

6. Procesos

6.1 *Definición: proceso y programa.*

Del mismo modo que las aplicaciones de usuario conducen a la realización de determinadas actividades, también el sistema operativo desempeña, como se ha visto, una serie de funciones, entre las que destaca la gestión de recursos, el ofrecimiento de una máquina amigable al usuario y la obtención de un rendimiento óptimo de la máquina. El desempeño de estas funciones – tanto del sistema operativo como de usuario – en el momento en que son requeridas, conlleva la ejecución de uno o más fragmentos de código.

Facultad de Ingeniería

6.1.1 DEFINICIÓN DE PROCESO. ENTORNO DE UN PROCESO.

El término proceso se define como la secuencia de acciones llevadas a cabo a través de la ejecución de una serie de instrucciones o fragmentos de código, con el fin de proporcionar una función.

Sin embargo, la ejecución de código implica la existencia de un entorno en el cual se pueda dar, y que se define como el *conjunto de valores que deben tomar los parámetros que definen la situación del sistema en cada momento*. El entorno es por tanto dinámico.

Para ilustrar todos estos conceptos, supóngase por ejemplo que se desea elaborar un postre siguiendo una receta de Karlos Arguiñano.

- La función que se desea lograr mediante las actividades a realizar es *obtener un postre*.
- El conjunto de acciones a desempeñar, es decir, el código a ejecutar, lo constituye las *instrucciones de la receta*.
- Además, son necesarios una serie de recursos, como *el horno, la cocina, etc.*
- Estos recursos deberán tener un estado determinado, es decir, es necesario lograr un entorno adecuado, haciendo que *la temperatura del horno tenga determinado valor, etc.*
- El proceso consistiría en la realización, propiamente dicha, del postre citado, en ese entorno y con dichos recursos.

Programas, procesos e hilos

Es importante desligar el concepto de proceso del concepto de programa o conjunto de instrucciones, puesto que un mismo proceso puede implicar a varios programas y viceversa, un programa puede intervenir en un número variable de procesos. Por tanto, no es posible identificar un proceso a partir de un programa en ejecución.

Un examen más detallado del concepto de *proceso* permite contemplarlo como una secuencia de acciones que pueden ser interrumpidas, en un orden y forma impredecible, por lo que a priori se desconoce cuándo se ejecutarán – si se ejecutan – (indeterminismo).

Se podría establecer una analogía con el trabajo de un secretario/a en una oficina. Esta persona, que sería el procesador, debe realizar una serie de funciones y en consecuencia, ejecutar los procesos correspondientes (mecanografiar cartas, tomar nota de pedidos, anotar dictados, etc.). Esta persona puede recibir un número aleatorio de llamadas telefónicas al azar; desconoce el tiempo que tardará en atender cada llamada y sus repercusiones, si bien, es consciente de que deberá retomar las tareas que se interrumpan puesto que, en principio, deben ser finalizadas.

En Linux, como en otros sistemas operativos, la unidad de trabajo en el sistema es el proceso. Durante el tiempo que dure la ejecución de un proceso, éste usará diferentes recursos del sistema: la UCP para la ejecución de sus instrucciones, la memoria del sistema y el sistema de archivos, para el mantenimiento y almacenamiento de datos, periféricos del sistema,... Es decir, un proceso es la unidad de asignación de recursos; los recursos del sistema se asignan a procesos que los necesitan. Además, tradicionalmente, un proceso es la unidad planificable, es decir, la unidad más pequeña que puede ser sometida a ejecución a un procesador.

Sin embargo, en los sistemas operativos modernos, se tiende a separar el concepto de unidad de asignación del de unidad de planificación, de manera que el primero de ellos hace referencia a un proceso¹⁹, mientras que la unidad de planificación es el *hilo* o procesos ligero²⁰. Así, un proceso puede constar de varios hilos (Podría considerarse que una vez que se ha creado un proceso a partir de cierto código, éste se subdivide en varios procesos ligeros o hilos.

¹⁹ También conocido como proceso “pesado” o tarea. En inglés: *task*, *process*,...

²⁰ En inglés, *thread*.

Tanto en la ejecución de procesos como en la ejecución de hilos se ven involucrados elementos como registros máquina, pilas que contienen datos temporales, etc. Consecuentemente, en un sistema que soporte multiprogramación cada proceso – pesado o ligero – debe ser considerado como una entidad independiente, con sus propios derechos y responsabilidades, de manera que si termina su ejecución anormalmente, no debería afectar o influir a otros procesos concurrentes o posteriores. Cada proceso se ejecuta en su propio espacio de direcciones y no puede interactuar con otros procesos si no a través de los mecanismos de comunicación manejados por el núcleo.

