Relazione sul Progetto dell'Esame di

Sistemi Operativi

Anno Accademico 2016/17

Bacciarini Yuri - `5654547` - *yuri.bacciarini@stud.unifi.it*
Bindi Giovanni - `5530804` - *giovanni.bindi@stud.unifi.it*
Puliti Gabriele- `5300140` - *gabriele.puliti@stud.unifi.it*

June 25, 2017

Contents

1 Primo Esercizio : Simulatore di chiamate a procedura

1.1 Descrizione dell'implementazione

L'obiettivo del primo esercizio é quello di implementare uno scheduler di processi. Quest'ultimo deve permettere all'utente di poter creare, eseguire ed eliminare i processi stessi secondo una politica di priorità od esecuzioni rimanenti.
Abbiamo organizzato il codice in tre files: due librerie *config.h* e *taskmanager.h* ed un programma, *scheduler.c*. All'interno di *config.h*1 vengono unicamente definite due stringhe utilizzate nella formattazione dell'output. All'interno di *taskmanager.h*2 abbiamo invece definito la `struct` *TaskElement*, ovvero l'elemento ***Task***, descritto da 5 campi fondamentali che rappresentano un processo all'interno della nostra implementazione:

1. *ID* : Un numero intero univoco che viene automaticamente assegnato alla creazione del task.
2. *nameTask* : Nome del task, di massimo 8 caratteri, scelto dall'utente alla creazione.
3. *priority* : Numero intero che rappresenta la prioritá del task.
4. *remainingExe* : Numero intero che rappresenta il numero di esecuzioni rimanenti (burst) del task.
5. **nextTask* : Puntatore al task successivo

Sempre all'interno di *taskmanager.h*2 vi sono le implementazioni delle operazioni che il nostro scheduler sará in grado di effettuare, definite dalle seguenti funzioni:

- `setExeNumber(void)` : Permette l'inserimento del numero di esecuzioni rimanenti n, effettuando i controlli sulla legalitá dell'input ($1 < n < 99$).
- `setPriority(void)` : Permette l'inserimento della prioritá p, effettuando i controlli sulla legalitá dell'input ($1 < p < 9$).
- `setTaskName(Task*)` : Permette l'inserimento del nome del task, effettuando i controlli sulla lunghezza massima della stringa inserita (al massimo 8 caratteri).
- `isEmptyTaskList(Task*)` : Esegue il controllo sulla lista di task, restituendo 0 nel caso sia vuota.
- `selectTask(Task*)` : Restituisce il task con il PID richiesto dall'utente, dopo aver eseguito la ricerca nella lista.
- `modifyPriority(Task*)` : Permette di modificare la prioritá del task selezionato.
- `modifyExecNumb(Task*)` : Permette di modificare il numero di esecuzioni rimanenti del task selezionato.

- `newTaskElement(Task*,int)` : Permette la creazione di un nuovo task, allocandolo in memoria con l'utilizzo di una `malloc`.
- `printTask(Task*)` : Esegue la stampa degli elementi del task coerentemente con la richiesta nella specifica dell'esercizio.
- `printListTask(Task*)` : Esegue la stampa dell'intera lista dei task, richiamando la funzione `printTask`.
- `deleteTask(Task*, Task*)` : Permette l'eliminazione di un task dalla lista, semplicemente collegando il puntatore *nextTask* dell'elemento precedente al task successivo a quello che deve essere eliminato.
- `executeTask(Task*)` : Esegue il task in testa alla coda, eseguendo i controlli sul numero di esecuzioni rimanenti.

Le operazioni legate allo scheduling sono state poi affidate a *scheduler.c*3, il quale contiene le funzioni:

- `getChoice(void)` : Stampa il menu di scelta delle operazioni eseguibili e restituisce la risposta data in input dall'utente.
- `switchPolicy(char)` : Permette di modificare la politica di scheduling, passando da prioritá ad esecuzioni rimanenti.
- `sortListByPriority(Task*)` : Ordina la lista dei task per valori decrescenti della prioritá ($max(p) = 9$).
- `sortListByExecution(Task*)` : Ordina la lista dei task per valori decrescenti del numero di esecuzioni rimanenti ($max(n) = 99$).
- `swapTask(Task*, Task*, Task*)` : Permette l'inversione dell'ordine di due task.
- `main()` : Main del programma.

1.2 Codice

```
#ifndef CONFIG_H_
#define CONFIG_H_

#define POINTSHEAD "
    .....................................................................\n\r"
#define SEPARATOR "+----+----------+----------+--------------------+\n\r"

#endif /* CONFIG_H_ */
```

Listing 1: config.c

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

#include "config.h"

typedef struct TaskElement {
  int ID;
  char nameTask[9]; // the ninth element of the name must be
  int priority;
  int remainingExe;
  struct TaskElement *nextTask;
} Task;

int setExeNumber(void);
int setPriority(void);
void setTaskName(Task*);
int isEmptyTaskList(Task*);
Task* selectTask(Task*);
void modifyPriority(Task*);
void modifyExecNumb(Task*);
```

```
22 Task* newTaskElement(Task*, int);
23 void printTask(Task*);
24 void printListTasks(Task*);
25 Task* deleteTask(Task*, Task*);
26 int executeTask(Task*);
27
28 /*
29 *
30 * PURPOSE : Setter remainingExe member of struct TaskElement, this function is use by
       newTaskElement
31 * RETURN : int -> number of remaining execution
32 *
33 */
34 int setExeNumber() {
35   int exeNum = 0;
36   do {
37     printf("\n\rInsert the number of remaning executions : ");
38     scanf("%i", &exeNum);
39     if ((exeNum < 0) || (exeNum > 99)) {
40       printf("\n\rError! It must be a number between 1 and 99. \n\r");
41     }
42   } while ((exeNum <= 0) || (exeNum > 99));
43   return exeNum;
44 }
45
46 /*
47 *
48 * PURPOSE : Setter priority member of struct TaskElement, this function is use by
       newTaskElement
49 * RETURN : int -> number of priority
50 *
51 */
52 int setPriority() {
53   int priority = 0;
54   do {
55     printf("\n\rInsert the priority (ascending order): ");
56     scanf("%i", &priority);
57     if (((priority < 0) || (priority > 9))) {
58       printf("\n\rError! It must be a number between 1 and 9\n\r");
59     }
60   } while ((priority < 0) || (priority > 9));
61   return priority;
62 }
63
64 /*
65 *
66 * PURPOSE : Setter name member of struct TaskElement, this function is use by
       newTaskElement
67 * PARAMS : Task* actualTask -> pointer of the task that want to set name
68 * RETURN : void
69 *
70 */
71 void setTaskName(Task *actualTask) {
72   char name[256];
73   do {
74     printf("\n\rName this task (max 8 chars) : ");
75     scanf("%s", name);
76     if (strlen(name)>0 && strlen(name)<=8) {
77       strcpy(actualTask->nameTask, name);
78       for (int i = strlen(name); i<=9 ;i++) {
79         actualTask->nameTask[i] = '\0'; // this for cycle set all character of the name
      task with the null character
80       }
81       return;
82     } else {
83       printf("\n\rThe name of the task must be less than 8");
84     }
85   } while (strlen(name)>8);
86   printf("Something went wrong\n\rThe name of the task it will be setted -default-");
87   strcpy(actualTask->nameTask,"default\0");
88   return;
89 }
90
```

```
91 /*
92 *
93 * PURPOSE : Check if the list is empty
94 * PARAMS : Task* firstTask -> pointer of the first (head) task of the list
95 * RETURN : int -> return 0 if the list is empty, return 1 if the list is not empty
96 *
97 */
98 int isEmptyTaskList(Task *firstTask) {
99   return !(firstTask->ID);
100 }
101
102 /*
103 *
104 * PURPOSE : It ask the ID of the Task and finds the Task with that ID
105 * PARAMS : Task* firstTask -> pointer of the first (head) task of the list
106 * RETURN : Task* -> return the pointer of the task found
107 *
108 */
109 Task* selectTask(Task* firstTask) {
110   int id;
111   printf("Select the task...\nInsert the ID : ");
112   scanf("%d", &id);
113   while (firstTask->ID != id) {
114     firstTask = firstTask->nextTask;
115     if (firstTask == NULL) {
116       printf("\n\rError! No tasks with this ID!\n\r");
117       return firstTask;
118     }
119   }
120   return firstTask;
121 }
122
123 /*
124 *
125 * PURPOSE : Modify the priority of the task
126 * PARAMS : Task* thisTask -> pointer of the task to change
127 * RETURN : void
128 *
129 */
130 void modifyPriority(Task *thisTask) {
131   thisTask = selectTask(thisTask);
132   if (thisTask == NULL) {
133     return;
134   }
135   thisTask->priority = setPriority();
136   return;
137 }
138
139 /*
140 *
141 * PURPOSE : Modify the execution number of the task
142 * PARAMS : Task* thisTask -> pointer of the task to change
143 * RETURN : void
144 *
145 */
146 void modifyExecNumb(Task *thisTask) {
147   thisTask = selectTask(thisTask);
148   if (thisTask == NULL) {
149     return;
150   }
151   thisTask->remainingExe = setExeNumber();
152   return;
153 }
154
155 /*
156 *
157 * PURPOSE : Allocate a new item in the list
158 * PARAMS : Task* actualTask -> pointer of the last task of the list
159 * PARAMS : int idT -> the id of the new task
160 * RETURN : Task* -> pointer of the new last task of the list
161 *
162 */
163 Task* newTaskElement(Task *actualTask, int idT) {
```

```
164   actualTask->ID = idT;
165   setTaskName(actualTask);
166   actualTask->priority = setPriority();
167   actualTask->remainingExe = setExeNumber();
168   (*actualTask).nextTask = malloc(sizeof(Task));
169   return (*actualTask).nextTask;
170 }
171
172 /*
173 *
174 * PURPOSE : Print a single Task
175 * PARAMS : Task* thisTask -> pointer of the task to print
176 * RETURN : void
177 *
178 */
179 void printTask(Task *thisTask) {
180   printf("| %d  + %d          + %s         + %d                   | \n\r",
181       thisTask->ID, thisTask->priority , thisTask->nameTask,
182       thisTask->remainingExe);
183   printf(SEPARATOR);
184 }
185
186 /*
187 *
188 * PURPOSE : Print the list of the task
189 * PARAMS : Task* first -> printing start from this task
190 * RETURN : void
191 *
192 */
193 void printListTasks(Task *first) {
194   printf(SEPARATOR);
195   printf("| ID + PRIORITY + TASK NAME + REMAINING EXEC |\n\r");
196   printf(SEPARATOR);
197   Task* tmp = first;
198   while (tmp->ID != 0) {
199     printTask(tmp);
200     tmp = (*tmp).nextTask;
201   }
202 }
203
204 /*
205 *
206 * PURPOSE : Delete a Task
207 * PARAMS : Task* first -> pointer of the first task of the list
208 * PARAMS : Task* thisTask -> pointer of the task to delete
209 * RETURN : Task* -> return the pointer of the first task of the list
210 *
211 */
212 Task* deleteTask(Task *first , Task *thisTask) {
213   if (thisTask != NULL) {
214     Task *tmpTask = first;
215     if (thisTask == first) {
216       tmpTask = thisTask->nextTask;
217       thisTask->ID = thisTask->priority = thisTask->remainingExe = 0;
218       strcpy(thisTask->nameTask, "\0");
219       thisTask->nextTask = NULL;
220       return tmpTask;
221     } else {
222       while (tmpTask->nextTask == NULL) {
223         if (tmpTask->nextTask == thisTask) {
224           tmpTask->nextTask = thisTask->nextTask;
225           thisTask->ID = thisTask->priority = thisTask->remainingExe =
226               0;
227           strcpy(thisTask->nameTask, "\0");
228           thisTask->nextTask = NULL;
229           return first;
230         }
231         tmpTask = tmpTask->nextTask;
232       }
233     }
234   }
235   printf("There is no task to delete!\n\r");
236   return first;
```

