Sistemi di Calcolo - Modulo 2 (A.A. 2014-2015)

Primo appello - 9 giugno 2015

Tempo a disposizione: 1h 30'.

Attenzione: assicurarsi di compilare il file **studente.txt** e che il codice prodotto non contenga **errori di compilazione**, pena una valutazione negativa dell'elaborato.

Realizzazione di un client per il recupero parallelo di testo da un server

L'obiettivo di questa prova è completare il codice di un client che lancia N *reader thread* concorrenti, ognuno dei quali comunica via FIFO con un server per recuperare delle porzioni di testo. In parallelo, un *reconstructor thread* (unico nel client) si occupa di riassemblare queste porzioni per formare il testo completo. Il server quando parte crea una server FIFO e si mette in attesa di messaggi su di essa. Ogni *reader thread* nel client invece:

- ha un identificatore intero univoco `tidx`
- crea una propria client FIFO avente come nome `tidx` e la apre in lettura
- scrive `tidx` nella server FIFO per consentire al server di inviargli messaggi (il server a questo scopo aprirà infatti in scrittura la FIFO avente come nome `tidx`)
- legge messaggi dalla propria client FIFO nel formato `i,c` (`i` è l'indice del carattere `c` nel testo completo)
- salva questi messaggi in un buffer circolare, dal quale il *reconstructor thread* li recupera per riassemblare il testo completo
- quando riceve un messaggio con indice `-1`, rilascia le proprie risorse e termina

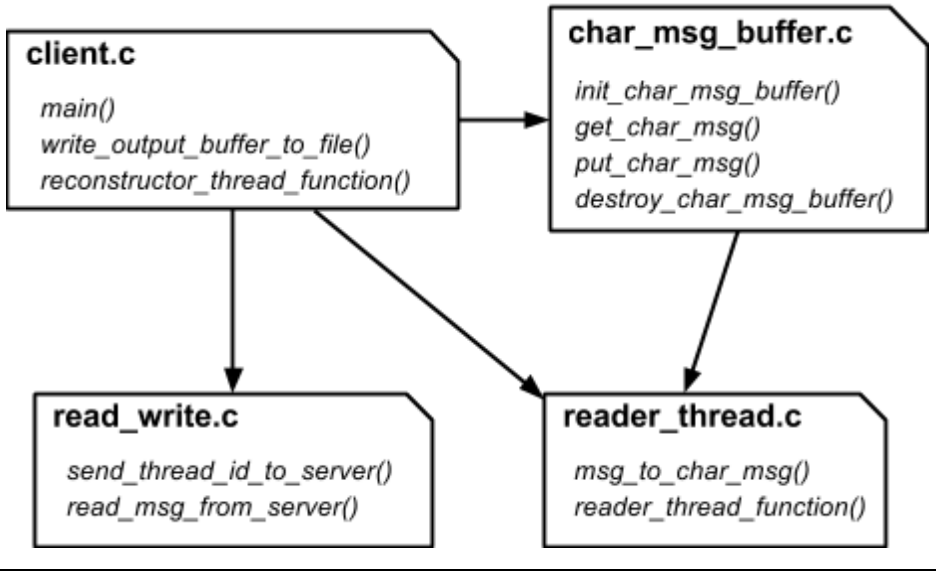
Quando tutti i *reader thread* sono terminati, il client mette nel buffer circolare un messaggio `NULL` per segnalare al *reconstructor thread* di terminare, poi scrive nella server FIFO un messaggio per segnalare al server che tutte le operazioni sono concluse. Infine, il client scrive su file (`output.txt`) il contenuto del buffer ricostruito dal *reconstructor thread*.

Descrizione del codice del client

Per ottimizzare la stesura e la leggibilità del codice, il client è organizzato in modo modulare, con due file di header `.h` e quattro file contenenti i sorgenti `.c` veri e propri:

common.h	Definizione delle strutture dati e delle macro (incluse quelle per la gestione degli errori) in uso nei sorgenti `.c`
methods.h	Contiene i prototipi dei metodi definiti in ciascun modulo
client.c	Lancio del *constructor thread* e dei *reader thread*, implementazione della logica del *constructor thread*, scrittura su file del testo riassemblato
read_write.c	Funzioni per la scrittura dell'identificatore di un *reader thread* nella server FIFO, e per la lettura dei messaggi inviati dal server sulle client FIFO

char_msg_buffer.c	Funzioni per la gestione (inizializzazione, inserimento, recupero, rilascio risorse) del buffer circolare nel quale i *reader thread* inseriscono messaggi e dal quale il *reconstructor thread* li recupera
reader_thread.c	Implementazione della logica del *reader thread*
Makefile	Compilazione automatizzata del codice



Architettura del codice sorgente. Una freccia indica che il modulo di origine utilizza uno o più metodi definiti nel modulo di destinazione.

Obiettivi
Vi viene richiesto di completare porzioni di codice - contrassegnate anche da blocchi di commenti nei rispettivi file - nei sorgenti C per implementare le seguenti operazioni:

(1) All'interno della funzione `main()` nel modulo `client.c`, si gestisca l'apertura e la chiusura della server FIFO, ed il lancio e l'attesa del termine sia del *reconstructor thread* che dei *reader thread*.

(2) Nel corpo della funzione `reader_thread_function()` nel modulo `reader_thread.c`, si implementino creazione, apertura, chiusura e rimozione della client FIFO.

(3) Il modulo `char_msg_buffer.c` include le funzioni da usare per la gestione del buffer circolare. Si implementi la dichiarazione ed inizializzazione dei semafori necessari, la gestione concorrente *race-free* delle operazioni di inserimento/rimozione di elementi nel/dal buffer, ed infine il rilascio delle risorse acquisite durante l'inizializzazione.

