

PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

0. COMANDI SHELL

pwd [opzioni]

Il comando **pwd** serve per mostrare la directory in cui ci si trova. Opzioni utilizzabili con il comando **pwd**:

- | | |
|----|--|
| -L | utilizza PWD dall'ambiente, anche se contiene collegamenti simbolici |
| -P | evita tutti i collegamenti simbolici |

ls [opzione] [directory]

Il comando **ls** serve per elencare il contenuto di una directory. Alcune opzioni da utilizzare con il comando **ls**:

[directory]	elenca il contenuto della directory specificata, se non specificata viene considerata la directory corrente
-a	elenca anche i file nascosti
-l	elenco dettagliato di file e sotto directory con i loro attributi
-R	elenca ricorsivamente i file nella directory indicata e in tutte le sottodirectory
-s	mostra la dimensione dei file
-S	ordina i file per dimensione partendo dal più grande
-u	ordina i file per data e ora di accesso partendo dal più recente
-X	ordina i file per estensione e ordine alfabetico
-r	elenca i file invertendone l'ordine
--color	mostra i file con colori differenti

cd [directory]

Il comando **cd** serve per spostarsi all'interno delle directory del filesystem.

- | | |
|---------|------------------------------------|
| cd .. | spostarsi alla directory superiore |
| cd /etc | spostarsi nella directory /etc |

mkdir [opzioni] directory

- | | |
|-----------------------------------|--|
| mkdir prova | creare la directory prova/ all'interno della directory corrente |
| mkdir -p prova1/prova2/prova3/bin | Qualora non esistessero, verranno create anche tutte le directory intermedie, a partire da quella corrente |

rm [opzioni] file ...

Il comando **rm** serve per cancellare file o directory dal file system.

- | | |
|----|--|
| -i | chiede conferma prima di cancellare |
| -f | forza la cancellazione del file senza chiedere conferma |
| -r | abilita la modalità ricorsiva usata per la cancellazione delle directory |

rmdir directory

Il comando **rmdir** serve per cancellare directory dal file system.

touch [opzioni] file

Il comando **touch** serve per aggiornare la data dell'ultimo accesso o quello dell'ultima modifica di un file. Se seguito da un nome di file non ancora presente, ne crea uno vuoto con l'estensione indicata.

- | | |
|----|--|
| -a | cambia solo la data dell'ultimo accesso |
| -c | non creare il file |
| -m | cambia solo la data dell'ultima modifica |

od [opzioni] file

Il comando **od** visualizza i file in ottale e altri formati.

- | | |
|----|-------------------|
| -b | formato ottale |
| -c | formato carattere |
| -f | formato float |
| -i | formato intero |
| -l | formato long |

chmod [ugo]a] [+ -] [rwxX] file

User			Group			Others		
R	W	X	R	W	X	R	W	X
4	2	1	4	2	1	4	2	1

Esempi:

chmod u+x script.sh: aggiunge il diritto di esecuzione per il proprietario del file script.sh.

chmod 755 pippo.sh: assegna diritto di lettura, scrittura ed esecuzione all'utente proprietario, diritto di lettura ed esecuzione al gruppo ed agli altri utenti.

chmod 644 pluto.txt: assegna diritto di lettura e scrittura all'utente proprietario ed il solo diritto di lettura al gruppo ed agli altri utenti.

1.0 FILE DESCRIPTOR

Quando si tratta l'argomento di I/O non bufferizzato, ci riferiamo sempre al **file descriptor**, che sono degli interi non negativi (partono da 0). Ogni volta che all'interno di un processo viene aperto un file, il kernel gli assegna un file descriptor. Tutte le system call elencate in precedenza, esclusa la open, prenderanno in input il file descriptor del file e non il nome del file stesso. A differenza del normale utilizzo dei file in C che abbiamo sempre visto, in cui quando veniva invocata la fopen ci veniva restituito un puntatore al file, in questo caso ci viene quindi restituito un file descriptor, che è diverso. Il numero massimo di file che possono essere aperti al giorno d'oggi in un file sono 63.

Ogni nuovo processo apre 3 file standard di default: input, output, error e vi si riferisce ad essi con 3 file descriptor:

- **0 (STDIN_FILENO)**
- **1 (STDOUT_FILENO)**
- **2 (STDERR_FILENO)**

Tenendo conto che all'apertura di un processo vengono generati di default questi file descriptor, è chiaro, il primo file che apriamo all'interno di un processo avrà come indice il 3.

1.1 OPEN

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Permette di aprire un file. Per apertura s'intende il venirsi restituito un file descriptor se l'operazione è andata bene. Altrimenti viene restituito -1. *mode_t mode* non è obbligatorio. *fd* restituito è il più piccolo numero non usato come *fd*.

L'argomento **oflag** è formato dall'OR di uno o più dei seguenti flag di stato:

- **Una ed una sola costante tra:**
 - *O_RDONLY*; *O_WRONLY*; *ORDWR*
- **Una qualunque tra (sono opzionali):**
 - *O_APPEND*: tutto ciò che verrà scritto sarà posto alla fine.
 - *O_CREAT*: usato quando si usa open per creare un file.
 - *O_EXCL*: messo in OR con *O_CREAT* per segnalare errore se il file già esiste. Se il file non esiste non vengono segnalati errori.
 - *O_TRUNC*: se il file già esiste, aperto in write oppure read-write tronca la sua lunghezza a 0, cioè elimina qualsiasi contenuto precedente del file.
 - *O_SYNC (SVR4)*: se si sta aprendo in write, fa completare prima I/O.
 - *O_NOCTTY*, *O_NONBLOCK*

L'argomento **mode** viene utilizzato quando si crea un nuovo file utilizzando *O_CREAT* per specificare i permessi di accesso del nuovo file che si sta creando. Se il file già esiste questo argomento viene ignorato. I permessi si assegnano in questo modo:

Ci sono delle costanti come si vede dall'immagine. Concentriamoci sulla seconda tripla: *S_IRUSR* indica il permesso di lettura per lo user, *S_IWUSR* permessi di scrittura per lo user e così via. La tripla successiva è riferita al gruppo e l'ultima tripla per other. Tutto ciò perché i permessi di un file sono organizzati in tre triplett: tripla di permessi per l'utente, tripla di permessi per il gruppo, tripla di permessi per other.

Alla destra di ogni modalità vi è associato il contenuto reale di ogni costante: per esempio *S_IRUSR* è composto da 4 cifre ottali (in totale 12 bit).

Se per esempio mettiamo in OR *S_IRUSR*, *S_IWUSR* otteniamo:

000 100 000 000 || 000 010 000 000 = 000 110 000 000 → 0600, cioè permessi di lettura e scrittura per l'utente.

0644 indica che il proprietario può leggere e scrivere, mentre gruppo e other possono solo leggere.

Vediamo un esempio pratico →

In questo caso, fd varrà 3 e subito dopo viene fatto exit. All'uscita del processo, tutti i file vengono chiusi.

In questo caso aggiungiamo dei flag messi in OR.

In particolare, vogliamo creare (*O_CREAT*) un file in wronly (*O_WRONLY*) e se per caso il file già esiste allora viene segnalato un errore (*O_EXCL*). Poiché stiamo creando un file, mettiamo anche i permessi: in questo caso con 0600 l'utente può leggere e scrivere, mentre gruppo e other non possono fare nulla. In caso di errore fd varrà -1. Se per caso si decidesse di mettere il file *O_CREAT* senza mettere *O_EXCL* e il file già esiste, quello che succede è che il file non viene ricreato ma si apre quello esistente, e i permessi che sono stati inseriti non vengono considerati, bensì vengono tenuti i precedenti.

mode	Description
<i>S_ISUID</i>	set-user-ID on execution
<i>S_ISGID</i>	set-group-ID on execution
<i>S_ISVTX</i>	saved-text (sticky bit)
<i>S_IRWXU</i>	read, write, and execute by user (owner)
<i>S_IRUSR</i>	read by user (owner)
<i>S_IWUSR</i>	write by user (owner)
<i>S_IXUSR</i>	execute by user (owner)
<i>S_IRWXG</i>	read, write, and execute by group
<i>S_IRGRP</i>	read by group
<i>S_IWGRP</i>	write by group
<i>S_IXGRP</i>	execute by group
<i>S_IRWXO</i>	read, write, and execute by other (world)
<i>S_IROTH</i>	read by other (world)
<i>S_IWOTH</i>	write by other (world)
<i>S_IXOTH</i>	execute by other (world)

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

int main(void)
{
    int    fd;

    fd=open("FILE", O_RDONLY);
    exit(0);
}
```

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

int main(void)
{
    int    fd;
    fd=open("FILE1", O_CREAT|O_EXCL|O_WRONLY, 0600);
    exit(0);
}
```

1.2 CREAT

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int create(const char *pathname, mode_t mode);
```

Viene citata solo per motivi storici. Quello che fa è creare un file dal nome pathname con i permessi descritti in mode. Se va tutto bene, viene restituito il fd del file aperto come write-only, -1 altrimenti.

Questo significa che questa chiamata è esattamente uguale a: `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)`.

Con la creat non posso aprire un file in modalità read-write, per cui questa non viene mai usata.

1.3 UMASK

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

Questa system call serve per impostare la maschera di creazione per l'accesso ad un file. Il valore di ritorno è la maschera di creazione precedente (non restituisce un valore di errore).

Una maschera di creazione di un file permette di disabilitare alcuni permessi in maniera tale che erroneamente l'utente li setti. Viene usato ogni volta che il processo crea un nuovo file o directory, secondo il seguente criterio:

- Setta la maschera di creazione (cmask).
- Comando: umask
- Alla creazione del file viene fatto l'AND tra la maschera negata e il mode della creazione del file.

Dovunque ci sia un 1 nella maschera, tale permesso non verrà dato ad un file/directory.

Mettendo come maschera 0777, tutti i nuovi file non potranno avere alcun permesso. Con umask 0000, qualunque sia il permesso scelto dall'utente, verrà applicato senza restrizione.

Vediamo un esempio →

Se mettiamo umask(0), che corrisponde a 0000, con la creat del file *foo*, non verrà disabilitato alcun permesso che l'utente ha inserito nella creat. Dunque nella pratica il file *foo* avrà permessi 0666.

Con umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH) che corrisponde a umask(0066), vengono disabilitati lettura e scrittura per gruppo e other, a prescindere dal fatto che l'utente li inserisca nei permessi alla creazione del file. Infatti, se creiamo un file *bar*, avremo che i suoi permessi sono 0600 in quanto quelli di gruppo e other sono disabilitati.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
/* esempio di utilizzo di umask */

int main(void)
{
    umask(0);
    if(creat("foo", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
        printf("creat error for foo \n");

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH); /* 0066 */

    if (creat("bar", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
        printf("creat error for bar \n");
    exit(0);
}
```

1.4 CLOSE

```
#include <unistd.h>
int close (int filedes);
```

Banalmente chiude il file con file descriptor filedes. Restituisce 0 se la procedura va a buon fine, -1 altrimenti. Quando un processo termina, tutti i file aperti vengono automaticamente chiusi dal kernel.

1.5 OFFSET

Quando si apre un file si vorrebbe scrivere o leggere dentro. Sorge il problema di dove iniziare a leggere o dove iniziare a scrivere. Questo è il concetto di **offset**. Normalmente, ogni file aperto ha assegnato un **current offset** (intero >0) che misura in numero di byte la posizione nel file. Operazioni come open e creat settano il current offset all'inizio del file a meno che O_APPEND sia specificato. Operazioni come read e write partono dal current offset e causano un incremento pari al numero di byte letti o scritti. Nonostante tutto ciò si ha la possibilità di potersi spostare liberamente all'interno di un file.

