Progetto Del Corso di Sistemi Operativi

Il progetto ha lo scopo di creare un editor di testo da terminale, che implementi più comandi possibili per l'utilizzo dello stesso.
Il cuore dell'Editor di Testo è rappresentato dalla struct config, presente nel file *utilities.h*, così definita:

```
typedef struct config{
    int x, y;   /*Indice di Riga e di Colonna del Terminale*/
    int rx; /* Indice del campo di rendering, se non vi sono TAB rx == x, se ci sono rx > x*/
    int offsetRiga;     /*Tiene traccia della riga in cui sono */
    int offsetColonna;  /*Tiene traccia della colonna in cui sono. Rappresenterà l'indice dei caratteri*/
    int righe, colonne;
    int numRighe;
    EditorR* row;   /*Mi serve un puntatore ai dati di carattere da scrivere*/
    int sporco; /*Si occuperà di mostrare se il file è stato modificato*/
    char* nomeFile;
    char statusmsg[80];  /*Stringa che utilizzo per la ricerca bella barra di stato*/
    time_t statusmsg_time;  /*Timestamp per messaggio, in modo tale in poco tempo posso cancellarlo*/
    struct editorSyntax *syntax;    /*Contiene tutto ciò che mi serve per riconosce il tipo di file*/
    struct termios initialState;    // Salvo lo stato iniziale del terminale e tutti i suoi flag
}config;
```

Per l'implementazione dell'editor, si può suddividere il progetto il 5 macro sezioni:

1. Modifica del Terminale con funzioni che lo implementano

I files *termFunc.h* e *termFunc.c* contengono le funzioni che vengono utilizzate per settare determinati flag sul terminale.

Per prima cosa si scrive una funzione chiamata *"abilitaRawMode"* per uscire dalla classica modalità "cooked mode" del terminale ed entrare in modalità "Raw Mode". Occorre quindi:

- Disabilitare tutti i tasti *ctrl* che utilizza di default
- Disabilitare gli accapo e la funzionalità di elaborazione dell'output (comprese le printf)
- Disabilitare la gestione dei segnali
- Settare il numero minimo di byte prima che la *read* ritorni e il tempo massimo di attesa
- Eliminare qualsiasi input non letto

Ovviamente, una volta terminata la scrittura nell'Editor, servirà la funzione *disabilitaRawMode* per riassegnare tutti gli attributi che originariamente possedeva il terminale.
Il *main*, una volta abilitata la modalità *RawMode*, dovrà continuamente svuotare lo schermo e processare ogni singolo char messo in input sul terminale. Per fare ciò si utilizzano le *"Sequenze di Escape"*, le quali iniziano sempre con un carattere escape `\x1b`, seguito sempre da `[`. In questo modo si comunica al terminale di spostare il cursore, cambiare il colore del font, cancellare parti dello schermo,...

> *Molto Utile è stata la guida sul sito* ***https://vt100.net/docs/vt100-ug/chapter3.html#ED****, che mostra il significato di ogni singola sequenza di escape.*

Per svuotare lo schermo, occorre scrivere sullo standard output 4 byte:

- Il Primo `\x1b` (27 in decimale) è il carattere di escape
- Il Secondo `[` è un altro carattere di escape
- Il Terzo `?25` indica che si vuole cancellare l'intero schermo
- Il Quarto `J` indica che si vuole eliminare <*esc*>

Dopo aver impostato tutto l'occorrente, ci serviremo di una struct `StringBuffer` e delle relative funzioni associate ad essa:

```
struct StringBuffer {
    char *b;
    int len;
};
```

La struct servirà a creare una *write* dinamica, in cui scrivere, tramite la funzione *memcpy*, l'input inserito nel terminale nel `char* b`, riallocando i byte necessari per la stringa e

aggiornando anche la sua rispettiva lunghezza. Per ottenere il risultato si usano due funzioni:

- Il costruttore:

```
void sbAppend(struct StringBuffer *sb, const char *s, int len)
```

