UNIVERSITA' DI PISA – A.A. 2020/2021

APPELLO STRAORDINARIO

RELAZIONE PROGETTO SISTEMI OPERATIVI

realizzato da

SIMONE IOZIA N.549108 CORSO A

1 Introduzione

Il progetto consiste nell'implementazione di un modello che rappresenta un supermercato, con K casse, gestito da un direttore che decide se aprire o chiudere le casse e dare il permesso di uscita ai clienti senza acquisti. I clienti che posso "entrare" nel supermercato sono potenzialmente infiniti se non fosse per l'arrivo dei segnali SIGQUIT o SIGHUP, che decidono la chiusura del supermercato secondo modalità diverse. Il modello è costituito da un'architettura client-server.

2 Implementazione Server: il Direttore

All'avvio del processo direttore, si configurano i vari parametri di configurazione passando per argomento durante la compilazione e l'esecuzione un file di testo che rappresenta il file di configurazione.

A seconda del tipo di test che si va a scegliere, il direttore può eseguire il processo supermercato come processo figlio tramite una **fork** e una **execl** oppure agendo come un vero e proprio server, indipendente dal processo supermercato.

A questo punto il direttore associa un **handler** ai segnali mediante **sigaction** e, oltre ai segnali del SIGQUIT e del SIGHUP, ricevuti dal S.O., troviamo il segnale di permesso di uscita del cliente, che inoltrerà lo stesso segnale ricevuto dal processo supermercato allo stesso, e il segnale

di chiusura del supermercato. Da sottolineare la presenza del flag **SA_RESTART** che permetterà di "riavviare" funzioni interrompibili, come readn, che, con l'arrivo di un segnale, possono avere strani comportamenti.

Dopo la gestione dei segnali, il processo direttore crea una **socket** e si mette in attesa di connessione del processo supermercato. Non appena il processo supermercato si connetterà, il processo creerà un nuovo thread (man_casse) che andrà a gestire la decisione di aprire o chiudere una cassa. Non appena riceverà il segnale di SIGQUIT o il segnale di chiusura del supermercato, il thread ritornerà (atteso dal thread main tramite una join) e, a seconda del test scelto, aspetterà che il processo supermercato si sia effettivamente chiuso o tramite una **waitpid** (caso fork) o con un while che cicla sul controllo del segnale di chiusura.

Assicuratosi che il processo supermercato si sia chiuso, il processo chiuderà la socket terminando la sua vita.

2.1 Thread man_casse

Il thread man_casse decide l'apertura o la chusura di una cassa ricevendo, tramite una **readn** (bloccante), dal processo supermercato un array di interi che rappresentano il numero di clienti in coda nelle varie casse.

In caso si verifichino le condizioni di apertura o chiusura di una cassa, il thread manderà un array di 2 posizioni, la cui prima posizione rappresenta la chiusura di una cassa mentre la seconda l'apertura di una cassa.

Se non è necessaria alcuna azione, manderà comunque un array contenente due 0.

Il thread ritornerò in caso di SIGQUIT, di chiusura del supermercato o della non-lettura della readn.

3 Implementazione Client: il Supermercato

Al suo avvio, il processo supermercato, come il direttore, setterà i vari parametri di configurazione tramite il file di configurazione.

Dopodichè allocherà, oltre ad alcuni array di sostegno, la struttura dati principale del modello, che rappresenta l'insieme delle casse, realizzata tramite un array di una particolare struct che contiene, oltre ai dati della cassa, una **lista concatenata** di struct (che rappresenta il cliente e i suoi dati) che costituirà la coda della cassa, e una lock che penserà a gestire al meglio la concorrenza dei vari thread per accedere ai dati, e in particolare, alla coda.

Allocate le strutture dati, il processo supermercato costruirà un file .log (stampato a schermo dallo script analisi.sh dopo la chiusura dei processi) che conterrà tutti i dati finali dei clienti e delle casse "a fine giornata".

Si connetterà alla **socket** del processo direttore e gestirà tramite un **handler** i segnali di SIGQUIT, SIGHUP e permesso di uscita cliente, tutti ricevuti dal processo direttore. Presenza del flag SA_RESTART.

Infine il processo supermercato crea i thread per la gestione delle casse e dei clienti e aspetterà il loro ritorno fino al momento dell'arrivo di un segnale di chiusura, dopo il quale deallocherà, chiuderà i vari file aperti o socket e manderà al processo direttore il segnale di effettiva chiusura.

