Lab. di Sistemi Operativi

Esercitazioni proposte per le lezioni del 20-21/05/10

Utilizzando il compilatore gcc in Linux e disponendosi in gruppi di due persone per ogni PC del laboratorio.

1. Invocare la system call pipe e visualizzare il valore dei file descriptor generati (es81.c).
2. Modificare es81.c in modo da immettere qualche carattere nel lato di scrittura e prelevarlo da quello di lettura.
3. Modificando es81.c, verificare cosa succede se si tenta di utilizzare la pipe in direzione opposta.
4. Verificare cosa succede se si tenta di scrivere su una pipe con il lato di lettura chiuso.
5. Verificare con un nuovo sorgente (es82.c) la capacità della pipe scrivendo su di essa fino alla saturazione.
6. Creazione di N-pipe, con N valore del primo argomento. Allocare (dinamicamente) la memoria per i 2N file descriptor e riempire l'array con i file descriptor ritornati dalla pipe(). Visualizzarne i valori su stdout (es83.c).
7. Una pipe come strumento di comunicazione fra processo padre e figlio. Passare un carattere, un intero ed una struttura da padre a figlio (es84.c).
8. Una pipe per N+1 processi (es85.c). Far creare al padre N processi (con N valore del primo argomento) ed utilizzare una singola pipe per passare una struttura di dati da ogni figlio al padre.
9. N pipe per N+1 processi. Far creare al padre N processi ed utilizzare una pipe per ogni processo figlio. Leggere le strutture in modo ordinato, prima dall'ultimo figlio poi dal penultimo e così via fino al primo.
10. N pipe per N+1 processi. Far creare al padre N processi ed utilizzare una pipe per ogni processo figlio ma diretta dal padre verso il figlio. Ogni figlio attende un carattere poi stampa il proprio pid su stdout. Il padre manda un carattere ad ogni figlio dall'ultimo al primo.
11. N pipe per N+2 processi. Far creare al padre N processi ed utilizzare una pipe per collegare ogni processo figlio al successivo. Il primo processo figlio emette il proprio pid poi invia un carattere al secondo che, dopo aver stampato il proprio pid, attiva il successivo ...
12. N pipe per N+2 processi. Come prima ma il primo processo a stampare il pid è l'ultimo creato

Soluzione

Esercitazioni proposte per le lezioni del 20-21/05/10

Utilizzando il compilatore gcc in Linux e disponendosi in gruppi di due persone per ogni PC del laboratorio.

1. Invocare la system call pipe e visualizzare il valore dei file descriptor generati (es81.c).

   Soluzione:

```c
/* file: es81.c
 * job: pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
char msg[256];
int main(int argc, char **argv) {
    int pfd[2]; /* contenitore per i 2 fd */
    if (pipe(pfd)!=0) {
        write(2,"Errore in pipe()\n",17);
        return(1);
    }
    snprintf(msg,sizeof(msg),"pfd[0]=%d, pfd[1]=%d\n",
        pfd[0],
        pfd[1]);
    write(1,msg,strlen(msg));
    return(0);
}
```

2. Modificare es81.c in modo da immettere qualche carattere nel lato di scrittura e prelevarlo da quello di lettura.

   Soluzione:

```c
/* file: es81.c
 * job: pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
char msg[256];
char msg2[256];
int main(int argc, char **argv) {
    int pfd[2]; /* contenitore per i 2 fd */
    if (pipe(pfd)!=0) {
        write(2,"Errore in pipe()\n",17);
        return(1);
    }
    snprintf(msg,sizeof(msg),"pfd[0]=%d, pfd[1]=%d\n",
        pfd[0],
        pfd[1]);
    write(1,msg,strlen(msg)); /* su stdout */
    write(pfd[1],msg,strlen(msg)); /* su msg->pipe */
```

```
    read(pfd[0],msg2,sizeof(msg2)); /* da pipe->msg2 */
    write(1,msg2,strlen(msg2)); /* msg2 su stdout */
    return(0);
}
```

3. Modificando es81.c, verificare cosa succede se si tenta di utilizzare la pipe in direzione opposta.

   Soluzione:

```
/* file: es81.c
 * job: pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
char msg[256];
char msg2[256];
int main(int argc, char **argv) {
    int pfd[2]; /* contenitore per i 2 fd */
    int nw,nr; /* ritorni di read e write */
    if (pipe(pfd)!=0) {
        write(2,"Errore in pipe()\n",17);
        return(1);
    }
    snprintf(msg,sizeof(msg),"pfd[0]=%d, pfd[1]=%d\n",
        pfd[0],
        pfd[1]);
    write(1,msg,strlen(msg)); /* su stdout */
    /* Attenzione all'inversione [0]<->[1] */
    nw=write(pfd[0],msg,strlen(msg)); /* su msg->pipe */
    nr=read(pfd[1],msg2,sizeof(msg2)); /* da pipe->msg2 */
    write(1,msg2,strlen(msg2)); /* msg2 su stdout */
    snprintf(msg,sizeof(msg),"nw= %d, nr= %d\n",
        nw,nr);
    write(1,msg,strlen(msg)); /* su stdout */
    return(0);
}
```

