Programmazione di sistema

domenica 23 maggio 2021 14:16

Architettura del sistema

I processi vengono eseguiti in modo user, quindi non possono accedere direttamente alle zone di memoria riservate o alle porte di I/O. Il kernel e' il solo programma ad essere eseguito in modalita' privilegiata, con il completo accesso all'hardware. Questo e' possibile grazie ai processori moderni che forniscono gestione hardware della memoria e modalita' protetta (modo user, modo kernel).
In UNIX bisogna distinguere:

- **User space**: l'ambiente in cui sono eseguiti i programmi.
- **Kernel space**: l'ambiente in cui viene eseguito il kernel.

Questo permette di far vedere al processo come se avesse il pieno controllo della CPU e della memoria, completamente ignaro del fatto che altri programmi possono essere messi in esecuzione dal kernel.

System call e librerie

Le interfacce con cui i programmi possono accedere all'hardware vanno sotto il nome di **system call**. Queste procedure possono essere richiamate, generando successivamente una trap che passa il controllo al kernel. Quindi le system call lavorano in kernel space. Normalmente le system call sono mappate in un insieme di funzioni contenute nella libreria fondamentale del sistema, la **C Standard Library**. In generale, anche se sono differenti dal punto di vista concettuale, al programmatore non cambia molto tra l'usare una system call o una funzione implementata tramite system call. Nel caso in cui non sia presente una specifica funzione di libreria e' possibile eseguire una generica system call tramite la funzione `syscall`

```
#include <unistd.h>
#include <sys/syscall.h>
/*Esegue la system call indicata da number*/
int syscall(int number, ...)
```

`number` e' il numero della system call, ma in genere ogni system call ha associato una costante nella forma SYS_* dove al posto di * viene aggiunto il nome della system call. Queste costanti sono riportate nel file `sys/syscall.h` In questo modo viene ritornato un valore, nullo in genere a significare successo, altrimenti un valore negativo simboleggia un errore.

Portabilita'

Uno dei problemi di portabilita' del codice e' quello dei tipi di dato utilizzati, che spesso variano in base all'architettura. Per questo motivo le funzioni di libreria non fanno riferimento ai tipi elementari ma ad una serie di **tipi primitivi** del sistema. Queste sono riportate nel file `sys/types.h`.

Standard POSIX

Lo standard POSIX concerne sia kernel che librerie che comandi. Ha come sottoinsieme l'ANSI C ma ne amplia le funzioni e ne aggiunge di nuove. Per scrivere un programma conforme a POSIX bisogna includere gli header file richiesti dalle varie primitive usate. Se si vuol compilare un programma in modo tale che dipenda solo dallo standard POSIX si puo' usare la seguente istruzione per il preprocessore

```
#define _POSIX_SOURCE 1
```

Interfaccia con i processi

Un processo esegue sempre e solo un programma, avra' la sua copia del codice (che puo' essere condivisa da piu' processi se eseguono lo stesso programma), il suo spazio di indirizzi e variabili proprie.

Funzione `main`

Quando un programma viene lanciato, il kernel esegue un opportuno codice di avvio contenuto nel programma `ld-linux.so`. Questo programma prima carica le librerie condivise che servono, poi effettua il linking dinamico (a meno che non si usi il flag `-static` durante la compilazione) ed infine esegue il programma. Il sistema fa partire un programma chiamando la funzione `main`. Secondo lo standard la funzione `main` puo' prendere o meno argomenti passati nella linea di comando

```
int main (int argc , char * argv [])
```

Come chiudere un programma

Quando `main` ritorna viene chiamata automaticamente la funzione `exit`. Alternativamente si puo' chiamare la system call `_exit` che restituisce il controllo direttamente alla funzione di conclusione dei processi del kernel. Il valore tornato da queste funzioni, chiamato **valore di uscita**, e' ritornato a chi chiama il programma, quindi in genere la shell. In `stdlib.h` ci sono le costanti `EXIT_SUCCESS` e `EXIT_FAILURE`.
La funzione `exit` ha il seguente prototipo

```
#include <stdlib.h>
void exit(int status)
```

E' pensata per eseguire una conclusione pulita del programma. Essa passa il controllo al kernel chiamando `_exit` e restituendo il valore `status` come stato di uscita.
Un programma puo' anche essere interrotto dall'esterno grazie all'uso di un segnale.

I processi e l'uso della memoria

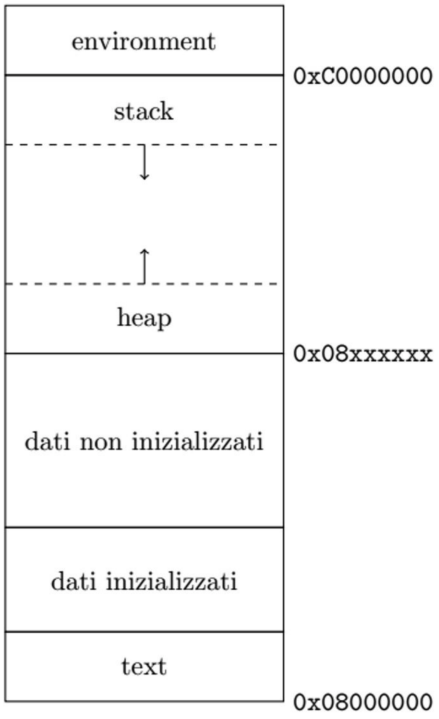
I processi hanno una visione virtuale della memoria. Questa viene divisa in pagine di dimensione fissa, e ciascuna pagina di memoria virtuale e' associata ad una pagina in memoria reale o in memoria secondaria. Ciascun processo possiede una mappa di queste corrispondenze detta **page table**. Una pagina in memoria reale puo' fare da supporto a diversi processi, come nel caso di librerie condivise.

Struttura della memoria di un processo

Sebbene lo spazio di indirizzi di un processo sia molto ampio, solo alcuni di essi sono effettivamente allocati. La memoria virtuale di un processo viene divisa in **segmenti**:

- **Segmento testo**: contiene il codice del programma ed e' condiviso da tutti i processi che eseguono lo stesso programma.
- **Segmento dati**: contiene le variabili globali e quelle statiche e si divide nella parte che contiene le variabili inizializzate e uelle non inizializzate.
- **Heap**: utilizzato per l'allocazione dinamica della memoria.
- **Stack**: contiene tutte le chiamate di procedura e le relative variabili locali e indirizzi di

ritorno. La dimensione di questo segmento aumenta seguento la crescita dello stack del programma, ma non viene ridotta quando quest' ultimo si restringe.



Allocazione della memoria per i programmi C

Il C supporta direttamente soltanto due modalita' di allocazione della memoria: statica e automatica. L'allocazione **statica** e' riservata alle variabili globali e statiche, ovvero quelle poste nel segmento dati. L'allocazione **dinamica** e' fatta per i parametri di funzione e le sue variabili locali, contenute nello stack. L'allocazione **dinamica** non e' prevista direttamente all'interno di C ed e' necessaria in alcuni casi. Le librerie C hanno una serie di funzioni per eseguire l'allocazione dinamica nello heap. L'accesso a queste variabili e' permesso solo attraverso i puntatori alla zona di memoria assegnata. Le funzioni sono quattro:

```
void *calloc(size_t nmemb, size_t size)
```

Alloca nello heap un'area di memoria per un vettore di `nmemb` membri di `size` byte di dimensione. La memoria viene inizializzata a 0. La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e `NULL` in caso di fallimento.

```
void *malloc(size_t size)
```

Alloca `size` byte nello heap. La memoria non viene inizializzata. La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e `NULL` in caso di fallimento.

```
void *realloc(void *ptr, size_t size)
```

Cambia la dimensione del blocco allocato all'indirizzo `ptr` portandola a `size`.
La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e `NULL` in caso di fallimento. Il blocco di memoria restituito puo' non essere un'estensione di quello passato in input, per questo si dovranno gestire i puntatori all'interno del blocco ridimensionato.

```
void free(void *ptr)
```

Disalloca lo spazio di memoria puntato da `ptr`. La funzione non ritorna nulla e non riporta errori.

