Progetto didattico – Sistemi Operativi

Andrea Maggiordomo
Informatica applicata – Università di Pisa

2013

Sommario

Questo documento descrive gli aspetti rilevanti della realizzazione del progetto didattico `fats` per il corso di Sistemi Operativi e Laboratorio e costituisce, insieme ai commenti al codice, la documentazione del sistema realizzato. La sezione 1 tratta delle specifiche implementate dal sistema, mentre la sezione 2 illustra l'organizzazione dei file consegnati e la sezione 3 menziona brevemente alcuni aspetti relativi all'implementazione della libreria. Le sezioni 4, 5, e 6 documentano rispettivamente il funzionamento del comando per la formattazione di un file system, del server e del client: particolare attenzione è rivolta alla descrizione dell'architettura del server, che si è rivelata la componente del sistema più complessa da realizzare. La sezione 7 illustra il processo di compilazione e l'esecuzione dei semplici test implementati.

1 Specifiche del sistema

Argomento del progetto didattico di sistemi operativi è lo sviluppo di un sistema software che gestisca un file system attraverso un'architettura di tipo client-server. Il sistema deve operare su dispositivi virtuali memorizzati come comuni file Unix; nel resto del documento i termini *dispositivo* o *device* verranno usati con riferimento a questi file.

Le specifiche del progetto prevedono un insieme di funzionalità di base la cui implementazione è obbligatoria, e la possibilità di estendere il sistema con alcune funzionalità avanzate. La soluzione proposta implementa oltre alle funzionalità di base anche quelle descritte nella sezione 10 della specifica: realizza cioè un ambiente multi-utente nel quale il server gestisce un insieme di dispositivi, ciascuno assegnato ad un proprietario. I client si identificano con un nome utente e hanno la possibilità di scegliere il dispositivo con cui interfacciarsi. Il sistema prescrive che un client possa operare solo su un file system che abbia come proprietario il nome utente dichiarato.

Per realizzare questa caratteristica è stata modificata la libreria `fats` in modo da associare un utente ad ogni device: la scelta è stata quella di aggiungere alla struttura che definisce il boot sector il campo `usr` dove memorizzare il nome del proprietario

```
#define FAT_MAXUSRLEN 16

struct boot_sector {
  char fs_type;
  int block_size;
  unsigned int num_block;
  char usr[FAT_MAXUSRLEN];
};
```

In questo modo è possibile associare un utente al file system quando si effettua il mounting (cioè quando il server carica le informazioni di controllo).

Il server è realizzato con un'architettura multi-thread per permettere la gestione di molteplici file system e conversazioni contemporaneamente. Sono inoltre stati definiti opportuni meccanismi di sincronizzazione per evitare interferenze in caso di accessi concorrenti allo stesso device.

La API utilizzata per la programmazione multi-thread è quella definita dallo standard POSIX (Pthreads).

2 Organizzazione dei file

La root directory del progetto contiene il Makefile ed il file `README`.

I file sorgente risiedono nella cartella `src` e sono raggruppati tenendo conto delle componenti del sistema che realizzano: le cartelle `client`, `server` e `create` contengono i file relativi all'implementazione degli omonimi moduli; in `common` sono implementate le funzionalità condivise da client e server (in realtà solo il controllo sulla validità sintattica dei percorsi); `fat` ospita le definizioni relative al file system e l'implementazione delle operazioni mentre `com` contiene l'implementazione delle funzioni per la comunicazione tra processi definite dalla libreria.

La cartella `tmp` ospita a runtime il socket di ascolto del server come da specifica, mentre la cartella `test` contiene alcuni file utilizzati per verificare automaticamente il funzionamento del sistema (vd. sezione 7).

3 Implementazione della libreria

Le funzioni per la comunicazione tra processi sono sostanzialmente dei wrapper per le analoghe chiamate di sistema Unix (`socket` e `bind`, `accept`, `connect`), e due funzioni specializzate per lo scambio di oggetti `message_t` attraverso i socket, basate su `read` e `write`.

Per quanto riguarda l'implementazione delle operazioni sul file system `fats`, trattarne in maniera discorsiva risulta difficile e si rimanda pertanto al codice, che è opportunamente commentato. La navigazione del file system usando i riferimenti parent (specificare percorsi del tipo `/dir1/../dir2` per indicare `/dir2`) dovrebbe essere pienamente supportata.

