# PROVE PRIMITIVE SOeLAB A.A. 2020-21

**Nota bene:**

1) I comandi saranno indicati in carattere courier normale, mentre il risultato dell'esecuzione dei comandi in courier italico!
2) Poiché verranno usati vari file di lucidi/slide, l'indicazione del numero del lucido/slide sarà sempre specificata; comunque i file cui si farà riferimento sono i seguenti:
   - Slide introduttive su File System (con password di lettura)
   - Slide sulle primitive UNIX per file facenti parte della libreria standard del linguaggio C (con password di lettura)
   - Slide sulle tabelle di UNIX per l'interazione con i file (con password di lettura)
   - Slide sui processi UNIX (con password di lettura)
   - Slide sulle pipe e fifo UNIX (con password di lettura)
   - Slide sui segnali UNIX (con password di lettura)
3) Sono indicati in evidenziato giallo delle cose o che non sono state dette a lezione oppure se sono state corrette rispetto a quanto visto a lezione.

## Sommario

Lezione Mercoledì 14/04/2021 (seconda ora) ➔ corrisponde a una video-registrazione ..... 2
Lezione Lunedì 19/04/2021 ➔ corrisponde a due video-registrazioni ..... 3
Lezione Mercoledì 21/04/2021 ➔ corrisponde a due video-registrazioni ..... 17
Lezione Lunedì 26/04/2021 ➔ corrisponde a due video-registrazioni ..... 30

## Lezione Mercoledì 14/04/2021 (seconda ora) ➔ corrisponde a una video-registrazione

**Visione di insieme: si veda il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/figuraPuntoDiVistaEsternoSHELL_C.mp4**
**(Luc. C/Unix Primitive per i file 1-3)**
**(Luc. File System 9-11)**
**(Luc. C/Unix Primitive per i file 4-7)**
**(Luc. Unix. Tabelle per l'interazione con i file 1-3)**

**Verifichiamo quali file system fisici sono montati sul nostro server (lica04): comando df (disk free, disco libero). NOTA BENE: RIPORTIAMO SOLO LE LINEE CHE INTERESSANO!**

```
soELab@Lica04:~$ df
Filesystem     1K-blocks    Used Available Use% Mounted on
…
/dev/sda2       16445308 10252880   5337340  66% /                  <== punto di mount
…
/dev/sdb1      256979396  3424760 240431156   2% /home               <== punto di mount
…
```

**Usiamo anche l'opzione –i per verifichiamo il numero di inode totali, usati e liberi**

```
soELab@Lica04:~$ df -i
Filesystem       Inodes  IUsed    IFree IUse% Mounted on
…
/dev/sda2       1048576 228468   820108   22% /
…
/dev/sdb1      16384000  75138 16308862    1% /home
…      ↑ nome del dispositivo montato
```

**Verifichiamo le caratteristiche dei due dispositivi montati:**

```
soELab@lica04:~$ ls -l /dev/sda2 /dev/sdb1
brw-rw---- 1 root disk 8,  2 Apr 14 08:54 /dev/sda2
brw-rw---- 1 root disk 8, 17 Apr 14 08:54 /dev/sdb1    <== b indica che sono dispositivi organizzati a BLOCCHI!
```

**(Luc. Unix. Tabelle per l'interazione con i file 4-6)**

**(fine lezione di mercoledì 14/04/2021)**

## Lezione Lunedì 19/04/2021 ➔ corrisponde a due video-registrazioni

**(ancora Luc. C/Unix Primitive per i file 7)**

**Possiamo usare il comando man anche verificare il funzionamento delle primitive (in questo caso dobbiamo usare l'opzione –s per usare la sezione giusta del manuale che per le primitive è la sezione 2): controlliamo il manuale della open e della close.**

```
soELab@Lica04:~$ man -s 2 open
OPEN(2)                          Linux Programmer's Manual                          OPEN(2)

NAME
       open, …, creat - open and possibly create a file

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

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

**<== questa versione la vedremo poi!**

```
       int creat(const char *pathname, mode_t mode);

…

DESCRIPTION
The open() system call opens the file specified by pathname.  If the specified file does not exist,
it may optionally (if O_CREAT is specified in flags) be created by open().

The return value of open() is a file descriptor, a small, nonnegative integer that  is  used  in
subsequent  system  calls  (read(2),  write(2),  lseek(2), fcntl(2), etc.) to refer to the open file.
The file descriptor returned by a successful call will be the lowest-numbered file descriptor not
currently open for the process.

soELab@lica04:~/file$ man -s 2 close
CLOSE(2)                 Linux Programmer's Manual                          CLOSE(2)

NAME
       close - close a file descriptor

SYNOPSIS
       #include <unistd.h>
```

```
        int close(int fd);
DESCRIPTION
close()  closes a file descriptor, so that it no longer refers to any file and may be reused. …
```

**Vediamo di verificare in pratica i concetti che abbiamo visto nella scorsa lezione.**
**Per prima cosa verifichiamo il funzionamento di open e dei parametri di invocazione (argc e argv) ==> usare directory ~/file**

```
soELab@Lica04:~/file$ cat provaopen.c
#include <stdio.h>        <== deve essere incluso per poter usare printf e puts
#include <stdlib.h>       <== deve essere incluso per poter usare la primitiva exit
#include <unistd.h>       <== deve essere incluso per poter usare la primitiva close
#include <fcntl.h>        <== deve essere incluso per poter usare le costanti per la open (O_RDONLY, O_WRONLY e O_RDWR)

int main(int argc, char **argv)
{
        int fd1, fd2, fd3;

if (argc != 2) { puts("Errore nel numero di parametri");  <== per prima cosa controlliamo il numero di parametri:
vogliamo avere un numero uguale a 1 (il nome del file che vogliamo aprire) e quindi il numero dei parametri deve essere 1+1 (nome del comando stesso)=2
                 exit(1); }    <== in caso di errore, usiamo la primitiva exit con un numero diverso ogni volta (in modo assolutamente analogo a quanto veniva fatto nella shell)!

if ((fd1 = open(argv[1], O_RDONLY)) < 0)          <== invochiamo la open e controlliamo se il valore di ritorno è minore di 0; FARE MOLTA ATTENZIONE ALLE PARENTESI INDICATE! Unix controllerà i diritti sul file del processo in esecuzione controllando UID e GID effettivi (che si trovano nel descrittore di processo) nei confronti di UID e GID del file (che si trovano nell'inode)
                { puts("Errore in apertura file");
                  exit(2); }
else
        printf("Valore di fd1 = %d\n", fd1);          <== stampiamo il valore ritornato dalla open!