4. Verificare cosa succede se si tenta di scrivere su una pipe con il lato di lettura chiuso.

   Soluzione:

```
/* file: es81.c
 * job: pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
char msg[256];
char msg2[256];
int main(int argc, char **argv) {
    int pfd[2]; /* contenitore per i 2 fd */
    int nw,nr; /* ritorni di read e write */
    if (pipe(pfd)!=0) {
        write(2,"Errore in pipe()\n",17);
        return(1);
```

```
    }
    snprintf(msg,sizeof(msg),"pfd[0]=%d, pfd[1]=%d\n",
        pfd[0],
        pfd[1]);
    write(1,msg,strlen(msg)); /* su stdout */
    close(pfd[0]); /* chiudo lato lettura */
    nw=write(pfd[1],msg,strlen(msg)); /* su msg->pipe */
    /* questa syscall scatena un segnale di
     * "broken pipe" che provoca la terminazione del
     * processo */
    nr=read(pfd[0],msg2,sizeof(msg2)); /* da pipe->msg2 */
    write(1,msg2,strlen(msg2)); /* msg2 su stdout */
    snprintf(msg,sizeof(msg),"nw= %d, nr= %d\n",
        nw,nr);
    write(1,msg,strlen(msg)); /* su stdout */
    return(0);
}
```

5. Verificare con un nuovo sorgente (es82.c) la capacità della pipe scrivendo su di essa fino alla saturazione.

Soluzione:

```
/* file: es82.c
 * job: dimensione buffer pipe
 */
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
char msg[256];
int main(int argc, char **argv) {
    int pfd[2]; /* contenitore per i 2 fd */
    int count; /* contatore */
    int nw; /* ritorno della write */
    if (pipe(pfd)!=0) {
        write(2,"Errore in pipe()\n",17);
        return(1);
    }
    /* Set pfd[1] to non blocking mode */
    if (fcntl(pfd[1],F_SETFL,O_NONBLOCK)!=0) {
        write(2,"Errore in fcntl()\n",18);
        return(1);
    }
    /* Attenzione: se non avessimo modificato il flag
     * O_NONBLOCK, il processo corrente si sarebbe bloccato
     * indefinitamente in attesa di spazio nella pipe,
     * provare per credere rimuovento fcntl() */
    for (count=0; ; count++) {
        nw=write(pfd[1],"A",1);
        if (nw!=1) break;
    }
    snprintf(msg,sizeof(msg),"Ho scritto %d caratteri nella pipe\n",
        count);
    write(1,msg,strlen(msg)); /* su stdout */
    return(0);
```

```
}
```

6. Creazione di N-pipe, con N valore del primo argomento. Allocare (dinamicamente) la memoria per i 2N file descriptor e riempire l'array con i file descriptor ritornati dalla pipe(). Visualizzarne i valori su stdout (es83.c).

   Soluzione:

```c
/* file: es83.c
 * job: array di pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
char msg[256];
int *pfds;
int main(int argc, char **argv) {
    int npipe;
    int i;
    if (argc<2) { /* controllo num args */
        snprintf(msg,sizeof(msg),
            "Uso: %s N\n",argv[0]);
        write(2,msg,strlen(msg));
        return(1);
    }
    npipe=atoi(argv[1]);
    if (npipe<=0) { /* controllo N */
        snprintf(msg,sizeof(msg),
            "N(%d) non valido\n",npipe);
        write(2,msg,strlen(msg));
        return(1);
    }
    pfds = malloc(2*sizeof(int)*npipe);
    if (pfds==NULL) { /* controllo malloc */
        snprintf(msg,sizeof(msg),
            "Allocazione fallita\n");
        write(2,msg,strlen(msg));
        return(1);
    }
    for (i=0; i<npipe; i++) { /* per ogni pipe */
        /* creo la coppia di fds */
        if (pipe(&(pfds[2*i]))!=0) { /* errore */
            snprintf(msg,sizeof(msg),
                "pipe fallita in %d\n",i);
            write(2,msg,strlen(msg));
            return(1);
        }
    }
    for (i=0; i<npipe; i++) { /* per ogni pipe */
        /* stampo i file descriptor */
        snprintf(msg,sizeof(msg),"pfds[2*%d+0]=%d, pfds[2*%d+1]=%d\n",
            i,pfds[2*i+0],
            i,pfds[2*i+1]);
        write(1,msg,strlen(msg)); /* su stdout */
```

```
	}
	return(0);
}
```

7. Una pipe come strumento di comunicazione fra processo padre e figlio. Passare un carattere, un intero ed una struttura da padre a figlio (es84.c).

