PROVE PRIMITIVE SOeLAB A.A. 2019-20

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)

Sommario

Lezione Sesto Lunedì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Gio. 2 e Ven. 3/04/2020

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 1-2)

(Luc. Unix. Tabelle per l'interazione con i file 3-4)
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 8976644   6613576  58% /              <== punto di mount
…
/dev/sdb1      256979396 2978696 240877220   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 155571   893005   15% /
…
/dev/sdb1      16384000  66862 16317138    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 Mar  3 10:59 /dev/sda2
brw-rw---- 1 root disk 8, 17 Mar  3 10:59 /dev/sdb1    <== b indica che sono dispositivi organizzati a BLOCCHI!
```

(Luc. Unix. Tabelle per l'interazione con i file 5-6)
(fine seconda video-registrazione di venerdì 3/04/2020)

Lezione Sesto Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Sab. 4 e Dom. 5/04/2020

(Luc. Unix. Tabelle per l'interazione con 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):

```
soELab@lica04:~/file$ man -s 2 sync
SYNC(2)                          Linux Programmer's Manual                          SYNC(2)

NAME
       sync, syncfs - commit filesystem caches to disk

SYNOPSIS
       #include <unistd.h>
       void sync(void);
…
DESCRIPTION
       sync() causes all pending modifications to filesystem metadata and cached file data to be
written to the underlying filesystems.
…
```

Ora passiamo a verificare il funzionamento delle primitive viste finora, ma prima controlliamo il manuale anche della open e della close.

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

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

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

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

<== questa versione la vedremo poi!

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

…
```

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

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

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

NAME
close - close a file descriptor

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

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

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

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

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

```
if (argc != 2) { puts("Errore nel numero di parametri");
```
<== per prima cosa controlliamo il numero di parametri: vogliamo avere un numero uguale a 1 (il nome del file che vogliamo aprire) e quindi il numero dei parametri deve essere 1+1 (nome del comando stesso)=2

```
                exit(1); }
```
<== in caso di errore, usiamo la primitiva exit con un numero diverso ogni volta (in modo assolutamente analogo a quanto veniva fatto nella shell)!

```
if ((fd1 = open(argv[1], O_RDONLY)) < 0)
```
<== invochiamo la open e controlliamo se il valore di ritorno è minore di 0; FARE MOLTA ATTENZIONE ALLE PARENTESI INDICATE! Unix controllerà i diritti sul file del processo in esecuzione controllando UID e GID effettivi (che si trovano nel descrittore di processo) nei confronti di UID e GID del file (che si trovano nell'inode)

```
                { puts("Errore in apertura file");
                  exit(2); }
else
        printf("Valore di fd1 = %d\n", fd1);
```
<== stampiamo il valore ritornato dalla open!

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

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

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

```
if ((fd3 = open(argv[1], O_RDONLY)) < 0)
```
<== NOTA BENE: stessa nota di cui sopra!

```
                { puts("Errore in apertura file");
                  exit(2); }
else
        printf("Valore di fd3 = %d\n", fd3);
```
<== verifichiamo il valore ritornato dalla open!

```
return 0;
```
<== ritorniamo 0 che è il valore di successo in UNIX!

```
}
```

Ricordiamo come si fa ad ottenere una versione eseguibile da un programma C:

```
soELab@Lica04:~/file$ gcc -Wall -o provaopen provaopen.c
```
<== con l'opzione –Wall andiamo a verificare tutti i warning!
Ricordarsi che NON ci devono essere warning, oltre che chiaramente errori di compilazione e di linking!

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

```
soELab@Lica04:~/file$ provaopen
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);
```
<== ciclo infinito!

```
                exit(2); }
```
<== 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!

```
        else i++;
return 0;
}
soELab@Lica04:~/file$ proveopen
Errore nel numero di parametri
soELab@Lica04:~/file$ proveopen proveopen.c
Errore in apertura file
Valore di i = 1021
```
<== la dimensione totale della tabella si ricava da 1021 + 3 che sono i primi 3 elementi della tabella dei file aperti di ogni processo, usati per standard input, standard output e standard error: quindi = 1024!

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

(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:

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 file e direttori 9-10)
Dal man della read e della write:

```
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.*

(Saltati Luc. C/Unix Primitive per i file 12-14)

(Luc. C/Unix Primitive per i file 15)
Vediamo un primo esempio di uso delle primitive read e write: del codice seguente fatto vedere nella prima ora solo il codice della funzione main.
(fine prima video-registrazione di sabato 4/04/2020)

```
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  5 12:36 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
```

NOTA BENE: il comando umask è un comando interno (built-in o special command) sia nella Bourne Shell che nella bash e quindi NON esiste il manuale del comando ma la sua descrizione la si trova nel manuale della shell corrispondente!

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  5 12:36 F2
-rw-r--r-- 1 soELab users 24 Apr  5 12:43 F2-new
```

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

(Luc. C/Unix Primitive per i file 16)

Vediamo un esempio di uso del concetto di ridirezione ==> usare sempre directory ~/file

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

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

```
        int nread;

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

<== Si legge da standard input!

<== Si scrive su standard output!

```
soELab@Lica04:~/file$ copiarid
ciao
ciao
come stai?
come stai?
```

<== Tutto quello che si scrive su standard input!

<== viene riportato su standard output!

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  5 12:54 F1-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 28 Apr  5 12:55 FF
```
<== **N.B.: i file creati dalla ridirezione hanno come diritti proprio 0644**

4) nessuna ridirezione

```
soELab@Lica04:~/file$ copiarid
vDVd
vDVd
dfbadf
dfbadf
dfADF
dfADF
```

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) 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)!

2) come comando

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

3) come commando con ridirezione in uscita

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

4) come filtro con ridirezione in ingresso

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

5) come filtro con ridirezione in ingresso e in uscita

```
soELab@Lica04:~/file$ mycat < F1-mycat > 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  5 13:02 F1-mycat
-rw-r--r-- 1 soELab users 24 Apr  5 13:02 F2-mycat
```

(fine seconda video-registrazione di domenica 5/04/2020)

Lezione Settimo Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Dom. 12/04/2020

(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 riveda osservazione inserita nel codice dell'esecizio provaopen.c della lezione scorsa.

(Luc. C/Unix Primitive per i file 19) ==> vedere anche figura che si trova nel file http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraRidirezione-implementazione.pdf
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;
/* ERRORE se non si riesce ad aprire in LETTURA il primo file */
```

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

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

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

<== 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 )
{ if (write(outfile, buffer, nread) < nread)
/* ERRORE se non si riesce a SCRIVERE */
```

<== infile = 0!
<== outfile = 1!

```
    { 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 F1-ridir

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

Verifichiamone le informazioni del file F1-ridir:

```
soELab@Lica04:~/file$ ls -l F1-ridir
-rw-r--r-- 1 soELab users 24 Apr 12 12:10 F1-ridir
```

```
soELab@Lica04:~/file$ cat F1-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)
```

<== 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!

```
  write(fd, "Ci sono stati degli errori durante la copia\n", 44);
```
<== 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!

```
return status;
}
```

Verifichiamone il funzionamento:

1) corretto

```
soELab@Lica04:~/file$ ridir-constampesudev F1 FF1
Valore di infile 0 e di outfile 1
soELab@Lica04:~/file$ echo $?
0
```

Verifichiamone le informazioni del file FF1:

```
soELab@lica04:~/file$ ls -l FF1
-rw-r--r-- 1 soELab users 24 Apr 12 12:19 FF1
soELab@lica04:~/file$ cat FF1
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 12 12:36 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 F7
-rw-r--r-- 1 soELab users 62 Apr 12 12:36 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!

(fine prima video-registrazione di domenica 12/04/2020)

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

```
            { lseek(fd, -1L, 1);
        /* SI DEVE RIPORTARE INDIETRO IL FILE POINTER */
```
<== 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)!

```
              write(fd, " ", 1);
```
<== scrittura di un singolo carattere che si trova all'inizio del buffer di memoria costituito dalla stringa che contiene un singolo carattere spazio/blank!

```
            }
return 0;
}
```

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 pap
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 12 18:12 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$ 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 12 18:21 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 servirà per identificare se è 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 è 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", 49);
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", 49);
        }

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", 49);
        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   49 Apr 12 18:34 paperina
-rw-r--r-- 1 soELab users   49 Apr 12 18:34 paperino
-rw-r--r-- 1 soELab users   49 Apr 12 18:34 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
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   <== N.B. paperina ha variato la dimensione (e data)
-rw-r--r-- 1 soELab users   49 Apr 12 18:34 paperino   <== N.B. paperino è rimasto uguale
-rw-r--r-- 1 soELab users   49 Apr 12 18:36 pippo      <== 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
::::::::::::::
seconda volta che scriviamo su file
```

<== N.B: Il file paperina risulta sovrascritto completamente!

(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 seconda video-registrazione di domenica 12/04/2020)

Lezione Ottavo Lunedì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Gio. 16 e Ven. 17/04/2020

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

(fine prima video-registrazione di giovedì 16/04/2020)

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");
exit(0);
}
```

<== eseguita da entrambi i processi (padre e figlio) dato che il codice è condiviso!

<== 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, ...

(Luc. Unix: azioni primitive per gestione processi 19)

2) 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
```
<== scritta dal padre!
```
DUE. PID = 10685, UID = 1003, GID = 100
```
<== scritta dal padre!
```
DUE. PID = 10686, UID = 1003, GID = 100
```
<== scritta dal figlio!
```
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
```
<== mescolamento come prima!

(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.

3) 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 */
else
        printf("Ho creato figlio con PID = %d\n", pid); /* padre */
exit(0);
}
```

<== questa sezione la esegue solo il processo figlio!

<== questa sezione la esegue solo il processo padre!

<== 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 seconda video-registrazione di venerdì 17/04/2020)

Lezione Ottavo Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Dom. 19/04/2020

(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' di 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)
    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);
}
```

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

<== se negli 8 bit meno significativi NON abbiamo 0 allora vuol dire che il figlio è terminato in modo involontario cioè anomalo!

<== si eliminano gli 8 bit (il byte) meno significativo che abbiamo verificato valgono 0

<== 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

Vediamo come funziona:

```
soELab@Lica04:~/processi/PrimiEsempi$ status1
Generato figlio con PID = 13191
Esecuzione del figlio
Terminato figlio con PID = 13191
Per il figlio 13191 lo stato di EXIT e` 5
```

<== 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!

<== 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!

(fine prima video-registrazione di domenica 19/04/2020)

Precisazione su errore (ora corretto) nell'ultima slide di http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraWait.mp4!

(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

```
/* 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
```

2) 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:

```
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 seconda video-registrazione di domenica 19/04/2020)

Lezione Nono Lunedì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Gio. 23 e Ven. 24/04/2020

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

(Luc. Unix: azioni primitive per gestione processi 32-33)
Ripreso discorso su fork ed exec ➔ 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 47032 Jan 27  2016 /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 1798 Feb 15 10:46 /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);  <== il programma leggiPippo1 deriva dal sorgente
leggiPippo.c, ma avrà il set-user-id settato!
        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);
}
```

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.
```

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 = 1003
effective-user id = 1003
Esecuzione di programma (con suid settato) che visualizza file (leggibile solo dal proprietario)
real-user id = 1003
effective-user id = 1004
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 = 1003
effective-user id = 1003
Esecuzione di programma che tenta di visualizzare file (leggibile solo dal proprietario)
real-user id = 1003
effective-user id = 1003
SONO leggiPippo
Sto per leggere il file pippo
ERRORE in apertura
Il figlio con pid=25061 ha ritornato 1
```

effective-user id = 1003 **<== il programma leggiPippo NON ha il set-user-id settato e quindi NON è cambiato l'effective UID**

(fine prima video-registrazione di giovedì 23/04/2020)

(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 */
        }
}
}
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/errno.h>
#include <string.h>
#include <sys/wait.h>
```

```
int main  (int argc, char **argv)
{       int stato, atteso, pid, exit_s;
        char st[80];

for (;;)        /* ciclo infinito */
{
        printf("inserire il comando da eseguire:\n");
        scanf("%s", st);                                                    <== ①
        if ((pid = fork()) < 0)  { perror("fork"); exit(1);}

        if (pid == 0)
                {  /*  FIGLIO: esegue i comandi */
                execlp(st, st, (char *)0);
                perror("errore");
                exit (errno);
                }
        else {          /* PADRE */
                        atteso=wait (&stato); /*attesa figlio: sincronizzazione */
                        exit_s = stato >> 8;
                        exit_s &= 0xFF; /* selezione degli 8 bit piu' significativi */
                        printf("Per il figlio %d lo stato di EXIT e` %d\n", atteso, exit_s);
                        printf ("eseguire altro comando? (si/no) \n");
                        scanf ("%s", st);                                   <== ②
                        if (strcmp(st, "si")) exit(0);
                }
        }
}
```

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

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

<== una sorta di PROMPT dei comandi!

```
ls
comando-bis     leggiPippo     leggiPippo1   myGrepConFork     mylsConFork     myopen        myopen-new.c
myopen.c  prova     prova1     scarti  suid.c     suid1
```

```
comando-bis.c  leggiPippo.c  makefile      myGrepConFork.c  mylsConFork.c  myopen-new  myopen-new1
pippo     prova.c  prova1.c  suid    suid.old  suid1.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=1004(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
27491 pts/1    00:00:00 bash
28459 pts/1    00:00:00 comando-bis
28460 pts/1    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
```
<== il valore della variabile errno tornato dal figlio è 2!
```
eseguire altro comando? (si/no)
no
```

```
soELab@Lica04:~/exec/processi$
```

(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): ==> usare directory ~/pipe

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

```
int main (int argc, char** argv)
{
        int fd1, fd2;
        int piped[2];

fd1= open(argv[0], O_RDONLY);
printf("aperto file con fd1=%d \n", fd1);
fd2= open(argv[0], O_RDONLY);
printf("aperto file con fd2=%d \n", fd2);
close(fd1);

/* si crea una pipe */
if (pipe (piped) < 0 )  { 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$ prova
aperto file con fd1=3
aperto file con fd2=4
creato pipe con piped[0]= 3
creato pipe con piped[1]= 5
```

<== dopo la close(fd1) viene riutilizzato l'elemento di indice 3!

(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 lungpipe1.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$ lungpipe1
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$
```

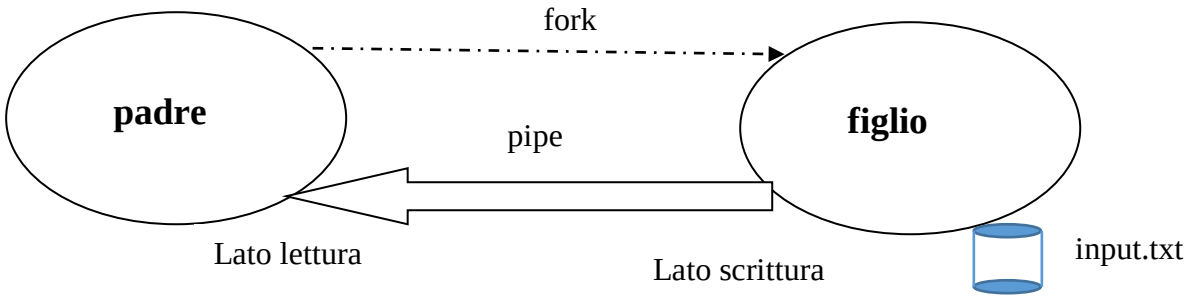
Attenzione che per ora non abbiamo una informazione precisa di quanto sia la lunghezza della pipe: sappiamo comunque che è limitata e nel sistema Linux che stiamo usando è compresa fra 65536 e 65536+1023! Quando parleremo dei segnali riprenderemo questo esempio per realizzare una versione che non si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

(fine seconda video-registrazione di venerdì 24/04/2020)

Lezione Nono Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Dom. 26/04/2020

(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:

```
soELab@Lica04:~/pipe$ pipe-new input.txt
Padre 22952 sta per iniziare a leggere i messaggi dalla pipe
Figlio 22953 sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza 5, sulla pipe
dopo averli letti dal file passato come parametro
Figlio 22953 scritto 10 messaggi sulla pipe
0: cane
1: ciao
2: neve
3: lato
4: tela
5: poco
6: cosa
7: dado
```

```
8: nodo
9: casa
Padre 22952 letto 10 messaggi dalla pipe
Il figlio con pid=22953 ha ritornato 0
```

Vediamo il contenuto di un altro file, input.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 23043 sta per iniziare a leggere i messaggi dalla pipe
Figlio 23044 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 23044 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 23043 letto 11 messaggi dalla pipe
Il figlio con pid=23044 ha ritornato 0
```

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 24214 sta per iniziare a leggere i messaggi dalla pipe
Figlio 24215 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
```

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 22759 sta per iniziare a leggere i messaggi dalla pipe
```

<== il padre termina e torna il prompt!

```
soELab@Lica04:~/pipe$ ps
```

<== appena torna il prompt chiediamo l'esecuzione del comando ps

```
  PID TTY          TIME CMD
22377 pts/2    00:00:00 bash
23086 pts/2    00:00:00 pipe-newSenzale
23087 pts/2    00:00:00 ps
soELab@Lica04:~/pipe$ Figlio 22760 sta per iniziare a scrivere una serie di messaggi, ognuno di
lunghezza 5, sulla pipe dopo averli letti dal file passato come parametro
```

<== 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!

