Università di Pisa

Dipartimento di Informatica

Corso di Laurea in Informatica (L31)

Progetto di Laboratorio di Sistemi Operativi

Sessione invernale a.a. 2014-15

Relazione sul progetto dello studente Simone Pignotti

Matricola 489911

Domenica, 11 Gennaio 2015 - Pisa

In questa breve relazione vengono descritte maggiormente le scelte progettuali effettuate durante la realizzazione del progetto, e vengono lasciati alla lettura del codice, adeguatamente commentato, i dettagli implementativi meno importanti. Ogni paragrafo consiste della descrizione del contenuto e del funzionamento di ciascuno dei 7 file di cui è composto l'operato.

libacc.h

Questo header contiene solamente i prototipi delle funzioni implementate nell'ononimo .c, ed è necessario per poter utilizzare tali funzioni nel codice del client (*include*). Sono inoltre dichiarate qui le variabili globali intere *skt* e *max*, che rappresentano rispettivamente il file descriptor del socket a cui il client si connetterà (globale per essere mantenuto disponibile tra le varie chiamate a funzione) e la lunghezza massima delle stringhe usate nel protocollo di comunicazione (inizializzato a MAX_LEN+20 nella funzione *os_connect* perchè ogni messaggio, sia di errore che di richiesta/risposta, non contiene mai più di 20 caratteri oltre al nome del client).

libacc.c

L'implementazione delle funzioni di libreria contenute in libacc.h è in questo file, ed è realizzata in questo modo:

- *os_connect*: qui viene creato il socket per la comunicazione con lo store, vengono inizializzate *skt* e *max* e viene assemblato e mandato il messaggio di registrazione allo store, contenente il nome del client. A seconda della risposta dello store essa ritornerà 1 (ricevuto un OK) o 0, in questo caso stampando il messaggio di errore ricevuto in risposta dallo store e chiudendo la comunicazione. Questo comportamento per il controllo dell'errore vale anche per tutte le altre.

- *os_store*: la particolarità di questa funzione è il metodo con cui comunica allo store l'oggetto da memorizzare: il blocco viene infatti diviso in parti da 100 bytes l'una, in modo da cercare di evitare scritture parziali per carenza di spazio sul buffer del socket.
- *os_retrieve*: l'acquisizione avviene a blocchi di 100 bytes come nella store. In più questa volta va allocato lo spazio per l'oggetto in memoria, esattamente della sua dimensione acquisita nella prima parte della risposta dello store, e il puntatore a esso viene restituito al chiamante, che si occuperà poi di liberarlo con una *free()*.
- *os_delete*: consiste semplicemente nell'invio del messaggio per la cancellazione di un oggetto.
- *os_disconnect*: ancora più semplice della precedente, non contiene neanche un controllo per il tipo di errore che potrebbe verificarsi, e il socket viene chiuso in ogni caso.

client.c

Nel file *client.c* è contenuta l'implementazione del client di test. Il client è un processo single-threaded che effettua in sequenza queste operazioni:

- *os_connect*
- una serie di *os_store/os_retrieve/os_delete* a seconda del codice con cui viene lanciato l'eseguibile
- *os_disconnect*

Durante tutta l'esecuzione vengono mantenuti i contatori delle operazioni effettuate e di quelle eseguite con successo, dati che vengono stampati al termine insieme al tipo di test che è stato eseguito. Queste informazioni saranno utili allo script *testsum.sh* per ricavare una misura delle performances del sistema.

store.c

La maggior parte del codice contenuto in questo file riguarda il comportamento dei thread worker. Il main infatti è composto da poche righe e consiste nel set-up dell'ambiente necessario per l'esecuzione del programma, ossia la creazione della cartella *data*, l'inizializzazione del socket, e la chiamata della funzione *dispatcher*: questa consiste invece di un ciclo in cui il processo si mette in attesa di connessioni da parte dei client (con la funzione *accept*) e ogni volta che questo accade crea un thread worker che si occuperà della relativa comunicazione. Se viene superato il limite di connessioni (arbitrario, specificato dalla costante MAX_CONN) lo store termina. Nel caso in cui si volesse portare in rete il sistema si potrebbe controllare quali sono i canali di comunicazione liberi e indirizzare le nuove richieste su quelli, ma nel dominio AF_UNIX quest'operazione è molto più complessa e non ho ritenuto necessario implementarla, anche perchè avrebbe influito molto sulle prestazioni del sistema. Una volta che al client è stato assegnato un worker, entrambi rimangono attivi e comunicano finchè non arriva un messaggio di disconnessione o si verifica un errore, nel qual caso il socket viene chiuso e il thread termina la sua esecuzione. Le funzioni che rispondono alle varie richieste sono speculari alle funzioni della libreria di accesso, e i passaggi più ambigui sono stati commentati. Per quanto riguarda la terminazione del server non ho provveduto a gestire segnali per il semplice motivo che non ci sono risorse da liberare o altri problemi legati a essa, quindi il processo può semplicemente terminare con un segnale di interruzione, lanciato preferibilmente quando non ci sono più client connessi. In ogni caso i client restituirebbero semplicemente un errore al primo tentativo di connessione/lettura/scrittura e terminerebbero anch'essi.

Makefile

Il *Makefile* è molto semplice: il target di default, *all*, compila *libacc.c* in codice oggetto per poi inserirlo nella libreria *libacc.a* e compila i file *store.c* e *client.c*, il target *perm* assegna il permesso di esecuzione agli script *test.sh* e *testsum.sh*, *clean* ripulisce la directory, e *test* esegue i due script bash già citati, il primo che lancia gli eseguibili e ridireziona il loro *stdout* sul file *testout.log* e il secondo che esamina i dati che trova in quel file.

test.sh

In questo file è definito il comportamento del target *test* del *Makefile.* Anche questo script è molto semplice e non fa altro che lanciare lo store e vari client e ridirezionare i loro *stdout* nel file *testout.log.* I client vengono lanciati in una subshell in modo da poter eseguire il comando wait senza parametri aggiuntivi per attendere la terminazione di tutti i processi in esecuzione sulla shell corrente, così da aspettare la creazione dei file (test di store) prima di effettuare test di retrieve e delete e di mandare un segnale di interruzione allo store.

testsum.sh

Questo script analizza il contenuto del file *testout.log*, ricavando informazioni sulla percentuale di successo dei vari tipi di test e quella generale del sistema. Il procedimento che attua per ottenere questi dati è molto ben descritto nei commenti al codice.