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

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);
       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.
```

<== questa versione la vedremo poi!

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

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

if (argc != 2) { puts("Errore nel numero di parametri");
                 exit(1); }
if ((fd1 = open(argv[1], O_RDONLY)) < 0)
                { puts("Errore in apertura file");
                  exit(2); }
else
        printf("Valore di fd1 = %d\n", fd1);

if ((fd2 = open(argv[1], O_RDONLY)) < 0)
```

<== deve essere incluso per poter usare printf e puts
<== deve essere incluso per poter usare la primitiva exit
<== deve essere incluso per poter usare la primitiva close
<== deve essere incluso per poter usare le costanti per la open (O_RDONLY, O_WRONLY e O_RDWR)

<== 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

<== in caso di errore, usiamo la primitiva exit con un numero diverso ogni volta (in modo assolutamente analogo a quanto veniva fatto nella shell)!

<== 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)

<== stampiamo il valore ritornato dalla open!

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

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 )                    <== Si legge da  standard input!
/* lettura dallo standard input fino a che ci sono caratteri */
                write(1, buffer, nread);                          <== Si scrive su standard output!
/* scrittura sullo standard output dei caratteri letti */
return 0;
}
```

```
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)

Lezione Mercoledì 28/04/2021 ➔ corrisponde a due video-registrazioni

(Luc. Unix: azioni primitive per gestione processi 18) Fork in Linux con ottimizzazione.

(Luc. Unix: azioni primitive per gestione processi 20)
Un processo padre, dopo aver creato un processo figlio (o più processi figli), può decidere quando e se attendere la terminazione dei processi figli utilizzando la primitiva wait: dal man di wait

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

NAME
       wait, … - wait for process to change state
SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);
DESCRIPTION
The  wait()  system  call  suspends  execution  of  the  calling  process  until  one  of  its  children
terminates…
```

Questa primitiva ritorna un errore (valore -1) nel caso il padre non abbia figli da aspettare o non ne abbia più da aspettare: sempre dal man di wait:

```
RETURN VALUE
wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.
```

Per prima cosa vediamo un paio di casi di errori (LASCIATI DA GUARDARE COME ESERCIZI AGLI STUDENTI):

1) un processo usa la primitiva wait senza aver creato alcun processo:

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

int main ()
{
/* padre (!?!) */
```

```
if (wait((int *)0) < 0)
```
<== in questo caso usiamo la versione in cui al padre NON interessa il valore di ritorno del figlio (che comunque non è stato creato!)
```
{
printf("Errore in wait\n");
exit (1);
}
exit (0);
}
```
Vediamo come funziona:
```
soELab@Lica04:~/processi/PrimiEsempi$ padresenzafigli
Errore in wait
```
2) un processo padre crea un solo processo figlio e quindi usa la primitiva wait due volte e la seconda volta si evidenzia un errore:
```
soELab@Lica04:~/processi/PrimiEsempi$ cat padresenzafigli1.c
/* FILE: padresenzafigli1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main ()
{

int pid, pidFiglio; /* pid per fork e pidFiglio per wait */

if ((pid = fork()) < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{ /* figlio */
        printf("Esecuzione del figlio\n");
        sleep(4);       /* si simula con un ritardo di 4 secondi che il figlio faccia qualcosa! */
        exit(5);         /* si torna un valore che si suppone possa essere derivante dall'esecuzione
di un compito assegnato al figlio */
```

```
}

/* padre */
printf("Generato figlio con PID = %d\n", pid);
/* il padre aspetta il figlio disinteressandosi del valore della exit del figlio */
if ((pidFiglio=wait((int *)0)) < 0)
{
        printf("Errore in wait\n");
        exit(2);
}

if (pid == pidFiglio)
        printf("Terminato figlio con PID = %d\n", pidFiglio);
else
        {       /* problemi */
                printf("Il pid della wait non corrisponde al pid della fork!\n");
                exit(3);
        }

/* padre fa un'altra wait: MA NON CI SONO PIU' FIGLI DA ASPETTARE */
if (wait((int *)0) < 0)
{
        printf("Errore in wait\n");
        exit(4);
}

exit(0); /* N.B. Non si arrivera' mai qui perche' il padre uscira' con la exit(4)! */
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/PrimiEsempi$ padresenzafigli1
Generato figlio con PID = 26756          <== scritta dal padre!
Esecuzione del figlio                    <== scritta dal figlio!
Terminato figlio con PID = 26756         <== scritta dal padre!
Errore in wait                           <== scritta dal padre!
```

(ancora Luc. Unix: azioni primitive per gestione processi 20 e 21, secondo caso)
Vediamo ora alcuni esempi corretti di funzionamento della wait:
1) prima il padre non considera il valore di ritorno del padre (il codice è simile al precedente, senza la seconda wait fatta dal padre):

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

int main ()
{
int pid, pidFiglio; /* pid per fork e pidFiglio per wait */

if ((pid = fork()) < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{       /* figlio */
        printf("Esecuzione del figlio\n");
        sleep(4);       /* si simula con un ritardo di 4 secondi che il figlio faccia qualcosa! */
        exit(5);         /* si torna un valore che si suppone possa essere derivante dall'esecuzione
di un compito assegnato al figlio */ <== NOTA BENE: il figlio esegue una exit nel suo codice e quindi la sezione seguente di
codice non è necessario che specifichi l'else di questo if!
}

/* padre */ <== NOTA BENE: questa sezione di codice viene eseguita solo dal padre perché il il figlio esegue una exit nel suo codice!
printf("Generato figlio con PID = %d\n", pid);
/* il padre aspetta il figlio disinteressandosi del valore della exit del figlio */
if ((pidFiglio=wait((int *)0)) < 0)
{
        printf("Errore in wait\n");
```

```
        exit(2);
}

if (pid == pidFiglio)
        printf("Terminato figlio con PID = %d\n", pidFiglio);
else
        {       /* problemi */
                printf("Il pid della wait non corrisponde al pid della fork!\n");
                exit(3);
        }

exit(0);
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/PrimiEsempi$ provawait
Generato figlio con PID = 13111
Esecuzione del figlio
```

<== il figlio, dopo questa write su standard output, esegue una sleep e quindi si sospende; allo stesso tempo il padre si sospende a causa della wait in attesa della terminazione del figlio!

```
Terminato figlio con PID = 13111
```

<== il figlio dopo l'attesa indicata dalla sleep esegue la exit(5) e quindi termina; la sua terminazione sblocca dalla wait il padre che quindi può scrivere su standard output questa stringa.

(Luc. Unix: azioni primitive per gestione processi 22) Spiegato terminazione normale e anormale.
(Luc. Unix: azioni primitive per gestione processi ancora 20, 21, primo caso e 23)
Per la spiegazione del parametro di ouput della wait (status), del suo uso e della sua 'pulizia' si può fare riferimento a http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraWait.mp4
Proseguiamo negli esempi di funzionamento corretto della wait:
2a) il padre considera il valore di ritorno del padre (il codice è simile al precedente, ma il padre ricava a mano il valore tornato dal figlio con la wait, similitudine con l'uso della variabile $? della shell):

```
soELab@Lica04:~/processi/PrimiEsempi$ cat status1.c
/* FILE: status1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main ()
{

int pid, pidFiglio, status, exit_s; /* pid per fork, pidFiglio e status per wait, exit_s per
selezionare valore di uscita figlio */

if ((pid = fork()) < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{       /* figlio */
        printf("Esecuzione del figlio\n");
        sleep(4);       /* si simula con un ritardo di 4 secondi che il figlio faccia qualcosa! */
        exit(5);         /* si torna un valore che si suppone possa essere derivante dall'esecuzione
di un compito assegnato al figlio */
}

/* padre */
printf("Generato figlio con PID = %d\n", pid);

/* il padre aspetta il figlio in questo caso interessandosi del valore della exit del figlio */
if ((pidFiglio=wait(&status)) < 0)      <== si deve controllare sempre il valore di ritorno della wait!
{
        printf("Errore in wait\n");
        exit(2);
}

if (pid == pidFiglio)
        printf("Terminato figlio con PID = %d\n", pidFiglio);
else
        {       /* problemi */
                printf("Il pid della wait non corrisponde al pid della fork!\n");
```

```
                exit(3);
        }
```

```
if ((status & 0xFF) != 0)
```
<== si maschera tutto a parte gli 8 bit (il byte) meno significativo (0xFF in esadecimale ha 8 bit a 1 nella parte bassa)

```
    printf("Figlio terminato in modo involontario (cioe' anomalo)\n");
```
<== se negli 8 bit meno significativi NON abbiamo 0 allora vuol dire che il figlio è terminato in modo involontario cioè anomalo!

```
    else
    {
        /* selezione del byte "alto" */
        exit_s = status >> 8;
```
<== si eliminano gli 8 bit (il byte) meno significativo che abbiamo verificato valgono 0

```
        exit_s &= 0xFF;
```
<== di nuovo si maschera tutto a parte gli 8 bit (il byte) meno significativo che adesso sono il valore ritornato dal figlio con la exit, andando a cambiare valore a exit_s

```
        printf("Per il figlio %d lo stato di EXIT e` %d\n", pid, exit_s);
    }

exit(0);
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/PrimiEsempi$ status1
Generato figlio con PID = 13191
Esecuzione del figlio
```
<== come prima (la parte di codice del figlio NON è variata) il figlio esegue, dopo questa write su standard output, una sleep e quindi si sospende; allo stesso tempo il padre si sospende a causa della wait in attesa della terminazione del figlio!

```
Terminato figlio con PID = 13191
Per il figlio 13191 lo stato di EXIT e` 5
```
<== diversamente da prima (la parte di codice del padre è variata) il padre una volta risvegliato dalla wait, avendo passato un puntatore ad intero (&status) ha modo di recuperare nell'intero status, usando in modo opportuno delle istruzioni che agiscono sui singoli bit, il valore tornato dal figlio con la exit!

(Luc. Unix: azioni primitive per gestione processi ancora 21, in fondo)
LASCIATO DA GUARDARE COME ESERCIZIO AGLI STUDENTI:
2b) il padre considera il valore di ritorno del padre; il codice è simile al precedente, ma il padre invece di ricavare a mano il valore tornato dal figlio con la wait, usa delle opportune macro fornite a livello di libreria: attenzione si deve aggiungere un altro file di include rispetto a prima.
Riportiamo la spiegazione di due macro dal man di wait:

```
WIFEXITED(status)
returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by
returning from main().

