4	A	Instrucción ilegal
SIGTRAP	5	CG	Trampa para rastreo o puntos de ruptura
SIGKILL	9	AEF	Señal para matar
SIGUSR1	10	A	Señal definida por usuario – 1
SIGUSR2	12	A	Señal definida por usuario – 2
SIGPIPE	13	A	Tubería rota: escritura en tubería sin lectores. (Broken pipe)
SIGALRM	14	A	Señal de alarma (ver llamada *alarm*)
SIGTERM	15	A	Señal de finalización
SIGCHLD	17	B	Descendiente terminado o parado
SIGCONT	18		Continuar si parado (asociada a SIGSTOP)
SIGSTOP	19	DEF	Parar proceso
SIGTSTP	20	D	Petición de teclado de parar proceso (^Z)

Los tipos de acciones se refieren a la situación por defecto de las señales:

A: matar el proceso actual.
B: ignorar la señal.
C: Volcar la memoria del proceso a un fichero para depuración. (CORE DUMP)
D: Se para el proceso.
E: Esta señal no puede ser capturada. (VER *signal*)
F: Esta señal no puede ser ignorada. (VER *signal*)

signal

Asocia una función a una señal. ***(NO POSIX)***
Sintaxis: `anterior_func=signal(señal,nueva_func)`
Devuelve la función previamente asociada o –1 si error. Alternativamente, para ignorar señales se pasa la constante SIG_IGN en lugar de *función*, y para devolver el funcionamiento por defecto SIG_DFL. Existen señales a las que no afecta, por ejemplo SIGKILL

kill

Envía una señal a un proceso (ident) dado. El proceso se identifica con su PID.
Un proceso sólo puede enviar señales a procesos de su mismo UID.
Sintaxis: `n=kill(ident,señal)`

pause

El proceso se para hasta que llegue una señal.
Sintaxis: `n=pause()`
Siempre devuelve –1.

alarm

Se planifica la recepción de una señal SIGALRM (14) dentro de un intervalo.
Sintaxis: `resto=alarm(tiempo)`
Dentro de *tiempo* segundos, el sistema operativo mandará la señal SIGALRM al proceso. Un proceso sólo puede tener un intervalo de alarma cada vez.
Devuelve el tiempo que quedaba para el vencimiento de la anterior alarma.
El instante exacto de envío de la señal depende del nivel de carga del sistema.

sigreturn

Regresa de una señal.
No usada por los programadores.
La utiliza el sistema operativo como última función después de procesar una señal capturada.

Señales en POSIX

POSIX no utiliza signal() para modificar el modo de procesar las señales entrantes. En vez de utilizar un valor para definir las señales, POSIX trabaja con máscaras, conjuntos de bits que representan varias señales.
Las llamadas al sistema definidas por POSIX son: sigaction(), sigprocmask(), sigpending() y sigsuspend().
Para la gestión de las máscaras utilizadas en estas funciones se suministran funciones de librería, como sigsetempty(). Estas funciones se pueden consultar en el manual en línea (orden man). Para hacerlo probar la orden: `man sigsetops`.

2.3 Administración de Archivos

open, close, lseek, fstat, pipe, ...

En UNIX los ficheros se ven como un vector de bytes o caracteres. Las operaciones de lectura o escritura se realizan sobre bytes consecutivos. La posición donde se realizan estas operaciones se puede cambiar con la llamada al sistema adecuada.
Los conceptos fundamentales a tener en cuenta en la administración de archivos en los sistemas operativos tipo UNIX son:

- INODE: todos los ficheros son accedidos usando una estructura de datos especial denominada INODE.
- DESCRIPTOR: estructura de datos que sirve para poder encontrar el INODE de un fichero en uso en el proceso actual y el lugar donde se está operando dentro de él.
- TABLA DE DESCRIPTORES. Tabla asociada a un proceso que almacena todos los descriptores pertenecientes al mismo.

Los descriptores se utilizan a través de la tabla de descriptores, por lo que lo más importante es el índice que ocupan dentro de la misma. En muchas ocasiones la bibliografía utiliza "descriptor de fichero" cuando se refiere al índice de la entrada de la tabla de descriptores.
Cuando un proceso empieza, hereda la tabla de descriptores de su padre, por lo tanto, puede acceder a los mismos ficheros que el padre. Más aún, los accesos se realizan en la misma posición que el padre.
Las primeras tres posiciones de la tabla de descriptores tienen un significado especial:

- Posición 0 de la tabla ó STDIN. Fichero que el proceso utiliza como su entrada estándar. Por ejemplo, un shell interactivo lee del teclado a través de este fichero.

- Posición 1 de la tabla ó STDOUT. Salida estándar, la usada cuando se realiza un `printf()`, por ejemplo.
- Posición 2 de la tabla ó STDERR. Salida estándar de error. En este fichero se supone que los programas escriben los mensajes de error que detectan. En muchas ocasiones apunta al mismo fichero que STDOUT. Normalmente se utiliza con la función `perror()`.

open

Abre o crea un fichero para su utilización en lectura, escritura o lectura/escritura.
Sintaxis:

```
fd = open(nombref,flags)
fd = open(nombref,flags, ini_perm)
```

En *nombref* se indica el fichero sobre el que se desea trabajar.
El parámetro *flags* determina para que acción se abre el fichero. Este campo se forma mediante una OR de constantes (el operador OR bit a bit en C es |).
Algunas de estas constantes son:

O_WRONLY: El fichero se abre para escribir en él.
O_RDONLY: El fichero se abre para leer en él.
O_RDWR: El fichero se abre para leer y escribir en él
O_TRUNC: Borra el contenido del fichero. Si se abre el fichero en lectura no se utilza.
O_APPEND: Se posiciona la posición de trabajo al final del archivo.
O_CREAT: Si el fichero no existe se crea.

NOTA: Si se ha especificado O_CREAT es obligatorio poner un tercer parámetro con el mismo significado que el segundo parámetro de la llamada *creat()*.

creat

Crea un fichero vacío para su utilización en escritura.
Sintaxis: `fd = creat(nombref, ini_perm)`
Crea un nuevo fichero denominado *nombref*. En *ini_perm* se definen los permisos iniciales del fichero.
Devuelve su descriptor (índice en la tabla de descriptores). Con el descriptor podremos hacer otras operaciones sobre el fichero.
En caso de error devuelve -1 (fd = -1).
NORMALMENTE SE SUELE USAR *open* para esta acción.

access

Comprueba si un fichero es accesible en un determinado modo.
Sintaxis: `n = access(nombref, modo)`
Modo es una máscara que indica un tipo de acceso.
Devuelve 0 si se puede acceder al fichero en ese modo.

dup

Duplica un descriptor.
Sintaxis: `nfdes = dup(fdes)`
Busca la entrada libre de la tabla de descriptores de ficheros con un índice más bajo y crea en ella un descriptor exactamente igual al pasado como parámetro, *fdes*. Devuelve la posición en la tabla del nuevo descriptor.

close

Cierra un descriptor de fichero.
Sintaxis: `n = close(fd)`
Cierra el descriptor de fichero de la entrada *fd*. Este entrada queda libre.
Devuelve 0 si va bien y –1 en caso de error.

read

Lee un número especificado de bytes de un fichero.
Sintaxis: `leidos = read(fd,buffer,num)`
Lee num bytes desde el fichero apuntado por *fd* y los pone en buffer.
Devuelve el número real de bytes leidos o –1 si hay error.

write

Escribe un número especificado de bytes en un fichero.
Sintaxis: `escritos = write(fd,buffer,num)`
Escribe num bytes del *buffer* en el fichero apuntado por *fd*.
Devuelve el número real de bytes escritos o –1 si hay error.

lseek

Mueve el puntero de trabajo del fichero.
Sintaxis: `pos = lseek(fd, cuanto, desde)`
Mueve el puntero de trabajo del fichero apuntado por fd, tantas posiciones como se indica en *cuanto*, desde la posición indicada por el parámetro *desde (SEEK_SET absoluto, SEEK_CUR relativo y SEEK_END desde el final)*.
Devuelve la posición actual del puntero de trabajo, medida desde el primer byte del fichero. Si el valor devuelto es –1 ha ocurrido un error.

stat y fstat

Devuelven información sobre un fichero (probad el programa *stat*)
Sintaxis:

```
res =  stat(nombre, &info)
res = fstat(fd, &info)
```

Rellena la estructura *info* con información sobre el fichero designado por el primer parámetro. Devuelve –1 en caso de error, y si no cero.
La estructura rellenada es del siguiente tipo:

```
struct stat{
   dev_t         st_dev;     /* device */
   ino_t         st_ino;     /* inode */
   mode_t        st_mode;    /* protection */
   nlink_t       st_nlink;   /* number of hard links */
   uid_t         st_uid;     /* user ID of owner */
   gid_t         st_gid;     /* group ID of owner */
   dev_t         st_rdev;    /* device type (if inode device) */
   off_t         st_size;    /* total size, in bytes */
   unsigned long st_blksize; /* blocksize for filesystem I/O */
   unsigned long st_blocks;   /* number of blocks allocated */
   time_t        st_atime;    /* time of last access */
   time_t        st_mtime;    /* time of last modification */
   time_t        st_ctime;    /* time of last change */
};
```

pipe

Crea una **tubería** (mecanismo básico de comunicación entre procesos en UNIX)
Sintaxis: `res = pipe(descr[])`
Crea una tubería. Las tuberías tienen dos descriptores, para poder devolverlos es necesario pasar como parámetro un vector de dos enteros.
El primer descriptor (descr[0]) es el descriptor de entrada al proceso y se usa para leer de la tubería y con el descr[1] (descriptor de salida) se escribe en la tubería.
Devuelve 0 si todo ha ido bien, y –1 en caso contrario.

mknod

Creación de "ficheros" especiales.
Sintaxis: `res = mknod(nombre_path, modo, num_disp)`
Crea un "fichero" especial con el nombre y la localización especificados por el parámetro *nombre_path*. El fichero se crea con unos permisos y características especificados por *modo*. Si el fichero creado hace referencia a un dispositivo, el tercer parámetro indica el número de dispositivo. **En principio, esta llamadas sólo se puede usar en modo super usuario.**
Si la operación ha tenido éxito devuelve un cero.

ioctl

Operaciones especiales con ficheros de dispositivo.
Sintaxis: `res = ioctl(fd, peticion, argumentos)`
Sirve para la realización de operaciones especiales sobre el fichero cuyo descriptor es *fd*. Hay que tener en cuenta que los dispositivos se direccionan como ficheros.
Devuelve cero si la operación se ha ejecutado correctamente.

2.4 Sistema de archivos y directorios

mkdir, chdir, link, unlink, ...

mkdir

Crea un nuevo directorio.
Sintaxis: `res = mkdir(nombre, permisos)`
Crea el directorio *nombre* con permisos de acceso iniciales *permisos*.
Si error devuelve –1, y si no cero

rmdir

Elimina un directorio.
Sintaxis: `res = rmdir(nombre)`
Elimina el directorio *nombre*, que debe estar vacío antes de la operación.
Si error devuelve –1, y si no cero

chdir

Cambia el directorio de trabajo actual.
Sintaxis: `res = chdir(camino)`
Cambia el directorio de trabajo actual a *camino*, la ruta puede ser absoluta (empieza por /) o relativa al directorio de trabajo actual.
Si error devuelve –1, y si no cero

chroot

Cambia el directorio raíz para el proceso actual.
Sintaxis: `res = chroot(camino)`
Cambia el directorio que el proceso actual considerará a partir de este momento como inicio del sistema de ficheros, es decir el directorio raíz (/).
Si error devuelve –1, y si no cero

link

Crea una nuevo enlace a un fichero.
Sintaxis: `res = link(nombre1, nombre2)`
Crea un nuevo hardlink al fichero *nombre1*, el link creado tiene como nombre *nombre2* y comparte el nodo-i con el fichero original.
El nodo-i contiene un campo en el que cuenta el número de enlaces a este nodo-i.
Si error devuelve –1, y si no cero.

unlink

Elimina un enlace a un fichero.
Sintaxis: `res = unlink(nombre)`
Elimina el link *nombre1*. Decrementa el campo del nodo-i que cuenta el número de enlaces a este nodo-i, si este campo se hace cero el nodo-i y la información del fichero son eliminados del sistema de ficheros.
Si error devuelve –1, y si no cero.

mount

Añade un sistema de archivos al actual.
Sintaxis: `res = mount(special,name,flag)`
Añade el sistema de archivos presente en el dispositivo *special* al sistema de ficheros actual. Al acceder al directorio *name* se accede al directorio raiz del sistema de ficheros montado. El campo *flag* permite modificar algunas opciones de montaje: solo lectura, lectura escritura, propietario del directorio…
Si error devuelve –1, y si no cero.

umount

Elimina un sistema de archivos al actual.
Sintaxis: `res = umount(dir)`
Elimina el sistema de ficheros que se haya montado sobre el directorio *dir*, del sistema de ficheros actual. Los ficheros del sistema desmontado permanecen en el dispositivo.
Si error devuelve –1, y si no cero.

sync

Actualiza la información de los dispositivos de almacenamiento..
Sintaxis: `res = sync()`
Para mejorar los tiempos de acceso a los ficheros, el sistema operativo utiliza una caché con política de actualización write-back. Con esta llamada nos aseguramos que los contenidos del sistema de ficheros y de la caché son los mismos. Existe una llamada similar para la sincronización selectiva de ficheros (llamada fsync())

2.5 Protección

UID y PID

Se basa en cuatro identificadores por cada proceso, el UID (real y efectivo) y el GID (real y efectivo).

getuid

Obtiene el identificador del usuario que ejecutó el proceso.
Sintaxis: `ident = getuid()`
Devuelve el identificador real. El efectivo, el que se usa para permisos de accesos, se puede obtener con *geteuid*.
Si error devuelve –1.

getgid

Obtiene el identificador de grupo del usuario que ejecutó el proceso.
Sintaxis: `ident = getgid()`
Devuelve el identificador real. Devuelve el identificador real. El efectivo, el que se usa para permisos de accesos, se puede obtener con *getegid*.
Si error devuelve –1.

setuid

Modifica el identificador del usuario que ejecutó el proceso.
Sintaxis: `res = setuid()`
Modifica el identificador efectivo.
Si error devuelve –1.

setgid

Modifica el identificador de grupo del usuario que ejecutó el proceso.
Sintaxis: `res = setgid()`
Modifica el identificador efectivo.
Si error devuelve –1.

chmod

Modifica la protección de un fichero.
Sintaxis: `res = chmod(fich, mode)`
Modifica la máscara de protección del fichero *fich*, reemplazándola por la pasada en el parámetro *mode*.
Sólo un proceso con un GID efectivo igual al del propietario del fichero puede caminar la máscara del mismo.
Si error devuelve –1.

chown

Modifica la propiedad de un archivo.
Sintaxis: `res = chown(fich, quien, grupo)`
El fichero *fich* tendrá como propietario a *quien* y tendrá como grupo propietario *grupo*.
Si error devuelve –1.

umask

Modifica la mascara de permisos por defecto.

Sintaxis: `o_mask = umask(n_mask)`
Modifica la máscara de permisos que se aplica por defecto a los ficheros que se creen. Devuelve –1 en caso de error, o el viejo valor de la máscara si todo a dio correctamente.

2.6 Administración del tiempo

En los sistemas tipo UNIX el instante de tiempo actual se mide mediante un contador de segundos. El valor de este contador se corresponde con el número de segundos transcurridos desde el 1 de enero de 1970. Con un contador de 32 bis, se dispone de un total de un contador de 0 a 2^{32}, lo que quiere decir que deben pasar más de 63 años para que el contador vuelva a cero.

time

Obtiene el número de segundos desde el 1 de enero de 1970.
Sintaxis: `seg = time(&seg)`

stime

Establece el número de segundos desde el 1 de enero de 1970.
Sintaxis: `res = stime(&seg)`
Devuelve 0 si todo ha ido bien, -1 en caso de error.

NOTA:
Las páginas de "man" `pueden estar en varios idiomas, aunque lo usual es que originalmente se escribieran en inglés. Por este motivo las páginas en castellano no siempre tienen una traducción "buena" y la página original en inglés aporta en ocasiones matices perdidos en la traducción. Para ver las diferencias, probas en el servidor de prácticas las siguientes órdenes que consultan la misma página de manual en los dos indiomas:

- `man -L es getuid`
- `man -L en getuid`

La opción –L de man determina el lenguaje a utilizar, si están instaladas dichas páginas. También son útiles las ordenes `apropos` y `whatis` y las opciones `-f` y `-k` de `man`.

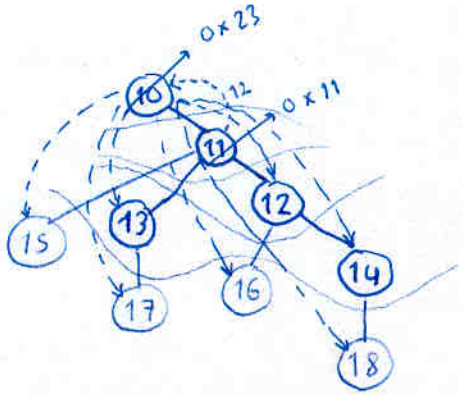
Problema de examen. Llamadas al sistema y señales

```
int recibidas=0, nivel=0;
void fun1(int s){
 printf("SENYAL(%d,%d,%d,%d)\n",getpid(),s,recibidas,nivel);
 if(s==12) recibidas++;
 if(nivel==0){
       signal(s,fun1);
 }else if(nivel<recibidas){
       signal(s,fun1);
 }
}

int main(){
 int fd[2],i, a, motivo, quien, miPid,j,n;
 pipe(fd);
 signal(12,fun1);
 a=fork();
 if(a==0){
       for(i=0;i<3;i++){
             a=fork();
             if(a==0){  nivel=i+1; }
       }
       sleep(1);
       if(nivel==0) kill(getppid(),12);
       sleep(1);
       printf("Yo soy (%d) en nivel (%d)\n",getpid(),nivel);
       sleep(9-nivel);
       miPid=getpid();
       while(recibidas<3){
         write(fd[1],&miPid,sizeof(miPid)); = transmito por la tuberia mi propio PID
         pause();
       }
       close(fd[1]); close(fd[0]);
       /******************/
       do{
         quien=wait(&motivo);
         if(quien>0)
           printf("ha muerto (%d) por (%04x)\n",quien,motivo);
       }while(quien>0);
       printf("acabo (%d)\n",getpid());
 }else{
   close(fd[1]); = close(4)
   sleep(5); → cuando llega la señal te liberan del sleep (por ex hay otro)
   sleep(10);
   do{                                          /*** PUNTO d) ***/
     n=read(fd[0],&j,sizeof(j)); = lee de la tuberia a j
     sleep(1);
     if(n>0){
       printf("he leido %d\n",j);
       kill(j,12);
     }
   }while(n!=0); = n será cero cuando todos los procesos al otro lado de la tuberia hayan cerrado su descriptor de fichero
   exit(0x23);
 }
 exit(0x11);
}
```

Dado el siguiente proceso, suponiendo pid = 10

a) ¿qué procesos se generan y cuales envían señales a cuales?

	10	11	12	13	14	15	16	17	18
recibidas	ø ~~1~~ ~~2~~ 3	ø 1	ø 1	ø 1	ø 1	ø 1	ø 1	ø 1	ø 1
nivel	0	0	ø 1	ø 2	~~1~~ 2	ø 3	~~1~~ 3	2 3	2 3
señal_12	fun 15	fun 1	~~fun 1~~	~~fun 1~~	~~fun 1~~	~~fun 1~~	~~fun 1~~	~~fun 1~~	~~fun 1~~
fd[0]	3	3	3	~~3~~ 3	3	3	3	3	3
fd[1]	4	4	4	4	4	4	4	4	4
i		ø ~~1~~ 2	ø ~~1~~ 2	1 2	~~1~~ 2	2	2	2	2
a	11	0 ~~12 13~~ 15	ø ~~14~~ 16	ø 17	ø 18	0	0	0	0
motivo									
quien									
miPID		11	12	13	14	15	16	17	18
i									
n									
desc. 0	X	X	X	X	X	X	X	X	X
1	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X
3	t[0]	t[0]	t[0]	t[0]	t[0]	t[0]	t[0]	t[0]	t[0]
4	~~t[1]~~	t[1]	t[1]	t[1]	t[1]	t[1]	t[1]	t[1]	t[1]
5									



1seg → SENYAL(10, 12, 0, 0)
2seg → Yo soy 11 en nivel 0
Yo soy 12 en nivel 1
⋮
Yo soy 18 en nivel 3
12seg → He leido 15 → SENYAL(15, 12, 0, 3)
He leido 16 → " (16 3)
He leido 17 → " (17 3)
He leido 18 → " (18 3)
13seg → He leido 13 → " (13 2)
He leido 14 → " (14 2)
14seg → He leido 12 → " (12 1)
15seg → He leido 11 → " (11 0)

He leido 15
" 16
" 17 ← los está matando
" 18
" 13
" 14
" 12
" 11
SENYAL(11, 12, 1, 0)

He leido 11
SENYAL(11, 12, 2, 0) ← se cierran tuberías (n=0)
ha muerto 15 motivo 12
ha muerto 13 motivo 12
ha muerto 12 motivo 12
} sólo sus hijos (no los nietos)
acabo (11)

No quedan zombies, ya que o bien el padre ha hecho wait o bien el padre ha muerto, y por tanto, adoptados por INIT, INIT ha hecho wait

Problema Julio 2003

1. El listado de un directorio, es el siguiente:

```
SUID SGID PERMISOS    UID    GID    Nombre
  -    -  drwxr-xr-x julio  profe  .
  -    -  drwxrwxr-x root   staff  ..
  -    -  -rw-r-----  julio  staff  Notas
  -    -  -rw-r-----  pepe   profe  Trabajo
  -    -  -rw-------  julio  staff  Examen
  -    -  ----------  pepe   profe  F1
  1    -  -r-xr-xr-x  root   profe  rm1
  -    -  -r-xr-xr-x  julio  profe  rm2
  1    1  -r-xr-xr-x  julio  profe  rm3
  -    -  -r-xr-xr-x  julio  profe  cat1
  -    1  -r-xr-xr--  julio  profe  cat2
  1    1  -r-xr--r--  pepe   staff  cat3
  -    -  -r-xr-xr-x  julio  profe  mv1
  1    1  -r-xr--r--  pepe   staff  mv2
  -    -  -r-xr-xr-x  julio  profe  chmod1
  1    -  -r-xr-xr-x  julio  profe  chmod2
  -    -  -r-xr-xr-x  julio  profe  cp1
  1    -  -r-xr-xr--  julio  alumn  cp2
  1    1  -r-xr-xr-x  root   profe  cambia1
  -    1  -r-xr-xr-x  julio  staff  cambia2
```

Suponiendo que los programas **rm***, **cat***, **mv***, **chmod*** y **cp*** son copias de las ordenes estándar de UNIX, y **cambia*** es un programa que modifica el fichero que se le pasa como parámetro, suponer que el usuario **pepe** del grupo **alumn** accede a este directorio e intenta ejecutar las líneas de órdenes que se especifican (suponer que antes de ejecutar cada una el directorio siempre se encuentra en el estado inicial).
En el directorio /tmp todos los usuarios tienen permiso de escritura.
Marcar con un círculo **V** si la orden se lleva a cabo con éxito y **F** si no puede realizarse.

2 ptos:

acertada	**+0.1,**
fallada	**-0.1,**
no contestada	**0**

(V)	F	chmod1 444 F1
V	(F)	cambia2 Notas
(V)	F	cp1 Trabajo $HOME/Trab2
(V)	F	chmod2 444 Notas
V	(F)	cat1 Notas
V	(F)	cp2 Examen ../Exam2
V	(F)	cat3 Notas > ../Notas2
V	(F)	mv1 Trabajo Trab2
(V)	F	cambia1 Trabajo
V	(F)	mv2 Examen /tmp/Exam2

(V)	F	cambia1 Notas
V	(F)	cp2 Trabajo T2
V	(F)	chmod2 F1
(V)	F	rm1 Trabajo
V	(F)	cp1 Notas Notas2
(V)	F	rm3 Trabajo
V	(F)	mv1 Notas Notas2
V	(F)	rm2 Examen
V	(F)	chmod1 777 Notas
(V)	F	cp2 Examen /tmp/examen

Problema permisos: Julio 2003

- ¿qué permisos requiere cada orden?

rm: requiere permiso de escritura en el directorio
cat: requiere permiso de lectura del archivo
mv: requiere permiso de escritura en el directorio origen y destino
chmod: requiere que yo sea el propietario (nada que ver con permisos) ⚠
cp: requiere permiso de lectura en el origen y de escritura en el destino

- cuando hay redirecciones, se hacen antes de procesar las órdenes, y necesita permiso de escritura en el directorio destino o lectura en el archivo de entrada.
- si hay tuberías, tienen que cumplirse los permisos de ambas órdenes

chmod2 444 F1 :
- tengo permiso para ejecutar chmod? si
- setuid : asumo personalidad de julio
- ¿soy (yo, ahora julio) propietario de Notas? Si
  Si se puede

cambia1 Notas :
cambia2 Trabajo
- puedo ejecutarlo (todos pueden)
- setuid + setgid : soy julio, profe
- puedo leer y escribir notas (soy julia)

cp2 Trabajo T2 :
- tengo permiso para ejecutar
- setuid + setgid → soy pepe, staff
- requiero escribir en directorio actual (ni pepe ni staff puede)
- requiero leer trabajo : pepe puedo

cp1 Trabajo /tmp/Trab2 :
- tengo permiso para ejecutar
- setuid + setgid → soy julio, profe
- puedo leer trabajo y escribir en tmp

mv2 Examen /tmp/Examen:
- puedo ejecutar mv2 (soy grupo alumno)
- setuid → soy julio
- puedo escribir en directorio actual ya que soy julio

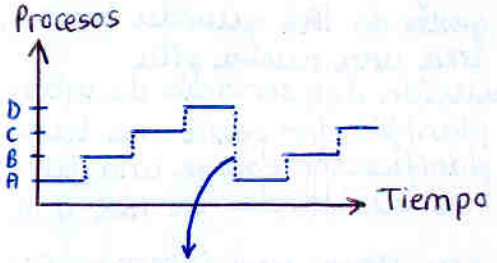
UT2. Gestión de Procesos

Tema 3. Planificación de Procesos

1. Introducción

1 CPU = 1 Proceso en ejecución

Vamos saltando entre programas secuencialmente



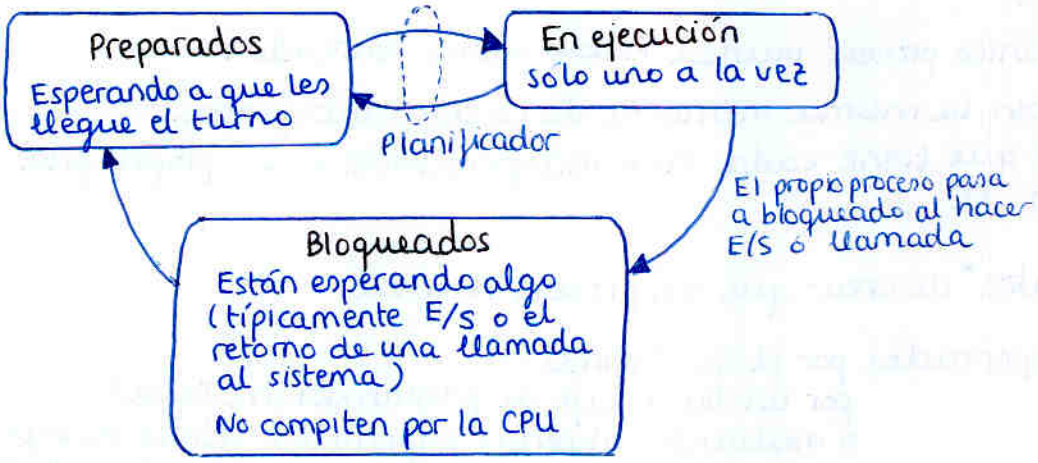
Cambio de contexto

Al cambiar de proceso hay que guardar todo el contexto del anterior (PC, registros, ...) y cargar el nuevo contexto.