(fine prima video-registrazione di domenica 26/04/2020)

(Luc. Comunicazione in Unix: Pipe 12-14)

Passiamo ora a capire come la shell (la BOURNE shell. NOTA BENE: NON È DETTO CHE TUTTE LE SHELL SCELGANO QUESTA IMPLEMENTAZIONE; AD ESEMPIO LA BASH IMPLEMENTA IL PIPING IN MODO DIVERSO) implementa il piping dei comandi: scriviamo quindi un programma che simula l'esecuzione in piping di due comandi passati come parametri con un carattere particolare che specifica che devono essere eseguiti in piping; per evitare problemi con il metacarattere che nella shell viene usato per indicare il piping utilizziamo un carattere alternativo (il punto esclamativo '!'). Il primo comando (con i suoi eventuali parametri) viene memorizzato in un array chiamato com1 e il secondo comando (con i suoi eventuali parametri) viene memorizzato in un array chiamato com2: entrambi i comandi devono essere eseguiti (tramite una exec) da un processo diverso. Vediamo come:

```
soELab@Lica04:~/pipe$ cat shellpipe.c
/* FILE: shellpipe.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
```

```
int join (char *com1[], char *com2[])
{
        int pid, piped[2];                      /* pid per fork, piped per pipe */
        int pidFiglio, status, ritorno;         /* per wait padre */

/* processo P1: simulazione del processo shell */
        /* creazione del figlio per eseguire il comando in pipe */
        switch (fork ())
        {
        case -1:    /* errore */ printf("Errore nella prima fork\n"); return 1;
        case 0:     /* figlio ===> processo P2 */       break;
        default:    /* padre P1:  attende il figlio P2 */
                if ((pidFiglio=wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        return 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\n", pidFiglio, ritorno);
                        return ritorno;
                }
        }

        /* il figlio P2 esegue il comando intero: crea la pipe */
        if (pipe (piped) < 0 )
        {       printf("Errore in creazione pipe\n");
                exit(-1) /* torniamo un valore che verra' interpretato 255 dal padre */;
        }
        /*  CREAZIONE di un nuovo FIGLIO: processo P3  */
        if ((pid = fork()) < 0)
        {       printf("Errore nella seconda fork\n");
```

```
                exit(-1) /* torniamo un valore che verra' interpretato 255 dal padre */;
        }
        else
        if (pid == 0)       /* P3: figlio del FIGLIO P2 */
        {       close(1);  /* lo standard output va messo sulla pipe */
                dup(piped[1]);
                close(piped[0]); close(piped[1]); /* non servono piu' */
                execvp(com1[0], com1);
                exit(-1);   /* errore in caso si ritorni qui */
        }
        else  /*  processo P2 (padre di P3) */
        {       close(0);  /* lo standard input va preso dalla pipe */
                dup(piped[0]);
                close(piped[0]); close(piped[1]); /* non servono piu' */
                execvp(com2[0], com2);
                exit(-1);   /* errore in caso si ritorni qui */
        }
}

int main  (int argc, char **argv)
{
        int integi, j, i;               /* integi per valore di ritorno, i e j come indici */
        char *temp1[10], *temp2[10];    /* array di 10 stringhe, supposte sufficienti per copiare il
primo comando (con i suoi parametri) e il secondo comando (con i suoi parametri) che sono presenti
nel piping */
/* si devono fornire nella linea comandi due  comandi distinti, separati dal carattere !. Non si usa
direttamente il metacarattere |, perche` questo viene direttamente interpretato dallo shell come una
pipe */
        if (argc > 4)   /* ci devono essere almeno tre stringhe, due che corrispondono ai due comandi
in piping e una che deve contenere il carattere ! */
        {       for (i=1; i < argc && strcmp (argv[i], "!"); i++)
                        temp1[i-1] = argv[i];
                temp1[i-1] = (char *)0;  /* terminatore */
                i++;
                for (j = 1; i < argc ; i++, j++)
```

```
                    temp2[j-1] = argv[i];
                temp2[j-1] = (char *)0;  /* terminatore */
        }
        else { printf("Errore numero di parametri insufficiente\n"); exit(3); }
        integi = join(temp1, temp2);
        exit(integi);
}
```

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

Verifichiamone il funzionamento:

1) caso di nessun parametro

```
soELab@Lica04:~/pipe$ shellpipe
Errore numero di parametri insufficiente
soELab@Lica04:~/pipe$ echo $?
3
```

2) caso di numero di parametri corretti

```
soELab@Lica04:~/pipe$ shellpipe ls -lR $HOME ! grep pippo
-rw-r--r--  1 soELab users     50 Mar  9  2016 pippo.txt
-rw------- 1 soELab users  2009 Jun 18  2014 pippo.txt
-rw-r--r-- 1 soELab users  2009 May 15  2017 pippo.txt.sig
-rw------- 1 soELab users 1621 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users 41 Jun 18  2014 pippo
-rw------- 1 soELab users 1621 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users    0 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users 0 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users 0 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users 0 Jun 18  2014 pippo
-rw------- 1 soELab users     0 Jun 18  2014 minnie.pippo
-rw------- 1 soELab users   211 Jun 18  2014 p.pippo
-rw------- 1 soELab users 211 Jun 18  2014 prova.pippo
-rw------- 1 soELab users 211 Jun 18  2014 ecco.pippo
-rw-r--r-- 1 soELab users    32 May 24  2015 pippo
-rw-r--r-- 1 soELab users    32 Jun 18  2014 pippo
-rw------- 1 soELab users   58 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users   49 Apr 18 12:49 pippo
```

```
-rw-r--r-- 1 soELab users   25 Jun 18  2014 pippo
-rw-r----- 1 soELab users    0 Jun 18  2014 pippo
-rw------- 1 soELab users  100 Jun 18  2014 pippo
-rw-r----- 1 soELab users 11 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users 112 Mar  6  2015 pippo
-rw-r----- 1 soELab users 371 Jun 18  2014 pippo.paperino.topolino.pluto
-rw-r----- 1 soELab users   70 Jun 18  2014 pippo
-rw-r-x--- 1 soELab users  20 Jun 18  2014 pippo2
-rw-r-xr-- 1 soELab users  70 Jun 18  2014 pippor
-rw------- 1 soELab users   71 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users    30 Jun 18  2014 pippo
-rw-r--r-- 1 soELab users   759 Jun 18  2014 pippo.txt
-rw-r--r-- 1 soELab users    58 Apr 28  2015 pippo
-rw-r--r-- 1 soELab users   32 Jun 10  2014 pippo
Il figlio con pid=8668 ha ritornato 0
soELab@Lica04:~/pipe$ echo $?
0
```

3) caso di numero di parametri corretti con uso di ps per vedere 'in chiaro' la relazione dei processi (anche se può sembrare una strana richiesta!)

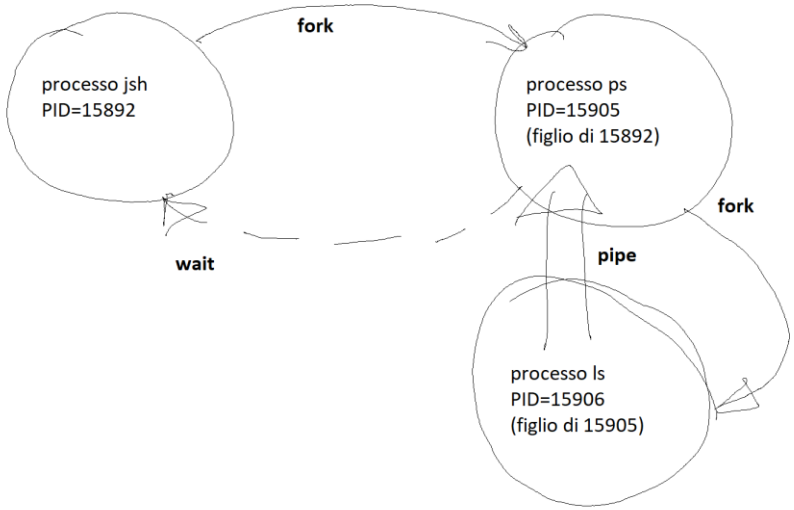
```
soELab@Lica04:~/pipe$ shellpipe ls -lR /home/soELab ! ps -lf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S soELab   24422 24421  0  80   0 -  5594 wait   17:09 pts/0    00:00:00 -bash
0 S soELab   24631 24422  0  80   0 -  1047 wait   18:15 pts/0    00:00:00 shellpipe ls -lR /home/soELab ! ps -lf   (P1)
0 R soELab   24632 24631  0  80   0 -  3891 -      18:15 pts/0    00:00:00 ps -lf                                   (P2)
0 R soELab   24633 24632  0  80   0 -  5891 -      18:15 pts/0    00:00:00 ls -lR /home/soELab                      (P3)
Il figlio con pid=8671 ha ritornato 0
```

Verifichiamo usando jsh (che sostanzialmente implementa la Bourne shell) che la relazione di parentela dei processi che eseguono i comandi in pipe sia uguale a quella realizzata dal nostro programma:

```
soELab@Lica04:~/pipe$ jsh
$ ls -lR /home/soELab | ps -lf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S soELab   15878 15877  0  80   0 -  5594 wait   17:09 pts/0    00:00:00 -bash
0 S soELab   15892 15878  0  80   0 -  1074 wait   18:17 pts/0    00:00:00 jsh     <== corrisponde al nostro processo P1
0 R soELab   15905 15892  0  80   0 -  3891 -      18:17 pts/0    00:00:00 ps -lf  <== corrisponde al nostro processo P2
```

`0 R soELab 15906 15905 0 80 0 - 5891 - 18:17 pts/0 00:00:00 ls -lR /home/soELab` **<== corrisponde al nostro processo P3**

Vediamo in un disegno le varie relazioni fra i processi:



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.

(fine seconda video-registrazione di domenica 26/04/2020)

Lezione Decimo Lunedì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Gio. 30 e Ven. 01/05/2020

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 assegnato un compito molto semplice 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 31209 e sto per generare 4 figli
Sono il processo figlio di indice 0 e pid 31210 sto per creare il nipote che recuperera' l'ultima
linea del file pippo     <== N.B. NON VERO!
Sono il processo figlio di indice 1 e pid 31212 sto per creare il nipote che recuperera' l'ultima
linea del file prova1.txt     <== N.B. NON VERO!
Sono il processo figlio di indice 3 e pid 31215 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 2 e pid 31214 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
```

Ripreso quindi il testo completo della seconda prova in itinere del 5 Giugno 2015 e discusso gli elementi più rilevanti della soluzione legati alla presenza dei nipoti:
- necessità da parte del nipote di usare una delle primitive della famiglia exec per eseguire il comando tail -1;
- necessità di una pipe (p) creata dal figlio prima di creare il nipote;
- necessità da parte del nipote di usare close(1) e dup(p[1]) per fare in modo che quello che normalmente il comando tail scrive sullo standard output venga in realtà scritto sulla pipe di comunicazione fra nipote-figlio,
OSSERVAZIONE: in totale vengono create M pipe di comunicazione fra ogni nipote e ogni figlio, ma conviene che la creazione avvenga localmente ad ogni figlio in modo che la pipe esista SOLO per quella coppia di processi e NON per le altre coppie. Nel caso in cui invece si dovesse decidere di fare creare al padre un secondo array dinamico di M pipe, si avrebbe la necessità di effettuare molte più chiusure di lati di

pipe non usate da parte dei processi: ad esempio il padre, dovrebbe chiudere tutti i lati di tutte le pipe create per la comunicazione fra nipoti e figli!

Nel video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/5Giu2015-IIProvaInItinere.mp4 si trovano ulteriori dettagli che illustrano le specifiche e le scelte implementative.

La soluzione di questa prova in itinere si trova alla URL http://didattica.agentgroup.unimore.it/didattica/SOeLab/SoluzioniCompiti/5Giu15/5Giu15.c

Mostrato il funzionamento:

```
soELab@lica04:~/5Giu15$ 5Giu15 pippo prova1.txt prova2.txt prova3.txt
Sono il processo padre con pid 31483 e sto per generare 4 figli
Sono il processo figlio di indice 0 e pid 31484 sto per creare il nipote che recuperera' l'ultima
linea del file pippo
Sono il processo figlio di indice 1 e pid 31485 sto per creare il nipote che recuperera' l'ultima
linea del file prova1.txt
Sono il processo nipote del figlio di indice 0 e pid 31488 e sto per recuperare l'ultima linea del
file pippo
Sono il processo figlio di indice 2 e pid 31486 sto per creare il nipote che recuperera' l'ultima
linea del file prova2.txt
Sono il processo figlio di indice 3 e pid 31487 sto per creare il nipote che recuperera' l'ultima
linea del file prova3.txt
Il nipote con pid=31488 ha ritornato 0
Sono il processo nipote del figlio di indice 2 e pid 31490 e sto per recuperare l'ultima linea del
file prova2.txt
Sono il processo nipote del figlio di indice 1 e pid 31489 e sto per recuperare l'ultima linea del
file prova1.txt
Sono il processo nipote del figlio di indice 3 e pid 31491 e sto per recuperare l'ultima linea del
file prova3.txt
Il nipote con pid=31490 ha ritornato 0
Il nipote con pid=31489 ha ritornato 0
Il figlio di indice 3 ha comunicato il valore 40 per il file prova3.txt
Il figlio di indice 2 ha comunicato il valore 33 per il file prova2.txt
Il figlio di indice 1 ha comunicato il valore 35 per il file prova1.txt
Il figlio di indice 0 ha comunicato il valore 7 per il file pippo
Il figlio con pid=31484 ha ritornato 0
Il figlio con pid=31485 ha ritornato 0
Il figlio con pid=31486 ha ritornato 0
```

```
Il nipote con pid=31491 ha ritornato 0
Il figlio con pid=31487 ha ritornato 0
```

Verifichiamo la lunghezza, ad esempio, dell'ultima linea di prova3.txt:

```
soELab@lica04:~/5Giu15$ tail -1 < prova3.txt
Ecco sono la seconda linea di prova3.txt
soELab@lica04:~/5Giu15$ tail -1 < prova3.txt | wc -c
41
```

Quindi 40 risulta corretto dato che è 41-1!

Mostrato il funzionamento anche passando un file che NON esiste (pap)!

