Progetto di Laboratorio di Sistemi Operativi

OleksiyNedobiychuk
matricola: 597455 Corso B

June 16, 2021

1 Introduzione

Il progetto è stato svolto con parti opzionali tranne che per compressione dei file. Si è pensato di realizzare la politica LRU con una struttura dati apposta per gestire i puntatori di file per trovare una vittima in tempo migliore di quello lineare. Si è cercato di esperimentare con questo progetto di conseguenza non si riesce a garantire che la politica LRU è stata implementata a dovere pero si è tentato di gestire al meglio la cache per avere un gruppo di file usati meno recentemente come vittime. Si è tentato di implementare con la fantasia e con l'obiettivo di fare qualcosa di non standard però che si comportasse bene. Si è usato una funzione di hash dell'autore: Dan Bernstein, presa da: http://www.cse.yorku.ca/ oz/hash.html. Si è basato il file storage, cache in memoria centrale sulla tabella hash di Jacub Kurzak presa da Didawiki infine si è usato le funzioni di utilità del professore Massimo Torquati.
Github: https://github.com/lesi-nedo/another_brick_in_the_wall.

2 files_s.c

Si è voluto sfruttare i vantaggi della tabella hash per implementare un file storage . Per avere mutua esclusione tra i diversi thread si è deciso di avere un lock per ogni file, quindi per modificare oppure leggere si deve acquisire il lock associato alla posizione in cui si vuole inserire il file. Per evitare problemi di inconsistenza cioé un thread potrebbe acquistare il lock della cella ma allo stesso tempo un altro thread potrebbe procurasi il lock di un blocco diverso con lo stesso pathname del file si è pensato se si vuole inserire allora si richiede il primo lock per assicurare che solo un thread alla volta può associare un blocco a un file per livello. La decisione è stata di allocare per ogni cella iniziale della tabella un recipiente con l'intento di avere sempre il primo lock inizializzato. Per evitare eventuali cicli di attesa infiniti la decisione è stata di liberare saltato il filename e data dalla struct senza rilasciare la memoria allocata per il recipiente, quando si vuole inserire un file si vede se il file è presente e si cerca le celle vuote salvandole in una variabile temporanea per poi utilizzare una qualunque presa dalla variabile. Purtroppo il prezzo che si paga è quello di ricerca, cioè si deve controllare più celle del dovuto. Si è posto il problema che le celle vuote potrebbero occupare troppo però testando si è osservato che le dimensione si stabilizza è si riempe le celle vuote quindi la dimensione è stata accetta per evitare il deadlock e non solo. Per rispondere alla richiesta della funzione **lock** e **unlock** dell'API del poggetto si è pensando di avere una variabile O_LOCK e OWNER per gestire questi casi.O_LOCK può essere settato a 1 o a 0 soltanto con il lock per evitare situazioni spiacevoli. Ogni nuovo file viene inserito prima nella file store per poi legare la posizione in cache con la cella nella tabella.

3 new.c

Rappresenta la cache che è organizzata a livelli di riferimenti, più il file è stato ricercato\scritto più il livello è basso quindi al livello 0 ci si troveranno i file più usati. Poiché il massimo numero di file è noto si alloca lo spazio necessario a contenere tutti i puntatori all'inizio per non fare inutilmente free e acquisire lock con il rischio del deadlock senza avere vantaggi. Supponiamo che la cache è quasi piena allora i seguenti passi sono svolti dalla funzione **cach_hash_insert_bind**: Un nuovo file viene inserito se è presente una cella vuota e il thread riesce ad acquisire il lock, oppure viene sostituito il file che non dovrebbe appartenere al livello 0 , quindi

