Progetto

Osservazioni preliminari

Il progetto deve essere svolto da ognuno di voi individualmente: la presenza di porzioni di codice con alta similarità comporta l'annullamento della consegna e l'assegnazione di un diverso progetto.

Il corso ha anche l'obiettivo di mettervi in grado di leggere autonomamente la documentazione quindi per il progetto vi è richiesto di usare funzioni che non abbiamo visto in dettaglio a lezione. Lo sviluppo va fatto su macchine Linux: MacOS non supporta ad esempio i semafori unnamed, e Windows è ancora meno compatibile. La correzione avviene sulla macchina `laboratorio2.di.unipi`

Dato che parte della correzione viene fatta in maniera automatizzata dovete seguire le indicazioni alla lettera per quanto riguarda il nome dei file eseguibili. Leggete con attenzione la parte sul test di funzionamento da effettuare prima della consegna: se non viene superato quello la valutazione è insufficiente e non siete ammessi all'orale.

E' vostro compito progettare i dettagli del protocollo di comunicazione: ad esempio quando si parla di inviare una sequenza di byte è solitamente necessario prima inviare la lunghezza della sequenza. Potete assumere che al massimo la singola sequenza sia lunga al più 2048 bytes e che di conseguenza per indicare le lunghezze siano necessari sufficienti 2 byte.

La chiarezza del codice è più importante dell'efficienza, ma ha un grande impatto sulla valutazione l'efficenza in termini di concorrenza: ad esempio l'uso di mutex in situazioni in cui non sono necessari sarà penalizzato perché, quando possibile, dovete sempre sfruttare al meglio la disponbilità di thread multipli.

Fanno parte del testo dell'esercizio anche tutte le precisazioni fornite sul Forum sul progetto finale: si consiglia di leggerle prima di iniziare a scrivere il codice e di consultarle in caso di dubbi.

Costanti

- `Num_elem 1000000` dimensione della tabella hash
- `PC_buffer_len 10`: lunghezza dei buffer produttori/consumatori
- `PORT 5XXXX`: porta usata dal server dove `XXXX` sono le ultime quattro cifre del vostro numero di matricola
- `Max_sequence_length 2048` massima lunghezza di una sequenza che viene inviata attraverso un socket o pipe

Il programma C archivio

Il file `archivio.c` deve contenere il codice C di un programma multithread che gestisce la memorizzazione di stringhe in una tabella hash. La tabella hash deve associare ad ogni stringa un intero; le operazioni che devono essere suportate dalla tabella hash sono:

- `void aggiungi(char *s)`: se la stringa `s` non è contenuta nella tabella hash deve essere inserita con valore associato uguale a 1. Se `s` è già contenuta nella tabella allora l'intero associato deve essere incrementato di 1.
- `int conta(char *s)` restituisce l'intero associato ad `s` se è contenuta nella tabella, altrimenti 0.

Le operazioni sulla tabella hash devono essere svolte utilizzando le funzioni descritte su `man hsearch`. Si veda il sorgente `main.c` per un esempio. Si noti che la tabella hash è mantenuta dal sistema in una sorta di variabile globale (infatti ne può esistere soltanto una).

Il programma `archivio` riceve sulla linea di comando due interi che indicano il numero `w` di thread scrittori (che eseguono solo l'operazione `aggiungi`), e il numero `r` di thread lettori (che eseguono solo l'operazione `conta`). L'accesso concorrente di lettori e scrittori alla hash table deve essere fatto utilizzando le condition variables usando lo schema che favorisce i lettori visto nella lezione 40 (o un altro schema più equo a vostra scelta).