```
soELab@lica04:~/5Giu15$ 5Giu15 pippo prova1.txt prova2.txt prova3.txt pap
Sono il processo padre con pid 31577 e sto per generare 5 figli
Sono il processo figlio di indice 0 e pid 31578 sto per creare il nipote che recuperera' l'ultima
linea del file pippo
Sono il processo figlio di indice 1 e pid 31579 sto per creare il nipote che recuperera' l'ultima
linea del file prova1.txt
Sono il processo nipote del figlio di indice 1 e pid 31584 e sto per recuperare l'ultima linea del
file prova1.txt
Sono il processo nipote del figlio di indice 0 e pid 31581 e sto per recuperare l'ultima linea del
file pippo
Sono il processo figlio di indice 2 e pid 31580 sto per creare il nipote che recuperera' l'ultima
linea del file prova2.txt
Sono il processo figlio di indice 3 e pid 31582 sto per creare il nipote che recuperera' l'ultima
linea del file prova3.txt
Sono il processo figlio di indice 4 e pid 31583 sto per creare il nipote che recuperera' l'ultima
linea del file pap
Sono il processo nipote del figlio di indice 4 e pid 31587 e sto per recuperare l'ultima linea del
file pap
Errore nella open del file pap
Il figlio di indice 4 ha comunicato il valore 0 per il file pap
Il nipote con pid=31587 ha ritornato 252
Sono il processo nipote del figlio di indice 2 e pid 31585 e sto per recuperare l'ultima linea del
file prova2.txt
Sono il processo nipote del figlio di indice 3 e pid 31586 e sto per recuperare l'ultima linea del
file prova3.txt
```

```
Il nipote con pid=31585 ha ritornato 0
Il figlio di indice 3 ha comunicato il valore 40 per il file prova3.txt
Il figlio di indice 2 ha comunicato il valore 33 per il file prova2.txt
Il nipote con pid=31586 ha ritornato 0
Il nipote con pid=31581 ha ritornato 0
Il figlio di indice 1 ha comunicato il valore 35 per il file prova1.txt
Il figlio di indice 0 ha comunicato il valore 7 per il file pippo
Il figlio con pid=31578 ha ritornato 0
Il figlio con pid=31580 ha ritornato 0
Il figlio con pid=31582 ha ritornato 0
Il figlio con pid=31583 ha ritornato 252 e quindi vuole dire che il nipote non e' riuscito ad
eseguire il tail oppure il figlio o il nipote sono incorsi in errori
Il nipote con pid=31584 ha ritornato 0
Il figlio con pid=31579 ha ritornato 0
```

(fine prima video-registrazione di giovedì 30/04/2020)

Leggiamo ora il testo della parte C dell'esame del 11 Luglio 2018, che era stato indicato la lezione scorsa e commentiamolo:

La parte in C accetta un numero variabile **1+N** di parametri (con **N** maggiore o uguale a **2**, da controllare**)** che rappresentano rispettivamente: il primo parametro un singolo carattere **CZ** (da controllare), mentre gli altri **N** nomi di file (**F1, F2, ... FN-1**).

Il processo padre deve generare **N processi figli** (**P0, P1, … PN-1**): i processi figli **Pi (con i che varia da 0 a N-1)** sono associati ai **N** file **Ff** (con f= i+2). Ogni processo figlio **Pi** deve leggere tutti i caratteri del file associato **Ff cercando le occorrenze del carattere CZ**; appena viene trovata una occorrenza del **carattere** CZ, il processo figlio **Pi** deve comunicare al padre la **posizione** di tale carattere (in termini di *long int*) e deve ricevere dal padre l'indicazione di stampare o meno su standard output delle informazioni (*vedi dopo* *). Il padre deve ricevere, rispettando l'ordine dei file **Ff** , da ogni figlio via via i valori *long int* che rappresentano la **posizione** all'interno del file della occorrenza **corrente** trovata. Quindi, al processo padre deve arrivare **un insieme di valori long int** (che via via potrebbe diminuire in quantità in dipendenza della terminazione dei figli): sicuramente il primo insieme (questo viene garantito dalla parte Shell) è costituito dalla prima posizione inviata dal figlio **P0**, dalla prima posizione inviata dal figlio **P1**, … , dalla prima posizione inviata dal figlio **PN-1**. Per ogni insieme ricevuto, il padre deve determinare il valore **massimo** e, SOLO AL PROCESSO FIGLIO CHE HA INVIATO TALE VALORE, deve indicare (*) di stampare su standard output **il carattere trovato, assieme all'indice d'ordine del processo, il suo pid, la posizione e il nome del file associato**, mentre a tutti gli altri processi figli deve indicare di non stampare[+].

Al termine, ogni processo figlio **Pi** deve ritornare al padre il numero di occorrenze trovate (supposto strettamente minore di 255) del carattere **CZ**; il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

([+]) Per questo tipo di interazione, volendo si possono usare i segnali. **Nel caso si usino le pipe, fare attenzione che il padre deve inviare l'indicazione SOLO ai figli che non sono terminati!**

Illustrato in dettaglio i parametri e i vari controlli che si devono fare. Poi passato ad illustrare i vari schemi di comunicazione: si parte con una serie di N pipe orientate da ogni figlio al padre e, quindi, specificato che per l'interazione fra padre e figli non sono stati usati i segnali ma un altro insieme di N pipe orientate dal padre ai figli. Sulle pipe figli-padre devono essere inviati dei long int che rappresentato via via le posizioni trovate del carattere CZ: tali posizione sono calcolate manualmente (in alternativa, si poteva usare la primitiva lseek) e si è assunto che la posizione del primo carattere del file abbia valore 0. Sulle pipe padre-figlio basta mandare un singolo carattere di controllo che se vale 'S' significherà che il corrispondente processo figlio deve riportare su standard output quanto richiesto, mentre se vale 'N' il processo figlio non deve fare nulla: nota bene, che per ogni insieme di posizioni ricevute dal padre, il padre deve inviare solo ad un figlio (quello che ha calcolato il massimo) l'indicazione di stampare, mentre a tutti gli altri processi NON FINITI deve inviare l'indicazione di non stampare. Per identificare quali figli sono finiti o meno, si deve introdurre un array finito con dimensione uguale al numero di processi e con la seguente semantica: ogni elemento vale 0 se il corrispondente processo NON è finito, altrimenti vale 1; chiaramente inizialmente ogni elemento dell'array finito viene posto a 0!
Nel video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/11Lug2018.mp4 si trovano ulteriori dettagli che illustrano le specifiche e le scelte implementative.
Quindi fatta vedere la soluzione completa sul sistema lica04.

La soluzione si trova sul sito alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/11Lug18/11Lug18.c
Alla URL si trova il contenuto di due semplici file di prova che consentono di capire il funzionamento della soluzione http://didattica.agentgroup.unimore.it/didattica/SOeLab/TestiEsami/A.A.2017-18/F1eF2x11Lug18.pdf
La soluzione che invece dal lato del padre colcola il minimo si trova sempre sul sito alla URL http://didattica.agentgroup.unimore.it/didattica/SOeLab/SoluzioniCompiti/11Lug18/11Lug18.c

(fine seconda video-registrazione di venerdì 1/05/2020)

Lezione Decimo Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Dom. 3/05/2020

Prima di vedere altri esempi di esami, passiamo a considerare un argomento molto importante che è quello relativo alla sincronizzazione tramite i segnali.
(Luc. Sincronizzazione in Unix: segnali 1-4)
Verifichiamo per prima cosa il funzionamento di alcuni dei segnali usati normalmente da UNIX.
Iniziamo con il verificare cosa succede con l'uso del CTRL-C: per farlo prendiamo in considerazione un semplice esempio di programma C che contiene un ciclo infinito ==> usare directory ~/segnali/PrimiEsempi

```
soELab@Lica04:~/segnali/PrimiEsempi$ cat loop.c
#include <stdio.h>

int main()
{
puts("SONO UN PROGRAMMA CHE VA IN LOOP");
while (1)
        puts("Sono nel ciclo");
}
```

Mandiamo in esecuzione il programma eseguibile corrispondente (avendo cura di ridirigere lo standard output per poter vedere bene l'echo sul video del CTRL-C):

```
soELab@Lica04:~/segnali/PrimiEsempi$ loop > /dev/null
^C
```

Spieghiamo che cosa è successo: la parte del kernel responsabile dell'input da tastiera riconosce che sono stati premuti questi due tasti. Il kernel virtualizza il tutto spedendo il segnale SIGINT a tutti i processi associati al terminale corrente e quindi sia al processo figlio della shell che sta eseguendo il programma loop (che chiameremo processo loop) che al processo di shell. Il processo loop alla ricezione del segnale SIGINT esegue l'azione di default che è la terminazione anormale del processo. Invece, il processo di shell, dato che è una shell interattiva che deve rimanere in vita, prima di cominciare ad accettare comandi dall'utente predispone le cose in modo da ignorare il segnale SIGINT e quindi alla ricezione del segnale appunto lo ignora. Si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FigureSegnali.mp4

Proviamo ora a mandare in esecuzione lo stesso programma di prima (sempre con ridirezione dello standard output), ma usando questa volta l'esecuzione di background:

```
soELab@Lica04:~/segnali/PrimiEsempi$ loop > /dev/null &
[1] 10275
soELab@Lica04:~/segnali/PrimiEsempi$ ^C
```

```
soELab@Lica04:~/segnali/PrimiEsempi$ ^C
soELab@Lica04:~/segnali/PrimiEsempi$ ^C
soELab@Lica04:~/segnali/PrimiEsempi$ ps
  PID  TTY          TIME CMD
10046  pts/2    00:00:00 bash
10275  pts/2    00:00:20 loop
10291  pts/2    00:00:00 ps
```

Purtroppo (o per fortuna …) non riusciamo ad abortire il processo loop (cioè a farlo terminare) con il CTRL-C: perché? Il processo creato dalla shell per eseguire il programma in background, esattamente come la shell interattiva, predispone le cose in modo da ignorare il segnale SIGINT e quindi alla ricezione di questo segnale appunto lo ignora. Quindi in questo caso abbiamo sempre due processi che ricevono il segnale SIGINT, ma entrambi lo ignorano. Si veda ancora anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FigureSegnali.mp4

Dobbiamo quindi trovare un altro metodo per far terminare il processo loop lanciato in background; lo possiamo fare usando il comando kill che invia al processo specificato (di default) il segnale SIGTERM:

```
soELab@Lica04:~/segnali/PrimiEsempi$ kill 10275
```

Verifichiamo che sia terminato il processo:

```
soELab@Lica04:~/segnali/PrimiEsempi$ ps
  PID  TTY          TIME CMD
10046  pts/2    00:00:00 bash
10294  pts/2    00:00:00 ps
[1]+  Terminated              loop > /dev/null
```

NOTA BENE: Il S.O. riporta appunto che il processo che stava eseguendo il comando loop (con la ridirezione dello standard output) è terminato (in modo anomalo).

Se per caso non risultasse abortito, allora possiamo intervenire in maniera più decisa usando sempre il comando kill, ma specificando di mandare il segnale SIGKILL; questo si ottiene specificando una opzione nel comando kill e indicando come valore della opzione il 9 che corrisponde appunto al numero del SIGKILL:

```
soELab@Lica04:~/segnali/PrimiEsempi$ loop > /dev/null &
[1] 10302
soELab@Lica04:~/segnali/PrimiEsempi$ kill -9 10302
soELab@Lica04:~/segnali/PrimiEsempi$ ps
  PID  TTY          TIME CMD
10046  pts/2    00:00:00 bash
10303  pts/2    00:00:00 ps
 [1]+  Killed                  loop > /dev/null
```

NOTA BENE: Il S.O. riporta appunto che il processo che stava eseguendo il comando loop (con la ridirezione dello standard output) è stato ucciso (killed).

Passiamo ora a considerare il segnale che UNIX invia per segnalare che è stata eseguita una azione illegale, SIGILL; per farlo prendiamo in considerazione un semplice esempio di programma C che contiene un errore nella gestione della memoria (è l'esempio più semplice da realizzare):

```
soELab@Lica04:~/segnali/PrimiEsempi$ cat illegal.c
#include <stdio.h>
#include <stdlib.h>

char *s=(char *)64000;

int main()
{
puts("SONO UN PROGRAMMA CHE VA IN ERRORE e CREA UN CORE");
puts("Dammi una stringa");
puts("ATTENZIONE NON ABBIAMO ALLOCATO LA MEMORIA PER LA STRINGA");
scanf("%s", s);
puts("Non si arrivera' mai qui!!!!!!");
exit(0);
}
```

Vediamo quindi cosa succede mandando in esecuzione l'eseguibile:

```
soELab@Lica04:~/segnali/PrimiEsempi$ illegal
SONO UN PROGRAMMA CHE VA IN ERRORE e CREA UN CORE
Dammi una stringa
ATTENZIONE NON ABBIAMO ALLOCATO LA MEMORIA PER LA STRINGA
Fgzgfjgfd                    <== stringa a caso …
Segmentation fault (core dumped)
soELab@Lica04:~/segnali/PrimiEsempi$
```

Il kernel al tentativo di scrivere in un'area di memoria non allocata ha inviato al processo che esegue il programma illegal il segnale SIGILL che di default provoca la terminazione (anomala) con la generazione di un file di core. Vediamo che cosa è usando man core:

CORE(5) Linux Programmer's Manual CORE(5)

NAME
core - core dump file

DESCRIPTION
The default action of certain signals is to cause a process to terminate and produce a core dump file, a disk file containing an image of the process's memory at the time of termination. This image can be used in a debugger (e.g., gdb(1)) to inspect the state of the program at the time that it terminated. A list of the signals which cause a process to dump core can be found in signal(7).
A process can set its soft RLIMIT_CORE **resource limit** to place an upper limit on the size of the core dump file that will be produced if it receives a "core dump" signal; see getrlimit(2) for details.

Dovrebbe quindi essere stato creato un file di nome core nel direttorio corrente; verifichiamolo:

```
soELab@Lica04:~/segnali/PrimiEsempi$ ls
illegal  illegal.c  loop  loop.c  old  read.me
```

Ma in questo caso non troviamo nessun file di nome core! La ragione è che il limite per la lunghezza del file core è stata settata a zero e lo possiamo verificare con il seguente comando:

```
soELab@Lica04:~/segnali/PrimiEsempi$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31431
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31431
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

(Luc. Sincronizzazione in Unix: segnali 5-7)
Vediamo ora quali primitive possiamo usare per trattare i segnali, partendo dalla primitiva signal; vediamo il man signal:

SIGNAL(2) Linux Programmer's Manual SIGNAL(2)

NAME
signal - ANSI C signal handling

SYNOPSIS
```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION
The behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. *…*

signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler").

If the signal signum is delivered to the process, then one of the following happens:
** If the disposition is set to SIG_IGN, then the signal is ignored.*
** If the disposition is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs.*
** If the disposition is set to a function, then first either the disposition is reset to SIG_DFL, or the signal is blocked (see Portability below), and then handler is called with argument signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.*
The signals SIGKILL and SIGSTOP cannot be caught or ignored.

RETURN VALUE
signal() returns the previous value of the signal handler, or SIG_ERR on error. In the event of an error, errno is set to indicate the cause.

Si veda di nuovo anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FigureSegnali.mp4

(fine prima video-registrazione di domenica 3/05/2020)

(Luc. Sincronizzazione in Unix: segnali 8-9)
Vediamo ora un primo esempio dove gestiamo il segnale SIGINIT (quello corrispondente al CTRL-C): ==> usare directory ~/segnali

```
soELab@Lica04:~/segnali$ cat psignalNew.c
/* FILE: psignalNew.c */
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

void catchint(int); /* prototipo/dichiarazione della funzione che rappresenta il gestore/handler del
segnale SINGINT */

int main ()
{       int i;

        signal(SIGINT, catchint);          <== il nome di una funzione corrisponde al puntatore al suo codice!
/* aggancia il segnale alla funzione gestore/handler di nome catchint */

for (;;)        /* ciclo infinito */       <== ciclo infinito, anche se diverso dal programma loop.c
        for (i =0 ; i < 32000;  i++)
                printf (" i vale %d\n", i);
}

void catchint (int signo)
{
int fd;
char s[100] = "Segnale nro  ";
char s1[5];
char s2[] = " e count = ";
char s3[] = "\n";
static int count = 0;  /* variabile definita static e quindi allocata non nello stack, ma nei dati
globali che hanno tempo di vita pari a quella del programma e quindi del processo! */
```

```
 /* NON si disabilita il segnale SIGINT: dato che abbiamo verificato che si usi la semantica BSD */
        sprintf(s1, "%d", signo);
        strcat(s, s1);
        strcat(s, s2);
        count++;
        sprintf(s1, "%d", count);
        strcat(s, s1);
        strcat(s, s3);
        if ((fd = open("sig.log", O_WRONLY)) < 0) exit(1); <== si usa il file sig.log in scrittura!
        lseek(fd, 0L, 2);       /* se la open ha avuto sucesso, spostiamo l'I/O pointer alla fine in
modo da appendere la stringa che abbiamo 'costruito' */
        write(fd, s, strlen(s));
 /* non si prevedono azioni di terminazione: ritorno al segnalato, SENZA BISOGNO DI RIPRISTINARE la
catch function sempre secondo la semantica BSD */
}
```

Verifichiamo per prima cosa se il file sig.log è vuoto:

```
soELab@Lica04:~/segnali$ ls -l sig.log
-rw-r--r-- 1 soELab users 0 May  3 14:16 sig.log
```

Quindi ora possiamo verificare il funzionamento del programma (avendo cura di ridirigere lo standard output per poter vedere bene l'echo sul video del CTRL-C):

```
soELab@Lica04:~/segnali$ psignalNew > /dev/null
^C^C^C^C^C^C                          <== abbiamo usato 6 volte il CTRL-C
^Z                                    <== ora dobbiamo usare un altro segnale che non abbiamo 'catturato', proviamo con CTRL-Z!
[1]+  Stopped                 psignalNew > /dev/null
soELab@Lica04:~/segnali$
```

Siamo riusciti ad avere di nuovo il prompt dei comandi; vediamo cosa è successo al processo:

```
soELab@Lica04:~/segnali$ ps -lf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S soELab     10046   10045  0  80   0 -  5772 wait   16:35 pts/2    00:00:00 -bash
0 T soELab     11217   10046 80  80   0 -  1127 signal 18:43 pts/2    00:00:17 psignalNew
0 R soELab     11232   10046  0  80   0 -  9176 -      18:45 pts/2    00:00:00 ps -lf
```

Dove **T** dal man di ps risulta: *stopped, either by a **job control signal** or because it is being traced*

Dato che il processo non ci serve più lo terminiamo:

```
soELab@Lica04:~/segnali$ kill 11217
```

Verifichiamo che sia terminato:

```
soELab@Lica04:~/segnali$ ps -lf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S soELab     10046   10045  0  80   0 -  5772 wait   16:35 pts/2    00:00:00 -bash
0 T soELab     11217   10046 80  80   0 -  1127 signal 18:43 pts/2    00:00:17 psignalNew
0 R soELab     11232   10046  0  80   0 -  9176 -      18:45 pts/2    00:00:00 ps -lf
```

Poiché non risulta terminato, dobbiamo intervenire più decisamente, usando l'opzione -9 nel comando kill che corrisponde (come visto prima) ad inviare il segnale SIGKILL al processo:

```
soELab@Lica04:~/segnali$ kill -9 11217
[1]+  Killed                  psignalNew > /dev/null
```

Vediamo che cosa è stato scritto sul file:

```
soELab@Lica04:~/segnali$ cat sig.log
Segnale nro  2 e count = 1
Segnale nro  2 e count = 2
Segnale nro  2 e count = 3
Segnale nro  2 e count = 4
Segnale nro  2 e count = 5
Segnale nro  2 e count = 6
```

(Luc. Sincronizzazione in Unix: segnali 10-13)
Discorso legato alla sincronizzazione dei processi: molto spesso possono essere usati in alternative le pipe (si consideri ad esempio il compito visto la volta scorsa), ma ci saranno dei testi di compiti che richiedono esplicitamente di usare la primitiva kill (con il segnale SIGKILL) dato che si dovranno uccidere dei processi!

Accennato a cosa farà l'esempio di uso della primitiva kill, della primitiva pause e della primitiva sleep: un processo padre e un processo figlio si scambiano il segnale SIGUSR1 'simulando' una specie di ping-pong; i processi eseguono entrambi due cicli infiniti e quindi dovremo terminarli con il CTRL-C (segnale SIGINT).

(fine seconda video-registrazione di domenica 3/05/2020)

Lezione Undicesimo Lunedì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Mer. 06 e Gio. 07/05/2020

(Luc. Sincronizzazione in Unix: segnali 10-13)

Vediamo un esempio di uso della primitiva kill, della primitiva pause e della primitiva sleep: un processo padre e un processo figlio si scambiano il segnale SIGUSR1 'simulando' una specie di ping-pong; i processi eseguono entrambi due cicli infiniti e quindi dovremo terminarli con il CTRL-C (segnale SIGINT): ==> usare directory ~/segnali/Pause

Si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraEsempioKillEPause.mp4

```
soELab@Lica04:~/segnali/Pause$ cat pausekillNew.c
/* FILE: pausekillNew.c  */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int ntimes = 0; /* variabile globale */

void handler(int signo)
{
     printf ("Processo %d ricevuto #%d volte il segnale %d\n", getpid(), ++ntimes, signo);
```

<== il codice di questa funzione di trattamento di segnale riporta solo delle informazioni all'utente: per gli scopi di questo esempio poteva anche essere un codice nullo

```
}

int main ()
{
int pid, ppid;
```

signal(SIGUSR1, handler); **<== il padre aggancia il segnale SIGUSR1 alla funzione handler: in questo modo alla ricezione di tale segnale né il padre né il figlio eseguiranno l'azione di default che provocherebbe la morte del processo, ma verrà eseguita la funzione di nome handler!**

```
if ((pid = fork()) < 0)
```

```
        /* fork fallita */
        exit(1);
else if (pid == 0)  /* figlio */
        { /* l'aggancio al segnale viene ereditato */
                ppid = getppid(); /* PID del padre */
                for (;;) /* ciclo infinito */
                {
                        printf("FIGLIO %d\n", getpid());
                        sleep(1);
                        kill(ppid, SIGUSR1);
                        pause();
                }
        }
        else
        /* padre */
        {
                for(;;) /* ciclo infinito */
                {
                        printf("PADRE %d\n", getpid());
                        pause();
                        sleep(1);
                        kill(pid, SIGUSR1);
                }
        }
```

<== la sleep serve per essere sicuri che all'invio del segnale con la kill, l'altro processo (il padre) sia in pause!

<== la sleep serve per essere sicuri che all'invio del segnale con la kill, l'altro processo (il figlio) sia in pause!

} **<== NOTA BENE: il padre dato che è in un ciclo infinito NON può attendere la terminazione del figlio (anche lui nella stessa situazione) e né il figlio né il padre tornano alcun valore con exit!**

Verifichiamone il funzionamento:

```
soELab@Lica04:~/segnali/Pause$ pausekillNew
PADRE 9493
FIGLIO 9494
Processo 9493 ricevuto #1 volte il segnale 10
PADRE 9493
Processo 9494 ricevuto #1 volte il segnale 10
```

```
FIGLIO 9494
Processo 9493 ricevuto #2 volte il segnale 10
PADRE 9493
Processo 9494 ricevuto #2 volte il segnale 10
FIGLIO 9494
Processo 9493 ricevuto #3 volte il segnale 10
PADRE 9493
Processo 9494 ricevuto #3 volte il segnale 10
FIGLIO 9494
Processo 9493 ricevuto #4 volte il segnale 10
^C
```

<== abbiamo usato il CTRL-C, quindi il segnale SIGINT è stato inviato in questo caso a tre processi: il processo padre 9493 e il processo figlio 9494 che terminano entrambi e il processo shell, che non termina!

```
soELab@Lica04:~/segnali/Pause$
```

Vediamo cosa succede se eliminiamo le sleep(1): quindi nel codice andiamo a commentare le due linee corrispondenti!

```
soELab@Lica04:~/segnali/Pause$ cat pausekillWrong.c
/* FILE: pausekillWrong.c  */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int ntimes = 0; /* variabile globale */

void handler(int signo)
{
     printf ("Processo %d ricevuto #%d volte il segnale %d\n", getpid(), ++ntimes, signo);
}

int main ()
{
int pid, ppid;

signal(SIGUSR1, handler);
```

```
if ((pid = fork()) < 0)
        /* fork fallita */
        exit(1);
else if (pid == 0)  /* figlio */
        { /* l'aggancio al segnale viene ereditato */
                ppid = getppid(); /* PID del padre */
                for (;;) /* ciclo infinito */
                {
                        printf("FIGLIO %d\n", getpid());
                /*      sleep(1);  */            <== COMMENTATA!
                        kill(ppid, SIGUSR1);
                        pause();
                }
        }
        else
        /* padre */
        {
                for(;;) /* ciclo infinito */
                {
                        printf("PADRE %d\n", getpid());
                        pause();
                /*      sleep(1);  */            <== COMMENTATA!
                        kill(pid, SIGUSR1);
                }
        }
}
```

Verifichiamone il funzionamento:

```
soELab@Lica04:~/segnali/Pause$ pausekillWrong
PADRE 9639
FIGLIO 9640
PADRE 9639
Processo 9639 ricevuto #1 volte il segnale 10
```

<== entrambi i processi (padre 9639 e figlio 9640) sono bloccati: DEADLOCK!!! Sia il padre che il figlio sono in attesa del verificarsi di una situazione (l'arrivo di un segnale che li sblocca dalla primitiva pause) che non si può verificare!

^C **<== come prima dobbiamo usare il CTRL-C!**

```
soELab@Lica04:~/segnali/Pause$
```

NOTA BENE: le stampe che riescono a scrivere i due processi sullo standard input non è costante nel senso che esecuzioni diverse possono provocare stampe diverse: dipende tutto dalla velocità relativa di esecuzione dei due processi!

Verifichiamo provando ad esempio un'altra esecuzione:

```
soELab@Lica04:~/segnali/Pause$ pausekillWrong
PADRE 9754
FIGLIO 9755
Processo 9754 ricevuto #1 volte il segnale 10
PADRE 9754
Processo 9755 ricevuto #1 volte il segnale 10
FIGLIO 9755
PADRE 9754
Processo 9754 ricevuto #2 volte il segnale 10
```

<== entrambi i processi (padre 9754 e figlio 9755) sono di nuovo bloccati: DEADLOCK!!!

Verifichiamo usando un altro terminale che i due processi sono entrambi bloccati:

```
soELab@Lica04:~$ ps -elf | grep pause
0 S soELab   9754 9246  0  80   0 -  1127 pause  18:10 pts/2    00:00:00 pausekillWrong
1 S soELab   9755 9754  0  80   0 -  1127 pause  10:24 pts/2    00:00:00 pausekillWrong
0 S soELab   9765 9344  0  80   0 -  2218 pipe_w 10:24 pts/3    00:00:00 grep pause
```

(Ancora Luc. Sincronizzazione in Unix: segnali 11 e Luc. Sincronizzazione in Unix: segnali 14-15)

Vediamo un esempio di uso della primitiva alarm e della primitiva pause: un processo padre genera un processo figlio che installa un allarme: ==> usare sempre directory ~/segnali/Pause

```
soELab@Lica04:~/segnali/Pause$ cat pauseala.c
/* FILE: pauseala.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

#define TRUE  1
#define FALSE 0
#define BELLS "\007\007\007"

int alarm_flag = FALSE;
```

```
void setflag()
{
         printf("STAMPA DI DEBUGGING: Processo con pid = %d ha ricevuto il segnale SIGALRM che e' il
nro %d\n", getpid(), signo);
        alarm_flag = TRUE;
}

int main (int argc, char **argv)
{
        int pid, nsecs, j;

        if (argc <= 2) { printf ("Errore nel numero di parametri\n");
                         exit(1);
                       }
/* il primo parametro deve essere il numero di minuti che deve passare prima di far scattare
l'allarme e il resto dei parametri sono le stringhe che saranno scritte sullo standard output quando
scatta l'allarme */
        if ((nsecs = atoi(argv[1]) * 60 ) <= 0)
                { printf ("Errore nel valore del tempo\n");
                  exit(2);
                 }

        switch (pid = fork())
        {
        case -1: /* fork fallita */
                 printf("Errore in fork\n");
                 exit(1);
        case 0:  /* figlio */
                 break;
        default: /* padre */
                 printf("Creazione del processo %d\n", pid);
                 exit(0);
        }
```

<== il padre termina senza attendere il figlio e quindi simula l'esecuzione in background e il figlio viene ereditato da init (lo vedremo più avanti)!

```
        /* figlio */
        /* installa l'azione specifica da eseguire alla ricezione del segnale SIGALRM */
        signal(SIGALRM, setflag);
        /* attiva l'allarme per il numero di secondi calcolato dal primo parametro (fornito in
minuti) */
        alarm(nsecs);
        /* attende l'arrivo di un qualunque segnale */
        pause();
        /* se il segnale arrivato era quello dell'allarme, allora stampa su stadard output le
stringhe passate come parametri dal secondo in poi */
        if (alarm_flag == TRUE)
              { printf(BELLS); /* viene suonata la 'campanella' del terminale per 3 volte */
                for (j=2; j < argc; j++)
                        printf("%s ", argv[j]);
                printf("\n");
              }
        exit (0);
}
```

Verifichiamone il funzionamento:

1) senza parametri

```
soELab@Lica04:~/segnali/Pause$ pauseala
Errore nel numero di parametri
```

2) con parametri giusti

```
soELab@Lica04:~/segnali/Pause$ pauseala 1 passato un minuto
Creazione del processo 24870
soELab@Lica04:~/segnali/Pause$ ps -lf
```

<== dato che il processo padre è terminato, ed è tornato il prompt, possiamo verificare cosa sta facendo il figlio

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S soELab    9246  9245  0  80   0 -  5610 wait   09:28 pts/1    00:00:00 -bash
1 S soELab    9862     1  0  80   0 -  1047 pause  11:06 pts/1    00:00:00 pauseala 1 passato un minuto
0 R soELab    9866  9246  0  80   0 -  3891 -      11:06 pts/1    00:00:00 ps -lf
```

<== rimaniamo in attesa e dopo un minuto (o poco più) si ha la scritta da parte del figlio!

```
soELab@Lica04:~/segnali/Pause$STAMPA DI DEBUGGING: Processo con pid = 26919 ha ricevuto il segnale
SIGALRM che e' il nro 14
```

```
passato un minuto
```
<== subito prima di questa scritta è suonata la 'campanella'!

(Luc. Comunicazione in Unix: Pipe 16)
Torniamo a considerare l'esempio per calcolare la lunghezza di una pipe, dove useremo la primitiva alarm per fare terminare il processo quando si bloccherà sulla scrittura sulla pipe: ==> usare di nuovo il directory ~/pipe

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

int count;
void alrm_action(int signo);

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

if (pipe(p) < 0) { printf("Errore\n"); exit (1); }
```
<== N.B. l'unico processo si comporta come scrittore e come potenziale lettore e quindi NON riceve il segnale SIGPIPE!
```
signal(SIGALRM, alrm_action);

for (count = 0;;)
    {
/* settiamo l'allarme */
        alarm(10);
/* scrittura sulla pipe */
        write(p[1], &c, 1);
/* resettiamo l'allarme */
        alarm(0);
```
<== se si arriva qui, vuole dire che la write non ha bloccato il processo!
```

        if ((++count % 1024) == 0)
```

```
                printf ("%d caratteri nella pipe\n", count);
        }
}

void alrm_action(int signo)
{
        printf ("write bloccata dopo %d caratteri (ricevuto segnale SIGALRM che e' il nro %d)\n",
count, signo);
        exit(0);
}
```

Verifichiamone il funzionamento:

```
soELab@Lica04:~/pipe$ lungpipeConSegnali
1024 caratteri nella pipe
2048 caratteri nella pipe
3072 caratteri nella pipe
4096 caratteri nella pipe
5120 caratteri nella pipe
…
64512 caratteri nella pipe
65536 caratteri nella pipe
write bloccata dopo 65536 caratteri (ricevuto segnale SIGALRM che e' il nro 14)
soELab@Lica04:~/pipe$
```

(fine prima video-registrazione di mercoledì 6/05/2020)

(Luc. Comunicazione in Unix: Pipe 14-15)
Torniamo a considerare l'esempio di una pipe senza lettore e gestiamo il segnale SIGPIPE che viene mandato al processo scrittore (nel nostro esempio il figlio): ==> usare di nuovo il directory ~/pipe

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

```
#define MSGSIZE 5
#define TRUE 1
#define FALSE 0

int flag = FALSE; /* variabile globale */

void Azione(int sig)
{       printf("Arrivato segnale # %d\n", sig);
        flag = TRUE;
}

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

if (argc != 2)
{   printf("Numero dei parametri errato\n");
    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;
        signal(SIGPIPE, Azione); /* si aggancia Azione per trattare il segnale SIGPIPE */
        close (piped [0]);      /* il figlio CHIUDE il lato di lettura */
        if ((fd = open(argv[1], O_RDONLY)) < 0)
```

<== alla ricezione del segnale SIGPIPE si cambia valore alla variabile flag!

```
        {   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); /* andiamo a introdurre un ritardo per essere sicuri che quando comincia a scrivere
il processo padre sia terminato */
        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);
        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';
                if (!flag) write (piped[1], mess, MSGSIZE);
                else
                { puts("NON C'E' LETTORE");
                 exit(-1); /* torniamo al padre un -1 che sara' interpretato come 255 e quindi
identificato come errore */
                }
                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 COMMENTARE LA wait
((int *)0); */ /* padre NON aspetta il figlio */
exit (0);
```

```
}
```

Verifichiamone il funzionamento:

```
soELab@Lica04:~/pipe$ pipe-newSenzalettoreConHandler input.txt
Padre 22889 sta per iniziare a leggere i messaggi dalla pipe
```

<== dato che il processo padre è terminato, è tornato il prompt!