1.1 El planificador (scheduler)

- Algoritmo que decide qué proceso pasa a ejecución
- se diseña para que la elección sea la mejor posible
- El planificador también consume CPU

1.2 Estados de un proceso



1.3 Tipos de planificadores

2 grandes grupos:

- No expulsivos: el planificador no puede sacar a un proceso de ejecución. El proceso debe salir por su propio pie (por programación para ello o por bloquearse esperando algo)
  ej: Windows 3.11 usaba planificador no expulsivo

- Expulsivos: pueden sacar a un proceso de ejecución y dejarlo en "Preparados"

  Nota: lógicamente, en el caso no expulsivo, el proceso que salga de estado "En ejecución" irá siempre a "Bloqueados", nunca a "Preparados"

1.4 Interrupciones y Procesos

Para hacer un cambio de contexto se hace una interrupción, tras la cual

HW: Pone en la pila el PC (contador de Programa) y otros registros de la CPU
HW: Se carga un nuevo PC
SW: El gestor de IRQ guarda los registros de la CPU en la pila
SW: Se crea una nueva pila
SO: Ejecución del servicio de interrupciones
SO: El planificador pasa una tarea a "preparado"
SO: El planificador escoge una tarea para su ejecución
SO y HW: Se vuelve de la IRQ y se arranca el proceso

Cada vez que llega una interrupción (por ejemplo de E/S) se hacen estos pasos, ya que la ejecución podría desbloquear a alguien que estaba esperándola.

1.5 Threads o hilos (procesos ligeros)

La llamada fork crea un proceso completo (autónomo e independiente, con su propia entrada en la tabla de procesos, su propio environ, etc...) sin embargo a veces puede interesar que un mismo programa haga algo en paralelo sin tener que crear un nuevo proceso
→ Ello se consigue con los threads

- Dentro de un único proceso pueden haber varios threads:
  - comparten la misma memoria de código, datos, etc...
  - lo único que tiene cada hilo independiente es su propia pila y propio PC

- Son más "baratos" de crear que un proceso normal

- Pueden ser soportados por el SO (Win32)
  por un lenguaje de programación (Java)
  o mediante librerías especiales (Linux no soporta threads a menos que uses la librería pthreads)

- Nuevos problemas de planificación de recursos (el planificador tendrá que ser más complejo)

2. Tipos de Procesos

- batch o por lotes
  - no requieren intervención del usuario
  - mucho uso CPU Poca E/S

  ej: cálculo largo, simulación
- interactivos
  - requieren intervención del usuario
  - entre acciones del usuario están bloqueados esperando
  - el usuario tiene que esperar desde que hace una acción hasta que surte efecto (intentar que no sea molesto)

  ej: Word
- de tiempo real
  - se deben cumplir unas restricciones temporales

Todos los procesos realizan E/S; por tanto los procesos se ejecutan en ráfagas de CPU y E/S

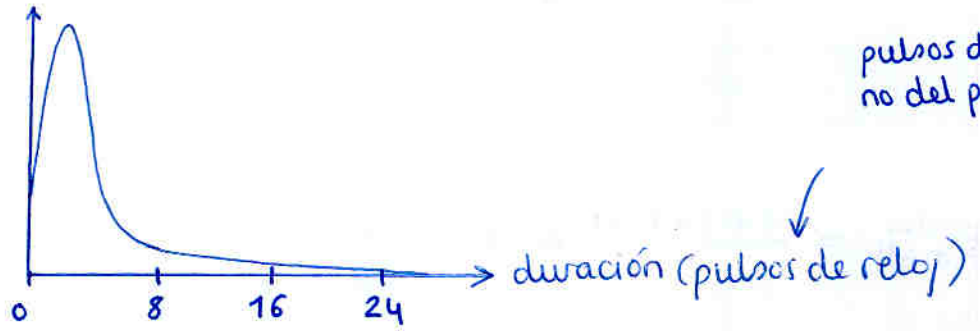
Proceso orientado a CPU: E/S | CPU | E/S | CPU ··

Proceso orientado a E/S: E/S | CPU | E/S | CPU | E/S | CPU | E/S ···

Nota: las ráfagas de E/S en la práctica duran MUCHISIMO más que las de CPU, por tanto en lo que hay que fijarse es en la frecuencia y duración de las ráfagas de CPU

Análisis estadístico de procesos

Histograma duración ráfagas de CPU



pulsos de reloj del SO, no del procesador

Las ráfagas de CPU duran casi siempre menos de 8 pulsos de reloj antes de bloquearse

3. Tipos de Planificación

Criterios:

- Usuario → respuesta aplicaciones
  - tiempo de respuesta
  - tiempo de retorno
  - previsibilidad
- Sistema → utilización eficiente de recursos
  - productividad
  - utilización de CPU
  - tiempo de espera
  - tiempo de respuesta normalizado
- Otros
  - equidad
  - importancia de ejecución de los procesos

ejemplo: desde que pulso una tecla hasta que la veo en pantalla

tiempo transcurrido desde el inicio del proceso hasta su finalización

tiempo de respuesta o de retorno (según sea interactivo o no) dividido por el tiempo necesario para la ejecución del proceso (i.e. lo que pierdo por estar en un S.O. multiproceso)

tiempo en la cola de preparados

3.1 Primero en llegar (First come first served, FCFS)

- El más simple que se puede implementar
- No expulsivo (aunque podríamos implementar versión expulsiva)

Desventajas:

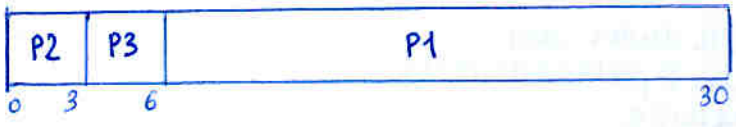
- No optimiza el tiempo de espera
- No es adecuado para sistemas interactivos (una ráfaga de CPU larga provoca espera larga a otros usuarios)

ejemplo

P1, P2, P3
↓ ↓ ↓
24 3 3

$$\text{Tiempo de espera medio} = \frac{0 + 24 + 27}{3} = 17$$

P2, P3, P1



$$\text{Tiempo de espera medio} = \frac{0 + 3 + 6}{3} = 3$$

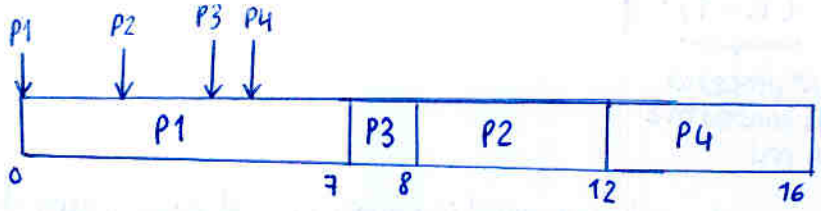
El tiempo de espera sería óptimo en este caso, pero sólo porque ha dado la casualidad de que han llegado en el buen orden

3.2 Primero el más corto (Shortest Job First SJF ó Shortest Process Next SPN)

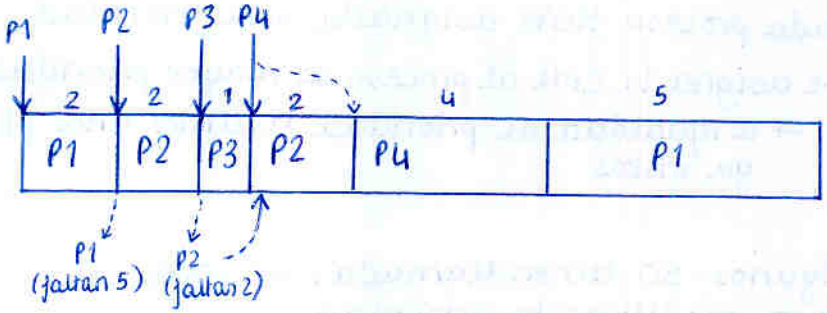
- se necesita una estimación de cuanto tiempo necesita cada proceso

ejemplo:

Proceso	Duración
P1	7
P2	4
P3	1
P4	4



Existen versiones expulsivas del algoritmo que mejoran las prestaciones
- ejecutan al que menos tiempo le falta para acabar
- lo comprueban cada vez que entra uno nuevo



Ventajas: Optimiza el tiempo de espera

Inconvenientes:
- Tiempo del siguiente intervalo de CPU difícil de predecir
- Posibilidad de inanición: los trabajos largos no se ejecutarán nunca mientras haya trabajos cortos

3.3 Turno Rotatorio (Round Robin RR)

- Algoritmo expulsivo
- Trata de minimizar el tiempo de respuesta
- se basa en turno rotativo y un quantum de tiempo q
  - q ↑: tiende a FCFS
  - q ↓: sobrecarga por cambios de contexto

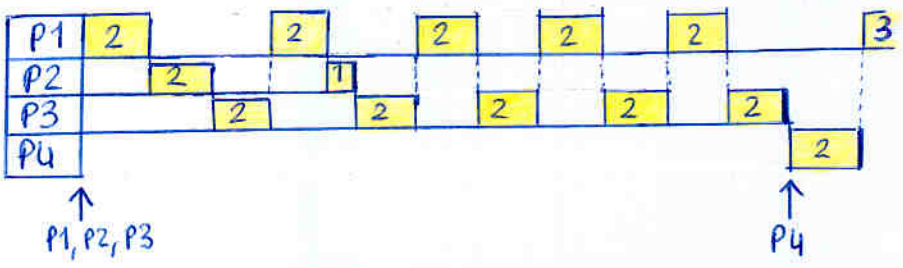
regla práctica: que el 80% de las ráfagas de CPU sean inferiores a q

- un proceso está en ejecución un quantum entero (a no ser que se bloquee por E/S)
- Si hay n procesos, cada uno obtiene 1/n del tiempo de CPU en intervalos de q unidades como máximo

ejemplo:

Proceso	Duracion	T.llegada
P1	13	0
P2	3	0
P3	10	0
P4	2	19

q=2



Propiedades de Round Robin

- Equitativo
- Tiempo de espera máximo

$$(n-1)\cdot q$$

($n-1$: nº procesos sin contarme a mi)

- Tiempo de retorno medio varía con el quantum de tiempo

$q \uparrow$ → Degenera en FCFS
$q \downarrow$ → sobrecarga por cambios de contexto

Peores prestaciones que el SJF con expulsión, pero no existe la posibilidad de inanición

3.4 Prioridades

- Cada proceso tiene asignada una prioridad
- se asigna la CPU al proceso de mayor prioridad
  → a igualdad de prioridad usamos una planificación como las ya vistas

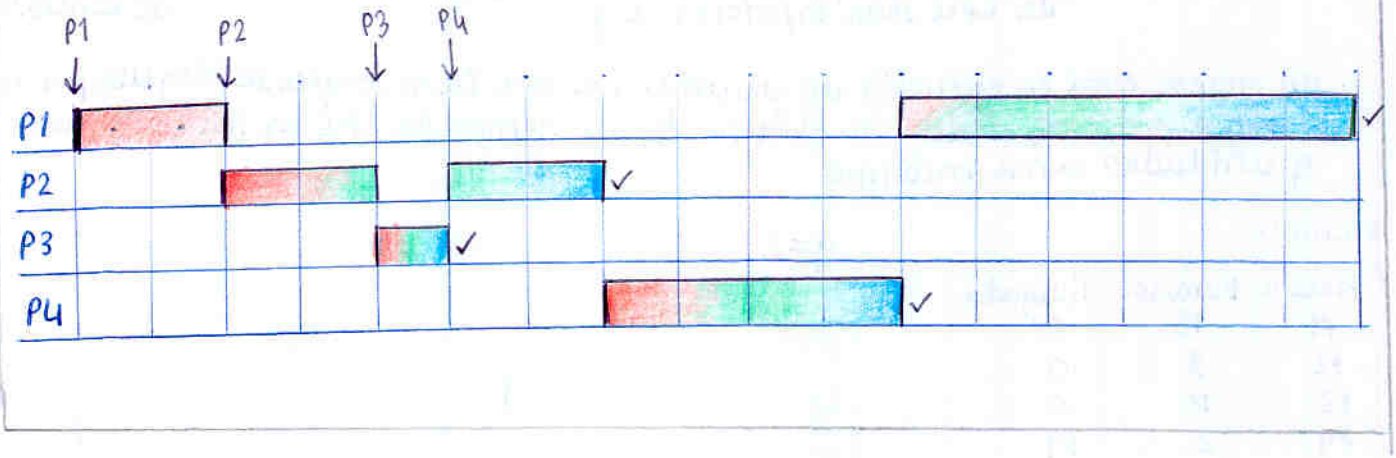
- Algunos SO tienen llamadas al sistema para modificar la prioridad
- Puede ser con expulsión o sin expulsión
- Puede provocar inanición

ejemplo

nice n ↑ baja la prioridad en n; n puede ser negativo

ejemplo: prioridades con expulsión

Proceso	T. llegada	Duracion	Prioridad
P1	0	7	5
P2	2	4	10
P3	4	1	15
P4	5	4	10



3.4 Prioridades dinámicas

Todos los procesos tienen asignada una prioridad base y una efectiva;
la prioridad efectiva se calcula en función de la prioridad base y puede cambiar debido a:

- El proceso hace mucho tiempo que no se ejecuta (sistema de actualización de prioridades)
- El proceso se ha ejecutado durante mucho tiempo
- Las características del proceso cambian
- Evitar que procesos de baja prioridad bloqueen otros de más alta prioridad (exclusividad de recursos: ejemplo: proceso de baja prioridad quiere usar E/S y cuando empieza a usarla llega un proceso de alta prioridad que pasa a ejecución quitando al de baja prioridad; si ahora el proceso de alta prioridad quiere usar esa misma E/S es posible que no pueda porque el recurso está en uso por el proceso de baja: se llega a un bloqueo donde ambos procesos necesitan esperar a que el otro acabe para continuar.)

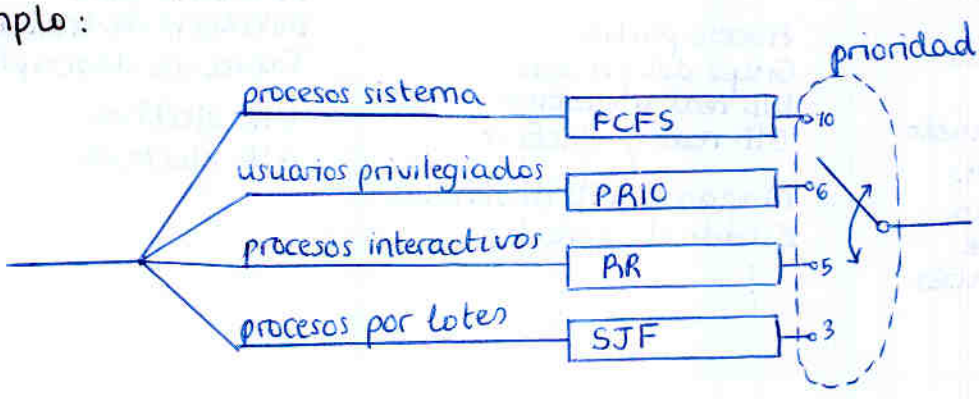
3.5 Colas múltiples

Los procesos se agrupan en distintas colas según tipo y características.

Entonces aparecen dos niveles de planificación

- La planificación de qué cola debe provenir el proceso a ejecutarse (típicamente prioridades: colas más prioritarias que otras)
- La planificación de qué elemento dentro de una cola debe ser el primero (la planificación puede ser distinta para cada cola)

ejemplo:

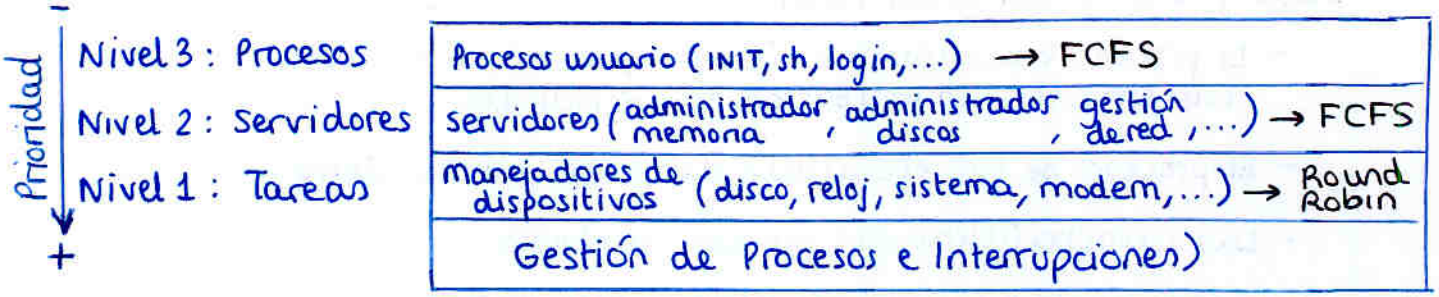


nota: si un nivel más privilegiado saca de CPU a un Round-Robin que aún no había acabado sus 'quantum' de tiempo; cuando vuelva los hará todos otra vez (i.e. no recuerda los que hizo, sino que se resetea al salir de CPU)

4. Planificación en MINIX 2.0

4.1. Planificación

Utiliza colas múltiples



ejemplo: un proceso de usuario hace una llamada al SO, lo cual despierta a un servidor el cual entra al procesador por ser de más nivel que cualquier proceso de usuario, los cuales se bloquean.

Si ahora el servidor quiere hacer uso de algún dispositivo, despierta a un manejador y por tanto se bloquea; el manejador pedirá, por ejemplo, una lectura al disco, y mientras el disco lee se bloqueará esperando una interrupción; entonces puede volverse a algun otro proceso de usuario que esté esperando.

Cuando el disco termina la lectura → interrupción, el driver (manejador) se desbloquea, termina su tarea y se bloquea, dando paso al servidor, que terminará su tarea y se bloqueará, entonces el proceso de usuario podrá continuar (si le toca a él según FCFS) habiendo realizado la llamada al sistema.

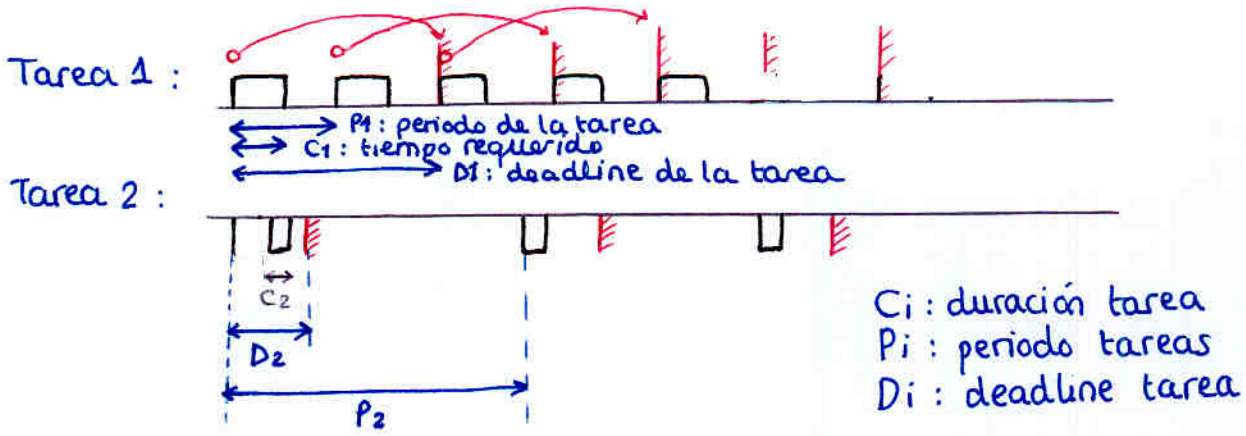
4.2 Tabla de procesos

Información que se tiene de un proceso

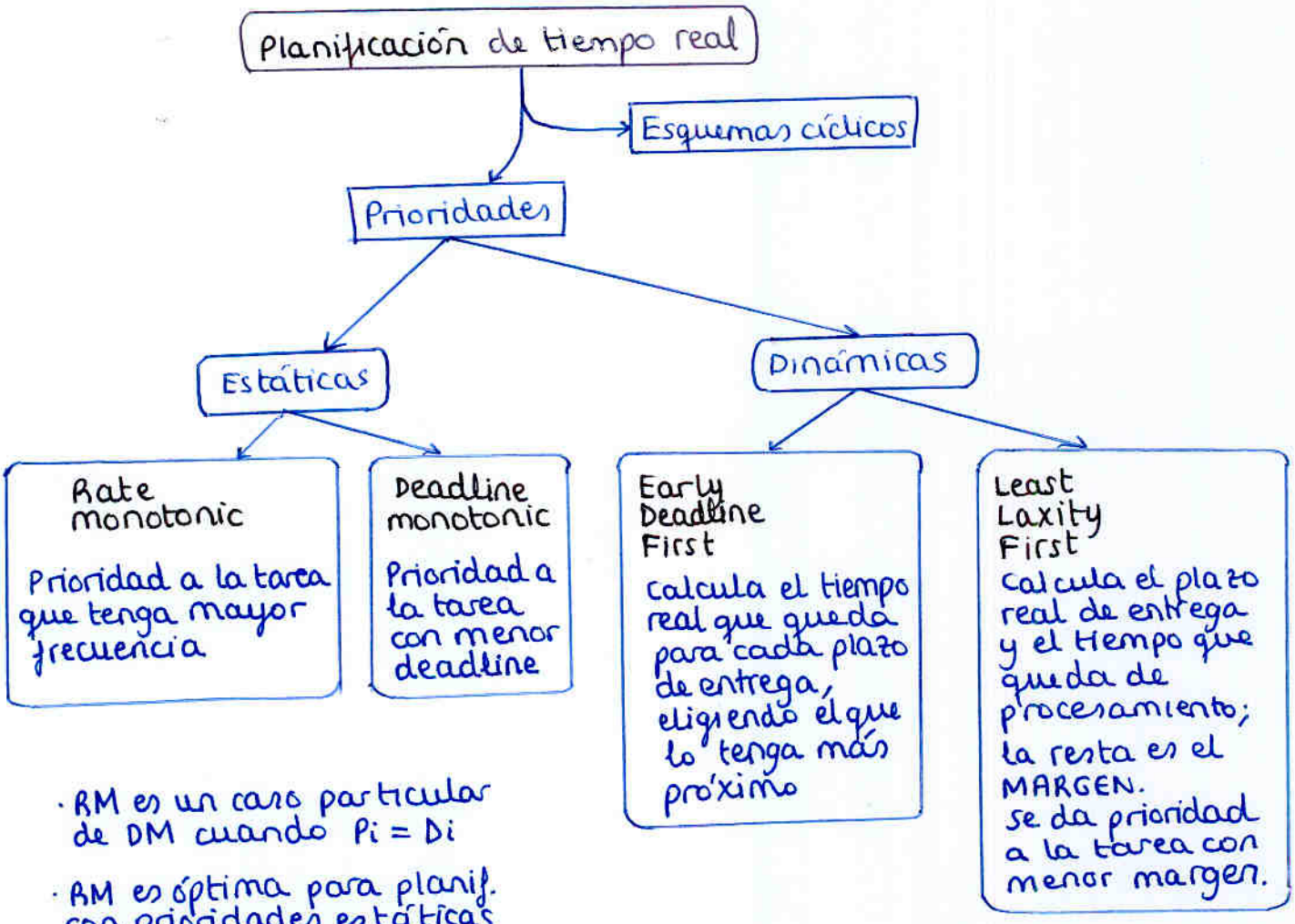
Nucleo	Sist. memoria	Sist. archivos
PID Registros PC (cont. programa) Palabra Estado Puntero de pila Estado del proceso Tiempo de inicio Tiempo de CPU usado T. de CPU de hijos Tiempo de alarma Cola de mensaje Señales pendientes ⋮	PID Mapa de memoria Estado de finalización Proceso padre Grupo del proceso UID real y efectivo GID real y efectivo Mapas de bits de señales Estado de señales ⋮	PID Máscara UMASK Directorio raíz Directorio de trabajo Tabla de descriptores UID efectivo GID efectivo ⋮

5. Planificación de Tiempo Real

supondremos que todas las tareas son periódicas



Ci : duración tarea
Pi : periodo tareas
Di : deadline tarea



- RM es un caso particular de DM cuando $P_i = D_i$
- RM es óptima para planif. con prioridades estáticas

Tema 4. Programación Concurrente

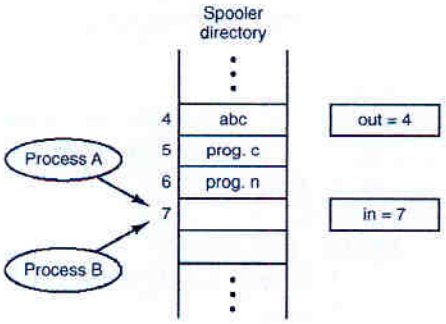
1. Problema de comunicación de procesos. Soluciones:
- *1. Inhibición de Interrupciones*
- *2. Espera Activa*
- *3. Espera con bloqueo*

2. Implementación en MINIX

3. Ejemplo de paso de mensajes: PVM
- *1. Utilización de la PVM*
- *2. Aplicación paralela: FRACTALES*

1. Problema comunicación de procesos

El problema aparece cuando existen accesos simultáneos un mismo recurso (espacio de memoria o dispositivos), y alguno es de escritura.



Ejemplo: 'A' va a escribir en 7 pero justo entonces le interrumpe 'B' que también escribe en 7; cuando 'A' vuelva, sobreescribirá lo que ha escrito 'B'

Región critica: parte del código en el que se realizan accesos a un recurso compartido.

Soluciones

Condiciones para obtener una buena solución:
- Dos procesos no pueden estar a la vez dentro de su región crítica.
- Ningún proceso que se encuentre fuera de una región critica puede bloquear a otros procesos.
- Ningún proceso deberá esperar indefinidamente para entrar en su región crítica.
- Velocidad de proceso y nº de CPUs desconocidos.

1.1 Inhibición de interrupciones

Cuando un proceso entre en región crítica, inhibimos las interrupciones.
Si se deshabilitan las interrupciones el SO no puede cambiar el proceso en ejecución.
- No es útil en sistemas multiprocesadores.
- Confiere a los procesos de usuario la posibilidad de quedarse con la CPU.
- Determinadas IRQs no se pueden inhibir.

En la práctica, esto lo realiza a veces el SO de forma controlada

1.2 Espera Activa

- Variables candado (SW + HW)
- Alternancia estricta (SW)
- Solución de Peterson (SW)

Espera activa: Variables candado:

Cuando entro en sección crítica:
Si candado=0; entro y 'pongo el candado' (a la vez → instrucción indivisible)
Si candado=1; me quedo esperando mirando el candado (while) (gasto CPU)

Nadie debe poder interrumpirme desde que entro hasta que pongo el candado; necesito indivisibilidad:
Se necesita que el procesador tenga intrucción indivisible **Test and Set Lock (TSL)**
No sólo indivisible por interrupciones, sino también en un mismo ciclo de bus

```
enter_region:
      tsl R1, lock_var //copia lock_var a R1 y pone lock_var a 1
      cmp R1, #0 // ¿R1=0? == ¿lock_var ERA 0? (ahora mismo es 1)
      jne enter_region // si lock_var ERA 1 (por tanto tsl lo ha
                           dejado igual y la zona crítica
                           está ocupada) sigo esperando
            // si no era 1, ya lo he puesto yo a 1, y sigo hacia
               la ejecución de la zona crítica
      ret

leave_region:
      move lock_var, #0
      ret
```

Espera activa: Alternancia estricta:

<u>En el proceso A:</u>

```
while(TRUE){
      while(turn!=0); // me quedo esperando
      region_critica();
      turn=1;
      otro_codigoA();
}
```

<u>En el proceso B:</u>

```
while(TRUE){
      while(turn!=1);
      region_critica();
      turn=0;
      otro_codigoB();
}
```

La mecánica es buena, nunca puede fallar.
Desventaja: el acceso a la región crítica solo puede hacerse alternándose A y B;
A nunca puede hacer dos accesos seguidos si entre medias no accede B, lo cual puede ser un gasto de tiempo si B está haciendo otras cosas y A quiere acceder una 2ª vez.

Espera activa: Peterson:

Algoritmo del año 81, generalizable a N procesos. Algoritmo software que funciona bien y cumple todos los requisitos sin usar ningún truco hardware.

