Programmazione di sistema in Unix

Possiamo studiare Unix da diversi punti di vista:

Visione dei servizi (programmazione di sistema):

- I **comandi** utilizzabili dagli utenti
- Struttura della macchina astratta (**primitive di sistema**)

Visione funzionale (del progettista):

- Struttura generale del **file system**
- Struttura **a tempo d'esecuzione** (a processi)

Visione realizzativa (dell'implementatore):

- Struttura **del programma sistema operativo**
  - Architettura generale
  - Strutture dati
  - Strutture di controllo (politiche, meccanismi)
- Struttura del **codice oggetto** di Unix creato dal compilatore

Programmazione di sistema

Il modello a processi implica
un modo di programmare diverso dal sequenziale

Si parlerà di programmazione

- *parallela* se si presuppongono più sequenze di calcolo
- *concorrente*: se più sequenze di calcolo parallele devono interagire
- *distribuita*: se più sequenze vengono eseguite da più processori

I linguaggi di programmazione sequenziali **non** permettono di:

- schedulare processi (attivare, terminare, dare priorità)

- sincronizzarsi o comunicare con altri processi

- competere su risorse condivise

- distribuire computazioni su insiemi di processori

Linguaggi di programmazione di sistemi non sequenziali:

- con costrutti appositi (e.g. Ada, CSP, Linda)

- con librerie di system calls (e.g. C + Unix)

Unix: i servizi disponibili

L'utente di Unix può usare le chiamate di sistema per gestire processi e manipolare file

Un **processo** è una attività che esegue un programma; il processo usa servizi di sistema operativo mediante chiamate di sistema

Una **chiamata di sistema** è una invocazione di procedura che richiede un servizio

Ad esempio, le chiamate di sistema che gestiscono i processi sono quelle che trattano la loro *creazione* e *terminazione*

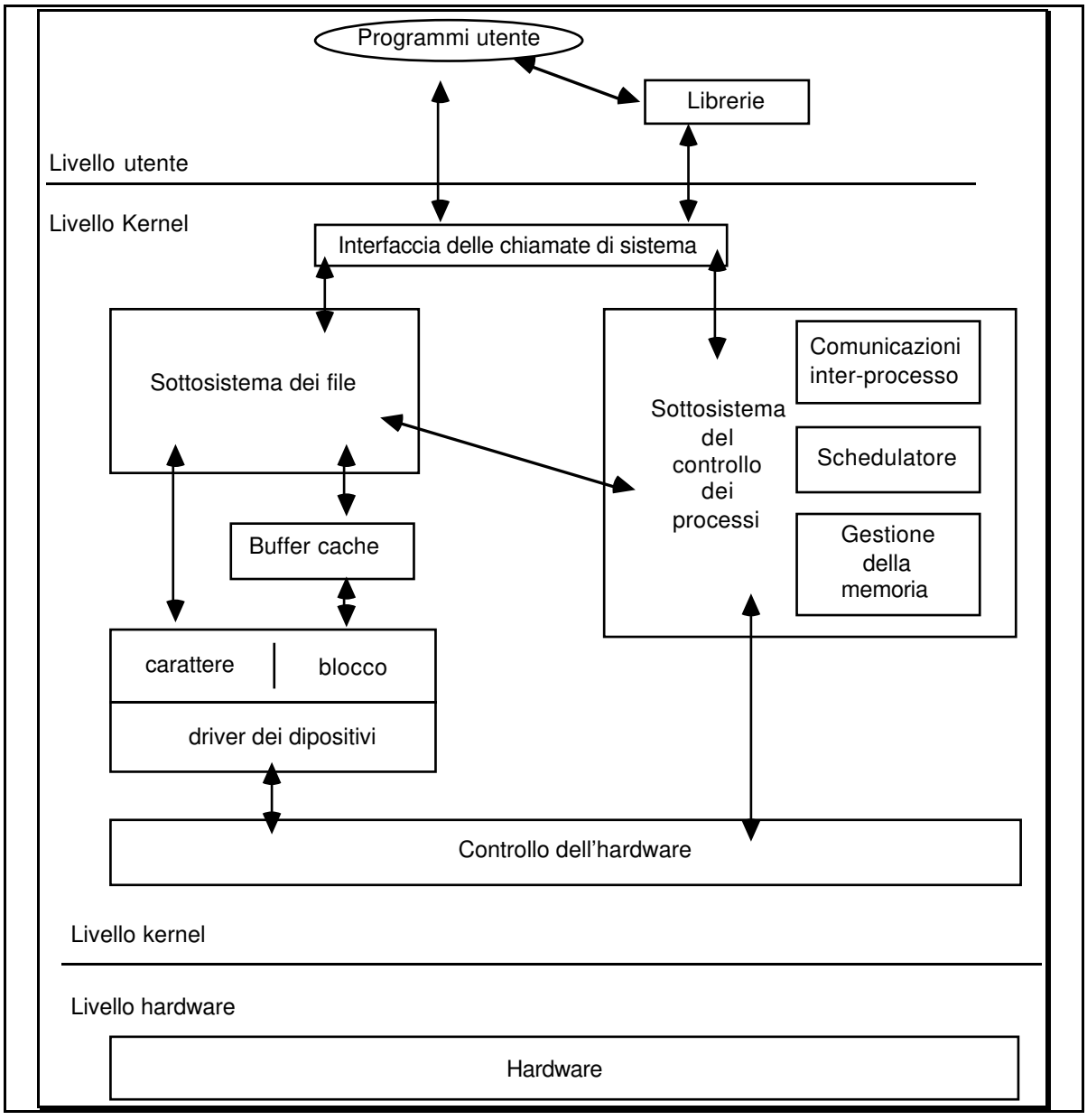
Sono disponibili altre chiamate di sistema per

attendere il termine di un processo figlio
sovrapporre il proprio programma ad un altro
spedire un segnale ad un processo
gestire un segnale
alterare la struttura dell'immagine di un processo