4 Formattazione dei dispositivi

La sintassi del comando per la formattazione dei dispositivi è modificata per accomodare la gestione degli utenti con l'aggiunta del parametro `usr` che dichiara il proprietario del file system

```
fats_create usr name nblocks size
```

La formattazione di un dispositivo è estremamente semplice: viene compilato il boot sector con le informazioni specificate, costruita la FAT (un array di `unsigned int` con `nblocks` elementi) e la directory table per la cartella di root (che inizialmente occupa un solo blocco con le entry "`.`" e "`..`"). Sul file designato come dispositivo viene scritto inizialmente il boot sector, successivamente la FAT con tutti gli indici eccetto il primo inizializzati a `BLOCK_FREE` ed infine la data region composta dal primo blocco che contiene la directory table della cartella di root e `nblocks-1` blocchi vuoti.

5 Il processo server

Il server viene lanciato con il comando

```
fats_server fsdir
```

dove `fsdir` indica la cartella dalla quale caricare i dispositivi.

Il funzionamento del server è relativamente semplice: in fase di avvio indicizza i dispositivi contenuti in `fsdir`; successivamente attende le richieste di connessione da parte dei client. Il protocollo di comunicazione prevede un handshake a due vie nel quale il client invia una richiesta di connessione al server specificando un nome utente (e opzionalmente il nome di un particolare device richiesto); il server accetta la sessione e la affida ad un thread dedicato soltanto se trova un device associato all'utente, oppure se il device richiesto esplicitamente esiste ed è associato al nome utente specificato.

5.1 Gestione dei dispositivi

Quando un device viene caricato, il server inizializza alcune informazioni di supporto; queste sono raggruppate in una struttura apposita

```
struct file_handle {
  char *name;
  FILE *fs;
  struct fat_ctrl fsctrl;
  int num_readers;
  pthread_mutex_t num_mutex;
  pthread_mutex_t write_mutex;
};
```

Il campo `name` denota il nome del file fisico con cui è realizzato il device; i campi `fs` e `fsctrl` memorizzano le informazioni di controllo relative al file system (puntatore al dispositivo e metadati); i campi `num_readers`, `num_mutex`, `write_mutex` sono utilizzati per regolare gli accessi concorrenti al dispositivo, e verranno menzionati nella sezione dedicata. Intuitivamente, utilizzare una struttura di questo tipo permette di tenere facilmente traccia di tutte le informazioni necessarie ad un thread per accedere e operare su un file system.

I `file_handle` relativi ai file system controllati dal server vengono memorizzati in una tabella con visibilità globale `fstab` che viene inizializzata durante la fase di indicizzazione ed è rappresentata con la struttura

```
struct fstab_st {
  int numfiles;
  int tabsize;
  struct file_handle *fsystem;
};
```

dove `fsystem` è un puntatore ad un array di `file_handle` il cui ultimo elemento ha indice `numfiles-1`. Gli elementi della tabella vengono riempiti mano a mano che il server esamina le entry di `fsdir` e la dimensione effettiva dell'array (salvata in `tabsize`) è estesa dinamicamente quando necessario.

5.2 Threads

La funzione `main` del server (thread principale) è composta sostanzialmente da un ciclo infinito che attende le richieste di connessione da parte dei client sul socket del server. Quando un client si connette e completa l'handshake viene creato un nuovo thread `client_handler` (*worker*) che prende in gestione la conversazione mentre `main` rimane in ascolto di nuove richieste. I thread worker ricevono come argomento il puntatore ad una struttura `th_info` che contiene le informazioni necessarie per proseguire la conversazione con il client ed eseguire le operazioni

```
struct th_info_st {
  pthread_t tid;
  channel_t client_c;
  struct file_handle *handle;
  struct th_info_st *next;
};
typedef struct th_info_st th_info;
```

Il campo `client_c` denota il descrittore del socket di comunicazione con il client, mentre `handle` punta al `file_handle` per il dispositivo richiesto. Poiché per implementare la terminazione asincrona del server si è reso necessario tenere traccia degli identificatori di tutti i `client_handler` attivi,

gli oggetti di questo tipo vengono salvati in una lista globale: `tid` contiene l'identificatore del thread creato e il campo `next` punta all'elemento successivo nella lista. Il tipo di dato lista di thread è definito ed implementato nei file `thlist.h` e `thlist.c` (situati nella cartella `src/server`).

I thread `client_handler` ciclano indefinitamente in attesa dei comandi inviati dai client. Quando un client interrompe la conversazione, il thread rimuove la propria entry dalla lista dei thread attivi e termina l'esecuzione.

5.3 Gestione della concorrenza

Nella fase di indicizzazione viene compilata la tabella dei file system `fstab` con gli opportuni `file_handle`: ad ogni file system vengono associati un puntatore al dispositivo fisico ed una struttura di controllo che contiene la relativa tabella di allocazione. Per permettere a più client di interagire con lo stesso file system, è necessario che queste informazioni siano condivise dai thread e possono quindi verificarsi *race conditions* tra i `client_handler` che vogliono accedervi.