- Il distruttore:

```
void sbFree(struct StringBuffer *sb)
```

La funzione *sbAppend* verrà utilizzata anche per svuotare lo schermo, per nascondere il cursore `"sbAppend(&sb, "\x1b[?25l", 6);"` e successivamente per riposizionarlo in alto a sinistra nel terminale.
A questo punto si otterrà l'effettiva dimensione del terminale (larghezza e altezza), per far si che:

- Il file in ingresso sia perfettamente centrato nel terminale;
- Mostrare un messaggio di benvenuto nel caso in cui non passi alcun file;
- Scrivere due righe sottostanti per mostrare sia la lista dei comandi implementati che per abilitare la ricerca nel file o l'apertura di un altro nella stessa finestra;
  void disegnaRighe(struct StringBuffer * sb)

Per conoscere l'effettiva dimensione del file si usa la `struct winsize` che contiene il numero di righe e di colonne e i pixel in orizzontale e verticale. Ora è il momento di posizionare il cursore sullo schermo, tramite la comoda funzione.

```
int posizioneCursore(int* righe, int* colonne)
```

Tale funzione funge da appoggio per posizionare il cursore in posizione (0, 0).
Successivamente si deve verificare che il cursore si sposti all'esterno della finestra (altrimenti potrebbe non funzionare lo scroll verticale) per posizionarlo poi all'interno della finestra "visibile".
È il momento di inizializzare e disegnare la *status bar*:

```
void statusBarInit(struct StringBuffer *sb)
```

Innanzitutto occorre invertire il colore dell'ultima riga del terminale per creare contrasto e renderla visibile `sbAppend(sb, "\x1b[7m", 4)`. La status bar dovrà mostrare il nome del file, il tipo (se conosciuto dall'editor), se è stato modificato, il numero di righe del file, l'indice di riga in cui ci si trova e una seconda barra che servirà per la ricerca del testo, per l'apertura

di un nuovo file e per mostrare un messaggio contenente i comandi implementati che verrà visualizzato solo per 5 secondi.

2. Funzioni di Utility per Editor

L'Editor utilizza principalmente una struct, contenente tutto ciò che necessario per la gestione dello stesso.

```
typedef struct EditorR{
    int index;  // Per gestire i commenti /* */ su più linee gestendo l'indice all'interno del file
    int size;    /*Conterrà la size che occuperanno le stringhe*/
    int effSize;    /*Gestisco le effettive tabulazioni, mostrando gli spazi come dico io e non...*/
    char* chars;
    char* effRow;   /*... come fa di default il terminale, altrimenti un TAB occuperebbe 7 caratteri circa*/
    unsigned char *color;   /*conterrà valori da 0 a 255 e vedrà se ogn carattere matcherà con un stringa definita da me, per l'highlight*/
    int is_comment; /*Variabile boolean per gestione commento*/
} EditorR;
```

Prima di tutto occorre inizializzare la struct config per resettare ogni dato presente in essa e posizionare il cursore all'inizio del file. Inizialmente si gestisce il posizionamento del cursore attraverso la combinazione di tasti *W-A-S-D* ma successivamente tale implementazione verrà sostituita con la seguente struttura:

```
enum editorKey {
    BACKSPACE = 127,    /*ASCII == 127*/
    FRECCIA_SINISTRA = 1000,    /* Dalla prossima chiave in poi i numeri incrementeranno di uno*/
    FRECCIA_DESTRA,
    FRECCIA_SU,
    FRECCIA_GIU,
    CANC,   /*<esc> [3 ~*/
    HOME,   /*Fn + ←*/
    END,    /*Fn + →*/
    PAGINA_SU,
    PAGINA_GIU
};
```

In questo modo basterà concatenare la sequenza di escape `\x1b` con char che vanno da '1' a '8' per gestire i tasti *HOME, END, CANC PAGE-UP, PAGE-DOWN* e da 'A' a 'F' per gestire i tasti *FRECCIA-SU, FRECCIA-GIU, FRECCIA-SINISTRA, FRECCIA-DESTRA* tramite un semplicissimo switch nella funzione:

```
int letturaPerpetua()
```

Tale funzione lavorerà insieme a...

```
void processaChar()
```

...la quale si occuperà di gestire ogni carattere passato controllando se questo rappresenta un carattere speciale, se viene premuto *CTRL* o se semplicemente dovrà scrivere. Per gestire i tasti *CTRL*, si usa la seguente macro: `#define CTRL_KEY(k) ((k) & 0x1f)`.