Architettura di Unix

Il nucleo di Unix è un programma monolitico (non suddiviso in moduli);
ciascun processo utente condivide con gli altri la parte di nucleo

Il seguente schema a blocchi descrive a grandi linee la struttura di un s.o. di tipo Unix



Struttura standard del file system

`root` (che è sempre allocato sul disco RAM, in memoria centrale) di solito contiene:

`/bin` – contiene i più importanti programmi in binario
`/dev` – contiene i file speciali per i programmi di I/O
`/etc` – contiene vari programmi per la gestione del sistema
`/lib` – contiene alcuni programmi di libreria
`/tmp` – usato per gestire file temporanei
`/user` – usato per montare altri file system
`/usr` – qui viene montato il disco di sistema

In `/bin` risiedono i programmi usati più di frequente come `cat`, `cp` e `ls`, più altri programmi, come `login` e `sh`

Il directory `/dev` contiene i file speciali per i dispositivi di I/O:

`/dev/ram`	– il disco RAM
`/dev/mem`	– la memoria assoluta
`/dev/kmem`	– la memoria riservata al nucleo
`/dev/null`	– il dispositivo nullo
`/dev/fd0`	– il dispositivo di dischetto 0
`/dev/fd1`	– il dispositivo di dischetto 1
`/dev/hd0`	– il dispositivo minore del disco rigido 0
`/dev/hd1`	– il dispositivo minore del disco rigido 1
`/dev/lp`	– la stampante
`/dev/tty0`	– il terminale 0 (console)

Tutti i file speciali hanno un *numero di dispositivo maggiore* che si riferisce alla classe del dispositivo e un *numero di dispositivo minore* che specifica un particolare dispositivo tra quelli della stessa classe

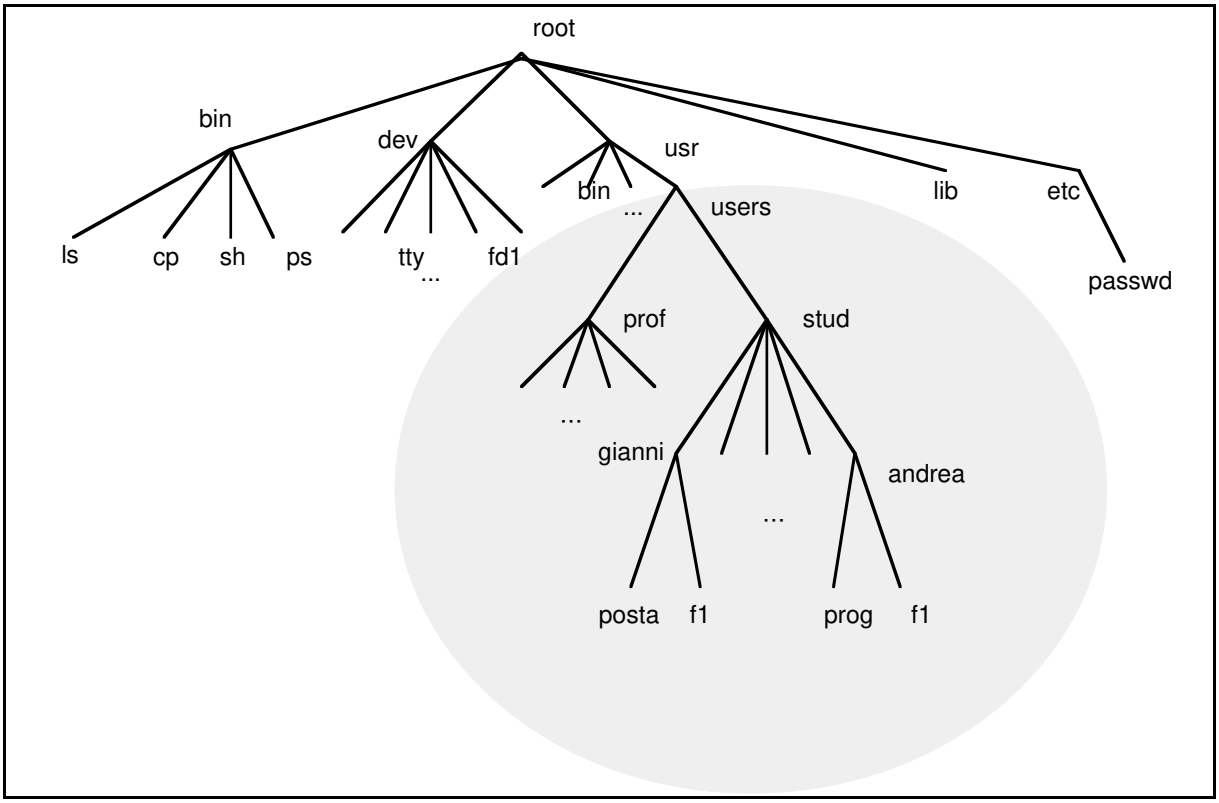
Allocazione fisica del file system

Dal punto di vista del programmatore, i processi agiscono e comunicano unicamente tramite il file system

Il file system è un oggetto allocato fisicamente su dispositivi:

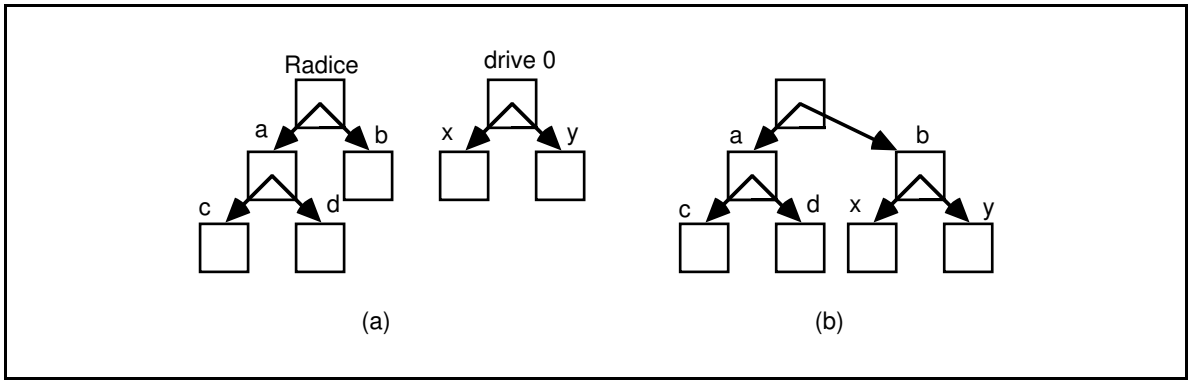
- disco RAM (quindi in memoria centrale)
- disco rigido (o partizioni di esso)
- dischetti, CDROM, dischi removibili, ecc.

Sul disco RAM risiede la parte alta del file system
(`/bin, /dev, /etc, ...`)
montata automaticamente al momento del bootstrapping



Montaggio di un file system

Un file system residente su un dischetto per essere reso accessibile va *montato* logicamente (oltre che immesso fisicamente nel dispositivo)



(a) Prima del montaggio, i file sul drive 0 non sono accessibili
(b) Dopo il montaggio fanno parte del file system gerarchico

Alcuni file sono *file speciali* e rappresentano dispositivi

Esistono due tipi di file speciali:

I file speciali a **blocchi** vengono usati per i dispositivi costituiti da una serie di blocchi indirizzabili in modo diretto, come i dischi

I file speciali a **carattere** sono usati per i modelli di dispositivo costituiti da sequenze di caratteri, come i terminali, le stampanti e la interfacce di rete

Le chiamate di sistema

L'interfaccia tra il sistema operativo e i programmi dell'utente è definita da un insieme di "istruzioni estese" dette **chiamate di sistema**

Tratteremo le chiamate di Unix Versione 7 (Le chiamate di sistema per gli altri Unix - come System V, 4.3BSD, Linux - sono simili, ma ciascuno ha ulteriori chiamate che non tratteremo)

In C, tutte le chiamate di sistema sono sintatticamente invocazioni di funzione; il servizio richiesto si ottiene come effetto laterale; il risultato della richiesta di servizio viene restituito come valore finale della funzione

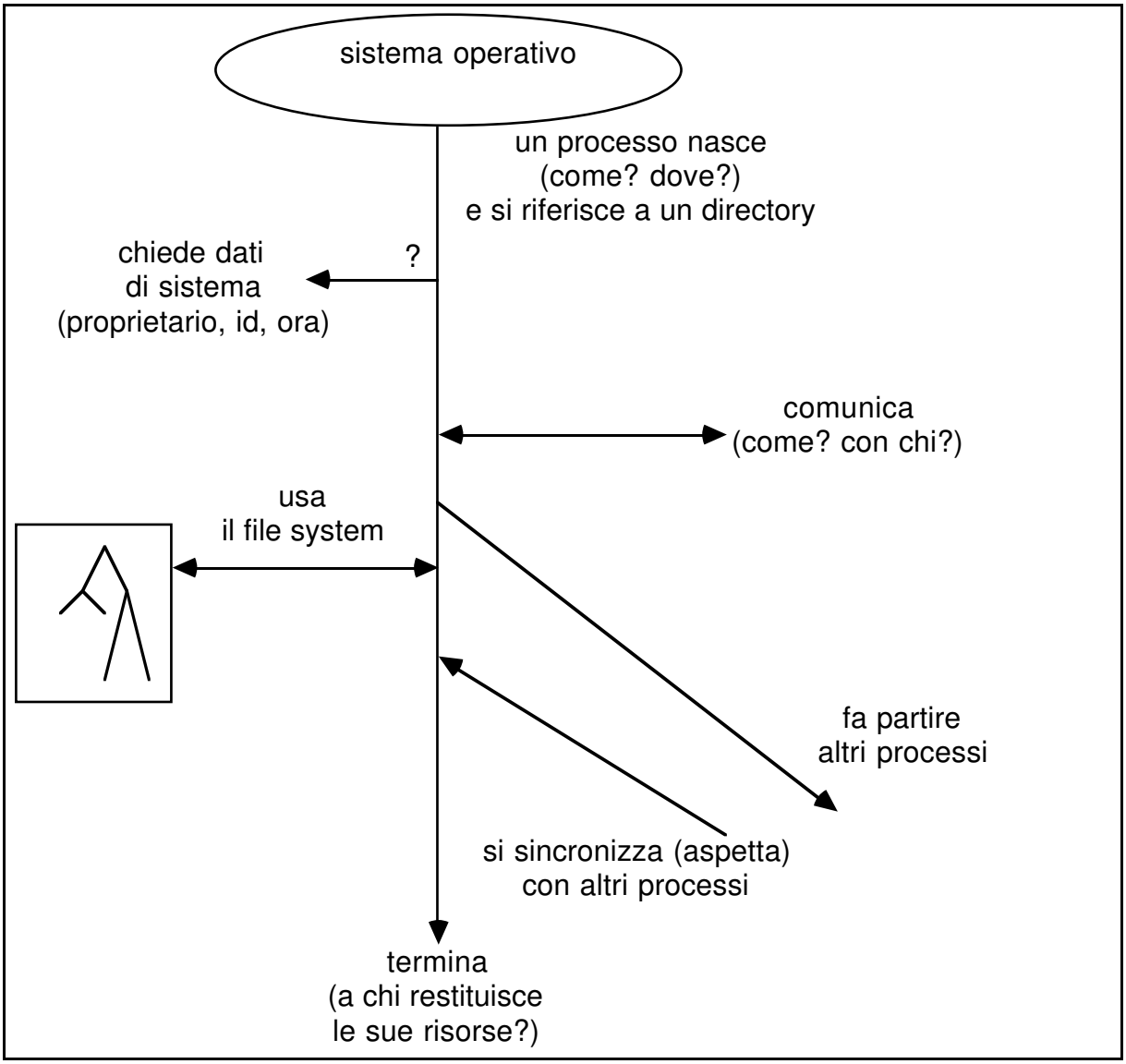
Se la chiamata di sistema non può essere portata a termine il numero dell'errore viene assegnato ad una variabile globale, *errno*