6.1.2 CONCURRENCIA.

Linux es un sistema operativo multiusuario y multitarea, es decir, es posible que varias personas estén usando el mismo computador simultáneamente (en contraposición con sistemas operativos como MS-DOS) y es capaz de llevar a cabo de forma concurrente varias tareas (simultáneamente y compitiendo por los recursos).

En aquellos sistemas en los que se dispone de un procesador para cada proceso la concurrencia es real, sin embargo, si el número de procesos es superior al de procesadores es necesario conmutarlos entre los primeros, dándose así una concurrencia aparente.

En cualquier caso, al coexistir varios procesos, el sistema operativo debe tener constancia en todo momento de las acciones que lleva a cabo cada uno, así como de los recursos del sistema que utilizan, para evitar así posibles conflictos derivados del hecho de la compartición de los recursos disponibles en el sistema.

6.1.3 REPRESENTACIÓN DE LOS PROCESOS

Internamente, cada proceso suele representarse mediante un *PCB* o Bloque de Control de Proceso, que puede definirse como la zona de memoria en el sistema con toda la información relevante de un proceso.

Un PCB contiene generalmente el estado actual del proceso (se está ejecutando, está esperando por un recurso, etc.), información de re arranque (información contenida en el entorno volátil del sistema: registros máquina, registros relativos a la memoria ocupada, etc.), información relativa al estado de operaciones de E/S (peticiones pendientes, dispositivos asignados, ficheros abiertos, etc.) e información de planificación (punteros a colas de planificación, prioridad, etc.).

6.2 Los procesos en el sistema operativo Linux

6.2.1 FUNCIONAMIENTO DE LA MULTITAREA

Un objetivo de un sistema operativo multiusuario es hacer creer a cada usuario que es el único que está siendo atendido, que tenga la impresión de que dispone del 100% de los recursos – tanto físicos como lógicos – del sistema. Esto es posible gracias al tiempo de reflexión, o tiempo que tarda cada usuario en elaborar la siguiente petición. Este tiempo puede ser empleado en atender a cientos de peticiones de otros usuarios – en una fracción de segundo un procesador es capaz de ejecutar cientos o miles de instrucciones.

En el momento en que existe más de una tarea “viva” en el sistema es preciso tener información actualizada de cada una de ellas, ya que compartirán el tiempo de UCP, la capacidad de almacenamiento, los dispositivos de E/S, datos y variables, etc.

Para sacar mayor rendimiento del hardware, Linux combina técnicas de multiprogramación y de tiempo compartido, lo cual favorece los trabajos interactivos.

Estructuras de datos

Linux guarda por cada proceso una estructura de datos denominada *task_struct*, en la que se almacena toda la información relacionada con el proceso: aspectos de planificación, identificadores, relación con otros procesos, memoria utilizada por el proceso, archivos abiertos, etc. Es decir, el PCB o *Bloque de Control de un Proceso*, se implementa en Linux con la estructura *task_struct*.

Todas las *task_struct* conforman la “Tabla de Control de Procesos” o “Tabla de Control de Tareas”, y cada *task_struct* individual de la “Tabla de Control de Tareas”, denominada *task* está apuntada desde una matriz de punteros.

La cantidad de entradas no nulas de esta matriz, determina el grado de multiprogramación del sistema, hasta un máximo definido por la constante *NR_TASKS*, que toma por defecto el valor 512.

Todas estas estructuras de datos se encuentran declaradas en el archivo de cabecera “*/include/linux/sched.h*”.

Estados de los procesos.

Los procesos van pasando por una serie de estados discretos a lo largo de su existencia. Por ejemplo, si desde un proceso se solicita que se pulse una tecla, éste debe quedar esperando hasta que dicha tecla sea pulsada. En este caso, no tiene

sentido que el proceso, mientras espera, consume tiempo de UCP; lo lógico es bloquearlo y reanudarlo cuando se detecte una interrupción del teclado que indique la llegada de la información deseada.

