Documentazione

Implementare una programma che riceva in input, tramite argv[], il nome di un file F ed N stringhe S_1 .. S_N (con N maggiore o uguale ad 1).

Per ogni stringa S_i dovra' essere attivato un nuovo thread T_i, che fungerà da gestore della stringa S_i.

Il main thread dovrà leggere indefinitamente stringhe dallo standard-input.

Ogni nuova stringa letta dovrà essere comunicata a tutti i thread T_1 .. T_N tramite un buffer condiviso, e ciascun thread T_i dovrà verificare se tale stringa sia uguale alla stringa S_i da lui gestita. In caso positivo, ogni carattere della stringa immessa dovrà essere sostituito dal carattere '*'. Dopo che i thread T_1 .. T_N hanno analizzato la stringa, ed eventualmente questa sia stata modificata, il main thread dovrà scrivere tale stringa (modificata o no) su una nuova linea del file F.

In altre parole, la sequenza di stringhe provenienti dallo standard-input dovrà essere riportata sul file F in una forma *"epurata"* delle stringhe S1 .. SN, che verranno sostituite da stringhe della stessa lunghezza costituite esclusivamente da sequenze del carattere '*'.

Inoltre, qualora già esistente, il file F dovrà essere troncato (o rigenerato) all'atto del lancio dell'applicazione.

L'applicazione dovrà gestire il segnale *SIGINT* in modo tale che quando il processo venga colpito esso dovrà riversare su standard-output il contenuto corrente del file F.

OUTPUT

Per facilitare la comprensione del software di seguito viene riportato un esempio di output su sistemi UNIX.

```
gcc file.c -lpthread
./a.out file ciao1 ciao2 ciao3
```

Come è facile notare l'applicazione prevede che vengano passati come parametri due tipologie di argomenti:

- **Il nome di un file, in questo caso è il nome "file" stesso;**
- **Un insieme di stringhe la cui cardinalità è N, in questo caso N = 3.**

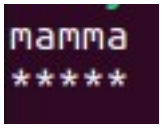
In totale vi sono 5 argomenti: ad esclusione dei parametri appena discussi viene aggiunto il nome del programma appena compilato che per default è a.out.

Passati questi argomenti come bisogna procedere?

Vogliamo realizzare un software di questa tipologia:

```
Numero di argomenti passati = 5
Numero di stringhe effettivamente passate = 4

Inserisci la stringa >> mamma

-------------------------------------------------------------------------------
La stringa che gestisce il thread corrente è : file [*]
Confronto tra la stringa acquisita  mamma  e la stringa che gestisce  file  [*]
Dimensione della stringa acquisita = 5
 Dimensione della stringa gestita = 4
Stringhe non uguali. (!=)
Buffer condiviso = mamma
-------------------------------------------------------------------------------

Inserisci la stringa >> ciao1

-------------------------------------------------------------------------------
La stringa che gestisce il thread corrente è : ciao1 [*]
Confronto tra la stringa acquisita  ciao1  e la stringa che gestisce  ciao1  [*]
Dimensione della stringa acquisita = 5
 Dimensione della stringa gestita = 5
Stringa acquisita e stringa gestita sono uguali. Check OK [*]
La stringa acquisita viene sostituita nel seguente modo : *****
Buffer condiviso = *****
-------------------------------------------------------------------------------
```

Visualizziamo il file:



***Nota**: come è facile notare il file è in una versione "epurata" perché la stringa che per prima è stata inserita (mamma) non è uguale alla stringa che gestisce il thread corrente, che in questo caso è proprio il nome del file (file), pertanto il check è negativo e ciò comporta semplicemente la scrittura sul file della stringa inserita senza opportune modifiche.*
Diversamente accade nell'inserimento della seconda stringa (ciao1): il thread corrente ora si ritrova a dover gestire la stringa "ciao1" che, guarda caso, è uguale alla stringa digitata da tastiera e pertanto il check tra la stringa acquisita e la stringa gestita è positivo e ciò comporta la scrittura della stringa su una nuova linea del file epurato sostituita da "".*