Si el otro no está interesado: siempre puedo yo

Si ambos están interesados: sistema puro de turnos

```
#define FALSE 0;
#define TRUE 1;

int turn; // quien tiene el turno?
int interested[2]; //todos a 0 (FALSE)

void enter_region(int process) { // process = 1 ó 0
      int other;
      other = 1 - process;
      interested[process] = TRUE;
      turn = process; // set flag (ya me he asignado el turno
                           para cuando acabe el actual)
      while ((turn == process) && (interested[other] == TRUE))
            // null statement (esperar a que el otro acabe y
                 ponga interested[other] = FALSE)
}

void leave_region(int process) {
      interested[process] = FALSE;
}
```

• **Principales problemas de espera activa:**

- Los procesos que no pueden entrar en una sección crítica siguen consumiendo tiempo de CPU.

- Inversión de prioridad. Procesos de alta prioridad que están simplemente esperando por sección crítica (debido a que está siendo usada) no dejan ejecutarse a otros procesos que harían cosas más útiles que esperar.

• Como solución se proponen esquemas en los que los procesos que quieren entrar en una región crítica son bloqueados hasta que pueden entrar en la misma.

1.3 Espera con bloqueo

ejemplo
- 2 procesos:
- productor (producir + llevar al almacén)
- consumidor (coger del almacén + consumir)
- acceso al almacén = sección crítica (i.e. no pueden verse dentro)
- el almacen tiene capacidad limitada

Dos circunstancias bloquean al productor:
- almacen está lleno
- consumidor está dentro del almacén

Dos circunstancias bloquean al consumidor:
- almacén está vacío
- productor está dentro del almacén

```
#define N 100 // number of slots in the buffer
int count = 0; //number of items in the buffer

void producer(void) {
      while (TRUE) { // repeat forever
            produce_item(); // generate next item
            if (count == N) { // almacén lleno
                  sleep();
                  // if buffer is full, go to sleep until consumer wakes you
                  //Problema: si desde que veo count==N hasta que
                  //hago sleep() cambia N, el consumidor ya no me despierta
            }
            enter_item(); // put item in buffer
                        // suponemos que el código de enter_item() ya
                        //incluye dormir si el otro está dentro
            count = count + 1; // increment count of items in buffer
            if (count == 1) //was buffer empty? (just filled)
                  wakeup(consumer);
      }
}

void consumer(void) {
      while (TRUE) {
            if (count == 0)
                  sleep(); // if buffer is empty, got to sleep
            remove_item(); // take item out of buffer
            count = count - 1; // decrement count of items in
                        buffer
            if (count == N-1)
                  wakeup(producer); // was buffer full?
            consume_item(); // print item
      }
}
```

- Esquemas más utilizados :
  - Semáforos. Normalmente en sistema operativo.
    ejemplo: en Unix, Linux, Windows...
  - Monitores. Normalmente necesitan un soporte del lenguaje de programación.
    ejemplos: Ada, Java...
  - Paso de Mensajes.

- Teniendo uno, se pueden simular los otros.

1.3.1 Espera con bloqueo: Semáforos

• Dos primitivas atómicas (no divisibles) (ejemplo: implementado por el SO) sobre una variable especial: un semáforo.
• Primitivas:
- SUBIR().
Incrementa en uno el valor del semáforo.
Procesos bloqueados pueden despertar.
A veces se llama Primitiva V(). (truco para recordar: Victoria)

- BAJAR().
Decrementa en uno el valor del semáforo.
Si el valor del semáforo era <= 0 el proceso se duerme hasta que otro proceso haga un SUBIR()
A veces se llama Primitiva P(). (truco para recordar: puede Pararte)

ejemplo: Sala con 3 ordenadores - semáforo inicialmente a 3
- cada vez que entra alguien baja el semáforo (BAJAR)
- según vaya entrando gente, se irá bajando el semáforo
- cuando quiera entrar uno pero el semáforo ya esté a 0, al intentarlo BAJAR se dormirá (se puede implementar que lo baje hasta números negativos para saber el número de procesos bloqueados, o que lo deje en 0)
- cada vez que alguien sale de la sala, hace SUBIR:
  - si no hay nadie esperando: sube el semáforo
  - si hay alguien en cola: despierta a uno para que entre a su sitio
    (no subo el semáforo a menos que se haya implementado el semáforo con números negativos)

ejemplo: Productor/consumidor implementado con semáforos

Recuerda, dos circunstancias bloquean al productor:
- almacen está lleno
- consumidor está dentro del almacén

dos circunstancias bloquean al consumidor:
- almacén está vacío
- productor está dentro del almacén

Para cumplir con todo ello se utilizan 3 semáforos:

- **semáforo empty** : número de huecos (permite bloqueo del productor al llegar a cero por estar almacén vacío)

- **semáforo full** : número de elementos (permite bloqueo del consumidor al llegar a cero por estar almacén vacío)

- **semáforo mutex** : es un semáforo binario que implementa la exclusión mutua (que no coincidan nunca a la vez dentro del almacen)

```
#define N 100 /* number of slots in the buffer */
semaphore mutex=1;
semaphore empty=N;
semaphore full =0;

void producer(void) {
      while (TRUE) { /* repeat forever */
            produce_item(&item); /* generate next item */
            down(&empty);
            down(&mutex);
            enter_item(item); /* put item in buffer */
            up(&mutex);
            up(&full); // CUIDADO con el orden
      }

void consumer(void) {
      while (TRUE) {
            down(&full);
            down(&mutex);
            remove_item(&item);
            up(&mutex);
            up(&empty);
            consume_item(item); / * print item * /
      }
}
```

Fijate en la elegancia de los semáforos; la única condición para poder usarlos es que existan las primitivas indivisibles UP y DOWN; proporcionadas normalmente por el SO o por el lenguaje de programación (ejemplo: JAVA)

1.3.2 Espera con bloqueo: Monitores

Permite "abstraerse" de los problemas de la zona critica.

Usar un nuevo tipo de datos : **Monitor**
- Incluye datos y código, siendo los datos accesibles sólo a través del código asociado.
- Hay exclusión mútua de acceso al código del monitor, y por tanto cualquier proceso que haga uso de una función interna de un monitor tiene garantizada la exclusión mutua
- Tienen variables especiales a las que se asocian colas de procesos bloqueados.
- Primitivas WAIT y SIGNAL.

ejemplo: Productor/consumidor implementado con monitor

```
// Primero definimos al monitor

monitor ProductorConsumidor

      // Variables internas

      integer count;
      condition full, empty; // causa por la cual haces wait o signal

      // Funciones internas

      proc enter(datatype item)
      begin
            if(count=N) then wait(full); // dormir por la causa "full"

            enter_item(item);
            count:=count+1;
            if(count=1) then signal(empty); // despertar a quien
                              // estuviera dormido por la causa "empty"
      end;

      proc remove():datatype;
      begin
            if(count=0) then wait(empty);
            remove:=remove_item();
            count:=count-1;
            if(count=(N-1)) then signal(full);
                  // que el signal siempre esté al final de todo lo que
                        hagas, para que si el otro entra justo despues, yo
                        lo único que haga sea salir, sin molestar al otro
      end;

begin

      // Inicializacion del monitor
      count :=0;

end monitor;
```

```
// Ahora los procesos consumidor y productor son muy sencillos
// totalmente abstraidos del problema del acceso a un mismo
// recurso

Var ProductorConsumidor buffer;

// Proceso Productor
proc productor
begin
      while (true) do begin
            item=produce_item();
            buffer.enter(item);
      end
end

// Proceso Consumidor
proc consumidor
begin
      while (true) do begin
            item=buffer.remove();
            consume_item(item);
      end
end
```

1.3.3 Espera con bloqueo: Paso de mensajes

- Primitivas de comunicación explícitas:
  - Send(...): envía una información.
  - Receive(...): recibe información.
- Diferentes tipos según el canal de comunicación:
  bloqueantes, asíncronos, tamaño del buffer...
  - Sin buffer: rendez vous.
- Fáciles de portar a sistemas distribuidos y programación en red.
- Implementados en Sistema Operativo.

ejemplo: productor/consumidor con paso de mensajes

Además del paso de "cajas" desde el productor al consumidor, la capacidad limitada del almacén se logra "simular" con el el paso de "cajas vacías" del consumidor al productor
El productor no envía mensaje al consumidor hasta que no recibe una caja vacía.
El consumidor inicialmente envía 100 cajas vacias al productor, simulando así el almacén; además, cada caja llena que recibe el consumidor, la vacía y reenvía al productor.

```
#define N 100 // number of slots in the buffer

void producer(void) {
      item_type item;
      message_1 m;
      while (TRUE) { // repeat forever
            produce_item(&item); // generate next item
            receive(consumer,&m) // el productor no produce
                                 mientras no reciba cajas vacias
                                 del consumidor
            build_item(&m,item); // put item in buffer
            send(consumer,&m);
      }
}

void consumer(void) {
      item_type item;
      message_1 m;
      int i;
      for(i=0;i<N;i++) send(producer,&m); // Inicialmente envia
                                             100 cajas vacias
      while (TRUE) {
            receive(producer,&m); // Recibe la caja
            extract_item(&m,&item); // Vacia la caja
            send(producer,&m): // Devuelve la caja vacía
            consume_item(item); // print item
      }
}
```

2. Implementación en MINIX

- Utilización de mecanismo de paso de mensajes basado en primitivas bloqueantes
  (primitivas bloqueantes = rendez-vous = no hay buffer
  = si el que recibe no quiere recibir, el que envía se queda esperando)
- Primitivas:
  - Send(...)
  - Receive(...)
  - SendRec(...)
- Se permite intercambio de mensajes en la misma capa o con las capas adyacentes.
  - Las aplicaciones sólo entre ellas o con la capa de servicios.

3. Ejemplo de paso de mensajes: PVM

- PVM: Parallel Virtual Machine
  - Herramienta de programación paralela
  - Entorno de ejecución: multiprocesador o NOWs (red de estaciones de trabajo)
  - Programación paralela:

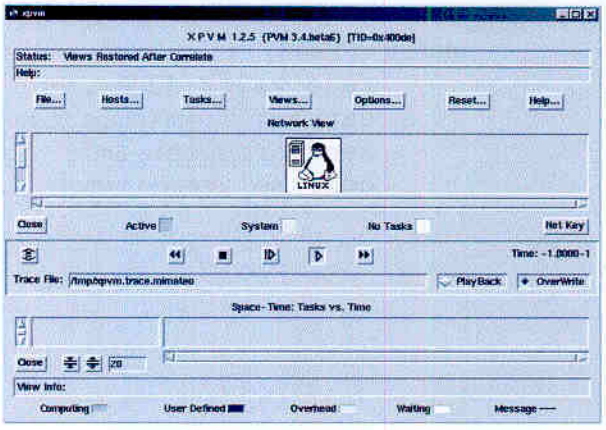
Dividir un problema (programa) en subproblemas (subprogramas) más o menos independientes. Cada uno será un proceso. Los distintos procesos (subprogramas) se reparten entre los distintos procesadores (ordenadores) del multiprocesador (NOW). Es necesaria comunicación y sincronización de procesos para resolver el problema conjuntamente.

- Características de PVM:
  - Comunicación por paso de mensajes.
  - **PVM Daemon** (servidor). Programa residente que ofrece los servicios de una máquina paralela. Debe estar en marcha, uno en cada máquina.

- Comunicación entre pvmd basada en TCP/IP
  - Librería. Permite a las aplicaciones pedirle al servidor local servicios: comunicación entre tareas, configuración, gestión, etc.
  - Se suele utilizar un protocolo de ejecución remota de aplicaciones (RSH, SSH, etc.) para arrancar los servidores en las máquinas de la NOW

3.1 Utilización de la PVM

- Arrancar un servidor PVM local mediante:
  - Llamadas a librería desde una aplicación. (i.e. la propia aplicación arranca el demonio)
  - Consola de la máquina virtual:
    - Modo texto (pvm)
    - Modo gráfico (xpvm)



La consola es un interfaz con el demonio para lanzarle ordenes (normalmente de configuración). Una vez en marcha y configurado el demonio, la consola se puede cerrar.

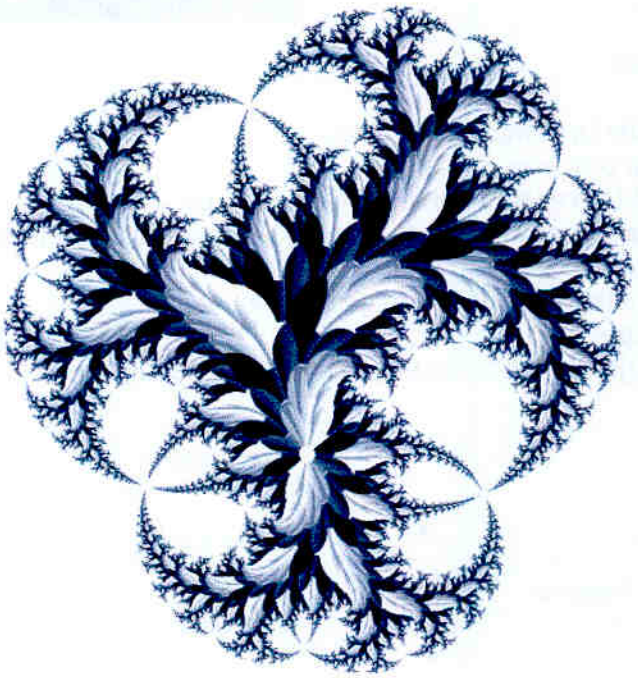
- Configurar la PVM
  - Añadir ordenadores, ver estado, etc.
  - Permite incluir máquinas de diferentes características
    (NOW o cluster heterogéneo)

- Ejecución de aplicaciones:
  - Desde la consola.
  - Desde una aplicación normal que "llama" a funciones de librería de PVM
    - Se auto-añade a la máquina virtual.
    - Utiliza los servicios de PVM para arrancar otras tareas (librería).
    - Las tareas se pueden repartir entre las máquinas de la PVM.
      - Comunicación entre tareas independiente de su localización

3.2 Aplicación paralela: Fractales

- Fractales:
  - Objetos geométricos cuya estructura básica se repite en diferentes escalas
    (B. Mandelbrot, 1975)
  - Definibles recursivamente.
  - Presentes en muchas estructuras de la naturaleza
  (hoja de helecho, caracol de mar, brócoli).

  - Usos:
    artísticos, compresión de datos (imágenes), biología, sociología, economía...



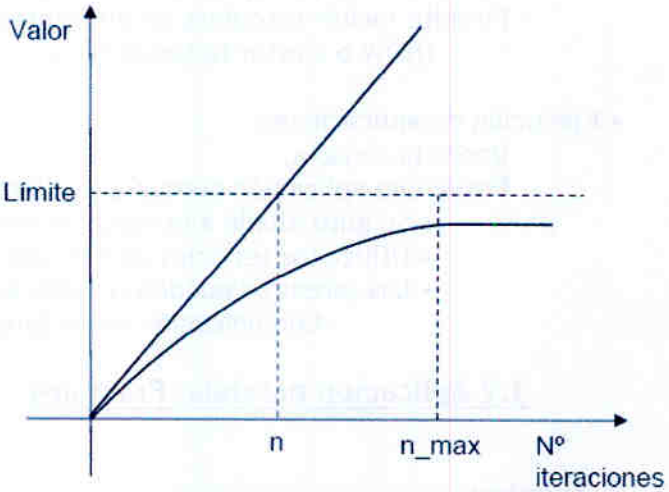
Nosotros vamos a usar un fractal muy conocido:

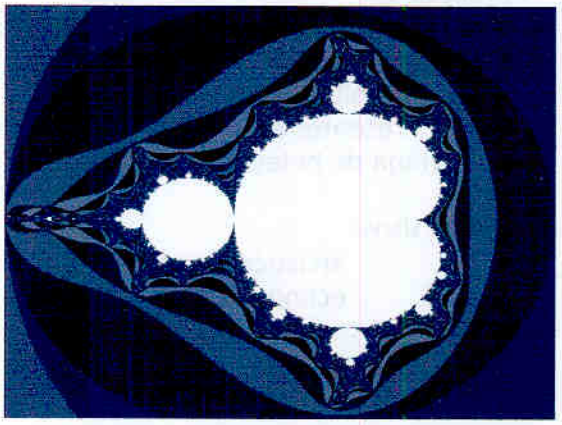
- **Conjunto de Mandelbrot :**

Para cada punto del plano complejo z = (x,y), se aplica iterativamente una cierta función compleja:

$$a = 2ab + y$$
$$b = a^2 - b^2 + x$$

Se cuenta el Nº de iteraciones n hasta que (a+b) sobrepase un umbral (Limit) o hasta que se llegue a un cierto número de iteraciones (n_max == n_col). De ésta forma a cada punto del plano complejo le corresponde un número entre [0, n_max].

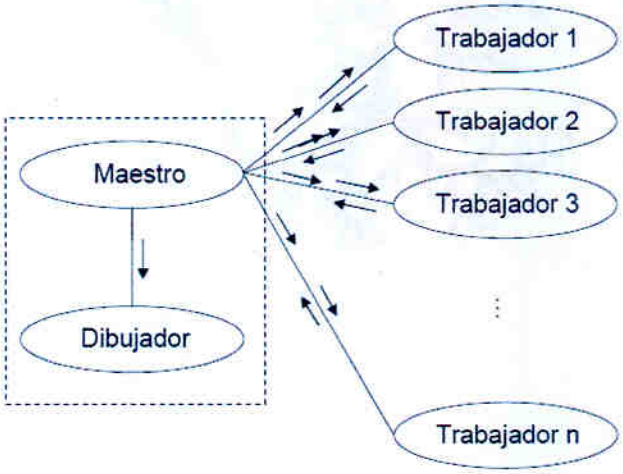




Se asigna un color a cada valor de [0, n_max] (blanco si se alcanza n_max). Ése será el color asociado a ese punto.

El cálculo de un punto es independiente del resto

- Aplicación paralela:
  - maestro.c:
    - Divide la imagen en bandas
    - Crea procesos de cálculo.
    - PVM los reparte entre los ordenadores.
    - Organiza el reparto de trabajo y recoge los resultados.
  - trabajador.c:
    - Calcula líneas de la imagen una a una.
  - dibujador.cpp:
    - Dibuja el gráfico en una ventana (opcional).



UT 3 : GESTIÓN DE MEMORIA

Tema 5. Multiprogramación

1. Conceptos básicos
   1.1 Monoprogramación
   1.2 Multiprogramación
      - Particiones fijas
      - Particiones variables
2. Estructuras de datos
3. Algoritmos de asignación de memoria
4. Intercambio
5. MINIX
   5.1 Gestión de Memoria
   5.2 Llamadas al sistema

1. Conceptos básicos

Un programa necesitará cierta cantidad de memoria; hay que ver de dónde se la damos.

Tamaño → Hay que repartir la memoria total entre los programas

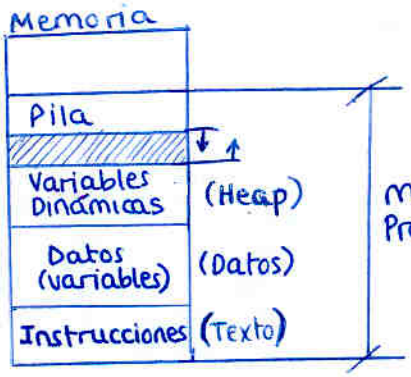
puede ser muy grande si uso memoria virtual

memoria: recurso escaso que hay que gestionar

En este tema nos olvidamos de ésto

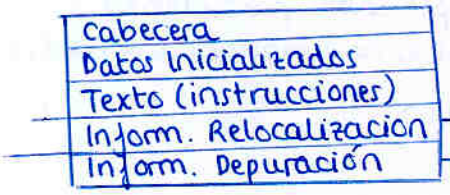
La memoria se usa para cosas diversas

Mapa de memoria de un proceso



A un sólo programa se le asigna un trozo FIJO de la memoria

El archivo ejecutable de un programa tiene:



1.1 Monoprogramación:

Sólo hay UN proceso: toda la memoria para un proceso no hay que gestionar nada

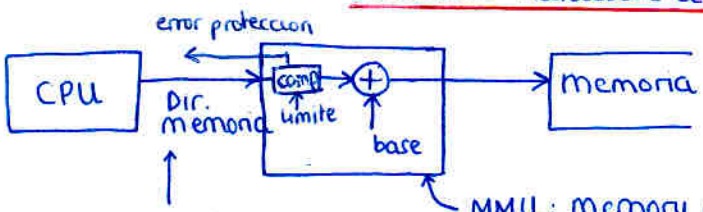
1.2 Multiprogramación

- varios procesos
- Hay que poder repartir la memoria entre ellos

Relocalización:
Permite que un programa pueda colocarse en cualquier lugar de la memoria; es una tabla que indica que partes del Texto hay que cambiar (allí donde se acceda a variables de forma absoluta)

opción -g
nombres de variables
incluir código fuente
⋮

solución hardware a la relocalización



Ya no hace falta la inform. de relocalización

El compilador pone las direcciones como si el programa comenzase en cero

MMU: Memory Management Unit
En la actualidad suele estar integrado en el procesador; sumar registro base.
Incluye también un comparador con la dirección máxima para controlar "errores de protección" (registro límite)

Hay dos formas de repartir la memoria:

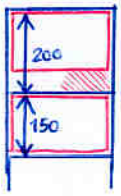
- Particiones fijas:
  Inicialmente divides la memoria en N particiones de los tamaños que quieras. Sólo permite la ejecución de N programas, cada uno se mete en la partición que más se ajuste → fragmentación interna

- Particiones variables:
  Ir dando la memoria 'a medida'
  Ir reservando y liberando → fragmentación externa

Fragmentación:

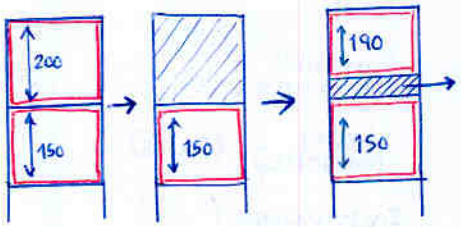
fragmentación interna:
Fragmento que no estoy usando de una partición asignada a mi
ej: partición de 300 ocupada por un programa de 280

ej:



fragmentación externa:
Fragmentos que se quedan por entre particiones (i.e. no asignados a nadie)

ej



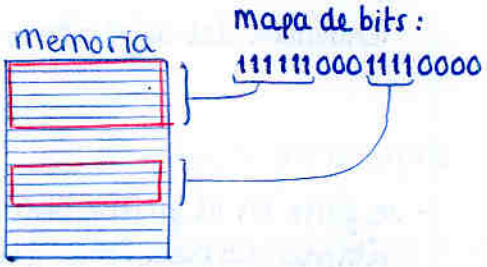
2. Estructuras de datos

Vamos a ver dos alternativas { - mapa de bits
- Listas enlazadas

· mapas de bits:

Primero divido la memoria en bloques

Reservo una zona de memoria donde cada bit representa un bloque
1 = ocupado
0 = libre

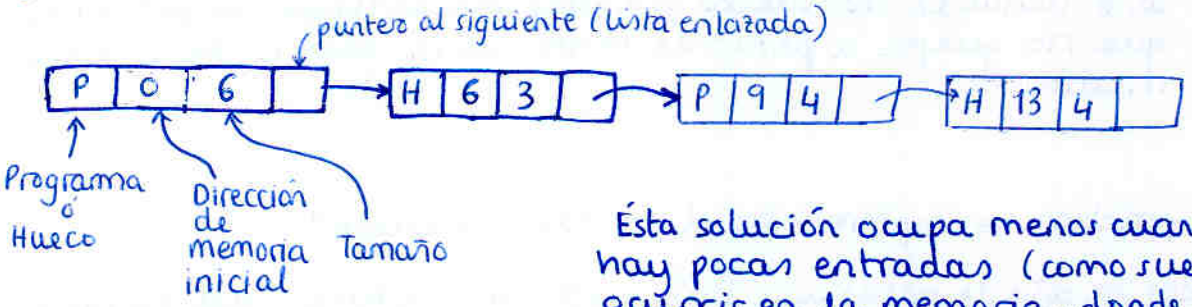


Si un programa pide 5 bloques, yo busco 5 ceros seguidos y le asigno los bloques asociados

Ventajas:
- El propio mapa de bits ocupa muy poco espacio (si cada bloque son 1000 bits, ocupamos una milésima parte de la memoria para el mapa de bits) (1 bit por cada 1000 bits)
- Es muy sencillo; los espacios libres se 'funden' automaticamente

· Listas enlazadas

Voy describiendo una por una las zonas de memoria



Se suele usar
- mapa de bits para el disco
- Lista enlazada para la memoria

Ésta solución ocupa menos cuando hay pocas entradas (como suele ocurrir en la memoria, donde hay pocos procesos) sin embargo si empiezan a haber muchos procesos se hace muy grande.

En realidad bastaría con almacenar los huecos

3. Algoritmos de asignación de memoria

- Primer hueco
  - Se pone en el primer hueco donde quepa
  - Es el algoritmo más rápido

  Tendencia del algoritmo: va haciendo pequeños los huecos del principio

- Siguiente hueco
  - se pone en el primer hueco donde quepa buscando desde el último donde me quedé (hacemos la lista enlazada circular)
  - Es igual de rápido
  - se ha visto que da las mismas prestaciones

- Mejor hueco
  - se pone en el hueco más pequeño en el que quepa
  - tiende a dejar huequitos inservibles
  - si la lista de huecos es ordenada por tamaño (fácil) el mejor hueco es igual al primer hueco

- Peor hueco
  - se pone en el hueco más grande que haya
  - lo que sobra suele ser servible
  - al ir llenando los huecos grandes puede llegar un programa grande que no quepa a pesar de tener varios huecos de tamaño medio

En la práctica se implementa el de "Primer hueco"

A veces se añade antes una búsqueda de un hueco del tamaño exacto al que se pide; pero esto sólo merece la pena si la probabilidad de que ésto ocurra es alta.

