

# Relazione progetto

## SO A.A. 2016-2017

Autori:

- Federico Moncini, 5936828, [federico.moncini@stud.unifi.it](mailto:federico.moncini@stud.unifi.it)
- Tommaso Capecchi, 5943118, [tommaso.capecchi@stud.unifi.it](mailto:tommaso.capecchi@stud.unifi.it)

-----22/01/2018-----

### Esercizio 1: Scheduler di processi

Inizialmente lo scheduler stampa il menù delle operazioni disponibili richiedendo che l'utente inserisca un comando valido (intero tra 1 e 7 compresi). Per poter tener conto dei vari processi si mantiene una coda, il cui riferimento iniziale è salvato nella struttura Task con nome 'coda'. In base ai casi scelti lo scheduler completerà l'operazione richiesta. Di default lo scheduler implementa la politica di priorità maggiore.

#### Metodi principali:

- void printPrompt():  
Viene stampato il menù delle operazioni disponibili, attraverso la funzione printMenu(). Si richiede all'utente di inserire un valore accettabile compreso tra 1 e 7 inclusi e attraverso uno switch vengono analizzati i casi disponibili:

Caso 1) Si richiede l'inserimento di un nuovo Task specificando il nome, la priorità e il numero di esecuzioni rimanenti, attraverso la funzione createTask(). Al suo interno viene creata una variabile univoca utilizzata per tener di conto dell'identificativo del processo, tale variabile con nome 'id' viene incrementata di una unità ogni volta che si crea un

nuovo processo. Si crea inoltre un puntatore di tipo Task, allocando la memoria dinamicamente attraverso la funzione malloc(), e infine con le funzioni assignTaskName, assignTaskPriority e assignTaskExecLeft si inserisce rispettivamente il nome, la priorità e le esecuzioni rimanenti del processo. Dopo di che si aggiunge alla coda il nuovo Task appena creato, inserendolo nella posizione giusta in base all'attuale politica di scheduling; per fare ciò si utilizza la funzione ordering\_list();

Caso 2) Poiché questo caso corrisponde all'esecuzione in testa si richiama la funzione executeTaskInHead(), questa controlla che ci sia almeno un processo in testa alla coda da eliminare e si controlla che il numero di esecuzioni rimanenti sia maggiore di 1:

- se è maggiore, allora si decrementa di una unità il numero delle esecuzioni rimanenti e se la politica di scheduling non è per priorità viene riordinata la coda attraverso la funzione sort();
- altrimenti si elimina il processo in testa;

Caso 3) In questo caso si richiede l'id (che dovrà essere immesso all'utente tramite la funzione promptID la quale restituisce 1 se il processo è presente all'interno della coda altrimenti 0) del processo da eseguire. In caso positivo il processo viene eseguito attraverso la funzione executeTask(), che va alla ricerca del processo nella coda, lo esegue ed eventualmente lo elimina se il numero delle esecuzioni rimanenti è minore di 1;

Caso 4) Analogo al caso 3, ovvero si richiede l'inserimento dell'ID da parte dell'utente e attraverso la funzione eliminateID() si elimina il processo che corrisponde a tale ID;

Caso 5) Si modifica la priorità del processo avente ID inserito dall'utente, se questo esiste si richiama la funzione changePriority() che chiederà la nuova priorità modificando la precedente;

Caso 6) Si modifica lo scheduling attuale, richiamando la funzione changeSchedulingPolicy(), e la coda viene riordinata in base alla nuova politica;

## Caso 7) Uscita dal programma.

- void ordering\_list(Task \*task):

In questa funzione si controlla l'attuale politica di scheduling, dopo di che si inserisce il nuovo Task nella coda attraverso i metodi insertByPriority e insertByExecLeft.

- void insertByPriority(Task \*task):

Si controlla che la coda non sia vuota o che la priorità del nuovo Task sia minore della coda, in questo caso il nuovo Task viene inserito in coda, altrimenti si scorre la lista finchè non si trova un Task avente una priorità minore; se lo si trova allora il nuovo Task viene inserito prima di esso.

Nel caso in cui gli ID siano uguali si richiama la funzione getNodeWithMinId() che restituisce il Task avente ID minore e si procede con l'inserimento. Infine se non viene trovata alcuna priorità minore del nuovo Task, si inserisce in testa alla coda.

