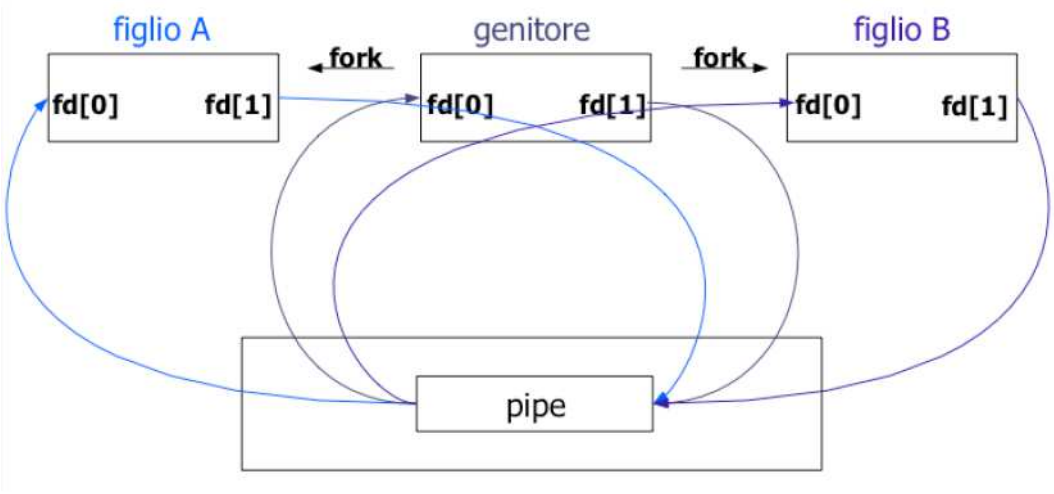
Pipe tra due Programmi

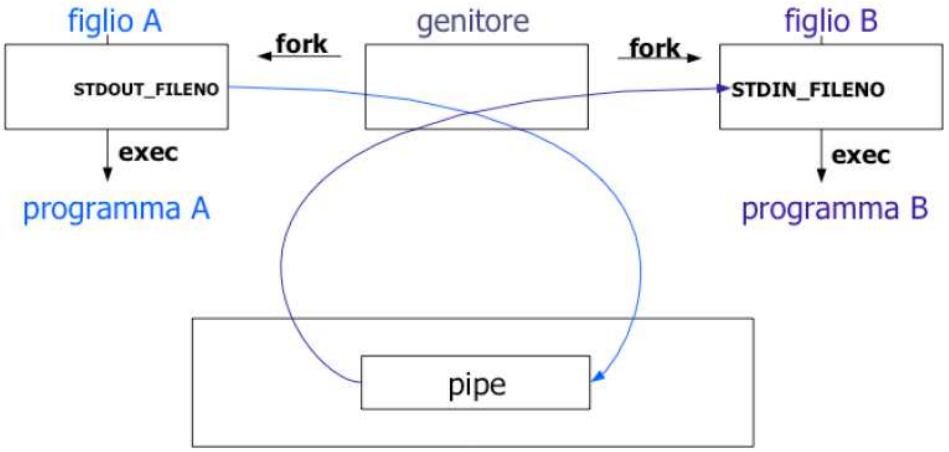
Vediamo ora di emulare una pipeline ossia il comportamento di una singola pipe "|" sulla bash quindi creiamo un programma pipeline a cui passando la stringa “ls -x- wc” da lo stesso input del comando “ls | wc” sostituendo "|" con "-x-"

Il programma pipeline agirà così:

- il padre genera due figli ed una pipe che i figli ereditano



- I due figli sostituiscono rispettivamente il proprio stdout e stdin con i relativi ingressi della pipe.
- Successivamente cambiano i rispettivi processi in esecuzione di modo che la pipe venga eseguita.



- Il padre si occupa delle terminazione dei figli.

Per ridirezionare un file un C si utilizza la funzione dup e dup2

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

La funzioni dup() e la funzione dup2() creano una copia del file descriptor oldfd, dup() attribuisce al nuovo file descriptor, il più piccolo intero non usato, dup2() crea newfd come copia di oldfd, chiudendo prima newfd se è necessario.