   Soluzione:

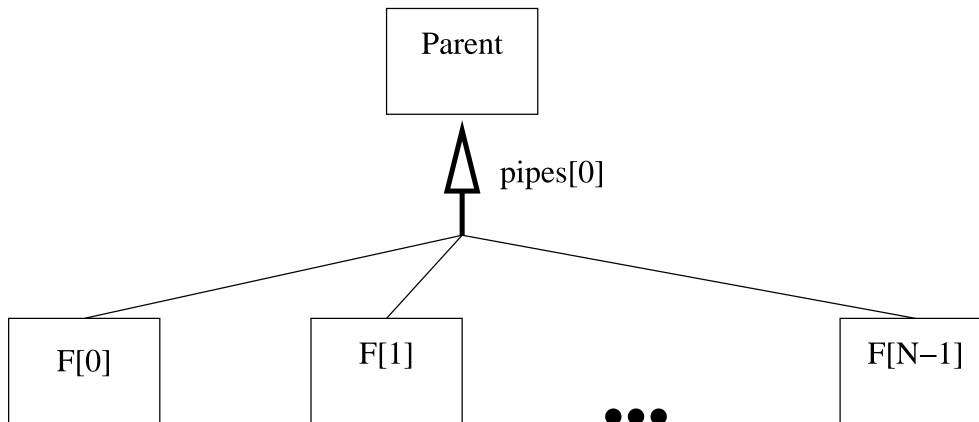
```
/* file: es84.c
 * job: pipe fra processi
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
typedef int pipe_t[2]; /* tipo per pipe */
char msg[256];
char ch; /* carattere da inviare */
int intero; /* intero da inviare */
struct {
	int unint;
	char unchar;
	char string[16];
} struttura; /* struttura da inviare */
pipe_t pfd; /* contenitore per i 2 fd */
int figlio(void); /* funzione figlio */
int main(int argc, char **argv) {
	if (pipe(pfd)!=0) {
		write(2,"Errore in pipe()\n",17);
		return(1);
	}
	switch (fork()) {
		case 0: /* figlio */
			return(figlio());
		case -1:
			break;
	}
	/* solo il padre arriva qui */
	ch='A';
	if (write(pfd[1],&ch,sizeof(ch))!=sizeof(ch)) {
		snprintf(msg,sizeof(msg),"Errore invio ch\n");
		write(2,msg,strlen(msg));
	}
	intero=1250;
	if (write(pfd[1],&intero,sizeof(intero))!=sizeof(intero)) {
		snprintf(msg,sizeof(msg),"Errore invio intero\n");
		write(2,msg,strlen(msg));
	}
	struttura.unchar='B';
	struttura.unint=334;
	strncpy(struttura.string,"Ciao Mondo!",sizeof(struttura.string));
	if (write(pfd[1],&struttura,sizeof(struttura))!=sizeof(struttura)) {
		snprintf(msg,sizeof(msg),"Errore invio struttura\n");
		write(2,msg,strlen(msg));
	}
```

```
    return(0);
}
int figlio() {
    int nr;
    /* chiudo il lato di scrittura */
    close(pfd[1]); /* cosa succede se non lo faccio?*/
    nr=read(pfd[0],&ch,sizeof(ch));
    if (nr==sizeof(ch)) {
        snprintf(msg,sizeof(msg),
            "Ricevo il char %c\n",ch);
        write(1,msg,strlen(msg));
    } else {
        snprintf(msg,sizeof(msg),
            "Attendendo ch, ottengo %d\n",nr);
        write(2,msg,strlen(msg));
    }
    nr=read(pfd[0],&intero,sizeof(intero));
    if (nr==sizeof(intero)) {
        snprintf(msg,sizeof(msg),
            "Ricevo l'intero %d\n",intero);
        write(1,msg,strlen(msg));
    } else {
        snprintf(msg,sizeof(msg),
            "Attendendo intero, ottengo %d\n",nr);
        write(2,msg,strlen(msg));
    }
    nr=read(pfd[0],&struttura,sizeof(struttura));
    if (nr==sizeof(struttura)) {
        snprintf(msg,sizeof(msg),
            "Ricevo la struttura:\n%c\n%d\n%s\n",
            struttura.unchar,
            struttura.unint,
            struttura.string);
        write(1,msg,strlen(msg));
    } else {
        snprintf(msg,sizeof(msg),
            "Attendendo struttura, ottengo %d\n",nr);
        write(2,msg,strlen(msg));
    }
    nr=read(pfd[0],&ch,1);
    if (nr==0) {
        snprintf(msg,sizeof(msg),
            "Ora la pipe e' chiusa\n");
        write(1,msg,strlen(msg));
    }
    return(0);
}
```

8. Una pipe per N+1 processi (es85.c). Far creare al padre N processi (con N valore del primo argomento) ed utilizzare una singola pipe per passare una struttura di dati da ogni figlio al padre.

