GIANLUCA PANZANI - CORSO A – 550358 – PROGETTO19/20

SUDDIVISIONE DEI FILE.C/FILE.H:

La suddivisione che ho scelto per i file é basata sulle funzionalitá offerte da questi.
Infatti il file supermercato.c, essendo il processo principale, ha il compito di creare tutti i processi secondari basandosi sui valori letti dal file di configurazione, e di deallocarli a fine esecuzione. Inoltre ha il compito di gestire i segnali SIGHUP e SIGQUIT attraverso la modifica della gestione di default di tali segnali che avverrá settando le variabili signalH o signalQ a 1 in base al tipo di segnale ricevuto.
Il file cassa.c contiene l'implementazione delle funzioni pushCassa, popCassa, chiudiCassa, apriCassa e cashUpdate, oltre all'implementazione dei thread fcasse e cashHandler che sfruttano le funzioni appena citate. Il thread fcasse contiene l'implementazione di ogni singola cassa (che da adesso chiameró per semplicitá: thread cassa), mentre il thread cashHandler si occupa della gestione delle casse stesse.
Il file cliente.c contiene le funzioni popDir e pushDir, che sono una versione di pop e push riadattate alla gestione della coda dei clienti senza acquisti, e la funzione clientUpdate, che gestisce i clienti usciti dal supermercato inserendo i dati ottenuti da questi nel file di log col fine di poterli sostituire con altri (se necessario). Inoltre contiene i thread: fclienti (che per semplicitá chiameró: thread cliente), clientsHandler e noBoughtHandler. Questi corrispondono rispettivamente: all'implementazione di un singolo cliente, alla gestione dei clienti con numero di prodotti acquistati diverso da 0 e alla gestione dei clienti senza acquisti.
Il file myPthreads.c contiene le implementazioni delle funzioni pthread che ho maggiormente usato nel progetto col fine di rendere l'implmentazione piú rapida e il codice piú leggibile.
Il file shared_data.h contiene tutti i dati condivisi (variabili esterne, struct, dichiarazioni di funzioni, ecc..) tra i file appena elencati.

LOGFILE E FILE DI CONFIGURAZIONE:

Il formato del file di configurazione dovrá essere il seguente: “logfileName:K:C:E:T:P:k:Tp:Td:S1:S2”.
Il file supermercato.c si occuperá di leggere le informazioni dalla prima riga del file di configurazione e di interpretarle nel modo corretto. I valori che ho deciso di aggiungere a quelli richiesti dal progetto sono k, il numero di casse aperte al momento dell'apertura del supermercato, Tp, il tempo che impiega ogni cassa ad elaborare ogni singolo prodotto, e Td, il tempo che indica ogni quanto una cassa invierá un segnale al cashHandler (che dovrá effettuare controlli per chiudere o aprire altre casse).
Il logfile sará scritto, stampato, eliminato e ricreato (vuoto) ad ogni esecuzione del programma. La scrittura avviene grazie alle funzioni cashUpdate e clientsUpdate, mentre la stampa, l'eliminazione e la creazione del nuovo file sará affidata allo script analisi.sh. Se nel file di configurazione viene specificato il nome di un file diverso dal nome di default (logfile) questo non verrá eliminato. In tal caso spetta all'utente l'eventuale rimozione del file. La decisione é stata presa in modo tale da permette all'utente di avere la possibilitá di salvare i dati ottenuti dall'esecuzione del programma.

SHARED_DATA.H:

Questo file é incluso in tutti i file “.c” del progetto. Contiene: la dichiarazione delle struct (List, Queue, Cassa, Cliente e Pthread_t), la dichiarazione delle variabili condivise da 2 o piú file, la dichiarazione delle funzioni condivise (tutte le funzioni Pthread definite nel file myPthreads.c e la funzione pushCassa in quanto usata sia nel file cliente.c che cassa.c) e la dichiarazione delle funzioni che verranno passate alla pthread_create al momento della creazione dei thread.

