

# Lab. di Sistemi Operativi

## Esercitazioni proposte per le lezioni del 27-28/05/2010

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

1. Determinare con un programma c (es91.c) il segnale che causa la terminazione di un processo figlio quando questo tenta di scrivere su una pipe con lato di lettura chiuso (completamento dell'esercizio della scorsa settimana)
2. Modificare il precedente sorgente (in es92.c) in modo che il segnale di broken pipe (man 7 signal: SIGPIPE,13,Broken pipe) venga intercettato da un opportuno handler.
3. (facoltativo) Ottenere un ritardo con l'uso del segnale SIGALRM. Progettare un applicativo che si sospenda per 10 secondi senza utilizzare la funzione sleep().
4. Progettare un applicativo concorrente con due processi (padre e figlio) che si alternano nella scrittura su stdout mediante segnali (es94.c)
5. i modifichi il codice precedente per verificare cosa accade quando un segnale arriva mentre se ne sta servendo un altro
6. Soluzione al compito del 8 aprile 2005 (es95.c)

La parte in C accetta un numero variabile di parametri che rappresentano nomi di file F1...FM e un numero K strettamente positivo, pari e strettamente minore di 7 che rappresenta il numero di linee di ogni file. Il processo padre deve generare M processi figli (P1..PM): ogni processo figlio associato ad uno dei file Fi. Ognuno di tali processi figli deve creare un file il cui nome (FiK) risulti dalla concatenazione del nome del file associato (Fi) con la stringa che corrisponde al numero K; quindi, ogni figlio deve creare un processo nipote: la coppia figlio e nipote esegue concorrentemente leggendo, rispettivamente, il figlio le linee dispari e il nipote le linee pari. Il processo nipote, dopo la lettura di ogni linea pari, calcola la sua lunghezza (LP) e la comunica al processo figlio; il processo figlio, dopo la lettura di ogni linea dispari, calcola la sua lunghezza (LD) e riceve dal nipote LP: se LD minore di LP, allora il processo figlio scrive la sua linea sul file FiK(\*). Ogni processo figlio deve ritornare al padre il numero di linee scritte sul file FiK.

Il padre, dopo che i figli sono terminati, deve stampare su standard output i PID di ogni figlio con il corrispondente valore ritornato.

(\*) Le letture dal singolo file da parte di una coppia di processi figlio-nipote non devono essere sincronizzate.

7. Soluzione al compito del 12 marzo 2004 (es96.c)

La parte in C accetta un numero variabile di parametri che rappresentano un nome di file F e N caratteri C1 ... CN. Il processo padre deve generare N processi figli (P1..PN): ogni processo figlio è associato ad uno dei caratteri Ci. Ognuno di tali processi figli esegue concorrentemente e verifica se il file F contiene il carattere associato Ci; in particolare, ogni processo figlio deve riportare sullo standard output il numero d'ordine di ogni linea di F che contiene il carattere associato. La scrittura dei processi figli deve essere sincronizzata (senza l'intervento del padre) affinché venga stampata prima una informazione di P1, poi di P2, e cos via fino a PN, dopodiché si ricomincia da P1 finché tutti i processi non hanno verificato tutto il file. Al termine, ogni processo figlio deve ritornare al padre il numero di linee che contengono il carattere associato. Il padre visualizza su standard output i valori ritornati dai figli.

NOTA BENE:

Si consideri ipotizzabile che i processi figli terminano in cascata, dal primo all'ultimo.

8. Invertire l'ordine di sincronizzazione dei figli nell'esercizio es96.c (es97.c)
9. Evidenziare le differenze fra i due file

# Soluzione

## Esercitazioni proposte per le lezioni del 27-28/05/2010

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

1. Determinare con un programma c (es91.c) il segnale che causa la terminazione di un processo figlio quando questo tenta di scrivere su una pipe con lato di lettura chiuso (completamento dell'esercizio della scorsa settimana)

### Soluzione:

```
/* file: es91.c
 * job: identificazione del segnale broken pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.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 */
    int st;
    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 */
    /* utilizziamo la fork per lasciare al processo figlio
     * l'incombenza di scrivere sulla pipe e recuperiamo
     * le informazioni di terminazione con la wait */
    switch (fork()) {
        case 0: /* figlio */
            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);
        case -1: /* errore fork */
            write(2,"Errore_fork\n",12); /* su stderr */
            return(1);
        break;
    }
}
```

```

}
/* padre: attendo il figlio (unico, non devo recuperare il pid) */
wait(&st);
if (WIFEXITED(st)) { /* terminazione "naturale", exit value valido */
    snprintf(msg, sizeof(msg), "Figlio_torna_%d\n",
        WEXITSTATUS(st));
    write(1, msg, strlen(msg));
} else { /* terminazione forzata, no exit value ma id segnale */
    snprintf(msg, sizeof(msg), "Figlio_terminato_da_%d\n",
        WTERMSIG(st));
    write(1, msg, strlen(msg));
}
return(0);
}

```

2. Modificare il precedente sorgente (in es92.c) in modo che il segnale di broken pipe (man 7 signal: SIGPIPE,13,Broken pipe) venga intercettato da un opportuno handler.

**Soluzione:**

```

/* file: es92.c
 * job: intercettazione del segnale broken pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
char msg[256];
char msg2[256];
/* prototipo dell'handler */
void intercetta(int segnale);
int main(int argc, char **argv) {
    int pfd[2]; /* contenitore per i 2 fd */
    int nw, nr; /* ritorni di read e write */
    int st;
    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 */
    /* utilizziamo la fork per lasciare al processo figlio
     * l'incombenza di scrivere sulla pipe e recuperiamo
     * le informazioni di terminazione con la wait */
    switch (fork()) {
        case 0: /* figlio */
            /* aggancio intercetta() a SIGPIPE */
            signal(SIGPIPE, intercetta);
            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);
case -1: /* errore fork */
write(2,"Errore_fork\n",12); /* su stderr */
return(1);
break;
}
/* padre: attendo il figlio (unico, non devo recuperare il pid) */
wait(&st);
if (WIFEXITED(st)) { /* terminazione "naturale", exit value valido */
snprintf(msg,sizeof(msg),"Figlio_torna_%d\n",
        WEXITSTATUS(st));
write(1,msg,strlen(msg));
} else { /* terminazione forzata, no exit value ma id segnale */
snprintf(msg,sizeof(msg),"Figlio_terminato_da_%d\n",
        WTERMSIG(st));
write(1,msg,strlen(msg));
}
return(0);
}
}
void intercetta(int segnale) {
snprintf(msg,sizeof(msg),"Figlio:_intercetto_%d\n",
        segnale);
write(1,msg,strlen(msg));
return;
}
}

```

3. (facoltativo) Ottenere un ritardo con l'uso del segnale SIGALRM. Progettare un applicativo che si sospenda per 10 secondi senza utilizzare la funzione sleep().

