

Gestione di periferici e file

Dispositivi di I/O

- Gestore di sottosistemi di I/O: esiste una differenza concettuale tra i compiti di un gestore di CPU o memoria e un gestore di un sottosistema di I/O.
- I gestori di memoria e CPU servono per virtualizzare la risorsa, mentre un gestore di I/O deve soltanto preoccuparsi di assegnare la risorsa.
 - Un processo per tutta la sua vita avrà bisogno di CPU e memoria, ma solo saltuariamente avrà bisogno di dispositivi di I/O.

Gestore di I/O

- Ad un processo si assegnano sempre e comunque CPU e memoria, mentre non gli si assegna un dispositivo di I/O se non ne fa esplicita richiesta.
- Strategie di allocazione e revoca dei dispositivi diverse a seconda del tipo.
- In genere un dispositivo di I/O è considerato come risorsa nonpreemptable, cioè viene assegnato al processo finché questi non ha terminato.
 - minore flessibilità.

Gestori di I/O

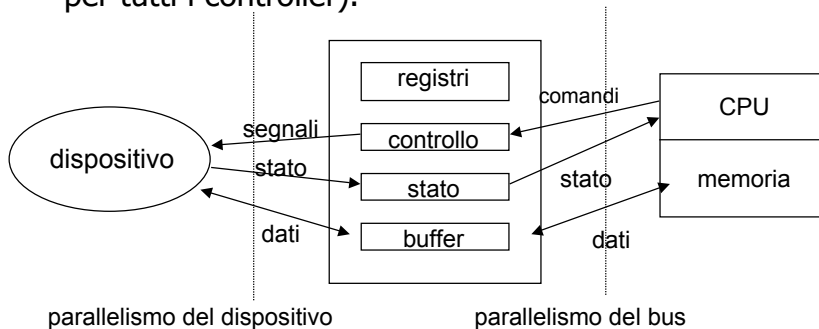
- Devono mascherare all'utente i problemi HW legati al dispositivo.
- Devono appiattire il più possibile le differenze tra i dispositivi (es. floppy e hard disk).
- Devono gestire le interruzioni provenienti dai dispositivi.
- Devono essere in grado di gestire errori e/o malfunzionamenti.
- Devono fornire un insieme di comandi per poter operare in modo efficace con i periferici.

Architettura HW di riferimento

- Dispositivi a blocchi:
 - servono per memorizzare informazioni in blocchi di dimensione fisica specificata: da 128 a 2kB. Ogni blocco può essere riferito indipendentemente dagli altri blocchi (accesso casuale).
- Dispositivi a carattere:
 - Inviano o accettano stringhe di caratteri. Il flusso di byte non può essere alterato nell'ordine.
- Dispositivi speciali:
 - Ad esempio il clock. Non è caratterizzato da blocchi indirizzabili e non può accettare o mandare stringhe di caratteri; può essere programmato per generare interruzioni.

Controller

- Doppia interfaccia:
 - dispositivo controller (a parallelismo proprio del dispositivo).
 - controller bus (a parallelismo proprio del bus -> uguale per tutti i controller).



Controller

- La CPU scrive nel registro di controllo:
 - nel registro di controllo c'è un bit per abilitare le interruzioni (quando il dispositivo ha finito manda un'interruzione).
 - I registri possono essere visti memory mapped o I/O mapped.
- Il dispositivo esegue il compito e scrive nel registro di stato delle informazioni riguardanti l'operazione che ha eseguito.
 - nel registro di stato vi sono due bit: uno viene automaticamente settato quando il dispositivo ha terminato e l'altro in caso di errore.
- La CPU legge il registro di stato.

Architettura SW di riferimento

- Il sottosistema SW è costituito da componenti organizzati gerarchicamente.
 - livello utente
 - device independent sw
 - device driver.
 - gestori interrupt

Le peculiarità dei dispositivi sono confinate in questi due livelli



livello utente
device independent sw
device driver
gestori interrupt

Architettura SW di riferimento

```
processo esterno /*I/O*/
{
    while(1)
    {
        <attesa invio del comando>
        <esegue comando>
        <verifica l'esito del comando>
        bit_di_flag=1;
    }
}
processo interno
{
    while(1)
    {
        <prepara il comando>
        <invia il comando>
        do{
        }while(flag==0);
        <verifica l'esito>
    }
}
```