STRUCT USATE:

Ho scelto di usare un vettore di casse di dimensione K e un vettore di clienti di dimensione C. Quest'ultimo perché a mio parere la loro gestione risulta molto piú semplice. In pratica, quando un cliente esce, il thread cliente non termina ma rimane in attesa affinché vengano inseriti i dati del cliente uscito nel file di log, per poi essere “eliminato” e sostituito da un altro (con ID diverso). Se non avessi fatto cosí avrei dovuto creare ed eliminare thread cliente concorrentemente a tutte le altre operazioni e quindi sarebbe stato, sia piú dispendioso a livello di risorse, che piú complesso a livello di implementazione.
Per quanto riguarda la struct Pthread_t, ho scelto di usarla sia per i clienti che per le casse in modo tale da permettere il passaggio dell'indice del vettore cassa e del vettore cliente come argomento dei rispettivi thread.
Mentre le struct Queue e List risultano essenziali per implementare le code in cassa (una coda per ogni struct Cassa), la coda dei clienti senza acquisti e la lista delle casse aperte (chiamata “open”). Quest'ultima risulta essenziale per la chiusura/apertura delle casse e per la scelta della coda da parte del cliente.

DIRETTORE:
Il direttore é suddiviso in 4 parti:

1) Processo principale (supermercato.c):
   Al momento dell'esecuzione puó ricevere come argomento il nome del file di configurazione. Altrimenti il programma leggerá di default il file “config.txt”.
   Successivamente si metterá in attesa dei segnali SIGHUP e SIGQUIT con l'uso delle funzioni “signal()”. La gestione del segnale SIGHUP é definita nella funzione “signalH_Set”, mentre quella di SIGQUIT in “signalQ_Set”. Al momento della ricezione del segnale, ognuna di queste funzioni setta la rispettiva variabile a 1 (signalQ o signalH in base al segnale ricevuto).
   A questo punto avviene la lettura e il parsing della prima riga del file di configurazione. Se non sono rispettati gli intervalli che ho definito (commentati nel file di configurazione) il programma terminerá con errore.
   Adesso avverrá la creazione dei i thread e successivamente non fará niente finché non termineranno. Quando i thread avranno terminato la loro esecuzione avverrá la deallocazione di tutte le risorse allocate attraverso la chiamata della funzione “clean_all”. Fatto questo il programma terminerá.
2) Thread cashHandler:
   Questo thread si occupa della gestione delle casse. Ad ogni ciclo while si mette in attesa incondizionata finché non riceve un segnale. Questo riattiva il cashHandler affinché faccia i controlli sulle casse per la loro apertura/chiusura in base alla dimensione della propria coda.
   Per l'apertura di una cassa viene chiamata la funzione “apriCassa”, che aprirá la prima trovata chiusa in ordine di indice da i = 0 a K-1. Mentre per la chiusura di una cassa viene chiamata la funzione “chiudiCassa” la quale é un pó piú complessa della prima. Infatti questa deve inizialmente rimuovere dalla lista delle casse aperte la cassa da chiudere, poi deve scegliere la cassa con dimensione di coda minore, spostare i clienti dalla cassa chiusa a quella appena scelta ed infine aggiornare i campi delle struct di entrambe le casse modificate.