4. Intercambios

Podríamos plantearnos que cuando un proceso pase a estar bloqueado (esperando E/s por ejemplo, como vimos) le saquemos de la memoria (ésto es muy parecido a la memoria virtual pero a nivel más alto; no confundir los conceptos ya que pueden darse simultáneamente)

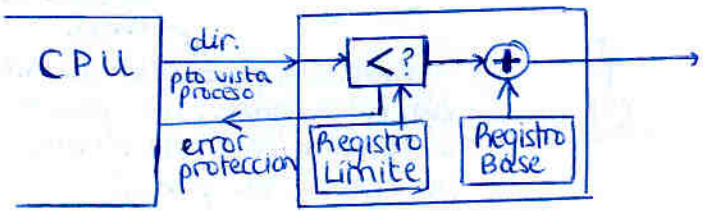
SWAPPING: permite arrancar más procesos de los que cabrían en memoria

Se le copia la memoria al disco; cuando quiera volver a ejecutarse tendrá que traerlo de vuelta

5. MINIX

5.1 Gestión de memoria

Utiliza lo de registro límite y registro base



el espacio que el SO ve como 'memoria' que puede repartir entre procesos, hacer swapping, etc... es en realidad la memoria virtual (q puede estar parte en discos, con lo cual el swapping podría estar sacando del disco para meterlo en el disco)

La memoria de un programa la divide en 2 trozos
- Texto (Instrucciones)
- Todo lo demás (Datos + Pila)

Llamada fork

- ¿Hay hueco en la tabla de procesos? si
- Buscar hueco en memoria para Datos+Pila
- Copiamos todos los Datos+Pila del padre al hijo (el programa lo comparten; cualquier dirección de retorno de subrutina que haya en la pila será válida)
- Se crea en la tabla de procesos una nueva entrada para el hijo, copiando inicialmente todos los valores de la entrada del padre
  - modifica entrada nueva con el mapa de memoria del hijo
  - modifica entrada nueva con el PID que corresponda al hijo
  - se actualiza nº de ref a entradas de la tabla FILP (lo veremos)
- Informa al Kernel y otros servidores
  - nuevo proceso creado
  - al Kernel se le da el mapa de memoria
- Enviamos mensajes (en plural, uno al padre y otro al hijo) de respuesta: al padre el PID del hijo y al hijo un cero

Llamada exec

Debe tener mucho cuidado y hacer muchas comprobaciones antes de machacar el programa antiguo (punto de no retorno)

parámetros: fichero, argv, envp

1. Verificar permisos del fichero
2. Leer cabecera de fichero: tamaño de código, pila y cola ← chequeos de si va a caber
3. Obtener argv y envp pasados a EXEC por P
4. Liberar memoria antigua y reservar la nueva → Punto sin retorno

↓

el algoritmo de comprobación puede suponer que la memoria actual está ya libre

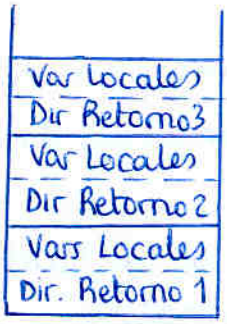
↙

En el caso particular de MINIX éste no se hace, ya que es complejo y poco probable (no merece la pena)

5. Crear una nueva pila con los contenidos necesarios para iniciar el programa
6. Copiar desde el fichero del programa los datos y el código
   (el código se podría compartir con otros procesos que ejecuten mismo código)
7. Gestionar SET_UID y SET_GID del fichero ejecutado
8. Modificar la tabla de procesos:
   - mapa de memoria nueva
   - eliminar capturas de señales
   - cambios en UID/GID

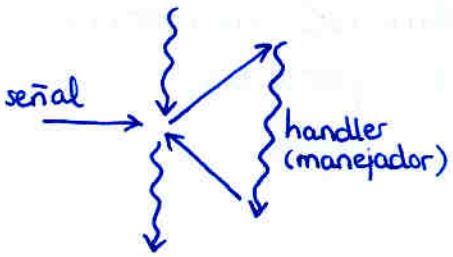
   Nota: no se renueva TODA la table, lo que se puede conservar se conserva
   ej: descriptores de ficheros abiertas
   ej: protecciones contra señales
9. Informar al kernel del nuevo mapa de memoria y de que el proceso puede pasar a preparado

Recuerda: Las sucesivas llamadas a subrutinas de un proceso se pueden 'ver' en la pila de un proceso, ya que antes de saltar se guardan en ésta las variables locales y dirección de retorno



De ésta forma al hacer RTS (return from subroutine) no hay más que leer de la pila el PC y variables locales
(justo lo que hacíamos en LAB. SED en el MC68000)

Señales

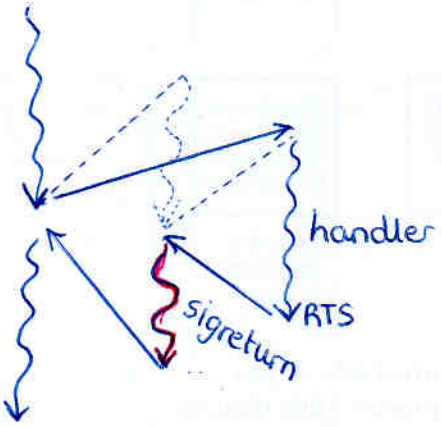


La ejecución principal de un programa no debe verse afectada al volver de un handler
i.e. hay que recuperar el entorno (registros CPU) y las variables locales que tuviéramos

¿Cómo se logra ésto?
Una señal siempre le llega al proceso mandada por otro proceso o por el sistema Operativo; en cualquier caso el proceso que la recibe no estaba en ejecución y por tanto cabe esperar que en su tabla de procesos tenga el entorno (registros de CPU), llamado Stackframe

Al llegar la señal, se guarda el stackframe de la tabla de procesos a la pila, y luego se hace un pequeño 'truco':

- Se escriben direcciones de retorno y variables en la pila, simulando que se han producido dos llamadas a subrutina que en realidad no se han hecho, pero a todos los efectos el proceso 'cree' que sí (de forma que sucesivos RTS irán haciendo que el proceso vuelva por subrutinas que en realidad no ha empezado)
- se pone el PC del handler en la tabla del proceso



Cuando le toque ejecutarse al proceso que le acaba de llegar la señal; se leerá el PC del handler, por tanto se ejecutará handler. Cuándo éste haga RTS al acabar, gracias al truco de la pila, se saltara a la función llamada SIGRETURN

Ésta rutina, que es del SO, se hace a sí misma una interrupción para poder estar en modo supervisor, y se encarga de todo lo necesario para volver al lugar original con todo en orden (copia el stackframe que se guardó en la pila a la tabla de procesos, y se hace un salto al lugar original)

NOTA: SIGRETURN no requiere hacer RTS sino que salta directamente al punto adecuado para continuar desde donde llegó la señal, por tanto el hecho de haber guardado la dirección de retorno en la pila se hace sólo por consistencia (por si un debugger quiere trazar las llamadas)

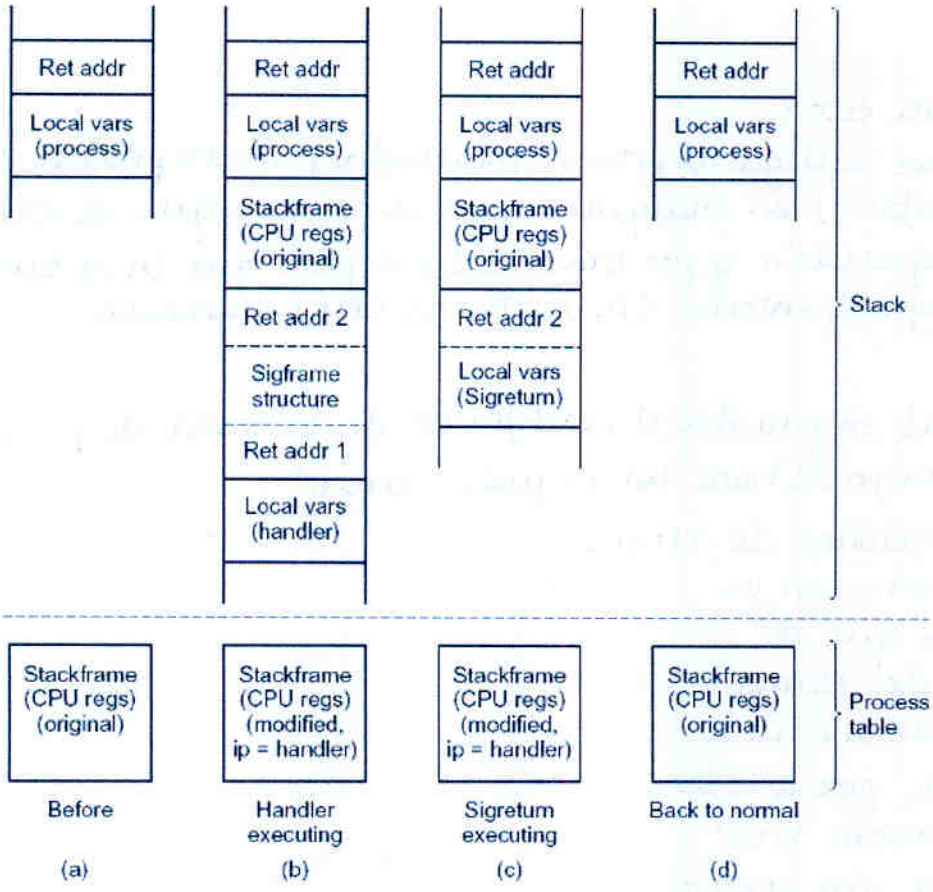
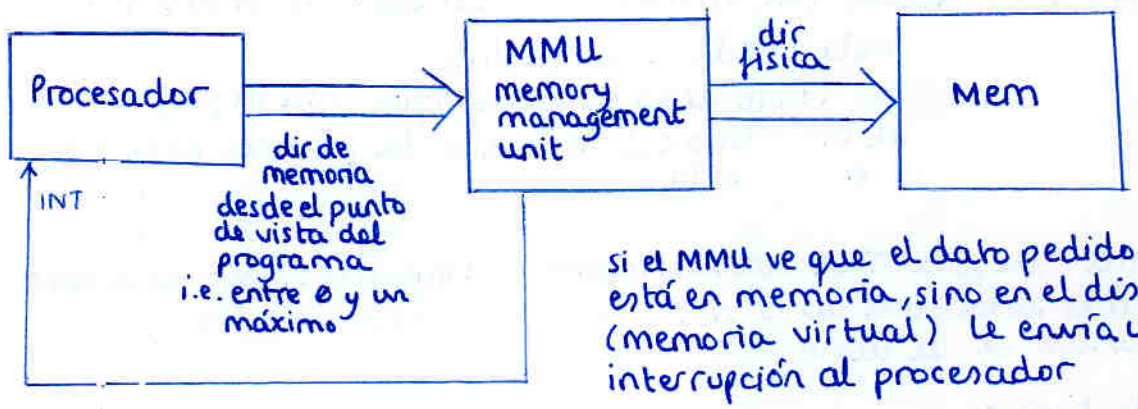


Figure 4-42. A process' stack (above) and its stackframe in the process table (below) corresponding to phases in handling a signal. (a) State as process is taken out of execution. (b) State as handler begins execution. (c) State while SIGRETURN is executing. (d) State after SIGRETURN completes execution.

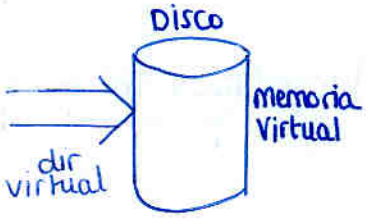
El stackframe se guarda en la tabla de proceso y normalmente se usa para volver a meter un proceso en CPU después de haber estado esperando o bloqueado; por eso ante una señal le cambio el PC al stackframe pero guardo en la pila el stackframe original, y lo restauro luego en SIGRETURN

Tema 6. Memoria Virtual

1. Concepto de MV



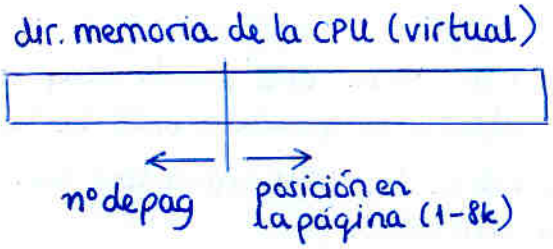
si el MMU ve que el dato pedido no está en memoria, sino en el disco (memoria virtual) le envía una interrupción al procesador



será entonces el SO, por SOFTWARE, quien se encarga de llevar el bloque a memoria;
Tiempo durante el cual el programa se bloquea esperando E/S
(idéntico a cuando quiere esperar un dato por teclado)

como vemos, el concepto es distinto al de cache, donde todo el proceso es transparente al procesador

2. Paginación



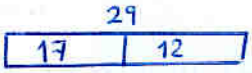
se logra dividir la memoria virtual en páginas

A su vez la memoria física se divide en marcos de páginas

ej 32 bits
páginas de 4k → 12 bits

2^{20} pág = 1M pag

si la memoria son 512 Mb → 29 bits



$\begin{array}{r} 29 \\ -\ 12 \\ \hline 17 \end{array}$

tengo 2^{17} marcos de página

Cualquier página puede ir a cualquier marco de página

Para ello requiere una tabla en memoria

en el ejemplo: tabla con un millón de entradas de 17 bits + 1

cada entrada es una página

los 17 bits dicen en qué marco está la página

el bit adicional indica si la página está o no en memoria

Ello implica que para TODO acceso a memoria hay que hacer un acceso previo a memoria para saber si la página está en memoria, consultando en la tabla.

ejemplo de tiempos

Tacc
Tdisco
ρ : prob de que dato esté en disco

$$\overline{T} = (1-\rho)[T_{acc} + T_{acc}] + \rho[T_{acc} + T_{disco} + T_{acc}]$$

↑ acceso a tabla en memoria ↑ acceso al dato en sí

Posible mejora : localidad

Es probable que un acceso a memoria tenga el dato en la misma página que algún acceso anterior

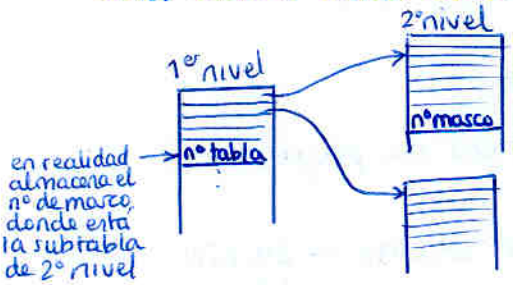
MMU puede tener una tabla (de 3 o 4 entradas) con el marco de página correspondiente a las últimas páginas usadas (TLB's)

Así sólo hago acceso a la tabla en memoria la primera vez que accedo a una página

TLB : tablas multinivel

La tabla de un millón de entradas de varios bits puede ocupar varios megas (en el ejemplo 3MB), (si el ordenador fuera 64 bits la tabla no cabe en memoria!)

¿Y si subdivido la tabla en una tabla de 1024 entradas con una tabla cada una de 1024 entradas?



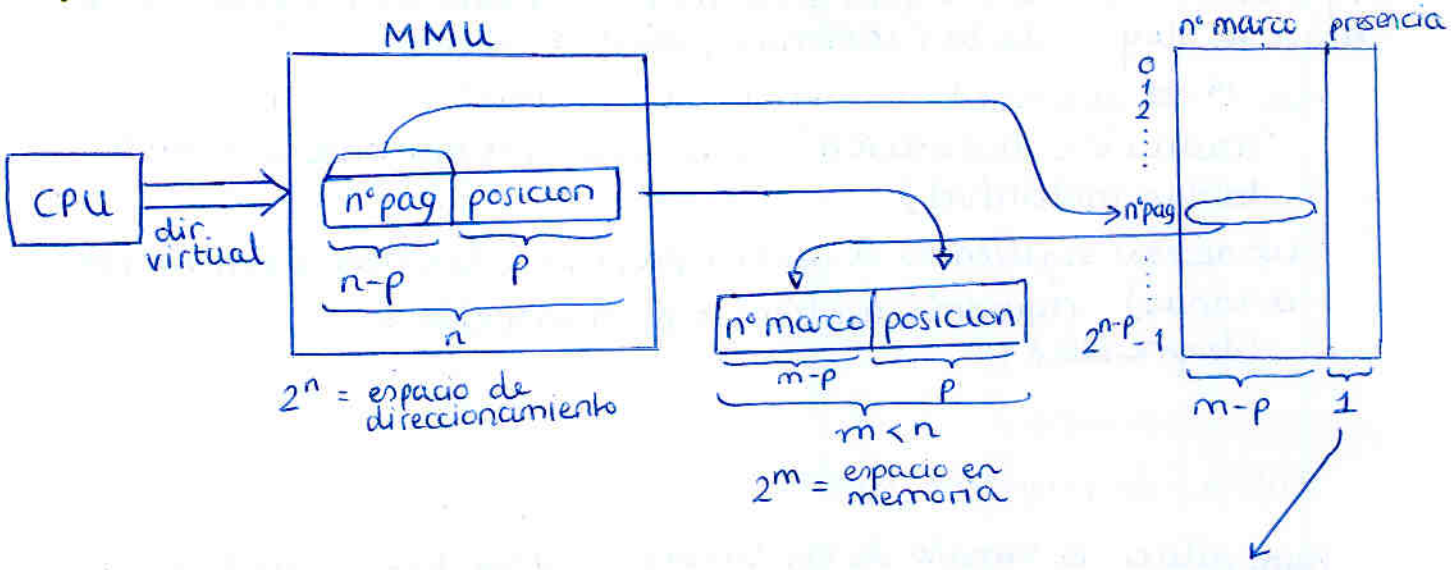
Ocupo más espacio aún

solución : almacenar sólo las tablas 2°nivel que esté utilizando

Tengo mucha menos memoria ocupada

Ahora cada acceso a memoria tiene un Tacc más (en total 3, uno por cada tabla y el propio dato) y además pueden darse fallos de tabla, que requieran crear tabla que no existía

Paginación (Resumen)

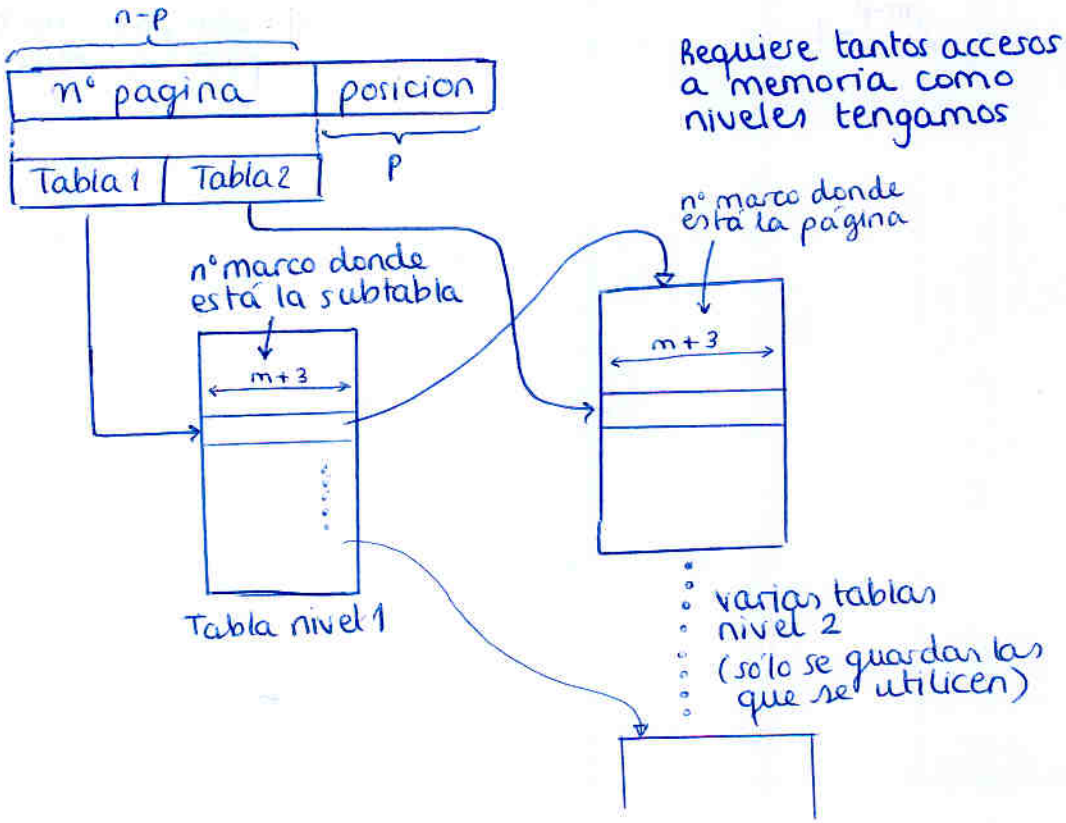


También puede tener la tabla ciertos atributos
- sólo lectura (proteger zonas de memoria, típicamente código)

Pueden haber otros bits de información/estadística
- bit R: se activa al usarse la página, se puede poner a 0 cuando quieras
- bit M: avisa cuando la página ha sido modificada (lo que hay en memoria es distinto al disco)

La tabla tiene tantas entradas como páginas $2^{n-p} - 1$
lo cual puede ser demasiada sobrecarga

solución: dividir la tabla en trozos (tablas multinivel)



Requiere tantos accesos a memoria como niveles tengamos

TLB: Translation Lookaside Buffers

Mejora: utilizar 2-8 registros de alta velocidad donde guardemos el nº de marco de las últimas páginas: (almacena nº pag y nº marco) ← con algún algoritmo de sustitución tipo LRU

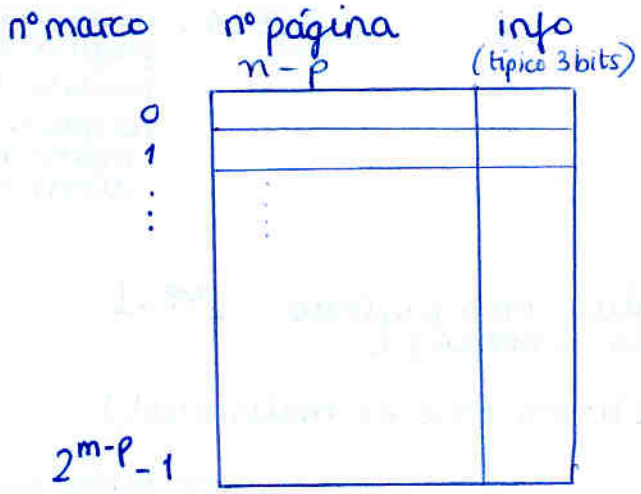
- La 1ª vez que accedo a una dirección, requiere gran tiempo de 'traducción de dirección' (varios accesos a memoria si usamos tablas multinivel)
- Los accesos siguientes a misma página (direcciones de memoria cercanas) requerirán un tiempo de traducción $\simeq 0$ (despreciable)

Tablas de páginas invertidas

Para reducir el tamaño de las tablas, usamos tablas invertidas

- una entrada por cada marco
- cada entrada contiene el nº de página

me obliga a leer todas las entradas hasta encontrar mi página



Ventaja:
ocupa muchísimo menos

Desventaja:
requiere recorrer la tabla

↓

se soluciona usando técnicas asociativas
ej: sólo poder meter páginas pares en marcos pares

2.3 Algoritmos de sustitución de páginas

Algoritmo que decide, de todos los marcos que tenemos, cual vamos a sobreescribir cuando, tras un fallo de página, haya que traerse una página a memoria

Por localidad, lo ideal es sustituir el LRU (least recenty used)
Pero algoritmos LRU son muy costosos de llevar a cabo

Tenemos la ventaja de que este algoritmo es software y ocurre cuando ha habido interrupción por fallo de página; tenemos tiempo para él. (ya que es despreciable frente al tiempo que se tarda en traernos la página del disco)

sin embargo el algoritmo debe ser muy rápido en los casos que NO HAYA fallo de página, ya que se ejecuta CADA vez que accedemos a memoria

- **FIFO**

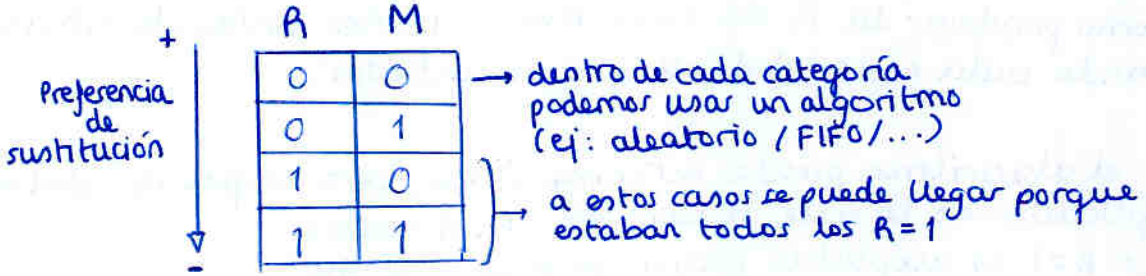
cola ordenada con los números de página
sustituir el primero de la cola

Inconveniente: Tiene en cuenta la vejez, pero no la utilidad
i.e. no se vuelve a meter en cola cada vez que se usa, ya que eso implicaría complejidad del algoritmo cuando NO hay fallo de página, lo cual NO interesa.

- **NRU (Not Recenty Used)**

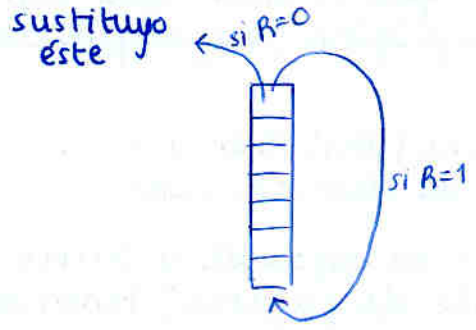
No tiene ningun tipo de ordenación; sólo divide en dos niveles (se ha usado hace poco o no)

- se puede conseguir poniendo periódicamente los bits R a cero y escoger para sustituir uno cualquiera de los que aún tienen R=0 (no usados) (o por ej usar algoritmo FIFO sólo con los R=0)
- También podemos decidir, de entre las R=0, las que tengan M=0 (no modificado → menos costosa de sustituir)



Segunda oportunidad o Reloj

FIFO con cola circular



aparte ponemos R=0 cada cierto tiempo (interrupción de reloj cada pocos ms)

Realmente no es mas que usar FIFO en la categoria R=0

LRU

Queremos de algún modo más categorías

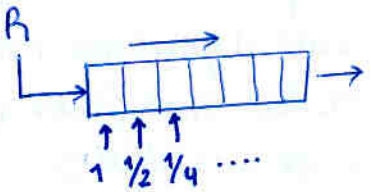
En la interrupción de reloj que ponemos todos los R=0, guardar un histórico de cómo lo tenía antes

ej: cada página tiene un contador, y antes de resetear todos los R a 0, sumar a cada contador el valor de R

Por tanto cuanto mayor sea el valor del contador, más ha sido utilizado

Problema: página se usa mucho pero de pronto 'pasa de moda' se quedará con un valor alto cuando no debería (ya no se usará mas)

Solución: En lugar de contador, usar registro de desplazamiento



Favorece que una página se haya usado hace poco para no cambiarla

Es como ponderar la R de hace más o menos ciclos de interrupción contando cada ciclo el doble que el anterior

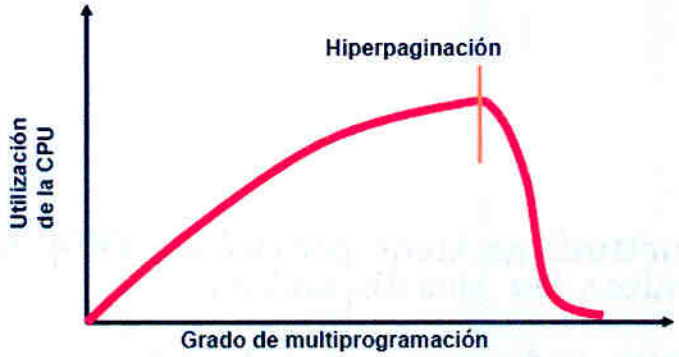
Nota: el algoritmo puede ser complicado, pero lo que no debe ser complicado es lo que hago en CADA acceso
(poner R=1 es aceptable porque se hace por HW)
(incrementar contadores cada interrupción de reloj es aceptable porque ocurre cada ciertos ms)

Hiperpaginación

Un suceso que ocurre por exceso de paginación

La paginación permite tener en ejecución más procesos de los que realmente caben en la memoria física del procesador. Los procesos utilizan la memoria física sólo para los datos que están utilizando, cuando acceden a un dato que no se encuentra en memoria se produce un fallo de página que pone al proceso en un estado de "NO PREPARADO".
Si las necesidades de memoria de los procesos aumentan o el número de procesos es demasiado grande, el número de fallos de página aumenta también.
Supongamos que el proceso A genera un fallo de página. Durante la gestión del fallo de página no se podrá ejecutar. La memoria está llena, y para traer la página del proceso A, se elimina una página del proceso B. El planificador escoge un proceso para que se ejecute durante el tiempo que el sistema de E/S tarda en traer la página de A. Es posible que el proceso seleccionado sea B. B empieza a ejecutar se y provoca un fallo de página, se escoge para el reemplazo una página de C. Durante el tiempo de intercambio, se escoge C como siguiente proceso…

Llevada al límite, la situación anterior haría que el uso de la CPU cayera debido al número de operaciones de E/S necesarias para la paginación. Si además se arrancan más procesos el proceso se agrava. Esta situación se la denomina **hiperpaginación** o **trashing**.
Para evitarla se utilizan técnicas basadas en limitar el número de procesos o asignar a cada proceso un número mínimo de marcos.



3. Segmentación

sabemos que la memoria de un proceso:

Pero no tienen porque estár asi juntitos

Se puede tener todo muy separado y tener numeración propia, etc...

