Esercitazione di Venerdì 4 Maggio 2018

1. (provaPipe.c) Scrivere un programma C che a) apra un file (lo studente può decidere in autonomia quale file aprire) e riporti su standard output il file descriptor assegnato; b) apra un file (il file può anche essere lo stesso del caso a)) e riporti su standard output il file descriptor assegnato; chiuda il file del punto a) e quindi crei una pipe e riporti su standard output i file descriptor assegnati alla pipe.

```
soELab@Lica02:~/pipe/terzaEsercitazione$ cat provaPipe.c
/* FILE: provaPipe.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main (int argc, char** argv)
{
        int fd1, fd2;   /* file descriptor per le due open */
        int piped[2];   /* array di due interi per la pipe */

if ((fd1= open(argv[0], O_RDONLY)) < 0)          /* si fa una open del
programma stesso, tanto questa open serve solo per occupare un posto nella
TFA (Tabella dei File Aperti) del processo corrente */
        {
                printf("Errore nella prima open\n");
                exit(1);
        }

printf("aperto file con fd1=%d \n", fd1);

if ((fd2= open(argv[0], O_RDONLY)) < 0)         /* idem come sopra */
        {
                printf("Errore nella seconda open\n");
                exit(2);
        }

printf("aperto file con fd2=%d \n", fd2);
close(fd1);     /* chiudiamo fd1 e quindi liberiamo un posto nella TFA */

/* si crea una pipe */
if (pipe (piped) < 0 )
        {
                printf("Errore nella creazione pipe\n");
                exit (3);
        }
printf("creato pipe con piped[0]= %d \n", piped[0]);
printf("creato pipe con piped[1]= %d \n", piped[1]);
exit (0);
}
```

2. (pipe-newGenerico.c e pipe-newGenerico1.c) Partendo dall'esercizio pipe-new.c mostrato a lezione, cambiare il protocollo di comunicazione per considerare la possibilità di scambiare linee/stringhe di lunghezza generica: 1) in pipe-newGenerico.c il figlio manda per prima cosa la lunghezza della linea/stringa al padre e quindi il padre usa questa informazione per leggere solo il numero di caratteri che costituiscono la linea/stringa; 2) in pipe-newGenerico1.c il processo figlio

manda linee/stringhe di lunghezza qualunque al padre e il padre le riceve carattere per carattere fino a terminatore di linea/stringa.

```
soELab@Lica02:~/pipe/terzaEsercitazione$ cat pipe-newGenerico.c
/* FILE: pipe-newGenerico.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main (int argc, char **argv)
{       int pid, j, piped[2];
        char mess[512];          /* array che serve per salvare la linea
letta dal figlio, supposta non piu' lunga di 512 caratteri compreso il
terminatore di linea */
        int L;                    /* indice per la lettura di un singolo
carattere da file */
        char inpbuf [512];     /* array usato dal padre; NOTA BENE: poteva
essere usato sempre l'array usato dal figlio cioe' mess */
        int Lun;                 /* lunghezza letta dal padre dalla pipe;
NOTA BENE: poteva essere usata sempre la variabile usata dal figlio cioe'
L */
        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 )  { exit (2); }
if ((pid = fork()) < 0)  { exit (3); }
else 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(4);
}

printf("Figlio %d sta per iniziare a scrivere una serie di messaggi, di
lunghezza non nota, sulla pipe dopo averli letti dal file passato come
parametro\n", getpid());
/* con un ciclo leggiamo tutte le linee e ne calcoliamo la lunghezza */
L=0; /* valore iniziale dell'indice */
j=0; /* il figlio inizializza la sua variabile j per contare i messaggi
che ha mandato al padre */
while(read(fd,&(mess[L]),1) != 0)
{
        if (mess[L] == '\n') /* siamo arrivati alla fine di una 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[L]='\0';