```
237 }
238
239 /*
240 *
241 * PURPOSE : Execute a Task
242 * PARAMS : Task* thisTask -> pointer of the task to execute
243 * RETURN : Task* -> return the number of the remaining execution
244 *
245 */
246 int executeTask(Task *thisTask) {
247   if (thisTask != NULL) {
248     thisTask->remainingExe -= 1;
249     return thisTask->remainingExe;
250   } else if (thisTask->remainingExe == 0) {
251     printf("This task has no more executions to be done\n\r");
252     return 0;
253   }
254   printf("There is no task to execute!\n\r");
255   return 0;
256 }
257
258 /*
259 *
260 * PURPOSE : To find previous Task
261 * PARAMS : Task* first -> pointer of the first task of the list
262 * PARAMS : Task* thisTask -> pointer of the following task to find
263 * RETURN : Task* -> return prevoius task of thisTask
264 *
265 */
266 Task* findPreviousTask(Task* first, Task* thisTask) {
267   if (first==thisTask) {
268     printf("This Task have no Previous\n\r");
269     return first;
270   } else if (thisTask==NULL) {
271     printf("This Task is NULL\n");
272     return first;
273   } else {
274     Task* tempTask = first;
275     while (tempTask->nextTask!=thisTask && tempTask->nextTask!=NULL) {
276       tempTask = tempTask->nextTask;
277     }
278     if (tempTask->nextTask==NULL) {
279       printf("No Task found\n");
280     }
281     return tempTask;
282   }
283 }
```

Listing 2: Task Manager

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "taskmanager.h"
5
6 int getChoice(void);
7 char switchPolicy(char pol);
8 Task* sortListByPriority(Task*);
9 Task* sortListByExecution(Task*);
10 Task* swapTask(Task*, Task*, Task*);
11
12 int main() {
13   int idTraker = 1;
14   int flag = 1;
15   char policy = 'p';
16   Task *firstTask = malloc(sizeof(Task));
17   Task *lastTask = NULL; // the last Task is always empty
18   printf(POINTSHEAD);
19   printf("                This is a process scheduler\n\r");
20   printf(POINTSHEAD);
21   while (flag == 1) {
22     Task *tmpTask;
23     switch (getChoice()) {
```

```
24      case 0:
25        printf("Bye!\n\r");
26        return 0;
27      case 1:
28        if (firstTask->ID == 0) {
29          lastTask = newTaskElement(firstTask, idTraker);
30        } else {
31          lastTask = newTaskElement(lastTask, idTraker);
32          if (policy == 'p') {
33            firstTask = sortListByPriority(firstTask);
34          } else if (policy == 'e') {
35            firstTask = sortListByExecution(firstTask);
36          }
37        }
38        idTraker += 1;
39        break;
40      case 2:
41        if (!isEmptyTaskList(firstTask)) {
42        tmpTask = findPreviousTask(firstTask, lastTask);
43          if (executeTask(tmpTask) == 0) {
44            lastTask = tmpTask;
45            lastTask->ID = lastTask->priority = 0;
46            strcpy(lastTask->nameTask,"/0");
47            lastTask -> nextTask = NULL;
48          }
49          printf("\n\r");
50        }
51        break;
52      case 3:
53        if (!isEmptyTaskList(firstTask)) {
54          tmpTask = selectTask(firstTask);
55          if (executeTask(tmpTask) == 0) {
56            firstTask = deleteTask(firstTask,tmpTask);
57          }
58        }
59        break;
60      case 4:
61        if (!isEmptyTaskList(firstTask)) {
62          firstTask = deleteTask(firstTask,selectTask(firstTask));
63        }
64        break;
65      case 5:
66        if (!isEmptyTaskList(firstTask)) {
67          modifyPriority(firstTask);
68          if (policy == 'p') {
69            firstTask = sortListByPriority(firstTask);
70          }
71        }
72        break;
73      case 6:
74        policy = switchPolicy(policy);
75        if (policy == 'p') {
76          firstTask = sortListByPriority(firstTask);
77        } else if (policy == 'e') {
78          firstTask = sortListByExecution(firstTask);
79        }
80        break;
81      case 7:
82        if (!isEmptyTaskList(firstTask)) {
83          modifyExecNumb(firstTask);
84          if (policy == 'e') {
85            firstTask = sortListByExecution(firstTask);
86          }
87        }
88        break;
89      default:
90        flag = 0;
91        break;
92      }
93      if (!isEmptyTaskList(firstTask)) {
94        printf("\n\rScheduling Policy: ");
95        if (policy == 'p') {
96          printf("PRIORITY \n\r");
```

```
97        } else if (policy == 'e') {
98          printf("REMAINING EXECUTIONS \n\r");
99        }
100       printListTasks(firstTask);
101     } else {
102       printf("\n\rList is empty! Please insert a task first...\n\r");
103     }
104   }
105   return 0;
106 }
107
108 /*
109 *
110 * PURPOSE : Print menu and get the choice
111 * RETURN : int -> choice of the menu
112 *
113 */
114 int getChoice() {
115   printf("\n\rPlease select an option:\n\r");
116   printf(" 0) Exit\n\r 1) Create a new task\n\r 2) Execute a task\n\r");
117   printf(
118       " 3) insert ID task and execute\n\r 4) Delete a task\n\r 5) Modify the PRIORITY of
      a task\n\r");
119   printf(
120       " 6) Switch policy (default : PRIORITY)\n\r 7) Modify the REMAINING EXECUTIONS of a
       task");
121   int res = 0;
122   printf("\n\r> ");
123   scanf("%i", &res);
124   return res;
125 }
126
127 /*
128 *
129 * PURPOSE : Switch the policy of the scheduler
130 * PARAMS : char pol -> actual policy of the scheduler
131 * RETURN : char -> new policy of the scheduler
132 *
133 */
134 char switchPolicy(char pol) {
135   printf("\n\rYou switched the policy of scheduling from ");
136   if (pol == 'p') {
137     printf("PRIORITY to REMAINING EXECUTIONS\n\r");
138     return 'e';
139   } else if (pol == 'e') {
140     printf("REMAINING EXECUTIONS to PRIORITY \n\r");
141     return 'p';
142   }
143   return 'p';
144 }
145
146 /*
147 *
148 * PURPOSE : Sort list by priority (highest priority, task most important)
149 * PARAMS : Task* headTask -> pointer of the first task of the list
150 * RETURN : Task* -> new pointer of first (head) task
151 *
152 */
153 Task* sortListByPriority(Task *headTask) {
154   Task *tempTask = headTask;
155   Task *previousTempTask = tempTask;
156   int flag = 0;
157   while (!flag) {
158     flag = 1;
159     tempTask = headTask;
160     previousTempTask = tempTask;
161     while (tempTask->ID != 0) {
162       if (tempTask->priority < tempTask->nextTask->priority) {
163         if (tempTask == headTask) {
164           headTask = swapTask(headTask, tempTask, tempTask->nextTask);
165         } else {
166           previousTempTask = swapTask(previousTempTask, tempTask,
167               tempTask->nextTask);
```

```
168           }
169           flag = 0;
170         }
171         previousTempTask = tempTask;
172         tempTask = tempTask->nextTask;
173       }
174     }
175     return headTask;
176 }
177
178 /*
179 *
180 * PURPOSE : Sort list by priority (lowest remaining execution, task most important)
181 * PARAMS : Task* headTask -> pointer of the first task of the list
182 * RETURN : Task* -> new pointer of first (head) task
183 *
184 */
185 Task* sortListByExecution(Task* headTask) {
186   Task *tempTask = headTask;
187   Task *previousTempTask = tempTask;
188   int flag = 0;
189   while (!flag) {
190     flag = 1;
191     tempTask = headTask;
192     previousTempTask = tempTask;
193     while (tempTask->ID != 0) {
194       if ((tempTask->remainingExe > tempTask->nextTask->remainingExe)
195           && (tempTask->nextTask->remainingExe != 0)) {
196         if (tempTask == headTask) {
197           headTask = swapTask(headTask, headTask, headTask->nextTask);
198         } else {
199           previousTempTask = swapTask(previousTempTask, tempTask,
200               tempTask->nextTask);
201         }
202         flag = 0;
203       }
204       previousTempTask = tempTask;
205       tempTask = tempTask->nextTask;
206     }
207   }
208   return headTask;
209 }
210
211 /*
212 *
213 * PURPOSE : Swap two task
214 * PARAMS : Task* previousTask -> pointer of the first task of the list
215 * PARAMS : Task* taskSwap1 -> pointer of first task to swap
216 * PARAMS : Task* taskSwap2 -> pointer of second task to swap
217 * RETURN : Task* -> pointer of the previous task
218 *
219 */
220 Task* swapTask(Task *previousTask, Task *taskSwap1, Task *taskSwap2) {
221   if (previousTask != taskSwap1) {
222     previousTask->nextTask = taskSwap2;
223     taskSwap1->nextTask = taskSwap2->nextTask;
224     taskSwap2->nextTask = taskSwap1;
225     return previousTask;
226   }
227   taskSwap1->nextTask = taskSwap2->nextTask;
228   taskSwap2->nextTask = taskSwap1;
229   return taskSwap2;
230 }
```

Listing 3: scheduler.c

2 Secondo Esercizio : Esecutore di comandi

2.1 Descrizione dell'implementazione

L'obiettivo del secondo esercizo é quello di creare un esecutore di comandi `UNIX` che scriva, sequenzialmente o parallelamente, l'output dell'esecuzione su di un file.
Tutte le funzionalitá del programma sono incluse all'interno della libreria *functions.h*5 e fanno uso a loro volta della libreria *unistd.h*. La funzione `initDataFolder()` si occupa di creare la cartella ed inserirvi il file di output. Essa viene generata all'interno della directory *"../commandexe/data/[pid]"* dove il *pid* é il process ID del chiamante in questione, ritornato dal `getpid()`. Il comando inserito dall'utente viene poi eseguito attraverso una `popen()`, la quale apre uno stream di scrittura/lettura su di una pipe, inserendovi l'output del comando. La funzione `execCommandAndLog(char,int)` genera due *char[]*, rispettivamente il *path* ed il *filename*, quest'ultimo viene nominato attraverso il pid e l'indice di esecuzione, come richiesto dalla specifica di implementazione. Viene poi eseguito il comando ed il log dell'output: l'esecuzione viene affidata ancora una volta ad una `popen()` mentre la scrittura dell'output viene eseguita mediante le usuali funzioni dello `stdin` attraverso il descrittore di file generato precedentemente. Il codice viene eseguito nel *cmd.c*4, all'interno del quale vi é un ciclo while che itera fino a quando non viene inserita la stringa vuota dall'utente.

2.2 Codice

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "functions.h"

#define MAX_CMD_LEN 100

int main() {
  int k = 1;
  initDataFolder();
  while (1) {
    char cmd[MAX_CMD_LEN] = "";
    printf("\nEnter the %d-cmd: ", k);

    //read chars until \n
    scanf("%[^\n]", cmd);
    getchar();
    printf("Cmd entered : %s\n", cmd);
    if (strlen(cmd) == 0) {
      printf("Bye!\n");
      exit(1);
    }
    execCommandAndLog(cmd, k);
    k = k + 1;
  }

  return (0);
}
```

Listing 4: cmd.c

```c
/*
*
* PURPOSE : Create the data folder to store outputs
*
* RETURN : void
*
*/
int initDataFolder() {
  char cmd[30];
  FILE *fp;
  sprintf(cmd, "%s%i", "mkdir -p ../commandexe/data/", getpid());
  fp = popen(cmd, "r");
  if (fp == NULL) {
    printf("[Error] - Error initialing process folder\n");
    exit(1);
```

```
16    }
17    return 0;
18  }
19
20  /*
21  *
22  * PURPOSE : Function that execute the c command and log the output in ../commandexe/data
        /[pid]/out.[index]
23  *
24  * PARAMS : char* -> command string
25  * PARAMS : int -> index of out.[index] log
26  * RETURN : int -> 0
27  *
28  */
29  int execCommandAndLog(char* c, int index) {
30    FILE *fp;
31    char path[1035];
32    char filename[7];
33
34    sprintf(filename, "data/%i/%s.%i", getpid(), "out", index);
35    FILE *f = fopen(filename, "w");
36    if (f == NULL) {
37      printf("[Error] - Error opening file!\n");
38      exit(1);
39    }
40
41    // command open to read
42    sprintf(c, "%s %s", c, "2>&1");
43    fp = popen(c, "r");
44
45    if (fp == NULL) {
46      fprintf(f, "[Error] - Error executing the command\n");
47    }
48
49    // read the output a line at a time - output it.
50    while (fgets(path, sizeof(path) - 1, fp) != NULL) {
51      fprintf(f, "%s", path);
52    }
53
54    // closing files
55    pclose(fp);
56    fclose(f);
57
58    return 0;
59  }
```

Listing 5: functions.h

3 Terzo Esercizio : Message passing

3.1 Descrizione dell'implementazione

L'obiettivo del terzo esercizio é l'implementazione di un client e un server che comunicano tramite pipe con nome ed eseguono routine dipendentemente dal tipo di richiesta.
Inizialmente viene eseguito il *server.c*12 che racchiude le sue funzionalitá nei file inclusi *functions.server.h*8 e *listmanage.h*9. Il client invece include la libreria *functions.client.h*7. Sia *client.c* che *server.c* condividono due librerie, rispettivamente *functions.inc.h*10 e *config.h*6.
Una volta mandato in esecuzione, il server aspetta comandi dal client in linea con il formato del protocollo definito. Il protocollo definito per permettere la comunicazione tra client e server é composto da una stringa in cui le informazioni sono separate da spazio. Il server quindi si preoccupa di splittare la stringa ricevuta per avere tutte le informazioni della richiesta. Ogni richiesta é composta da una prima informazione che identifica l'ID del comando (1-5), la seconda invece l'ID del client richiedente. L'unica eccezione é fatta dal comando n °3 che oltre a queste due informazioni segue con l'ID del client destinatario e il messaggio da inviare. Abbiamo definito che l'ID di ogni client corrisponde al suo PID. Per ogni funzionalitá é definito il formato per richiederla e un esempio:

1. Connessione, con la quale il client si registra presso il server. → "1 [pid_richiedente]" → "1 5555"
2. Richiesta elenco ID dei client registrati, con la quale si richiede al server l'elenco dei client attualmente registrati. → "1 [pid_richiedente]" → "1 5555"
3. Invio di un messaggio testuale a un altro client o a un insieme di client (specificandone l'ID). → "1 [pid_richiedente] [pid_destinatario] [messaggio]" → "1 5555 4444 Questo é un messaggio da recapitare"
4. Disconnessione, con la quale il client richiede la cancellazione della registrazione presso il server. → "1 [pid_richiedente]" → "1 5555"
5. Uscita dal programma. → "1 [pid_richiedente]" → "1 5555"

Infrastrutture e segnali previsti:
La comunicazione tra client e server per richiedere l'esecuzione di una routine avviene tramite la pipe con nome "data/pipe" (definita nel *config.h*6). Il server quando riceve una richiesta di connessione o di disconnessione aggiorna la sua lista concatenata nella quale si mantiene i client connessi. Le funzioni per interagire con questa lista sono definite in listmanage.h.
Quando il server ha la necessità di inviare informazioni ad uno specifico client si crea una pipe "data/[PID_DEST]", scrive sulla pipe e segnala l'evento al relativo client tramite segnale **SIGUSR1**. Il segnale **SIGUSR2** invece é inviato dal server ad un client quando quest'ultimo ha richiesto l'invio di un messaggio ad un client non connesso al server. Un altro segnale intercettato sia dal client che dal server é il **SIGINT**. Quando questo segnale viene intercettato da un client, esso si disconnette dal server prima di terminare mentre in caso sia intercettato dal server fa terminare tutti I client connessi inviandogli lo stesso segnale **SIGINT**.

3.2 Codice