   Soluzione:

```
/* file: es85.c
 * job: 1 pipe, N+1 processi
```



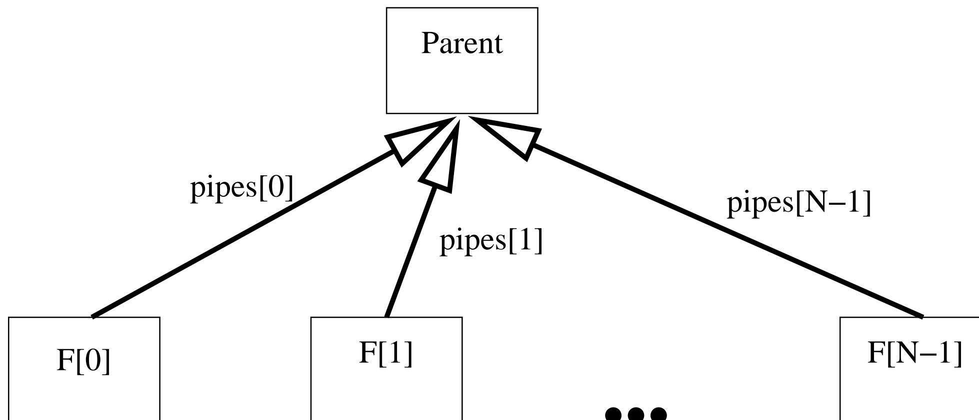

```
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef int pipe_t[2];
char msg[256];
pipe_t *pipes;
struct {
    int indice;
} struttura;
int figlio(int arg);
int main(int argc, char **argv) {
    int npipe;
    int nfigli;
    pid_t pid;
    int i;
    int nr;
    if (argc<2) { /* controllo num args */
        snprintf(msg,sizeof(msg),
            "Uso: %s N\n",argv[0]);
        write(2,msg,strlen(msg));
        return(1);
    }
    nfigli=atoi(argv[1]);
    if (nfigli<=0) { /* controllo N */
        snprintf(msg,sizeof(msg),
            "N(%d) non valido\n",nfigli);
        write(2,msg,strlen(msg));
        return(1);
    }
    npipe=1;
    pipes = malloc(sizeof(pipe_t)*npipe);
    if (pipes==NULL) { /* controllo malloc */
        snprintf(msg,sizeof(msg),
            "Allocazione fallita\n");
        write(2,msg,strlen(msg));
        return(1);
    }
```

```c
for (i=0; i<npipe; i++) { /* per ogni pipe */
    /* creo la coppia di fds */
    if (pipe(pipes[i])!=0) { /* errore */
        snprintf(msg,sizeof(msg),
            "pipe fallita in %d\n",i);
        write(2,msg,strlen(msg));
        return(1);
    }
}
for (i=0; i<nfigli; i++) { /* per ogni figlio */
    pid=fork(); /* lo creo */
    switch(pid) {
        case 0: /* ogni figlio esegue la funzione
                   e termina */
            return(figlio(i));
        case -1:
            snprintf(msg,sizeof(msg),
                "fork() fallita in %d\n",i);
            write(2,msg,strlen(msg));
            return(1);
    }
}
/* solo il padre arriva qui */
close(pipes[0][1]); /* il padre non scrive, chiudo il fd di scritt*/
for (i=0; i<nfigli; i++) { /* per ogni figlio */
    nr=read(pipes[0][0],&struttura,sizeof(struttura));
    /* Nella pipe vengono inserite dai figli dei "blocchi"
     * di byte la cui dimensione e' pari a quella del tipo
     * "struttura". Visto che l'operazione di write e'
     * atomica, il padre o legge una intera struttura o
     * si sospende fino al momento della disponibilita'
     * o, ancora, ottiene 0 per indicare che la pipe
     * non ha dati e non ne potra' mai piu' avere, ovvero
     * quando tutti i processi che avevano il file descriptor
     * del lato di scrittura lo hanno chiuso. */
    switch (nr) {
        case 0: /* pipe chiusa, tutti i figli sono terminati
        ma qualcuno non ha inviato la struttura */
            snprintf(msg,sizeof(msg),
                "Ricevo solo %d strutture\n",i);
            write(2,msg,strlen(msg));
            return(1);
        case sizeof(struttura): /* lettura di una struct */
            snprintf(msg,sizeof(msg),
                "Ricevo la struttura:%d\n",struttura.indice);
            write(1,msg,strlen(msg));
            break;
        default: /* errore */
            snprintf(msg,sizeof(msg),
                "Ricevo solo %d byte\n",nr);
            write(2,msg,strlen(msg));
            return(1);
    }
}
```

```
    return(0);
}
/* funzione figlio */
int figlio(int arg) {
    int nw;
    close(pipes[0][0]); /* figlio non legge */
    struttura.indice=arg;
    nw=write(pipes[0][1],&struttura,sizeof(struttura));
    if (nw!=sizeof(struttura)) {
        snprintf(msg,sizeof(msg),
            "Figlio con indice %d scrive solo %d bytes\n",arg,nw);
        write(2,msg,strlen(msg));
        return(1);
    }
    return(0);
}
```

9. N pipe per N+1 processi. Far creare al padre N processi ed utilizzare una pipe per ogni processo figlio. Leggere le strutture in modo ordinato, prima dall'ultimo figlio poi dal penultimo e così via fino al primo.

