



Relazione sul Progetto dell'Esame di
Sistemi Operativi
Anno Accademico 2016/17

Contino Niccolò - 5831593 – niccolo.contino@stud.unifi.it
D'Autilia Mattia - 5765968 – mattia.dautilia@stud.unifi.it
Di Falco Davide - 5121746 – davide.difalco@stud.unifi.it

1 Dicembre 2017

ESERCIZIO 1

EVIDENZA DEL CORRETTO FUNZIONAMENTO DEL PROGRAMMA RISPETTO ALLE SPECIFICHE FORNITE

L'obiettivo del primo esercizio è 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 o esecuzioni rimanenti.

La nostra soluzione è stata implementata utilizzando una lista di strutture collegata con puntatori, inoltre per l'ordinamento è stato implementato l'algoritmo di quicksort.

Abbiamo organizzato il codice in due file : una header file struct.h ed un programma, Scheduler.c.

All'interno di struct.h viene definito la struct T_structure, ovvero una struttura che descrive il task. Ogni task è formato da 5 campi che rappresentano un processo all'interno della nostra implementazione:

1. ID : Un numero intero univoco che viene automaticamente assegnato alla creazione del task.
2. Name: Nome del task, di massimo 8 caratteri, scelto dall'utente alla creazione.
3. Priority : Numero intero che rappresenta la priorità del task.
4. Execution : Numero intero che rappresenta il numero di esecuzioni rimanenti (detti burst) del task.
5. *Next : Puntatore al task successivo.

Questa struttura verrà utilizzata dal programma Scheduler.c per l'implementazione dello scheduler.

Avviato il programma la prima funzione che viene eseguita è il **main()**, nel quale si esegue le chiamate a tutte le funzioni, la prima funzione è quella di **stampa_menu()** che manda in visualizzazione una lista di operazioni da poter eseguire, qui di seguito elencate:

- inserimento di una task;
- esecuzione task in coda;
- esecuzione task con ID;
- eliminazione task con ID;
- modifica della priorità task con ID;
- cambiamento politico di scheduling;
- uscita dal programma.

A fine di ogni operazione il main chiama la funzione **print_task_list()** che stampa la lista di tutte le task all'interno dello scheduler.

Nel caso in cui non siano presenti task all'interno della lista gli unici comandi che sarà possibile eseguire saranno, l'inserimento di un task e l'uscita dal programma.

Presupponendo che l'utente voglia inserire un nuovo task, in corrispondenza del comando 1, verrà eseguito il metodo **insert_new_task()**.

Con questa funzione il programma, utilizzando **malloc**, alloca dinamicamente lo spazio in memoria per memorizzare il nuovo task. A questo punto si richiede all'utente di inserire tutti i dati necessari a creare il nuovo task. Ricevuti i dati, il programma, associa al task un ID univoco. Il programma torna nella schermata in cui l'utente può scegliere l'operazione che desidera eseguire.

A questo punto è permesso all'utente l'esecuzione delle altre operazioni.

Di seguito analizziamo le operazioni di:

- esecuzione task in coda.

Questa operazione avviene con il metodo **execute_tail()**.

In questa funzione il programma ricava il numero delle esecuzioni rimaste.

Se quest'ultima è pari a 1, viene direttamente eseguita la funzione, **delete_by_ID()**, metodo che rimuove il task dalla lista. Altrimenti la funzione decrementerà unitariamente il numero di esecuzioni del task tornando successivamente al menù principale.

- esecuzione task con ID.

Questa operazione è molto simile alla precedente e avviene tramite la funzione **execute_by_ID()**.

Inizialmente il programma crea un puntatore alla struttura che punterà alla task dell'ID scelto da tastiera, per prendere la task corrispondente a questo ID tale funzione utilizza il metodo **find_task_by_ID()**.

Questa scorre tutti gli ID contenuti nella lista dei task presenti nello scheduler fino a che non identifica l'ID del task scelto. Nel caso in cui l'ID non fosse presente nella lista, restituisce un messaggio d'errore e annulla tutta l'operazione.

- Eliminazione task con ID.

Tale operazione avviene con la funzione **delete_by_ID()**.

Il metodo è eseguito o in corrispondenza di una richiesta di eliminazione del task da parte dell'utente o quando si esegue l'ultima esecuzione di un task.

Nel primo caso il task viene cercato con la funzione **find_task_by_ID()**, nel secondo caso l'ID viene passato da uno dei seguenti metodi: **execute_by_ID()**, **execute_tail()**.

L'eliminazione di una task avviene con l'aggiornamento del puntatore del task che lo precede facendolo puntare al task che lo segue.

- Modifica della priorità task con ID.

Il metodo che si occupa di modificare la priorità è denominato con **change_priority_by_ID()**.

anche in questo caso il task viene individuato con la funzione **find_task_by_ID()**. Una volta trovata il task si procede con la modifica del campo priority, inserendo una nuova priorità da tastiera.

- Cambiamento politico di scheduling.

L'operazione è eseguita dalla funzione **switch_scheduling()**, che tramite un flag identifica se l'ordinamento deve essere effettuato per priorità in ordine decrescente (di default) o per numero di esecuzioni rimaste in ordine crescente.

Di seguito illustriamo il comportamento del programma in corrispondenza di un cambio della politica di scheduling.

Nell'esempio seguente abbiamo associato ai task gli stessi valori di priorità ed esecuzioni rimanenti dell'esempio riportato nella consegna.

Da notare che tale ordinamento differisce da quello in consegna, poiché a parità di priorità o esecuzioni rimanenti, l'ordinamento è basato sul valore ID.

```
dave@dave-VirtualBox: ~/Scrivania

ORDINAMENTO PER PRIORITA'

| ID | NOME | PRIORITA' | ESECUZIONI |
| 0 | task1 | 7 | 3 |
| 1 | task2 | 5 | 11 |
| 3 | task4 | 2 | 3 |
| 2 | task3 | 2 | 2 |

Scegli un'operazione:
1 = Inserimento nuovo task
2 = Esecuzione task in coda
3 = Esecuzione task con ID
4 = Eliminazione task con ID
5 = Modifica della priorita' task con ID
6 = Cambiamento politica di scheduling
7 = Uscita dal programma.

Inserire un intero corrispondente alle precedenti:
6
```

```
dave@dave-VirtualBox: ~/Scrivania

ORDINAMENTO PER ESECUZIONI'

| ID | NOME | PRIORITA' | ESECUZIONI |
| 2 | task3 | 2 | 2 |
| 3 | task4 | 2 | 3 |
| 0 | task1 | 7 | 3 |
| 1 | task2 | 5 | 11 |

Scegli un'operazione:
1 = Inserimento nuovo task
2 = Esecuzione task in coda
3 = Esecuzione task con ID
4 = Eliminazione task con ID
5 = Modifica della priorita' task con ID
6 = Cambiamento politica di scheduling
7 = Uscita dal programma.

Inserire un intero corrispondente alle precedenti:

```

- Uscita dal programma.

In questo caso viene semplicemente effettuata una `exit(0)`.

Ogni funzione richiama, come ultima operazione, il metodo di ordinamento, in modo che per ogni task inserito o modificato, lo scheduler tiene la lista dei task sempre aggiornata e ordinata.

A questo punto parliamo del metodo di **ordinamento()**.

Una volta determinato se l'ordinamento deve essere effettuato per priorità o per esecuzioni rimaste, viene chiamata una delle seguenti:

- **massima_priority()**.

Individua il task con la priorità massima, e se questo non si trova in testa alla lista, invoca la funzione **switch_two()**.

Continua in questo modo fino a che l'intera lista non è ordinata.

- **minimo_execution()**.

Individua illa task con le esecuzioni rimaste minime, e se questa non si trova in testa alla lista, invoca la funzione **switch_two()**.

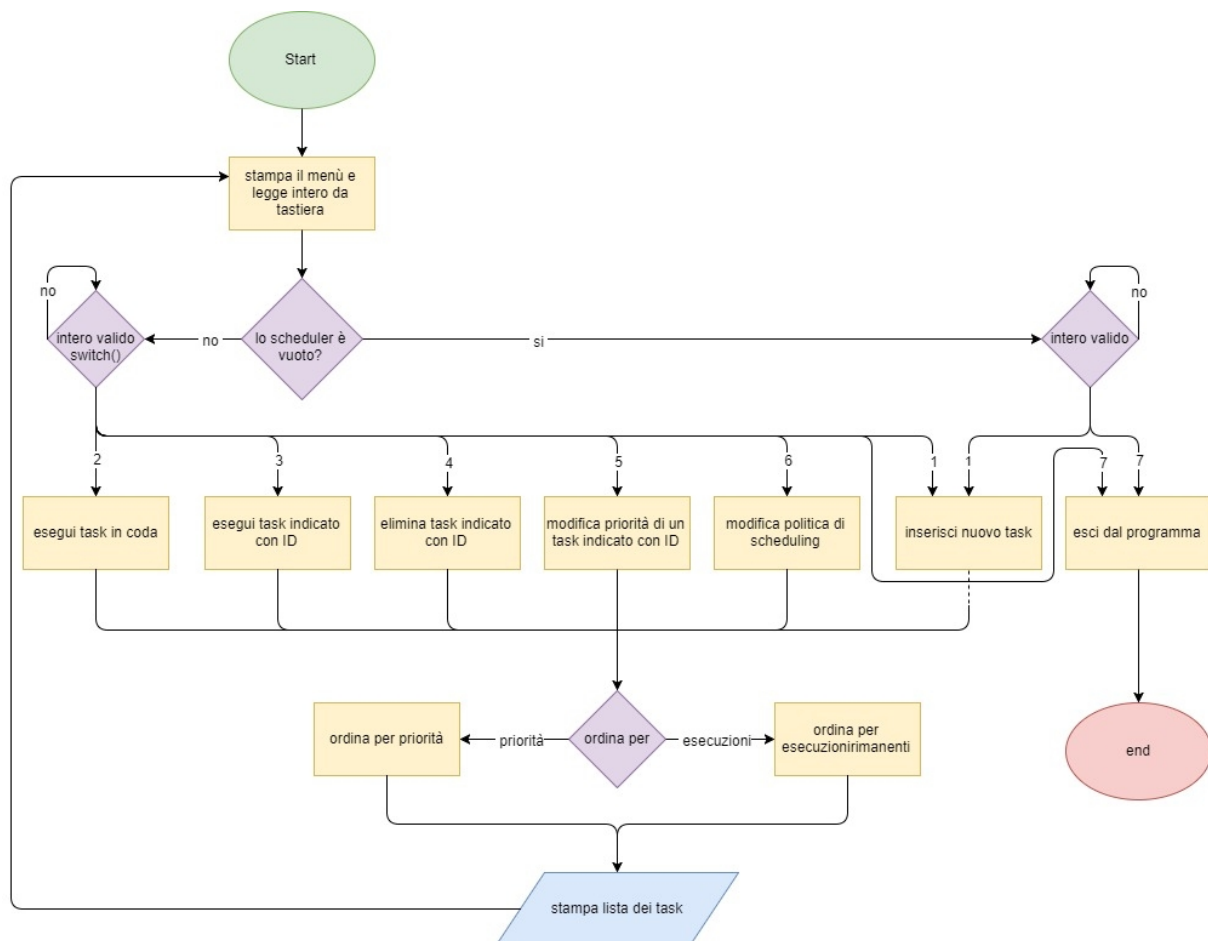
Continua in questo modo fino a che l'intera lista non è ordinata.

Concludiamo spiegando come agisce il metodo **switch_two()**.

Quest'ultimo esegue l'effettivo spostamento di due task nello scheduler.

L' operazione avviene aggiornando i puntatori di ogni struttura facendoli puntare ai task nell'ordine corretto.

DIAGRAMMA DI FLUSSO



DESCRIZIONE DELL'IMPLEMENTAZIONE

```
1.  #include "struct.h" /*includiamo la libreria della struttura delle task*/
2.  #include<stdio.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.
6.  /*VARIABILI GLOBALI*/
7.      typedef struct T_structure *LIST; /*crea un puntatore a lista utilizzato per lo switch dei
task*/
8.      struct T_structure *head = NULL; /*puntatore che conterrà la testa dello scheduler*/
9.      struct T_structure *newhead = NULL; /*puntatore d'appoggio che conterrà la nuova
testa*/
10.     struct T_structure *tail = NULL; /*puntatore che conterrà la coda dello scheduler*/
11.     int sch = 0; /*variabile con la quale specifico se l'ordinamento deve essere per esecuzioni
rimaste o per priorità(di default)*/
12.     int n= 0; /*variabile con la quale il programma capisce che lo scheduler è vuoto*/
13.
14.  /*FUNZIONE DI STAMPA MENÙ*/
15.  /*visualizza quali sono le operazioni che sono consentite */
16.  int stampa_menu(){
17.
18.      printf("1 = Inserimento nuovo task\n2 = Esecuzione task in coda\n3 = Esecuzione task
con ID\n4 = Eliminazione task con ID\n5 = Modifica della priorita' task con ID\n6 = Cambiamento
politica di scheduling\n7 = Uscita dal programma.\n\n");
19.      return 0;
20.  }
21.
22.  /*FUNZIONE CHE PRESI 3 VALORE ESEGUE LO SWITCH PER ORDINARLI IN BASE
AL TIPO DI ORDINAMENTO*/
23.
24.  void switch_two(LIST a, LIST b, LIST c){
25.
26.      printf("entro switch\n");
27.      printf("head %p\n", head);
28.      printf("a %p e b %p\n", a, b); /* a = point, cioè la testa da dove inizia l'ordinamento , b =
prec, cioè l'elemento precedente al minore , c = cont5, cioè puntatore alla struttura dal quale si inizia
l'ordinamento*/
29.
30.      if(a == head && b == a){ /*verifico se l'elemento in testa è il più piccolo, in caso
negativo esegue lo switch tra i primi due task */
31.
32.          /*esegue l'effettivo spostamento dei task per metterli in ordine, aggiornando i
puntatori dei task */
33.          LIST x = a;
34.          LIST y = a -> next;
35.          LIST z = a -> next -> next;
36.
37.          a = y;
38.          a -> next = x;
39.          a -> next -> next = z;
```

```

40.
41.
42.     head = a;
43.     newhead = head;
44.     tail = a -> next;
45.
46.
47.     }else if(a == head && b != a){/*verifico se l'elemento in testa è il più piccolo, in caso
negativo esegue lo switch tra la testa e l'elemento individuato più piccolo */
48.
49.         /*esegue l'effettivo spostamento dei task per metterli in ordine, aggiornando i
puntatori dei task*/
50.         LIST x = b;
51.         LIST t = a;
52.
53.         LIST y = b -> next;
54.         LIST z = b -> next -> next;
55.
56.         a = y;
57.         a -> next = t;
58.
59.         head = a;
60.         newhead = head;
61.         tail = x;
62.         tail -> next = z;
63.
64.     }else if(a != head && a==b){/*quando la lista contiene già alcuni elementi in ordine,
l'ordinamento continuerà dall'elemento successivo (all'ultimo, che è già nella
posizione finale), quindi l'elemento precedente corrisponde all'elemento puntato*/
65.
66.         /*esegue l'effettivo spostamento dei task per metterli in ordine, aggiornando i
puntatori dei task */
67.         LIST x = a;
68.         LIST y = a -> next;
69.         LIST z = a -> next -> next;
70.
71.
72.         a = y;
73.         a -> next = x;
74.         a -> next -> next = z;
75.         c -> next = a;
76.         tail = a -> next;
77.
78.     }else if(a != head && a!= b){/*quando la testa contiene già l'elemento più
piccolo/maggiore, l'ordinamento inizia da un task successivo a quello iniziale*/
79.
80.         /*esegue l'effettivo spostamento dei task per metterli in ordine, aggiornando i
puntatori dei task */
81.         LIST x = a;
82.         LIST y = b -> next;
83.         LIST z = b -> next -> next;
84.