Una volta gestiti tutti questi casi, ci occuperemo dell'**apertura di un file** tramite la funzione

```
void openFile(char* nomeFile);
```

Appena aperto il file, si dovrà liberare la memoria allocata per il `char* nomeFile` presente nella struct principale dell'Editor; successivamente la si riallocherà con il nome del file appena aperto, tramite la funzione *strdup* che gestirà automaticamente la memoria che occorre. Il file sarà aperto in lettura, come fanno la maggior parte degli editor, e il salvataggio su disco sarà gestito successivamente da un' altra funzione. Per mostrare il contenuto del file sullo schermo si dovrà scandire ogni sua linea, tramite la funzione `ssize_t getline(char ** restrict linep, size_t * restrict linecapp, FILE * restrict stream)`, gratuitamente offerta da <*stdio.h*> ,ed "iniettare" tante righe sul terminale quante sono quelle scandite dal file.
Per **modificare il contenuto del file** si utilizzano principalmente le seguenti funzioni:

```
void inserisciRiga(int at, char *s, size_t len);
void aggiornaRiga(EditorR* row);
int xToRx(EditorR* row, int x);
void scriviInRiga(EditorR *row, int at, int c);
void inserisciChar(int c);
char *rowToString(int *buflen);
```

- La prima funzione **gestisce l'allocazione della memoria** delle stringhe presenti su una riga e dei rispettivi indici di riga, per processare ogni `char *` presente nell'Editor incrementando il numero di righe e la sua lunghezza, se presente un carattere di tabulazione;
- La seconda funzione è ausiliaria, utilizzata per l' **aggiornamento degli spazi su una riga**, riempiendo il contenuto della stringa e copiando ogni carattere per reindirizzarlo non appena modificato. Per fare ciò, occorre scorrere tutti i caratteri della riga per contare quanta memoria allocare per gli spazi e per le tabulazioni. Dato che ogni carattere di tabulazione occupa 8 char, per ogni riga occorre allocare `row->size + tabs*(STOP_TAB -1)+1`, in modo tale che ogni carattere letto venga copiato interamente nella *struct EditorR*.

- La terza funzione è anch'essa di appoggio e servirà per aggiornare il valore *x* della *struct config* in un valore *rx* , per **calcolare** l'effettivo **offset di ogni tab** e tramutarlo in un vero e proprio spazio. Per fare ciò occorre sapere quante colonne sono alla destra del *TAB* e quante ne sono a sinistra (*8 - 1*); quindi si farà un controllo in un ciclo *for* incrementando il valore di *rx* per cercare il successivo *TAB*.
- Dalla quarta funzione in poi ci si occuperà dell'effettiva **scrittura di caratteri su** una **riga**, dato che precedentemente sono state gestite le tabulazioni e l'inserimento delle righe sul terminale. Questa fungerà da funzione ausiliaria per la prossima funzione e si userà per l'effettiva scrittura di caratteri nella struct dell'Editor.
- La quinta funzione **inserisce** le **stringhe** precedentemente lette **sulle righe del terminale**. Per fare ciò si verifica dapprima la posizione del cursore sullo schermo e se quest'ultimo si trova alla fine del file si dovrà aggiungere una nuova riga per dare la possibilità di scrivere oltre la fine del file. Si sposterà successivamente la posizione del cursore in avanti in modo tale che il prossimo carattere inserito capiti subito dopo il carattere precedentemente aggiunto.
- La sesta funzione invece **incapsulerà una riga** presente nel terminale **convertendola in una** vera e propria **stringa**. Un primo ciclo *for* sommerà le lunghezze di ogni riga di testo salvando il suo valore in una variabile in modo tale che si possa allocare l'effettiva memoria necessaria per la stringa. Servirà anche un secondo ciclo *for* per copiare il contenuto di ogni riga all'interno del buffer precedentemente allocato, aggiungendo un ulteriore carattere alla fine di ogni riga.