Le chiamate di sistema rientrano in 2 grandi classi: quelle che riguardano i *processi* e quelle che riguardano il *file system*

Unix v7 ha un totale di 41 chiamate di sistema (altre ne sono state aggiunte nelle versioni successive) raggruppate per convenienza in sei categorie

- gestione processi (generazione, terminazione, soppressione)
- sincronizzazione
- gestione file
- gestione file system
- protezione
- gestione del tempo

Vita e opere di un processo



Chiamate di sistema per gestire processi

```
pid = fork();
```
crea un processo figlio “quasi” identico al genitore

```
exit(status);
```
termina l’esecuzione di un processo e riporta lo stato di uscita

```
s = execve(nome, argv, envp);
```
sostituisce l’immagine di memoria del processo

```
size = brk(addr);
```
imposta la dimensione del segmento dati ad “addr”

```
pid = getpid();
```
riporta l’identificatore del processo

Creazione di processi

```
pid = fork();
```

La chiamata crea un processo duplicato esatto del processo originale (inclusi tutti i descrittori di file e i registri)

`fork` restituisce un valore, che è 0 nel figlio ed uguale all'identificatore del processo figlio, o *pid*, nel genitore; usando il pid, i due processi stabiliscono il proprio ruolo

Nota: la chiamata `getpid()` riporta il pid del chiamante (es.: se il figlio vuole sapere il proprio pid, deve usare `getpid`)

Spesso dopo una fork per prima cosa il processo figlio cambia la propria immagine con una chiamata `execve(file,argv,envp)`

La chiamata di sistema `execve` sostituisce l'intera immagine di memoria con il file passato come suo primo parametro

Una mini shell (si osservi l'uso di `fork` ed `exec`):

```
while (TRUE) {          /* ripete all'infinito */
   read_command(command,parameters);
                        /* input da terminale */
   if (fork() !=0) {          /* genera il figlio */
     wait(&status);           /* codice genitore */
   } else {
   execve(command,parameters,0); /* codice figlio */
   }
}
```

Esempio di esecuzione attivata da shell

Consideriamo il comando

```
cp file1 file2
```

La shell non esegue direttamente questo comando esterno: attiva un processo apposito mediante fork. Il processo creato cerca nel file system il codice eseguibile del programma *cp* . Questo (come tutti altri programmi che realizzano comandi) contiene la dichiarazione

```
main(argc, argv, envp)
```

*argc*c indica il numero di elementi presenti nella linea di comando, incluso il nome del programma (nell'esempio precedente *argc* = 3)

argv, è il puntatore ad un vettore
il cui i-esimo elemento è il puntatore all'i-esima stringa del comando
Nell'esempio:
argv[0] punta alla stringa "cp"
argv[1] punta alla stringa di 5 caratteri "file1"
argv[2] alla stringa di 5 caratteri "file2"

envp è un puntatore alla shell:
è un vettore di stringhe che contiene assegnamenti del tipo

nome = *valore*

contiene informazioni come il tipo di terminale o il directory attivo

Terminazione di processo (`exit`)

`exit` ha un solo parametro:

```
exit(status);
```

lo stato di uscita (un intero da 0 a 255) viene riportato nella variabile *status* della chiamata `wait`

Il byte di ordine basso di *status* contiene lo stato di uscita
- è 0 in caso di uscita normale
- è un altro valore se ci sono condizioni di errore

Il byte di ordine alto contiene lo stato di uscita del figlio (da 0 a 255)

Esempio:
se un processo genitore esegue il comando

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

verrà interrotto fino a quando non terminerà il processo figlio

Se il figlio termina con 4 come parametro di *exit*,
il genitore sarà risvegliato con $n = 0$ (nessun errore)
e *status* = 0x0400

La chiamata `brk`

Ricordiamo che l'immagine in memoria di ogni processo è suddivisa in tre segmenti:

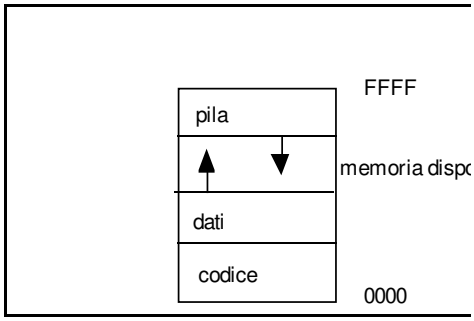
il *segmento codice*
il *segmento dati*
il *segmento pila*

Il segmento **codice** resta fisso; il segmento **pila** (de)cresce da solo
Il **segmento dati** (de)cresce usando la chiamata `brk(addr)`

Il parametro fornisce l'indirizzo al quale il segmento dati deve andare

Questo indirizzo può essere
- maggiore del valore corrente (il segmento dati cresce)
- oppure minore (il segmento dati si sta restringendo)

Il parametro deve essere minore del puntatore alla pila



Chiamate di sistema per sincronizzazione tra processi

```
oldfunc = signal(sig,func);
```
cattura, ignora, ecc. un segnale

```
s = kill(pid,sig);
```
invia un segnale ad un processo

```
residual = alarm(second);
```
invia un segnale SIGALRM dopo un certo tempo

```
s = pause();
```
sospende il processo fino al prossimo segnale

```
s = wait(&status);
```
attende che termini un processo figlio, e testa il suo stato di uscita