```c
#define DEBUG 1 //debug mode
#define CMD_PIPE_NAME "data/pipe" //path to store the main named pipe
#define PIPES_PATH "data/" //path to store clients pipes
#define MAX_MSG_LEN 100
#define MAX_PID_LEN 100
```

Listing 6: config.h

```c
/*
*
* PURPOSE : Show menu options
*
* RETURN : void
*
*/
void menu(){
```

```
9       int menuChoice;
10
11      printf("Menu 1\n");
12      printf("~~~~~~~~~~\n");
13      printf("1. Connect to server.\n");
14      printf("2. Get clients connected to server.\n");
15      printf("3. Write to client/s.\n");
16      printf("4. Disconnect from server.\n");
17      printf("5. Exit.\n");
18      scanf("%d", &menuChoice);
19      printf("\n");
20      switch ( menuChoice ){
21          case 1:
22              connect();
23              break;
24          case 2:
25              getClientsID();
26              break;
27          case 3:
28              sendMessage();
29              break;
30          case 4:
31              disconnect();
32              break;
33          case 5:
34              disconnect();
35              clientExit();
36              break;
37          default:
38              printf("Please.. is not a joke. \n");
39      }
40
41      printf("\n");
42      menu();
43  }
44
45  /*
46  *
47  * PURPOSE : Connect to server
48  *
49  * RETURN : void
50  *
51  */
52  void connect(){
53      if(connected == 1){
54          printf("%s\n", "Already connected");
55          return;
56      }
57
58      printf("%s\n", "Connected");
59
60      char str[7];
61
62      sprintf(str, "1 %d", getpid()); // puts string into buffer
63
64      write(fd, str, sizeof(str));
65
66      connected = 1;
67
68      return;
69  }
70
71  /*
72  *
73  * PURPOSE : Request the list of the clients connected to the server
74  *
75  * RETURN : void
76  *
77  */
78  void getClientsID(){
79      char str[7];
80      char *s_pid;
81
```

```
82      sprintf(s_pid, "%d", getpid());
83      sprintf(str, "2 %s", s_pid); // puts string into buffer
84
85      write(fd, str, sizeof(str));
86
87      return;
88  }
89
90  /*
91  *
92  * PURPOSE : Clear old chars on a stream
93  *
94  * RETURN : void
95  *
96  */
97  void clear_stream(FILE *in){
98      int ch;
99
100     clearerr(in);
101
102     do
103         ch = getc(in);
104     while (ch != '\n' && ch != EOF);
105
106     clearerr(in);
107 }
108
109 /*
110 *
111 * PURPOSE : Get the lenght of an int
112 *
113 * PARAMS : int -> int to measure
114 * RETURN : int -> lenght of the int passed
115 *
116 */
117 int get_int_len(int value){
118   int l=1;
119   while(value>9){ l++; value/=10; }
120   return l;
121 }
122
123 /*
124 *
125 * PURPOSE : Send message to client(s) menu
126 *
127 * RETURN : void
128 *
129 */
130 void sendMessage(){
131     char msg[MAX_MSG_LEN];
132     int confirm;
133     int i = 1;
134
135     //read the string message from STDIN
136     clear_stream(stdin);
137     printf("\nEnter the message: ");
138     scanf("%[^\n]s", msg);
139
140     //output the message to the STDOUT
141     printf("Message entered : %s\n", msg);
142
143     //message confirm
144     printf("\nPress 1 to confirm, others to delete the message.. ");
145     if(scanf("%d", &confirm)==1 && confirm==1){
146         //confirmed
147         if(DEBUG)
148             printf("%s\n", "Start sending..");
149
150         int k = 1;
151         int pid;
152
153         while (1) {
154             printf("\nEnter the %dth pid destination (letters to return to Menu): ", k);
```

```
155
156             if(scanf("%d", &pid)==1){
157                 if(DEBUG)
158                     printf("[DEBUG] pid entered: %d\n", pid);
159
160                 //send to pid
161                 int size = 5 + get_int_len(getpid()) + get_int_len(pid) + strlen(msg);
162                 char str[size];
163
164                 if(DEBUG)
165                     printf("[DEBUG] str size: %d\n", size);
166
167                 sprintf(str, "3 %d %d %s", getpid(), pid, msg); // puts string into
    buffer
168
169                 if(DEBUG)
170                     printf("[DEBUG] string to send: '%s'\n", str);
171
172                 write(fd, str, sizeof(str));
173
174
175                 k = k + 1;
176             }else{
177                 clear_stream(stdin);
178                 break;
179             }
180         }
181
182         if(DEBUG)
183             printf("%s\n", "[DEBUG] End sending");
184
185     }else{
186         clear_stream(stdin);
187         printf("%s\n", "Message aborted");
188     }
189
190
191     return;
192 }
193
194 /*
195 *
196 * PURPOSE : Disconnect from the server
197 *
198 * RETURN : void
199 *
200 */
201 void disconnect(){
202     if(connected == 0){
203         printf("\n%s\n", "Already disconnected");
204         return;
205     }
206
207     printf("\n%s\n", "Client disconnected");
208
209     char str[7];
210
211     sprintf(str, "4 %d", getpid()); // puts string into buffer
212
213     write(fd, str, sizeof(str));
214
215     connected = 0;
216
217     return;
218 }
219
220 /*
221 *
222 * PURPOSE : Function to exit from client execution
223 *
224 * RETURN : void
225 *
226 */
```

```
227 void clientExit(){
228     close(fd);
229     printf("%s\n", "Bye");
230     exit(0);
231 }
232 
233 /*
234 *
235 * PURPOSE : Manage SIGINT signal (CTRL+C)
236 *
237 * PARAMS : int -> signal number
238 * RETURN : void
239 *
240 */
241 void sigHandler_1(int signumber) {
242     if(signumber == SIGINT){
243         printf("\n{SIGNAL}\n");
244 
245         if(DEBUG)
246             printf("    [DEBUG] SIGINT catched\n");
247 
248         printf("{/SIGNAL}\n");
249         disconnect();
250         clientExit();
251     }
252     return;
253 }
254 
255 /*
256 *
257 * PURPOSE : Manage SIGUSR1 signal. When a SIGUSR1 is catched the client read from its
      named pipe
258 *
259 * PARAMS : int -> signal number
260 * RETURN : void
261 *
262 */
263 void sigHandler_2(int signumber){
264 
265     if(signumber == SIGUSR1) {
266         printf("\n{SIGNAL}\n");
267         if(DEBUG)
268             printf("    [DEBUG] SIGUSR1 catched\n");
269 
270         //READ RESPONSE FROM PROCESS PIPE
271         char s_pid[10];
272         int fd_client;
273         char pipeName[20];
274         char response[100];
275         char *p_pipeName;
276 
277         sprintf(s_pid, "%d", getpid());
278 
279 
280         p_pipeName = concat(PIPES_PATH, s_pid);
281 
282         // sprintf(p_pipeName,"%s %s",PIPES_PATH, s_pid);
283 
284         strcpy(pipeName, p_pipeName); /* BANG!!! */
285         fd_client = open(pipeName, O_RDONLY); /* Open it for writing */
286 
287         if(DEBUG)
288             printf("    [DEBUG] Reading from: %s ...\n", pipeName);
289 
290         readLine(fd_client, response);
291         printf("    Received: %s\n", response);
292 
293         close(fd_client);
294 
295         char c[50];
296         sprintf(c, "rm -f  %s",pipeName);
297         FILE *fp = popen(c, "r");
298 
```

```
299         // closing files
300         pclose(fp);
301         printf("{/SIGNAL}\n");
302 
303     }
304 }
305 
306 /*
307 *
308 * PURPOSE : Manage SIGUSR2 signal. When a SIGUSR2 is catched the client had tried to send a message to a non existing client
309 *
310 * PARAMS : int -> signal number
311 * RETURN : void
312 *
313 */
314 void sigHandler_3(int signumber){
315 
316     if(signumber == SIGUSR2) {
317         printf("\n{SIGNAL}\n");
318 
319         if(DEBUG)
320             printf("    [DEBUG] SIGUSR2 catched.\n");
321 
322         printf("    Pid not found.\n");
323 
324         printf("{/SIGNAL}\n");
325     }
326 }
```

Listing 7: functions.client.h

```
1  /*
2  *
3  * PURPOSE : Create the data folder to store named pipes
4  *
5  * RETURN : void
6  *
7  */
8  int initDataFolder() {
9    FILE *fp;
10 
11   fp = popen("mkdir -p data/", "r");
12   if (fp == NULL) {
13     printf("[Error] - Error initialing process folder\n");
14     exit(1);
15   }
16 
17     pclose(fp);
18 
19   return 0;
20 }
21 
22 /*
23 *
24 * PURPOSE : Manage SIGINT signal (CTRL+C)
25 *
26 * PARAMS : int -> signal number
27 * RETURN : void
28 *
29 */
30 void sigHandler_1(int signumber) {
31     if(signumber == SIGINT){
32         printf("\n{SIGNAL}\n");
33         if(DEBUG)
34             printf("  [DEBUG] SIGINT catched\n");
35 
36         int killed;
37         remove(CMD_PIPE_NAME);
38         // here we have to insert a while to send SIGINT to all clients to disconnect to this server
39       killed = clients_killer(n);
40       if(DEBUG)
```

```
41          printf("  [DEBUG] killed %d clients\n", killed);
42          printf("{/SIGNAL}\n");
43        exit(0);
44      }
45      return;
46  }
47
48  /*
49  *
50  * PURPOSE : Send text to a pid client. Create a named pipe with the destination pid value
        , write the text and send.
51  *
52  * PARAMS : char* -> destination pid
53  * PARAMS : char* -> string to send
54  * RETURN : void
55  *
56  */
57  void sendTextToClient(char* pid, char* p_textToSend){
58    char pipeName[20];
59    char* p_pipeName;
60    int fd_client;
61    char c_textToSend[strlen(p_textToSend)+1];
62
63    strcpy( c_textToSend, p_textToSend );
64
65    p_pipeName = (char*)malloc(strlen(PIPES_PATH)+strlen(pid)+1);
66    p_pipeName = concat(PIPES_PATH, pid);
67    strcpy(pipeName, p_pipeName); /* BANG!!! */
68    mknod(pipeName, S_IFIFO|0666, 0); /* Create named pipe */
69    if(DEBUG)
70      printf("[DEBUG] Writing '%s' (size: %lu) to '%s'\n", c_textToSend, sizeof(
        c_textToSend), pipeName);
71    fd_client = open(pipeName, O_RDWR); /* Open it for writing */
72
73    int res = write(fd_client, c_textToSend, sizeof(c_textToSend));
74    if(DEBUG)
75      printf("[DEBUG] Written %d elements\n", res);
76    // close(fd_client); /* Close pipe */
77
78  }
```

Listing 8: functions.server.h

```
1  //struct
2  struct node{
3      char data[10];
4      struct node *next;
5  }*head;
6
7
8  /*
9  *
10 * PURPOSE : Get the number of clients connected
11 *
12 * RETURN :  int -> number of clients in the list
13 *
14 */
15 int clients_count(){
16     struct node *n;
17     int c=0;
18     n=head;
19     while(n!=NULL){
20         n=n->next;
21         c++;
22     }
23     return c;
24 }
25
26 /*
27 *
28 * PURPOSE : Append a node to the end of the concat list
29 *
30 * PARAMS : char *pid -> pid to add in the list
```