   Soluzione:



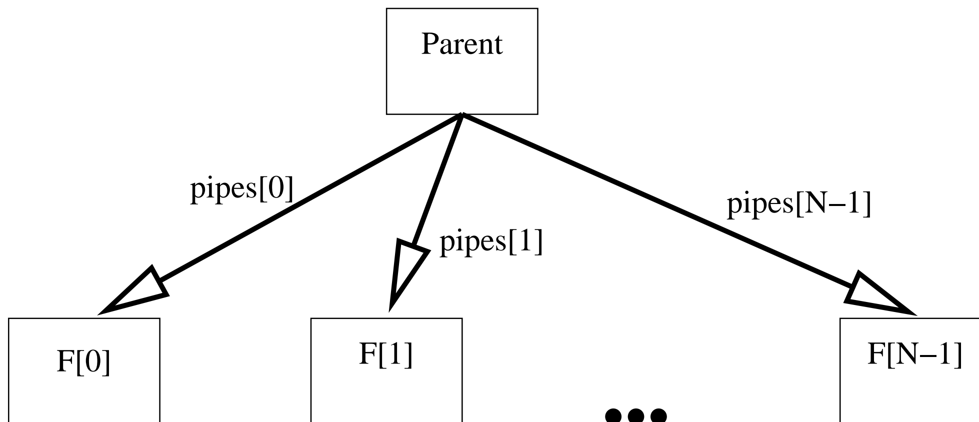

```
/* file: es86.c
 * job: N pipe, N+1 processi
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef int pipe_t[2];
char msg[256];
pipe_t *pipes;
struct {
    int indice;
} struttura;
int figlio(int arg);
int main(int argc, char **argv) {
    int npipe;
    int nfigli;
```

```
pid_t pid;
int i,j;
int nr;
if (argc<2) { /* controllo num args */
    snprintf(msg,sizeof(msg),
        "Uso: %s N\n",argv[0]);
    write(2,msg,strlen(msg));
    return(1);
}
nfigli=atoi(argv[1]);
if (nfigli<=0) { /* controllo N */
    snprintf(msg,sizeof(msg),
        "N(%d) non valido\n",nfigli);
    write(2,msg,strlen(msg));
    return(1);
}
npipe=nfigli;
pipes = malloc(sizeof(pipe_t)*npipe);
if (pipes==NULL) { /* controllo malloc */
    snprintf(msg,sizeof(msg),
        "Allocazione fallita\n");
    write(2,msg,strlen(msg));
    return(1);
}
for (i=0; i<npipe; i++) { /* per ogni pipe */
    /* creo la coppia di fds */
    if (pipe(pipes[i])!=0) { /* errore */
        snprintf(msg,sizeof(msg),
            "pipe fallita in %d\n",i);
        write(2,msg,strlen(msg));
        return(1);
    }
}
for (i=0; i<nfigli; i++) { /* per ogni figlio */
    pid=fork(); /* lo creo */
    switch(pid) {
        case 0: /* ogni figlio esegue la funzione
                  e termina. Prima della chiamata chiudo i fd
                  che il figlio non deve usare */
            for (j=0; j<npipe; j++) {
                close(pipes[j][0]); /* figlio non legge */
                if (j!=i) {
                    /* figlio non scrive sulle altre */
                    close(pipes[j][1]);
                }
            }
            return(figlio(i));
        case -1:
            snprintf(msg,sizeof(msg),
                "fork() fallita in %d\n",i);
            write(2,msg,strlen(msg));
            return(1);
    }
}
```

```
    /* solo il padre arriva qui */
    for (i=0; i<npipe; i++) { /* per tutte le pipe */
        close(pipes[i][1]); /* il padre non scrive, chiudo il fd di scritt*/
    }
    for (i=npipe-1; i>=0; i--) { /* per ogni pipe, dall'ultima alla prima */
        nr=read(pipes[i][0],&struttura,sizeof(struttura));
        /* Nella pipe vengono inserite dai figli dei "blocchi"
         * di byte la cui dimensione e' pari a quella del tipo
         * "struttura". Visto che l'operazione di write e'
         * atomica, il padre o legge una intera struttura o
         * si sospende fino al momento della disponibilita'
         * o, ancora, ottiene 0 per indicare che la pipe
         * non ha dati e non ne potra' mai piu' avere, ovvero
         * quando tutti i processi che avevano il file descriptor
         * del lato di scrittura lo hanno chiuso. */
        switch (nr) {
            case 0: /* pipe chiusa, tutti i figli sono terminati
            ma qualcuno non ha inviato la struttura */
                snprintf(msg,sizeof(msg),
                    "Ricevo solo %d strutture\n",i);
                write(2,msg,strlen(msg));
                return(1);
            case sizeof(struttura): /* lettura di una struct */
                snprintf(msg,sizeof(msg),
                    "Ricevo la struttura:%d\n",struttura.indice);
                write(1,msg,strlen(msg));
                break;
            default: /* errore */
                snprintf(msg,sizeof(msg),
                    "Ricevo solo %d byte\n",nr);
                write(2,msg,strlen(msg));
                return(1);
        }
    }
    return(0);
}
/* funzione figlio */
int figlio(int arg) {
    int nw;
    struttura.indice=arg;
    nw=write(pipes[arg][1],&struttura,sizeof(struttura));
    if (nw!=sizeof(struttura)) {
        snprintf(msg,sizeof(msg),
            "Figlio con indice %d scrive solo %d bytes\n",arg,nw);
        write(2,msg,strlen(msg));
        return(1);
    }
    return(0);
}
```

