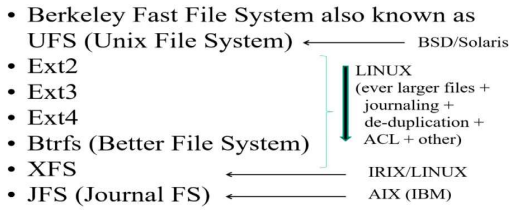
Virtual file system 5 UNIX

martedì 4 novembre 2025 11:47

Ricordiamoci l'albero genealogico di unix :

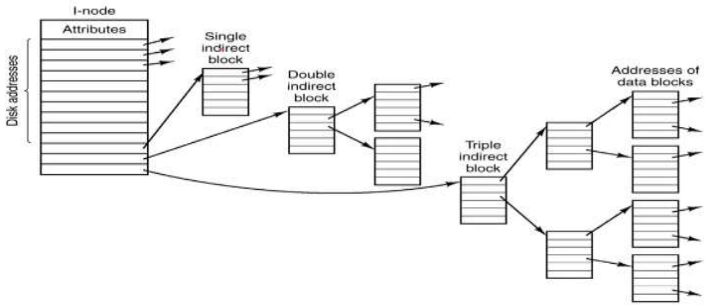


Vediamo in dettaglio alcune caratteristiche :

1. Un file è una sequenza di singoli byte
2. Metodo di accesso diretto (riposizionamento in qualunque punto)
3. Il record di sistema viene chiamato **i-node** (indexed node)
4. Gli i-node indicano informazioni su dati strutturali sul file system
5. Devono essere riportati su hard drive se si spenge il so
6. Esiste un vettore di i-node, il quale viene associato ad ogni file (esempio directory)
7. La struttura delal directory è composta da 4 o più bytes (numero i-node) + 2 ulteriori bytes per la lunghezza e nome + 2 per lo spiazzamento
8. La zona iniziale di un file system contiene le bit map (quali sono i blocchi liberi) e gli i-node liberi
   a. 

9. Attenzione però alla saturazione

Vediamo ora la struttura di un i-node (indicizzazione a livelli) :



I primi 10 record del primo i-node contengono le info su dove sono i file; mentre gli ultimi 3 (indicizzate indirettamente) contengono puntatori/riferimenti ad altri indici . Andiamo a vedere un esempio :

Parametri (dipendenti dalla versione di file system ed hard drive)

- Blocchi su disco da 512 byte
- Indirizzi su disco 4 byte
- In un blocco: 512/4 = 128 indirizzi
  - ind. 1-10 10 blocchi dati
  - ind. 11 128 blocchi dati
  - ind. 12 128^2 blocchi dati
  - ind. 13 128^3 blocchi dati

Maxfile = $10+128+128^2+128^3$ =
2.113.664 blocchi * 512 bytes =
1.082.195.968 ≈ 1Gbyte.

Tornando al discorso degli attributi dell'i-node :

{-,d,b,c,p}	**tipologia di file**: normale, directory, block-device, character-device, pipe
UID (2/4 byte) GID (2/4 byte)	**identificatori** del proprietario e del suo gruppo
rwx rwx rwx	**permessi di accesso** per proprietario, gruppo, altri (codifica ottale)
SUID (1 bit) SGID (1 bit)	**specifica di identificazione dinamica** per chi utilizza il file
Sticky (1 bit)	**per le directory rimuove la possibilita' di cancellare files se non si e' l'owner**

