

RELAZIONE DI PROGETTO

Autori:

Gabriele Bertini 5793664 – gabriele.bertini3@stud.unifi.it

Lorenzo Pratesi Mariti 5793319 – lorenzo.pratesi@stud.unifi.it

Data di consegna:

17/09/2017

SCHEDULER DI PROCESSI

Introduzione:

La funzionalità principale di questo programma è quella di simulare uno scheduler di processi, tramite un menù a scelta, l'utente decide quale operazione desidera eseguire. L'utente ha la possibilità di scegliere tra 6 operazioni fondamentali che permettono di manipolare i task inseriti.

I task verranno memorizzati in una lista alla quale è possibile accedere tramite il puntatore alla testa, e ogni task verrà collegato in posizione adeguata tramite un puntatore.

Il programma termina quando viene scelto l'apposito comando dal menù, chiamando un return 0 che chiude il main.

Implementazione dei task:

I "task" memorizzati all'interno della lista sono una struct di tipo *Task* e hanno questa struttura:

- int *id*
- char *nomeTask*[9] → array di char per simulare una stringa
- int *priorita*
- int *esecuzioniRimanenti*
- struct *Task* **next* → puntatore al task successivo

nomeTask è un array di 9 elementi: 8 per i caratteri, 1 per il carattere di fine stringa ('\0').

Descrizione delle funzioni:

int main()

Parametri in ingresso: nessuno

Descrizione:

- istanzia una variabile *head* come puntatore a TASK e una variabile TASK *t*
- inizializza il contatore *ida* a 1 e *priorita* a 0
- stampa il menù tramite la funzione **void printMenu()**
- legge l'intero preso in input da tastiera e lo memorizza nella variabile *menuChoice*
- tramite il costrutto switch scelgo la case a cui saltare a seconda della scelta fatta
- etichette per lo switch:
 1. inserimento nuovo task
 2. esecuzione task in testa
 3. esecuzione task a scelta
 4. eliminazione task a scelta
 5. modifica priorità task a scelta
 6. modifica politica scheduling
 7. esci
- una volta eseguita la funzione adeguata torna a stampare il menu fintanto che non si esce dal programma

Valori di ritorno: 0, per indicare la terminazione del programma senza errori.

DESCRIZIONE DELLE CASE ACTIONS CON RELATIVE FUNZIONI

1. Inserimento nuovo task:

- assegna a *t* il task restituito dalla funzione **TASK choiceInsert(int *id)**
- assegna a *head* la lista ordinata dopo l'inserimento del task *t* operato dalla funzione **TASK *push(TASK *head, TASK task)**

- torna al menu

TASK choiceInsert(*int* *id)

Parametri in ingresso: id del task da creare

Descrizione:

- assegna **id* al campo id del task da creare e lo incrementa di 1
- assegna al campo nomeTask del task la stringa presa in input da tastiera
- assegna al campo priorita del task l'intero preso in input da tastiera
- assegna al campo esecuzioniRimanenti l'intero preso in input da tastiera
- torna alla case 1

Valori di ritorno: task creato con i campi inseriti dall'utente

TASK *push(TASK *head, TASK task)

Parametri in ingresso: puntatore alla testa della lista, task creato

Descrizione:

- inserisce il task *t*, in ordine secondo la politica di scheduling corrente, a partire da *head*, puntatore alla testa della lista
- a parità di priorità o esecuzioni rimanenti si inserisce in ordine decrescente secondo l'id del task
- torna alla case 1

Valori di ritorno: puntatore alla testa della lista

2. *Esecuzione del task in testa*

- assegna a *head* la lista ordinata dopo l'esecuzione del task in testa operata dalla funzione **TASK *executeTaskHead(TASK *head)**
- torna al menù

TASK *executeTaskHead(TASK *head)

Parametri in ingresso: puntatore alla testa della lista

Descrizione:

- diminuisce di 1 le esecuzioni rimanenti del task in testa
- controlla se le esecuzioni rimanenti sono terminate e in tal caso avanza il puntatore *head* al task successivo liberando la memoria occupata dalla vecchia testa
- torna alla case 2

Valori di ritorno: puntatore alla testa della lista

3. *Esecuzione task specifico*

- prende l'intero in input da tastiera e lo memorizza in *selId*
- assegna a *head* la lista ordinata dopo l'esecuzione del task scelto operata dalla funzione **TASK *executeTaskId(TASK *head, int *id)**
- torna al menù

TASK *executeTaskId(TASK *head, int *id)

Parametri in ingresso: puntatore alla testa della lista, ID del task da eseguire

Descrizione:

- avanza un puntatore *current* (utilizzato per scorrere la lista) finché non si posiziona sul task con id uguale a quello inserito
- diminuisce di 1 le esecuzioni rimanenti e controlla se sono terminate e se l'elemento corrente è la testa della lista; in tal caso avanza il puntatore *head* al task successivo liberando la memoria occupata dalla vecchia testa
- controlla se le esecuzioni rimanenti sono terminate e in tal caso avanza il puntatore *current* al task successivo liberando la memoria occupata dal vecchio elemento

- torna alla case 3

Valori di ritorno: puntatore alla testa della lista ordinata utilizzando la funzione
`TASK *sortingByPolicy(TASK *head)`

`TASK *sortingByPolicy(TASK *head)`

Parametri in ingresso: puntatore alla testa della lista

Descrizione:

- alloca memoria della dimensione di un *TASK* per *new_head*, cioè la testa della lista ordinata
- scorre la lista con un puntatore *current* a partire da *head* e inserisce ogni elemento della lista nella nuova lista, in ordine utilizzando la funzione
`TASK *push(TASK *head, TASK task)`
- libera la memoria occupata da *head*
- torna alla funzione chiamante

Valori di ritorno: puntatore alla testa della lista ordinata

4. *Elimina task specifico*

- prende l'intero in input da tastiera e lo memorizza in *selId*
- assegna a *head* la lista ordinata dopo l'eliminazione del task scelto operata dalla funzione **`TASK *deleteTaskId(TASK *head, int *id)`**
- torna al menù

`TASK *deleteTaskId(TASK *head, int *id)`

Parametri in ingresso: puntatore alla testa della lista, ID del task da eliminare

Descrizione:

- avanza un puntatore *current* (utilizzato per scorrere la lista) finché non si posiziona sul task con id uguale a quello inserito
- se l'elemento corrente è la testa della lista in tal caso avanza il puntatore *head* al task successivo liberando la memoria occupata dalla vecchia testa
- altrimenti avanza il puntatore *current* al task successivo liberando la memoria occupata dal vecchio elemento
- torna alla case 4

Valori di ritorno: puntatore alla testa della lista

5. *Modifica priorità task specifico*

- prende l'intero in input da tastiera e lo memorizza in *selId*
- prende l'intero in input da tastiera e lo memorizza in *priority*
- chiama **`void modifyTaskIdPriority(TASK *head, int *id, int *priority)`**, funzione che modifica la priorità del task con id scelto
- riordina la lista utilizzando la funzione **`TASK *sortingByPolicy(TASK *head)`**
- torna al menù

`void modifyTaskIdPriority(TASK *head, int *id, int *priority)`

Parametri in ingresso: puntatore alla testa della lista, ID del task a cui modificare la priorità, nuovo valore per la priorità

Descrizione:

- avanza un puntatore *current* (utilizzato per scorrere la lista) finché non si posiziona sul task con id uguale a quello inserito
- modifica il campo priorità di *current* con il nuovo valore
- torna al case 5

Valori di ritorno: nessuno

6. Cambia la politica di scheduling

- assegna a *head* il puntatore alla lista ordinata dopo il cambio di politica di scheduling restituito dalla funzione **TASK *changePolicy(TASK *head)**
- torna al menù

TASK *changePolicy(TASK *head)

Parametri in ingresso: puntatore alla testa della lista

Descrizione:

- cambia la politica di scheduling attuale

Valori di ritorno: puntatore alla testa della lista ordinata secondo la nuova politica

7. Esci dal programma

- stampa messaggio d'uscita
- esce dal programma (return 0)

DESCRIZIONE DELLE FUNZIONI DI STAMPA

Abbiamo deciso di elencare le funzioni di stampa senza però descriverle vista la loro banalità e forma:

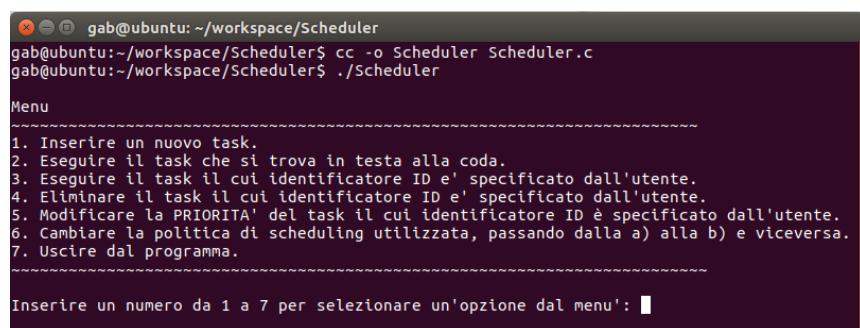
```
void printMenu()  
  
void print_list(TASK *head)  
  
void getSpaced(char str[])
```

Simulazione:

La simulazione mostra alcune situazione critiche attraverso l'applicazione di tutte le operazioni effettuabili dal menù.