```
31 * RETURN : void
32 *
33 */
34 void clients_append(char* pid){
35     struct node *temp,*right;
36     temp= (struct node *)malloc(sizeof(struct node));
37     strcpy(temp->data, pid);
38 
39     right=(struct node *)head;
40     while(right->next != NULL)
41         right=right->next;
42 
43     right->next =temp;
44     right=temp;
45     right->next=NULL;
46 }
47 
48 /*
49 *
50 * PURPOSE : Add the first node in the concat list
51 *
52 * PARAMS : char *pid -> pid to add in the list
53 * RETURN : void
54 *
55 */
56 void clients_add( char* num ){
57     struct node *temp;
58     temp=(struct node *)malloc(sizeof(struct node));
59     strcpy(temp->data, num);
60     // temp->data=num;
61     if (head== NULL){
62       head=temp;
63       head->next=NULL;
64     }
65     else{
66       temp->next=head;
67       head=temp;
68     }
69 }
70 
71 /*
72 *
73 * PURPOSE : Add the (i+1)-node in the concat list
74 *
75 * PARAMS : char *pid -> pid to add in the list
76 * RETURN : void
77 *
78 */
79 void clients_addafter(char* num, int loc){
80     int i;
81     struct node *temp,*left,*right;
82     right=head;
83     for(i=1;i<loc;i++){
84         left=right;
85         right=right->next;
86     }
87     temp=(struct node *)malloc(sizeof(struct node));
88     // temp->data=num;
89     strcpy(temp->data, num);
90 
91     left->next=temp;
92     left=temp;
93     left->next=right;
94 }
95 
96 /*
97 *
98 * PURPOSE : Insert a pid in the concat list
99 *
100 * PARAMS : char *pid -> pid to add in the list
101 * RETURN : void
102 *
103 */
```

```
104 void clients_insert(char* pid){
105     int c=0;
106     struct node *temp;
107     temp=head;
108     if(temp==NULL){
109     // printf("%s\n", "ok");
110       clients_add(pid);
111     }
112     else{
113       while(temp!=NULL){
114           if(temp->data<pid)
115           c++;
116           temp=temp->next;
117       }
118       if(c==0)
119           clients_add(pid);
120       else if(c<clients_count())
121           clients_addafter(pid,++c);
122       else
123           clients_append(pid);
124     }
125 }
126
127 /*
128 *
129 * PURPOSE : Search a client in the concat list
130 *
131 * PARAMS : char *pid -> pid to add in the list
132 * RETURN : void
133 *
134 */
135 int clients_search(char* pid){
136     struct node *temp, *prev;
137     temp=head;
138     while(temp!=NULL){
139         if(strcmp(temp->data,pid)==0)
140             return 1;
141
142         prev=temp;
143         temp= temp->next;
144     }
145
146     return 0;
147 }
148
149 /*
150 *
151 * PURPOSE : Remove a client in the concat list
152 *
153 * PARAMS : char *pid -> pid to remove
154 * RETURN : int -> 1 if removed, 0 otherwise
155 *
156 */
157 int clients_delete(char* num){
158     struct node *temp, *prev;
159     temp=head;
160     while(temp!=NULL){
161         if(strcmp(temp->data,num)==0){
162             if(temp==head){
163                 head=temp->next;
164                 free(temp);
165                 return 1;
166             }
167             else{
168                 prev->next=temp->next;
169                 free(temp);
170                 return 1;
171             }
172         }
173         else{
174             prev=temp;
175             temp= temp->next;
176         }
```

```
177     }
178     return 0;
179 }
180 
181 /*
182 *
183 * PURPOSE : Get the pid list of the clients connnected
184 *
185 * PARAMS : struct node *r
186 * RETURN : char* -> string of clients connected
187 *
188 */
189 char*  clients_display(struct node *r){
190     char* clients = NULL;
191     char*  toAdd;
192 
193     clients = malloc(0);
194 
195     r=head;
196     if(r==NULL)
197         return "No Clients Connected";
198 
199     while(r!=NULL){
200         toAdd = &(r->data)[0];
201         sprintf(clients, "%s %s", clients, toAdd);
202         r=r->next;
203     }
204 
205     return clients;
206 }
207 
208 /*
209 *
210 * PURPOSE : Kill all clients connected to the server sending them a SIGINT signal
211 *
212 * PARAMS : struct node *r
213 * RETURN : int -> number of clients killed
214 *
215 */
216 int  clients_killer(struct node *r){
217     int pid;
218     int clientsKilled=0;
219 
220     r=head;
221     if(r==NULL)
222       return 0;
223 
224     while(r!=NULL){
225         pid = atoi(r->data);
226         if(DEBUG)
227             printf("    [DEBUG] pid to kill: %d\n",pid);
228 
229         kill(pid,SIGINT);
230         clientsKilled += 1;
231         r=r->next;
232     }
233 
234     return clientsKilled;
235 }
```

Listing 9: listmanage.h

```
1 /*
2 *
3 * PURPOSE : Concat two strings
4 *
5 * PARAMS : char* s1 -> first string
6 * PARAMS : char* s2 -> second string
7 * RETURN : char* -> string concatenated
8 *
9 */
10 char* concat(const char *s1, const char *s2){
11     int i_s1 = 0;
```

```
12      int i_s2 = 0;
13
14      if(&(s1)[0] != NULL)
15          i_s1 = strlen(s1);
16
17      if(&(s2)[0] != NULL)
18          i_s2 = strlen(s2);
19
20      char *result = malloc(i_s1+i_s2+1);//+1 for the zero-terminator
21
22      //in real code you would check for errors in malloc here
23      if(&(s1)[0] != NULL)
24          strcpy(result, s1);
25
26      if(&(s2)[0] != NULL)
27          strcat(result, s2);
28
29      return result;
30  }
31
32  /*
33  *
34  * PURPOSE : read a line from a file and write it in str
35  *
36  * PARAMS : int fd -> file descriptor
37  * PARAMS : char* str -> where the line read will go
38  * RETURN : int -> 1 if ok, 0 end-of-input
39  *
40  */
41  int readLine(int fd, char *str) {
42      /* Read a single '\0'-terminated line into str from fd */
43      /* Return 0 when the end-of-input is reached and 1 otherwise */
44      int n;
45      do { /* Read characters until '\0' or end-of-input */
46          n = read (fd, str, 1); /* Read one character */
47      } while (n > 0 && *str++ != '\0');
48      return (n > 0); /* Return false if end-of-input */
49  }
```

Listing 10: functions.inc.h

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6  #include <unistd.h>
7  #include <string.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <signal.h>
11 #include <math.h>
12 #include "config.h"
13
14 int connected = 0;
15 char * myfifo = CMD_PIPE_NAME;
16 int fd;
17
18 void menu();
19 void connect();
20 void getClientsID();
21 void sendMessage();
22 void disconnect();
23 void clientExit();
24 void ex_program(int);
25 void sigHandler_1(int);
26 void sigHandler_2(int);
27 void sigHandler_3(int);
28
29 #include "functions.inc.h"
30 #include "functions.client.h"
31
32 int main(){
```

```
33      if(DEBUG)
34          printf("[DEBUG] pid: %d\n", getpid());
35
36      signal(SIGINT, sigHandler_1);
37      signal(SIGUSR1 ,sigHandler_2);
38      signal(SIGUSR2 ,sigHandler_3);
39
40      fd = open(myfifo, O_WRONLY);
41
42      if(fd==-1){
43          printf("%s\n", "[ERR]: server is not running");
44          return 0;
45      }
46
47      printf("Hello. Welcome to the client.\n");
48      printf("Press RETURN key to continue...\n");
49      getchar();
50      menu();
51
52      return 0;
53  }
```

Listing 11: client.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6  #include <unistd.h>
7  #include <string.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <signal.h>
11 #include "config.h"
12
13 struct node *n;
14
15 int initDataFolder(void);
16 void sigHandler_1(int);
17 void sendTextToClient(char*,char*);
18
19 #include "functions.inc.h"
20 #include "listmanage.h"
21 #include "functions.server.h"
22
23 int main(){
24   initDataFolder();
25
26   signal(SIGINT, sigHandler_1);
27
28   int i_fd;
29   int i_pid_m;
30   int i_pid_d;
31
32   char cmd[100];
33   char pid[10];
34
35   char* p_pid_m;
36   char* p_pid_d;
37   char* p_msg;
38   char* p_clientsList;
39   char* p_token;
40
41   //init clients list
42     head=NULL;
43
44   int res = mknod(CMD_PIPE_NAME, S_IFIFO|0666, 0); /* Create named pipe */
45   if(res!=0){
46     printf("%s\n", "[ERR] Problem creating pipe");
47     return 1;
48   }
49
```

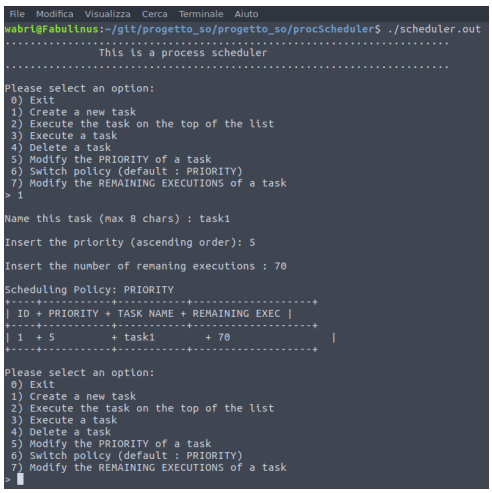
```
50  i_fd = open(CMD_PIPE_NAME, O_RDONLY); /* Open it for reading */
51  if(i_fd==-1){
52    printf("%s\n", "[ERR] Problem reading pipe");
53    return 1;
54  }
55
56  if(DEBUG)
57    printf("[DEBUG] Reading from pipe %s ..\n\n", CMD_PIPE_NAME);
58
59  while (readLine(i_fd, cmd)){
60    /* Receiving messages */
61        if(DEBUG)
62      printf("[DEBUG] Received: %s\n", cmd);
63
64    char* cmd_detected = strtok(cmd, " ");
65    // printf("%s\n", cmd_detected);
66    char* p_pid_m = strtok(NULL, " ");
67    // printf("%s\n", p_pid_m);
68
69
70    strcpy(pid, p_pid_m); /* BANG!!! */
71    i_pid_m = atoi(pid);
72
73    if(DEBUG)
74      printf("[DEBUG] cmd->%s | pid_m->%s\n", cmd_detected, p_pid_m);
75
76    switch (cmd_detected[0]){
77      case '1':
78        clients_insert(pid);
79        break;
80      case '2':
81        p_clientsList = clients_display(n);
82
83        sendTextToClient(pid, p_clientsList);
84
85        if(DEBUG)
86          printf("[DEBUG] SIGUSR1 TO PID %d\n", i_pid_m);
87
88        //notify client through SIGNAL SIGUSR1
89        kill(i_pid_m, SIGUSR1);
90        break;
91      case '3':
92
93        p_pid_d = strtok(NULL, " ");
94        p_msg = strtok(NULL, " ");
95
96        while(p_token = strtok(NULL, " "))
97          sprintf(p_msg, "%s %s", p_msg, p_token);
98
99
100       if(DEBUG){
101         printf("[DEBUG] pid_d->%s\n", p_pid_d);
102         printf("[DEBUG] msg received: %s\n", p_msg);
103       }
104
105
106       if(clients_search(p_pid_d)==1){
107         //if the client is connected, send the message
108         i_pid_d = atoi(p_pid_d);
109
110         sendTextToClient(p_pid_d, p_msg);
111         //notify client through SIGNAL SIGUSR1
112         kill(i_pid_d, SIGUSR1);
113       }else{
114         //otherwise notify the client who request to send this message
115         if(DEBUG)
116           printf("[DEBUG] Client %s is not connected\n", p_pid_d);
117
118         //send SIGUSR2 to client
119         kill(i_pid_m, SIGUSR2);
120
121       }
122
```