3.1 Threads g_casse e cassiere_t

Il thread di gestione delle casse ha 3 funzioni:

1) Allocare e attivare i threads cassieri, il cui numero è specificato nel file di configurazione;

2) Attivare il thread **news_t**, che manderà gli array con il numero di clienti in coda alle casse al processo direttore ogni intervallo di tempo prestabilito tramite una writen, e riceverà il codice di chiusura/apertura/non far nulla tramite una readn e un array di 2 posizioni settato a 0 o a 1 in modo opportuno;

3) Chiudere la cassa dal minor numero di clienti in coda o aprire una cassa (in tal caso “riciclando” uno dei thread cassieri momentaneamente chiusi) in seguito all’arrivo di un codice dal direttore.

Dopo la loro attivazione, i threads cassieri (a cui è passato per argomento della pthread_create il loro id) si metteranno in attesa con una **wait** sulla condition variable *checkout_cond* dell’arrivo di clienti nella rispettiva coda; l’arrivo di uno di questi sveglierà il thread iniziando a eliminare ad uno ad uno i clienti in prima posizione nella coda, settando a 1 la variabile *loading* quando stanno svolgendo il servizio, e la variabile *servito* quando hanno terminato il loro lavoro con quel particolare cliente. All’arrivo di segnale SIGQUIT, il cassiere terminerà il servizio sul cliente in corso e chiuderà la cassa; invece all’arrivo di segnale SIGHUP, il cassiere completerà la richiesta di tutti i clienti in coda in quel momento prima di chiudere e terminare il thread.

3.2 Thread g_clienti e cliente_t

La funzione del thread g_clienti è quella di “immettere” nel supermercato sempre nuovi clienti, all’inizio in un numero fissato, e poi mano a mano che la variabile *cl_attivi* raggiunge il numero prestabilito nel file di configurazione (C-E) crea nuovi E threads clienti, riallocando le strutture dedicate.

Appena creato, il thread cliente (a cui viene passato come parametro il suo id) stabilirà quanti prodotti acquistare.

Se il numero generato in modo randomico sarà 0, allora il thread manderò un segnale al processo direttore al fine di far terminare quel thread cliente. Il processo direttore risponderà con un segnale uguale e il cliente, stampati i suoi dati sul file .log, potrà terminare.

Se il numero generato randomicamente non è 0, allora il cliente sceglierebbe in modo randomico una cassa inserendosi così nella sua coda. Durante questa attesa, ogni intervallo di tempo prestabilito, se il cliente non risultasse nelle prime 2 posizioni della coda, si attiverebbe un algoritmo che porta il cliente a scegliere la coda con meno clienti, a meno che quest'ultima non sia quella in cui il cliente vi è presente. Inoltre se durante tale attesa, la cassa del cliente si chiudesse, egli sceglierà un'altra cassa in modo casuale.

Avvisato dalla variabile *loading*, che il cliente "sta passando" i suoi acquisti dalla cassa, si pone in uno stato di **wait** (sulla condition variable *client_cond*) in attesa che il cassiere ponga a 1 la sua variabile *servito*.

A questo punto, stamperà i suoi dati nel file .log (tra cui il numero di casse visitate, salvate per mezzo di un array) e terminerà.

Terminerà immediatamente in caso di segnale SIGQUIT.

4 Librerie

4.1 Structs.c

Libreria che contiene la struttura dati del supermercato e un insieme di funzioni utili, come l'immissione o la rimozione di un cliente dalla coda, il conteggio dei clienti in coda o l'inizializzazione dei dati per casse e clienti.

4.2 Config.c

Libreria che contiene la struct per memorizzare i dati presenti nel file di configurazione, la funzione per leggere i dati al suo interno e la funzione che controlla che questi siano corretti.

4.3 Rwn.c

Libreria che contiene le funzioni readn e writen.

5 Makefile

Per compilare il programma, aprire il terminale nella cartella contenente i file principali e inviare il comando **make**, che compilerà i processi direttore e supermercato creando i loro corrispettivi file .o. Per eseguire il programma sono previsti 2 test:

1) **make test1**
   controllo tramite valgrind di memory leaks sul processo supermercato, processo direttore e supermercato indipendenti, arrivo di un segnale SIGQUIT dopo 15 sec.

2) **make test2**
   esecuzione dello script analisi.sh che stamperà a schermo il file .log in seguito all'esecuzione del processo direttore che, tramite una fork e una execl, creerà il processo supermercato, arrivo di un segnale SIGHUP dopo 25 sec.

Dopo la compilazione o l'esecuzione del programma, è bene ripulire la cartella con tutti i file creati tramite il comando **make clean**.