Mostriamo come compilare il programma e lanciarlo in esecuzione. Abbiamo scelto l'opzione **-o** del comando **cc** per poter rinominare il file da eseguire in modo che avesse un nome pertinente.

Al lancio del programma possiamo notare che viene immediatamente stampato il menù con le 7 opzioni disponibili e viene richiesto di scegliere una di queste.



```
gab@ubuntu: ~/workspace/Scheduler  
gab@ubuntu:~/workspace/Scheduler$ cc -o Scheduler Scheduler.c  
gab@ubuntu:~/workspace/Scheduler$ ./Scheduler  
Menu  
-----  
1. Inserire un nuovo task.  
2. Eseguire il task che si trova in testa alla coda.  
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.  
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.  
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.  
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.  
7. Uscire dal programma.  
-----  
Inserire un numero da 1 a 7 per selezionare un'opzione dal menu':
```

Abbiamo selezionato l'opzione 1 e inserito i dati per la creazione del primo task che viene stampato insieme alla politica di scheduling attuale, cioè priorità.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 1
Inserisci stringa di max 8 caratteri per nome task: first
Inserisci numero compreso tra 0 e 9 per priorita' task: 0
Inserisci esecuzioni rimanenti task: 2

La Politica di Scheduling attuale e': Priorita'
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+-----+-----+-----+-----+
| 1 | 0 | first | 2 |
+-----+-----+-----+
```

Abbiamo selezionato l'opzione 1 e inserito i dati per la creazione del secondo task in ordine decrescente di priorità. Il task inserito si posiziona in testa seguendo la politica di scheduling corrente.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 1
Inserisci stringa di max 8 caratteri per nome task: second
Inserisci numero compreso tra 0 e 9 per priorita' task: 1
Inserisci esecuzioni rimanenti task: 1

La Politica di Scheduling attuale e': Priorita'
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+-----+-----+-----+-----+
| 2 | 1 | second | 1 |
+-----+-----+-----+
| 1 | 0 | first | 2 |
+-----+-----+
```

Abbiamo selezionato l'opzione 5 e indicato il task con ID 1 per cambiarne la priorità. Essendo in vigore la politica di scheduling per esecuzioni rimanenti, però, non si avranno modifiche nell'ordine dei task in lista.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 5
Inserire ID task: 1
Inserire nuova priorita': 2

La Politica di Scheduling attuale e': Esecuzioni Rimanenti
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+-----+-----+-----+-----+
| 2 | 1 | second | 1 |
+-----+-----+-----+
| 1 | 2 | first | 2 |
+-----+-----+
```

Abbiamo selezionato l'opzione 5 e indicato il task con ID 3 per cambiarne la priorità, ma un task con tale ID non è presente; verrà perciò stampato un messaggio di errore.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 5
Inserire ID task: 3
Inserire nuova priorita': 1
ERRORE! L'ID scelto non e' presente.

La Politica di Scheduling attuale e': Esecuzioni Rimanenti
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+---+-----+-----+-----+
| 2 | 1 | second | 1 |
+---+-----+-----+-----+
| 1 | 0 | first | 2 |
+---+-----+-----+-----+
```

Abbiamo selezionato l'opzione 6 che cambia politica di scheduling corrente; ciò ha portato a ordinare la lista per priorità, cambiando, così, l'ordine dei task presenti.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 6

La Politica di Scheduling attuale e': Priorita'
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+---+-----+-----+-----+
| 1 | 2 | first | 2 |
+---+-----+-----+-----+
| 2 | 1 | second | 1 |
+---+-----+-----+-----+
```

Abbiamo selezionato l'opzione 2 ed eseguito il task in testa, che ha diminuito di 1 le proprie esecuzioni rimanenti.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 2

La Politica di Scheduling attuale e': Priorita'
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+---+-----+-----+-----+
| 1 | 2 | first | 1 |
+---+-----+-----+-----+
| 2 | 1 | second | 1 |
+---+-----+-----+-----+
```

Abbiamo selezionato l'opzione 3 ed eseguito il task con ID 1, che ha diminuito di 1 le proprie esecuzioni rimanenti; essendo rimasto senza esecuzioni rimanenti, il task con ID 1 è stato eliminato dalla lista.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 3
Inserire ID task: 1

La Politica di Scheduling attuale e': Priorita'
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+---+-----+-----+-----+
| 2 | 1 | second | 1 |
+---+-----+-----+-----+
```

Abbiamo selezionato l'opzione 4 e indicato il task con ID 2, il quale è stato eliminato dalla lista.

Verrà stampato un messaggio per indicare che non è presente alcun task nella lista.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 4
Inserire ID task: 2

Lista vuota! Nessun task e' presente.
La Politica di Scheduling attuale e': Priorita'
```

Abbiamo selezionato l'opzione 1 e inserito i dati per la creazione del terzo task; stavolta la politica di scheduling corrente è esecuzioni rimanenti.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 1
Inserisci stringa di max 8 caratteri per nome task: third
Inserisci numero compreso tra 0 e 9 per priorita' task: 3
Inserisci esecuzioni rimanenti task: 99

La Politica di Scheduling attuale e': Esecuzioni Rimanenti
+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+-----+-----+-----+
| 3 | 3 | third | 99 |
+-----+-----+-----+
```

Abbiamo selezionato l'opzione 1 e inserito i dati per la creazione del quarto task, che viene inserito in testa secondo la politica di scheduling per esecuzioni rimanenti.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 1
Inserisci stringa di max 8 caratteri per nome task: fourth
Inserisci numero compreso tra 0 e 9 per priorita' task: 9
Inserisci esecuzioni rimanenti task: 1

La Politica di Scheduling attuale e': Esecuzioni Rimanenti
+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+-----+-----+-----+
| 4 | 9 | fourth | 1 |
+-----+-----+-----+
| 3 | 3 | third | 99 |
+-----+-----+-----+
```

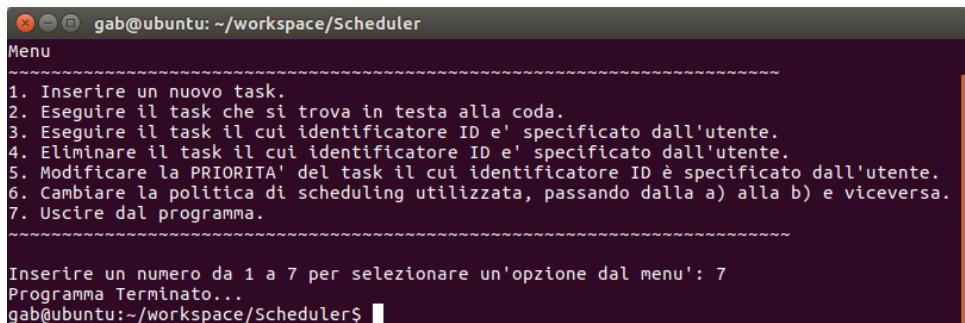
Abbiamo selezionato l'opzione 2 ed eseguito il task in testa, che ha diminuito di 1 le proprie esecuzioni rimanenti; essendo rimasto senza esecuzioni rimanenti, il task con ID 4 è stato eliminato dalla lista.

```
gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.

Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 2

La Politica di Scheduling attuale e': Esecuzioni Rimanenti
+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|
+-----+-----+-----+
| 3 | 3 | third | 99 |
+-----+-----+-----+
```