```
123             break;
124           case '4':
125           case '5':
126             clients_delete(pid);
127             break;
128       }
129
130       printf("\n\r");
131     }
132
133     if(DEBUG)
134       printf("%s\n", "End..");
135
136     close (i_fd); /* Close pipe */
137       remove(CMD_PIPE_NAME);
138 }
```

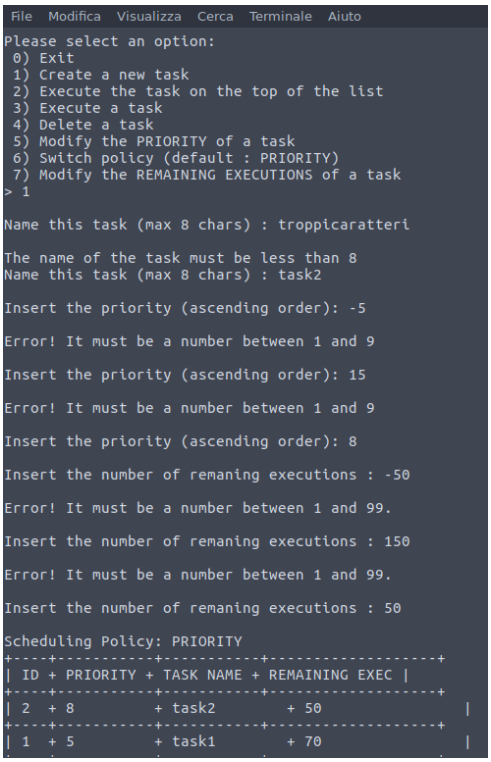
Listing 12: server.c

4 Evidenza del corretto funzionamento dei programmi

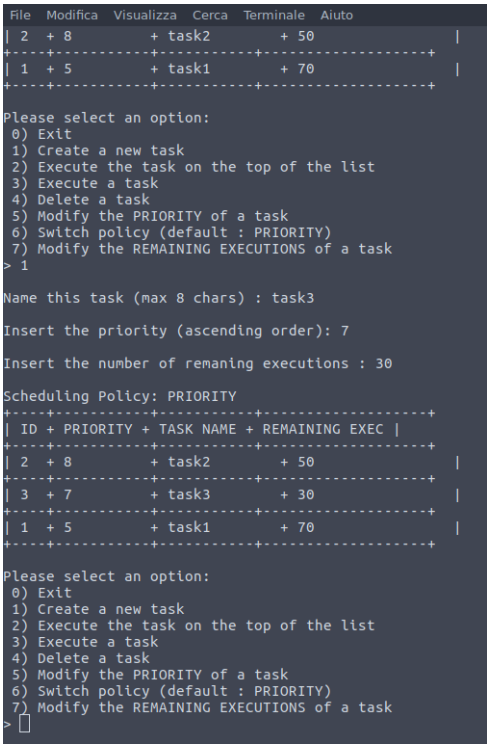
4.1 I Esercizio



(a) inserimento

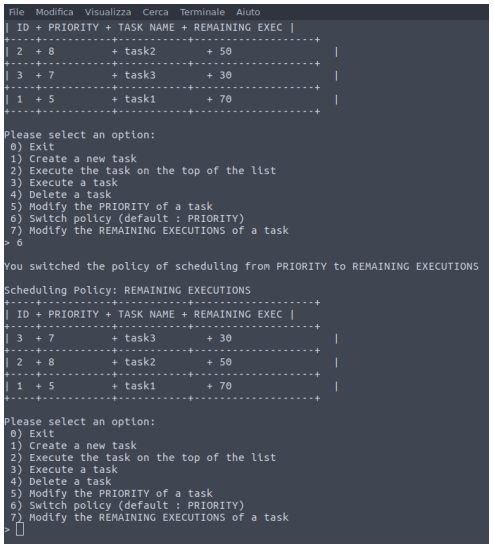


(b) inseriemento con errore

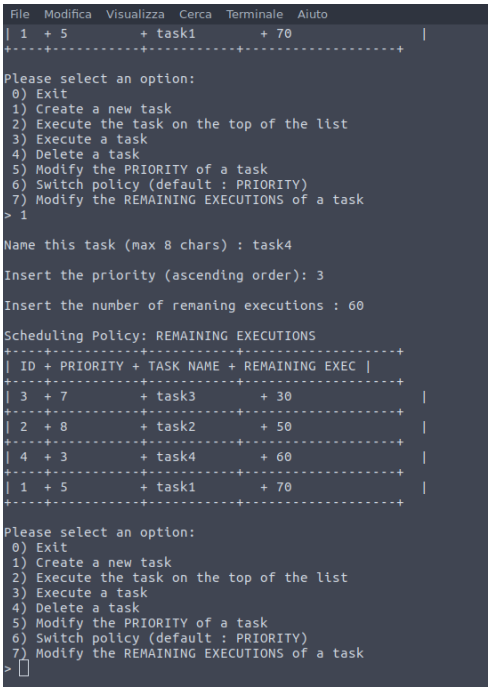


(c) inserimento e ordinamento

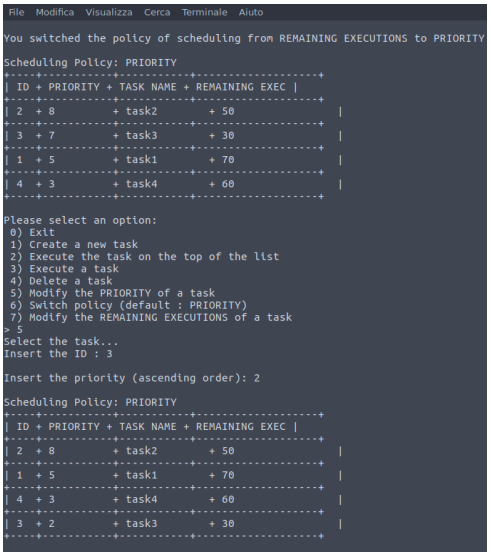
Figure 1: Creazione di un nuovo task



(a) switch della politica di scheduling

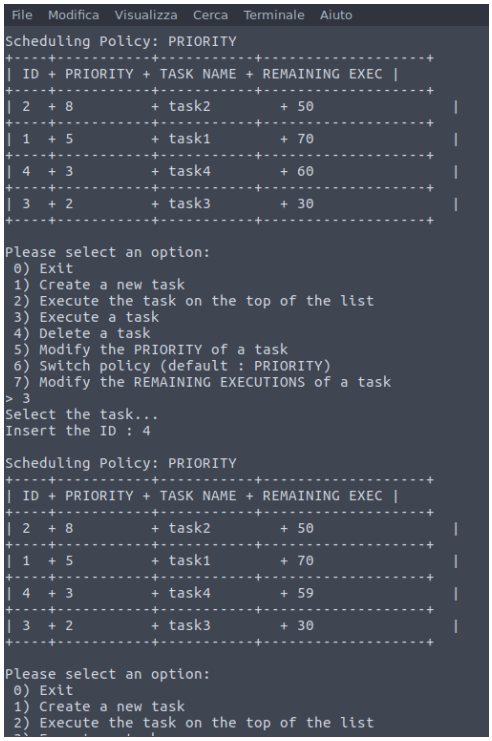


(b) inserimento e ordinamento

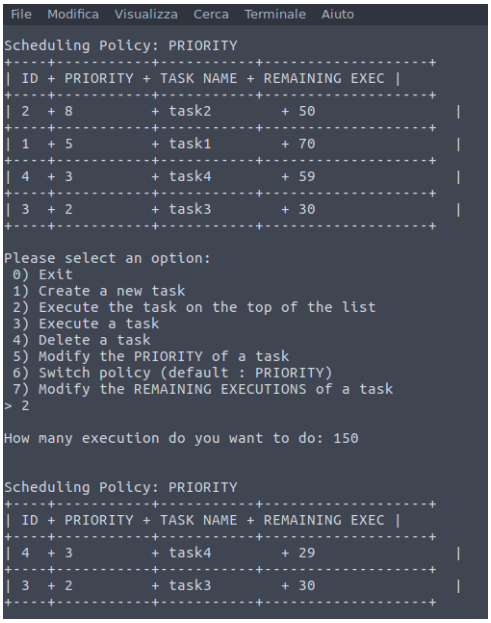


(c) switch della policy e modifica della prioriá

