<u>CAPÍTULO I.</u> Conceptos fundamentales de UNIX.

Tipos de archivos

Programas y procesos

Señales

Process−ID

Permisos

Otros atributos de proceso

TIPOS DE ARCHIVOS

Ficheros ordinarios: contienen bytes de datos organizados en vectores lineales. No tienen nombre, se nombran por medio de números (i−number). Estos números son los índices a la tabla de i−nodes. Cada i−node consta de las siguientes partes:

- Tipo de fichero (ordinario, directorio o especial)
- Número de links
- user−ID y group−ID
- Permisos para el propietario, grupo y otros
- Tamaño en bytes
- Fecha del último uso, última modificación y último cambio de permisos y
- Punteros a los bloques de memoria donde se encuentran los datos del fichero

Directorios: Se permite usar nombres en los directorios para designar a los ficheros que contienen. Cada directorio consta de una con el nombre del fichero y otra con el i−number del fichero correspondiente, un link será el par nombre/i−nodo.

Cuando se le manda al kernel acceder a un fichero por nombre, se busca en el directorio el nombre y su correspondiente i−number, a continuación se busca el i−nodo del fichero que se corresponde con ese i−number y el i−nodo nos dice donde encontrar el fichero en el disco. Por ejemplo, con el path memo/julio/oscar se busca el i−nodo del directorio actual para localizar sus bytes de datos, encontramos memo entre ellos, extraemos su i−number y de él sacamos su i−nodo para encontrar los bytes de datos de memo, entre ellos buscamos julio y cogemos su i−number, de él sacamos el i−nodo para localizar sus bytes de datos y encontramos oscar, cogiendo su correspondiente i−nodo.

Ficheros especiales: se dividen en block special files que contienen vectores de bloques de tamaño fijo para controlar I/O y character special files que no necesitan regalas especiales para su funcionamiento y pueden hacer I/O.

PROGRAMAS Y PROCESOS

Programa: son datos e instrucciones guardados en un fichero ordinario o en disco. Su i−nodo lo marca como ejecutable. Para crearlo se escribe el programa en un lenguaje de programación (C para UNIX), lo guardamos como fichero de texto y por medio de un compilador se traduce a lenguaje máquina obteniendo un fichero objeto y si no tiene fallos se crea el ejecutable.

Procesos: Para ejecutar un programa, el kernel crea un entorno donde se pueda desarrollar (proceso), que consta de segmento de instrucciones, segmento de datos de usuario y segmento de datos del sistema. El proceso proporciona recursos como ficheros abiertos, más memoria... que el programa no presenta.

SEÑALES

Se mandan del kernel al proceso para indicar por ejemplo una violación de segmento, alarma de reloj...

Hay unas 19 señales, el proceso podrá aceptar la acción asociada por defecto a la señal recibida (finalización del proceso), puede ignorar la señal o bien la puede capturar pasándola como parámetro (número del 1 al 19) a una subrutina de tratamiento de la señal.

PROCESS−ID (PID)

Todos procesos se designan con un entero positivo único llamado pid. Todos procesos menos uno (proceso 0 que es creado por el propio kernel) tienen un proceso padre. Los datos del sistema de un proceso guarda su ppid (pid del proceso padre). Si un hijo se queda huérfano al finalizar su padre antes que él, toman como ppid 1 que es el pid del proceso de inicialización INIT.

PERMISOS

El user−ID (uid) es un entero positivo asociado con el login de un usuario en el fichero de passwords (/etc/passwd). Cuando un usuario entra al sistema, el comando login hace este ID el uid del primer proceso creado (login shell). Los procesos que descienden tienen el mismo uid.

Los usuarios también se dividen en grupos que tienen sus propios Ids (gid). Los grupos están definidos en /etc/group.

Un usuario se puede cambiar de grupo, lo que hace que su gid cambie, este es heredado por los hijos del proceso que cambia de grupo. Estos dos Ids vistos hasta ahora se llaman ruid (real user ID) y rgid (real group ID).

Otros Ids son el effective user−ID (euid) y el effective group−ID (egid) que normalmente son iguales a los anteriores. El effective ID indica permisos y el real ID la identidad. Cada fichero tiene en su i−nodo (como se ha visto antes) un uid del propietario y un gid del grupo del propietario.

Existen 9 bits de permisos (3 para el usuario, para el grupo y para otros de lectura, escritura y ejecución). Si euid es 0, el usuario va a ser el superuser (administrador del sistema)

OTROS ATRIBUTOS DE PROCESO

Otros atributos serán grabados en el segmento de datos de sistema del proceso.