```

```

85.
86.     a = y;
87.     a -> next = c -> next;
88.     b -> next = z;
89.     c -> next = a;
90.     tail = b;
91.
92.     }
93.
94.
95. }
96. /*FUNIZIONE CHE ORDINA I TASK IN BASE AL NUMERO DI ESECUZIONI RIMASTE
IN MODO CRESCENTE */
97.
98. int minimo_execution(LIST point, LIST precPoint){
99.     LIST cont1 = point; /* puntatore d'appoggio dal quale iniziare l'ordinamento e che
scorre la lista, =cont3*/
100.    LIST cont2 = point -> next; /* puntatore d'appoggio al successivo con il quale effettuare
il controllo*/
101.    LIST cont5 = precPoint; /*puntatore fisso all'inizio della struttura, =cont4*/
102.
103.    LIST prec = point; /* puntatore per elemento precedente, usate per non modificare la
struttura della lista*/
104.
105.    int min = point -> execution; /* assegnazione ad una variabile, l'esecuzione
dell'elemento puntato, il possibile elemento con priorità minima*/
106.    LIST succ = point -> next; /* puntatore per l'elemento successivo, usate per non
modificare la struttura della lista*/
107.
108.    int c=0; /* contatore spostamenti, che individua se ci deve effettuare uno spostamento tra
i task*/
109.    int minore = min; /*variabile d'appoggio, assegna il minimo numero di esecuzioni (burst)
del task*/
110.
111.    /*ciclo che controlla se c'e un elemento successivo a quello puntato attualmente*/
112.    while(cont2 != NULL){
113.        /*controllo se il numero delle esecuzioni del task successivo è minore al valore min*/
114.        if(cont2 -> execution <= min){
115.
116.            prec = cont1; /*assegno a prec il task puntato da cont1*/
117.
118.            min = cont2 -> execution; /*assegno a min il nuovo numero d'esecuzione
minore del task puntato dal puntatore della lista cont2*/
119.
120.            succ = cont2 -> next; /*assegno al succ il task successivo a quello puntato da
cont2*/
121.
122.            c=c+1; /*incremento il contatore degli spostamento poichè ho eseguito lo
spostamento*/
123.
124.
125.        }

```



```

126.         cont1 = cont2; /* Puntatore al possibile precedente del minimo*/
127.         cont2 = cont2 -> next; /* Puntatore al possibile futuro minimo*/
128.     }
129.
130.
131.     if(c!=0 && min != minore){/*se è stato riscontrato uno spostamento e il minimo
ritrovato è diverso da quello della testa, si chiama la procedura di switch tra la testa da dove iniziare
l'ordinamento (point), elemento precedente al minimo(prec), puntatore della struttura dalla quale si
inizia l'ordinamento(cont5)*/
132.         switch_two(point, prec, cont5);
133.         return 1;
134.     }else if(c!=0 && point -> ID < prec -> next -> ID){/*in caso ci sono due task con uno
stesso numero di esecuzioni rimaste, si effettua lo switch controllando l'id del task*/
135.         switch_two(point, prec, cont5);
136.         return 1;
137.     }
138.
139.     return 0;
140. }
141.
142.
143. /*FUNZIONE CHE ORDINA IN BASE ALLA PRIORITÀ MASSIMA*/
144.
145. int massima_priority(LIST point, LIST precPoint){
146.
147.     LIST cont1 = point; /* puntatore d'appoggio dal quale iniziare l'ordinamento e che scorre
la lista, =cont3*/
148.     LIST cont2 = point -> next; /* puntatore d'appoggio al successivo con il quale effettuare
il controllo*/
149.     LIST cont5 = precPoint; /*puntatore fisso all'inizio della struttura, =cont4*/
150.
151.     LIST prec = point; /* puntatore per elemento precedente, usate per non modificare la
struttura della lista*/
152.
153.     int max = point -> priority; /* assegnazione ad una variabile, la priorità dell'elemento
puntato, il possibile elemento con priorità massima*/
154.     LIST succ = point -> next; /* puntatore per l'elemento successivo, usate per non
modificare la struttura della lista*/
155.
156.     int c=0; /* contatore spostamenti, che individua se ci deve effettuare uno spostamento tra
i task*/
157.     int massimo = max; /*variabile d'appoggio, contiene la priorità del primo elemento*/
158.
159.
160.     while(cont2 != NULL){/* controlla che la lista abbia altri task*/
161.
162.         if(cont2 -> priority >= max){/*controlla se la priorità dell'elemento successivo è
maggiore di quello precedente, in caso affermativo effettua lo scambio dei valori
d'appoggio per il futuro switch, altrimenti lascia tutto invariato*/
163.
164.             prec = cont1;
165.

```

```

166.         max = cont2 -> priority;
167.
168.         succ = cont2 -> next;
169.
170.         c=c+1;/*notifico che si effettuerà uno switch*/
171.
172.     }
173.     cont1 = cont2; /* Puntatore al possibile precedente del massimo*/
174.     cont2 = cont2 -> next; /* Puntatore al possibile futuro massimo*/
175. }
176.
177.
178.     if(c!=0 && max != massimo){/*se è stato riscontrato uno spostamento e il massimo
ritrovato è diverso da quello della testa, si chiama la procedura di switch tra la testa da dove iniziare
l'ordinamento (point), elemento precedente al massimo(prec), puntatore della struttura dalla quale si
inizia l'ordinamento(cont5)*/
179.         switch_two(point, prec, cont5);
180.         return 1;
181.     }else if(c!=0 && point -> ID < prec -> next -> ID){/*in caso due priorità hanno lo stesso
valore, si effettua lo switch controllando l'id dei task*/
182.         switch_two(point, prec, cont5);
183.         return 1;
184.     }
185.
186.     return 0;
187. }
188.
189. //FUNZIONE CHE ORDINA LA LISTA DEI TASK
190.
191. void ordinamento(struct T_structure *head, int sch){
192.
193.     int control=0;/* flag che individua eventuali switch*/
194.     LIST cont3 = head; /* puntatore scorrevole della lista */
195.     LIST cont4 = head; /* utilizzato come puntatore alla testa della lista*/
196.
197.     if(sch==0){ /*ordinamento priorità = false*/
198.         printf("ORDINAMENTO PRIORITA'\n");
199.
200.
201.         while(cont3 != NULL){/*controlla se la lista dei task ha altri elementi*/
202.
203.             control = massima_priority(cont3, cont4); /* chiamata alla funzione massima
priorità, e passa i valori da confrontare*/
204.
205.
206.             if( control != 0 && newhead != NULL){ /*controlla se la testa ha subito
modifiche*/
207.                 cont4 = newhead; /* assegno la nuova head e faccio ripartire il ciclo
dalla testa*/
208.                 cont3 = newhead ;/* assegno la nuova head e faccio ripartire il ciclo
dalla testa*/
209.             }else{

```

```

210.          /* se non si è verificato lo switch, passa agli elementi successivi */
211.          cont4 = cont3;
212.          cont3 = cont3 -> next;
213.      }
214.
215.  }
216.
217.      tail = cont4; /* cont4 è l'ultimo elemento prima di null quindi è sicuramente la coda,
poichè cont4 punterà ad un elemento null*/
218.
219.  }else{ /*ordinamento esecuzioni = true*/
220.      printf("ORDINAMENTO ESECUZIONE\n");
221.
222.      while(cont3 != NULL){/*controlla se la lista dei task ha altri elementi*/
223.
224.
225.          control = minimo_execution(cont3, cont4); /* chiamata alla funzione minore
numero di esecuzioni rimaste e passa i valori da confrontare*/
226.
227.
228.          if(control != 0){/*controlla se ci sono state delle modifiche*/
229.              cont4 = newhead; /* assegno la nuova head e faccio ripartire il ciclo
dalla testa*/
230.              cont3 = newhead ;/* assegno la nuova head e faccio ripartire il ciclo
dalla testa*/
231.          }else{
232.              /* se non si è verificato lo switch, passa agli elementi successivi */
233.              cont4 = cont3;
234.              cont3 = cont3 -> next;
235.          }
236.      }
237.
238.      tail = cont4; /* cont4 è l'ultimo elemento prima di null quindi è sicuramente la coda,
poichè cont4 punterà ad un elemento null*/
239.  }
240. }
241.
242.
243. /* FUNZIONE CHE CONSENTE DI IMMETTERE I DATI DEL NUOVO TASK DA
ESEGUIRE */
244.
245. LIST insert_new_task(int x){
246.
247.     struct T_structure *new_task = malloc(sizeof(T_structure)); /*puntatore a struttura che
punta ad una allocazione di memoria*/
248.
249.     /*riempimento della memoria allocata per il nuovo task con i valore presi da tastiera
e l'ID*/
250.     new_task -> ID = x;
251.     printf("inserire nome task (massimo 8 caratteri):\n");
252.     scanf("%8s", new_task -> name);
253.

```

```

254.     while(getchar() != '\n');
255.     printf("%s\n", new_task -> name);
256.     printf("inserire priorit  task (compreso tra 0 e 9):\n");
257.     scanf("%1d", &new_task -> priority);
258.
259.     while(getchar() != '\n');
260.     printf("%s, %d\n", new_task -> name, new_task -> priority);
261.
262.     do{
263.         printf("inserire numero di esecuzioni (compreso tra 1 e 99):\n");
264.         scanf("%2d", &new_task -> execution);
265.
266.         while(getchar() != '\n');
267.
268.     }while(new_task -> execution < 1);
269.
270.     printf("%s, %d, %d\n", new_task -> name, new_task -> priority, new_task ->
execution);
271.     new_task -> next = NULL; /* assegna al nuovo task un puntatore che punta a nulla,
poich  il task verr  inserita in fonda alla lista dei task*/
272.
273.     if (head == NULL){/*controlla se la testa sia vuota, serve a identificare se   il primo
task inserito, in caso affermativo assegna sia alla testa che alla coda il
nuovo task creato*/
274.         head = new_task;
275.         tail = head;
276.     }else{
277.         tail -> next = new_task;/*crea il puntatore al task in coda con il nuovo task
inserito nello scheduler*/
278.         tail = new_task;/*inserisce il task appena creata in coda nello scheduler*/
279.     }
280.
281.     ordinamento(head, sch); /*chiama la funzione d'ordinamento mandandogli come
parametro, la testa e il tipo d'ordinamento (per priorit  o rimanenze )*/
282. }
283.
284. /*FUNZIONE CHE INDIVIDUA L'ID DEL TASK RICERCATO DALLA LISTA DEI
TASK DELLO SCHEDULER */
285.
286. LIST find_task_by_ID(int inner_ID){
287.
288.     struct T_structure *find_struct = head;/*punatore alla struttura che punta alla testa dello
scheduler*/
289.
290.     /*cicla tutta la lista dei task contenuti nello scheduler fino a trovare l'ID ricercato, in
caso di risposta negativa restituisce un messaggio d'errore*/
291.     do{
292.
293.         if(find_struct -> ID == inner_ID){/*verifica che l'ID del task inserito da tastiera
corrisponde a quello in testa alla lisat del task*/
294.             printf("L'ID corrisponde al task %s\n\n", find_struct -> name);
295.             return find_struct;/*restituisco la struttura del task con l'ID ricercato*/

```

```

296.         }else if(find_struct -> ID != inner_ID && find_struct -> next == NULL){/*se l'ID
non è presente nella lista dei task, risponde con un messaggio di errore*/
297.             printf("Non esiste un task con ID %d\n", inner_ID);
298.
299.             return NULL;
300.
301.         }else{
302.             find_struct = find_struct -> next;/* passaggio al successivo elemento*/
303.         }
304.
305.
306.     }while(find_struct != NULL );
307.
308.
309. }
310.
311. /*FUNZIONE DI ELIMINAZIONE TASK TRAMITE ID DA TASTIERA*/
312.
313. void delete_by_ID(int x){
314.
315.     struct T_structure *tmp_struct = NULL;/*inizializza un puntatore a struttura che verrà
utilizzato per il task con l'ID selezionato*/
316.     struct T_structure *struct_pointer = head;/*viene assegnato al puntatore il task in testa
dello scheduler*/
317.     printf("Elimino il task con ID %d.\n", x);
318.     tmp_struct = find_task_by_ID(x);/*viene assegnato al puntatore l'ID ottenuto dalla
funzione find_task_by_ID*/
319.
320.     if(tmp_struct != NULL){/*controlla se è stato trovato l'ID ricercato*/
321.
322.
323.         if(tmp_struct == head){/*controlla che l'ID da eliminare corrisponde alla testa*/
324.             if(tmp_struct == tail){/*controlla se l'ID corrisponde anche alla coda e quindi
imposta sia la testa che la coda a null in quanto la lista dei task
diventerà vuota*/
325.                 head = NULL;
326.                 tail = NULL;
327.             }else if(tmp_struct -> next == tail){/*controlla se l'elemento successivo
all'elemento da eliminare (testa),corrisponde all'elemento in coda, in caso affermativo la coda
corrisponde alla testa( è rimasto un solo task), altrimenti imposta come testa l'elemento successivo
all'elemento eliminato*/
328.                 head = tail;
329.             }else{
330.                 head = head -> next;
331.             }
332.         }else {/*caso in cui l'elemento da eliminare non fosse quello in testa*/
333.             while(struct_pointer -> next != tmp_struct){/*scorrimento della lista fino a
che l'elemento è diverso dall'elemento da eliminare*/
334.                 struct_pointer = struct_pointer -> next;/*passa all'elemento
successivo*/
335.
336.             }if(tmp_struct != tail){/*verifica che l'elemento da eliminare sia diverso

```

```

dall'elemento in coda, in caso negativo l'elemento precedente punterà a nulla e
diventerà la nuova coda dei task*/
337.         struct_pointer -> next = tmp_struct -> next;
338.     }else{
339.         struct_pointer -> next = NULL;
340.         tail = struct_pointer;
341.     }
342.
343.
344.     }
345. }
346.     ordinamento(head, sch); /*chiama la funzione d'ordinamento mandandogli come
parametro, la testa e il tipo d'ordinamento (per priorità o rimanenze )*/
347. }
348.
349. /*FUNZIONE CHE ESEGUE IL TASK IN CODA*/
350.
351. void execute_tail(){
352.
353.
354.
355.     if(tail -> execution == 1){/*controlla se il numero delle esecuzioni del task in coda è
uguale a 1, ciò porta all'eliminazione di quest'ultima*/
356.         int x = (*tail).ID;/*assegnazione ad una variabile l'ID del task in coda */
357.         printf("Al task in coda rimane una sola esecuzione, dopo la quale sarà eliminato.\n");
358.         delete_by_ID(x);/*passa alla funzione di eliminazione l'ID del task in coda */
359.     }else{
360.         tail -> execution -= 1;/*diminuzione del numero di esecuzioni rimaste dal task in
coda*/
361.
362.     }
363.     ordinamento(head, sch); /*chiama la funzione d'ordinamento mandandogli come
parametro, la testa e il tipo d'ordinamento (per priorità o rimanenze )*/
364. }
365.
366. /* FUNZIONE CHE ESEGUE IL TASK TRAMITE LA SCELTA DELL'ID*/
367.
368. void execute_by_ID(int x){
369.
370.     struct T_structure *tmp_struct = NULL;/*inizializza un puntatore a struttura che verrà
utilizzato per il task con l'ID selezionato*/
371.     tmp_struct = find_task_by_ID(x);/*viene assegnato al puntatore l'ID ottenuto dalla
funzione find_task_by_ID*/
372.
373.     if(tmp_struct != NULL){
374.         printf("Eseguo task con ID %d.\n", x);
375.         if(tmp_struct -> execution == 1){/*controllo che verifica se è rimasta una sola
esecuzione*/
376.             printf("Al task in coda rimane una sola esecuzione, dopo la quale sarà
eliminato.\n");
377.             delete_by_ID(x);/*passa alla funzione di eliminazione l'ID del task scelto*/
378.         }else{