Oltre ai thread lettori e scrittori, il programma archivio deve avere:

- un thread "capo scrittore" che distribuisce il lavoro ai thread scrittori mediante il paradigma produttore/consumatori
- un thread "capo lettore" che distribuisce il lavoro ai thread lettori mediante il paradigma produttore/consumatori
- un thread che gestisce i segnali mediante la funzione `sigwait()`

I thread scrittori e il loro capo

Il thread "capo scrittore" legge il suo input da una FIFO (named pipe) `caposc`. L'input che riceve sono sequenze di byte, ognuna preceduta dalla sua lunghezza. Per ogni sequenza ricevuta il thread capo scrittore deve aggiungere in fondo un byte uguale a 0; successivamente deve effettuare una tokenizzazione utilizzando `strtok` (o forse `strtok_r`?) utilizzando `".,:; \n\r\t"` come stringa di delimitatori. Una copia (ottenuta con `strdup`) di ogni token deve essere messo su un buffer produttori-consumatori per essere gestito dai thread scrittori (che svolgono il ruolo di consumatori). I thread scrittori devono semplicemente chiamare la funzione `aggiungi` su ognuna delle stringhe che leggono dal buffer.

Il buffer produttori-consumatori consiste quindi di puntatori a `char` e deve essere di lunghezza `PC_buffer_len`.

Naturalmente tutti gli array intermedi usati nel processo devono essere deallocati.

Non appena la FIFO `caposc` viene chiusa in scrittura, il thread "capo scrittore" deve mandare un valore di terminazione ai thread scrittori e terminare lui stesso.

I thread lettori e il loro capo

Il thread "capo lettore" si comporta in maniera simile al "capo scrittore" tranne che:

- Riceve il suo input dalla FIFO `capolet`
- Scrive i token su un buffer (sempre di lunghezza `PC_buffer_len`) che è condiviso con i thread lettori.

I thread lettori devono chiamare la funzione `conta` per ognuna delle stringhe lette dal buffer, e scrivere una linea nel file `lettori.log` contenente la stringa letta e il valore restituito dalla funzione `conta`; ad esempio se `conta("casa")` restituisce 7 il thread deve scrivere la stringa `casa 7` (seguita da un cararattere `\n`) nel file `lettori.log`.

Non appena la FIFO `capolet` viene chiusa in scrittura, il thread "capo lettore" deve mandare un valore di terminazione ai thread lettori e terminare lui stesso.

Il thread gestore dei segnali

Tutti i segnali ricevuti dal programma `archivio` devono essere gestiti da questo thread.

- Quando viene ricevuto il segnale `SIGINT` il thread deve stampare su `stderr` il numero totale di stringhe distinte contenute dentro la tabella hash (questo richiede che in qualche modo manteniate questo numero durante le operazioni `aggiungi`); il programma non deve terminare.
- Quando viene ricevuto il segnale `SIGTERM` il thread deve attendere la terminazione dei thread "capo lettore" e "capo scrittore"; successivamente deve stampare su `stdout` il numero totale di stringhe distinte contenute dentro la tabella hash, dellocare la tabella hash (e il suo contenuto per il **progetto completo**, vedere sotto) e far terminare il programma. Questa è l'unica modalità "pulita" con cui deve terminare il programma. Durante queste operazioni di terminazione non devono essere gestiti ulteriori segnali.
- **[Solo per il progetto completo]** Quando viene ricevuto un segnale `SIGUSR1` il thread gestore deve ottenere l'accesso in scrittura alla tabella hash, deallocare tutti i dati memorizzati nella tabella, e chiamare le funzioni `hdestroy` seguita da `hcreate(Num_elem)`. In pratica questo corrisponde a cancellare tutti i vecchi dati dalla tabella e ripartire con una tabella vuota.

Deallocazione della memoria

Il programma deve deallocare tutta la memoria utilizzata (a parte la tabella hash per il progetto ridotto, vedi sotto). Lanciando il server con l'opzione `-v` (vedi sotto) viene generato un file `valgrind-NNN.log` contenente il report della memoria persa con anche l'indicazione del punto in cui la memoria persa era stata allocata.