   Il cashHandler terminerá quando si attiverá uno dei flag di uscita, signalQ o signalH, e al contempo la somma dei clienti in coda alle casse sará uguale a 0. A quel punto dovrá risvegliare eventuali clienti rimasti nel supermercato e tutte le casse che erano in attesa e pronte ad uscire.
   A questo punto, prima della terminazione, avremo due nanosleep (una prima del risveglio delle casse e l'altra dopo). La prima nanosleep é dovuta al fatto che i clienti dovranno avere il tempo di uscire prima che le casse vengano risvegliate per la chiusura. Mentre la seconda é dovuta al fatto che, affinché le casse possano essere inserite correttamente all'interno del logfile, dovranno terminare la propria esecuzione. Se questo non avvenisse verrebbero inseriti dei valori uguali a 0, e di conseguenza errati, all'interno del logfile.
   Infatti, successivamente alla nanosleep, avverrá la chiamata della funzione cashUpdate la quale avrá il compito di inserire nelle ultime righe del file di log le informazioni ricavate dalle casse.
3) Thread clientsHandler:
   Questo thread direttore si occupa della gestione dei clienti. Ogni volta che un cliente esce dal supermercato (ovvero ogni volta che termina il ciclo acquisti-coda-uscita) lo segnala al clientsHandler il quale viene attivato ed effettua un controllo. Se il numero di clienti nel supermercato é inferiore o uguale a C-E chiama la funzione “clientUpdate”, la quale avrá il compito di aggiornare il file di log con le informazioni dei clienti usciti e permettere a nuovi clienti di entrare. I nuovi clienti avranno la stessa posizione (nel vettore clienti) dei clienti usciti ma risulteranno diversi grazie all'assegnazione di un nuovo ID. Quest'ultimo peró verrá aggiornato nel thread cliente. Fatto ció tornerá in attesa incondizionata finché un altro cliente non uscirá dal supermercato il quale lo sveglierá nuovamente.
   L'uscita dal while avverrá quando i clienti usciti dal supermercato saranno esattamente quanto la capienza di questo, ovvero C.
   Uscito dal ciclo segnalerá ai thread noBoughtHandler e cashHandler che anch'essi possono terminare in quanto i clienti usciti sono C.
4) Thread noBoughtHandler:
   Questo é l'ultimo dei thread direttori ed ha il compito piú semplice, ovvero quello di gestire i clienti senza acquisti. Essendo raro che un cliente non faccia acquisti é molto difficile che ci sia piú di un cliente nella coda gestita da questo thread (dirqueue), peró é comunque possibile ed é quindi gestita esattamente come le altre.
   L'attesa é condizionata e il risveglio avviene solo da parte del segnale di un cliente con 0 acquisti il quale ha appena chiamato la funzione “pushDir” per l'inserimento nella coda. Ho preferito distiguere