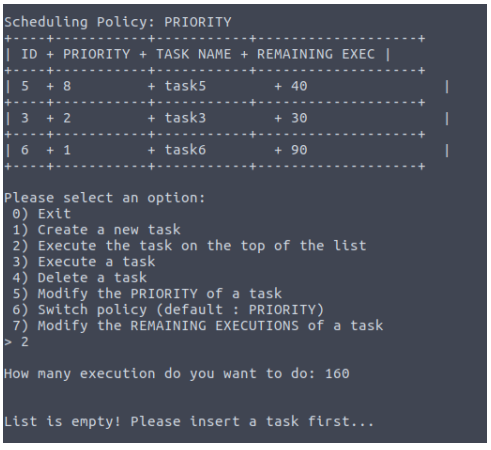
Figure 2: Mantenimento dell'ordinamento della lista dei task e modifica ai parametri



(a) singola esecuzione di un task

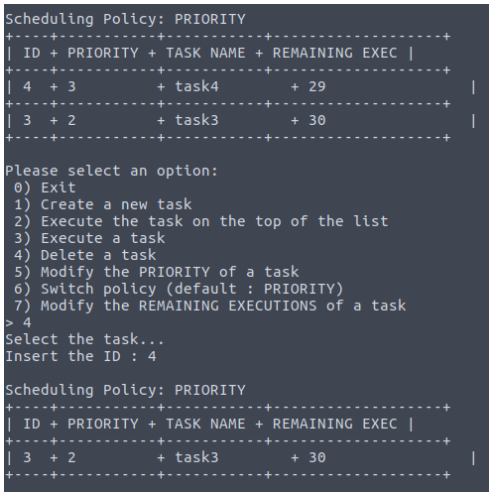


(b) 150 esecuzioni dei task in testa

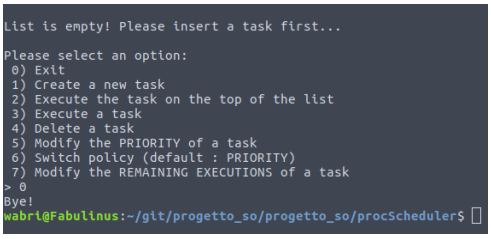


(c) esecuzione di tutti i task

Figure 3: Esecuzioni varie dei task

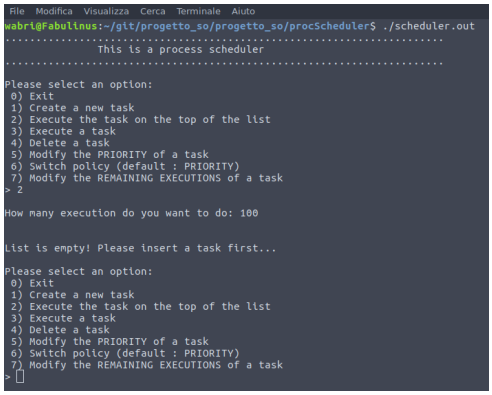


(a) eliminazione di tutti i task

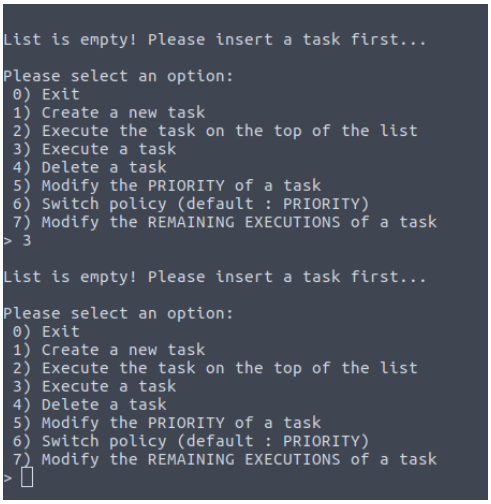


(b) uscita dal programma

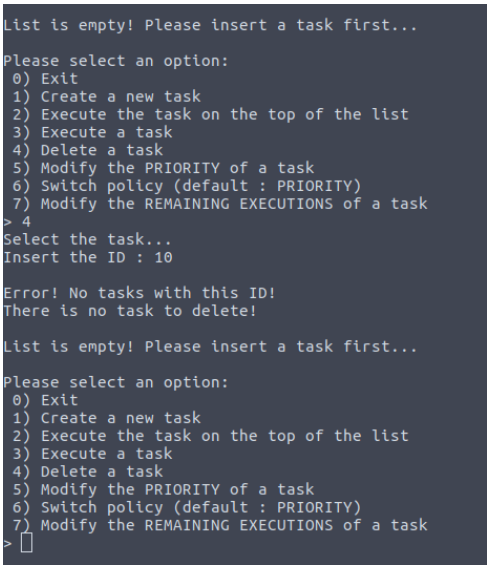
Figure 4: Eliminazione dei task ed uscita dal programma



(a) esecuzione a lista vuota

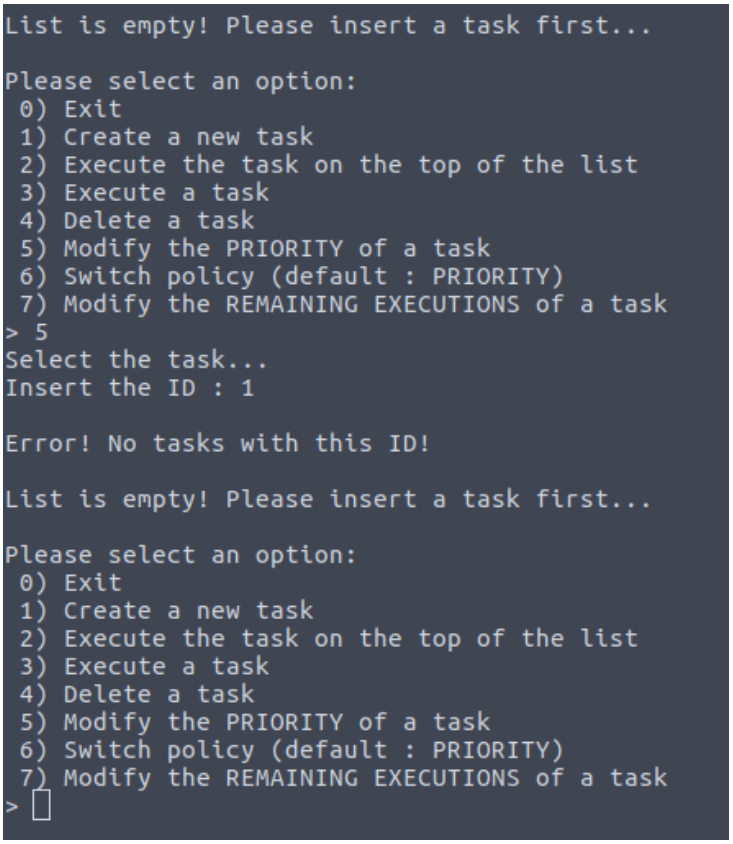


(b) esecuzione per ID a lista vuota

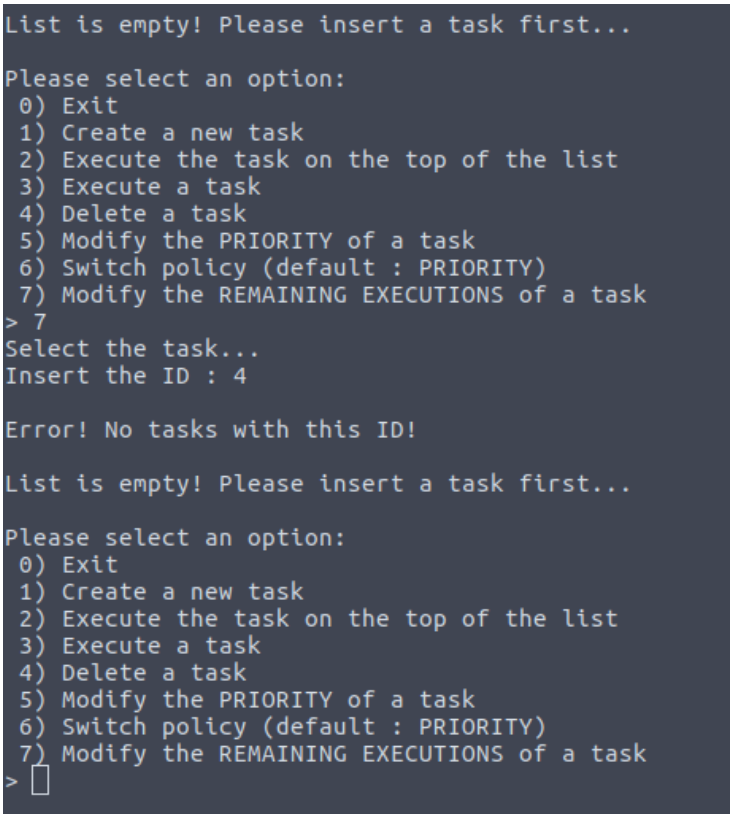


(c) eliminazione per ID a lista vuota

Figure 5: Esecuzioni a lista vuota



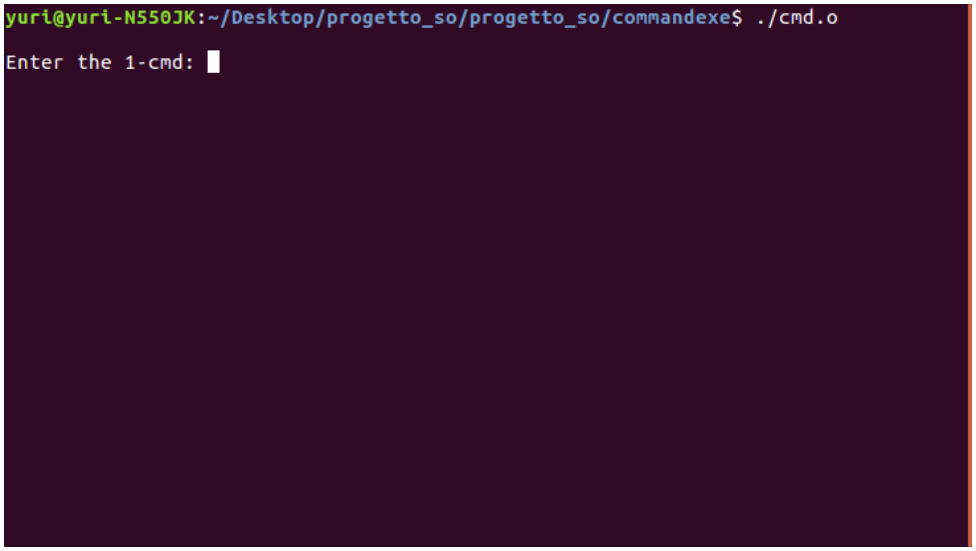
(a) modifica della prioritá a lista vuota