- void insertByExecLeft(Task \*task):

Si controlla che la coda non sia vuota o che le esecuzioni rimanenti del nuovo Task siano minori di quelli della coda, in questo caso il nuovo Task viene inserito in coda, altrimenti si scorre la lista finchè non si trova un Task avente una esecuzione rimanente minore; se lo si trova allora il nuovo Task viene inserito prima di esso. Nel caso in cui gli ID siano uguali si richiama la funzione getNodeWithMinId() che restituisce il Task avente ID minore e si procede con l'inserimento. Infine se non viene trovata alcuna esecuzione rimanente minore del nuovo Task, si inserisce in testa alla coda.

- void sort():

Si crea una variabile di tipo Task avente nome 'sorted' che conterrà la nuova lista ordinata. Si scorre tutta la coda attuale non ordinata e per ogni elemento si richiama l'insertion sort in base al tipo di politica di scheduling, che ha lo scopo di inserirlo nella coda ordinata 'sorted'.

Infine la coda viene posta uguale a 'sorted' in maniera che si abbia il riferimento alla nuova lista ordinata.

- void insertSortExecLeft(Task \*\*coda, Task \* new\_node):

La variabile 'coda', in questo caso, rappresenta il puntatore al riferimento in memoria alla variabile 'sorted' che contiene il riferimento all'ultimo processo della coda ordinata.

Se questo risulta essere nullo, non sono stati ancora inseriti Task al suo interno, dunque il nuovo nodo viene posto in coda in questo modo:

Si controlla che la coda abbia un numero di esecuzioni maggiori o uguali del nuovo nodo; se la coda e il nuovo nodo hanno lo stesso numero di esecuzioni rimanenti allora si procede ad ordinare la coda in base all'ID (ponendo il processo avente ID maggiore in coda) chiamando la funzione orderHeadByID(), che restituirà il riferimento alla coda. Se la coda è maggiore il nuovo nodo viene inserito prima della coda, aggiornando la nuova lista di elementi. Altrimenti si deve scorrere la lista per trovare un elemento avente numero di esecuzioni rimanenti maggiore rispetto al nodo da inserire.

Se il successivo del nodo corrente e il nuovo nodo hanno numero di esecuzioni rimanenti uguali, si controlla che l'id del nuovo nodo sia minore del successivo del nodo corrente e se risulta vero, si continua il ciclo fino a che non si trova un elemento con numero di esecuzioni rimanenti maggiore. Infine si effettua lo scambio posizionando il nuovo nodo prima dell'elemento avente numero di esecuzioni rimanenti maggiore.

La funzione insertSortPriority() è analoga, solo che l'ordinamento è effettuato in base alla priorità.

- void eliminateTask(Task \*current):

Elimino il Task nel caso in cui è stato scelto il caso 4 dall'utente oppure quando il numero di esecuzioni rimanenti di un processo è pari a 1 e viene eseguito. Viene salvato il successivo e il precedente dell'elemento da eliminare. Nel caso in cui si debba eliminare la coda si libera la memoria attraverso la funzione free() e la coda diventerà l'elemento successivo della lista. Nel caso in cui elimino dopo un'esecuzione avendo scelto il suo ID, controllo che quell'elemento non sia la coda; se lo è richiamo eliminateTask() impostando choice = 2 che indica l'eliminazione

in coda. Altrimenti elimino l'elemento corrente, facendo puntare il successivo del precedente al successivo del corrente.

## **Motivazioni delle scelte di implementazione:**

Nel file 'Config.h' si crea un nuovo tipo Task, utilizzato per inserire all'interno della coda i vari elementi; poiché la coda è una lista concatenata che effettua gli inserimenti in coda e le estrazioni in testa, all'interno della struttura Task oltre ai vari dati del processo, vi è anche il puntatore di tipo Task che tiene il riferimento dell'elemento successivo. Per effettuare lo scambio di politica si è utilizzato un enum che ci permette di rendere l'implementazione a livello del codice più sicura e leggibile. Inoltre abbiamo creato un altro enum di nome 'ExecutionChoice' utilizzato nel caso in cui si debba eseguire un processo in base all'ID scelto dall'utente ed eliminarlo quando questo non ha più esecuzioni rimanenti oppure nel caso 4 dove si elimina un processo in base all'ID. Per quanto riguarda l'ordinamento della coda dopo un cambio di politica di scheduling, si crea un nuovo riferimento in memoria che inizialmente è vuoto, dopo di che scorrendo la vecchia coda gli elementi verranno posizionati correttamente aggiornando la nuova coda ogni volta attraverso un algoritmo di ordinamento che si basa sull'Insertion Sort. Così facendo si cambiano i riferimenti ai puntatori degli elementi della coda già esistenti in modo da riordinarli evitando di allocare nuova memoria per una nuova coda.