Esistono due funzioni che vengono utilizzate soltanto quando e' necessario effettuare direttamente la gestione della memoria associata allo spazio dati di un processo, ad esempio qualora si volesse implementare la propria versione delle funzioni di allocazione della memoria.

```
int brk(void *end_data_segment)
```

Sposta la fine del segmento dei dati. Questa funzione e' un'interfaccia dell'omonima system call e sposta la fine del segmento dati all'indirizzo specificato in `end_data_segment`.

```
void *sbrk(ptrdiff_t increment)
```

Incrementa la dimensione dello spazio dati. Restituisce il nuovo indirizzo finale.

Gestione processi

In Unix qualunque processo puo' generarne altri, detti processi figli. Ogni processo e' associato ad un **process identifier** (**pid**). Un'altra caratteristica e' che lancio di un processo e sua creazione sono due operazioni separate. Ogni processo e' generato da un altro tranne il processo **init** e il cui pid e' 1. Il kernel mantiene una **process table** dei processi attivi, con una voce per ogni processo.
I processi vengono creati tramite la funzione `fork`. Il nuovo processo e' una copia di suo padre ma con un nuovo pid. Se si vuole che il processo padre attendi la fine del processo figlio, questo deve essere specificato dopo la `fork` tramite la funzione `wait` o `waitpid`. Un processo figlio puo' essere terminato completamente solo quando la notifica della sua conclusione viene ricevuta dal processo padre, rilasciando dunque tutte le risorse. Per far eseguire ad un processo figlio un altro programma si usa la funzione `exec`. Le funzioni appartenenti a questa famiglia permettono di caricare un altro programma da disco sostituendo quest'ultimo all'immagine di quello corrente. La `fork` ritorna due volte (una nel processo padre e una nel processo figlio) mentre l'`exec` non ritorna mai.

Identificatori dei processi

Il pid e' un tipo di dato `pid_t`. Questo viene assegnato in maniera progressiva, fino ad un massimo di 32678. Oltre questo valore l'assegnazione riparte da un numero piu' basso. I pid piu' bassi sono riservati ai processi del kernel. Il `ppid` e' il pid del genitore ed e' memorizzato dai processi figli. Vi sono due funzioni per ottenere queste informazioni, ovvero

```
pid_t getpid(void)
```

Restituisce il pid del processo corrente.

```
pid_t getppid(void)
```

Restituisce il pid del padre del processo corrente

`fork`

```
pid_t fork(void)
```

Crea un nuovo processo. In caso di successo restituisce il pid del figlio al padre e zero al figlio

oppure in caso di errore ritorna -1 al padre (senza creare il figlio).
Dopo l'esecuzione della `fork`, sia il padre che il figlio continuano l'esecuzione dall'istruzione successiva alla `fork`. Il figlio eredita testo, stack e dati ma ha una memoria sua. Normalmente la chiamata a `fork` puo' fallire solo per due ragioni, o ci sono gia' troppi processi nel sistema o si e' ecceduto il limite sul numero totale di processi permessi all'utente. Generalmente la fork viene usata o per creare processi che eseguano determinate parti di un programma o per creare processi che eseguano nuovi programmi. L'ordine di esecuzione tra child e parent dipende dal meccanismo di scheduling. Bisogna osservare inoltre il modo in cui avviene l'interazione dei processi con i file. Le funzioni come `printf` prevedono output bufferizzato: su disco il buffer e' scaricato solo quando necessario, mentre su terminale viene scaricato ad ogni carattere di new line. Con la redirezione su file quindi, l'output resta nel buffer, e dato che ogni figlio riceve una copia della memoria del padre, esso ricevera' anche quanto c'e' nel buffer, comprese le linee scritte sino ad allora. La posizione corrente sul file e' condivisa tra tutti i processi figli e il padre. Infatti la funzione `fork` duplica tutti i file descriptor aperti nel processo padre, il che comporta che padre e figli condividano le stesse voci nella **file table**.

Terminazione di un processo

Abbiamo gia' visto che un programma termina o con la `return` del `main`, o con le chiamate ad `exit` o `_exit`. Oltre alle conclusioni normali esistono anche modalita' di arresto anomalo che usano la funzione `abort`. In realta' questa invia un segnale di `SIGABRT`. Quando si conclude un processo bisogna chiudere tutti i file aperti, rilasciare la memoria e cosi' via. Oltre a queste operazioni e' necessario disporre di un meccanismo ulteriore che consenta di sapere se la terminazione e' avvenuta, ovvero il **termination status**. Se il processo termina in maniera anomala non puo' specificare nessun **exit status** (tramite `exit` ad esempio), ed il kernel deve generare un termination status. Il termination status contraddistingue lo stato di chiusura di un processo e viene riportato attraverso le funzioni `wait` e `waitpid`. Nel caso di conclusione normale il kernel usa l'exit status per generare il termination status. Quando un processo conclude non e' detto che questo abbia un padre, che potrebbe essere gia' terminato lasciandolo **orfano**. I processi orfani sono adottati da `init`, sostituendo il ppid con il pid di `init`. Inoltre i processi potrebbero terminare prima che il padre possa ricevere lo status (processi **zombie**). Per questo il kernel deve mantenere alcune informazioni in memoria. Evitare i processi zombie ha il vantaggio di diminuire il numero di entry nella process table.

Funzioni di attesa

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

Sospende il processo corrente finche' un figlio non e' uscito, o finche' un segnale termina il processo o chiama una funzione di gestione. La funzione restituisce il pid del figlio in caso di successo e -1 in caso di errore.
Ritorna non appena un processo figlio qualsiasi termina. Se un figlio e' gia' terminato ritorna subito, se sono terminati in tanti bisogna chiamarla piu' volte. Il termination status e' salvato in `status`.

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

Attende la conclusione di un processo figlio `pid`. La funzione restituisce il pid del processo che e' uscito, 0 se e' stata specificata l'opzione `WNOHANG`
e il processo non `e uscito e -1 per un errore. `WNOHANG` permette di evitare il blocco della funzione qualora nessun figlio sia uscito. Usando la costante `WAIT_ANY` (-1) si aspetta un qualunque figlio. Sono disponibili macro per la valutazione dello `status`, come `WEXITSTATUS(status)` che

riporta gli 8 bit meno significativi del codice di ritorno del child.

exec

Quando un processo chiama una delle funzioni `exec` esso viene completamente sostituito dal nuovo programma: il pid non cambia, la funzione semplicemente rimpiazza stack, heap, dati e il codice. Ci sono sei diverse funzioni di `exec` che in realta' sono tutte un front-end a `execve`.

```
int execve(const char *filename, char *const argv[], char *const envp[])
```

La funzione ritorna solo in caso di errore, restituendo -1. La funzione `exec` esegue lo script indicato in `filename`, passandogli argomenti `argv` e un ambiente `envp`.
Le altre funzioni della famiglia forniscono all'utente diverse interfacce per la creazione di un processo.

```
int execl(const char *path, const char *arg, ...)
int execv(const char *path, char *const argv[])
int execle(const char *path, const char *arg, ..., char * const envp[])
int execlp(const char *file, const char *arg, ...)
int execvp(const char *file, char *const argv[])
```

La prima differenza riguarda la modalita' di passaggio dei valori. Differenziamo `v` e `l` che stanno rispettivamente per vector e list. Nel primo caso gli argomenti sono passati tramite il vettore di puntatori `argv[]` che deve essere terminato con un puntatore nullo. Nel secondo caso le stringhe sono passate alla funzione come una lista di puntatori nella forma

```
char * arg0 , char * arg1 , ... , char * argn , NULL
```

In entrambi i casi il primo argomento indica il file contenente il codice. La seconda distinzione riguarda la modalita' con cui si specifica il programma da eseguire. `p` indica che se file non contiene un "/" esso viene considerato come un nome di programma e viene eseguita una ricerca nelle directory nella variabile di ambiente PATH.
Per quanto riguarda i file aperti, se il flag `close-on-exec` e' true questi vengono chiusi nel nuovo processo, altrimenti vengono lasciati aperti.