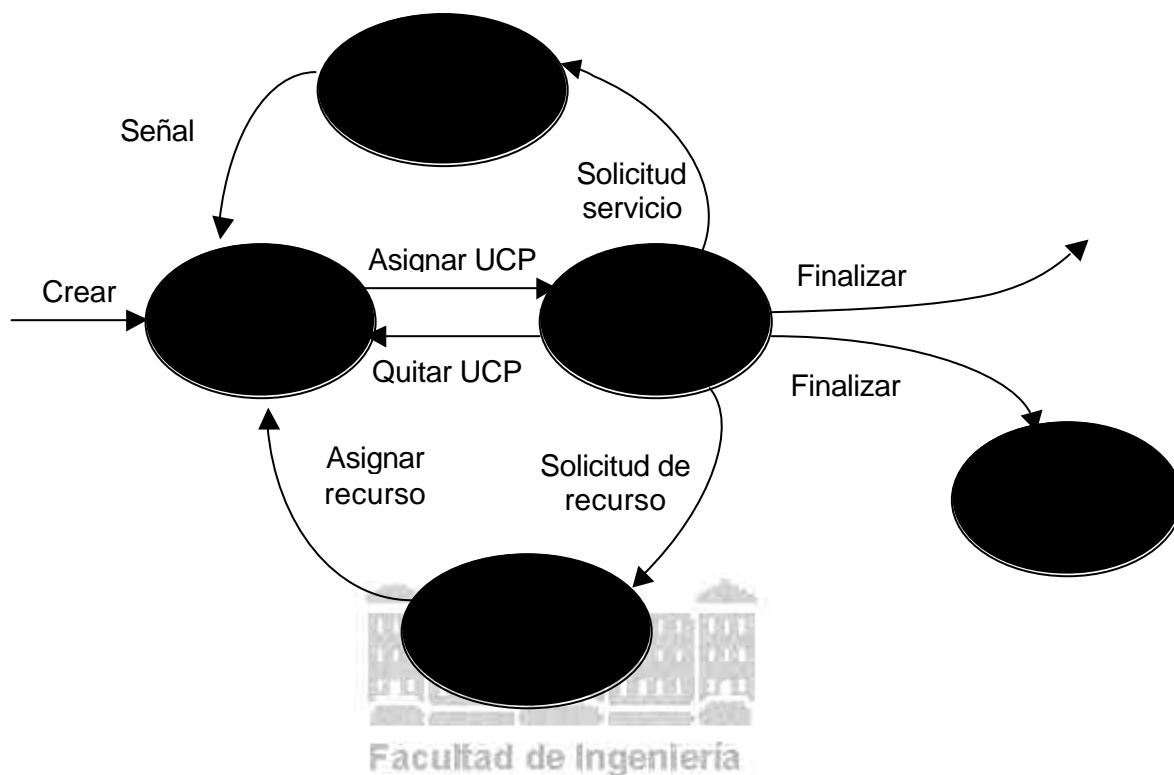


Figura 7: Estados de los procesos en Linux

El estado en que se encuentra un proceso en cada momento se almacena en el campo *state* de la estructura *task_struct*. Este campo puede tomar los siguientes valores:

- *task_running*: Indica que el proceso en cuestión se está ejecutando o listo para ejecutarse. En este segundo caso, el proceso dispone de todos los recursos necesarios excepto el procesador.
- *task_interrumpible*: el proceso está suspendido a la espera de alguna señal para pasar a listo para ejecutarse. Generalmente se debe a que el proceso está esperando a que otro proceso del sistema le preste algún servicio solicitado.
- *task_uninterrumpible*: el proceso está bloqueado esperando a que se le conceda algún recurso hardware que ha solicitado (cuando una señal no es capaz de “despertarlo”).

- *task_zombie*: el proceso ha finalizado pero aún no se ha eliminado todo rastro del mismo del sistema; por ejemplo su *task_struct* de la “Tabla de Control de Tareas”.

Identificadores de un proceso

Linux asigna a cada proceso un identificador exclusivo llamado identificador de proceso (*PID* o *Process ID*) y otro que identifica a su proceso padre denominado *PPID* (*Parent Process Identifier*).

Un *PID* es simplemente un número entero que identifica de manera unívoca a cada uno de los procesos existentes en el sistema. El comando *ps* permite la visualización en pantalla de los distintos procesos del sistema, de su identificador.

Aparte de estos dos identificadores existen otros adicionales que permiten determinar quién ha lanzado el proceso (*uid* o *user identifier*) y a qué grupo pertenece (*gid* o *group identifier*), y por tanto los derechos del proceso respecto de los diferentes recursos del sistema.

Para cada recurso se mantiene una lista de control de acceso que determina los derechos que tiene el propietario del recurso, los del grupo al que pertenece el propietario, y los derechos por defecto. Por ello, cada recurso del sistema tiene asociados también dos identificadores que permiten determinar cuál es su propietario y el grupo del mismo. Comparando los identificadores *uid* y *gid* con los de un recurso al que se pretende acceder y en función de la lista de control de acceso (ACL) del propio recurso Linux determina si un proceso tiene derecho de acceso al mismo o no.