10. N pipe per N+1 processi. Far creare al padre N processi ed utilizzare una pipe per ogni processo figlio ma diretta dal padre verso il figlio. Ogni figlio attende un carattere poi stampa il proprio pid su stdout. Il padre manda un carattere ad ogni figlio dall'ultimo al primo.

Soluzione:



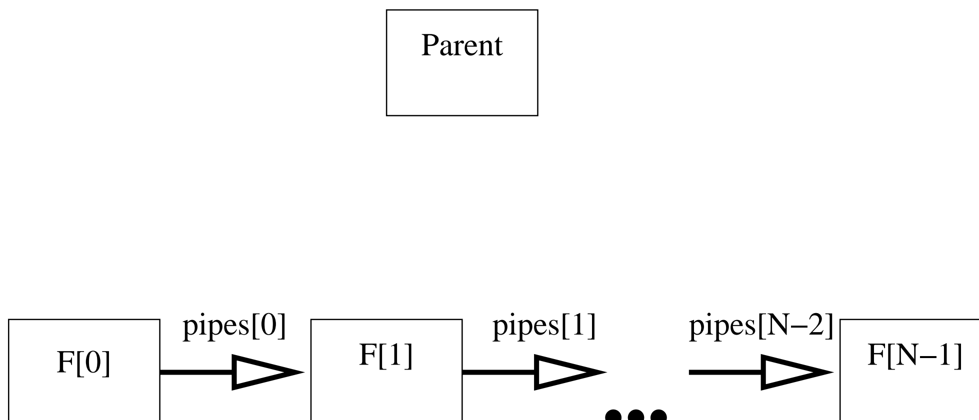

```
/* file: es87.c
 * job: N pipe, N+1 processi, da padre a figlio
 */
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef int pipe_t[2];
char msg[256];
char ch;
pipe_t *pipes;
int figlio(int arg);
int main(int argc, char **argv) {
    int npipe;
    int nfigli;
    pid_t pid;
    int i,j;
    int nw;
    if (argc<2) { /* controllo num args */
        snprintf(msg,sizeof(msg),
            "Uso: %s N\n",argv[0]);
        write(2,msg,strlen(msg));
        return(1);
    }
    nfigli=atoi(argv[1]);
    if (nfigli<=0) { /* controllo N */
        snprintf(msg,sizeof(msg),
            "N(%d) non valido\n",nfigli);
        write(2,msg,strlen(msg));
        return(1);
    }
    npipe=nfigli;
    pipes = malloc(sizeof(pipe_t)*npipe);
    if (pipes==NULL) { /* controllo malloc */
        snprintf(msg,sizeof(msg),
            "Allocazione fallita\n");
        write(2,msg,strlen(msg));
        return(1);
```

```
    }
    for (i=0; i<npipe; i++) { /* per ogni pipe */
        /* creo la coppia di fds */
        if (pipe(pipes[i])!=0) { /* errore */
            snprintf(msg,sizeof(msg),
                "pipe fallita in %d\n",i);
            write(2,msg,strlen(msg));
            return(1);
        }
    }
    for (i=0; i<nfigli; i++) { /* per ogni figlio */
        pid=fork(); /* lo creo */
        switch(pid) {
            case 0: /* ogni figlio esegue la funzione
                     e termina. Prima della chiamata chiudo i fd
                     che il figlio non deve usare */
                for (j=0; j<npipe; j++) {
                    close(pipes[j][1]); /* figlio non scrive */
                    if (j!=i) {
                        /* figlio non legge dalle altre */
                        close(pipes[j][0]);
                    }
                }
                return(figlio(i));
            case -1:
                snprintf(msg,sizeof(msg),
                    "fork() fallita in %d\n",i);
                write(2,msg,strlen(msg));
                return(1);
        }
    }
    /* solo il padre arriva qui */
    for (i=0; i<npipe; i++) { /* per tutte le pipe */
        close(pipes[i][0]); /* il padre non legge, chiudo il fd di lett*/
    }
    for (i=npipe-1; i>=0; i--) { /* per ogni pipe, dall'ultima alla prima */
        nw=write(pipes[i][1],&ch,1);
        switch (nw) {
            case 0: /* impossibile, se non c'e' spazio attende */
                snprintf(msg,sizeof(msg),
                    "Non scrivo il carattere\n");
                write(2,msg,strlen(msg));
                return(1);
            case 1: /* OK */
                snprintf(msg,sizeof(msg),
                    "Do il via al figlio %d\n",i);
                write(1,msg,strlen(msg));
                break;
            default: /* errore */
                snprintf(msg,sizeof(msg),
                    "Errore in write\n");
                write(2,msg,strlen(msg));
                return(1);
        }
```

```
    }
    return(0);
}
/* funzione figlio */
int figlio(int arg) {
    int nr;
    nr=read(pipes[arg][0],&ch,1);
    if (nr!=1) {
        snprintf(msg,sizeof(msg),
            "Figlio con indice %d ottiene %d da read\n",arg,nr);
        write(2,msg,strlen(msg));
        return(1);
    }
    /* se qui, ho ricevuto l'ok e stampo il mio pid */
    snprintf(msg,sizeof(msg),"Figlio con pid %d\n",getpid());
    write(1,msg,strlen(msg));
    return(0);
}
/* Attenzione: questo sistema non assicura che le write dei figli su stdout
avvengano in sequenza! Una volta ottenuto l'ok dal padre i figli diventano
processi "eseguibili" e l'ordine con il quale il sistema assegna le slice
ai figli e' casuale. Una soluzione esatta del problema si ottiene con il
prossimo sorgente */
```