La chiamata `signal`

Esistono situazioni in cui il programmatore ha bisogno di "interruzioni software": si usano i **segnali** (corrispondenti sw alle interruzioni hw)

Quando viene inviato (con alla chiamata `kill(sig,pid)`) un segnale ad un processo che non ha dichiarato la sua disponibilità ad accettare quel segnale, il processo viene ucciso

Un processo può usare la chiamata `signal(sig,function)` per dichiarare che è pronto ad accettare alcuni tipi di segnale e per fornire l'indirizzo della procedura di gestione del segnale; oppure si può specificare o la costante SIG_IGN per ignorare tutti i successivi segnali o la costante SIG_DFL ristabilisce l'azione standard dell'uccisione del processo quando arriva un segnale

Nella tabella che segue,gli asterischi denotano segnali che, se non catturati o ignorati, provocano un dump della memoria

(1)	SIGHUP	Rileva una connessione telef. interrotta
2	SIGINT	E` stato battuto sulla tastiera il tasto DEL
3*	SIGQUIT	Segnale di abbandono dalla tastiera
(4)*	SIGILL	Istruzione illegale
(5)*	SIGTRAP	Eccezione di traccia
(6)*	SIGIOT	Istruzione IOT
(7)*	SIGEMT	Istruzione EMT
(8)*	SIGFPE	Overflow/underflow in virgola mobile
9	SIGKILL	Uccisione
(10)*	SIGBUS	Errore sul bus
(11)*	SIGEGV	Violazione di segmentazione
(12)*	SIGSYS	Argomento errato per una chiamata di sistema
13	SIGPIPE	Scrittura in un pipe privo di un lettore
14	SIGALRM	Allarme
15	SIGTERM	Segnale di terminazione generato dal software
16		non assegnato

La chiamata `kill`

La chiamata `kill(pid,sig)` permette ad un processo di mandare segnali ad un altro processo (sempre che abbiano la stessa uid: processi non appartenenti allo stesso uid non possono mandarsi segnali)

```
r = kill(pid,SIGKILL);
```

Inviando il segnale 9 (SIGKILL) ad un processo di background, lo si può uccidere

`alarm(T)` specifica un intervallo T in secondi, dopo di che viene inviato al processo un segnale SIGALARM

`pause()` sospende il processo fino a quando non arriva il successivo segnale; si può usare per evitare l'attesa attiva

Esempio: il comando kill (kill.c)

```
/* kill - send a signal to a process */

#include <signal.h>

main(argc,argv)
int argc; char **argv;
{
   int proc, signal = SIGTERM;
   if (argc < 2) usage();
   if (argc > 1 && *argv[1] == '-') {
     signal = atoi(&argv[1][1]);
     if (!signal)  usage();
     argv++;
     argc--;
   }
   while(--argc) {
     argv++;
     proc = atoi(*argv);
     if (!proc && strcmp(*argv, "0"))
     usage();
     if (kill(proc,signal)) {
     prints("Kill: %s no such process\n", itoa(proc));
         exit(1);
     }
   }
   exit(0);
}

usage()
{  prints("Usage: kill [-sig] pid\n");
   exit(1);
}
```

Chiamate di sistema per la gestione di file

```
fd = creat(name,mode);
```
crea un nuovo file o tronca un file esistente

```
fd = open(file,how);
```
apre un file in lettura e/o scrittura

```
s = close(fd);
```
chiude un file aperto

```
n = read(fd,buffer,nbytes);
```
copia i dati da un file in un buffer

```
n = write(fd,buffer,nbytes);
```
copia i dati da un buffer in un file

```
pos = lseek(fd,offset,whence);
```
muove il puntatore entro nel file

```
s = stat(name,&buf);
```
legge e riporta lo stato del descrittore di un file

```
s = fstat(fd,buf);
```
legge e riporta lo stato del descrittore di un file

```
fd = dup(fd1);
```
alloca un altro descrittore di file per un file gia' aperto

```
s = pipe(&fd[0]);
```
crea un pipe

```
s = ioctl(fd,request,argp);
```
esegue operazioni speciali su file speciali

La gestione dei file

Per creare un nuovo file si usa la chiamata `creat`

```
fd = creat("abc",0751);
```
primo parametro: nome
secondo parametro: modalità di protezione

L'esempio crea un file chiamato *abc* con modalità 0751 ottale
7 significa permesso `rwx` per il proprietario
5 significa `rx` per il gruppo
1 significa `x` per gli altri

`creat` non solo crea un nuovo file, ma lo apre in scrittura e restituisce il descrittore che può essere usato per scrivere il file

Se `creat` viene fatta su un file esistente, questo viene troncato a lunghezza nulla (se ci sono tutti i diritti)

`open` serve per leggere o scrivere un file esistente

```
fd = open(file,c);
```

Primo parametro: nome del file da aprire
(cammino assoluto o relativo al directory di lavoro)
Secondo parametro: un codice 0, 1 o 2
(0=aperto in lettura, 1=scrittura, 2=entrambi)

Il descrittore di file restituito viene usato in `read` o `write`

Il file viene chiuso con `close`, che rende il descrittore di file disponibile per un riutilizzo con `open` o `creat` successive

```
s = close(fd);
```

Le chiamate `read` e `lseek`

`read` ha tre parametri:

```
count = read(file, buffer, nbyte);
```

il primo specifica il file,
il secondo specifica il buffer
il terzo specifica il numero di byte da leggere

L'esempio riporta in *count* il numero di byte appena letti dal file entro il buffer; questo valore normalmente è lo stesso di `nbyte` ma può essere più piccolo, se ad esempio durante la lettura viene incontrata la fine del file

Associato a ciascun file c'è un puntatore alla posizione corrente nel file; durante la lettura (o scrittura) sequenziale, questo normalmente punta al successivo byte da leggere (o scrivere)