Controllo di accesso

Unix e' fondato sui concetti di utenti e grupi e separa root che puo' fare ogni cosa dagli utenti normali. Ogni utente ha associato un uid e un gid, che servono al kernel per identificare uno specifico utente. Per motivi di flessibilita', Unix associa ad i processi tre identificatori:

- **Real user/group id**: indica chi ha lanciato il programma.
- **Effective user/group id**: indica l'utente usato per il controllo di accesso.
- **Saved user/group id**: e' una copia dell'euid iniziale.

Generalmente real ed effective id coincidono, tranne se il programma e' eseguito in **suid** o **sgid**, in tal caso sono impostati a chi il file appartiene. Tutti questi campi hanno delle funzioni per poter essere letti

```
uid_t getuid(void)
```

Restituisce l'user-ID reale del processo corrente.

```
uid_t geteuid(void)
```

Restituisce l'user-ID effettivo del processo corrente.

```
gid_t getgid(void)
```

Restituisce il group-ID reale del processo corrente.

```
gid_t getegid(void)
```

Restituisce il group-ID effettivo del processo corrente.

Ogni processo deve tenere in ogni momento i privilegi minimi necessari, per questo si salva l'effective id iniziale nel saved id. In questo modo se successivamente il programma deve tornare a privilegi elevati, euid viene posto al valore contenuto in suid. Le funzioni piu' comuni per cambiare identita' sono le seguenti

```
int setuid(uid_t uid)
```

Imposta l'user-ID del processo corrente.

```
int setgid(gid_t gid)
```

Imposta il group-ID del processo corrente.

Se il processo ha privilegi di super utente, queste funzioni cambiano tutti gli id in quello specificato. Se il processo non ha privilegi di superutente, cambia l'effective id solo se `uid` e' uguale al real id o al saved id. In ogni altro caso da' errore. Queste funzioni servono per far tornare un processo a privilegi minimi uando non ha bisogno di quelli elevati.

```
int getgroups(int size, gid_t list[])
```

Legge gli identificatori dei gruppi supplementari. Se `size = 0` restituisce il numero di gruppi supplementari.

Clone

Con l'introduzione del supporto dei thread a livello kernel si e' resa necessaria un'interfaccia che garantisse maggior controllo sulle modalita' con cui vengono creati i nuovi processi. La system call `clone` svolge questo compito. A differenza di fork, questa permette al processo figlio di condividere parte dell'esecuzione con il processo chiamante, ad esempio la memoria, la tabella dei file descriptor e dei segnali. Nel caso si crei un nuovo thread va specificato un indirizzo di stack, in quanto questo condivide la stessa memoria del padre, e quindi si mischierebbero i due stack.

```
int clone(int (*fn)(void *), void *child_stack, int flags, void
*arg, ...
/* pid_t *ptid, struct user_desc *tls, pid_t *ctid */)
```

Il primo argomento e' un puntatore a funzione, che verra' messa in esecuzione dal nuovo processo. Gli argomenti `ptid`, `tls` e `ctid` sono opzionali. Il comportamento di `clone` dipende dall'argomento `flags`.

Gestione errori

Generalmente il kernel non avvisa mai direttamente un processo dell'occorrenza di un errore ma

viene semplicemente riportato come valore di ritorno.

errno

Per riportare il tipo di errore il sistema usa la variabile globale `errno`, definita in `errno.h`. Il valore di `errno` e' sempre 0 all'avvio di un programma. In generale si controlla sempre `errno` subito dopo aver verificato il fallimento di una funzione attraverso il suo codice di ritorno.

perrno

```
void perror(const char *message)
```

Stampa il messaggio di errore relativo al valore corrente di `errno` sullo standard error preceduto dalla stringa `message`.

File system

Il sistema deve fornire un'interfaccia per accedere all'informazione tenuta allo stato grezzo all'interno dei dischi gestendola come file. Questo viene fatto mediante l'astrazione di **file system**. Le directory sono file trattati in modo speciale dal kernel. Esse contengono una lista di altri file con nomi e informazioni associate. Potendo inserire directory le une dentro le altre si crea una struttura ad albero. Un **pathname** identifica un certo file. Se inizia dalla radice allora la ricerca parte dalla directory root ed e' un path assoluto. In altri casi la ricerca parte dalla directory corrente e si parla di path relativo.

Interfaccia ai file

In Linux le modalita' di accesso ai file sono due basate su due meccanismi con cui e' possibile accedere al loro contenuto. La prima e' rappresentata dai **file descriptor**, ovvero dei numeri interi. Questo tipo di accesso non e' bufferizzato in quanto la scrittura e la lettura vengono eseguite direttamente tramite le system call. La seconda interfaccia sono gli **stream**. Essa fornisce funzioni piu' evolute e bufferizzazione, costruite sopra i file descriptor. Gli stream sono oggetti complessi rappresentati tramite dei puntatori ad una struttura definita nelle librerie C. Si accede ad essi sempre in maniera indiretta utilizzando il tipo `FILE *`.

Filesystem linux

Linux e' in grado di supportare svariati file system. Una delle caratteristiche di base di un file system Unix sono gli **inode**. Essi contengono tutti i metadati riguardanti i file. In una directory vi sono solo il nome del file e il numero di inode associato. Ogni inode ha un link count che indica da quante directory entry e' puntato. Quando si cambia nome di un file viene semplicemente sostiuita la voce nella directory e fatta puntare all'inode.

link e unlink

In Unix e' possibile creare degli alias per uno stesso file e sono chiamati **link**. La realizzazione di un link su Unix e' immediata, basta creare un'associazione nuova con un inode. Questo tipo di collegamento e' detto **hard link**.

```
int link(const char *oldpath, const char *newpath)
```

Crea un nuovo collegamento diretto. Crea su `newpath` un collegamento diretto al file indicato in `oldpath`. Aumenta il numero di riferimenti al file di 1.

```
int unlink(const char *pathname)
```

Cancella un file. In realta' cancella solo la voce che lo referenzia all'interno di una directory e decrementa il link count. Solo quando il link count scende a 0 lo spazio occupato su disco viene rimosso. Il file in realta' continua ad esistere finche' i processi che lo stanno adoperando non lo chiudono.

remove e rename

In Linux non e' possibile usare `unlink` sulle directory. Per farlo si puo' utilizzare `remove`.

```
int remove(const char *pathname)
```

Cancella un nome dal filesystem. Se applicata sui file ha lo stesso comportamento di `unlink`.

```
int rename(const char *oldpath, const char *newpath)
```

Rinomina un file o una directory. Se necessario effettua lo spostamento tra directory diverse. Se ci si riferisce ad un file, `newpath` non deve essere una directory. Se `newpath` indica un file esistente questo viene cancellato e rimpiazzato. Se `oldpath` e' una directory `newpath` deve essere una directory vuota. La funzione rename ha il vantaggio di essere atomica, ovvero non puo' esistere in nessun istante in cui un altro processo puo' trovare attivi entrambi i nomi per lo stesso file.

Link simbolici

Esistono un altro tipo di link detti **soft link**. Essi contengono semplicemente un riferimento ad un altro file o directory. Questi link sono riconosciuti come tali dal kernel, per cui uando si usano delle funzioni di libreria che ricevono come parametro un link simbolico, vengono automaticamente applicate al file puntato.

```
int symlink(const char *oldpath, const char *newpath)
```

Crea un nuovo link simbolico di nome `newpath` il cui contenuto e' `oldpath`. Non si fanno controlli sull'esistenza di `oldpath`, quindi possono crearsi dei **dangling link**. Per leggere le informazioni di un link, e non quelle del file a cui punta si usa la funzione `readlink`.

```
int readlink(const char *path, char *buff, size_t size)
```

Legge il contenuto del link simbolico indicato da `path` nel buffer `buff` di dimensione `size`.