Il vecchio e nuovo file descriptor possono essere utilizzati interscambiabilmente, essi condividono locks, puntatori di file position e flag, ad eccezione del flag close-on-exec.

Per esempio è possibile effettuare una lseek() su un file descriptor e ritrovarsi la posizione modificata su entrambi i file descriptor.

Le funzioni dup() e dup2() ritornano il nuovo file descriptor in caso di sucesso, oppure -1 in caso di errore.

Non è necessario fornire il percorso assoluto dei programmi da eseguire poiché utilizziamo execvp.

Vediamo il codice pipeline.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

char **make_args(char *argv[],int l, int r);

// data la struttura argv (vettore di puntatori a stringhe, terminante
// con NULL, la funzione make_args genera una nuova struttura argv che
//contiene un intervallo dell'argv originale, dove gli estremi dell'intervallo
//sono dati input l,r
char **make_args(char *argv[],int l, int r)
{char **new_argv;
 int i, j;
 // alloca lo spazio necessario tenendo pure conto del NULL finale
 new_argv=(char **)malloc(sizeof(char *)*(r-l));
 // copia il puntatore della stringa dall'argv originario a quello
 // nuovo (notare che non viene generata una nuova stringa per ogni
 // puntatore
 for (i=l,j=0;i<r;new_argv[j++]=argv[i++]);
 new_argv[j]=NULL; // inserisce il NULL finale
 return new_argv;
}
```

```c
int main(int argc, char *argv[])
{int tube[2];
 int i;
 int k=0;
 pid_t ok_fork[2]={-1,-1};
 char **new_argv;
 int w;
 for (i=1;i<argc;i++)// per ogni parametro della line di comando
  // se corrisponde alla stringa di pipe "-x-"
  // conserva la posizione se era la prima volta, altrimenti
  // segna l'errore (si emula solo una singola pipe)
  if (!strcmp(argv[i],"-x-")) k=(!k)?i:-1;
  // esci se non hai trovato piu' di un simbolo di pipe "-x-"
  if (k<0)
   {printf("Error!!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
  // esci se non ci sono simboli di pipe
  if (!k)
   {printf("Error!!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
  if (pipe(tube)) // crea la pipe {
   {printf("Error!!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
  if ((ok_fork[0]=fork())<0) // genera un figlio
   {printf("Error!!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
  if (ok_fork[0])   // se non sono il figlio genera il secondo figlio
   if ((ok_fork[1]=fork())<0)
   {printf("Error!!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
  // se sono il primo figlio
  if (!(ok_fork[0]))
   {new_argv=make_args(argv,1,k);// genera il nuovo argv per il 1 comando
```

```
    // rimpiazza lo stdout con il ripsettivo ingresso della pipe
    dup2(tube[1],STDOUT_FILENO);
    // chiudi i descrittori superflui
    close(tube[0]);close(tube[1]);
    // sostituisci il processo corrente con quello fornito dal nuovo argv
    execvp(new_argv[0],new_argv);
    }
   // se sono il secondo figlio
   if (!(ok_fork[1]))
    {new_argv=make_args(argv,k+1,argc);//genera il nuovo argv per il 2 comando
     // rimpiazza lo stdout con il rispettivo ingresso della pipe
     dup2(tube[0],STDIN_FILENO);
     // chiudi i descrittori superflui
     close(tube[0]);close(tube[1]);
     // sostituisci il processo corrente con quello fornito dal nuovo argv
     execvp(new_argv[0],new_argv);
    }
   // se sono il padre
   if (ok_fork[0] && ok_fork[1])
    {close(tube[0]);close(tube[1]);// chiude i descrittori superflui
    // attendi che i figli abbiano terminato
    for (i=0;i<2;)
     {wait(&w);
      if (WIFEXITED(w))i++;
     }
    printf("All done\n");
   }
 exit(EXIT_SUCCESS);}
```

I Segnali

I Segnali costituiscono una forma di comunicazione primitiva tra processi permettendo una comunicazione asincrona tra il processo che invia il segale e quelli che lo ricevono, quindi:

- Processo che invia il segnale:
  - invia un segnale e prosegue il suo avanzamento indisturbato
- Processo che riceve il segnale:
  - il processo cattura il segnale
  - interrompe il normale avanzamento, gestisce il segnale eseguendo opportune funzioni dedicate
  - ritorna al punto in cui era stato interrotto

I segnali vengono lanciati attraverso la chiamata della funzione kill che ha la seguente sintassi