`lseek` cambia il valore della posizione del puntatore, cosicché le chiamate seguenti a `read` o `write` possono cominciare ovunque nel file, o anche dopo la fine dello stesso

`lseek` ha tre parametri:

```
pos = lseek(fd,offset,whence);
```

il primo è un descrittore di file
il secondo individua una posizione nel file
il terzo specifica se la posizione è relativa all'inizio, alla posizione corrente o alla fine del file

Il valore riportato da `lseek` è la posizione assoluta nel file dopo che il puntatore è stato modificato

Accesso agli attributi dei file

Un file possiede una serie attributi (contenuti nel suo descrittore, **che non è parte del file**); certi programmi li usano
Per ciascun file, Unix tiene traccia del modo, della dimensione, della data dell'ultima modifica e altre informazioni

Esiste una struttura dati standard che contiene gli attributi dei file

```
struct stat {
     short st_dev;        /* dispositivo dell'i-nodo */
     unsigned short st_ino      /* numero di i-nodo */
     unsigned short st_mode;    /* parola di modo */
     short st_nlink;      /* numero di collegamenti */
     short st_uid;        /* id dell'utente */
     short st_gid;        /* id del gruppo */
     short st_rdev;/* dispositivo primario o /*
         /* secondario per file spec. */
     long st_size;        /* dimensione del file */
     long st_atime;/* uguale a st_mtime */
     long st_mtime;/* data dell'ultima modifica */
     long st_ctime;/* uguale a st_mtime */
};
```

Le chiamate `stat` e `fstat` accedono a queste informazioni

```
s = stat(name,&buf);
s = fstat(fd,buf);
```

La prima chiamata specifica il file mediante il nome
La seconda accetta un descrittore di file, utilizzabile per i file ereditati di cui non si conosce il nome

Entrambe le chiamate vogliono come secondo parametro un puntatore ad una struttura nella quale deve essere messa l'informazione

Comunicazione tra processi

La comunicazione tra processi si basa sui pipe

```
cat file1 file2 | sort
```

Nell'esempio la shell crea un pipe:

a) lo standard output del primo processo scrive nel pipe
b) lo standard input del secondo processo legge dal pipe

la chiamata `pipe(&fd[0])` crea un pipe e restituisce due descrittori di file: uno per la scrittura e uno per la lettura

Esempio:
schema di procedura per attivare un pipe tra due processi:

```
#define STD_INPUT 0    /* descr. file per standard in */
#define STD_OUTPUT 1  /* descr. file per standard out */

pipeline(process1, process2)
char *process1, *process2;  /* punt. ai nomi dei progr. */
{ int fd[2];
 pipe(&fd[0]);                          /* crea un pipe*/
 if (fork() != 0)  /* Il genitore esegue questi comandi */
{
    close(fd[0]); /* il pr. 1 non deve leggere dal pipe */
    close(STD_OUTPUT);   /* prepara un nuovo output st. */
    dup(fd[1]);     /* imposta standard output su fd[1] */
    close(fd[1]);     /* il pipe non è più necessario */
    execl(process1, process1, 0);
 } else {             /* il figlio esegue questi comandi */
    close(fd[1]);     /* il proc. 2 non scrive sul pipe */
    close(STD_INPUT);     /* prepara un nuovo st. input */
    dup(fd[0]);      /* imposta standard input su fd[0] */
    close(fd[0]);       /* il pipe non serve più */
    execl(process2, process2, 0);
 } }
```

La chiamata `pipe`

Il programma precedente si basa sulla chiamata

```
pipe(&fd[0]);
```

`fd` è un vettore di due interi: fd[0] è il descrittore di file per la lettura e fd[1] è quello per la scrittura

Viene creato un pipe, poi la procedura genera due processi
(il genitore diventa il primo processo e il figlio diventa il secondo)

I descrittori di file vanno manipolati in modo che l'output del primo processo sia il pipe e l'input del secondo sia lo stesso pipe

Si chiude lo standard output e si esegue una chiamata `dup` che abilita il descrittore di file 1 a scrivere sul pipe; `dup` dà come risultato il minimo descrittore di file disponibile (in questo caso è 1)

Il programma chiude il descrittore di file

Dopo `exec`, il processo iniziale avrà i descrittori di file 0 e 2 immutati e il descrittore di file 1 per scrivere sul pipe
Il codice del figlio è analogo

Il parametro di *execl* è ripetuto perché il primo è il file che deve essere eseguito e il secondo è il parametro che molti programmi si aspettano che corrisponda al nome del file.

La chiamata `dup` (duplicazione del descrittore)

Quando si lavora con i descrittori di file, risulta utile la chiamata `dup`

```
fd = dup(1);
```

`dup` alloca un nuovo descrittore, *fd;* `dup` restituisce sempre il descrittore disponibile con indice minimo

Nell'esempio, corrisponde allo standard output; l'idea è che in seguito verrà chiuso lo standard output e aperto un nuovo file su cui verrà redirezionato l'output del processo

Quando è il momento di ristabilire la situazione originale, si può chiudere il descrittore di file 1; poi si esegue la chiamata

```
n = dup(fd);
```

per assegnare il più basso descrittore di file, cioè 1, allo stesso file di *fd.* Infine *fd* può essere chiuso e quindi si tornerà nella situazione di partenza

