Documentazione Relativa al Progetto di Sistemi Operativi

Studenti :

Mattia Santangelo	151776
Luca Amadori	matricola

Progetto sviluppato: Threads

Introduzione

L'obiettivo del progetto è quello di creare un programma in grado di gestire l'esecuzione di 4 threads, che devono rispettivamente svolgere le seguenti funzioni:

- T1 : legge da tastiera una sequenza di caratteri (S) arbitrariamente lunga fino a che non riconosce il carattere <CR>, ossia ENTER.
- T2 : legge dal device *"/dev/random"* una stringa di caratteri R della dimensione di S, e successivamente, ne fa lo XOR byte-per-byte con S ottenendo Se=XOR(R,S).
- T3 : accede ai dati contenuti in R e Se e calcola Sd=XOR(R,Se).
- T4 : stampa a schermo l'ultima stringa generata Sd.

Inoltre il programma genera nella cartella *"/var/log/threads"* quattro file di log, che servono per verificare, a esecuzione terminata, la riuscita delle operazioni di ciascun thread.

Linguaggio Utilizzato : C

Istruzioni

- All'interno del terminale accedere alla cartella in cui si trovano i file del progetto
- Mediante il comando *make* e *make clean* si compileranno tutti i file con estensione .c ottenendo un eseguibile , chiamato *thread*.
- Lanciare il comando *./thread* , con permessi di root , per poter scrivere nella cartella di sistema */var/log/threads.*
- Terminare inserendo la stringa "*quit*".

Perché scrivere in C?

La sua struttura minimale, costituita da un numero di parole chiave molto ristretto, il fatto che il primo sistema Unix sia stato scritto in C, sono due fattori determinanti nella scelta di tale linguaggio. Inoltre il codice risulta essere facilmente trasferibile da macchina a macchina e vanta un gran numero di librerie, che permettono l'esecuzione di numerose operazioni. Da tutto ciò si ricava un'ottima efficienza e stabilità dei programmi.

L'automazione

L'utilizzo dei comandi *make* e *make clean* semplificano la compilazione dei file, eseguendo una serie di operazioni automatizzate seguendo una descrizione di regole e dipendenze (Makefile).
Tali regole sono state descritte per garantire la compilazione dei file con estensione .c e la rimozione dei file object.
Uno dei vantaggi più appariscenti sta nella possibilità di evitare che vengano ricompilati i file sorgenti che non sono stati modificati, abbreviando quindi il tempo di compilazione necessario quando si procede a una serie di modifiche limitate.
Per eseguire una regola specifica , come ad esempio clean, è necessario specificarne il nome , indicandolo al comando *make* → *make clean*.

Struttura del programma

La prima riga del file principale thread.c richiama il file *declaration.h*, contenente l'importazione di tutte le librerie e le dichiarazioni di funzione che sono state utilizzate per la stesura del testo in codice.
Successivamente vengono dichiarati globalmente tutti i tipi di dato che saranno utilizzati nel corso dell'esecuzione del programma.
All'inizio del main è stata implementata la funzione getOpt() che dà la possibilità al programma di ricevere argomenti e opzioni quando viene lanciato. Il passaggio degli argomenti e delle opzioni è effettuato attraverso gli argomenti *argc* e *argv* della funzione *main*, che vengono passati al programma dalla shell quando questo viene messo in esecuzione. Un argomento che da linea di comando inizia con '-' o '—' viene considerato un'opzione. Per gestire le opzioni all'interno degli argomenti a linea di comando le librerie standard del C forniscono questa funzione:

```
#include <unistd.h>
Int getopt(int argc, char *const argv[], const char*optstring)
```

L'opzione inserita nel suddetto file è '-h' e mostra nel terminale come utilizzare correttamente l'eseguibile per non incorrere in errori di creazione/scrittura file.
Infatti subito dopo,nel main(), viene chiamata la funzione *try_catch(File *f1_log, File *f2_log, File *f3_log, File *f4_log, char *f1, char *f2, char *f3, char *f4)* che crea la cartella */var/log/threads* e dentro di essa quattro file di log, che serviranno a controllare l'andamento del programma; tale operazione richiede i permessi di root che devono essere garantiti, altrimenti il processo terminerà restituendo un errore.