Architettura SW di riferimento

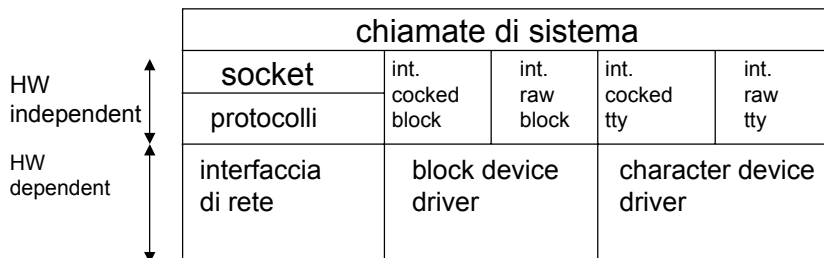
```
struct semaforo fine={0};
processo interno
{
    do{
        <prepara il comando>
        <invia il comando>
        wait(&fine);
        <verifica l'esito>
        }while(trasf_non_completato);
    }
}
processo esterno
{
    while(1){
        <attesa invio del comando>
        <esegue il comando>
        <registra l'esito del comando>
        bit_di_flag=1; /*interruzione*/
    }
}
interrupt handler /*livello kernel*/
{
    <salvataggio stato>
    signal(&fine);
    <ripristino stato>
    <ritorno da interruzione>
}
```

Device driver

- Il device driver è un processo di sistema attivato via interrupt e non dallo scheduler tradizionale.
- Conviene vedere il device driver come processo di sistema e non come parte integrante del kernel.
- Compiti del device driver:
 - invia i comandi al dispositivo
 - riceve le interruzioni
 - gestisce gli errori
 - gestisce la competizione
 - sincronizza processo applicativo (utente) e dispositivi mediante bufferizzazione.

I/O in Unix

- In Unix i device driver non sono processi di sistema ma particolari routine del nucleo. Più efficiente ma meno modulare.
- Due interfacce tra chiamate di sistema e i device driver:
 - raw: trasferisce le stringhe di caratteri così come sono.
 - cocked: (più raffinata) permette l'uso del backspace.



File system

- Necessità di memorizzare quantità di informazioni che non possono essere tenute in memoria centrale e che devono sopravvivere al processo che le ha generate.
 - file: contenitore di informazioni di memoria di massa.
- Caratteristiche di un file system:
 - permette di identificare univocamente un file
 - fornisce metodi per accedere a blocchi elementari di informazioni in un file.
 - maschera le caratteristiche fisiche dei dispositivi.
 - realizza meccanismi di controllo.
 - garantisce la consistenza delle informazioni.

Realizzazione del file system

- struttura fisica del disco:
 - disk drive: parte meccanica.
 - disk controller: parte elettronica.
 - settore (blocco fisico): unità di informazione che può essere letta e/o scritta (da 32 a 4096 byte).
 - traccia (da 4 a 32 settori)
 - superficie (da 20 a 1500 tracce)
 - per indirizzare un settore: seek time per raggiungere la traccia e latency time per portare il settore sotto la testina.
 - indirizzo visto come terna di numeri: numero di cilindro, numero di traccia nel cilindro e numero di settore all'interno della traccia.

Gestione dello spazio libero

- Bit map: ogni blocco è rappresentato da un bit
 - facile trovare sequenze consecutivi di blocchi liberi.
 - la bit map deve essere tenuta in memoria per motivi di efficienza.
- Lista concatenata di blocchi: ogni blocco contiene la quantità massima possibile di numeri di blocchi di disco liberi
 - non è efficiente perché devo scandire i blocchi in sequenza e non riesco facilmente ad individuare sequenze di blocchi liberi

Metodi di allocazione

- Un file è memorizzato in blocchi logici di dimensione finita. Ogni blocco logico può contenere 1,2, blocchi fisici.
 - concetto di cluster
 - ogni operazione (lettura o scrittura) riguarda blocchi logici.
- Allocazione contigua
 - pregi: riduco il tempo di seek, posso avere accessi sia sequenziali sia diretti.
 - difetti: frammentazione esterna, devono conoscere a priori la dimensione massima del file.

Metodi di allocazione

- Allocazione a lista: in ogni blocco ho l'indirizzo del blocco successivo.
 - pregi: non ho frammentazione esterna e posso allocare dinamicamente blocchi man mano che il file cresce.
 - difetti: impossibile l'accesso diretto, e se perdo un blocco della lista, perdo tutto il file.
- Allocazione ad indice: in un blocco di indice ho l'elenco dei blocchi, in ordine, che costituiscono il file
 - dimensione massima di un file = al numero di indirizzi che possono stare in un blocco (risolvo con l'indirizzamento indiretto).
 - risolvo il problema della frammentazione e dell'accesso diretto.