Abbiamo selezionato l'opzione 7 per uscire dal programma; ciò terminerà il programma stampando un messaggio di uscita.



```

gab@ubuntu: ~/workspace/Scheduler
Menu
-----
1. Inserire un nuovo task.
2. Eseguire il task che si trova in testa alla coda.
3. Eseguire il task il cui identificatore ID e' specificato dall'utente.
4. Eliminare il task il cui identificatore ID e' specificato dall'utente.
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente.
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa.
7. Uscire dal programma.
-----
Inserire un numero da 1 a 7 per selezionare un'opzione dal menu': 7
Programma Terminato...
gab@ubuntu:~/workspace/Scheduler$ 
```

Codice:

```

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

#define P_SPACE 11

struct Task {
    int id;
    char nomeTask[9];
    int priorita;
    int esecuzioniRimanenti;

    struct Task *next;
};

typedef struct Task TASK;

enum policyType {
    PRIORITA, ESECUZIONIRIMANENTI
};

int policy = PRIORITA;
int numTask = 0;

void printMenu();
TASK choiceInsert(int *id);
void print_list(TASK *head);
void getSpaced(char str[]);
TASK *push(TASK *head, TASK task);
TASK *sortingByPolicy(TASK *head);
TASK *executeTaskHead(TASK *head);
TASK *executeTaskId(TASK *head, int *id);
TASK *deleteTaskId(TASK *head, int *id);
void modifyTaskIdPriority(TASK *head, int *id, int *priority);
TASK *changePolicy(TASK *head);
TASK *sortingByPolicy(TASK *head);

int main() {
    TASK *head;
    TASK t;
    head = malloc(sizeof(TASK)); // alloca memoria della dimensione di TASK
    per la testa della lista

    int menuChoice = 0, id = 1, selId = 0, priority = 0;

    while (menuChoice != 7) {
        printMenu(); 
```

```

        printf("Inserire un numero da 1 a 7 per selezionare un'opzione dal
menu: ");
        scanf("%d", &menuChoice);
        while (menuChoice < 1 || menuChoice > 7) {
            /*
             * ripulisce lo scanf se c'e' rimasto un '\n'
             * dovuto ad un inserimento di un carattere al posto di un
numero intero
            */
            while (getchar() != '\n');
            printf("ERRORE! Inserire un numero da 1 a 7: ");
            scanf("%d", &menuChoice);
        }
        switch (menuChoice) {
    case 1: // inserimento nuovo task
        t = choiceInsert(&id);
        head = push(head, t);
        numTask++;
        break;
    case 2: // esecuzione task in testa
        if (numTask > 0) {
            head = executeTaskHead(head);
        } else {
            printf("ERRORE! Inserire prima un task.\n");
        }
        break;
    case 3: // esecuzione task a scelta
        if (numTask > 0) {
            printf("Inserire ID task: ");
            while (scanf("%d", &selId) == 0) { // se la scanf
restituisce 0 allora l'input non è un intero
                while (getchar() != '\n');
                printf("ERRORE! Inserire nuovamente ID task: ");
            }
            head = executeTaskId(head, &selId);
        } else {
            printf("ERRORE! Inserire prima un task.\n");
        }
        break;
    case 4: // eliminazione task a scelta
        if (numTask > 0) {
            printf("Inserire ID task: ");
            while (scanf("%d", &selId) == 0) { // se la scanf
restituisce 0 allora l'input non è un intero
                while (getchar() != '\n');
                printf("ERRORE! Inserire nuovamente ID task: ");
            }
            head = deleteTaskId(head, &selId);
        } else {
            printf("ERRORE! Inserire prima un task.\n");
        }
        break;
    case 5: // modifica priorita' task a scelta
        if (numTask > 0) {
            printf("Inserire ID task: ");
            while (scanf("%d", &selId) == 0) { // se la scanf
restituisce 0 allora l'input non è un intero
                while (getchar() != '\n');
                printf("ERRORE! Inserire nuovamente ID task: ");
            }
            printf("Inserire nuova priorita': ");
            scanf("%d", &priority);
        }
    }
}

```

```

        while (priority < 0 || priority > 9) {
            while (getchar() != '\n');
            printf("ERRORE. Inserire nuovamente la nuova
priorita': ");
            scanf("%d", &priority);
        }
        modifyTaskIdPriority(head, &selId, &priority);
        head = sortingByPolicy(head);
    } else {
        printf("ERRORE! Inserire prima un task.\n");
    }
    break;
case 6: // modifica politica scheduling
    head = changePolicy(head);
    break;
case 7: // esce
    printf("Programma Terminato...");
    return 0;
    break;
}
if (numTask > 0) {
    print_list(head);
} else {
    printf("\nLista vuota! Nessun task e' presente.\n");
    printf("La Politica di Scheduling attuale e': %s \n", policy
== 0 ? "Priorita'" : "Esecuzioni Rimanenti");
}
}

return 0;
}

void printMenu() {
    printf("\nMenu\n");

printf("~~~~~\n");
    printf("1. Inserire un nuovo task.\n");
    printf("2. Eseguire il task che si trova in testa alla coda.\n");
    printf("3. Eseguire il task il cui identificatore ID e' specificato
dall'utente.\n");
    printf("4. Eliminare il task il cui identificatore ID e' specificato
dall'utente.\n");
    printf("5. Modificare la PRIORITA' del task il cui identificatore ID è
specificato dall'utente.\n");
    printf("6. Cambiare la politica di scheduling utilizzata, passando dalla
a) alla b) e viceversa.\n");
    printf("7. Uscire dal programma.\n");

printf("~~~~~\n\n");
}

TASK choiceInsert(int *id) {
    TASK task;
    task.id = (*id)++;

    printf("Inserisci stringa di max 8 caratteri per nome task: ");
    scanf("%s", task.nomeTask);
    while (strlen(task.nomeTask) > 8) {
        printf("ERRORE. Inserisci nuovamente nome task: ");
        scanf("%s", task.nomeTask);
    }
}

```

```

}

printf("Inserisci numero compreso tra 0 e 9 per priorita' task: ");
scanf("%d", &task.priorita);
while (task.priorita < 0 || task.priorita > 9) {
    while (getchar() != '\n');
    printf("ERRORE. Inserisci nuovamente priorita' task: ");
    scanf("%d", &task.priorita);
}

printf("Inserisci esecuzioni rimanenti task: ");
scanf("%d", &task.esecuzioniRimanenti);
while (task.esecuzioniRimanenti < 1 || task.esecuzioniRimanenti > 99) {
    while (getchar() != '\n');
    printf("ERRORE. Inserisci nuovamente esecuzioni rimanenti task: ");
    scanf("%d", &task.esecuzioniRimanenti);
}
return task;
}

void print_list(TASK *head) {
    TASK *current = head;
    printf("\nLa Politica di Scheduling attuale e': %s \n", policy == 0 ?
"Priorita'" : "Esecuzioni Rimanenti");
    printf("+-----+-----+-----+-----+");
    printf("\n| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI|\n");
    printf("+-----+-----+-----+-----+\n");

    while (current != NULL && current->id != 0) {
        if (current->id % 10 == current->id) {
            printf("| %d ", current->id);
        } else {
            printf("| %d ", current->id);
        }

        printf("|      %d      ", current->priorita);

        printf("|%s", current->nomeTask);
        getSpaced(current->nomeTask);

        if (current->esecuzioniRimanenti % 10 == current-
>esecuzioniRimanenti) {
            printf("|      %d      |", current->esecuzioniRimanenti);
        } else {
            printf("|      %d      |", current->esecuzioniRimanenti);
        }

        current = current->next;
        printf("\n+-----+-----+-----+-----+\n");
    }
}

void getSpaced(char str[]) {
    int length, space;
    char s[11];

    length = P_SPACE - strlen(str);
    for (space = 0; space < length; space++) {
        strcat(s, " ");
    }
    printf("%s", s);
}