```
soELab@Lica04:~/pipe$ Figlio 22890 sta per iniziare a scrivere una serie di messaggi, ognuno di lunghezza 5, sulla pipe dopo averli letti dal file passato come parametro
Arrivato segnale # 13
NON C'E' LETTORE
```

OSSERVAZIONE IMPORTANTE: ci sono dei testi di esame dove si parla di comunicazione/sincronizzazione e di solito in questi testi si trova una nota (ma potrebbe anche non esserci) che chiarisce che per questo tipo di comunicazione/sincronizzazione lo studente può utilizzare i segnali.
Esistono invece dei testi (nel seguito un esempio) in cui l'utilizzo dei segnali non è eludibile e quindi vanno per forza utilizzati!

Testo parte C esame del 15 Luglio 2015:
La parte in C accetta un numero variabile **N** di parametri (con **N** maggiore o uguale a **2)** che rappresentano **N** nomi di file (**F1, F2,... FN**).
Il processo padre deve, per prima cosa, creare un file di nome "Merge" e, quindi, deve generare **N processi figli** (**P0** … **PN-1**) ognuno dei quali è associato ad uno dei file **Fi**. Ogni processo figlio **Pi** deve leggere i caratteri del file associato **Fi** ~~**sempre** fino alla fine~~ ***solo*** ***dopo*** aver ricevuto l'indicazione dal padre di procedere. Infatti, i processi figli devono attenersi a questo **schema di comunicazione/sincronizzazione con il padre**: il figlio **P0**, ricevuta l'indicazione dal padre che può procedere, legge il primo carattere e lo comunica al padre che lo scrive sul file "Merge"; il figlio **P1**, ricevuta l'indicazione dal padre che può procedere, legge il primo carattere e lo comunica al padre che lo scrive sul file "Merge" etc. fino al figlio **PN-1**, ricevuta l'indicazione dal padre che può procedere, legge il primo carattere e lo comunica al padre che lo scrive sul file "Merge"; questo schema deve continuare per gli altri caratteri e deve terminare appena un processo **Pi** non riesce più a leggere dal proprio file **Fi** a causa del raggiungimento della fine del file: il processo padre, appena si accorge che un processo **Pi ha concluso la propria lettura e quindi è terminato**, **non deve inviare più indicazioni agli altri figli che possono procedere e deve terminare forzatamente gli altri figli (con un apposito segnale).**
Al termine, il processo figlio **Pi** associato al file **Fi** più corto fra quelli passati deve ritornare al padre l'ultimo carattere letto dal file associato **Fi**; il padre deve stampare su standard output il PID di ogni figlio con l'indicazione di terminazione anormale o normale e in questo caso il valore ritornato dal figlio.
La soluzione si trova sul sito alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/15Lug15/15Lug15.c

Consideriamo di usare i seguenti due file, creati in questa maniera per rendere più evidente il 'mescolamento' dei caratteri:

```
soELab@lica04:~/15Lug15$ more F1 F2
::::::::::::::
F1
::::::::::::::
1234567890
::::::::::::::
F2
::::::::::::::
abcdefghilmnopqrstuvz
```

Vediamone il funzionamento in vari casi:

a)
```
soELab@lica04:~/15Lug15$ 15Lug15 F1 F2
Numero di processi da creare 2
Sono il figlio 23245 di indice 0
Sono il figlio 23246 di indice 1
Valore di primoindice 0
Il figlio con pid=23245 ha ritornato il carattere 0
Figlio con pid 23246 terminato in modo anomalo
soELab@lica04:~/15Lug15$ cat Merge
1a2b3c4d5e6f7g8h9i0lsoELab@lica04:~/15Lug15$ more Merge
1a2b3c4d5e6f7g8h9i0l
soELab@lica04:~/15Lug15$ rm Merge
rm: remove regular file 'Merge'? y
```

b)
```
soELab@lica04:~/15Lug15$ 15Lug15 F1 F1 F2 F2
Numero di processi da creare 4
Sono il figlio 23284 di indice 0
Sono il figlio 23286 di indice 2
Sono il figlio 23285 di indice 1
Sono il figlio 23287 di indice 3
Valore di primoindice 0
Il figlio con pid=30105 ha ritornato il carattere 0
Figlio con pid 23286 terminato in modo anomalo
Figlio con pid 23285 terminato in modo anomalo
Figlio con pid 23287 terminato in modo anomalo
```

```
soELab@lica04:~/15Lug15$ more Merge
11aa22bb33cc44dd55ee66ff77gg88hh99ii00ll
soELab@lica04:~/15Lug15$ rm Merge
rm: remove regular file 'Merge'? y
```

c) *soELab@lica04:~/15Lug15$ 15Lug15 F2 F1*

```
Numero di processi da creare 2
Sono il figlio 23295 di indice 0
Sono il figlio 23296 di indice 1
Valore di primoindice 1
Il figlio con pid=23296 ha ritornato il carattere 0
Figlio con pid 23295 terminato in modo anomalo
soELab@lica04:~/15Lug15$ more Merge
a1b2c3d4e5f6g7h8i9l0m
```

(fine seconda video-registrazione di giovedì 7/05/2020)

Lezione Undicesimo Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Dom. 10/05/2020

(Luc. Comunicazione in Unix: Pipe 17-20)

Passiamo ora l'ultimo argomento delle pipe che era rimasto indietro: vediamo le FIFO (o pipe con nome, named pipe) e per prima cosa verifichiamo cosa risulta da man fifo

```
FIFO(7)                    Linux Programmer's Manual                    FIFO(7)

NAME
       fifo - first-in first-out special file, named pipe

DESCRIPTION
A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the
filesystem.  It can be  opened  by multiple processes  for  reading or writing.  When processes are
exchanging data via the FIFO, the kernel passes all data internally without writing  it to  the
filesystem.  Thus, the FIFO special file has no contents on the filesystem; the filesystem entry
merely serves as a reference point  so that processes can access the pipe using a name in the
filesystem.

The kernel maintains exactly one pipe object for each FIFO special file that is opened by at least
one process.  The FIFO  must  be  opened  on both  ends  (reading and writing) before data can be
passed.  Normally, opening the FIFO blocks until the other end is opened also.
```

Vediamo ora con un semplice esempio di uso, un processo (recfifo) che si comporta come server/gestore/consumatore e che riceve le richieste di eseguire un qualche compito da vari processi clienti/sender/produttori : ==> usare directory ~/fifo

Si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/FiguraEsempioFIFO.mp4

soELab@Lica04:~/fifo$ cat recfifo.c **<== codice del processo server**

```
/* FILE: recfifo.c */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MSGSIZ 60
```

```
int main()
{
int fd;
char msgbuf[MSGSIZ+1];

/* Apertura FIFO */
if ((fd = open ("/tmp/fifo", O_RDWR)) < 0)
        { printf("Errore in open\n"); exit(1); }

/* Ricezione messaggi */
for  (;;)
{      if (read(fd, msgbuf, MSGSIZ+1) < 0)
        { printf("Errore in lettura\n"); exit(2); }

        printf("Messaggio ricevuto: %s\n", msgbuf);
  }
}
```

<== N.B. questo programma leggerà solo dalla fifo, ma è importante che faccia finta di poter anche scrivere in modo che la read sia bloccante (e si eviti il fenomeno di attesa attiva)!

<== un server normalmente ha un codice che corrisponde ad un ciclo infinito!

Verifichiamone il funzionamento:

```
soELab@Lica04:~/fifo$ recfifo &
[1] 29070
Errore in open
```

Il problema che la fifo (/tmp/fifo) NON esiste ==> verifichiamolo:

```
soELab@Lica04:~/fifo$ ls -l /tmp/fifo
ls: cannot access /tmp/fifo: No such file or directory
```

Dobbiamo quindi crearla, o con un comando apposito:

```
soELab@Lica04:~/fifo$  mkfifo /tmp/fifo
soELab@Lica04:~/fifo$ ls -l /tmp/fifo
prw-r--r-- 1 soELab users 0 Jun  6 09:17 /tmp/fifo
```

oppure ad esempio con questo programma:

```
soELab@Lica04:~/fifo$ cat creafifo.c
/* FILE: creafifo.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <stdlib.h>
#include <errno.h>

int main()
{
if (mkfifo("/tmp/fifo", 0660) < 0)
        perror("Errore\n");
exit(errno);
}
```

<== entra sempra in gioco il valore di umask (vedi man –s 3 mkfifo)!

Se proviamo ad usare questo programma adesso che /tmp/fifo esiste già, avremo un errore:

```
soELab@Lica04:~/fifo$ creafifo
Errore
: File exists
```

Quindi per provare ad usare questo programma prima dobbiamo cancellare /tmp/fifo:

```
soELab@Lica04:~/fifo$ rm /tmp/fifo
rm: remove fifo '/tmp/fifo'? y
```

A questo punto proviamo di nuovo a creare la fifo con il programma e verifichiamo che esista:

```
soELab@Lica04:~/fifo$ creafifo
soELab@Lica04:~/fifo$ ls -l /tmp/fifo
prw-r----- 1 soELab users 0 Jun  6 09:21 /tmp/fifo
```

Se ci interessa che i processi che possono spedire sulla fifo siano anche processi del gruppo del proprietario, allora dobbiamo modificare i diritti di /tmp/fifo:

```
soELab@Lica04:~/fifo$ chmod g+w /tmp/fifo
soELab@Lica04:~/fifo$ ls -l /tmp/fifo
prw-rw---- 1 soELab users 0 Jun  6 09:21 /tmp/fifo
```

Ora che abbiamo creato la fifo che ci serve, possiamo attivare il processo server (NOTA BENE IN BACKGROUND):

```
soELab@Lica04:~/fifo$ recfifo &
[1] 15483
```

Verifichiamo che sia stato creato:

```
soELab@Lica04:~/fifo$ ps -lf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S soELab   13997 13996  0  80   0 -  5709 wait   12:10 pts/3    00:00:00 -bash
0 S soELab   15483 13997  0  80   0 -  1094 pipe_w 14:15 pts/3    00:00:00 recfifo
0 R soELab   15484 13997  0  80   0 -  9176 -      14:16 pts/3    00:00:00 ps -lf
```

Vediamo un esempio di possibile processo client che scrive sulla fifo dei messaggi che quindi verranno ricevuti dal processo server (recfifo): i messaggi saranno delle semplici stringhe che verranno richieste all'utente!

```
soELab@Lica04:~/fifo$ cat sendfifo.c
/* FILE: sendfifo.c */
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MSGSIZ 60

int main(argc, argv)
int argc;
char **argv;
{
int fd, i, nwrite;
char msgbuf[MSGSIZ+1];

if (argc != 1) { printf ("Errore nel numero parametri (NON CI VOGLIONO PARAMETRI)\n"); exit (1); }

/* Apertura FIFO */
if ((fd = open ("/tmp/fifo", O_WRONLY | O_NDELAY)) < 0)
```

<== spieghiamo dopo come mai c'è bisogno anche di O_NDELAY!

```
        { printf("Errore in open\n"); exit(2); }

/* Spedizione messaggi dopo che abbiamo chiesto all'utente cosa vuole spedire (per semplicita' usiamo
solo delle stringhe) */
i=0;
while (printf("Fornisci una stringa che verra' inviata sulla FIFO\n"),scanf("%s", msgbuf) != EOF)
{
        if ((nwrite = write(fd, msgbuf, MSGSIZ+1)) <= 0)
                { printf("Errore in scrittura\n"); exit(3); }
        else i++;
}
```

```
printf("SONO IL CLIENT E HO SPEDITO %d MESSAGGI SULLA FIFO\n", i);
exit(0);
}
```

<== Da un altro terminale e usando anche un altro utente ad esempio sonod, andiamo a lanciare alcuni processi client (sendfifo)

```
sonod@Lica04:/home/soELab/fifo$ ./sendfifo
Fornisci una stringa che verra' inviata sulla FIFO
ciao
Fornisci una stringa che verra' inviata sulla FIFO
come
Fornisci una stringa che verra' inviata sulla FIFO
stai?
Fornisci una stringa che verra' inviata sulla FIFO
SONO IL CLIENT E HO SPEDITO 3 MESSAGGI SULLA FIFO
sonod@Lica04:/home/soELab/fifo$
```

<== Usiamo ^D per indicare la fine delle standard input!

Sul terminale al quale è agganciato recfifo (quello che stavamo usando prima), vedremo questo:

```
soELab@Lica04:~/fifo$ Messaggio ricevuto: ciao
Messaggio ricevuto: come
Messaggio ricevuto: stai?
```

Facciamo ora la prova che sia importante nel processo client andare a fare la open con O_NDELAY: terminiamo quindi il processo server recfifo

```
soELab@Lica04:~/fifo$ kill 15483
[1]+  Terminated              recfifo
soELab@Lica04:~/fifo$ ps
  PID TTY          TIME CMD
13997 pts/1    00:00:00 bash
15639 pts/1    00:00:00 ps
```

Torniamo ora sull'altro terminale:

<== Proviamo di nuovo a lanciare un processo client (sendfifo)

```
sonod@Lica04:/home/soELab/fifo$ ./sendfifo
Errore in open
```

<== La open torna -1 e quindi il processo termina e torna il pompt dei comandi

```
sonod@Lica04:/home/soELab/fifo
```

Consideriamo quindi un programma uguale a sendfifo.c ma che nella open non abbia l'OR bit-a-bit con O_NDELAY (programma sendfifo1.c); l'effetto sarà questo:

<== Proviamo di nuovo a lanciare un processo client (sendfifo1)

```
sonod@Lica04:/home/soELab/fifo$ ./sendfifo1
```

<== La open non fallisce e il processo rimane bloccato sulla open in attesa che venga creato il servitore!

Ricreiamo a questo punto, sul terminale precedente, il servitore recfifo (sempre in background):

```
soELab@Lica04:~/fifo$ recfifo &
[1] 15622
```

<== TORNIAMO NEL TERMINALE DI sonod DOVE TROVEREMO CHE IL PROCESSO SI È SBLOCCATO

```
sonod@Lica04:/home/soELab/fifo$ ./sendfifo1
Fornisci una stringa che verra' inviata sulla FIFO
ancora
Fornisci una stringa che verra' inviata sulla FIFO
delle
Fornisci una stringa che verra' inviata sulla FIFO
stringhe
Fornisci una stringa che verra' inviata sulla FIFO
Per
Fornisci una stringa che verra' inviata sulla FIFO
provare
Fornisci una stringa che verra' inviata sulla FIFO
SONO IL CLIENT E HO SPEDITO 5 MESSAGGI SULLA FIFO
sonod@Lica04:/home/soELab/fifo$
```

<== Usiamo ^D per indicare la fine delle standard input!

```
soELab@Lica04:~/fifo$ Messaggio ricevuto: ancora
Messaggio ricevuto: delle
Messaggio ricevuto: stringhe
Messaggio ricevuto: per
Messaggio ricevuto: provare
```

Il servitore riceve immediatamente i messaggi inviati dal processo cliente sendfifo1 (che si sblocca dalla open) e il terminale di sonod si sblocca mostrando di nuovo il prompt dei comandi!

(fine prima video-registrazione di domenica 10/05/2020)

Illustrato il testo della seconda prova in itinere del 26 Maggio 2017: si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/26Mag2017-IIProvaInItinere.mp4

Analizzato lo schema di comunicazione a pipeline, in particolare il numero di pipe che servono (N, e cioè quanti i processi figli) e il fatto che nella soluzione ogni figlio Pi legge dalla pipe i-1 e scrive sulla pipe i; al padre arriva una singola struttura (con 3 campi); è necessario che il padre salvi in un array creato dinamicamente i pid dei figli.
La soluzione si trova alla URL:- http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/26Mag17/26Mag17.c

Vediamo il contenuto di 3 file di prova che contengono solo caratteri alfabetici (ma è assolutamente non significativo, potrebbero essere presenti qualunque carattere, alfanumerico, numerico, caratteri di punteggiatura):

```
soELab@lica04:~/26Mag17$ more F1 F2 F3
::::::::::::::
F1
::::::::::::::
abcdeeeefghileeemnopqrsteeuvz

::::::::::::::
F2
::::::::::::::
abcdefghileeemnopqrsteeuvz

::::::::::::::
F3
::::::::::::::
abcdefghileemnopqrsteeuvz
```

Mostrato alcune invocazioni di prova che daranno tutte lo stesso risultato finale:

```
soELab@lica04:~/26Mag17$ 26Mag17 F1 F2 F3 e
Carattere da cercare e
Numero di processi da creare 3
Sono il figlio 5086 e sono associato al file F1
Sono il figlio 5087 e sono associato al file F2
Sono il figlio 5088 e sono associato al file F3
Il figlio di indice 0 e pid 5086 ha trovato il numero massimo di occorrenze 9 del carattere e nel file F1
```