if ((fd2 = open(argv[1], O_RDONLY)) < 0)          <== NOTA BENE: usiamo lo stesso nome di file; verrà occupato un ulteriore elemento libero della TFA dei processo e un ulteriore elemento della TFA di sistema; sia questo elemento che il precedente (derivante dalla precedente open) farà riferimento all'unico elemento della Tabella degli I-NODE attivi che contiene la copia dell'I-NODE del file il cui nome è argv[1]!
```

```
                { puts("Errore in apertura file");
                  exit(2); }
else
        printf("Valore di fd2 = %d\n", fd2);

close(fd1);          <== chiudiamo il primo file aperto

if ((fd3 = open(argv[1], O_RDONLY)) < 0)          <== NOTA BENE: stessa nota di cui sopra!
                { puts("Errore in apertura file");
                  exit(2); }
else
        printf("Valore di fd3 = %d\n", fd3);      <== verifichiamo il valore ritornato dalla open!

return 0;                                         <== ritorniamo 0 che è il valore di successo in UNIX!
                                                  <== NOTA BENE: Equivalente a exit(0);
}
```

**Ricordiamo come si fa ad ottenere una versione eseguibile da un programma C utilizzando l'utility make (come spiegato nella video-registrazione a cura di Stefano Allegretti che si trova nella sezione Laboratorio di Dolly):**

```
soELab@lica04:~/file$ make
make: Nothing to be done for 'all'.
```

**Poiché ho già la versione eseguibile, il make mi dice che non ha nulla da fare per il target "all". Quindi andiamo a 'fingere' di cambiare il file provaopen.c usando il comando touch e quindi rilanciamo l'utility make:**

```
soELab@lica04:~/file$ touch provaopen.c
soELab@lica04:~/file$ make
gcc   -Wall  provaopen.c -o provaopen   <== con l'opzione –Wall andiamo a verificare tutti i warning!
```

**che ci mostra il comando che viene eseguito: ricordarsi che NON ci devono essere warning, oltre che chiaramente errori di compilazione e di linking!**

**Ora invochiamo il programma** `provaopen`**, che ricordiamo verrà eseguito da un processo, figlio del processo di shell!**

```
soELab@Lica04:~/file$ provaopen     <== invocazione sbagliata: ci vuole in parametro!
Errore nel numero di parametri
soELab@Lica04:~/file$ echo $?
1
```

```
soELab@Lica04:~/file$ provaopen provaopen.c
```
**<== passiamo come parametro un file che sappiamo essere leggibile dal proprietario!**

```
Valore di fd1 = 3
Valore di fd2 = 4
Valore di fd3 = 3
```
**<== dopo la close, si verifica il riutilizzo dell'elemento di indice 3!**

**Passiamo ora a verificare la dimensione della tabella dei file aperti di ogni singolo processo (tramite un altro programma che si chiama proveopen.c):**

```
soELab@Lica04:~/file$ cat proveopen.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
        int i=0, fd;

if (argc != 2) { puts("Errore nel numero di parametri");
                 exit(1); }

while (1)
        if ((fd = open(argv[1], O_RDONLY)) < 0)
               { puts("Errore in apertura file");
                 printf("Valore di i = %d\n", i);
                 exit(2); }
        else i++;
return 0;
}
```
**<== ciclo infinito!**

**<== si terminerà solo quando si avrà un errore della open che deriverà, nel nostro caso, solo da esaurimento dello spazio nella tabella dei file aperti del processo!**

```
soELab@Lica04:~/file$ proveopen
```
**<== invocazione sbagliata: ci vuole in parametro!**

```
Errore nel numero di parametri
soELab@Lica04:~/file$ echo $?
1
soELab@Lica04:~/file$ proveopen proveopen.c
Errore in apertura file
```

*Valore di i = 1021* **<== la dimensione totale della tabella si ricava da 1021 + 3 che sono i primi 3 elementi della tabella dei file aperti di ogni processo, usati per standard input, standard output e standard error: quindi = 1024!**

```
soELab@Lica04:~/file$ echo $?
2
```

**(Luc. C/Unix Primitive per i file 8 per ora saltato)**

**(Luc. File System 12-13)**
**Per poter capire le prossime primitive dobbiamo capire prima in generale i metodi di accesso**
**(Luc. C/Unix Primitive per i file 9)**
**Quindi vediamo il concetto di I/O pointer (o file pointer), necessario per l'accesso sequenziale, implementato in UNIX/LINUX.**
**Leggiamo un altro pezzo del manuale della primitiva open (LASCIATO AGLI STUDENTI DA GUARDARE):**

*A call to open() creates a new open file description, an entry in the system-wide table of open files. The open file description records the* ***file offset*** *and the* ***file status flags*** *…*

**Nota bene: il file offset del manuale è quello che chiamiamo I/O pointer (o file pointer)**
**(di nuovo Luc. Unix. Tabelle per l'interazione con i file 1-2)**

**(Luc. C/Unix Primitive per i file 9-11)**
**Dal man della read e della write: (LASCIATO AGLI STUDENTI DA GUARDARE):**

*READ(2) Linux Programmer's Manual READ(2)*

*NAME*
*read - read from a file descriptor*

*SYNOPSIS*

```
#include <unistd.h>

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

*DESCRIPTION*
*read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. On files that support seeking, the read operation commences at the* ***current file offset****, and the* ***file offset is incremented by the number of bytes read****. If the current file offset is at or past the end of file, no bytes are read, and read() returns zero.*
*…*
*RETURN VALUE*

***On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.*** *It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file,… On error, -1 is returned, and errno is set appropriately.*

*WRITE(2) Linux Programmer's Manual WRITE(2)*

*NAME*
*write - write to a file descriptor*

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

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

*DESCRIPTION*
*write() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.*

*The number of bytes written may be less than count if, for example, there is insufficient space on the underlying physical medium, …*
*For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the* ***current file offset****, and the* ***file offset is incremented by the number of bytes actually written****.*
*If the file was open(2)ed with O_APPEND, the file offset is first set to the end of the file before writing.*
*The adjustment of the file offset and the write operation are performed as an atomic step.*
*…*
*RETURN VALUE*
***On success, the number of bytes written is returned*** *(zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.*