```

```
                L++; /* incrementiamo L per tenere conto anche del
terminatore di linea */
                /* comunichiamo L al processo padre */
                write(piped[1],&L,sizeof(L));
                /* comunichiamo la stringa corrispondente alla linea al
processo padre */
                write(piped[1],mess, L);
                L = 0;  /* azzeriamo l'indice per la prossima linea */
                j++;        /*  se  troviamo  un  terminatore  di  linea
incrementiamo il conteggio */
        }
        else L++;
}

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], &Lun, sizeof(Lun)))
    {
        /* ricevuta la lunghezza, il padre puo' andare a leggere la
linea/stringa */
        read(piped[0], inpbuf, Lun);
        /* 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\n",  pidFiglio,
ritorno);
}
exit (0);
}
```

```
soELab@Lica02:~/pipe/terzaEsercitazione$ cat pipe-newGenerico1.c
/* FILE: pipe-newGenerico1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main (int argc, char **argv)
{       int pid, j, piped[2];
        char mess[512];          /* array che serve per salvare la linea
letta dal figlio, supposta non piu' lunga di 512 caratteri compreso il
terminatore di linea */
        int L;                    /* indice per la lettura di un singolo
carattere da file */
        char inpbuf [512];     /* array usato dal padre; NOTA BENE: poteva
essere usato sempre l'array usato dal figlio cioe' mess */
        int Lun;                /* lunghezza letta dal padre dalla pipe;
NOTA BENE: poteva essere usata sempre la variabile usata dal figlio cioe'
L */
        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 )  { exit (2); }
if ((pid = fork()) < 0)  { exit (3); }
else 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(4);
}

printf("Figlio %d sta per iniziare a scrivere una serie di messaggi, di
lunghezza non nota, sulla pipe dopo averli letti dal file passato come
parametro\n", getpid());
/* con un ciclo leggiamo tutte le linee e ne calcoliamo la lunghezza */
L=0; /* valore iniziale dell'indice */
j=0; /* il figlio inizializza la sua variabile j per contare i messaggi
che ha mandato al padre */
while(read(fd,&(mess[L]),1) != 0)
{
        if (mess[L] == '\n') /* siamo arrivati alla fine di una 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[L]='\0';
                L++;  /*  incrementiamo  L  per  tenere  conto  anche  del
terminatore di linea */
```

```
                /* comunichiamo la stringa corrispondente alla linea al
processo padre */
                write(piped[1],mess, L);
                L = 0;  /* azzeriamo l'indice per la prossima linea */
                j++;        /*  se  troviamo  un  terminatore  di  linea
incrementiamo il conteggio */
        }
        else L++;
}

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());
Lun=0; /* valore iniziale dell'indice */
j=0; /* il padre inizializza la sua variabile j per verificare quanti
messaggi ha mandato il figlio */
while (read(piped[0], &(inpbuf[Lun]), 1))
{
        if (inpbuf[Lun] == '\0') /* siamo arrivati alla fine di una stringa
*/
        {
                /* 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++;
                Lun = 0;  /* azzeriamo l'indice per la prossima linea */
        }
        else Lun++;
}
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\n",  pidFiglio,
ritorno);
}
exit (0);
}
```

3. (partec190201.c) Prova di esame del 19 Febbraio 2001 concentrandoci sul testo della sola parte C: La parte in C accetta un numero variabile di parametri che rappresentano nomi assoluti di file F1, F2 .. FN. Il processo padre deve generare N processi figli, ognuno dei quali associato ad un file. I processi figli eseguono concorrentemente leggendo dal file associato e comunicando al padre una