**Soluzione:**

```

/* file: es93.c
 * job: attesa temporizzata
 */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <setjmp.h>
void sigfun(int sigid);
char msg[256];
jmp_buf stato; /* salvataggio stato */
int main(int argc, char **argv) {
/* colleghiamo SIGALRM alla nostra sigfun */
signal(SIGALRM, sigfun);
if (setjmp(stato)) {
/* se non zero, allora torno */
snprintf(msg, sizeof(msg),

```

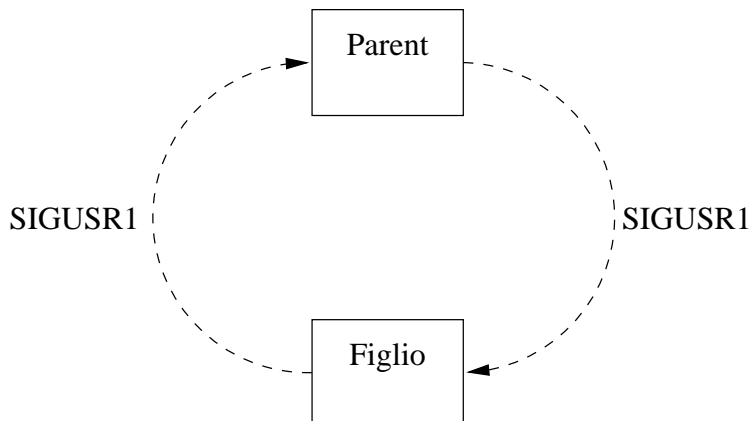
```

        "Attesa_scaduta\n");
    write(1,msg,strlen(msg));
} else {
    /* se zero, primo passaggio */
    snprintf(msg,sizeof(msg),
        "Attesa_iniziata\n");
    write(1,msg,strlen(msg));
    alarm(10);
    snprintf(msg,sizeof(msg),
        "Sveglia_impostata\n");
    write(1,msg,strlen(msg));
    for (;;) {
        pause();
    }
}
return(0);
}
void sigfun(int sigid) {
    if (sigid==SIGALRM) {
        longjmp(stato,1);
    }
}
}

```

4. Progettare un applicativo concorrente con due processi (padre e figlio) che si alternano nella scrittura su stdout mediante segnali (es94.c)

**Soluzione:**



```

/* file: es94.c
 * job: sincronizzazione con segnali
 */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
void sigfun(int sigid);
int figlio(void);
volatile int consenso;
char msg[256];

```

```

int main(int argc, char **argv) {
    pid_t pidfiglio;

    /* colleghiamo SIGUSR1 alla nostra sigfun */
    signal(SIGUSR1, sigfun);
    consenso=0; /* consenso assente */
    switch(pidfiglio=fork()) {
        case 0: /* figlio */
            return(figlio());
        case -1: /* errore */
            snprintf(msg, sizeof(msg),
                "Errore_fork()\n");
            write(2, msg, strlen(msg));
            return(1);
    }
    /* padre */
    consenso=1; /* il padre inizia */
    for (;;) {
        snprintf(msg, sizeof(msg), "Padre\n");
        while(!consenso) pause();
        write(1, msg, strlen(msg));
        sleep(1); /* rallentatore! */
        consenso=0; /* mi blocco */
        kill(pidfiglio, SIGUSR1); /* sblocco */
    }
    return(0);
}

int figlio() {
    for (;;) {
        snprintf(msg, sizeof(msg), "Figlio\n");
        while(!consenso) pause();
        write(1, msg, strlen(msg));
        sleep(1); /* rallentatore! */
        consenso=0; /* mi blocco */
        kill(getppid(), SIGUSR1); /* sblocco */
    }
    return(0);
}

void sigfun(int sigid) {
    if (sigid==SIGUSR1) {
        consenso=1;
    }
}

```

5. i modifichi il codice precedente per verificare cosa accade quando un segnale arriva mentre se ne sta servendo un altro

**Soluzione:**

```

/* file: es94b.c
 * job: test per rientranza handler
 */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

```

```

#include <sys/types.h>
#include <string.h>
void sigfun(int sigid);
int figlio(void);
char msg[256];
char msgS[256];
int main(int argc, char **argv) {
    pid_t pidfiglio;
    int i;

    /* colleghiamo SIGUSR1 alla nostra sigfun */
    signal(SIGUSR1, sigfun);
    /* colleghiamo anche SIGUSR2 a sigfun */
    signal(SIGUSR2, sigfun);
    switch(pidfiglio=fork()) {
        case 0: /* figlio */
            return(figlio());
        case -1: /* errore */
            snprintf(msg, sizeof(msg),
                "Errore_fork()\n");
            write(2, msg, strlen(msg));
            return(1);
    }
    /* padre */
    for (i=0; i<2; i++) {
        sleep(1); /* rallentatore! */
        snprintf(msg, sizeof(msg), "Padre, _invio_SIGUSR1\n");
        write(1, msg, strlen(msg));
        kill(pidfiglio, SIGUSR1);
        sleep(1); /* rallentatore! */
        snprintf(msg, sizeof(msg), "Padre, _invio_SIGUSR2\n");
        write(1, msg, strlen(msg));
        kill(pidfiglio, SIGUSR2);
    }
    return(0);
}
int figlio() {
    for (;;) {
        snprintf(msg, sizeof(msg), "Figlio, _attendo_con_pause()\n");
        write(1, msg, strlen(msg));
        pause();
    }
    return(0);
}
void sigfun(int sigid) {
    /* nota1: msg e' condiviso fra contesto principale e contesto
     * di segnale, quindi devo usare un buffer diverso */
    if (sigid==SIGUSR1) {
#ifdef LOOP_SIGUSR1
        snprintf(msgS, sizeof(msgS), "Figlio, _ricevo_SIGUSR1_e_non_esco\n");
        write(1, msgS, strlen(msgS));
        for (;;)
#else
        snprintf(msgS, sizeof(msgS), "Figlio, _ricevo_SIGUSR1\n");