## Screenshot:

```
federico@federico: ~/Documenti/Esercizio1_Finale
1) Crea un nuovo Task
2) Esegui Task in testa alla coda
3) Esegui Task in base all'ID
4) Elimina Task in base all'ID
5) Modificare la priorità di un Task in base all'ID
6) Cambiare politica di scheduling
7) Esci dal programma

Inserire un numero tra 1 e 7 compresi
1
  Inserire nome
proc1
  Inserire priorit  
11
  Inserire priorit  
5
  Inserire esecuzioni rimanenti
100
  Inserire esecuzioni rimanenti
11
+-----+-----+-----+-----+
| ID | PRIORITY | TASK NAME | REMAINING EXEC |
+-----+-----+-----+-----+
| 1 | 5 | proc1 | 11 |
```

- Inserimento di un nuovo processo a coda vuota. L'inserimento di una priorit   errata (la quale deve essere compresa tra 0 e 9) oppure di un numero di esecuzioni rimanenti errato (il quale pu   essere al pi   99 cicli) ripropone all'utente la richiesta.

```
+-----+-----+-----+-----+
| ID | PRIORITY | TASK NAME | REMAINING EXEC |
+-----+-----+-----+-----+
| 4 | 7 | proc4 | 3 |
+-----+-----+-----+-----+
| 1 | 5 | proc1 | 11 |
+-----+-----+-----+-----+
| 3 | 2 | proc3 | 2 |
+-----+-----+-----+-----+
| 2 | 2 | proc2 | 3 |
+-----+-----+-----+-----+

1) Crea un nuovo Task
2) Esegui Task in testa alla coda
3) Esegui Task in base all'ID
4) Elimina Task in base all'ID
5) Modificare la priorit   di un Task in base all'ID
6) Cambiare politica di scheduling
7) Esci dal programma

Inserire un numero tra 1 e 7 compresi
```

```

Inserire un numero tra 1 e 7 compresi
2
+-----+-----+-----+-----+
| ID | PRIORITY | TASK NAME | REMAINING EXEC |
+-----+-----+-----+-----+
| 4 | 7 | proc4 | 3 |
+-----+-----+-----+-----+
| 1 | 5 | proc1 | 11 |
+-----+-----+-----+-----+
| 3 | 2 | proc3 | 2 |
+-----+-----+-----+-----+
| 2 | 2 | proc2 | 2 |
+-----+-----+-----+-----+
1) Crea un nuovo Task
2) Esegui Task in testa alla coda
3) Esegui Task in base all'ID
4) Elimina Task in base all'ID
5) Modificare la priorità di un Task in base all'ID
6) Cambiare politica di scheduling
7) Esci dal programma
Inserire un numero tra 1 e 7 compresi

```

- Esecuzione di un Task in testa alla coda, in questo caso il processo con ID=2 sarà il primo ad essere eseguito e infatti successivamente il suo numero di esecuzioni rimanenti è decrementato di 1. Inizialmente la politica di scheduling è sempre per priorità.

<pre> Inserire un numero tra 1 e 7 compresi 3 Inserisci ID del task 5 Errore: id non trovato 1) Crea un nuovo Task 2) Esegui Task in testa alla coda 3) Esegui Task in base all'ID 4) Elimina Task in base all'ID 5) Modificare la priorità di un Task in base all'ID 6) Cambiare politica di scheduling 7) Esci dal programma Inserire un numero tra 1 e 7 compresi </pre>	<pre> Inserire un numero tra 1 e 7 compresi 8 Errore: comando non disponibile 1) Crea un nuovo Task 2) Esegui Task in testa alla coda 3) Esegui Task in base all'ID 4) Elimina Task in base all'ID 5) Modificare la priorità di un Task in base all'ID 6) Cambiare politica di scheduling 7) Esci dal programma </pre>
---	--