Uno de ellos va a ser el descriptor de ficheros (fd) que es un entero del 0 al 19 (un descriptor de ficheros para cada proceso que se abre y 2 para cada tubería que se crea). El hijo no hereda los descriptores de fichero de los ficheros abiertos por el padre sino que hereda copias de ellos. Son índices a la tabla de procesos abiertos (padre e hijo tienen el mismo puntero de fichero)

<u>CAPÍTULO II.</u> Conceptos básicos de ficheros de E/S

Descriptores de fichero

creat

open

write

read

close

lseek

DESCRIPTORES DE FICHERO

Cada proceso tiene un conjunto de 20 descriptores de fichero (fd) numerados del 0 al 19. Los tres primeros se abren automáticamente cuando el proceso comienza (0 es la entrada estándar, 1 la salida estándar y 2 la salida estándar de errores). Un proceso UNIX leerá de 0, escribirá en 1 y usará 2 para mensajes importantes. Estos tres fd están preparados para ser usados inmediatamente con read y write; los otros 17 se pueden usar con ficheros, tuberías (pipes) y ficheros especiales que el proceso abre para su uso propio.

Hay cinco llamadas que producen fd, serán creat, open, fcntl, pipe y dup.

CREAT

int creat(path,permisos) crea un fichero

char *path nombre del path

int permisos bits de permiso

Devuelve fd o −1 si ha habido error.

Se comporta de distinta forma dependiendo de si el fichero existe o no.

Si el fichero no existe, se crea un nuevo i−nodo y un link a él se pone en el directorio en donde se va a crear; euid y egid del proceso que llama deben tener permiso en este directorio, convirtiéndose en propietarios del fichero.

Si el fichero ya existe, no se necesita permiso de escritura en el directorio al que se linka, solamente permiso de ejecución (búsqueda), de todos modos euid y egid deben tener permiso de escritura en el fichero.

Un ejemplo es fd0=creat(temp,0666)

OPEN

Hay dos formas para esta llamada. La antigua forma es:

int open(path,flags) abrir el fichero

char *path; nombre del path

int flags; read,write o ambas

Devuelve fd o −1 si hay error. El fichero dado por path debe existir previamente.

Los flags son 0 para lectura, 1 para escritura y 2 para lectura/escritura. El fd devuelto se podrá usar luego para otras llamadas como lseek, que veremos más adelante; euid y egid deben tener permiso de lectura y/o escritura dependiendo de los flags que se tengan.

Una versión más moderna de esta llamada es la siguiente:

#include <fcntl.h>

int open(path,flags,permisos)

char *path;

int flags;

int permisos;

Devuelve fd o −1 si hay error. En lugar de usar números para los flags hay constantes simbólicas incluidas en usr/include/fcntl.h

WRITE

int write(fd,buf,nbytes) escritura en el fichero

int fd; descriptor de fichero

char *buf; dirección del buffer

unsigned nbytes; número de bytes a escribir

Devuelve el número de bytes escritos o −1 si se ha cometido algún fallo. Esta orden escribe nbytes apuntados en buf al fichero abierto representado por fd. Se puede usar para escribir en tuberías, pero esto se verá más adelante.

READ

int read(fd,buf,nbytes) se lee de fichero

int fd; descriptor de fichero

char *buf; dirección del buffer

unsigned nbytes; número de bytes a leer

Devuelve el número de bytes leído o 0 en caso de EOF o −1 si se produce error.

CLOSE

int close(fd) se cierra el fichero

int fd; descriptor de fichero

Devuelve 0 si se lleva a cabo correctamente o −1 si no es así. No hace falta usar esta llamada ya que el fichero se cierra automáticamente cuando termina el proceso.

LSEEK

long lseek(fd,offset,interp) mueve el puntero de fichero

int fd; descriptor de ficheros

long offset; offset en el fichero

int interp; interpretación del offset

Devuelve el valor del puntero de fichero o −1 si hay error. Sitúa el puntero de fichero para la próxima lectura o escritura.

<u>CAPÍTULO V.</u> Procesos

exec

fork

exit

wait

Llamadas para conocer Ids (get Ids)

Generación de Ids y proceso de arranque

Antes e empezar, debemos recordar que un programa es un conjunto de instrucciones y datos usados para inicializar la instrucción y el segmento de datos de un proceso, mientras que el entorno es la instrucción y los segmentos de datos del usuario y del sistema asociados a ella.