```

```

        write(1,msgS,strlen(msgS));
    #endif
    }
    if (sigid==SIGUSR2) {
        snprintf(msgS,sizeof(msgS),"Figlio ,ricevo SIGUSR2\n");
        write(1,msgS,strlen(msgS));
    }
}

```

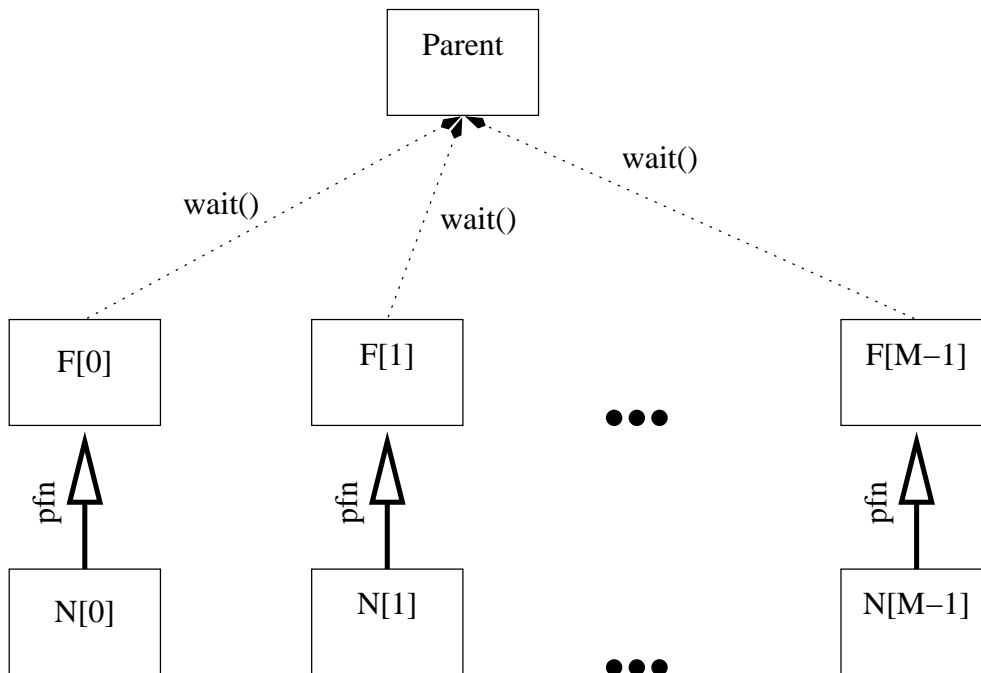
6. Soluzione al compito del 8 aprile 2005 (es95.c)

La parte in C accetta un numero variabile di parametri che rappresentano nomi di file F1...FM e un numero K strettamente positivo, pari e strettamente minore di 7 che rappresenta il numero di linee di ogni file. Il processo padre deve generare M processi figli (P1..PM): ogni processo figlio associato ad uno dei file Fi. Ognuno di tali processi figli deve creare un file il cui nome (FiK) risulti dalla concatenazione del nome del file associato (Fi) con la stringa che corrisponde al numero K; quindi, ogni figlio deve creare un processo nipote: la coppia figlio e nipote esegue concorrentemente leggendo, rispettivamente, il figlio le linee dispari e il nipote le linee pari. Il processo nipote, dopo la lettura di ogni linea pari, calcola la sua lunghezza (LP) e la comunica al processo figlio; il processo figlio, dopo la lettura di ogni linea dispari, calcola la sua lunghezza (LD) e riceve dal nipote LP: se LD minore di LP, allora il processo figlio scrive la sua linea sul file FiK(\*) . Ogni processo figlio deve ritornare al padre il numero di linee scritte sul file FiK.

Il padre, dopo che i figli sono terminati, deve stampare su standard output i PID di ogni figlio con il corrispondente valore ritornato.

(\*) Le letture dal singolo file da parte di una coppia di processi figlio-nipote non devono essere sincronizzate.

**Soluzione:**



```

/* file: es95.c
 * job: soluzione 8 aprile 2005
 */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdarg.h>
int M; /* numero di processi figli */
int K; /* ultimo argomento */
pid_t pid; /* appoggio per valore pid */
int status; /* appoggio per wait */
int pfn[2]; /* pipe figlio-nipote, attenzione ogni processo figlio
ha la sua istanza di questa variabile quindi le pipe create sono "M" */
int fdr,fdw; /* file descriptors per read e write */
int indice; /* indice su linea */
char linea[256]; /* buffer di linea */
char FiK[80]; /* nome file di uscita */
int LP,LD; /* lunghezze linea */
int lineacorr; /* linea corrente */
int nonscritte; /* numero di linee non scritte */
/* abbinamento snprintf+write */
void zprintf(int fd, const char *fmt, ...);
/* funzione figlio */
int figlio(char *filename, int i);
/* funzione nipote */
int nipote(char *filename, int i);
int main(int argc, char **argv) {
    int i; /* int di appoggio */
    /* controllo numero di argomenti */
    if (argc<3) {
        zprintf(2,"Errore_nel_numero_di_argomenti\n");
        return(1);
    }
    M=argc-2; /* numero di processi figli */
    K=atoi(argv[argc-1]); /* numero K, ultimo argomento */
    if ((K<=0)||((K>=7)||((K%2)!=0))) {
        zprintf(2,"Parametro_K_non_valido:%d\n",K);
        return(2);
    }
    /* creazione figli e attribuzione argomenti */
    for (i=0; i<M; i++) {
        switch (fork()) {
            case 0: /* figlio */
                return(figlio(argv[i+1],i));
            case -1:
                /* errore */
                zprintf(2,"Errore_nella_%d_fork\n",i);
                return(3);
        }
    }
    /* recupero exit status dei figli (non dei nipoti) */
    for (i=0; i<M; i++) {
        pid=wait(&status);
        if (WIFEXITED(status)) {
            zprintf(1,

