Relazione conclusiva progetto sistemi operativi – progetto 2

Anno accademico 2012-2013

Dellagiacoma Daniele 152027

Lissoni Davide 151855

Carli Luca 152011

Introduzione sull'ambiente di sviluppo e linguaggio di programmazione

L'ambiente dove il nostro progetto verrà virtualizzato nella fase della sua presentazione sarà quello utilizzato anche durante le lezioni in laboratorio, ovvero un sistema GNU/Linux basato sulla distribuzione Debian 6.0.5 "Squeeze" – kernel 2.3.4.
Il sistema GNU/Linux è un sistema operativo formato da GNU, il sistema operativo di tipo Unix distribuito come software libero, e da Linux, che fornisce il kernel.
GNU deriva da un progetto nato nel 1983 il cui scopo era quello di creare un sistema operativo composto solamente da software libero, chiamato appunto GNU, tutto questo software prodotto è sviluppato da una comunità di programmatori che condivide regolarmente tutte le modifiche al codice.
Il kernel Linux è il nucleo del sistema operativo, ovvero colui che si occupa dei tempi di accesso all'hardware, essendo Linux un kernel monolitico, ovvero un kernel contenuto in un unico file che ne rappresenta un vantaggio, ha lo svantaggio di essere rigido e di non permettere la liberazione delle risorse quando le unità periferiche gestite non servono.

Il linguaggio di programmazione utilizzato per questo progetto è il C, un linguaggio nato nel 1972.
Questo linguaggio già dalla sua nascita si è rivelato molto efficiente, diventando ben presto il linguaggio di riferimento per la creazione di software di sistema e piattaforme hardware.
Il C viene spesso definito un linguaggio a medio livello in quanto, nonostante sia un linguaggio molto più vicino all'utente, quindi di alto livello, mantiene alcune relazioni semantiche con il linguaggio macchina.
Il C è anche stato il linguaggio da cui sono derivati altri linguaggi di programmazione come il C++, che didatticamente ha sostituito il C, in quanto quest'ultimo, essendo molto vicino al funzionamento dell'hardware, è molto più complicato e meno intuitivo per i principianti.

Introduzione sul progetto

La scelta iniziale del nostro gruppo è stata quella di dividere subito il nostro progetto in diversi file per avere una più chiara struttura del codice, quindi abbiamo inizialmente creato il file denominato 'Threads_main.c'; questo file contiene l'inizializzazione e la creazione dei nostri quattro thread:

- Tr, un produttore di dati, ovvero stringhe prese in input dal terminale con lunghezza arbitrariamente lunga, nel nostro caso 500 caratteri
- Te, un consumatore di dati che genera una stringa casuale e ne fa lo XOR con la stringa inserita nel thread Tr
- Td, che fa semplicemente il decriptaggio dello XOR del thread Te, restituendo lo XOR tra il risultato di Te e la stringa casuale

- Tw, stampa solamente il risultato del thread Td, che deve essere la stringa inizialmente inserita dall'utente

Prima dell'inizializzazione e della creazione dei thread necessari allo sviluppo del codice nella fase di sistemazione del progetto abbiamo aggiunto un controllo sull'input con la getoption, infatti in fase di esecuzione il programma dovrà essere lanciato con l'opzione –q seguita da un intero, questo intero sarà poi la dimensione del nostro buffer fatto a coda che conterrà gli elementi che verranno inseriti in input, qualora nell'opzione –q verrà inserito un valore non numerico o lo zero, il programma verrà chiuso con la creazione di un messaggio d'errore, nel caso in cui il programma fosse eseguito senza l'opzione –q viene data alla coda una dimensione di default pari a 50.