1.5.1 LSEEK

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
```

Questa è una system call che ci permette di decidere dove posizionarsi all'interno del file per leggere o scrivere. Restituisce il nuovo offset se OK (possiamo vederlo come intero), -1 altrimenti. Il primo parametro della system call è il file descriptor del file già aperto (si noti che non ci si riferisce più attraverso il nome).

L'argomento **whence** può assumere uno dei seguenti tre valori:

- **SEEK_SET**: ci si sposta del valore di offset a partire dall'inizio.
- **SEEK_CUR**: ci si sposta del valore di offset (positivo o negativo) a partire dalla posizione corrente.
- **SEEK_END**: ci si sposta del valore di offset (positivo o negativo) a partire dalla fine (taglia del file).

Quando si scrive all'interno di un file di testo per esempio "Ciao", e lo si salva all'interno di un file Ciao.txt, all'interno di questo file, a seconda di quale sia la rappresentazione dei set di caratteri c'è un determinato contenuto.

Quando si inserisce SEEK_END con una costante positiva, ci si sposta in avanti e solo se dopo questo spostamento c'è un'operazione scrittura allora tutti i caratteri precedenti tra la fine vecchia del file e il nuovo punto di inizio saranno riempiti da un carattere speciale: il carattere ASCII 0 che corrisponde a \0.

Iseek permette dunque di settare il current offset oltre la fine dei dati esistenti nel file. Se vengono inseriti successivamente dei dati in tale posizione, si crea un buco. Una lettura nel buco restituirà byte con valore 0 (\0). Se Iseek fallisce (restituisce -1), il valore del current offset rimane inalterato.

La chiamata di Iseek non aumenta la taglia del file. Appena si scrive un nuovo carattere, la taglia aumenta di offset definita nella chiamata.

Con questo esempio di codice →

Si apre un file di nome FILE in modalità lettura. Appena si apre un file il current offset è all'inizio. Successivamente c'è una Iseek a cui passiamo il fd ricavato precedentemente. Con SEEK_CUR vogliamo spostarci in avanti a partire dalla posizione corrente, in questo caso di 50 byte. Dopo l'esecuzione della Iseek, se si volesse stampare il valore di i, questo varrà 50.

In questo caso →

Si sta creando un file denominato *buco*, nel caso esista il valore precedente viene troncato. La modalità di apertura è in read-write e i permessi sono di lettura e scrittura per l'utente, lettura per group e other. Il current offset è a 0 in questo caso

Con la write si scrive all'interno del file che ha file descriptor fd, "ABCDEF". Con questa system call, il current offset attuale vale 6.

Con la Iseek, rispetto alla posizione corrente, ci si sposta in avanti di 10 caratteri. Il current offset attuale è ancora 6, anche se virtualmente siamo alla posizione 16.

Con la successiva write, a partire dalla posizione 16, si scrive "GHILMN". In questo modo il nuovo current offset vale 22.

```
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
    int fd,i;
    fd=open("FILE",O_RDONLY);
    i=lseek(fd,50,SEEK_CUR);
    exit(0);
}

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    int fd, i;
    fd= open("buco", O_CREAT|O_RDWR|O_TRUNC, 0644);
    write(fd,"ABCDEF",6);
    lseek(fd,10,SEEK_CUR);
    write(fd,"GHILMN",6);
}
```

1.6 READ

```
#include <unistd.h>
ssize_t read(int filedes, void *buff, size_t nbytes);
```

Con la read, si legge dal file descriptor *filedes* un numero di nbyte e li mette in *buff*.

Il valore di ritorno è il numero di bytes letti, 0 se alla fine del file, -1 in caso di errore.

Naturalmente la lettura parte dal current offset, alla fine dell'operazione il current offset è incrementato del numero di byte letti.

Se il current offset è alla fine del file o anche dopo, viene restituito 0 e non vi è alcuna lettura.

In questo caso →

Viene aperto un file denominato FILE in modalità lettura. Si sposta il current offset in avanti di 50 byte. Vengono letti 20 caratteri e vengono memorizzati in buf. Alla fine di questa operazione il current offset vale 70.

```
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
    int fd,i;
    char buf[20];
    fd=open("FILE",O_RDONLY);
    i=lseek(fd,50,SEEK_CUR);
    read(fd,buf,20);
    exit(0);
}
```

1.7 WRITE

```
#include <unistd.h>
ssize_t write(int filedes, void *buff, size_t nbytes);
```

Con la write, si scrivono *nbyte* presi dal *buff* sul file con file descriptor *filedes*.

Vengono restituiti il numero di byte scritti se l'operazione va a buon fine, -1 in caso di errore.

La posizione da cui si comincia a scrivere è current offset. Alla fine della scrittura current offset è incrementato di *nbytes* e se tale scrittura ha causato un aumento della lunghezza del file anche tale parametro viene aggiornato.

Se viene richiesto di scrivere più byte rispetto allo spazio a disposizione (limite fisico), solo lo spazio disponibile è occupato e viene restituito il numero effettivo di byte scritti.

Se *filedes* è stato con O_APPEND allora current offset è settato alla fine del file in ogni operazione di write.

In questo esempio →

Viene aperto un file di nome FILE in modalità read-write. Ci posizioniamo alla posizione 50 rispetto all'inizio. Vengono letti al massimo 20 caratteri e vengono salvati in buf. La variabile n può valere al più 20.

La write scrive su standard output gli n caratteri che sono stati messi all'interno di buf.

```
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
    int fd,i,n;
    char buf[20];
    fd=open("FILE",O_RDWR);
    i=lseek(fd,50,SEEK_CUR);
    n=read(fd,buf,20);
    write(1,buf,n);
    exit(0);
}
```

1.8 EFFICIENZA DI I/O

Le operazioni di lettura e scrittura sono molto onerose. La CPU è in grado di eseguire molte istruzioni al secondo che operano sulla memoria interna. Quando l'operazione avviene nella memoria di massa (operazione I/O), il processo viene swappato dal processore, entra in una coda speciale per effettuare tale operazione e poi rientra nel processore per continuare l'esecuzione del codice. Ogni volta che si accede al disco è di conseguenza un'operazione molto lenta in confronto ad un accesso alla memoria centrale.

In questo programma →

Viene definita una costante BUFFSIZE con valore 8192. Si dichiara un array di dimensione BUFFSIZE. Parte poi il ciclo: fintantoché il valore di ritorno di una read è maggiore di 0, scrivi ciò che hai letto da qualche altra parte. La read legge da standard input (tastiera) 8192 caratteri e li memorizza in buf. Successivamente vengono scritti su standard output gli n caratteri letti da buf.

```
#define BUFFSIZE 8192
int main(void)
{
    int n;
    char buf[BUFFSIZE];
    while((n = read(STDIN_FILENO, buf, BUFFSIZE))>0)
        if (write(STDOUT_FILENO, buf, n) != n)
            printf("write error");
        if (n < 0)  printf("read error");
    exit(0);
}
```

Questo esempio è stato mostrato per questo motivo →

Se si legge un carattere alla volta, il tempo di esecuzione in termini di CPU e sistema è di 117 secondi (2 minuti) che è molto tempo. Più caratteri si leggono alla volta, più il tempo di esecuzione si dimezza in ordine di potenze di 2.

Superati 8192 (migliore performance) non avviene più una diminuzione ma avviene un piccolo aumento del tempo, questo perché c'è un limite dell'hardware che non dipende più da quanto spesso la CPU chiede i dati, ma da quanto tempo l'hard disk ci mette a rispondere con i dati.

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

1.9 CONDIVISIONE DI FILE

C'è la possibilità per più processi di aprire lo stesso file. Supponiamo di avere il seguente scenario:

2 processi aprono lo stesso file, ognuno si posiziona alla fine e scrive (in 2 passi):

- lseek(fd, 0, SEEK_END);
- write(fd, buff, 100);

Cosa succede se i due processi partono contemporaneamente? Potrebbe accadere che si sovrascrive il contenuto di uno dei due processi. Per risolvere il problema, bisogna aprire il file con il flag O_APPEND in modo che questo fa posizionare l'offset alla fine, prima di ogni write. In questo modo l'operazione diventa atomica (passi che o sono eseguiti tutti insieme o non ne è eseguito nessuno).

1.9.1 DUP & DUP2

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Consentono di assegnare un altro file descriptor ad un file che già ne possedeva uno, cioè filedes.

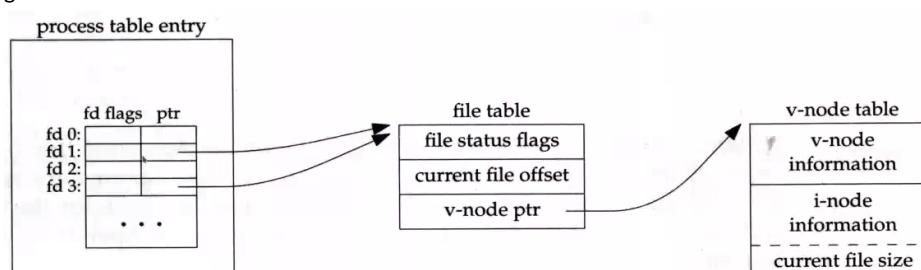
Restituiscono entrambe il nuovo fd se l'operazione va a buon fine.

Ciò significa che quando si lancia in esecuzione dup(3) succede che viene creato un nuovo file descriptor cioè 4 che sarà una copia di 3: entrambi punteranno allo stesso file.

La differenza tra dup e dup2 è che con la dup si copia filedes sul primo file descriptor libero che c'è nella tabella dei file aperti dal processo. Con dup2 si assegna al file avente già file descriptor filedes anche il file descriptor filedes2.

Se filedes2 è già open esso è prima chiuso e poi è assegnato a filedes. Se filedes2=filedes viene restituito direttamente filedes2.

L'effetto di dup/dup2 è il seguente:



O uso fd1 o fd3 è la stessa identica cosa (ad esempio hanno lo stesso current file offset). Questo è diverso dall'avere 2 processi distinti che operano sullo stesso file.

2.0 STRUTTURA STAT

In un file system di tipo Linux abbiamo a disposizione diversi tipi di file:

- **Regular file:** dal punto di vista del kernel un file regolare contenente testo oppure binario.
- **Directory file:** contiene nomi e puntatori ad altri file; solo il kernel può scrivere per motivi di sicurezza.
- **Character special file:** usato per individuare alcuni dispositivi del sistema per esempio il display, la tastiera (/dev/tty), etc...
- **Block special file:** usato per individuare i dischi, ad esempio /dev/sda1 che indica la prima partizione del disco sda.
- **Pipe e FIFO:** usati per la comunicazione tra processi.
- **Symbolic link:** un tipo di file che punta ad un altro file.
- **Socket:** usato per la comunicazione in rete tra processi.

Per ogni file che abbiamo all'interno del file system dobbiamo avere delle informazioni che ci permettono di usare al meglio i file. In particolare:

- **Tipo di file:** (normale, directory, link, speciale) utile alle system call per permettere di capire con che file bisogna lavorare.
- **Permessi:** (lettura, scrittura, esecuzione) ci permettono cosa possiamo fare su un certo file.
- **Tempo ultimo accesso, modifica e cambiamento dello stato (permessi).**
- **User ID e Group ID del proprietario del file.**
- **Numero di link che puntano al file**

La struttura contiene al suo interno tutte le informazioni descritte in precedenza. Ad esempio, vediamo il primo campo di questa struttura che è **st_mode**: contiene al suo interno informazioni quali i permessi e il tipo di file. Il secondo campo, **st_ino**, il numero di i-node che contiene queste informazioni. **st_dev** il numero di device, cioè su quale partizione del nostro file system si trova. **st_nlink** il numero di link, **st_uid** e **st_gid** lo user ID e il group ID del proprietario. **st_size** ci fornisce la size in bytes del nostro file. **st_blocks** il numero di blocchi allocato per il file., **st_blksize** ci dice la dimensione del blocco ottimale per le operazioni di I/O.