le push/pop in cassa dalle push/pop in questo thread per il fatto che nel primo caso si devono anche aggiornare campi specifici della struct cassa, cosa ovviamente non necessaria nella coda senza acquisti.
L'uscita é causata dalla signal inviata dal clientsHandler che avverte dell'avvenuta ricezione di uno dei segnali SIGHUP/SIGQUIT e dell'uscita di tutti i clienti dal supermercato.

THREAD FCLIENTI:
Questo thread svolge le funzioni del cliente nell'indice del vettore clienti passatagli come argomento. Infatti ho creato un vettore di struct Pthread_t contenente l'ID del thread cliente e l'indice corrispondente nel vettore clienti. Si é rivelata necessaria questa scelta in quanto se avessi passato il valore “i” del ciclo for sarebbe stato scorretto visto che questo viene continuamente modificato durante i cicli successivi, non permettendo quindi l'accesso al cliente corretto.
Prima del ciclo infinito ho definito alcune variabili: msec, tstart, tend, i e seed.
Ho usato “msec” per l'assegnamento del valore del tempo passato in coda dal cliente, “tstart” e “tend” (di tipo struct timespec) per calcolare il tempo passato in coda (permesso dalla funzione clock_gettime), “i” per distinguere i clienti rientrati nello stesso thread dai precedenti e “seed” per il calcolo dei valori random con la funzione rand_r.
Vorrei specificare che la funzione clock_gettime non fa parte del corso ma che, dopo molte ricerche sul browser, mi é sembrata la funzione piú adatta per il calcolo dei tempi (con l'uso della variabile CLOCK_REALTIME).
All'inizio del ciclo viene impostato il valore dell'ID del cliente con l'uso della funzione pthread_self(), la quale restituisce un valore univoco identificativo del thread. Peró non poteva bastare in quanto i nuovi clienti avrebbero avuto lo stesso id dei precedenti elaborati sullo stesso thread. Quindi é per questo che ho scelto di moltiplicare il valore restituito dalla funzione per la variabile “i” (che viene incrementata ad ogni ciclo). Cosí facendo l'ID di ogni cliente risulterá unico.
Successivamente ho calcolato (con rand_r) il valore corrispondente al tempo impiegato dal cliente nel fare acquisti e poi ho calcolato (sempre con rand_r) il numero di prodotti acquistati. A questo punto l'if distingue le diverse gestioni del cliente in base al numero di prodotti\: se uguali a 0 finisce nella coda del direttore (thread noBoughtHandler), altrimenti finisce nella coda di una cassa la quale viene decisa randomicamente scorrendo la lista delle casse aperte.
Vorrei far notare che in tutto il progetto ho applicato la mutua esclusione solo se la modifica di un valore condiviso implicava il rischio che venisse letto o modificato in altre parti del programma. Ma, ad esempio, nel caso dell'incremento “clienti[index].ncode++;” (come in altri casi) non vi é necessitá di mutex in quanto valore mai letto o modificato se non al momento dell'uscita dal supermercato del cliente stesso.
Tornando al thread, fatta la push sulla cassa scelta (o sulla coda dei clienti senza acquisti), il cliente si mette in attesa settando il valore “tstart” con la funzione “clock_gettime”. Al momento della fine dell'attesa viene settato il valore “tend” con la stessa funzione. Facendo adesso la differenza dei tempi di sistema ottenuti, il calcolo restituirá il valore del tempo passato in coda.
A quel punto viene avvertito il clientsHandler che un cliente é uscito, il quale fará i controlli visti precedentemente, e poi, il cliente, uscirá dal supermercato mettendosi in attesa. Se il cliente viene risvegliato significa che il clientsHandler ha provveduto ad inserire i suoi dati nel file di log e a ripulire i campi del cliente settandoli a 0 (o comunque al valore assegnatogli nel main al momento della inizializzazione). Adesso puó entrare come nuovo cliente nel supermercato.
Il thread cliente terminerá al momento della ricezione di uno dei due segnali. Nel caso della ricezione di SIGQUIT il cliente uscirá immediatamente al di lá della sua situazione all'interno del supermercato, altrimenti dopo l'attesa in coda.

THREAD FCASSE:
Nel caso delle casse ho definito inizialmente le coppie di variabili “tstart”/“tend” e “dirstart”/“direnD” (di tipo struct timespec) per il calcolo, rispettivamente, del tempo di apertura totale delle casse e dell'intervallo di tempo oltre il quale deve essere risvegliato il thread direttore (cashHandler) affinché possa fare i suoi controlli sulla dimensione delle code. Oltre a queste ho definito le varibili “t1” e “t2” (sempre di tipo struct timespec) le quali rappresentano rispettivamente il tempo compreso tra 20 e 80 ms che la cassa deve attendere per ogni cliente e il tempo per l'elaborazione di un singolo prodotto. Inoltre ho definito un flag “openflag” che indica lo stato precedente della cassa. Se la cassa va in attesa e risulta essere chiusa, ed abbiamo “openflag = 1”, significa che era aperta ed é stata chiusa. Se invece va in attesa e la cassa risulta essere chiusa, ma “openflag = 0”, allora era giá stata chiusa. Questo é utile per decidere se far risettare o