La memoria utilizzata dalla tabella hash non viene restituita automaticamente con la chiamata `hdestroy` di conseguenza chi fa il progetto ridotto è normale che, se nella tabella sono state memorizzate *N* stringhe si ritrovi *N* blocchi da 16 byte ciascuno `definitely lost` e *6N* blocchi di dimensione variabile `indirectly lost`.

[Solo per il progetto completo] Quando termina il programma deve deallocare anche la memoria utilizzata per memorizzare gli oggetti nella tabella hash. La funzione `hdestroy` si limita a deallocare la tabella; i dati in essa contenuti devono invece essere deallocati dal vostro programma. A questo scopo è necessario che gli oggetti inseriti nella tabella hash siano mantenuti in una linked list che deve essere usata per deallocare tutti gli oggetti al momento della terminazione del programma (o quando viene ricevuto il segnale `SIGUSR1`). Una possibile struttura di questa linked list è mostrata qui sotto ed è realizzata all'interno del file `main_linked.c`. Con questo modifica l'output di `valgrind` non dovrebbe mostrare nessun blocco perso.

"assets/lista.png" is not created yet. Click to create.

Il server

Il server deve essere scritto in Python e si deve mettere in attesa su `127.0.0.1` sulla porta `5XXXX` dove `XXXX` sono le ultime quattro cifre del vostro numero di matricola. Ad ogni client che si connette il server deve assegnare un thread dedicato. I client posso essere di due tipi

- **Tipo A**: inviano al server una singola sequenza di byte. Il server deve scrivere tale sequenza nella FIFO `capolet`
- **Tipo B**: inviano al server un numero imprecisato di sequenze di byte; quando il client ha inviato l'ultima sequenza esso segnala che non ce ne sono altre inviandone una di lunghezza 0. Il server deve scrivere ognuna di queste sequenze (tranne quella di lunghezza zero) nella FIFO `caposc`. **Solo per il progetto completo:** successivamente il server deve inviare un intero al client che indica il numero totale di sequenze ricevute durante la sessione.

Il server deve usare il modulo `logging` per la gestione di un file di log di nome `server.log`. Per ogni connessione, il server deve scrivere sul file di log il tipo della connessione e il numero totale di byte scritti nelle FIFO `capolet` o `caposc`.

Il server deve essere scritto in Python in un file *eseguibile* di nome `server.py` e deve usare il modulo `argparse` per la gestione degli argomenti sulla linea di comando, e deve richiedere come argomento obbligatorio un intero positivo che indica il numero massimo di thread che il server deve utilizzare contemporaneamente per la gestione dei client (usate la classe `ThreadPoolExecutor` vista a lezione).

Altre operazioni che deve svolgere il server:

- All'avvio, se non sono già presenti nella directory corrente, deve creare le FIFO `caposc` e `capolet`
- Deve accettare due parametri positivi `-r` e `-w` sulla linea di comando e deve lanciare il programma `archivio` passandogli questi due parametri sulla linea di comando che rappresentano rispettivamente il numero di thread lettori e scrittori (esclusi i capi). Il valore di default per entrambi questi parametri è 3. Usare `subprocess.Popen` per lanciare `archivio`, vedere `manager.py` per un esempio.
- Deve accettare l'opzione `-v` sulla linea di comando che forza il server a chiamare il programma `archivio` mediante `valgrind` con opzioni `valgrind --leak-check=full --show-leak-kinds=all --log-file=valgrind-%p.log`, vedere ancora `manager.py` per un esempio.
- Se viene inviato il segnale `SIGINT`, il server deve terminare l'esecuzione chiudendo il socket con l'istruzione `shutdown`, cancellando (con `os.unlink`) le FIFO `caposc` e `capolet` e inviando il segnale `SIGTERM` al programma `archivio` (il segnale `SIGINT` in Python genera l'eccezione `KeyboardInterrupt`)

Il client tipo 1

Questo client deve accettare sulla linea di comando il nome di un file di testo e inviare al server, una alla volta, le linee del file di testo con una connessione di tipo A. L'eseguibile si deve chiamare `client1`:

- Per il progetto ridotto questo client può essere scritto in Python; si deve comunque chiamare `client1` senza l'estensione `.py` ed essere un file *eseguibile*.