```
I figli hanno trovato in totale 20 occorrenze del carattere e nei file
Il figlio con pid=5086 ha ritornato 0 (se > di 2 problemi)
Il figlio con pid=5088 ha ritornato 2 (se > di 2 problemi)
Il figlio con pid=5087 ha ritornato 1 (se > di 2 problemi)
soELab@lica04:~/26Mag17$ 26Mag17 F3 F2 F1 e
Carattere da cercare e
Numero di processi da creare 3
Sono il figlio 5090 e sono associato al file F3
Sono il figlio 5091 e sono associato al file F2
Sono il figlio 5092 e sono associato al file F1
Il figlio di indice 2 e pid 5092 ha trovato il numero massimo di occorrenze 9 del carattere e nel
file F1
I figli hanno trovato in totale 20 occorrenze del carattere e nei file
Il figlio con pid=5090 ha ritornato 0 (se > di 2 problemi)
Il figlio con pid=5091 ha ritornato 1 (se > di 2 problemi)
Il figlio con pid=5092 ha ritornato 2 (se > di 2 problemi)
soELab@lica04:~/26Mag17$ 26Mag17 F2 F1 F3 e
Carattere da cercare e
Numero di processi da creare 3
Sono il figlio 5094 e sono associato al file F2
Sono il figlio 5095 e sono associato al file F1
Sono il figlio 5096 e sono associato al file F3
Il figlio di indice 1 e pid 5095 ha trovato il numero massimo di occorrenze 9 del carattere e nel
file F1
I figli hanno trovato in totale 20 occorrenze del carattere e nei file
Il figlio con pid=5094 ha ritornato 0 (se > di 2 problemi)
Il figlio con pid=5096 ha ritornato 2 (se > di 2 problemi)
Il figlio con pid=5095 ha ritornato 1 (se > di 2 problemi)
```

(fine seconda video-registrazione di domenica 10/05/2020)

Lezione Dodicesimo Lunedì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Mer. 13 e Gio. 14/05/2020

(Ripreso Luc. Unix: azioni primitive per gestione processi 9)

(Luc. Unix: azioni primitive per gestione processi 25)
Passiamo ora a capire a cosa serve lo stato ZOMBIE di un processo: il processo figlio può terminare prima che il padre faccia la wait e quindi, dato che il figlio (e il SO) non può sapere se il padre avrà bisogno del valore tornato dalla exit (e comunque il SO NON sa se viene tornato un valore specifico) il SO, in questo caso, fa transitare il processo nelle stato ZOMBIE, in cui sono già state recuperate tutte le risorse assegnata al processo a parte di descrittore di processo (al cui interno viene mantenuto il valore tornato da un processo con la exit). Il seguente esempio ci farà capire questa transizione di stato:
<== (usare directory ~/processi/InitZombie):

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

int main ()
{
int pid, pidFiglio, n, status;

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

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

```
        /* facciamo terminare subito il figlio dato che ci serve termini PRIMA che il padre abbia
effettuato la wait */
        exit(0);                                                    <== figlio termina subito per la nostra simulazione!
        }
/* padre */
printf("Ho generato il figlio %d\n", pid);
printf("Esecuzione del padre %d\n", getpid());
/* per fare in modo che il padre non faccia subito la wait gli facciamo aspettare un dato dall'utente
*/
printf("Fornisci un valore intero\n");
scanf("%d", &n);              <== facciamo in modo che il padre passi in stato sleeping in modo da essere sicuri che il figlio
termini prima che il padre faccia la wait!
printf("Numero letto dal padre %d\n", n);                           <== stampa solo di controllo!
if ((pidFiglio=wait(&status)) < 0)
{
        printf("Errore in wait\n");
        exit (2);
}

if (pid == pidFiglio) printf("Terminato figlio con PID = %d\n", pidFiglio);
else exit(3);

if (WIFEXITED(status) == 0)
    printf("Errore in status\n");
else
    printf("Per il figlio %d lo stato di EXIT e` %d\n", pid, WEXITSTATUS(status));
exit(0);
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/InitZombie$ figlio-zombieConStato
Ho generato il figlio 24240
Esecuzione del padre 24239
Fornisci un valore intero          <== il padre risulta a questo punto in attesa (ha effettuato la transizione di stato da running a
sleeping)
Esecuzione del figlio 24240        <== il figlio ora termina!
```

<== Da un altro terminale controlliamo la situazione del figlio (e del padre)

```
soELab@Lica04:~$ ps -elf | grep pts/3
5 S soELab   23785 23701  0  80   0 - 26997 -      15:27 ?        00:00:00 sshd: soELab@pts/3
0 S soELab   23786 23785  0  80   0 -  5707 wait   15:27 pts/3    00:00:00 -bash
0 S soELab   24239 23785  0  80   0 -  1127 wait_w 16:02 pts/3    00:00:00 figlio-zombieConStato
1 Z soELab   24240 24239  0  80   0 -     0 -      16:02 pts/3    00:00:00 [figlio-zombieCo] <defunct>
0 R soELab   24262 23686  0  80   0 -  2866 pipe_w 16:03 pts/2    00:00:00 grep pts/3
soELab@Lica04:~$
```

NOTA BENE:
1) si vede che la shell di partenza (processo con PID 23786) sta eseguendo una wait in attesa della terminazione del comando mandato in foreground e quindi è in stato sleeping!
2) si vede che il padre (processo con PID 24239) sta eseguendo una scanf in attesa del dato, da standard input, che deve fornire l'utente e quindi è in stato sleeping!
3) si vede che il figlio (processo con PID 24240), che ha eseguito una exit, in attesa che il padre recuperi le informazioni di terminazione, è quindi in stato zombie!

Ora torniamo al terminale precedente e forniamo il numero atteso dal padre:

```
12980
Numero letto dal padre 12980
```

<== Una volta fornito il numero, il processo padre viene risvegliato (transizione di stato da sleeping a running) e il padre, dopo aver stampato il valore fornito sullo standard output, passa ad eseguire la wait

```
Terminato figlio con PID = 24240
Per il figlio 24240 lo stato di EXIT e` 0
```

(Luc. Unix: azioni primitive per gestione processi 26)
Vediamo ora di capire cosa succede se un processo padre non aspetta i propri figli e termina, quindi, prima di loro.
Ricordiamo che è buona norma, però, che un processo padre (cioè un processo che ha eseguito delle fork) aspetti SEMPRE i propri figli.
Chiaramente, questo non può essere garantito se un processo muore a causa della ricezione di un segnale!
Si può derogare da questa regola solo se si vuole simulare/emulare il comportamento della esecuzione in background.
<== usare directory ~/processi/InitZombie: nel codice seguente, abbiamo invertito il comportamento del padre con quello del figlio!

```
soELab@Lica04:~/processi/InitZombie$ cat figlio-init.c
/* figlio-init.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{

int pid;

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

if (pid == 0)
        { /* figlio */
        int n=0;
        printf("Esecuzione del figlio %d\n", getpid());
        /* per fare in modo che il figlio non termini subito gli facciamo aspettare un dato
dall'utente */
        printf("Fornisci un valore intero\n");
        scanf("%d", &n);
        printf("Numero letto dal figlio %d\n", n);
        exit(0);
        }
/* padre */
printf("Ho generato il figlio %d. BYE BYE!!!\n", pid);
/* il padre termina senza fare la wait del figlio! */
exit(0);
}
```

Vediamo come funziona:

```
soELab@Lica04:~/processi/InitZombie$ figlio-init
Ho generato il figlio 24420. BYE BYE!!!
Esecuzione del figlio 24420
Fornisci un valore intero
soELab@Lica04:~/processi/InitZombie$
```

<== **torna il prompt perché il processo atteso dalla shell (il padre) è terminato!**

<== **Di nuovo, dall'altro terminale controlliamo la situazione del figlio (del padre no perché è terminato, ma dato che la shell lo aspetta non ci sono problemi), usando lo stesso comando in piping di prima. N.B. Dobbiamo usare comunque un altro terminale perché dal precedente abbiamo il figlio 'agganciato' che aspetta un numero intero e si potrebbero creare delle interferenze …**

```
soELab@Lica04:~$ ps -elf | grep pts/3
5 S soELab   23785 23701  0  80   0 - 26997 -      15:27 ?        00:00:00 sshd: soELab@pts/3
0 S soELab   23786 23785  0  80   0 -  5707 wait   15:27 pts/3    00:00:00 -bash
1 S soELab   24420     1  0  80   0 -  1127 wait_w 16:15 pts/3    00:00:00 figlio-init
0 R soELab   24429 23686  0  80   0 -  3891 pipe_w 16:16 pts/2    00:00:00 grep pts/03
soELab@Lica04:~$
```

NOTA BENE: si vede che il processo figlio (con PID 28095), che è in stato sleeping (perché sta eseguendo la scanf) è stato ereditato dal processo init (PID=1)!

Ora torniamo al terminale precedente e forniamo il numero atteso dal figlio:

```
soELab@Lica04:~/processi/InitZombie$ Numero letto dal figlio 4
```

OSSERVAZIONE: si hanno un po' di problemi di interferenza fra l'echo del numero digitato dall'utente e la printf del figlio. Notare che il prompt è sempre quello di prima!

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

(fine prima video-registrazione di mercoledì 13/05/2020)

Per i seguenti due comandi usiamo un server diverso (lica02 e non lica04) perché in particolare per il secondo dei due si vede meglio quello che è stato spiegato nelle slide.

Verifichiamo la presenza del processo init (quello con PID=1):

```
letizia@Lica02$ ps -elf | head -2
F S UID          PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1     0  0  80   0 -  8346 -      Feb20 ?        00:02:21 /sbin/init
```

Vediamo di verificare, in pratica, il fatto che il processo init crei un insieme di processi pronti per consentire l'accesso da una serie di terminali fisici (ricordarsi di usare lica02, dato che la gestione è cambiata nella versione più recente istallata su lica04):

```
letizia@Lica02$ ps -elf | grep getty
4 S root       901     1  0  80   0 -  3954 -      Feb20 tty4     00:00:00 /sbin/getty -8 38400 tty4
4 S root       879     1  0  80   0 -  3954 -      Feb20 tty5     00:00:00 /sbin/getty -8 38400 tty5
4 S root       889     1  0  80   0 -  3954 -      Feb20 tty2     00:00:00 /sbin/getty -8 38400 tty2
4 S root       890     1  0  80   0 -  3954 -      Feb20 tty3     00:00:00 /sbin/getty -8 38400 tty3
4 S root       892     1  0  80   0 -  3954 -      Feb20 tty6     00:00:00 /sbin/getty -8 38400 tty6
4 S root      1138     1  0  80   0 -  3954 -      Feb20 tty1     00:00:00 /sbin/getty -8 38400 tty1
0 S letizia  10385  9617  0  80   0 -  2218 pipe_w 16:27 pts/0    00:00:00 grep getty
```

NOTA BENE: A ogni processo getty (generato dal processo init, con PID 1) viene passata, in particolare, l'indicazione di quale terminale deve gestire!

Letto il testo della prova in itinere dell'8 Giugno 2011, in particolare la prima versione che si trova sul sito (versione a) e commentiamolo:
La parte in C accetta un numero variabile di parametri N+2 che rappresentano le seguenti informazioni: i primi N nomi di file (F0, F1, ... FN-1), mentre gli ultimi due devono essere considerati due singoli caratteri C1 e C2 (da controllare). Il processo padre deve generare N processi figli (P0 ... PN-1): ogni processo figlio è associato al corrispondente file Fi e al carattere C1. Ognuno di tali processi figli deve creare a sua volta un processo nipote (PP0 ... PPN-1) associato al corrispondente file Fi e al carattere C2. Ogni processo figlio Pi e ogni nipote PPi esegue concorrentemente andando a cercare nel file associato Fi tutte le occorrenze del carattere C1 per il figlio e C2 per il nipote, tenendo traccia delle rispettive posizioni PC1 e PC2 (in termini di long int) nel file corrispondente. In particolare, sia il figlio che il nipote deve comunicare una struttura secondo quanto specificato nel seguito: ogni volta che il figlio Pi trova una occorrenza di C1, comunica al nipote PPi una struttura con il carattere C1 e la posizione PC1 (in termini di long int) dell'occorrenza di C1 nel file corrispondente; il nipote a sua volta riceve questa informazione e, se verifica che la posizione PC2 dell'occorrenza corrente di C2 è minore, comunica al padre una struttura comprendente PC2 e C2 altrimenti una struttura comprendente PC1 e C1. Il padre ha il compito di stampare su standard output, rispettando l'ordine dei file, tutte le posizioni P ricevute e il carattere C associato riportando anche il nome del file corrispondente. Al termine, ogni processo nipote PPi deve ritornare al figlio Pi il numero di occorrenze di C2 trovate e ogni processo figlio Pi deve ritornare al padre il numero di occorrenze di C1 trovate (NOTA: si può supporre che il numero di occorrenze sia minore di 255): sia ogni figlio Pi e sia il padre devono stampare su standard output il PID di ogni nipote/figlio e il valore ritornato.

Si noti che la differenza fra i due testi caricati sul sito (testo a e testo b) riguarda solo il verso della comunicazione fra figli e nipote: nel caso del testo sopra riportato (testo a) il verso è dal figlio al nipote e poi è il nipote che comunica al padre, nel caso del testo non riportato (testo b) il verso è dal nipote al figlio e poi è il figlio che comunica al padre.

Il problema che si può presentare nella soluzione di questo testo (versione a) riguarda il caso in cui il processo nipote trovi un numero di occorrenze del carattere C2 maggiore del numero di occorrenze del carattere C1 trovate dal figlio; infatti, in questo caso, la situazione sarebbe la seguente:
- il nipote si troverebbe bloccato sulla read dal figlio in attesa della posizione calcolata dal figlio per confrontarla e poi mandare al padre;
- il figlio dal canto suo NON invia nulla al nipote perché ha esaurito le occorrenze del suo carattere e quindi termina il while di lettura del file e si bloccerebbe in attesa della terminazione del nipote (richiesta in esplicito dal testo).
La situazione prospettata in termini tecnici viene indicata come un DEADLOCK (o BLOCCO CRITICO) perché i due processi (nipote e figlio) sono bloccati in attesa di una condizione che non si può verificare: il nipote è bloccato in attesa di una posizione da figlio (che però non gliela può inviare perché non ha più trovato occorrenze) e il figlio è bloccato in attesa della terminazione del nipote (che però essendo bloccato non può terminare).
La soluzione scelta e che il figlio, appena termina la lettura del file alla ricerca del suo carattere, prima di fare la wait, invia un segnale SIGUSR1 al nipote. A questo punto, se il nipote fosse bloccato sulla read, la funzione handler agganciata a tale segnale lo fa terminare in modo normale riportando il numero di occorrenze (come richiesto dal testo, anche se NON sarà il conteggio corretto di tutte le occorrenze), altrimenti se il nipote fosse già terminato il segnale va perso (ma non ci sono problemi a riguardo).

La soluzione si trova sul sito alla URL http://didattica.agentgroup.unimore.it/didattica/SOeLab/SoluzioniCompiti/8Giu11/8Giu11-a.c

Usiamo per provare la soluzione i seguenti due file:

```
soELab@lica04:~/8Giu11$ more F3 F4
::::::::::::::
F3
::::::::::::::
abcdefghilmnopqrstuvz
ea
::::::::::::::
F4
::::::::::::::
abcdefghilmnopqrstuvz
e
```

I due file contengono sostanzialmente i caratteri dell'alfabeto italiano, in più, F3 contiene una ulteriore occorrenza del carattere 'e' e del carattere 'a', mentre il file F3 contiene una ulteriore occorrenza solo del carattere 'e'. Chiaramente i caratteri che passeremo, oltre ai suddetti file, saranno il carattere 'a' e il carattere 'e' che saranno rispettivamente cercati dai due figli e dai due nipoti.
Verifichiamone il funzionamento:

```
soELab@lica04:~/8Giu11$ 8Giu11-a F3 F4 a e
Caratteri da cercare a e e
Numero di processi da creare 2
Sono il processo figlio di indice 0 e pid 20412 sto per creare il nipote che leggera' sempre dal mio stesso file F3
Sono il processo figlio di indice 1 e pid 20413 sto per creare il nipote che leggera' sempre dal mio stesso file F4
Sono il processo nipote del figlio di indice 0 e pid 20414, associato al file F3
Il figlio di indice 0 ha trovato il carattere a nella posizione 0 nel file F3
Il nipote del figlio di indice 0 ha trovato il carattere e nella posizione 22 nel file F3
Sono il processo nipote del figlio di indice 1 e pid 20415, associato al file F4
Il figlio di indice 1 ha trovato il carattere a nella posizione 0 nel file F4
Il nipote con pid=20414 ha ritornato 2
STAMPA DI DEBUGGING-Processo 19486 ricevuto il segnale 10 e uscira' con 1
Il figlio con pid=20412 ha ritornato 2
Il nipote con pid=20415 ha ritornato 1
Il figlio con pid=20413 ha ritornato 1
```

Poiché nel file F4 ci s'è una sola occorrenza del carattere 'a' il figlio associato a tale file, termina il ciclo di lettura dal file e passerebbe ad aspettare il nipote, ma prima di farlo manda il segnale SIGUSR1 al nipote per sbloccarlo; infatti, il nipote, dato che trova una occorrenza in più del carattere 'e' si mette in attesa di ricevere la struct dal figlio, ma viene sbloccato dalla read appunto perché gli attiva il segnale, quindi esegue l'handler ed esce con la exit(occ) che riporta un numero di occorrenze non corrette, ma accettabile per non complicare troppo la soluzione.

Sempre sul sito si trova la soluzione (senza il problema sopra evidenziato) del testo b alla URL
http://didattica.agentgroup.unimore.it/didattica/SOeLab/SoluzioniCompiti/8Giu11/8Giu11-b.c

Nella prossima lezione (che sarà in streaming registrata) si analizzerà la soluzione dei seguenti due compiti (solo parte C):
- esame del 17 Luglio 2017;
- esame del 12 Febbraio 2016.

(fine seconda video-registrazione di giovedì 14/05/2020)

Lezione Dodicesimo Mercoledì di lezione (2 ore) ➔ in streaming Mer. 20/05/2020 (oltre che registrata)

Leggiamo il testo della parte C dell'esame del 12 Luglio 2017:

La parte in C accetta un numero variabile pari **2N** di parametri maggiore o uguale a 2 (*da controllare*) che rappresentano **N** nomi assoluti di file **F1**, ... **FN** intervallati da numeri interi strettamente positivi **X1, X2**, ... **XN** che rappresentano la lunghezza in linee dei file (si può supporre che i parametri di posizione pari siano numeri e si deve *solo controllare che siano strettamente positivi*). Il processo padre deve generare **N** processi figli: i processi figli **Pi** sono associati agli **N** file **Fh** e al numero **Xh** (con h= i+1). Ognuno di tali figli deve creare a sua volta un processo nipote **PPi**: ogni processo nipote **PPi** esegue concorrentemente e deve, per prima cosa, inizializzare il seme per la generazione random di numeri (come illustrato nel retro del foglio), quindi deve, usando in modo opportuno la funzione mia_random() (riportata sul retro del foglio), individuare un intero che rappresenterà il numero di linee del file **Fh** da selezionare, a partire dall'inizio del file, e inviare al figlio usando in modo opportuno il comando ***head*** di UNIX/Linux**.**

Ogni processo figlio **Pi** deve ricevere tutte le linee inviate dal suo processo nipote **PPi (ogni linea si può supporre che abbia una lunghezza massima di 250 caratteri, compreso il terminatore di linea e, se serve, il terminatore di stringa)** e, per ogni linea ricevuta, deve inviare al processo padre una **struttura** dati, che deve contenere tre campi: 1) *c1*, di tipo *int*, che deve contenere il pid del nipote; *c2*, di tipo *int*, che deve contenere il numero della linea; 3) c3, di tipo char[250], che deve contenere la linea corrente ricevuta dal nipote.

Il padre deve ricevere le strutture inviate dai figli nel seguente ordine: prima deve ricevere dal figlio **P0** la prima struttura inviata, poi deve ricevere dal figlio **P1** la prima struttura inviata e così via fino a che deve ricevere dal figlio **PN-1** la prima struttura inviata; quindi deve procedere a ricevere le seconde strutture inviate dai figli (se esistono) e così via. La ricezione di strutture da parte del padre deve terminare quando ha ricevuto tutte le strutture inviate da tutti i figli **Pi**. Il padre deve stampare su standard output, per ogni struttura ricevuta, ognuno dei campi.

Al termine, ogni processo figlio **Pi** deve ritornare al padre il valore random calcolato dal proprio processo nipote **PPi** e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

Chiamata alla funzione di libreria per inizializzare il seme:

```
#include <time.h>

srand(time(NULL));
```

Funzione che calcola un numero random compreso fra 1 e n:

```
#include <stdlib.h>

int mia_random(int n)
{
int casuale;
        casuale = rand() % n;
        casuale++;
        return casuale;
}
```

Illustrato in dettaglio la specifica e quindi come identificare i parametri file e parametri numeri interi, come controllare che i numeri interi siano strettamente positivi, che cosa devono eseguire i nipoti, cosa devono eseguire i figli e cosa deve eseguire il padre, e i vari schemi di comunicazione: si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/12Lug2017.mp4

(fine prima video-registrazione di mercoledì 20/05/2020)

La soluzione si trova sul sito alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/12Lug17/12Lug17.c
Vediamone il funzionamento:

```
soELab@lica04:~/12Lug17$ 12Lug17 /home/soELab/12Lug17/ger/f1 8 /home/soELab/12Lug17/ger/f4 6
/home/soELab/12Lug17/ger/d1/prova3.txt 10 /home/soELab/12Lug17/ger/d2/d3/prova1.txt 7
/home/soELab/12Lug17/ger1/d21/d3/prova4.txt 11 <== N.B. questi file sono quelli che trova la parte Shell!
Il nipote con pid 21447 ha letto dal file /home/soELab/12Lug17/ger/f1 nella riga 1 questa linea:
sono la prima linea di f1
Il nipote con pid 21453 ha letto dal file /home/soELab/12Lug17/ger/f4 nella riga 1 questa linea:
sono la prima linea di f4
Il nipote con pid 21450 ha letto dal file /home/soELab/12Lug17/ger/d1/prova3.txt nella riga 1 questa linea:
Sono la prima linea di prova3.txt
Il nipote con pid 21454 ha letto dal file /home/soELab/12Lug17/ger/d2/d3/prova1.txt nella riga 1 questa linea:
Sono la prima linea di prova1.txt
Il nipote con pid 21452 ha letto dal file /home/soELab/12Lug17/ger1/d21/d3/prova4.txt nella riga 1 questa linea:
Sono la prima linea di prova4.txt in ger1
Il nipote con pid 21447 ha letto dal file /home/soELab/12Lug17/ger/f1 nella riga 2 questa linea:
Sono la seconda linea di f1
Il nipote con pid 21450 ha letto dal file /home/soELab/12Lug17/ger/d1/prova3.txt nella riga 2 questa linea:
Sono la seconda linea di prova3.txt
…
Il figlio con pid=21445 ha ritornato 7 (se 255 problemi nel figlio o nel nipote)
Il figlio con pid=21446 ha ritornato 1 (se 255 problemi nel figlio o nel nipote)
Il figlio con pid=21448 ha ritornato 9 (se 255 problemi nel figlio o nel nipote)
Il figlio con pid=21449 ha ritornato 2 (se 255 problemi nel figlio o nel nipote)
Il figlio con pid=21451 ha ritornato 4 (se 255 problemi nel figlio o nel nipote)
```

Leggiamo ora il testo della parte C dell'esame del 12 Febbraio 2016:
La parte in C accetta un numero variabile **N+1** di parametri (con **N** maggiore o uguale a **2**, da controllare**)** che rappresentano **N** nomi di file (**F1, F2. ... FN**), mentre l'ultimo rappresenta un carattere alfabetico minuscolo (**Cx**) (da controllare).

Il processo padre deve generare **N processi figli** (**P0, P1, … PN-1**): i processi figli **Pi (con i che varia da 0 a N-1)** sono associati agli **N** file **Fj** (con j= i+1). Ogni processo figlio **Pi** deve leggere i caratteri del file associato **Fj** cercando il carattere **Cx**. I processi figli e il processo padre devono attenersi a questo **schema di comunicazione a pipeline**: il figlio **P0** comunica con il figlio **P1** che comunica con il figlio **P2** etc. fino al figlio **PN-1** che comunica con il **padre**. Questo schema a pipeline deve prevedere l'invio in avanti di un array di **strutture** dati ognuna delle quali deve contenere due campi: 1) *c1*, di tipo int, che deve contenere l'indice d'ordine dei processi; 2) *c2*, di tipo long int, che deve contenere il numero di occorrenze del carattere Cx calcolate dal corrispondente processo. *Gli array di strutture DEVONO essere creati da ogni figlio della dimensione minima necessaria per la comunicazione sia in ricezione che in spedizione.* Quindi la comunicazione deve avvenire in particolare in questo modo: il figlio **P0** passa in avanti (cioè comunica) un array di strutture **A1**, che contiene una sola struttura con *c1* uguale a 0 e con *c2* uguale al numero di occorrenze del carattere **Cx** trovate da **P0** nel file **F1**; il figlio seguente **P1**, dopo aver calcolato numero di occorrenze del carattere **Cx** nel file **F2**, deve leggere (*con una singola read*) l'array **A1** inviato da **P0** e quindi deve confezionare l'array **A2** che corrisponde all'array **A1** aggiungendo all'ultimo posto la struttura con i propri dati e la passa (*con una singola write*) al figlio seguente **P2**, etc. fino al figlio **PN-1**, che si comporta in modo analogo, ma passa al **padre**. Quindi, al processo padre deve arrivare l'array **AN** di **N** strutture (uno per ogni processo **P0** … **PN-1)**. Il padre deve riportare i dati di ognuna delle **N** strutture su standard output insieme al **pid** del processo corrispondente e al carattere **Cx**.
Al termine, ogni processo figlio **Pi** deve ritornare al padre il valore intero corrispondente al proprio indice d'ordine (**i**); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

Illustrato in dettaglio la specifica mettendo in evidenza che la soluzione è molto simile a quella della prova del 26 Maggio 2017 dove però al padre arriva una singola struttura (con 3 campi), mentre nella prova del 12 Febbraio 2016 al padre arriva un array di strutture (ognuna con 2 campi) e che da un figlio all'altro la dimensione di questo array cresce: da P0 a P1 l'array ha dimensione 1, da P1 a P2 ha dimensione 2 e così via. In entrambe le soluzioni, è necessario che il padre salvi in un array creato dinamicamente i pid dei figli.
Si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/12Feb2016.mp4
Analizzato lo schema di comunicazione a pipeline, in particolare il numero di pipe che servono (N, e cioè quanti i processi figli) e il fatto che nelle soluzioni dei due esercizi ogni figlio Pi legge dalla pipe i-1 e scrive sulla pipe i. Analizzato poi le differenze,

Esaurito il tempo prima di poter far vedere la soluzione che comunque si trova sul sito alla URL
http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/12Feb16/12Feb16.c

Nella prossima lezione (che sarà in streaming registrata) si analizzerà la soluzione dei seguenti due compiti (solo parte C):
- seconda prova in itinere del 31 Maggio 2019;
- esame del 11 Febbraio 2011.

(fine seconda video-registrazione di mercoledì 20/05/2020)

Lezione Tredicesimo Lunedì di lezione (2 ore) ➔ in streaming Lun. 25/05/2020 (oltre che registrata)

Rivisto rapidamente le specifiche del testo dell'ultimo esercizio presentato nella lezione scorsa (quello del 12 Febbraio 2016): mostrato nel dettaglio il codice che si ricorda si trova sul sito alla URL
http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/12Feb16/12Feb16.c

Vediamone il funzionamento:

```
soELab@lica04:~/12Feb16$     12Feb16     /home/soELab/12Feb16/ger/a     /home/soELab/12Feb16/ger/d1/f1
/home/soELab/12Feb16/ger/d1/f2      /home/soELab/12Feb16/ger/d1/f3      /home/soELab/12Feb16/ger/d1/f4
/home/soELab/12Feb16/ger/d2/dd2/F1 /home/soELab/12Feb16/ger/d2/dd2/F2 a
Carattere da cercare a
Numero di processi da creare 7
Sono il figlio 25041
Sono il figlio 25043
Sono il figlio 25047
Sono il figlio 25044
Sono il figlio 25045
Sono il figlio 25046
Sono il figlio 25042
Padre ha letto un numero di strutture 7
Il figlio di indice 0 e pid 25041 ha trovato 1 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/a
Il figlio di indice 1 e pid 25042 ha trovato 1 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/d1/f1
Il figlio di indice 2 e pid 25043 ha trovato 4 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/d1/f2
Il figlio di indice 3 e pid 25044 ha trovato 5 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/d1/f3
Il figlio di indice 4 e pid 25045 ha trovato 2 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/d1/f4
Il figlio di indice 5 e pid 25046 ha trovato 1 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/d2/dd2/F1
Il figlio di indice 6 e pid 25047 ha trovato 4 occorrenze del carattere a nel file
/home/soELab/12Feb16/ger/d2/dd2/F2
```

```
Il figlio con pid=25041 ha ritornato 0 (se > di 6 problemi)
Il figlio con pid=25042 ha ritornato 1 (se > di 6 problemi)
Il figlio con pid=25043 ha ritornato 2 (se > di 6 problemi)
Il figlio con pid=25044 ha ritornato 3 (se > di 6 problemi)
Il figlio con pid=25045 ha ritornato 4 (se > di 6 problemi)
Il figlio con pid=25046 ha ritornato 5 (se > di 6 problemi)
Il figlio con pid=25047 ha ritornato 6 (se > di 6 problemi)
```
← ATTENZIONE il fatto che siano riportati in ordine è assolutamente casuale!

Verichichiamo il contenuto di alcuni file:

```
soELab@lica04:~/12Feb16$ more /home/soELab/12Feb16/ger/d2/dd2/F2
sono il file
f2 e Servo come
prova
ho altri caratteri
```
← 4 occorrenze in totale

```
soELab@lica04:~/12Feb16$ more /home/soELab/12Feb16/ger/a
Sono il file
di nome a
```
← 1 sola occorrenza in totale

Verifichiamo anche alcune invocazioni scorrette:

```
soELab@lica04:~/12Feb16$ 12Feb16 /home/soELab/12Feb16/ger/a /home/soELab/12Feb16/ger/d1/f1
/home/soELab/12Feb16/ger/d1/f2 /home/soELab/12Feb16/ger/d1/f3 /home/soELab/12Feb16/ger/d1/f4
/home/soELab/12Feb16/ger/d2/dd2/F1 /home/soELab/12Feb16/ger/d2/dd2/F2 abc
Errore ultimo parametro non singolo carattere
soELab@lica04:~/12Feb16$ 12Feb16 /home/soELab/12Feb16/ger/a /home/soELab/12Feb16/ger/d1/f1
/home/soELab/12Feb16/ger/d1/f2 /home/soELab/12Feb16/ger/d1/f3 /home/soELab/12Feb16/ger/d1/f4
/home/soELab/12Feb16/ger/d2/dd2/F1 /home/soELab/12Feb16/ger/d2/dd2/F2 A
Errore ultimo parametro non alfabetico minuscolo
soELab@lica04:~/12Feb16$ 12Feb16
Errore numero di parametri
soELab@lica04:~/12Feb16$ 12Feb16 F1 a
Errore numero di parametri
```

Leggiamo ora il testo della parte C dell'esame del 11 Febbraio 2011, come previsto nella lezione scorsa:
La parte in C accetta un numero variabile di parametri N+1: il primo parametro rappresenta un numero intero positivo dispari (da verificare) H, mentre gli altri N parametri rappresentano nomi assoluti di file F0...FN-1: si può ipotizzare che la lunghezza di tutti i file sia uguale, pari e multiplo intero H (senza verificarlo). Il processo padre deve generare 2N processi figli (P0 … P2N-1); tali processi figli costituiscono N coppie di processi: ogni coppia Ci è composta dal processo Pi (primo processo della coppia) e dal processo Pi+N (secondo processo della coppia), con i variabile da 0 a N-1. Ogni coppia di processi figli Ci è associata ad uno dei file Fi*. Il primo processo della coppia deve creare un file il cui nome (FCreato) risulti dalla concatenazione del nome del file associato alla coppia con la stringa ".mescolato" (ad esempio se Fi è /tmp/pippo.txt il file FCreato si deve chiamare /tmp/pippo.txt.mescolato). Tutte le coppie di processi figli eseguono concorrentemente leggendo il proprio file associato: in particolare, il primo processo della coppia deve leggere la prima metà del file associato, mentre il secondo processo la seconda metà del file; inoltre, per entrambi i processi della coppia la lettura deve avvenire a blocchi di dati di grandezza uguale a H byte. Il primo processo di ogni coppia, dopo la lettura di ogni blocco di dati B1 della sua prima metà del file, lo scrive (con un'unica write!) sul file FCreato; quindi deve ricevere (con un'unica read!) dal secondo processo della coppia il suo corrispondente blocco di dati B2 e quindi deve scriverlo (sempre con un'unica write!) sul file FCreato; viceversa, il secondo processo di ogni coppia, dopo la lettura di ogni blocco di dati B2 della sua seconda metà del file, lo comunica (con un'unica write!) al primo processo della coppia. Al termine, ogni processo di ogni coppia deve ritornare al padre il numero di blocchi letti dalla propria metà del file. Il padre, dopo che i figli sono terminati, deve stampare su standard output i PID di ogni figlio con il corrispondente valore ritornato.
*** Se N è 3 (i varia da 0 a 2), le coppie di processi e i file associati sono P0-P3 con F0, P1-P4 con F1 e P2-P5 con F2.**

Illustrato in dettaglio la specifica, in particolare, la comunicazione fra le coppie di processi figli: si veda anche il video caricato qui **http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/11Feb2011.mp4**

(fine prima video-registrazione di mercoledì 20/05/2020)

Mostrato quindi nel dettaglio il codice che si ricorda si trova sul sito alla URL
http://www.didattica.agentgroup.unimo.it/didattica/soNOD/Esercizi/c/11Feb11.c

Vediamone il funzionamento, considerando i seguenti due file di prova:

```
soELab@lica04:~/11Feb11$ more F1 F2
::::::::::::::
F1
::::::::::::::
abcdefghilmnopqrstuvza01234567890xxxxxxxxxx
```

← l'undicesimo carattere del secondo blocco della seconda metà è lo '\n'

```
::::::::::::::
F2
::::::::::::::
01234567891234567890123456789012YYYYYYYYYYwwwwwwwwww
```
← l'undicesimo carattere del secondo blocco della seconda metà è lo '\n'

Vediamo prima due invocazioni scorrette:

```
soELab@lica04:~/11Feb11$ 11Feb11 11 F1
Errore numero parametri 3
soELab@lica04:~/11Feb11$ 11Feb11 -11 F1 F2
Errore numero H -11
```

Ora quella giusta:

```
soELab@lica04:~/11Feb11$ 11Feb11 11 F1 F2
Numero processi da creare 4 con H=11
PRIMO DELLA COPPIA-Figlio di indice 0 e pid 25993 associato al file F1
SECONDO DELLA COPPIA-Figlio di indice 3 e pid 25996 associato al file F2
SECONDO DELLA COPPIA-Figlio di indice 2 e pid 25995 associato al file F1
Il figlio con pid=25996 ha ritornato 2 (se 0 problemi)
PRIMO DELLA COPPIA-Figlio di indice 1 e pid 25994 associato al file F2
Il figlio con pid=25994 ha ritornato 2 (se 0 problemi)
Il figlio con pid=25993 ha ritornato 2 (se 0 problemi)
Il figlio con pid=25995 ha ritornato 2 (se 0 problemi)
```

Vediamo il contenuto dei due file creati:

```
soELab@lica04:~/11Feb11$ more F1.mescolato F2.mescolato
::::::::::::::
F1.mescolato
::::::::::::::
abcdefghilm01234567890nopqrstuvzaXXXXXXXXXX
```
← l'undicesimo carattere dell'ultimo blocco è lo '\n'

```
::::::::::::::
F2.mescolato
::::::::::::::
01234567891YYYYYYYYYY23456789012wwwwwwwwww
```
← l'undicesimo carattere dell'ultimo blocco è lo '\n'

Leggiamo ora il testo della parte C della seconda prova in itinere di Venerdì scorso, 31 Maggio 2019, come previsto nella lezione scorsa: sul sito sono state caricate 4 versioni che differiscono in alcune cose. Si precisa che la soluzione è molto simile ad un esercizio già svolto.
Versione 1 dei turni 1 e 2:

La parte in C accetta un numero variabile **N** di parametri maggiore o uguale a 3 (*da controllare*) che rappresentano nomi assoluti di file **F1**, ... **FN**. Il processo padre deve generare **N** processi figli: i processi figli **Pi** sono associati agli **N** file **Fh** (con h= i+1). Ognuno di tali figli deve creare a sua volta un processo nipote **PPi**: ogni processo nipote **PPi** esegue concorrentemente e deve ordinare il file **Fh** secondo il normale ordine alfabetico, senza differenziare maiuscole e minuscole, usando in modo opportuno il comando ***sort*** di UNIX/Linux**.**
Ogni processo figlio **Pi** deve ricevere solo la **PRIMA** linea inviata dal suo processo nipote **PPi** e deve inviare al processo padre una **struttura** dati, che deve contenere tre campi: 1) *c1*, di tipo *int*, che deve contenere il pid del nipote; *c2*, di tipo *int*, che deve contenere la lunghezza della linea compreso il terminatore di linea; 3) *c3*, di tipo char[250] [*], che deve contenere la linea corrente ricevuta dal nipote.
Il padre deve ricevere, rispettando l'ordine dei file, le singole strutture inviate dai figli e deve stampare su standard output, per ogni struttura ricevuta, ognuno dei campi insieme al nome del file cui le informazioni si riferiscono: **si faccia attenzione al fatto che è demandato al padre il compito di trasformare, in una stringa, la linea ricevuta nel campo c3 di ogni struttura!**
Al termine, ogni processo figlio **Pi** deve ritornare al padre la lunghezza della linea inviata al padre, ma non compreso il terminatore di linea e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

[*] **Ogni linea si può supporre che abbia una lunghezza massima di 250 caratteri, compreso il terminatore di linea e, se serve, il terminatore di stringa.**

Illustrato in dettaglio i parametri e i vari controlli che si devono fare. Poi passato ad illustrare i vari passi che devono essere svolti dal padre, dai figli e dai nipoti. In particolare, illustrato i vari schemi di comunicazione: si parte con una serie di N pipe orientate da ogni figlio al padre. Sulle pipe figli-padre deve essere inviata una singola struttura per ogni figlio. Poi c'è una pipe fra ogni nipote e ogni figlio orientata dal nipote al figlio: il nipote scrive sulla pipe il risultato della sort –f (come il testo riportato, oppure del sort –r come il testo della versione 2 dei turni 1 e 2) per effetto della close(1) e della dup(p[1]). ATTENZIONE CHE IL NIPOTE IN CASO DI FALLIMENTO DELLA EXEC NON POTEVA USARE LA printf, MA EVENTUALMENTE LA perror (SEMPRE CHE NON SI FOSSE RIDIRETTO LO STANDARD ERROR SU /dev/null!).
Il figlio di tutte le linee inviate dal nipote DEVE considerare solo la PRIMA linea; al padre deve essere mandata una struttura che ha nel campo c1 il pid del nipote, nel campo c2 la lunghezza della linea compreso il terminatore e nel campo c3 la PRIMA linea.
ATTENZIONE CHE INDIPENDENTEMENTE SE IL CONTEGGIO DELLA LUNGHEZZA DELLA LINEA CONTASSE O MENO IL TERMINATORE NELL'ARRAY DI CARATTERI DOVEVA SEMPRE ESSERE PRESENTE ANCHE IL TERMINATORE DI LINEA (CIOÈ LO '\n') ALTRIMENTI NON SAREBBE STATA UNA LINEA!
Il figlio doveva aspettare sempre il nipote (facendo i soliti controlli e stampando o meno il valore di ritorno) per evitare la ricezione del segnale SIGPIPE e poi doveva tornare al padre la lunghezza della linea NON compreso il terminatore.
Il padre ricevuta ogni singola struttura dai figli doveva andare ad inserire il terminatore di stringa (cioè lo '\0') per poter usare la printf con %s senza problemi: ATTENZIONE CHE NON SI DOVEVA SOVRASCRIVERE IL TERMINATORE DI LINEA!
Si veda anche il video caricato qui http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/31Mag2019.mp4

Quindi fatta vedere velocemente la soluzione completa sul sistema lica04 della versione 1 dei turni 1 e 2 che si trova sul sito alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/31Mag19/31Mag19-1.c

Verifichiamone il funzionamento:

```
soELab@lica04:~/31Mag19$ 31Mag19-1 F1 F2 F3
Il nipote con pid 7545 ha letto dal file F1 questa linea 'amorevole
' che ha lunghezza (compreso il terminatore) di 10 caratteri:
Il nipote con pid 7546 ha letto dal file F2 questa linea 'ancora
' che ha lunghezza (compreso il terminatore) di 7 caratteri:
Il nipote con pid 7547 ha letto dal file F3 questa linea 'Bacioni
' che ha lunghezza (compreso il terminatore) di 8 caratteri:
Il figlio con pid=7542 ha ritornato 9 (se 255 problemi nel figlio o nel nipote)
Il figlio con pid=7543 ha ritornato 6 (se 255 problemi nel figlio o nel nipote)
Il figlio con pid=7544 ha ritornato 7 (se 255 problemi nel figlio o nel nipote)
```

Controlliamo il contenuto dei file:

```
soELab@lica04:~/31Mag19$ more F1 F2 F3
::::::::::::::
F1
::::::::::::::
zuzzurellone
Letizia
ombrellone
mare
bagnino
amorevole
costa
linea
Bacio
::::::::::::::
F2
::::::::::::::
ancora
Letizia
Anna
ombrellone
mare
bagnino
costa
```

amorevole ← **nell'ordinamento sort –f metterà questa linea come prima linea!**

ancora ← **nell'ordinamento sort –f metterà questa linea come prima linea!**

linea
Bacio
seta
::::::::::::::
F3
::::::::::::::
Letizia
ombrellone
mare
bagnino
costa
linea
bellissima
Bacioni **🡐 nell'ordinamento sort –f metterà questa linea come prima linea!**

(fine seconda video-registrazione di lunedì 25/05/2020)

Lezione Tredicesimo Mercoledì di lezione (2 ore) ➔ corrisponde a due registrazioni effettuate Mer. 27/05/2020

Leggiamo il testo di un compito completo, quello del 15 Gennaio 2020, chiaramente prima della parte Shell e poi della parte C.
Si realizzi un programma **concorrente** per UNIX che deve avere una parte in **Bourne Shell** e una parte in **C**

La parte in Shell deve prevedere un numero variabile di parametri **W+1** (con **W** maggiore o uguale a 2): il primo parametro deve essere considerato un intero strettamente positivo (**H**), mentre gli altri **W** devono essere **nomi assoluti di directory** che identificano **W** gerarchie (**G1, G2, …**) all'interno del file system. Il comportamento atteso dal programma, dopo il controllo dei parametri, è organizzato in **W** fasi, una per ogni gerarchia.
Il programma, per ognuna delle **W** fasi, deve esplorare la gerarchia **Gg** corrispondente - tramite un file comandi ricorsivo, **FCR.sh** - e deve contare **globalmente per ogni singola** gerarchia **Gg** tutti i file che saranno cercati secondo quanto di seguito specificato. Il file comandi ricorsivo **FCR.sh** deve cercare in ogni gerarchia **Gg** che esista almeno un file (**F)** la cui lunghezza in caratteri sia esattamente uguale a **H**: appena trovato un file che soddisfa la specifica, si deve riportare, contestualmente, il suo nome assoluto sullo standard output. Al termine di OGNUNA delle **W** fasi, si deve riportare sullo standard output il numero di file (**F)** trovati e, solo nel caso tale numero sia *pari* si deve invocare la parte in C, passando come parametri i nomi assoluti dei file ***trovati*** (**F1**, **F2**, ... **FN**).

La parte in C accetta un numero variabile **N** di parametri (con **N** maggiore o uguale a **2** e **pari**, da controllare**)** che rappresentano nomi di file **F1, F2. ... FN** (tutti con uguale lunghezza, che non deve essere controllata). Il processo padre deve generare **N/2** processi figli (P0 ... PN/2-1) e ognuno dei processi figli deve generare un *processo nipote* (PP0 ... PPN/2-1): i processi figli Pi sono associati ai file Fi+1 mentre i processi nipoti PPi ai file FN/2+i+1 (con *i* che, in entrambi i casi, varia da 0 a N/2). Ogni processo figlio Pi deve, *prima di creare il proprio nipote*, creare un file **FOut** il cui nome deve risultare dalla concatenazione della stringa "merge" e

della stringa corrispondente a i (numero d'ordine di creazione del processo figlio). Una volta creato il processo nipote, ogni figlio e ogni nipote eseguono concorrentemente; in particolare, ognuno dei due 'tipi' di processi deve leggere, dal suo file associato, un carattere alla volta e quindi lo deve scrivere sul file FOut: **la scrittura deve avvenire in modo strettamente alternato**, iniziando dal figlio Pi. In altre parole, ogni figlio Pi legge il primo carattere dal file Fi+1 e lo scrive sul file FOut e quindi deve comunicare l'avvenuta scrittura sul file al proprio nipote PPi, quindi il processo PPi che ha concorrentemente letto il primo carattere dal file FN/2+i+1 lo può scrivere sul file FOut e può comunicare al proprio figlio l'avvenuta scrittura; tale schema di comunicazione/sincronizzazione* deve continuare per tutti i caratteri dei due file associati. Al termine, ogni processo nipote PPi deve ritornare al figlio il valore dell'ultimo carattere scritto nel file FOut e, a sua volta, ogni processo figlio Pi lo deve ritornare al padre. Il padre, dopo che i figli sono terminati, deve stampare, su standard output, i PID di ogni figlio con il corrispondente valore ritornato.

* Se si vuole per la sincronizzazione si possono usare i segnali.

Quindi fatta vedere prima la soluzione della parte Shell sul sistema lica04 e che si trova sul sito alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/15Gen20/FCP.sh per il file comandi principale e alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/15Gen20/FCR.sh per il file comandi ricorsivo e poi la soluzione della parte C sul sistema lica04 e che si trova sul sito alla URL http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SoluzioniCompiti/15Gen20/15Gen20.c

Vediamo quindi il funzionamento complessivo sul sistema:

```
soELab@lica04:~/15Gen20$ FCP.sh 5 `pwd`/ger `pwd`/ger1
Numero parametri corretti 3
fase per /home/soELab/15Gen20/ger
Trovato file /home/soELab/15Gen20/ger/F1
Trovato file /home/soELab/15Gen20/ger/d1/F3
Trovato file /home/soELab/15Gen20/ger/d2/F4
Trovato file /home/soELab/15Gen20/ger/d2/d21/F2
Il numero globale di file trovati nelle gerarchia /home/soELab/15Gen20/ger che rispettano le specifiche e' 4
stiamo per invocare la parte C con /home/soELab/15Gen20/ger/F1 /home/soELab/15Gen20/ger/d1/F3 /home/soELab/15Gen20/ger/d2/F4 /home/soELab/15Gen20/ger/d2/d21/F2
Sono il processo figlio di indice 0 e pid 28287 e leggero' dal file /home/soELab/15Gen20/ger/F1
Sono il processo nipote del figlio di indice 0 e ho pid 28289 e leggero' dal file /home/soELab/15Gen20/ger/d2/F4
Sono il processo figlio di indice 1 e pid 28288 e leggero' dal file /home/soELab/15Gen20/ger/d1/F3
Sono il processo nipote del figlio di indice 1 e ho pid 28290 e leggero' dal file /home/soELab/15Gen20/ger/d2/d21/F2
Il nipote con pid=28289 ha ritornato 6
```

```
Il nipote con pid=28290 ha ritornato k
Il figlio con pid=28287 ha ritornato 6
Il figlio con pid=28288 ha ritornato k
fase per /home/soELab/15Gen20/ger1
Trovato file /home/soELab/15Gen20/ger1/d1/F3
Trovato file /home/soELab/15Gen20/ger1/d2/F4
Trovato file /home/soELab/15Gen20/ger1/d2/d21/F2
Il numero globale di file trovati nelle gerarchia /home/soELab/15Gen20/ger1 che rispettano le specifiche e' 3
3 non pari
```

Verifichiamo che il contenuto dei file letti dai nipoti sia tale che l'ultimo carattere sia '6' e 'k':

```
soELab@lica04:~/15Gen20$ more /home/soELab/15Gen20/ger/d2/F4
09876
soELab@lica04:~/15Gen20$ more /home/soELab/15Gen20/ger/d2/d21/F2
zyxwk
```

Verifichiamo anche il contenuto dei file letti dai figli e il contenuto dei file creati (dimenticata di farli vedere a lezione):

```
soELab@lica04:~/15Gen20$ more /home/soELab/15Gen20/ger/F1
abcde
soELab@lica04:~/15Gen20$ more /home/soELab/15Gen20/ger/d1/F3
12345
soELab@lica04:~/15Gen20$ more merge0
a0b9c8d7e6
soELab@lica04:~/15Gen20$ more merge1
1z2y3x4w5k
```

← deriva da /home/soELab/15Gen20/ger/F1 e /home/soELab/15Gen20/ger/d2/F4

← deriva da /home/soELab/15Gen20/ger/d1/F3 e /home/soELab/15Gen20/ger/d2/d21/F2

(fine prima video-registrazione di mercoledì 27/05/2020)

Mostrato schemi sia per i testi della parte Shell che per la comunicazione della parte C: si vedano i due video caricati qui:

- **http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/SchemiShell.mp4**
- **http://www.didattica.agentgroup.unimo.it/didattica/SOeLab/SessioniInterattive/SchemiDiComunicazione.mp4**

(fine seconda video-registrazione di mercoledì 27/05/2020) ➔ FINE DELLE LEZIONI!