```

```

379.             tmp_struct -> execution -= 1; /*diminuzione del numero di esecuzione rimaste
dal task scelto */
380.         }
381.     }
382.     ordinamento(head, sch); /*chiama la funzione d'ordinamento mandandogli come
parametro, la testa e il tipo d'ordinamento (per priorità o rimanenze) */
383. }
384.
385.
386. //FUNZIONE CHE CAMBIA LA PRIORITÀ DI UN TASK TRAMITE LA SCELTA DI UN
ID
387.
388. void change_priority_by_ID(int x){
389.
390.     struct T_structure *tmp_struct = NULL; /*inizializzo un puntatore a struttura che verrà
utilizzato per il task con l'ID selezionato*/
391.     tmp_struct = find_task_by_ID(x); /*viene assegnato al puntatore l'ID ottenuto dalla
funzione find_task_by_ID*/
392.
393.     if(tmp_struct != NULL){
394.         printf("Inserire nuova priorit :\n");
395.         scanf("%1d", &tmp_struct -> priority); /*riceve da tastiera la nuova priorit  e la
sostituisce alla vecchia*/
396.
397.         while(getchar() != '\n');
398.
399.     }
400.     ordinamento(head, sch); /*chiama la funzione d'ordinamento mandandogli come
parametro, la testa e il tipo d'ordinamento (per priorit  o rimanenze) */
401. }
402. /* FUNIZIONE PER CAMBIO DI POLITICA DI SCHEDULER*/
403. /* Tramite l'assegnazione di un valore compreso tra 0 e 1 questa funzione mi permette di
passare da un ordinamento per priorit  (0) a un ordinamento per rimanenze (1)*/
404.
405. void switch_scheduling(){
406.
407.     printf("Cambio politica di scheduling.\n");
408.     if(sch==1){
409.         sch=0;
410.     }else{
411.         sch=1;
412.     }
413.     ordinamento(head, sch); /*chiama la funzione d'ordinamento mandandogli come
parametro, la testa e il tipo d'ordinamento (per priorit  o rimanenze) */
414. }
415.
416. /*FUNZIONE CHE STAMPA LA LISTA DEI TASK ALL'INTERNO DELLO
SCHEDULER*/
417.
418. void print_tasks_list(){
419.     struct T_structure *pointer = head; /*crea un puntatore a struttura che punta alla testa */
420.

```

```

printf("\n");
421. printf("| ID | NOME | PRIORITA' | ESECUZIONI\n");
422.
423. while(pointer != NULL){/**/
424.
printf("\n");
425. printf("| %d | %s | %d | %d |\n",
pointer -> ID, pointer -> name, pointer -> priority, pointer -> execution);
426. pointer = pointer -> next;/**sposta il puntatore al task successivo*/
427. }
428.
429. }
430.
431. /*FUNZIONE MAIN */
432.
433. int main(void){\
434.
435. int autoincremental_task_ID = 0;/**inizializzazione della variabile per l'ID di una task*/
436. struct T_structure *new_task = NULL;/**puntatore a struttura per la memorizzazione di
una nuova task*/
437.
438. while(1){
439. int comando=0;/**variabile in cui salva il comando da eseguire*/
440. int command = 0;/**variabile in cui salva il comando da eseguire,usato per non
permettere altre operazioni esclusa quella di inserimento e uscita dal programma nel
caso in cui la lista dei task sia vuota*/
441. int ID_search = -1;/**variabile che conterrà il l'id incerito da tastiera per cercare il
task*/
442.
443. if(head != NULL){/**verifica che la testa sia diversa da 'NULL' condizione in cui lo
scheduler è vuoto*/
444. n=1;
445. print_tasks_list();
446.
447. }else{
448. n= 0;
449. }
450.
451. if( n == 0){/** controlla se lo scheduler sia vuoto*/
452.
453.
454. do{
455. printf("\n Scegli un'operazione: \n");
456. stampa_menu();
457. printf("Inserire un intero corrispondente alle precedenti:\n");
458. scanf("%1d", &command);/**riceve dal tastiera l'esecuzione che dovrà
esser eseguita*/
459. getchar();/**attesa che l'utente inserisca un carattere da tastiera*/
460. if(command == 1){
461. printf("sono nel new task\n");

```



```

462.             n=1;
463.             new_task = insert_new_task(autoincremental_task_ID);
autoincremental_task_ID += 1; /* new_task non serve più, quindi insert_task può
    tornare void*/
464.
465.             }else if (command == 7){
466.                 printf("Esco dal programma.\n");
467.                 exit(1);
468.             }else if (command>1 && command<7) {
469.
470.                 printf("operazione non valida, non ci sono task in
memoria.\n\n");
471.             }else{
472.                 printf("Inserimento errato. Riprova.\n\n");
473.             }
474.
475.         }while ( n!=1);
476.
477.
478.     }else{
479.         printf("\n Scegli un'operazione:\n");
480.         stampa_menu();
481.         printf("Inserire un intero corrispondente alle precedenti:\n");
482.         scanf("%1d", &comando);/*riceve dal tastiera l'esecuzione che dovrà esser
eseguita*/
483.         getchar();/*attesa che l'utente inserisca un carattere da tastiera*/
484.
485.
486.         switch(comando){
487.
488.             case 1: new_task = insert_new_task(autoincremental_task_ID);
autoincremental_task_ID += 1; /* new_task non serve più, quindi insert_task può tornare void*/
489.                 break;
490.
491.             case 2: execute_tail();
492.                 break;
493.
494.             case 3: printf("Inserire l'ID del task da eseguire.\n");
495.                 scanf("%d", &ID_search);
496.                 execute_by_ID(ID_search);
497.                 break;
498.
499.             case 4: printf("Inserire l'ID del task da eliminare.\n");
500.                 scanf("%d", &ID_search);
501.                 delete_by_ID(ID_search);
502.                 break;
503.
504.             case 5: printf("Inserire l'ID del task del quale si desidera cambiare la
priorita'.\n");
505.                 scanf("%d", &ID_search);
506.                 change_priority_by_ID(ID_search);

```

```

507.             break;
508.
509.             case 6: switch_scheduling();
510.                 break;
511.
512.             case 7: printf("Esco dal programma.\n");
513.                 exit(1);
514.                 break;
515.             default: printf("Inserimento errato. Riprova.\n");
516.         }
517.     }
518. }
519.
520. }
521.

```

struct.h

```

1.  #ifndef SRC_ESERCIZIO1_H_
2.  #define SRC_ESERCIZIO1_H_
3.  // CREAZIONE DELLA STRUTTURA DI OGNI SINGOLA TASCK
4.  //in questa header file definisco la struttura di una task definendo i campi che essa deve avere
5.      extern struct T_structure{
6.          int ID;
7.          char name[9];
8.          int priority;
9.          int execution;
10.         struct T_structure *next;
11.     }T_structure;
12.
13. #endif

```

ESERCIZIO 2

EVIDENZA DEL CORRETTO FUNZIONAMENTO DEL PROGRAMMA RISPETTO ALLE SPECIFICHE FORNITE

L'obiettivo del secondo esercizio è quello di creare un esecutore di comandi UNIX che scriva, sequenzialmente o parallelamente, l'output dell'esecuzione su di un file.

Tutte le funzionalità del programma sono incluse all'interno del programma stesso **cmdexec.c** .

- La funzione **main()** si occupa di avviare il programma su terminale e di mostrare a video il menù con le due scelte che l'utente può fare. A sua volta l'utente ha il compito di inserire un valore che deve essere uguale o a "s" (i comandi saranno eseguiti *sequenzialmente*) o a "p", (i comandi saranno eseguiti *parallelamente*); in caso contrario richiama se stessa. L'inserimento del carattere viene gestito tramite una **scanf()** e successivamente una **getchar()**, in quanto la **scanf()** lascia nel buffer un carattere di fine riga '\n'. Quando chiamo **getchar()**, in pratica viene tolto "\n" dal buffer facendo sì che la **fgets()** funzioni senza problemi. Senza il **getchar()**, la **fgets()** legge come primo carattere un '\n' e il programma si interrompe prima che l'utente possa eseguire qualunque azione.
- La funzione **change_sdoutput()** si occupa di creare il file output "out.i" (con i =1, 2, ...) relativo al i-esimo comando inserito dall'utente, all'interno della cartella del programma. Se il file è già presente lo rimuove per poi ricrearlo per il nuovo comando. Creato il file gli si assegna il file_descriptor di output, per poi restituirlo.
- La funzione **execution(char *v[], int fd)** tramite la chiamata di funzione **fork()** trasforma un singolo processo in due processi identici, riconoscibili come processo padre e processo figlio. Dopo di che, si controlla se l'esecuzione deve avvenire in modo sequenziale o parallelo. Se deve avvenire in modo sequenziale si effettua una **wait()**, affinché il processo padre attenda la terminazione del processo figlio prima di passare al successivo processo figlio, invece se i comandi devono essere svolti in parallelo , anche se un processo figlio è lungo , se arriva nel frattempo un secondo comando, esso viene eseguito in parallelo con un altro processo figlio, in quanto non si utilizza nessuna **wait()** come controllo sulla terminazione o meno del processo figlio. Dopo di che il comando viene eseguito con la chiamata di funzione **execvp()** , passandoglielo tramite il vettore " v ". Tramite l'intero restituito dalla funzione **execvp()** si controlla se il comando eseguito è andato a buon fine e in caso contrario viene stampato un intero definito dalla variabile " **errno** " la quale indica il tipo di errore. In più, tramite la funzione **strerror()**, che ha come parametri il valore di **errno**, viene stampata una stringa con il messaggio di errore relativo ad **errno**. Infine, per evitare processi figli orfani, (i processi figli zombi non vengono considerati una volta che il padre termina), quest'ultimi vengono uccisi tramite la funzione **kill()** che come parametri ha il pid del processo figlio orfano e il segnale **SIGKILL** (Segnale che indica la terminazione immediata del processo. Questo segnale non può essere ignorato ed il processo che lo riceve non può eseguire delle operazioni di chiusura morbida).

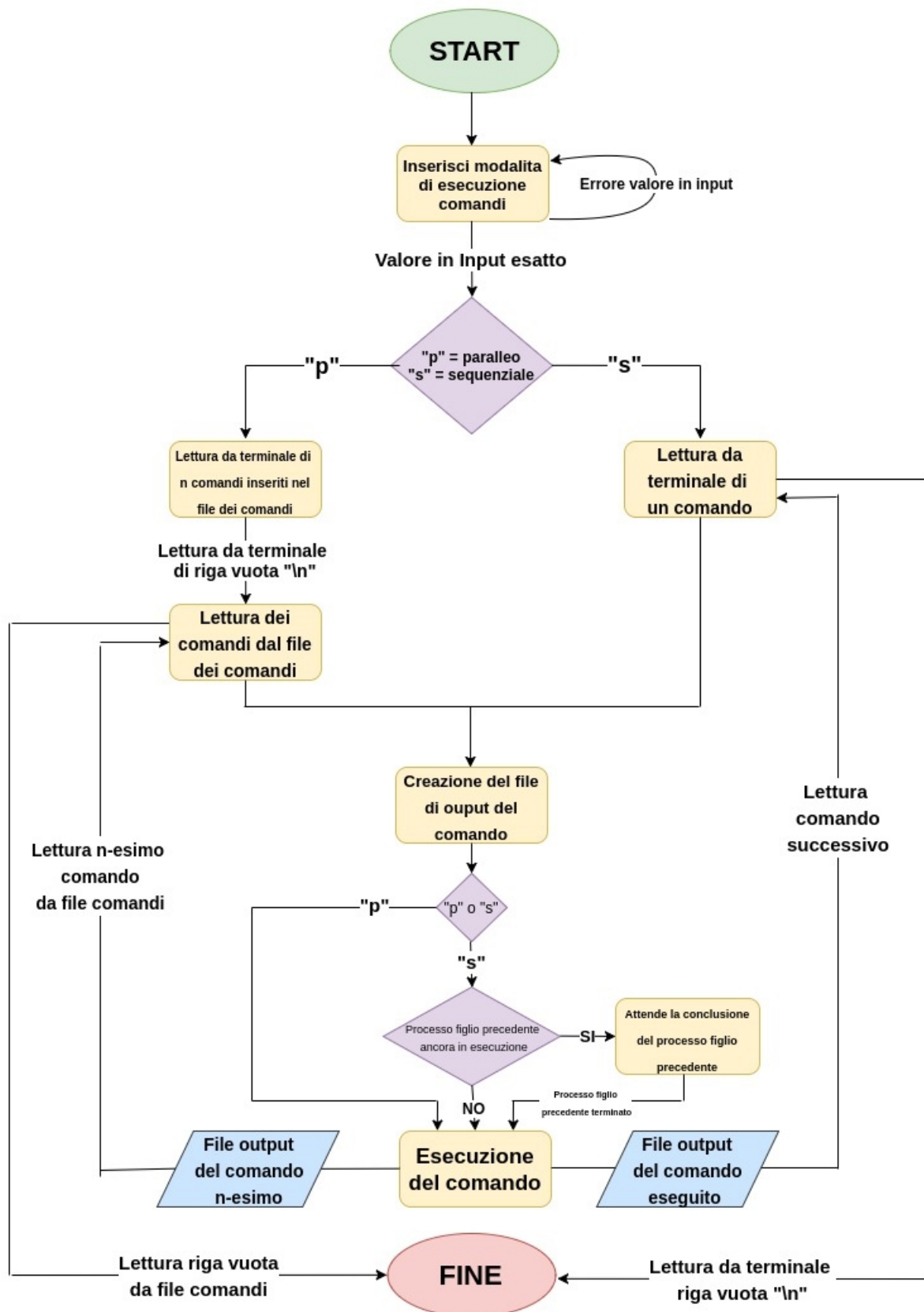
- La funzione **sequenziale()** tramite un ciclo while effettua la chiamata della funzione **comand_seq()** eseguendo sequenzialmente le operazioni inserite dall'utente, finché quest'ultimo non inserisce una riga vuota. In tal caso avviene l'uscita dal programma. Vedremo che qui non si usa nessun file intermedio, ciò rende più l'idea di sequenzialità, oltre a utilizzare nella **execution()** la funzione **wait()** che ci permette lo svolgimento sequenziale dei comandi
- La funzione **parallelo()** crea, se non è già esistente, il file "**file_comand.txt**". Se il file è già presente lo rimuove per poi crearlo nuovamente. Dopo di che, tramite un ciclo while esegue **fgets()**, permettendo all'utente di immettere un riga di comando, la quale sarà inserita in un vettore. Se la lunghezza di tale vettore è uguale a "0", vuol dire che è stata inserita una riga vuota, in caso contrario la stringa verrà scritta nel file dei comandi specificato inizialmente. Ad ogni scrittura sul file, segue l'aggiunta di un "**new line**". Quando l'utente inserisce una riga vuota, il file dei comandi viene chiuso ed è chiamata la funzione **comand_par()** la quale esegue parallelamente i comandi, leggendoli dal file. Nel parallelo usiamo un file d'appoggio affinché ciò renda l'idea di un potenziale parallelismo sulla creazione del file, oltre al fatto che come vedremo nella **execution()**, per il parallelo per come lo abbiamo implementato, non effettua nessuna attesa della terminazione di un processo figlio prima di crearne un altro da parte del processo padre.
- La funzione **comand_seq()** sequenzialmente legge da tastiera, tramite una **fgets()**, una riga di comando inserendola nel suo vettore. Una volta nel vettore la riga comando viene modificata tramite la chiamata di funzione **strtok()** (spiegata in seguito), per rendere tale riga un stringa di comando eseguibile. Dopo di che, il vettore, contenente il comando finale, viene passato come parametro insieme al file di output, alla funzione **execution(char *v[], int fd)**, dove verrà eseguito. Finché non è inserito il carattere "**new_line**", sarà restituito valore "1", altrimenti "0".
- La funzione **comand_par()** effettua l'apertura del "**file_comand.txt**", e tramite un ciclo while ne avviene la lettura di ogni riga, la quale viene modificata tramite la chiamata di funzione **strtok()** (spiegata in seguito) per rendere tale riga un stringa di comando eseguibile. Dopo di che il vettore, contenente il comando finale, viene passato come parametro insieme al file di output alla funzione **execution(char *v[], int fd)**, dove verrà eseguito. Il ciclo viene interrotto quando dal file dei comandi viene letta una riga vuota, in tal caso viene chiuso il file dei comandi e tornando alla funzione **parallelo()** si effettua l'uscita dal programma.