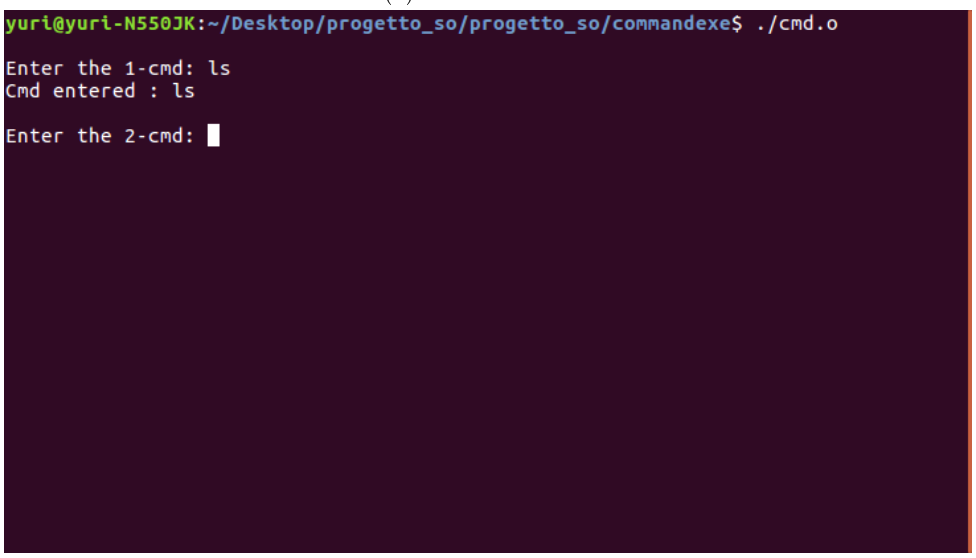
(b) modifica del n.esec. a lista vuota

Figure 6: Modifiche a lista vuota

4.2 II Esercizio

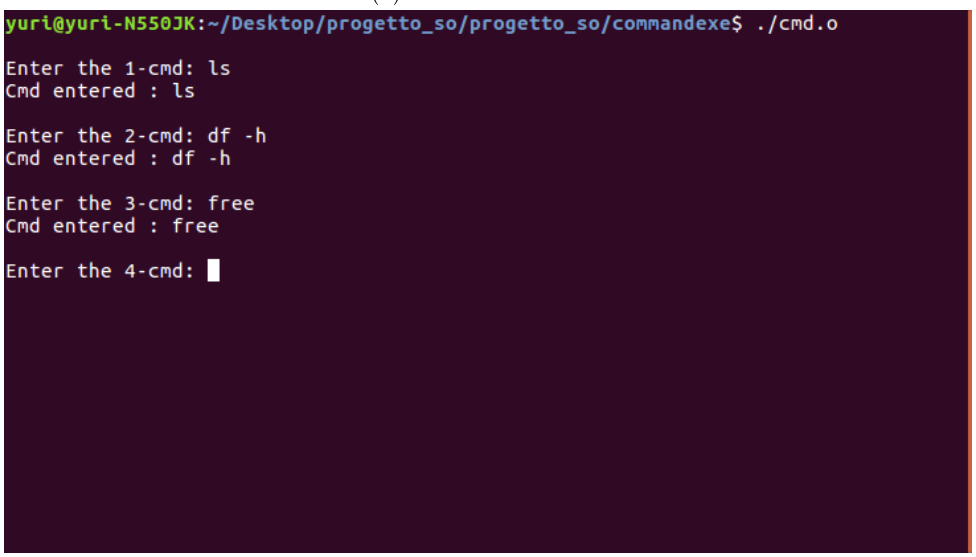


(a) esecuzioni di `ls`



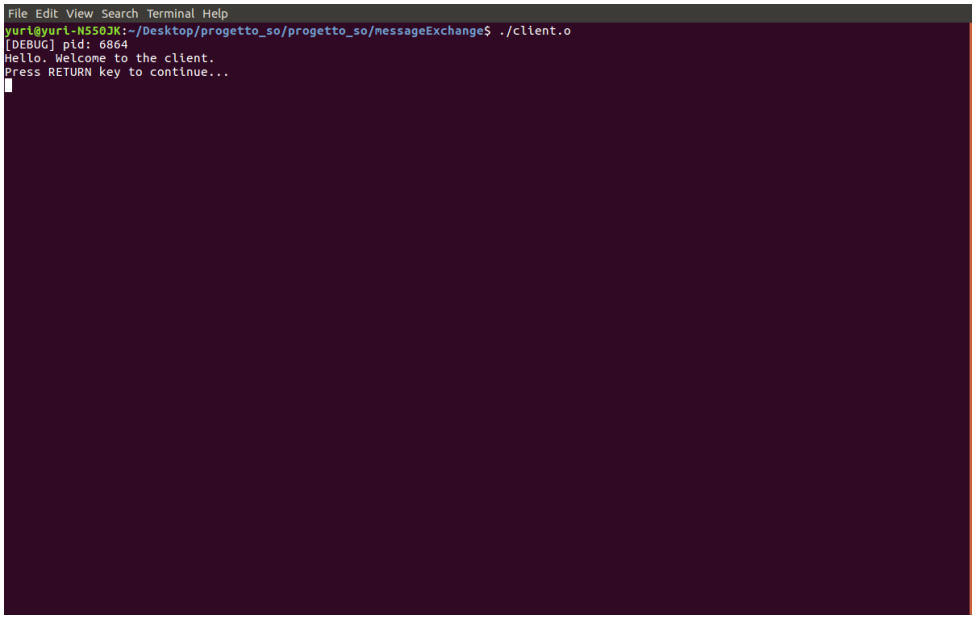
(b) esecuzioni di `df -h`

(a) esecuzioni di `free`

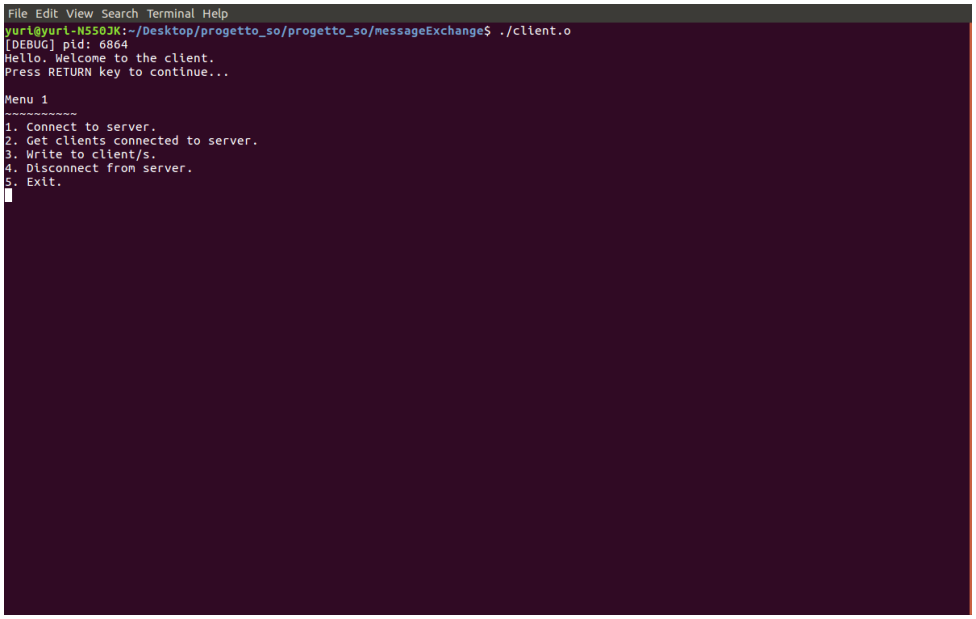


(b) uscita dal programma

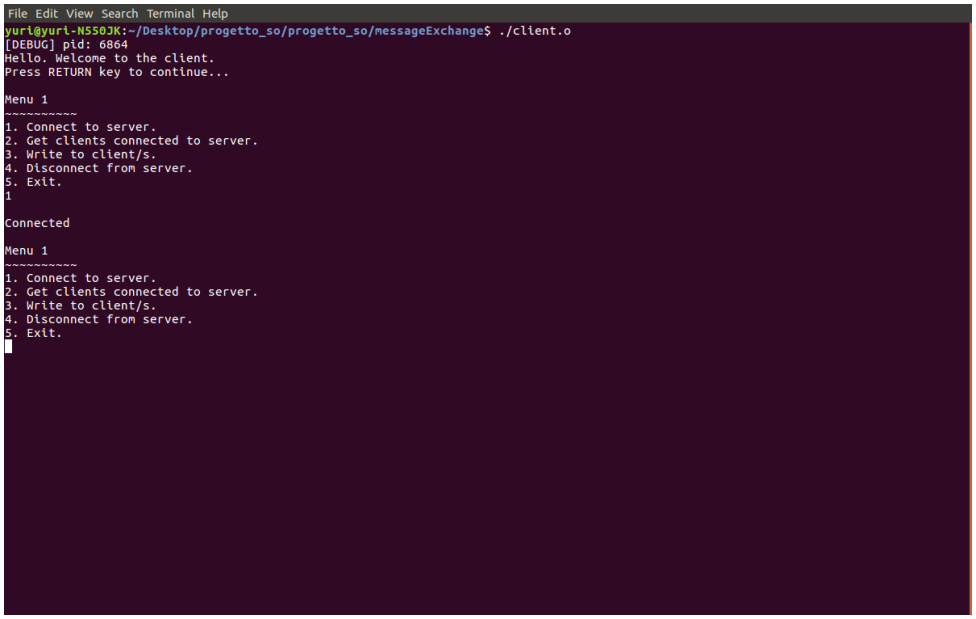
4.3 III Esercizio



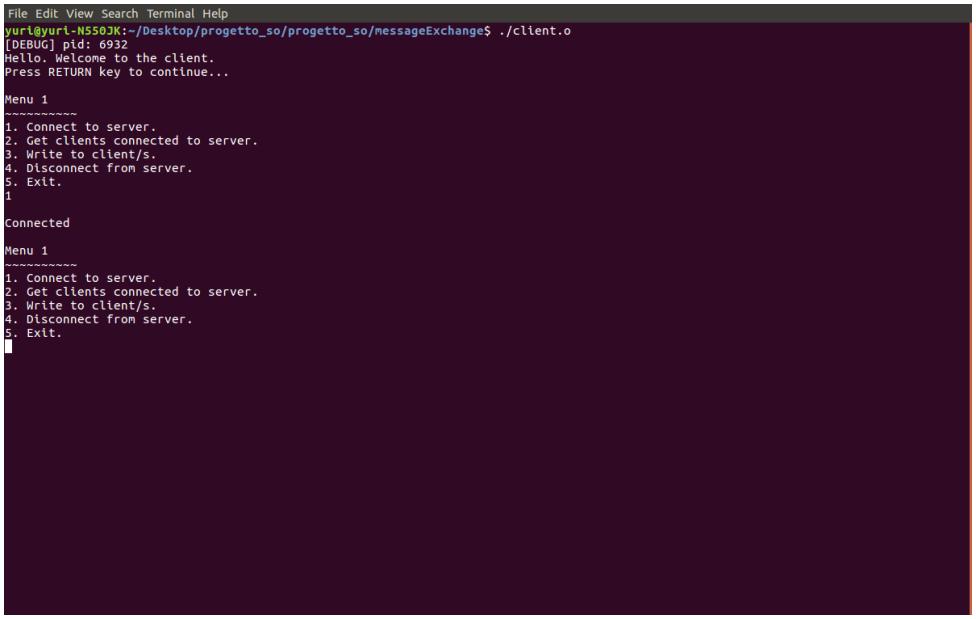
(a) lancio del client con pid 6864



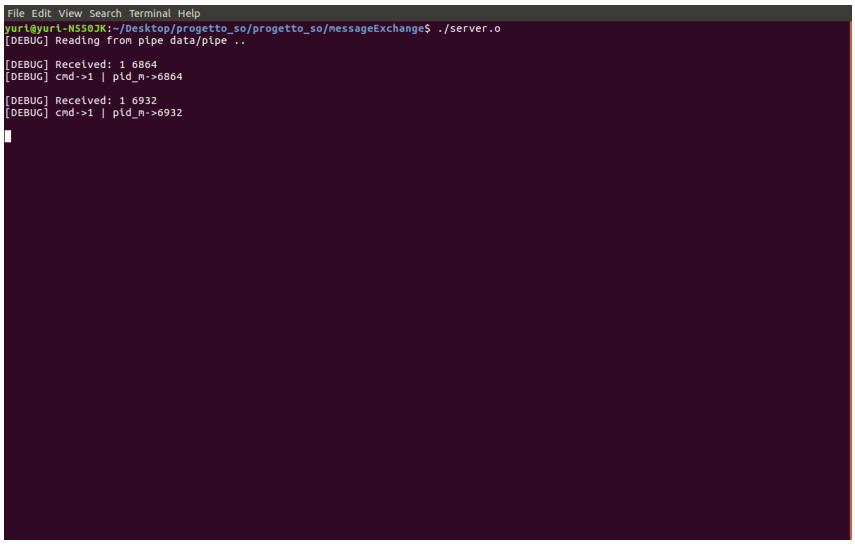
(b) comparsa del menu



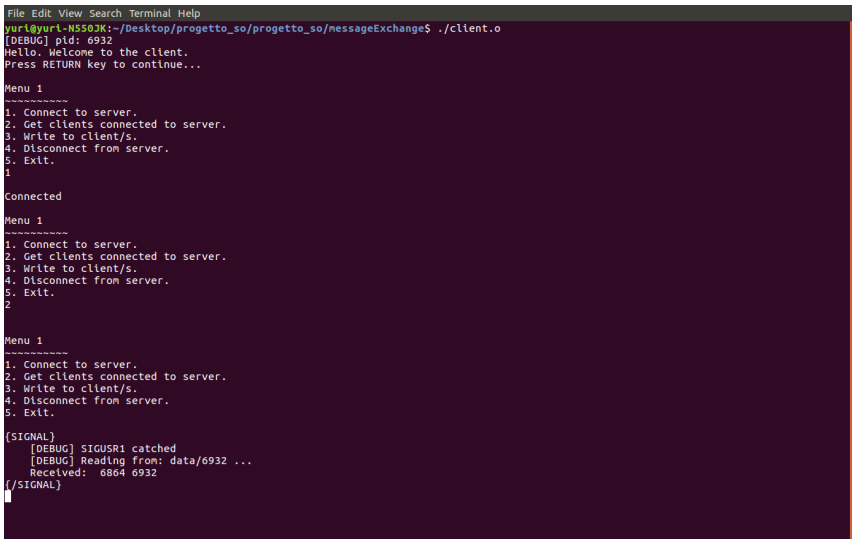
(a) connessione del client 6864 al server



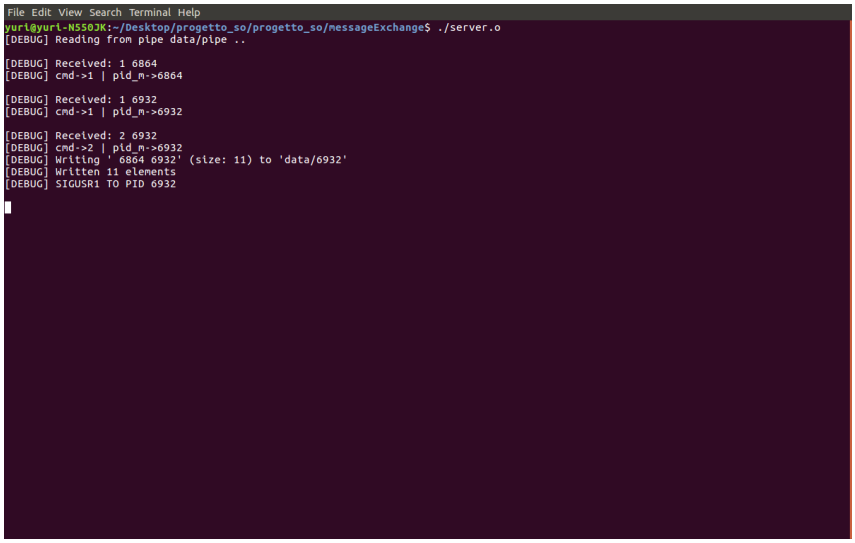
(b) connessione del client 6932 al



(a) Connessioni dei client 6864 e 6932 lato server

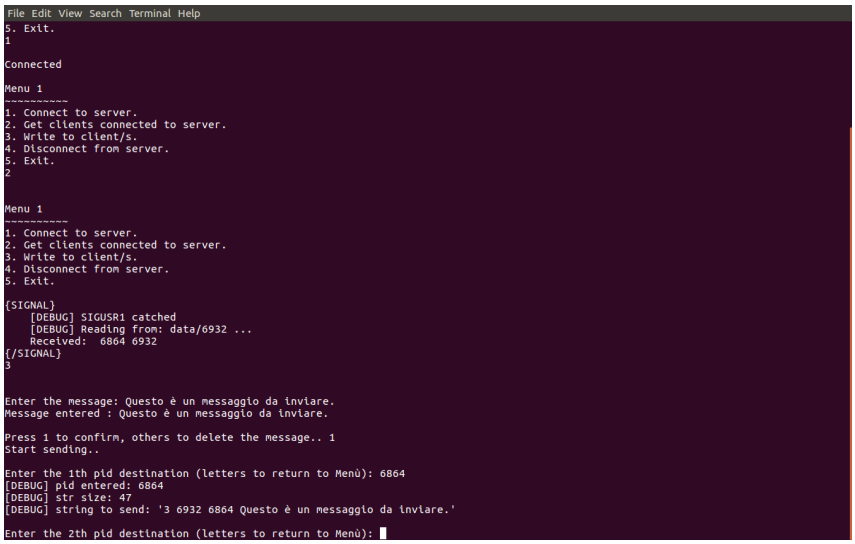


(b) Richiesta da parte di 6932 dei client connessi

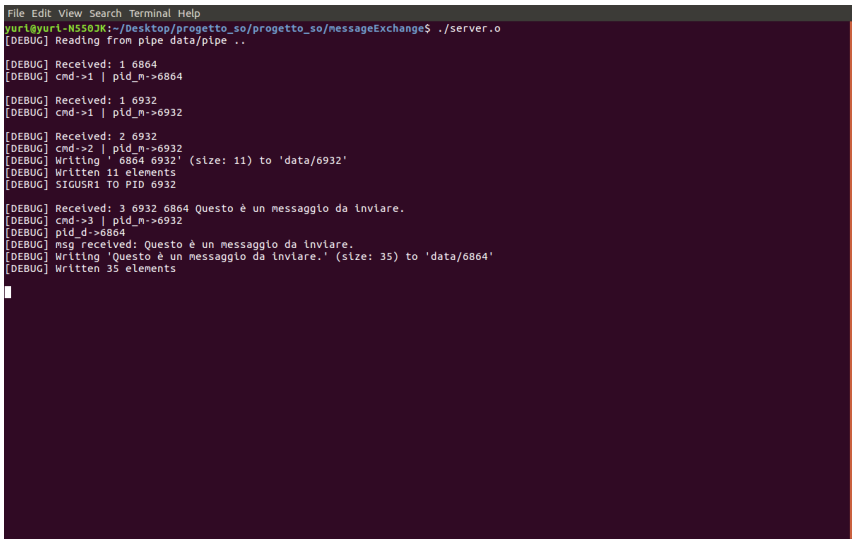