L'altro file ausiliario del nostro progetto è il file 'Threads.h' che è stato creato per salvare tutte le dichiarazioni delle funzioni e della struct 'prodcons'.
La struttura 'prodcons', viene dichiarata con il nome 'buffer' ed è stata creata per salvare in un buffer di lettura e scrittura tutti gli elementi inseriti in input dall'utente.
Le variabili che compongono la struct 'prodcons' sono:

- char ** elem, ovvero l'array degli elementi contenuti della nostra coda
- int testa, il primo elemento della coda
- int coda, l'ultimo elemento della coda
- int size
- int mas, il numero massimo di elementi dell'array della coda
- pthread_mutex_t lock, il semaforo che amministra il funzionamento di Tr e Te
- pthread_cond_t notempty, una condizione che modifica il mutex nel caso che la coda sia vuota
- pthread_cond_t notfull, una condizione che modifica il mutex nel caso che la coda sia piena

Abbiamo scelto di utilizzare un mutex in quanto è un semaforo esclusivo ed essenziale per risolvere il problema del produttore-consumatore dei primi due thread.

Per evitare che questo file venga compilato più di una volta abbiamo incluso tutto il codice del file 'Threads.h' all'interno di un #IFNDEF.

Il cuore del nostro progetto è il file 'Threads.c' che contiene tutte le istruzioni delle funzioni e dei thread che vengono eseguite dal programma per arrivare al risultato finale.
Oltre alle funzioni abbiamo creato alcune variabili che per forza di cose dovevano essere dichiarate come globali, in quanto utilizzate in più funzioni, queste sono:

- char S[500], la variabile in cui verranno salvate le stringhe inserite dall'utente
- int controllo, la variabile che farà uscire dall'esecuzione il programma nel caso l'utente digiti la stringa "quit"
- char* R, Se, Sd, le variabili che conterranno rispettivamente la stringa random creata nel thread Te, la stringa derivata dallo XOR tra la stringa in input e la stringa casuale, e la stringa, che risulterà uguale ad S, che deriva dallo XOR tra la stringa random e la stringa Se
- int lungS, la variabile che conterrà la lunghezza delle stringhe prese in input

In aggiunta alle variabili globali abbiamo aggiunto anche dei semafori globali che gestiscono l'attesa e l'avvio dei thread Td e Tw.

Spiegazione funzionamento funzioni ausiliarie

All’interno del file ‘Threads.c’ sono implementate alcune funzioni necessarie allo svolgimento del problema.
La prima funzione richiamata dal programma è la funzione **init**(prodcons *b, int opzione), dove passo come parametri la coda creata per contenere gli input e ‘opzione’, ovvero l’intero che l’utente deve inserire nell’opzione –q, questa variabile darà la grandezza massima della coda.
All’interno di questa funzione vengono inizializzati la coda ed i semafori necessari all’esecuzione, con la gestione degli eventuali errori.
La funzione **dataora**(int nthread), dove nthread è la variabile che contiene il numero del thread che richiama questa funzione, crea inizialmente, se non è già presente, la cartella nella directory ‘/var/log/Threads/’; nella gestione dell’errore della mkdir il programma darà sempre un errore dicendo che la cartella è già stata creata, nonostante questo il processo continua normalmente con la sua esecuzione.
Creata la cartella il programma va ad interagire con la libreria ‘time.h’ salvando nella variabile char[30] orario la data e l’ora attuali del sistema.
La successiva switch è basata sulla variabile passata come parametro, infatti in base a quale thread ha richiamato la funzione, la switch farà aprire in append il file di log relativo su cui verrà scritto il valore della variabile orario e successivamente verrà chiuso il file, anche in questo caso vengono gestiti eventuali errori di apertura o chiusura.
La funzione **top**(prodcons *b) ritorna l’elemento in testa alla coda, senza toglierlo dalla stessa, ovviamente se la coda dovesse essere vuota questa funzione non ritorna nulla, il parametro passato è il nostro buffer di elementi.
La funzione **dequeue**(prodcons *b) toglie l’elemento in testa alla coda, qualora la coda fosse vuota non elimina nulla, anche qui, come per la top, il parametro è lo stesso.
La funzione **enqueue**(char *S, prodcons *b) aggiunge la stringa S, quella presa in input, in coda.
La funzione **ctrlS**(char *d) è una funzione di controllo sulla stringa d, infatti, nel caso in cui la stringa passata come parametro dovesse risultare uguale alla parola di escape ‘quit’ il programma restituisce 1, in caso contrario restituisce 2.
Abbiamo scelto di adottare questo tipo di controllo al posto dello string compare in quanto utilizzando questa funzione ritornava valori diversi per ogni dispositivo su cui facevamo girare il nostro programma.
La funzione **inserisci**(prodcons *b) come prima cosa controlla se l’array del buffer è pieno, in tal caso richiama il secondo thread fino a che il buffer non è vuoto e poi riparte, nel caso che il buffer non sia pieno allora aggiunge la stringa S in coda.
La funzione **estrai**(prodcons *b) è una funzione analoga alla funzione ‘inserisci’, in quanto anche qui prima di tutto viene effettuato un controllo sullo stato della coda, qualora il buffer risulti vuoto, la funzione richiama il thread produttore fino a che la coda non è in parte piena, solo dopo questo la funzione riprende la sua esecuzione.
Infatti dopo aver salvato in una variabile locale, chiamata ‘stringa’ il contenuto del primo elemento dell’array che contiene tutte le stringhe inserite dall’utente lo elimino dalla coda e ritorno il valore di ‘stringa’.
L’ultima funzione richiamata è la **deinit**(), questa funzione libera i blocchi di memoria allocati dinamicamente e azzera la variabile della lunghezza delle stringhe.