**(Luc. C/Unix Primitive per i file 12-14 LASCIATI AGLI STUDENTI DA GUARDARE):)**

**(Luc. C/Unix Primitive per i file 15)**
**Vediamo un primo esempio di uso delle primitive read e write:**

```
soELab@Lica04:~/file$ cat copia.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>                    <== deve essere incluso per poter usare le primitive read, write e close
#include <fcntl.h>
#define PERM 0644   /* in UNIX */      <== diritti espressi in OTTALE: read-write per proprietario, read per gruppo e read per tutti gli altri!

int copyfile (char *f1, char * f2)
{    int infile, outfile, nread;
     char buffer [BUFSIZ];  /* usato per i caratteri */

if ((infile = open(f1, O_RDONLY)) < 0) return 2;
/* ERRORE se non si riesce ad aprire in LETTURA il primo file */

if ((outfile = creat(f2, PERM)) <0 )
/* ERRORE se non si riesce a creare il secondo file */
                        {close(infile); return 3; }

while ((nread = read(infile, buffer, BUFSIZ)) > 0 )   <== FARE MOLTA ATTENZIONE ALLE PARENTESI INDICATE!
{ if (write(outfile, buffer, nread) < nread )
/* ERRORE se non si riesce a SCRIVERE */
    { close(infile); close(outfile); return 4; }
}
close(infile); close(outfile); return 0;
/* se arriviamo qui, vuol dire che tutto e' andato bene */
}

int main (int argc, char** argv)
{   int status;
if (argc != 3)  /* controllo sul numero di argomenti */
     { printf("Errore: numero di argomenti sbagliato\n");
       printf("Ci vogliono 2 argomenti: nome-file-sorgente e nome-file-destinazione\n");
       exit (1); }
```

```
status = copyfile (argv[1], argv[2]);
if (status != 0)
 printf("Ci sono stati degli errori durante la copia\n");
return status;
}
```

**Vediamo come funziona:**

```
soELab@Lica04:~/file$ copia
Errore: numero di argomenti sbagliato
Ci vogliono 2 argomenti: nome-file-sorgente e nome-file-destinazione
soELab@Lica04:~/file$ echo $?
1
soELab@Lica04:~/file$ ls -l pippo
ls: cannot access pippo: No such file or directory
soELab@Lica04:~/file$ copia pippo pap
Ci sono stati degli errori durante la copia
soELab@Lica04:~/file$ echo $?
2
soELab@Lica04:~/file$ ls -l F1
-rw------- 1 soELab users 24 Feb 21  2019 F1
soELab@Lica04:~/file$ cat F1
Sono il file di
nome F1
soELab@Lica04:~/file$ copia F1 F2
soELab@Lica04:~/file$ echo $?
0
soELab@Lica04:~/file$ ls -l F2
-rw-r--r-- 1 soELab users 24 Apr 19 12:27 F2
soELab@Lica04:~/file$ cat F2
Sono il file di
nome F1
```

**(Luc. C/Unix Primitive per i file 17)**

**In questo esempio, abbiamo letto BUFSIZ byte alla volta dal file sorgente perché tanto li dovevamo scrivere sul file destinazione senza intervenire in alcun modo; verifichiamo ora il valore di BUFSIZ:**

```
soELab@Lica04:~/file$ cat provaBUFSIZ.c
```

```
#include <stdio.h>

int main()
{
printf("Il valore di BUFSIZ is %d\n", BUFSIZ);
return 0;
}
soELab@Lica04:~/file$ provaBUFSIZ
Il valore di BUFSIZ is 8192
```

**Vediamo ora cosa succede se cambiamo nel programma precedente i diritti che assegniamo al file che creiamo: consideriamo il file copia-new.c, ma prima leggiamo un altro pezzo del manuale della primitiva open/creat:**

…
*The **mode** argument specifies the file mode bits be applied when a new file is created. …*
*The effective mode is modified by the process's umask in the usual way: … the mode of the created file is **(mode & ~umask)**.*
…

```
soELab@Lica04:~/file$ cat copia-new.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
```
***#define PERM 0666*** */* in UNIX */* **<== diritti espressi in OTTALE: read-write per proprietario, e idem per gruppo e per tutti gli altri!**

```
/* rispetto al file copia.c abbiamo cambiato i permessi: peccato che se il comando umask riporta come
nel nostro caso
soELab@lica04:~/file$ umask
```
***0022***

*il risultato e' comunque che il file viene creato con i diritti rw per U, e r per G e O e quindi 0644.*
*Vediamo perche'. Dal manuale della open/creat scopriamo che:*
*The effective permissions are modified by the process's umask in the usual way: The permissions of the created file are (mode & ~umask)*
*Quindi se mode = 0666 e umask = 0022, tralasciando lo 0 iniziale abbiamo che*

```
666 in binario e'110110110 e 022 è 000010010 e quindi ~umask è 111101101 e quindi mode & ~umask e'
110110110 &
111101101
=========
110100100
e quindi proprio 644 */

int copyfile (char *f1, char * f2)
{    int infile, outfile, nread;
     char buffer [BUFSIZ];  /* usato per i caratteri */

if ((infile = open(f1, O_RDONLY)) < 0) return 2;
/* ERRORE se non si riesce ad aprire in LETTURA il primo file */

if ((outfile = creat (f2, PERM)) <0 )
/* ERRORE se non si riesce a creare il secondo file */
                        {close(infile); return 3; }

while ((nread = read (infile, buffer, BUFSIZ)) > 0 )
{ if (write(outfile , buffer, nread) < nread)
/* ERRORE se non si riesce a SCRIVERE */
    { close(infile); close(outfile); return 4; }
}
close(infile); close(outfile); return 0;
/* se arriviamo qui, vuol dire che tutto e' andato bene */
}

int main  (int argc, char** argv)
{   int status;
if (argc != 3)  /* controllo sul numero di argomenti */
     { printf("Errore: numero di argomenti sbagliato\n");
       printf("Ci vogliono 2 argomenti: nome-file-sorgente e nome-file-destinazione\n");
       exit (1); }
status = copyfile(argv[1], argv[2]);
if (status != 0)
```

```
 printf("Ci sono stati degli errori durante la copia\n");
return status;
}
```

**Verifichiamo il valore ritornato dal comando umask nel nostro caso:**

```
soELab@Lica04:~/file$ umask
0022
```

**Vediamo quindi cosa capiterà quando manderemo in esecuzione il programma copia-new: si veda il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/figura-umask.mp4**

**Proviamo ora il funzionamento del nuovo programma di copia-new:**

```
soELab@Lica04:~/file$ copia-new F1 F2-new
soELab@Lica04:~/file$ ls -l F2*
-rw-r--r-- 1 soELab users 24 Apr 19 12:27 F2
-rw-r--r-- 1 soELab users 24 Apr 19 13:01 F2-new
```

**<== Come spiegato anche nel video, nonostante la costante PERM valga 0666, l'effetto dell'umask è tale che i diritti sono comunque read-write per proprietario e solo read per gruppo e altri!**

**(Luc. C/Unix Primitive per i file 16)**
**Vediamo un esempio di uso del concetto di ridirezione ==> usare sempre directory ~/file**

```
soELab@Lica04:~/file$ cat copiarid.c
#include <stdio.h>
#include <unistd.h>

int main()
{       char buffer [BUFSIZ];
        int nread;

while ((nread = read(0, buffer, BUFSIZ)) > 0 )
/* lettura dallo standard input fino a che ci sono caratteri */
                write(1, buffer, nread);
/* scrittura sullo standard output dei caratteri letti */
return 0;
}
```