Per la creazione del file che per semplicità si chiama "epurata" viene utilizzato un descrittore.

```
file_descriptor = open(nome_file, O_RDWR|O_CREAT|O_TRUNC, 0666);
file = fdopen(file_descriptor, "w+");
```

La variabile file è un tipo FILE* (dichiarato globalmente) che servirà come primo parametro nella funzione fprintf per effettuare la scrittura (ciascuna in una nuova linea) della stringa gestita, indipendentemente se essa sia epurata o meno.
Per completezza viene mostrato il nome del file dichiarato globalmente:

```
char *nome_file="epurata";
```

Ora il file verrà creato ed esisterà.

Prima di immergere il lettore nella struttura vera del codice è opportuno richiamare alcune tecniche di programmazione base sulla struttura dei puntatori.

```
int main(int argc, char** argv)
{
        return 0;
}
```

Soffermiamoci sul secondo parametro argv.
Esso rappresenta la lista vera degli argomenti passati dall'utente durante la sua esecuzione e permette al programmatore di recuperare questi parametri effettuando semplicemente un'assegnazione di questo tipo:

```
file_names = argv;
```

Dove file_names è dichiarato globalmente nel seguente modo:

```
char **file_names;
```

In questo modo il programmatore è in grado di accedere a grana fine all'argomento I-esimo che l'utente ha inserito.

```
file_names[0] = a.out
file_names[1] = file
file_names[2] = ciao1
file_names[3] = ciao2
file_names[4] = ciao3
```

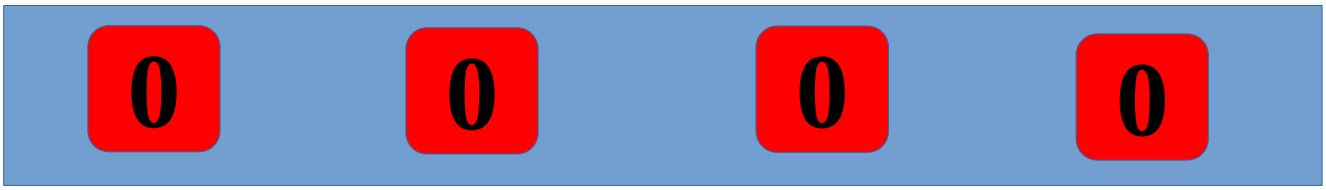
Dopo questa lunga premessa ed effettuati vari controlli di natura banale si giunge alla porzione viva del codice.

La struttura presenta la gestione di due semafori: un semaforo gestito dal main thread e un array semaforico che garantirà l'accesso atomico ad uno solo degli N threads che gestiranno la stringa I-esima e che effettueranno il confronto con la stringa acquisita dal main thread.

```
array_semaforico = semget(IPC_PRIVATE,numero_di_stringhe_effettivamente_passate, IPC_CREAT|IPC_EXCL|0666);
semaforo_main = semget(IPC_PRIVATE, 1, IPC_CREAT|IPC_EXCL|0666);
```

Queste due istruzioni servono a dichiarare le nostre strutture semaforiche IPC.
L'array semaforico è un vettore di N elementi, in questo caso N = 4;
Il semaforo del main è un singolo distributore di gettoni.
Di seguito l'impostazione dei valori.

semaforo_main



array_semaforico.

In questo modo l'unico che può essere eseguito in un primo istante di tempo è il main thread e pertanto tutti i threads rimarranno bloccati in attesa di uno sblocco da parte di quest'ultimo.