```

```

TASK *push(TASK *head, TASK task) {
    TASK *current = head;
    TASK *previous = NULL;
    if (policy == PRIORITA) {
        while (current->priorita >= task.priorita) {
            if (current->priorita == task.priorita && current->id <
task.id) {
                break;
            }
            previous = current;
            current = current->next;
        }
    } else {
        while (current->next != NULL && current->esecuzioniRimanenti <=
task.esecuzioniRimanenti) {
            if (current->esecuzioniRimanenti == task.esecuzioniRimanenti
&& current->id < task.id) {
                break;
            }
            previous = current;
            current = current->next;
        }
    }
    TASK *temp = malloc(sizeof(TASK));
    temp->id = task.id;
    temp->priorita = task.priorita;
    strncpy(temp->nomeTask, task.nomeTask, 8);
    temp->esecuzioniRimanenti = task.esecuzioniRimanenti;
    temp->next = current;
    if (numTask == 0) { // con lista vuota restituisco il task creato
        return temp;
    }
    if (previous != NULL) {
        previous->next = temp;
    } else {
        head = temp;
    }
    return head;
}

TASK *executeTaskHead(TASK *head) {
    if (--head->esecuzioniRimanenti <= 0) {
        TASK* new_head = head->next;
        free(head);
        head = new_head;
        numTask--;
    }
    return head;
}

TASK *executeTaskId(TASK *head, int *id) {
    TASK *current = head;
    TASK *previous = current;
    while (current->id != *id && current->next != NULL) {
        previous = current;
        current = current->next;
    }
    if (current->id == *id) {
        if (--current->esecuzioniRimanenti <= 0 && current == head) {
            TASK *new_head = head->next;
            head = NULL;
        }
    }
}

```

```

        free(head);
        head = new_head;
        numTask--;
        return head;
    } else if (current->esecuzioniRimanenti <= 0) {
        TASK *new_current = current->next;
        current = NULL;
        free(current);
        current = new_current;
        previous->next = current;
        numTask--;
    }
} else {
    printf("ERRORE! L'ID scelto non e' presente.\n");
}
return sortingByPolicy(head);
}

TASK *deleteTaskId(TASK *head, int *id) {
    TASK *current = head;
    TASK *previous = current;
    while (current->id != *id && current->next != NULL) {
        previous = current;
        current = current->next;
    }
    if (current->id == *id) {
        if (current == head) {
            TASK *new_head = head->next;
            head = NULL;
            free(head);
            head = new_head;
        } else {
            TASK *new_current = current->next;
            current = NULL;
            free(current);
            current = new_current;
            previous->next = current;
        }
        numTask--;
    } else {
        printf("ERRORE! L'ID scelto non e' presente.\n");
    }
    return head;
}

void modifyTaskIdPriority(TASK *head, int *id, int *priority) {
    TASK *current = head;
    while (current->id != *id && current->next != NULL) {
        current = current->next;
    }
    if (current->id == *id) {
        current->priorita = *priority;
    } else {
        printf("ERRORE! L'ID scelto non e' presente.\n");
    }
}

TASK *changePolicy(TASK *head) {
    if (policy == PRIORITA) {
        policy = ESECUZIONIRIMANENTI;
    } else {

```

```
        policy = PRIORITA;  
    }  
    return sortingByPolicy(head);  
}  
  
TASK *sortingByPolicy(TASK *head) {  
    TASK *new_head = malloc(sizeof(TASK));  
    TASK *current = head;  
    while (current->next != NULL) {  
        new_head = push(new_head, *current);  
        current = current->next;  
    }  
    head = NULL;  
    free(head);  
    return new_head;  
}
```

ESECUTORE DI COMANDI

Introduzione:

La funzionalità principale di questo programma è quella di eseguire qualsiasi comando shell e scrivere il risultato dell'esecuzione in un file, uno per ogni comando richiesto.

I file verranno creati nella directory dove è situato il file sorgente ("CommandExecutor.c") ed ordinati in base all'ordine di arrivo dei comandi richiesti. L'ordine dei file è realizzato tramite un prefisso "out." uguale per tutti ed un numero intero che aumenta ogni volta che si esegue un nuovo comando.

Il programma può essere eseguito in due modi:

- Sequentiale, con comando "-s" passato come argomento.

In questa tipologia di esecuzione, i comandi vengono eseguiti uno alla volta, salvando il relativo standard output nel file opportuno.

oppure

- Parallelo, con comando "-p" passato come argomento;

In questa tipologia di esecuzione, si salvano tutti i comandi inseriti dall'utente in modo tale da eseguirli tutti insieme in parallelo, ovvero generando n processi (dove n è il numero complessivo di righe di comando inserite) ognuno dei quali dedica la sua esecuzione ad una singola riga di comando e salva nel file appropriato lo standard output.

Prima di eseguire le richieste da input, vengono rimossi (se ci sono) i file "out.*" (dove "*" indica il numero associato), in maniera tale da non incorrere in una crescita inopportuna di memoria.

Descrizione delle funzioni:

int main(int argc, char **argv)

Parametri in ingresso: argomento "-s" o "-p"

Descrizione:

- elimina tutti i file "out.*" (se ci sono) dalla directory tramite il comando "rm out.*"
- controlla che ci siano stati errori nell'inserimento degli argomenti al momento dell'esecuzione del programma
- se è stato eseguito il programma con argomento "-s" ovvero in maniera sequenziale, genera un solo processo figlio tramite la funzione **generateProcess()** che si occuperà dell'esecuzione e relativa scrittura su file del comando richiesto
- se è stato eseguito il programma con argomento "-p" ovvero in maniera parallela, crea un buffer che contiene tutti i comandi inseriti dall'utente, genera n processi figli dove n è il numero dei comandi inseriti tramite la funzione **generateProcess()**. Ogni processo si occuperà di eseguire e scrivere sul file appropriato il risultato di tale esecuzione del comando
- aspetta che tutti i figli siano terminati tramite la funzione **wait()**

Valori di ritorno: 0, per indicare la terminazione del programma senza errori.

int strInput(char str[])

Parametri in ingresso: stringa comando

Descrizione:

- prende la richiesta del comando dallo standard input (massimo 100 caratteri)
- aggiunge il carattere '\0' al termine della stringa

Valori di ritorno: lunghezza della stringa inserita

void generateProcess(char *command, int fileNum)

Parametri in ingresso: stringa comando, numero suffisso file

Descrizione:

- genera un processo figlio tramite la funzione di libreria **fork()**
- controlla che la **fork()** sia andata a buon fine
- il processo generato chiama la funzione **executeProcess()** passandogli la stringa del comando e il suffisso del file

Valori di ritorno: nessuno

void executeProcess(char *command, int fileNum)

Parametri in ingresso: stringa comando, numero suffisso file

Descrizione:

- chiama la funzione **getCommand()** passandogli la stringa del comando ed un puntatore ad array di char usato per salvare il comando ed i suoi eventuali argomenti
- apre il file "out.*" dove l'asterisco viene sostituito da fileNum, il file viene aperto con i flags: O_CREAT (se il file "out.*" non esiste viene creato), O_RDWR (apre in lettura e scrittura), O_TRUNC (viene troncato se il file esiste già), S_IRUSR (permessi in lettura solo per il proprietario del file), S_IWUSR (permessi in scrittura solo per il proprietario del file)
- controlla che la **open()** si andata a buon fine
- copia il descrittore del file nel descrittore dello standard input/error tramite la funzione **dup2()** in modo tale da reindirizzare lo standard output e lo standard error al file "out.*"
- esegue il comando tramite la system call **execvp()** passandogli il comando e la lista degli eventuali argomenti
- controlla che la funzione **execvp()** sia andata a buon fine

Valori di ritorno: nessuno

char *myStrDup(char *str)

Parametri in ingresso: stringa comando

Descrizione:

- crea una nuova stringa, alloca tramite **malloc()** n+1 byte dove n è la lunghezza di str
- copia tramite **strcpy()** il contenuto di str nella nuova stringa creata precedentemente

Valori di ritorno: la nuova stringa copia di str

void getCommand(char* command, char* commandExec[MAX])

Parametri in ingresso: stringa comando, puntatore ad array di char che conterrà il comando

Descrizione:

- spezza la stringa command, prende tutto quello che c'è prima del carattere spazio " "
- tramite la funzione di libreria **strtok()**, tale funzione restituisce il puntatore al carattere spazio
- copia in commandExec il contenuto della stringa spezzata, richiama **strtok()**, questa volta chiamandola con il parametro NULL in maniera tale che il puntatore punti sempre allo spazio trovato in precedenza; cicla fintanto che non è stata esaminata tutta la stringa command

Valori di ritorno: nessuno

Simulazione:

La simulazione mostra un esempio di funzionamento del programma, inserendo 3 comandi e visualizzando il risultato all'interno dei file creati.

