Relazione Progetto del Laboratorio di Sistemi Operativi

Emanuele Cuzari

Matricola: 579477

A.A. 2019/2020

Introduzione

Lo scopo del progetto è quello di realizzare un sistema che implementi un supermercato avente K casse e in cui entrino C clienti. All'inizio ci sarà un numero di casse iniziali prestabilito che indicherà quante casse dovranno essere aperte nel momento in cui il processo supermercato viene lanciato ed "entreranno" nel supermercato i C clienti previsti. I clienti, prima di inserirsi nella coda della cassa, spendono un certo tempo T nel fare acquisti nel supermercato (acquistano al massimo P prodotti), dopodiché si metteranno casualmente nella coda di una delle casse se hanno comprato dei prodotti, dove attenderanno il loro turno. Per ogni cliente verrà calcolato il tempo di attesa in cassa, ovvero il tempo fino a quando il cliente non viene servito dal cassiere; verrà poi, per ogni cliente, calcolato il tempo di servizio impiegato dalla cassa per servire il cliente. Fatto ciò il cliente verrà considerato come "servito" e uscirà dal supermercato. Se il cliente non ha acquistato nulla, non si metterà in coda e attenderà che il direttore gli dia il permesso per lasciare il supermercato (azione non richiesta se il cliente ha fatto acquisti). Non entreranno nuovi clienti fino a che non ne sono usciti almeno E, con $0<E<C$ (NOTA: non possono esserci nel supermercato più di C clienti). Il processo termina nel momento in cui arriva un segnale di SIGHUP o di SIGQUIT con due gestioni diverse a seconda del segnale ricevuto. Terminato il processo vengono stampati a video i dati dei clienti e delle casse, salvati precedentemente in un file di log, tramite l'esecuzione di un file script.

File di configurazione: config.txt e altri

Nel file di configurazione sono definiti tutti quei dati necessari a far partire correttamente il processo:

- K, per il numero di casse totali del processo
- C, per il massimo numero di clienti
- E, per il numero di clienti che entrano ed escono
- T, per il tempo massimo di acquisto (T > 10 millisecondi)
- P, per il numero massimo di prodotti acquistati (P > 0)
- S1, per indicare quando posso chiudere una cassa
- S2, per indicare quando posso aprire una cassa
- INTERVALLO, per il tempo di trasmissione delle casse
- TEMPO_GESTIONE_PROD, per il tempo di gestione di un prodotto da parte della cassa
- CASSE_INIZIALI, per il numero di casse aperte all'inizio
- FILENAME, per il nome del file di log

Se all'esecuzione non viene specificato nulla, il programma viene eseguito con il file congif,txt come default; se si specifica: nome_eseguibile –f nome_file_config, il programma verrà eseguito con il file di configurazione specificato.

Cassieri

Per cominciare, definisco tre variabili extern che utilizzerò per gestire la chiusura delle casse, una macro per controllare i valori NULL e due funzioni, una per calcolare il tempo di servizio e una per gestire l'invio della notifica al direttore.

Ho proseguito iniziando a calcolare il tempo di apertura della cassa e creando un ciclo "infinto" all'interno del quale uno dei primi passi è quello di controllare lo stato della cassa: questo perché nel caso di chiusura della cassa da parte del direttore interrompo il ciclo while.

Se la cassa risulta essere aperta, estraggo un cliente dalla coda e termino di calcolare il suo tempo di attesa in coda. A questo punto calcolo il tempo di servizio con la formula fornita da specifica (t_fisso + n_prodotti*t_gestione_prodotto, con t_fisso nel range 20-80 msec) e lo inserisco all'interno della coda unbounded che utilizzo per salvare tutti i tempi di servizio relativi a una cassa: questa scelta della coda unbounded è stata fatta perché, non sapendo quanti clienti il processo dovrà gestire in ogni esecuzione, era necessario poter inserire un tempo di tempi di servizio che fosse variabile, da molto piccolo a molto grande.