```
int kill(pid_t pid, int sig);
```

pid indica il numero del processo al quale si vuole inviare un determinato segnale, e sig il segnale che si vuole inviare.

pid può anche essere:

- 0 per tutti i processi del gruppo di quello corrente
- -1 per tutti i processi tranne il primo nella tavola dei processi
- <-1 il segnale è spedito a tutti i processi del gruppo -pid

sig il segnale che si vuole inviare è può essere uno di questi:

Nome segnale	Valore	Azioni	Commenti
SIGHUP	1	A	Hangup detected
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	A	Quit from keyboard
SIGILL	4	A	Illegal Instruction
SIGTRAP	5	CG	Trace/breakpoint trap
SIGABRT	6	C	Abort
SIGUNUSED	7	AG	Unused signal
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Terminal signal
SIGUSR1	10	A	User defined signal 1
SIGSEGV	11	C	Invalid memory reference
SIGUSR2	12	A	User defined signal 2
SIGPIPE	13	A	Write to pipe with no readers
SIGALRM	14	A	Timer signal from alarm(1)
SIGTERM	15	A	Termination signal
SIGSTKFLT	16	AG	Stack fault on coprocessor
SIGCHILD	17	B	Child terminated
SIGCONT	18		Continue if stopped
SIGSTOP	19	DEF	Stop process
SIFTSTP	20	D	Stop typed at tty
SIGTTIN	21	D	tty input for background pr.
SIGTTOU	22	D	tty output for background p
SIGIO	23	AG	I/O error
SIGXCPU	24	AG	CPU time limit exceeded
SIGXFSZ	25	AG	File size limit exceeded
SIGVTALRM	26	AG	Virtual time alarm
SIGPROF	27	AG	Profile signal
SIGWINCH	29	BG	Window resize signal

Per saperne di più sui segnali consulta la seguente pagina web

http://docsrv.sco.com:507/en/man/html.M/signal.M.html

Esempio di kill kill.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(int argc, char *argv[])
 { int pid;
  if ((pid = fork()) == 0)
    while (1); // il figlio attende per sempre
  else
   {sleep(10);// il padre attende 10 secondi
    // … e quindi invia il segnale di terminazione
    kill(pid, SIGINT);
   }
}
```

L’intercettazione di un segnale utente avviene attraverso una funzione definita dal programmatore attraverso La funzione signal() che stabilisce a quale funzione il controllo deve passare quando il segnale è catturato.

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

La funzione signal() stabilisce che quando il processo corrente riceve il segnale signum, il controllo deve passare come è specificato dal signal handler sighandler.

Per specificare il segnale signum può essere impiegato uno dei valori costanti definiti nell'header standard (o in un altro header da questo richiamato) oppure un valore numerico.

sighandler puoò essere o una funzione utente oppure uno dei valori SIG_IGN o SIG_DFL.

Quando il processo riceve il segnale signum, esso può comportarsi in 3 modi differenti a seconda che sighandler sia stato impostato come:

- SIG_IGN Il segnale viene ignorato.
- SIG_DFLViene eseguita l'azione di default prevista per quel segnale.
- Funzione utente Viene chiamata la funzione utente, avente come argomento il numero del segnale (signum) che ne ha provocato la chiamata.

N.B. - I segnali SIGKILL e SIGSTOP non possono essere né ignorati né catturati.

In caso di successo la funzione ritorna il precedente valore della funzione signal handler , SIG_ERR in caso di errore.

Il prototype della funzione signal può essere semplificato nella seguente forma:

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

sighandler_t signal(int signum, sighandler_t handler);
```

Ora appare più leggibile quali siano il tipo degli argomenti passati alla funzione e quale sia il tipo tornato da signal().

La funzione accetta due argomenti:

- uno di tipo int
- uno di tipo puntatore a funzione (la quale a sua volta accetta un argomento intero e torna void).

Il tipo ritornato da signal è un puntatore a funzione che accetta un argomento intero e torna void.

Esempio 1 es1_signal.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void int_handler(int sig);

void int_handler(int sig)
{printf("Ricevuto!\n");
 exit(2);
}
int main(int argc, char *argv[])
{int pid;
 signal(SIGINT, int_handler);
 if ((pid = fork()) < 0)
  {perror("fork fallita\n");exit(-1); }
 if(pid== 0)
  {while(1)
    { sleep(1); printf("In esecuzione!!!\n");}
  }
 sleep(3);
 kill(pid, SIGINT);exit(0);
}
```