Compilazione del programma tramite gcc e rinominando l'eseguibile tramite -o.

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/CommandExecutor/src
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ gcc CommandExecutor.c -o CommandExecutor
```

Esecuzione del programma in modo sequenziale tramite l'argomento "-s".

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/CommandExecutor/src
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ gcc CommandExecutor.c -o CommandExecutor
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ ./CommandExecutor -s
```

Possiamo vedere come i 3 comandi vengono eseguiti sequenzialmente uno dopo l'altro. Ogni volta che viene inserito un nuovo comando viene generato un processo figlio che si occupa della sua esecuzione.

Il programma termina quando viene inserita una stringa vuota.

I tre comandi nell'esempio sono: ls -la

man rm
cat /etc/password

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/CommandExecutor/src
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ gcc CommandExecutor.c -o CommandExecutor
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ ./CommandExecutor -s
Command Executor: SEQUENTIAL, insert an empty string for exit
Insert command/s:
ls -la
Process 3467, generate...
Insert command/s:
man rm
Process 3472, generate...
Insert command/s:
cat /etc/password
Process 3492, generate...
Insert command/s:

Exit...
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$
```

Come possiamo vedere il file out.1 contiene il risultato del comando ls -la il quale mostra la lista dei file presenti nella directory.

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/CommandExecutor/src
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ cat out.1
totale 28
drwxr-xr-x 2 lorenzo lorenzo 4096 set 17 05:58 .
drwxr-xr-x 5 lorenzo lorenzo 4096 set 16 16:22 ..
-rw-r-xr-x 1 lorenzo lorenzo 13688 set 17 05:58 CommandExecutor
-rw-r--r-- 1 lorenzo lorenzo 2990 set 17 05:58 CommandExecutor.c
-rw----- 1 lorenzo lorenzo 0 set 17 05:58 out.1
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$
```

Il file out.2 contiene il risultato del comando man rm il quale ci mostra il manuale di rm.

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/CommandExecutor/src
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ cat out.2
RM(1)                                         User Commands
NAME
      rm - remove files or directories
SYNOPSIS
      rm [OPTION]... [FILE]...
DESCRIPTION
      This manual page documents the GNU version of rm.  rm removes each specified file.  By
      default, it does not remove directories.

      If the -I or --interactive=once option is given, and there are more than three files or the
      -r, -R, or --recursive are given, then rm prompts the user for whether to proceed with the
      entire operation.  If the response is not affirmative, the entire command is aborted.

      Otherwise, if a file is unwritable, standard input is a terminal, and the -f or --force
      option is not given, or the -i or --interactive=always option is given, rm prompts the user
```

Infine il file out.3 contiene il risultato dell'esecuzione del comando cat etc/password. Genera un errore in quanto la directory richiesta non esiste, ciò lo si nota andando ad analizzare il contenuto del file out.1.

```
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ cat out.3
cat: /etc/password: File o directory non esistente
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$
```

Mettiamo in evidenza anche un'esecuzione del programma in modalità parallela come indicato dall'argomento "-p". I 3 comandi vengono prima memorizzati e successivamente eseguiti in parallelo dai 3 processi figli.

```
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ gcc CommandExecutor.c -o CommandExecutor
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$ ./CommandExecutor -p
Command Executor: PARALLEL, insert an empty string for exit
Insert command/s:
ls -la
Insert command/s:
man rm
Insert command/s:
cat /etc/password
Insert command/s:

Process 3983, generate...
Process 3982, generate...
Process 3981, generate...
lorenzo@lorenzo-VirtualBox:~/workspaceC/CommandExecutor/src$
```

Codice:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAX 100
#define SEQUENTIAL "-s"
#define PARALLEL "-p"

int strInput(char str[]);
void executeProcess(char *command, int fileNum);
char *myStrDup(char *str);
void getCommand(char* command, char* commandExec[MAX]);
void generateProcess(char *command, int fileNum);

int main(int argc, char **argv) {
    int fileNum = 0;
    char command[MAX];
    int status;

    /* rimuove tutti i file creati precedentemente con prefisso "out.", per
    non appesantire la directory */
    system("rm -f out.*");

    if (argc > 1 && !strcmp(argv[1], SEQUENTIAL)) {
```

```

        printf("Command Executor: SEQUENTIAL, insert an empty string for
exit\n");
        while (strInput(command) > 0) {
            fileNum++;
            generateProcess(command, fileNum);
            wait(&status);
        }
    } else if (argc > 1 && !strcmp(argv[1], PARALLEL)) {
        printf("Command Executor: PARALLEL, insert an empty string for
exit\n");
        int i = 0, process = 0;
        char buffer[MAX][MAX];
        while (strInput(command) > 0) {
            memcpy(&buffer[i++], &command[0], sizeof(buffer[i]));
        }
        for (process = 0; process < i; process++) {
            generateProcess(buffer[process], ++fileNum);
        }
        wait(&status);
    } else {
        perror("call the program with -s or -p arguments");
    }

    printf("\nExit...\n");
    usleep(10000);
    exit(0);
}

void generateProcess(char *command, int fileNum) {
    pid_t pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    if (pid == 0) { // child
        printf("Process %d, generate...\n", getpid());
        executeProcess(command, fileNum);
    }
}

int strInput(char str[]) {
    printf("Insert command/s:\n");
    fgets(str, MAX, stdin);

    str[strlen(str) - 1] = '\0';
    return strlen(str);
}

void executeProcess(char *command, int fileNum) {
    char *commandExec[MAX];
    getCommand(command, commandExec);

    char catedString[10];
    sprintf(catedString, "out.%d", fileNum);
    int fd = open(catedString, O_CREAT | O_RDWR | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("open error");
    }

    dup2(fd, STDOUT_FILENO);      // make stdout go to file
    dup2(fd, STDERR_FILENO);      // make stderr go to file
    close(fd);
}

```

```
if (execvp(commandExec[0], commandExec) < 0) {
    system(commandExec[0]);
    perror("Error exec");
    exit(-1);
}

void getCommand(char* command, char* commandExec[MAX]) {
    int i = 0;
    char* str = strtok(command, " ");
    while (str != NULL) {
        commandExec[i++] = myStrDup(str);
        str = strtok(NULL, " ");
    }
    commandExec[i] = NULL;
}

char *myStrDup(char *str) {
    char *other = malloc(strlen(str) + 1);
    if (other != NULL) {
        strcpy(other, str);
    }
    return other;
}
```

SCAMBIO DI MESSAGGI

Introduzione:

La funzionalità principale di questo programma è quella di permettere lo scambio di messaggi tra processi client gestito da un processo server.

La comunicazione tra client è realizzata come segue: un client invia un messaggio a un altro client o a un gruppo di client inviandolo al server, il quale si occupa di recapitarlo.

Il server svolge le seguenti funzioni:

- mantiene la lista dei client connessi.
- riceve messaggi testuali da inoltrare ai client.

Il client esegue le seguenti attività:

- si connette al server, si disconnette dal server.
- invia e riceve messaggi testuali.

Per realizzare ciò abbiamo utilizzato dei segnali (**handle_ctrlC_signal** e **handle_message_signal**) e pipe con nome: un unico pipe con nome (creato dal server) utilizzato da tutti i client in modalità scrittura e dal server in modalità lettura (**client_to_server**), un pipe con nome per ogni client (creato dal server) utilizzato dal server in modalità scrittura e dal client in modalità lettura (**server_to_client[BUFSIZ]**, cioè un array in cui verranno memorizzate le pipe).

server_to_client[0] è una pipe utilizzata dal server per comunicare ai client appena connessi il proprio ID e di appoggio per altre operazioni minori.

Il server verrà avviato in un terminale distinto, così come ogni client, e attenderà informazioni da parte di essi.

Ciascun client presenterà un menù con 5 possibili scelte:

- 1. *Connect to Server*
- 2. *Get Client list*
- 3. *Send message to another Client or a group of Client*
- 4. *Disconnect*
- 5. *Exit*

Il server riceve la richiesta dal client ed esegue ciò che essa prevede.