```
struct stat {  
    mode_t st_mode; /* file type & mode (permissions) */  
    ino_t st_ino; /* i-node number (serial number) */  
    dev_t st_dev; /* device number (filesystem) */  
    dev_t st_rdev; /* device number for special files */  
    nlink_t st_nlink; /* number of links */  
    uid_t st_uid; /* user ID of owner */  
    gid_t st_gid; /* group ID of owner */  
    off_t st_size; /* size in bytes, for regular files */  
    time_t st_atime; /* time of last access */  
    time_t st_mtime; /* time of last modification */  
    time_t st_ctime; /* time of last file status change */  
    long st_blksize; /* best I/O block size */  
    long st_blocks; /* number of 512-byte blocks allocated */  
};
```

Il campo **st_mode** è l'unico campo di questa struttura che contiene 2 informazioni al suo interno. In questo campo sono inclusi i 9 bit che regolano i permessi di accesso al file cui esso si riferisce →

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

2.0.1 stat, fstat e Istat

```
#include <sys/types.h>  
#include <sys/stat.h>  
int stat (const char *pathname, struct stat *buf);  
int fstat(int fd, struct stat *buf);  
int Istat(const char *pathname, struct stat *buf);
```

Questa system call ci permette di leggere le informazioni relative ad un file che sono memorizzate nell'i-node. Questa system call ha diverse varianti in questo caso rispetto alla **stat** ci sono anche **fstat** e **Istat**.

La **stat** prende come primo parametro un nome di un file e poi passa come secondo parametro un puntatore ad una struttura stat che è stata precedentemente dichiarata. L'effetto è che le informazioni contenute nell'i-node del file, vengono ricopiate all'interno di *buf*. Questa invocazione restituisce 0 se è andato tutto bene, -1 in caso di errore.

La **fstat** si usa quando non si ha a che fare con un nome di un file, ma con un file descriptor che magari è stato dichiarato in precedenza. La funzionalità è la stessa della **stat**.

La **Istat**, che per il momento citiamo solamente, ha a che fare con i link simbolici. Se il primo parametro che passiamo è un link simbolico, ci darà le informazioni sul link simbolico e non sul file puntato.

Nell'uso reale della **stat**, bisogna dichiarare prima la struttura e poi passare il puntatore. Se non viene dichiarata prima, si corre il rischio di non avere a disposizione dello spazio allocato.

L'utilizzatore tipico di queste informazioni che vengono caricate è il comando **ls -l** in quanto le informazioni che ricaviamo da questa chiamata sono contenute nell'i-node.

Se volessimo scoprire di quale tipo è un file che viene passato come parametro, possiamo usare delle **Macro**, che sono funzioni booleane che aiutano ad identificare il tipo di un file verificando ciò che è contenuto nel campo **st_mode** della struttura stat del file. Ne sono 7, una per ogni tipo di file, per verificare se un determinato file è un regular file, una directory e così via. La funzionalità pratica banalmente è la seguente: dalla struttura **stat** recuperiamo il campo **st_mode** e viene passato a queste Macro. Queste macro daranno vero (valore di ritorno 1) se il tipo di file che abbiamo passato è uguale alla Macro che abbiamo invocato, falso altrimenti (**S_ISREG(buf.st_mode)**).

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link (not in POSIX.1 or SVR4)
S_ISSOCK()	socket (not in POSIX.1 or SVR4)

Questa porzione di codice prende da terminale un nome di un file, e all'interno di un if verifica che la lstat che prende in input il nome del file passato da terminale e il puntatore buf (la struttura è stata creata precedentemente) sia minore di 0, nel caso errore.

Successivamente c'è una sequenza di if else if per verificare il file di che tipo è. Questo lo si fa perché sicuramente uno e un solo else if sarà vero e verrà stampato il tipo del file.

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

int main(int argc, char *argv[])
{
    int i; struct stat buf;

    for (i=1;i<argc;i++)
        printf("%s:", argv[i]);
    if (lstat(argv[i],&buf) <0) {
        printf("lstat error");
        continue;
    }
    if (S_ISREG(buf.st_mode)) printf("regular");
    else if (S_ISDIR(buf.st_mode)) printf("directory");
    else if (S_ISCHR(buf.st_mode)) printf("character special");
    else if (S_ISBLK(buf.st_mode)) printf("block special");
    .....
}
exit(0);
```

Vediamo ora quelli che sono gli ID associati ai processi. Così come ci sono degli ID associati ad ogni file (accessibili attraverso *st_uid*, *st_gid*) per dire chi sia il proprietario, ovviamente ci sono anche degli ID associati ai processi (ne sono 6) che sono: **real u/g ID**, **effective u/g ID**, **saved set-u/g-ID**. Quello che capita nella maggioranza dei casi è che il real u ID coincide con l'effective u ID, cioè chi sta eseguendo il nostro processo (ID dell'utente) sarà uguale sia a real u ID, sia a effective u ID.

Il **real**, come suggerisce il nome, ci dice chi siamo realmente nel nostro sistema.

L'**effective** determina i permessi di accesso ai file. Avere accesso ad un file, capire quindi chi è l'utente e chi è il suo gruppo, tutto ciò si fa effettuando un confronto tra l'ID del proprietario del file (presente nell'i-node, *st_uid*, *st_gid*), con l'ID del processo che tenta di accedere al file. Per fare questo test non si usa il real user ID, ma l'effective user ID.

Per i programmi standard eseguiti normalmente, l'effective user ID coincide sempre con il real user ID. C'è però un flag speciale nel campo **st_mode**, che se viene settato, fa in modo che l'effective invece di essere uguale al real, diventa uguale al proprietario (o group) dell'eseguibile. Questi bit possono essere testati usando le costanti **S_ISUID** e **S_ISGID**.

Facciamo un esempio per capire meglio la situazione:

Supponiamo di avere a disposizione un file chiamato **ippo.doc** che è un file dell'utente pippo sul quale solo pippo può scrivere (0644). Prendiamo a disposizione un eseguibile detto **scrivi** che è un word-processor di pippo che può essere usato da tutti. La prima domanda che ci poniamo è: pippo può modificare pippo.doc usando scrivi? Chiaramente sì, in quanto sia il file sia l'editor sono di sua proprietà.

La domanda più interessante è: l'utente pluto può modificare pippo.doc usando scrivi di pippo? Sembrerebbe di no, a meno che scrivi abbia il set-user-id flag settato, poiché in questo modo chiunque esegua scrivi, verrà visto come pippo. Questa caratteristica è utilizzata quando cambiamo la nostra password del sistema, in quanto la password è modificabile solo da root. Settando il set-user-id, il sistema ci vedrà come root e ci permetterà di cambiare la password.

Tutto ciò serve per capire come un processo accede ad un file. Gli ID del proprietario sono una proprietà del file, infatti sono contenuti nell'i-node e di conseguenza c'è un campo nella struct stat. Gli effective ID sono proprietà del processo che utilizza quel file. Per aprire un file (lettura o scrittura) bisogna avere permesso di esecuzione in tutte le directory contenute nel path assoluto del file. Per creare un file bisogna avere permessi di scrittura ed esecuzione nella directory che conterrà il file.

Fatte queste premesse, ogni volta che si cerca di accedere ad un file viene eseguito un vero e proprio algoritmo:

Per prima cosa si controlla che il processo che vuole accedere al file abbia un effective uid=0, nel caso allora l'accesso è libero. Nel caso in cui fallisce la condizione precedente, si effettua un ulteriore controllo: se l'effective uid= owner ID allora si avrà l'accesso in accordo ai permessi "User". Se fallisce anche questo controllo, controllo se l'effective gid= group ID, cioè se chi vuole accedere al file faccia parte almeno dello stesso gruppo del proprietario del file. In caso positivo si ha l'accesso al fine in accordo ai permessi "Group". Se fallisce anche questo controllo allora significa che si rientra nella categoria Other e di conseguenza l'accesso al file si ha in accordo ai permessi "Other".

Supponiamo di avere un file test che ha i seguenti permessi:

`_r__r w _r w _(466)`

Chi può modificare il file? Tutti tranne il proprietario.

Ogni volta che si crea un nuovo file, l'uid è settato come l'effective ID del processo che sta creando il file, cioè significa che un processo che sta creando un nuovo file dovrà crearlo come se fosse un proprio processo quindi uguale all'effective.

Il gid è il group ID della directory nel quale il file è creato oppure il gid del processo.

- eff. uid = 0 --> accesso libero
- eff. uid = owner ID
 - accesso in accordo ai permessi "User"
- eff. gid = group ID
 - accesso in accordo ai permessi "Group"
- accesso in accordo ai permessi "Other"

2.0.2 access

```
#include <unistd.h>
int access (const char *pathname, int mode);
```

Questa system call serve a verificare se il **real** ID ha accesso al file *pathname* nella modalità specificata da *mode*. Restituisce 0 se OK, -1 in caso di errore.

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

Questa porzione di codice controlla se l'utente che esegue il processo per un certo file di cui passiamo il *pathname* da linea di comando abbia i permessi di lettura.

Successivamente prova ad aprire il file in lettura. Vediamo l'esempio di questo eseguibile:

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

int main(int argc, char *argv[])
{
    if (argc != 2)
        printf("usage: a.out <pathname>");

    if (access(argv[1], R_OK) < 0)
        printf("access error for %s", argv[1]);
    else
        printf("read access OK\n");

    if (open(argv[1], O_RDONLY) < 0)
        printf("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");

    exit(0);
}
```

Primo caso:

a.out che è di Rescigno come proprietà. Se lanciamo a.out passando come argomento a.out abbiamo che i permessi di lettura si hanno e quindi il file viene aperto normalmente. Questo succede perché rescigno lancia qualcosa di rescigno.

Secondo caso:

prendiamo il file di un altro utente, basile, che ha come permessi rw solo per il proprietario. Se facciamo a.out su prova avremo access error in quanto l'utente rescigno non può accedere al seguente file in lettura come user id.

Terzo caso:

con il comando *su* si diventa momentaneamente super user. Quello che si fa è modificare il proprietario di un file. Con *chown* abbiamo fatto diventare a.out di basile. Successivamente è stato aggiunto il set user id bit, il quale permette nel momento in cui si esegue un eseguibile di essere visti come effective uid come se fossi il proprietario del file. A questo punto, se si lancia a.out su prova la access fornisce un valore minore di 0, di conseguenza access error, perché ricordiamo che la access verifica il real ID. E' vero che noi precedentemente abbiamo settato il set user id, ma esso ci permette di essere visti come effective uid, no real. L'accesso al file avviene normalmente secondo l'algoritmo di apertura di un file.

2.0.3 chmod e fchmod

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Ci consentono di modificare i bit di permesso del primo argomento. Come negli altri casi è possibile attraverso il nome del file oppure il fd del file. Restituiscono 0 se OK, -1 in caso di errore.

Per cambiare i permessi, l'effective uid del processo deve essere uguale all'owner del file, cioè deve essere un mio file, o il processo deve avere i permessi di root.

Il *mode* è specificato come l'OR bit a bit di costanti che rappresentano i vari permessi.

Questi sono elencati nella tabella qui di fianco.

mode	Description	
S_ISUID	set-user-ID on execution	4000
S_ISGID	set-group-ID on execution	2000
S_ISVTX	saved-text (sticky bit)	1000
S_IRWXU	read, write, and execute by user (owner)	700
S_IRUSR	read by user (owner)	400
S_IWUSR	write by user (owner)	200
S_IXUSR	execute by user (owner)	100
S_IRWXG	read, write, and execute by group	070
S_IRGRP	read by group	040
S_IWGRP	write by group	020
S_IXGRP	execute by group	010
S_IRWXO	read, write, and execute by other (world)	007
S_IROTH	read by other (world)	004
S_IWOTH	write by other (world)	002
S_IXOTH	execute by other (world)	001

Questa porzione di codice ci permette di capire che ci sono 2 modi diversi di modificare i permessi: il primo chmod che prende in input il nome di un file *foo*, mentre il secondo parametro, nella parentesi c'è un AND tra statbuf.st_mode che rappresenta i permessi attuali e ~S_IXGRP che rappresenta il negato di S_IXGRP (dunque tutti 1 e 0 nel campo dell'eseguibilità del gruppo). Questo AND mi consente di eliminare l'esecuzione per il gruppo. Dopo questo AND c'è un OR con S_ISGID, cioè set-group-ID on execution, poiché è un OR stiamo aggiungendo questo permesso.

L'altra modalità invece la vediamo applicata sul file "bar", in cui si passa una sequenza di permessi. I permessi di "bar" sono quelli indicati. Con questa modalità i permessi diventeranno il classico 0644.

```
#include <sys/types.h>
#include <sys/stat.h>
int main(void)
{
    struct stat    statbuf;
    /* turn on set-group-ID and turn off group-execute */
    if (stat("foo", &statbuf) < 0)
        printf("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        printf("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        printf("chmod error for bar");
```

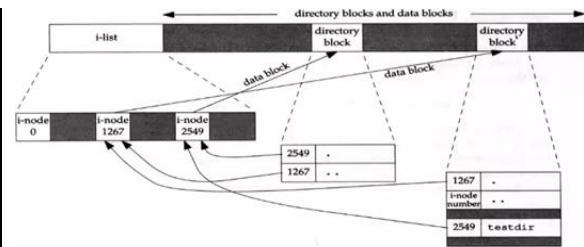
2.0.4 chown

```
#include <sys/types.h>
#include <sys/stat.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

Questa system call è l'equivalente di chmode solo che serve a cambiare il proprietario del primo argomento e li setta uguali a owner e group.

Quando si crea una directory, vengono sempre creati 2 file speciali: **dot** (.) e **dot dot** (..) che servono al file system per poter gestire quelle che chiamiamo directory padre e directory figlio.