Esempio 2 es2_signal.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int count = 0;
void gotCntrlBSlash();
void gotINT();
int main (int argc, char* argv[])
 {int i;
  signal(SIGTSTP,SIG_IGN); // ignora ^Z
  signal(SIGQUIT,gotCntrlBSlash);// gestore per '^\'
  signal(SIGINT,gotINT); // gestore per ^C
  for(i=1;i<100;i++) {sleep(1);printf((i%10==0)?".\n":".");}
  return 0;}
void gotINT()
 {switch(++count)
   {case 1:printf("Ricevuto Primo SIGINT\n");
           signal(SIGINT,gotINT);break;
    case 2:printf("Ricevuto Secondo SIGINT\n");
           signal(SIGINT,gotINT);break;
    case 3:printf("Ricevuto Terzo SIGINT\n");
           signal(SIGINT,SIG_DFL);break;}}
void gotCntrlBSlash()
  {signal(SIGQUIT,gotCntrlBSlash);
    printf("Ricevuto un SIGQUIT\n");}
```

Raise

L'altra funzione da considerare è raise(), con la quale si attiva volontariamente un segnale, dal quale poi dovrebbero o potrebbero sortire delle conseguenze, come stabilito in una fase precedente attraverso signal(). La funzione raise() è molto semplice:

```
int raise (int sig);
```

La funzione ha come argomento il segnale da attivare e restituisce:

- 0 in caso di successo,
- !=0 altrimenti.

Vediamo un esempio di raise, con un programma che all'invio dei segnali relativi mostra un messaggio sullo schermo.

Quindi dopo l'esecuzione del programma, viene inviato il segnale SIGUSR1 con il comando kill -s SIGUSR1 <pid>.

Se si inviano altri segnali SIGUSR1 finché il signal handler relativo non termina tramite invio seguito da "bg <numero job>" (il segnale di stop SIGSTOP è invocato dallo stesso processo con raise), questi non vengono eseguiti ma restano sospesi.

Dopo aver pressato l'invio comunque il gestore verrà eseguito soltanto una volta, anche se sono stati ricevuti più di un segnale SIGUSR1.

La ricezione di SIGUSR2 (kill -s SIGUSR2 <pid>), invece interrompe il signal handler di SIGUSR1.

Attenzione, la funzione printf non è sicura con i segnali.

Vediamo il codice usr_signal1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void sig_user_print1(int signum);
void sig_user_print2(int signum);
int main(int argc, char *argv[]);
// SIGUSR1 signal handler
void sig_user_print1(int signum)
{printf("SIGUSR1 sent...\n"); //il segnale e' arrivato
 printf("Press enter\n");// attendi un carattere (invio)
 getchar();
 raise(SIGSTOP);// invia a te stesso SIGSTOP
                //(bg <numero job> per continuare)
 printf("Exit SIGUSR1\n");    // segnala l'uscita dall'handler
}
// SIGUSR1 signal handler
void sig_user_print2(int signum)
{printf("SIGUSR2 sent...\n");
 printf("Press enter\n");// attendi un carattere (invio)
 getchar();
 printf("Exit SIGUSR2\n");// segnala l'uscita dall'handler
}
```

```c
int main(int argc, char *argv[])
{// puntatori alle funzioni dei signal handler
 void (*sig_old1)(int);
 void (*sig_old2)(int);
 // stampa il pid da utilizzare col kill
 printf("My pid is: %d\n",getpid());
 // sostituisci i signal handler di default
 sig_old1=signal(SIGUSR1,sig_user_print1);
 sig_old2=signal(SIGUSR2,sig_user_print2);
 printf("Wait or type enter to exit...\n");
 getchar();//attendi invio
 // rimetti i signal handler di default
 signal(SIGUSR1,sig_old1);
 signal(SIGUSR2,sig_old2);
 exit(EXIT_SUCCESS);}
```