(c) Risposta del server a 6932

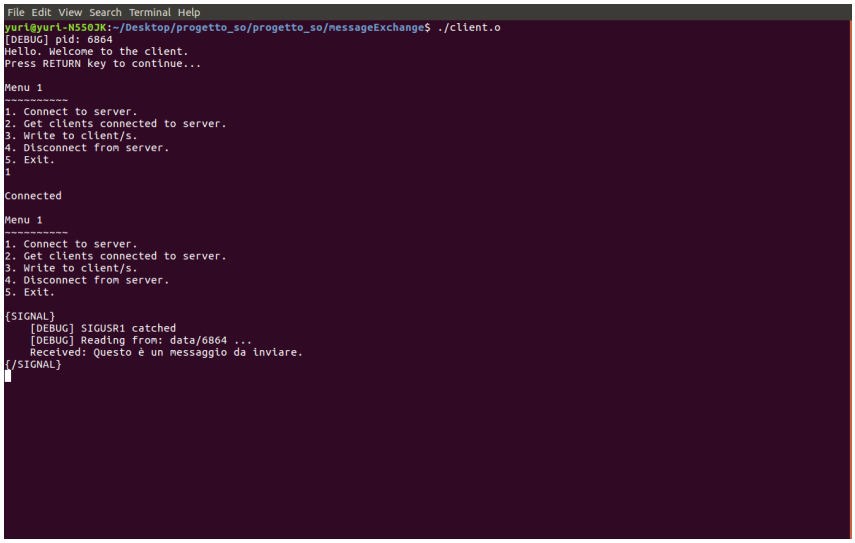
Figure 11: Connessioni e richiesta dei client connessi



(a) 6932 invia un messaggio a 6864

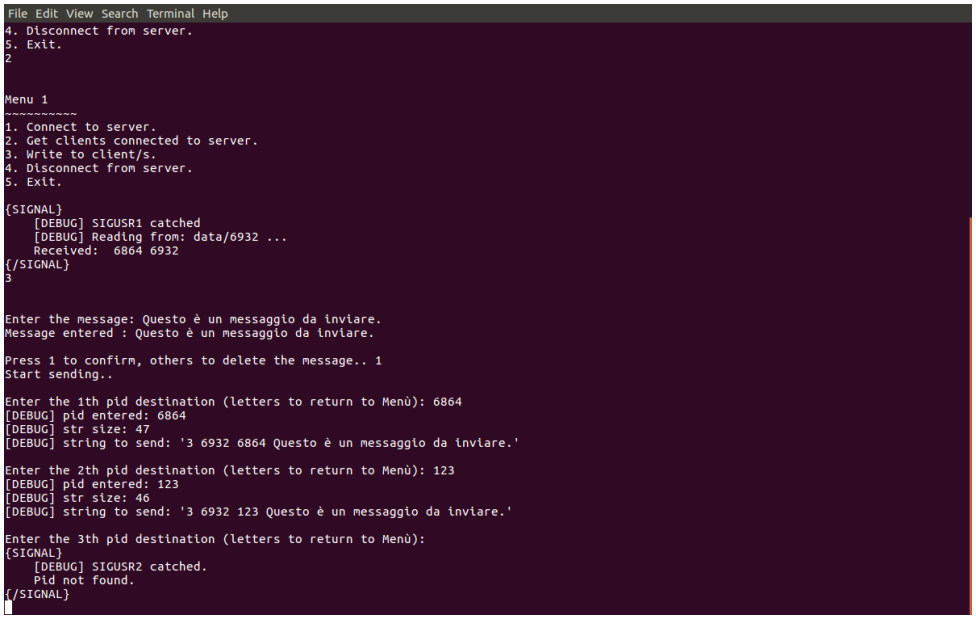


(b) Risposta del server alla richiesta di 6932

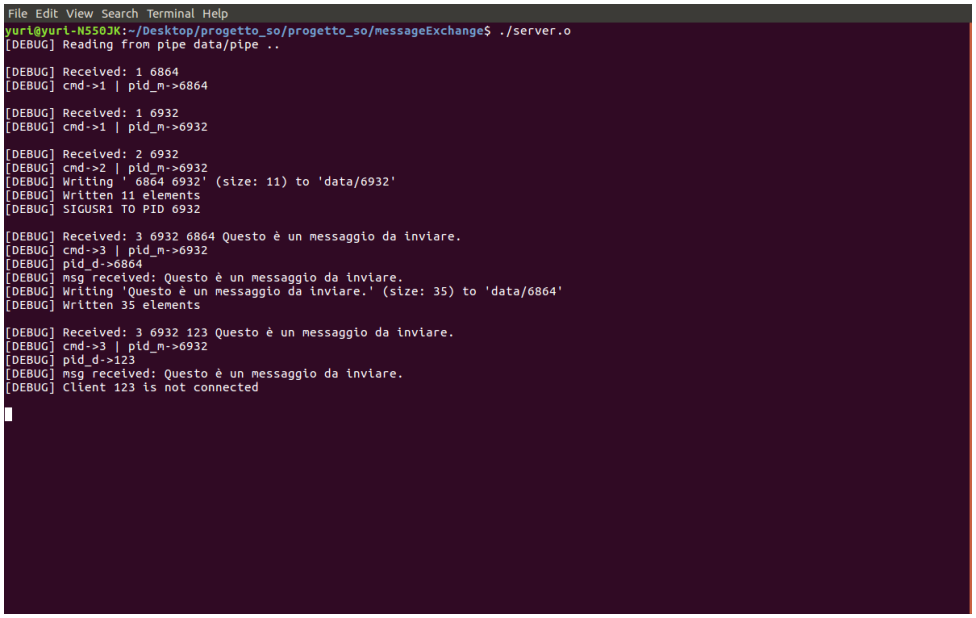


(c) Ricezione del messaggio da parte di 6864

Figure 12: Message passing

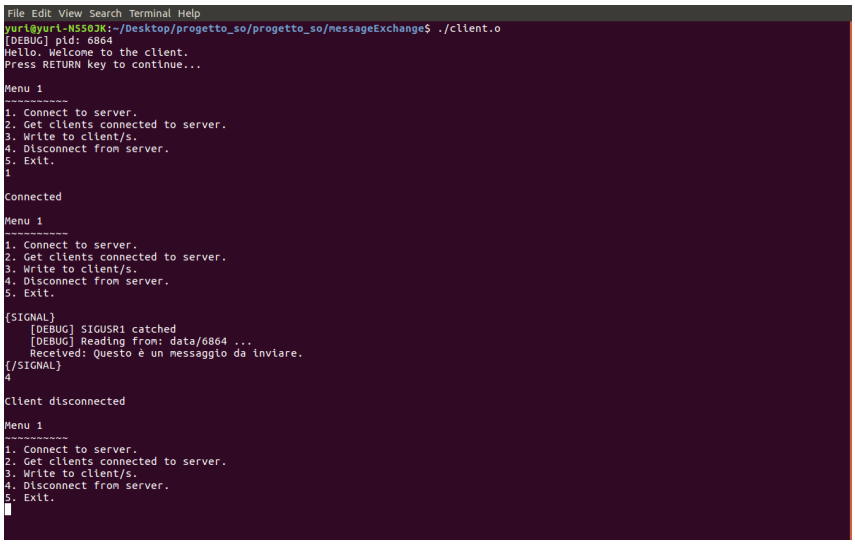


(a) Gestione dell'errore nell'invio di un messaggio ad un client inesistente

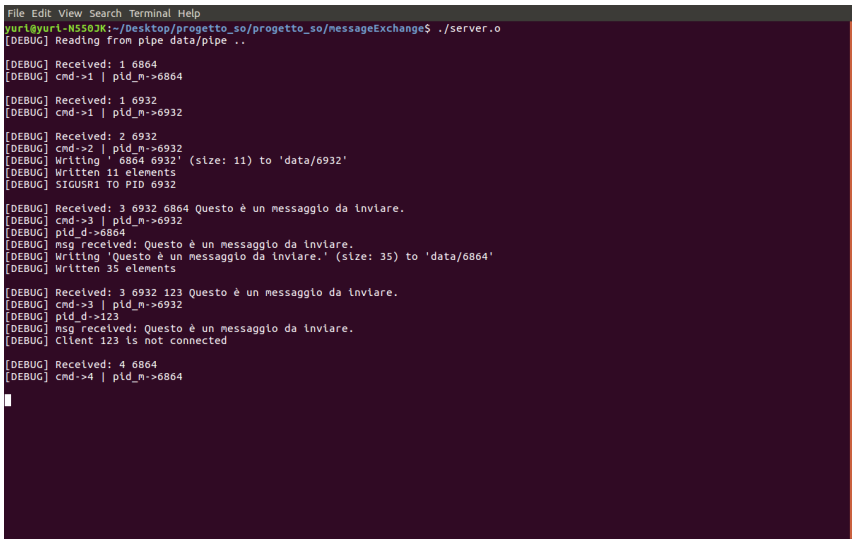


(b) Gestione dell'errore nell'invio di un messaggio ad un client inesistente lato server

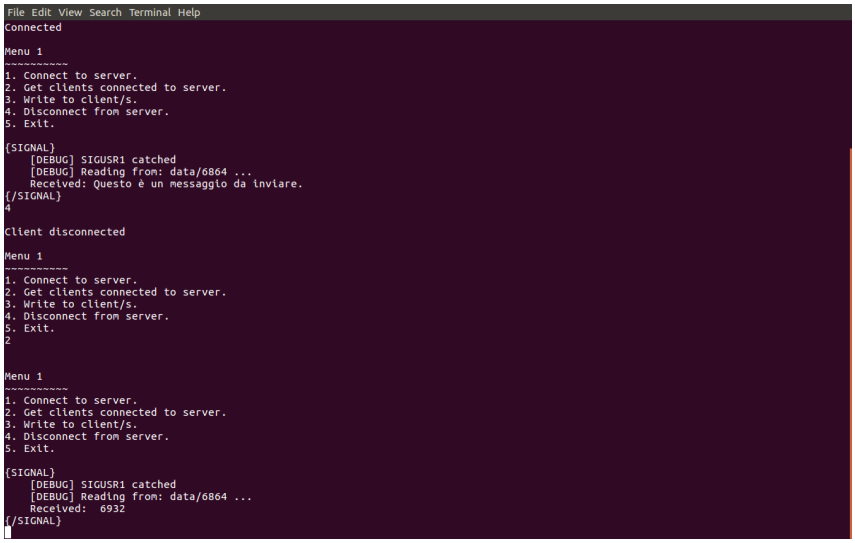
Figure 13: Errori



(a) Disconnessione di 6864 dal server

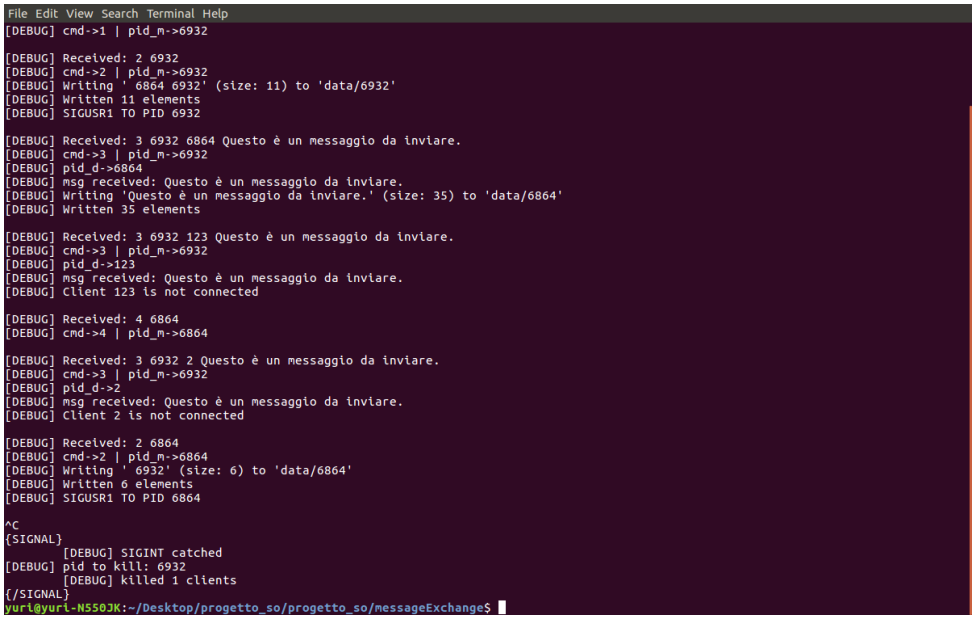


(b) Risposta del server

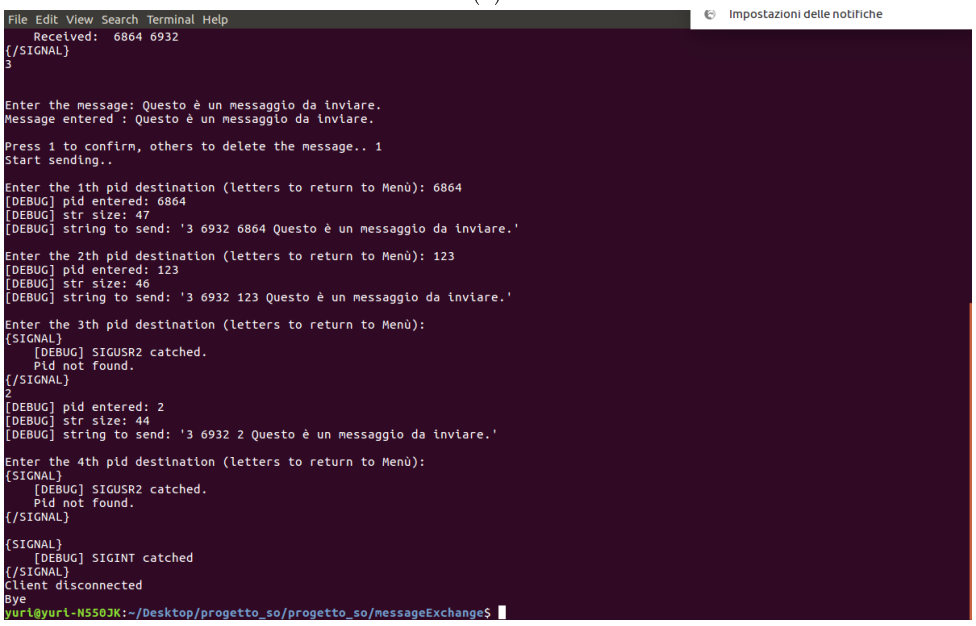


(c) Richiesta dei client connessi al server

Figure 14: Disconnessione



(a)



(b)

Figure 15: Errori