Autore: Simone Cipriani, matricola 5951907
sim.cipr@gmail.com, s.cipriani@aem-progetti.com

Data di consegna: 13/06/2017

Repository progetto: https://github.com/gesucca/os-class-homework

ESERCIZIO 1: scheduler di processi

Descrizione dell'implementazione

File globals.h

```
#include <stdlib.h>

#define LIST_INIT_LEN 5
#define LIST_ENLARGE_STEP 5

typedef struct {
        int i_id;
        int i_priority;
        char c_name[8];
        int i_cycles;
} task_t;

task_t* task_list;
size_t task_size;
int i_num_task;

int i_assign_id;

#define PRIORITY 1
#define LJF 0
int i_policy;
```

Questo header contiene la dichiarazione delle strutture globali usate in varie parti del programma.

L'inclusione alla libreria standard di C avviene in questo file, in quanto le funzioni in essa contenute saranno utili in molti altro sorgenti.

Il concetto di *processo* è stato rappresentato con una struttura contente gli attributi caratteristici, mentre l'insieme dei processi attivi nel sistema è mantenuto in una lista di tasks allocata dinamicamente nello heap.

In questo header vengono inoltre impartite direttive al preprocessore per quanto riguarda la policy di scheduling (rappresentata con un valore numerico intero) e i valori predefiniti per la modifica della memoria allocata alla lista di processi.

File cmd.h

```
int insertTask(int i_priority, char c_name[], int i_exec);
int execTopTask();
int execTask(int i_id);
int deleteTask(int i_id);
int alterPriority(int i_id, int i_new_priority);
int switchSchedulingPolicy();
```

File io.h

```
void printProgHead();
void printCmds();
void printTaskList();

int dispatchCMD();
```

File list.h

```
int initTaskList(int i_len);
int alterListSize(int i_len);
int releaseEmptySpace();

void sortList();
```

Questi tre headers contengono le dichiarazioni delle funzioni implementate nei file sorgenti e usate nel corso dell'esecuzione del programma.

File main.c

```c
#include "headers/globals.h"
#include "headers/list.h"
#include "headers/io.h"

int main() {

        printProgHead();

        // init list
        int i_ret_code;
        i_ret_code = initTaskList(LIST_INIT_LEN);
        if (i_ret_code!=EXIT_SUCCESS)
                return i_ret_code;

        //default policy is priority
        i_policy = PRIORITY;

        //main program loop
        while(1) {
                printCmds();
                i_ret_code = dispatchCMD();
                if (i_ret_code!=EXIT_SUCCESS)
                        return i_ret_code;
                sortList();
                printTaskList();
        }

        //should never reach here, but in case..
        return EXIT_FAILURE;
};
```

Questo file sorgente contiene il metodo main.

Esso comincia stampando nello standard output l'intestazione del programma ed inizializza la lista dei processi chiamando delle apposite funzioni. Viene inoltre impostato un valore di default per la politica di scheduling utilizzata.

Successivamente, il metodo main si occupa di gestire il loop principale del programma, nel quale vengono eseguite in un ciclo infinito le seguenti operazioni: stampa dei comandi, esecuzione del comando inserito, ordinamento della lista di processi secondo la politica di scheduling impostata ed infine stampa dei processi immessi nel sistema.

Non è prevista uscita dal loop perché si prevede che l'utente termini il programma immettendo l'apposito comando o inviando un SIGINT al processo tramite CTR-C (oppure direttamente killando il processo, come non poche volte è stato necessario fare in fase di sviluppo).

File list.c

```
#include "headers/globals.h"

int initTaskList(int i_len)
{
        i_assign_id = 1;
        i_num_task = 0;
        task_size = i_len;

        task_list = malloc(i_len * sizeof(task_t));
        if (task_list)
                return EXIT_SUCCESS;
        else
                return EXIT_FAILURE;
}

int alterListSize(int i_len)
{
        task_size += i_len;
        task_t* realloc_task = realloc(task_list, task_size * sizeof(task_t));
        if (realloc_task)
        {
                free(task_list);
                task_list = realloc_task;
                return EXIT_SUCCESS;
        }
        else
                return EXIT_FAILURE;
}

int releaseEmptySpace()
{
        task_t* clean_task_list = malloc( (i_num_task +1) * sizeof(task_t));

        if (!task_list)
                return EXIT_FAILURE;

        int i;
        for (i=0; i<task_size; i++)
                clean_task_list[i] = task_list[i];

        free(task_list);
        task_list = clean_task_list;
        task_size = i_num_task + 1;

        return EXIT_SUCCESS;
}

int compare(task_t a, task_t b)
{
        if (i_policy==PRIORITY)
        {
                if (a.i_priority==b.i_priority)
                        return (a.i_id<b.i_id);
                else
                        return (a.i_priority<b.i_priority);
```

```
        }

        if (i_policy==LJF)
        {
                if (a.i_cycles==b.i_cycles)
                        return (a.i_id<b.i_id);
                else
                        return (a.i_cycles>b.i_cycles);
        }

        //default case, should never happen
        return 0;
}

//bubble sort, terrible in performances but easy to implement
void sortList()
{
        int i,k;
        task_t temp;
        for(i = 0; i<i_num_task-1; i++)
        {
                for(k = 0; k<i_num_task-1-i; k++)
                {
                        if (compare(task_list[k],task_list[k+1]))
                        {
                                temp = task_list[k];
                                task_list[k] = task_list[k+1];
                                task_list[k+1] = temp;
                        }
                }
        }
}
```

In questo modulo del programma sono definite le implementazioni delle funzioni che si occupano di gestire la lista di processi immessi.

La funzione *initTaskList* si occupa di allocare memoria sufficiente a contenere una lista di processi cntenente un determinato numero di elementi (fornito alla funzione come parametro) utilizzando la funzione di libreria standard *malloc.*

La funzione *alterListSize* invece viene invocata qualora la memoria allocata per la lista dei processi non dovesse più essere sufficiente. Essa usa la funzione di libreria standard *realloc* per riallocare la lista in una nuova area di memoria più grande.

Si noti che la funzione permette di essere invocata con argomento negativo, casando di fatto un riallocamento della lista in un'area di memoria più piccola. Tuttavia questa possiblità non è stata poi utilizzata nel flusso di esecuzione del programma, preferendo invece scrivere una apposita funzione, come spiegato sotto.

La funzione *releaseEmptySpace* serve a liberare la memoria occupata dalla lista qualora il numero degli elementi in essa contenuti dovesse scendere anziché aumentare. Essa alloca una nuova area di memoria di dimensione appena sufficiente, copia la lista di processi in essa e rilascia la memoria occupata dalla vecchia lista con la funzione di libreria standard *free*.