Esempio: una mini shell

```
/* leggi linea comandi */
while (read(stdin, buffer, numchars))
{  /* analisi sintattica del comando /*
   if (/* comando contiene '&'/*) amper = 1; /*
flag */
                  else amper = 0;
/* comandi non riconosciuti come parte del linguaggio di shell
*/
if (fork() == 0)
{  if (/* occorre la redirezione dell'output? */
      {  fd = creat(newfile, fmask);
         close(stdout); dup(fd); close(fd);
      }  /* stdout è stato rediretto */
   if (/* occorre gestire pipe ? */)
      {  pipe(fildes);
         if (fork() == 0)
         {/* primo componente della linea comandi */
         close(stdout);  /*  stdout viene diretto sul
pipe */
            dup(fildes[1]);
            close(fildes[1]);
         close(fildes[0]);
   /* il processo figlio esegue il comando */
         }
/* secondo componente della linea comandi */
         close(stdin);  /*  stdin viene rediretto dal
pipe */
         dup(fildes[0]);
         close(fildes[0]); close(fildes[1]);
      }
      execve(command2,command2,0);
   } /* il genitore continua qui e aspetta */
   if (amper == 0) retid = wait(&status);}
```

Controllo dell'I/O: la chiamata `ioctl`

`ioctl` ha 3 parametri:

```
ioctl(fd, TIOCSETP, &sgttyb);
```

Il primo specifica un file
il secondo specifica un'operazione
il terzo è l'indirizzo di una struttura contenente vari flag

`ioctl` si applica solo ai file speciali di caratteri, soprattutto **terminali;** si usa per cambiare alcuni parametri di terminale: modo, kill char, ecc. (vedi il comando *stty*)

In **modo cooked**

- i caratteri "cancella" e "uccidi" lavorano normalmente
- CTRL-S e CTRL-Q possono essere usati per interrompere e far ripartire l'output sul terminale
- CTRL-D significa "fine del file"
- DEL genera un segnale di interruzione
- CTRL-\ forza una copia su disco dell'immagine di memoria

In **modo raw** tutte queste funzioni sono disabilitate: ogni carattere viene trasmesso senza nessuna particolare elaborazione
Una lettura restituirà al processo qualunque carattere sia stato battuto, anche una linea parziale, invece che aspettare una linea completa come in modo cook

Il **modo cbreak** è una via di mezzo
I caratteri "cancella", "uccidi" e CTRL-D sono disabilitati
Sono abilitati CTRL-S, CTRL-Q, DEL e CTRL-\; Come in modo raw, possono essere riportate ai programmi linee parziali

Chiamate di sistema di gestione del file system

```
fd = mknod(name,mode,addr);
```
crea un descrittore di file speciale, regolare o directory

```
s = link(name1,name2);
```
crea un nuovo elemento, name2, nel directory per il file name1

```
s = unlink(name);
```
elimina un elemento del directory

```
s = mount(special,name,rwflag);
```
monta un file system

```
s = umount(special);
```
smonta un file system

```
s = sync();
```
trasferisce tutti i blocchi dalla memoria cache al disco

```
s = chdir(dirname);
```
cambia il directory di lavoro; la chiamata specifica il nuovo directory

```
s = chroot(dirname);
```
cambia il directory radice

La chiamata `mknod`

I file speciali sono creati con `mknod` (invece che con `creat`)

```
fd = mknod("/dev/tty2", 020744, 0x0402);
```

crea un file di nome */dev/tty2* (terminale 2) gli assegna modo 020744 (un file speciale a caratteri con bit di protezione *rwxr--r--*)

Il terzo parametro contiene
- nel byte di ordine più alto il dispositivo principale (4)
- nel byte di ordine più basso il dispositivo secondario (2)

Il dispositivo principale avrebbe potuto essere uno qualunque
un file di nome */dev/tty2* deve avere un dispositivo secondario numero 2

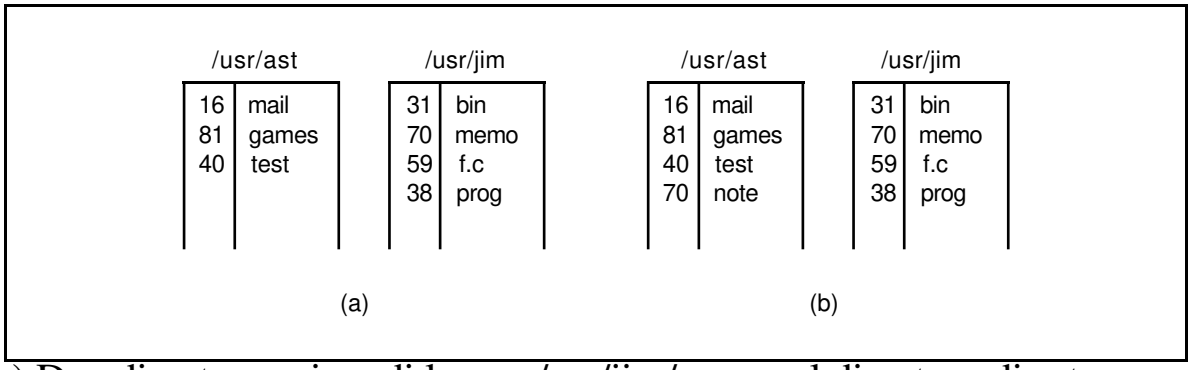
La chiamata `link`

```
s = link(nome1,nome2);
```

`link` permette allo stesso file di apparire sotto due o più nomi (spesso in directory differenti)

Viene usata per permettere a membri dello stesso gruppo di condividere un file comune, in modo che ciascuno lo veda nel proprio directory, eventualmente con nomi diversi

Qualunque cambiamento fatto da alcuni membri del gruppo è immediatamente visibile agli altri membri



(a) Due directory prima di legare /usr/jim/memo al directory di ast.
(b) Gli stessi directory dopo il legamento

Dopo la chiamata

```
link("/usr/jim/memo", "/usr/ast/note");
```

il file *memo* nel directory di *jim* ha adesso un elemento nel directory di *ast* sotto il nome *note*; inoltre, */usr/jim/memo* e */usr/ast/note* si riferiscono allo stesso file.