Come funzioni particolari abbiamo usato **strtok(char *s, char *sep)** della libreria "**string.h**" che serve per spezzare una stringa in token (parola), in base a dei caratteri di separazione e restituisce, ogni volta, un puntatore all'ultimo token trovato nella stringa. Un puntatore nullo viene restituito se non ci sono token rimasti da recuperare.

Nel nostro caso l'abbiamo usata per analizzare gli input inseriti dall'utente, in modo da poter essere convertiti in dei comandi da immettere in un vettore per essere eseguiti dalla funzione **execution()**.

Infine, ogni volta che viene eseguita una **fgets()** si pone, "**acr[strlen(acr)-1] = '\0'**", in quanto **fgets()** inserisce al penultimo valore "\n", ma l'operazione **strtok()**, necessita che tale valore sia "\0".

DIAGRAMMA DI FLUSSO



DESCRIZIONE DELL'IMPLEMENTAZIONE

```
1. #include <stdio.h> /* Libreria che contiene operazioni di input/output */
2. #include <fcntl.h> /* Libreria che contiene opzioni di controllo dei file */
3. #include <string.h> /* Libreria che contiene operazione di manipolazioni stringhe e
   memoria */
4. #include <stdlib.h> /* Libreria che dichiara funzioni e costanti di utilità generale */
5. #include <unistd.h> /* Libreria che consente l'accesso alle API dello standard POSIX */
6. #include <errno.h> /* Libreria che contiene definizioni di macro per la gestione delle
   situazioni di errore */
7. #include <signal.h> /* Libreria che contiene macro per gestire segnali diversi riportati
   durante l'esecuzione di un programma */
8. #include <sys/wait.h> /* Libreria che contiene opzioni di controllo della wait() */
9. #define BUFFER 1024 /* Grandezza del buffer nel quale inserire l'input da tastiera */
10.
11. int num_file = 1; /* Inizializzo il numero del file di output */
12. char newline[1] = "\n"; /* Definisco un vettore contenente carattere di nuova linea */
13.
14. /* execution() effettua l'esecuzione del comando inserito dall'utente sottoforma di input,
15. controllando se devono essere svolti sequenzialmente o no, tramite una wait(), se ci sono
   eventuali errori,
16. e infine chiude il file di output specifico */
17.
18. void execution(char *ar[], int f, char control){
19.
20.     printf("\n\nSoluzione :\n\n");
21.
22.     if(fork() != 0){ /* Viene creato un processo figlio per ogni comando inserito*/
23.
24.         if(control == 's'){ /* Se i comandi devono essere eseguiti sequenzialmente
           utilizzo una wait() affinché il processo padre attenda la terminazione del processo
           figlio prima di passare al successivo processo figlio, invece se i
           comandi devono essere svolti in parallelo , anche se un processo figlio è
           lungo , se arriva nel frattempo un secondo comando, esso viene
           eseguito in parallelo con un altro processo figlio, in quanto non si utilizza
           nessuna wait() come controllo sulla terminazione o meno del
           processo figlio */
25.
26.             wait(NULL);
27.
28.         }
29.
30.     }else{
31.
```

```

32.
33.         int fpid = getpid(); /* Restituisce l' ID del processo figlio del comando */
34.
35.         int er = execvp(ar[0], &ar[0]); /* Viene eseguito il comando con i rispettivi
parametri e restituisci -1 se il comando è errato */
36.
37.         if(er == -1){
38.
39.             printf("Errore = %d\n\n", errno); /* La variabile errno contiene il
numero dell' errore commesso all' esecuzione del comando che non è andato a buon
fine*/
40.
41.             char *p=strerror(errno); /* Puntatore alla stringa di errore che viene
restituita dalla funzione*/
42.
43.             write(f, p, strlen(p)); /* Scrivo l'errore sul file di out.i specifico*/
44.
45.             kill(fpid, SIGKILL); /* Invia al processo figlio il segnale SIGKILL
cioè il processo viene terminato in maniera sicura */
46.         }
47.
48.         close(f); /* Il file di output del comando viene chiuso in quanto compilato*/
49.     }
50.
51. }
52.
53. /* change_sdoutput() effettua la creazione del file specifico di output e gli assegna il
file_descriptor, il quale viene restituito come ritorno */
54.
55. int change_sdoutput(){
56.
57.     int fd1;
58.     int fd2;
59.
60.
61.     char out[6]; /* Vettore contenente il nome del file output */
62.     sprintf(out, "out.%d", num_file); /* Creazione e inserimento del nome del file output
nel vettore */
63.
64.     remove(out); /* Viene azzerato il contenuto del file con quel nome se già esistente */
65.
66.     fd1 = open(out, O_CREAT | O_RDWR, 0666); /* Viene effettuata l'apertura del
file , se non esiste si crea e
67.         gli si assegna i permessi di lettura e scrittura al
proprietario e al gruppo */
68.

```

```

69.     num_file++; /* Si incrementa il numero che indentifica il file */
70.
71.     close (1); /* Viene effettuata la chiusura del file di output definito dal sistema */
72.
73.     fd2 = dup (fd1); /* Creo una copia del descrittore di file fd1, utilizzando il descrittore
    di file inutilizzato con il numero più basso,
74.         che è quello di output, per il nuovo descrittore. */
75.
76.     return fd2; /* Valore di ritorno è il descrittore di file di output del comando */
77. }
78.
79. /* comand_seq() elabora in maniera sequenziale ogni riga di comando inserita dall'utente */
80.
81. int comand_seq(){
82.
83.     char acr[BUFFER]; /* Buffer nel quale inserire la riga di comando */
84.     fgets(acr, BUFFER, stdin); /* Lettura da tastiera della riga di comando la quale viene
    salvata nel buffer */
85.
86.     acr[strlen(acr)-1] = '\0'; /* fgets() inserisce al penultimo valore "\n", ma per
    l'operazione successiva ho bisogno che tale valore sia "\0" */
87.
88.     if(strlen(acr) != 0){ /* Controllo se la lunghezza della riga inserita è uguale o diversa
    da zero */
89.
90.         int f = change_sdoutput();
91.
92.         const char s[2]= " "; /* Vettore contenente il carattere di separazione */
93.         char *token; /* Puntatore usato per l'ultimo token della lista */
94.         char *ar[BUFFER]; /* Buffer contenente la riga comando rivisionata */
95.         token = strtok(acr, s); /* Assegna al puntatore il primo token */
96.
97.         printf("Comando Inserito :\n\n");
98.
99.         int i=0;
100.
101.         while(token != NULL){ /* Finche il puntatore non punta a nulla
    esegue la ricerca del successivo token */
102.
103.             ar[i]=token;
104.             printf("%s ", token);
105.             token=strtok(NULL,s);
106.             i++;
107.         }
108.         ar[i]=NULL; /* L'ultimo valore di un vettore che contiene un
    comando eseguibile deve essere NULL */

```



```

109.
110.             execution(ar, f, 's'); /* Esegue il comando inserito nel vettore ar,
           passandogli anche il file di output */
111.
112.             return 0; /* Indica che il comando c'è, che sia esatto o meno */
113.
114.         }else{
115.
116.             return 1; /* Indica che è stato inserito un nuova linea vuota */
117.
118.         }
119.     }
120.
121.     /* sequenziale() cicla finche l'utente non inserisce una riga vuota, altrimenti esegue il
           comando inserito. Vedremo che qui non si usa nessun file intermedio,
122.     ciò rende piu l'idea di sequenzialità, oltre a utilizzare nella execution() la funzione
           wait() che ci permette lo svolgimento sequenziale dei comandi */
123.
124.     void sequenziale(){
125.
126.         printf("Inserisci i comandi da svolgere in modo sequenziale : \n");
127.
128.         while(1){ /* Cicla finche non viene inserita una riga vuota */
129.
130.             int control = comand_seq();
131.
132.             if(control == 1){
133.
134.                 exit(1); /* Indica l'uscita dal programma a causa di inserimento
           di riga vuota */
135.             }
136.         }
137.     }
138.
139.     /* comand_par() elabora in maniera "parallela" ogni riga di comanda inserita
           dall'utente, tramite l'uso di un file intermedio */
140.
141.     void comand_par(){
142.
143.         FILE *fd; /* Puntatore al file che conterra tutti i comandi che sono stati
           inseriti precedentemente dall'utente*/
144.         char buf[BUFFER]; /* Buffer nel quale inserire la riga di comando letta dal
           file dei comandi */
145.         char *res; /* Puntatore al buffer che conterrà la riga di comando letta dal file
           dei comandi */
146.

```

```

147.         fd=fopen("file_comand.txt", "r");
148.
149.         if( fd==NULL ){
150.
151.             perror("Errore in apertura del file"); /* Vuol dire che il file non si è
             aperto in maniera corretta quindi si esce dal programma*/
152.             exit(1);
153.
154.         }
155.
156.         while(1){ /* Cicla finché non viene letta dal file una riga vuota */
157.
158.             res=fgets(buf, BUFFER, fd); /* Restituisce il puntatore al buffer
             contenente la riga di comanda*/
159.
160.             if( res==NULL ){
161.                 break; /* Esce dal while in quanto è stata inserita una riga
                 vuota nel file dei comandi, quindi punta a NULL */
162.             }
163.
164.             int f = change_sdoutput();
165.
166.             /* Stesso procedimento di creazione del vettore comando, di
             comand_seq() */
167.
168.             buf[strlen(buf)-1] = '\0';
169.
170.             const char s[2]= " ";
171.             char *token;
172.             char *ar[BUFFER];
173.             token = strtok(buf, s);
174.
175.             printf("Comando Inserito :\n\n");
176.
177.             int i=0;
178.
179.             while(token != NULL){
180.
181.                 ar[i]=token;
182.                 printf("%s ", token);
183.                 token=strtok(NULL,s);
184.                 i++;
185.             }
186.
187.             ar[i]=NULL; /* L'ultimo valore di un vettore che contiene un
             comando eseguibile deve essere NULL */

```

```

188.
189.             execution(ar, f, 'p'); /* Esegue il comando inserito nel vettore ar,
           passandogli anche il file di output */
190.             }
191.
192.             fclose(fd); /* Il file dei comandi viene chiuso */
193.         }
194.
195.     /* parallelo() cicla finche l'utente non inserisce una riga vuota, salvando ogni
           comando in un file dei comandi per poi eseguirlo in "parallelo".
196.     Nel parallelo usiamo un file d'appagio affinche ciò renda l'idea di un potenziale
           parallelismo sulla creazione del file, oltre al fatto che come
197.     vedremo nella execution(), per il parallelo per come lo abbiamo implementato , non
           effettua nessuna attesa della terminazione di un processo figlio
198.     prima di crearne un altro da parte del processo padre */
199.
200.     void parallelo(){
201.
202.
203.         printf("Inserisci i comandi da svolgere in modo parallelo : \n");
204.
205.         remove("file_comand.txt"); /* Viene azzerato il contenuto del file dei
           comandi se già esistente */
206.
207.         int fdc = open("file_comand.txt", O_CREAT | O_RDWR , 0666); /* Viene
           effettuata l'apertura del file dei comandi, se non esiste si crea e
208.                                     gli si assegna i
           permessi di lettura e scrittura al proprietario e al gruppo */
209.
210.         int control = 0; /* Variabile di controllo sul ciclo */
211.
212.         while(control == 0){ /* Cicla finchè l'utente non inserisce una riga vuota*/
213.
214.             char arcom[BUFFER]; /* Buffer contenente la riga di comando */
215.             fgets(arcom, BUFFER, stdin); /* Lettura da tastiera della riga di
           comando la quale viene salvata nel buffer */
216.
217.             arcom[strlen(arcom)-1] = '\0'; /* fgets() inserisce al penultimo valore
           "\n", ma per l'operazione successiva ho bisogno che tale valore sia "\0" */
218.
219.             if(strlen(arcom) == 0){ /* Controllo sulla lunghezza del buffer */
220.
221.                 control = 1; /* Vuol dire che la riga è vuota */
222.
223.             }else{
224.

```

```

225.                write(fdc, arcom, strlen(arcom)); /* Inserisco la riga scritta su
    terminale nel file dei comandi */
226.                write(fdc, newline, strlen(newline)); /* Inserisco il valore di
    newline */
227.                }
228.            }
229.
230.        close(fdc); /* Chiudo il file dei comandi */
231.
232.        comand_par(); /* Richiamo il metodo per far eseguire in "parallelo" i
    comandi inseriti nel file dei comandi */
233.
234.        exit(1); /* Indica l'uscita dal programma a causa della lettura di tutti i
    comandi */
235.    }
236.
237.    /* main() avvia il programma stampando a video il menù con la scelta effettuabile
    dall'utente e prende in input un valore che identifica la modalità
238.    di esecuzione dei comandi successivamente inseriti "s" = sequenziale e "p" =
    parallelo */
239.
240.    void main(){
241.
242.        printf("Seleziona 's' se vuoi eseguire una serie di comandi in modo
    sequenziale\n");
243.        printf("Seleziona 'p' se vuoi eseguire una serie di comandi in modo
    parallelo\n");
244.
245.        /* La scanf lascia nel buffer un carattere di fine riga '\n'. Quando chiami
    getchar(), in pratica toglie questo carattere dal buffer facendo sì che la fgets
    funzioni senza problemi. Senza il getchar(), la fgets legge come primo
    carattere un '\n' e si interrompe subito */
246.
247.        char sp[1024];
248.        char *p = sp;
249.
250.        scanf("%s", p);
251.        getchar();
252.
253.        if(strlen(sp) == 1){
254.
255.            switch(sp[0]){ /* Esegue lo switch sul valore inserito */
256.
257.                case 's': sequenziale(); /* Esegue i comandi inseriti da tastiera
    in modo sequenziale */
258.
                break;

```

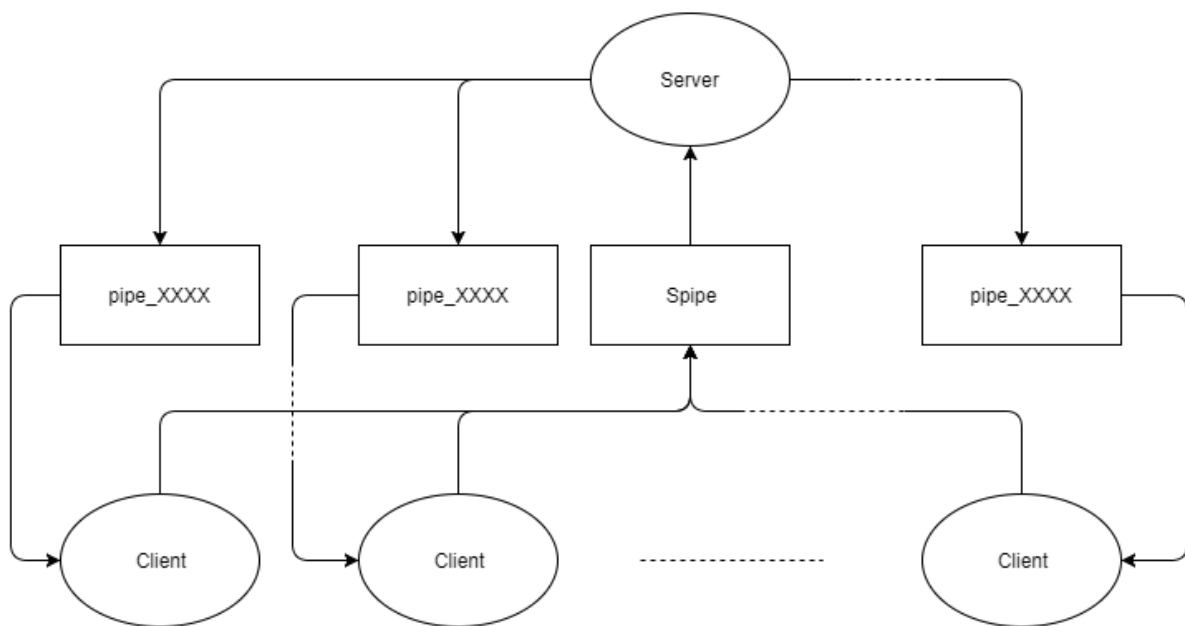
```
259.
260.                case 'p': parallelo(); /* Esegue i comandi inseriti da tastiera in
    modo "parallelo" */
261.                break;
262.
263.                default : printf("Riprova\n");
264.                main(); /* Richiama il metodo main() in quanto il dato
    inserito non è tra quelli indicati */
265.
266.                }
267.
268.            }else{
269.
270.                printf("Riprova\n");
271.                main(); /* Richiama il metodo main() in quanto il dato inserito non è
    tra quelli indicati */
272.            }
273.
274. }
```

ESERCIZIO 3

EVIDENZA DEL CORRETTO FUNZIONAMENTO DEL PROGRAMMA RISPETTO ALLE SPECIFICHE FORNITE

L'esercizio proposto mira a fornire le basi per la creazione del paradigma client-server avvalendosi di scambio di messaggi attraverso l'uso di pipe con nome all'interno dello stesso Sistema Operativo.