Metodi di accesso

- Metodo di accesso "byte stream":
 - modello del file: sequenza di byte.
 - le operazioni di lettura e scrittura specificano il numero di byte che devono essere letti o scritti.
 - puntatore alla posizione corrente (viene incrementato del numero di byte letti o scritti).
 - Conoscendo: l'indirizzo fisico del primo blocco del file, la dimensione del blocco e la posizione corrente all'interno del file, è possibile risalire al blocco che si deve leggere o scrivere.
- Metodo di accesso append:
 - consente di aggiungere byte alla fine di un file.



Metodi di accesso

- Metodo di accesso seek:
 - posiziona il puntatore alla posizione corrente in un ben preciso punto del file.
 - si usa per effettuare letture o scritture di byte non consecutivi.
- Organizzazione a record: il file è visto come un insieme di record, suddivisi in campi e identificati da un nome simbolico.
 - i record possono essere a lunghezza fissa o a lunghezza variabile.
- Tre metodi di accesso ad alto livello fondamentali:
 - sequenziale
 - diretto
 - a chiave



Metodi di accesso

- Accesso sequenziale:
 - Il file è visto come una successione di record cui è associata una posizione corrente.
 - il file sequenziale è un'astrazione che modella il comportamento di periferiche di tipo nastro.
- Accesso diretto:
 - il file è visto come un array di record.
 - il file ad accesso diretto modella il comportamento di periferiche ad accesso casuale (disco).
- Accesso a chiave:
 - è visto logicamente come una collezione di record all'interno dei quali esiste un campo chiave.
 - il record viene identificato specificando il nome del campo chiave
 - possono esistere più campi chiave (chiave primaria , chiave secondaria).

Metodi di accesso

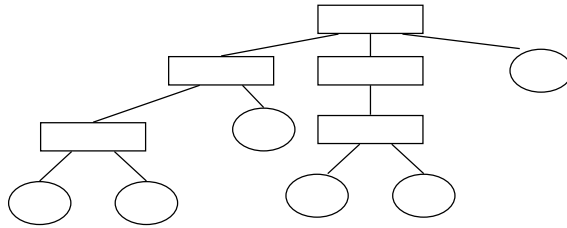
- Ad una chiave primaria può corrispondere solo un record, mentre ad una chiave secondaria possono corrispondere più record.
- File ad indice (indexed) è usato come sinonimo di file a chiave.
- File sequenziale con indice è un file su cui si possono eseguire accessi a chiave ed in più vi è associato il concetto di posizione corrente.

Proprietà dei file

- **Descrittore di file:**
 - corrispondenza tra nome logico e indirizzo fisico.
 - contiene gli attributi del file.
 - un insieme di descrittori è un direttorio.
- **Attributi:**
 - protezione -lunghezza record
 - parola chiave -pos. chiav. nel rec.
 - creatore -data creazione
 - proprietario -data ultima modif.
 - read only flag -numero di byte
 - ASCII o BIN -dimensione max
 - lock flag -versione (VMS)

Organizzazione del file system

- Un direttorio è visto come un file, identificato da un nome simbolico.
- Organizzazione gerarchica ad albero:
 - tutti i nodi dell'albero sono direttori e le foglie i file veri e propri.
- Identificazione univoca di un file = pathname.



File system di Unix

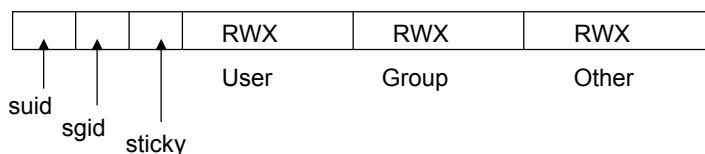
- Non esistono limiti nel livello di annidamento dei direttori.
- Alcuni direttori predefiniti hanno delle funzioni specifiche indipendentemente della versione (etc, usr, etc.)
- Caratteristica di omogeneità: tutte le risorse del sistema sono viste come file:
 - file ordinari
 - direttori
 - file speciali (dispositivi fisici).
- Caratteristica di trasparenza: ogni utente può accedere ad un dispositivo allo stesso modo in cui accede a un file.