```

```

        " Il figlio con pid %d ritorna %d\n",
        pid, WEXITSTATUS(status));
    } else {
        zprintf(1,
        " Il figlio con pid %d e' terminato per il segnale %d\n",
        pid, WTERMSIG(status));
    }
}
return(0);
}
void zprintf(int fd, const char *fmt, ...) {
    /* printf wrapper using write instead */
    static char msg[256];
    va_list ap;
    int n;
    va_start(ap, fmt);
    n=vsprintf(msg, 256, fmt, ap);
    write(fd, msg, n);
    va_end(ap);
}
int figlio(char *filename, int i) {
    char ch;
    /* creo la pipe figlio-nipote */
    if (pipe(pfn)!=0) {
        zprintf(2, "Errore creazione pipe figlio-nipote %d\n", i);
        abort(); /* termino per SIGABRT, il padre se ne accorge */
        return(4);
    }
    /* creo il nipote */
    switch (fork()) {
        case 0:
            return(nipote(filename, i));
        case -1:
            /* errore */
            zprintf(2, "Errore nella %d fork (nipote)\n", i);
            abort();
    }
    close(pfn[1]); /* non scrivo mai */
    /* legge le linee 1,3,... evitando quelle pari */
    fdr=open(filename, O_RDONLY);
    if (fdr<0) {
        zprintf(2, "Impossibile aprire %s\n", filename);
        abort();
    }
    sprintf(FiK, sizeof(FiK), "%s%d", filename, i);
#ifdef DEBUG
    zprintf(1, "Creato %s\n", FiK);
#endif
    fdw=open(FiK, O_WRONLY|O_CREAT|O_EXCL, 0666);
    if (fdw<0) {
        zprintf(2, "Impossibile aprire %s\n", FiK);
        abort();
    }
    lineacorr=1;
}

```

```

    indice=0;
    for(;;) {
        if (read(fdr,&ch,1)!=1) {
            return(nonscritte);
        }
#ifdef DEBUG1
        zprintf(1,"Char_%d\n",ch);
#endif
        if (ch=='\n') {
            /* fine linea */
            if ((lineacorr%2)==1) {
                LD=indice;
                /* attendo info da nipote */
#ifdef DEBUG
                zprintf(1,"Attendo_info_per_linea_%d_lung_%d\n",
                    lineacorr,LD);
#endif
                if (read(pfn[0],&LP,sizeof(LP))!=sizeof(LP)) {
                    zprintf(2,"Errore_lettura_pipe\n");
                    abort();
                }
                if (LD>LP) {
                    write(fdw,linea,LD);
                    write(fdw,"\n",1); /* fine linea */
                } else {
                    nonscritte++;
                }
            }
            indice=0;
            lineacorr++;
        } else if ((lineacorr%2)==1) {
            /* linea dispari, memorizzo */
            if (indice<sizeof(linea)) {
                linea[indice++]=ch;
            }
        }
    }
    /* fine figlio, mai qui */
    abort();
    return(0);
}
int nipote(char *filename, int i) {
    char ch;
    /* nipote, linee pari */
    fdr=open(filename,ORDONLY);
    if (fdr<0) {
        zprintf(2,"Impossibile_aprire_%s\n",filename);
        return(1); /* recuperabile con wait dal figlio */
    }
    lineacorr=1;
    indice=0;
    for(;;) {
        if (read(fdr,&ch,1)!=1) {
            /* fine file */

```

```

        return(0);
    }
    if (ch=='\n') {
        /* fine linea */
        if ((lineacorr%2)==0) {
            LP=indice;
            write(pfn[1],&LP, sizeof(LP));
#ifdef DEBUG
            zprintf(1,"Inviata_lung_linea_%d,_pari_a_%d\n",
                lineacorr,LP);
#endif
        }
        indice=0;
        lineacorr++;
    } else {
        indice++;
    }
}
return(0);
}

```

#### 7. Soluzione al compito del 12 marzo 2004 (es96.c)

La parte in C accetta un numero variabile di parametri che rappresentano un nome di file F e N caratteri C1 ... CN. Il processo padre deve generare N processi figli (P1..PN): ogni processo figlio è associato ad uno dei caratteri Ci. Ognuno di tali processi figli esegue concorrentemente e verifica se il file F contiene il carattere associato Ci; in particolare, ogni processo figlio deve riportare sullo standard output il numero d'ordine di ogni linea di F che contiene il carattere associato. La scrittura dei processi figli deve essere sincronizzata (senza l'intervento del padre) affinché venga stampata prima una informazione di P1, poi di P2, e cos via fino a PN, dopodiché si ricomincia da P1 finché tutti i processi non hanno verificato tutto il file. Al termine, ogni processo figlio deve ritornare al padre il numero di linee che contengono il carattere associato. Il padre visualizza su standard output i valori ritornati dai figli.

NOTA BENE:

Si consideri ipotizzabile che i processi figli terminano in cascata, dal primo all'ultimo.

**Soluzione:**

```

/* Possibile soluzione della prova in itinere del 12 marzo 2004
 * analizzata nelle lezioni di Laboratorio di Sistemi Operativi
 * del 14 e 15 marzo 2005 */
/* $Id: es96.c,v 1.1 2009/03/09 07:30:21 valealex Exp $ */

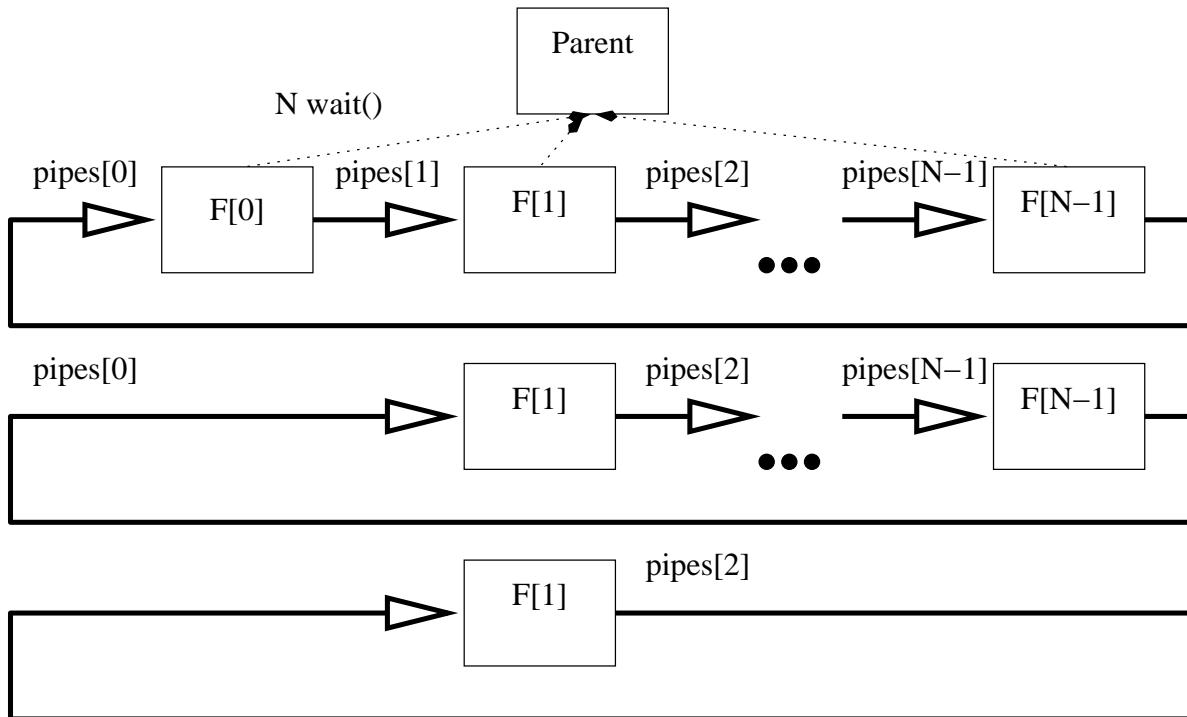
```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdarg.h>