Notiamo dalle specifiche che, dovendo esistere un'unica pipe, denominata “Spipe”, per la comunicazione da client a server, e molteplici pipe, create solo all'occorrenza e indicate con pipe_XXXX, per la comunicazione dal server ai client, la struttura che andremo a creare può essere rappresentata come segue:



Una scelta implementativa su cui volgiamo porre subito attenzione è il modo in cui il server si occupa di mantenere le informazioni riguardanti i client. A tale scopo, nel programma server, dopo aver incluso tutti gli header file di cui esso ha bisogno per funzionare, è stata dichiarata una struttura adibita a rappresentare i metadati inerenti ai client.

```
struct client{
    int p; /*bit di presenza*/
    int pid; /*process id del client*/
    char client_name[9]; /*nome identificativo della pipe su cui il server può scrivere al client*/
}client;
```

Di seguito è dichiarato un vettore composto dalle suddette strutture, che via via terrà memoria dei client connessi, delle locazioni utilizzate ma di nuovo disponibili e di quelle non ancora utilizzate e dunque sempre disponibili.

Appena lanciato il programma server, viene dunque stampata una rappresentazione grafica del vettore, come segue:

A questo punto il server crea e apre in lettura il file descriptor della pipe su cui i client scriveranno le azioni che desiderano effettuare.

```
dave@dave-VirtualBox: ~/Scrivania
dave@dave-VirtualBox:~/Scrivania$ ./SERVER
NULL-->NULL-->NULL-->NULL-->NULL-->NULL-->NULL-->NULL-->Fine lista.
```

Il programma client, quindi, inizia la sua esecuzione stampando a video un menù che associa i numeri da 1 a 5 ad una serie di operazioni che l'utente può effettuare:

- 1 = Connessione al server.
- 2 = Richiesta elenco ID dei client connessi.
- 3 = Invio messaggio testuale ad un client o ad un gruppo di client.
- 4 = Disconnessione dal server.
- 5 = Chiusura del client.

In questa situazione iniziale le uniche operazioni che il client potrà effettuare sono la connessione al server e la chiusura del programma client. Qualunque altra richiesta (immissione del numero 3 nell'esempio) verrà respinta, e sarà stampato a video il seguente messaggio:

```
dave@dave-VirtualBox: ~/Scrivania
dave@dave-VirtualBox:~/Scrivania$ ./CLIENT

Inserire un intero corrispondente alle seguenti istruzioni:

1 = Collegati al server
2 = Richiesta elenco dei Client connessi
3 = Invio messaggio testuale a un client o un gruppo di client
4 = Scollegati dal server
5 = Esci dal programma

3

È RICHIESTA UNA CONNESSIONE AL SERVER.

Inserire un intero corrispondente alle seguenti istruzioni:

1 = Collegati al server
2 = Richiesta elenco dei Client connessi
3 = Invio messaggio testuale a un client o un gruppo di client
4 = Scollegati dal server
5 = Esci dal programma
```

Quindi, presupponendo che l'utente non chiuda il programma (funzione che in questo caso si occupa solo di stampare il messaggio di uscita e poi eseguire la `exit(0)`), in corrispondenza di una richiesta di connessione, il client tramite il metodo `connessione()`, procede ad aprire a sua volta la pipe su cui scrivere il comando da effettuare e il numero corrispondente al suo Process ID.

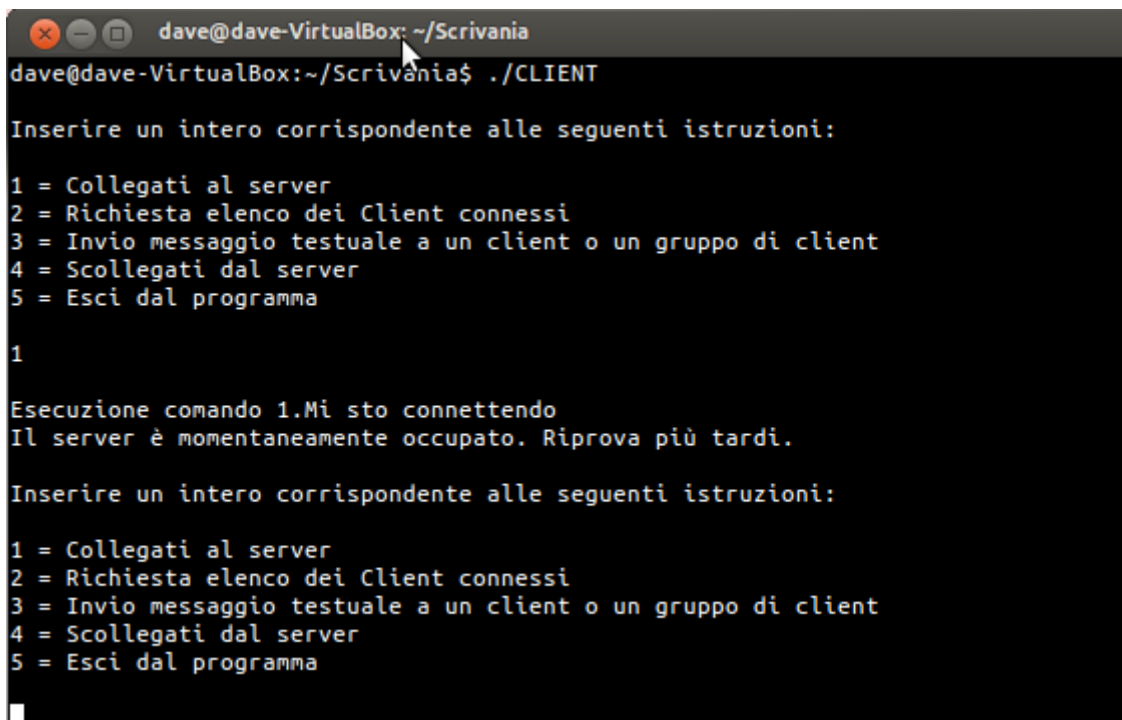
A questo punto il server, leggendo tali informazioni dalla pipe, inizia il processo di memorizzazione dei dati riguardanti il client all'interno del suo vettore di strutture lanciando i metodi `new_connection()` che a sua volta si avvale del metodo `new_client()`.

Vogliamo porre particolare attenzione al fatto che il server dovendo associare al nuovo client connesso un ID, farà uso dell'indice del suo vettore, così che nello svolgimento di tutte le operazioni future, sarà usato l'ID del client richiedente una funzione, per andare a identificare i suoi metadati in modo diretto e univoco.

A questo punto, dal lato server non resta che comunicare al client, l'identificativo con cui il server lo ha registrato. Dunque, viene aperta la pipe specifica al nuovo client connesso, il quale leggerà quale ID gli è stato associato.

Da questo punto in poi per effettuare qualunque altra operazione, il client comunicherà al server l'operazione da svolgere e il proprio ID.

Come abbiamo detto, il server mantiene le informazioni dei client all'interno di un vettore, che dunque ha dimensione finita. Nel caso in cui il vettore risultasse pieno, il server non sarebbe in grado di gestire una nuova connessione. In questo scenario dunque l'ID che viene restituito al client sarebbe uguale a `[-1]`. Una volta letta la risposta del server, il client controlla che l'ID ad esso associato sia valido e quindi diverso da `-1`. Se così non fosse, il client si occupa di stampare un messaggio per informare l'utente che il server risulta pieno, a quindi di riprovare in un secondo momento.



```
dave@dave-VirtualBox: ~/Scrivania
dave@dave-VirtualBox:~/Scrivania$ ./CLIENT

Inserire un intero corrispondente alle seguenti istruzioni:

1 = Collegati al server
2 = Richiesta elenco dei Client connessi
3 = Invio messaggio testuale a un client o un gruppo di client
4 = Scollegati dal server
5 = Esci dal programma

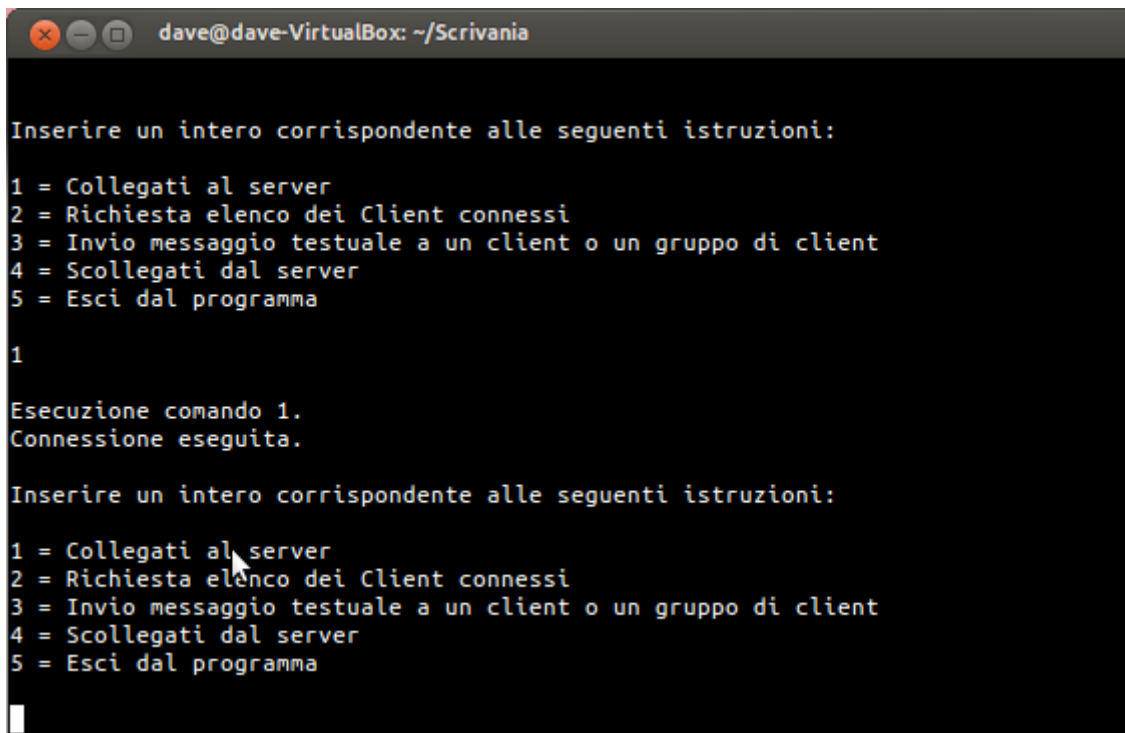
1

Esecuzione comando 1.Mi sto connettendo
Il server è momentaneamente occupato. Riprova più tardi.

Inserire un intero corrispondente alle seguenti istruzioni:

1 = Collegati al server
2 = Richiesta elenco dei Client connessi
3 = Invio messaggio testuale a un client o un gruppo di client
4 = Scollegati dal server
5 = Esci dal programma
```

Se invece il server risulta disponibile e la connessione va a buon fine, le operazioni che il client può effettuare vanno dalla 2 alla 5 (a questo punto infatti la richiesta di connessione genera una stampa che avverte che la connessione è già stata eseguita).



```
dave@dave-VirtualBox: ~/Scrivania

Inserire un intero corrispondente alle seguenti istruzioni:

1 = Collegati al server
2 = Richiesta elenco dei Client connessi
3 = Invio messaggio testuale a un client o un gruppo di client
4 = Scollegati dal server
5 = Esci dal programma

1

Esecuzione comando 1.
Connessione eseguita.

Inserire un intero corrispondente alle seguenti istruzioni:

1 = Collegati al server
2 = Richiesta elenco dei Client connessi
3 = Invio messaggio testuale a un client o un gruppo di client
4 = Scollegati dal server
5 = Esci dal programma
```

Analizzeremo dapprima cosa accade in corrispondenza delle operazioni 2, 4 e 5, per poi andare a descrivere il meccanismo per l'invio e la ricezione dei messaggi.

Supponendo dunque che l'utente voglia visualizzare la lista dei client attualmente connessi al server, digitando il comando 2, il client lancia il metodo `require_client_list()`, comunicando al server l'operazione da svolgere e il proprio ID.

Quindi il server dopo aver letto il contenuto della pipe di comunicazione (Spipe), si avvale del metodo `send_client_list()`.

In questo metodo abbiamo modo di vedere per la prima volta la struttura del “protocollo” utilizzato per la comunicazione di informazioni complesse, che non siano semplici interi o caratteri.

Dovendo infatti comunicare un'intera lista di ID, e dovendola presentare in maniera quantomeno comprensibile, viene da subito compreso che prima di poter inviare la lista, il server dovrà comunicare al client la dimensione di tale lista. Così facendo il client ha modo di creare un vettore della dimensione sufficiente a memorizzare la lista che riceverà.

Fatto ciò, il client riceve la lista sulla sua pipe (pipe_XXXX) e si occupa di stamparla a video.

Se invece l'utente avesse voglia di disconnettersi dal server, l'esecuzione del comando 4 equivarrebbe a richiamare il metodo `connectio_off()`, che scriverebbe sulla pipe (Spipe) l'intero corrispondente a tale operazione e l'ID del client che desidera disconnettersi.

Il server dunque non farà altro che utilizzare tale ID come indice per settare il bit di presenza nel suo vettore pari a 2 (la funzione che si occupa di realizzare il meccanismo appena descritto è chiamata `connection_off()` anche dal lato server).

La scelta di porre tale valore a 2, deriva dal fatto che in una prima fase di sviluppo del codice, era molto utile fare distinzione tra indici del vettore utilizzati, ma tornati disponibili e indici mai utilizzati e quindi sempre disponibili. In questo modo è stato più facile monitorare il corretto assegnamento degli ID (soprattutto quelli tornati disponibili).