In caso di connessione il client verrà identificato tramite un ID univoco a partire da 1 e inserito nella lista dei client connessi, mentre in caso di disconnessione (opzione 4) o terminazione (sia con opzione 5 che premendo Ctrl-C) il client verrà eliminato da tale lista.

Descrizione delle funzioni – Server:

int main()

Parametri in ingresso: nessuno

Descrizione:

- dichiara un path (rappresentato dalla stringa ***cspath**) con descrittore di file la variabile **int** **cs**, utilizzato per la connessione del client e dichiara una matrice **clientConnected[BUFSIZ][4]** in cui memorizzare i client (massimo 3 cifre).
- dichiara un descrittore di file **int** **client_to_server** e un **_path** per la FIFO (named pipe) **char** ***myfifo** a cui si assegna la stringa **/tmp/client_to_server_fifo**.

- dichiara un array di descrittori di file **int** `server_to_client[BUFSIZ]` e un path per la FIFO (named pipe) **char** *`myfifo2` a cui si assegna la stringa `/tmp/server_to_client_fifo`.
- crea le pipe con nome con la funzione **mkfifo** (0666 come permessi), apre e restituisce il fd: uno in modalità lettura **open**(`myfifo`, `0_RDONLY`), l'altro in modalità scrittura **open**(`myfifo2`, `0_WRONLY`).
- legge dal client l'opzione scelta ed entra nel costrutto if corrispondente.
- l'opzione 1 connette un client assegnandogli un ID univoco e aggiungendolo alla matrice `clientConnected` dei client connessi.
- l'opzione 2 invia, al client richiedente, la lista dei client connessi uno per uno.
- l'opzione 3 riceve, dal client richiedente, una lista di client a cui inviare un messaggio (controlla e segnala se un ID non è presente) e il messaggio stesso, e lo recapita ai client selezionati.
- le opzioni 4 e 5 eliminano da `clientConnected` il client disconnesso o terminato.
- al termine dell'esecuzione dell'opzione scelta pulisce i buffer utilizzati, riporta la stringa `cspath` alla condizione iniziale (`/tmp/csFifo`) e chiude il fd `server_to_client[0]` reinizializzandolo a 0.

Valori di ritorno: 0, per indicare la terminazione del programma senza errori.

Descrizione delle funzioni – Client:

int main()

Parametri in ingresso: nessuno

Descrizione:

- dichiara un path (rappresentato dalla stringa `*cspath`) con descrittore di file la variabile **int** `cs`, utilizzato per la connessione del client.
- dichiara un descrittore di file **int** `client_to_server` e un path per la FIFO (named pipe) **char** *`myfifo` a cui si assegna la stringa `/tmp/client_to_server_fifo`.
- dichiara un array di descrittori di file **int** `server_to_client[BUFSIZ]` e un path per la FIFO (named pipe) **char** *`myfifo2` a cui si assegna la stringa `/tmp/server_to_client_fifo`.
- l'opzione 1 connette un client se non già connesso e riceve un ID univoco dal server.
- l'opzione 2 richiede al server la lista dei client connessi e la stampa.
- l'opzione 3 invia al server una lista di client (uno alla volta) a cui inviare un messaggio (riceve dal server una richiesta di reinserimento se il client inserito non è presente) e il messaggio stesso.
- l'opzione 4 invia al server il proprio ID per essere eliminato dalla lista dei client connessi, impostando a 0 la variabile booleana **int** `connected`.
- l'opzione 5 invia al server il proprio ID per essere eliminato dalla lista dei client connessi e ritorna il valore 0 per terminare l'esecuzione del programma.

Valori di ritorno: 0, per indicare la terminazione del programma senza errori.

Tralasciando la funzione di stampa del menù **void printMenu()**, passiamo ad analizzare le funzioni utilizzate per la realizzazione dei segnali:

- **void handle_ctrlC_signal(int signal)**
- **void handle_message_signal(int signal)**

Per la gestione dei segnali utilizziamo la funzione `sigaction` molto più flessibile e robusta rispetto alla classica `signal`.

Definiamo quindi due variabili di tipo struct `sigaction`, in modo da far puntare il campo interno della struct "sa_handler" alla funzione che verrà eseguita come handler del segnale invocato.

Il campo `sa_handler` di `sa1` punterà alla funzione `handle_ctrlc_signal()` mentre il campo `sa_handler` di `sa2` punterà alla funzione `handle_message_signal()`.

A questo punto si esegue la funzione sigaction() passandogli la macro del segnale interessato; la variabile sa1 che contiene l'indirizzo della funzione handler e il valore NULL in modo da installare la funzione handler specificata da sa1.

Infine si controlla che tutto sia andato a buon fine, verificando il valore di ritorno della sigaction. In caso di errore (-1) si stampa un messaggio comunicando il fallimento all'utente con il relativo codice errno.

Lo stesso si esegue per sa2 che gestirà un eventuale segnale di ricezione messaggio da parte di un client.

Simulazione:

La simulazione mostra un esempio di funzionamento del programma, lanciandolo viene mandato in esecuzione Server.c, che rimane in attesa di connessioni, disconnessioni e richieste da parte dei programmi Client.c

Compilazione ed esecuzione di Server.c, a questo punto il server è acceso e attende una connessione da parte dei client.

```
Server
x - o lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src$ gcc Server.c -o Server
lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src$ ./Server
Server ON.
```

Compilazione ed esecuzione di Client.c, viene eseguito un processo client, il quale si interfaccia con il server in attesa tramite un menu a scelta.

E' banale dire che l'unica scelta che il client può fare in questo momento è la connessione al server (1) oppure l'uscita dal programma (5)

Viene eseguita la connessione al server tramite il comando (1) del menu.

Il server riceve la richiesta di connessione da parte del client, associandogli un ID univoco e creando una pipe specifica che verrà utilizzata per la comunicazione.

```
x - o lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src$ gcc Client.c -o Client
lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src$ ./Client
===== Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit
Status: not connected
=====
```

```
Client:1
x - o lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src$ gcc Client.c -o Client
lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src$ ./Client
===== Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit
Status: not connected
=====
1
Connected, your id is Client:1
```

In un altro terminale viene eseguito un nuovo client e creata la connessione con il server. Come si nota questo client ha un diverso ID rispetto al client eseguito precedentemente.

```
Client:2
x - o lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src$ ./Client
=====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit
Status: not connected
=====
1
Connected, your id is Client:2
```

Dunque al momento abbiamo due client diversi in due terminali distinti, entrambi connessi al server. A questo punto il Client:1 richiede al server la lista di tutti i client connessi tramite il comando 2, dopodiché invierà un messaggio al Client:2 con il comando 3.

Client:1

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/Client/src
=====
1
Connected, your id is Client:1
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: connected
=====
2
Connected IDs: 1 2
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: connected
=====
3
Insert Client IDs, 0 to quit.
Insert Client ID: 2
Insert Client ID: 0
Insert done.
Input message to server: Hi I'm Client:1
```

Client:2

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/Client/src
Connected, your id is Client:2
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: connected
=====
You have a new message from Client 1: Hi I'm Client:1
```

Come possiamo vedere il messaggio “Hi I'm Client:1” viene mandato al server dal Client 1, ed inoltrato al Client 2 da parte del server. Ovviamente il server comunica al Client 2 anche il nome del mittente.

Il messaggio che Client 2 riceve è il seguente:

“You have a new message from Client 1: Hi I'm Client:1”

Simuliamo adesso un messaggio inviato ad un gruppo di client.

Supponiamo che venga mandato in esecuzione un nuovo client (Client:3), venga connesso al server e che decida di mandare un messaggio sia al Client:1 che al Client:2. La situazione è la seguente:

Client:3

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/Client/src
lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src$ ./Client
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: not connected
=====
1
Connected, your id is Client:3
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: connected
=====
3
Insert Client IDs, 0 to quit.
Insert Client ID: 1
Insert Client ID: 2
Insert Client ID: 0
Insert done.
Input message to server: Hi I'm client 3
```

Client:1

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/Client/src
=====
1
Connected, your id is Client:1
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: connected
=====
You have a new message from Client 3: Hi I'm client 3
```

Client:2

```
x - lorenzo@lorenzo-VirtualBox: ~/workspaceC/Client/src
Connected, your id is Client:2
=====
Menu =====
1. Connect to Server
2. Get Client list
3. Send message to another Client or a group of Client
4. Disconnect
5. Exit

Status: connected
=====
You have a new message from Client 3: Hi I'm client 3
```

Possiamo notare come il messaggio da parte di Client:3 venga recapitato sia a Client:1 sia a Client:2.