- Esecuzione di un Task con ID non esistente: il programma ripropone il menù, questo perché riferendosi alla lista di processi precedente non esiste nessun processo con ID=5. Questo avviene anche nel caso 4 e 5 nel caso si inserisca un ID non esistente. Nell'immagine a destra invece il programma ripropone il menù dopo che è stato inserito per errore un comando non disponibile.

```
1) Crea un nuovo Task
2) Esegui Task in testa alla coda
3) Esegui Task in base all'ID
4) Elimina Task in base all'ID
5) Modificare la priorità di un Task in base all'ID
6) Cambiare politica di scheduling
7) Esci dal programma

Inserire un numero tra 1 e 7 compresi
2
Nessun Task disponibile
1) Crea un nuovo Task
2) Esegui Task in testa alla coda
3) Esegui Task in base all'ID
4) Elimina Task in base all'ID
5) Modificare la priorità di un Task in base all'ID
6) Cambiare politica di scheduling
7) Esci dal programma
Inserire un numero tra 1 e 7 compresi
```

- Nel caso in cui non esiste alcun processo in coda e si prova ad eseguire il Task in testa la coda, la richiesta viene rifiutata.

```
6
Politica di scheduling: esecuzioni rimanenti
```

ID	PRIORITY	TASK NAME	REMAINING EXEC
3	2	proc3	2
2	2	proc2	2
4	7	proc4	3
1	5	proc1	11

- Cambio di politica di scheduling da priorità a esecuzioni rimanenti, facendo riferimento alla coda, dove era stato eseguito il Task con ID=2, precedentemente inserita.

## Istruzioni per compilare i file sorgenti:

Per quanto riguarda il main: `cc -o main Main.c`

Lanciare in esecuzione il programma: `./main`



## Esercizio 2: Esecutore di comandi

La nostra soluzione prevede inizialmente una stampa di un menù, nella quale l'utente può scegliere la modalità di esecuzione dei programmi:

- esecuzione sequenziale = 0:

A ogni comando inserito il programma lo esegue, creando il file di output out.<numero comando>. Il programma termina quando l'utente inserisce una riga di testo vuota.

- esecuzione in parallelo = 1:

Si richiedono i comandi da eseguire fino a che l'utente non inserisce una riga di testo vuota. In tal caso il programma genera tanti file quanti sono i comandi inseriti, ponendovi all'interno i rispettivi risultati. Dopo di che termina il programma.

### Metodi principali:

- void execSequentialMode(int n):

Questa funzione legge il comando inserito salvandolo nella variabile 'line'. Si effettua il controllo su tale variabile per controllare se vi è effettivamente un comando inserito. Dopo di che si sostituiscono gli spazi vuoti con il carattere '\0', per poter leggere prima l'istruzione e successivamente i vari argomenti, attraverso la funzione eliminateEmptySpace(). In seguito si effettua la lettura della riga appena modificata e si costruisce l'istruzione da poter eseguire. Infine si richiama la funzione createChild() che creerà un figlio del processo corrente che sarà incaricato di eseguire l'istruzione appena costruita con i vari argomenti e scrivere il risultato su file.

- void readLine(char line[], char \*instruction[]):

Attraverso un while si scandisce la variabile 'line' che contiene il comando inserito, fino a che non si trova uno '\n' che rappresenta il simbolo di fine riga. Al suo interno viene creato un altro ciclo necessario per copiare dal carattere i-esimo fino al carattere '\0' o '\n' la parte di istruzione necessaria per costruire il comando e salvarlo in 'instruction', che conterrà in ordine: l'istruzione e gli eventuali argomenti.

- void createChild(char\* instruction[], int n):

Si crea un processo figlio attraverso la funzione fork(), verificando che sia andato a buon fine controllando che il pid restituito sia maggiore uguale di 0.

Nel caso in cui il pid sia 0 (processo figlio), verrà creato un file per mezzo della funzione createFile() nella quale si ridirige lo standard output e lo standard error che ci consentono di scrivere rispettivamente il risultato o l'errore dell'esecuzione del comando. Per fare ciò è necessario utilizzare la funzione 'dup2' che come secondo argomento può ricevere l'intero '1' (ridirige lo standard output) oppure l'intero '2' (ridirige lo standard error). Attraverso la funzione 'execvp' viene eseguita l'istruzione ricevendo come primo parametro l'istruzione stessa e come secondo parametro i vari argomenti.