Consideriamo la seconda directory block nell'immagine. Supponiamo di creare una nuova directory che si chiama *testdir*. Quando viene creata le viene assegnata un i-node number (2549). Questo vuol dire che questa directory punterà all'i-node 2549. Visto che abbiamo creato una directory, l'i-node punterà a quello che chiamiamo directory block. Il contenuto della directory block sono 2 record: dot (puntatore a se stesso) e dot dot (describe la directory padre). Il padre della directory che ha i-node 2549 è l'i-node 1267. Tutto questo ha portato che appena creiamo una nuova directory, creo un **hard link** che permette dalla directory figlio di risalire alla directory padre e viceversa.



Ogni i-node ha un contatore di link che contiene il numero di directory entry che lo puntano. **Solo** quando scende a zero, allora il file può essere cancellato (i blocchi sono rilasciati mediante funzione **unlink**). Il contatore è nella struct stat nel campo *st_nlink*.

Un file può avere più di una directory entry che punta al suo i-node, cioè più hard link il cui numero è contenuto in *st_nlink*. Gli hard link possono essere creati usando la funzione **link**.

2.0.5 link

```
#include <unistd.h>
int link(const char *path, const char *newpath);
```

Questa system call consente di creare una nuova directory entry *newpath* che si riferisce a *path*. Restituisce 0 se OK, -1 in caso di errore (anche se *newpath* già esiste).

Con la link crea automaticamente il nuovo link ed incrementa anche di uno il contatore dei link *st_nlink* (questa è un'operazione atomica). Il vecchio ed il nuovo link, riferendosi allo stesso i-node, condividono gli stessi diritti di accesso al "file" cui essi si riferiscono.

2.0.6 unlink

```
#include <unistd.h>
int unlink(const char *pathname);
```

Questa system call banalmente rimuove la directory entry specificata da *pathname* e decrementa il contatore dei link del file. Restituisce 0 se OK, -1 in caso di errore. È consentita solo se si hanno i permessi di scrittura ed esecuzione nella directory dove è presente la directory entry.

Attenzione a queste due condizioni:

- Se tutti i link ad un file sono stati rimosse e nessun processo ha ancora il file aperto, allora tutte le risorse allocate per il file vengono rimosse e non è più possibile accedere al file.
- Se però uno o più processi hanno il file aperto, quando l'ultimo link è stato rimosso, pur essendo il contatore dei link a 0 il file continua ad esistere e sarà rimosso solo quando tutti i riferimenti al file saranno chiusi.

Con questa porzione di codice, si apre un file *tempfile*. Subito dopo con l'unlink lo si cancella, con la *printf("file unlinked")* ci serve a notificare sullo standard output questa operazione. Subito dopo c'è una sleep di 15 secondi che serve ad addormentare, bloccare temporaneamente, il nostro processo. Dopo 15 secondi verrà stampato "done". Questa operazione serve a far sì che nel momento in cui si fa un run di questo programmino, nei 15 secondi che abbiamo a disposizione dalla sleep, osserveremo che il file non è più presente nella directory, ma da un punto di vista fisico risiede ancora sul disco. Nel momento in cui termina l'esecuzione del programma, e di conseguenza viene chiuso il file, si libera lo spazio dalla memoria.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "ourhdr.h"

int main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");

    if (unlink("tempfile") < 0)
        err_sys("unlink error");

    printf("file unlinked\n");
    sleep(15);
    printf("done\n");

    exit(0);
}
```

2.0.7 remove

```
#include <stdio.h>
int remove(const char *pathname);
```

Questa system call rimuove il file specificato da *pathname*. Restituisce 0 se OK, -1 in caso di errore. È equivalente alla **unlink**.

2.0.8 rename

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Questa system call assegna un nuovo nome *newname* ad un file od ad una directory data come primo argomento (*oldname*). Restituisce 0 se OK, -1 in caso di errore.

2.1 LINK SIMBOLICI

Un link simbolico (soft link) sono dei file che possono essere visti come un puntatore indiretto ad un file, oppure quelli che possiamo vedere come dei collegamenti. Quando si fa un collegamento sul desktop ad un programma, di fatto viene creato un nuovo file che ci consentirà in maniera veloce di puntare ad un file che si trova in una directory particolare.

Creare un soft link vuol dire creare un nuovo file, creare un nuovo file significa avere un nuovo i-node che punta ad un data block con un contenuto.

Creare un hard link, vuol dire creare una nuova directory entry che punta ad un i-node già esistente.

Quando si usano funzioni che si riferiscono a file (open, read, stat, etc.), si deve sapere se seguono il link simbolico oppure no.

In questa tabella sono riportate varie funzioni, e per ognuna di essa, se segue il link simbolico oppure no. Tutte seguono il link simbolico ad esclusione di quelle che cominciano con la lettera **I** e di altre visibili nella tabella. Ad esempio la **Istat**. Se do in pasto alla stat un link simbolico, la stat mi darà le informazioni del file puntato. Ma se volessi le informazioni del link simbolico, cosa faccio? Con la stat non potrò mai ottenerlo e per questo uso la **Istat**.

Remove e rename lavorano sul link simbolico e non sul file puntato altrimenti non ci sarebbe modo di lavorare sui link simbolici.

La readlink legge il contenuto del link simbolico, è l'equivalente di una open, read e close in un colpo solo.

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat		•
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

2.1.0 symlink

```
#include <unistd.h>
int symlink(const char *path, const char *sympath);
```

Questa system call crea un link simbolico *sympath* che punta a *path*. Restituisce 0 se OK, -1 altrimenti.

I link simbolici possono essere creati anche su file che non esistono.

2.1.1 readlink

```
#include <unistd.h>
int readlink(const char *pathname, char *buf, int bufsize);
```

Questa system call legge il contenuto del link simbolico del primo argomento e lo copia in *buf*, la cui taglia è *bufsize*. Restituisce il numero di byte letti se OK, -1 in caso di errore. Legge il contenuto del link e non del file cui esso si riferisce. Se la lunghezza del link simbolico è > *bufsize* viene restituito errore.

2.1.2 mkdir

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

Questa system call crea una directory i cui permessi di accesso vengono determinati da *mode* e dalla mode creation mask del processo. Restituisce 0 se OK, -1 in caso di errore. La directory creata avrà come:

- owner ID = l'effective ID del processo.
- group ID = group ID della directory padre.

La directory sarà vuota ad eccezione di . e ..

2.1.3 rmdir

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int rmdir(const char *pathname);
```

Con questa system call viene decrementato il numero di link al suo i-node. Se esso diventa uguale a 0 allora si libera la memoria solo se nessun processo ha quella directory aperta. Restituisce 0 se OK, -1 in caso di errore.

Sappiamo che dal punto di vista del sistema operativo, le directory sono dei file speciali e per questo esiste la possibilità di leggerne il contenuto. Per leggere il contenuto di una directory, esistono delle system call ad hoc:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int closedir(DIR *dp);
```

STRUCT DIRENT:

```
struct dirent {
    ino_t d_ino; /* i-node number */
    .....
    char d_name[256]; /* Null-terminated filename */
};
```

Con **opendir** si effettua l'operazione equivalente a quella della **open** su un file, però viene effettuata su una directory. Restituisce il puntatore se OK, NULL in caso di errore. Con **readdir** verrà restituita una struct dirent, OK se tutto va bene, NULL in caso di errore.

Con **rewinddir** si riparte da 0, con **closedir** si chiude la directory.

Per scandire una directory si effettua un while.

La **struct dirent** è fatta almeno da questi due parametri: un campo *d_ino* che rappresenta l'i-node number, e un'array di caratteri che contiene il nome. Se volessi leggere tutti i file contenuti in una directory devo effettuare una opendir della directory, fare un ciclo while readdir diverso da NULL e visualizzare il campo *d_name*.

2.1.4 chdir, fchdir, getcwd

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
char *getcwd(char *buf, size_t size);
```

Le prime 2 system call cambiano la cwd del processo chiamante a quella specificata come argomento. Restituiscono 0 se OK, -1 in caso di errore.

La terza system call ottiene in *buf* il path assoluto della cwd. Restituisce *buf* se OK, NULL in caso di errore.

3.0 LIBRERIE STANDARD I/O

Parliamo di libreria standard perché rientra nello standard C ed è stata implementata su molti sistemi operativi oltre che su UNIX. La modalità con cui si accede ai file è attraverso quelli che vengono chiamati **stream** (flusso di dati) e non si utilizza il File Descriptor come visto precedentemente. C'è quindi un passaggio logico da FD a Stream quando si passa da I/O di base a standard I/O.

Uno stream è un puntatore a una struttura di tipo FILE, questa struttura al suo interno contiene molte informazioni: File descriptor usato per l'I/O, Puntatore al buffer per lo stream, Dimensione del buffer, Contatore di caratteri, Etc....

Ogni processo che viene creato ha sempre 3 stream predefiniti che sono già aperti quando il nostro processo viene eseguito. Questi 3 sono:

- **stdin** che punta allo *standard input*.
- **stdout** che punta allo *standard output*.
- **stderr** che punta allo *standard error*.

Stiamo ritrovando la stessa situazione descritta per i File Descriptor, solo che in questo caso parlando di Stream. In effetti, questi 3 Stream si riferiscono esattamente ai File Descriptor descritti nei precedenti appunti.

Quello che si aggiunge quando parliamo di questo argomento, è il concetto di **Buffering**. Tutte le funzioni di standard I/O utilizzano questo concetto di immagazzinare temporaneamente le informazioni da scrivere e poi al momento giusto chiamare *read* e *write*. Le librerie standard automaticamente allocano il buffer chiamando *malloc*. Ci sono 3 tipi di buffering: **fully buffered**, **line buffered**, **unbuffered**.

3.0.1 FULLY BUFFERED

In questa modalità, le operazioni di I/O avvengono nella realtà soltanto quando il Buffer è pieno. Il termine **flush** descrive la scrittura di un buffer standard di I/O, più in particolare esso significa "writing out" il contenuto di un buffer. In generale tutti gli stream sono di questo tipo, a meno che non si riferiscono ad un terminale.

3.0.2 LINE BUFFERED

In questa modalità, le operazioni di I/O avvengono quando si incontra il carattere di *newline* sull'input o output, oppure se si riempisse il buffer. Questa modalità è usata tipicamente su stream che si riferiscono ad un terminale (standard input o output).

3.0.3 UNBUFFERED

In questo caso, la libreria standard di I/O non bufferizza i caratteri, di fatto significa che le operazioni avvengono immediatamente. Tipicamente, lo stream **standard error** utilizza questa modalità, in quanto vogliamo che un messaggio di errore venga visto immediatamente e non aspettare che sia abilitata la scrittura in un certo momento. È possibile modificare la modalità di buffering associata ad un dispositivo.

Con questa porzione di codice, innanzitutto si dichiara una stringa con un certo contenuto di 15 caratteri. Con il ciclo while, ad ogni iterazione c'è una chiamata putchar, che va a stampare o almeno ci aspettiamo che lo faccia, un carattere. Quello che accade effettivamente è che putchar viene eseguita, mentre l'operazione di I/O non viene effettuata. Questo perché associato ad uno standard output, c'è un buffer. Ogni chiamata di putchar scrive un carattere nel buffer. Nei primi 15 secondi di questo ciclo, verrà scritto un carattere per volta il testo nel buffer. Appena esce dal ciclo, inserirà nel buffer il carattere "\n", l'effetto sarà un flush di tutto il buffer. Dunque, dopo 15 secondi viene stampato in un colpo solo "Uno alla volta?".