Infine la disconnessione da parte di un client al server può essere effettuata in 3 modi diversi, ossia, tramite i comandi (4) e (5) oppure tramite il comando CTRL+C (comando usato in unix per terminare un processo inviando il segnale SIGINT).

Mostriamo dunque il funzionamento di questi comandi:

(4). Disconnessione senza uscita dal programma Client.c

Il server riceve il comando 4 da parte del Client:1 ed esegue la disconnessione.

Client:1 <pre>x - lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src ===== Menu ===== 1. Connect to Server 2. Get Client list 3. Send message to another Client or a group of Client 4. Disconnect 5. Exit Status: connected ===== 4 ===== Menu ===== 1. Connect to Server 2. Get Client list 3. Send message to another Client or a group of Client 4. Disconnect 5. Exit Status: not connected =====</pre>	Server <pre>x - lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src\$./Server Server ON. Received: 1 Connected: OK Received: 4 Client:1 has disconnected from the Server. []</pre>
---	---

(5). Uscita dal programma Client.c e relativa disconnessione dal server.

Il server riceve il comando 5 da parte del Client:1 ed esegue la disconnessione.

Client:1 <pre>x - lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src 3. Send message to another Client or a group of Client 4. Disconnect 5. Exit Status: not connected ===== 1 Connected, your id is Client:1 ===== Menu ===== 1. Connect to Server 2. Get Client list 3. Send message to another Client or a group of Client 4. Disconnect 5. Exit Status: connected ===== 5 lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src\$ []</pre>	Server <pre>x - lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src\$./Server Server ON. Received: 1 Connected: OK Received: 5 Client:1 has disconnected from the Server. []</pre>
---	---

(CTRL+C). Terminazione del programma Client.c tramite ctrl+c.

Il programma termina, invia un segnale di SIGINT al server che immediatamente disconnette il client (se connesso).

Client:1 <pre>x - lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src 4. Disconnect 5. Exit Status: not connected ===== 1 Connected, your id is Client:1 ===== Menu ===== 1. Connect to Server 2. Get Client list 3. Send message to another Client or a group of Client 4. Disconnect 5. Exit Status: connected ===== ^C Signal catch. Exit... lorenzo@lorenzo-VirtualBox:~/workspaceC/Client/src\$ []</pre>	Server <pre>x - lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src lorenzo@lorenzo-VirtualBox:~/workspaceC/Server/src\$./Server Server ON. Received: 1 Connected: OK Received: 5 Client:1 has disconnected from the Server. []</pre>
---	---

Codice:

-Server.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

int main() {
    int clientID = 0;
    int cs; // file descriptor per csPath
    char *csPath = "/tmp/csFifo"; // utilizzato per connessione client
    char clientConnected[BUFSIZ][4]; // max 3 cifre per ID client
    int id;
    int pidProc[BUFSIZ];
    int numConnected = 0;

    int client_to_server; // fd per myFifo
    char *myfifo = "/tmp/client_to_server_fifo";

    int server_to_client[BUFSIZ]; // array di fd per myFifo2
    char *myfifo2 = "/tmp/server_to_client_fifo";

    char buf[BUFSIZ], buf2[BUFSIZ]; // buffer temporanei di appoggio
    char csFifo[BUFSIZ], scFifo[BUFSIZ];

    printf("Server ON.\n");

    /* crea la FIFO (pipe con nome) */
    mkfifo(myfifo, 0666);
    mkfifo(myfifo2, 0666);

    /* apre le due FIFO */
    client_to_server = open(myfifo, O_RDONLY);
    server_to_client[0] = open(myfifo2, O_WRONLY);

    while (1) {
        server_to_client[0] = open(myfifo2, O_WRONLY);
        read(client_to_server, buf, sizeof(buf));

        if (strcmp("1", buf) == 0) { // 1. Connect to Server
            char client[10];
            printf("Received: %s\n", buf); // stampa l'opzione ricevuta
            clientID++;
            sprintf(clientConnected[clientID - 1], "%d", clientID);
            numConnected++;
            write(server_to_client[0], &clientID, sizeof(clientID));
            sprintf(buf, "your id is %d", clientID);
            sprintf(client, "Client:%d", clientID);

            unlink(myfifo2); // elimina il file per ripulirlo
            mkfifo(myfifo2, 0666); // ricrea la fifo
            write(server_to_client[0], "OK", sizeof("OK")); // aspetta il
Client
            server_to_client[clientID] = open(myfifo2, O_WRONLY);

            write(server_to_client[clientID], client, BUFSIZ);

            /* nuova connessione per dare un id al client */
        }
    }
}
```

```

        char *tmp = client;
        strcat(strcat(csFifo, csPath), tmp);
        write(server_to_client[clientID], csFifo, BUFSIZ);
        read(client_to_server, buf, sizeof(buf)); // aspetta il Client

        csPath = csFifo;
        mkfifo(csPath, 0666);
        cs = open(csPath, O_RDONLY);

        /* prendo il pid del processo client */
        int pid;
        read(cs, &pid, sizeof(pid));
        pidProc[clientID] = pid;

        read(cs, client, sizeof(client));
        printf("Connected: %s\n", client);

    } else if (strcmp("2", buf) == 0) { // 2. Get Client list
        int i = 0, j = 0;
        printf("Received: %s\n", buf);
        write(server_to_client[0], &numConnected,
sizeof(numConnected));
        while (i < numConnected) { // invia i client connessi uno per
uno
            if (strcmp(clientConnected[j], "") != 0) {
                write(server_to_client[0], clientConnected[j],
BUFSIZ);
                i++;
            }
            j++;
        }
    }

    } else if (strcmp("3", buf) == 0) { // 3. Send message to another
Client or a group of Client
        char clientIDs[BUFSIZ], message[BUFSIZ], clients[BUFSIZ][3];
        int presentID; // booleano per presenza ID
        int countPresentIDs = 0;
        int i, j;
        int seqTerminated;
        int idRecipient;
        printf("Received: %s\n", buf);
        /* riceve il nome del cliente che invia il messaggio */
        read(client_to_server, &id, sizeof(id));
        do {
            i = j = 0;
            presentID = 1;
            /* riceve la lista dei client a cui inviare il messaggio
*/
            read(client_to_server, clientIDs, sizeof(clientIDs));
            seqTerminated = strcmp(clientIDs, "0") == 0; // si
termina con il carattere 0

            while (i < numConnected) {
                if (strcmp(clientConnected[j], "0") != 0) {
                    /* se le due stringhe sono uguali la
funzione ritorna 0 */
                    if (strcmp(clientConnected[j], clientIDs) !=
0) {
                        presentID = 0; // ID non presente
                    } else {
                        presentID = 1; // ID presente
                    }
                }
            }
        }
    }
}

```

```

clientIDs);
                                strcpy(clients[countPresentIDs++],
                                break;
                            }
                            i++;
                        }
                        j++;
                    }
                    if(!seqTerminated) {
                        write(server_to_client[0], &presentID,
sizeof(presentID));
                    } else {
                        presentID = 0;
                        write(server_to_client[0], &presentID,
sizeof(presentID));
                        break; // esce se nessun ID e' presente e la
sequenza e' terminata
                    }
                } while (!seqTerminated);

                if (countPresentIDs > 0) {
                    /* riceve il messaggio da recapitare ai client */
                    read(client_to_server, message, sizeof(message));
                    /* invia messaggio ai client presenti */
                    for (i = 0; i < countPresentIDs; i++) {
                        idRecipient = atol(clients[i]);
                        write(server_to_client[idRecipient], &id,
sizeof(id));
                        write(server_to_client[idRecipient], message,
sizeof(message));
                        kill(pidProc[idRecipient], SIGUSR1);
                    }
                }
            }

} else if (strcmp("4", buf) == 0 || strcmp("5", buf) == 0) { // 4.
Disconnect e 5. Exit
    printf("Received: %s\n", buf);
    read(client_to_server, &id, sizeof(id));
    close(server_to_client[id]);
    memset(clientConnected[id - 1], 0, sizeof(clientConnected[id -
1]));
    numConnected--;
    printf("Client:%d has disconnected from the Server.\n", id);
}