Creazione e cancellazione delle directory

```
int mkdir(const char *dirname, mode_t mode)
```

Crea una nuova directory vuota con nome indicato da `dirname` che puo' essere un path assoluto o relativo. L'argomento `mode` indica i permessi di accesso.

```
int rmdir(const char *dirname)
```

Cancella una directory vuota.

Creazione di file speciali

Unix prevede un insieme di file speciali. La cancellazione e' analoga a quella per i file regolari, mentre la creazione necessita di funzioni apposite.

```
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Crea un inode del tipo specificato sul filesystem. `mode` specifica sia il tipo di file che si vuole creare che l relativi permessi. I tipi di file ammessi sono:

- `S_IFREG` per i file regolari.
- `S_IFBLK` per un dispositivo a blocchi.
- `S_IFCHR` per un dispositivo a caratteri.
- `S_IFSOCK` per un socket.
- `S_IFIFO` per una fifo.

Se si crea un file di tipo dispositivo `dev` deve indicare a quale dispositivo si fa riferimento. Questo valore indica il numero di dispositivo. Il kernel associa a ciasun device un valore numerico. Originariamente era un intero a 16bit diviso in due, chiamati rispettivamente **major number** e **minor number**. Il primo identifica una classe di dispositivi e server al kernel per indicare qualche modulo gestisce quella classe, il minor number invece identifica uno specifico dispositivo. Si possono ottenere questi valori con delle apposite funzioni

```
int major(dev_t dev)
```

Restituisce il major number del dispositivo `dev`.

```
int minor(dev_t dev)
```

Restituisce il minor number del dispositivo `dev`.

Gestione directory

Nonostante siano dei file, le directory non possono essere scritte ad esempio da un processo, questo puo' essere fatto solo dal kernel per ragioni di sicurezza. Lo standard POSIX ha introdotto un'apposita interfaccia per la lettura delle directory chiamata **directory stream**.

```
DIR * opendir(const char *dirname)
```

Apre un directory stream. Questa funzione opera associando un file descriptor sottostante.

```
int dirfd(DIR * dir)
```

Restituisce il file descriptor associato ad un directory stream.
Viceversa, se si e' aperto un file descriptor associato ad una directory e' possibile associarvi un directory stream con la funzione `fopendir`

```
DIR * fdopendir(int fd)
```

Associa un directory stream al file descriptor fd.
Una volta aperto un directory stream la lettura del contenuto di una directory avviene tramite la funzione `readdir`

```
struct dirent *readdir(DIR *dir)
```

Legge una voce dal directory stream. I dati sono memorizzati in una struttura `dirent`. La funzione legge la voce corrente della directory posizionandosi sulla voce successiva. Per leggere l'intero contenuto della directory bisogna ripetere l'esecuzione.

```
void rewinddir(DIR *dir)
```

Si posiziona all'inizio di un directory stream.

```
int closedir(DIR * dir)
```

Chiude un directory stream.

Directory di lavoro

A ciascun processo e' associata una directory che e' chiamata **current directory**. E' da questa directory che si parte quando si esprime un pathname in forma relativa. Il kernel tiene traccia per ogni processo dell'inode della current directory. Per ottenere il pathname si usa la funzione `getcwd`

```
char *getcwd(char *buffer, size_t size)
```

Legge il pathname della directory di lavoro corrente e lo memorizza in `buffer`.
Per cambiare la directory di lavoro si puo' usare la funzione `chdir`

```
int chdir(const char *pathname)
```

Cambia la directory di lavoro in `pathname`. E' possibile riferirsi ad una directory anche tramite il file descriptor

```
int fchdir(int fd)
```

Identica a `chdir`, ma usa il file descriptor `fd` invece del pathname.

Manipolazione delle caratteristiche dei file

Per leggere le informazioni dei file contenute negli inode si usa la famiglia funzioni `stat`.

```
int stat(const char *file_name, struct stat *buf)
```

Legge le informazioni del file il cui pathname e' specificato da `file_name` e le memorizza in `buf`.

```
int lstat(const char *file_name, struct stat *buf)
```

Identica a `stat` ma si riferisce ad un link simbolico e legge le informazioni relative ad esso e non al file puntato.

```
int fstat(int filedes, struct stat *buf)
```

Identica a `stat` ma usa un file descriptor leggendo le informazioni su in file gia' aperto.
Nel campo `st_mode` viene ritornata una maschera binaria che indica il tipo di file. Esistono delle macro che restituiscono true se il file e' di un determinato tipo, false altrimenti. Ad esempio `S_ISREG(m)` controlla se e' un file regolare.

Dimensioni del file

Il campo `st_size` contiene le dimensioni di un file in byte (per le fifo e' sempre nullo). Il campo `st_block` definisce la lunghezza del file in blocchi di 512 byte. Il campo `st_blksize` indica la dimensione preferita per I trasferimenti sul file. In alcuni casi si vuole scartare una parte di dati presenti nel file. Per questo esistono due funzioni

```
int truncate(const char *file_name, off_t length)

int ftruncate(int fd, off_t length))
```

Entrambe le funzioni troncano il file in modo che la sua dimensione abbia valore massimo `length`, ma alla prima e' passato il pathname, alla seconda il file descriptor. Se viene specificata una dimensione maggiore del file questo viene allargato riempiendo la parte in eccesso di zeri.

Gestione delle proprieta' del file system

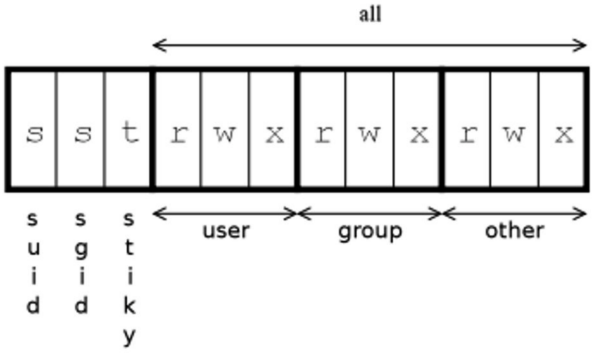
L'operazione di attivazione del file system e' chiamata montaggio ed esiste una funzione apposita per farla su Linux

```
mount(const char *source, const char *target, const char
*filesystemtype, unsigned long mountflags, const void *data)
```

Monta il filesystem di tipo `filesystemtype` contenuto in `source` sulla directory `target`. La directory di destinazione e' detta mount point. Ciascun filesystem e' dotato di caratteristiche specifiche che possono essere attivate o meno tramite l'argomento `mountflags`. Ad esempio `MS_RDONLY` monta in sola lettura.

Permessi

Ogni file ha un owner ed un gruppo di appartenenza. L'insieme dei permessi e' epsresso con un numero a 12 bit. I 9 meno significativi indicano i permessi concessi a owner, gruppi e altri, mentre i restanti tre **suid bit**, **sgid bit**, e **sticky bit** servono ad indicare alcune caratteristiche piu' complesse.



Lo sticky bit e' usato per le directory: se e' impostato (solo root puo' farlo) un file potra' essere rimosso dalla directory solo se l'utente ha il permesso di scrittura sulla stessa. La presenza di questo bit e' indicata dalla lettera *t* nei permessi di esecuzione di altri.
In alcuni casi il controllo sui permessi vuole essere fatto con i real uid/gid e non quelli effettivi. Questo puo' essere fatto grazie alla seguente funzione

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

Il campo `mode` puo' essere epsresso come combinazione di costanti numeriche attraverso l'OR. Ad esempio `F_OK` indica se si vuole verificare l'esistenza del file, `R_OK` verifica il permesso di scrittura, e lo stesso `W_OK` e `X_OK`. Questa funzione ormai e' deprecata in quanto non e' atomica, quindi

qualcuno potrebbe cambiare i permessi di un file dopo aver avuto l'accesso (Time-of-check to time-of-use).
Per cambiare I permessi il sistema mette a disposizione due funzioni