File io.c

```
#include <stdio.h>

#include "headers/globals.h"
#include "headers/cmd.h"

void printProgHead()
{
        printf("\n*** Scheduler Simulator v1.0 - Simone Cipriani ***\n\n");
}

void printCmds()
{
        printf("Enter one of the following commands:\n");
        printf("        1 [insert new task]\n");
        printf("        2 [execute first-in-queue task]\n");
        printf("        3 [execute specified task]\n");
        printf("        4 [delete specified task]\n");
        printf("        5 [alter specified task's priority]\n");
        printf("        6 [switch scheduling policy]\n");
        printf("        7 [exit]\n");
}

#define SEPARATOR "+----+----------+-------+----------+\n"

void printTaskList()
{
        printf("\n");
        printf(SEPARATOR);
        printf("| ID | PRIORITY | EXEC. |   NAME   |\n");
        printf(SEPARATOR);
        int i =0;
        for (i=0; i<i_num_task; i++)
        {
                printf("| %d", task_list[i].i_id);

                if (task_list[i].i_id < 10)
                        printf("  |");
                else
                        printf(" |");

                printf(" %d", task_list[i].i_priority);

                printf("        | %d", task_list[i].i_cycles);
                if (task_list[i].i_cycles < 10)
                        printf("     ");
                else
                        printf("    ");

                printf("| %s", task_list[i].c_name);

                printf("\n");

                printf(SEPARATOR);
        }
        printf("\n");
```

```
}

int dispatchCMD()
{
        int i_cmd;
        scanf( "%d", &i_cmd );

        if (i_cmd==1)
        {
                int i_priority, i_exec;
                char c_name[8];

                printf("Insert task priority:\n");
                scanf( "%d", &i_priority );
                if (i_priority > 9 || i_priority < 0)
                {
                        printf("ERROR: please inster a number between 0 and 9!\n");
                        return EXIT_SUCCESS;
                }

                printf("Insert task name:\n");
                scanf("%s", &c_name);

                printf("Insert remaining executions:\n");
                scanf( "%d", &i_exec );

                if (insertTask(i_priority, c_name,  i_exec)!=EXIT_SUCCESS)
                {
                        printf("ERROR: could not insert task!");
                        printf("Probably something has gone very bad with memory,
                                the program is closing.");
                        return EXIT_FAILURE;
                }
                return EXIT_SUCCESS;

        }

        if(i_cmd==2)
        {
                if (execTopTask() !=EXIT_SUCCESS)
                {
                        printf("ERROR: could not execute first-in-queue task!");
                        printf("Probably something has gone very bad with memory,
                                the program is closing.");
                        return EXIT_FAILURE;
                }
                return EXIT_SUCCESS;
        }

        if(i_cmd==3)
        {
                int i_id;
                printf("Insert task id:\n");
                scanf( "%d", &i_id );
                if (execTask(i_id) !=EXIT_SUCCESS)
                {
                        printf("ERROR: could not execute specified task!");
```

```
                return EXIT_SUCCESS;
        }
        return EXIT_SUCCESS;

    }

    if(i_cmd==4)
    {
        int i_id;
        printf("Insert task id:\n");
        scanf( "%d", &i_id );
        if (deleteTask(i_id) !=EXIT_SUCCESS)
        {
            printf("ERROR: could not delete specified task!");
            return EXIT_SUCCESS;
        }
        return EXIT_SUCCESS;
    }

    if(i_cmd==5)
    {
        int i_id, i_priority;
        printf("Insert task id:\n");
        scanf( "%d", &i_id );
        printf("Insert task priority:\n");
        scanf( "%d", &i_priority );
        if (alterPriority(i_id, i_priority) !=EXIT_SUCCESS)
        {
            printf("ERROR: could not alter priority of specified
                    task!");
            return EXIT_SUCCESS;
        }
        return EXIT_SUCCESS;
    }

    if(i_cmd==6)
    {
        if (switchSchedulingPolicy() == EXIT_SUCCESS)
        {
            if (i_policy == PRIORITY)
                printf("\nScheduling policy now set to
                        PRIORITY\n");
            if (i_policy == LJF)
                printf("\nScheduling policy now set to LONGEST JOB
                        FIRST\n");

            return EXIT_SUCCESS;
        }
    }

    if(i_cmd==7)
        exit(0);

    //unknown command
    printf("Unknown command. Please enter a valid one:\n");
    return dispatchCMD();
}
```

In questo file sorgente sono definite le funzioni che si occupano di gestire gli input ricevuti dal programma attraverso lo *stdin* e di inviare l'output allo *stdout*. Questo sorgente è infatti l'unico ad includere la libreria **<stdio.h>**

Le funzioni *printProgHead*, *printCmds* e *printTaskList* sono quelle che si occupano di scrivere l'output nello standard output (che, nel caso di questo programma, è la console). La prima viene invocata soltanto all'avvio del programma, le altre due dopo ogni operazione richeista dall'utente.

La funzione *dispatchCMD* invece si occupa di leggere l'input dell'utente e di invocare la funzione appropriata per eseguire il comando immesso.

File cmd.c

```
#include <string.h>

#include "headers/globals.h"
#include "headers/list.h"
#include "headers/cmd.h"

int insertTask(int i_p, char c_n[], int i_ex)
{
        task_t t;

        t.i_id = i_assign_id;
        i_assign_id++;
        t.i_priority = i_p;
        strncpy(t.c_name, c_n, 8); // any character beyond 8th will be ignored
        t.i_cycles = i_ex;

        if (task_size > i_num_task) {
                task_list[i_num_task] = t;
                i_num_task++;
                return EXIT_SUCCESS;
        }
        else {
                int iError = alterListSize(LIST_ENLARGE_STEP);
                task_list[i_num_task] = t;
                i_num_task++;
                return iError;
        }
}

int execTopTask()
{
        task_list[i_num_task-1].i_cycles--;

        if (task_list[i_num_task-1].i_cycles <= 0)
                return deleteTask(task_list[i_num_task-1].i_id);
        else return EXIT_SUCCESS;
}

int execTask(int i_task_id)
{
        int i;
        for (i=0; i<i_num_task; i++)
        {
                if (task_list[i].i_id == i_task_id)
                {
                        task_list[i].i_cycles--;
                        if (task_list[i].i_cycles <= 0)
                                return deleteTask(task_list[i].i_id);
                        else return EXIT_SUCCESS;
                }
        }
        //if id does not exist, simply do nothing
        //and let cmd dispatcher print an error msg
        return EXIT_FAILURE;
}
```

```
int deleteTask(int i_task_id)
{
        int k;
        for(k = 0; k<i_num_task; k++) {
                if (task_list[k].i_id == i_task_id)
                {
                        task_list[k].i_id = 0;
                        task_list[k].i_priority = 0;
                        strncpy(task_list[k].c_name, "", 8);
                        task_list[k].i_cycles = 0;
                }
        }
        i_num_task--;
        return releaseEmptySpace();
}

int alterPriority(int i_task_id, int i_new_p)
{
        int k;
        for(k = 0; k<i_num_task; k++) {
                if (task_list[k].i_id == i_task_id)
                {
                        task_list[k].i_priority = i_new_p;
                        return EXIT_SUCCESS;
                }
        }
        return EXIT_FAILURE;
}

int switchSchedulingPolicy()
{
        if (i_policy == PRIORITY)
                i_policy = LJF;
        else if (i_policy == LJF)
                i_policy = PRIORITY;

        return EXIT_SUCCESS;
}
```

Le funzioni contenute in questo sorgente sono quelle che effettivamente implementano i comandi richiesti dalle specifiche dell'esercizio: parafrasando un'espressione diventata un meme su alcuni social network, *this is where the magic happens.*

E' stata scritta una funzione per ogni comando. Esse realizzano l'implementazione richeista invocando le procedure i cui prototipi sono stati definiti negli header inclusi ed eseguendo operazioni elementari.

Viene inclusa la libreria **<string.h>** per avvalersi di *strncpy*.

Istruzioni per la compilazione

Il file *build_run.sh* contiene uno script bash che esegue i comandi necessari per compilare ed eseguire il programma. Il contenuto del file è il seguente:

```
#!/bin/sh

# clean
rm -rf bin
mkdir bin

# compile
cc -c src/main.c -o bin/main.o
cc -c src/list.c -o bin/list.o
cc -c src/cmd.c -o bin/cmd.o
cc -c src/io.c -o bin/io.o

# link
cc bin/main.o bin/list.o bin/cmd.o bin/io.o -o bin/main

# run
./bin/main
```

La prima operazione che viene eseguita è una pulizia della cartella ./bin, onde rimuovere ogni file generato da compilazioni precedenti. Per fare ciò viene usato il comando **rm** con il flag **-rf**, che elimina la cartella argomento e tutto il suo contenuto. Successivamente, la stessa cartella viene ricreata con il comando **mkdir**.

La fase di compilazione prevede l'invocazione del comando **cc** con il flag **-c** per generare i file oggetto dei vari moduli del programma a partire dai sorgenti.

Per generare finalmente un file eseguibile, viene invocato nuovamente il comando **cc**, che provvederà a linkare i file oggetto generati precedentemente, producendo un solo file che potrà essere lanciato.