typedef int pipe_t[2]; /* tipo per pipe */
#define SCRIT 1
#define LETT 0

```



```

char *F; /* Nome del file , variabile globale */
int N; /* Numero di caratteri */
char *C; /* Elenco dei caratteri , allocazione dinamica */
int Npipe; /* Numero di pipe da creare */
pipe_t *Pfds; /* Array di pipe , allocazione dinamica */
int Nfigli; /* Numero di figli da forkare */
int *Pids; /* Array di pid , allocazione dinamica */

/* abbinamento snprintf+write */
void zprintf(int fd, const char *fmt, ...);
/* funzione figlio */
int figlio(int);

char ok='Q'; /* carattere (casuale) che inviamo sulla pipe
per la sincronizzazione */

int main(int argc, char **argv, char **envp) {
    int i,j; /* qualche contatore */

    /* Determinazione dei dati dalla command line */
    switch (argc) {
        case 0:
        case 1:
        case 2:
            zprintf(2, "Inserire il nome di file ed almeno un carattere\n");
            return(1);
    }
    F = argv[1];
    N = argc-2; /* argc vale 1 se non ci sono argomenti */

```

```

C = malloc(N*sizeof(char));
if (C==NULL) {
    zprintf(2,"Impossibile allocare C\n");
    return(1);
}
for (i=0; i<N; i++) {
    C[i]=argv[i+2][0]; /* prendo il primo carattere della stringa
                        argv[i+2] */
}
#ifdef DEBUG
zprintf(1,"Nome del file: %s\n",F);
zprintf(1,"Numero di caratteri %d\n",N);
for (i=0; i<N; i++) {
    zprintf(1,"Carattere di indice %d = %c\n",i,C[i]);
}
#endif
/* Predisposizione del sistema di sincronizzazione
 * fra i processi figli: si utilizza una pipe per
 * collegare ogni figlio al successivo; la sincronizzazione
 * del figlio di indice 'k' si ottiene attendendo un
 * carattere dalla pipe che lo collega al figlio di
 * indice 'k-1'. */
Npipe = N; /* una pipe in uscita da ogni figlio */
Pfds = malloc(Npipe*sizeof(pipe_t));
if (Pfds==NULL) {
    zprintf(2,"Impossibile allocare Pfds\n");
    return(1);
}
for (i=0; i<Npipe; i++) {
    if (pipe(Pfds[i])!=0) {
        zprintf(2,"Impossibile creare la pipe %d\n",i);
        return(1);
    }
}
/* Creazione processi figli:
 * dal testo, bisogna creare un processo figlio per ogni carattere
 * quindi Nfigli=N
 * Visto che il padre deve stampare i valori di uscita dei processi
 * figli, si devono memorizzare i pid in ordine di indice per poi
 * abbinare i risultati delle wait. L'array di pid deve essere
 * allocato dinamicamente */
Nfigli = N;
Pids=malloc(Nfigli*sizeof(int));
if (Pids==NULL) {
    zprintf(2,"Impossibile allocare Pids\n");
    return(1);
}
for (i=0; i<Nfigli; i++) {
    Pids[i]=fork();
    switch (Pids[i]) {
        case 0: /* figlio di indice i */
            return(figlio(i)); /* chiamo funzione figlio
            e uso valore di ritorno come exit del processo*/
        case -1:

```

```

        zprintf(2,"Impossibile_forkare_il_processo_%d\n",i);
        return(1);
        break;
    }
#ifdef DEBUG
    /* solo il processo padre esegue queste linee */
    zprintf(1,"Forkato_processo_%d_per_indice_%d\n",Pids[i],i);
#endif
}
/* Il padre non utilizza nessuna pipe in quando la sincronizzazione
 * viene fatta totalmente fra i figli. Eventualmente (ma ci sono
 * anche altri modi) il padre puo' dare il via al primo figlio.
 * Utilizziamo questa possibilita' ed utilizziamo un singolo
 * carattere memorizzato in ok per l'invio */
if(write(Pfds[0][SCRIT],&ok,1)!=1) {
    /* controllo di scrittura avvenuta */
    zprintf(2,"Impossibile_inviare_su_%d\n",Pfds[0][SCRIT]);
    return(1);
}
/* ora chiudiamo tutte le pipe inutilizzate dal padre, compresa
 * quella in cui abbiamo appena scritto (il carattere rimane!) */
for (i=0; i<Npipe; i++) {
    close(Pfds[i][SCRIT]);
    close(Pfds[i][LETT]);
}
/* Recupero delle informazioni dai processi figli tramite la wait */
for (i=0; i<Nfigli; i++) {
    int pid_x;
    int status;

    pid_x = wait(&status);
    /* abbinamento delle informazioni all'indice del figlio */
    for (j=0; j<Nfigli; j++) {
        if (pid_x==Pids[j]) {
            if (WEXITSTATUS(status)==255) {
                /* convenzione: segnalazione d'errore */
                zprintf(1,
                    "Il_figlio_che_cerca_il_carattere_%c_ha_incontrato_un_errore\n",
                    C[j]);
            } else {
                zprintf(1,
                    "Il_figlio_che_cerca_il_carattere_%c_ha_trovato_%d_linee\n",
                    C[j],WEXITSTATUS(status));
            }
        }
    }
}
return(0);
}

/* Funzione figlio */
int figlio(int indice) {
    /* Ogni figlio deve cercare le occorrenze di C[i] nel file F,
     * quindi il file NON deve essere condiviso ma aperto indipendentemente

```