```
int chmod(const char *path, mode_t mode)

int fchmod(int fd, mode_t mode)
```

Queste funzioni differiscono perche' ad una si passa il pathname, all'altra il file descriptor. Il valore di mode puo' essere ottenuto con un OR binario.

mode	Valore	Significato
S_ISUID	04000	Set user ID .
S_ISGID	02000	Set group ID .
S_ISVTX	01000	Sticky bit .
S_IRWXU	00700	L'utente ha tutti i permessi.
S_IRUSR	00400	L'utente ha il permesso di lettura.
S_IWUSR	00200	L'utente ha il permesso di scrittura.
S_IXUSR	00100	L'utente ha il permesso di esecuzione.
S_IRWXG	00070	Il gruppo ha tutti i permessi.
S_IRGRP	00040	Il gruppo ha il permesso di lettura.
S_IWGRP	00020	Il gruppo ha il permesso di scrittura.
S_IXGRP	00010	Il gruppo ha il permesso di esecuzione.
S_IRWXO	00007	Gli altri hanno tutti i permessi.
S_IROTH	00004	Gli altri hanno il permesso di lettura.
S_IWOTH	00002	Gli altri hanno il permesso di scrittura.
S_IXOTH	00001	Gli altri hanno il permesso di esecuzione.

Il sistema associa ad ogni processo una maschera di bit chiamata **umask** che viene utilizzata per impedire che alcuni permessi possano essere assegnati ai nuovi file in sede di creazione.

```
mode_t umask(mode_t mask)
```

Imposta la maschera dei permessi dei bit al valore specificato da `mask` (di cui vengono presi solo i 9 bit meno significativi) e ritorna la maschera precedente.
Per definizione, quando si crea un nuovo file l'uid corrisponde all'uid effettivo del processo (idem per gid). Il sistema fornisce anche delle funzioni che permettono di cambiare sia l'utente sia il gruppo a cui un file appartiene

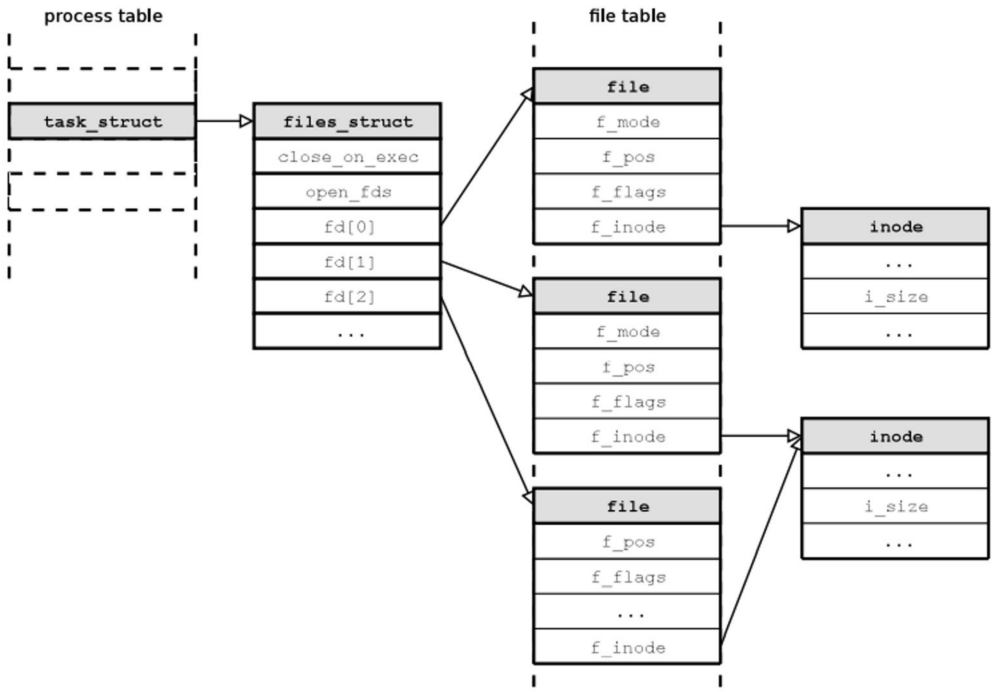
```
int chown(const char *path, uid_t owner, gid_t group)

int fchown(int fd, uid_t owner, gid_t group)

int lchown(const char *path, uid_t owner, gid_t group)
```

Queste funzioni cambiano l'owner e il gruppo di un file in quelli specificati. La seconda sia pplica su un file aperto, nella terza si cambia il possessore di un link simbolico non del file stesso. Se si vole cambiare solo uno dei due valori si inserisce -1 nell'altro.

Interfaccia con file descriptor

Per poter accedere ad un file bisogna aprire un canale di comunicazione con il kernel attraverso una open. All'interno di ogni processo i file aperti sono identificati tramite un file descriptor. Il kernel mantiene un elenco dei processi attivi nella process table ed un elenco dei file aperti nella file table. Nelle voci della process table e' contenuto un puntatore ad una struttura che contiene le informazioni relative ai file che il processo ha aperto tra cui i flag relativi ai file descriptor, il

numero di file aperti e una tabella che contiene un puntatore alla relativa voce nella file table per ogni file aperto. Il file descriptor non e' altro che un indice di questa tabella. Nella file table invece le voci contengono lo stato del file, il valore della posizione corrente e un puntatore al suo inode.



Per convenzione ogni processo lanciato dalla shell ha almeno tre file aperti (con indici 0, 1, 2). Il primo file e' lo standard input, ovvero il file da cui il processo si aspetta di ricevere i dati, il secondo e' lo standard output, ovvero il file su cui si aspetta di inviare i dati e il terzo e' lo standard error, ovvero quello su cui si aspetta di inviare gli errori in uscita.
Possiamo avere due casi:

- Due processi condividono condividono lo stesso offset. Ad esempio in seguito ad una `fork`, i file aperti da un processo rimangono aperti anche per l'altro condividendo gli elementi nella table, quindi anche l'offset.
- I processi sono indipendenti, tuttavia si riferiscono allo stesso inode, quindi allo stesso file.

Open

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

Apre il file indicato da `pathname` nella modalita' indicata da `flags`, e, nel caso il file sia creato, con gli eventuali permessi specificati da `mode`.

Flag	Descrizione
O_RDONLY	Apre il file in sola lettura, le *glibc* definiscono anche O_READ come sinonimo.
O_WRONLY	Apre il file in sola scrittura, le *glibc* definiscono anche O_WRITE come sinonimo.
O_RDWR	Apre il file sia in lettura che in scrittura.
O_CREAT	Se il file non esiste verrà creato, con le regole di titolarità del file viste in sez. 5.3.4. Con questa opzione l'argomento mode deve essere specificato.
O_EXCL	Usato in congiunzione con O_CREAT fa sì che la precedente esistenza del file diventi un errore[2] che fa fallire open con EEXIST.
O_NONBLOCK	Apre il file in modalità non bloccante, e comporta che open ritorni immediatamente anche quando dovrebbe bloccarsi (l'opzione ha senso solo per le fifo, vedi sez. 11.1.4).
O_NOCTTY	Se pathname si riferisce ad un dispositivo di terminale, questo non diventerà il terminale di controllo, anche se il processo non ne ha ancora uno (si veda sez. 10.1.3).
O_SHLOCK	Apre il file con uno shared lock (vedi sez. 12.1). Specifica di BSD, assente in Linux.
O_EXLOCK	Apre il file con un lock esclusivo (vedi sez. 12.1). Specifica di BSD, assente in Linux.
O_TRUNC	Se usato su un file di dati aperto in scrittura, ne tronca la lunghezza a zero; con un terminale o una fifo viene ignorato, negli altri casi il comportamento non è specificato.
O_NOFOLLOW	Se pathname è un link simbolico la chiamata fallisce. Questa è un'estensione BSD aggiunta in Linux dal kernel 2.1.126. Nelle versioni precedenti i link simbolici sono sempre seguiti, e questa opzione è ignorata.
O_DIRECTORY	Se pathname non è una directory la chiamata fallisce. Questo flag è specifico di Linux ed è stato introdotto con il kernel 2.1.126 per evitare dei *DoS*[3] quando opendir viene chiamata su una fifo o su un dispositivo associato ad una unità a nastri, non deve dispositivo a nastri; non deve essere utilizzato al di fuori dell'implementazione di opendir.
O_LARGEFILE	Nel caso di sistemi a 32 bit che supportano file di grandi dimensioni consente di aprire file le cui dimensioni non possono essere rappresentate da numeri a 31 bit.
O_APPEND	Il file viene aperto in *append mode*. Prima di ciascuna scrittura la posizione corrente viene sempre impostata alla fine del file. Con NFS si può avere una corruzione del file se più di un processo scrive allo stesso tempo.[4]
O_NONBLOCK	Il file viene aperto in modalità non bloccante per le operazioni di I/O (che tratteremo in sez. 12.2.1): questo significa il fallimento di read in assenza di dati da leggere e quello di write in caso di impossibilità di scrivere immediatamente. Questa modalità ha senso solo per le fifo e per alcuni file di dispositivo.
O_NDELAY	In Linux[5] è sinonimo di O_NONBLOCK.
O_ASYNC	Apre il file per l'I/O in modalità asincrona (vedi sez. 12.3.3). Quando è impostato viene generato il segnale SIGIO tutte le volte che sono disponibili dati in input sul file.
O_SYNC	Apre il file per l'input/output sincrono: ogni write bloccherà fino al completamento della scrittura di tutti i dati sull'hardware sottostante.
O_FSYNC	Sinonimo di O_SYNC, usato da BSD.
O_DSYNC	Variante di I/O sincrono definita da POSIX; presente dal kernel 2.1.130 come sinonimo di O_SYNC.
O_RSYNC	Variante analoga alla precedente, trattata allo stesso modo.
O_NOATIME	Blocca l'aggiornamento dei tempi di accesso dei file (vedi sez. 5.2.4). Per molti filesystem questa funzionalità non è disponibile per il singolo file ma come opzione generale da specificare in fase di montaggio.
O_DIRECT	Esegue l'I/O direttamente dai buffer in user space in maniera sincrona, in modo da scavalcare i meccanismi di caching del kernel. In genere questo peggiora le prestazioni tranne quando le applicazioni[6]ottimizzano il proprio caching. Per i kernel della serie 2.4 si deve garantire che i buffer in user space siano allineati alle dimensioni dei blocchi del filesystem; per il kernel 2.6 basta che siano allineati a multipli di 512 byte.
O_CLOEXEC	Attiva la modalità di *close-on-exec* (vedi sez. 6.3.1 e 6.3.6).[7]

Uno dei primi tre flag deve essere sempre specificato (bisogna includere `fcntl.h`>.

Close