L'ultimo comando, infatti, lancia l'eseguibile.

Esempio di compilazione:

- aprire una shell bash e posizionarsi nella cartella principale del programma con il comando **cd**:

```
:~$ cd Projects/os-class-homework/ES1/
```

- lanciare lo script bash contenuto nella cartella invocandolo da bash:

```
:~/Projects/os-class-homework/ES1$ ./build_run.sh
```

Evidenza di funzionamento

Compilazione e lancio del programma (vedere sezione precedente):

```
*** Scheduler Simulator v1.0 - Simone Cipriani ***

Enter one of the following commands:
        1 [insert new task]
        2 [execute first-in-queue task]
        3 [execute specified task]
        4 [delete specified task]
        5 [alter specified task's priority]
        6 [switch scheduling policy]
        7 [exit]
```

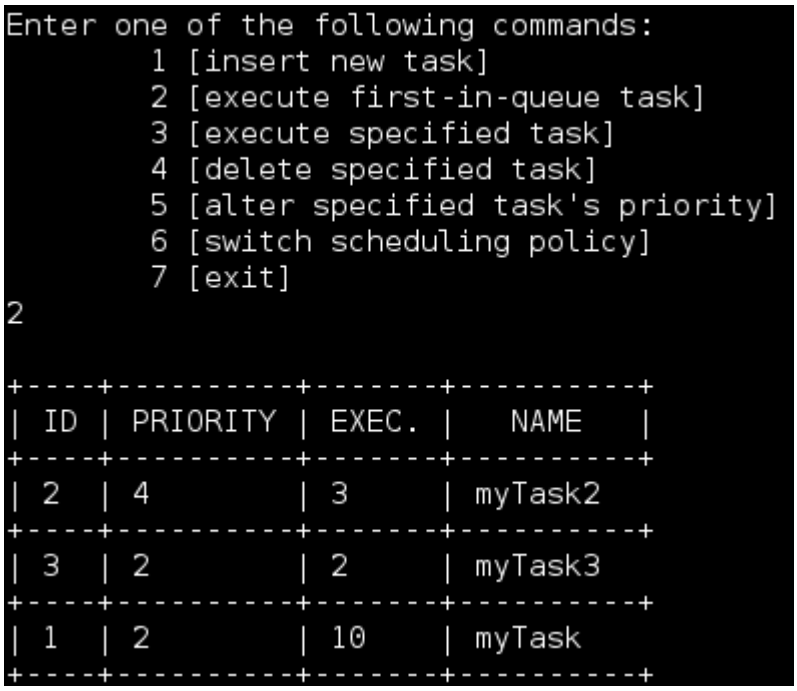
Inserimento di un task:

```
Enter one of the following commands:
        1 [insert new task]
        2 [execute first-in-queue task]
        3 [execute specified task]
        4 [delete specified task]
        5 [alter specified task's priority]
        6 [switch scheduling policy]
        7 [exit]
1
Insert task priority:
2
Insert task name:
myTask
Insert remaining executions:
11

+----+----------+-------+----------+
| ID | PRIORITY | EXEC. |   NAME   |
+----+----------+-------+----------+
| 1  | 2        | 11    | myTask
+----+----------+-------+----------+
```

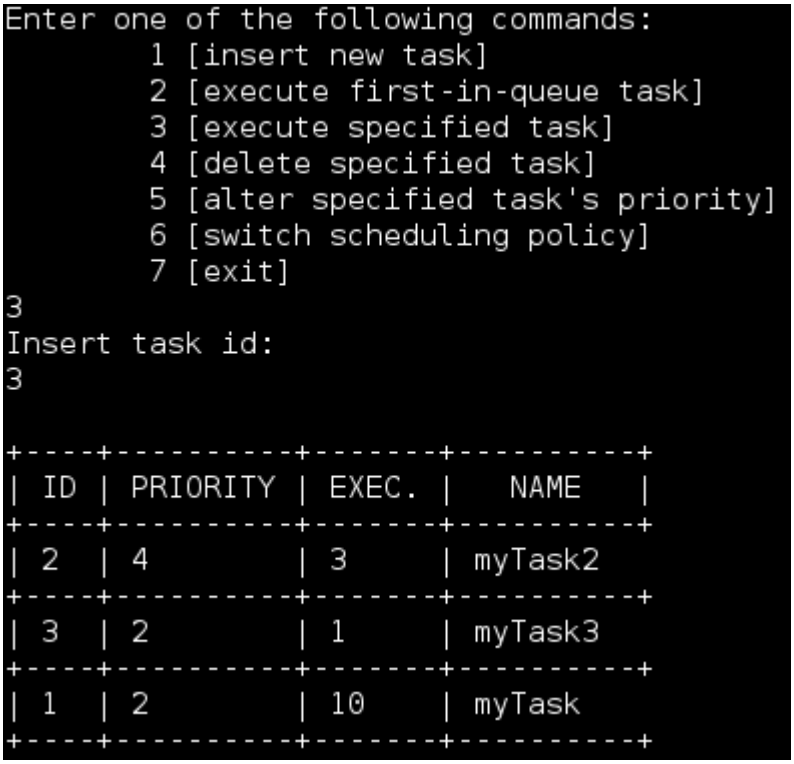
Inserimento di altri task:

```
+----+----------+-------+----------+
| ID | PRIORITY | EXEC. |   NAME   |
+----+----------+-------+----------+
| 2  | 4        | 3     | myTask2
+----+----------+-------+----------+
| 3  | 2        | 2     | myTask3
+----+----------+-------+----------+
| 1  | 2        | 11    | myTask
+----+----------+-------+----------+
```

Esecuzione del primo task nella coda:



Esecuzione di un task specificato:



Modifica della priorità di un task:

```
Enter one of the following commands:
        1 [insert new task]
        2 [execute first-in-queue task]
        3 [execute specified task]
        4 [delete specified task]
        5 [alter specified task's priority]
        6 [switch scheduling policy]
        7 [exit]
5
Insert task id:
2
Insert task priority:
1

+----+----------+-------+----------+
| ID | PRIORITY | EXEC. |   NAME   |
+----+----------+-------+----------+
| 3  | 2        | 1     | myTask3
+----+----------+-------+----------+
| 1  | 2        | 10    | myTask
+----+----------+-------+----------+
| 2  | 1        | 3     | myTask2
+----+----------+-------+----------+
```

Cambio della politica di scheduling:

```
Enter one of the following commands:
        1 [insert new task]
        2 [execute first-in-queue task]
        3 [execute specified task]
        4 [delete specified task]
        5 [alter specified task's priority]
        6 [switch scheduling policy]
        7 [exit]
6

Scheduling policy now set to LONGEST JOB FIRST

+----+----------+-------+----------+
| ID | PRIORITY | EXEC. |   NAME   |
+----+----------+-------+----------+
| 3  | 2        | 1     | myTask3
+----+----------+-------+----------+
| 2  | 1        | 3     | myTask2
+----+----------+-------+----------+
| 1  | 2        | 10    | myTask
+----+----------+-------+----------+
```

Esecuzione del primo task nella coda:

```
Enter one of the following commands:
        1 [insert new task]
        2 [execute first-in-queue task]
        3 [execute specified task]
        4 [delete specified task]
        5 [alter specified task's priority]
        6 [switch scheduling policy]
        7 [exit]
2

+----+----------+-------+----------+
| ID | PRIORITY | EXEC. |   NAME   |
+----+----------+-------+----------+
| 3  | 2        | 1     | myTask3
+----+----------+-------+----------+
| 2  | 1        | 3     | myTask2
+----+----------+-------+----------+
| 1  | 2        | 9     | myTask
+----+----------+-------+----------+
```