**<== Si legge da standard input!**

**<== Si scrive su standard output!**

```
soELab@Lica04:~/file$ copiarid
ciao                                    <== Tutto quello che si scrive su standard input!
ciao                                    <== viene riportato su standard output!
come stai?
come stai?
```

**NOTA BENE: copiarid si comporta come il filtro cat!**

**Vediamone quindi anche l'uso utilizzando la ridirezione della shell:**

**1) ridirezione dello standard input e dello standard output**

```
soELab@Lica04:~/file$ copiarid < F1 > F1-rid
soELab@Lica04:~/file$ ls -l F1-rid
-rw-r--r-- 1 soELab users 24 Apr 19 12:56 F2-rid
```

**2) ridirezione dello standard input**

```
soELab@Lica04:~/file$ copiarid < F1-rid
Sono il file di
nome F1
```

**3) ridirezione dello standard output**

```
soELab@Lica04:~/file$ ls FF
ls: cannot access FF: No such file or directory
soELab@Lica04:~/file$ copiarid > FF
ciao
come stai?
io bene ...
```

**<== ricordarsi che la fine dello standard input si ottiene con ^D (CTRL-D), di cui non viene fatto l'echo sul terminale! MEGLIO A LINEA NUOVA!**

```
soELab@Lica04:~/file$ copiarid < FF
ciao
come stai?
io bene ...
soELab@Lica04:~/file$ ls -l FF
-rw-r--r-- 1 soELab users 43 Apr 19 12:57 FF <== N.B.: i file creati dalla ridirezione hanno come diritti proprio 0644!
```

**Vediamo ora di rendere ancora più simile il nostro programma al comando/filtro cat del sistema: quindi si deve comportare come il filtro cat e come il comando cat (limitandoci al caso di voler visualizzare un solo file: PROVARE A REALIZZARNE, COME ESERCIZIO, UNA VERSIONE CHE ACCETTA UN NUMERO QUALSIASI DI FILE ESATTAMENTE COME FA IL COMANDO CAT). ==> usare sempre directory ~/file**

```
soELab@Lica04:~/file$ cat mycat.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char **argv)
{       char buffer [BUFSIZ];
        int nread, fd = 0;
```

**<== Inizializzato a 0: se non si passa un parametro, allora si legge da standard input!**

```
if (argc > 2) { puts("Errore nel numero di parametri");
                exit(1); }
```

**<== Se più di 1 parametro, errore!**

```
if (argc == 2)
```

**<== Nel caso si passi un parametro, allora si deve aprire il file passato!**

```
/* abbiamo un parametro che deve essere considerato il nome di un file */
        if ((fd = open(argv[1], O_RDONLY)) < 0)
                { puts("Errore in apertura file");
                  exit(2); }

/* se non abbiamo un parametro, allora fd rimane uguale a 0 */
while ((nread = read (fd, buffer, BUFSIZ)) > 0 )
/* lettura dal file o dallo standard input fino a che ci sono caratteri */
                write(1, buffer, nread);
/* scrittura sullo standard output dei caratteri letti */
return 0;
}
```

**Vediamone l'uso:**

**1) invocazione sbagliata (diversamente dal comando cat!)**

```
soELab@lica04:~/file$ mycat F*
Errore nel numero di parametri
soELab@lica04:~/file$ echo $?
1
```

**2) come filtro (senza ridirezione)**

```
soELab@Lica04:~/file$ mycat
fn zgn
```

```
fn zgn
dfnadf
dfnadf
```

**<== ricordarsi che la fine dello standard input si ottiene con ^D (CTRL-D)!**

**3) come comando**

```
soELab@Lica04:~/file$ mycat F1
Sono il file di
nome F1
```

**4) come commando con ridirezione in uscita**

```
soELab@Lica04:~/file$ mycat F1 > F1-mycat
```

**5) come filtro con ridirezione in ingresso**

```
soELab@Lica04:~/file$ mycat < F1-mycat
Sono il file di
nome F1
```

**6) come filtro con ridirezione in ingresso e in uscita**

```
soELab@Lica04:~/file$ mycat < F1 > F2-mycat
soELab@Lica04:~/file$ mycat < F2-mycat
Sono il file di
nome F1
soELab@Lica04:~/file$ ls -l *-mycat
-rw-r--r-- 1 soELab users 24 Apr 19 18:35 F1-mycat
-rw-r--r-- 1 soELab users 24 Apr 19 18:36 F2-mycat
```

**(fine lezione di lunedì 19/04/2021)**

## Lezione Mercoledì 21/04/2021 ➔ corrisponde a due video-registrazioni

**(ancora Luc. C/Unix Primitive per i file 17)**
**Chiarimento su numero di byte da leggere in base alle specifiche.**

**(Luc. C/Unix Primitive per i file 18)**
**Precisato che i controlli sui diritti di accesso vengono effettuati su UID e GID effettivi del processo che esegue un certo programma: si veda anche osservazione inserita nel codice dell'esecizio provaopen.c della lezione scorsa.**

**(Luc. C/Unix Primitive per i file 19)**
**Passiamo ora a considerare di voler realizzare in un programma C il funzionamento della parte della shell che implementa la ridirezione sia in ingresso che in uscita**

```
soELab@Lica04:~/file$ cat ridir.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define PERM 0644   /* in UNIX */

int copyfile (char *f1, char * f2)
{    int infile, outfile, nread;
     char buffer [BUFSIZ];  /* usato per i caratteri */
```

***close(0);*** **<== Prima di aprire il file sorgente in lettura, chiudiamo il file descriptor 0 e quindi liberiamo il primo elemento della tabella dei file aperti del processo corrente (quello che eseguirà il programma nella sua forma eseguibile)!**

```
if ((infile = open(f1, O_RDONLY)) < 0) return 2;
```

**<== Se ha successo andrà ad occupare la posizione con fd = 0!**

```
/* ERRORE se non si riesce ad aprire in LETTURA il primo file */
```

***close(1);*** **<== Prima di creare il file destinazione, chiudiamo il file descriptor 1 e quindi liberiamo il primo elemento della tabella dei file aperti del processo corrente (quello che eseguirà il programma nella sua forma eseguibile)!**

```
if ((outfile = creat(f2, PERM) ) <0)
```

**<== Se ha successo andrà ad occupare la posizione con fd = 1!**

```
/* ERRORE se non si riesce a creare il secondo file */
                        { close(infile); return 3; }
```