selezione dei caratteri del file associato: i processi associati ai file dispari (F1, F3, ...) devono selezionare solo i caratteri alfabetici, mentre quelli associati ai file pari (F2, F4, ...) solo i caratteri numerici. Il padre deve scrivere i caratteri ricevuti dai figli su standard output, alternando un carattere alfabetico e uno numerico. Una volta ricevuti tutti i caratteri, il padre deve stampare su standard output il numero totale di caratteri scritti. Al termine, il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

```
soELab@Lica02:~/19feb01$ cat partec190201.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <ctype.h> /* per usare isdigit() e isalpha() */

int main(int argc, char *argv[])
{
  int p[2][2];  /* p e' un array di due pipe: l'elemento di posto 0 (p[0]) serve per la comunicazione fra i processi pari e il padre, mentre l'elemento di posto 1 (p[1]) serve per la comunicazione fra i processi dispari e il padre */
  int N;        /* N e' il numero di processi da creare (uno per ogni file passato) */
  int pid;      /* pid variabile usata per il valore di ritorno della fork */
  int fdr;      /* fdr file descriptor usato da ogni figlio per aprire in lettura il proprio file */
  int i;        /* indice */
  int tot=0;    /* tot variabile usata dal padre per contare i caratteri scritti su standard output */
  int nr0, nr1; /* nr0 numero di caratteri letti dalla pipe 'pari' e nr1 numero di caratteri letti dalla pipe 'dispari' */
  char ch, ch0, ch1; /* ch e' il carattere letto da ogni file; ch0 e' il carattere letto dal padre sulla pipe 'pari' e ch1 e' il carattere letto dal padre su quella 'dispari' */
  int pidFiglio, status, ritorno;    /* variabili per wait*/

  if (argc < 3)          /* non e' specificato, ma l'esercizio ha senso solo se sono passati almento due nomi di file! */
  { printf("Errore nel numero dei parametri: %d\n", argc);
    exit(1);
  }

N=argc-1;
/* creazione delle due pipe: quella 'pari' e quella 'dispari' */
  if (pipe(p[0]) < 0)
/* pipe per i caratteri numerici */
  { printf("Errore nella creazione della prima pipe\n");
    exit(2);
  }
  if (pipe(p[1]) < 0)
/* pipe per i caratteri alfabetici */
  { printf("Errore nella creazione della seconda pipe\n");
    exit(3);
  }

  for (i=0; i < N; i++)
```

```
  {
/* si creano N figli con N uguale ad argc -1  */
    if ((pid = fork()) < 0)
    { printf ("Errore nella fork\n");
      exit(4);
    }

    if (pid == 0) /* ogni figlio */
    {
      close(p[0][0]); /* chiude i lati di lettura di entrambe le pipe */
      close(p[1][0]);
      close(p[i % 2][1]); /* chiude il lato di scrittura della pipe non
usata */
      if ((fdr = open(argv[i+1], O_RDONLY)) < 0) /* il processo di indice
i deve aprire il file di indice i+1 */
/* apertura del file associato */
      { printf("Errore nell'apertura del file %s\n", argv[i+1]);
        exit(4);
      }
      while(read(fdr, &ch, 1) > 0)
      {
        if ((((i % 2) == 0) && isalpha(ch)) ||  /* il file di indice
dispari e' letto in realta' da un processo di indice pari e quindi si
controlla che il carattere letto sia un alfabetico */
/* il processo pari controlla che il carattere letto sia un numerico */
            (((i % 2) == 1) && isdigit(ch)))             /* il file di
indice pari e' letto in realta' da un processo di indice dispari e quindi
si controlla che il carattere letto sia un numerico */
                write(p[(i+1) % 2][1], &ch, 1); /* comunica il carattere
al padre */
      }
      exit(0); /* non viene specificato di tornare un valore particolare
e quindi si torna 0 intendendo successo, per la semantica UNIX! */
      /* NOTA BENE: questa exit e' importantissima altrimenti ogni processo