Il meccansimo di gestione della concorrenza prevede di dividere le operazioni offerte dal file system in operazioni di lettura (`ls`, `fread`) e scrittura (`mkdir`, `mkfile`, `append`, `cp`): l'osservazione è che gli accessi concorrenti in lettura non causano interferenze tra loro, mentre gli accessi in scrittura devono essere esclusivi dal momento che modificano il contenuto del dispositivo fisico e la tabella di allocazione. Questo problema di sincronizzazione (problema dei lettori-scrittori) è risolto con un semplice algoritmo che tiene traccia dei tipi di accesso al dato condiviso: la lock di lettura può essere acquisita da molteplici operazioni mentre quella di scrittura è esclusiva. Per impedire letture e scritture contemporanee, la lock di scrittura viene acquisita dalla prima operazione di lettura che accede al dato, e viene rilasciata dall'ultima operazione di lettura attiva che si conclude: in questo modo non è mai disponibile se ci sono dei lettori attivi, ed in caso di scrittura in corso un eventuale lettore rimane bloccato in attesa di acquisirla.

Questa politica di accesso ai dati è realizzata mantenendo per ogni device un contatore dei lettori attivi (il cui incremento/decremento è protetto da un `mutex`) ed un secondo `mutex` per proteggere gli accessi in scrittura; tali informazioni sono memorizzate come campi della struttura `file_handle`. Il meccanismo per acquisire i lock di lettura e scrittura è codificato attraverso le procedure `read_lock`, `read_unlock`, `write_lock`, `write_unlock` che ricevono come parametro il riferimento ad un `file_handle`. L'algoritmo implementato rende le operazioni di scrittura soggette a *starvation*.

5.4 Terminazione asincrona

Tra le specifiche del progetto è richiesta la possibilità di terminare il server in maniera asincrona durante l'esecuzione inviando un segnale oppor-

tuno (`SIGINT` o `SIGTERM`): il server deve portare a termine le eventuali richieste pendenti, terminare le conversazioni con i client ancora connessi ed interrompere l'esecuzione.

L'aspetto problematico di tale caratteristica risiede nel fatto che le funzioni per la comunicazione tra client e server (richieste di connessione e I/O sui socket) incapsulano chiamate di sistema bloccanti (`accept`, `read`, `write`).

Gestire la terminazione asincrona con un semplice signal handler che disattiva un flag globale – usato come protezione dei cicli di accettazione delle nuove connessioni e delle conversazioni con i client – comporta che la guardia non venga esaminata fino alla successiva iterazione, dal momento che i thread si sospendono sulle funzioni di cui sopra. Questo violerebbe la specifica di portare a compimento le sole richieste pendenti: il server dovrebbe invece interrompere le conversazioni momentaneamente "inattive" che causano la sospensione di alcuni `client_handler`. Inoltre, terminare i thread direttamente da un signal handler è una soluzione difficilmente gestibile: i segnali sono intercettati in maniera completamente asincrona, e terminare un thread quando un mutex non è stato rilasciato o parte delle operazioni sul file system non sono state completate può causare lo stallo del server o la corruzione del device in uso.

La soluzione adottata è stata quella di ricorrere alla cancellazione dei thread attivi. Questa scelta risolve infatti il problema delle system call bloccanti: `accept`, `read` e `write` sono definite dallo standard POSIX come *cancellation point*, e la cancellazione di un thread sospeso su una di queste chiamate viene eseguita immediatamente. A ciò si aggiunge il fatto che un thread può temporaneamente ignorare le richieste di cancellazione se ad esempio si trova ad eseguire una sequenza di operazioni durante la quali non deve essere interrotto

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &state);
/* sezione critica */
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &state);
```

In questo caso le richieste vengono esaminate solamente dopo che la cancellazione è stata riabilitata.

Nel codice proposto i segnali vengono utilizzati unicamente per innescare la procedura di cancellazione. In fase di avvio il server aggiunge alla propria maschera dei segnali `SIGINT` e `SIGTERM` invocando `pthread_sigmask`: questo blocca la loro ricezione in ogni thread del server attivato successivamente; a questo punto viene attivato un nuovo thread – `sig_handler` – che ha il compito di intercettare tali segnali in maniera sincrona invocando `sigwait`. Quando uno dei due segnali viene inviato al processo, `sig_handler` si risveglia, ed invia richieste di cancellazione al `main` e a tutti i `client_handler` in esecuzione, i cui identificatori sono memorizzati nella lista dei thread attivi.

Operazioni come l'inserimento di un nuovo thread nella lista, l'esecuzione di un comando sul file system e in generale qualunque frammento di codice che acquisisca un mutex, sono considerate sezioni critiche e sono protette dalla cancellazione. Le operazioni di clean-up (chiusura dei socket, rimozione dalla lista dei thread attivi) sono implementate installando nei thread dei *clean-up handler* che vengono invocati all'uscita.

6 Il processo client

Il client realizza una semplice interfaccia a riga di comando verso il server e viene lanciato con il comando