Successivamente, prima di chiamare la funzione msleep per far attendere al processo che passi il tempo previsto dal tempo di servizio calcolato, eseguo un controllo sulla durata del tempo di invio della notifica: questo perché, se il tempo di invio della notifica al direttore è minore del tempo di servizio, verrà prima inviata la notifica tramite la chiamata della funzione invia_notifica e poi verrà fatta una msleep della differenza tra il tempo di servizio e il tempo di invio, altrimenti verrà chiamata una msleep sul tempo di servizio e sottrarò quest'ultimo valore al tempo di invio prima di rieseguire il ciclo while relativo a questo controllo per l'invio della notifica. Successivamente il cliente viene risvegliato e termina la sua esecuzione.

Quando la cassa chiude o arriva un segnale tra SIGHUP e SIGQUIT, il direttore viene risevegliato come se la cassa avesse inviato la notifica al direttore e viene gestita la chiusura: se è dovuta all'arrivo di un segnale, nel caso di SIGHUP non entreranno nuovi clienti e servirò i rimanenti, nell'altro caso verranno solo risvegliati (tutto questo all'interno di un ciclo while dove eseguo attesa attiva fino a che tutti i clienti non sono usciti dal supermercato); se la chiusura è una chiusura standard, invece, faccio semplicemente cambiare cassa a tutti i clienti.

Cliente

Inizialmente, per il cliente ho definito una macro per controllare i valori a NULL e tre funzioni per l'attesa in coda, l'uscita dal supermercato e l'attesa di autorizzazione.

All'inizio, comincio con calcolare il tempo che il cliente passa nel supermercato e faccio attendere il tempo di acquisto dei prodotti tramite una chiamata di msleep (il tempo di acquisto è randomico e viene valutato nel main). Se il cliente ha acquistato dei prodotti faccio partire un while(1) all'interno del quale controllo se una variabile set_signal del cliente è stata settata a 1: questo controllo mi serve per sapere se il cliente deve terminare subito o no, cioè se deve o meno mettersi in coda.

Successivamente, in un altro while(1), il cliente sceglie la cassa in cui inserirsi in modo casuale, verificando se ha già controllato o no la cassa il cui indice è stato generato randomicamente: se ho controllato tutte le casse o ne ho trovata una aperta, esco dal while (in caso non abbia trovato una cassa valida in cui inserirsi, una volta uscito dal while della scelta esce pure dal while principale).

Se la cassa scelta è valida, il cliente viene inserito in coda tramite apposite funzioni di inserimento definite nel file CodaCassa.c e viene chiama la funzione wait_to_pay che mette in attesa il cliente su una variabile di condizionamento stabilita. Alla fine del while(1) controllo se il cliente deve cambiare cassa, questo per decidere se uscire dal ciclo o se il cliente deve scegliere nuovamente l'indice della cassa in cui inserirsi.

Se il cliente non ha acquisti, viene chiamata subito la wait_authorize, che lo mette in attesa fino a che non ha ricevuto il via libera dal direttore e successivamente viene chiamata la exit_market, per qualunque cliente non stia eseguendo il while(1), che conta quanti clienti sono usciti e manda una cond_signal al processo supermercato, che ripartirà quando il numero di uscite equivale a E.

Infine, prima di terminare viene calcolato il tempo totale che il cliente ha passato nel supermercato e vengono stampati i dati utili sul file di log. Ho scelto di far stampare i proprio dati a ogni cliente di volta in volta per semplicità di scrittura nel file supermercato.c e perché ho ritenuto fosse più logico in questo modo, simulando una specie di checkout del cliente.

Direttore

Anche per il direttore come per il cassiere, ho definito due variabili extern per la gestione dei segnali e due funzioni, una per aspettare la notifica e una per concedere l'autorizzazione ai clienti che vogliono uscire.