figlio rimarrebbe nel ciclo di creazione e a sua volta creerebbe dei figli
che a loro volta creerebbero dei figli */
    }
  }
  /* padre */
  close(p[0][1]); /* chiude i lati di scrittura */
  close(p[1][1]);
/* dato che il testo non lo specifica si decide di continuare a leggere
dalle due pipe anche se non si riesce a garantire l'alternanza in output
a causa del termine dei processi pari o dei processi dispari; appena per
finiscono entrambi i 'tipi' di processi il padre smette di leggere dalle
pipe */
  nr0 = read(p[0][0], &ch0, 1);
  nr1 = read(p[1][0], &ch1, 1);
  while ((nr0 != 0) || (nr1 != 0))
  {
    tot = tot + nr0 + nr1; /* il totale viene ricavato sommando via via i
valori di ritorno delle read da pipe, ricordando che uno dei due puo' anche
essere 0 */
    write(1, &ch1, nr1); /* ATTENZIONE ALL'ORDINE: prima un alfabetico e
poi un numerico (secondo la specifica) */
    write(1, &ch0, nr0);
    nr0 = read(p[0][0], &ch0, 1);
```

```
    nr1 = read(p[1][0], &ch1, 1);
  }
  printf("\nNumero di caratteri scritti: %d\n", tot); /* stampa finale del
numero di caratteri scritti */
  /* Attesa della terminazione dei figli */
  for(i=0;i<N;i++)
  {
   pidFiglio = wait(&status);
   if (pidFiglio < 0)
   {
      printf("Errore wait\n");
      exit(9);
   }
   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);
    }
  }
  exit(0);
}
```

4. (partec191201-a.c e partec191201-b.c) Prova di esame del 19 Dicembre 2001 concentrandoci sul testo della sola parte C: La parte in C accetta un numero variabile di parametri (almeno 2) che rappresentano un nome di file F e singoli caratteri C1 ...CN. Il processo padre deve creare un numero di figli pari al numero di caratteri passati come parametri. Ogni processo figlio esegue in modo concorrente ed esamina il file F contando le occorrenze del carattere assegnato Cx: terminato il file, ogni figlio **comunica** al padre quante occorrenze di Cx sono presenti e termina. Il padre deve riportare sullo standard output il numero totale delle occorrenze specificando a quale carattere Cx si riferisce il conteggio. Al termine, il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.
   Provare a realizzare due soluzioni: 1) partec191201-a.c che faccia uso di N pipe (una per ogni processo figlio su cui viene scritto il numero di occorrenze del carattere Cx assegnato: in tale modo, il padre possa individuare il numero inviato a quale carattere si riferisce);

```
soELab@Lica02:~/19dic01so$ cat partec191201-a.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

/* versione con N pipe */

/* tipo che definisce una pipe */
typedef int pipe_t[2];

int main(int argc, char *argv[])
{
  int pid;      /* pid per fork */
  int nchar;    /* numero di caratteri e quindi numero di processi */
  int fdr;      /* per open */
  int i, k;     /* indici */
  int cont;     /* per conteggio */
```

```
char c;       /* per leggere dal file */
pipe_t *p;    /* array dinamico di pipe */
int pidFiglio, status, ritorno;    /* variabili per wait*/

if (argc < 3)
{
  printf("Errore nel numero dei parametri\n");
  exit(1);
}

/* numero dei caratteri passati sulla linea di comando */
nchar = argc - 2;

/* alloco l'array di pipe */
p = (pipe_t *)malloc(sizeof(pipe_t) * nchar);
if (p == NULL)
{ printf("Errore nella creazione array dinamico\n");
    exit(1);
}

/* creo nchar pipe */
for (i=0; i<nchar; i++)
  if (pipe(p[i]) < 0)
  { printf("Errore nella creazione della pipe\n");
    exit(1);
  }

printf("Padre con pid %d\n", getpid());

for (i=0; i<nchar; i++)
{
  /* creazione dei figli */
  if ((pid = fork()) < 0)
  {
    printf ("Errore nella fork\n");
    exit(1);
  }

  if (pid == 0) /* figlio */
  {
    printf("Figlio %d con pid %d\n", i, getpid());