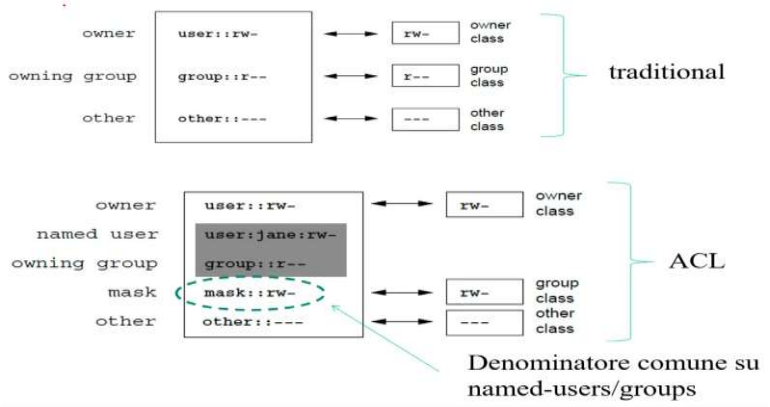
La prima tripletta rappresenta i permessi del proprietario del file; la seconda agli utenti appartenenti allo stesso gruppo del proprietario; l'ultima specifica i permessi per tutto il resto del mondo. UID-> User id ; GID->Group id. Possibile cambiare UID? Certo attraverso

```
uid_t getuid()   <-- Accessibile a tutte le utenze

int setuid(uid_t)   <-- Accessibile a "euid" root
```

Andiamo a vedere un esempio :

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

int main(int a ,char** b){

        uid_t id;
        uid_t euid;

        id = getuid();
        euid = geteuid();
        printf("I'm running on behalf of user %d - euid user %d\n",(int)id,(int)euid);
        printf(".. who would you like to become? ");
        scanf("%d",&id);
        setuid(id);
        id = getuid();
        euid = geteuid();
        printf("I'm now running on behalf of user %d - euid user %d\n",(int)id,(int)euid);
        pause();

}
```

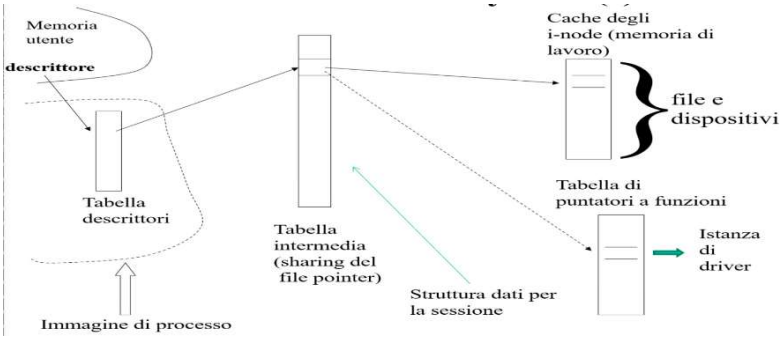
la quale se la eseguo io non fa il cambio : questo cambio lo può fare solo l'utente root. Quindi se lo faccio girare come root lo effettua il cambio. Vediamo ora in dettaglio : **ACL (access control list) :** servono a specificare i permessi a grana molto più fine. Si assegna ad un file un ACL, e vi si assegna un file shadow (ulteriori info e non accessibile alle applicazioni): quindi sia per il file che per quello shadow si hanno i-node. Quindi i comandi usati per gestire questa lista sono **getfacl e setfacl**. Vediamo ora come è fatto un ACL:



Attenzione alla mask (ulteriori info nel file shadow): serve o meno per dare/levare granting . Vediamo ora un esempio :

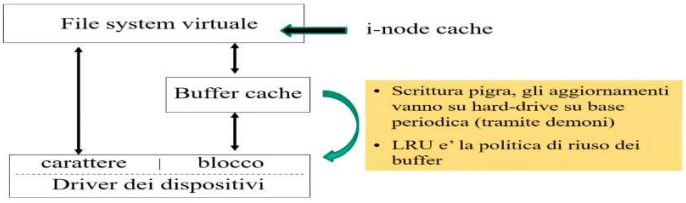
```
getACL:
        getfacl pippo