PARES DE IDENTIFICADORES

Cada proceso tiene cuatro pares de identificadores de usuario y grupo:

- *uid* y *gid*: son los identificadores de usuario y grupo del propietario del proceso y pueden “verse” accediendo al archivo */etc/passwd*, que contiene información relativa a todos los usuarios del sistema (campos 3 y 4 de cada entrada).
- *euid* y *egid* (*uid* y *gid* efectivos): son los identificadores correspondientes al propietario del programa. Éstos pueden ser asignados a un proceso al ser lanzado, en vez de los correspondientes al usuario que lo lanza, al ser lanzado.
- *fuid* y *fgid* (*uid* y *gid* de sistema de archivos): son los identificadores utilizados al trabajar con un sistema de archivos en red, como el NFS (*Network File System*).

- *suid y sgid (uid y gid de salvado): se emplean para almacenar los uid y gid originales cuando un proceso trabaja con identificadores efectivos y pide modificar sus identificadores a través de una llamada al sistema.*

Planificación

El elemento del sistema operativo encargado de repartir el tiempo de la UCP, es decir, de determinar a qué proceso debe asignarse la UCP es el planificador.

Linux supervisa las listas (colas implementadas en forma de listas enlazadas) de las tareas que están pendientes. Estas colas de tareas permiten separar tareas de usuario, tareas de sistema operativo, etc. Linux reparte el tiempo de uso del procesador por las listas de manera rotatoria de manera que cada tarea recibe más o menos tiempo según su prioridad hasta que termine (las tareas de una misma cola reciben el mismo tiempo de uso o *quantum*, pero éste varía entre las distintas colas).

Esta manera de planificar el tiempo del procesador se denomina *Round-Robin* o asignación rotatoria, e implica la repartición de los recursos del sistema entre todas las tareas, en base a un esquema de reparto de tiempos (*time slicing*). Este método es simple y equitativo que evita problemas de inanición. Si un proceso no libera voluntariamente el procesador al acabar su *quantum* se le requisa para ser asignado a otro proceso activo, evitando así que un proceso se adueñe del procesador y bloquee el sistema completo.

Cuando a un proceso se le requisa la UCP, Linux almacena en su estructura *task_struct* el contenido de su entorno volátil, y vuelca sobre él, desde su respectiva estructura *task_struct*, lo correspondiente a la tarea o proceso al que el planificador conceda el control del procesador.

En esta estructura, también hay un campo denominado *policy*, que permite distinguir si una tarea es “ordinaria” o de “de tiempo real”; es decir, si tiene o no requisitos temporales de procesamiento. Tanto entre los procesos “ordinarios” como entre los de “tiempo real” existen distintos niveles de prioridad, pero estos últimos, los procesos con restricciones temporales, son siempre más prioritarios que los otros. En el caso de procesos ordinarios, Linux utiliza el campo *priority* para almacenar la prioridad de cada proceso y en el caso de los procesos de tiempo real, el campo *rt_priority* (ambos campos pueden ser modificados mediante llamadas al sistema).

Dentro de la Tabla de Control de Tareas, las estructuras *task_struct* de todos los procesos en estado *listo* se unen mediante punteros, formando listas de procesos en estado listo, una por cada tipo de proceso, ordenadas por prioridades.

Tipo de proceso	Descripción
Ordinario	Proceso que no tiene restricciones temporales de procesamiento.
De tiempo real	Proceso con restricciones temporales de procesamiento.
Interactivo	Proceso que permite una interacción directa con el usuario que lo pone en marcha. Puede ejecutarse en primer o segundo plano.
Por lotes	Conjunto de procesos que ha sido planificado para ser ejecutados en cierto orden basándose en un guión.
Daemon	Proceso que suele iniciarse al arrancar el sistema y que no suele finalizar hasta que éste es apagado. Suele atender a ciertas funciones solicitadas al sistema operativo como <code>LPD</code> , en el momento de imprimir.

Tabla 8: Tipos de procesos existentes en Linux

SISTEMAS MULTIPROCESADOR

Linux también tiene soporte para sistemas con varios procesadores en configuración simétrica (*Symmetric Multi-Processing – SMP*), pudiendo repartir los procesos entre ellos con el objeto de aumentar el rendimiento total del sistema. En este tipo de sistemas pueden existir tantos procesos en ejecución como procesadores tenga el sistema. Esto es posible, gracias a la información que, relacionada con cada procesador específico se mantiene en cada *task_struct*.