```
int close(int fd)
```

Chiude il descrittore `fd`. Se viene chiuso l'ultimo riferimento ad un file nella file table, questo viene rimosso da essa e le risorse rilasciate. Se il file descriptor era l'ultimo riferimento ad un file su disco questo viene eliminato.

lseek

A ciascun file aperto e' associata una posizione corrente nel file che indica in byte l'offset

dall'inizio. Tutte le operazioni di lettura e scrittura avvengono a partire da questa posizione. E' possibile impostare questo valore ad una posizione qualsiasi tramite la funzione `lseek`

```
off_t lseek(int fd, off_t offset, int whence)
```

Imposta la posizione attuale nel file. Il valore `offset` indica la nuova posizione sommato al riferimento indicato in `whence`. Quest'ultimo puo' assumere i seguenti valori:

- `SEEK_SET` fa riferimento all'inizio del file.
- `SEEK_CUR` fa riferimento alla posizione corrente.
- `SEEK_END` fa riferimento alla posizione finale.

Dato che la funzione ritorna la nuova posizione si puo' ottenre l'offset corrente chiamando `lseek(fd, 0, SEEK_CUR)`.

`read` e `pread`

```
ssize_t read(int fd, void * buf, size_t count)
```

Cerca di leggere `count` byte dal file `fd` al buffer `buf`. Dopo la lettura la posizione sul file e' spostata automaticamente in avanti del numero di byte letti.

ssize_t pread(int fd, void * buf, size_t count, off_t offset)

Cerca di leggere `count` byte dal file `fd` al buffer `buf` a partire dalla posizione `offset`. Il comportamento di questa funzione e' identico a quello di una `read` seguita da una `lseek` che riporti la posizione corrente ad un certo `offset`. Tuttavia grazie a `pread` questo viene fatto atomicamente siccome non comportano la modifica dell'offset.

`write` e `pwrite`

```
ssize_t write(int fd, void * buf, size_t count)
```

Scrive `count` byte dal buffer `buf` sul file `fd`. Anche in questo caso si sposta in avanti il current offset. Se il file e' aperto in O_APPEND mode I dati vengono scritti sempre alla fine del file.

```
ssize_t pwrite(int fd, void * buf, size_t count, off_t offset)
```

Cerca di scrivere sul file `fd`, a partire dalla posizione `offset`, `count` byte dal buffer `buf`.

`preadv` e `pwritev`

Spesso ci si ritrova ad effettuare una serie multipla di operazioni di I/O come una serie di scritture e letture da vari tipi di buffer e vogliamo controllare l'atomicita' di queste operazioni.

```
int readv(int fd, const struct iovec *vector, int count)
int writev(int fd, const struct iovec *vector, int count)
```

Eseguono rispettivamente una lettura o una scrittura vettorizzata. `iovec` e' una struttura con due campi che indicano l'indirizzo del buffer e la sua dimensione. `vector` e' un vettore di strutture `iovec`, count la sua dimensione. Per effettuare le operazioni in maniera atomica esistono anche in questo caso i corrispettivi `preadv` e `pwritev`.

`dup` e `dup2`

```
int dup(int oldfd)
```

Crea una copia del file descriptor `oldfd`. Il risultato e' che si hanno due file descriptor che puntano alla stessa voce nella file table. L'unica differenza tra i due file descriptor sono i flag (close on exec viene sempre cancellato nella copia). L'uso principale di questa funzione e' quello di redirezionare l'input e l'output fra l'esecuzione di una `fork` e la successiva `exec`.

```
int dup2(int oldfd, int newfd)
```

Rende `newfd` una copia del file descriptor `oldfd`. Se `newfd` e' gia aperto viene chiuso.

Funzioni `at`

Le principali funzioni viste finora permettono di passare pathname relativi alla directory corrente. Spesso pero' sarebbe utile che ogni thread abbia la sua directory di lavoro. Per questo sono state introdotte delle funzioni che terminano con `at` che permettono l'apertura di un file (o altre operazioni) usando un pathname relativo ad una directory specificata. Queste funzioni prevedono l'apertura iniziale della directory che sara' la baase dei pathname relativi, cosi' da evitare problemi di race condition e migliorare le prestazioni. Il comportamento delle nuove funzioni e' del tutto analogo a quello delle corrispettive classiche, con la sola eccezione del fatto che se fra i loro argomenti si utilizza un pathname relativo questo sara' risolto rispetto alla directory sotto forma di file descriptor indicato come primo parametro.

Funzione	Flags	Corrispondente
faccessat	•	access
fchmodat	•	chmod
fchownat	•	chown,lchown
fstatat	•	stat,lstat
utimensat	•	utimes,lutimes
linkat	•[25]	link
mkdirat	–	mkdir
mknodat	–	mknod
openat	–	open
readlinkat	–	readlink
renameat	–	rename
symlinkat	–	symlink
unlinkat	•	unlink,rmdir
mkfifoat	–	mkfifo

`fcntl`