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

int main(void)
{
    char *stringa="Uno alla volta?";
    while (*stringa) {
        putchar(*stringa++);
        sleep(1); /* fermiamo il processo per un secondo */
    }
    putchar('\n');
    sleep(4);
    return(0);
}
```

Un modo per poter realmente stampare un carattere alla volta è forzare la scrittura modificando il buffering. Se si modifica il buffer si può modificare l'output. In questo caso ci interessa un funzionamento come quello di unbuffered, coerentemente alla sua definizione.

3.1 setbuf – setvbuf

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

Restituiscono 0 se OK, un valore diverso da 0 in caso di errore.

La **setbuf**, prende come primo parametro un puntatore ad un FILE aperto. La **setvbuf** è di livello un po' più basso in quanto consente di effettuare delle operazioni più precise, il primo parametro sarà sempre un puntatore ad uno stream.

Se alla **setbuf** gli si passa un *buf* NULL lo stream passato come primo parametro diventa unbuffered. Viceversa, se come secondo parametro passiamo un buffer diverso da NULL, avviene che sarà associato al nostro stream un buffer di lunghezza ben definita e il tipo del buffering sarà fully buffered o line buffered a seconda di cosa sia il nostro buffer (automaticamente l'stdio effettua questa associazione considerando il nostro FILE di che tipo è).

Function	mode	buf	Buffer & length	Type of buffering
setbuf		nonnull	user buf of length BUFSIZ	fully buffered or line buffered
		NULL	(no buffer)	unbuffered
setvbuf	_IOFBF	nonnull	user buf of length size	fully buffered
		NULL	system buffer of appropriate length	
	_IOLBF	nonnull	user buf of length size	line buffered
	_IONBF	NULL	system buffer of appropriate length	
		(ignored)	(no buffer)	unbuffered

Con la **setvbuf** abbiamo la possibilità di cambiare il buffering, ma anche di cambiare la dimensione del buffer. Con il campo *mode* si specifica quale buffer di vuole associare al file. Se scegliamo una modalità **_IONBF**, il *buf* che passiamo viene ignorato. Se invece scegliamo una modalità **_IOFBF** e passiamo un *buf* NULL, vuol dire che la dimensione sarà quella di default scelta dallo standard I/O. Valore diverso da NULL indica che scegliamo la size.

Queste due funzioni devono essere chiamate dopo che lo stream è stato aperto, e prima di ogni altra operazione sullo stream.

Tornando all'esempio precedente:

Un **setbuf(stdout, NULL)**, ci permette di ottenere quello che volevamo.

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

int main(void)
{
    char *stringa="Uno alla volta?"; /*questa volta SI*/
    setbuf(stdout, NULL); /*stdout unbuffered*/
    while (*stringa) {
        putchar(*stringa++);
        sleep(1); /*fermiamo il processo per un secondo*/
    }
    return(0);
}
```

3.2 fflush

```
#include <stdio.h>
int fflush(FILE *fp);
```

Questa funzione forza il flushing di uno stream. Se chiamo questa funzione passandogli uno stream, l'effetto sarà di scrivere il contenuto del buffer sul file puntato da fp. Restituisce 0 se OK, EOF in caso di errore.

Se passassimo NULL a questa funzione, si effettua il flush di tutti gli stream aperti.

Se mettiamo la fflush dopo il while, verrebbe stampato in un solo colpo la stringa su standard output.

Vediamo quelle che sono le differenza tra una system call e le funzioni di libreria I/O:

- Una system call di I/O viene invocata ed immediatamente eseguita.
- L'esecuzione di una funzione di libreria di I/O passa attraverso il buffer.

```
#include <stdio.h>

int main(void)
{
    char *stringa="Uno alla volta?";
    while (*stringa) {
        putchar(*stringa++);
        sleep(1); /*fermiamo il processo per un secondo*/
    }
    fflush(stdout); /*invece di putchar('\n') */
    sleep(4);
    return(0);
}
```

3.3 APERTURA E CHIUSURA DI UNO STREAM

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type, FILE *fp);
FILE *fdopen(int fd, const char *type);
int fclose(FILE *fp);
```

La funzione **fopen**, preso un *pathname*, assocerà ad esso un puntatore a file, con il *type* specifichiamo in che modalità vogliamo aprire il file.

La funzione **freopen** apre il file *pathname* sullo stream *fp*, chiudendo questo se era già aperto.

La funzione **fdopen** prende un file descriptor (che è stato ottenuto per esempio con una open) e gli associa uno standard I/O stream.

Con queste funzioni non indichiamo mai quali sono i permessi di accesso di un file, banalmente vengono usati i permessi standard per i file che apriamo con queste funzioni. Si ricordi che nella open invece si decidono i permessi per i file.

type	Description
r or rb	open for reading
w or wb	truncate to 0 length or create for writing
a or ab	append; open for writing at end of file, or create for writing
r+ or r+b or rb+	open for reading and writing
w+ or w+b or wb+	truncate to 0 length or create for reading and writing
a+ or a+b or ab+	open or create for reading and writing at end of file

La funzione **fclose** chiude uno stream aperto. Restituisce 0 se OK, EOF in caso di errore. Ogni dato output presente nel buffer è “flushed” prima che il file sia chiuso. Quando un processo termina:

- Tutti gli stream di I/O con dati bufferizzati non scritti sono “flushed”.
- Tutti gli stream di I/O aperti sono chiusi.

3.4 LETTURA E SCRITTURA DI UNO STREAM

Una volta che uno stream è stato aperto possiamo scegliere tra:

- I/O non formattato:
 - Un carattere alla volta: **getc**, **getchar**, **putc**, **putchar** ...
 - Una linea alla volta: **fgets**, **fputs** ...
 - Diretto (I/O binario, record oriented, structure oriented): **fread** ...
- I/O formattato: **scanf**, **printf**, **fscanf**, **fprintf**

3.5 EOF

Le funzioni precedenti restituiscono EOF sia su errore che quando incontrano la fine del file. Possiamo dire che la EOF è un segnale che ci dice che c'è stato un errore di qualche tipo. In molte implementazioni sono mantenuti due flag per ogni stream: **flag di errore** e **flag di end-of-file**.

Sono flag binari, quando una funzione restituisce EOF perché si trova alla fine del file, il flag di end-of-file diventerà 1, al contrario se una funzione restituisce un errore allora il flag di errore diventerà 1. Per testare il flag settato da queste due funzioni si ricorre alle seguenti funzioni:

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
```

Si testa il flag contenuto nel flag di errore o end-of-file.

3.6 POSIZIONAMENTO IN UNO STREAM

```
#include <stdio.h>
long ftell(FILE *fp);
long fseek(FILE *fp, long offset, int whence); /* simile a lseek */
void rewind(FILE *fp);
int fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos)
int fileno(FILE *fp);
```

La funzione **ftell** restituisce l'indicatore della posizione corrente (misurato in byte), -1 in caso di errore.

La funzione **fseek** è identica alla lseek, cambia solo il primo parametro in quanto non si passa un File Descriptor ma un puntatore a file.

La funzione **rewind** riporta ci fa riposizionare all'inizio del file.

La funzione **fgetpos** pone nell'oggetto puntato da *pos* l'indicatore della posizione del file fp.

La funzione **fsetpos** prende dall'oggetto puntato da *pos* l'indicatore della posizione del file fp.

La funzione **fileno** restituisce il file descriptor associato allo stream, utile se vogliamo chiamare per esempio la **dup**.

4.0 PROCESSI IN UNIX

Ogni processo ha un **identificatore unico** non negativo. Il primo processo è quello init che ha ID uguale a 1 ed è localizzato in `/sbin/init`. Quest'ultimo è il padre di tutti i processi. Più in dettaglio viene invocato dal kernel alla fine del boot, legge il file di configurazione `/etc/inittab` dove sono elencati i file di inizializzazione del sistema (rc files) e dopo legge questi rc file portando il sistema in uno stato predefinito (multi user). Questo processo non muore mai. Ci sono delle system call su quelle che sono gli identificatori che hanno a che fare con i processi:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void); /* PROCESS ID DEL PROCESSO CHIAMANTE*/
pid_t getppid(void); /*PROCESSO ID DEL PADRE DEL PROCESSO CHIAMANTE*/
uid_t getuid(void); /*REAL USER ID DEL PROCESSO CHIAMANTE*/
uid_t geteuid(void); /*EFFECTIVE USER ID DEL PROCESSO CHIAMANTE*/
uid_t getgid(void); /*REAL GROUP ID DEL PROCESSO CHIAMANTE*/
uid_t getegid(void); /*EFFECTIVE GROUP ID DEL PROCESSO CHIAMANTE*/
```

La system call **getpid** ci restituisce il process ID del processo chiamante.

La system call **getppid** ci restituisce il process ID del padre del processo chiamante. Questa relazione padre-figlio è l'essenza fondamentale della struttura dei processi all'interno di un sistema. Ogni processo può creare un processo figlio creando un ramo.

Banalmente questo programma stampa il pid del processo corrente →

Per osservare la ramificazione dei processi in un sistema, c'è il comando **ps** che permette di vedere i processi attivi all'interno di un sistema con tutta una serie di opzioni. Senza alcuna opzione ci dice quali sono i processi che sono in stato di running nella shell attuale. Possiamo vedere che ci viene fornito il PID di ogni processo:

```
[geronimo@fujitsu ~]$ ps
  PID TTY      TIME CMD
11561 pts/0    00:00:00 bash
13487 pts/0    00:00:00 ps
```

Se vogliamo sapere qualcosa in più possiamo eseguire il seguente comando: **ps l**:

```
[geronimo@fujitsu ~]$ ps l
F  UID   PID  PPID PRI  NI   VSZ RSS WCHAN STAT TTY      TIME COMMAND
4 1000  1487  1465 200  0 157996 5660 - Ssl+ tty2    0:00 /usr/lib/gdm-x-session --run-s
4 1000  1489  1487 200  0 1415124 175096 - S+  tty2    11:22 /usr/lib/Xorg vt2 -displayfd 3
0 1000  1503  1487 200  0 257832 14028 - S+  tty2    0:00 /usr/lib/gnome-session-binary
0 1000  11561 11554 200  0 8448 5160 - Ss  pts/0    0:00 bash
4 1000  13496 11561 200  0 9652 3236 - R+  pts/0    0:00 ps l
```

Oltre PID di ogni processo c'è pure il PPID che è colui che ha generato un certo processo. Ad esempio, possiamo notare il PPID del processo **ps l** che corrisponde a 11561, suo padre, che è la **bash**, ha proprio il suo PID uguale a 11561. Con **ps lx** lista tutti i processi associati ad un real user ID.

4.1 FORK

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

La seguente funzione non prende argomenti. Restituisce un tipo *pid_t* che possiamo semplificare ad un normale valore intero. Restituisce:

- 0 nel figlio.
- **pid** del figlio nel padre.
- -1 in caso di errore.

L'effetto di una fork è quello di creare un nuovo processo (entità attiva) che avrà un suo pid e inizia la sua esecuzione nel sistema. Un nuovo processo creato all'interno di un programma utilizzerà un certo spazio di memoria, il processo figlio diventa una copia (eredita le stesse istruzioni, dati, etc.) del processo padre, non eredita il pid in quanto ogni processo ha il proprio pid. Visto che è una copia del padre, entrambi i processi avranno lo stesso program counter che sarà l'istruzione successiva a quella della fork. Il valore di ritorno della fork è duplice nel caso in cui la funzione abbia esito positivo: il valore di ritorno della fork nel processo padre sarà il pid del figlio, mentre il valore di ritorno della fork nel processo figlio sarà 0. Entrambi i processi continueranno ad eseguire le istruzioni di un certo programma. La scelta di assegnare 0 al figlio e il pid del figlio nel padre ha un senso logico. Infatti, si potrà mantenere traccia di entrambi i processi durante l'esecuzione di un programma. All'atto della creazione di un processo figlio non avviene immediatamente una copia di tutti i dati, ma alcuni vengono condivisi. Se uno dei due processi ha necessità di modificare questi dati avverrà la copia vera e propria. In generale non si sa se il figlio è eseguito prima del padre, questo dipende dall'algoritmo di scheduling usato dal kernel.

Analizziamo la seguente porzione di codice →

Fuori dal main c'è la dichiarazione di una variabile globale *glob* inizializzata a 10. All'interno viene inizializzata una variabile locale *var* uguale a 100. Viene istanziato *pippo* che è un pid. C'è una printf in cui viene stampata una sequenza di caratteri e si presume avvenga una sola volta. Alla variabile *pippo* viene eseguita la fork. Entrambi, padre e figlio, continueranno dall'istruzione successiva. Il valore di *pippo* varrà 0 per il figlio mentre per il padre sarà il pid del figlio. Tenendo a mente questa informazione andiamo dentro l'if, che viene affrontato da entrambi i processi. Per uno dei due il risultato dell'if sarà vero mentre per l'altro sarà falso. In particolare, il figlio avrà valore vero nella condizione e dunque siccome sta andando a modificare i valori delle variabili *glob* e *var* ci sarà una copia dei due valori che verranno anche incrementati. Il padre invece fallirà il controllo, dunque il processo padre rimarrà fermo per 2 secondi. In questo modo possiamo essere abbastanza sicuri che il figlio verrà eseguito prima del padre. Terminato l'if-else, il figlio stampereà immediatamente tramite printf i suoi valori pid, glob e var. Dopo circa 2 secondi ci sarà la printf fatta dal processo padre che stamparerà il suo valore pid, glob e var.

Poiché il processo figlio ha effettuato una modifica alle variabili *glob* e *var*, ha effettuato una copia delle variabili del processo padre e le ha modificate per lui (anche le variabili globali), dunque l'esecuzione di questo processo ci fornirà il seguente output:

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

int main(void)
{
    printf("pid del processo = %d\n", getpid());
    return (0);
}
```

```
[geronimo@fujitsu ~]$ ./a.out
pid del processo = 227
```