A questo punto si **salverà** il contenuto del file **sul disco** tramite la funzione

```
void salvaSuDisco();
```

Anche il salvataggio sarà dinamico, infatti prima di tutto si verifica se il file è esistente (se si conoscerà il suo nome) e dove salvarlo, altrimenti si dovrà far immettere il nome per il suo successivo salvataggio e si dovrà anche gestire l'interruzione di salvataggio in caso di ripensamento dall'utente. Si aprirà successivamente il file in modalità lettura e scrittura se esiste (altrimenti verrà creato) tramite il flag `O_RDWR | O_CREAT`. Si imposterà quindi la dimensione effettiva del file uguale alla lunghezza specificata dalla funzione *rowToString* e con la funzione `int ftruncate(int fildes, off_t length)` di <*unistd.h*> si imposterà una dimensione statica al file; se è più corto inserisce *0* di padding, se più lungo verrà tagliato fino alla lunghezza specificata, non troncandolo completamente. Ora si può usare la `write` per salvare il file sul disco mostrando anche all'utente quanti byte sono stati scritti sul disco.

Finora l'Editor sarà in grado solamente di scrivere il testo, gestendo le tabulazioni e l'inserimento di caratteri concatenandoli tra loro. A questo punto occorre gestire la **cancellazione del testo**, utilizzando le seguenti funzioni:

```
void cancellaCharInRiga(EditorR* row, int at);
void cancellaChar();
void liberaRiga(EditorR* row);
void cancellaRiga(int at);
void appendiStringaInRiga(EditorR* row, char* s, size_t len);
```

Le precedenti funzioni dovranno gestire sia il tasto *CANC* che il tasto *DEL*. Per fare ciò una funzione "mangerà" il testo dalla destra del prossimo char che si trova in corrispondenza del cursore, attraverso *memmove*, e un'altra consumerà il carattere da sinistra, utilizzando la stessa funzione ma giostrando gli indici della stringa in modo accurato. Occorre trattare anche il caso in cui un carattere si trovi o in posizione *(0, 0)* dello schermo o in posizione *(0, n)*. Nel primo caso la cancellazione non dovrà essere effettuata e il cursore dovrà essere ri-posizonato al suo posto; nel secondo caso invece una funzione ausiliaria concatenerà le due righe unendole, cancellerà la riga sottostante e aggiornerà gli indici di riga.
L'ultima funzione invece si occuperà dell'aggiunta della stringa modificata nell'opportuno campo della *struct conf*.

Per il **tasto invio** invece basterà intercettare l'inserimento del carattere `\r` e `\n` tramite la funzione

```
void inserisciNewLine()
```

Anche in questo caso se ci si trova all'inizio del file si aggiunge una riga al campo *y* della struct; altrimenti si splitta la riga in 2, inserendo la prima con i caratteri che si trovano sulla sinistra e la seconda con quelli che sono a destra. A questo punto si sposta il cursore in posizione *(0, n)* (con *n* = inizio riga successiva) e si aggiorna il contenuto della riga troncando il contenuto della riga corrente, poiché la *realloc* potrebbe invalidare il puntatore che si sta utilizzando. Si tronca il contenuto della riga corrente e si aggiorna usando la funzione

```
void aggiornaRiga(EditorR* row)
```

3. Funzioni di utility ausiliarie: Ricerca del Testo e Apertura file da Prompt

Per gestire funzioni ausiliarie quali la ricerca nel testo e l'apertura di un nuovo file nella schermata principale dell'Editor, si considerano come appoggio la funzione

```
char *promptComando(char *prompt, void (*callback)(char *, int))
```

Tale funzione si occuperà della scrittura sul "prompt di comando" dell'Editor, ovvero sull'ultima riga del terminale. La funzione prende in input una stringa e un puntatore a funzione che a sua volta ritorna void e prende in input un `char*` e un `int`. Dare in input un puntatore a funzione da' la possibilità di gestire sia la ricerca, in modo da potergli passare la stringa da cercare e la sua *size*, che l'apertura di un file, passandogli come valore al puntatore a funzione *NULL*.
Tale funzione memorizzerà l'input inserito dall'utente in un buffer appositamente allocato e attraverso un ciclo while si occuperà di verificare:

- La cancellazione del testo
- La gestione del tasto invio
- Il riconoscimento dei caratteri, in modo tale che nel prompt possa inserire solo caratteri *ASCII* riconoscibili dal terminale

Per la **ricerca del testo** si deve immettere al puntatore a funzione di *promptComando* il metodo

```
void cercaTestoCallback(char *toFind, int key)
```

Tale funzione si occuperà di gestire la ricerca direzionale (tramite i tasti freccia) settando opportunamente due variabili intere che memorizzeranno l'indice di riga su cui si trova l'ultimo risultato trovato (se non esiste *-1*), e la direzione della ricerca (*1* per cercare in avanti e *-1* per cercare indietro) tramite un ciclo *for* che servirà a scandire l'intero contenuto del file.

- Se si preme invio o un qualsiasi altro carattere di escape basta uscire dalla ricerca
- Altrimenti:
    - Se si preme *FRECCIA DESTRA* o *FRECCIA GIÙ* ci si sposterà in avanti
    - Se si preme *FRECCIA SINISTRA* o *FRECCIA SU* ci si sposterà indietro

Per verificare se la stringa inserita è contenuta in una riga si utilizza la comodissima funzione `char * strstr(const char *haystack, const char *needle);` gratuitamente offerta da <*string.h*>. Se si trova la stringa in questione, verrà colorata di blu, altrimenti il suo colore sarà quello di default.
Tale funzione di callback verrà presa in input dalla funzione *promptComando* che però verrà invocata dalla funzione

```
void cercaTesto();
```

Quest'ultima si occuperà solamente di salvare la posizione che aveva il cursore prima della ricerca e di ripristinarlo a ricerca terminata.

Per **aprire un nuovo file** nella schermata dell'Editor si verifica se il file esiste, tramite la system call `int access(const char *path, int mode);` con il flag di modalità settato ad `F_OK`

- Se esiste, verrà inizializzato l'Editor aprendo il file in sola lettura
- Altrimenti si prenderà in input il file, salvando opportunamente il suo nome e lo si aprirà in scrittura

Tutto ciò verrà verificato attraverso la funzione

```
void openNewFileFromPrompt()
```

4. Funzioni per Riconoscimento del Tipo di File e Colorazione di Sintassi

Riconoscere il tipo di **file in input** è abbastanza semplice, basterà solamente vedere cosa contiene `argv[1]` tramite la funzione

```
void selezionaSintassiDaColorare();
```

Se `argv[1]`:

- È *NULL*, si esce
- Altrimenti si ritorna salvando il puntatore all'ultima occorrenza dei caratteri nella stringa, ovvero l'estensione, tramite la funzione *strrchr* di <*string.h*>. Si verifica se questa meccia con la mia struct e si procede con la colorazione del testo.

```
struct editorSyntax HLDB[] = {
    {
            "c",
            ESTENSIONI_C,
            PAROLE_C,
            "//", "/*", "*/",
            COLORA_NUMERI | COLORA_STRINGHE
    },
};
```

Per **impostare** determinati **colori** in base al tipo di file verranno utilizzate le seguenti funzioni:

```
void aggiornaSintassi(EditorR *row);
int daSintassiAColore(int color);
```

- La prima funzione prenderà un' intera riga del file e per ogni stringa presente in essa riallocherà la memoria necessaria, dato che colorare il testo incrementerà la dimensione della stringa in questione. Tale funzione si occuperà anche di verificare se nel file sono presenti i caratteri `//` e `/*` o `*/`, per colorare di *cyan* i commenti trovati.
- La seconda funzione invece prende in input un intero e restituisce un altro intero corrispondente al carattere *ANSI* da abbinare alla sequenza di escape, che permetterà di colorare il testo

5. Gestione commenti singoli e multilinea

Gestire i commenti singoli è molto semplice, infatti si crea una funzione 'booleana' che verifica, tramite la funzione *strchr*, se il carattere preso in input è considerato un carattere di separazione (`.()+-/*=~%<>[];`).

```
int is_separator(int c);
```