```

    * da ogni figlio */
int fd;
int linenum; /* contatore di linea corrente */
char ch; /* carattere di appoggio */
int retval; /* valore di ritorno */
int nr; /* numero di caratteri letti dalla read */
int flag; /* se 0 carattere non trovato, se 1 trovato */
int nrp; /* numero di car letti da read su pipe */
char dummy; /* carattere di appoggio per lett/scritt pipe */
int i; /* un contatore */
int pipe_di_sinc; /* indice della pipe di sincronizzazione da
cui fare la read */

/* Chiusura dei lati delle pipe per evitare il deadlock quando
* un figlio termina prima degli altri e non puo' proseguire
* nell'attivita' di sincronizzazione. Se almeno i lati
* di scrittura sono posseduti ognuno da un solo processo
* la terminazione di questo provoca un comportamento non
* bloccante per la read */
for (i=0; i<Npipe; i++) {
    /* chiudo i lati di scrittura se non e' la pipe
    * che usa questo figlio */
    if (indice!=Nfigli-1) {
        if (i!=(indice+1)) {
            close(Pfds[i][SCRIT]);
        }
    } else {
        if (i!=0) {
            close(Pfds[i][SCRIT]);
        }
    }
}

fd=open(F,ORDONLY);
if (fd<0) {
#ifdef DEBUG
    zprintf(2,"Impossibile aprire il file %s\n",F);
#endif
    return(-1);
}
/* il file deve essere letto per linee */
linenum=1; /* prima linea */
retval=0; /* linee che soddisfano=0 */
flag=0;
/* sincronizzazione 'estesa' in grado di assicurarci l'alternanza delle
* informazioni anche in seguito alla terminazione di uno o piu' figli
* si usa una variabile per memorizzare l'indice della pipe da cui
* attendere l'ok. Se la read da tale indice restituisce 0 -> allora il
* figlio relativo e' terminato e bisogna 'agganciarsi' al precedente.
* Nel caso estremo in cui un solo processo figlio rimanga in essere,
* la pipe di lettura e quella di scrittura vanno a coincidere
* sbloccando continuamente l'output di tale processo */
pipe_di_sinc = indice; /* iniziamo con quella 'canonica' */
for (;;) {

```

```

        nr=read(fd,&ch,1);
        if (nr==0) {
            return(retval);
        }
        if (nr<0) {
            /* errore */
#ifdef DEBUG
            zprintf(2,"Impossibile leggere dal file %s\n",F);
#endif
            return(-1);
        }
        /* carattere letto, controllo se fine linea */
        if (ch=='\n') {
            linenum++;
            flag=0;
        }
        /* dal testo: se trovo il carattere assegnato C[indice], devo riportare
         * il numero di linea su stdout. Se il carattere si trova piu'
         * volte nella stessa linea devo riportare il numero solo
         * una volta, quindi uso un flag che controllo a fine linea */
        if ((ch==C[indice])&&flag==0) {
            flag=1;
            retval++; /* incremento contatore */
            /* aggiungo sincronizzazione, attendo carattere da pipe
             * con indice pipe_di_sinc e gestisco l'eventuale
             * terminazione del figlio associato */
            for (;;) {
                nrp=read(Pfds[pipe_di_sinc][LETT],&dummy,1);
                if (nrp==0) {
                    /* pipe chiusa, agganciamoci al figlio precedente */
                    if (pipe_di_sinc>0) {
                        pipe_di_sinc--;
                    } else {
                        pipe_di_sinc=Npipe-1;
                    }
                }
                continue;
            }
            if (nrp==-1) {
                /* errore, segnalare */
#ifdef DEBUG
                zprintf(2,"Errore pipe di sincronizzazione\n");
#endif
                return(-1);
            }
            break;
        }
        /* se sono qui posso scrivere */
        zprintf(1,"Linea %d contiene %c\n",linenum,C[indice]);
        /* ora do il via al figlio successivo: l'indice della pipe
         * in cui scrivere e' indice+1 a meno che no si tratti dell'ultimo
         * figlio (quello con indice=(Nfigli-1)) che deve scrivere sulla
         * pipe di indice 0 */
        if (indice!=Nfigli-1) {
            write(Pfds[indice+1][SCRIT],&dummy,1);

```

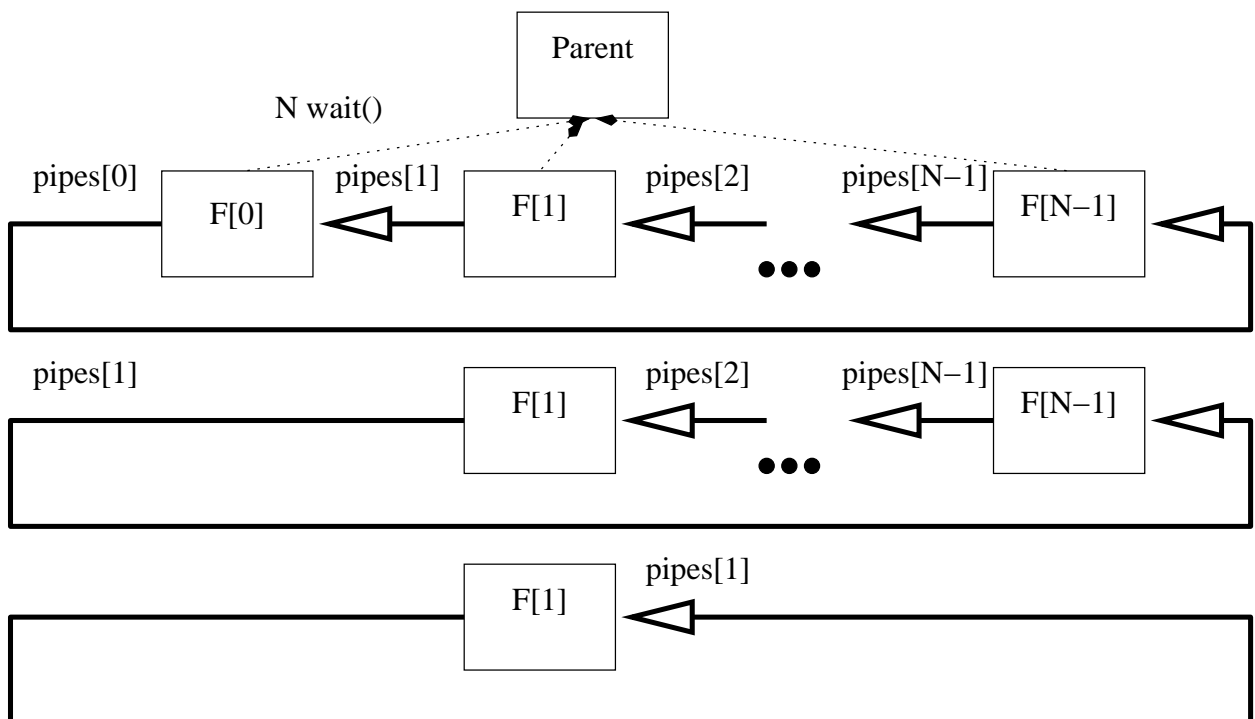
```