Nel caso in cui invece il pid è maggiore di 0 (processo padre), essendo la versione sequenziale, si attende che il figlio termini l'esecuzione attraverso la wait() e inoltre si controlla che il figlio sia correttamente terminato attraverso la macro WIFEXITED(status).

- void execParallelMode(int n):

Si apre il file necessario per poter salvare i comandi inseriti dall'utente, con nome 'instruction'.

Dopo di che si richiede all'utente di inserire i vari comandi, fino a che una riga vuota non viene inserita, in tal caso si eseguono i comandi richiamando la funzione execCmdFromInstruction(). Per ogni comando inserito si eseguono le seguenti operazioni:

- conteggio dei caratteri che compongono l'istruzione completa salvandone il numero nella variabile 'c';

- il comando è poi salvato all'interno del file 'instruction', aprendolo con l'opzione 'a' che permette di porlo quindi alla fine del file, in questo modo:

[istruzione – argomenti – '\n'];

- con la funzione 'fseek()' si sposta l'indicatore di posizione del file, necessario per poter scrivere l'eventuale comando successivo.

- void execCmdFromInstruction(int n):

Si apre il file 'instruction' in lettura, dopo di che si legge ogni riga all'interno di esso, attraverso la funzione 'fgets' la quale immagazzina nel buffer 'command' i numeri di caratteri indicati dalla macro LENGTH, dallo stream 'newFile'. Attraverso una 'fork()' si crea un processo figlio che avrà il compito di eseguire il comando dentro 'commands' opportunamente modificato, per poi salvarlo nel file di output rispettivamente nominato con il numero del comando. Il padre in questo caso non attende il figlio per permettere nel modo più efficiente l'esecuzione parallela. Dopo aver letto tutti i comandi si chiude il file contenente tutte le istruzioni inserite dall'utente.

## **Motivazioni delle scelte di implementazione:**

Per facilitare la scelta dell'esecuzione abbiamo creato un menù che permette di sceglierne la modalità:

- 1) esecuzione sequenziale;
- 2) esecuzione parallela.

In entrambi i casi per poter leggere le istruzioni, si è usato il metodo 'eliminateEmptySpace()' che sostituisce gli spazi vuoti con il carattere '\0', necessario per poter memorizzare le componenti dell'istruzione. In questo modo si scandisce la riga salvandosi i primi n caratteri che fanno parte dell'istruzione e i rimanenti argomenti, separati anch'essi da uno '\0'.

Nel caso dell'esecuzione in parallelo, tutti i comandi inseriti dall'utente vengono memorizzati in un file creato appositamente che conterrà un comando con i suoi argomenti, per riga. Le istruzioni verranno poi lette ed eseguite da questo file riga per riga, creando un processo per ogni istruzione attraverso la 'fork()'.

Per eseguire i comandi si è utilizzata la funzione 'execvp' che permette di passare gli argomenti all'interno di un array di caratteri, poiché questi

sono variabili in numero. Inoltre questa funzione utilizza la variabile PATH di ambiente per eseguire il comando nella directory corrente.

## Screenshot:

```
Scegliere modalità di esecuzione
0 = Esecuzione sequenziale
1 = Esecuzione in parallelo
0
Inserisci un comando
ls -l
Eseguo figlio
Il processo figlio è terminato correttamente, status = 1
Inserisci un comando
cat /etc/pas
Eseguo figlio
Il processo figlio è terminato correttamente, status = 1
Inserisci un comando
cat /etc/passwd
Eseguo figlio
Il processo figlio è terminato correttamente, status = 1
Inserisci un comando

Programma terminato
federico@federico:~/Documenti/Esercizio2_FINALE$ ls
Config.h  main  Main.c  out.0  out.1  out.2
```

- Esecuzione di comandi in versione sequenziale: si inserisce un comando per volta, creando un processo figlio che si occuperà di eseguirlo, controllando che non ci siano stati errori nella sua terminazione.

```
GNU nano 2.2.6          File: out.1
cat: /etc/pas: File o directory non esistente
```

- Nel caso in cui si inserisca un comando inesistente, lo standard error viene scritto sul file associato al comando.

```

Scegliere modalità di esecuzione
0 = Esecuzione sequenziale
1 = Esecuzione in parallelo
1
Inserisci un comando
ls -l
done
Inserisci un comando
date
done
Inserisci un comando
cat /etc/passwd
done
Inserisci un comando

Programma terminato
federico@federico:~/Documenti/Esercizio2_FINALE$ ls
Config.h instruction main Main.c out.0 out.1 out.2

```

- Esecuzione di comandi in versione parallela: Ogni comando inserito in questo caso viene inserito all'interno del file "instruction", per permettere l'esecuzione in parallelo dei comandi appena inseriti.

```

GNU nano 2.2.6          File: instruction
ls -l
date
cat /etc/passwd

```

- Contenuto del file "instruction" al termine dell'inserimento dei comandi, in versione parallela.