lo sono anche i file che sono i più referenziati cioè hanno il numero minore della variabile **min_and_you_in**, però chiaramente non si può andare oltre lo 0. Si risolve questa situazione con la scansione del livello 0 fino a trovare un puntatore che dovrebbe essere rimosso poiché il numero di riferimento è maggiore uguale alla variabile **max_and_you_out**, cioè il numero di scritture letture fatte su queste file recentemente non permette di tenerlo alla posizione 0. Si calcola la nuova posizione in base al riferimento e si ricomincia il processo nel livello corretto. Se non si riesce a trovare nessun puntatore allora si è deciso di randomizzare il calcolo dell'indice dove cercherà, cioè si esce dal ciclo while si controlla se la cache è piena se si viene chiamato la funzione di rimpiazzamento oppure si chiama un'altra funzione che calcola un indice che è il punto di partenza per la ricerca.
Esistono due funzioni per la ricerca della vittima, una viene usata quando si raggiunge il limite di files in cache e l'altra quando il numero di bytes è maggiore o uguale al limite di bytes. Si parte dall'ultimo livello provando ad acquistare il lock di blocco per rimuovere il puntatore e il file dalla file store. Per testare l'uguaglianza dei puntatori, l'assenza del deadlock , la correttezza dei vari incrementi\decrementi, variabili che dovrebbero tenere traccia del numero di files in cache\file store si è scritto un test unit però è necessario avere il framework check, con make check_store si può avviare il controllo.

4 parsing_conf.c, thread_for_log.c

Si è cercato di essere efficienti anche nel parsing del file di setting attraverso la tecnica di hashing . Durante il processo di scansione si controlla la correttezza del file eventualmente si abortisce se si trova qualcosa che non dovrebbe esserci però con certi valori assegnati si potrebbe mandare in tilt il server. E stata presa la decisione di avere un thread per gesite il file di log, riportando ciò che era richiesto nel progetto. Ogni thread worker scrive le operazioni fatte attraverso la funzione **dprintf** nella pipe bloccante. Per testare il processo di parsing si è scritto un unit test però non riportato nel make principale poiché ritenuto non fondamentale.

5 api_sock.c

Racchiude le funzioni richieste dal progetto. openConnection salva in due variabili globali esterne il fd del socket e il fd prodotta dalla funzione epoll. La decisione di usare **epoll** è stata motivata dal fatto che la select si è vista a lezione quindi non si voleva fare copia e incolla, per la sua efficienza e anche per capire come funzionasse. Si è cambiato O_CREATE e O_LOCK in O_CREATE_M e O_LOCK_M. Se si chiama la funzione openFile con O_CREATE_M | O_LOCK_M si deve leggere dal fd la risposta per capire se il server ha selezionato una vittima per fare posto al nuovo file creato se non presente. Se si passa soltanto O_LOCK_M allora non si deve aspettare una risposta. Con la prima chiamata di openFIle il server genera un certo id per identificare il cliente e gestire i permessi, la funzione salva la chiave in una variabile globale. La decisione per rispondere alla richiesta del progetto di chiamare la writeFile solo dopo la openFile è stata quella d'incorporare la chiamata della openFile all'interno della funzione writeFile, la funzione chiama anche appendFile. Ogni chiamata dell'API invia al server una array di grandezza due contente l'operazione richiesta e l'id. Si rispettano tutte le richieste del progetto come per esempio che con la chiamata lock se non si riesce a settare la variabile O_LOCK a 1 per avere accesso esclusivo, il thread che serve la richiesta si mette in attesa fin quando non acquista la lock che puo essere anche infinita, pero a ogni ciclo testa se il cliente è ancora vivo uscendo dal ciclo in caso negativo, si esce anche quando si riceve il segnale SIGINT-SIGQUIT. Se eventualmente il cliente che detiene la risorsa chiama unlock allora un qualsiasi cliente in attesa può acquistare la lock è settare le variabili O_LOCK e OWNER.