Spiegazione funzionamento thread

La parte centrale del progetto sono i thread, ovvero dei sottoprocessi che condividono concorrentemente tra loro le risorse.
Con la creazione di questi thread si è posto un problema di sincronizzazione che noi abbiamo risolto introducendo dei semafori.

Subito dopo l'avvio dell'esecuzione, il programma è vincolato ad entrare nel thread Tr, in realtà potrebbe entrare anche nel thread Te, ma nella sua esecuzione sarebbe costretto a ritornare nel thread Tr in quanto la coda degli elementi sarebbe vuota.
Entrato nel thread **Tr** il processo richiama la funzione dataora passando come parametro il numero del suo thread, ovvero l'1, questo fa si che venga creato il file di log Tr.log e ci venga scritto la data e l'ora relativa al momento dell'esecuzione del thread.
A questo punto si attende che l'utente inserisca una stringa seguita poi dal carattere <CR> (ENTER), al quale il nostro programma aggiunge '/0' come carattere successivo all'ultimo inserito, successivamente viene richiamata la funzione inserisci per aggiungere alla coda la stringa appena letta.
Finita quest'operazione viene richiamata la funzione ctrlS sulla stringa inserita dall'utente e il valore di ritorno viene salvato nella variabile globale 'controllo'.
Tutto questo thread è contenuto in un ciclo che termina solo quando la parola inserita dall'utente sarà uguale alla parola di escape.
Conclusa l'esecuzione del thread Tr, il secondo thread continua la sua esecuzione fino a quando la coda non è vuota.
Il secondo thread, **Te**, può quindi avviare la sua esecuzione creando un file di log, Te.log, analogo a quello creato per il primo thread.
Già detto del suo primo controllo sul contenuto dell'array 'elem' contenuto nella coda, estrae il primo elemento della coda e lo copia in una variabile locale, 'd', questa stringa viene comparata con la stringa di escape tramite la funzione ctrlS, nel caso dovesse risultare proprio la parola 'quit', il programma termina.
Fatto il controllo il thread alloca dinamicamente due stringhe, R e Se, lunghe entrambe come la stringa d.
Subito dopo viene aperto il device dev/random, che ci serve per inserire caratteri casuali all'interno della stringa R.
Il passo successivo è quello di scrivere nella stringa Se il valore dello XOR byte-per-byte tra R e d.
Prima della fine vengono stampate a video prima la variabile R e poi la variabile Se.
Tutto questo thread è contenuto all'interno di un ciclo che esce solamente quando la lista degli elementi del buffer è vuota.
Solamente dopo la conclusione del thread Te, viene attivato il semaforo ed è possibile quindi l'avvio del terzo thread.
Il thread Td, come i due precendenti thread, richiama la funzione dataora che scrive nel file di log Te.log la data e l'ora attuali.
Dopo di questo il processo alloca dinamicamente un'altra variabile, Sd, dove sarà contenuto il risultato dello XOR byte-per-byte tra R e Se.
Dopo questa operazione viene attivato il semaforo che manda il programma all'interno del quarto thread.
Il thread **Tw**, crea e modifica il suo file di log, Tw.log, contenuto nella cartella dove sono contenuti tutti i file di questo tipo del nostro programma, ovvero '/var/log/Threads/', dopo di che stampa a video la stringa Sd e richiama la funzione deinit() che libera tutte le variabili allocate dinamicamente e riporta a 0 lungS.
Ovviamente qualora l'utente inserisse la stringa di escape, che causerebbe la terminazione del thread Tr, gli altri thread continuano normalmente la loro esecuzione fino a che dalla coda degli elementi verrà estratta la parola 'quit', a quel punto il processo terminerà.
Un altro punto in comune tra tutti quattro i thread è la presenza in ogni system call di un controllo che gestisce eventuali errori e ne restituisce una stringa di informazione nel momento in cui si verifichino.