(4) Il modulo `read_write.c` contiene due funzione per la comunicazione via FIFO.

Si completi la funzione `send_thread_id_to_server()` affinché i byte della stringa contenuta nel buffer `msg` (incluso il carattere `\0` di terminazione) vengano scritti sul descrittore di FIFO `server_fifo_desc`. Una implementazione che assicuri che tutti i byte siano stati scritti verrà premiata in sede di valutazione.

Si completi poi la funzione `read_msg_from_server()` affinché scriva nel buffer `buf` una stringa di lunghezza fino a `max_len` byte leggendolo dal descrittore di FIFO `client_fifo_desc`. La stringa da leggere deve includere il carattere `\0` di terminazione. Nel caso in cui la FIFO venga chiusa dal server, la funzione deve restituire immediatamente `-1`, altrimenti il numero di byte realmente letti (incluso il carattere `\0` di terminazione se effettivamente letto da FIFO).

Testing
La compilazione è incrementale (ossia avviene un modulo per volta) ed automatizzata tramite `Makefile`, pertanto in assenza di errori di compilazione un binario `client` verrà generato quando si esegue `make`.

Viene fornito un eseguibile `server` precompilato che può essere utilizzato per testare rapidamente il corretto funzionamento del proprio client. Per lanciare il server, digitare da terminale

```
./server <input_filename>
```

Dove `<input_filename>` è il path del file di testo da usare come input. Vengono forniti tre file di input di taglie diverse: `small_input.txt` (42 byte), `medium_input.txt` (1301 byte) e `big_input.txt` (23463 byte).

Una volta lanciato il server, si può lanciare il proprio client su un altro terminale con:

```
./client <numero_reader_thread>
```

Dove `<numero_reader_thread>` è il numero di *reader thread* da lanciare. Se omesso, il numero di *reader thread* da lanciare viene impostato di default a 30 (macro `DEFAULT_READER_THREAD_COUNT`).

Al termine dell'esecuzione, sia il processo client che il processo server dovrebbero terminare senza errori. Il fatto che almeno uno dei due rimanga in esecuzione è sintomo di qualche malfunzionamento lato client. Per testare il corretto funzionamento del client, è possibile effettuare un confronto tra il contenuto del file di input (argomento `<input_filename>` del server) ed il contenuto del file di output generato dal client (`output.txt`) lanciando da terminale il comando

```
cmp <input_filename> output.txt
```

Se l'output di tale comando è vuoto, allora i due file sono identici. Altrimenti viene stampata a video la prima differenza riscontrata tra i due file.

Viene inoltre fornito uno script `reset.sh` che si occupa di eliminare il file `output.txt`, se presente, e tutte le eventuali FIFO non rimosse presenti nella cartella corrente.

Altro

- in caso di cancellazione accidentale di uno o più file, nella cartella `backup/` è presente una copia di sicurezza della traccia di esame

- il file `dispensa.pdf` contiene una copia della dispensa *Primitive C per UNIX Sytem Programming* preparata dai tutor di questo corso

---

Raccomandazioni

Seguono alcune considerazioni sugli errori riscontrati più di frequente tra gli elaborati dei partecipanti, nonché delle raccomandazioni per un buon esito delle prove al calcolatore:

- dato un `char* buf`, `sizeof(buf)` non restituirà il numero di byte presenti nel buffer, bensì il numero di byte occupati da un puntatore (4 su IA32, 8 su x86_64); se `buf` contiene una stringa NULL-terminated è possibile utilizzare `strlen(buf)`
- quando si deve invocare una funzione per la programmazione di sistema:
    - fare attenzione se i singoli parametri siano o meno dei puntatori
    - verificare dalla documentazione se può essere soggetta ad interruzioni, e in tal caso gestirle opportunamente
    - verificare il comportamento della funzione in caso di errori: inserire codice idoneo a trattare errori gestibili, utilizzare la macro opportuna per gli altri
    - per funzioni della libreria pthread, si vedano il capitolo 2 e l'appendice A.2 della dispensa di UNIX system programming
- quando viene acquisita una risorsa (socket aperta, area di memoria allocata, etc.), questa deve poi essere rilasciata in maniera opportuna
- utilizzare puntatori non inizializzati porta ragionevolmente un Segmentation fault!
- i cast di puntatori da e verso `void*` sono impliciti in C, renderli espliciti o meno è una scelta di natura puramente stilistica nella programmazione
- fare molta attenzione nell'aggiornamento dell'indice (o del puntatore) con cui si opera su un buffer: disallineamenti ±1 sono tra le cause più comuni di errori
- quando si vuole passare un puntatore ad una funzione, aggiungere al nome della funzione una coppia di parentesi tonde equivale invece ad invocarla!
- **i commenti nel codice contengono molte informazioni utili per lo svolgimento della prova, si consiglia quindi di tenerli in debita considerazione**

Regole Esame

- Domande ammesse
  Le domande possono riguardare solo la specifica dell'esame e la struttura di alto livello del codice, nessuna domanda può riguardare singole istruzioni.
- Oggetti vietati
  I seguenti oggetti non devono essere presenti sulla scrivania, né tantomeno usati: smartphone, telefonini, tablet, portatili, dispositivi di archiviazione USB, copie cartacee della dispensa, astucci e qualsiasi forma di libri ed appunti. **Chi verrà sorpreso ad usare uno di questi oggetti verrà automaticamente espulso dall'esame.**
- Azioni vietate
  È assolutamente vietato comunicare in qualsiasi modo con gli altri studenti. **Chi verrà sorpreso a comunicare con gli altri studenti per la prima volta verrà richiamato, la seconda volta verrà invece automaticamente espulso dall'esame**.