Exec reinicializa un proceso de un determinado programa; el programa cambia mientras el proceso se mantiene. Fork crea un nuevo proceso exactamente igual a otro existente, copiando la instrucción y los segmentos de datos del usuario y del sistema, el nuevo proceso no es inicializado desde un programa.

Exec es la única forma de que un programa se ejecute en UNIX, y fork es la única manera de crear procesos.

EXEC

El proceso 1, que ya existe, ejecuta un programa, exec reemplaza el proceso 1 con el nuevo programa. El pid del proceso 1 no cambia (el proceso pasa a ejecutar B en lugar de A). Hay un nuevo programa ejecutándose en el mismo contexto, el ppid no cambia, la instrucción exec sólo devuelve el control al llamador si hay error. Hay distintos tipos, vamos a ver los más importantes:

- **execl**

int execl(path,arg0,arg1,...,argn,0)

char *path;

char *arg0; primer argumento (nombre de fichero)

char *arg1;

.....

char *argn;

Devuelve −1 si hay error, el programa busca en path, siempre hay un 0 al final; fin de los argumentos: el entorno se pasa automáticamente.

- **execv**

int execv(path,argv)

char *path;

char *argv[] punteros a los argumentos

No busca en path, los argumentos son dados como un vector de punteros (tengo en mi programa un vector llamado argv cuyos elementos son punteros a memoria)

- **execle**

int execle(path,arg0,...,argn,0,envp)

char *path;

char *ârg0;

.....

char *argn;

char *envp[]; punteros a las variables de entorno

- **execlp**

int execlp(file,arg0,...,argn,0)

char *file nombre del fichero de programa

char *arg0;

.....

char *argn;

Busca en path (ls)

- **execvp**

int execvp(file,argv)

char *file;

char *argv[];

Busca en path.

FORK

int fork() se crea un nuevo proceso

Devuelve el pid del hijo al padre y un 0 al hijo si hay éxito o un −1 si hay error. Fork es esencialmente lo opuesto a exec, es decir, crea un nuevo proceso, pero no inicializado desde un nuevo programa, por lo tanto, la instrucción del nuevo proceso y los segmentos de datos del usuario y del sistema son copias exactas del viejo proceso. El valor de retorno es diferente en el padre y en el hijo, así se permite diferenciar acciones siguientes.

El hijo recibe un 0 de fork, el padre recibe el pid del hijo (−1 si hay error). La única causa de error es la falta de recursos en el sistema (demasiados procesos ejecutándose, falta de memoria...), podría pensarse que si faltan recursos, el padre va a esperar, pero no es así, se imprime un mensaje dando la causa del error.

Debemos tener en cuenta que el pid del padre y del hijo son distintos (obvio, ya que son procesos diferentes), el hijo consigue copias de los fd del padre. Si el hijo cambiara el puntero con lseek, el puntero del padre estaría en la nueva posición; el fd es distinto así si el hijo lo cierra, la copia del padre no cambia. También el tiempo de ejecución es diferente, cuando se crea el hijo, su tiempo de ejecución empieza de cero (obvio).

Aquí tenemos un ejemplo:

forktest()

{

int pid;

printf(Oscar es el mejor \n);

pid=fork();

printf(Devuelto %d\n,pid);

}

La salida es la siguiente:

Oscar es el mejor

Devuelto 0

Devuelto 93

Primero se imprime la frase, a continuación se crea un proceso hijo con el fork(), ahora se ejecuta el hijo con lo que se realiza la siguiente línea y nos escribe Devuelto 0, es decir, el proceso ha sido creado con éxito, con

lo que en la variable pid hemos almacenado un 0, que es lo que se nos da por pantalla, el proceso hijo ya ha terminado, por lo que volvemos a la línea printf(Devuelto %d\n,pid) que ahora corresponde al proceso padre y escribiremos en pantalla el valor que se ha almacenado en la variable pid, que es el pid correspondiente al proceso hijo, como se explicaba al principio de esta sección.

EXIT

void exit(estado) terminamos el proceso

int estado; estado de salida

No hay valor devuelto. La variable estado es un entero entre 0 y 255. Si la salida es 0, la terminación ha sido normal, si no, ha habido alguna anomalía. Cuando se llama a exit, se cierran todos los fd. Si un proceso hijo está vivo cuando se llama a exit, el ppid cambia a 1 (pid de INIT)

WAIT

int wait(statusp) se espera a un hijo

int *statusp; estado de salida

Si hay algún proceso hijo, wait duerme hasta que uno de ellos termina. No se puede especificar el hijo al que se espera.