        } else {
            write(Pfds[0][SCRIT], &dummy, 1);
        }
    }
}
/* mai qui! */
return (-1);
}
void zprintf(int fd, const char *fmt, ...) {
    /* printf wrapper using write instead */
    static char msg[256];
    va_list ap;
    int n;
    va_start(ap, fmt);
    n = vsnprintf(msg, 256, fmt, ap);
    write(fd, msg, n);
    va_end(ap);
}

```

8. Invertire l'ordine di sincronizzazione dei figli nell'esercizio es96.c (es97.c)

**Soluzione:**



```

/* file: es97.c
 * job: inversione d'ordine
 */

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

```

```

#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdarg.h>

typedef int pipe_t[2]; /* tipo per pipe */
#define SCRIT 1
#define LETT 0

char *F; /* Nome del file , variabile globale */
int N; /* Numero di caratteri */
char *C; /* Elenco dei caratteri , allocazione dinamica */
int Npipe; /* Numero di pipe da creare */
pipe_t *Pfds; /* Array di pipe , allocazione dinamica */
int Nfigli; /* Numero di figli da forkare */
int *Pids; /* Array di pid , allocazione dinamica */

/* abbinamento snprintf+write */
void zprintf(int fd, const char *fmt, ...);
/* funzione figlio */
int figlio(int);

char ok='Q'; /* carattere (casuale) che inviamo sulla pipe
             per la sincronizzazione */

int main(int argc, char **argv, char **envp) {
    int i,j; /* qualche contatore */

    /* Determinazione dei dati dalla command line */
    switch (argc) {
        case 0:
        case 1:
        case 2:
            zprintf(2," Inserire _il _nome _di _file _ed _almeno _un _carattere\n");
            return(1);
    }
    F = argv[1];
    N = argc-2; /* argc vale 1 se non ci sono argomenti */
    C = malloc(N*sizeof(char));
    if (C==NULL) {
        zprintf(2," Impossibile _allocare _C\n");
        return(1);
    }
    for (i=0; i<N; i++) {
        C[i]=argv[i+2][0]; /* prendo il primo carattere della stringa
                           argv[i+2] */
    }
#ifdef DEBUG
    zprintf(1,"Nome _del _file : %s\n",F);
    zprintf(1,"Numero _di _caratteri %d\n",N);
    for (i=0; i<N; i++) {
        zprintf(1," Carattere _di _indice %d _= %c\n",i,C[i]);
    }
#endif
}

```

```

/* Predisposizione del sistema di sincronizzazione
 * fra i processi figli: si utilizza una pipe per
 * collegare ogni figlio al successivo; la sincronizzazione
 * del figlio di indice 'k' si ottiene attendendo un
 * carattere dalla pipe che lo collega al figlio di
 * indice 'k-1'. */
Npipe = N; /* una pipe in uscita da ogni figlio */
Pfds = malloc(Npipe*sizeof(pipe_t));
if (Pfds==NULL) {
    zprintf(2,"Impossibile_allocare_Pfds\n");
    return(1);
}
for (i=0; i<Npipe; i++) {
    if (pipe(Pfds[i])!=0) {
        zprintf(2,"Impossibile_creare_la_pipe_%d\n",i);
        return(1);
    }
}
/* Creazione processi figli:
 * dal testo, bisogna creare un processo figlio per ogni carattere
 * quindi Nfigli=N
 * Visto che il padre deve stampare i valori di uscita dei processi
 * figli, si devono memorizzare i pid in ordine di indice per poi
 * abbinare i risultati delle wait. L'array di pid deve essere
 * allocato dinamicamente */
Nfigli = N;
Pids=malloc(Nfigli*sizeof(int));
if (Pids==NULL) {
    zprintf(2,"Impossibile_allocare_Pids\n");
    return(1);
}
for (i=0; i<Nfigli; i++) {
    Pids[i]=fork();
    switch (Pids[i]) {
        case 0: /* figlio di indice i */
            return(figlio(i)); /* chiamo funzione figlio
            e uso valore di ritorno come exit del processo*/
        case -1:
            zprintf(2,"Impossibile_forkare_il_processo_%d\n",i);
            return(1);
            break;
    }
}
#ifdef DEBUG
    /* solo il processo padre esegue queste linee */
    zprintf(1,"Forkato_processo_%d_per_indice_%d\n",Pids[i],i);
#endif
}
/* Il padre non utilizza nessuna pipe in quando la sincronizzazione
 * viene fatta totalmente fra i figli. Eventualmente (ma ci sono
 * anche altri modi) il padre puo' dare il via al primo figlio.
 * Utilizziamo questa possibilita' ed utilizziamo un singolo
 * carattere memorizzato in ok per l'invio */
if(write(Pfds[0][SCRIT],&ok,1)!=1) {
    /* controllo di scrittura avvenuta */

```

```

        zprintf(2, "Impossibile_inviare_su_%d\n", Pfds[0][SCRIT]);
        return(1);
    }
    /* ora chiudiamo tutte le pipe inutilizzate dal padre, compresa
     * quella in cui abbiamo appena scritto (il carattere rimane!) */
    for (i=0; i<Npipe; i++) {
        close(Pfds[i][SCRIT]);
        close(Pfds[i][LETT]);
    }
    /* Recupero delle informazioni dai processi figli tramite la wait */
    for (i=0; i<Nfigli; i++) {
        int pid_x;
        int status;

        pid_x = wait(&status);
        /* abbinamento delle informazioni all'indice del figlio */
        for (j=0; j<Nfigli; j++) {
            if (pid_x==Pids[j]) {
                if (WEXITSTATUS(status)==255) {
                    /* convenzione: segnalazione d'errore */
                    zprintf(1,
                        "Il_figlio_che_cerca_il_carattere_%c_ha_incontrato_un_errore\n",
                        C[j]);
                } else {
                    zprintf(1,
                        "Il_figlio_che_cerca_il_carattere_%c_ha_trovato_%d_linee\n",
                        C[j], WEXITSTATUS(status));
                }
            }
        }
    }
}
return(0);
}