Al fine di garantire un corretto e ordinato accesso ai dati in memoria condivisa da parte dei threads sono stati implementati dei semafori sem_t. Questo tipo di dato è possibile utilizzarlo mediante l'importazione della libreria <*semaphore.h*> . L'inizializzazione della suddetta variabile accetta in ingresso 3 parametri, che sono costituiti da: l'indirizzo del semaforo da creare, un intero che stabilisce se la sua condivisione avverrà tra più processi ed infine un intero con il suo valore iniziale. Le funzioni utilizzate nei threads per modificare il valore dei semafori (ossia bloccarli o sbloccarli) sono :

```
sem_wait(sem_t  *sem)
sem_post(sem_t *sem)
```

I Pthreads
Le API standard POSIX (Portable Operating System Interface)(IEEE 1003.1c) sono utilizzate per la creazione e sincronizzazione di thread. Un singolo processo può contenere più thread, i quali seguono gli standard chiamati POSIX Threads (PThreads), e condividendo la stessa memoria globale, avendo la possibilità di modificare i dati in essa.

La creazione dei quattro threads avviene tramite la chiamata :
*int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)*
dove:

- "*thread"* corrisponde all'argomento di output, in cui viene riposto l'id del nuovo thread;
- *"attr"* è l'argomento di input che specifica gli argomenti di input del thread creato (NULL= attributi di default);
- "*start_routine*" è la funzione computata dal thread;
- *arg*" sono gli argomenti da passare alla funzione calcolata dal thread.

Nel caso in cui la funzione dovesse richiedere parametri multipli, essi devono essere passati tramite array o struct.

Quando finisce un thread
Un thread può terminare quando finisce l'esecuzione del codice della routine specificata, all'atto della creazione del thread stesso, oppure quando, nel codice della routine si chiama la syscall di terminazione: *void pthread_exit(void *value)*.
Quanto il thread termina la sua esecuzione, o viene restituito il valore di return specificato nella routine, oppure, se si chiama la funzione pthread_exit(), si avrà il valore passato a questa syscall come argomento.

Un thread può sospendersi, in attesa della terminazione di un altro thread chiamando la syscall:

*int pthread_join(pthread_t thread, void **value_ptr)*

dove thread è l'identificativo del thread di cui si attende la terminazione e value_ptr è il valore del thread restituito dal thread che termina.

In questo caso tutti e quattro i thread necessari per il progetto sono stati creati nel main.
L'esecuzione del programma termina nel momento in cui il primo thread esegue il codice della sua routine, ovvero fino all'immissione della stringa di uscita *"quit"*.

Quali Pthread vengono utilizzati e come elaborano i dati

Primo Thread
Il primo thread, come da specifiche, accetta in ingresso una stringa di caratteri . La funzione richiamata alla creazione del thread è divisa in due parti: la prima chiede all'utente di inserire il testo e la seconda verifica il testo inserito. Attraverso il comando *getline(&s, &nbytes, stdin)* l'immissione del testo terminerà nel momento in cui verrà premuto il tasto ENTER.
Al fine di risparmiare memoria l' allocazione della stringa in C, come array di caratteri, avviene attraverso il comando *void *malloc(size_t)*.
Se l'operazione ha successo viene restituito un puntatore al blocco di memoria, mentre in caso contrario verrà restituito un puntatore a *null*. La memoria allocata con malloc è persistente: ciò significa che continuerà ad esistere fino alla fine del programma o fino a quando non sarà esplicitamente deallocata, mediante il comando *void free(void* pointer).*
Una volta allocata la stringa, il codice controlla se essa corrisponde a *"quit"* e quindi procede con la terminazione del programma, altrimenti con l'utilizzo di *sem_post(sem_t *sem)* viene data la possibilità al secondo thread di accedere ai dati salvati in memoria finora.