```
fats_client usr [device]
```

dove `usr` indica un nome utente e l'argomento opzionale `device` specifica il nome del device sul quale si vuole operare.

In fase di avvio il client invia al server una richiesta di connessione `MSG_CONNECT` specificando nel buffer il nome utente e l'eventuale device richiesto.

Una volta avviata la sessione con il server, viene lanciato un thread `server_listener` che si occupa di stampare a video i messaggi di risposta ricevuti sul socket. Questo thread ha anche il compito di terminare l'esecuzione del client se il server per qualche motivo interrompe la conversazione (in questo caso la chiamata `receiveMessage` che viene invocata su un socket per ricevere un messaggio restituisce il valore `SEOF`).

La gestione dell'input dell'utente è invece compito del thread principale. La funzione `handle_input` memorizza la stringa in ingresso in un array allocato staticamente e la passa alla funzione `parse_cmd` che ha il compito di identificare il comando specificato e, in caso sia valido, convertirlo in un `message_t` opportuno che viene poi inviato al server. Dal momento che il buffer in cui l'input è memorizzato ha dimensione fissa (definita dalla macro `MAXLINE` in `client_impl.h`), esiste un limite sul numero di caratteri che possono costituire un comando; questo limita ad esempio la quantità di caratteri scrivibili su un file da una singola invocazione del comando `fats_append`.

7 Compilazione del sistema

Il sistema si compila dalla root directory del progetto utilizzando l'utility `make`. I target del Makefile rilevanti ai fini della compilazione del sistema sono `build`, che compila il sistema ed elimina i file temporanei generati (file oggetto, etc ...) e `test`, che esegue alcuni semplici test di funzionamento del sistema e ne riporta l'esito sul terminale.

L'invocazione del comando '`make build`' genera gli eseguibili `fats_create`, `fats_server` e `fats_client`.

Quando viene lanciato, il server crea il socket di connessione nella cartella locale `tmp`.

7.1 Il client `fats`

I comandi del client vengono invocati esclusivamente su percorsi assoluti. La sintassi dei comandi e' la seguente

`mkdir dir`
: Crea la cartella `dir`, se alcuni elementi intermedi del percorso non esistono l'operazione fallisce

`mkfile file`
: Crea il file `file`, se alcuni elementi intermedi del percorso non esistono l'operazione fallisce

`ls dir`
: Elenca il contenuto della cartella `dir`

`append file string`
: Aggiunge in coda al file `file` il contenuto di `string`, escludendo gli apici che la delimitano

`fread file offset len`
: Legge `len` caratteri di `file` a partire da `offset` e li stampa a video

`cp source dest`
: Crea il file `dest` e vi copia per intero il contenuto di `source`

`quit`
: Interrompe la conversazione con il server ed esce dal programma

In caso di fallimento di un'operazione viene stampato sullo schermo un messaggio che descrive l'errore notificato dal server.

7.2 Test del sistema

Il test del sistema e' effettuato invocando il comando '`make test`' e consiste nell'esecuzione di alcune operazioni dimostrative, e nel confronto delle risposte del server con quelle attese. I file usati dai test sono contenuti nella cartella `test`: i comandi sono letti dai file `test*.input`, e le risposte del server sono confrontate con i risultati salvati nei file `test*.ok` . Il target `test` del Makefile invoca quattro target (`test1`, `test2`, `test3`, `test4`) il cui compito e' quello di testare il funzionamento del sistema con file system che usano ogni dimensione di blocchi ammessa. I test consistono ciascuno

nell'esecuzione di uno script shell che crea un file system, attiva un server fats, invoca le operazioni di prova salvando le risposte del server in `test/test*.output` per ogni run `test/test*.input`, confronta queste con i risultati attesi, notifica l'esito dei test all'utente e termina il server fats.

I run di prova simulano una semplice sessione:

- `test1x.input` verificano che le cartelle e i file vengano creati e listati correttamente
- `test2x.input` verificano la correttezza delle operazioni di scrittura e di lettura
- `test3x.input` verificano la correttezza delle operazioni di copia

La creazione di cartelle genera una directory table con dimensione maggiore di un singolo blocco; la scrittura e la copia interessano files con un'estensione di due blocchi. Nei test viene utilizzata una versione del client che attende 2 secondi dopo la ricezione del comando `quit` per assicurarsi di ricevere le risposte dal server prima di chiudere la comunicazione.