WEXITSTATUS(status)
returns the exit status of the child. This consists of the least significant 8 bits of the
status argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a
return statement in main(). This macro should be employed only if WIFEXITED returned true.
```

Vediamo ora un programma che usa queste due macro:

```
soELab@Lica04:~/processi/PrimiEsempi$ cat status2.c
/* FILE: status2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main ()
{

int pid, pidFiglio, status; /* pid per fork, pidFiglio e status per wait */

if ((pid = fork()) < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{       /* figlio */
        printf("Esecuzione del figlio\n");
```

```
        sleep(4);       /* si simula con un ritardo di 4 secondi che il figlio faccia qualcosa! */
        exit(5);         /* si torna un valore che si suppone possa essere derivante dall'esecuzione
di un compito assegnato al figlio */
}

/* padre */
printf("Generato figlio con PID = %d\n", pid);

/* il padre aspetta il figlio in questo caso interessandosi del valore della exit del figlio */
if ((pidFiglio=wait(&status)) < 0)
{
        printf("Errore in wait\n");
        exit(2);
}

if (pid == pidFiglio)
        printf("Terminato figlio con PID = %d\n", pidFiglio);
else
        {       /* problemi */
                printf("Il pid della wait non corrisponde al pid della fork!\n");
                exit(3);
        }

if (WIFEXITED(status) == 0)
    printf("Figlio terminato in modo involontario (cioe' anomalo)\n");
    else
    {
        printf("Valore di WIFEXITED(status) %d\n", WIFEXITED(status)); <== stampa solo di controllo!
        printf("Per il figlio %d lo stato di EXIT e` %d\n", pid, WEXITSTATUS(status));
    }

exit(0);
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/PrimiEsempi$ status2
Generato figlio con PID = 13235
Esecuzione del figlio
Terminato figlio con PID = 13235
Valore di WIFEXITED(status) 1
Per il figlio 13235 lo stato di EXIT e` 5
```

2c) il padre considera il valore di ritorno del padre; il codice è simile al precedente, ma il figlio invece che ritornare con la exit un valore costante, ritorna un valore intero richiesto all'utente. Nota bene: se l'utente fornisce un valore maggiore di 255, il valore che riceve il padre risulterà troncato dato che il padre riesce a ricevere solo un valore rappresentabile in 8 bit!

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

int main ()
{

int pid, pidFiglio, status, exit_s;      /* pid per fork, pidFiglio e status per wait, exit_s per
selezionare valore di uscita figlio  */
int valore;                              /* valore per scanf da parte de figlio */

if ((pid = fork()) < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{       /* figlio */
        printf("Esecuzione del figlio\n");
        /* questa volta facciamo fare qualche cosa aò figlio */
        printf("Dammi un valore intero per provare la exit:\n");
```

```
        scanf("%d", &valore);
        if ((valore > 255) || (valore < 0)) printf("ATTENZIONE IL VALORE SARA' TRONCATO!\n");
        else printf("Il valore fornito non verra' troncato!\n");
        exit(valore);
}

/* padre */
printf("Generato figlio con PID = %d\n", pid);

/* il padre aspetta il figlio in questo caso interessandosi del valore della exit del figlio */
if ((pidFiglio=wait(&status)) < 0)
{
        printf("Errore in wait\n");
        exit(2);
}

if (pid == pidFiglio)
        printf("Terminato figlio con PID = %d\n", pidFiglio);
else
        {       /* problemi */
                printf("Il pid della wait non corrisponde al pid della fork!\n");
                exit(3);
        }

if ((status & 0xFF) != 0)
    printf("Figlio terminato in modo involontario (cioe' anomalo)\n");
    else
    {
        /* selezione del byte "alto" */
        exit_s = status >> 8;
        exit_s &= 0xFF;
        printf("Per il figlio %d lo stato di EXIT e` %d\n", pid, exit_s);
    }

exit(0);
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/PrimiEsempi$ provaValoriWait
Generato figlio con PID = 13276
Esecuzione del figlio
Dammi un valore intero per provare la exit:
10
Il valore fornito non verra' troncato!
Terminato figlio con PID = 13276
Per il figlio 13276 lo stato di EXIT e` 10
soELab@Lica04:~/processi/PrimiEsempi$ provaValoriWait
Generato figlio con PID = 13279
Esecuzione del figlio
Dammi un valore intero per provare la exit:
125
Il valore fornito non verra' troncato!
Terminato figlio con PID = 13279
Per il figlio 13279 lo stato di EXIT e` 125
soELab@Lica04:~/processi/PrimiEsempi$ provaValoriWait
Generato figlio con PID = 13281
Esecuzione del figlio
Dammi un valore intero per provare la exit:
255
Il valore fornito non verra' troncato!
Terminato figlio con PID = 13281
Per il figlio 13281 lo stato di EXIT e` 255
soELab@Lica04:~/processi/PrimiEsempi$ provaValoriWait
Generato figlio con PID = 13283
Esecuzione del figlio
Dammi un valore intero per provare la exit:
256
ATTENZIONE IL VALORE SARA' TRONCATO!
Terminato figlio con PID = 13283
Per il figlio 13283 lo stato di EXIT e` 0          <== ATTENZIONE VALORE EVIDENTEMENTE TRONCATO!
soELab@Lica04:~/processi/PrimiEsempi$ provaValoriWait
Generato figlio con PID = 13291
```

```
Esecuzione del figlio
Dammi un valore intero per provare la exit:
355
ATTENZIONE IL VALORE SARA' TRONCATO!
Terminato figlio con PID = 13291
Per il figlio 13291 lo stato di EXIT e` 99
soELab@Lica04:~/processi/PrimiEsempi$ provaValoriWait
Generato figlio con PID = 13293
Esecuzione del figlio
Dammi un valore intero per provare la exit:
-1
ATTENZIONE IL VALORE SARA' TRONCATO!
Terminato figlio con PID = 13293
Per il figlio 13293 lo stato di EXIT e` 255
```

Vedere ancora http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraWait.mp4 per la spiegazione dell'uso della wait nel processo shell padre (nei confronti del processo sotto-shell figlio) in caso di esecuzione in foreground, mentre in caso di esecuzione in background il processo padre NON attende la terminazione del processo sotto-shell figlio!

(Luc. Unix: azioni primitive per gestione processi 24 lasciato da guardare agli studenti)
(Luc. Unix: azioni primitive per gestione processi 25 e 26 per ora saltati)

(Luc. Unix: azioni primitive per gestione processi 27-30)
Mostrato di nuovo applicazione scaricabile da http://www.didattica.agentgroup.unimo.it/didattica/TesiSOeLab/Sentimenti/UnixFunctionHelper.jar e in particolare funzionamento di una delle primitive della famiglia exec.
Proseguiamo nel capire come la shell manda in esecuzione un comando/programma: dopo aver usato la primitiva fork (per creare un processo figlio), la shell si mette in attesa con la primitiva wait (nel caso di esecuzione di foreground) e il processo figlio (dopo aver operato tutte le sostituzioni e aver trattato la ridirezione mette in esecuzione il comando/programma con una delle primitive della famiglia exec.
<== usare directory ~/exec

Vediamo lo stesso problema (mettere in esecuzione il comando /bin/ls) risolto:

1) usando execv

```
2) soELab@Lica04:~/exec$ cat myls1.c
/* FILE: myls1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{
        char *av[3];    /* array di puntatori a char che serve per passare i parametri alla execv */

av[0]="ls";              /* av[0] e' un puntatore a char cui viene assegnato il puntatore alla stringa
"ls" */
av[1]="-l";             /* av[1] e' un puntatore a char cui viene assegnato il puntatore alla stringa "-
l" */
av[2]= (char *)0;       /* av[2] e' un puntatore a char cui viene assegnato il valore 0 come puntatore a
char */

printf("Esecuzione di ls: prima versione\n");
execv("/bin/ls", av);   /* controllare con which se ls sta in effetti in /bin */

/* si esegue l'istruzione seguente SOLO in caso di fallimento della execv */
printf("Errore in execv\n");
exit(1);        /* torniamo quindi un valore diverso da 0 per indicare che ci sono stati dei problemi */
}
```

Verifichiamo il funzionamento:

```
soELab@Lica04:~/exec$ myls1
Esecuzione di ls: prima versione
total 112
-rwxr-xr-x 1 soELab users 8392 Apr 19 15:59 callecho
-rw-r--r-- 1 soELab users  910 Apr 19 19:05 callecho.c
-rw-r--r-- 1 soELab users  201 Apr 19 15:33 makefile
-rwxr-xr-x 1 soELab users 8384 Apr 19 15:52 myecho
-rw-r--r-- 1 soELab users  257 Apr 19 15:51 myecho.c
-rwxr-xr-x 1 soELab users 8384 Apr 19 15:59 myls1
```

```
-rw-r--r-- 1 soELab users  826 Apr 19 19:06 myls1.c
-rwxr-xr-x 1 soELab users 8384 Apr 19 15:58 myls2
-rw-r--r-- 1 soELab users  394 Apr 19 19:06 myls2.c
-rwxr-xr-x 1 soELab users 8392 Apr 19 15:44 mylsErrato
-rw-r--r-- 1 soELab users  493 Apr 19 19:06 mylsErrato.c
drwxr-xr-x 2 soELab users 4096 Feb  3 13:01 old-sun
drwxr-xr-x 3 soELab users 4096 Apr 19 19:10 processi
-rwxr-xr-x 1 soELab users 8440 Apr 19 16:00 prova
-rw-r--r-- 1 soELab users  868 Apr 19 19:07 prova.c
-rw-r--r-- 1 soELab users  102 Feb 21  2019 temp
```