Eliminazione di un task specificato:

```
Enter one of the following commands:
        1 [insert new task]
        2 [execute first-in-queue task]
        3 [execute specified task]
        4 [delete specified task]
        5 [alter specified task's priority]
        6 [switch scheduling policy]
        7 [exit]
4
Insert task id:
1

+----+----------+-------+----------+
| ID | PRIORITY | EXEC. |   NAME   |
+----+----------+-------+----------+
| 3  | 2        | 1     | myTask3
+----+----------+-------+----------+
| 2  | 1        | 3     | myTask2
+----+----------+-------+----------+
```

ESERCIZIO 2: esecutore di comandi

Descrizione dell'implementazione – versione seriale

File main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFERSIZE 512

char buffer[BUFFERSIZE];
int out_num;

int main (int argc, char *argv[])
{
        out_num = 1;

        while(fgets(buffer, BUFFERSIZE , stdin) != NULL)
        {
                // exit if empty line
                if (buffer[0]=='\n')
                        return 0;

                // prepare std out redirection
                char to_file[BUFFERSIZE];
                snprintf(to_file, sizeof to_file, ">  out.%d", out_num);
                out_num++;

                // remove newline and form complete shell command
                strtok(buffer, "\n");
                strcat(buffer, to_file);

                // rock it!
                system(buffer);

        }
        return 0;
}
```

Vista la semplicità del programma, l'intero codice soegente è contenuto in questo unico file.

Il loop principale legge una riga dallo standard input con la funzione *fgets* ed esegue operazioni su di essa per redirigere lo standard output del comando che contiene in un file. A questo proposito ci si avvale di alcune funzioni della libreria **<string.h>**

Il programma si interrompe non appena viene letta una stringa vuota (ovvero contenente il solo carattere *newline*) dallo standard input, oppure, quando la funzione *fgets* non ritorna alcunché (ad esempio nel caso in cui lo standard input è un file che è già stato letto fino alla fine).

Non sono state gestite le situazioni di errore generate da comandi errati mandati in esecuzione, in quanto la shell propagherà tali errori nello standard output imposto dal programma.

Descrizione dell'implementazione – versione parallela

File main.c

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#define BUFFERSIZE 512

char buffer[BUFFERSIZE];
int out_num;
int pid;

int main (int argc, char *argv[])
{
        out_num = 1;

        while(fgets(buffer, BUFFERSIZE , stdin) != NULL)
        {
                // exit if empty line
                if (buffer[0]=='\n')
                        return 0;

                // prepare std out redirection
                char to_file[BUFFERSIZE];
                snprintf(to_file, sizeof to_file, ">  out.%d", out_num);
                out_num++;

                // remove newline and form complete shell command
                strtok(buffer, "\n");
                strcat(buffer, to_file);

                // create a child and make him run the shell command
                if ((pid = fork()) == -1)
                {
                        perror("Impossible to fork!"); // something went wrong
                        exit(1);
                }
                else if (pid==0) // I'm the child, run the stuff
                {
                        system(buffer);
                        break;
                }
        }
        return 0;
}
```

Molto simile alla precedente, questa risoluzione dell'esercizio invoca la system call *fork* per creare un processo figlio che si occuperà di lanciare il comando recuperato dal file. In questo modo il processo padre potrà continuare la scansione del file, senza dover aspettare il completamento dell'istruzione in esso contenuta.

E' stata gestita la situazione di errore generata dall'impossibilità di effettuare il fork chiudendo il programma, proppagando un errore (di codice 1) e stampando un messaggio.

Istruzioni per la compilazione

Essendo composto da un unico file, la compilazione del programma è banale. In ogni caso, per risparmiare tempo, ci si può avvalere di uno script bash che compila e lancia il programma, impostando come standard input un file di testo precedentemente creato.

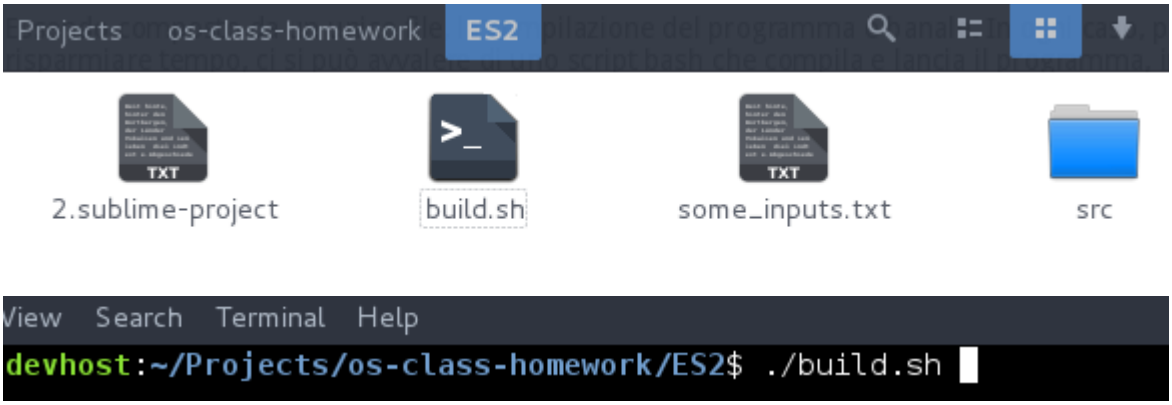
```
#!/bin/sh

# clean
rm -rf bin
mkdir bin

# compile
cc -c src/main.c -o bin/main.o

# link
cc bin/main.o -o bin/main

# run
cd bin      # output is generated in current dir
cat ../some_inputs.txt | ./main
```

Il file *some_inputs.txt* contiene tre comandi bash ed una riga vuota:

```
ls -la
date
echo "I'm working!"
```

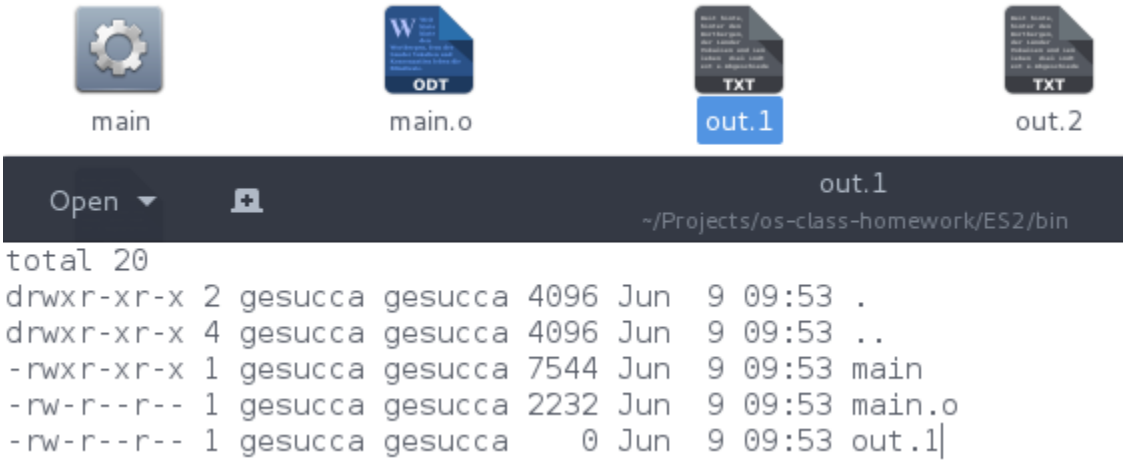
Evidenze di funzionamento

Lancio dello script bash per la compilazione e l'esecuzione:



Viene creata la cartella bin, contenente il file oggetto, il file eseguibile ed i file dove è rediretto lo standard output dei comandi contenuti nel file di testo di input:

Ad esempio, nel file *out.1* è contenuto lo standard output del primo comando, *ls- la*.



Il comportamento esterno è il medesimo sia per la versione sequenziale che quella parallea.

ESERCIZIO 3: scambio di messaggi

Descrizione dell'implementazione

codice in comune : *pipe.h*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>


#define CLIENT_REQ "req_pipe"
#define PIPE_READ_ERROR_MSG "ERROR: unable to read from pipe!"