Inizialmente, ho costruito un ciclo while(1) al cui interno gestisco l'attesa della notifica, mettendo in attesa il direttore fino a che tutte le casse non hanno inviato la notifica: sebbene le casse inviino la notifica in modo asincrono, prima di far fare qualcosa al direttore ho preferito aspettare che tutte avessero comunicato i propri dati, per un scelta più accurata delle casse che vanno chiuse e quelle che vanno aperte.

Dopodiché viene eseguito l'algoritmo decisionale secondo il quale il direttore va ad aprire o chiudere una cassa: ho deciso di stabilire un tempo X per quando il direttore può chiudere una cassa, in questo modo certifico che una cassa appena aperta, prima di poter essere chiusa, deve essere rimasta aperta per almeno un tempo X. L'algoritmo controlla quindi se ci sono delle casse con al più un cliente in coda, se sì incrementa un contatore e dopo controlla se il numero di casse aperte è maggiore del minimo stabilito (ovvero 1) e se il contatore è maggiore o uguale alla soglia S1 stabilita nel file di configurazione. Se queste condizioni preliminari sono verificate, viene calcolato un tempo diff dato dalla differenza tra il tempo di ultimo aggiornamento della cassa e il tempo attuale calcolato; se questo è maggiore di X, allora verrà salvato l'indice della cassa come possibile cassa da chiudere. Se il numero di clienti in coda alla cassa non è minore o uguale a 1, allora controllo se in quella coda il numero di clienti è maggiore di S2, soglia di apertura; se è così, metto a 1 un flag per indicare la possibilità di apertura. Per l'apertura, se la cassa è chiusa, salvo l'indice della prima cassa chiusa che trovo, dopodiché ogni volta calcolo la differenza tra i tempi di ultimi aggiornamento della cassa di cui precedentemente ho salvato l'indice e quella attuale: apro quella chiusa da più tempo. In seguito gestisco le azioni da eseguire in caso di apertura e chiusura e in fondo al while(1) chiamo authorize, dove tramite una broadcast risveglio tutti i thread che aspettavano l'autorizzazione.

Quando arriva il segnale, esco dal while(1), autorizzo tutti i possibili clienti rimasti in attesa e termino il direttore (eseguo questa autorizzazione in quanto dopo l'arrivo del segnale potrei avere dei possibili clienti ancora in attesa).

Supermercato

All'interno del supermercato sono dichiarate una serie di variabili globali che sono condivise tra thread differenti (NOTA: sig_hup e sig_quit sono di tipo volatile sig_tomic_t perché vengono settate all'interno del signal handler e devono quindi essere signal safe).

Successivamente ho definito una serie di macro per il controllo dei valori nulli, per la stampa di configurazione e per l'inizializzazione delle mutex e delle variabili di condizionamento (questo perché essendo che queste macro vengono utilizzate molte volte, ho cercato di eliminare un po' di overhead che le chiamate di funzione al loro posto avrebbero potuto generare).

Inizialmente, viene eseguito il controllo sul metodo di esecuzione dell'eseguibile (se devo usare il file di configurazione di default o se è specificato tramite -f); subito dopo stampo i valori contenuti dal file di configurazione usato per l'esecuzione: ho fatto questa scelta per dare un'idea all'utente dei dati su cui sto lavorando.

Segue l'allocazione e l'inizializzazione di tutte le variabili che mi serviranno nel corso dell'esecuzione; dopodiché vengono inizializzate le strutture dei thread e vengono creati tramite pthread_create quest'ultimi.

Dopo l'inizializzazione preliminare, ho costruito un ciclo while(1) per l'esecuzione delle azioni svolte dal processo supermercato: in particolare attendo, su una variabile di condizionamento, che il numero di clienti usciti sia E (questo per rispettare le specifiche fornite); quando questo accade, riciclo i thread usciti (attendo la terminazione tramite pthread_join) e li reinserisco nel supermercato, con indice incrementato e valori settati nuovamente. Ho scelto di riciclare i thread per non sovraccaricare il sistema eccessivamente.