setACL:
        setfacl -m u:8765:r pippo
        setfacl -m g:8765:r pippo

maskACL:
        chmod 000 pippo

resetACL:
        setfacl -b pippo
```

Il quale se eseguito e senza permessi di root non si ha il cambio ; altrimenti se seguito come root sì . Inoltre per vedere i permessi ed il file shadow si usa il comando **ls –lai** . Vediamo ora come è fatta l'architettura vera e propria del virtual file system unix :



Dove la tabella intermedia rappresenta la sessione. Ogni collegamento ha un riferimento numerico.
Riassumendo il tutto :

1. I dispositivi in generale sono gestiti come file
2. Le operazioni eseguite vengono registrate all'interno del driver che andiamo ad utilizzare
3. Gli i-node associati a quei file vengono riportati su hard drive allo spegnimento
4. Ma può capitare che alcuni i-node vengano rimossi allo spegnimento del so
5. Esiste sempre un **buffer-cache** che serve a tenere temporaneamente i blocchi di dati relativi ai file in memoria d lavoro. Vediamo come è fatto in dettaglio :

a. 

b. È area di memoria che serve a mantenere i blocchi che arrivano dalle interazioni con i dispositivi (dati in ram)

1. Andiamo ora a vedere le system call per gestire i file :
   1. **Creazione**

int creat(char *file_name, int mode)	
Descrizione	invoca la creazione un file
Argomenti	1) *file_name: puntatore alla stringa di caratteri che definisce il nome del file da creare 2) mode: specifica i permessi di accesso al file da creare (codifica ottale)
Restituzione	-1 in caso di fallimento; un descrittore di file altrimenti

   a. **Esempio**

```
#include <stdio.h>

void main() {
        if(creat("pippo",0666) == -1) {
             printf("Errore nella chiamata creat \n");
             exit(1);
        }
}
```

   b. Quindi aggiorniamo sia gli i-node sia il file speciale (directory)
   2. **Apertura/chiusura**

a.

int open(char *file_name, int option_flags [, int mode])	
Descrizione	invoca la creazione un file
Argomenti	1) *file_name: puntatore alla stringa di caratteri che definisce il nome del file da aprire 2) option_flags: specifica la modalita' di apertura (read, write etc.) 3) mode: specifica i permessi di accesso al file in caso di creazione contestuale all'apertura
Restituzione	-1 in caso di fallimento, altrimenti un descrittore per l'accesso al file

int close(int descriptor)	
Descrizione	invoca la chiusura di un file
Argomenti	descriptor: descrittore del file da chiudere
Restituzione	-1 in caso di fallimento

   b. Dove i valori di option_flags sono i seguenti (in possibile combinazione tramite "|" :
      i.

O_RDONLY: apertura del file in sola lettura;

O_WRONLY: apertura del file in sola scrittura;

O_RDWR: apertura in lettura e scrittura;

O_APPEND: apertura del file con puntatore alla fine del file; ogni scrittura sul file sara' effettuata a partire dalla fine del file;

O_RDONLY: apertura del file in sola lettura;

O_WRONLY: apertura del file in sola scrittura;

O_RDWR: apertura in lettura e scrittura;

O_APPEND: apertura del file con puntatore alla fine del file; ogni scrittura sul file sara' effettuata a partire dalla fine del file;

O_CREAT : crea il file con modalita' d'accesso specificate da *mode* sole se esso stesso non esiste;

O_TRUNC : elimina il contenuto del file se esso gia' esistente.

O_EXCL : (exclusive) serve a garantire che il file sia stato effettivamente creato dalla chiamata corrente.

- definiti in **fcntl.h**
- combinabili tramite l'operatore "|" (OR binario)

ii. Con l'opzione truncate si resetta il file e si cancella il collegamento tra file e quello nel hard disk , il quale non viene minimamente modificato (modifica solo nell'i-node)

3. **Lettura/scrittura**

a.

ssize_t read(int descriptor, char *buffer, size_t size)	
Descrizione	invoca la lettura da un file
Argomenti	1) descriptor:descrittore relativo al file da cui leggere 2) buffer: puntatore al buffer dove memorizzare i byte letti 3) size: quantita' di byte da leggere
Restituzione	-1 in caso di fallimento, altrimenti il numero di byte realmente letti

ssize_t write(int descriptor, char *buffer, size_t size)	
Descrizione	invoca la scrittura su un file
Argomenti	1) descriptor:descrittore relativo al file su cui scrivere 2) buffer: puntatore al buffer dove prendere i byte da scrivere 3) size: quantita' di byte da scrivere
Restituzione	-1 in caso di fallimento, altrimenti il numero di byte realmente scritti

b. Quindi attenzione alla scrittura : se puramente scrittura i byte saranno sovrascritti ; altrimenti se siamo in coda al file le nuove info verranno aggiunte in coda (simil append).

4. Vediamo un esempio ovvero copiamo un file nel file system : il contenuto del secondo file è copia del primo : nel caso di shell si usa il comando **cp** . In dettaglio :

a.

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFSIZE 1024
int main(int argc, char *argv[]) {
        int sd, dd, size, result;
        char buffer[BUFSIZE];
         if (argc != 3) { /* check the number of arguments */
                printf("usage: copy source target\n");
                exit(1);
         }
         /* read only opening of the source file */
         sd=open(argv[1],O_RDONLY);
         if (sd == -1) {
                printf("source file open error\n");
                exit(1);
         }
         /* destination file creation */
         dd=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0660);
         if (dd == -1) {
                printf("destination file creation error\n");
                exit(1);
         }
        /* let's start with the copy operations */
        do {
                 /* read up to BUFSIZE from source */
                size=read(sd,buffer,BUFSIZE);
                if (size == -1) {
                  printf("source file read error\n");
                  exit(1);
                }
                /* write up to BUFSIZE to destination file */
                result = write(dd,buffer,size);
                if (result == -1) {
                        printf("destination file write error\n");
                        exit(1);
                }
        } while(size > 0);
        close(sd);
        close(dd);
  }/* end main*/
```