3) usando execl

```
soELab@Lica04:~/exec$ cat myls2.c
/* FILE: myls2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{
printf("Esecuzione di ls: seconda versione\n");
execl("/bin/ls", "ls", "-l", (char *)0);
```

<== passiamo direttamente i parametri (compreso il nome del programma stesso!)

```
/* si esegue l'istruzione seguente SOLO in caso di fallimento della execl */
printf("Errore in execl\n");
exit(1);        /* torniamo quindi un valore diverso da 0 per indicare che ci sono stati dei problemi */
}
```

Verifichiamo il funzionamento:

```
soELab@Lica04:~/exec$ myls2
Esecuzione di ls: seconda versione
total 112
-rwxr-xr-x 1 soELab users 8392 Apr 19 15:59 callecho
-rw-r--r-- 1 soELab users  910 Apr 19 19:05 callecho.c
-rw-r--r-- 1 soELab users  201 Apr 19 15:33 makefile
-rwxr-xr-x 1 soELab users 8384 Apr 19 15:52 myecho
-rw-r--r-- 1 soELab users  257 Apr 19 15:51 myecho.c
-rwxr-xr-x 1 soELab users 8384 Apr 19 15:59 myls1
```

```
-rw-r--r-- 1 soELab users  826 Apr 19 19:06 myls1.c
-rwxr-xr-x 1 soELab users 8384 Apr 19 15:58 myls2
-rw-r--r-- 1 soELab users  394 Apr 19 19:06 myls2.c
-rwxr-xr-x 1 soELab users 8392 Apr 19 15:44 mylsErrato
-rw-r--r-- 1 soELab users  493 Apr 19 19:06 mylsErrato.c
drwxr-xr-x 2 soELab users 4096 Feb  3 13:01 old-sun
drwxr-xr-x 3 soELab users 4096 Apr 19 19:10 processi
-rwxr-xr-x 1 soELab users 8440 Apr 19 16:00 prova
-rw-r--r-- 1 soELab users  868 Apr 19 19:07 prova.c
-rw-r--r-- 1 soELab users  102 Feb 21  2019 temp
```

Vediamo ora un caso di errore nell'uso di una delle primitive exec:

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

int main ()
{
printf("Esecuzione di ls: seconda versione\n");
execl("/bin/ls1", "ls", "-l", (char *)0);          /* abbiamo scritto il nome dell'eseguibile in modo
scorretto e quindi la execl fallira' */

/* si esegue l'istruzione seguente SOLO in caso di fallimento della execl */
printf("Errore in execl\n");
exit(1);          /* torniamo quindi un valore diverso da 0 per indicare che ci sono stati dei problemi
*/
}
```

Verifichiamo il funzionamento:

```
soELab@Lica04:~/exec$ mylsErrato
Esecuzione di ls: seconda versione
Errore in execl    <== La exec fallisce e quindi viene eseguita l'istruzione seguente, che in caso di successo NON viene mai eseguita!
soELab@Lica04:~/exec$ echo $?    <== Lo verifichiamo stampando il valore di $?
1
```

Cambiamo ora problema e proviamo il funzionamento delle primitive con la p ad esempio execvp; il programma callecho mette in esecuzione il programma myecho:

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

int main ()
{
        char *argin[4]; /* array di puntatori a char che serve per passare i parametri alla execvp */

argin[0]="myecho";        /* argin[0] e' un puntatore a char cui viene assegnato il puntatore alla
stringa "myecho" */
argin[1]="hello";         /* argin[1] e' un puntatore a char cui viene assegnato il puntatore alla
stringa "hello" */
argin[2]="world!";        /* argin[2] e' un puntatore a char cui viene assegnato il puntatore alla
stringa "world!" */
argin[3]=(char *)0;       /* argin[3] e' un puntatore a char cui viene assegnato il valore 0 come
puntatore a char */

printf("Esecuzione di myecho\n");
execvp(argin[0], argin);

/* si esegue l'istruzione seguente SOLO in caso di fallimento della execvp */
printf("Errore in execvp\n");
exit(1);            /* torniamo quindi un valore diverso da 0 per indicare che ci sono stati dei
problemi */
}
```

Il programma myecho deriva dal sorgente myecho.c:

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

```
int main (int argc, char **argv)
{
        int i;  /* indice per for */

printf("Sono myecho\n");
for (i=0; i < argc; i++)
        printf("Argomento argv[%d]= %s\n", i, argv[i]);
exit(0);
}
```

Verifichiamo il funzionamento, precisando che sia il programma callecho che myecho si trovano nella stessa directory e che il valore della variabile PATH per l'utente corrente contiene il direttorio corrente, dato che:

```
soELab@Lica04:~/exec$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games: /snap/bin:.
soELab@Lica04:~/exec$ callecho
Esecuzione di myecho
Sono  myecho
Argomento argv[0]= myecho
Argomento argv[1]= hello
Argomento argv[2]= world!
```

Chiaramente le primitive exec consentono anche di mettere in esecuzione lo stesso programma che si sta eseguendo; vediamolo in un esempio (LASCIATO DA GUARDARE AGLI STUDENTI):

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

int main ()
{
        char *argin[2]; /* array di puntatori a char che serve per passare i parametri alla execvp */
        int x;          /* per leggere valore dallo standard input */

argin[0]="prova";         /* argin[0] e' un puntatore a char cui viene assegnato il puntatore alla
stringa "prova" */
```

```
argin[1]= (char *)0;     /* argin[1] e' un puntatore a char cui viene assegnato il valore 0 come
puntatore a char */

printf("Esecuzione di prova\n");
printf("Dimmi se vuoi finire (valore 0)!\n");
scanf("%d", &x);
if (x != 0)
{
        execvp(argin[0], argin);

        /* si esegue l'istruzione seguente SOLO in caso di fallimento della execvp */
        printf("Errore in execvp\n");
        exit(1);        /* torniamo quindi un valore diverso da 0 per indicare che ci sono stati dei
problemi */
}
else exit(x);   /* notare che x sara' 0! */
}
```

Verifichiamo il funzionamento:

```
soELab@Lica04:~/exec$ prova
Esecuzione di prova
Dimmi se vuoi finire (valore 0)!
3
Esecuzione di prova
Dimmi se vuoi finire (valore 0)!
7
Esecuzione di prova
Dimmi se vuoi finire (valore 0)!
3
Esecuzione di prova
Dimmi se vuoi finire (valore 0)!
0
soELab@Lica04:~/exec$ echo $?
0
```

(Luc. Unix: azioni primitive per gestione processi 32-33)
Consideriamo di voler simulare ancora di più il comportamento della shell e quindi non solo usiamo una delle primitive exec, ma anche la primitiva fork (come esempio usiamo ancora il comando ls):
<== usare directory ~/exec/processi

```
soELab@Lica04:~/exec/processi$ cat mylsConFork.c
/* FILE: mylsConFork.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main ()
{
        int pid;                                /* per fork */
        int pidFiglio, status, ritorno;         /* per wait padre */

/* generiamo un processo figlio dato che stiamo simulando di essere il processo di shell! */
pid = fork();
```

<== il nostro programma sta simulando una shell e quindi per prima cosa crea un figlio per eseguire il comando (non interno che abbiamo scelto e cioè ls)

```
if (pid < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{       /* figlio */
        printf("Esecuzione di ls da parte del figlio con pid %d\n", getpid());
        execlp("ls", "ls", "-l", (char *)0); /* il processo sotto-shell usa sempre la versione con p!
*/
```

<== il processo figlio si trasforma in ls (nuova area utente: dati e codice)

```
        /* si esegue l'istruzione seguente SOLO in caso di fallimento della execlp */
        printf("Errore in execlp\n");
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi identificato
come errore */
```

```
}

/* padre aspetta subito il figlio appunto perche' deve simulare la shell e la esecuzione in foreground! */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(2);
}
if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);
}

exit(0);
}
```

<== **N.B. in questo caso le operazioni di pulizia di status vengono fatte con una singola istruzione!**

Verifichiamo il funzionamento:

```
soELab@Lica04:~/exec/processi$ mylsConFork
Esecuzione di ls
total 180
-rw-r--r-- 1 soELab users 8712 Apr 16  2019 comando
-rwxr-xr-x 1 soELab users 9043 May 15  2018 comando-bis
-rw-r--r-- 1 soELab users 1151 Feb 21  2019 comando.c
-rwxr-xr-x 1 soELab users 8424 Apr 19 19:00 leggi
-rw-r--r-- 1 soELab users  308 Apr 19 19:09 leggi.c
-rwxr-xr-x 1 soELab users 8922 Feb 21  2019 leggiPippo
-rw-r--r-- 1 soELab users  521 Feb 21  2019 leggiPippo.c
-rwxr-xr-x 1 soELab users 8922 Feb 21  2019 leggiPippo1
-rw-r--r-- 1 soELab users  201 Apr 19 16:03 makefile
-rwxr-xr-x 1 soELab users 8648 Apr 19 16:17 myGrepConFork
```

```
-rw-r--r-- 1 soELab users 1664 Apr 19 19:05 myGrepConFork.c
-rwxr-xr-x 1 soELab users 8568 Apr 19 16:17 mylsConFork
-rw-r--r-- 1 soELab users 1292 Apr 19 19:04 mylsConFork.c
-rwxr-xr-x 1 soELab users 8600 Apr 19 16:23 myopen
-rw-r--r-- 1 soELab users 1587 Apr 19 19:04 myopen.c
-rw-r--r-- 1 soELab users   58 Feb 21  2019 pippo
-rwxr-xr-x 1 soELab users 8661 Feb 21  2019 prova1
-rw-r--r-- 1 soELab users  209 Feb 21  2019 prova1.c
drwxr-xr-x 2 soELab users 4096 Feb  3 13:01 scarti
-rwxr-xr-x 1 soELab users 8600 Apr 17  2019 suid
-rw-r--r-- 1 soELab users 1128 Apr 17  2019 suid.c
-rwxr-xr-x 1 soELab users 8600 Apr 17  2019 suid1
-rw-r--r-- 1 soELab users 1119 Apr 17  2019 suid1.c
Il figlio con pid=15296 ha ritornato 0 (se 255 problemi!)
```

Consideriamo un altro esempio di simulazione del comportamento della shell, usando fork ed exec con il comando grep):
<== usare directory ~/exec/processi