#define BUFFER_SIZE 512

int createReqPipe();
int openReqPipe();
int createResPipe();
int openResPipe();
int writeInPipe(int pipe, char* toBeWritten);
```

codice in comune : *pipe.c*

```
#include "pipe.h"

#include <string.h>

//server
int createReqPipe()
{
        unlink(CLIENT_REQ); // remove named pipe if already exists
        printf("        unlink errno: %d\n",errno);
        errno = 0;

        mkfifo(CLIENT_REQ, 0666);
        printf("        mkfifo errno: %d\n",errno);

        printf("Almost done, waiting for a client to open req pipe...\n");
        int req_d = open (CLIENT_REQ, O_RDONLY);
        printf("        open descriptor: %d\n", req_d);

        return req_d;
}

//client
int openReqPipe()
{
```

```
        int req_d;
        do {
                req_d = open(CLIENT_REQ, O_WRONLY);
                printf("        open request pipe: %d\n", req_d);
                if (req_d == -1)
                {
                        printf("        will try again\n");
                        sleep(1);
                }
        } while (req_d == -1);
        return req_d;
}

//server
int createResPipe(char* name)
{
        unlink(name);
        printf("        unlink errno: %d\n",errno);
        errno = 0;

        mkfifo(name, 0666);
        printf("        mkfifo errno: %d\n",errno);

        int res_d = open(name, O_WRONLY);
        printf("        open descriptor: %d\n", res_d);

        return res_d;
}

//client
int openResPipe()
{
        int req_d;
        do {
                char str[10];
                sprintf(str, "%d", getpid());
                req_d = open(str, O_RDONLY);
                printf("        open response pipe: %d\n", req_d);
                if (req_d == -1)
                {
                        printf("        will try again\n");
                        sleep(1);
                }
        } while (req_d == -1);
        return req_d;
}

int writeInPipe(int pipe, char* toBeWritten)
{
        int error = write(pipe, toBeWritten, strlen(toBeWritten) * sizeof(char));
        sleep(1); //let the other process read from from the pipe
        return error;
}
```

codice in comune : *custsignal.h*

```
#include <signal.h>

#define SIG_MSG SIGUSR1
#define SIG_N_EX SIGUSR2

int req_d; // need this to catch ctrl-c closing

void setupSignals();
```

Questa sezione di codice, composta da due headers ed un solo file sorgente, viene condivisa dal modulo server e da quello client per ragioni di consistenza e di efficienza del processo di compilazione.

In particolare, l'header *custsignal.h* viene condiviso perché contiene le macro che definiscono i segnali usati per comunicare la ricezione di un messaggio e l'assenza del destinatario specificato.

Il modulo *pipe.c* ed il rispettivo header invece definisce i metodi per creare, aprire e scrivere dentro determinati pipes fifo. Il modulo viene compilato autonomamente ed il suo file oggetto linkato con quelli di del programma client e del programma server (si veda la sezione "Istruzioni per la compilazione"), i quali incluideranno l'header di queso modulo nei propri sorgenti per poterne chiamare le funzioni.

Nel merito delle operazioni offerte dal modulo, questo esegue la creazione di un pipe chiamando prima la syscall *unlink*, per eliminare eventuali pipe preesistenti nella stessa locazione ad aventi lo stesso nome, chiede al kernel la creazione vera e propria tramite la syscall *mkfifo* ed infine lo apre con la syscall *open*, restituendo al chiamante il descrittore cosicché lo possa utilizzare.

Le funzioni che offrono l'apertura di determinati pipe prevedono di tentare ciclicamente ad aprire il pipe in questione tramite un polling con frequenza di 1 secondo, il quale si interrompe non appena l'operazione di apertura ha successo.

server *: main.h*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

// by fixing this value I can avoid using malloc and going for a simple int array
#define MAX_NUM_CLIENTS 256

int pid[MAX_NUM_CLIENTS];
int pid_n;
```

server *: main.c*

```
#include "../common/pipe.h"
#include "../common/custsignal.h"
#include "main.h"

static int init()
{
        printf("Starting server...\n");
        pid_n = 0;
        int req_d = createReqPipe();
        printf("All set up. Server running...\n");
        return req_d;
}

static void req1(int req_d)
{
        char buffer[BUFFER_SIZE];
        if (read(req_d, buffer, BUFFER_SIZE))
        {
                int temp_pid = atoi(buffer);
                int already_there = 0;
                int i;

                for (i=0; i<pid_n; i++)
                {
                        if (pid[i]==temp_pid)
                        {
                                already_there=1;
                                printf("        client %d already logged!\n",
                                        temp_pid);
                        }
                }


                if (!already_there)
                {
                        pid[pid_n] = atoi(buffer);
                        pid_n++;
```

```
                        for (i=0; i<pid_n; i++)
                                printf("        client n° %d logged in\n", pid[i]);
                }
        }
        else
                printf(PIPE_READ_ERROR_MSG);
}

static void req2(int req_d)
{
        // read pid
        char buffer[sizeof(int)];
        if (read(req_d, buffer, sizeof(int)))
        {
                // signal the client about the pipe
                printf("        signal to %d ", atoi(buffer));
                printf("returned code: %d\n", kill(atoi(buffer), SIG_MSG));
                int res_d = createResPipe(buffer);

                char list[BUFFER_SIZE] = {'\0'};
                int error, i;

                if (pid_n <1)
                {
                        strcpy(list, "No clients logged!\n");
                }

                for (i=0; i<pid_n; i++)
                {
                        char temp[sizeof(int)];
                        sprintf(temp, "%d ", pid[i]);
                        strcat(list, temp);
                }

                strcat(list, "\n\0");

                error = writeInPipe(res_d, list);
                printf("        write in pipe returned code: %d\n", error);
        }
        else
                printf(PIPE_READ_ERROR_MSG);
}

static void req3(int req_d)
{
        char buffer[BUFFER_SIZE];
        if (read(req_d, buffer, BUFFER_SIZE))
        {
                char msg[BUFFER_SIZE];
                msg[0] = '\0';
                const char *start = strrchr(buffer, '_') + 1;
                strncat(msg, start, strcspn(start, "#"));

                char sender[10];
                sender[0] = '\0';
                const char *s = strrchr(buffer, '+') + 1;
                strncat(sender, s, strcspn(s, "*"));
```

```
                while (strchr(buffer, '_'))
                {
                        // exrtact the destinatary from buffer string
                        char dest[10];
                        dest[0] = '\0';
                        strncat(dest, buffer, strcspn(start, "_")-1);

                        //strip away the destinatary taken
                        const char *start = strchr(buffer, '_') + 1;
                        char new_buffer[BUFFER_SIZE];
                        new_buffer[0] = '\0';
                        strncat(new_buffer, start, strcspn(start, "#"));
                        strcpy(buffer, new_buffer);

                        // check if destinatary exists
                        int i, ok = 0;
                        for (i=0; i<pid_n; i++)
                        {
                                if (pid[i]==atoi(dest))
                                        ok = 1;
                        }

                        if (ok)
                        {
                                printf("        signal to %d ", atoi(dest));
                                printf("returned code: %d\n", kill(atoi(dest),
                                                                SIG_MSG));

                                //this may not be elegant, but it wil clean dest
                                  pid from dirty chars
                                sprintf(dest, "%d", atoi(dest));
                                int res_d = createResPipe(dest);

                                int error = writeInPipe(res_d, msg);
                                printf("        write in pipe returned code: %d\n",
                                       error);
                        }
                        else
                        {
                                printf("        signal to %d ", atoi(sender));
                                printf("returned code: %d\n", kill(atoi(sender),
                                                                SIG_N_EX));
                        }
                }

        }
        else
                printf("Something went terribly wrong...");
}

static void req4(int req_d)
{
        char buffer[BUFFER_SIZE];
        if (read(req_d, buffer, BUFFER_SIZE))
        {
                int temp_pid = atoi(buffer);
```

```
                int i, index = 0;

                for (i=0; i<pid_n; i++)
                {
                        if (pid[i]==temp_pid)
                        {
                                pid[i] = 0;
                                index=i+1;
                                printf("        client n° %d logged out\n",
                                        temp_pid);
                                break;
                        }
                }

                if (index)
                {
                        index--;
                        for (; index<pid_n; index++)
                        {
                                pid[index] = pid[index+1];
                        }
                        pid_n--;
                        printf("        cleaned process list\n");
                }
        }
        else
                printf(PIPE_READ_ERROR_MSG);
}

int main()
{
        int req_d = init();

        // main server loop
        while (1)
        {
                char buffer[1];
                if (read(req_d, buffer, 1))
                {
                        int req_to_serve = atoi(buffer);
                        printf("asked for req n° %d\n", req_to_serve);

                        if (req_to_serve ==1)
                                req1(req_d);

                        if (req_to_serve ==2)
                                req2(req_d);

                        if (req_to_serve ==3)
                                req3(req_d);

                        if (req_to_serve ==4)
                                req4(req_d);
                }
                else {
                        printf("None is connected: nothing to do...\n");
                        sleep(3);
```

```
                }
            }
}
```