File system di Unix

- Un riferimento ad un file può essere fatto in due modi:
 - riferimento relativo (riferito al direttorio corrente).
 - riferimento assoluto (riferito al direttorio radice).
- Ad ogni file è associato un solo descrittore (i-node), identificato da un indice di tabella (i-number).
- Ad ogni file possono essere associati più nomi simbolici (linking)
- Convenzioni:
 - "." direttorio corrente
 - ".." direttorio padre.

Protezioni

- Multiutenza:
 - username e password
 - classificazione degli utenti in gruppi.
 - Esempio: username anna [user_id:1234], group [group_id:22]
- Ad ogni file è associato il nome dell'utente che lo ha creato e quello del gruppo.
- Tre modalità di accesso:
 - lettura, scrittura ed esecuzione.
- 12 bit di protezione per i file:

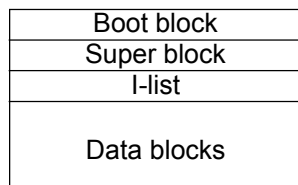


Protezioni

- I tre bit più significativi hanno senso solo per file eseguibili:
 - suid: set user id
 - sgid: set group id
 - sticky: save text image bit
- File eseguibili che contengono codice che può essere eseguito, generando un processo a cui viene assegnato dinamicamente l'uid e il gid del proprietario del file (es: posso essere temporaneamente root).
- Normalmente quando si lancia un eseguibile, il processo che si genera ha uid e gid dell'utente che lancia il file.
- sticky: l'immagine del processo rimane in memoria anche dopo la sua terminazione (es: comandi più frequenti).

File system di Unix

- Metodo di allocazione ad indice
- Blocco: unità elementare di allocazione (da 512 a 4096 byte).
- Partizionamento del disco in blocchi.
- Realizzazione del file system attraverso la suddivisione del disco in 4 regioni:
 - Boot Block
 - Super Block
 - I-list
 - Data blocks.



File system di Unix

- Boot block: contiene le procedure di inizializzazione del sistema.
- Super block contiene:
 - i limiti delle 4 regioni.
 - il puntatore a una lista dei blocchi liberi.
 - il puntatore a una lista degli i-node liberi
- Data blocks: area effettivamente disponibile per la memorizzazione dei file.
- I-list: contiene la lista di tutti gli i-node del file system (accesso con l'indice i-number).

I-node

- I-node: è il descrittore del file.
- Attributi contenuti nell'i-node:
 - tipo del file (ordinario, direttorio, speciale).
 - proprietario, gruppo.
 - dimensione.
 - data dell'ultima modifica.
 - 12 bit di protezione.
 - numero di links.
 - 13 indirizzi di blocchi.

Indirizzamento

- L'allocazione del file non è fisicamente su blocchi contigui.
- 13 indirizzi di cui:
 - 10 indirizzi di blocchi di dati (0-9).
 - 11° indirizzo contenente l'indirizzo di un blocco dove sono registrati gli indirizzi di blocchi di dati.
 - 12° indirizzo (secondo livello di indirettezza).
 - 13° indirizzo (tre livelli di indirettezza).
- Se la dimensione di un blocco è di 512 byte, con indirizzi su 4 byte, → un blocco contiene 128 indirizzi.
- Capacità di indirizzamento:
 $10+128+128*128+128*128*128$
- Dimensione massima di un file dell'ordine del Gbyte.

Gestione dei file

- System call creat:
 - cerca spazio sul disco.
 - inserisce il descrittore nel direttorio identificato dal pathname del file.
 - inserisce nel descrittore l'indirizzo fisico, gli attributi e il metodo di accesso.
- system call delete:
 - rilascia lo spazio su disco.
 - cancella il descrittore dal direttorio.
- System call open:
 - porta in memoria il descrittore del file in un'apposita tabella.
 - restituisce l'identificatore di file, che corrisponde alla posizione del descrittore nella tabella.



Gestione dei file

- System call close:
 - il descrittore del file, eventualmente modificato, è ricopiato su disco.
- System call get attributes:
 - fornisce gli attributi del file.
- System call set attributes:
 - modifica gli attributi del file.
- System call rename:
 - cambia il nome del file.
- System call read e write:
 - dipendono dal metodo di accesso.
 - Informazioni necessarie: identificatore del file, area di memoria dove inserire o prelevare i dati, identificatore del blocco interessato al trasferimento (dipende dal metodo di accesso).