```
soELab@Lica04:~/exec/processi$ cat myGrepConFork.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

/* FILE: myGrepConFork.c */
int main (int argc, char** argv)
{
        int pid;                                /* per fork */
        int pidFiglio, status, ritorno;         /* per wait padre */

if (argc != 3)
{
       printf("Errore nel numero di parametri che devono essere due (stringa da cercare e nome del file
dove cercare): %d\n", argc);
       exit(1);
```

```
}

/* generiamo un processo figlio dato che stiamo simulando di essere il processo di shell! */
pid = fork();
if (pid < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(2);
}

if (pid == 0)
{       /* figlio */
        printf("Esecuzione di grep da parte del figlio con pid %d\n", getpid());
        /* ridirezionamo lo standard output su /dev/null perche' ci interessa solo se il comando grep ha successo o meno */
        close(1);
        open("/dev/null", O_WRONLY);
        execlp("grep", "grep", argv[1], argv[2], (char *)0);

        /* si esegue l'istruzione seguente SOLO in caso di fallimento della execlp */
        /* ATTENZIONE SE LA EXEC FALLISCE NON HA SENSO FARE printf("Errore in execlp\n"); DATO CHE LO STANDARD OUTPUT E' RIDIRETTO SU /dev/null */
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi identificato come errore */
}

/* padre aspetta subito il figlio appunto perche' deve simulare la shell e la esecuzione in foreground! */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(3);
}
if ((status & 0xFF) != 0)
```

```
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);
}

exit (0);
}
```

Verifichiamo il funzionamento:

1) caso con numero di parametri errato

```
soELab@Lica04:~/exec/processi$ myGrepConFork
Errore nel numero di parametri che devono essere due (stringa da cercare e nome del file dove
cercare): 1
soELab@Lica04:~/exec/processi$ echo $?
1
```

2) caso di stringa presente nel file

```
soELab@Lica04:~/exec/processi$ cat pippo
Ciao.
Sono il file pippo.
Sono leggibile solo da soELab.

soELab@Lica04:~/exec/processi$ myGrepConFork il pippo
Esecuzione di grep da parte del figlio con pid 15362
Il figlio con pid=15362 ha ritornato 0 (se 255 problemi!)
```

3) caso di stringa NON presente nel file

```
soELab@Lica04:~/exec/processi$ myGrepConFork zz pippo
Esecuzione di grep da parte del figlio con pid 15387
Il figlio con pid=15387 ha ritornato 1 (se 255 problemi!)
```

4) caso di file NON esistente

```
soELab@Lica04:~/exec/processi$ myGrepConFork il pap
Esecuzione di grep da parte del figlio con pid 15395
grep: pap: No such file or directory                    <== viene scritto su standard error
Il figlio con pid=15395 ha ritornato 2 (se 255 problemi!)
```

Modificare il programma in modo che venga ridiretto anche lo standard error.

Modificare il programma in modo che venga ridiretto anche lo standard input in modo che venga direttamente letto il file corrispondente ad argv[2] e quindi passando un parametro in meno alla grep invocata tramite la primitiva execlp.

(Luc. Unix: azioni primitive per gestione processi 32)
Continuiamo nell'obiettivo di voler simulare il comportamento della shell e consideriamo anche la ridirezione: l'esempio mostrerà la ridirezione in input, viene lasciato come esercizio di provare a simulare anche la ridirezione in output; vogliamo simulare l'esecuzione del seguente comando: PROMPT$ prova < pippo, dove prova è un programma che sostanziamente riporta il contenuto dello standard input sullo standard output (una sorta di cat) e pippo è un file di testo.
<== usare sempre directory ~/exec/processi

```
soELab@Lica04:~/exec/processi$ cat myopen.c
/* FILE: myopen.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main ()
{
        int pid;                                /* per fork */
        int pidFiglio, status, ritorno;         /* per wait padre */

/* generiamo un processo figlio dato che stiamo simulando di essere il processo di shell! */
pid = fork();

if (pid < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
        { /* figlio */
        int fd;
```

```
        /* simuliamo ridirezione dello standard input */
        close(0);
        if ((fd = open("pippo", O_RDONLY)) < 0)
                {       puts("ERRORE in apertura");
                        exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi
identificato come errore */
                }
        printf("Ho aperto il file pippo con fd = %d\n", fd);
        printf("Esecuzione di programma che visualizza file gia` aperto\n");
        execl("leggi", "leggi", (char *)0);

        /* si esegue l'istruzione seguente SOLO in caso di fallimento della execlp */
        printf("Errore in execl\n");
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi identificato
come errore */
}

/* padre aspetta subito il figlio appunto perche' deve simulare la shell e la esecuzione in
foreground! */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(2);
}
if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi)\n", pidFiglio, ritorno);
}

exit(0);
}
```

Il programma leggi deriva dal sorgente leggi.c:

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

int main()
{
        char c; /* per leggere caratteri da standard input */

puts("SONO leggi");
puts("Sto per leggere da fd = 0 e scrivere su standard output");

while(read(0, &c, 1))
  write(1, &c, 1);

exit(0);
}
```

Verifichiamo il funzionamento ricordando che abbiamo già mostrato il contenuto del file pippo:

```
soELab@Lica04:~/exec/processi$ myopen
Ho aperto il file pippo con fd = 0                          <== scritto da my-open (figlio)
Esecuzione di programma che visualizza file gia` aperto     <== scritto da my-open (figlio)
SONO leggi                                                  <== scritto da leggi
Sto per leggere da fd = 0 e scrivere su standard output     <== scritto da leggi
Ciao.                                                       <== scritto da leggi
Sono il file pippo.                                         <== scritto da leggi
Sono leggibile solo da soELab.                              <== scritto da leggi
                                                            <== scritto da leggi
Il figlio con pid=15483 ha ritornato 0(se 255 problemi)     <== scritto da my-open (padre)
```

(fine lezione di mercoledì 28/04/2021)

Lezione Lunedì 3/05/2021 ➔ corrisponde a due video-registrazioni

(Luc. Unix: azioni primitive per gestione processi 31)
Riassunto su famiglia primitive exec.

(Luc. Unix: azioni primitive per gestione processi 33)
Ripreso discorso su fork ed exec e ricordato giustificato ottimizzazione di Linux (da Luc. Unix: azioni primitive per gestione processi 18)

(Ritorniamo alle slide sulla SHELL - Luc. 17)
Due dei 3 bit speciali per i file eseguibili (SUID e SGID)
All'interno del descrittore di processo UID reale e UID effettivo oltre che GID reale e GID effettivo: nei controlli sui diritti di accesso ai file/directory UNIX/Linux utilizza UID e GID effettivi!
Spiegazione anche tramite il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraSUID.mp4
Vediamo l'uso del SUID nel comando passwd:

```
soELab@Lica04:~/prime-prove-sh/suid$ which passwd
/usr/bin/passwd
soELab@Lica04:~/prime-prove-sh/suid$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 59640 Mar 22  2019 /usr/bin/passwd
```

<== NOTARE la lettera s nella tripletta del proprietario del file (che è root cioè il super-utente!) che significa che c'è il SUID settato.
Il comando passwd deve leggere/scrivere nel file /etc/shadow su cui ha l'accesso in scrittura e lettura solo l'utente root (super-utente) e quindi il SUID settato serve per fare in modo che quando un semplice utente chiama il comando passwd il processo che esegue passwd sia in grado di leggere e scrivere il file /etc/shadow, dato che che si ha:

```
soELab@Lica04:~/prime-prove-sh/suid$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1674 Mar 26 12:15 /etc/shadow
```

Vediamo ora il funzionamento delle primitive exec in presenza di set-user-id settato (analogo comportamento ci sarebbe con set-group-id settato)
<== usare sempre directory ~/exec/processi

```
soELab@Lica04:~/exec/processi$ cat suid.c
/* FILE suid.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main ()
{
        int pid;                                /* per fork */
        int pidFiglio, status, ritorno;         /* per wait padre */

/* generiamo un processo figlio dato che stiamo simulando di essere il processo di shell! */
pid = fork();

if (pid < 0)
{       /* fork fallita */
        printf("Errore in fork\n");
        exit(1);
}

if (pid == 0)
{       /* figlio */
        printf("real-user id = %d\n", getuid());
        printf("effective-user id = %d\n", geteuid());
        printf("Esecuzione di programma (con suid settato) che visualizza file (leggibile solo dal
proprietario)\n");
        execl("leggiPippo1", "leggiPippo1", (char *)0);
        printf("Errore in execl\n");
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi identificato
come errore */
}

/* padre aspetta subito il figlio appunto perche' deve simulare la shell e la esecuzione in
foreground! */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(2);
}
if ((status & 0xFF) != 0)
```

<== il programma leggiPippo1 deriva dal sorgente leggiPippo.c, ma avrà il set-user-id settato!

```
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);
}
exit (0);
}
```

Vediamo cosa fa il programma eseguibile leggiPippo1, che deriva dal seguente sorgente:

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

int main(int argc, char **argv)
{
        int fd; /* per la open */
        char c; /* per leggere i caratteri dal file pippo */

printf("real-user id = %d\n", getuid());
printf("effective-user id = %d\n", geteuid());
printf("SONO %s\n", argv[0]);
puts("Sto per leggere il file pippo");

if ((fd=open("pippo", O_RDONLY)) < 0)
{       puts("ERRORE in apertura");
        exit(1);
}

while(read(fd, &c, 1))
        write(1, &c, 1);

exit(0);
}
```

Verifichiamo i diritti di leggiPippo1 e leggiPippo:

```
soELab@Lica04:~/exec/processi$ ls -l leggiPippo*
-rwxr-xr-x 1 soELab users 8608 Apr 23 17:01 leggiPippo
-rw-r--r-- 1 soELab users  518 Apr 23 17:01 leggiPippo.c
-rwsr-xr-x 1 soELab users 8608 Apr 23 17:01 leggiPippo1
```

<== il programma leggiPippo1 è una copia di leggiPippo, ma gli è stato settato il set-user-id! Per farlo si deve usare il comando chmod u+s leggiPippo1

Verifichiamo i diritti di pippo e il suo contenuto:

```
soELab@Lica04:~/exec/processi$ ls -l pippo
-rw------- 1 soELab users 58 Jun 18  2014 pippo
```

<== il file pippo è leggibile (e scrivibile) solo dal proprietario!