Il modulo server è stato sviluppato in un unico file sorgente, con rispettivo header. Il motivo di questa scelta è stato limitare la complessità del sistema, già di per se complicato dato l'argomento dell'esercizio, che sarebbe accresciuta notevolmente nel caso in cui il programma server fosse stato scomposto in vari moduli (si pensi ad esempio alle dipendenze causate dalla macro *#include*).

Nel merito del funzionamento, il modulo presenta un loop principale nel metodo main, il quale legge ciclicamente dal pipe delle richieste ed esegue l'operazione corrispondente al valore che legge. Questo pipe viene predisposto nel metodo *init()*, il quale viene invocato come prima operazione del programma e che si occupa anche di inizializzare la lista dei processi connessi al server.

Riguardo a questa lista, è stato scelto di implementarla con un array di interi fissando un numero massimo di processi disponibili, per evitare una gestione dinamica della memoria e permettere allo sviluppatore di concentrare il focus sulle questioni centrali dell'esercizio.

Le operazioni effettuate per eseguire le richieste sono contenute in funzioni apposite, allo scopo di raggrupparle e rendere il codice più leggibile. Nel dettaglio, nelle operazioni 1 e 4 si va ad agire sulla lista di processi collegati, mentre nelle operazioni 2 e 3 si crea un pipe apposito per la comunicazione con il processo richiedente (e/o con il processo destinatario del messaggio, per quanto riguarda la richeista numero 3), inviando attraverso esso le informazioni richieste.

Riguardo alle funzioni implementanti la soddisfazione delle richieste 2 e 3, oltre alla creazione del pipe si ha l'invio di un segnale al processo client interessato, per notificare l'avvenuta creazione di un pipe di comunicazione e l'inserimento in esso di un'informazione da elaborare. Per la funzione 3, nel caso in cui il destinatario inviato al server fosse inesistente, viene utilizzato attraverso un segnale apposito al mittente e nessun pipe di comunicazione viene creato.

client *: custsignal.c*

```c
#include "../common/custsignal.h"
#include "interact.h" // for macros

static void msgHandler(int sig)
{
        char msg[BUFFER_SIZE];

        char c_pid[10];
        sprintf(c_pid, "%d", getpid());

        int req_d = openResPipe();

        if (read(req_d, msg, BUFFER_SIZE))
        {
                printf("\nReceived message from server:\n%s\n",msg);
        }
}

static void destinataryHandler(int sig)
{
        printf("\nERROR: chosen destinatary does not exists!\n");
}

static void closeHandler(int sig)
{
        printf("Client is closing. Disconnecting...");
        char cmd[] = "4";
        int error = writeInPipe(req_d, cmd);
        printf("        write returned code: %d\n", error);
        char c_pid[10];
        sprintf(c_pid, "%d", getpid());
        error = writeInPipe(req_d, c_pid);
        printf("        write returned code: %d\n", error);
        exit(0);
}

void setupSignals(int r)
{
        req_d = r;
        printf("...setting up signals...\n");
        signal(SIG_MSG, msgHandler);
        signal(SIG_N_EX, destinataryHandler);
        signal(SIGINT, closeHandler);
        printf("...done...\n");
}
```

Questo file sorgente implementa la funzione *setupSignals(int r)* definita nel suo header, la quale innesta i signal handler richeisti dalle specifiche dell'esercizio. Il funzionamento di tali handler risulta abbastanza intellegibile dal codice sorgente, per cui non verranno spese parole in merito.

Si noti che l'handler del segnale SIGINT inoltra al server una richeista di disconnessione del proprio PID prima di chiudere effettivamente il programma.

client : *interact.h*

```
#include "../common/pipe.h"
#include <stdlib.h>
#include <string.h>

void printGreetings();
void printMenu();
void dispatchCmd(int req);
```

client : *interact.c*

```
#include  "interact.h"

void printGreetings() {
        printf("CLIENT PROCESS v1.0 - Simone Cipriani\n");
}

void printMenu() {
        printf("\nSelect one of the following options: \n");
        printf("        [1] : connect to server\n");
        printf("        [2] : requests client list\n");
        printf("        [3] : send text message to other clients\n");
        printf("        [4] : disconnect from server\n");
        printf("        [5] : exit program\n");
}

static void message(int req_d)
{
        char msg[BUFFER_SIZE];
        char separator[1] = "_";
        char end_m[1] = "#";
        int error;

        printf("Please insert your message:\n");
        scanf( "%s", msg );
        strcat(msg, end_m);

        char me[10];
        sprintf(me, "+%d*", getpid());
        strcat(msg, me);

        while (1)
        {
                char dest[BUFFER_SIZE];
                printf("Please insert destinatary: [0 for stop]\n");
                scanf( "%s", dest );

                if (atoi(dest)==0)
                        break;

                strcat(dest, separator);
                strcat(dest, msg);
                strcpy(msg, dest);
        }
```

```
        error = writeInPipe(req_d, msg);
        printf("        write in pipe returned code: %d\n", error);
}

static void readIdList(int req_d)
{
        //pass pid to server
        char c_pid[10];
        sprintf(c_pid, "%d", getpid());
        int error = writeInPipe(req_d, c_pid);
        printf("        write returned code: %d\n", error);

        //server will rise signal
}

void dispatchCmd(int req_d)
{
        char c_cmd[1];
        int error;

        scanf( "%s", c_cmd );

        // handle immediately close action
        int i_cmd = atoi(c_cmd);
        if (i_cmd == 5)
        {
                kill(getpid(), SIGINT);
                return;
        }

        error = writeInPipe(req_d, c_cmd);
        printf("        write returned code: %d\n", error);

        //log in, log out
        if (i_cmd == 1 || i_cmd == 4)
        {
                char c_pid[10];
                sprintf(c_pid, "%d", getpid());
                error = writeInPipe(req_d, c_pid);
                printf("        write returned code: %d\n", error);
        }

        if (i_cmd == 2)
                readIdList(req_d);

        if (i_cmd == 3)
                message(req_d);
}
```

In questo file sorgente sono implementate le funzioni che si occupano di comunicare con l'utente attraverso la console e di inviare le richieste al server attraverso l'apposito pipe.

Si noti che il comportamento del metoo *dispatchCmd* non effettua (almeno non direttamente) alcuna scrittura nel pipe di comunicazione con il server per soddisfare la richiesta numero 5, ma si limita a lanciare il segnale di interruzione a sé stesso.