Para ello se usan segmentos

La memoria se divide en segmentos (que pueden hasta solapase)
y entonces podemos tener
segmento de pila
segmento de datos
⋮

Para decir la dirección, dices el nº del segmento y la posición dentro del segmento

segmentación paginada

Dividir segmentos en páginas
Es como tablas multinivel donde el primer nivel son los segmentos

Se utiliza segmentación paginada desde el 386 hasta hoy dia

Nota final:
La memoria virtual no tiene porqué ser simplemente todos los gigas que permitan los bits disponibles

Podemos hacer que cada proceso del sistema crea tener todo el rango de memoria para él

UT 4. Gestión de Ficheros

Tema 7. SW de E/S

1. Generalidades

1.1 Concepto

Controlador HW : interfaz de cara al bus del procesador
proporciona registros donde el procesador puede leer y escribir

Para acceder a estos controladores; 2 alternativas: (el procesador requiere instrucciones HW específicas)
- Espacio E/S separado (bus especial para E/S) →
- Direcciones de memoria (en el mismo espacio de memoria mapeamos los controladores)

El diálogo con la E/S requiere avisos
- Interrupciones vectorizadas
- Enmascaramiento

DMA (direct memory access)
permite intercambiar datos entre la memoria y los controladores sin intervención del procesador: le liberamos

1.2 Tipos de dispositivo

Por bloques : transferencia en bloques de tamaño fijo
ej: típicamente discos

Por caracteres (bytes) : transferencia por bytes
ej : ratón, teclado, ...

2 Elementos SW

2.1 Objetivos

→ Independencia del dispositivo ← en alto nivel quiero leer y escribir lo que quiera

- Espacio de nombres uniforme : cada dispositivo se identifica con un nombre
- Comunicación SÍNCRONA : el acceso al disco en realidad es asíncrono (interrupción) queremos que el usuario lo vea síncrono

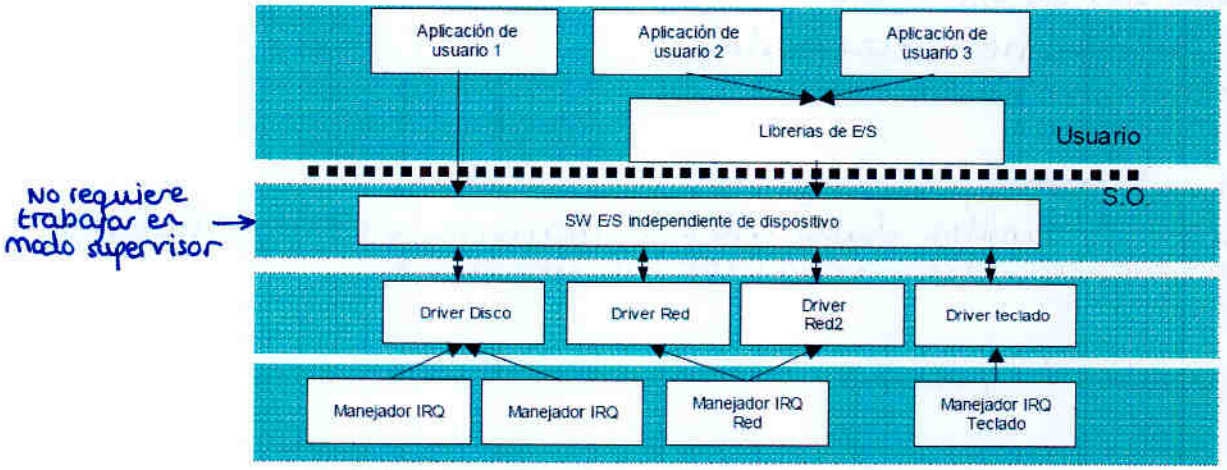
- Esquema de tratamiento de errores
- Mecanismos de uso exclusivo de recursos
  Si estoy utilizando un disposito (ej: grabando CD) que no pueda venir otro y usarlo
  quiero que el dispositivo sea exclusivo para mí

Clasificación

nivel bajo ↓ nivel alto

- manejador de interrupciones
- controlador de dispositivo (SW que sabe hablar con HW)
- SW del SO independiente del HW
- SW de usuario
  - aplicaciones
  - librerías de E/S

en MINIX:



2.1.1 SW independiente del HW

- Interfaz uniforme para los controladores de dispositivo (drivers)
- Nombres de dispositivos cara al usuario ej: PRN en MS-DOS /dev/printer en UNIX
- Protección: · importante en SO multiusuario
  · permisos de acceso para los distintos usuarios
- Definir unidad de transferencia con dispositivos (tamaño de bloque o caracter)
- Almacenamiento intermedio (buffers)
  ej: teclado: si escribimos en aplicación que no está leyendo del teclado, no queremos que se pierdan las teclas
- Gestión del espacio de almacenamiento en dispositivos
  - El usuario sólo tiene que decir lo que quiere escribir
  - Si el usuario va escribiendo caracteres en el disco, se va llenando un buffer hasta que se completa un bloque y entonces escribe
  - Tiene el peligro de que haya algún error antes de tener todo un bloque (de ahí el "quitar hardware con seguridad")
  - Un montón de cuestiones (localización, fragmentación,...)
- Asignar y liberar recursos dedicados
- Errores: tratamiento y, de no poder resolverlo, informar

2.2 Interbloqueos

Un conjunto de procesos está en bloqueo mutuo si cada proceso del conjunto está esperando un evento que sólo puede disparar un proceso del mismo conjunto (que a su vez está esperando)

Condiciones: - exclusión mutua - no expropiación (dispositivos no expropiables)
- retener y esperar - espera circular

Solución: No hacer nada → Algoritmo del avestruz

Se da tan pocas veces que no merece la pena el esfuerzo de solucionarlo; ya reiniciará el usuario

3. MINIX

- Kernel (Capa 0): Manejadores de IRQ
- Tareas (Capa 1): Drivers de dispositivo
- Servidores (Capa 2): SW independiente del dispositivo
- Aplicaciones (Capa 3) : SW de usuario y librerías E/S en estado de usuario.
  incluye 'Demonios': nombre **UNIX** para aplicaciones que acceden de forma especial a dispositivos para realizar tareas especiales: red, protocolos IP, imprimir... (ej: servidor web, lo asociamos al SO pero en realidad es programa de usuario aunque venga con el SO)

ejemplo: IRQ handler del disco (a lo que llamamos al haber interrupción)

```
/*===========================================================*/
* w_handler
* Recepción de IRQ desde disco duro.
*===========================================================*/
PRIVATE int w_handler(int irq)
{
        /* Disk interrupt, send message to winchester task and reenable
        interrupts. */
        w_status = in_byte(w_wn->base + REG_STATUS);/* acknowledge
                                                    interrupt */
        interrupt(WINCHESTER); /* ENVIO DE MENSAJE A TAREA DE DISCO */
        return 1;
}
```

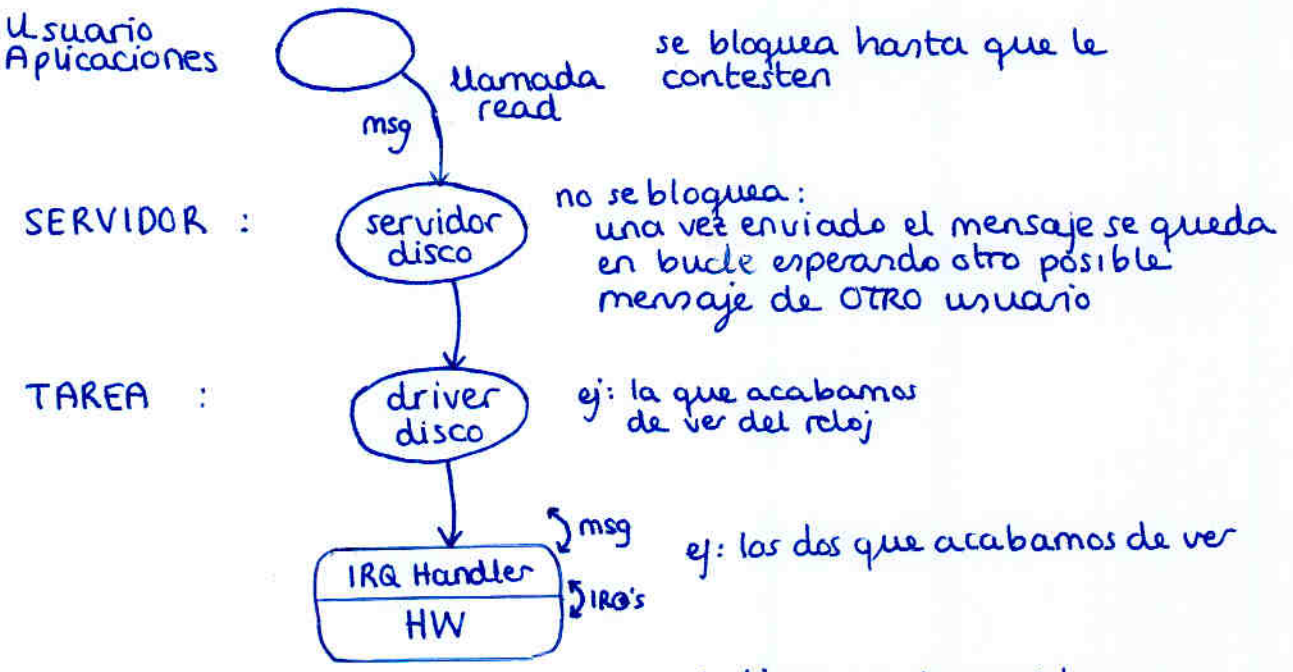
ejemplo: IRQ handler del teclado (con buffer)

```
PRIVATE int kbd_hw_int(int irq){
        // A keyboard interrupt has occurred. Process it.

        int code; unsigned km; register struct kb_s *kb;
        code = scan_keyboard();

        /* El teclado de los PC compatibles realizan dos irq: cuando se presiona
        y cuando se suelta una tecla. Se filtra lo segundo, ignorando
        todas menos las teclas especiales 29, 42, 54, 56, 58 (shift, ctrl, …)

        // máscara 0200
        if (code & 0200) { /* A key has been released (high bit is set). */
                km = map_key0(code & 0177);
                if (km != CTRL && km != SHIFT && km != ALT && km != CALOCK
                        && km != NLOCK && km != SLOCK && km != EXTKEY)
                        return 1; //no hacemos caso porq la hemos cogido al pulsar
        }

        kb = kb_addr(); /* Store the character in memory */
        if (kb->icount < KB_IN_BYTES) {
                *kb->ihead++ = code;
                if (kb->ihead == kb->ibuf + KB_IN_BYTES) kb->ihead = kb->ibuf;
                kb->icount++;
                tty_table[current].tty_events = 1;
                force_timeout();
        }

        /* Else it doesn't fit - discard it. */
        return 1; /* Reenable keyboard interrupt */
}
```

ejemplo de tarea: tarea a la que se llama cuando hay interrupción del reloj

```
PUBLIC void clock_task()
{
        int opcode;
        init_clock(); /* initialize clock task */
        // Main loop of the clock task. Get work, process it, sometimes reply.
        while (TRUE) {
                receive(ANY, &mc); /* go get a message */
                opcode = mc.m_type; /* extract the function code */
                lock(); /* Disable interrupts */
                realtime += pending_ticks; // transfer ticks from low level
                                                                handler
                pending_ticks = 0; /* so we don't have to worry about them */
                unlock(); /* Enable interrupts */
                witch (opcode) {
                        case HARD_INT: do_clocktick(); break; /* HW interrupt */
                        case GET_UPTIME: do_getuptime(); break;
                        case GET_TIME: do_get_time(); break;
                        case SET_TIME: do_set_time(&mc); break;
                        case SET_ALARM: do_setalarm(&mc); break;
                        case SET_SYNC_AL:do_setsyn_alrm(&mc); break;
                        default: panic("clock task got bad message", mc.m_type);
                }
        }
}
```

Recuerda:



Cuando el disco ya ha leido
- Interrupción
  - mensaje al driver (tarea)
    - mensaje al servidor
      - desbloquea a la aplicación de usuario habiendo efectuado read

4. E/S y Tiempo Real

Para que la aplicación funcione necesita dispositivos que tengan el tiempo de respuesta acotado

ej: disco duro en lectura
ej: transmisión en red ethernet no sirve ya que no tiene tiempo de respuesta acotado

Acceso a SW independiente de dispositivo

- Los tiempos deben ser conocidos y acotados para todas las operaciones
- Las operaciones en los dispositivos deben ser fiables

Tema 8. Sistema de archivos

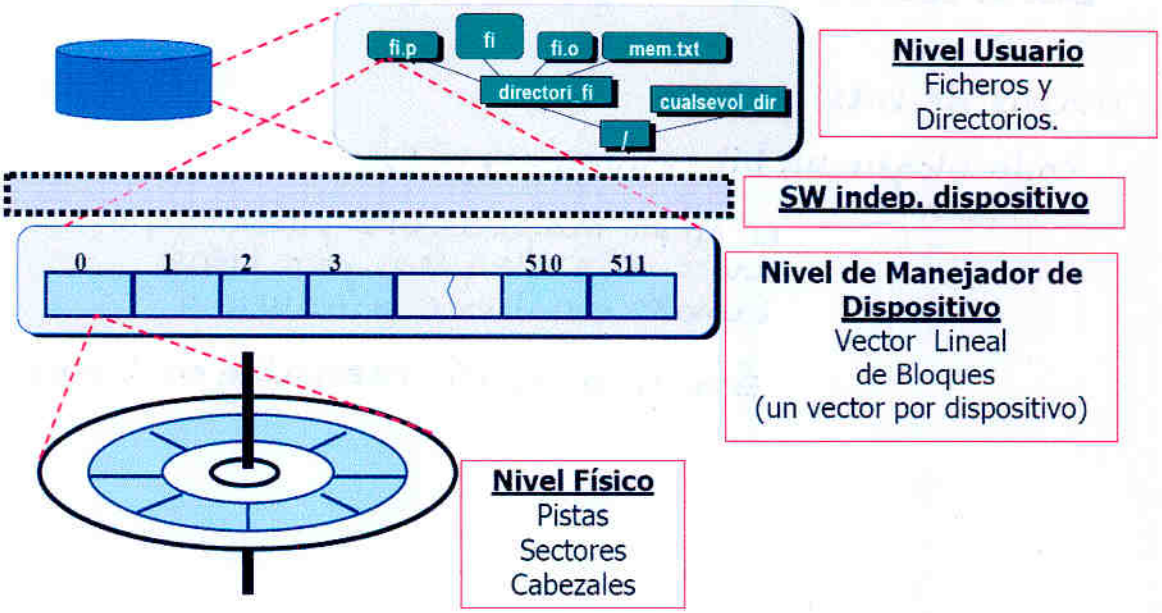
1. Sistemas de archivos
   - Gestión de espacio en disco
   - Localización de archivos y de datos
     - enlaces
     - localización de datos
   - Ejemplos: FAT32 y Nodos-i
   - consistencia
   - Rendimiento
   - seguridad
2. MINIX

1. Sistema de ficheros

Fichero: - abstracción de almacenamiento
- representa una colección de datos relacionados
- la relación de los datos es conocida por: el SO, una aplicación, o el usuario que lo creó.

Disco: - dispositivo típico de almacenamiento
- es un vector de sectores (numerados de 0 a N-1)

¿cómo asigno espacio a los ficheros?
¿cómo encuentro un fichero en un disco?
¿cómo utilizo varios dispositivos?



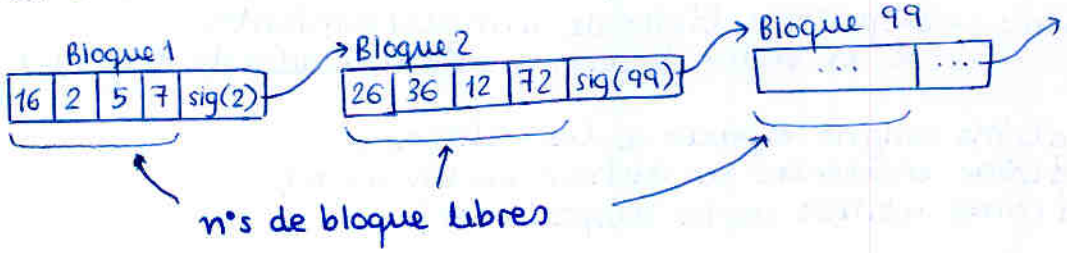
Manejo del disco (espacio + localización ficheros y datos)

- ¿Cómo localizo los archivos en un dispositivo? (1.2)
  - Organización jerárquica con directorios
    directorio = archivo especial que tiene una lista de archivos
  - Para discos muy grandes: particiones
- ¿Cómo localizo los datos de un archivo? (1.3)
  - FAT vs NODOS-i
- ¿Cómo gestiono el espacio libre del dispositivo? (1.1)
  - Lista enlazada vs. mapa de bits

1.1 Gestión del espacio libre

- Problema muy parecido al que teníamos en memoria
- Hay que indicar qué bloques están libres y cuáles no
  ¿Cómo?
  ↓
  si hay muchos podemos agrupar en clusters (FAT) o zonas (MINIX)

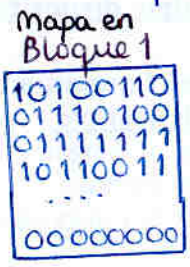
- Lista enlazada con todos los bloques libres



ocupan un tamaño variable
Ésta es la solución que elegíamos en memoria

- Mapas de bits

  cada bloque un bit (libre/ocupado)



El propio mapa de bits puede ocupar uno o más bloques, pero tiene tamaño constante y reducido

Ésta es la opción razonable en discos

1.2 Localización archivos

Los directorios contienen el nombre de fichero y un 'puntero'

ejemplos de directorios :

- Directorio en MSDOS (mucha información para cada fichero)

Bytes

8 Nombre	3 Extens	1 Atrib.	10 Reservado	2 Tiempo	2 Fecha	2 1er cluster	4 Tamaño
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

nota: se vió que era demasiado pequeño. Por compatibilidad se usa un byte reservado para indicar que el fichero tiene una versión larga del nombre guardada en algún otro lugar

nº del cluster donde EMPIEZA el fichero; además como veremos este es el nº de índice de la tabla FAT donde se inicia la 'lista enlazada' para localizar los datos del fichero

- Directorio en UNIX

nombre	puntero a nodo i
⋮	

y es el nodo-i quien tiene todos los datos (fecha, atributos, ...)

Enlaces :

hard links :
- entradas de directorio con la misma información de localización de archivos
- no hay forma de diferenciar la entrada creada normalmente y la creada como enlace
- en UNIX tan sencillo como

fich1	119
fich2	119

- en MSDOS no se puede ya que habría que mantener actualizados todos los campos (fecha, tamaño, ...) de todos los enlaces cada vez que se modificara un fichero.

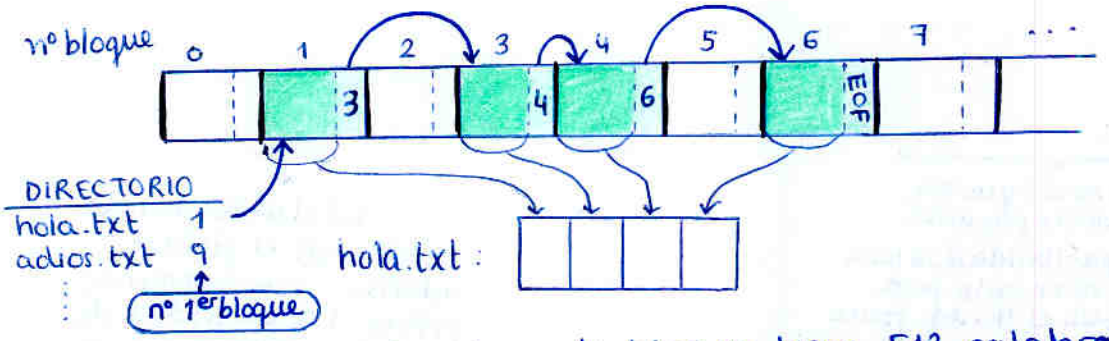
soft links (enlace simbólico) :
- archivo creado por el sistema operativo para hacer referencia a otro archivo (dato almacenado : nombre del archivo al que hace referencia)
  ejemplo : acceso directo de Windows

1.3. Localización de datos

Se trata de mantener una lista de los bloques utilizados por un fichero (no tienen porque ser consecutivos).

- De forma indexada: se guarda la lista en una estructura (ejemplo: nodos-i)
- De forma enlazada: se guarda en cada bloque información sobre el siguiente

ejemplo enlazado: los ultimos n bytes de cada bloque son un puntero al siguiente bloque o a EOF

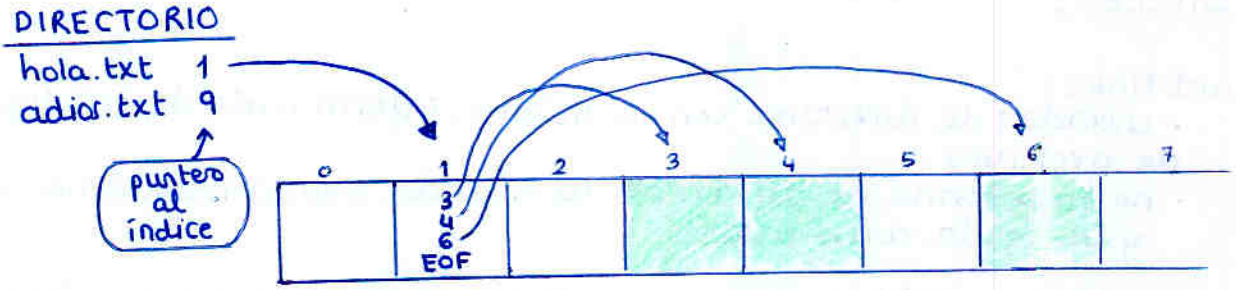


Nota: Si en este ejemplo cada bloque tiene 512 palabras, y las últimas 2 son el puntero al siguiente bloque, entonces el usuario ve bloques de 510 palabras.

Puede ser molesto ya que elimina la simplicidad de dividir el nº de palabra por 2^n para hallar el nº de bloque

ejemplo indexado:

. cada fichero tiene un bloque de índice donde hay ubicado un vector con todos los punteros a los bloques del fichero



ejemplo: FAT

- se usa la lista enlazada, pero NO en los propios clusters (evitando así la molestia de no poder dividir por 2^n para hallar el nº bloque)
- La lista enlazada se encuentra en una tabla en un bloque al inicio del disco
- FATxx : File Allocation Table
  xx bits por cada entrada
  tantas entradas como clusters tenga el disco

  ↓

  Se aprovecha para indicar los clusters que están libres (requiriendo xx bits para ello en lugar de uno, pero ahorrándonos por completo el mapa de bits)

Se mantienen dos copias de la tabla FAT

DIRECTORIO

Nombre fichero	...	1er cluster
f1	...	6
f2	...	5
f3	...	10

0	reservado
1	reservado
2	EOF
3	13
4	2
5	9
6	8
7	libre
8	4
9	12
10	3
11	error
12	EOF
13	EOF
14	libre

Tamaño del disco

f1: 6 → 8 → 4 → 2

f2: 5 → 9 → 12

f3: 10 → 3 → 13

cada entrada tiene xx bits — ej: 32 bits en FAT32

↓

Éste impuca un máximo de 2^{xx} posibles 'punteros', esto es una limitación muy importante al tamaño máximo de bloques en el disco, por eso lo que se hace es trabajar con punteros a 'clusters'

cluster = 2^n bloques

↓ unidad mínima de asignación de espacio en disco (i.e. mínimo tamaño de un fichero)

↓ unidad de intercambio con el disco

cluster grande: { desventaja: fragmentación enorme
ventaja: los bloques de un mismo cluster son consecutivos para el mismo fichero

Atributos	FAT12	FAT16	FAT32
Se utiliza en	Disquetes y particiones (pequeñas)	Particiones (pequeñas y medianas)	Particiones (medianas y grandes)
Tamaño de entrada	12 bits	16 bits	32 bits
Máximo número de Clusters	4,078	65,518	~268,435,456
Tamaño de Cluster	0.5 KB a 4 KB	2 KB a 32 KB	4 KB a 32 KB
Máximo tamaño de la partición	16,736,256	2,147,123,200	Aprox. 2^41

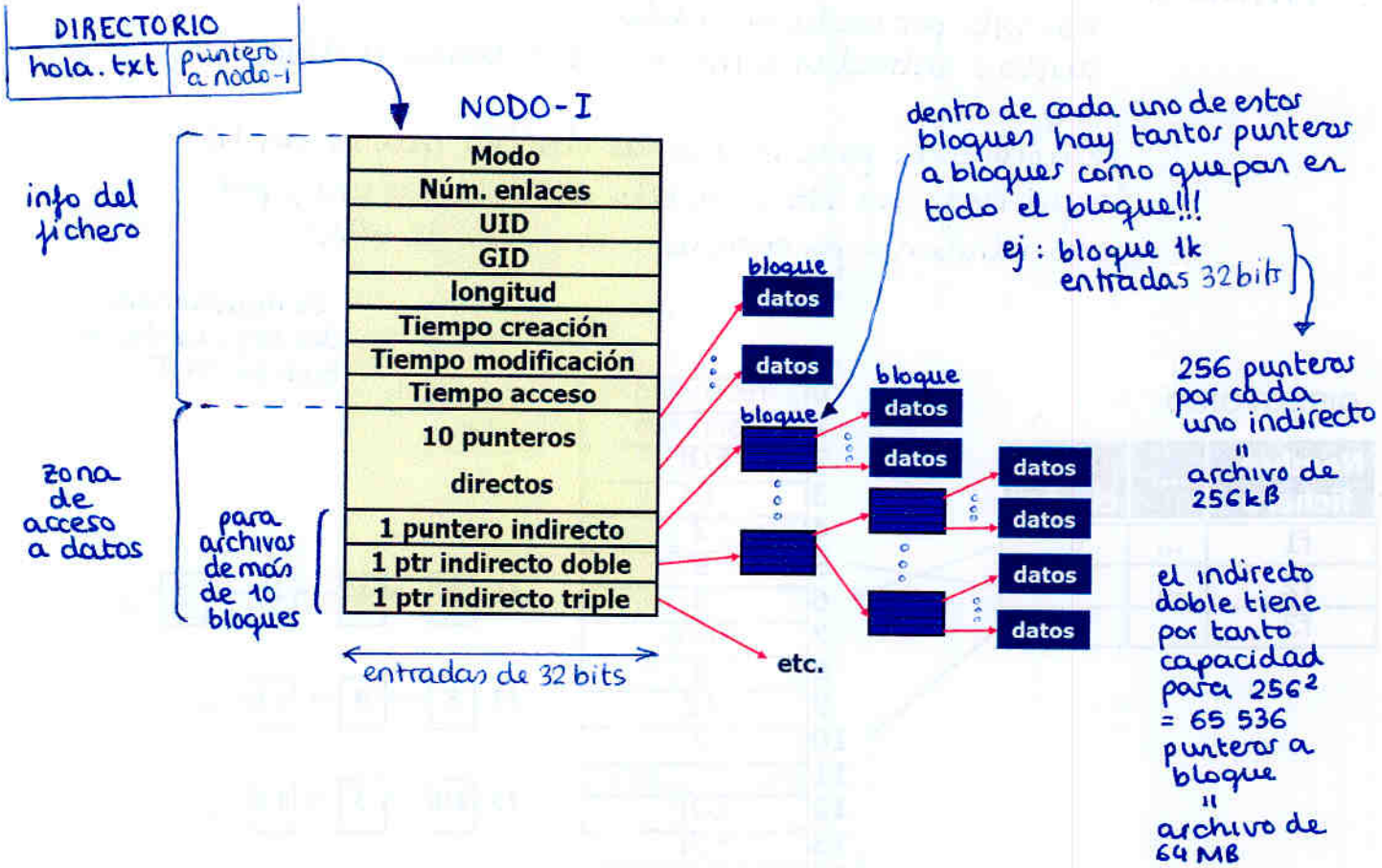
Tamaños de FAT con FAT32 (lo que ocupa la tabla)

Clusters / Tamaño Partición			16 KB	32 KB
8 GB	8 MB	4 MB	2 MB	1 MB
16 GB	16 MB	8 MB	4 MB	2 MB
32 GB	32 MB	16 MB	8 MB	4 MB
64 GB	64 MB	32 MB	16 MB	8 MB
2 TB (2,048 GB)	--	*1,024 MB*	*512 MB*	*256 MB*

ejemplo: Nodos-i

Asignación indexada: los nodos-i están todos en uno o más bloques al inicio del disco duro. Ya que son un recurso finito hay que gestionarlos y tener una estructura mapa de bits para anotar si cada uno está ocupado o libre:

Cada fichero tiene uno de éstos nodos-i



1.4 Consistencia

Duplicar información para lograr redundancia:

- Dos tablas FAT siempre sincronizadas
- Además de tener bitmap con nodos-i ocupados/libres, que cada nodo-i diga si está ocupado o libre

  (en UNIX, si se encuentra nodo-i que según el bitmap está libre pero el propio nodo-i dice que no, se añade como fichero al directorio lost+found)

Comprobaciones típicas:

- un bloque ocupado debe pertenecer a un único fichero
- un bloque libre no debe pertenecer a ningún fichero
- la longitud del fichero debe ser acorde con su lista de bloques
- un fichero que aparece en un directorio debe existir
- si un fichero existe debe aparecer en algún directorio

se pueden construir programas que comprueben la consistencia de un sistema de ficheros

1.5 Rendimiento

Caching: · Mantener en memoria copias de las estruct. de datos del disco
· Pretende minimizar el número de accesos al disco

La cache de bloques de disco es un elemento fundamental sin el cual los computadores no irían cara al aire.