```
soELab@lica04:~/exec/processi$ cat pippo
Ciao.
Sono il file pippo.
Sono leggibile solo da soELab.
```

Verifichiamo ancora una volta le informazioni su UID (e GID) dell'utente soELab:

```
soELab@lica04:~/exec/processi$ id
uid=1003(soELab) gid=100(users) groups=100(users)
```

Ora verifichiamone il funzionamento di suid da un altro account (sonod) con un altro terminale:

```
sonod@Lica04:/home/soELab/exec/processi$ cat pippo
cat: pippo: Permission denied
sonod@Lica04:/home/soELab/exec/processi$ ./suid
real-user id = 1002
effective-user id = 1002
Esecuzione di programma (con suid settato) che visualizza file (leggibile solo dal proprietario)
real-user id = 1002
effective-user id = 1003
SONO leggiPippo1
Sto per leggere il file pippo
Ciao.
Sono il file pippo.
Sono leggibile solo da soELab.

Il figlio con pid=25061 ha ritornato 0
```

<== il programma leggiPippo1 ha il set-user-id settato e quindi è cambiato l'effective UID

Passiamo ora a fare il controesempio, e quindi vediamo il programma suid-sbagliato.c che manda in esecuzione il programma leggiPippo (quello originale) che NON ha il set-user-id settato:

`soELab@Lica04:~/exec/processi$ cat suid-sbagliato.c` **<== questo programma è uguale a suid.c a parte che viene messo in esecuzione leggiPippo (e non leggiPippo1)**

```
/* FILE suid-sbagliato.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main ()
{
        int pid;        /* per fork */
        int pidFiglio, status, ritorno;         /* per wait padre */

/* generiamo un processo figlio dato che stiamo simulando di essere il processo di shell! */
pid = fork();

if (pid < 0)
{ /* fork fallita */
printf("Errore in fork\n");
exit(1);
}

if (pid == 0)
{ /* figlio */
        printf("real-user id = %d\n", getuid());
        printf("effective-user id = %d\n", geteuid());
        printf("Esecuzione di programma che tenta di visualizzare file (leggibile solo dal proprietario)\n");
        execl("leggiPippo", "leggiPippo", (char *)0);
        printf("Errore in execl\n");
        exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi identificato come errore */
```

<== si mando in esecuzione leggiPippo che NON ha il set-user-id settato!

```
}

/* padre aspetta subito il figlio appunto perche' deve simulare la shell e la esecuzione in
foreground! */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(2);
}
if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);
}
exit (0);
}
```

Verifichiamone sempre il funzionamento da un altro account (sonod), nel terminale usato precedentemente:

```
sonod@Lica04:/home/soELab/exec/processi$ ./suid-sbagliato
real-user id = 1002
effective-user id = 1002
Esecuzione di programma che tenta di visualizzare file (leggibile solo dal proprietario)
real-user id = 1002
effective-user id = 1002
SONO leggiPippo
Sto per leggere il file pippo
ERRORE in apertura
Il figlio con pid=25061 ha ritornato 1
```

<== il programma leggiPippo NON ha il set-user-id settato e quindi NON è cambiato l'effective UID

(Luc. Unix: azioni primitive per gestione processi 34 e 35, per ora saltati)

(Luc. Unix: azioni primitive per gestione processi 36 e 37)
Vediamo ora un programma che simula un 'embrione' di shell (sul sito è presente anche un programma molto più completo, chiamato smallsh.c che si trova sempre nella sezione small sh, ma che potrà essere compreso appieno solo alla fine delle lezioni); in questo programma usiamo la variabile errno, si veda man errno:

ERRNO(3) Linux Programmer's Manual ERRNO(3)

NAME
errno - number of last error

SYNOPSIS
#include <errno.h>

DESCRIPTION
The <errno.h> header file defines the integer variable errno, which is set by system calls and some library functions in the event of an error to indicate what went wrong. Its value is significant only when the return value of the call indicated an error (i.e., -1 from most system calls; -1 or NULL from most library functions); a function that succeeds is allowed to change errno.
Valid error numbers are all nonzero; errno is never set to zero by any system call or library function.
…

E la funzione perror, si veda man perror:

PERROR(3) Linux Programmer's Manual PERROR(3)

NAME
perror - print a system error message

SYNOPSIS
#include <stdio.h>

void perror(const char *s);
…

```
DESCRIPTION
The  routine  perror() produces a message on the standard error output, describing the last error
encountered during a call to a system or library function. First (if s is not NULL and *s is not a
null byte ('\0')) the argument string s is printed, followed by a colon and a blank.  Then the
message  and  a  new-line.
To  be  of most use, the argument string should include the name of the function that incurred the
error.  The error number is taken from the external variable errno, which is set when errors occur
but not cleared when successful calls are made.
```

…

Vediamo ora il codice di questo 'embrione' di una shell:

```
soELab@Lica04:~/exec/processi$ cat comando.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>

int main  (int argc, char **argv)
{
        int pid;                                /* per fork */
        int pidFiglio, status, ritorno;         /* per wait padre */
        char st[80];                            /* array di caratteri per leggere (da standard input)
il comando (SENZA OPZIONI O PARAMETRI) immesso dall'utente */

for (;;)        /* ciclo infinito: stiamo simulando una shell che deve restare sempre in attesa di un
comando */
{
        printf("inserire il comando da eseguire:\n");
        scanf("%s", st);
        /* una volta che la nostra shell simulata riceve un comando, delega un processo per eseguirlo
(come fa la shell!) */
        if ((pid = fork()) < 0)  { perror("fork"); exit(errno);}

        if (pid == 0)
```

```
        {       /*  FIGLIO: esegue i comandi */
                execlp(st, st, (char *)0);
                perror("errore esecuzione comando");
                exit(errno);
        }

        /* padre aspetta subito il figlio, dato che siamo simulando l'esecuzione in foreground */
        pidFiglio = wait(&status);
        if (pidFiglio < 0)
        {
                perror("errore wait");
                exit(errno);
        }
        if ((status & 0xFF) != 0)
                printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
        else
        {
                ritorno=(int)((status >> 8) & 0xFF);
                printf("Il figlio con pid=%d ha ritornato %d\n", pidFiglio, ritorno);
                /* se siamo arrivati qui vuole dire che tutto e' andato bene e quindi chiediamo
all'utente se si vuole proseguire: stiamo simulando il comando interno exit! */
                printf ("eseguire altro comando? (si/no) \n");
                scanf ("%s", st);
                if (strcmp(st, "si"))
                        exit(0); /* se l'utente non vuole proseguire terminiamo tornando il valore 0
(successo) e quindi questo ci fa uscire dal ciclo infinito */
        }
}
}
```

Verifichiamone il funzionamento, facendo attenzione che possiamo usare solo comandi (NON INTERNI) che non hanno bisogno di parametri:

```
soELab@Lica04:~/exec/processi$ comando
inserire il comando da eseguire:
ls
```

<== una sorta di PROMPT dei comandi!

```
comando     leggi     leggiPippo     leggiPippo1   myGrepConFork     mylsConFork     myopen     pippo
prova1.c  suid            suid-sbagliato.c
comando.c   leggi.c   leggiPippo.c   makefile      myGrepConFork.c   mylsConFork.c   myopen.c   prova1
scarti    suid-sbagliato  suid.c
Per il figlio 16086 lo stato di EXIT e` 0
eseguire altro comando? (si/no)
```
<== questo chiaramente non c'è in una vera shell!
```
si
inserire il comando da eseguire:
id
uid=1003(soELab) gid=100(users) groups=100(users)
Per il figlio 16088 lo stato di EXIT e` 0
eseguire altro comando? (si/no)
si
inserire il comando da eseguire:
ps
  PID TTY          TIME CMD
   PID TTY          TIME CMD
 61720 pts/0    00:00:00 bash
 62477 pts/0    00:00:00 comando
 62493 pts/0    00:00:00 ps
Per il figlio 28460 lo stato di EXIT e` 0
eseguire altro comando? (si/no)
si
inserire il comando da eseguire:
cat
nzgf
nzgf
tjata
tjata
```
<== su una riga vuota usiamo il CTRL-D per terminare lo standard input
```
Per il figlio 16089 lo stato di EXIT e` 0
eseguire altro comando? (si/no)
si
inserire il comando da eseguire:
cd
```
<== il comando cd è interno e quindi si ha un errore tentando di eseguirlo con un exec e quindi come comando esterno!

errore: No such file or directory **<== questa riga viene scritta dalla perror sullo standard error!**

```
Per il figlio 16090 lo stato di EXIT e` 2
eseguire altro comando? (si/no)
no
soELab@Lica04:~/exec/processi$
```

<== il valore della variabile errno tornato dal figlio è 2!

(Luc. Unix: azioni primitive per gestione processi 38 e 39)
Riassunto di dove siamo arrivati e cosa ci manca.

(Luc. Comunicazione in Unix: Pipe 1-4)
Leggiamo cosa dice il manuale sulla primitiva pipe:

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

NAME
       pipe - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);
…
DESCRIPTION
pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication.
The array pipefd is used to return two file descriptors referring to the ends of the pipe.  pipefd[0]
refers to the read end of the pipe.  pipefd[1] refers to the write end of the pipe.  Data written to
the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.
For further details, see pipe(7).
…

RETURN VALUE
On success, zero is returned.  On error, -1 is returned, and errno is set appropriately.
```