Se devuelve el pid del hijo (o −1 si hay fallo) y se almacena su código de estado en el entero señalado por statusp, a menos que el argumento sea 0, en cuyo caso no se almacena ningún estado. Wait también nos indicará si una señal de captura es mandada o si un hijo para en modo de traza, pero esto se verá más adelante.

Debemos fijarnos en que no hay nietos si no hay hijos, al morir un hijo, sus hijos son adoptados por el proceso INIT, no por su abuelo, así un proceso no recibirá nada de wait cuando se produzca la terminación de un hijo, el padre del nieto está todavía vivo y tiene el privilegio de esperar.

Un proceso podría terminar en un momento en el que el padre no estaba esperándole, el Kernel se debe asegurar de que todo proceso es esperado, ese proceso no esperado se convierte en zombie, su instrucción y segmento de datos de usuario y sistema desaparecen, pero todavía ocupa una posición de la tabla de procesos del kernel. Cuando es esperado, finalmente desaparece.

Si no hay hijos (porque nunca los ha habido o porque un wait anterior ya devolvió sus Ids), la instrucción wait devuelve −1.

Hay tres formas de que un proceso se acabe: con una llamada exit, con una señal fatal o caída del sistema. El código de estado devuelto nos indica cual de las dos primeras se produce, el tercer caso se produce cuando el kernel y el proceso padre mueren a la vez.

LLAMADAS PARA CONOCER Ids (GET IDs)

Hay distintos tipos dependiendo de qué ID queramos conocer:

− int getuid()

− int getgid()

− int geteuid()

− int getegid()

− int getpid()

− int getpgrp() process−group−id

− int getppid()

Todos ellos devuelven un ID. El pid es frecuentemente usado para crear nombres de fichero temporales ya que como el ID es único, nos aseguramos de que ese fichero tiene nombre único.

GENERACIÓN DE IDs Y PROCESO DE ARRANQUE

int setuid(uid) generación de uid

int uid; uid

int setgid(gid) generación de gid

int gid; gid

Devuelve 0 si ha habido éxito y −1 si no lo ha habido para ambos procesos.

Si el superuser es el que realiza la llamada, se crean los uid y gid reales y efectivos. Estas llamadas permiten al superusuario convertirse en otro usuario.

Esta llamada es usada por login en el proceso de arranque. Se introduce un login name y un password, login lo verifica consultando /etc/passwd, si son válidos, ejecuta setuid y setgid para crear los uid y gid efectivos y reales de acuerdo con los números en /etc/passwd; después login coloca como directorio actual login y usa exec para ejecutar el shell asignado al usuario.

CAPÍTULO VI. Tuberías (pipes)

Creación de tuberías

Duplicado de descriptores de fichero

Aquí solamente se va a ver un par de instrucciones que son útiles para el manejo de tuberías (unnamed pipes, las named pipes no son vistas en el curso), hay ejemplos en los archivos de merlin.

CREACIÓN DE TUBERÍAS

int pipe(fd) crea tubería

int pfd[2] descriptores de fichero

Esta llamada va a devolver un 0 si se realiza correctamente o un −1 si no lo hace. La tubería es un canal representado por dos fd que son devueltos al vector pfd, escribiendo en pfd[1] se pone un dato en la tubería, y se leen de ella por medio de pfd[0]. Las operaciones que se pueden realizar sobre las tuberías y que nosotros usaremos (hay muchas más) serán las siguientes:

- **write:** Los datos se escriben en el orden de llegada. Cuando se llena la tubería no se puede escribir hasta que algún dato se haya leído. Una tubería tiene un tamaño variable, pero el mínimo es de 4096 bytes.
- **read:** Los datos se leen por orden de llegada, una vez leídos, los datos se eliminan de la tubería. Si se pretende leer con la tubería vacía, se deberá esperar hasta que se introduzca algún dato, a menos que fd esté cerrado y se devuelva un 0.
- **close:** Libera el fd para un nuevo uso y cuando el fd está cerrado actúa como un EOF para el lector. Si un fd está cerrado, una escritura dará error.