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int glob=10; /* dati inizializzati globalmente */

int main(void)
{
    int var=100; /* vbl locale */
    pid_t pippo;
    printf("prima della fork\n");
    pippo=fork();
    if (pippo == 0) {glob++; var++;}
    else sleep(2);
    printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
    exit(0);
}
```

```
$ a.out
prima della fork
pid=227, glob=11, var=101
pid=226, glob=10, var=100
```

Vediamo ora questa porzione di codice che **si differenzia dalla precedente nella printf in quanto qui non c'è lo \n** →

Notiamo che viene eseguita la *printf* ("prima della fork"), in questo caso sappiamo che non viene stampato nulla su standard output (perché non c'è il \n che svuota il buffer) e che l'informazione da stampare viene salvata nel buffer.

Successivamente viene eseguita la fork e sappiamo già come viene salvata in pippo. Il figlio incrementerà le variabili, mentre il padre aspetterà 2 secondi. Adesso il figlio effettua la *printf* (si consideri che c'è un \n che svuota il buffer), però si consideri che al momento della fork è stata effettuata una copia anche del buffer e dunque anche il processo figlio stamperà "prima della fork". L'output di questo programma sarà dunque:

```
$ a.out
prima della fork pid=227, glob=11, var=101
prima della fork pid=226, glob=10, var=100
```

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int glob=10; /* dati inizializzati globalmente */

int main(void)
{
    int var=100; /* vbl locale */
    pid_t pippo;
    printf("prima della fork");
    pippo=fork();
    if( (pippo == 0) {glob++; var++;}
       else sleep(2);
    printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
    exit(0);
}
```

Dobbiamo tenere conto di come sono gestiti i file aperti all'interno del processo. Sappiamo che esiste la tabella dei file aperti da un processo e ad ogni file è associato un puntatore al cosiddetto file table in cui ci sono tutte le informazioni del file. Attraverso la file table, al suo interno c'è un campo *v-node pointer* che punta al file fisico.

2 processi distinti hanno puntatori uguali (tranne 0 1 2 che sono gli standard che sono sempre gli stessi per tutti i processi), dunque punteranno alla stessa file table (condividono anche l'offset dunque). Se il padre prima della creazione del figlio aveva aperto un file e aveva eseguito una certa operazione, il figlio, quando verrà creato erediterà questa informazione.

Da questo capiamo che nella porzione di codice precedenti, lo standard output del figlio è copiato da quello del padre. Tutti i file aperti del padre vengono condivisi al figlio.

Quest'ultima affermazione può creare problemi di sincronizzazione: chi scrive per prima? Se non si sincronizzasse potremmo avere un output intermixed tra i due processi. Nel codice precedente abbiamo messo una pezza con una *sleep(2)*.

C'è una regola: il padre aspetta che il figlio termini o viceversa. In generale è più naturale il primo caso, infatti, se un processo decide di creare un figlio, si aspetta che il figlio faccia qualcosa e che dunque termini. Altrimenti il processo padre che motivo aveva per crearlo?

```
int main(void)
{
    pid_t pid1, pid2;
    pid1 = fork();
    pid2 = fork();
    exit(0);
}
```

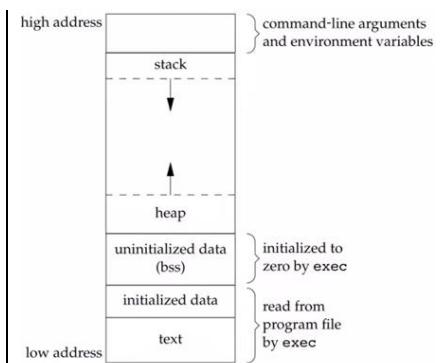
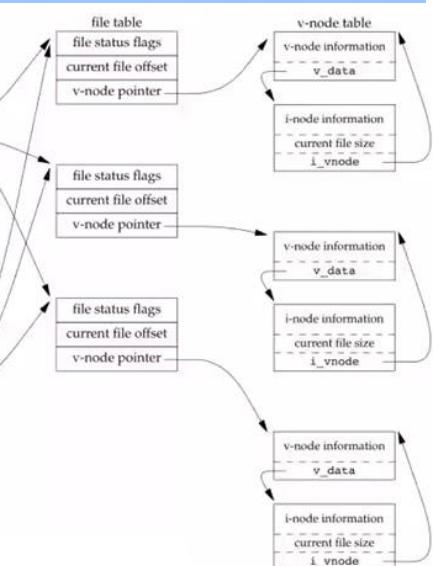
In questa porzione di codice, vengono dichiarate due variabili di tipo *pid_t* che sono *pid1* e *pid2*. Viene eseguita una prima fork e il risultato viene immagazzinato all'interno di *pid1*. In questo momento abbiamo quindi 2 processi. Questi 2 processi continuano l'esecuzione del programma, dall'istruzione successiva che è un'altra fork che sarà eseguita da entrambi. Dunque, il numero di processi che vengono generati in totale sono 4.

La fork in generale si usa quando un processo attende richieste (per esempio da parte di client), dunque decide di duplicarsi: il figlio gestisce la richiesta e il padre si mette in attesa di nuove richieste. Quando viene avviato un processo:

- Si esegue prima una routine di start-up speciale che prende i valori passati dal kernel alla linea di comando (in *argv[]* se il processo si riferisce ad un programma C) e le variabili d'ambiente.
- Successivamente viene chiamata la funzione principale da eseguire (il *main*)

Questa è una tipica occupazione di memoria di un processo. In alto c'è la zona per le variabili d'ambiente e variabili da linea di comando. Poi c'è un area condivisa da heap stack. Lo stack va verso il basso e l'heap verso l'alto in modo da avere lo stesso spazio di memoria a disposizione. Ci sono poi i dati non inizializzati, quelli inizializzati e le istruzioni di un programma.

Le variabili d'ambiente sono variabili fondamentali per il nostro processo (SHELL, PATH, USER, etc.)



4.2 EXIT

La terminazione di un processo può essere: **normale** o **anormale**. Con terminazione normale s'intende un **return** dal *main*, una chiamata a ***exit*** o ***_exit***. Con terminazione anormale s'intende una chiamata ***abort*** oppure l'arrivo di un segnale (per esempio mandare un CTRL+C).

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

La ***exit*** restituisce *status* al processo che chiama il programma includente *exit*. Effettua prima una pulizia e poi ritorna al kernel. Più in particolare viene effettuato uno shutdown delle funzioni di libreria standard di I/O (fclose di tutti gli stream lasciati aperti) e tutto l'output è flushed.

La ***_exit***, invece, si ritorna direttamente al kernel, dunque tutte le *printf* correttamente eseguite, ma senza \n, non verranno stampate.

Con questa porzione di codice, dunque, non vedremo mai alcuna stampa su standard output →

```
#include <stdio.h>

int main(void)
{
    printf("Ciao a tutti");

    _exit(0);
    exit(0);
}
```

4.3 EXIT HANDLER

```
#include <stdlib.h>
int atexit (void (*funzione) (void));
```

Questa system call prende come parametro una funzione di tipo void. Restituisce 0 se OK, diverso da 0 se errore.

Questa system call fa in modo che la funzione che gli è stata passata come parametro, venga eseguita solo all'atto di una exit. Quindi significa che si mettono da parte delle funzioni che vengono eseguite esattamente quando viene chiamata una exit, solo in quel momento. Come se programmassi un'uscita fatta in un certo modo. Si possono dichiarare più exit handlers, che vengono messi in uno stack.

In questa porzione di codice che una prima atexit che prende come parametro la funzione *my_exit2*. Successivamente ci sta un'altra atexit che prende come parametro la funzione *my_exit1*. C'è una printf che stampa una sequenza di caratteri. Al momento in cui c'è il return viene prima eseguito *my_exit1* e poi *my_exit2*.

```
$ a.out
ho finito il main
sono il primo handler
sono il secondo handler
```

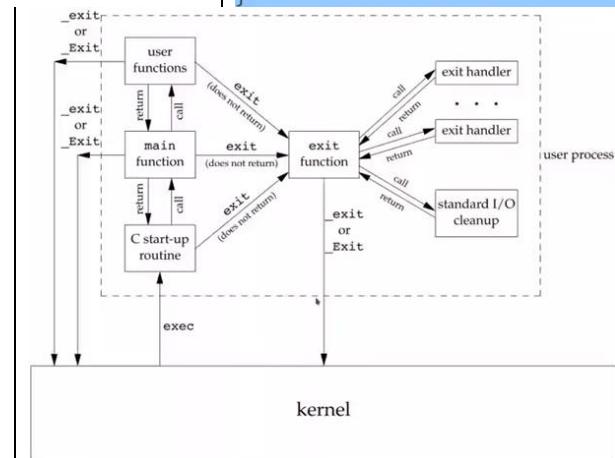
Se invece di `return(0)` avessimo messo `_exit(0)` l'output sarebbe stampato solo "ho finito il main\n" (si consideri che viene stampato perché ci sta \n che svuota il buffer).

```
int main(void)
{
    atexit(my_exit2);    atexit(my_exit1);
    printf("ho finito il main\n");
    return(0);
}

static void my_exit1(void)
{
    printf("sono il primo handler\n");
}

static void my_exit2(void)
{
    printf("sono il secondo handler\n");
}
```

Questa figura mostra realmente quello che accade quando si sta avvia, esegue e termina un processo. Si inizia il main, al suo interno possono esserci delle funzioni utente a cui si passa il controllo e successivamente si ritorna al main. Alla fine del main c'è poi una exit che chiama tutti gli exit handler che ci sono eventualmente, poi fa il cleanup di I/O. Alla fine verrà eseguita una `_exit` che passerà il controllo al kernel.



4.4 WAIT E WAITPID

Il kernel all'atto della terminazione del processo figlio invia al padre un segnale *SIGCHLD*. Il padre con questo segnale può: ignorarlo (default) o lanciare una funzione (signal handler). In più, il processo padre può ottenere una serie di informazioni sullo stato di uscita del figlio, utilizzando due funzioni:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

La `wait` restituisce il pid del processo che ha terminato la sua esecuzione e gli si può passare come parametro un puntatore ad un intero. Ogni volta che viene invocata tale funzione il processo padre si blocca in attesa che un processo figlio termini la sua esecuzione, una volta terminato, la `wait` restituisce il PID se OK, -1 in caso di errore. All'interno della variabile `statloc` è memorizzato il cosiddetto **stato di terminazione del figlio**.

La `waitpid` oltre a contenere la `statloc`, contiene altri 2 parametri che sono un `pid` e `options`. Una volta che la `waitpid` viene chiamata dal processo padre che chiede lo stato di terminazione all'interno della variabile `statloc` del figlio specificato dal valore `pid`. Il processo padre si blocca in attesa o meno a seconda del contenuto di `options`. Il valore di ritorno è il PID del processo di cui si sta aspettando la terminazione, 0 oppure -1 in caso di errore.

Vediamo quali sono i valori attribuibili a `pid`:

- `pid > 0`: si vuole aspettare la terminazione del figlio che abbiamo process ID uguale al valore di pid.
- `pid == -1`: qualsiasi figlio. La rende simile a `wait`.
- `pid == 0`: si vuole aspettare la terminazione di un figlio che abbia GroupID uguale al padre.
- `pid < 0`: si vuole aspettare la terminazione di un figlio che abbia GroupID uguale a `abs(pid)` (il valore assoluto di pid).

Il valori attribuibili a `options` sono:

- 0: non accade nulla, come nella `wait`.
- **WNOHANG**: il processo che ha chiamato la `waitpid`, non blocca se il figlio indicato non è disponibile.

La differenza tra le due funzioni è che quando si usa la `wait` l'idea è che il processo si blocca e aspetta che un figlio termini la sua esecuzione. Se qualche figlio ha terminato la propria esecuzione quando si lancia la `wait`, viene ritornato lo stato di un figlio. Nel caso in cui si lancia la `wait` e un processo non ha figli viene ritornato immediatamente un errore. Se un processo padre ha più figli, con la `waitpid` posso scegliere quale figlio aspettare (1° argomento). Attraverso il 3° parametro posso evitare che il processo padre che sta aspettando la terminazione del figlio, si blocchi.