Se invece l'utente ha intenzione di uscire dal programma, in corrispondenza del comando 5, il client non solo si occupa di stampare il messaggio di chiusura come in precedenza, ma essendo adesso

ancora connesso, viene richiamato anche il metodo `connection_off()` che si comporta esattamente come precedentemente descritto.

Passiamo ora alla descrizione dell'invio e la ricezione dei messaggi.

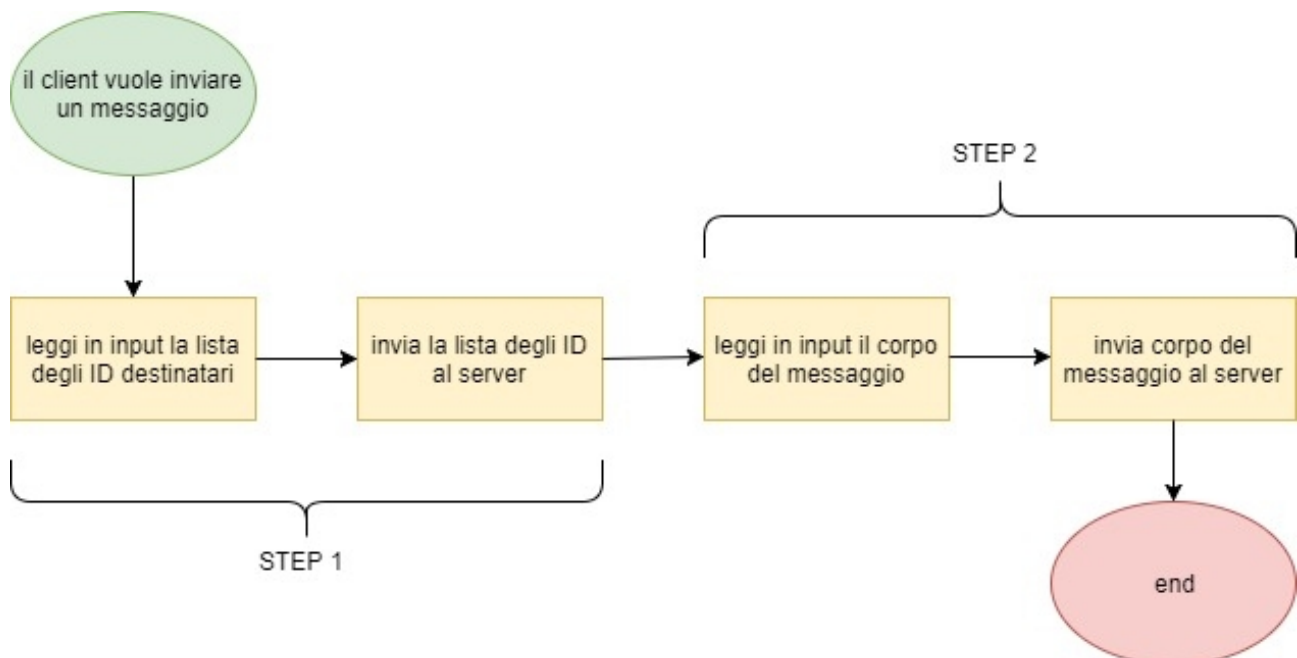
Dapprima specifichiamo che per inserire la lista degli ID destinatari, bisogna rispettare la seguente sintassi:

ID<spazio>ID<spazio>...ID<\n>

Anche qui, ovviamente, ci avvaliamo del “protocollo” precedentemente descritto di invio dimensione e invio contenuti.

Riteniamo di rilevante importanza sottolineare un problema di concorrenza tra client incontrato durante lo sviluppo di questa funzione.

In un primo momento la realizzazione della comunicazione tra client e server avveniva nel modo rappresentato di seguito.



Guardando il diagramma, notiamo come la richiesta da parte del client e dunque la ricezione da parte del server, fossero in un primo momento divise in due step consecutive:

CLIENT STEP 1:

- Si richiede all'utente di inserire la lista dei client.
- Si invia la dimensione di tale lista al server.
- Si invia al lista al server.

SERVER STEP1:

- Viene letta la lunghezza della lista degli ID destinatari.
- Viene letta la lista dei destinatari.

CLIENT STEP2:

- Si richiede al client di inserire il corpo del messaggio
- Si invia la dimensione del messaggio
- Si invia i corpo del messaggio

SERVER STEP2:

- Viene letta la dimensione del messaggio.
- Viene letto il corpo del messaggio.

Questa scelta implementativa si è rivelata un vero e proprio flop nel momento in cui 2 client avrebbero provato ad inviare un messaggio nello stesso momento.

Siano CLIENT1 e CLIENT 2 i due mittenti, basta osservare cosa accade quando gli step vengo eseguiti nell'ordine seguente:

CLIENT1 STEP1 → SERVER STEP1 → CLIENT2 STEP1 → SERVER STEP2 → ...
(il server eseguirà comunque i suoi 2 passi nell'ordine prestabilito).

Quello che accade è che entrambi i client avrebbero inviato sulla pipe (Spipe) a lista dei destinatari, ma il CLIENT2 andrebbe a disturbare il server nel momento in cui quest'ultimo fosse in attesa di leggere (prima la dimensione e poi) il messaggio del CLIENT1.

Si è dunque arrivati alla conclusione che il programma client, prima di comunicare qualunque informazione al server avrebbe dovuto avere a sua disposizione tutte le informazioni necessarie all'esecuzione dell'operazione. In corrispondenza del comando 5 infatti, prima di richiamare il metodo `send_msg()`, che si occupa di effettuare le 4 write del client sopra descritte, viene chiesto all'utente di inserire sia la lista degli ID destinatari, sia il corpo del messaggio. A questo punto il programma client può finalmente comunicare al server (ovviamente l'operazione che vuole svolgere e) la lista degli ID destinatari e il corpo del messaggio in modo "atomico". Abbiamo così ovviato al problema di concorrenza tra client che ci si era presentato.

Una volta che il server è in possesso di tutte le informazioni di cui ha bisogno, dopo aver parsato la lista degli ID, utilizzando gli spazi che la caratterizzano, si occupa di generare un segnale SIGUSR1 a tutti i client corrispondenti agli ID validi inseriti nella lista. Discuteremo il metodo SIGHANDLER per il segnale SIGUSR1 in seguito.

Infine parliamo dei SIGHANDLER utilizzati per la gestione di alcune eccezioni:

Partiamo dalla gestione del segnale SIGINT (generato alla pressione Ctrl-C) da parte del client.

Se questo segnale viene generato, viene richiamato il metodo `get_interrupt()` che preventivamente controlla lo stato della connessione. Se la connessione è attiva viene richiamato il metodo `connection_off()` per comunicare al server la disconnessione del client e quindi il via libera alla liberazione della struttura che gli era assegnata. Quindi esce dal programma.

Se invece la connessione non era attiva, `get_interrupt()` semplicemente esce dal programma.

Per quanto riguarda la gestione del segnale SIGINT da parte del server invece, la funzione nominata anche qui `get_interrupt()` si occupa di scorrere l'intero vettore per comunicare ai client l'avvenuto crash inatteso. Nel farlo il server invia a sua volta il segnale SIGUSR2 ad ogni client,

Da parte loro, i client ricevendo SIGUR2 si avvalgono del metodo `manage_server_crash()`, che dopo aver stampato all'utente un messaggio riguardo alla situazione inattesa, esce dal programma.

La scelta di chiudere anche il programma client deriva dal fatto che, lasciando il client aperto, e dandogli quindi la possibilità di effettuare ulteriori operazioni, anche con un server non attivo, porterebbe il client in una situazione di totale inconsistenza.

Avremmo provato a gestire meglio questa situazione se l'apertura in scrittura della pipe (Spipe) con la label `O_NONBLOCK` non fosse bloccante (vedi commenti nel codice riga: 38).

Infine non ci rimane che esaminare l'ultimo SIGHANDLER rappresentato dal metodo

read_message() in corrispondenza del segnale SIGUSR1 ricevuto dal server.

Quest'ultimo, dopo aver notificato la presenza di un messaggio in arrivo al client, apre la pipe ad esso associata (pipe_XXXX) e vi invia il corpo del messaggio.

Nel metodo read_message() quindi, viene in un primo momento eseguita una stampa per notificare all'utente l'arrivo di un nuovo messaggio, e conseguentemente aperta la pipe (pipe_XXXX) dalla quale leggerne il corpo, che viene infine stampato a video.

Al termine di ognuna delle operazioni sopra esposte (fatta eccezione per quelle che terminano con una exit(0)), il flusso del programma torna nel main() e viene stampato il menù delle operazioni che è possibile eseguire.

DIAGRAMMA DI FLUSSO DEL PROGRAMMA CLIENT

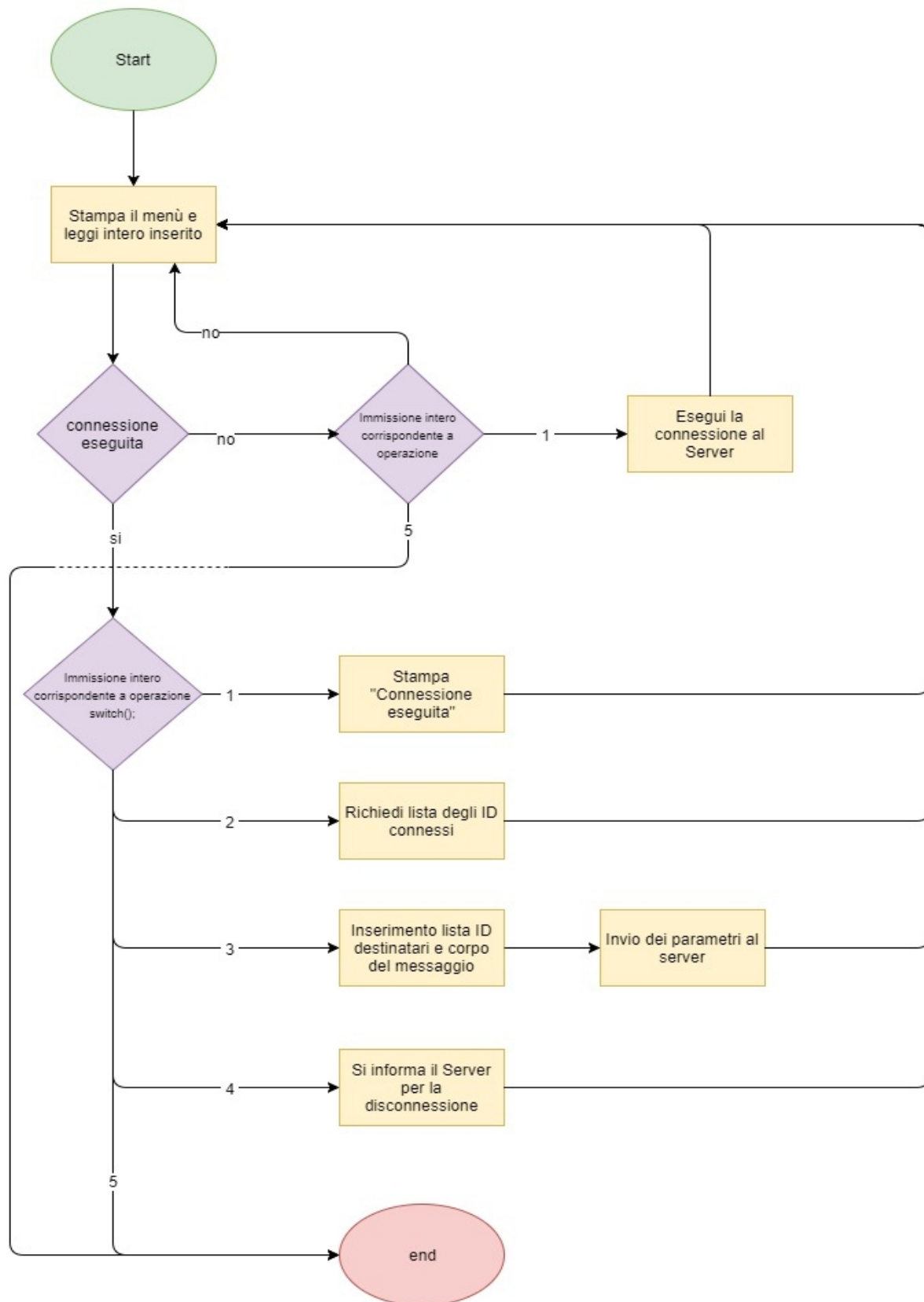
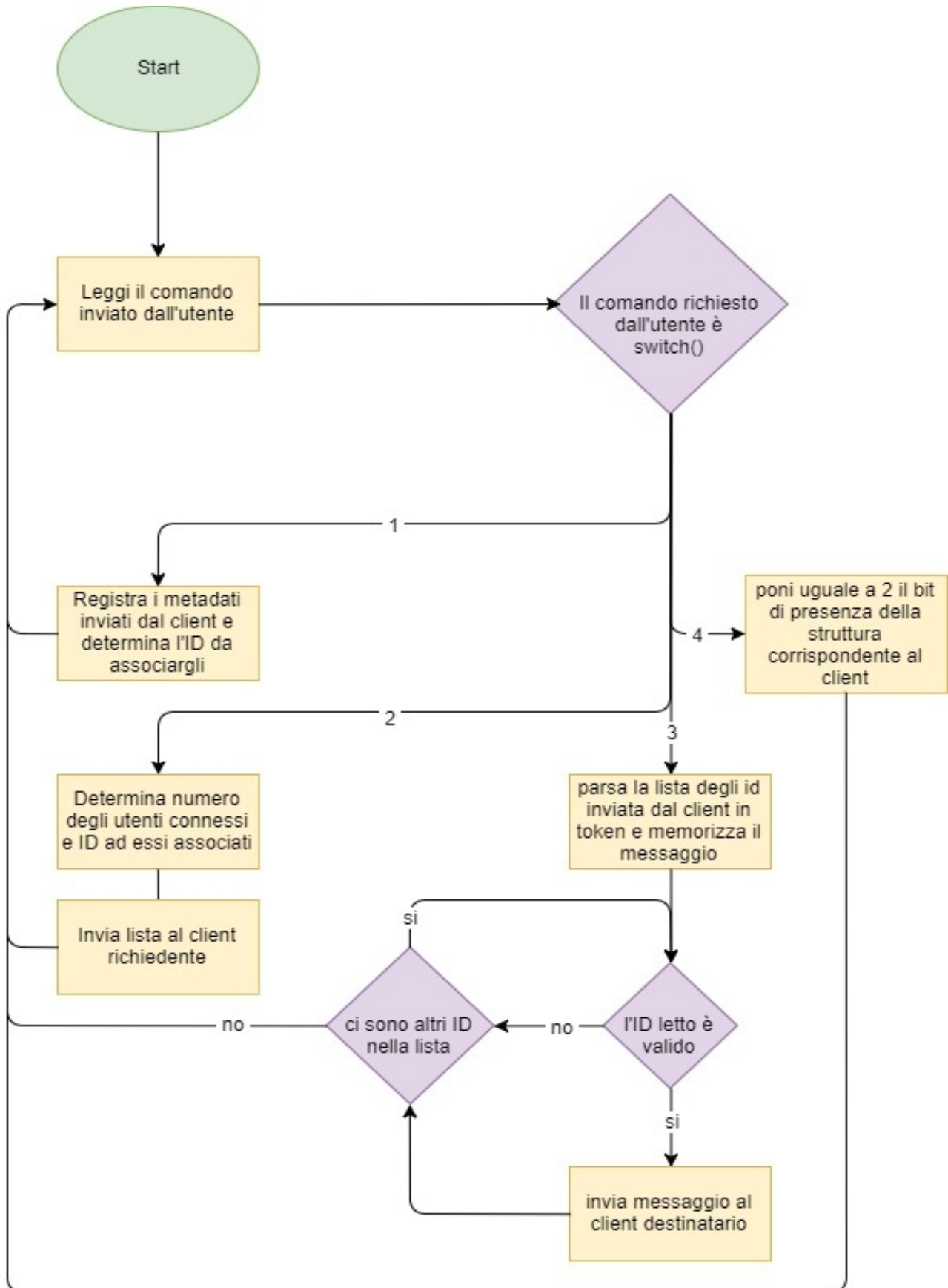


DIAGRAMMA DI FLUSSO DEL PROGRAMMA SERVER



Entrambi i digrammi di flusso precedentemente illustrati non considerano i SIGHANDLER presenti nel codice.