    /* chiude tutte le pipe che non usa */
    for (k = 0; k < nchar; k++)
    {
      close(p[k][0]);
      if (k != i)
        close(p[k][1]);
    }

    /* apre il file */
    if ((fdr = open(argv[1], O_RDONLY)) < 0)
    { printf("Errore nella apertura del file %s\n", argv[1]);
      exit(1);
    }
    cont = 0;
    /* conta le occorrenze del carattere assegnato */
```

```
      while(read(fdr, &c, 1) > 0)
      {
        if (c == argv[i+2][0])
          cont++;
      }
      /* comunica al padre usando la i-esima pipe */
      write(p[i][1], &cont, sizeof(cont));
      exit(0);
    }
  }

  /* padre */
  /* chiude tutte le pipe che non usa */
  for (k=0; k<nchar; k++)
  {
    close(p[k][1]);
  }
  /* legge dalle pipe i messaggi */
  for (i=0; i<nchar; i++)
  {
    if (read(p[i][0], &cont, sizeof(int)) > 0)
      printf("%d occorrenze del carattere %c nel file %s\n", cont,
argv[i+2][0], argv[1]);
  }
  /* Attesa della terminazione dei figli */
  for(i=0;i<nchar;i++)
  {
   pidFiglio = wait(&status);
   if (pidFiglio < 0)
   {
      printf("Errore wait\n");
      exit(9);
   }
   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);
   }
  }
  exit(0);
}
```

2) partec191201-b.c che faccia uso di una sola pipe su cui ogni figlio scrive una struct con due campi (carattere e numero occorrenze) dato che il testo non richiede che le informazioni inviate dal figlio vengano recuperate dal padre in un ordine specifico.

```
soELab@Lica02:~/19dic01so$ cat partec191201-b.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
/* versione con 1 pipe e uso di una struttura */

int main(int argc, char *argv[])
{
  int pid;      /* pid per fork */
  int nchar;    /* numero di caratteri e quindi numero di processi */
  int fdr;      /* per open */
  int i;        /* indice */
  char c;       /* per leggere dal file */
  int p[2];     /* singola pipe */
  int pidFiglio, status, ritorno;    /* variabili per wait*/
  /* struttura per la comunicazione tra figli e padre */
  struct {
    int n; /* numero di occorrenze del carattere */
    char c; /* carattere controllato */
  } msg;

  if (argc < 3)
  {
    printf("Errore nel numero dei parametri\n");
    exit(1);
  }

  /* numero dei caratteri passati sulla linea di comando */
  nchar = argc - 2;

  /* creo la pipe */
  if (pipe(p) < 0)
  { printf("Errore nella creazione della pipe\n");
    exit(1);
  }

  printf("Padre con pid %d\n", getpid());

  for (i=0; i<nchar; i++)
  {
    /* creazione dei figli */
    if ((pid = fork()) < 0)
    {
      printf ("Errore nella fork\n");
      exit(1);
    }

    if (pid == 0) /* figlio */
    {
      printf("Figlio %d con pid %d\n", i, getpid());

      /* chiude il lato della pipe che non usa */
      close(p[0]);

      /* apre il file */
      if ((fdr = open(argv[1], O_RDONLY)) < 0)
      { printf("Errore nella apertura del file %s\n", argv[1]);
        exit(1);
      }
      /* inizializza la struttura */
      msg.c = argv[i+2][0];
```

```
      msg.n = 0;
      /* conta le occorrenze del carattere assegnato */
      while(read(fdr, &c, 1) > 0)
      {
        if (c == argv[i+2][0])
          msg.n++;
      }
      /* comunica al padre */
      write(p[1], &msg, sizeof(msg));
      exit(0);
    }
  }

  /* padre */
  /* chiude il lato della pipe che non usa */
  close(p[1]);
  /* legge dalla pipe i messaggi */
  while (read(p[0], &msg, sizeof(msg)) > 0)
    printf("%d occorrenze del carattere %c nel file %s\n", msg.n, msg.c,
argv[1]);
  /* Attesa della terminazione dei figli */
  for(i=0;i<nchar;i++)
  {
   pidFiglio = wait(&status);
   if (pidFiglio < 0)
   {
      printf("Errore wait\n");
      exit(9);
   }
   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);
   }
  }
  exit(0);
}
```