· Cuidado en la escritura.
Ahorramos movimiento del cabezal escribiendo sólo al final

Clustering: · Se inventó con FAT para poder tener más memoria con un nº limitado de elementos direccionables.

· Direccionamos clusters formados por varios bloques
desventaja: fragmentación
ventaja: los bloques están seguiditos
↳ minimiza tiempo de acceso en lecturas/escrituras secuenciales

Algoritmo del ascensor:
Reordenar las peticiones de datos para minimizar los movimientos del cabezal

(igual que un ascensor con memoria reorganiza el orden de las peticiones para minimizar su movimiento)

1.6. Seguridad

• Motivos de ataques: fisgoneo casual, reto personal, fines destructivos, fines lucrativos.

• Fallos famosos: virus, problemas en comprobación de contraseñas (comprobarlas siempre toda a la vez, no carácter a carácter), DoS (Denial of Service = hacer inaccesible un recurso a los usuarios, por ej, consumiéndolo el atacante)

• Principios de diseño de mecanismos de seguridad:

-Diseño público.
(Si un sistema de seguridad debe ser secreto para funcionar, entonces no es un buen sistema de seguridad)

-Verificar vigencia de la autorización
(ej: que caduque la sesión)

-Denegar el acceso por defecto y minimizar el nivel concedido

• Ataques genéricos:puertos abiertos, realizar acciones prohibidas (hacer justamente lo que el manual diga que no se debe hacer), búsqueda en la basura...

2. MINIX

Recuerda: MINIX cabe en un disquette de 5 1/4 y funciona en PC con 640kB de RAM (sin usar memoria virtual)

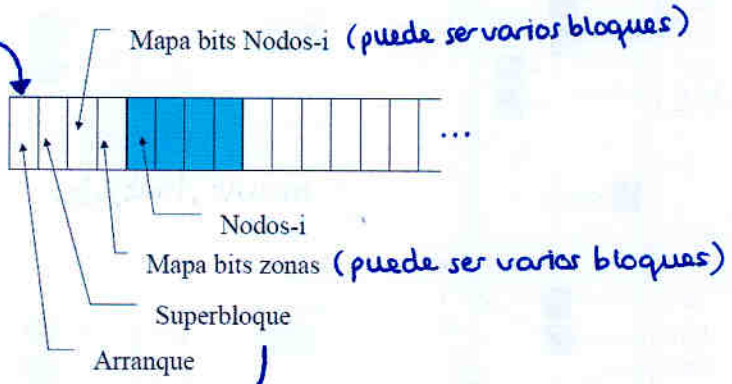
Sistema de ficheros basado en UNIX.

- jerarquía de directorios
- único espacio de nombres (todo fichero tiene ruta desde el directorio raíz /)
- gestión del espacio libre con mapa de bits
- utilización de nodos-i
  - Punteros de 32 bits a zonas
    - zona = 2^n bloques (equivalente a clusters en FAT)
      - se almacena n
      - típicamente $n=0$
  - Posibilidad de doble indirección

2.1. Organización y superbloque

Propio de la arquitectura del PC. Lo lee la BIOS

A partir de ese bloque 0 el resto lo decide el SO



SUPERBLOQUE

Presentes en disco y almacenados en memoria al montar el dispositivo.	Número de nodos
	Número de Zonas (V1)
	Núm. bloques del mapa de nodos-i
	Núm. bloques del mapa de zonas
	Primera zona de datos
	log2 (bloque/zona) = n
	Máximo tamaño de archivo
	Número mágico
	Relleno
	Número de zonas
Presentes sólo en la memoria temporalmente mientras el disco esté montado	Puntero al nodo-i del directorio raiz del sistema de ficheros
	Puntero al nodo-i del directorio sobre el que se monto el dispositivo
	Nodos-i/bloque
	Número de dispositivo (menor)
	Sólo lectura
	FS es Big-Endian
	Versión de FS
	Zonas directas/nodo-i
	Zonas indirectas/bloque indirecto
	Primer bit libre en mapa de nodos-i
	Primer bit libre en mapa de zonas

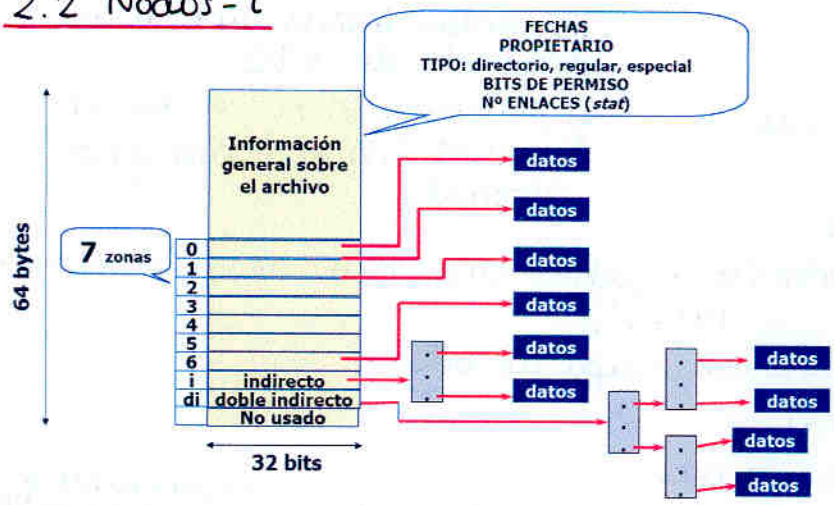
pactado, es lo que busca el SO para saber si es un superbloque

al montar se hace getblock de los bloques que contengan estos nodos-i para poder tener punteros a ellos en caché

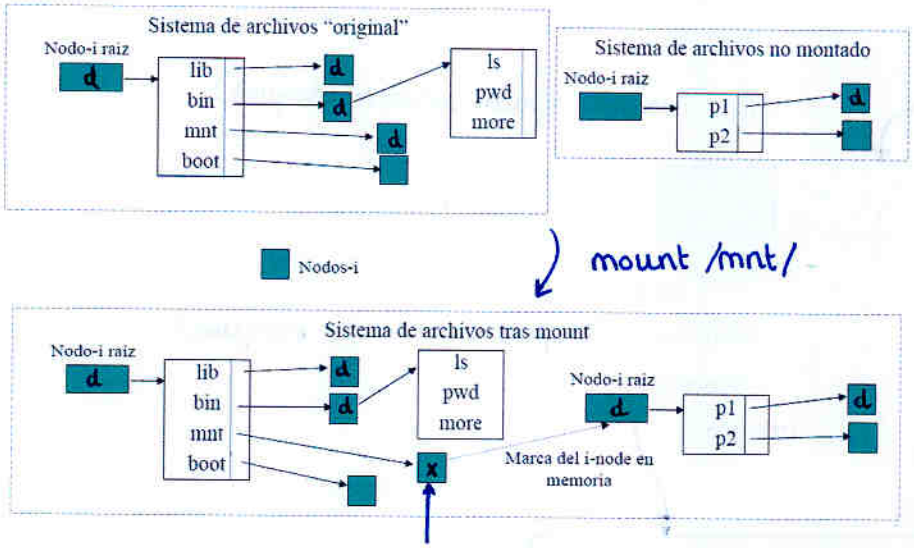
Cada dispositivo en MINIX tiene:
- nº principal (mayor)
  - tipo de dispositivo
  - indica que controlador usar
- nº secundario (menor)
  - identifica al dispositivo

atajo para buscar bits libres sin tener que recorrer el bitmap (hay que ir actualizándolo)

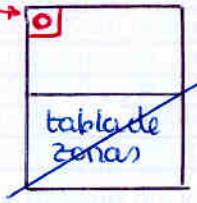
2.2 Nodos-i



2.3 Organización de directorios



nodo-i con una marca especial que indica que no es un directorio normal sino un dispositivo de bloques



nota: el bloque q contiene el nodo-i de un directorio sobre el cual se monta un dispositivo siempre está en cache de bloques (el superbloque tiene puntero a el, por tanto ha hecho GETBLOCK sin PUTBLOCK)

↓

Implica que para hallar las zonas de este directorio (recuerda que un directorio TAMBIEN es un archivo con zonas) no hay que buscar en la tabla de zonas del nodo-i, sino que:

Habrá que <u>buscar</u> en cada superbloque de <u>cada dispositivo</u> hasta hallar aquel cuyo campo "puntero al nodo-i del directorio sobre el que se montó el dispositivo" coincida con el nodo-i;

entonces para éste directorio habrá que utilizar el nodo-i apuntado por "puntero al nodo-i del directorio raiz del sistema de ficheros" del dispositivo. (que será un nodo-i que ya estará en un bloque de la cache de bloques al cual se hizo GETBLOCK cuando se montó el dispositivo; por tanto no necesitaremos ningún GETBLOCK adicional cuando utilicemos un directorio sobre el cual hay un dispositivo montado.)

2.4 Apertura de ficheros

¿Qué pasa en la memoria cuando un proceso abre un fichero?

Solución que se nos ocurre:
Tener en la tabla de descriptores de fichero el puntero al nodo-i del fichero y el puntero lseek (posición dentro del fichero)

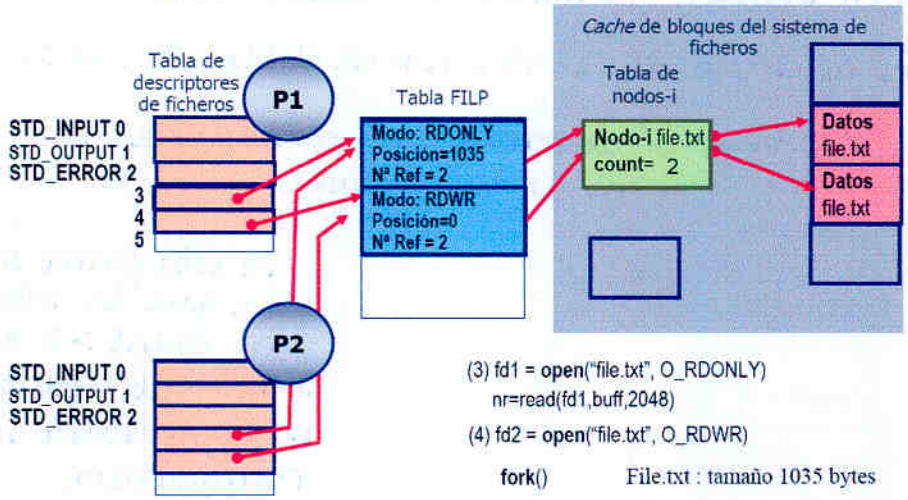
Problema:
Aunque puede interesar que dos procesos distintos usen distinto lseek, también podría interesar que dos procesos compartan lseek (ej: colaborando para escribir un fichero ej: open+fork)
¿Cómo podemos conseguir que puedan darse ambas situaciones?

Solución: Tabla intermedia, tabla FILP.

→ La orden open crea una nueva entrada en la tabla FILP (tipo de acceso, puntero a nodo-i del fichero abierto y puntero lseek) y añade a la tabla de descriptores de fichero un puntero a esta entrada. (y se incrementa el campo "Nº ref" en la entrada)

→ Ahora un fork duplicará la tabla de descriptores de fichero, por lo tanto, el padre y el hijo tendrán ambos un puntero a la misma entrada en la tabla FILP, y por tanto comparten lseek
(y se actualiza el campo "Nº ref" en la entrada FILP)

- Cada orden OPEN crea una nueva entrada en FILP y permite la independencia
- Los hijos y duplicados de un proceso que ha hecho OPEN comparten la entrada en la tabla FILP, permitiendo la colaboración. (comparten posición en el fichero)



Cache de bloques

Forma parte del "SW independiente del dispositivo"

se basa en almacenar en memoria los bloques que nos solicita el SO. Cuando el "SW independiente del dispositivo" necesita un bloque, en lugar de pedirlo al driver, lo pide a la cache de bloques (la cual a su vez se lo pedirá al driver si no tiene el bloque)

Funciones GETBLOCK y PUTBLOCK

No son llamadas al sistema, sino funciones internas del SO, no destinadas al usuario.

función GETBLOCK (nº bloque)

↳ devuelve dirección de memoria de ese bloque en cache (si no está en cache se trae del disco)

Con la dirección de memoria ya podemos leer y escribir lo que queramos en el bloque

> Nota importante: Éste comportamiento es distinto al típico read, donde el usuario pasa un buffer al cual copiar los datos. En este caso se devuelve la dirección de la propia cache de bloques.
> ¡Hay que implementar algo que nos garantice que la cache de bloques no va a quitar o sustituir el bloque que estamos usando tras haber hecho GETBLOCK!

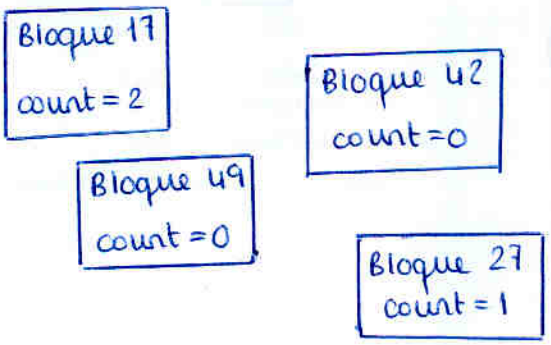
Para garantizar que el bloque seguirá en memoria mientras lo utilizamos; cada bloque de la cache tiene asociado un contador

- GETBLOCK (nº bloque): incrementa el contador en 1

una vez se ha acabado de leer/escribir en el bloque, el SO hace

- PUTBLOCK (nº de bloque) decrementa el contador en 1
  no devuelve nada

cache de bloques:



De ésta forma la cache sabe que los bloques con count > 0 están siendo utilizados y no los quitará ni reemplazará

Si count=0, el bloque no lo está usando nadie, y por tanto la cache puede quitarlo de memoria (copiándolo al disco si había sido modificado) cuando lo necesite (algoritmo LRU o similar)

Cuidado: hacer un PUTBLOCK que ponga count=0 no quiere decir que el bloque vaya a ser quitado de la cache y actualizado en el disco; tan sólo quiere decir que PUEDE serlo si la cache de bloques lo cree necesario.

Para forzar a un bloque a que se copie al disco se puede usar fflush.

ejemplo: open (/dir1/dir2/archivo)

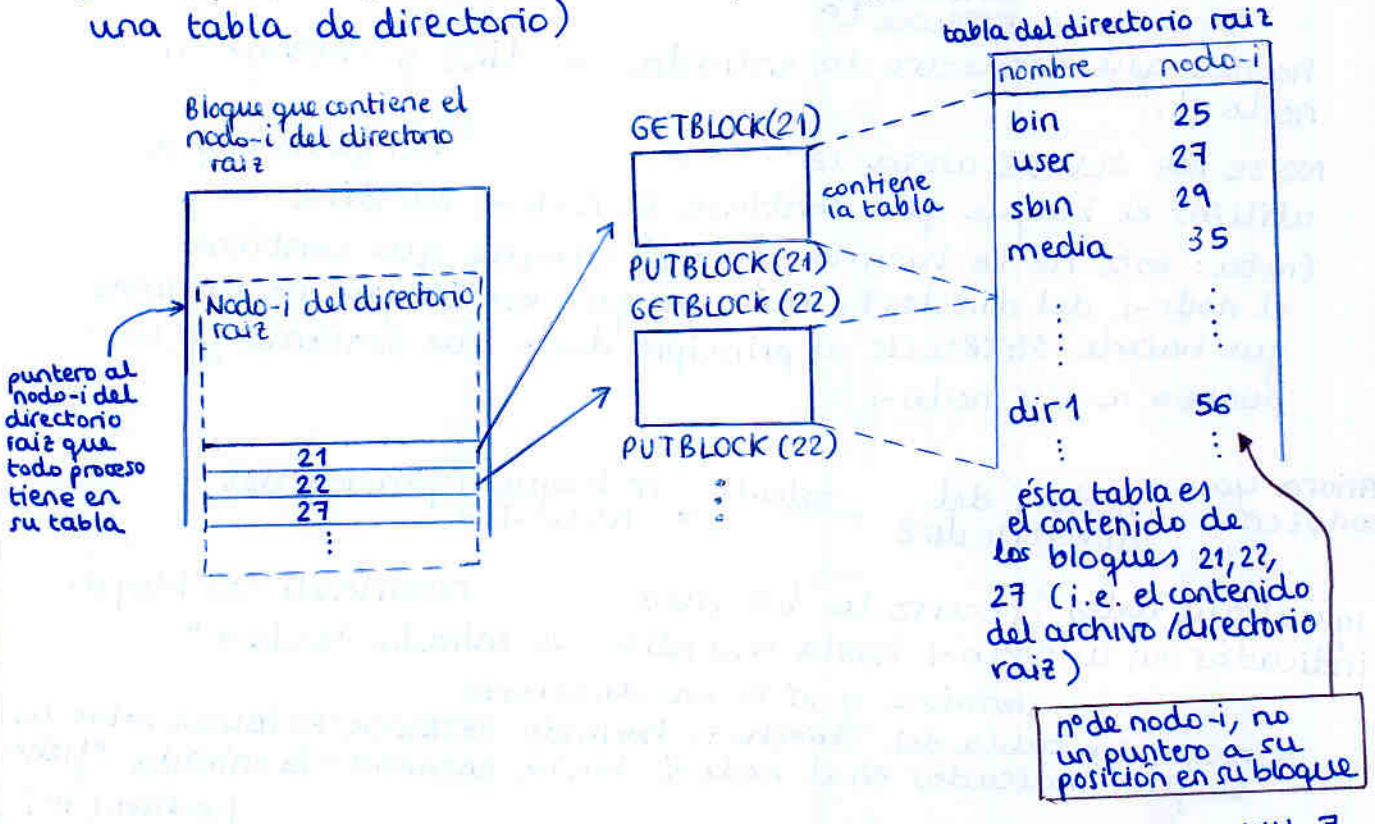
Nota inicial: en la tabla de un proceso se tienen los punteros a los nodos-i del directorio actual y del directorio raíz; ésto quiere decir que el(los) bloque(s) donde estén esos nodos-i estarán en la cache de bloques con un count > 0, ya que previamente se habrá hecho un GETBLOCK de ellos, para poder tener dichos punteros.

/ dir1 / dir2 / archivo

se obtiene el nodo-i del directorio raiz (el cual ya está en un bloque de la cache en una determinada dirección de la cual ya disponemos)

recuerda que los nodos-i estaban todos en algunos de los primeros bloques del disco

En el nodo-i del directorio raiz se indican los bloques en los cuales está el CONTENIDO de ese directorio raiz (el cual no es más que un fichero que contiene una tabla de directorio)



Por tanto conociendo la lista de bloques de la tabla del directorio raiz, puedo ir haciendo GETBLOCK's y PUTBLOCK's buscando la entrada dir1 a lo largo de la tabla.

GETBLOCK
buscar dir1 ¿lo encuentro?
- si: obtengo nº nodo-i
  PUTBLOCK
- No: PUTBLOCK
  GETBLOCK siguiente ← indicado en el nodo-i

Como ves, buscar un archivo que no existe en un directorio con muchos archivos, es costoso

Una vez encontrada la entrada | dir1 56 | ya sé el nº de nodo-i

nº nodo-i (numeración de 0 a MAX_NODOS) —calculo→ nº de bloque y posición dentro del bloque donde está el nodo-i

56 ⟶ 13

recuerda que los nodos-i están numerados y almacenados en los primeros bloques del disco, el cálculo es muy sencillo.

Hago GETBLOCK(13), y con la dirección de memoria del bloque que me devuelve le sumo las posiciones de memoria que hagan falta para localizar el nodo-i 56 dentro del bloque
Leo el nodo-i del directorio dir1 (que es el nodo-i nº 56)
- compruebo si el usuario tiene permisos
- compruebo si es un directorio

Recorro la tabla del directorio dir1 haciendo GETBLOCK y PUTBLOCK de los bloques indicados en el nodo-i de dir1

GETBLOCK
PUTBLOCK

hasta que encuentro la entrada de dir2 y obtengo su | dir2 59 | nodo-i.

NO SE ME OLVIDE ahora hacer PUTBLOCK(13) i.e. ya no voy a utilizar el bloque que contiene el nodo-i de dir1
(nota: esto no lo hicimos para el bloque que contiene el nodo-i del directorio raiz, ya que ese bloque no tuvimos que hacerle GETBLOCK al principio dado que teníamos ya un puntero a su nodo-i)

- Ahora ya conozco: nodo-i del directorio dir2 —calculo→ nº bloque y posición del nodo-i

  59 ⟶ 15

Hago GETBLOCK(15) y leo el nodo-i de "dir2"
- compruebo permisos y si es un directorio
- recorro la tabla del directorio haciendo GETBLOCK/PUTBLOCK sobre los bloques indicados en el nodo-i hasta encontrar la entrada "archivo"

| archivo 70 |

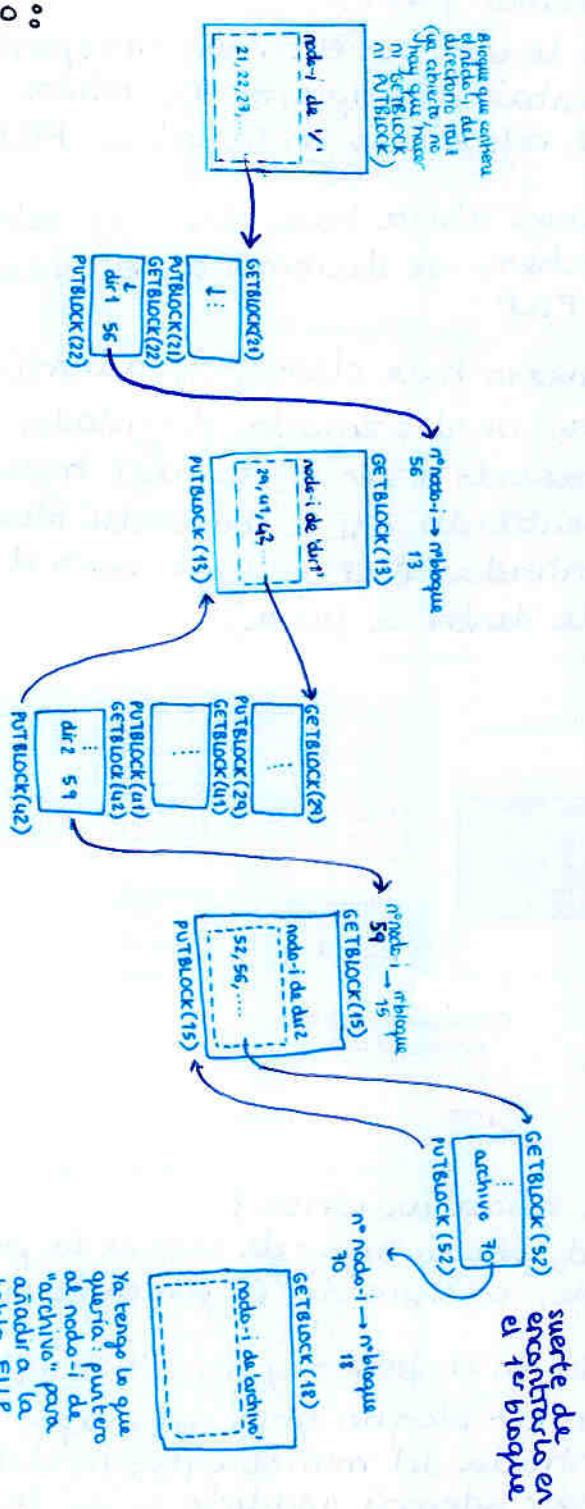
- me acuerdo de cerrar el bloque que contiene el nodo-i de "dir 2" PUTBLOCK(15)

Ya conozco el nº nodo-i de "archivo" 70 → calculo → nº bloque y posición donde está el nodo-i 19

Hago GETBLOCK(19) y genero con la dirección que me devuelve un puntero al nodo-i de "archivo"

- Creo una entrada en la tabla FILP que contenga ese puntero al nodo-i, y añado en la tabla de descriptores de fichero del proceso un puntero a dicha entrada FILP.

Resumen del proceso:



como vemos, la orden open es una de las pocas órdenes que no tienen el mismo número de GETBLOCKs que de PUTBLOCKs, sino uno más (lógico)

El objetivo es que repasemos los pasos que seguiría cualquier llamada al SO de las que hemos estudiado.

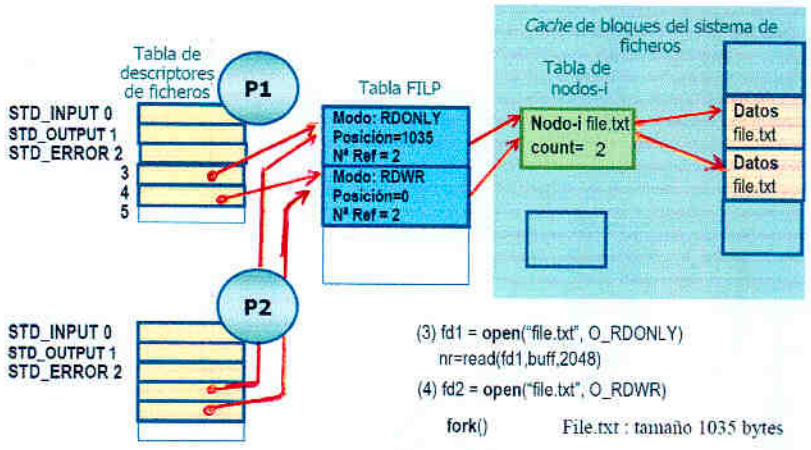
ejemplo: tras el f1 = open (/dir1/dir2/archivo)
hacemos nr = read (f1, buff, 2048)

- Descriptor de fichero → Entrada de la tabla FILP → dirección de memoria del nodo-i → listado de bloques del cual leer los datos (GETBLOCK + copiar a buffer + PUTBLOCK)

ejemplo: ahora hacemos fork()
Además de todo lo visto en el tema correspondiente, al duplicar la tabla de descriptores de fichero incrementamos un contador de referencias en la entrada FILP (pasa a 2)

Si uno de los 2 procesos ahora hace close(f1); además de borrar el descriptor de fichero, se decrementa el contador de referencias en la entrada FILP

Si ahora el OTRO proceso hace close(f1); vamos a la entrada FILP por el descriptor de fichero, se decrementa el contador de referencias en la entrada FILP pasando a ser 0, lo cual hace que se haga PUTBLOCK, decrementando así el count del bloque en 1.
Luego borro la entrada FILP y luego borro el descriptor f1
(siempre voy desde dentro a fuera)



ejemplo: ¿qué haría ahora un write?
casi idéntico a read, pero en lugar de leer de la porción de memoria del bloque en cache, escribe en él (y poniendo bit modificado = 1)

¿Y si el write no cabe en el fichero y necesito bloques nuevos?
Hay que buscar el primer bloque libre del mapa de bits de bloques (viene dado en una entrada del macrobloque, no olvidar actualizarla) y hacerle un GETBLOCK; además añadirlo a la lista de bloques del NODO-I

ejemplo: Problema de examen

5. Para cada una de las siguientes llamadas al sistema (que se ejecutan secuencialmente).
Indicar :
   - Número de getblock() y putblock() realizados.
   - Indicar todas las estructuras del SO y del sistema de ficheros implicadas en su ejecución y cómo se utilizan dichas estructuras.