/* Funzione figlio */
int figlio(int indice) {
    /* Ogni figlio deve cercare le occorrenze di C[i] nel file F,
     * quindi il file NON deve essere condiviso ma aperto indipendentemente
     * da ogni figlio */
    int fd;
    int linenum; /* contatore di linea corrente */
    char ch; /* carattere di appoggio */
    int retval; /* valore di ritorno */
    int nr; /* numero di caratteri letti dalla read */
    int flag; /* se 0 carattere non trovato, se 1 trovato */
    int nrp; /* numero di car letti da read su pipe */
    char dummy; /* carattere di appoggio per lett/scritt pipe */
    int i; /* un contatore */
    int pipe_di_sinc; /* indice della pipe di sincronizzazione da
     * cui fare la read */

    /* Chiusura dei lati delle pipe per evitare il deadlock quando
     * un figlio termina prima degli altri e non puo' proseguire
     * nell'attivita' di sincronizzazione. Se almeno i lati

```

```

    * di scrittura sono posseduti ognuno da un solo processo
    * la terminazione di questo provoca un comportamento non
    * bloccante per la read */
for (i=0; i<Npipe; i++) {
    /* chiudo i lati di scrittura se non e' la pipe
    * che usa questo figlio */
    if (i!=(indice)) {
        close(Pfds[i][SCRIT]);
    }
}

fd=open(F,ORDONLY);
if (fd<0) {
#ifdef DEBUG
    zprintf(2,"Impossibile_aprire_il_file_%s\n",F);
#endif
return(-1);
}
/* il file deve essere letto per linee */
linenum=1; /* prima linea */
retval=0; /* linee che soddisfano=0 */
flag=0;
/* sincronizzazione 'estesa' in grado di assicurarci l'alternanza delle
* informazioni anche in seguito alla terminazione di uno o piu' figli
* si usa una variabile per memorizzare l'indice della pipe da cui
* attendere l'ok. Se la read da tale indice restituisce 0 -> allora il
* figlio relativo e' terminato e bisogna 'agganciarsi' al precedente.
* Nel caso estremo in cui un solo processo figlio rimanga in essere,
* la pipe di lettura e quella di scrittura vanno a coincidere
* sbloccando continuamente l'output di tale processo */
pipe_di_sinc = (indice+1)%Npipe; /* iniziamo con quella 'canonica' */
for (;;) {
    nr=read(fd,&ch,1);
    if (nr==0) {
        return(retval);
    }
    if (nr<0) {
        /* errore */
#ifdef DEBUG
        zprintf(2,"Impossibile_leggere_dal_file_%s\n",F);
#endif
return(-1);
    }
    /* carattere letto, controllo se fine linea */
    if (ch=='\n') {
        linenum++;
        flag=0;
    }
    /* dal testo: se trovo il carattere assegnato C[indice], devo riportare
    * il numero di linea su stdout. Se il carattere si trova piu'
    * volte nella stessa linea devo riportare il numero solo
    * una volta, quindi uso un flag che controllo a fine linea */
    if ((ch==C[indice])&&flag==0) {
        flag=1;
    }
}

```

```

retval++; /* incremento contatore */
/* aggiungo sincronizzazione, attendo carattere da pipe
 * con indice pipe_di_sinc e gestisco l'eventuale
 * terminazione del figlio associato */
for (;;) {
    nrp=read(Pfds[ pipe_di_sinc ][LETT],&dummy,1);
    if (nrp==0) {
        /* pipe chiusa, agganciamoci al figlio successivo */
        if (pipe_di_sinc<(Npipe-1)) {
            pipe_di_sinc++;
        } else {
            pipe_di_sinc=0;
        }
        continue;
    }
    if (nrp==-1) {
        /* errore, segnalare */
#ifdef DEBUG
        zprintf(2,"Errore_pipe_di_sincronizzazione\n");
#endif
        return(-1);
    }
    break;
}
/* se sono qui posso scrivere */
zprintf(1,"Linea_%d_contiene_%c\n",linenum,C[indice]);
/* ora do il via al figlio successivo: l'indice della pipe
 * in cui scrivere e' indice+1 a meno che no si tratti dell'ultimo
 * figlio (quello con indice=(Nfigli-1)) che deve scrivere sulla
 * pipe di indice 0 */
if (indice!=Nfigli-1) {
    write(Pfds[indice+1][SCRIT],&dummy,1);
} else {
    write(Pfds[0][SCRIT],&dummy,1);
}
}
}
/* mai qui! */
return (-1);
}
void zprintf(int fd, const char *fmt, ...) {
    /* printf wrapper using write instead */
    static char msg[256];
    va_list ap;
    int n;
    va_start(ap, fmt);
    n=vsprintf(msg, 256, fmt, ap);
    write(fd,msg,n);
    va_end(ap);
}

```

9. Evidenziare le differenze fra i due file

**Soluzione:**

```

1,4c1,3
< /* Possibile soluzione della prova in itinere del 12 marzo 2004
< * analizzata nelle lezioni di Laboratorio di Sistemi Operativi
< * del 14 e 15 marzo 2005 */
< /* $Id: d96-97.diff,v 1.1 2009/03/09 07:30:21 valealex Exp $ */
-----
> /* file: es97.c
> * job: inversione d'ordine
> */
178,185c177,178
<     if (indice!=Nfigli-1) {
<         if (i!=(indice+1)) {
<             close(Pfds[i][SCRIT]);
<         }
<     } else {
<         if (i!=0) {
<             close(Pfds[i][SCRIT]);
<         }
-----
>     if (i!=(indice)) {
>         close(Pfds[i][SCRIT]);
208c201
<     pipe_di_sync = indice; /* iniziamo con quella 'canonica' */
-----
>     pipe_di_sync = (indice+1)%Npipe; /* iniziamo con quella 'canonica' */
239,241c232,234
<                                     /* pipe chiusa, agganciamoci al figlio precedente */
<                                     if (pipe_di_sync>0) {
<                                         pipe_di_sync--;
-----
>                                     /* pipe chiusa, agganciamoci al figlio successivo */
>                                     if (pipe_di_sync<(Npipe-1)) {
>                                         pipe_di_sync++;
243c236
<                                     pipe_di_sync=Npipe-1;
-----
>                                     pipe_di_sync=0;

```