**<== NOTA BENE: da qui in poi qualunque lettura/scrittura (di basso livello come di alto livello) risulta essere ridirezionata!**

```
while ((nread = read(infile, buffer, BUFSIZ)) > 0 )          <== infile = 0!
{ if (write(outfile, buffer, nread) < nread)                 <== outfile = 1!
/* ERRORE se non si riesce a SCRIVERE */
    { close(infile); close(outfile); return 4; }
}
close(infile); close(outfile); return 0;
/* se arriviamo qui, vuol dire che tutto e' andato bene */
}

int main(int argc, char** argv)
{   int status;
if (argc != 3)  /* controllo sul numero di argomenti */
     { printf("Errore: numero di argomenti sbagliato\n");
       printf("Ci vogliono 2 argomenti: nome-file-sorgente e nome-file-destinazione\n");
       exit (1); }
status = copyfile(argv[1], argv[2]);
if (status != 0)
 printf("Ci sono stati degli errori durante la copia\n"); <== NOTA BENE: possibili problemi se c'è un qualunque fallimento dalla creazione del file destinazione in poi!
return status;
}
```

**Verifichiamone il funzionamento:**

```
soELab@Lica04:~/file$ ridir
Errore: numero di argomenti sbagliato
Ci vogliono 2 argomenti: nome-file-sorgente e nome-file-destinazione
soELab@Lica04:~/file$ echo $?
1
soELab@Lica04:~/file$ ridir pippo pap
Ci sono stati degli errori durante la copia          <== ci ricordiamo che il file pippo NON esiste!
soELab@Lica04:~/file$ echo $?
2
soELab@Lica04:~/file$ ridir F1 F2-ridir

soELab@Lica04:~/file$ echo $?
0
```

**Verifichiamone le informazioni del file F2-ridir:**

```
soELab@Lica04:~/file$ ls -l F2-ridir
-rw-r--r-- 1 soELab users 24 Apr 21 11:24 F2-ridir
soELab@Lica04:~/file$ cat F2-ridir
Sono il file di
nome F1
```

**Se vogliamo verificare il valore delle variabili infile e outfile, bisogna che usiamo un dispositivo diverso dallo standard output (dato che è ridiretto). Ad esempio, come nella shell, possiamo pensare di usare il dispositivo standard /dev/tty che non è soggetto a ridirezione. Vediamo una possibile versione in cui dovremo 'divertirci' un po' con le stringhe:**

```
soELab@Lica04:~/file$ cat ridir-constampesudev.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#define PERM 0644   /* in UNIX */

int fd; /* per file/dispositivo /dev/tty; NOTA BENE: fd l'abbiamo definita come variabile globale perché serve sia in copyfile che nel main */

int copyfile (char *f1, char * f2)
{    int infile, outfile, nread;
     char s[100] = "Valore di infile ";
     char s1[5];
     char s2[] = " e di outfile ";
     char s3[] = "\n";
     char buffer [BUFSIZ];  /* usato per i caratteri */

fd = open("/dev/tty", O_WRONLY);   <== apriamo in scrittura il file/dispositivo /dev/tty (abbiamo omesso i controlli, perché diamo per scontato che non ci siano problemi ad aprire in scrittura questo dispositivo!)

close(0);
if (( infile = open (f1, O_RDONLY) ) < 0) return (2);
/* ERRORE se non si riesce ad aprire in LETTURA il primo file */
```

```
close(1);
if (( outfile = creat (f2, PERM) ) <0 )
/* ERRORE se non si riesce a creare il secondo file */
                        {close (infile); return (3); }

sprintf(s1, "%d", infile);
strcat(s, s1);
strcat(s, s2);
sprintf(s1, "%d", outfile);
strcat(s, s1);
strcat(s, s3);
write(fd, s, strlen(s));

while (( nread = read (infile, buffer, BUFSIZ) ) > 0 )
{ if ( write (outfile , buffer, nread) < nread )
/* ERRORE se non si riesce a SCRIVERE */
    { close (infile); close (outfile); return (4); }
}
close (infile); close (outfile); return (0);
/* se arriviamo qui, vuol dire che tutto e' andato bene */
}

int main  (int argc, char** argv)
{   int status;
if (argc != 3)  /* controllo sul numero di argomenti */
     { printf("Errore: numero di argomenti sbagliato\n");
       printf("Ci vogliono 2 argomenti: nome-file-sorgente e nome-file-destinazione\n");
       exit (1); }
status = copyfile (argv[1], argv[2]);
if (status != 0)
  write(fd, "Ci sono stati degli errori durante la copia\n", 44);
return status;
}
```

**<== se non vi ricordate il funzionamento di questa funzione di libreria, potete usare man sprintf!**

**<== riutilizziamo s1 dato che il contenuto precedente è stato già copiato in s**

**<== prepariamo la stringa che andiamo a scrivere su /dev/tty**

**<== qui possiamo usare la printf perché non abbiamo ancora chiamato la copyfile!**

**<== usiamo sempre fd se per caso ci sono stati errori, ad esempio nella creazione del file destinazione! N.B. 44 sono i caratteri di cui è costituita la stringa che stiamo scrivendo compreso lo \n!**

**Verifichiamone il funzionamento:**

**1) corretto**

```
soELab@Lica04:~/file$ ridir-constampesudev F1 FF2
Valore di infile 0 e di outfile 1
soELab@Lica04:~/file$ echo $?
0
```

**Verifichiamone le informazioni del file FF2:**

```
soELab@lica04:~/file$ ls -l FF2
-rw-r--r-- 1 soELab users 24 Apr 21 11:38 FF2
soELab@lica04:~/file$ cat FF2
Sono il file di
nome F1
```

**2) errato**

```
soELab@Lica04:~/file$ ridir-constampesudev pippo pap
Ci sono stati degli errori durante la copia
soELab@Lica04:~/file$ echo $?
2
```

**(Luc. C/Unix Primitive per i file 20): la primitiva lseek.**

**Vediamo ora degli esempi di uso della primitiva lseek che consente di agire sul file pointer.**

**(saltato Luc. C/Unix Primitive per i file 21)**

**(Luc. C/Unix Primitive per i file 22)**

**Per prima cosa vediamo come si può fare per aggiungere delle informazioni in un file se questo esiste, o crearlo se questo non esiste:**

```
soELab@Lica04:~/file$ cat append.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define PERM 0644               /* in UNIX */

int appendfile(char *f1)
{  int outfile, nread;
   char buffer [BUFSIZ];
if ((outfile = open( f1, O_WRONLY)) < 0)
        /* apertura in scrittura */
        { if ((outfile = creat(f1, PERM)) <0)
```

```
                /* se il file non esiste, viene creato */
                        return 1;       }
else lseek (outfile, 0L, 2);
```