Istruzioni per la compilazione

Il file *build_system.sh* contiene uno script bash che si occupa della compilazione e del lancio dei programmi che compongono l'esercizio.

```
#!/bin/sh

# clean
rm -rf bin
mkdir bin

# compile commons
cc -c src/common/pipe.c -o bin/pipe.o

# compile server
cc -c src/server/main.c -o bin/server.o

# link server
cc bin/server.o  bin/pipe.o -o bin/server

# compile client
cc -c src/client/main.c -o bin/cmain.o
cc -c src/client/interact.c -o bin/cinteract.o
cc -c src/client/custsignal.c -o bin/csignal.o

# link client
cc bin/cmain.o bin/cinteract.o bin/csignal.o bin/pipe.o -o bin/client

# clean folder from object files
rm bin/*.o

# run everything
./run.sh
```

Nel dettaglio, le istruzioni contenute nelfile prevedono la creazione di una cartella ./bin vuota, la compilazione di ogni modulo nel suo rispettivo file oggetto, il linking dei vari moduli per comporre gli eseguibili e infine la pulizia della cartella dai vari file oggetto generati.

Infine, lo script invoca un altro script col comando **`./run.sh`**

```
cd bin
xterm -e ./server &
xterm -e ./client &
xterm -e ./client
```

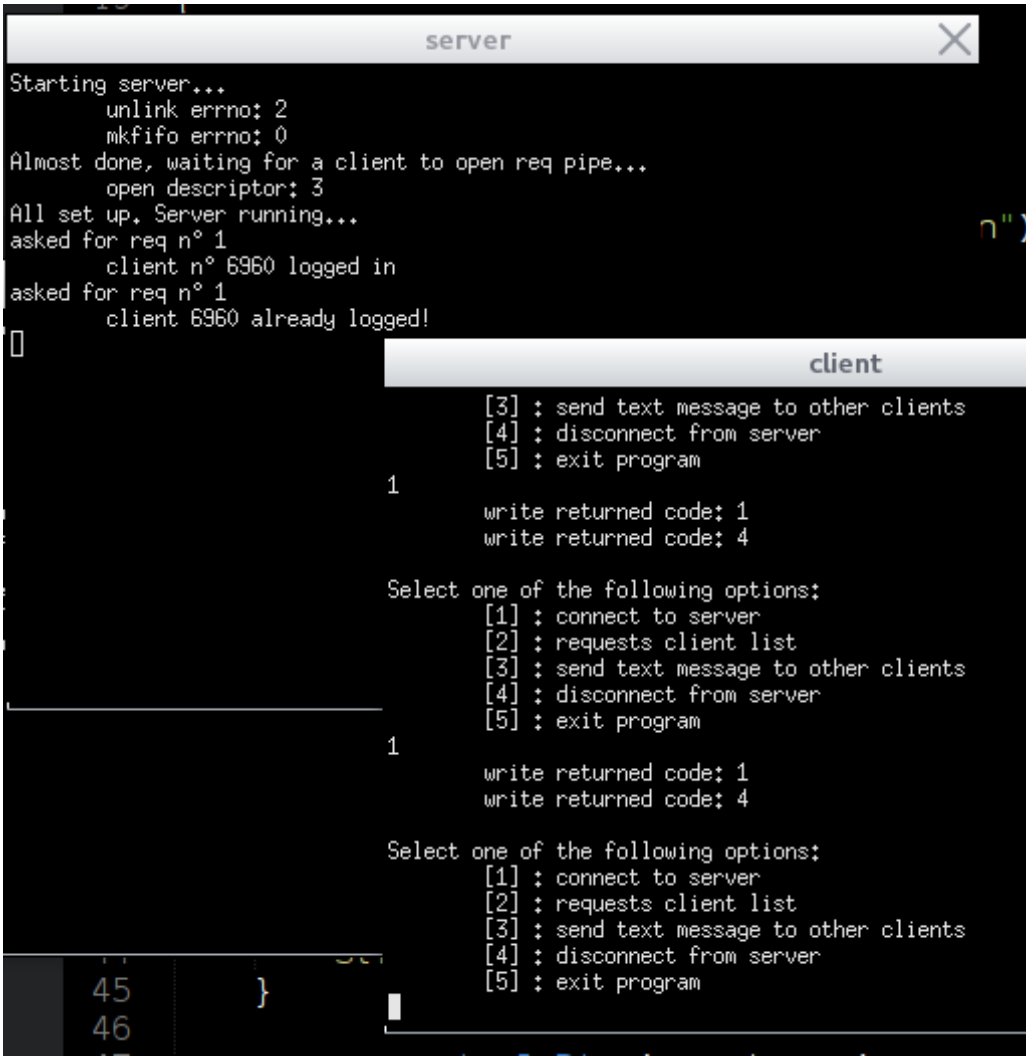
Questo script si occupa di posizionarsi nella cartella dove risiedono i file eseguibili e di lanciarli parallelamente in tre terminali diversi (usando l'emulatore standard di terminale *xterm*).

Se si preferisce lanciare manualmente il programma server e varie istanze di programma client, l'esecuzione di quest'ultimo script si può omettere.

Evidenze di funzionamento

Lancio dello script *build_system.sh*

Ripetute richeiste di login da parte di uno dei due client in esecuzione (operazione 1):



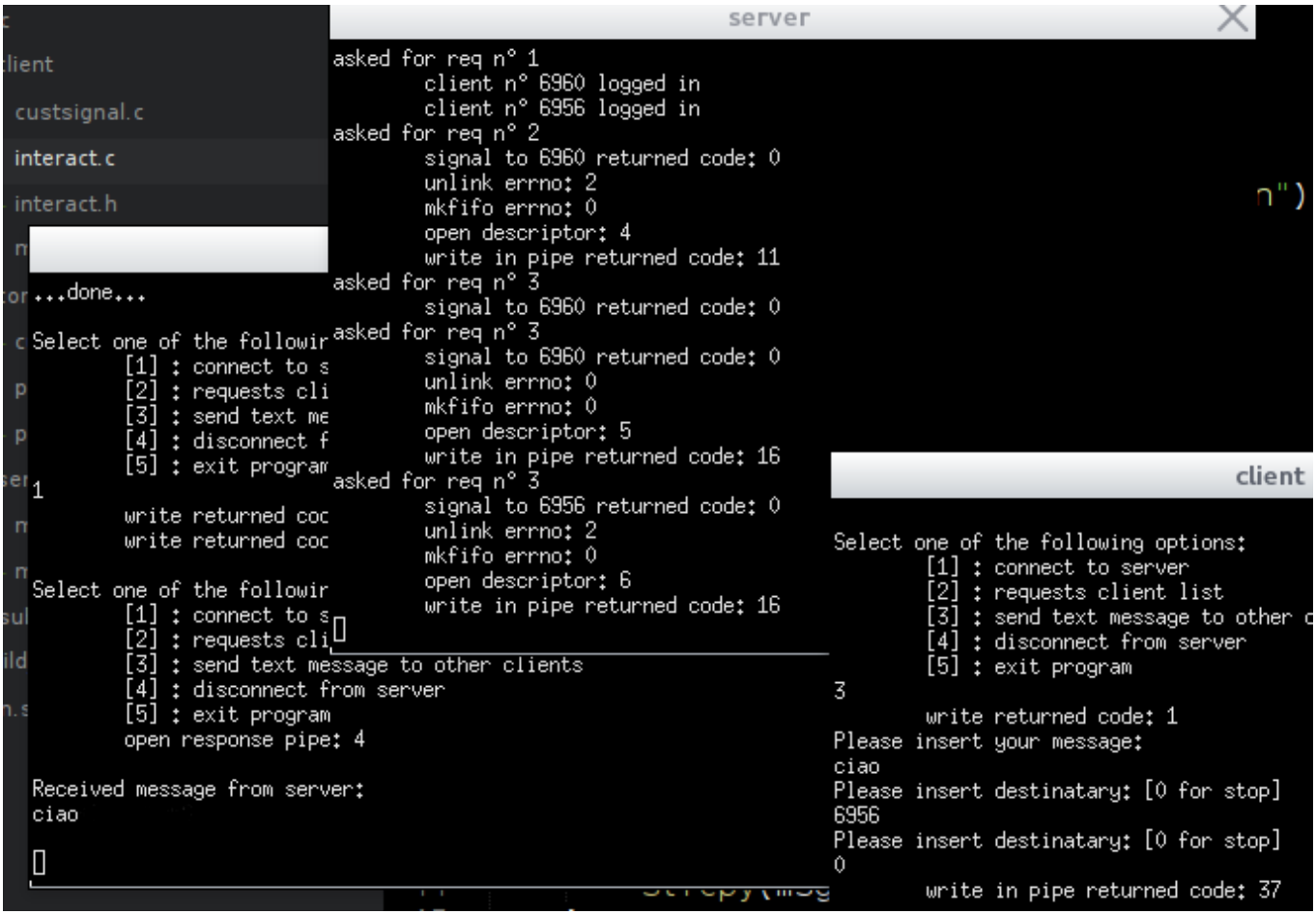
Si noti che il programma è sufficientemente robusto da “rimbalzare” eventuali tentativi di login successivi al primo (non preceduti da richeiste di logout).