LASCIATO DA FARE COME ESERCIZIO AGLI STUDENTI:
Verifichiamo con un semplice programma che vengano usati gli elementi liberi della Tabella dei File aperti del processo corrente (come avviene per la open e quindi anche per la creat) anche per la creazione di una pipe: ==> usare directory ~/pipe

```
soELab@lica04:~/pipe$ cat provaPipe.c
/* FILE: provaPipe.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main ()
{
        int piped[2];   /* array di due interi per la pipe */

/* si crea una pipe */
if (pipe (piped) < 0 )
        {
                printf("Errore nella creazione pipe\n");
                exit(1);
        }
printf("creato pipe con piped[0]= %d \n", piped[0]);
printf("creato pipe con piped[1]= %d \n", piped[1]);

exit (0);
}
```

Verifichiamone il funzionamento:

```
soELab@lica04:~/pipe$ provaPipe
creato pipe con piped[0]= 3
creato pipe con piped[1]= 4
```

<== viene utilizzato l'elemento di indice 3 che è il primo libero

<== viene utilizzato l'elemento di indice 4 che è l'ulteriore elemento libero

(Luc. Comunicazione in Unix: Pipe 5)

Calcoliamo quale è la lunghezza della pipe sul sistema operativo Linux che stiamo usando, andando a scrivere una serie di caratteri nella pipe senza che ci sia nessuno che li legge, e quindi andremo a saturare la capacità della mailbox/pipe:

```
soELab@Lica04:~/pipe$ cat lungpipe.c
/* FILE: lungpipe1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int count; /* variabile globale */

int main()
{
int p[2];
char c = 'x';

if (pipe(p) < 0) { printf("Errore\n"); exit (1); }


for (count = 0;;)
    {
        write(p[1], &c, 1);
/* scrittura sulla pipe */

        if ((++count % 1024) == 0)
                printf ("%d caratteri nella pipe\n", count);
    }
exit (0); /* non si arriverà mai qui dato che abbiamo un ciclo infinito con sospensione del processo
ad un certo punto sulla write */
}
```

Verifichiamone il funzionamento:

```
soELab@Lica04:~/pipe$ lungpipe
1024 caratteri nella pipe
2048 caratteri nella pipe
3072 caratteri nella pipe
4096 caratteri nella pipe
5120 caratteri nella pipe
6144 caratteri nella pipe
7168 caratteri nella pipe
8192 caratteri nella pipe
9216 caratteri nella pipe
10240 caratteri nella pipe
11264 caratteri nella pipe
```

```
12288 caratteri nella pipe
13312 caratteri nella pipe
14336 caratteri nella pipe
15360 caratteri nella pipe
16384 caratteri nella pipe
17408 caratteri nella pipe
18432 caratteri nella pipe
19456 caratteri nella pipe
20480 caratteri nella pipe
21504 caratteri nella pipe
22528 caratteri nella pipe
23552 caratteri nella pipe
24576 caratteri nella pipe
25600 caratteri nella pipe
26624 caratteri nella pipe
27648 caratteri nella pipe
28672 caratteri nella pipe
29696 caratteri nella pipe
30720 caratteri nella pipe
31744 caratteri nella pipe
32768 caratteri nella pipe
33792 caratteri nella pipe
34816 caratteri nella pipe
35840 caratteri nella pipe
36864 caratteri nella pipe
37888 caratteri nella pipe
38912 caratteri nella pipe
39936 caratteri nella pipe
40960 caratteri nella pipe
41984 caratteri nella pipe
43008 caratteri nella pipe
44032 caratteri nella pipe
45056 caratteri nella pipe
46080 caratteri nella pipe
47104 caratteri nella pipe
```

```
48128 caratteri nella pipe
49152 caratteri nella pipe
50176 caratteri nella pipe
51200 caratteri nella pipe
52224 caratteri nella pipe
53248 caratteri nella pipe
54272 caratteri nella pipe
55296 caratteri nella pipe
56320 caratteri nella pipe
57344 caratteri nella pipe
58368 caratteri nella pipe
59392 caratteri nella pipe
60416 caratteri nella pipe
61440 caratteri nella pipe
62464 caratteri nella pipe
63488 caratteri nella pipe
64512 caratteri nella pipe
65536 caratteri nella pipe
```

<== il prompt non appare dato che il processo lungpipe1 (figlio della shell che sta aspettando la sua terminazione con una wait) risulta bloccato sulla write su pipe piena: per sbloccare il processo dovremo usare CTRL-C!

```
^C
soELab@Lica04:~/pipe$
```

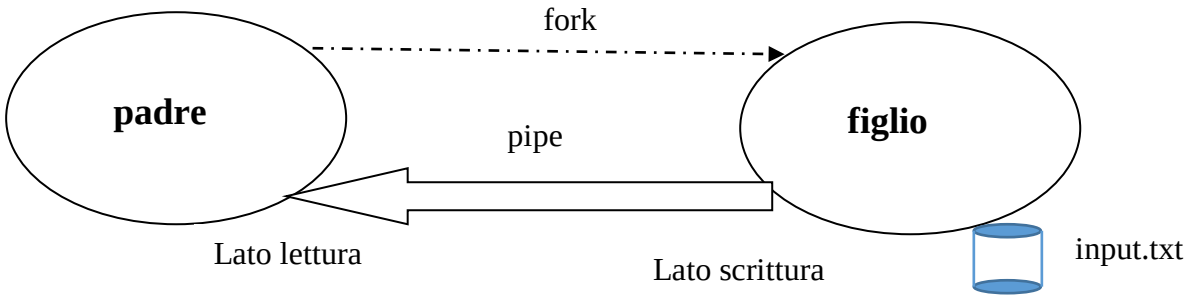
Attenzione che per ora non abbiamo una informazione precisa di quanto sia la lunghezza della pipe: sappiamo comunque che è limitata e nel sistema Linux che stiamo usando è compresa fra 65536 e 65536+1023! Quando parleremo dei segnali riprenderemo questo esempio per realizzare una versione che non si blocca, ma termina in modo controllato e che ci indicherà la lunghezza esatta di una pipe!

(fine lezione di lunedì 3/05/2021)

Lezione Mercoledì 5/05/2021 ➔ corrisponde a due video-registrazioni

(Luc. Comunicazione in Unix: Pipe 6-9)

Vediamo un primo semplice esempio di comunicazione: il processo padre (che si comporterà come un consumatore/receiver) e il processo figlio (che si comporterà come un produttore/sender) si accordano sul seguente PROTOCOLLO DI COMUNICAZIONE: il processo figlio invierà al padre una serie di stringhe C (cioè null-terminated) di lunghezza 4 caratteri (5 con il terminatore di stringa) e il padre le deve riportare su standard input. Il figlio ricava le stringhe leggendole da un file di testo (costituito da una serie di righe) il cui nome viene passato come parametro.



Spiegazione anche tramite il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraEsempioPipe.mp4

```
soELab@Lica04:~/pipe$ cat pipe-new.c
/* FILE: pipe-new.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#define MSGSIZE 5

int main (int argc, char **argv)
{       int pid, j, piped[2];           /* pid per fork, j per indice, piped per pipe */
        char mess[MSGSIZE];             /* array usato dal figlio per inviare stringa al padre */
```

```
        char inpbuf [MSGSIZE];            /* array usato dal padre per ricevere stringa inviata dal figlio: N.B: si poteva usare sempre mess, tanto il padre e il figlio agiscono sulla loro copia delle variabili! */
        int pidFiglio, status, ritorno;   /* per wait padre */

if (argc != 2)
{   printf("Numero dei parametri errato %d: ci vuole un singolo parametro\n", argc);
    exit(1);
}
/* si crea una pipe */
if (pipe (piped) < 0 )
{   printf("Errore creazione pipe\n");
    exit (2);
}

if ((pid = fork()) < 0)
{   printf("Errore creazione figlio\n");
    exit (3);
}
if (pid == 0)
{
        /* figlio */
        int fd;
        close (piped [0]);      /* il figlio CHIUDE il lato di lettura */
        if ((fd = open(argv[1], O_RDONLY)) < 0)
        {   printf("Errore in apertura file %s\n", argv[1]);
            exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi identificato come errore */
        }

        printf("Figlio %d sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza %d, sulla pipe dopo averli letti dal file passato come parametro\n", getpid(), MSGSIZE);
        j=0; /* il figlio inizializza la sua variabile j per contare i messaggi che ha mandato al padre */
```

```
        while (read(fd, mess, MSGSIZE)) /* il contenuto del file e' tale che in mess ci saranno 4
caratteri e il terminatore di linea */
        {
                /* il padre ha concordato con il figlio che gli mandera' solo stringhe e quindi
dobbiamo sostituire il terminatore di linea con il terminatore di stringa */
                mess[MSGSIZE-1]='\0';
                write (piped[1], mess, MSGSIZE);
                j++;
        }
        printf("Figlio %d scritto %d messaggi sulla pipe\n", getpid(), j);
        exit(0);
}

/* padre */
close (piped [1]); /* il padre CHIUDE il lato di scrittura */
printf("Padre %d sta per iniziare a leggere i messaggi dalla pipe\n", getpid());
j=0; /* il padre inizializza la sua variabile j per verificare quanti messaggi ha mandato il figlio
*/
while (read ( piped[0], inpbuf, MSGSIZE))
```

<== dato che il processo scrittore ad un certo punto termina, la primitiva read tornerà 0 e quindi il processo lettore terminerà il while!

```
    {
        /* dato che il figlio gli ha inviato delle stringhe, il padre le puo' scrivere direttamente
con una printf */
        printf ("%d: %s\n", j, inpbuf);
        j++;
     }
printf("Padre %d letto %d messaggi dalla pipe\n", getpid(), j);
/* padre aspetta il figlio */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(5);
}
if ((status & 0xFF) != 0)
```

```
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);}
exit (0);
}
```