**<== Nota bene: si deve usare il valore 0 indicato come LONG INTEGER e quindi 0L! L'origine è 2 cioè SEEK_END!**

```
        /* se il file esiste, ci si posiziona alla fine */
while ((nread = read(0, buffer, BUFSIZ)) > 0)
        /* si legge dallo standard input */
{   if (write (outfile, buffer, nread ) < nread)
        { close(outfile); return 2; /* errore */ }
}/* fine del file di input */
close(outfile); return 0;
}

int main (int argc, char ** argv)
{ int integi;
if (argc != 2) /* controllo sul numero di argomenti */
{ printf ("ERRORE: ci vuole un argomento \n"); exit (3); }
integi = appendfile(argv[1]);
exit(integi);
}
```

**Vediamo il funzionamento:**

**1) caso senza parametri: errore**

```
soELab@Lica04:~/file$ append
ERRORE: ci vuole un argomento
soELab@Lica04:~/file$ echo $?
3
```

**2) caso un parametro: file esistente**

```
soELab@Lica04:~/file$ cat F5
Sono il file
di nome F5 e servo come prova per
append.
soELab@Lica04:~/file$ append F5
ecco adesso aggiungiamo
qualche linea
```

**<== Nota bene: si deve usare la combinazione di tasti ^D per terminare la fase di input**

**Verifichiamo il contenuto del file F5:**

```
soELab@Lica04:~/file$ cat F5
Sono il file
di nome F5 e servo come prova per
append.
ecco adesso aggiungiamo
qualche linea
```

**3) caso un parametro: file NON esistente**

```
soELab@Lica04:~/file$ ls F6
ls: cannot access F6: No such file or directory
soELab@Lica04:~/file$ append F6
questo file non
esisteva.
Ma ora sì!
```

**<== Nota bene: si deve usare la combinazione di tasti ^D per terminare la fase di input**

```
soELab@Lica04:~/file$ ls -l F6
-rw-r--r-- 1 soELab users 38 Apr 21 11:59 F6
soELab@Lica04:~/file$ cat F6
questo file non
esisteva.
Ma ora sì!
```

**4) caso un parametro e utilizzando la ridirezione**

```
soELab@Lica04:~/file$ append F6 < F1
soELab@Lica04:~/file$ ls -l F6
-rw-r--r-- 1 soELab users 62 Apr 21 18:52 F6
soELab@Lica04:~/file$ cat F6
questo file non
esisteva.
Ma ora sì!
Sono il file di
nome F1
```

**Notiamo che questo programma, di fatto, ci fa capire come viene implementata la ridirezione in append dalla shell!**

**Vediamo un altro esempio di uso di lseek: sostituzione di caratteri all'interno di un file ==> usare directory ~/file/22SETT99**
**Leggiamo la specifica nel commento inserito all'inizio del programma C:**

```
soELab@Lica04:~/file/22SETT99$ cat 22sett99.c
/*
Si progetti, utilizzando il linguaggio C e le primitive di basso livello che operano sui file, un
filtro che accetta due parametri: il primo parametro deve essere il nome di un file (F), mentre il
secondo deve essere considerato un singolo carattere (C).  Il filtro deve operare una modifica del
contenuto del file F: in particolare, tutte le occorrenze del carattere C nel file F devono essere
sostituite con il carattere spazio.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv)
{       int fd;
        char c;
```

**<== usiamo un singolo carattere dato che faremo una lettura carattere per carattere, dato che dobbiamo verificare se abbiamo trovato il carattere cercato!**

```
if (argc != 3) { puts("Errore nel numero di parametri");
                 exit(1); }

if ((fd = open(argv[1], O_RDWR)) < 0)
```

**<== NOTA BENE: l'apertura la dobbiamo fare in LETTURA E SCRITTURA, dato che dobbiamo leggere per cercare il carattere e quindi dobbiamo poi scrivere!**

```
                { puts("Errore in apertura file");
                  exit(2); }

if (strlen(argv[2]) != 1)
```

**<== controllo che il secondo parametro sia una stringa di lunghezza 1 (quindi che contenga un singolo carattere); in alternativa si poteva verificare che argv[2][1] fosse o meno != '\0' cioè il carattere nullo!**

```
                { puts("Errore non carattere"); exit(3); }

while (read(fd, &c, 1))
```

**<== lettura di un singolo carattere alla volta; N.B. il controllo sul fatto se siamo arrivati all'end-of-file logico del file viene effettuato da UNIX sulla base del valore del file pointer (che si trova nell'elemento della Tabella dei File Aperti di Sistema, riferito dall'elemento di posto fd della Tabella dei File Aperti del processo) rispetto alla lunghezza del file (che si trova nella copia dell'i-node caricato nella tabella degli INODE Attivi di Sistema)**

```
        if (c == argv[2][0])
              { lseek(fd, -1L, 1);
            /* SI DEVE RIPORTARE INDIETRO IL FILE POINTER */
                  write(fd, " ", 1);
                }
return 0;
}
```

**<== ATTENZIONE: argv[2][0] è il carattere che dobbiamo cercare**

**<== NOTA BENE: quando verifichiamo che abbiamo trovato il carattere il file pointer è già avanzato ed è sul carattere successivo e quindi dobbiamo tornare indietro di 1 (-1L) rispetto alla posizione corrente (1 o SEEK_CUR)!**

**<== scrittura di un singolo carattere che si trova all'inizio del buffer di memoria costituito dalla stringa che contiene un singolo carattere spazio/blank!**

**Verifichiamone il funzionamento:**

**1) invocazione scorretta, senza parametri:**

```
soELab@Lica04:~/file/22SETT99$ 22sett99
Errore nel numero di parametri
soELab@Lica04:~/file/22SETT99$ echo $?
1
```

**2) invocazione scorretta, primo parametro file NON esistente:**

```
soELab@Lica04:~/file/22SETT99$ 22sett99 pippo p
Errore in apertura file
soELab@Lica04:~/file/22SETT99$ echo $?
2
```

**3) invocazione scorretta, secondo parametro non singolo carattere:**

```
soELab@Lica04:~/file/22SETT99$ 22sett99 prova pippo
Errore non carattere
soELab@Lica04:~/file/22SETT99$ echo $?
3
```

**4) invocazione corretta, controlliamo contenuto file prova prima e dopo l'esecuzione:**

```
soELab@lica04:~/file/22SETT99$ ls -l prova
-rw-r--r-- 1 soELab users 102 Apr 20 18:35 prova
soELab@Lica04:~/file/22SETT99$ cat prova
Sono il file prova, e ho al mio interno
alcune virgole, il programma
22sett99 me le togliera' tutte
soELab@Lica04:~/file/22SETT99$ 22sett99 prova ,
```

**<== passiamo come secondo parametro la stringa "," che contiene un singolo carattere!**

```
soELab@lica04:~/file/22SETT99$ ls -l prova
-rw-r--r-- 1 soELab users 102 Apr 21 12:28 prova
soELab@Lica04:~/file/22SETT99$ cat prova
Sono il file prova  e ho al mio interno
alcune virgole  il programma
22sett99 me le togliera' tutte
```