TERMINAZIONE DI UN PROCESSO:

Quando un processo termina il kernel che sta gestendo l'esecuzione del processo determina lo stato di terminazione **normale** o **anormale**. Se è **normale**, lo stato di terminazione è l'argomento della `exit`, `return` oppure `_exit`. Se è **anormale**, il kernel genera uno stato di terminazione che indica il motivo anormale. In entrambi i casi il padre del processo ottiene questo stato da `wait` o `waitpid` dalla variabile `statloc`.

Per leggere lo stato di terminazione si usano delle MACRO, perché lo stato di terminazione è l'unione di due informazioni:

(terminazione normale/anormale) + (exit status negli 8 bit meno significativi se la terminazione è normale ||).

La MACRO **WIFEXITED(status)** darà valore vero o falso se il figlio ha terminato come effetto della exit. Nel caso restituisca vero, posso usare la seconda MACRO **WEXITSTATUS(status)** che mi restituirà lo stato di exit (il valore usato nella exit).

Nel caso **WIFEXITED(status)** dia valore falso, posso tentare con la macro **WIFSIGNALED(status)** che darà valore vero se il processo figlio ha terminato la propria esecuzione per l'arrivo di un segnale, oppure falso altrimenti. Nel caso sia vero, posso usare **WTERMSIG(status)** che restituirà il numero del segnale che ha causato l'interruzione.

Se il processo figlio è stato stoppato posso controllarlo con la macro **WIFSTOPPED(status)**, nel caso sia vero posso vedere quale dei due segnali di stop ha stoppato il processo figlio attraverso la **WSTOPSIG(status)**.

Le macro **WIFEXITED**, **WIFSIGNALED**, **WIFSTOPPED**, sono quelle che ci riconoscono se è successo una terminazione normale, anormale o stop. Le altre dicono quale segnale o valore di exit è stato mandato.

In una situazione normale, un processo padre crea un figlio, fa la **wait**, quando il processo figlio termina il processo padre continua la sua esecuzione.

In una situazione anormale, se un processo figlio termina la sua esecuzione e il padre non ha ancora invocato la **wait**, si dice che viene creato uno **zombie**: un processo che ha terminato la sua esecuzione però ci sono ancora delle informazioni nel sistema che aspettano di essere restituite al padre (come lo stato di terminazione). Questo significa che nel momento in cui il processo termina, viene associato a quel processo lo status di terminazione.

Se il padre termina senza invocare la **wait** e il figlio è ancora in stato di running, il processo diventa un **orfano**.

In questa porzione di codice, si effettua una fork (creazione di un figlio). Entrambi i processi proseguiranno dalla riga successiva. In particolare, solo per il figlio la condizione dell'if sarà vera ed effettuerà una **exit(128)**.

A questo punto, questo secondo if in linea di principio sarà eseguita da entrambi i processi, ma di fatto il figlio è uscito precedentemente. Con il comando **wait(&status)** il processo padre si mette in attesa che il processo figlio termini la sua esecuzione (con la exit precedente). Una volta sbloccata la **wait**, nel caso in cui il valore di ritorno della **wait** è uguale alla variabile **pid** allora verrà stampato che la terminazione è avvenuta in maniera normale. In linea di principio non sappiamo come il figlio abbia terminato la sua esecuzione, per stampare un valore corretto di **status** si dovrebbero usare le macro precedenti.

```
pid_t pid;
int status;

pid=fork();
if (pid==0) /* figlio */
    exit(128); /* qualsiasi numero */
if (wait(&status) == pid)
    printf("terminazione normale\n: %d", status);
```

In questo caso si effettua una fork e nel figlio avviene una **abort()** che non fa nient'altro che generare (il kernel invia al processo chiamante) un segnale di **SIGABRT**. L'effetto di questo segnale è di far terminare immediatamente in maniera anormale l'esecuzione di un process. Questo processo figlio che era stato creato, il suo stato di terminazione manderà queste 2 informazioni: terminato in maniera anormale e negli 8 bit meno significativi ci sarà il valore del segnale che ha provocato la terminazione (il numero associato a **SIGABRT**). Anche in questo caso è necessario l'uso delle macro precedenti per sapere con precisione quale sia stata la terminazione del processo figlio.

```
pid_t pid;
int status;

pid = fork();
if (pid==0) /* figlio */
    abort(); /* genera il segnale SIGABRT */
if (wait(&status) == pid)
    printf("terminazione anormale con abort\n: %d"status);
```

Si effettua una fork. Successivamente nel processo figlio si effettua una banale printf in cui viene stampato il pid del processo figlio. Successivamente una **exit(0)**, dunque terminazione del processo figlio. Il processo padre continuerà da solo l'esecuzione del codice, e farà una **sleep(2)** che ci garantisce che il figlio riesca a terminare la sua esecuzione prima che il padre si risvegli dalla **sleep**. Subito dopo viene usata una system call che permette di eseguire un comando di sistema **ps -T** che mostra tutti i processi attivi all'interno della shell. L'opzione **-T** mostra anche lo status attuale di tutti i processi visibili all'interno della shell. In particolare, se si va a vedere che il processo figlio sarà ancora visualizzato pur avendo terminato la sua esecuzione e nella colonna di stato comparirà una **STAT Z**. Quindi si vedrà che è un processo zombie. All'atto dell' **exit(0)**, il processo figlio diventerà figlio di **init** che effettuerà una chiamata **wait**.

```
int main()
{ pid_t pid;

    if ((pid=fork()) < 0)
        printf("fork error");
    else if (pid==0){ /* figlio */
        printf("pid figlio= %d",getpid());
        exit(0);
    }
    sleep(2); /* padre */
    system("ps -T"); /* dice che il figlio è zombie... STAT Z*/
    exit(0);
}
```

Ogni volta che dei processi tentano di fare qualcosa con dati condivisi e il risultato finale dipende dall'ordine in cui i processi sono eseguiti sorgono delle **race conditions**.

La funzione **charatatime** si comporta nel seguente modo: innanzitutto si setta lo standard output ad **unbuffered**, dunque, non c'è più buffering associato allo standard output. A questo punto c'è un ciclo banale che scandisce ogni lettera dalla stringa passata in output e la stampa.

Il programma, banalmente effettua una fork. Per il processo figlio viene chiamata **charatatime** con una certa stringa, mentre per il padre un'altra stringa.

In questo caso l'output in linea di principio non si sa, potremo avere un mix qualunque di caratteri. La risorsa condivisa in questo caso è lo standard output, e infatti è capitata una race conditions. Questa è una problematica che si affronta con i semafori. Nel caso di padre-figlio, la gestione della race conditions avviene attraverso uno standard.

Volendo sincronizzare un processo padre ed un processo figlio possiamo tentare di utilizzare strumenti che abbiamo già introdotto: se un processo padre vuole aspettare che un figlio termini deve usare una delle **wait**.

Lo schema generale di quanto appena detto è il seguente. Si nota come la prima cosa che il padre fa è la **wait**. Si nota come in questa **wait** non viene passato alcun parametro al suo interno.

```
int main(void){
    pid_t pid;

    pid = fork();
    if (pid==0) {charatatime("output dal figlio\n");
    else { charatatime("output dal padre\n");
    exit(0);
}

static void charatatime(char *str)
{char *ptr; int c;
    setbuf(stdout, NULL); /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
    pid = fork();
    if (!pid){ /* figlio */
        /* il figlio fa quello che deve fare */
    }
    else{ /* padre */
        wait();
        /* il padre fa quello che deve fare */
    }
}
```

Se un processo figlio vuole aspettare che il padre termini si può tentare di utilizzare i segnali.

Con questa porzione di codice, innanzitutto viene creato un processo figlio attraverso la fork. Se siamo nel figlio, ci sarà un ciclo while. Fintantoché il padre (getppid restituisce il pid del padre) è diverso da 1 allora cicla inutilmente. Per quanto riguarda il padre, continuerà la sua esecuzione e prima o poi troverà una exit. All'atto della terminazione, il processo figlio diventerà il figlio di init, dunque getpid diventerà uguale a 1, fallirà il while e il figlio potrà continuare il suo codice.

```
#include <signal.h>
void catch(int);
int main (void) {
    pid = fork();
    if (!pid){ /* figlio */
        while ( getppid() != 1 );
        /* il figlio fa quello che deve fare */
    } else{ /* padre */
        /* il padre fa quello che deve fare */
    }
}
```

4.5 EXEC

La fork di solito è usata per creare un nuovo processo (figlio). Di norma, questo processo, una volta creato, esegue un nuovo programma che può eseguire attraverso la funzione **exec**, l'effetto è che il processo figlio è rimpiazzato dal nuovo programma ed inizia l'esecuzione col suo **main**. L'unico modo per creare un processo è attraverso la **fork**, mentre l'unico modo per eseguire un eseguibile (o comando) è attraverso la **exec**. La chiamata ad **exec** reinizializza un processo: il segmento di istruzioni ed il segmento dati utente cambiano (viene eseguito un nuovo programma) mentre il segmento dei dati di sistema rimane invariato.

```
#include <unistd.h>
int execl (const char *path, const char *arg0, .../* (char *) 0 */);
int execv (const char *path, char *const argv[ ]);
int execle (const char *path, const char *arg0, .../*(char *) 0, char *const envp[ ] */);
int execve (const char *path, char *const argv[ ], char *const envp[ ]);
int execlp (const char *file, const char *arg0, .../*(char *)0 */);
int execvp (const char *file, char *const argv[ ]);
```

Tutte queste funzioni cercano di eseguire all'interno di un processo corrente un nuovo programma. Nota: l sta per list mentre v sta per vector.

La funzione **execl**, come argomenti ha un *pathname* dell'eseguibile e successivamente un numero variabile di argomenti. Eseguire la **execl** significa all'interno del processo corrente, eseguire il comando che ha *pathname* come primo argomento, seguito dagli argomenti che sono indicati come parametri successivi. Questi argomenti sono essenzialmente quelli che ritroviamo nel nostro comune **argv[]**. Fondamentalmente gli argomenti della **execl** sono l'equivalente della linea di comando che eseguiremmo normalmente. La funzione **execv** segue la stessa linea di principio.

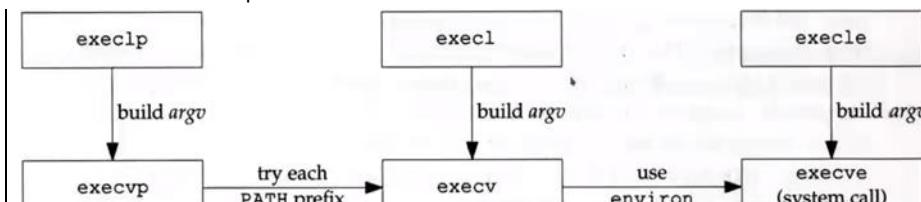
La funzione **execle** si basa sul seguente principio: se oltre al nome dell'eseguibile e agli argomenti bisogna anche passare l'environment list (variabili d'ambiente), allora si chiama questa funzione. Stesso ragionamento per **execve**.

La funzione **execlp** il primo argomento è un *filename* dell'eseguibile. In questo caso, quando si passa un nome di file di un eseguibile, quest'eseguibile sarà cercato all'interno delle variabili directory contenute nella variabile *path*. Si usa quando il comando da eseguire è un comando di sistema

Tutte queste funzioni restituiscono -1 in caso di errore, non ritornano se OK. Dal punto di vista pratico significa che quando viene chiamata una **exec** il codice chiamante non c'è più, e per questo che non ritornano niente in caso di funzione positiva.

In questa figura viene mostrato che di fatto che la vera system call che è utilizzata è la **execve**. Tutte le altre sono essenzialmente dei wrapper.

Logicamente, supponiamo di scrivere sulla shell **cat FILE1 FILE2**. Quando premiamo invio, tenendo conto che la shell è un programma, verrà individuato i pezzi del comando.



Viene effettuata una fork. Dopo la fork sarà costruita una **exec**. Se il primo argomento non inizia con un punto o con /, probabilmente sarà costruita una **execlp("cat", "cat", "FILE1", "FILE2", *0)**; a partire dalla **execlp**, una volta organizzati gli argomenti di **argv**, viene la **execvp("cat", argv)** dove argv non è nient'altro che l'organizzazione dei parametri di **execlp**. Successivamente di questa **execvp**, il kernel va a cercare all'interno della variabile *path* in quale directory si trova il comando da eseguire. Supponiamo sia "\usr\bin\cat". A questo punto viene generata la **execv("\usr\bin\cat", argv)**.