Gestione dei file in Unix

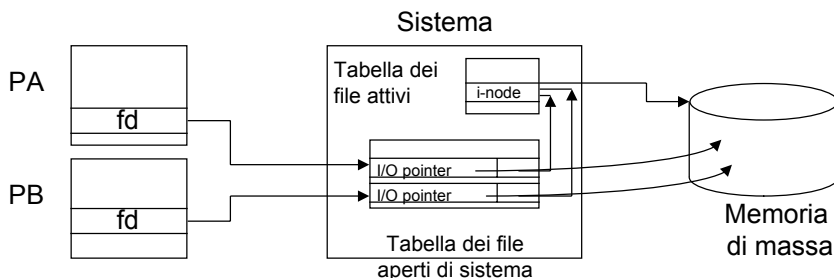
- Assenza di strutturazione: file = sequenza di byte
- I/O pointer : posizione corrente all'interno del file.
- Accesso:
 - sequenziale
 - diretto
- Varie modalità di accesso: lettura, scrittura, lettura/scrittura, etc.
- Accesso subordinato all'operazione di apertura.

File descriptor

- Ogni processo gestisce una tabella dei file aperti (generalmente max 20).
- Ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero (file descriptor).
- 0, 1, 2 individuano rispettivamente standard input, output error (aperti automaticamente alla creazione di un processo).
- la tabella dei file aperti di un processo è allocata nella user kernel area.

Gestione dei file

- Nella tabella dei file aperti vi è un elemento per ogni "apertura":
 - a processi diversi che accedono allo stesso file, corrispondono entry distinte.
- Ogni elemento contiene il puntatore alla posizione corrente (I/O pointer):
 - più processi possono accedere contemporaneamente allo stesso file, ma con I/O pointer diversi.
- L'operazione di apertura provoca la copia dell'i-node in memoria centrale
- La tabella dei file attivi contiene gli i-node di tutti i file aperti.
- Il numero dei record è pari al numero di file aperti.



System call: open e creat

- Apertura: creat, open.
- L'apertura di un file provoca:
 - l'allocazione di un elemento nella prima posizione libera dalla tabella dei file aperti del processo.
 - l'inserimento di un nuovo record nella tabella dei file aperti di sistema.
 - la copia dell'i-node nella tabella dei file attivi (se il file non è già stato aperto da un altro processo).
 - Apertura di file già esistenti:
int fd, flags;
char *nomefile;
fd=open(nomefile,flags);
 - flags esprimono le modalità di accesso: O_RDONLY, O_WRONLY, O_RDWR etc. definite in fcntl.h.
 - fd = file descriptor.

System call: open e creat

- Apertura di nuovi file:
 - fd=creat(nomefile,perm);
 - perm rappresenta i 12 bit di protezione.
 - se un file esiste già viene sovrascritto.
- Esempio:

```
#include <fcntl.h>
#define PERM 0744
main()
{
    int fd;

    if(fd=open("/home/miofile",O_RDONLY)<0)
    {
        printf("Il file non esiste\n");
        fd=creat("/home/miofile",PERM);
    }
    /*Accesso a file*/
    ...
}
```

System call: read

- Lettura e scrittura di file
- Accesso attraverso la specifica del file descriptor.
- Ogni operazione di lettura o scrittura agisce sequenzialmente sul file, a partire dalla posizione corrente del puntatore (I/O pointer).
- Possibilità di alternare operazioni di lettura e scrittura.
- Atomicità delle singole operazioni.
- Read:
int fd,n,let;
char *buf;
let=read(fd,buf,n);
- buf: area in cui trasferire i byte letti.
- let: numero di byte effettivamente letti.
- È previsto un EOF (^D)

System call: write

- Write:
int fd,n,scritti;
char *buf;
scritti=write(fd,buf,n);
- se scritti è < n → errore
- Esempio:
#include <fcntl.h>
main()
{
int fd,n;
char buf[10];
if
(fd=open("/home/miofile",O_RDONLY)<0)
{
printf("Il file non esiste\n"),
exit(-1);
}
while ((n=read(fd,buf,10))>0)
write(1,buf,10);
/*scrittura su stand. output */
.....
}

System call: lseek

- Accesso diretto: per spostare l'I/O pointer:
 - `int fd,offset,origine,dest;`
 - `dest=lseek(fd,offset,origine);`
 - `offset` = spostamento in byte dall'origine.
 - `origine`: 0 inizio file, 1 posizione corr., 2 fine file.
 - Chiusura file:
 - `int fd,rit;`
 - `rit=close(fd);`
 - `rit` = risultato dell'operazione.
- Esempio:
- ```
#include<fcntl.h>
main()
{
 int fd,n;
 char buf[10];

 if(fd=open("/home/miofile",O_RDWR)<0)
 {
 ...
 }
 lseek(fd,-3,2);
 ...
 close(fd);
}
```

## System call: pipe

- Pipe:
  - `int fd[2],retval;`
  - `retval=pipe(fd);`
- `fd` vettore di due elementi interi, `fd[0]` identifica il file aperto in lettura, `fd[1]` in scrittura.
- Comunicazione con pipe nell'ambito della gerarchia di processi.
- Strumenti di lettura e scrittura `read()` e `write()`.