**<== il carattere ‘,’ è stato sostituito dal carattere ‘ ’ (spazio/blank!)**
**<== il carattere ‘,’ è stato sostituito dal carattere ‘ ’ (spazio/blank!)**

**5) altra invocazione corretta e poi controlliamo contenuto file prova dopo l’esecuzione:**

```
soELab@Lica04:~/file/22SETT99$ 22sett99 prova m
soELab@Lica04:~/file/22SETT99$ cat prova
Sono il file prova  e ho al  io interno
alcune virgole  il progra  a
22sett99  e le togliera' tutte
```

**<== il carattere ‘m’ è stato sostituito dal carattere ‘ ’ (spazio/blank!)**
**<== i caratteri ‘,’ sono stati sostituiti dal carattere ‘ ’ (spazio/blank!)**
**<== il carattere ‘m’ è stato sostituito dal carattere ‘ ’ (spazio/blank!)**

**NOTA BENE: la dimensione del file prova non cambia:**

```
soELab@lica04:~/file/22SETT99$ ls -l prova
-rw-r--r-- 1 soELab users 102 Apr 21 12:30 prova
```

**(Luc. C/Unix Primitive per i file 8, che avevamo saltato)**
**Vediamo ora degli esempi di uso della primitiva open con 3 parametri ==> usare directory ~/file/openavanzate**
**Primo esempio: uso di open per creare un file, uso di open per creare un file solo se non esiste, uso di open per troncare un file**

```
soELab@Lica04:~/file/openavanzate$ cat proveopenavanzate.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#define PERM 0644               /* in UNIX */

int main (int argc, char **argv)
{
int fd; /* usiamo una sola variabile, tanto non ci serve agire contemporaneamente sui diversi file */

/* passiamo un solo parametro che ci servira’ per identificare se e’ la prima volta che invochiamo
questo programma */
if (argc != 2) { puts("Errore nel numero di parametri");
                 exit(1); }
```

```
if ( (fd= open ("pippo", O_CREAT | O_WRONLY, PERM)) < 0 )
        /* apertura in creazione */
        {
        puts("Errore in creazione pippo");
        exit(1);
        }
else
        puts("Creato il file pippo");
/* scriviamo nel file appena creato distinguendo se e' la prima volta o la seconda volta che
invochiamo il programma */
if (strcmp(argv[1], "prima") == 0)
        write(fd, "questa e' la prima volta che scriviamo sul file\n", 48);
else
        write(fd, "seconda volta che scriviamo su file\n", 36);

if ( (fd= open ("paperino", O_CREAT | O_EXCL | O_WRONLY, PERM)) < 0 )
        /* apertura in creazione solo se non esiste */
        {
        puts("Il file paperino esiste"); <== N.B. non terminiamo perché NON è un errore!
        }
else
        {
        puts("Il file paperino non esisteva: creato");
        write(fd, "questa e' la prima volta che scriviamo sul file\n", 48);
        }

if ( (fd= open ("paperina", O_TRUNC | O_WRONLY)) < 0 )
        /* apertura distruggendo il contenuto precedente */
        {
        puts("Il file paperina non esiste");
        exit(2);
        }
else
         {
        puts("Il file paperina esisteva: troncato");
```

```
        if (strcmp(argv[1], "prima") == 0)
                write(fd, "questa e' la prima volta che scriviamo sul file\n", 48);
        else
                write(fd, "seconda volta che scriviamo su file\n", 36);
        }
return 0;
}
```

**Verifichiamone il funzionamento, dopo aver verificato che non esistono file di nome pippo e paperino, ma esiste invece il file paperina:**

```
soELab@Lica04:~/file/openavanzate$ ls -l pippo paperino paperina
ls: cannot access 'pippo': No such file or directory
ls: cannot access 'paperino': No such file or directory
-rw-r--r-- 1 soELab users   54 Apr 12 18:25 paperina
soELab@lica04:~/file/openavanzate$ cat paperina
Sono il file
paperina
servo per vedere il troncamento
soELab@Lica04:~/file/openavanzate$ proveopenavanzate prima
Creato il file pippo
Il file paperino non esisteva: creato
Il file paperina esisteva: troncato
soELab@Lica04:~/file/openavanzate$ ls -l pippo paperino paperina
-rw-r--r-- 1 soELab users   48 Apr 21 12:47 paperina
-rw-r--r-- 1 soELab users   48 Apr 21 18:47 paperino
-rw-r--r-- 1 soELab users   48 Apr 21 18:47 pippo
```

**<== passiamo la stringa "prima"**

**Vediamo cosa c'è dentro ai file (che sono tutti della stessa lunghezza!):**

```
soELab@Lica04:~/file/openavanzate$ more pippo paperino paperina
::::::::::::::
pippo
::::::::::::::
questa e' la prima volta che scriviamo sul file
::::::::::::::
paperino
::::::::::::::
questa e' la prima volta che scriviamo sul file
::::::::::::::
```

```
paperina
::::::::::::::
questa e' la prima volta che scriviamo sul file
```

**Invochiamolo una seconda volta:**

```
soELab@Lica04:~/file/openavanzate$ proveopenavanzate seconda
```
**<== N.B. va bene qualunque stringa diversa da "prima"**

```
Creato il file pippo
Il file paperino esiste
Il file paperina esisteva: troncato
soELab@Lica04:~/file/openavanzate$ ls -l pippo paperino paperina
-rw-r--r-- 1 soELab users   36 Apr 12 18:36 paperina
-rw-r--r-- 1 soELab users   49 Apr 12 18:34 paperino
-rw-r--r-- 1 soELab users   49 Apr 12 18:36 pippo
```
**<== N.B. paperina ha variato la dimensione (e data)**
**<== N.B. paperino è rimasto uguale**
**<== N.B. la data di pippo è cambiata, ma non la dimensione!**

**Vediamo cosa c'è dentro ai file:**

```
soELab@Lica04:~/file/openavanzate$ more pippo paperino paperina
::::::::::::::
pippo
```
**<== N.B. il contenuto di pippo è stato parzialmente sovrascritto!**

```
::::::::::::::
seconda volta che scriviamo su file
mo sul file
```
**<== questo è un pezzo del contenuto precedente!**

```
::::::::::::::
paperino
```
**<== N.B: Il file paperino non risulta alterato!**

```
::::::::::::::
questa e' la prima volta che scriviamo sul file
::::::::::::::
paperina
```
**<== N.B: Il file paperina risulta sovrascritto completamente!**

```
::::::::::::::
seconda volta che scriviamo su file
```

**(Luc. Unix. Tabelle per l'interazione con i file 8-10)**
**Illustrato brevemente il codice del Kenel di Linux v. 2.0 relativo alle tabelle di interazione con i file.**