```
int fcntl(int fd, int cmd)
int fcntl(int fd, int cmd, long arg)
int fcntl(int fd, int cmd, struct flock * lock)
```

Esegue una delle possibili operazioni specificate da `cmd` sul file `fd`. Ad esempio `F_DUPFD` cerca il primo file descriptor disponibile di valore minore od uguale ad `arg` e ne fa una copia di `fd`. `F_SETFD` invece imposta il valore del flag a quello specificato con `arg` (si puo' immettere `FD_CLOEXEC`).

`ioctl`

Anche se i device vengono visti come file, ci sono molte operazioni sui dispositivi che non si

possono fare solo con operazioni di lettura e scrittura (Per esempio: controllare volume, alti e bassi di un dispositivo audio). Per ovviare a questo problema esiste la syscall `ioctl`

```
int ioctl(int fd, int request, ...)
```

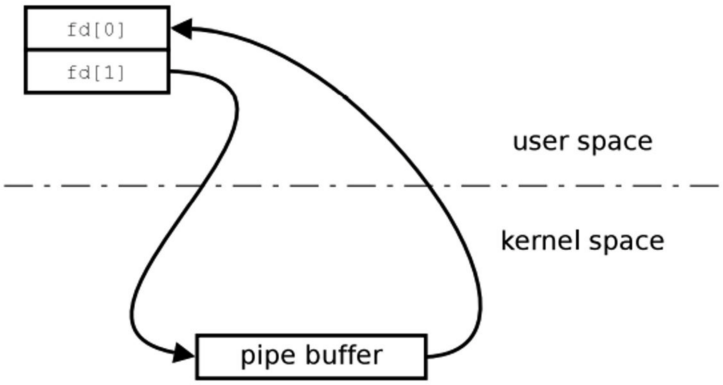
Esegue l'operazione di controllo specificata da `request` sul file descriptor `fd`. Il terzo argomento dipende dall'operazione scelta.

Intercomunicazione tra processi

Le **pipe** nascono con Unix e sono tuttora usate. Si tratta di una coppia di file descriptor connessi tra loro in modo che uello che uno scrive l'altro lo puo' leggere. Questa coppia la si puo' creare mediante una funzione

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

Crea una coppia di file descriptor associati ad una pipe. La coppia di file desrciptor e' restituita in `filedes`: il primo e' aperto in lettura, il secondo in scrittura. Cio' che viene scritto nel file descriptor aperto in scrittura viene ripresentato tale e quale in quello aperto in lettura. Questi descrittori sono connessi ad un buffer del kernel secondo il seguente schema



Visto che un processo figlio condivide i file descriptor, se uno dei processi scrive su un capo della pipe, l'altro puo' leggere. La lettura su una pipe puo' essere bloccante se non vi sono dati.

Named pipe (fifo)

Il limite delle pipe e' dato dal fatto che possono essere utilizzate solo tra processi parenti. Per superare questo problema sono state create le **fifo**. A differenza di usare strutture interne al kernel come le pipe, sono accessibili mediante un inode. Come per le pipe i dati passano in un bufer nel kernel, mentre l'inode serve a fornire solo un punto di riferimento per i processi. Oltre alla funzione `mknod` possiamo usare la funzione `mkfifo` per creare una fifo

```
int mkfifo(const char *pathname, mode_t mode)
```

Crea una fifo `pathname` con i permessi `mode`.
Per utilizzare la fifo il processo deve semplicemente aprire il relativo file: se viene aperto in lettura sara' collegato al capo di uscita, se viene aperto in scrittura al capo di entrata. Il kernel crea una singola pipe per ogni fifo che puo' essere acceduta da piu' processi. La `open` si blocca se viene eseguita quando l'altro capo non e' aperto. Si possono anche aprire le fifo in lettura/scrittura cosi'

da avere sempre successo, usato ad esempio per aprirne una in scrittura quando non ci sono ancora processi in lettura.

Select

```
int select(int ndfs, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout)
```

Attende che uno dei file descriptor degli insiemi specificati diventi attivo. La funzione mette in stato di sleep fintanto che almeno uno dei file descriptor negli insiemi specificati (`readfds`, `writefds`, `exceptfds`) non diventa attivo per un tempo massimo specificato da `timeout` (se e' NULL attende indefinitivamente). Per manipolare i `fd_set` si possono usare delle macro

```
void FD_ZERO(fd_set *set)
```

Inizializza l'insieme (vuoto).

```
void FD_SET(int fd, fd_set *set)
```

Inserisce il file descriptor fd nell'insieme.

```
void FD_CLR(int fd, fd_set *set)
```

Rimuove il file descriptor fd dall'insieme.

```
int FD_ISSET(int fd, fd_set *set)
```

Controlla se il file descriptor fd e' nell'insieme.

`ndfs` indica il valore piu' alto tra i file descriptor indicati negli insiemi (aumentato di 1). La funzione restituisce il numero di file descriptor pronti e gli insiemi vengono sovrascritti per indicare quali dei descrittori contenuti sono pronti con le operazioni relative.

poll

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout)
```

La funzione attende un cambiamento di stato su un insieme di file descriptor. Permette di tenere sotto controllo `nfds` file descriptor indicati in `ufds`. Per ciascun file da controllare va specificata una `pollfd`. La struttura ha tre campi: `fd` che indica il numero del file descriptor, `events` in cui si inserisce una maschera binaria di flag relativa agli eventi che si vogliono controllare e in `revents` il kernel restituira' il relativo risultato.

Segnali

I segnali rappresentano il piu' semplice meccanismo di comunicazione tra processi. Si tratta in sostanza di un'interruzione software portata da un processo. Questi sono usati per notificare ad un processo l'occorrenza di un evento. Questi eventi includono errori, terminazione di un processo figlio, richiesta dell'utente di terminare un programma ecc. Tutti questi eventi richiedono che il kernel generi un segnale. Quando un processo riceve un segnale interrompe la sua esecuzione

normale ed esegue il **signal handler**. Esistono due semantiche per i segnali. Nelle prima versioni di Unix i segnali erano **inaffidabili**, in quanto la ricezione di un segnale e la reinstallazione del suo gestore non sono operazioni atomiche, quindi si possono perdere dei segnali. Nelle implementazioni moderne si usa la semantica **affidabile**, il gestore una volta installato resta attivo e non si hanno i problemi precedenti. I segnali possono classificarsi in **asincroni**, ovvero generati da eventi fuori dal controllo del processo che li riceve, e **sincroni**, legati ad un azione specifica di un programma e inviati durante tale azione.
I segnali vengono ricevuti e il kernel ne prende nota nella `task_struct` del processo. Il segnale diventa dunque pendente fino a che non viene notificato al processo. Normalmente l'invio al processo che deve ricevere il segnale e' immediato, tranne se il segnale e' bloccato prima della notifica. Una volta notificato viene eseguita l'azione specifica. L'azione si puo' in genere selezionare dalle seguenti possibilita':

- Ignorare il segnale.
- Catturare il segnale ed utilizzare il gestore.
- Accettare l'azione predefinita del segnale (che in genere e' la terminazione).

Ogni segnale ha un identificatore che inizia con i caratteri `SIG`, i nomi simbolici sono definiti in `signal.h`. `SIGKILL` termina sempre il programma a qualunque condizione.
Alcuni esempi di generazione di segnali sono l'uso della combinazione di tasti `Ctrl+c` che genera il segnale `SIGINT`, oppure una divisione per 0 che genera un segnale `SIGFPE`.

```
int kill(pid_t pid, int sig)
```

Invia il segnale `sig` al processo specificato con `pid`. Root puo' spedire segnali a chiunque, altrimenti il real o l'effective uid della sorgente deve essere uguale al real o l'effective uid della destinazione. Se il segnale spedito e' null, `kill` esegue i normali meccanismi di controllo errore senza spedire segnali. Alternativamente esiste la funzione `raise` che spedisce un segnale al processo chiamante.