Se supone que el sistema de ficheros de TODOS los dispositivos es UNIX, que en el directorio / se encuentra montado un disco con bloques de 2Kbytes y en /usr/floppy otro dispositivo con bloques de 1Kbyte. Todos los directorios tienen una longitud total de 2 Kbytes y todas las referencias utilizadas son al último fichero de cada directorio.
Suponer así mismo, que inicialmente los ficheros referenciados en las líneas 3, 4 y 5 existían y tenían un tamaño de 2Kbytes. El directorio de trabajo del proceso es /usr/floppy/d1.
NOTA:

O_TRUNC: elimina el contenido del fichero y se escribe desde la posición 0.
O_APPEND: las escrituras se realizan a partir del final del fichero.

(2 ptos)

[1] `close(0);`
Cierra entrada estándar -> Decrementa el número de referencias en la tabla FILP, si es cero liberará entrada y hará **un putblock** del bloque del nodo-i asociado. Libera entrada 0 de la tabla de descriptores.

[2] `close(1);`
Cierra salida estándar -> decrementa número de referencias en la tabla FILP, si es cero liberará entrada y hará **un putblock** del bloque del nodo-i asociado. Libera entrada 1 de la tabla de descriptores.

[3] `b=open("/tmp/mifich",O_RDONLY)`
Abre el fichero /tmp/fich para sólo lectura ->
Accede al nodo-i del directorio raíz, lee el bloque de datos (2K) (1 getblock y 1 putblock) para localizar la entrada tmp. Accede al bloque que contiene el nodo-i de tmp (1 getblock y 1 putblock). Lee el bloque de datos de tmp (2K) (1 getblock y 1 putblock) para localizar la entrada mifich. Accede al bloque que contiene el nodo-i de mifich (1 getblock). **Total 4 getblocks 3 putblocks.**
Reserva una entrada en la FILP que apunta al nodo-i de mifich, inicializa a 1 el contador de referencias y a cero el puntero de posición.
Anota la referencia a la entrada de la FILP en la entrada 0 de la tabla de descriptores de ficheros.
La llamada devuelve 0.

[4] `a2=open("m1",O_WRONLY | O_TRUNC);`
Abre el fichero m1 para sólo escritura truncando (borrando el contenido previo) ->
Accede al nodo-i del directorio de trabajo, lee los bloques de datos (2x1K) (2 getblock y 2 putblock) para localizar la entrada m1. Accede al bloque que contiene el nodo-i de m1 (1 getblock).
Libera los bloques de datos del fichero, para lo cual ha de ir al mapa de bits de zonas y poner a cero los bits correspondientes a los dos bloques, 1 o 2 getblocks y 1 o 2 putblocks dependiendo de que los bits que representan los bloques en el mapa de bits de zonas estén en el mismo bloque o no. Posteriormente habrá que acceder a la tabla de superbloques para verificar si es necesario modificar la entrada que dice cual es el primer bit libre en el mapa de bits de zonas. **Total 4 ó 5 getblocks y 3 ó 4 putblocks.**

Reserva una entrada en la FILP que apunta al nodo-i de m1, inicializa a 1 el contador de referencias y a cero el puntero de posición. Se actualiza en el nodo-i el tamaño del fichero a 0 y la fecha de la última escritura a la actual.
Anota la referencia a la entrada de la FILP en la entrada 1 de la tabla de descriptores de ficheros.
La llamada devuelve 1.

[5] a3=open("/usr/floppy/m2",O_WRONLY | O_APPEND);
Abre el fichero /usr/floppy/m2 para sólo escritora posicionándose al final del fichero ->
Accede al nodo-i del directorio raíz, lee el bloque de datos (2K) (1 getblock y 1 putblock) para localizar la entrada usr. Accede al bloque que contiene el nodo-i de usr (1 getblock y 1 putblock). Lee el bloque de datos de usr (2K) (1 getblock y 1 putblock) para localizar la entrada floppy. Accede al bloque que contiene el nodo-i de floppy (1 getblock y 1 putblock). Ve que hay una marca que indica que se trata de un dispositivo montado. Accede a la tabla de superbloques para localizar la entrada que contenga la dirección de memoria del nodo-i de floppy. Localiza en la entrada anterior la dirección del nodo-i raíz del dispositivo montado. Accede al nodo-i dicho directorio, lee los bloques de datos (2x1K) (2 getblock y 2 putblock) para localizar la entrada m2. Accede al bloque que contiene el nodo-i de m2 (1 getblock). **Total 7 getblocks 6 putblocks.**
Reserva una entrada en la FILP que apunta al nodo-i de m2, inicializa a 1 el contador de referencias y a 2048 el puntero de posición (obtiene este valor de la entrada del nodo-i que indica el tamaño del fichero).
Anota la referencia a la entrada de la FILP en la entrada 3 (asumimos que es la primera entrada libre) de la tabla de descriptores de ficheros.
La llamada devuelve 3.

[6] n=read(0,buffer,10240);
Lee hasta 10240 bytes del fichero 0 (mifich), como el fichero tiene un tamaño de 2048, sólo leerá 2048 bytes.
Accede a través de la tabla de descriptores a la tabla FILP y a través de esta localiza en nodo-i. Lee el bloque de datos de 2k (**1 getblock y 1 putblock**) y copia el contenido en buffer. Actualiza el contador de posición de la FILP a 2048.
La llamada devuelve 2048.

[7] write(a3,buffer, n);
Escribe 2048 bytes en el fichero a3 (m2).
Como este fichero esta en el disquete tiene que escribir dos bloques, para lo cual tiene que buscar dos boques libres en el mapa de bits de zonas de este dispositivo. A través del superbloque sabe cual es el primer bloque libre, que deberá marcar como ocupado en el mapa de bits de zonas (1 getblock y 1 putblock) posteriormente deberá buscar el siguiente bit libre que podría estar en el mismo bloque del mapa de bits de zonas o en otro (n getblocks y n putblocks) y lo marcará como ocupado. Luego volverá a buscar el siguiente bloque libre para actualizar la tabla de superbloques (m getblocks y m putblocks). Posteriormente tiene que escribir los datos en los dos bloques obtenidos (2 getblocks y 2 putblocks) **Total 3+n+m getblocks y 3+n+m putblocks**
El puntero de posición de la FILP se incrementa en 2048, por lo que valdrá 4096. Se actualiza en el nodo-i el tamaño del fichero a 4096 y la fecha de la última escritura a la actual. En la tabla

de punteros a bloques de datos del nodo-i se escriben los números de los dos bloques nuevos.
La llamada devuelve 2048.

[8] write(1,"123456789 123456789 123456789 ",15);
Escribe 15 bytes en el fichero 1 (m1)
Tiene que escribir un bloque, para lo cual tiene que buscar un boque libres en el mapa de bits de zonas de este dispositivo. A través del superbloque sabe cual es el primer bloque libre, que deberá marcar como ocupado en el mapa de bits de zonas (1 getblock y 1 putblock) posteriormente deberá buscar el siguiente bit libre que podría estar en el mismo bloque del mapa de bits de zonas o en otro (n getblocks y n putblocks) para actualizar la tabla de superbloques. Posteriormente tiene que escribir los datos en el bloque obtenido (1 getblock y 1 putblock) **Total 2+n getblocks y 2+n putblocks**
El puntero de posición de la FILP se incrementa en 15, por lo que valdrá 15. Se actualiza en el nodo-i el tamaño del fichero a 15 y la fecha de la última escritura a la actual. En la tabla de punteros a bloques de datos del nodo-i se escribe el número del bloque nuevo.
La llamada devuelve 15.

[9] exit(0)
La llamada exit debe cerrar todos los ficheros abiertos, por lo que realizara **3 putblocks** (1 por cada nodo-i) También realizaría 1 putblock del nodo-i del directorio de trabajo y 1 putblock del nodo-i del directorio raíz.

Problema

Un sistema UNIX posee un disco duro, sobre el que esta el directorio raiz, y un disquette montado en el directorio /floppy. Tamaño de zona: 1K, y tamaño de `/floppy/dir/fil` de 990 bytes.

Para las siguientes llamadas, indica las estructuras del sistema de ficheros utilizadas señalando la operación que se realiza en las mismas.

Las llamadas son:

```
A. fd=open("/floppy/dir/fil",O_RDONLY);
B. chdir("/floppy/dir/")
C. fd2=open("fil",O_RDWR);
D. read(fd,buffer,100);
E. dup(fd2);
F. close(fd2);
G. close(fd);

D.0 pos=lseek(fd2,0, SEEK_END);
D.1a res=write(fd2,buffer,2);
D.1b res=write(fd2,buffer,500);
```

requerirá buscar un bloque nuevo

	A	B	C	D'	E	F	G
Tabla Superbloques							
Mapa Bits Nodos-i							
Mapa Bits Zonas							
Nodo-i fi1							
Bloque datos (fi1)							
Entrada tabla desc							
Entrada en FILP							
Ptr.Nodo-i Dir. Raiz							
Ptr.Nodo-i Dir. Traba							
Tarea HD							
Tarea FD							
GetBlocks()							
PutBlocks()							

contiene el puntero del próximo bit libre en el bitmap de zonas, y habrá que actualizarlo

	A	B	C	D'	E (dup(fd2))	F (close(fd2))	G (close(fd))	D0	D1a	D1b
Tabla Superbloques	Leer	Leer								Modif
Mapa Bits Nodos-i	-	-	-	-	-	-	-	-	-	-
Mapa Bits Zonas	-	-	-	-	-	-	-	-	-	Modif.
Nodo-i fi1	Leer	-	Leer	Leer	-	-	-	Leer	Modif.	Modif.
Bloque datos (fi1)	-	-	-		-	-	-	-	Modif.	Modif.
Entrada tabla desc	Crea	-	Crea	lee	Lee / Crea	Elimina	Elimina	Lee	Lee	Lee
Entrada en FILP	Crea	-	Crea	Mod Pos	Mod Refs	Mod Refs	Elimina	Mod Pos	Mod Pos	Mod Pos
Ptr.Nodo-i Dir. Raiz	Lee	Lee	-	-	-	-	-	-	-	-
Ptr.Nodo-i Dir. Trabajo	-	Modif.	Lee	-	-	-	-	-	-	-
Tarea HD	Prob Si	Prob No	No	No	No	No	No	No	No	No
Tarea FD	Prob Si	Prob No	Prob No	Prob Si	No	No	No	No	Prob Si	SI
GetBlocks()	6	4	2	1	-	-	-	-	1	n+1
PutBlocks()	5	3+1	1	1	-	-	1	-	1	n+1

hay que hacerse el dibujito

PutBlock del dir. trabajo antiguo

el bloque de los datos de fil aun no estaba en caché de bloques (sólo el bloque que contiene el nodo-i)

no elimina la entrada de la tabla FILP porque Refs = 2, sólo lo decrementa

en cambio en este caso Refs = 1, por eso elimina la entrada FILP y además hace un PutBlock

accesos al mapa de bits

actualizar el tamaño

Examen Junio 2005

ARQUITECTURA DE COMPUTADORES Y SISTEMAS OPERATIVOS II

Junio 2005

1. El listado de un directorio, es el siguiente:

```
SUID SGID PERMISOS    UID    GID    Nombre
  -    -  drwxr-xr-x julio  profe  .
  -    -  drwxrwxr-x root   staff  ..
  -    -  -rw-r-----  julio  staff  Notas
  -    -  -rw-r-----  pepe   profe  Trabajo
  -    -  -rw-------  julio  staff  Examen
  -    -  ----------  pepe   profe  F1
  1    -  -r-xr-xr--  root   profe  rm1
  -    -  -r-xr-xr-x  julio  profe  rm2
  1    1  -r-xr-xr-x  julio  profe  rm3
  -    1  -r-xr-xr-x  julio  profe  cat1
  -    -  -r-xr-xr--  julio  profe  cat2
  1    -  -r-xr-xr-x  pepe   staff  cat3
  -    1  -r-xr-xr-x  julio  staff  mv1
  1    1  -r-xr-xr--  julio  alumn  mv2
  1    1  -r-xr-xr-x  julio  profe  chmod1
  -    1  -r-xr-xr-x  julio  profe  chmod2
  1    1  -r-xr-xr-x  julio  profe  cp1
  1    1  -r-xr-xr--  pepe   staff  cp2
  1    1  -r-xr-xr-x  julio  profe  cambia1
  1    1  -r-xr--r--  pepe   staff  cambia2
  -    -  -r-xr-xr-x  pepe   staff  jun2005
```

Suponiendo que los programas **rm*, cat*, mv*, chmod* y cp*** son copias de las ordenes estándar de UNIX, y **cambia*** es un programa que modifica el fichero que se le pasa como parámetro, suponer que el usuario **pepe** del grupo **alumn** accede a este directorio e intenta ejecutar las líneas de órdenes que se especifican (suponer que antes de ejecutar cada una el directorio siempre se encuentra en el estado inicial).
En el directorio /tmp todos los usuarios tienen permiso de escritura.
Marcar con un círculo **V** si la orden se lleva a cabo con éxito y **F** si no puede realizarse.

2 ptos:

acertada	**+0.1,**
fallada	**-0.1,**
no contestada	**0**

V	(F)	chmod1 444 F1
(V)	F	cambia2 Trabajo
(V)	F	cp1 Trabajo /tmp/Trab2
V	(F)	chmod2 444 Notas
V	(F)	cat1 Notas
V	(F)	cp2 Examen ../Exam2
V	(F)	mv1 Trabajo Trab2
V	(F)	cat3 Notas > ./Notas2
V	(F)	cambia1 Trabajo
(V)	F	mv2 Examen /tmp/Exam2

(V)	F	cambia1 Notas
(V)	F	cp2 Trabajo ../T2
(V)	F	chmod2 444 F1
V	(F)	rm1 Trabajo
(V)	F	cp1 Notas Notas2
(V)	F	rm3 Examen
V	(F)	cat2 Trabajo
V	(F)	mv1 Notas Notas2
(V)	F	chmod1 777 Notas
(V)	F	cp1 Examen /tmp/examen

2. El programa de la hoja adjunta se ha llamado desde el shell del ejercicio 1 con la siguiente línea de órdenes :

```
./jun2005 /etc/passwd F1 Notas Trabajo /tmp/results 2> /tmp/errores
```

Sabiendo que en el directorio /tmp todos los procesos pueden escribir, y que los ficheros que se han pasado en la línea de órdenes tienen los siguientes tamaños en bytes:

/etc/passwd 2035 Notas 1025 Trabajo 234521 F1 2

Se pide:

- Árbol de procesos generado
- Salida por pantalla y contenido de los ficheros generados por los procesos
- Modificar el código según las instrucciones indicadas en el mismo.

Suponed que el primer proceso tiene un PID=10 y que a los demás se les asignarán identificadores consecutivos.

(2,5 puntos)

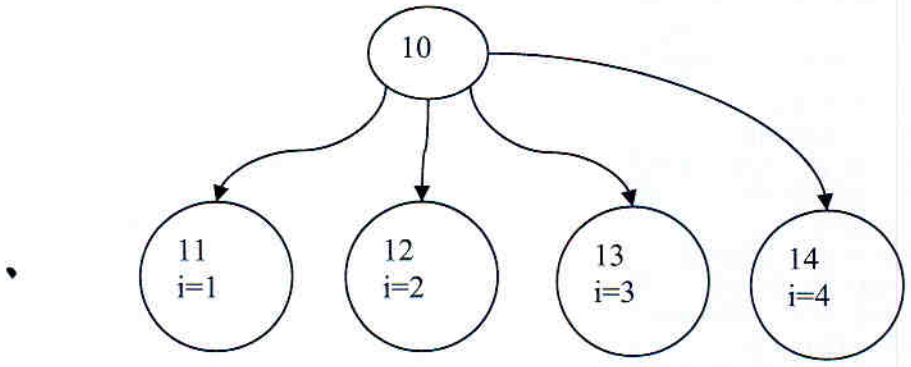
NOMBRE:

E1

Solución:

Lo primero es determinar que significa la línea de órdenes, en la que tenemos 5 parámetros para el programa y por lo tanto en argv recibiremos 6 cadenas (argc==6).
En la misma línea de ordenes hay una redirección de la salida de error, esta redirección la realizará el shell antes de hacer la llamada a execve() con la que empezará el programa. Como en /tmp se puede escribir, el shell creará el archivo /tmp/errores donde todos los procesos creados tendrán direccionada su salida de error.

Si seguimos la ejecución desde el principio observamos que el proceso 10, el inicial, lo primero que hace es comprobar que el número de parámetros sea correcto (entre 3 y 30) y después se protege de la señal 15, protección que heredarán sus hijos. Además crea una tubería de la que sólo 10 leerá.
En el bucle for() posterior creará (argc-2) hijos, cada uno de ellos tiene en la variable i el orden en que fue creado:



Después todos los hijos entran en el if(tipo==0) y el padre en el else.
Los hijos, realizan un printf y luego intentan abrir los archivos que se le habían pasado a 10 como parámetros. Para dos de ellos (F1 y Notas) no tenemos permiso de lectura, por lo que el open fallará y los procesos realizarán un exit(1) tras mandar al padre, a través de una tubería, su pid y un tamaño de fichero 0. Los procesos que hacen esto son 12 y 13.
11 y 14 mandan también un mensaje a 10 con su pid y el tamaño del fichero que han abierto (11 abre /etc/passwd y 14 Trabajo).
En el else, el padre redirecciona su salida estandar al fichero /tmp/results, ya no sacará nada por pantalla. Luego lee de la tubería tantos mensajes como hijos ha creado (4).
Cuando ha leído todos los mensajes, comprueba para cada uno si el tamaño (mes1.n_bytes) es cero, lo que indica que ese hijo habrá echo un exit(1), al resto les manda la señal 15 para que salgan del pause() en el que se habrán quedado.
Finalmente el padre se queda haciendo wait() en un bucle hasta que sepa con seguridad que han muerto todos sus hijos. Si tenía n hijos, necesita que wait devuelva n identificadores de procesos.

NOMBRE:

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include "stdlib.h"
#include "stdio.h"

struct mensa{
  int pid;
  int n_bytes;
} mes1,hijos[30];  // dos variables de este tipo.
                   // una de ellas un vector de 30 componentes


void foo(int n_sig){
  printf("Soy %d, me ha llegado %d\n",getpid(),n_sig);
  signal(n_sig,foo);
}

int main(int argc,char **argv, char **envp){
  int i,j,n,tipo,l,status;
  int fic1,fic2;
  int tubo[2];
  char buff[2048];


  if((argc<3)||(argc>30)){
    fprintf(stderr,"Pocos o demasiados parametros\n");
    exit(1);
  }
  signal(15,foo);
  pipe(tubo);
  for(i=1;i<(argc-1);i++){
    tipo=fork();
    if(tipo==0)
        break;
  }
  if(tipo==0){
    struct stat s1;
    printf("Soy %d y abrire %s\n",i,argv[i]);
    mes1.pid=getpid();
    fic1=open(argv[i],O_RDONLY);
    if(fic1<0){
      mes1.n_bytes=0;
      perror("fallo en el fichero");
      write(tubo[1],&mes1,sizeof(mes1));
      exit(1);
    }
    fstat(fic1,&s1);
    mes1.n_bytes=s1.st_size;          //tamanyo en bytes del archivo
    write(tubo[1],&mes1,sizeof(mes1));
    pause();
    printf("(%d)Pues parece que ha ido bien\n",mes1.pid);
```



```c
  }else{   // de if(tipo==0){
    close(1);
    fic2=creat(argv[i],0777);
    if(fic2<0)perror("En creat");
    j=0;
    while(i>1){
      read(tubo[0],&(hijos[j]),sizeof(struct mensa));
      printf("He leido el mensaje de %d (%d)\n",
             hijos[j].pid,
             hijos[j].n_bytes);
      i--;
      j++;
    }

    printf("tengo %d hijos\n",j);
    /* Completar con las lineas necesarias para todos los     */
    /* hijos bloqueados continuen su ejecucion                */
    for(i=0;i<j;i++){
        if(hijos[i].n_bytes>0){
              kill(hijos[i].pid,15);
        }
    }

    /* Completar con las lineas necesarias para que no quede  */
    /* ningun hijo zombie                                     */
    /* Se debe imprimir el pid de cada hijo que ha muerto     */
    /* y el motivo de su muerte                               */
    i=0;
    do{
      l=wait(&status);
        if(l>0){
          printf("hijo: PID(%d) razon(%04X)\n",l,status);
                i++;
        }
    }while(i<j);
  }
  exit(1);
}
```


E2

Salida por pantalla:

Soy 11 y abriré /etc/passwd
Soy 12 y abriré F1
Soy 13 y abriré Notas
Soy 14 y abriré Trabajo
Soy 11 me ha llegado 15
(11) Pues parece que ha ido bien
Soy 14 me ha llegado 15
(14) Pues parece que ha ido bien

Salida de error de los procesos, contenido de /tmp/errores:

Fallo en el fichero: permiso denegado
Fallo en el fichero: permiso denegado

Contenido de /tmp/results:

He leído el mensaje de 11(2035)
He leído el mensaje de 13(0)
He leído el mensaje de 14(234521)
He leído el mensaje de 12(0)
tengo 4 hijos
Hijo:PID(11) razon(0100)
Hijo:PID(12) razon(0100)
Hijo:PID(10) razon(0100)
Hijo:PID(14) razon(0100)

NOMBRE:

3. En la siguiente tabla se indican las ráfagas de CPU y E/S de un conjunto de procesos, donde los valores indicados son periódicos (se realizan de forma cíclica) y el proceso 3 termina tras la segunda ráfaga de CPU. También se muestra el instante de llegada y el valor de prioridad inicial asignado a cada uno. Indicad, en la tabla que se proporciona, el estado de cada proceso (CPU ó E/S sólo) y su prioridad en cada ciclo (a mayor número, mayor prioridad) si, en un sistema monoprocesador, se utiliza un algoritmo de planificación por **prioridades dinámicas expulsivas**. Un proceso incrementará su prioridad en 2 por cada ciclo completo que esté esperando en la cola de preparados. Cada vez que un proceso sale de la CPU recuperará su prioridad original, tanto si va a E/S como si vuelve a la cola de preparados.

PID	Prioridad	Instante llegada	Ráfagas
1	2	0	7 CPU 3 E/S
2	7	2	3 CPU 7 E/S
3	9	4	5 CPU 4 E/S
4	3	4	2 CPU 6 E/S

Para cada proceso indicad con una C si está usando la CPU y con una E si está en E/S. Dejad en blanco la casilla cuando el proceso esté en la cola de preparados y sombread las que queden tras la finalización del proceso 3.

NOTA: Los instantes de llegada indican el ciclo durante el que el proceso llega a la cola de preparados. En ese ciclo no podrán entrar en ejecución (ni se tendrá en cuenta para el aumento de prioridad), pues el planificador se ejecuta al final de cada ciclo incrementando primero la prioridad de los procesos que lo requieran y eligiendo después el siguiente proceso para ejecución. **(1ptos)**

		1				5					10					15					20					25					30					35
1	Estado	C	C						C						C	C	C	C	E	E	E									C	C	C	C	C			
	Prioridad	2	2	2	4	6	8	10	12	2	4	6	8	10	12	12	12	12				2	4	6	8	10	12	14	16	16	16	16	16	2	4	6
2	Estado			C	C			C	E	E	E	E	E	E	E				C	C	C	E	E	E	E	E	E	E						C	C	C
	Prioridad			7	7	7	9	11								7	9	11	13	13	13								7	9	11	13	15	17	17	17
3	Estado					C	C		C	C				C	E	E	E	E				C	C	C	C	C											
	Prioridad					9	9	9	11	13	13	9	11	13					9	11	13	15	15	15	15	15											
4	Estado											C	C	E	E	E	E	E	E								C	C	E	E	E	E	E	E		
	Prioridad					3	5	7	9	11	13	15	15							3	5	7	9	11	13	15	17	17							3	5

NOMBRE:

E3

4. En un sistema se utilizan semáforos para resolver el problema de la comunicación y sincronización de procesos. Las primitivas de utilización de semáforos son:

```
int crea_sem(int val_inicial);  // devuelve identfi de semáforo
int down(int sem_id);           // devuelve valor del semáforo
int up(int sem_id);             // devuelve valor del semáforo
```

Utiliza estas llamadas para completar el siguiente código de un proceso. Queremos que no hayan problemas en el acceso a la sección crítica utilizando sólo un semáforo.
Se deben escribir las funciones enter_region y leave_region (el nombre indica la funcionalidad de cada una de ellas), crear un semáforo con un valor inicial adecuado y colocar las llamadas a las funciones creadas en los puntos apropiados del código.
Se puede añadir el código adicional que se considere necesario.