11. N pipe per N+2 processi. Far creare al padre N processi ed utilizzare una pipe per collegare ogni processo figlio al successivo. Il primo processo figlio emette il proprio pid poi invia un carattere al secondo che, dopo aver stampato il proprio pid, attiva il successivo ...

    Soluzione:




```
/* file: es88.c
 * job: N pipe, N+2 processi, da figlio a figlio
 */
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
typedef int pipe_t[2];
```

```
char msg[256];
char ch;
pipe_t *pipes;
int figlio(int arg, int ultimo);
int main(int argc, char **argv) {
    int npipe;
    int nfigli;
    pid_t pid;
    int i,j;
    if (argc<2) { /* controllo num args */
        snprintf(msg,sizeof(msg),
            "Uso: %s N\n",argv[0]);
        write(2,msg,strlen(msg));
        return(1);
    }
    nfigli=atoi(argv[1]);
    if (nfigli<=1) { /* controllo N */
        snprintf(msg,sizeof(msg),
            "N(%d) non valido\n",nfigli);
        write(2,msg,strlen(msg));
        return(1);
    }
    npipe=nfigli-1;
    pipes = malloc(sizeof(pipe_t)*npipe);
    if (pipes==NULL) { /* controllo malloc */
        snprintf(msg,sizeof(msg),
            "Allocazione fallita\n");
        write(2,msg,strlen(msg));
        return(1);
    }
    for (i=0; i<npipe; i++) { /* per ogni pipe */
        /* creo la coppia di fds */
        if (pipe(pipes[i])!=0) { /* errore */
            snprintf(msg,sizeof(msg),
                "pipe fallita in %d\n",i);
            write(2,msg,strlen(msg));
            return(1);
        }
    }
    for (i=0; i<nfigli; i++) { /* per ogni figlio */
        pid=fork(); /* lo creo */
        switch(pid) {
            case 0: /* ogni figlio esegue la funzione
                   e termina. Prima della chiamata chiudo i fd
                   che il figlio non deve usare */
                for (j=0; j<npipe; j++) {
                    if (j!=i) { /* figlio[i] scrive su pipe[i] */
                        close(pipes[j][1]);
                    }
                    if (j!=(i-1)) { /* e legge da pipe[i-1] */
                        close(pipes[j][0]);
                    }
                }
                return(figlio(i,nfigli-1));
```

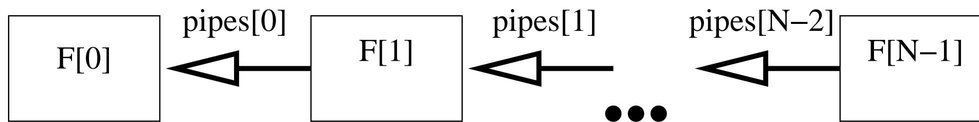
```
                case -1:
                    snprintf(msg,sizeof(msg),
                        "fork() fallita in %d\n",i);
                    write(2,msg,strlen(msg));
                    return(1);
            }
        }
        /* solo il padre arriva qui */
        for (i=0; i<npipe; i++) { /* per tutte le pipe */
            close(pipes[i][0]); /* il padre non legge, chiudo il fd di lett*/
            close(pipes[i][1]); /* e non scrive, chiudo il fd di scrittura */
        }
        /* volendo, posso attendere i figli per recuperare gli exit value */
        for (i=0; i<nfigli; i++) {
            wait(NULL);
        }
        return(0);
    }
    /* funzione figlio */
    int figlio(int arg, int ultimo) {
        int nr,nw;
        if (arg!=0) { /* F[0] non deve attendere */
            nr=read(pipes[arg-1][0],&ch,1);
            /* read bloccante */
            if (nr!=1) {
                snprintf(msg,sizeof(msg),
                        "Figlio con indice %d ottiene %d da read\n",arg,nr);
                write(2,msg,strlen(msg));
                return(1);
            }
        }
        /* se qui, ho ricevuto l'ok e stampo il mio pid */
        snprintf(msg,sizeof(msg),"Figlio con pid %d\n",getpid());
        write(1,msg,strlen(msg));
        /* ora do il via al figlio successivo a meno che questo non sia
         * l'ultimo figlio */
        if (arg!=ultimo) {
            nw=write(pipes[arg][1],&ch,1);
            if (nw!=1) {
                snprintf(msg,sizeof(msg),
                        "Figlio con indice %d ottiene %d da write\n",arg,nw);
                write(2,msg,strlen(msg));
                return(1);
            }
        }
        return(0);
    }
```