b. Il quale fa la copia del contenuto del file copy in pippo.

c. Attenzione se il contenuto copiato è minore della quantità voluta : dipende da quale driver D si utilizza (in generale dipende dal tipo di oggetto che si gestisce)

5. **Spostamento del file pointer**

a.

off_t lseek(int descriptor, off_t offset, int option)	
Descrizione	invoca il riposizionamento del file pointer
Argomenti	1) descriptor:descrittore relativo al file su cui riposizionarsi 2) offset:quantita' di caratteri di cui spostare il file pointer 3) option: opzione di spostamento (da inizio, da posizione corrente, da fine – valori relativi: 0, 1, 2)
Restituzione	-1 in caso di fallimento, altrimenti il nuovo valore del file pointer

Esempi

```
lseek(fd, 10, 0);  /* Spostamento di 10 byte dall'inizio di fd */
lseek(fd, 20, 1);  /* Spostamento di 20 byte in avanti dalla posizione
                      corrente  */
lseek(fd, -10, 1); /* Spostamento di 10 byte all'indietro dalla
                      posizione corrente */
lseek(fd, -10, 2); /* Spostamento di 10 byte all'indietro dalla fine
                      del file  */
lseek(fd, -10, 0); /* Fallisce e il valore del file pointer resta
```

6. **Duplicazione di descrittore e redirezione (canale)**

int dup(int descriptor)	
Descrizione	invoca la duplicazione di un descrittore
Argomenti	descriptor: descrittore da duplicare
Restituzione	-1 in caso di fallimento, altrimenti un nuovo descrittore

a. **Esempio**

Il descrittore restituito e' il primo libero nella tabella dei descrittori del processo

```
#define FNAME "info.txt"
#define STDIN 0

int main() {
        int fd;
        fd = open(FNAME,O_RDONLY);  /* Apro il file in lettura  */
        close(STDIN);               /* Chiudo lo standard input */
        dup(fd);                    /* Duplico il descrittore di file*/
        execlp("more","more",0);    /* Eseguo 'more' con input
                                       redirezionato */
    }
```

b. La duplicazione avviene nella prima entry libera
c. More legge da canale 0 e li scrive su canale 1
d. C'è una variante :

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

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

Specifica esplicita della posizione della "file-descriptor table" ove duplicare il canale originale

7. **Controllo canale i/o**
   a. Consente di vedere come i canali di i/o lavorano verso i file

b.