Considerazioni sulla sincronizzazione

Come si evince dal resoconto sul funzionamento dei thread, la sincronizzazione è stato uno dei problemi su cui il nostro gruppo ha lavorato di più, infatti inizialmente eravamo orientati verso una sincronizzazione diversa da quella finale, la quale comprendeva solamente tre semafori, i quali però non ci lasciavano far ripartire il nostro thread produttore.
Dopo vari tentativi in questo senso abbiano pensato ad una nuova implementazione, che poi si è rilevata quella finale, dove inizialmente blocchiamo solamente il terzo e il quarto thread, e quindi lasciamo liberi il primo e il secondo.
Questo è possibile grazie alla presenza nella funzione estrai, che viene chiamata nel caso in cui le risorse vengano date all'inizio al thread numero due, di un controllo, il quale esamina la coda degli elementi e nel caso essa sia vuota fa partire il thread produttore, che si occuperà di inserire nel buffer le stringhe inserite dall'utente.
A questo punto è possibile usare anche il secondo thread.
Dopo aver fatto tutte le proprie operazioni il thread Te, attiva il semaforo relativo al terzo thread che può così procedere.
Alla fine del terzo thread, analogamente a quello che succede nel thread precedente, viene attivato un secondo semaforo che ha lo scopo di far partire il quarto ed ultimo thread.

Considerazioni sui thread Tr e Te

I primi due thread, Tr e Te, abbiamo deciso di implementarli tramite il metodo del produttore-consumatore.
Nel nostro caso il thread Tr è il produttore, che ha il compito di generare dati ed inserirli nel buffer, il thread Te è invece il nostro consumatore, che ha lo scopo di utilizzare i dati prodotti ed estrarli dal buffer.
Nel caso in cui il buffer sia pieno, allora il processo produttore deve fermarsi fino a quando il buffer non sarà vuoto ,a quel punto il processo consumatore deve risvegliare il produttore per evitare un deadlock, allo stesso modo il thread consumatore dovrà fermarsi nel caso il buffer sia vuoto, e il produttore risvegliarsi dopo l'inserimento di dati nel buffer.

Considerazioni finali sul progetto

Le difficoltà principali che abbiamo riscontrato in questo progetto sono state implementare il problema del produttore-consumatore tra i primi due thread, che ci ha dato quasi una settimana di lavoro per capirne il funzionamento e il modo in cui noi volevamo fosse eseguito, e la sincronizzazione del terzo e del quarto thread che devono sbloccarsi solamente dopo che il secondo thread era concluso.
Questo progetto ci ha consentito di lavorare in gruppo, cosa che per alcuni aspetti non è mai facile, nonostante questo abbiamo lavorato sempre bene rispettando ognuno le idee degli altri componenti del gruppo.

Sequenza di comandi per l'esecuzione del progetto

Per la compilazione, dopo essere andati sulla cartella contenente il progetto, utilizzare il comando
make
per l'esecuzione utilizzare il comando con la getoption uguale a 15
./a.out –q 15
oppure il comando senza getoption, che quindi inizializza la coda a 50
./a.out

Per eseguire il nostro progetto bisogna avere i privilegi di root, qualora l’utente non li possedesse deve utilizzare lo stesso comando per la compilazione, mentre per l’esecuzione dovrà anteporre la parola **sudo** al comando ./a.out.