### Istruzioni per compilare i file sorgenti:

Per quanto riguarda il main: `cc -o main Main.c`

Lanciare in esecuzione il programma: `./main`

## Esercizio 3: Scambio di messaggi

L'implementazione della soluzione all'esercizio, è composta da due file: Client.c e Server.c.

- *Client.c*

Il client svolge due funzioni, la connessione/disconnessione al server e l'invio di messaggi ad altri clienti.

Il cliente prima di poter inviare messaggi, deve connettersi al server attraverso il comando 1 inserito dall'utente, dopo di che può visualizzare i clienti connessi allo stesso server (attraverso il comando 2) ed eventualmente, scambiare messaggi attraverso il comando 3. Quando lo ritiene necessario, il client può disconnettersi per mezzo del comando 4 oppure terminare la sua esecuzione col comando 5 o con il segnale CTRL-C (gestito dal sistema attraverso il segnale SIGINT).

La richiesta di connessione invoca la funzione "firstConnection()", la quale verifica se il cliente è già connesso e in tal caso effettua una stampa, altrimenti richiama la funzione "executeCommand(int command)" che attraverso uno switch invoca le procedure che corrispondono al comando inserito.

### **Metodi principali:**

- void sendMessageToServer():

Utilizzata per connettersi al server, richiedere la lista dei clienti connessi al server, disconnettersi dal server (coi comandi 4, 5 oppure col segnale CTRL-C).

In questa funzione si costruisce il messaggio da inviare al server concatenando il comando con il pid del cliente, che viene salvato nel buffer 'message'. Attraverso il descrittore fd si prova ad aprire la unica pipe con nome 'ServerPipe', creata dal server, in scrittura.

Se la pipe è stata aperta con successo, viene scritto il messaggio nella pipe e chiuso il descrittore da parte del cliente.

Per evitare che fosse stampato prima il menù della lista dei clienti connessi al server, il client viene messo in attesa di 3 secondi.

- void sendMessageToClients():

Utilizzata per costruire il messaggio e inviarlo al server. Al suo interno si dichiara la variabile 'text' che conterrà il messaggio scritto dall'utente. Clients è un array di stringhe contenente i pid dei clienti destinatari. E' necessario inizializzare 'text' per annullare l'eventuale contenuto di un messaggio precedente. Una volta inserito il messaggio, viene richiamata la funzione getNumClients(char \*client[]), la quale inserisce i clienti destinatari all'interno dell'array client[i], allocata precedentemente attraverso una malloc, e restituisce il numero dei clienti inseriti che viene salvato nella variabile 'numClient'. Se 'numClient' è diverso da 1, ciò significa che ho almeno un cliente a cui mandare il messaggio, quindi apro la pipe con nome(ServerPipe) in scrittura, costruisco il messaggio calcolando le dimensioni. SIZEINIT corrisponde all'intero 5, che indica i caratteri sempre presenti all'interno del messaggio(il comando, i 3 spazi tra i componenti del messaggio e il carattere di fine stringa). Infine scrivo il messaggio nella pipe e chiudo il descrittore. Uno sleep di 1 secondo è necessario per gestire il segnale di client destinatario non connesso al server.

- void handleMessages(int signum):

Utilizzata per leggere il messaggio ricevuto dal server attraverso la pipe con nome associata al client. Inoltre ci fornisce la lista dei clienti connessi allo stesso server, necessaria per il comando 2. Il nome della pipe aperta è costruita attraverso la concatenazione del buffer 'str' che contiene il pid del cliente. Viene aperta la pipe appena costruita in lettura e grazie alla funzione 'readLine' si legge il messaggio ricevuto dal server. A fine procedura si chiude il descrittore e si effettua la unlink della pipe, per cancellarla dal sistema.