NAME top

fcntl - manipulate file descriptor

SYNOPSIS top

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

int fcntl(int fd, int cmd, ... /* arg */ );
```

DESCRIPTION top

fcntl() performs one of the operations described below on the open file descriptor *fd*. The operation is determined by *cmd*.

fcntl() can take an optional third argument. Whether or not this argument is required is determined by *cmd*. The required argument type is indicated in parentheses after each *cmd* name (in most cases, the required type is *int*, and we identify the argument using the name *arg*), or *void* is specified if the argument is not required.

   c. Permette di eseguire dato un id di un canale di i/o un comando
8. **Hard link e rimozione file**

a.
```
int link(const char *oldpath, const char * newpath)
```
ritorna –1 in caso di fallimento

```
int unlink(const char *pathname)
```
ritorna –1 in caso di fallimento

Inseriscono/eliminano una directory-entry in file speciale rappresentante una directory

   b. Vediamo cosa è un hard link : ovvero collegamento diretto verso l'i-node
   c. L'hard link va bene solo con elementi dello stesso tipo
   d. Quindi la rimozione dell'i-node si elimina l'hard link
9. **Soft link (symbolic link)**

- Sono file il cui contenuto identifica una stringa che definisce parte o tutto un pathname
- Possono esistere indipendentemente dal target

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

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

10. **Gestione delle directory**

a.
```
int mkdir(const char *pathname,mode_t mode)

int rmdir(const char *pathname)
```
Creazione/rimozione di directory

b. Chiamata dalla shell

11. **Gestione dei permessi di accesso**

a.
NAME
chmod, fchmod - change permissions of a file

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

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

DESCRIPTION
The mode of the file given by path or referenced by fields is changed.

varianti **chown** gestiscono il cambio di proprieta'

b. Cambio il valore della proprietà nell'i-node del file

12. **Vediamo degli esempi**
    a. **Analyser**
        i. File analyser analizza il file dicendo righe parole ecc ecc
        ii. File writer permette di scrivere su stdin
        iii.
```c
#include <unistd.h>

#define MAXBUF 4096

int main () {
        char buffer[MAXBUF];
        int res_r,res_w,prev_w;

        res_r = read(0, buffer, MAXBUF);

        while(res_r) {
                //scrivo tutto qui anche i residui
                prev_w = 0;
                res_w = 0;
                do {
                        prev_w = prev_w + res_w;
                        res_w = write(1, &buffer[prev_w], res_r - prev_w);
                } while (res_w + prev_w < res_r);

                res_r = read(0, buffer, MAXBUF);
        }

        return 0;

}
```
        iv. Tutto quello che viene immesso su stdin viene emesso su stdout
        v. File redirector

i.

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

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

  int fd;

  if (argc!= 2) {
    printf("Syntax: write_on_file <file_name>\n");
    exit(-1);
  }

  //permessi -> o in alternativa codifica ottale
  fd=open(argv[1], O_CREAT| O_TRUNC|O_WRONLY,S_IRUSR|S_IWUSR);

  if (fd == -1) {
        //il messaggio viene mandato su stdout
    perror("Open error: ");
    exit(2);
  }

  printf("fd=%d\n",fd);
  //chiudo stdout
  close (1);
  dup(fd);
  execve("./writer", NULL, NULL);
  perror("Exec error: ");
  exit(3);
}
```

ii. Ridireziona l'output sul file (pippo che è passato come paramentro)

iii. File analyser

iv.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BUFFER 1000

int error_function() {
   printf("Syntax: analyzer [-p] <nome_file>\n");
   exit(EXIT_FAILURE);
}