12. N pipe per N+2 processi. Come prima ma il primo processo a stampare il pid è l'ultimo creato

Soluzione:

```
/* file: es89.c
 * job: N pipe, N+2 processi, da figlio a figlio in ordine inverso
 */
```

Parent




```c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
typedef int pipe_t[2];
char msg[256];
char ch;
pipe_t *pipes;
int figlio(int arg, int ultimo);
int main(int argc, char **argv) {
    int npipe;
    int nfigli;
    pid_t pid;
    int i,j;
    if (argc<2) { /* controllo num args */
        snprintf(msg,sizeof(msg),
            "Uso: %s N\n",argv[0]);
        write(2,msg,strlen(msg));
        return(1);
    }
    nfigli=atoi(argv[1]);
    if (nfigli<=1) { /* controllo N */
        snprintf(msg,sizeof(msg),
            "N(%d) non valido\n",nfigli);
        write(2,msg,strlen(msg));
        return(1);
    }
    npipe=nfigli-1;
    pipes = malloc(sizeof(pipe_t)*npipe);
    if (pipes==NULL) { /* controllo malloc */
        snprintf(msg,sizeof(msg),
            "Allocazione fallita\n");
        write(2,msg,strlen(msg));
        return(1);
    }
    for (i=0; i<npipe; i++) { /* per ogni pipe */
        /* creo la coppia di fds */
```

```
            if (pipe(pipes[i])!=0) { /* errore */
                snprintf(msg,sizeof(msg),
                    "pipe fallita in %d\n",i);
                write(2,msg,strlen(msg));
                return(1);
            }
        }
        for (i=0; i<nfigli; i++) { /* per ogni figlio */
            pid=fork(); /* lo creo */
            switch(pid) {
                case 0: /* ogni figlio esegue la funzione
                       e termina. Prima della chiamata chiudo i fd
                       che il figlio non deve usare */
                    for (j=0; j<npipe; j++) {
                        if (j!=(i-1)) { /* figlio[i] scrive su pipe[i-1] */
                            close(pipes[j][1]);
                        }
                        if (j!=i) { /* e legge da pipe[i] */
                            close(pipes[j][0]);
                        }
                    }
                    return(figlio(i,nfigli-1));
                case -1:
                    snprintf(msg,sizeof(msg),
                        "fork() fallita in %d\n",i);
                    write(2,msg,strlen(msg));
                    return(1);
            }
        }
        /* solo il padre arriva qui */
        for (i=0; i<npipe; i++) { /* per tutte le pipe */
            close(pipes[i][0]); /* il padre non legge, chiudo il fd di lett*/
            close(pipes[i][1]); /* e non scrive, chiudo il fd di scrittura */
        }
        /* volendo, posso attendere i figli per recuperare gli exit value */
        for (i=0; i<nfigli; i++) {
            wait(NULL);
        }
        return(0);
    }
    /* funzione figlio */
    int figlio(int arg, int ultimo) {
        int nr,nw;
        if (arg!=ultimo) { /* F[N-1] non deve attendere */
            nr=read(pipes[arg][0],&ch,1);
            /* read bloccante */
            if (nr!=1) {
                snprintf(msg,sizeof(msg),
                       "Figlio con indice %d ottiene %d da read\n",arg,nr);
                write(2,msg,strlen(msg));
                return(1);
            }
        }
        /* se qui, ho ricevuto l'ok e stampo il mio pid */
```

```
    snprintf(msg,sizeof(msg),"Figlio con pid %d\n",getpid());
    write(1,msg,strlen(msg));
    /* ora do il via al figlio successivo a meno che questo non sia
     * il primo figlio */
    if (arg!=0) {
        nw=write(pipes[arg-1][1],&ch,1);
        if (nw!=1) {
            snprintf(msg,sizeof(msg),
                    "Figlio con indice %d ottiene %d da write\n",arg,nw);
            write(2,msg,strlen(msg));
            return(1);
        }
    }
    return(0);
}
```