## System call: pipe

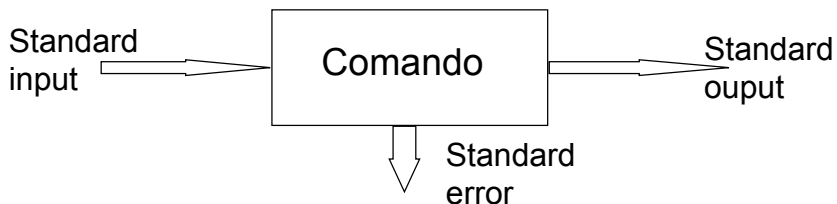
```
■ Esempio:
#include<stdio.h>
char *buf="Ciao";
main()
{
 int fd[2], rit, pid, n;
 rit=pipe(fd);
 if((pid=fork())!=0) { /*padre*/
 close(fd[0]);
 write(fd[1],buf,5);
 close(fd[1]); }
 else{
 close(fd[1]);
 read(fd[0],buf,5);
 printf("Ho letto %s dalla pipe \n",buf);
 close(fd(0));}
}
```

## System call: link e unlink

- Link:  
char \*oldpath, \*newpath;  
int rit;  
rit=link(oldpath,newpath);
- oldpath è un nome simbolico del file (preesistente);
- newpath è il nuovo nome che si desidera assegnare al file.
- rit valore restituito (-1 errore).
- Unlink: decrementa il numero di link dell'i-node; se è 0 provoca la cancellazione del file.  
char \*path;  
int rit;  
rit=unlink(path);

# Comandi

- Sintassi del tipo:
  - nome comando [opzioni] [argomenti].
  - esempio: `ls -la /usr/anna`
- Comandi di interazione con il file system:
  - gestione dei file e dei direttori.
- Comandi di gestione del sistema:
  - informazioni sulle risorse
  - modifica di dati di sistema



# Redirezione e piping

- Comandi filtro:
  - elaborano dati dallo standard input
  - producono risultati sullo standard output.
  - `more`, `sort`, `grep`, `head`.
- È possibile deviare standard input, output e error di un comando su file usando i caratteri `<`, `>`, `>>`:
  - `<` redirezione dello standard input
  - `>` redirezione dello standard output
  - `>>` redirezione in append dello standard output
  - `2>` redirezione dello standard error.
- È possibile deviare lo standard output di un comando nello standard input di un altro con il comando:
  - `comando1 | comando2`
  - i due comandi vengono eseguiti in parallelo.

## Esecuzione dei comandi

- Per ogni comando da eseguire la shell crea una shell figlio, dedicato all'esecuzione del comando.
- Esecuzione in foreground: il padre si sospende in attesa della terminazione del figlio.
- Esecuzione in background: il padre continua la propria esecuzione in parallelo con il figlio.

## Esecuzione comandi: schema implementativo

```
do{
 <accettazione comando>
 <interpretazione comando>
 pid=fork();
 if (pid==0) { /*figlio*/
 execl(<comando>);
 exit(-1);
 }
 else /*padre */
 if (<foreground>))
 wait (&status);
}while(!EOF) /*fine ciclo*/
```

## Redirezione: schema implementativo

```
.....
pid=fork();
if (pid == 0) { /* figlio */
 if (<redirezione input>){
 close(0);
 open(<filein>,O_RDONLY);
 }
 execl(<comando>);
 exit(-1);
}
.....
```

## Piping: schema implementativo

```
...
pipe(fd);
pid1=fork();
if (pid1==0){ /*1^ figlio*/
 close(1);
 dup(fd[1]);
 execl(<comando1>);
}
else{
 pid2=fork();
 if (pid2 ==0){ /*2^ figlio */
 close(0);
 dup(fd[0]);
 execl(<comando2>);
 }
 else{ /* padre */ }
}
}
```