La conexión entre dos procesos se llevará a cabo siguiendo estos pasos (para mejor comprensión, ver ejercicios resueltos en los apuntes):

- Hacer tubería
- Fork para crear un proceso hijo
- Cerrar el extremo de escritura para el hijo... (ver ejemplos)
- Ejecutar el programa hijo desde el proceso hijo
- Cerrar el extremo de lectura en el proceso padre
- Si un segundo hijo va a escribir en la tubería, deberemos crearlo, hacer los preparativos necesarios y ejecutar el programa. Si el padre va a escribir, seguir los pasos previos.

DUP

int dup(fd) duplicación del descriptor de fichero

int fd; fd existente

Se devuelve el nuevo fd o −1 en caso de error. Esta llamada duplica el fd ya existente, dando un nuevo fd abierto al mismo fichero o tubería. Los dos tienen el mismo puntero de fichero. La llamada falla si el argumento es malo (no abierto), o bien cuando los 20 fd ya están abiertos.

CAPÍTULO VIII. Señales

Principales señales

Llamadas al sistema asociadas

kill

pause

alarm

PRINCIPALES SEÑALES

En la mayoría de las versiones de UNIX hay 19 tipos de señales. Para todas excepto para las dos últimas, la acción que desarrollan por defecto es la terminación del proceso. Veremos las más importantes (el número de la señal va entre paréntesis, los nombres de las señales se dan en /usr/include/signal.h). Se debe usar siempre el nombre en lugar del número:

- **SIGINT (2):** Señal de interrupción. Es mandada a todos los procesos asociados con un terminal de control cuando la tecla es presionada. Esta acción de la tecla de interrupción puede ser suprimida o podemos cambiar la tecla de interrupción mediante la llamada ioctl. Debemos notar que suprimir la

tecla de interrupción es completamente diferente a ignorar la señal. Se produce cuando pulsamos simultáneamente CTRL C.

- **SIGQUIT (3):** Similar a SIGINT, pero mandada cuando se pulsa CTRL Y. Además se manda un core dump.
- **SIGKILL (9):** Señal de kill. La única forma segura de matar un proceso ya que esta señal es fatal (no puede ser ignorada o capturada). Se usa sólo en emergencias.
- **SIGPIPE (13):** Se produce cuando intentamos escribir en una pipe que no está abierta para lectura. Probablemente es debido a que el lector era un proceso padre que ya ha muerto.
- **SIGALRM (14):** Alarma de reloj. Se fija con la llamada alarm().
- **SIGTERM (15):** Es la señal de software termination. Es la señal estándar de terminación. Es la señal por defecto mandada por el comando kill. La acción conocida como shutdown produce que INIT mande una señal SIGTERM para que finalicen todos los hijos.
- **SIGUSR1 (16):** Es una señal user defined signal 2, definida por el usuario. Puede ser usada por programas de aplicación para la comunicación entre procesos. No es recomendable usarla y se usa raras veces.
- **SIGUSR2 (17):** Es la señal user defined signal 2, funciona de manera similar a la anterior.

LLAMADAS AL SISTEMA ASOCIADAS

#include <signal.h>

int (*signal(sig,fcn))() especificación de la señal

int sig; número de la señal

int (*fcn)(); acción en la recepción

Devuelve la acción previa o un −1 si hay fallo. La acción en recepción puede ser una de estas tres:

- **SIG_DFL:** Establece la acción por defecto de la señal.
- **SIG_IGN:** Se ignora la señal, el proceso es inmune a ella (no es válido para SIGKILL)
- **Puntero_a_funcion:** Permite capturar la señal. Todas señales menos SIGKILL pueden ser capturadas.

La acción de un padre frente a una señal es heredada por el hijo.

KILL

int kill(pid,sig) señal mandada

int pid; pid del proceso que recibe la señal

int sig; número de la señal mandada

Devuelve 0 en caso de que se realice correctamente o −1 en caso contrario. Si pides 0, la señal se envía a todos procesos del mismo grupo del que manda la señal. Si pid es −1, la señal se envía a todos los procesos cuyo ruid es igual al euid del proceso que manda la señal. Si el superuser ejecuta kill con pid igual a −1, todos procesos menos los de pid igual a 0 (swapper) ó 1 (INIT) mueren. Si el pid es negativo, pero distinto de −1, la señal se manda a todos los procesos cuyo pgid es igual al valor absoluto del pid.

PAUSE

void pause() esperamos una señal

La señal a la que más frecuentemente se espera es la alarma

ALARM

unsigned alarm(secs) establecemos la alarma del reloj

unsigned secs; número de segundos

Devuelve el número de segundos que quedaban anteriormente.