int main (int argc, char *argv[]){
 int fd, res_r, index;
 char buffer[MAX_BUFFER];
 char *filename;
 char nextchar;
 int continous = 0;
 int temp_charnum=0, temp_linenum=0, temp_wordnum=0;
 int charnum=0,linenum=0, wordnum=0;
 int newword = 0;

 if ((argc<2) || (argc>3)) error_function();
 if ((argc == 2) && (argv[1][0] == '-')) error_function();
 if ((argc == 3) && (strcmp(argv[1], "-p") !=0 )) error_function();
 if ((argc == 3) && (strcmp(argv[1], "-p") == 0)) continous= 1;
 if (argc == 2) filename = argv[1]; //caso limite
 else filename =argv[2];

 fd= open(filename, O_RDONLY| O_EXCL);
 if (fd== -1) {perror("Open error: ");exit(-2); }
 do {
    res_r = 1;
    temp_linenum=0; temp_wordnum=0; temp_charnum=0;
    newword=1;
    while (res_r){
       res_r = read (fd,buffer,MAX_BUFFER);
       index=0;
          while(index <res_r) {
             temp_charnum++;
             if ((buffer[index] != '\n') && (buffer[index] != ' ')){
                 if (newword) {
                        newword = 0;
                        temp_wordnum++;
                 }
              } else newword = 1;
            if (buffer[index] == '\n') temp_linenum++;
            index++;
       }
 }
 if ((linenum!= temp_linenum) || (wordnum!= temp_wordnum) ||(charnum!= temp_charnum)) {
    linenum = temp_linenum; wordnum = temp_wordnum;charnum = temp_charnum;
    printf("Number of characters: %d; number of words:%d; number of lines:%d\n",
    charnum,wordnum,linenum);    }
    if (continous) {
            sleep(1);
            lseek(fd, 0,SEEK_SET);
    }
 } while(continous);
 exit(0);
}
```

v. Per farla eseguire generiamo due nuove shell con il comando **xterminal &**, vedendo cosi "in real time" cosa accade , usando ./analyser -p nomefile

vi. Attenzione se rimuovo l'hard link del file (pippo) : rimuovo il collegamento ma l'applicazione continua ad eseguire codice : <u>può capitare che vi siano più sessioni</u>

a. **Tar di file**

i. Libreria che verrà usata

ii.

```c
#define BUFSIZE 1024
#define MAX_MANAGEABLE_FILES 128
#define MAX_FILE_NAME_LENGTH 128
#define PM_SIZE 12

char * the_magic = "\x11\x22\x33\x44\x55\x66\x77\x88\x99\xaa";//this is an arbitrary
sequence of values


typedef struct __attribute__((__packed__)) _position {
        char pos[PM_SIZE];
} position;

typedef struct __attribute__((__packed__)) _header{
        char magic[PM_SIZE];
        char num_files[PM_SIZE];
        char file_names[MAX_MANAGEABLE_FILES][MAX_FILE_NAME_LENGTH];
        //inizio e fine del file che rappresento
        position  start_position[MAX_MANAGEABLE_FILES];
        position  end_position[MAX_MANAGEABLE_FILES];
} header;