- **per il progetto completo:** questo client deve essere scitto in C e usare la funzione `getline` per la lettura delle singole linee del file di testo, e deve deallocare correttamente tutta la memoria utilizzata.

Il client tipo 2

Questo client deve accettare sulla linea di comando il nome di uno o più file di testo. Per ogni file di testo passato sulla linea di comando deve essere creato un thread che si collega al server e invia una alla volta le linee del file con una connessione di tipo B (si intende una connessione per ogni thread).

Questo client può esere scritto in C o Python a vostra scelta ma il file eseguibile deve chiamarsi `client2`

Materiale fornito

- I file `main.c`, `main_linked.c`, e `manager.py` con esempi di codice utile discusso nell'ultima lezione del corso e in questo testo.
- I file `file1`, `file2`, `file3`, e `lettori.esempio` da usare per il test di funzionamento base riportati qui sotto.

Consegna del progetto

La consegna deve avvenire esclusivamente mediante GitHub. Se non lo avete già createvi un account e create un repository **privato** dedicato a questo compito. Aggiungete come collaboratore al repository l'utente `Laboratorio2B` in modo che i docenti abbiano accesso al codice. **IMPORTANTE:** la consegna effettiva del progetto per un dato appello consiste nello scrivere l'url per la clonazione del vostro progetto nell'apposito contenitore consegna su moodle. L'url deve essere della forma `git@github.com:user/progetto.git` (**non** deve quindi iniziare per `https`). Dopo la data di consegna **non dovete** fare altri commit al progetto. Ricordatevi che almeno **cinque** prima della data di consegna dovete iscrivervi all'appello su esami.unipi.it.

Il repository deve contenere tutti i file del progetto. Il makefile deve essere scritto in modo che la seguente sequenza di istruzioni sulla linea di comando (`progetto` indica il nome del repository, voi usate un nome meno generico):

```
$ git clone git@github.com:user/progetto.git testdir

$ cd testdir

$ make

$ ./server.py 5 -r 2 -w 4 -v & # parte il server con 5 thread che

# a sua volta fa partire archivio

$ ./client2 file1 file2 # scrive dati su archivio
```

```
$ ./client1 file3 # interroga archivio

$ pkill -INT -f server.py # invia SIGINT a server.py

# che a sua volta termina archivio
```

non restituisca errori e generi i tre file `server.log`, `lettori.log` e `valgrind-XXX.log` con il contenuto corretto (si veda il file `lettori.esempio` per il contenuto corretto di `lettori.log` a meno dell'ordine delle righe). Al termine dell'esecuzione il programma `archivio` deve stampare che nella tabella hash ci sono 2010 stringhe distinte.

Fate la verifica descritta qui sopra sulla macchina `laboratorio2.di.unipi.it`, *partendo da una directory vuota*. Se tali operazioni non danno il risultato corretto il progetto è considerato non sufficiente e dovete ripresentarvi all'appello successivo.

Il repository deve contenere anche un file README.md contenente una breve relazione che descrive le principali scelte implementative. Si noti che è possibile gestire un repository GitHub anche da dentro Replit.

Nel repository dovete mettere solamente i file veramente essenziali per la compilazione del progetto: quindi non i file .o gli eseguibili o i file di configurazione di replit o del vostro editor. Per impedire l'inclusione di alcuni file potete indicare gli opportuni pattern nel file .gitgnore (doc).

Se il permesso di esecuzione per i programmi python non viene mantenuto sul repository è probabilmente necessario dare il comando `git config core.filemode` true come indicato qui.