Visto che si deve usare l'environment attuale, viene invocata la **execve("\usr\bin\cat", argv, environ)**.

Con la **exec** si chiude il ciclo delle primitive di controllo dei processi UNIX:

- **fork**: creazione nuovi processi.
- **exec**: esecuzione nuovi programmi.
- **exit**: trattamento fine processo.
- **wait/waitpid**: trattamento attesa fine processo.

Questo è un esempio estremamente atipico di una **exec**. Con la **exec** viene sostituito il codice visibile in figura e ci sarà il codice di **echo** che verrà caricato al posto di quello in figura. In più, questo programma avrà la sua lista degli argomenti in cui c'è **argv[0]=echo**, **argv[1] = "la"** e così via...

echo permette di stampare in sequenza i vettori che gli si passano come argomento. Alla fine, c'è un **exit(0)** che non esiste perché tutto quello che c'è dopo l'**exec** non verrà mai eseguito (in caso positivo della exec ovviamente).

L'output di questo programma sarà:

Sopra la panca
la capra campa

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

int main (void) {
    printf("Sopra la panca \n");
    execl("/bin/echo", "echo", "la", "capra", "campa", NULL);
    exit(0);
}
```

In questo programma l'output dipende dalla positività della **exec**. Se viene eseguita correttamente l'output sarà uguale a quello precedente.
Se la **exec** fallisce l'output sarà quello precedente con l'aggiunta di "sotto la panca crepa"

Sopra la panca
la capra campa

```
$a.out
Sopra la panca
la capra campa
sotto la panca crepa
$
```

```
$a.out
Sopra la panca
sotto la panca crepa
la capra campa
$
```

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

int main (void) {
printf("Sopra la panca \n");
execl("/bin/echo","echo","la","capra","campa",NULL);
printf("sotto la panca crepa \n");
exit(0);
}
```

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

int main (void) {
printf("Sopra la panca \n");
pid = fork();
if (pid==0){
    execl("/bin/echo","echo","la","capra","campa",NULL);
printf("sotto la panca crepa \n");
exit(0);
}
```

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

int main (void) {
printf("Sopra la panca \n");
pid = fork();
if (!pid){
    execl("/bin/echo","echo","la","capra","campa",NULL);
wait( );
printf("sotto la panca crepa \n");
exit(0);
}
```

In questo caso viene eliminata la race conditions che poteva esserci precedentemente.

```
$a.out
Sopra la panca
la capra campa
sotto la panca crepa
$
```

5. SEGNALI

Un segnale è un interrupt software e può essere generato da un processo utente o dal kernel a seguito di un errore software o hardware.

Ogni segnale ha un nome che comincia con SIG (ex. SIGABRT, SIGALARM) a cui viene associato una costante intera ($\neq 0$) positiva definita in **signal.h**.

Il segnale è un evento asincrono, esso può arrivare in un momento qualunque ad un processo ed il processo può limitarsi a verificare, per esempio, il valore di una variabile, o può fare cose più specifiche.

Quando arriva un segnale da un processo possono accadere 3 cose:

1. **Ignorare il segnale** (tranne che per SIGKILL e SIGSTOP);
2. **Catturare il segnale** (equivale ad associare una funzione utente quando il segnale occorre; ex. se il segnale SIGTERM è catturato possiamo voler ripulire tutti i file temporanei generati dal processo);
3. **Eseguire l'azione di default associata** (terminazione del processo per la maggior parte dei segnali).

5.1 SIGNAL

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

Restituisce SIG_ERR in caso di errore e il puntatore al precedente gestore del segnale se OK.

Prende due argomenti: il nome del segnale **signo** ed il puntatore alla funzione **func** da eseguire all'arrivo di signo (signal handler) e restituisce il puntatore ad una funzione che prende un intero e non restituisce niente, che rappresenta il puntatore al precedente signal handler.

Il valore di **func** può essere:

- **SIG_IGN** per ignorare il segnale (tranne che per SIGKILL e SIGSTOP);
- **SIG_DFL** per settare l'azione associata al suo default;
- L'indirizzo di una funzione che sarà eseguita quando il segnale occorre.

L'esecuzione di un programma tramite **fork+exec** ha le seguenti caratteristiche

- Se un segnale è ignorato nel processo padre viene ignorato anche nel processo figlio;
- Se un segnale è catturato nel processo padre viene assegnata l'azione di default nel processo figlio.

5.2 KILL E RAISE

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int signo);
int raise (int signo);
```

Mandano il segnale signo specificato come argomento e restituiscono 0 se OK, -1 in caso di errore.

Raise: consente ad un processo di mandare un segnale a sé stesso.

Kill: manda un segnale ad un processo o ad un gruppo di processi specificato da pid, quest'ultimo può avere valori:

- pid > 0 invia al processo pid;
- pid == 0 invia ai processi con lo stesso gid del processo sender;
- pid < 0 invia ai processi con gid uguale a |pid|.

5.3 ALARM

```
#include <unistd.h>
unsigned int alarm (unsigned int secs);
```

Invia al processo corrente il segnale SIGALRM (uccide il processo) dopo che siano trascorsi secs secondi e restituisce 0 o il numero di secondi rimasti da una precedente alarm.

5.4 PAUSE

```
#include <unistd.h>
int pause(void);
```

Sospende il processo finché non arriva un segnale, il corrispondente signal handler viene eseguito ed esce, e restituisce -1.

5.5 SLEEP

```
#include <unistd.h>
unsigned int sleep (unsigned int secs);
```

Restituisce 0 oppure, nel caso la funzione sia risvegliata da un segnale, il numero di secondi rimasti.

Come prima cosa nel main è la chiamata alla signal, proprio perché se si vuole gestire l'arrivo di un segnale con una funzione bisogna farlo il prima possibile perché non si sa quando arriverà tale segnale.

Ora se all'interno di tale processo arriverà uno dei due segnali esplicitati, allora verrà eseguita la funzione specificata.

```
[geronimo@fujitsu SO]$ ./sign1 &
[1] 73043
[geronimo@fujitsu SO]$ ps
 PID TTY      TIME CMD
 71359 pts/0    00:00:00 bash
 73043 pts/0    00:00:00 sign1
 73066 pts/0    00:00:00 ps
[geronimo@fujitsu SO]$ kill -USR1 73043
SIGUSR1
[geronimo@fujitsu SO]$ kill -USR2 73043
SIGUSR2
[geronimo@fujitsu SO]$ kill -2 73043
[geronimo@fujitsu SO]$ ps
 PID TTY      TIME CMD
 71359 pts/0    00:00:00 bash
 73225 pts/0    00:00:00 ps
[1]+  Interruzione          ./sign1
```

```
#include <signal.h>
void sig_usr(int);
int main (void) {
    signal(SIGUSR1, sig_usr);
    signal(SIGUSR2, sig_usr);
    for( ; ) pause();
}

void sig_usr(int signo) {
    if(signo == SIGUSR1) printf("SIGUSR1\n");
    else if (signo == SIGUSR2) printf("SIGUSR2\n");
    else printf("Segnale %d\n", signo);
}
```

6. PIPE

La comunicazione tra processi può avvenire:

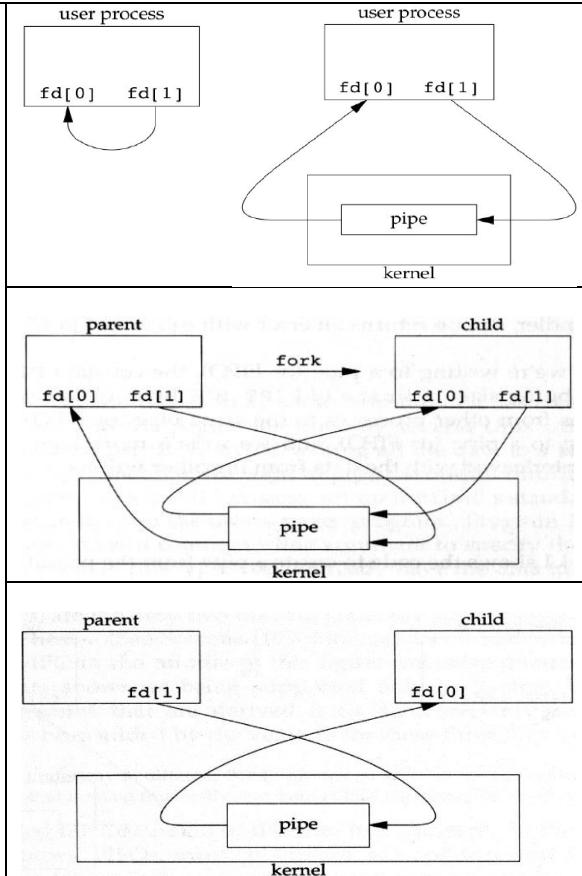
- Passando dei files aperti tramite fork
- Attraverso il filesystem
- Utilizzando le **pipe**
- Utilizzando le **FIFO**
- Utilizzando **IPC di System V**
- Utilizzando **stream e socket**

Le **PIPE** sono **half-duplex**, ovvero il flusso di dati è in una sola direzione. Possono essere utilizzate solo tra processi che hanno un antenato in comune.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Restituisce 0 se OK, -1 in caso di errore.

- **filedes[0]** è il file descriptor di un "file" aperto in **lettura**
- **filedes[1]** è il file descriptor di un "file" aperto in **scrittura**
- inoltre, l'output di **filedes[1]** corrisponde all'input di **filedes[0]**



L'utilizzo tipico delle pipe è il seguente:

```
int fd[2];
...
pipe(fd);
pid=fork();
...
}
```

NOTA: il figlio in tal caso eredita dal padre tutto il codice, ma in particolare eredita anche la tabella dei file aperti, quindi sia padre che figlio avranno nella tabella dei file aperti, oltre a STDIN/OUT/ERR, anche 2 file descriptor, uno in cui possono scrivere e uno leggere.

Una delle possibilità dopo la fork è la seguente:

```
if(pid>0) {           // padre
    close(fd[0]);
} else
    if(pid==0) {        // figlio
        close(fd[1]);
    }
```

Questo crea un canale dal padre verso il figlio.

NOTA: in questo modo abbiamo creato un canale dal padre verso il figlio.

6.1 I/O SU PIPE

Una volta che è stata creata la pipe e che è stato scelto il verso di comunicazione è possibile utilizzare le funzioni di I/O che lavorano con i file descriptor (*tranne open, creat e lseek*). Una pipe è un canale di comunicazione in cui i dati vengono letti nello stesso ordine in cui vengono scritti, ovvero in ordine FIFO (non si parla più di offset). La semantica di read e write è leggermente modificata.

FUNZIONE WRITE:

Quando la pipe si riempie (la costante **PIPE_BUF** specifica la dimensione, questa non è modificabile), la write si blocca fino a che la read non ha rimosso un numero sufficiente di dati. La scrittura è atomica se i dati sono \leq PIPE_BUF.

Se il descrittore del file che legge dalla pipe è chiuso, una write genererà un errore (segnale **SIGPIPE**).

FUNZIONE READ:

Legge i dati dalla pipe nell'ordine in cui sono scritti. Non è possibile rileggere o rimandare indietro i dati letti. Se la pipe è vuota la read si blocca fino a che non vi siano dati disponibili. Se il descrittore del file in scrittura è chiuso, la read restituirà un errore dopo aver completato la lettura dei dati.

FUNZIONE CLOSE:

La funzione close sul descrittore del file in scrittura agisce come end-of-file per la read.

La chiusura del descrittore del file in lettura causa un errore nella write.

```
...
int fd[2];
pipe(fd);
pid=fork();
if(pid>0) {           /* padre */
    close(fd[0]);
    write(fd[1],"ciao figlio\n",12);
} else{                /* figlio */
    close(fd[1]);
    n=read(fd[0], line, 12);
    write(STDOUT_FILENO, line, n);
}
...
```

NOTA.1: bisogna decidere a priori il tipo di dato, o struttura dati, usato tra i due processi durante la comunicazione.

NOTA.2: supponiamo per ora che il figlio, una volta che viene fatta la read ma non ha nulla da leggere e in questo caso si blocca, dando vita ad una sorta di sincronizzazione.

NOTA.3: se si vuole una comunicazione bidirezionale, bisogna creare 2 pipe, tale operazione va fatta prima di richiamare le **fork()**.

NOTA.4: se si omettono le **close()**, un processo non saprà mai che l'altro processo ha terminato la scrittura/lettura perché si avrà sempre il fd riferito in entrambi, per questo è meglio chiudere ciò che non servirà.