6.3 Manipulación de ficheros

En Linux, los procesos acceden a los archivos a través de *descriptores de fichero* que son números enteros no negativos, índices de un array de punteros a descriptores de fichero abiertos. Cada una de las posiciones de este array hace referencia mediante un puntero a una *descripción de fichero abierto*, que no es sino un conjunto de informaciones que permiten el acceso a un fichero, en el modo en que fue abierto (lectura, escritura o lectura/escritura), la posición actual dentro del mismo, su ubicación, etc.

Además, varios descriptores de fichero, bien sean del mismo proceso o no, pueden hacer referencia (pueden apuntar) a una misma *descripción de fichero abierto*, haciendo así posible la compartición de archivos a varios procesos del sistema.

LA LLAMADA OPEN.

Para obtener un descriptor de fichero para un archivo ya existente, se puede utilizar la llamada al sistema *open*:

```
# include <sys/types.h>

# include <sys/stat.h>

# include <fcntl.h>


int open (const char *path, int flags);

int open (const char *path, int flags, mode_t mode);
```

Como puede verse existen dos definiciones distintas de la función *open()* que dan acceso a la llamada al sistema correspondiente. La más habitual es la primera, aunque la segunda se utiliza cuando es posible que el fichero a acceder no exista y sea necesario crearlo.

El primer parámetro (*path*) debe ser un apuntador a una cadena que contenga la trayectoria – deseablemente absoluta – del fichero a abrir. El segundo de los parámetros (*flags*) permite especificar el modo de apertura deseado, y en principio puede tomar los valores siguientes:

Valor	Significado
O_RDONLY	Abrir fichero únicamente en modo lectura.
O_WRONLY	Abrir fichero únicamente en modo escritura.
O_RDWR	Abrir fichero en modo lectura / escritura.

Tabla 9: Posibles valores del parámetro *flags* en la llamada *open*.

Otros valores son los indicados en la tabla 10 y pueden combinarse con los anteriores mediante el operador de bit OR ("|").

Valor	Significado
O_CREAT	Crear el fichero si éste no existe.
O_EXCL	Hacer que la <i>open</i> falle si se especifica la opción O_CREAT y el fichero existe.
O_TRUNC	Hacer que el fichero pase a tener longitud 0 al abrir.
O_APPEND	Hacer que las posibles <i>write</i> se produzcan al final

del fichero.

Tabla 10: Valores opcionales del parámetro *flags* en la llamada *open*.

Tanto los valores anteriormente citados como los opcionales se encuentran especificados en el fichero `fcntl.h`. Siempre que se utilice la *open* con el propósito de crear un fichero, es posible hacer uso de un tercer parámetro (*mode*), para especificar los bits de permisos de acceso para el propietario, su grupo y el resto de usuarios del sistema. El fichero `sys/stat.h` contiene los posibles valores de *mode*.

Valor	Significado
<code>S_IRUSR</code>	Activar bit de lectura para el propietario del fichero.
<code>S_IWUSR</code>	Activar bit de escritura para el propietario del fichero.
<code>S_IXUSR</code>	Activar bit de ejecución para el propietario del fichero.
<code>S_IRGRP</code>	Activar bit de permiso de lectura para el grupo del propietario del fichero.
<code>S_IWGRP</code>	Activar bit de permiso de escritura para el grupo del propietario del fichero.
<code>S_IXGRP</code>	Activar bit de permiso de ejecución para el grupo del propietario del fichero.
<code>S_IROTH</code>	Activar bit de permiso de lectura para el resto de usuarios.
<code>S_IWOTH</code>	Activar bit de permiso de escritura para el resto de usuarios.
<code>S_IXOTH</code>	Activar bit de permiso de ejecución para el resto de usuarios.
<code>S_ISUID</code>	Activar bit <i>set_uid</i> .
<code>S_ISGID</code>	Activar bit <i>seg_gid</i> .
<code>S_IRWXU</code>	<code>S_IRUSR</code> <code>S_IWUSR</code> <code>S_IXUSR</code>
<code>S_IRWXG</code>	<code>S_IRGRP</code> <code>S_IWGRP</code> <code>S_IXGRP</code>
<code>S_IRWXO</code>	<code>S_IROTH</code> <code>S_IWOTH</code> <code>S_IXOTH</code>

Tabla 11: Valores posibles del parámetro *mode* en la llamada *open*.

Siempre que un proceso realiza una *open* se recorre su array de descriptores de fichero buscando la primera posición libre, de manera que *open* siempre devuelve el valor entero positivo más bajo posible.

El resultado de la ejecución de una *open()* es “-1” si se produce algún error durante su ejecución. En caso contrario, devuelve el descriptor de fichero a utilizar en las operaciones subsecuentes sobre el fichero.