#define AUDIT if(0)
```

iii. File minitar.c

1)

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include "mini-tar.h"

int descriptors[MAX_MANAGEABLE_FILES];
header tar_head;

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

        int dd, size, result;
        char buffer[BUFSIZE];
        int i;
        off_t seek;
        int num_files = 0;
         if (argc < 3) { /* check the number of arguments */
                printf("usage: mini-tar dest source_1 [source_2] .. [source_n]\n");
                exit(EXIT_FAILURE);
         }
         /* destination file creation */
         dd=open(argv[1],O_WRONLY|O_CREAT|O_TRUNC,0660);
         if (dd == -1) {
                printf("destination file creation error\n");
                return EXIT_FAILURE;
         }
        for (i=2;argv[i]!=NULL;i++){
          num_files++;
          if(strlen(argv[i])>=MAX_FILE_NAME_LENGTH){
            printf("File %s - name too long\n",argv[i]);
            return EXIT_FAILURE;
          }
          descriptors[i-2] = open(argv[i],O_RDONLY);
          if(descriptors[i-2] == -1){
            printf("Cannot access file %s\n",argv[i]);
            return EXIT_FAILURE;
          }
          printf("file %s correctly opened - access descriptor is %d\n",argv[i],descriptors[i-2]);
          memcpy(&(tar_head.file_names[i-2]),argv[i],strlen(argv[i])+1);

        }
        sprintf(tar_head.magic,"%s",the_magic);
        sprintf(tar_head.num_files,"%d",num_files);
        printf("Tar head audit - num files %s - file names:\n",tar_head.num_files);
        for(i=0;i<num_files;i++){
                printf("%s\n",tar_head.file_names[i]);
                fflush(stdout);
        }
```

2)

```c
         //qui comincia l'operazione di tar
         result = write(dd,&tar_head,sizeof(header));
         printf("tar header size is %ld - fseek is %d\n",sizeof(header),result);
         fflush(stdout);
         for(i=0;i<argc-2;i++){
                 printf("taring file %s\n",argv[i+2]);
                 fflush(stdout);
                 seek = lseek(dd,0,SEEK_CUR);
                 sprintf(tar_head.start_position[i].pos,"%lu",seek);
                 printf("start position is %ld\n",seek);
                 fflush(stdout);
                 /* let's start with the copy operations */
                 do {
                         AUDIT
                         printf("cycling - descriptor is %d\n",descriptors[i]);
                          /* read up to BUFSIZE from source */
                         size=read(descriptors[i],buffer,BUFSIZE);
                         if (size == -1) {
                           printf("file read error\n");
                           fflush(stdout);
                           exit(EXIT_FAILURE);
                         }
                         AUDIT
                         printf("read done\n");
                         fflush(stdout);
                         /* write up to BUFSIZE to destination file */
                         result = write(dd,buffer,size);
                         if (result == -1) {
                                 printf("mini-tar file write error\n");
                                 fflush(stdout);
                                 exit(EXIT_FAILURE);
                         }
                 } while(size > 0);
                 seek = lseek(dd,0,SEEK_CUR);
                 sprintf(tar_head.end_position[i].pos,"%lu",seek);
                 printf("end position is %ld\n",seek);
                 fflush(stdout);
         }
         printf("realigning the header\n");
         fflush(stdout);
         lseek(dd,0,SEEK_SET);
         result = write(dd,&tar_head,sizeof(header));

 }/* end main*/
```

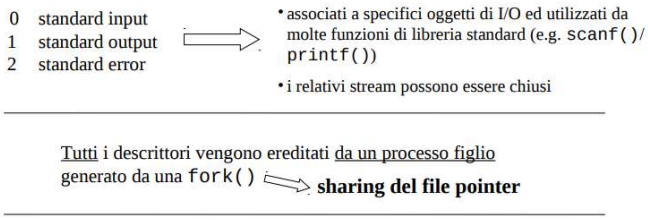
iv. Il quale per mandarlo in esecuzione si usa **./mtar tarfile pippo pluto**

v. Analogamente per il processo inverso si usa **./mextract tarfile pippo**

vi. Nota: questo programma è non portabile tra diverse macchine

13. Vediamo ora ultimo concetto : l'ereditarietà dei descrittori ovvero :

a.



Tutti i descrittori (inclusi 0, 1 e 2) restano validi quando avviene una sostituzione di codice tramite una chiamata execX() eccetto che nel caso in cui si specifichi operatività close-on-exec (tramite la system-call `fcntl()` o il flag `O_CLOEXEC` in apertura del file)

b. Il canale 0 è quello usato da scanf
c. Il canale 1 viene usato da printf
d. Il canale 2 serve a comunicare da parte delle librerie gli errori che possono sorgere
e. Si possono chiudere i canali??? Certo che si