La chiamata `mount`

Ogni file ha un identificatore unico, detto *i-numero*, che identifica l'i-nodo, cioè il descrittore del file

Un directory è un file contenente un insieme di coppie
(*i-numero*, *nome ASCII*)

`link` crea un nuovo elemento con un (nuovo) nome, usando l'i-numero di un file esistente

Se ne viene rimosso uno (usando `unlink`) l'altro rimane
Se vengono rimossi entrambi, non esiste più nessun puntatore al file: il file viene cancellato

`mount` permette di riunire due file system in uno solo

```
mount("/dev/df0", "/mnt", 0);
```

il primo parametro è il nome di un file speciale a blocchi per il drive 0
il secondo è la posizione nell'albero dove deve essere montato

`mount` permette di integrare dispositivi removibili in un singolo file system gerarchico

Possono essere montati in questo modo anche dischi rigidi o loro porzioni (spesso chiamate *partizioni* o *dispositivi minori*)

Quando un file system non serve più può essere disconnesso mediante il comando `unmount`

Esempio: il comando mount (mount.c)

```
/* mount - mount a file system  */

#include "errno.h"
extern int errno;

main(argc, argv)
int argc;
char *argv[];
{
  int ro;

  if (argc < 3 || argc > 4) usage();
  if (argc==4 && *argv[3]!='-' && *(argv[3]+1)!='r')
    usage();
  ro = (argc == 4 ? 1 : 0);
  if (mount(argv[1], argv[2], ro) < 0) {
    if (errno == EINVAL) {
        std_err("mount: ");
        std_err(argv[1]);
        std_err(" is not a valid file system.\n");
    } else {
        perror("mount");
    }
    exit(1);
  }
  std_err(argv[1]);
  std_err(" mounted\n");
  exit(0);
}

usage()
{
  std_err("Usage: mount special name [-r]\n");
  exit(1);
}
```

La gestione del file system: altre chiamate

Unix conserva in memoria i blocchi di disco usati recentemente per evitare di doverli leggere dal disco se vanno subito riutilizzati

Se viene modificato un blocco (da una chiamata `write` su un file) e il sistema subisce un crash prima che il blocco modificato sia scritto sul disco, il file system resterà danneggiato

Occorre quindi svuotare periodicamente questa parte di disco, in modo che la quantità di dati persi in seguito ad un crash sia piccola

La chiamata `sync` svuota tutti i blocchi che sono stati modificati

Le chiamate `chdir` e `chroot` sono relative ai directory
La prima cambia il directory di lavoro
La seconda cambia il directory radice (modifica il file system)

Dopo la chiamata

```
chdir ("/user/ast/test");
```

un'operazione di apertura del file *xyz* significa aprire */user/ast/test/xyz*

Una volta che un processo ha segnalato di cambiare il suo directory root, tutti i cammini assoluti (cominciano con un "/") inizieranno dal nuovo root

Chiamate di sistema per la protezione

```
s = chmod(name,mode);
```
cambia i bit di protezione associati ad un file

```
s = chown(name,owner,group);
```
cambia proprietario e gruppo di un file

```
oldmask = umask(complmode);
```
imposta una maschera per scoprire i bit di protezione

```
uid = getuid();
```
legge la uid del processo chiamante

```
gid = getgid();
```
legge la gid del processo chiamante

```
s = setuid(uid);
```
imposta la uid del processo chiamante

```
s = setgid(uid);
```
imposta la gid del processo chiamante

Chiamate di sistema per la protezione

A tutti gli utenti autorizzati viene assegnata dal superutente una *uid* (*user identification*)
Ogni processo fatto partire da un utente ha la *uid* di chi lo ha lanciato
Un processo figlio ha la stessa *uid* del suo genitore

Ad ogni file è associato un attributo di 11 bit; i primi 9 bit gestiscono lettura, scrittura ed esecuzione; gli altri due bit di protezione sono:

SETGID (set-group-id: 02000)
SETUID (set-user-id: 04000)

Quando un processo esegue un file che ha SETUID=1 (SETGID=1) acquisisce una uid (una gid) differente dalla sua uid (gid) reale

Quando un utente esegue un programma con SETUID=1, per tutta la durata di quel processo la uid effettiva dell'utente viene cambiata ed è uguale a quella del proprietario del file

Per ottenere queste informazioni esistono le chiamate `getuid` e `getgid`: ciascuna riporta sia la uid, o gid, reale che quella effettiva
Sono necessarie 4 procedure per estrarre le informazioni appropriate: *getuid*, *getgid*, *geteuid* e *getegid*
Le prime ottengono la uid/gid reale; le ultime due quella effettiva

Esempi:
`chmod("file",0644)` cambia la modalità di protezione di un file

`umask(022)` maschera alcuni attributi di protezione (che sono ereditati dai processi figli); per es., una `creat(file,0777)` dopo la `umask` precedente in realtà si comporta come `creat(file,0755)`

Chiamate di sistema per la gestione del tempo

```
seconds = time(&seconds);
```
calcola il tempo trascorso dal 1-1-1970 in secondi

```
s = stime(tp);
```
imposta il tempo trascorso dal 1 gennaio 1970 in secondi

```
s = utime(file,timep);
```
imposta la data dell'ultimo accesso al file

```
s = times(buffer);
```
legge il tempo utente e di sistema consumato finora

Queste 4 primitive riguardano l'orologio di sistema

`time` riporta l'ora in secondi (0 corrisponde alle 0.00 del 1-1-1970)

`stime` permette l'impostazione dell'orologio (solo al super-utente)

`utime` permette al proprietario di un file (o al super-utente) di cambiare la data memorizzata nell'i-nodo del file

Le applicazioni di queste chiamate di sistema sono abbastanza limitate, ma alcuni programmi ne hanno bisogno

`times` restituisce ad un processo le informazioni di addebito
(così può vedere quanto tempo di CPU ha usato direttamente e quanto è invece stato consumato dal sistema)

Vengono riportate anche le quantità di tempo di sistema e di utente consumate dai suoi figli