6 bunch_of_threads.c

Racchiude le funzioni server-side dell'API e una funzione che è eseguita da ogni worker. Si è deciso di far comunicare i thread worker con il main attraverso una pipe non bloccante per scrivere file descriptors che sono pronte per essere ascoltate, e nell'array fds vengono scritti i fd che si devono servire, viene assicurato la correttezza per via del meccanismo di muta esclusione con l'ausilio della variabile condizionale le varie letture e scritture sull'array. Ogni nuova connessione viene modificata in non bloccante però comunque un thread

serve un cliente fin a quando la connessione è aperta. Se si supera il limite di connessioni possibili il main thread si mette in attesa chiamndo **sched_yield** fin quando non c'è spazio disponibili per scrivere il nuovo fd nell'array.

7 server: main.c

Come specificato nel progetto si cerca di usare un protocollo di richiesta risposta. Si è deciso di gestire i tre segnali con un signalhandler che assegna 1 se si è ricevuto il segnale SIGINT, SIGQUIT e 2 se SIGHUP alla variabile globale del tipo sig_atomic.

8 cliente: main.c

Per gestire i vari commandi passabili e vincoli specificati si è scelto di utilizzare la getopt . Per ogni commando si è specificato una macro, un array di struct di grandezza del totale dei comandi. Questa organizzazione ci permette di gestire in modo semplice e corretto i comandi però lo svantaggio è che non si eseguono secondo la sequenza passata da linea di commando ma si eseguirà secondo l'ordine del progetto quindi prima **-h**, **-f** e cosi via. Per non riscrivere sempre gli stessi passaggi si è deciso di creare quattro macro; una per eseguire le funzioni dell'API, una per gestire i file passata da linea di commando, su ogni file viene chiamato la funzione realpath e ad ogni risposta dal server che indica un errore oppure se si verifica un errore durante una chiamata di funzione si è deciso di non continuare ma uscire usando una macro per stampare un messaggio di fallimento ma anche se non è specificato **-p** ma è minimale, oppure si stampa con l'altra macro il successo soltanto se si è passato il commando **-p**. Per gestire i comandi **-d**,**-D** si è deciso di chiamare la funzione chdir con l'argomento dirname e quindi creare il file estraendo il nome dal real path ricevuto dal server. Il cliente per permettere di buttare via i file letti con il commando **-r** controlla se è stato passato il commando **-d** in caso negativo si chiama una funzione che ha come dirname **/dev** di default e scrive tutto in **null**.

9 Considerazioni sulla gestione degli errori

Il server comunica la cortezza oppure l'errore con il cliente attraverso un array di due posizioni, la prima posizione indica l'esito e lo si controlla con le macro IS_NOT_ERROR, IS_ERROR, IS_EMPTY in realta IS_EMPTY non viene distinta nell'API ma si concentra sulla seconda poszione dell'array poiché la decisione è stata quello di passare gli stessi errori di errno. Si è posta molta attenzione sul server e sulla possibilità di perdita di memoria quindi si è cercato di prevedere ogni situazione speciale con la minima perdita . L'operatore che è stato di utilità notevole è stato il goto. Ogni fallimento viene propagata fino alla funzione principale, con il goto ci si reca alla sezione EXIT si imposta la variabile time_to _quit a 1 quindi tutti thread usciranno rilasciando la memoria anche per questo motivo si è deciso d'impostare ogni fd come non bloccante tranne che per la fd di log. Poiché si reputa di aver gestito in modo coretto tutti i valori passabili all'API, si fa un minimo controllo sul input per poi lasciare l'accertamento alle funzioni controllando il risultato per ogni chiamata.

10 Considerazioni finali

Per rispettare la specifica del primo test si deve avere solo un thread però non si riesce a testare nel mio caso il coretto funzionamento della funzione lockFile e unlockFile quindi si è deciso di aggiungere un altro thread. Per ogni test esiste un file config.txt in una sottocartelle della cartella tests , si prende dalla sottocartelle e lo si muove nella cartella server per poi rimettere nella sottocartelle corretta pero in caso di errori potrebbe succedere che l'operazione non viene eseguita.