**(Luc. C/Unix Primitive per i file 23)**
**Illustrato concetto di atomicità dell'effetto della singola primitiva read/write su un file se acceduto da processi diversi.**

**(fine lezione di mercoledì 21/04/2021)**

Lezione Lunedì 26/04/2021 ➔ corrisponde a due video-registrazioni

**(Luc. Unix: azioni primitive per gestione processi 1-11 e rivisto anche diagramma generale stati di un processo del Luc. 9 SOIntrod.pdf)**

**Mostrato applicazione scaricabile da http://www.didattica.agentgroup.unimo.it/didattica/TesiSOeLab/Sentimenti/UnixFunctionHelper.jar e in particolare funzionamento della primitiva fork().**
**(Saltate per ora Luc. Unix: azioni primitive per gestione processi 12 e 13)**

**(Luc. Unix: azioni primitive per gestione processi 14-15 sulla condivisione file e I/O pointer e 16-17 solo sulla condivisione file).**

**Vediamo ora i primi esempi semplici di funzionamento della primitiva fork() per la creazione di un processo figlio.**
**Controllare cosa dice il man di fork**

```
FORK(2)                    Linux Programmer's Manual                    FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
fork() creates a new process by duplicating the calling process.  The new process, referred to as the
child, is an exact duplicate of the calling process, referred to as the parent …
```

**OSSERVAZIONE il tipo pid_t nei sistemi Linux non è altro che una sorta di alias del tipo int.**

**1) primo esempio molto semplice**

```
soELab@Lica04:~/processi/PrimiEsempi$ cat unodue.c
/* FILE: unodue.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main ()
{
printf("UNO\n");
```
**<== eseguita dal processo che dopo chiameremo processo padre!**

```
fork();
```
**<== da qui in poi (se la primitiva ha successo, N.B. non l'abbiamo controllato) abbiamo due processi e non più un solo processo: il processo padre e il processo figlio!**

```
printf("DUE\n");
```
**<== eseguita da entrambi i processi (padre e figlio) dato che il codice è condiviso!**

```
exit(0);
```
**<== eseguita da entrambi i processi (padre e figlio) dato che il codice è condiviso!**

```
}
```

**Prima di vedere come funziona 'in diretta' vediamo su questo esempio i descrittori e gli spazi di indirizzamento dei processi padre e figlio e proviamo ad ipotizzare il possibile funzionamento:**
**http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/figuraUNODUE.mp4**
**Ora vediamo in effetti come funziona:**

```
soELab@Lica04:~/processi/PrimiEsempi$ unodue
UNO
DUE
DUE
soELab@Lica04:~/processi/PrimiEsempi$ unodue
UNO
DUE
soELab@Lica04:~/processi/PrimiEsempi$ DUE
```

**Nota Bene: può capitare che le write su standard output si mescolino a causa della condivisione dell'I/O pointer fra vari processi in parentela: la write fatta dal processo figlio rispetto alla write fatta dal processo di shell (cioè la stampa del prompt dei comandi, sottolineato per maggior chiarezza)!**

**La spiegazione si trova nei Luc. Unix: azioni primitive per gestione processi 14 e 15: ancora dal man di fork**

- The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see open(2)) as the corresponding file descriptor in the parent. This means that the two descriptors share open file status flags, current file offset, ...

**2) (Luc. Unix: azioni primitive per gestione processi 19)**
**per capire quale processo stampa la seconda riga con la scritta "DUE\n" e quale la terza riga, andiamo a riportare sullo standard input anche il pid del processo padre e del processo figlio, oltre che UID e GID (reali)**

```
soELab@Lica04:~/processi/PrimiEsempi$ cat unodueConPID-UID-GID.c
/* FILE: unodueConPID-UID-GID.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{
printf("UNO. PID = %d, UID = %d, GID = %d\n", getpid(), getuid(), getgid());

fork();
```

**<== N.B. non controlla il valore di ritorno della fork e quindi non controlliamo se la se la primitiva ha avuto o meno successo**

```
printf("DUE. PID = %d, UID = %d, GID = %d\n", getpid(), getuid(), getgid());
exit(0);
}
```

**Quindi, vediamo come funziona:**

```
soELab@Lica04:~/processi/PrimiEsempi$ unodueConPID-UID-GID
UNO. PID = 10685, UID = 1003, GID = 100
DUE. PID = 10685, UID = 1003, GID = 100
DUE. PID = 10686, UID = 1003, GID = 100
soELab@Lica04:~/processi/PrimiEsempi$ unodueConPID-UID-GID
UNO. PID = 10694, UID = 1004, GID = 100
DUE. PID = 10694, UID = 1004, GID = 100
soELab@Lica04:~/processi/PrimiEsempi$ DUE. PID = 10695, UID = 1004, GID = 100
```

**<== scritta dal padre!**
**<== scritta dal padre!**
**<== scritta dal figlio!**
**<== mescolamento come prima!**

**3) (Luc. Unix: azioni primitive per gestione processi 12, precedentemente saltata); ancora dal man di fork**

*RETURN VALUE*
*On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.*

**Ora usiamo anche il valore di ritorno della fork() sia per controllare che la fork sia andata a buon fine e sia per differenziare il comportamento del processo figlio da quello del comportamento del processo padre (eseguono sezioni diverse del codice condiviso, dato che non ha molto senso creare un processo che faccia le stesse cose del processo padre!):**

```
soELab@Lica04:~/processi/PrimiEsempi$ cat unoEdue.c
/* FILE: unoEdue.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{
int pid;

printf("UNO\n");                    <== a questo punto del programma avremo solo il processo padre!

if ((pid = fork()) < 0)  <== ATTENZIONE ALLE PARENTESI! Si controlla se per caso la fork è fallita (valore di ritorno -1)
{ /* fork fallita */ printf("Errore in fork\n"); exit(1);
}

if (pid == 0)
        printf("DUE\n");   /* figlio */      <== questa sezione la esegue solo il processo figlio!
else
        printf("Ho creato figlio con PID = %d\n", pid); /* padre */ <== questa sezione la esegue solo il processo
padre!
exit(0);                                    <== questa sezione la eseguono entrambi i processi (padre e figlio)!
}
```

**Vediamo come funziona:**

```
soELab@Lica04:~/processi/PrimiEsempi$ unoEdue
UNO                                         <== scritta dal padre (esiste solo lui)!
Ho creato figlio con PID = 10804            <== scritta dal padre!
DUE                                         <== scritta dal figlio!
soELab@Lica04:~/processi/PrimiEsempi$ unoEdue
UNO
Ho creato figlio con PID = 10814
soELab@Lica04:~/processi/PrimiEsempi$ DUE   <== mescolamento (sempre come prima)!
```

**(Luc. Unix: azioni primitive per gestione processi 13, precedentemente saltata): osservazioni su fork.**

**(fine lezione di lundì 26/04/2021)**