Il primo a partire è il main thread.
Esso troverà un gettone disponibile dal suo distributore e quindi verrà schedulato in CPU: in questo modo il semaforo del main esaurirà il suo unico gettone e si troverà azzerato dopo questa schedulazione.

semaforo_main

Il main thread è in esecuzione.

```
printf("Inserisci la stringa >> ");
if(fgets(buffer_condiviso, 1024, stdin))
{
  buffer_condiviso[strcspn(buffer_condiviso, "\n")] = 0;
}
```

In un while(1) richiederà continuamente in maniera indefinita l'acquisizione di una stringa.
Questa stringa "dovrà" necessariamente essere resa disponibile al thread che effettuerà il check dell'uguaglianza tra la stringa appena acquisita e la stringa da lui gestita, e per rendere disponibile la stringa appena acquisita al thread I-esimo occorre l'ausilio di un buffer_condiviso dichiarato globalmente in modo tale da rendere visibile la stringa anche nello spazio di indirizzamento del thread.
Pertanto la stringa appena acquisita viene memorizzata all'interno del buffer condiviso.
Dopo aver effettuato queste operazioni bisogna sbloccare il thread 0-esimo, ossia il primo thread che deve partire, e per effettuare questo sblocco bisogna impostare ad 1 il valore del primo elemento dell'array semaforico.

Parte il thread numero 0.
Ricordiamoci che è il main thread colui che deve effettuare la scrittura sul file dopo il check (positivo o negativo che sia) effettuato dal thread 0, pertanto appena parte il thread numero zero occorre fermare momentaneamente l'esecuzione del main thread.
A questo punto appena il thread numero 0 parte consuma il suo distributore.

A questo punto il programmatore può sbizzarrirsi come meglio crede per effettuare questo controllo.
Nell'esempio ho creato una funzione "fai da te ;)" nominata *check_equal* con la seguente firma:

```
int check_equal(char **,int);
```

Il primo parametro rappresenta il nome della stringa gestita mentre il secondo parametro il TID del thread – è fondamentale riconoscere quale thread sta gestendo quale stringa – e il valore di ritorno è un intero.
La funzione ritorna il valore 1 se la striga gestita è uguale alla stringa acquisita nel buffer condiviso, altrimenti -1.

A questo punto il thread zeresimo sblocca il main.
Il cuore del codice è tutto in questa seguente fase.
Il main viene sbloccato assegnando al semaforo del main il valore 2.

In questo modo il main thread può consumare il suo gettone.
Nota che questa volta il valore 2 è fondamentale.

Se avessimo impostato un valore pari ad 1 il main thread non sarebbe riuscito a sbloccarsi al secondo tentativo del while(1), poiché con un valore impostato ad 1 sarebbe riuscito solamente a scrivere i dati nel file ma non a ricominciare a richiedere l'acquisizione di una stringa.

Quindi il main thread usa il suo primo gettone per effettuare la scrittura della stringa nel file modificando il distributore nel seguente modo:

Dopo aver scritto nel file il main thread riprende il suo normale ciclo e avendo un gettone disponibile può proseguire con la sua esecuzione richiedendo la stringa da passare al thread successivo, il thread 1.
Una volta registrata la stringa nel buffer condiviso lo stato semaforico del main sarà nuovamente il seguente:

Verrà sbloccato il thread 1 e si metterà in attesa di uno sblocco (il main thread).

E ciò che succederà sarà esattamente ciò di cui abbiamo appena discusso.

Sin al momento in cui verrà sbloccato l'ultimo thread:

Finita la scansione di tutto l'array semaforico si ricomincerà dal primo elemento.

E così via!

Appena verrà premuto il segnale CTRL + C bisognerà portare il contenuto del file in output.
Questo lo si può semplicemente implementare tramite l'ausilio di una signal:

signal(SIGINT, sprint);

E la funzione che gestisce il segnale è la seguente:

```c
void sprint(int unused)
{
    sprintf(buffer_command, "cat %s", nome_file);
    printf("\n ---------\n");
    printf("\t \t |");
    printf("\n");
    system(buffer_command);
    printf("\n");
    printf("\t \t |");
    printf("\n ---------\n");
}
```

Documentazione didattica a cura di Simone Remoli.