Quando uno dei segnali sopraggiunge, esco dal corpo del while e avviso tutti i nuovi clienti entrati di non mettersi più in coda (ATTENZIONE: ho scelto di far terminare immediatamente i clienti che non si mettono in coda una volta arrivato il segnale, questo perché ho reputato non necessario salvare le loro informazioni, dato che il supermercato sta chiudendo e loro non possono più essere serviti. Dunque, il numero di clienti entrati nel supermercato risulterà "maggiore" del numero di clienti che il file script stamperà a video, ma il numero di clienti corretto è quello riportato tra i dati del supermercato, dove tengo il numero di clienti effettivi entrati).

Infine attendo la terminazione dei vari thread, assicurandomi tramite una variabile stop_casse, che le casse iniziano a terminare solo dopo che tutti i clienti hanno lasciato il supermercato e controllando che il direttore sia l'ultimi dei thread a terminare prima che termini il processo. Stampo, poi, sul file di log i dati relativi alle casse e chiamo una funzione di cleanup per deallocare la memoria allocata.

Makefile e analisi.sh

Il makefile è strutturato come un makefile generale, con le sue regole di compilazione per produrre l'eseguibile (assicurandomi che la compilazione sia compatibile con lo standard c99) e dei phony targets per le due batterie di test inerenti a SIGHUP e SIGQUIT e per cancellare tutti i file generati dalla chiamate make all del makefile (dove all è un altro phony target per la generazione dell'eseguibile). Uso pure una variabile DEBUGFLAG per abilitare le stampe di debug.

Analisi.sh è invece il file script che uso per la stampa a video del file di log. Inizialmente tramite pidof catturo il pid del supermercato per far si che con tail lo script attenda la terminazione del processo prima di essere eseguito.

Eseguo dei controlli sull'esistenza dei file atti alla stampa del file di log e dopodiché inizio la lettura del file di log tramite un ciclo al cui interno eseguo la tokenizzazione di ogni linea letta usando come separatore ':'.

Successivamente tramite una serie di controllo vado a verificare quali dati sto leggendo dal file do log e quindi decido quali azioni eseguire su di essi e come stamparli.

Statlog.c e parsing.c

Statlog.c è il file che utilizzo per la generazione del file di log. E' composto da due funzioni: una per i dati delle casse e del supermercato, una per i dati dei clienti.

Facendo stampare i dati al cliente volta per volta, ho dovuto controllare se il cliente che scrive fosse il primo a scrivere, questo per decidere il metodo con cui aprire il file di log (w o a, a seconda che il cliente sia il primo o vada a scrivere in append sul file).

Parsing.c è invece il file che utilizzo per il parsing del file di configurazione. Al suo interno ho definito diverse macro per liberare la memoria, settare le variabili e controllare i valori nulli. In particolare ho utilizzato un hash per salvare i dati relativi ai vari campi del file di configurazione in quanto questa scelta mi comportava un accesso molto più rapido ai dati salvati e non mi era necessario dover eseguire un nuovo accesso all'hash per modificare i dati salvati.

Bibliografia

I file .h e .c relativi alla definzione della coda bounded e della coda unbounded sono stati presi dai file già esistenti presenti nelle soluzioni di alcune esercitazioni svolte durante il laboratorio di sistemi operativi nel corso dell'anno accademico 2019/2020 (sono state eseguite solo alcune modifiche atte alla configurazione dei file per quanto concerne le funzionalità del progetto in questione); i file .h e .c per l'implementazione del file di hash sono stati presi anch'essi da una delle esercitazioni svolte durante il suddetto laboratorio nello stesso anno accademico, ma a questi non sono state apportate alcune modifiche relative all'implementazione.

Il codice relativo alla funzione msleep è stato preso da internet dal sito StackOverflow al link: https://stackoverflow.com/questions/1157209/is-there-an-alternative-sleep-function-in-c-to-milliseconds