- void handleCloseClient(int signum):

Utilizzata per disconnettere e chiudere il client. Sono 3 i casi disponibili:

1) command = 5

2) CTRL-C da parte del client

3) Chiusura del server e conseguente chiusura del client

Una volta che uno di questi casi si verifica, viene mandato un messaggio al server che provvederà a disconnettere il client.

- *Server.c*

Il server è responsabile della ricezione e smistamento dei messaggi ai vari client destinatari. Nel 'main' viene creata la pipe con nome 'ServerPipe' in lettura, utilizzata dai client per inoltrare messaggi. Inoltre si inizializza il segnale di chiusura del server. Il server rimane sempre in ascolto di un eventuale messaggio da un client, quando ne riceve uno viene interpretato dalla funzione 'parseMessage', che gestirà i vari casi.

## **Metodi principali:**

- void insertElement(char \*str):

Utilizzata per costruire la lista dei clienti ogni volta che richiedono una connessione al server. Ogni client viene allocato dinamicamente attraverso la malloc e inserito in fondo alla lista, facendo puntare l'elemento precedente al nuovo elemento creato.

- void sendList(char \* str):

Utilizzata per inviare la lista dei clienti connessi al server attualmente, al cliente che la richiede. Si costruisce il nome della pipe concatenandola col pid del client richiedente. Per costruire la lista dei clienti che dovranno essere inviati nella pipe si richiama la funzione 'fillClientList'. Se la lista non è vuota, attraverso un while si popola l'array 'clientList' ricavando il pid da 'str' (che contiene il pid del client nella lista) e separandolo da uno spazio vuoto.

L'operazione si ripete finchè non si è scandita tutta la lista.

Successivamente attraverso la funzione kill si invidia il segnale SIGUSR1 per permettere al client di leggere la lista. Infine la pipe viene aperta in scrittura e una volta scritto il messaggio viene chiuso il descrittore.



- void readPid(char \*str, char \*myPid):

Si cerca la prima occorrenza di uno spazio con la funzione 'strchr', e poi lo spazio viene sostituito con uno '\0' e questa operazione per 3 volte, poiché il messaggio ricevuto è così strutturato:

['command', " ", 'pidMittente', " ", 'pidDestinatario', " ", 'messaggio', '\0'].

Una volta costruito il messaggio da inviare al pidDestinatario, si invia attraverso la funzione sendMessage.

- void sendMessage(char \*myPid, char \*destPid, char \*message);

Si effettua un controllo sul cliente destinatario, che deve risultare connesso, altrimenti si effettua un kill al pidMittente, con il segnale SIGUSR2, che provvederà ad avvertire con una stampa il mittente che il destinatario non è connesso. Viceversa, se è connesso viene creato il nome della pipe concatenandolo con il pidDestinatario e si invia il segnale SIGUSR1 per avvertire il cliente destinatario che gli verrà inviato un messaggio.

Viene infine aperta la pipe in scrittura e tramite una 'write' nella quale si scriverà il messaggio al suo interno. Infine viene chiuso il descrittore.

- void disconnect(char \*pid):

Funzione usata per rimuovere il cliente (il quale pid è specificato nell'argomento 'pid') dalla lista dei client connessi e quindi disconnetterlo. Infine si libera la memoria occupata attraverso la funzione 'free()'.

## **Motivazioni delle scelte di implementazione:**

Client: All'interno del metodo 'executeCommand' i primi due casi risultano analoghi, perché in entrambi il client deve inviare semplicemente una richiesta al server contenente il comando e il suo pid, quindi il case 1 non necessita il break. Sarà poi il server a gestire la richiesta in base al comando inserito.