```
int mutex;

void enter_region(void){
     down(mutex);
}

void leave_region(void){
     up(mutex);
}



int main(void)
{
int continuar=1;
//crear semáforo

mutex=crea_sem(1);

parte_1();
fork();
fork();
while (continuar){
  parte_2(getpid());
  //Aquí se llama a…

  enter_region();

     //region critica
       parte_3(getpid());
     //fin region critica
  //Aquí se llama a…

  leave_region();

  continuar=parte4(getpid());
  } //fin del while



}//fin de main
```

(0,5 puntos)

NOMBRE:

5. El esquema de un disco con nodos-i (MINIX) es el siguiente:

Arranque	Superbloque	Mapa de bits Nodos-i	Mapa de bits Zonas	Nodos - i	Zonas de datos

Si sabemos que en el disco hay 5 archivos de 1 KB, 5 de 5 KB y 5 de 100KB (esta información incluye los directorios). Teniendo en cuenta los siguientes datos:

- Tamaño del disco : 2GBytes
- El tamaño del bloque de arranque y del superbloque es de 1 bloque
- El tamaño del nodo-i es de 64 bytes y que se reserva espacio para 4.096 nodos-i
- 1 zona = 1 bloque = 2 KB (2048 bytes)

Se pide:

- Tamaño en bloques de Mapa de bits nodos-i, Mapa de bits Zonas, Nodos-i, Zonas de datos
- Número de nodos-i ocupados y libres.
- Número de bloques de datos ocupados y libres.

(1,5 puntos)

Si el tamaño del bloque es de 2Kbyte y el del nodo i es de 64 bytes, tenemos que en cada bloque caben:

$2^{11}/2^6=2^5$ (32 nodos-i por bloque)

Si necesitamos 4096:

$2^{12}/2^5=2^7$ (128 bloques)

El mapa de bits de nodos-i es de 1 bloque ya que necesitamos 4096 bits que ocupan 512 bytes < 2K Bytes.

Para calcular el tamaño del mapa de bits de zonas, vamos a suponer que el número de zonas de datos es el necesario para almacenar 2Gbytes, aunque en realidad es algo menos pues al principio del disco tenemos bloques que nos son de datos, pero este número de bloques inicial es despreciable (al final verificaremos que es realmente despreciable)

Para 2Gbytes de datos en bloques de 2Kbyte, necesitamos 1MBloques por tanto para el mapa de bits de zonas necesitaremos 1Mbit, como en cada bloque caben 16Kbits, el número de bloques necesario es:

$2^{20}/2^{14}=2^6$ (64 bloques)

Por tanto tendremos:

Arranque 1 bloque	Superbloque 1 bloque	Mapa de bits Nodos-i 1 bloque	Mapa de bits Zonas 64 bloques	Nodos – i 128 bloques	Zonas de datos 1M-195 bloques

NOMBRE:

E4

Podemos observar que el número real de bloques necesarios para la zona de datos es 1M-195 bloques en lugar de 1M bloques que habíamos supuesto al principio del cálculo del número de bloques del mapa de bits de zonas. Como la diferencia es de 195 bloques (195 bits) el número de bloques del mapa de bits de zonas sigue siendo el mismo, ya que en cada bloque cabían 16Kbits > 195.

Como el sistema tiene 15 archivos, tendremos 15 nodos-i ocupados y el resto (4096-15=4081) libres

Respecto a los bloques de datos, vamos a calcular ahora los que están ocupados:

- los 5 ficheros de 1 Kbyte, necesitan cada uno de ellos un bloque (2Kbyte)
- los 5 ficheros de 5 Kbytes, necesitan cada uno 3 bloques (5/2=2,5 → 3)
- los 5 ficheros de 100 Kbyes, necesitan cada uno 50 bloques (100/2) y como en el nodo-i no caben los 50 números de bloques, necesitamos un bloque indirecto. Si los números de bloque fuesen de 32 bits, en un bloque indirecto de 2 Kbytes podríamos poner 512 números de bloques > 40 ó 43 que son los que necesitamos, luego es suficiente con un bloque indirecto por archivo de 100Kbytes.

En definitiva, el número de bloques de datos ocupados es (5*1+5*3+5*51=5*55=275)

El numero de bloques libres será: 1M-195-275=1M-470

NOMBRE:

6. Sea la llamada al sistema, que se ejecuta en un sistema de ficheros tipo MINIX :

descriptor=creat("fi2",0700)

a) Si el fichero fi2 no existía:
¿Qué estructuras de datos se utilizan durante se ejecución y cómo se utilizan?
Indicad el número de getblocks() y putblocks() y las modificaciones de las estructuras de datos del sistema operativo y del sistema de ficheros en el disco.

b) Si el fichero fi2 existía y tenía un tamaño de 1KB, cómo cambiaría la respuesta de las cuestiones anteriores

(1,5 puntos)

Solución a)

Lo primero es comprobar si el fichero "fi2" existe en el directorio actual.
Para ello hay que hace n_1 getblock+putblocks de los datos del directorio actual.
Una vez se ha comprobado que el fichero no existe, hay que añadir una nueva entrada al directorio de trabajo.
Para ello se comprueba que se tengan permisos de escritura en el directorio (nodo-i) y se mira si en el último bloque de datos del directorio cabría la nueva entrada de directorio.
Suponiendo que cabe: 1getblock+putblock para escribir la entrada en el directorio.
Suponiendo que no: 1getblock+putblock para escribir la entrada en el directorio, además de buscar un nuevo bloque libre. Esto siempre supone leer el superbloque del dispositivo para saber cual es el primero bloque libre, modificar el mapa de bits para marcarlo como ocupado y buscar en el mapa de bits de zonas el siguiente bloque libre (n_2 getblocks+putblocks) y almacenar este valor en el superbloque del dispositivo en memoria.
Además, tenemos que buscar un nodo-i libre para el fichero, lo que supone leer el superbloque del dispositivo para saber cual es el primer nodo-i libre, modificar el mapa de bits de nodos-i y buscar en el mapa de bits el siguiente nodo-i libre (n_3 getblocks+putblocks).
Traemos el nuevo nodo-i a memoria (1 getblock) y lo inicializamos.
Creamos una nueva entrada en la tabla FILP, campo POS=0, modo=WRONLY y una referencia.
Buscamos en la tabla de descriptores de ficheros del proceso la primera entrada libre y almacenamos en ella una referencia a la entrada de la tabla FILP creada.
Se devuelve el índice de la tabla de descriptores de ficheros que se ha usado como resultado de la operación o -1 si ha habido un error.
Getblocks: $n_1 + 1 + [\ n_2\ ] + n_3 + 1$ con $n_2 \geq 0$, $n_1 \geq 1$, $n_3 \geq 1$
Putblocks: $n_1 + 1 + [\ n_2\ ] + n_3$ con $n_2 \geq 0$, $n_1 \geq 1$, $n_3 \geq 1$

Solución b)

Lo primero es comprobar si el fichero "fi2" existe en el directorio actual.
Para ello hay que hace n_1 getblock+putblocks de los datos del directorio actual.
Una vez se ha comprobado que el fichero existe, hay que traer el nodo-i del fichero para comprobar permisos (1 getblock).

NOMBRE:

E5

Si se tienen permisos de escritura y el fichero contenía datos, hay que liberarlos. Para ello se lee del nodo-i la lista de bloques que ocupaba (en este caso 1 bloque si el tamaño de bloque es mayor o igual a 1Kbyte).
Se marca el bloque ocupado como libre en el mapa de bits de zonas (1 getblocks+putblocks) y se modifica el superbloque, si es necesario, para indicar cual es el primer bloque libre del sistema.
Se modifica el nodo-i para que tenga tamaño 0 y ningún bloque de datos asociado.
Creamos una nueva entrada en la tabla FILP, campo POS=0, modo=WRONLY y una referencia.
Buscamos en la tabla de descriptores de ficheros del proceso la primera entrada libre y almacenamos en ella una referencia a la entrada de la tabla FILP creada.
Se devuelve el índice de la tabla de descriptores de ficheros que se ha usado como resultado de la operación o -1 si ha habido un error.
Getblocks: n_1+1 +1 con $n_1 \geq 1$
Putblocks: n_1+ 1 con $n_1 \geq 1$

7. Justifica si las siguientes sentencias son ciertas o falsas.
   A. En una máquina PVM no existe riesgo de que N procesos entren en una situación de interbloqueo
      Es falsa.
      Con un contra ejemplo: la tarea A está bloqueada en un receive, esperando que la tarea B le envíe un mensaje. La tarea B está bloqueada en un receive, esperando que la tarea A le envíe un mensaje. Ninguna de las dos puede salir del receive, y ninguna envía el mensaje que la otra espera

   B. En un sistema de memoria virtual paginado, el número de marcos usados es igual a la suma de páginas accedidas por cada proceso.
      Es falsa.
      Por un lado, no todos los marcos son usados por los procesos, el sistema operativo también necesita marcos para ejecutarse y tener sus variables y estructuras de datos, como por ejemplo la cache de bloques.
      Por otro lados, existe la posibilidad de compartir páginas entre procesos, es decir, un marco contiene un página que es común (compartida) por varios procesos.

   C. Los TLB son un recurso HW utilizado para mejorar las prestaciones de los algoritmos de sustitución de páginas.
      En principio es falsa.
      El objetivo de los TLB es almacenar traducciones de número de página a número de marco, de forma que se ahorren accesos a memoria en sistemas de memoria virtual paginados.

   D. Que aparezca un problema de hiperpaginación no depende de la cantidad de memoria física instalada en el sistema.
      Es falsa.
      Para un mismo conjunto de procesos, cuanta más memoria física tengamos más difícil será la aparición del problema de hiperpaginación.

NOMBRE:

E. Los monitores se pueden usar en cualquier sistema multiprocesador.
Cierta.
Aunque los monitores necesitan soporte del lenguaje de programación, la implementación de los monitores se basará en el soporte que ofrezca el sistema operativo.
En un sistema multiprocesador el que dispongamos sólo de paso de mensajes el compilador/intérprete del lenguaje de programación usaría las primitivas de mensajes para implementar los monitores.
Un ejemplo real de esta situación es el lenguaje Java, que soporta monitores y se dispone de versiones de java para casi cualquier sistema operativo o máquina, incluyendo multicomputadores.

(1,5 puntos)

NOMBRE:

E6

ACSO II. Examen Junio 2005

Problema 1 (permisos)

chmod1 444 F1

julio profe todos
r-X r-X (r-X)

Puedo ejecutarlo
SetUID + SETGID → soy julio profe
F1 tengo permisos de escritura? No
de todas formas daría igual, chmod
sólo exige que seas el propietario, y
no lo soy

cambia2 Trabajo

pepe staff all
(r-x) r-- r--

Puedo ejecutarlo
setUID + setGID → soy pepe staff
¿Puedo modificar Trabajo?
↳ pepe profe all
(rw-) r-- ---
si puedo

cp1 Trabajo /tmp/Trab2

julio profe all
r-x r-x (r-x)
setUID
setGID
soy julio, profe

pepe profe all
rw- (r--) ---

all (rwx)

siendo pepe alumn, puedo ejecutar cp1, y ya que tiene sUID y sGID a 1, durante la ejecución me convierto en julio profe, y por tanto puedo leer el fichero origen

chmod2 444 Notas

julio profe all
r-x r-x (r-x)
SGID → soy profe

julio staff
no soy julio → no puedo

cat1 Notas

julio profe all
r-x r-x r-x
sGID → soy profe

julio staff all
rw- r-- (---)
no puedo leerlo

cp2 Examen ../Exam2

pepe staff all
(r-x) r-x r--
sUID → pepe
sGID → staff

julio staff all
rw- (---) ---

root staff all
rwx (rwx) r-x

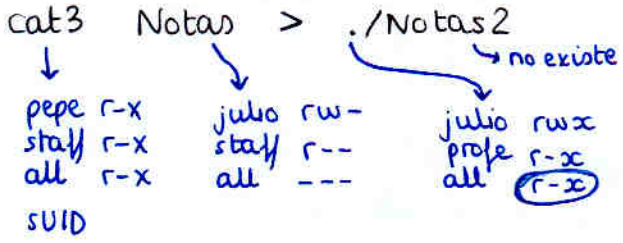
aunque si que puedo escribir en el directorio destino, no puedo leer Examen

mv1 Trabajo Trab2

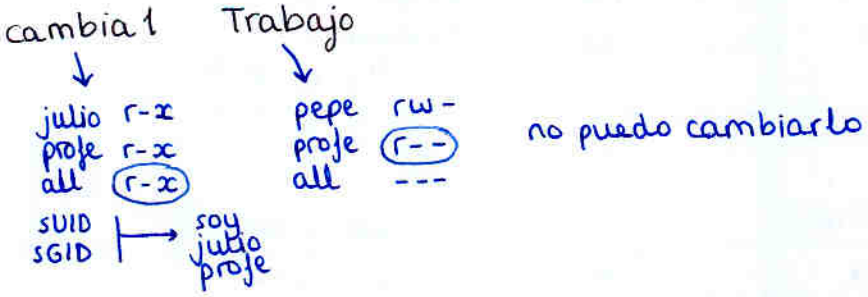
julio staff all
r-x r-x (r-x)
sGID → soy staff

pepe profe all
rw- r-- ---
para mv me dan igual los permisos del fichero

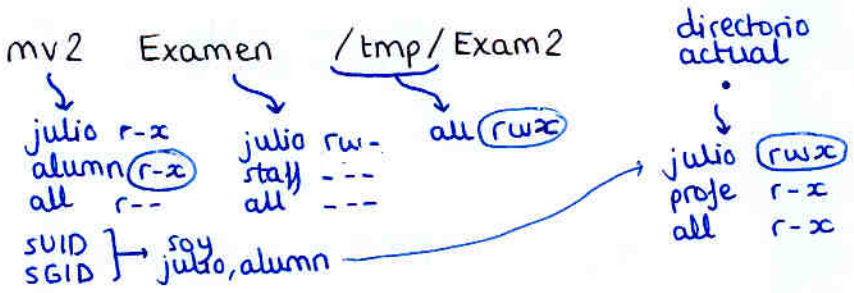
directorio actual .
julio profe all
rwx r-x (r-x)
→ me interesa sólo poder escribir en el directorio: no puedo



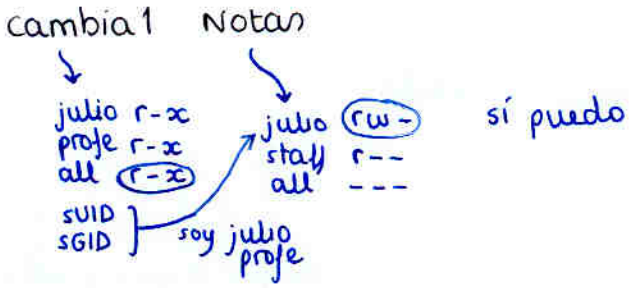
La redirección la hace el shell (siendo pepe alumn) ANTES de ejecutar cat, por tanto lo que interesa es ver si tengo permiso de escritura en el fichero de salida.
Como no existe habrá que crearlo, para eso necesito permiso de escritura en el directorio
→ No tengo



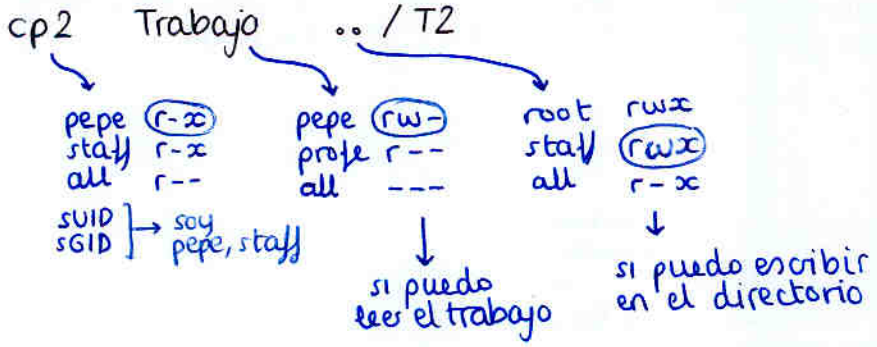
no puedo cambiarlo



De nuevo, para hacer mv no me importan los permisos del fichero, sino los permisos de los directorios.
Tengo permiso de escritura en ambos



sí puedo



si puedo leer el trabajo

si puedo escribir en el directorio

chmod2 444 F1

julio r-x
profe r-x
all (r-x)
sGID → soy profe

pepe
profe

soy el propietario, por tanto aunque no tuviera permisos de nada podría hacerle chmod

> chmod funciona ⟺ (si y sólo si) soy el propietario del fichero, independientemente de los permisos (i)

rm1 Trabajo

root r-x
profe r-x
all (r--)
SUID

→ no puedo ejecutar rm1.

Nota: si sí hubiera podido, luego habría que comprobar el permiso de escritura en el directorio, NO en el archivo.

> i.e. si tengo permiso de escritura en un directorio puedo mover y borrar sus archivos aunque no tenga permisos de nada en ellos (i)

cp1 Notas Notas2 → no existe.

julio r-x
profe r-x
all (r-x)
SUID
SGID → soy julio profe

julio (rw-)
staff r--
all ---

↓ Puedo leer

Por tanto miramos el directorio

julio (rwx)
profe r-x
all r-x

↓ Puedo escribir

rm3 Examen

julio r-x
profe r-x
all (r-x)
SUID
SGID → soy julio, profe

julio rw-
staff ---
all ---

No me importan los permisos del archivo, sólo del directorio

julio (rwx)
profe r-x
all r-x

cat2 Trabajo

julio r-x
profe r-x
all (r--)

pepe rw-
profe r--
all ---

No tengo permiso de ejecución de cat2

mv1 Notas Notas2

julio r-x
staff r-x
all (r-x)
sGID → soy staff

directorio .

julio rwx
profe r-x
all (r-x)

No puedo escribir en el directorio, por tanto no puedo mover

chmod1 777 Notas

julio r-x
profe r-x
all (r-x)
SUID
SGID → soy julio profe

(julio)
staff

si puedo

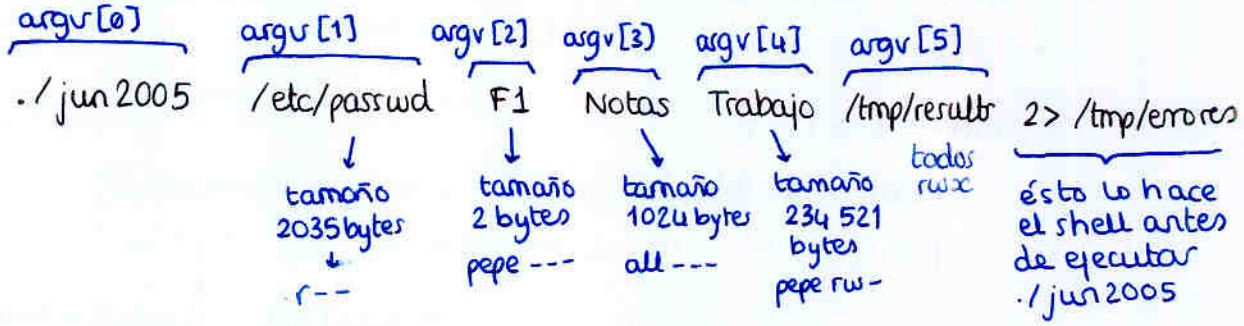
cp1 Examen /tmp/examen

julio r-x
profe r-x
all (r-x)
SUID
SGID → soy julio, profe

julio (rw-)
staff ---
all ---
↓
puedo leer

all (rwx)
↓
puedo escribir

Problema 2 (llamadas, señales, tuberías, hijos)

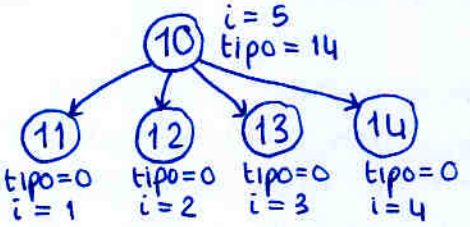


Vamos poco a poco en el código

- comprueba parámetros
- se protege contra la señal 15 (y por tanto todos sus hijos lo estarán)
- crea tubería (y por tanto todos sus hijos la tendrán)

```
for (i=1; i<(argc-1); i++) {
    tipo = fork();
    if (tipo==0)
        break;
}
```

Tras éste bucle se habrán creado 4 hijos, todos con tipo=0.
El padre se queda con tipo=14 e i=5



} todos los hijos están protegidos contra señal 15 y tienen tubería

if (tipo==0) : lo que hacen los hijos

- escriben: "Soy i y abriré argv[i]"
- cada uno se guarda mes1.pid = getpid() → recuerda: cada uno tiene su propia copia de mes1
- cada uno abre argv[i]
  - el 12 y el 13 no pueden ya que pepe no tiene permiso de lectura en F1 y Notas:
    - → escriben en salida de error "/tmp/errores/" "fallo en el fichero"
    - → escriben en la entrada de la tubería su mes1.pid = 12 | 13 ; mes1.n_bytes = 0 | 0
    - → mueren
  - el resto obtiene el nº de bytes a mes1.n_bytes y escriben su mes1 en la tubería

mes1.pid = 11 | 14
mes1.n_bytes = 2035 | 234 521

y se pausan

} else { lo que hace el padre (con i = 5)

- cierra su salida estándar
- crea tmp/results → lo pondrá como salida estándar porque es el hueco libre
- j = 0

```
while ( i > 1 ) {
    read ( tubo[0], &(hijos[j]), sizeof(struct mensa));
    printf ( "He leido el mensaje de %d (%d) \n"
                                    ↑          ↑
                              hijos[j].pid   hijos[j].n_bytes
    i--;
    j++;
}
```

lo hará para i=5 j=0 | i=4 j=1 | i=3 j=2 | i=2 j=3 | (i=1 j=4) → se sale antes de hacerlo

Guarda en su vector de mensajes los 4 mensajes que le han enviado sus hijos y los muestra por salida estándar (en este caso /tmp/results)

y ahora escribe en salida estándar "Tengo 4 hijos"

→ completar para desbloquear a los hijos

Tiene que enviar una señal a aquellos hijos que están bloqueados → i.e. hijos[j].n_bytes > 0

que les envíe la señal 15, para la cual están protegidos

```
for (i=0; i<j; i++)
    if (hijos[i].n_bytes > 0) { kill (hijos[i].pid, 15); }
```

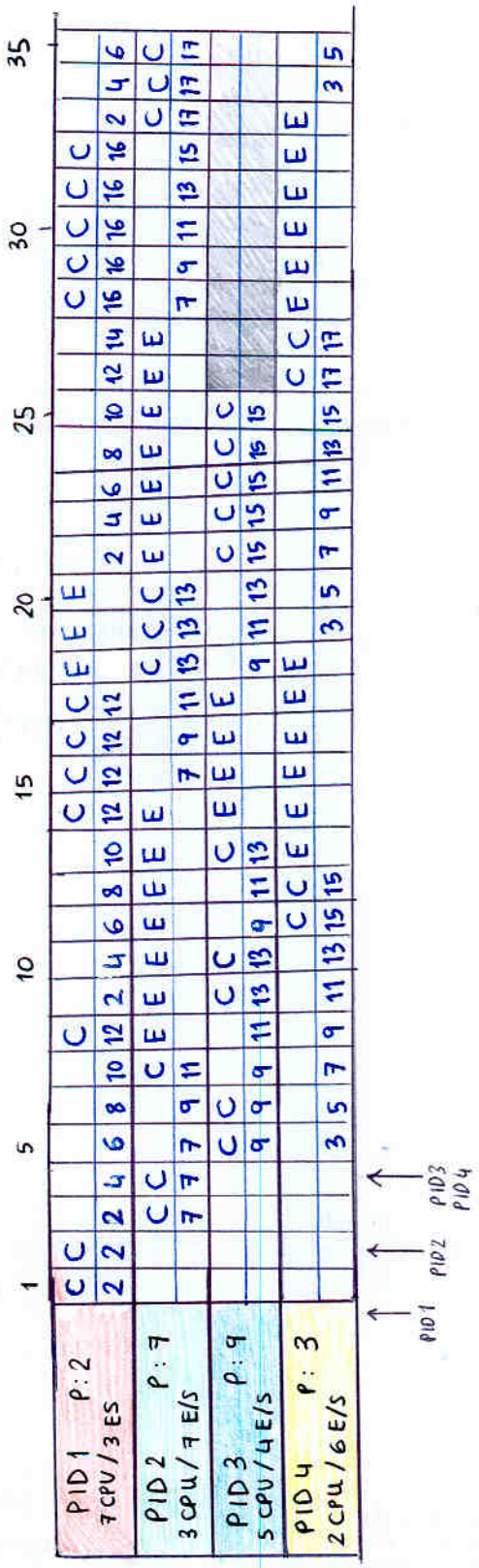
- Al recibir la señal, los 2 procesos que quedan vivos van a función foo y escriben en salida estándar (la suya aún es la pantalla) "soy getpid(), me ha llegado 15"
- Luego continuar ejecución

```
printf ("(mes1.pid) Pues parece que ha ido bien");
exit(1);   ← código terminación 1
```

→ Completar para que no queden hijos zombies

```
i=0;
do {
   l = wait(&status),
     if (l>0) { printf ("hijo l, razon status"); i++ }
} while (i<j);
```

Problema 3. (Planificador de procesos)



Prioridades dinámicas expulsivas

Al final de cada ciclo:

- aumentar prioridad en 2 si el proceso estaba esperando
- Entra a la CPU el proceso de mayor prioridad
- Si un proceso ha salido de CPU recupera su prioridad base

Problema 5 (sistema archivos MINIX)

- 1 zona = 1 bloque = 2kB (2048 bytes)
- bloque de arranque = 1 bloque
- superbloque = 1 bloque
- 4096 nodos-i, tamaño de nodo-i 64 bytes
- Tamaño disco 2GB
- Archivos: 5 de 1kB
  5 de 5kB
  5 de 100 kB

$$n^o \text{ bloques} = \frac{1 \text{ bit/nodo-i} \cdot 4096 \text{ nodos-i}}{8 \text{ bits/byte} \cdot 2048 \text{ bytes/bloque}} = \frac{2^{12}}{2^3 \cdot 2^{11}} = 0'25 \rightarrow 1 \text{ bloque}$$

$$n^o \text{ bloques} = n^o \text{ tot blocks} - (1+1+1+64+128) = \frac{2GB}{2kB/blog} - (195) = 2^{20} - 195 = 1M - 195$$

$$n^o \text{ bloques} = \frac{1 \text{ bit/zona} \cdot \left(\frac{2GBytes}{2kBytes/zona}\right)^{n^o \text{ zonas}}}{8 \text{ bits/byte} \cdot 2048 \text{ bytes/bloque}} = \frac{2^{20} \text{ bits}}{2^{14} \text{ bits/bloque}} = 2^6 \text{ bloque} \rightarrow 64 \text{ bloques}$$

$$n^o \text{ bloques} = \frac{64 \text{ bytes/nodo-i} \cdot 4096 \text{ nodos-i}}{2048 \text{ bytes/bloque}} = \frac{2^7 \cdot 2^{12}}{2^{11}} = 2^8 = 128 \text{ bloques}$$

Lo que ocupan los archivos:

- 1kB ⇒ 1 bloque/archivo ⇒ 5 bloques
- 5kB ⇒ 3 bloques/archivo ⇒ 3·5 = 15 bloques
- 100kB ⇒ 50 bloques/archivo > 7 ⇒ no caben en el nodo-i, necesitamos bloque indirecto
  + 1 bloque indirecto/archivo
  → 51 bloques/archivo ⇒ 51·5 = 255 bloques

total: 275 bloques ocupados

n^o bloques libres = (1M - 195) - 275 = 1M - 470 bloques libres

Problema 6 descriptor = creat ("fi2", 0700)

¿Qué estructuras de datos y cómo se utilizan?
Indicar número de getblocks() y putblocks()
Indicar modificaciones en las estructuras de datos y sistema de ficheros

a) Si el fichero "fi2" no existía

- Primero comprobar si fi2 existe en el directorio actual
  - Leer nodo-i del directorio actual (ya en cache de bloques gracias al puntero en la tabla del proceso)
  - Leer los bloques de datos del directorino buscando fi2
    ⇒ n_1 [getblock() + putblock()]
  - Si no lo encontramos ya sabemos que no existe

(*)

- Hallar ~~zona~~ y nodo-i libre para el nuevo fichero
  - Leer del superbloque cual es la ~~zona~~ y nodo-i libre
  - Actualizar ~~ambos~~ mapas de bits
    - ~~mapa bits zonas~~ : actualizar con un '1' y buscar siguiente libre
      ⇒ ~~n_2~~ [getblock() + putblock()]
    - mapa bits nodos-i : actualizar con un '1' y buscar siguiente libre
      ⇒ n_3 [getblock() + putblock()]
  - Actualizar el superbloque con la nueva siguiente ~~zona~~ y nodo-i libre.

→ me he equivocado; en el open/creat basta con tener el nodo-i, no hace falta reservar zonas para el fichero → ya se encargará write. ⚠

- Crear el nuevo nodo-i
  sabiendo ya cuál es el nodo-i libre, hacemos getblock() del bloque donde esté el nodo-i y rellenamos todo lo necesario → no hay putblock
- Añadir entrada a la tabla FILP con el puntero al nodo-i recién creado pos=0, modo WRONLY, nºrefs = 1
- Añadir entrada a la tabla de descriptores de fichero, almacenando la referencia a la entrada FILP

(*) ↙ esto se haría después de comprob. si fi2 existe

- Ya que hemos creado un nuevo fichero, habrá que añadirlo a la tabla del directorio
  - si cabe en el último bloque ⇒ 1 getblock() + putblock() para añadir la entrada
  - si no cabe en el último bloque ⇒
    buscar bloque libre en el superbloque, actualizar mapa de zonas, y actualizar el superbloque con el siguiente libre
    ⇒ (n_2 + 1)[getblock() + putblock()]
    ↳ para añadir la entrada en el nuevo bloque.
    y actualizar nodo-i del directorio con nuevo bloque

TOTAL:

getblock(): $n_1 + n_2 + 1 + n_3 + 1$
($n_1 \geq 1$, $n_2 \geq 0$, $n_3 \geq 1$)

putblock(): $n_1 + n_2 + 1 + n_3 + \emptyset$

b) si el fichero "fi2" sí existe

- comprobar si existe
  recorriendo los bloques del directorio indicados en el nodo-i
  n_1 [getblock() + putblock()]
- Al ver que sí existe, hay que 'borrarlo'
  - comprobar permisos con el nodo-i (1) (getblock) → no putblock ya que necesitaremos el puntero al nodo-i del nuevo fichero en blanco
  - eliminar los bloques listados en el nodo-i (dejar el archivo con 0 bloques asociados)
  - actualizar mapa de zonas/bloques
    ⇒ n_2 [getblock() + putblock()]
  - actualizar superbloque con nueva zona libre
- Añadir entrada a la FILP
  - POS = 0, nº ref = 1, modo WRONLY, referencia al nodo-i

¿Y si el archivo la habían abierto ya otros procesos? Da igual, eso estará en otra entrada FILP.
open ⟺ nueva entrada FILP

(i)

- Añadir nueva entrada en tabla de descriptores de fichero con referencia a la entrada de la FILP

TOTAL:

getblock(): $n_1 + 1 + n_2$

putblock(): $n_1 + \emptyset + n_2$

se podría asegurar que n_2 (nº de getblocks y putblocks para actualizar el mapa de zonas) es igual a uno, ya que el fichero ocupaba un bloque y por tanto sólo hay que actualizar un bit del mapa

pero ¿qué pasa con la actualización del "primer bloque libre" en el superbloque? ¿no requeriría esto n get/putblocks para recorrer el mapa de bits y ver el 1º libre?