Anche in questo caso si considera principale la funzione *void aggiornaSintassi(EditorR row)* per verificare se nel testo sono presenti caratteri considerati commenti in file *.c*. A questo punto serviranno 3 stringhe, che conterranno rispettivamente la stringa contenuta in un commento su una singola linea, la stringa di inizio del commento multilinea e la fine, e 3 variabili intere per contenere la loro lunghezza. Successivamente itero tutti i caratteri di ogni riga e se si trova una stringa che si riconosce si colorerà, solo se questa non è contenuta in un commento. Per tenere traccia nella scansione se ci si trova all'interno di un commento, si utilizza un'espressione ternaria `int in_comment = (row->index > 0 &&`

`Editor.row[row->index - 1].is_comment);` che varrà 1 (True) se la riga precedente è evidenziata, 0 (False) altrimenti.
Se si è all'interno di un commento multilinea, un ciclo *while* verificherà che:

- Le tre variabili stringhe non sono nulle e
    - Se `in_comment` vale 1, l'indice di riga in cui ci si trova sarà settato con la macro `COMMENTO_MULTILINEA`
    - Se `in_comment` vale 0, si imposta ad 1 e si continua a "mangiare" il contenuto della riga

Se si è all'interno di una stringa e il carattere corrispondente è '//' e c'è almeno un altro carattere all'interno di quella riga, tale carattere verrà evidenziato.
Se si trovano anche numeri decimali si colorano di rosso e ricorsivamente si invoca la funzione all'indice di riga successivo per aggiornare la sintassi di tutte le righe che seguono la riga corrente.

Infine, come richiesto espressamente dal professore, si gestisce il **lock di un file**. Appena invocato, il main farà in modo che l'apertura di un file sia bloccante, settando opportuni *flags* che si occuperanno di incapsulare il *file descriptor* . Per avere il "rock" esclusivo su un file si utilizzano la System Call `int fcntl(int fd, int cmd, ... arg e` e la `struct flock`.
La prima prende in input il descrittore di un file aperto e un valore che indica quale operazione deve essere eseguita (`F_SETLK` nel mio caso). Tale chiamata a sistema consente a uno ed un solo processo di inserire un blocco in lettura o in scrittura solo stesso file.
La *struct flock* così composta...

```
struct flock {
    ...
    short l_type;    ---> Tipo di lock (F_RDLCK, F_WRLCK, F_UNLCK)
    short l_whence;  ---> Come interpretare l_start (SEEK_SET, SEEK_CUR,
SEEK_END)
    off_t l_start;   ---> Offset di partenza per il lock
    off_t l_len;     ---> Numero di byte da lockare
    pid_t l_pid;     ---> Pid del processo bloccante
    ...
};
```

... permette di settare il desiderato blocco sul file. Infatti, il pid che imposterà il campo `l_type` con `F_WRLCK` ed il campo `l_whence` con `SEEK_SET` otterrà un blocco in scrittura dall'inizio del file preso in input. Se un altro processo contiene un blocco che impedisce l'acquisizione di altro, esso rimarrà in attesa affinché *fcntl* rilascerà la risorsa.
Il tutto si gestisce tramite la funzione...

```
int lockfile(const char *const filepath, int *const fdptr)
```

... che prende in input il nome di un file ed un descrittore che inizialmente si imposta a *-1* per renderlo non valido. Successivamente si utilizza la *fopen* in modalità *append*, si chiudono i descrittori `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO` e verrà restituito il risultato della *fcntl*. Infine in un ciclo *while* presente nel *main* si controllerà il valore di ritorno della precedente funzione.

- Se vale *0* la status bar mostrerà il classico messaggio di aiuto;
- Altrimenti l'utente verrà avvisato che lo stesso file è aperto da un altro processo, *errno* conterrà il rispettivo valore di errore e si continuerà la routine del processo *main* svuotando lo schermo e processando ogni *char* inserito sul terminale.

> ***Link Utili**:*
>
> - Tabella colori *ANSI*: **https://en.wikipedia.org/wiki/ANSI*escape*code#Colors** e **https://i.stack.imgur.com/7H7H9.png**
> - Guida VT100 e sequenze di escape: **https://vt100.net/docs/vt100-ug/chapter3.html**
> - Caratteri Speciali Prompt: **https://ss64.com/osx/syntax-prompt.html**