/* clean buf from any data */
memset(buf, 0, sizeof(buf));
memset(buf2, 0, sizeof(buf2));
memset(csFifo, 0, sizeof(csFifo));
memset(scFifo, 0, sizeof(scFifo));
csPath = "/tmp/csFifo";
close(server_to_client[0]);
server_to_client[0] = 0;
}

close(client_to_server);

unlink(myfifo);
unlink(myfifo2);
return 0;
}

```

-Client.c

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

#define flushscanf while ((getchar()) != '\n')
#define microsec 10000

void printMenu();
void handle_ctrlC_signal(int signal);
void handle_message_signal(int signal);

int id = 0, connected = 0;
int *ptrScBuffer; // utilizzato per segnali
int *ptrCsBuffer; // utilizzato per segnali

int main() {
    char clientID[BUFSIZ];
    char message[BUFSIZ];
    char clientServerPath[BUFSIZ];
    int cs; // file descriptor per csPath
    char *csPath = "/tmp/csFifo";
    int numConnected = 0;

    int client_to_server; // fd per myFifo
    ptrCsBuffer = &client_to_server;
    char *myfifo = "/tmp/client_to_server_fifo";

    int server_to_client[BUFSIZ]; // array di fd per myFifo2
    ptrScBuffer = server_to_client;
    char *myfifo2 = "/tmp/server_to_client_fifo";
    /* richiama la funzione di libreria sigaction su entrambi i tipi di struct */
    /*
    struct sigaction sa1, sa2;

    sa1.sa_handler = &handle_ctrlC_signal;
    sa2.sa_handler = &handle_message_signal;
    if (sigaction(SIGINT, &sa1, NULL) < 0) {
        perror("Error: cannot handle SIGINT"); // Should not happen
    }
    if (sigaction(SIGUSR1, &sa2, NULL) < 0) {
        perror("Error: cannot handle SIGUSR1"); // Should not happen
    }

    int choice = 0;
    while (choice != 5) {
        printMenu();
        choice = 0; // rимetto a 0 perche' dopo una signal salta la scanf e
        prende choice com'era prima
        scanf("%d", &choice);

        switch (choice) {
            /* Connessione, con la quale il client si registra presso il server.
    */
            case 1:
                if (!connected) {
```

```

client_to_server = open(myfifo, 0_WRONLY);
server_to_client[0] = open(myfifo2, 0_RDONLY);

write(client_to_server, "1", sizeof("1"));
read(server_to_client[0], &id, sizeof(id));

read(server_to_client[0], message, sizeof(message)); // aspetta il Server

server_to_client[id] = open(myfifo2, 0_RDONLY);

read(server_to_client[id], message, sizeof(message));
printf("Connected, your id is %s\n\n", message);
connected = 1;

read(server_to_client[id], message, sizeof(message));
sprintf(clientServerPath, "%s", message);
csPath = clientServerPath;

write(client_to_server, "OK", sizeof("OK"));

cs = open(csPath, 0_WRONLY);
int pid = getpid();
write(cs, &pid, sizeof(pid));
write(cs, "OK", 3);

} else {
    printf("Error: client already connected!\n\n");
}
break;

/*
 * Richiesta elenco ID dei client registrati, con la quale si
 * al server l'elenco dei client attualmente registrati.
 */
case 2:
    if (connected) {
        client_to_server = open(myfifo, 0_WRONLY);
        server_to_client[0] = open(myfifo2, 0_RDONLY);
        write(client_to_server, "2", sizeof("2"));
        /* riceve il numero di client connessi */
        read(server_to_client[0], &numConnected,
sizeof(numConnected));
        printf("Connected IDs:");
        int i = 0;
        while (i < numConnected) {
            read(server_to_client[0], message,
sizeof(message)); // riceve i client connessi uno per uno
            i++;
            printf(" %s", message);
        }
        printf("\n\n");
    } else {
        printf("Error: connect before running this
command\n\n");
    }
}
break;

/*
 * Invio di un messaggio testuale a un altro client o
 * a un insieme di client (specificandone l'ID).

```

```

        */
case 3:
    if (connected) {
        client_to_server = open(myfifo, 0_WRONLY);
        server_to_client[0] = open(myfifo2, 0_RDONLY);
        int presentID = 0; // booleano per presenza ID
        int countPresentIDs = 0;

        write(client_to_server, "3", sizeof("3"));
        usleep(microsec); // aspetta che il server elabori
        write(client_to_server, &id, sizeof(id)); // invia al

server il proprio ID

        printf("Insert Client IDs, 0 to quit.\n");
        do {
            printf("Insert Client ID: ");
            scanf("%s", clientID);

            countPresentIDs++;
            /* invia al server l'ID del client a cui inviare
            write(client_to_server, clientID,
            read(server_to_client[0], &presentID,
            if (presentID == 0 && strcmp(clientID, "0") != 0)
            { // il client inserito non e' connesso
                printf("Server feedback: this ID is not
connected, try another.\n");
                countPresentIDs--;
            }
            if (presentID == 0 && strcmp(clientID, "0") == 0)
            {
                printf("Insert done.\n");
                countPresentIDs--;
                break; // i client inseriti sono tutti
connessi ed esce dal ciclo
            }
        } while (countPresentIDs == 0 || strcmp(clientID, "0") !=
0);

        if (countPresentIDs > 0) {
            /* c'è rimanuto nello stdin uno '\n' dovuto
all'ultimo inserimento */
            flushscanf // macro, ripuliamo lo stdin
            memset(message, 0, sizeof(message));

            printf("Input message to server: ");
            fgets(message, 100, stdin); // 100 massima
lunghezza messaggio, scanf non riconosce gli spazi

            write(client_to_server, message, sizeof(message));
        } else {
            printf("No client selected.\n\n");
        }
    } else {
        printf("Error: connect before running this
command\n\n");
    }
}

break;

```

```

        /*
         * Disconnessione, con la quale il client richiede la
cancellazione
         * della registrazione presso il server.
        */
    case 4:
        if (connected) {
            client_to_server = open(myfifo, O_WRONLY);
            usleep(microsec); // aspetta che il server elabori
            write(client_to_server, "4", sizeof("4"));
            usleep(microsec); // aspetta che il server elabori
            write(client_to_server, &id, sizeof(id)); // invia al
server il proprio ID

            close(client_to_server);
            close(server_to_client[id]);
            connected = 0;
        } else {
            printf("Error: connect before running this
command\n\n");
        }
        break;

        /* Uscita dal programma. */
    case 5:
        if (connected) {
            client_to_server = open(myfifo, O_WRONLY);
            usleep(microsec); // aspetta che il server elabori
            write(client_to_server, "5", sizeof("5"));
            usleep(microsec); // aspetta che il server elabori
            write(client_to_server, &id, sizeof(id)); // invia al
server il proprio ID

            close(client_to_server);
            close(server_to_client[id]);
        }
        return 0;

    default:
        if (choice != 0) {
            printf("\nError: insert a number between 1 and 5\n\n");
        }
        break;
    }
}

close(client_to_server);
close(server_to_client[0]);
return 0;
}

void printMenu() {
    printf("===== Menu =====\n");
    printf("1. Connect to Server\n");
    printf("2. Get Client list\n");
    printf("3. Send message to another Client or a group of Client\n");
    printf("4. Disconnect\n");
    printf("5. Exit\n\n");

    if (connected == 1) {
        printf("Status: connected\n");
    } else {

```

```
        printf("Status: not connected\n");
    }
    printf("=====\\n");
}

/* segnale che entra in esecuzione quando viene premuto Ctrl-C */
void handle_ctrlC_signal(int signal) {
    printf("\nSignal catch. Exit...\\n");
    write(*ptrCsBuffer, "5", sizeof("5"));
    usleep(microsec);
    write(*ptrCsBuffer, &id, sizeof(id));

    close(*ptrCsBuffer);
    close(*(ptrScBuffer + id));
    exit(0);
}

/* segnale che entra in esecuzione quando viene ricevuto un messaggio */
void handle_message_signal(int signal) {
    char message[BUFSIZ];
    int idSender;
    read(*(ptrScBuffer + id), &idSender, sizeof(idSender));
    read(*(ptrScBuffer + id), message, sizeof(message));
    printf("\nYou have a new message from Client %d: %s\\n", idSender,
message);
}
```