LA LLAMADA ACCESS

Permite comprobar si un determinado proceso tiene acceso a un fichero en particular, y suele utilizarse antes de ejecutar una *open*. Su formato es el siguiente:

```
# include <unistd.h>

int access (char *pathname, int mode);
```

Siendo *pathname* el nombre del fichero cuyo posible acceso se desea comprobar, y *mode* el modo de acceso (ver tabla 12).

Valor	Significado
R_OK	Comprobar si el proceso que efectúa la llamada tiene permiso de lectura sobre el fichero indicado.
W_OK	Comprobar si el proceso que efectúa la llamada tiene permiso de escritura sobre el fichero indicado.
X_OK	Comprobar si el proceso que efectúa la llamada tiene permiso de ejecución sobre el fichero indicado.
F_OK	Comprobar si el fichero indicado existe.

Tabla 12: Valores posibles del parámetro *mode* en la llamada *access*.

LA LLAMADA CREAT

Se trata de una llamada que se incluye para mantener la compatibilidad de los sistemas Linux con los sistemas UNIX tradicionales. Esta llamada presenta la estructura siguiente:

```
# include <sys/types.h>

# include <sys/stat.h>
```

```
# include <fcntl.h>
```

```
int creat (const char *path, mode t_mode);
```

En las primeras versiones de UNIX la *open* se implementaba exclusivamente con el formato de dos parámetros, de manera que no tenía la capacidad de crear ficheros que no existían. En estos casos, la creación se llevaba a cabo mediante esta llamada. Como consecuencia, las dos líneas siguientes son equivalentes.

```
fd = creat(file, mode);
```

```
fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

LA LLAMADA READ

Una vez se dispone de un descriptor de fichero asociado a un fichero que está abierto, y siempre que se hayan especificado las opciones *O_RDONLY* o *O_RDWR* al ejecutar la llamada *open*, es posible leer *bytes* del fichero mediante la llamada *read*, cuya sintaxis es la siguiente:

```
# include <sys/types.h>
```

```
# include <unistd.h>
```



```
int read (int fd, void *buf, size_t nbytes);
```

Siendo *fd* el descriptor de fichero que hace referencia al fichero del cual se desea leer, *buf* un puntero a un bloque de memoria en el que se depositarán los bytes leídos, y *nbytes* es el número de bytes a leer o copiar desde el fichero al buffer *buf*.

Esta función devuelve el número de bytes copiados, pero si se intenta leer de un fichero cuya posición actual es la de *fin-de-fichero* devuelve un 0, y si se produce cualquier error durante la lectura, -1.

LA LLAMADA WRITE

Permite la escritura de datos en un fichero. Su formato es el siguiente:

```
# include <sys/types.h>
```

```
# include <unistd.h>
```

```
int write (int fd, void *buf, size_t nbytes);
```

Su efecto es la copia de *nbytes* bytes desde el bloque de memoria apuntado por *buf* al fichero al que se refiere el descriptor *fd*. Por defecto, esta copia se realiza a partir de la posición actual para el fichero. No obstante, si el fichero se abrió con el flag `O_APPEND`, la escritura tendrá lugar al final del fichero, de manera que no se sobrescribirá ningún dato ya existente.

Esta función devuelve el valor `-1` si se produce algún error durante su ejecución. En caso contrario devuelve el número de bytes que se han escrito en el fichero.

LA LLAMADA DUP

Su prototipo es:

```
#include <unistd.h>

int dup (int fd);
```

Esta llamada toma como parámetro un descriptor de fichero y devuelve otro descriptor de fichero que es un duplicado del primero, es decir, genera otro descriptor de fichero asociados al mismo fichero, y en el mismo modo de apertura. Al igual que ocurre con las funciones `creat` y `open`, el descriptor creado se ubicará en la posición más baja del array de descriptores de fichero que se encuentre libre.

La utilidad principal de esta llamada, es que si se encontraran libres los descriptores 0,1 ó 2, correspondiente a la entrada estándar, salida estándar y salida de error respectivamente estuvieran libres, el descriptor devuelto se ubicaría en estas posiciones, logrando así un redireccionamiento del valor afectado.

LA LLAMADA CLOSE

Cada vez que se asigna un descriptor de ficheros e incrementa en uno el contador de descriptores del fichero correspondiente, de forma que cada fichero *sabe* cuántos descriptores tiene asociados.

La invocación de la llamada *close* provoca un decremento en el citado contador, que al llegar a 0 provocaría un cierre del fichero. De esta manera, se permite la reutilización de descriptores de fichero. Su formato es el siguiente:

```
# include <unistd.h>

int close (int fd);
```

Esta función devuelve un 0 en caso de éxito y un -1 en caso de error, si bien, el único error posible se debería a que el parámetro pasado no es un descriptor de fichero válido.

Por otro lado, se recomienda a todo programador que cierre siempre los descriptors de fichero a medida que deja de utilizarlos, o al menos al final del programa, de manera explícita puesto que no todos los sistemas los cierran implícita y correctamente.

ESTRUCTURAS DE DATOS RELACIONADAS CON LOS ARCHIVOS

Dentro de la estructura *task_struct* de cada proceso se mantiene información relacionada con el sistema de archivos y con los archivos que ha abierto el proceso (campos *fs* y *files*).

El primero de ellos es un puntero a una estructura denominada *fs_struct*, que apunta a los i-nodos del directorio raíz y del directorio actual de trabajo del proceso correspondiente.

files es un puntero a una tabla denominada *file_struct* que contiene información relativa a los ficheros que el proceso correspondiente tiene abiertos en cada instante.

El array de punteros o descriptors de fichero de cada proceso, denominado *fd*, contiene índices o números de posición dentro de la estructura *file_struct*. Tres posiciones de ese array que están reservadas:

Cada *fd[x]* apunta a una estructura de tipo *file* que contiene información relacionada con el archivo cuyo descriptor se encuentra dicha posición o índice. Para todos los procesos, los tres primeros elementos de este array están reservados, y son:

- *fd[0]* hace referencia al archivo estándar de entrada
- *fd[1]* hace referencia al archivo estándar de salida
- *fd[2]* hace referencia al archivo estándar de error

6.4 Jerarquía de procesos: creación y eliminación

Como se ha ido citando a lo largo de los capítulos anteriores, siempre que Linux recibe instrucciones para ejecutar un programa, o bien recibe un comando, está recibiendo indicación explícita de que cree un nuevo proceso. Linux lleva a cabo esta tarea creando una copia exacta del proceso que efectúa la petición, es decir, solicitando al núcleo una bifurcación (la ejecución de la llamada *fork*).

6.4.1 BIFURCACIÓN DE UN PROCESO

Bifurcar un proceso ya existente consiste en duplicar absolutamente toda la información relativa al mismo: el código asociado, el espacio asignado de memoria – mismo tamaño y distribución –, su entorno (incluido el valor del contador de programa), los enlaces a los archivos que pudiera tener abiertos, etc. De hecho, la única diferencia inicial es que el nuevo proceso – proceso hijo – recibe un nuevo identificador o *PID*.

En consecuencia, con la creación de procesos se establece una jerarquía de procesos en la que la raíz es el proceso a partir del cual se crean todos los demás: el proceso *init*, cuyo *PID* es el número 1.

El proceso *init* es el único, de entre todos los que ejecuta el núcleo de Linux, con el que el usuario tiene algún contacto y es el padre de todos los procesos *shell* que se creen al establecer una sesión en el sistema (éstos procesos *shell* serán a su vez padres de los procesos de usuario).

6.5 Procesos en segundo plano

Un proceso en primer plano es aquel con el que el usuario puede interactuar y solamente puede existir un proceso en primer plano. La multitarea puede lograrse de manera explícita, puesto que el *shell* permite iniciar un proceso antes de que haya finalizado otro. En este momento, el primer proceso se pondría en segundo plano.

Esto se logra colocando al final de la línea de comando el símbolo "&", de manera que toda línea de comandos que acabe en este símbolo, al ser sometida a ejecución provocará que se muestre en pantalla el identificativo de proceso (*pid*) correspondiente, e inmediatamente después el *prompt* del sistema. Esto indica que el *shell* y el proceso se están ejecutando concurrentemente (normalmente, el *shell* suspende su ejecución hasta que finalice la del comando introducido); si bien, solamente el proceso que está en primer plano – el *shell* – tiene acceso a la entrada y salida estándares. Por ello, cuando el proceso en segundo plano finaliza no se muestra ninguna salida por pantalla.

Si un proceso en segundo plano fuera demasiado largo (o incluso infinito) perduraría en el sistema hasta que desapareciese el *shell*, es decir, hasta que el usuario que inició la sesión dejase de estar conectado al sistema, ya que todos los procesos resultantes de la ejecución de una línea de comandos, incluidos los procesos en segundo plano, son hijos del *shell*, y desaparecen cuando sus padres desaparecen.

6.6 Comunicación entre procesos

6.6.1 UTILIZACIÓN DE LOS PIPES PARA ENCADENAMIENTO DE ÓRDENES

Como se comentó en el capítulo 3, el *shell* de Unix permite la utilización de la salida estándar de un comando o proceso como entrada estándar a otro especificando el símbolo de tubería o *pipe* “|” entre los comandos a encadenar.

Ejemplo:

```
$ echo "No de usuarios conectados: " `who | wc -l` > salida
$ cat salida
```

El *shell* examina la primera de las líneas de comandos y al encontrar las comillas ejecutará los comandos que aparecen a continuación. Es decir, pasará la salida del comando *who* al *wc* como dato de entrada. Como resultado, sustituirá el fragmento entre comillas por un valor numérico, correspondiente al número de usuarios conectados al sistema. El resultado del comando *echo* será por tanto la introducción en un fichero denominado *salida* de la cadena No de usuarios conectados: <Nº>, que será visualizada mediante el comando “cat” de la sentencia siguiente.

Desde el punto de vista de la multitarea, todos los procesos indicados en la línea de órdenes son arrancados a la vez, siendo todos hijos o subordinados del *shell* actual, de manera que puede decirse que se trata de procesos “hermanos”.

6.6.2 UTILIZACIÓN DE PIPES JUNTO CON EL DESDOBLAMIENTO DE LA SALIDA

Existen ocasiones en las que se desea poder contemplar la salida de un programa por pantalla, a la vez que almacenarla en un archivo para, por ejemplo obtener más adelante una copia impresa del mismo o simplemente para volver a consultarla. Un método para lograrlo sería ejecutar la orden y observar su salida por pantalla, y a continuación volverla a ejecutar en combinación con el operador de redirección de la salida a un archivo.

Una alternativa, que logra el mismo resultado en menos tiempo y con menor esfuerzo, es utilizar el comando *tee* generalmente en combinación con el operador de *pipe* (|). Este comando desdobra su entrada generando dos salidas: una la lanza por pantalla, y la otra la introduce donde se le indique.

Ejemplo 1:


```
$ sort DNI.lst | tee DNI.sort
```

De esta manera, se pasa la salida de la orden *sort*, que ordena el contenido del fichero indicado, al comando *tee* y éste visualizará el contenido del fichero ordenado por pantalla y además lo almacenará en un fichero con el nombre especificado.

Es un comando indispensable para aquellas ocasiones en que se desea capturar el diálogo entre un usuario y un programa durante la ejecución de un programa interactivo.

En la tabla 13 se muestran las posibles opciones de este comando.

Opción	Descripción
-a	Añade la salida al archivo de manera que si éste ya existía no lo sobrescribe. Simplemente lo extiende.
-i	Ignora las interrupciones, es decir, no responde a las señales de interrupción generadas durante su ejecución.

Tabla 13: opciones del comando *tee*

Ejemplo 2:

```
$ who | tee -a DNI.sort
```

Esta línea de comandos permite visualizar los usuarios que actualmente se encuentran conectados al sistema, guardando la lista en un archivo denominado *DNI.sort*. Si este archivo no existe, simplemente se crea, pero si ya existe, el resultado de la ejecución del comando *who* se añade al final del mismo en vez de “machacar” su contenido.

6.6.3 PROGRAMACIÓN DE LA COMUNICACIÓN ENTRE PROCESOS

Un *pipe* es para Linux un fichero temporal de 4096 bytes, y se crea haciendo uso de la llamada al sistema *pipe()*, en vez de con la llamada *open()*, como en el caso de los ficheros. El formato de la llamada *pipe()* es el siguiente:

```
# include <unistd.h>

int pipe (int fd[2]);
```

A cada *pipe* se asocian dos descriptores de ficheros, de manera que uno de ellos se utiliza para leer del *pipe*, y el otro para escribir en él. Dado que una función (en este caso, la función *pipe()*) no puede devolver dos valores, el parámetro utilizado es un

puntero a un array de dos posiciones enteras en las que se depositarán los dos descriptores del pipe:

- `fd [0]` se utilizará como descriptor para la lectura del *pipe*
- `fd [1]` se utilizará como descriptor para la escritura del mismo.

Para los *pipes* no se proporciona como parámetro ninguna trayectoria, ni absoluta ni relativa, en la que almacenarlo ya que no se registra, en ningún directorio, una entrada para los *pipes*, y por tanto, ningún proceso tendrá acceso al *pipe* si carece de los descriptores de fichero asociados al mismo.

En consecuencia, la comunicación por medio de *pipes*, sólo está garantizada para procesos creados a partir del que creó anteriormente el *pipe* mediante la llamada `fork`, puesto que todos los procesos hijos de un cierto padre “heredan” sus descriptores de fichero.