Sigla	Significato
A	L'azione predefinita è terminare il processo.
B	L'azione predefinita è ignorare il segnale.
C	L'azione predefinita è terminare il processo e scrivere un *core dump*.
D	L'azione predefinita è fermare il processo.
E	Il segnale non può essere intercettato.
F	Il segnale non può essere ignorato.

Segnale	Standard	Azione	Descrizione
SIGHUP	PL	A	Hangup o terminazione del processo di controllo.
SIGINT	PL	A	Interrupt da tastiera (C-c).
SIGQUIT	PL	C	Quit da tastiera (C-y).
SIGILL	PL	C	Istruzione illecita.
SIGABRT	PL	C	Segnale di abort da abort.
SIGFPE	PL	C	Errore aritmetico.
SIGKILL	PL	AEF	Segnale di terminazione forzata.
SIGSEGV	PL	C	Errore di accesso in memoria.
SIGPIPE	PL	A	Pipe spezzata.
SIGALRM	PL	A	Segnale del timer da alarm.
SIGTERM	PL	A	Segnale di terminazione C-\.
SIGUSR1	PL	A	Segnale utente numero 1.
SIGUSR2	PL	A	Segnale utente numero 2.
SIGCHLD	PL	B	Figlio terminato o fermato.
SIGCONT	PL		Continua se fermato.
SIGSTOP	PL	DEF	Ferma il processo.
SIGTSTP	PL	D	Pressione del tasto di stop sul terminale.
SIGTTIN	PL	D	Input sul terminale per un processo in background.
SIGTTOU	PL	D	Output sul terminale per un processo in background.
SIGBUS	SL	C	Errore sul bus (bad memory access).
SIGPOLL	SL	A	*Pollable event* (Sys V); Sinonimo di SIGIO.
SIGPROF	SL	A	Timer del profiling scaduto.
SIGSYS	SL	C	Argomento sbagliato per una subroutine (SVID).
SIGTRAP	SL	C	Trappole per un Trace/breakpoint.
SIGURG	SLB	B	Ricezione di una *urgent condition* su un socket.
SIGVTALRM	SLB	A	Timer di esecuzione scaduto.
SIGXCPU	SLB	C	Ecceduto il limite sul tempo di CPU.
SIGXFSZ	SLB	C	Ecceduto il limite sulla dimensione dei file.
SIGIOT	L	C	IOT trap. Sinonimo di SIGABRT.
SIGEMT	L		
SIGSTKFLT	L	A	Errore sullo stack del coprocessore.
SIGIO	LB	A	L'I/O è possibile (4.2 BSD).
SIGCLD	L		Sinonimo di SIGCHLD.
SIGPWR	L	A	Fallimento dell'alimentazione.
SIGINFO	L		Sinonimo di SIGPWR.
SIGLOST	L	A	Perso un lock sul file (per NFS).
SIGWINCH	LB	B	Finestra ridimensionata (4.3 BSD, Sun).
SIGUNUSED	L	A	Segnale inutilizzato (diventerà SIGSYS).

Signal

```
void (*signal(int signo, void (*func)(int) ))(int)
```

signo indica il segnale che si vuole catturare mentre func la funzione che vuole essere eseguita. Se al posto della funzione mettiamo SIG_IGN ignora il segnale, mentre SIG_DFL esegue l'azione predefinita.

alarm e abort

```
unsigned int alarm(unsigned int seconds)
```

Predispone l'invio di SIGALRM dopo seconds secondi. Se si specifica per seconds un valore nullo non verra' inviato nessun segnale e siccome alla chiamata viene cancellato ogni precedente allarme, questo puo' essere usato per cancellare una programmazione precedente. Inoltre la funzione restituisce il numero di secondi rimanenti all'avvio precedente.

```
void abort(void)
```

Abortisce il processo corrente. La funzione non ritorna, il processo e' terminato inviando il segnale

di SIGABRT.

pause e sleep

```
int pause(void)
```

Questa funzione sospende il processo fino a quando un segnale non viene catturato.

```
unsigned int sleep(unsigned int seconds)
```

Pone il processo in stato di sleep per seconds secondi. La funzione restituisce zero se l'attesa viene completata, o il numero di secondi restanti se viene interrotta da un segnale.

Signal set

Le funzioni di gestione dei segnali nate con la semantica inaffidabile hanno dei limiti, in particolare non e' prevista nessuna funzione che permetta di gestire il blocco dei segnali o di verificare lo stato dei segnali pendenti. E' stato quindi introdotto un nuovo tipo di dato `sigset_t` che rappresenta un inseme di segnali. Questo e' rappresentato da un intero di dimensione opportuna, in cui ciascun bit e' associato ad uno specifico segnale.

```
int sigemptyset(sigset_t *set)
```

Inizializza un insieme di segnali vuoto (in cui non c'e' nessun segnale).

```
int sigfillset(sigset_t *set)
```

Inizializza un insieme di segnali pieno (in cui ci sono tutti i segnali).

```
int sigaddset(sigset_t *set, int signum)
```

Aggiunge il segnale `signum` all'insieme di segnali `set`.

```
int sigdelset(sigset_t *set, int signum)
```

Toglie il segnale `signum` dall'insieme di segnali `set`.

```
int sigismember(const sigset_t *set, int signum)
```

Controlla se il segnale `signum` e' nell'insieme di segnali `set`. Ritorna 1 in caso positivo, 0 altrimenti.

sigaction

```
int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact)
```

Installa una nuova azione per il segnale `signum`. Un'azione e' diversa da un gestore in quanto specifica varia caratteristiche in risposta ad un segnale. Quindi questa funzione permette un controllo piu' ampio rispetto a `signal`. Se `oldact` non e' nullo viene restituita l'azione precedente. Questo permette (specificando `act` nullo e `oldact` non nullo) di superare uno dei limiti di `signal`, che non consente di ottenere l'azione corrente senza installarne una nuova. La struttura `sigaction` e'

```
struct sigaction
{
    void (* sa_handler )( int );
    void (* sa_sigaction )( int , siginfo_t *, void *);
    sigset_t sa_mask ;
    int sa_flags ;
    void (* sa_restorer )( void );
}
```

`sa_mask` serve ad indicare I segnali che devono essere bloccati durante l'esecuzione del gestore. Quando il gestore ritorna la maschera viene ripristinata a prima dell'invocazione. Il primo campo e' simile al gestore della funzione `signal`, il secondo permette di specificare piu' informazioni. Installando un gestore di tipo `sa_sigaction` diventa allora possibile accedere alle informazioni restituite attraverso il puntatore alla struttura `siginfo_t`. Alcuni valori per `sa_flags` sono SA_RESTART che forza automatic restart per system call interrotte dal segnale e SA_INTERRUPT che elimina automatica restart per system call interrotte da uesto segnale.

Maschera dei segnali

La **signal mask** indica l'insieme dei segnali la cui consegna e' bloccata. Questa viene ereditata dal padre alla creazione di un proceso figlio. Questa maschera come abbiamo visto puo' essere modificata da `sigaction`.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

Cambia la maschera dei segnali del processo corrente. L'argomento how specifica il modo in cui viene modificata la maschera

Valore	Significato
SIG_BLOCK	L'insieme dei segnali bloccati è l'unione fra quello specificato e quello corrente.
SIG_UNBLOCK	I segnali specificati in **set** sono rimossi dalla maschera dei segnali, specificare la cancellazione di un segnale non bloccato è legale.
SIG_SETMASK	La maschera dei segnali è impostata al valore specificato da **set**.

```
int sigsuspend(const sigset_t *mask)
```

Imposta la signal mask specificata, mettendo in attesa il processo.