DESCRIZIONE DELL' IMPLEMENTAZIONE DEL PROGRAMMA CLIENT:

```
6. #include <stdio.h>
7. #include <fcntl.h>
8. #include <signal.h>
9. #include <stdlib.h>
10. #include <string.h>
11.
12.     int connection = 0; /* Indica lo stato della connessione: 0 = non connesso. 1 =
    connesso.*/
13.     int myID[1];
14.     int Spipe; /* Dichiarazione file descriptor della pipe per l'invio di informazioni al
    server. */
15.     char pipeName[9];
16.     int client_pipe; /* Dichiarazione file descriptor della pipe per la ricezione di
    informazioni dal server. */
17.     int name;
18. /*FUNZIONE DI STAMPA DEL MENÙ DELLE OPERAZIONI*/
19. void stampaMenu(){
20.
21.     printf("\nInserire un intero corrispondente alle seguenti istruzioni:\n\n");
22.     printf("1 = Connessione al server.\n");
23.     printf("2 = Richiesta elenco ID dei Client connessi.\n");
24.     printf("3 = Invio messaggio testuale ad un client o ad un gruppo di client.\n");
25.     printf("4 = Disconnessione dal server.\n");
26.     printf("5 = Chiusura del client.\n\n");
27.
28.
29. }
30.
31. /*FUNZIONE DI CONNESSIONE AL SERVER*/
32. void connessione(int *id){
33.     //int attempt_count = 0; /*Contatore che tiene traccia dei tentativi effettuati per
    aprire la pipe con cui comunicare informazioni al server.*/
34.     printf("Mi sto connettendo\n");
35.     int info[2]; /* Dichiarazione vettore che conterrà numero dell'operazione e nome
    client (PID). */
36.
37.     name = getpid(); /*Associazione PID al nome del client.*/
38.
39.     info[0] = 1; /* Numero operazione da inviare al server.*/
40.     info[1] = name; /*PID da inviare al server*/
41.
42.     do{
43.         Spipe = open("Spipe", O_WRONLY/*, O_NONBLOCK*/); /* Apertura
    della pipe con cui comunicare informazioni al server. In questo punto avremmo voluto
    utilizzare la label O_NONBLOCK per non bloccare l'apertura della pipe ed
    eventualmente stampare un messaggio di avviso per notificare che il server non è
    raggiungibile nel caso in cui un client provi a connettersi prima che il server sia stato
    avviato. Ciò non è stato possibile poichè anche inserendo la label O_NONBLOCK,
    l'apertura della pipe risulta comunque essere bloccante. Quindi provare a connettere un
```

client al server prima che quest'ultimo sia già avviato, porta il client ad un punto morto. L'unica recovery possibile è la combinazione di tasti Ctrl+C, che permette al client di terminare per essere poi rilanciato una volta che il server diventa operativo. Di seguito sono commentati frammenti di codice utilizzati nel caso in cui O_NONBLOCK funzionasse come desiderato.*/

```
44.
45.         //attempt_count ++;
46.         //printf("%d\n", attempt_count);
47.
48.     }while(Spipe == -1 /*&& attempt_count < 3*/);
49.     /*if(attempt_count == 3){
50.         printf("Il server non è momentaneamente raggiungibile.\nRiprova più
tardi.\n");
51.     }else{*/
52.
53.
54.         write(Spipe, &info, sizeof(info)); /* Invio dell'operazione e del PID del
client che si sta connettendo. */
55.         close(Spipe);
56.
57.         sprintf(pipeName, "pipe_%d", name); /* Concatenazione della stringa
"pipe_" e nome (PID) del client. Sarà il nome utilizzato per indicare la pipe dal quale il
client riceverà informazioni dal server. */
58.
59.         do{
60.             client_pipe = open(pipeName, O_RDONLY);
61.         }while(client_pipe == -1);
62.
63.         read(client_pipe, myID, sizeof(myID)); /*Lettura dell'ID con cui il server
ha identificato il client. Questo ID corrisponde all'indice del vettore mantenuto dal
server, in cui i metadati del nuovo client sono stati salvati.*/
64.
65.         if(myID[0] == -1){
66.             /* Un ID uguale a -1 indica una condizione in cui il server ha esaurito la capacità
del vettore in cui memorizzare i metadati dei client connessi. Viene quindi stampato il
seguente messaggio:*/
67.
68.             printf("Il server è momentaneamente occupato. Riprova più
tardi.\n");
69.
70.         }else{
71.             /*Altrimenti viene imposta la variabile globale connection = 1, che indica una
connessione avvenuta con successo, e stampato l'ID con cui il server ha identificato il
client.*/
72.             connection = 1;
73.             printf("L'ID associato a questa sessione è: %d\n\n", myID[0]);
74.         }
75.
76.         close(client_pipe);
77.         unlink(pipeName);
78.         //} /*È la parentesi di chiusura dell' if/else commentato in precedenza.*/
79. }
```



```

80.
    /*FUNZIONE PER LA DISCONNESSIONE DAL SERVER*/
81. void connection_off(){
82.     int info[2]; /* Dichiarazione vettore che conterrà numero dell'operazione e nome
    client (PID). */
83.     info[0] = 4; /*Numero operazione da inviare al server.*/
84.     info[1] = myID[0]; /*ID con cui il server potrà identificare quale client si sta
    disconnettendo.*/
85.     do{
86.         Spipe = open("Spipe", O_WRONLY);
87.
88.     }while(Spipe == -1);
89.
90.     write(Spipe, &info, sizeof(info)); /*Invio del numero dell'operazione e dell'ID
    del client che si sta disconnettendo.*/
91.     connection = 0; /*Viene quindi impostata la variabile globale connection = 0 per
    indicare che il client non è connesso al server.*/
92.     close(Spipe);
93.
94. }
95.
96.
    /*FUNZIONE SIGHANDLER LA LETTURA DEI MESSAGGI*/
97. void read_message(){
98.     int sender_id[1]; /*Dichiarazione del vettore che conterrà l'ID del client
    mittente.*/
99.
100.    do{
101.        client_pipe = open(pipeName, O_RDONLY);
102.    }while(client_pipe == -1);
103.
104.    read(client_pipe, sender_id, sizeof(sender_id)); /*Lettura dell'ID
    corrispondente al client mittente.*/
105.
106.    printf("\nHAI UN NUOVO MESSAGGIO.\n\nIl client con ID %d
    scrive:\n", sender_id[0]); /*Stampa l'avvenuta ricezione del messaggio, e l'ID del client
    mittente.*/
107.
108.    int message_dim[1]; /*Dichiarazione vettore che conterrà la dimensione
    del messaggio. Utilizzato in seguito per dichiarare il vettore in cui salvare il messaggio.
109.    */
110.    read(client_pipe, message_dim, sizeof(message_dim)); /*Lettura della
    dimensione del messaggio.*/
111.
112.    char message[message_dim[0] + 1]; /*Dichiarazione del vettore che
    conterrà il messaggio.*/
113.
114.    read(client_pipe, message, sizeof(message)); /*Lettura/ricezione del
    messaggio.*/
115.
116.    message[message_dim[0]] = '\0';
117.

```

```

118.         printf("\n[%s]\n\n", message); /*Stampa il messaggio ricevuto.*/
119.
120.         stampaMenu(); /*Richiamiamo infine la funzione di stampa del menù. */
121.
122.         unlink(pipeName);
123.         close(client_pipe);
124.
125.     }
126.
127.     /*FUNZIONE SIGHANDLER PER LA CHIUSURA FORZATA Ctrl-C*/
128.     void get_interrupt(){
129.         printf("\n");
130.         if(connection == 1){ /*SIGINT potrebbe essere sollevato a prescindere
dalla connessione del client al server.*/
131.             connection_off(); /*In presenza di connessione viene richiamata la
funzione per la disconnessione, così da informare il server della chiusura da parte del
client. Quindi si esce dal programma*/
132.             printf("Chiusura del client.\n");
133.             exit(0);
134.         }else{
135.
136.             printf("Chiusura del client.\n");
137.             exit(0);
138.         }
139.     }
140.
141.
142.     /*FUNZIONE PER LA RICHIESTA DEGLI ID CONNESSI*/
143.     void require_client_list(){
144.         int ans_dim[1], info[2]; /*Dichiarazione vettore che conterrà la dimensione
della lista degli ID. Dichiarazione vettore che conterrà numero dell'operazione e nome
client (PID). */
145.
146.         info[0] = 2; /*Numero dell'operazione da inviare al server*/
147.         info[1] = myID[0]; /*Invio dell'identificativo.*/
148.         do{
149.             Spipe = open("Spipe", O_WRONLY);
150.         }while(Spipe == -1);
151.         write(Spipe, &info, sizeof(info)); /*Invio dei parametri al server.*/
152.         close(Spipe);
153.
154.         do{
155.             client_pipe = open(pipeName, O_RDONLY); /*Apertura in lettura
della pipe in cui il server ha inviato la lista degli ID connessi.*/
156.         }while(client_pipe == -1);
157.
158.         read(client_pipe, ans_dim, sizeof(ans_dim)); /*Ricezione dimensione
della lista.*/
159.
160.         char ans[ans_dim[0] + 1];
161.         read(client_pipe, ans, sizeof(ans)); /*Ricezione della lista ID.*/

```

```

162.         ans[ans_dim[0]] = '\0';
163.         printf("Gli id connessi sono:\n%s\n", ans);
164.         close(client_pipe);
165.         unlink(pipeName);
166.     }
167.
168.     /*FUNZIONE DI INVIO DEI MESSAGGI*/
169.     void send_msg(char *id_list, char *message){
170.         int list_dim[1], message_dim[1], info[2]; /* Dichiarazione vettore che
        conterrà numero dell'operazione e nome client (PID). Dichiarazione vettore che conterrà
        dimensione del messaggio. Dichiarazione del vettore che conterrà i parametri da inviare
        al server.*/
171.         info[0] = 3;
172.         info[1] = myID[0];/*Settaggio dei parametri da inviare al sever.*/
173.
174.         do{
175.             Spipe = open("Spipe", O_WRONLY); /* Apertura della pipe per
        l'invio dei parametri.*/
176.             }while(Spipe == -1);
177.
178.             write(Spipe, info, sizeof(info));/*Invio dei parametri al server.*/
179.
180.             list_dim[0] = strlen(id_list);
181.
182.             write(Spipe, list_dim, sizeof(list_dim));/*Invio lunghezza del vettore su
        cui salvare la lista degli ID destinatari.*/
183.
184.             write(Spipe, id_list, strlen(id_list));/*Invio lista degli ID destinatari.*/
185.
186.             message_dim[0] = strlen(message);
187.             message[message_dim[0]] = '\0';
188.
189.             write(Spipe, message_dim, sizeof(message_dim));/*Invio lunghezza del
        vettore su cui salvare il messaggio.*/
190.
191.             write(Spipe, message, strlen(message));/*Invio del messaggio.*/
192.
193.             close(Spipe);
194.         }
195.
196.     /*FUNZIONE SIGHANDLER PER LA GESTIONE DI CRASH DA PARTE
        DEL SERVER*/
197.     void manage_server_crash(){
198.         /*In corrispondenza del segnale SIGUSR2 viene notificato il crash del server ed
        eseguita la chiusura del client.*/
199.         printf("Il server è stato chiuso forzatamente e la connessione è stata
        interrotta.\nChiusura del client.\nPRIMA DI TENTARE UNA NUOVA
        CONNESSIONE ASSICURARSI CHE IL SERVER SIA OPERATIVO\n");
200.         exit(0);
201.
202.     }
203.

```

```

204.     void main(){
205.         int comando = 0;
206.
207.         signal(SIGINT, get_interrupt);/**/
208.         signal(SIGUSR1, read_message);
209.         signal(SIGUSR2, manage_server_crash);
210.
211.         while(1){
212.
213.
214.             if(connection == 0){
215.                 do{
216.
217.                     stampaMenu();
218.                     scanf("%d", &comando);/*Lettura del comando da
eseguire.*/
219.                     printf("\n");
220.                     if(comando == 1){
221.                         printf("Esecuzione comando 1.");
222.                         connessione(myID);
223.                     }else if(comando == 5){
224.                         printf("Esecuzione comando 5.\n");
225.                         printf("Esco dal programma.\n");
226.                         exit(0);
227.                     }else{
228.                         printf("È RICHIESTA UNA
CONNESSIONE AL SERVER.\n\n");
229.                     }
230.
231.                 }while(connection != 1);
232.
233.
234.             }else{
235.                 do{
236.
237.
238.                     stampaMenu();
239.                     scanf("%d", &comando);
240.                     printf("\n");
241.
242.                     switch(comando){
243.
244.
245.                         case 1 : printf("Esecuzione comando
1.\n");
246.
247.                         printf("Connessione
eseguita.\n");
248.
249.                         break;
250.                         case 2 : printf("Esecuzione comando
2.\n");
251.
252.                         break;
253.
254.                     }
255.                     require_client_list();/*Richiamiamo funzione per la richiesta degli ID connessi.*/

```

```

250.                                     break;
251.                                     case 3 : printf("esecuzione comando
        3.\n");
252.
253.                                     char id_list[32],
        message[2048];      /*Sono rispettivamente il vettore per la memorizzazione degli ID
        destinatari e del messaggio. Per la lista degli ID è stato scelto di inserirli in un vettore di
        caratteri per poter utilizzare gli spazi come indici per il parsing.*/
254.
255.                                     printf("Inserire gli ID
        destinatari del messaggio.\n");
256.                                     getchar();
257.                                     fgets(id_list, 32, stdin);
        /*Immissione destinatari del messaggio*/
258.
259.                                     printf("Inserire corpo del
        messaggio.\n");
260.                                     fgets(message, 2048, stdin);
        /*Immissione corpo del messaggio*/
261.
262.                                     send_msg(id_list, message);
263.                                     break;
264.                                     case 4 : printf("Esecuzione comando
        4.\n");
265.                                     connection_off();/*Richiamo
        funzione per la disconnessione*/
266.                                     printf("Mi disconnetto dal
        server.\n\n");
267.                                     break;
268.                                     case 5 : printf("Esecuzione comando
        5.\n");
269.                                     printf("Esco dal
        programma.\n");
270.                                     connection_off();/*Richiamo
        funzione per la disconnessione*/
271.                                     exit(0);
272.                                     default : printf("Inserimento errato.
        Riprova.\n\n");
273.                                     }                                     main();
274.
275.
276.                                     }while(connection == 1);
277.                                     }
278.                                     }
279.                                     }

```

DESCRIZIONE DELL'IMPLEMENTAZIONE DEL PROGRAMMA SERVER:

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <sys/types.h>
4. #include <sys/stat.h>
5. #include <fcntl.h>
6. #include <stdlib.h>
7. #include <signal.h>
8. #include <string.h>
9.
10.
11. /* Dichiarazione della struttura che caratterizzerà il client all'interno del server. Questa
    struttura presenta rispettivamente: bit di presenza, PID del client connesso, nome associato
    alla pipe del client. */
12. struct client{
13.     int p; /*bit di presenza*/
14.     int pid; /*process id del client*/
15.     char client_name[9]; /*nome identificativo della pipe su cui il server può scrivere al
        client*/
16. }client;
17.
18. struct client client_vector[10]; /*Vettore delle strutture associate ad ogni client connesso.*/
19. int fdsc; /* Dichiarazione del file descriptor della pipe da cui il server leggerà istruzioni e
    contenuti inviati dal client.*/
20.
21.
22. /*FUNZIONE SIGHANDLER PER LA GESTIONE DEL CRASH DA PARTE DEL
    SERVER*/
23. void get_interrupt(){
24.     int index = 0; /*Indice per lo scorrimento sul vettore di strutture.*/
25.     while(index < 10){ /*Ciclo sul vettore per avvisare tutti i client connessi della
        chiusura forzata da parte del server.*/
26.         if(client_vector[index].p == 1){
27.             kill(client_vector[index].pid, SIGUSR2);
28.             index ++;
29.         }else{
30.             index ++;
31.         }
32.     }exit(1);
33. }
34.
35.
36. /*FUNZIONE DI REGISTRAZIONE E GESTIONE DEI METADATI DEL CLIENT*/
37. int new_client(char *name, int pid){
38.     int client_recorded;
39.     int index;
40.     client_recorded = 0; /*Viene trattato come un booleano per la guardia del while.
        Posto a 1 rompe la guardia e assicura che il salvataggio dei metadati è andato a buon fine.
        Altrimenti arriveremo alla fine del vettore (pieno).*/
41.
```

```

42.     index = 0; /*Indice di scorrimento sul vettore.*/
43.
44.     while(client_recorded == 0 && index < 10){
45.         if(client_vector[index].p != 1){
46.
47.             int i;
48.             client_vector[index].p = 1; /*Setto bit di presenza a 1. Indica che tale
indice è occupato dai metadati del client*/
49.             client_vector[index].pid = pid; /*Salvo il PID del client. Verrà
utilizzato per inviare segnali in caso di ricezione/invio messaggi*/
50.
51.             for(i = 0; i < 9; i++){ /*Ciclo sulla del nome per il salvataggio nella
struttura.*/
52.                 client_vector[index].client_name[i] = name[i];
53.
54.             }
55.             client_recorded = 1;
56.
57.         }else{
58.             index ++;
59.         }
60.         if(index == 10){
61.
62.             return -1;
63.         }
64.     }
65.     return index; /*Ritorno l'indice del vettore in cui sono stati salvati i metadati
riguardanti il nuovo client. Questo indice corrisponderà all'ID associato al client. Nel caso in
cui il vettore sia pieno, ritorna -1.*/
66. }
67.
68. /*FUNZIONE DI STAMPA DEI CLIENT CONNESSI AL SERVER*/
69.
70. /*Funzione di stampa del vettore dei client connessi. Ogni locazione del vettore è
rappresentata da ID|bit di presenza|. Tale coppia può essere: ID|1| in caso di presenza del
client, ID|2| in caso di locazione precedentemente utilizzata ma attualmente diponibile,
NULL altrimenti. Viene richiamata prima di ogni operazione richiesta dai client. Ha il solo
scopo di dare una rappresentazione grafica dello stato del server.*/
71.
72. void print_client_vector(){
73.     int i;
74.     for(i = 0; i < 10; i++){
75.         if(client_vector[i].p == 1 | client_vector[i].p == 2){
76.             printf("%d|%d|-->", i, client_vector[i].p);
77.         }else{
78.             printf("NULL-->");
79.         }
80.     }printf("Fine lista.\n\n");
81.
82. }
83.
84. /*FUNZIONE GESTIONE NUOVE CONNESSIONE*/

```

```

85. void new_connection(int *p){
86.
87.     char pipeName[9]; /*Dichiaro vettore di caratteri che conterrà il nome della pipe
relativa al client in connessione.*/
88.     int ID[1];
89.
90.     int pipe_XXXX; /* Dichiarazione file descriptor per la sudetta pipe.*/
91.
92.
93.     int name = p[1]; /* Contiene nome inviato dal client PID*/
94.
95.     printf("Un nuovo client vuole connettersi.\nIl suo PID è: %d.\n",name); /*Stampa
l'arrivo di una nuova connessione. Ha il solo scopo di informare lo stato di esecuzione in cui
è il server.*/
96.
97.     sprintf(pipeName, "pipe_%d", name); /* Concatenazione stringa "pipe_" e numero
PID. Verrà utilizzata per dare nome univoco alla pipe di comunicazione specifica per il
client. */
98.
99.     ID [0] = new_client(pipeName, name); /*Richiamiamo la funzione di registrazione
del nuovo client al server. Questa funzione tornerà l'ID da associare al client (corrispondente
alla posizione del vettore che avrà occupato), o -1 nel caso in cui il vettore di client sia
pieno.*/
100.
101.         mknod(pipeName,S_IFIFO, 0);
102.
103.         chmod(pipeName, 0660);
104.
105.
106.         do{
107.             pipe_XXXX = open(pipeName, O_WRONLY);
108.         }while(pipe_XXXX == -1);
109.
110.         write(pipe_XXXX, &ID, sizeof(ID)); /*Comunica al client (sulla sua relativa
pipe) l'ID che gli è stato assegnato. Può essere -1 in caso di saturazione del server.*/
111.         close(pipe_XXXX);
112.
113.         printf("L'ID ad esso associato è: %d.\n", ID[0]);/*Stampa risultato della
connessione. Ha il solo scopo di informare sull'esito della connessione dal lato server.*/
114.
115.
116.     }
117.
118.     /*FUNZIONE DISCONNESSIONE DAL SERVER*/
119.     void connection_off(int *ID){
120.         int free_id = ID[1];
121.         client_vector[free_id].p = 2; /*Settiamo il bit di presenza a 2. Indica che
l'indice è stato utilizzato ma è tornato disponibile per il salvataggio di nuovi metadati.*/
122.     }
123.
124.
125.     /*FUNZIONE DI INVIO DEGLI ID CLIENT CONNESSI AL SERVER*/

```



```

126. void send_client_list(int *ID){
127.     int count = 0; /*Contatore dei client attualmente connessi.*/
128.     char SINGLE_CLIENT[8]; /*Dichiarazione vettore stringa in cui salveremo
gli 8 caratteri corrispondenti a "ID:|%d|".*/
129.     int i = 0; /*Indice di scorrimento sul vettore.*/
130.     int client_id = ID[1];
131.
132.
133.     while(i < 10){ /*Ciclo sul vettore per il conteggio dei client attualmente
connessi e registrati.*/
134.         if(client_vector[i].p == 1){
135.             count ++;
136.             i ++;
137.         }else{
138.             i ++;
139.         }
140.     }
141.
142.     int dim[1];
143.     dim[0] = sizeof(SINGLE_CLIENT)*count; /*Calcolo lunghezza del vettore
che conterrà la stringa "ID:|%d|" per ogni client connesso. Tale vettore avrà dimensione pari
alla lunghezza della stringa "ID:|%d|" (8 caratteri), moltiplicata per il numero dei client
connessi.*/
144.
145.
146.     char CLIENT_LIST[dim[0] + 1]; /*Dichiarazione del sudetto vettore. Viene
aggiunto 1 alla sua dimensione, in modo da poter inserire il cattere di fine stringa '\0'*/
147.
148.     i= 0; /*L'indice sull'array viene reimpostato a zero per ricominciare il ciclo
sul vettore. Questa volta verrà rianalizzato l'intero vettore per poter concatenare tutte le
stringhe "ID:|%d|".*/
149.
150.     int j; /*Indice del for che ciclerà ricorsivamente la stringa "ID:|%d|" per
andarla a copiare nel vettore finale da inviare al client*/
151.     int index = 0;
152.
153.
154.     printf("Il client con ID %d richiede la lista dei client connessi.\n", client_id);
155.     while(i < 10){
156.         if(client_vector[i].p ==1){
157.
158.             sprintf(SINGLE_CLIENT, "ID:|%d|.\n", i);
159.
160.             for(j = 0; j < 8; j++){
161.
162.                 CLIENT_LIST[8*index + j] = SINGLE_CLIENT[j];
/*Viene calcolato l'indice in cui andare a scrivere il prissimo carattere.*/
163.
164.             }
165.
166.             index ++;
167.             i ++;

```

```

168.             CLIENT_LIST[dim[0]] = '\0'; /*Inserimento del carattere di
    fine stringa in fondo al vettore.*/
169.             }else{
170.                 i++;
171.             }
172.         }
173.
174.
175.         int pipe_XXXX; /* File descriptor pipe relativa al client.*/
176.
177.
178.         mknod(client_vector[client_id].client_name,S_IFIFO, 0);
179.
180.         chmod(client_vector[client_id].client_name, 0660);
181.
182.
183.         do{
184.
185.             pipe_XXXX = open(client_vector[client_id].client_name,
    O_WRONLY);
186.
187.             }while(pipe_XXXX == -1);
188.
189.             write(pipe_XXXX, &dim, sizeof(dim)); /*Invio informazione riguardante la
    dimensione della stringa che sarà inviata al passo successivo. In questo modo il client potrà
    creare a sua volta un vettore stringa su cui effettuare la read().*/
190.
191.             write(pipe_XXXX, &CLIENT_LIST, sizeof(CLIENT_LIST)); /*Invio della
    stringa contenente gli ID connessi.*/
192.
193.             close(pipe_XXXX);
194.
195.
196.         }
197.
198.         /*FUNZIONE GESTIONE RICEZIONE E INVIO MESSAGGI*/
199.         void manage_msg(int *ID){
200.             int vector_index;
201.             int client_id[1]; /*Conterrà l'ID del client che vuole inviare il messaggio.*/
202.             client_id[0] = ID[1];
203.
204.             int list_dim[1]; /*Conterrà la dimensione da assegnare al vettore in cui
    salvare la lista degli ID destinatari al momento della read().*/
205.             int message_dim[1]; /*Conterrà la dimensione del vettore in cui salvare il
    messaggio al momento della read().*/
206.
207.             printf("Il client con ID %d vuole inviare un messaggio.\n", client_id[0]);
208.
209.
210.             read(fdsc, list_dim, sizeof(list_dim)); /*Lettura della dimensione della lista
    degli ID destinatari che il server sta per ricevere.*/
211.

```

```

212.
213.         char id_list[list_dim[0]]; /*Dichiarazione del vettore che conterrà la lista
        degli ID connessi. Tale li sta sarà della forma ID<spazio>ID<spazio>...ID<\n>*/
214.
215.         read(fdsc, id_list, sizeof(id_list)); /*Lettura della lista degli ID destinatari.*/
216.
217.         id_list[list_dim[0] - 1] = '\0';
218.
219.         printf("La lista degli ID destinatari è:\n%s\n", id_list); /*Stampa della lista
        dal lato Server. Utile per monitorarne l'integrità.*/
220.
221.         read(fdsc, message_dim, sizeof(message_dim)); /*Lettura della dimensione
        del messaggio che il server sta per ricevere.*/
222.
223.
224.         char message[message_dim[0]]; /*Dichiarazione del vettore in cui verrà
        salvato il messaggio.*/
225.
226.         read(fdsc, message, sizeof(message)); /*Lettura del messaggio.*/
227.         message[message_dim[0] - 1] = '\0';
228.
229.         printf("Il messaggio inviato dall'utente è:\n%s\n", message); /*Stampa del
        messaggio dal lato Server. Utile per monitorarne l'integrità.*/
230.
231.         /*Poichè gli ID sono salvati in un array di caratteri, nel codice che segue
        viene effettuata la conversione da ACSII ad intero.*/
232.         int decade_count; /*contatore delle decine in caso di numeri a doppia cifra.*/
233.         int dest_id;      /*Conterrà l'ID del client destinatario.*/
234.         vector_index = 0; /*Indice per lo scorrimento su vettore.*/
235.
236.         while(vector_index < list_dim[0] - 1){
237.
238.             decade_count = 0;
239.             dest_id = 0;
240.             int pipe_XXXX;
241.             /*Nel while che segue si utilizzano gli spazi per parsare la stringa
        degli ID destinatari.*/
242.             while(id_list[vector_index] != ' ' && vector_index < list_dim[0] - 1){
243.
244.                 decade_count = decade_count * 10;
245.                 dest_id = dest_id * decade_count + (id_list[vector_index] -
        48);
246.
247.                 vector_index ++;
248.                 decade_count++;
249.             }
250.
251.             if(dest_id > sizeof(client_vector) || client_vector[dest_id].p != 1){
        /*Controllo validità degli ID destinatari. Tale controllo si avvale della dimensione del
        vettore e dei bit di controllo.*/
252.                 printf("ID %d non valido.\n", dest_id); /*Stampa dal lato

```

```

server il risultato della funzione.*/
253.
254.         }else{
255.
256.             printf("il PID del client destinatario è %d\n",
client_vector[dest_id].pid);/*Stampa Il PID del client destinatario*/
257.
258.             close(fdsc);
259.
260.             kill(client_vector[dest_id].pid, SIGUSR1);/*Invio del segnale
SIGUSR1 al client destiatario. Questo segnale permetterà a tale client di aprire la pipe ad
esso dedicata e leggere il messaggio che vi sarà inviato. */
261.
262.             mknod(client_vector[dest_id].client_name, S_IFIFO, 0); /*
Creazioe della pipe su cui il server invierà infomrazioni al client. */
263.
264.             chmod(client_vector[dest_id].client_name, 0660);
265.
266.             do{
267.
268.                 pipe_XXXX = open(client_vector[dest_id].client_name,
O_WRONLY); /* Apertura Pipe Client-Server specifica in scrittura */
269.
270.                 }while(pipe_XXXX == -1);
271.
272.                 write(pipe_XXXX, client_id, sizeof(client_id)); /*Invio al
destinatario ID del mittente.*/
273.                 write(pipe_XXXX, message_dim, sizeof(message_dim));
/*Invio al destinatario dimensione del messaggio per la creazione del vettore sul quale
effetture la read.*/
274.                 write(pipe_XXXX, message, sizeof(message)); /*Invio al
destinatario del messaggio.*/
275.                 close(pipe_XXXX);
276.                 printf("Il client con ID %d ha ricevuto il messaggio.\n",
dest_id); /*Stampa dal lato server il risultato della funzione.*/
277.
278.             }
279.             vector_index ++;
280.
281.         }
282.
283.
284.     }
285.
286.     void main(){
287.
288.         signal(SIGINT, get_interrupt);
289.
290.         while(1){
291.             close(fdsc);
292.             unlink("Spipe");
293.             print_client_vector(); /*Stampa iniziale dello stato dei client connessi al

```

```

server*/
294.
295.
296.         mknod("Spipe", S_IFIFO, 0); /* Creazione della pipe in sola scrittura per i
client e in sola lettura per il server.*/
297.
298.         chmod("Spipe", 0660); /* Assegnamento dei permessi sulla pipe*/
299.
300.
301.         do{
302.             fdsc = open("Spipe", O_RDONLY);
303.
304.         }while(fdsc == -1);
305.
306.
307.         int infoclient[2]; /* Vettore che conterrà le informazioni inviate dal client,
quali numero del comando da eseguire e nome( PROCESS ID DEL CLIENT ) */
308.
309.         read (fdsc, infoclient, sizeof(infoclient)); /* Lettura informazioni*/
310.
311.         switch(infoclient[0]){
312.
313.             case 1 :new_connection(infoclient);
314.                 break;
315.             case 2: send_client_list(infoclient);
316.                 break;
317.             case 3: manage_msg(infoclient);
318.                 break;
319.             case 4: printf("Il client con ID %d si sta disconnettendo.\n\n",
infoclient[1]);
320.                 connection_off(infoclient);
321.                 break;
322.
323.             }
324.         }
325. }

```