Il file che passeremo come parametro è il seguente (ogni linea contiene 4 caratteri e chiaramente il terminatore di linea \n):

```
soELab@Lica04:~/pipe$ cat input.txt
cane
ciao
neve
lato
tela
poco
cosa
dado
nodo
casa
soELab@lica04:~/pipe$ wc -l < input.txt
10
```

Verifichiamo quindi il funzionamento:

a) caso SBAGLIATO senza parametri

```
soELab@lica04:~/pipe$ pipe-new
Numero dei parametri errato 1: ci vuole un singolo parametro
```

b) caso SBAGLIATO nome file NON esistente

```
soELab@lica04:~/pipe$ pipe-new input
Padre 94580 sta per iniziare a leggere i messaggi dalla pipe
Errore in apertura file input
Padre 94580 letto 0 messaggi dalla pipe
Il figlio con pid=94581 ha ritornato 255 (se 255 problemi!)
```

c) caso GIUSTO

```
soELab@lica04:~/pipe$ pipe-new input.txt
Padre 94576 sta per iniziare a leggere i messaggi dalla pipe
```

```
Figlio 94577 sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza 5, sulla pipe
dopo averli letti dal file passato come parametro
0: cane
1: ciao
2: neve
3: lato
4: tela
5: poco
6: cosa
7: dado
8: nodo
9: casa
Figlio 94577 scritto 10 messaggi sulla pipe
Padre 94576 letto 10 messaggi dalla pipe
Il figlio con pid=94577 ha ritornato 0 (se 255 problemi!)
```

Vediamo il contenuto di un altro file, input1.txt :

```
soELab@Lica04:~/pipe$ cat input1.txt
cane
ciao
neve
lato
tela
poco
cosa
dado
nodo
casa
tata
```

<== uguale a input.txt ma ha questa linea in più

```
soELab@lica04:~/pipe$ wc -l < input1.txt
11
```

Verifichiamo di nuovo il funzionamento:

```
soELab@lica04:~/pipe$ pipe-new input1.txt
Padre 94606 sta per iniziare a leggere i messaggi dalla pipe
Figlio 94607 sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza 5, sulla pipe
dopo averli letti dal file passato come parametro
```

```
0: cane
1: ciao
2: neve
Figlio 94607 scritto 11 messaggi sulla pipe
```
<== N.B. Le stampe di padre e figlio si possono mescolare!
```
3: lato
4: tela
5: poco
6: cosa
7: dado
8: nodo
9: casa
10: tata
Padre 94606 letto 11 messaggi dalla pipe
Il figlio con pid=94607 ha ritornato 0 (se 255 problemi!)
```
PROVARE A CAMBIARE IL PROTOCOLLO DI COMUNICAZIONE, AD ESEMPIO MODIFICARE QUESTO PROGRAMMA IN MODO CHE IL PROCESSO FIGLIO INVII STRINGHE DI LUNGHEZZA QUALUNQUE AL PADRE!

(Luc. Comunicazione in Unix: Pipe 10)
Vediamo ora cosa succede ad un processo consumatore/receiver se MUORE il produttore/sender prima dell'invio corretto di tutti i messaggi che avrebbero dovuto essere mandati: nell'esempio di prima facciamo terminare il processo figlio, come caso limite, senza mandare alcun messaggio!

```
soELab@Lica04:~/pipe$ cat pipe-newSenzascrittore.c
/* FILE: pipe-newSenzascrittore.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#define MSGSIZE 5

int main (int argc, char **argv)
{       int pid, j, piped[2];                   /* pid per fork, j per indice, piped per pipe */
        char mess[MSGSIZE];                     /* array usato dal figlio per inviare stringa al
padre */
```

```
        char inpbuf [MSGSIZE];                  /* array usato dal padre per ricevere stringa inviata
dal figlio: N.B: si poteva usare sempre mess, tanto il padre e il figlio agiscono sulla loro copia
delle variabili! */
        int pidFiglio, status, ritorno;         /* per wait padre */

if (argc != 2)
{   printf("Numero dei parametri errato %d: ci vuole un singolo parametro\n", argc);
    exit(1);
}
/* si crea una pipe */
if (pipe (piped) < 0 )
{   printf("Errore creazione pipe\n");
    exit (2);
}

if ((pid = fork()) < 0)
{   printf("Errore creazione figlio\n");
    exit (3);
}
if (pid == 0)
{
        /* figlio */
        int fd;
        close (piped [0]);      /* il figlio CHIUDE il lato di lettura */
        if ((fd = open(argv[1], O_RDONLY)) < 0)
        {   printf("Errore in apertura file %s\n", argv[1]);
            exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi
identificato come errore */
        }

        printf("Figlio %d sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza %d,
sulla pipe dopo averli letti dal file passato come parametro\n", getpid(), MSGSIZE);
        /* IL FIGLIO TERMINA ===> PIPE SENZA SCRITTORE */
        exit (0);
}
```

```
/* padre */
close (piped [1]); /* il padre CHIUDE il lato di scrittura */
printf("Padre %d sta per iniziare a leggere i messaggi dalla pipe\n", getpid());
j =0; /* il padre inizializza la sua variabile j per verificare quanti messaggi ha mandato il
figlio*/
while (read ( piped[0], inpbuf, MSGSIZE))
    {
        /* dato che il figlio gli ha inviato delle stringhe, il padre le puo' scrivere direttamente
con una printf */
        printf ("%d: %s\n", j, inpbuf);
        j++;
     }
if (j != 0)
        printf("Padre %d letto %d messaggi dalla pipe\n", getpid(), j);
else { puts("NON C'E' SCRITTORE"); exit(4); }
/* padre aspetta il figlio */
pidFiglio = wait(&status);
if (pidFiglio < 0)
{
      printf("Errore wait\n");
      exit(5);
}
if ((status & 0xFF) != 0)
        printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
else
{
       ritorno=(int)((status >> 8) & 0xFF);
       printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);
}
exit (0);
}
```

Verifichiamo il funzionamento (che è poi uguale a prima a parte che il figlio NON invia alcun messaggio):

```
soELab@Lica04:~/pipe$ pipe-newSenzascrittore input.txt
Padre 94646 sta per iniziare a leggere i messaggi dalla pipe
```

```
Figlio 94647 sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza 5, sulla pipe
dopo averli letti dal file passato come parametro
NON C'E' SCRITTORE
```
<== la prima read eseguita dal padre torna 0 e non blocca il processo che quindi riporta che non c'è alcuno scrittore!

```
soELab@Lica04:~/pipe$ echo $?
```
<== il padre termina con errore (senza neanche aspettare il figlio che tanto è già terminato)!

```
4
```

(Luc. Comunicazione in Unix: Pipe 10)

Il funzionamento di questi programmi risulta corretto dato che il Sistema Operativo ha registrato, dopo la creazione della pipe e la creazione del figlio, che ci sono due processi che possono usare la pipe; entrambi questi processi all'inizio possono sia leggere che scrivere sulla pipe e quindi comportarsi potenzialmente come consumatori e come produttori; in seguito, ognuno dei processi, stabilisce il proprio ruolo, chiudendo il lato della pipe che non usa e quindi il SO sa che sulla pipe c'è un solo lettore (consumatore/receiver) che è il padre e un solo scrittore (produttore/sender) che è il figlio; quando il figlio termina, il SO registra che non c'è più alcuno scrittore e quindi può fare sì che il padre, che tenta di leggere da una pipe su cui c'è lui come unico processo (nel ruolo di lettore) non sia bloccato sulla read e che la read torni 0.

(Luc. Comunicazione in Unix: Pipe 11)

Vediamo ora il caso contrario e cioè cosa succede ad un processo il produttore/sender se MUORE il consumatore/receiver prima di ricevere tutti i messaggi: nell'esempio di prima facciamo terminare il processo padre, come caso limite, senza leggere alcun messaggio!

```
soELab@Lica04:~/pipe$ cat pipe-newSenzalettore.c
/* FILE: pipe-newSenzalettore.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#define MSGSIZE 5


int main (int argc, char **argv)
{       int pid, j, piped[2];                   /* pid per fork, j per indice, piped per pipe */
        char mess[MSGSIZE];                         /* array usato dal figlio per inviare stringa al
padre */

if (argc != 2)
{   printf("Numero dei parametri errato %d: ci vuole un singolo parametro\n", argc);
    exit(1);
```

```
}
/* si crea una pipe */
if (pipe (piped) < 0 )
{   printf("Errore creazione pipe\n");
    exit (2);
}

if ((pid = fork()) < 0)
{   printf("Errore creazione figlio\n");
    exit (3);
}
if (pid == 0)
{
        /* figlio */
        int fd;
        close (piped [0]);      /* il figlio CHIUDE il lato di lettura */
        if ((fd = open(argv[1], O_RDONLY)) < 0)
        {   printf("Errore in apertura file %s\n", argv[1]);
            exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi
identificato come errore */
        }
        sleep(1);       /* inseriamo questa sleep cosi' da essere sicuri che quando il figlio tenta
di scrivere, il padre sia gia' morto! */
        printf("Figlio %d sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza %d,
sulla pipe dopo averli letti dal file passato come parametro\n", getpid(), MSGSIZE);
        j=0; /* il figlio inizializza la sua variabile j per contare i messaggi che ha mandato al
padre */
        while (read(fd, mess, MSGSIZE)) /* il contenuto del file e' tale che in mess ci saranno 4
caratteri e il terminatore di linea */
        {
                /* il padre ha concordato con il figlio che gli mandera' solo stringhe e quindi
dobbiamo sostituire il terminatore di linea con il terminatore di stringa */
                mess[MSGSIZE-1]='\0';
                write (piped[1], mess, MSGSIZE);
                j++;
```

```
        }
        printf("Figlio %d scritto %d messaggi sulla pipe\n", getpid(), j); ← ☹
        exit (0);
}

/* padre */
close (piped [1]); /* il padre CHIUDE il lato di scrittura */
printf("Padre %d sta per iniziare a leggere i messaggi dalla pipe\n", getpid());
j =0; /* il padre inizializza la sua variabile j per verificare quanti messaggi ha mandato il figlio
*/
/* termina subito ===> PIPE SENZA PIU'LETTORE */
/* BISOGNA FARE IN MODO CHE IL PADRE TERMINI PRIMA DEL FIGLIO E QUINDI BISOGNA ELIMINARE LA PARTE
DOVE IL PADRE ASPETTEREBBE IL FIGLIO quindi padre NON aspetta il figlio */
exit (0);
}
```