Richiesta di avere una lista dei client connessi al server (operazione 2):

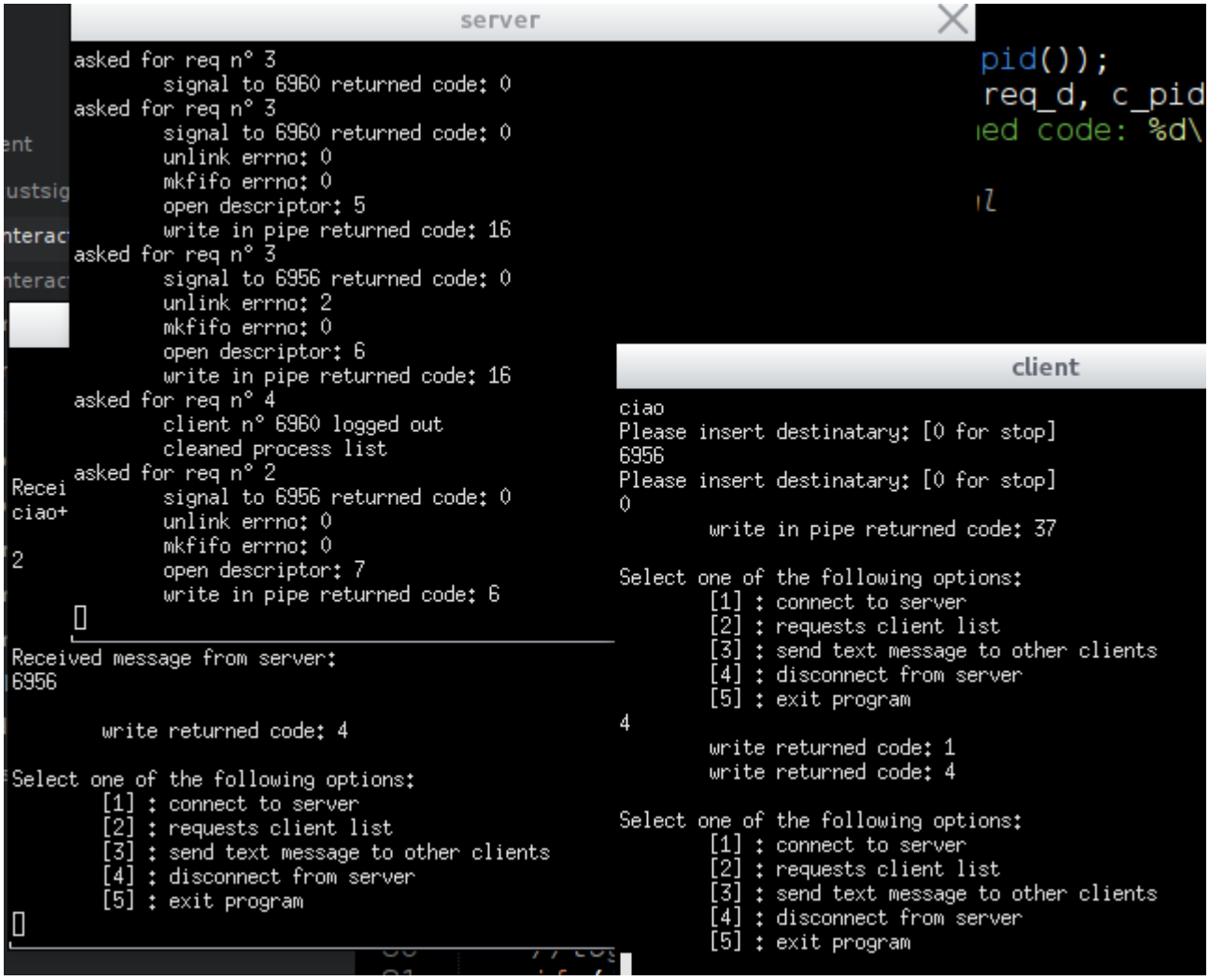
Richiesta di invio messaggio ad un client il cui PID non è connesso (operazione 3):

```
3
        write returned code: 1
Please insert your message:
ciao
Please insert destinatary: [0 for stop]
1234
Please insert destinatary: [0 for stop]
0

ERROR: chosen destinatary does not exists!
        write in pipe returned code: 37
```

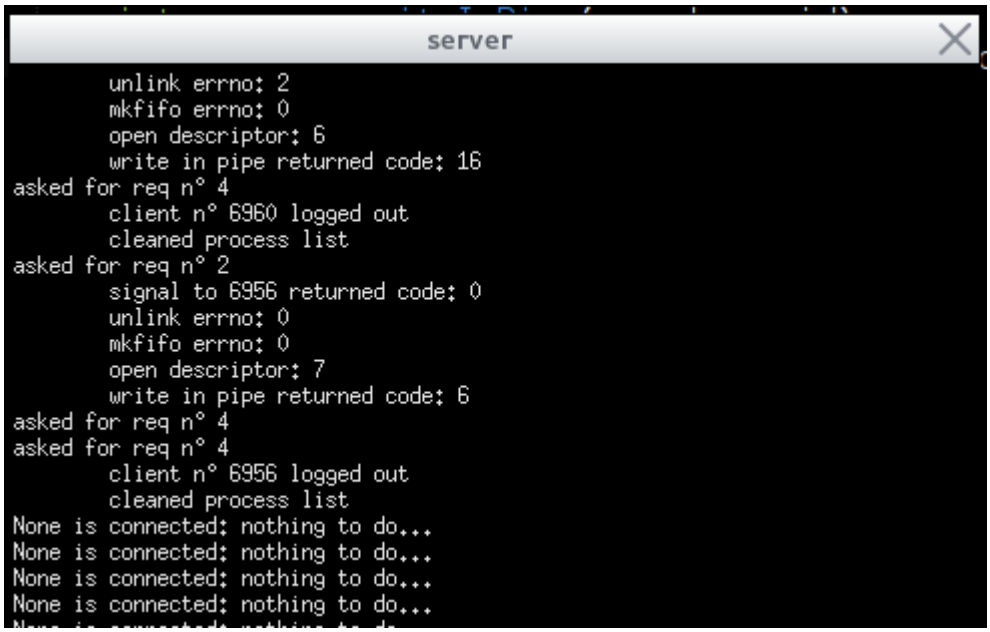
Richiesta di invio messaggio ad un client in esecuzione e connesso al server (operazione 3):



Richiesta di disconnessione da parte di un client connesso (operazione 4):



Chiusura di un programma client tramite l'operazione 5 e di un altro tramite CTRL+C:



Si noti che, in entrambi i casi, al server viene inviata una richeista di disconnessione (operazione 4) prima che il programma client venga effettivamente terminato.

ASPETTO INTERESSANTE:

nella working directory dove sono stati lanciati i programmi server e client, è possibile notare i file di pipe relativi alle richeiste al server (“req_pipe”) ed alla comunicazione di esso con i client (nominati con il pid del processo client in questione).