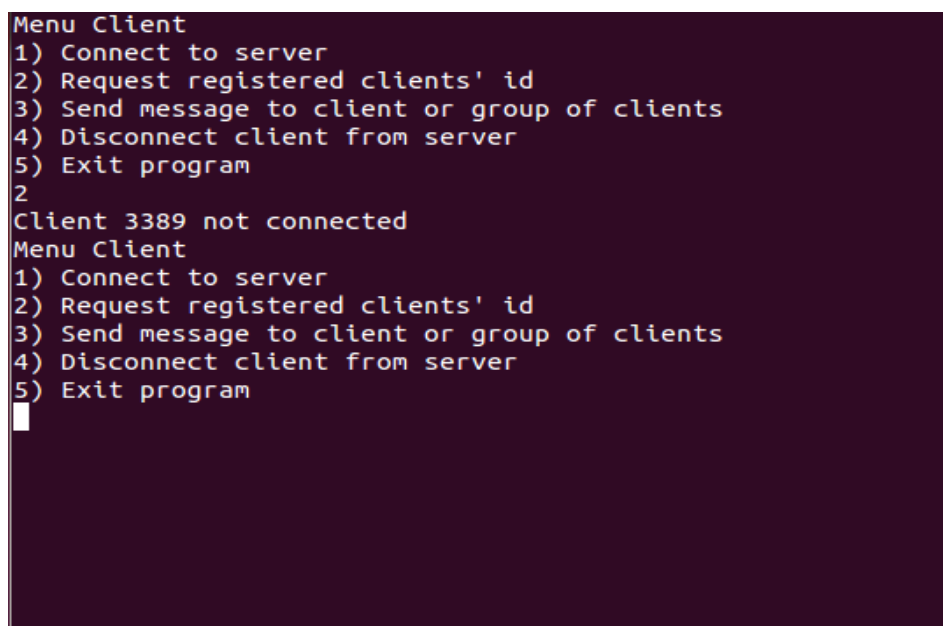
Per la gestione dei comandi, abbiamo realizzato un enum di nome "Commands" per poter rendere più chiaro e sicuro il codice, all'interno del Client.c. In questo modo invece di utilizzare gli interi da 1 a 5, si usano i nomi dei vari casi.

All'interno del metodo 'getNumClients', si alloca la i-esima posizione dell'array di stringhe client ad ogni ciclo del while, con una dimensione di 10. Inoltre nello scanf tramite la dicitura "%[^\n]s" si legge fino a '\n' e quindi l'utente può inserire il nuovo pid destinatario in una nuova riga. Se non intende più aggiungere client destinatari si deve inserire dalla tastiera il carattere '\n' (che corrisponde a invio), per poter inoltrare il messaggio.

Per la gestione del client connesso al server, abbiamo scelto di effettuare il controllo dell'eventuale connessione al server nel client stesso, così che se si prova ad inviare un messaggio quando non si è connessi il client, questo rifiuta l'operazione.

Server: La lista dei clienti connessi al server viene costruita attraverso la struttura dati 'Client' che è composta da due dati: 'id' che rappresenta il pid del cliente e 'nextClient' che indica l'elemento successivo nella lista.

## Screenshot:



```
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
2
Client 3389 not connected
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
█
```

- Le richieste da parte di un client non connesso, vengono rifiutate.

```

Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
1
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
2
RESPONSE:
3431 3640 3645
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program

```

- Richiesta della lista dei client connessi al server, da parte di un client: il server risponde inviando un segnale al client per notificarlo della risposta con la lista dei client.

```

3
Please insert a message:
Hello
Insert clients' id:
3431
3640

Send message to 3431...
Send message to 3640...
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of client
4) Disconnect client from server
5) Exit program

```

- Invio corretto da parte del client con pid 3645 a dei pid attualmente connessi al server. I client sono inseriti uno per volta.

```
1
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
RESPONSE:
Message from 3645: Hello
█
```

- Messaggio inoltrato da un client e ricevuto da parte di un client destinatario. Il server ne ha notificato la ricezione per leggerne il contenuto attraverso la pipe con nome creata appositamente.

```
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
3
Please insert a Message:
Hello from pid 3645
Insert clients' id:
1234

Send message to 1234...
Ops! Dest pid not connected at server
Menu Client
```

- Invio messaggio a pid non connesso al server. Il server notifica il client, che ha appena inviato il messaggio a un pid non attualmente connesso, con un segnale.

```
Menu Client
1) Connect to server
2) Request registered clients' id
3) Send message to client or group of clients
4) Disconnect client from server
5) Exit program
^Csignal 2: client closed
```

- Chiusura del programma dal lato client e conseguente disconnessione (nel caso il client è connesso) dal server che provvederà a eliminare il client appena chiuso dalla lista dei client attualmente connessi.

### **Istruzioni per compilare i file sorgenti:**

Per quanto riguarda il server: `cc -o server Server.c`

Lanciare in esecuzione il programma: `./server`

Per quanto riguarda il client: `cc -o client Client.c`

Lanciare in esecuzione il programma: `./client`