Secondo Thread
L'elaborazione dei dati di questo thread attende che la stringa venga salvata. Come prima operazione calcola la lunghezza della stessa, in quanto ,come da richieste, la chiave di codifica e il testo da cifrare devono avere la stessa dimensione.
Allocata dinamicamente una stringa *r*, che conterrà la chiave di codifica, viene aperto in lettura il device */dev/random*.
Il comando utilizzato è *FILE *fopen(const char *path, const char *mode)* presente nella libreria standard <*stdio.h*>. Se l'operazione non dovesse andare a buon fine, ossia *fopen* dovesse restituire il valore NULL, l'esecuzione del programma termina con un errore.
Verificata questa condizione, si procede con la lettura di *n* caratteri, pari alla lunghezza del testo inserito da terminale, e salvati nella stringa appena allocata.
Una volta chiusa la lettura dal *device*, verrà stampata a video la chiave di codifica.
Importante non farsi ingannare dai caratteri che si vedranno, in quanto per rendere la codifica più efficiente, verranno sostituiti tutti i caratteri non stampabili con '#'. Ciò non significa che si andrà a modificare, quindi semplificare, tale chiave, poiché i caratteri non stampabili non sono stati sostituiti in memoria, ma soltanto durante la stampa a video.
Questa operazione viene eseguita dalla funzione *print_string_ctrl(char *elem)*. In essa vengono utilizzate due chiamate a funzioni standard C , *isprint(int ch)* e *iscntrl(int c)* , che verificano rispettivamente se i caratteri della stringa letta dal device sono stampabili o se sono caratteri di controllo(*\n, \t, \a*).
La prima restituisce un valore diverso da 0 se il carattere passato come argomento è stampabile, inclusi gli spazi, diversamente ritorna 0; la seconda ,invece, restituisce 0 se il carattere analizzato è un *control character*.
Mi avvalgo di un'altra stringa *Se* allocata come le precedenti, che contiene lo XOR byte-per-byte dei due array di caratteri salvati in memoria. Quest'ultima verrà passata come argomento alla funzione di stampa *print_string_ctrl*, per evitare di avere sullo schermo caratteri indesiderati.
Infine sblocca l'accesso ai dati fino ad ora ottenuti al terzo thread.

Terzo Thread
Ottenuta la possibilità di elaborare i dati, il terzo thread alloca la quarta ed ultima stringa *Sd* che conterrà lo XOR byte-per-byte delle stringhe della chiave di codifica e della stringa criptata. Il risultato atteso è di ottenere la stringa inserita dall'utente.

Quarto Thread
L'ultimo thread creato si occupa soltanto di stampare a video l'ultima stringa allocata per verificare se effettivamente il risultato atteso corrisponde con quello ottenuto.
Inoltre vengono lanciati i comandi per liberare la memoria, affinché il primo thread possa richiedere un nuovo input ed avere tutte le variabili da allocare. Così facendo sarà possibile inizializzare le stringhe di dimensione adeguate senza incorrere in sprechi di memoria.
A questo punto un ciclo di elaborazione dati è concluso e ritorna al primo thread, pronto per ricevere nuovi dati.

I log come Monitor Esecuzione
Per ogni funzione dei thread il programma riporta l'esito sui rispettivi file di log, situati nella cartella */var/log/threads*. Ciascuna riga contiene data e ora, ottenuti attraverso l'importazione della libreria <*time.h*>.
Mediante l'utilizzo della formattazione di stringhe *strftime* è possibile ottenere il formato *datetime* desiderato. Recuperare le informazioni sulla data e ora corrente è compito della variabile time_t.
Sono state implementate due funzioni *void printEx(FILE *f, char * elem, int errnum)*
*void printOp(FILE *f, char * elem , int op)*

che accettano tre parametri in ingresso: il file su cui si vuole scrivere, la directory di tale file, e il codice errore/operazione desiderato.
Di seguito è riportato l'elenco dei codici per comprendere il comportamento delle due funzioni sopra citate.

```
/* LOG CODES */
/*

        Function : printEX()

    0 :     Execution Terminated by entering String Quit
    -1 :    Error while creating a thread
    -2 :    Error while joining first thread with main
    -3 :    Error while opening device /dev/random
    -4 :    Error while reading from device /dev/random

    1 :     The creation of 4 threads is done properly.
    2 :     The pthread_join is done properly.



        Function : printOp()

    1 :     The first thread reads the string from command line successfully.
    2 :     The second thread encrypts the read string.
    3 :     The third thread decrpyt the input string with the key.
    4 :     The fourth thread prints the result of the decryption.


*/
```