Verifichiamo il funzionamento:

```
soELab@Lica04:~/pipe$ pipe-newSenzalettore input.txt
Padre 94654 sta per iniziare a leggere i messaggi dalla pipe
```
<== il padre termina e torna il prompt!

```
soELab@Lica04:~/pipe$ ps
   PID TTY          TIME CMD
 94533 pts/0    00:00:00 bash
 94655 pts/0    00:00:00 pipe-newSenzale
 94656 pts/0    00:00:00 ps
soELab@lica04:~/pipe$ Figlio 94655 sta per iniziare a scrivere una serie di messaggi, ognuno di
lunghezza 5, sulla pipe dopo averli letti dal file passato come parametro
```
<== appena torna il prompt chiediamo l'esecuzione del comando ps

<== la prima write eseguita dal figlio comporta che il Sistema Operativo mandi al processo che la esegue (il figlio) un segnale specifico (si chiama SIGPIPE) che provoca di default la morte del processo: lo si capisce dal fatto che il figlio non esegue la stampa ulteriore che risulta nel codice (si veda ☹). Quando parleremo dei segnali riprenderemo questo esempio!

Applichiamo ora lo strumento di comunicazione delle pipe (oltre che chiaramente tutte le altre primitive viste finora) per risolvere degli esercizi di esame.

Esame del 5 Giugno 2015 (seconda prova in Itinere, quindi solo parte C):

La parte in C accetta un numero variabile di parametri (maggiore o uguale a 2, da controllare) che rappresentano M nomi assoluti di file F1...FM.

Il processo padre deve generare M processi figli (P0 … PM-1): ogni processo figlio è associato al corrispondente file Fi (con i=j+1). Ognuno di tali processi figli deve creare a sua volta un processo nipote (PP0 … PPM-1): ogni processo nipote PPj esegue concorrentemente e deve, usando in modo opportuno il comando tail di UNIX/Linux, leggere l'ultima linea del file associato Fi.
Ogni processo figlio Pj deve calcolare la lunghezza, in termini di valore intero (lunghezza), della linea scritta (escluso il terminatore di linea) sullo standard output dal comando tail eseguito dal processo nipote PPj; quindi ogni figlio Pj deve comunicare tale lunghezza al padre. Il padre ha il compito di ricevere, rispettando l'ordine inverso dei file, il valore lunghezza inviato da ogni figlio Pj che deve essere riportato sullo standard output insieme all'indice del processo figlio e al nome del file cui tale lunghezza si riferisce.
Al termine, ogni processo figlio Pj deve ritornare al padre il valore di ritorno del proprio processo nipote PPj e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

Per illustrare il testo della seconda prova in itinere del 5 Giugno 2015 letto la lezione scorsa, deciso di presentare, come premessa, una versione semplificata del testo che considera che i figli NON debbano creare i nipoti e quindi concentrata l'attenzione solo sui figli e sul padre. In particolare, ai figli è assegnato un compito molto semplice e cioè devono eseguire il calcolo 3000-j.
Quindi discusso gli elementi più rilevanti della soluzione:
- sottolineato che se il testo usa dei nomi simbolici specifici (come M e lunghezza) se nel codice si usano questi nomi per le variabili lo studente e chi corregge il compito ne trae vantaggio!
- necessità da parte del padre di creare M pipe (una per ogni figlio) prima di creare i figli, perché solo in questo modo si può soddisfare la specifica che richiede al padre di rispettare un ordine (in questo caso, inverso) nella ricezione delle informazioni dai figli. NOTA BENE: la coppia di file descriptor derivanti dalla creazione di ogni singola pipe, viene salvata in un array dinamico piped (la cui memoria quindi viene allocata con la malloc) di dimensione M.
Nel video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/5Giu2015-IIProvaInItinere-premessa.mp4 si trovano ulteriori dettagli che illustrano le specifiche e le scelte implementative.

Vediamo il codice di questa versione semplificata:

```
soELab@lica04:~/5Giu15$ cat 5Giu15-SoloFigli.c
/* Soluzione della Prova d'esame del 5 Giugno 2015 - SOLO Parte C */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

typedef int pipe_t[2];
```

```
int main(int argc, char **argv)
{
        /* -------- Variabili locali ---------- */
        int pid;                        /* process identifier per le fork() */
        int M;                            /* numero di file passati sulla riga di comando (uguale al
numero di file) */
        int status;                     /* variabile di stato per la wait */
        pipe_t *piped;                      /* array dinamico di pipe descriptors per comunicazioni
figli-padre  */
        int j, k;                       /* indici per i cicli */
        int lunghezza;                  /* variabile che viene comunicata da ogni figlio al padre */
        int ritorno;                    /* variabile usata dal padre per recuperare valore di ritorno
di ogni figlio */
/* ------------------------------------ */

        /* Controllo sul numero di parametri */
        if (argc < 3) /* Meno di due parametri */
        {
                printf("Errore nel numero dei parametri\n");
                exit(1);
        }

        /* Calcoliamo il numero di file passati */
        M = argc - 1;

        /* Allocazione dell'array di M pipe descriptors */
        piped = (pipe_t *) malloc (M*sizeof(pipe_t));
        if (piped == NULL)
        {
                printf("Errore nella allocazione della memoria\n");
                exit(2);
        }

        /* Creazione delle M pipe figli-padre */
```

```
        for (j=0; j < M; j++)
        {
                if(pipe(piped[j]) < 0)
                {
                        printf("Errore nella creazione della pipe\n");
                        exit(3);
                }
        }

        printf("Sono il processo padre con pid %d e sto per generare %d figli\n", getpid(), M);

        /* Ciclo di generazione dei figli */
        for (j=0; j < M; j++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork\n");
                        exit(4);
                }

                if (pid == 0)
                {
                        /* codice del figlio */
                        printf("Sono il processo figlio di indice %d e pid %d sto per creare il
nipote che recuperera' l'ultima linea del file %s\n", j, getpid(), argv[j+1]); <== N.B. NON VERO!
                        /* in caso di errori nei figli o nei nipoti decidiamo di tornare dei numeri
negativi (-1 che corrispondera' per il padre al valore 255, -2 che corrispondera' a 254, etc.) che
non possono essere valori accettabili di ritorno dato che il comando tail, usato avendo implementato
la ridirezione in ingresso, puo' tornare solo 0 (perche' avra' sempre successo) */

                        /* Chiusura delle pipe non usate nella comunicazione con il padre */
                        for (k=0; k < M; k++)
                        {
                                close(piped[k][0]);
                                if (k != j) close(piped[k][1]);
```

```
                        }

                        lunghezza=3000+j;
                        /* il figlio comunica al padre */
                        write(piped[j][1], &lunghezza, sizeof(lunghezza));

                        exit(0);
                }
        }

/* Codice del padre */
/* Il padre chiude i lati delle pipe che non usa */
        for (j=0; j < M; j++)
                close(piped[j][1]);

/* Il padre recupera le informazioni dai figli in ordine inverso di indice */
        for (j=M-1; j >= 0; j--)
        {
                /* il padre recupera tutti i valori interi dai figli */
                read(piped[j][0], &lunghezza, sizeof(lunghezza));
                printf("Il figlio di indice %d ha comunicato il valore %d per il file %s\n", j,
lunghezza, argv[j+1]);
        }

        /* Il padre aspetta i figli */
        for (j=0; j < M; j++)
        {
                pid = wait(&status);
                if (pid < 0)
                {
                printf("Errore in wait\n");
                exit(5);
                }

                if ((status & 0xFF) != 0)
```

```
                printf("Figlio con pid %d terminato in modo anomalo\n", pid);
                else
                { ritorno=(int)((status >> 8) & 0xFF);
                  if (ritorno!=0)
                        printf("Il figlio con pid=%d ha ritornato %d e quindi vuole dire che il
nipote non e' riuscito ad eseguire il tail oppure il figlio o il nipote sono incorsi in errori\n",
pid, ritorno);
                  else  printf("Il figlio con pid=%d ha ritornato %d\n", pid, ritorno);
                }
        }
exit(0);
}
```

Verifichiamo il funzionamento (passiamo dei nomi di file che in realtà non saranno usati se non semplicemente nelle printf):

```
soELab@lica04:~/5Giu15$ 5Giu15-SoloFigli pippo prova1.txt prova2.txt prova3.txt
Sono il processo padre con pid 109370 e sto per generare 4 figli
Sono il processo figlio di indice 0 e pid 109371 sto per creare il nipote che recuperera' l'ultima
linea del file pippo     <== N.B. NON VERO!
Sono il processo figlio di indice 3 e pid 109374 sto per creare il nipote che recuperera' l'ultima
linea del file prova3.txt     <== N.B. NON VERO!
Il figlio di indice 3 ha comunicato il valore 3003 per il file prova3.txt
Sono il processo figlio di indice 1 e pid 109372 sto per creare il nipote che recuperera' l'ultima
linea del file prova1.txt     <== N.B. NON VERO!
Sono il processo figlio di indice 2 e pid 109373 sto per creare il nipote che recuperera' l'ultima
linea del file prova2.txt     <== N.B. NON VERO!
Il figlio di indice 2 ha comunicato il valore 3002 per il file prova2.txt
Il figlio di indice 1 ha comunicato il valore 3001 per il file prova1.txt
Il figlio di indice 0 ha comunicato il valore 3000 per il file pippo
Il figlio con pid=31210 ha ritornato 0
Il figlio con pid=31212 ha ritornato 0
Il figlio con pid=31214 ha ritornato 0
Il figlio con pid=31215 ha ritornato 0
```

(fine lezione di mercoledì 5/05/2021)