meno il tempo “tstart” con la “clock_gettime” e aggiornare il campo “time” della cassa. Questo perché altrimenti verrebbe calcolato anche il tempo in cui la cassa viene risvegliata ma, essendo ancora chiusa, torna in attesa. Se non fosse usato il flag “openflag” risulterebbero tempi maggiori del dovuto.
Prima di entrare nel ciclo vengono settati i valori di “tstart”, “dirstart” e l'ID della cassa (quest'ultimo con pthread_self()).
Successivamente si entra nel ciclo infinito in cui, ad ogni interazione viene elaborato un cliente ed eventualmente risvegliato il cashHandler (in base al tempo passato). Infatti viene mandato il segnale solo se la differenza tra “direnD” e “dirstart” é maggiore o uguale al valore “Td” definito nel file di configurazione.
A questo punto viene fatta la pop sul cliente in coda alla cassa per poi segnalare a questo che puó uscire dal supermercato (il quale nel frattempo era in attesa).
Il thread cassa terminerá in due casi: se é stato ricevuto uno dei segnali e al contempo il numero dei clienti usciti é uguale a C, oppure se viene risvegliato dal cashHandler nel momento in cui, sia i thread clienti che i thread gestori dei clienti, sono stati chiusi. Nel secondo caso c'é da specificare che l'attesa era dovuta al fatto che la testa della coda in cassa era NULL oppure che la cassa era chiusa ancor prima di ricevere uno dei segnali di uscita.
Nel momento in cui una cassa esce dal while deve aggiornare il tempo di apertura ed incrementare il numero di chiusure.
Siccome i thread cassa sono gli ultimi thread rimasti aperti (oltre al cashHandler che peró é in fase di terminazione) non c'é bisogno di inviare segnali. Quando le casse terminano potrá avvenire la deallocazione nel main ed infine la terminazione del processo principale.

SCRIPT ANALISI.SH:
Lo script, al momento dell'esecuzione, deve ricevere come argomento il file di log.
Questo script permette il parsing del file di log e la sua stampa a video. Visto che i dati nel file di log sono stati inseriti separando le informazioni dal simbolo “/”, allora basterá fare una sostituzione di quel carattere con un “\n” e stampare il tutto.
Infine, elimina il “logfile” e ne crea uno vuoto con lo stesso nome. Se peró l'utente volesse usare un proprio file di log non verrebbe eliminato a patto che il nome sia diverso da “logfile”, anche se questo implicherebbe la modifica del Makefile sostituendo il nome del file passato come argomento ad analisi.sh.

MAKEFILE:
Nel Makefile ho definito i seguenti target: compile, test, testQ, all, supermercato.o, cassa.o, cliente.o, myPthreads.o e clean.
Tutti i target “.o” sono stati creati per la ricompilazione di tutti i file con estensione “.c” attraverso l'uso delle regole implicite, mentre tutti gli altri target sono fittizi (phony).
Il target “compile” é quello di default ed ha, nella dependency list, i target “all” e “clean” rispettivamente per la compilazione dei file e per la pulizia della directory dai file generati.
Il target “all” ha nella dependency list tutti i file “.o” ed ha come comando:

```
$(CC) $(CFLAGS) -o supermercato $^
```

In cui CC e CFLAGS sono varibili definite ad inizio file nel seguente modo:

```
CC=gcc
CFLAGS=-Wall -pthread -g -lm
```

Il target “test” (che corrisponde al target test2 indicato nel progetto) ha il compito di eseguire il programma, attendere 25 secondi, uccidere il processo col segnale SIGHUP, attendere la terminazione del processo ed infine eseguire il file analisi.sh che stamperá i valori scritti sul file durante l'esecuzione.
Il target “testQ” fa esattamente le stesse cose del target “test”ma a differenza di questo invia il segnale SIGQUIT dopo 15 secondi di attesa.

ALCUNE OSSERVAZIONI:
- Ad ogni variabile globale creata nei file con estensione “.c” corrisponde un commento che spiega esattamente il suo utilizzo (salvo quelle ovvie).
- Le mutex usate sono: una per ogni cassa, una per ogni cliente, una per il cashHandler (mtxdirK), una per il clientsHandler (mtxdirC), una per il noBoughtHandler (mtxdirqueue), una per la lista di casse aperte (mtxopen) ed infine una per i clienti usciti (mtxclienti). Quindi in sostanza una per ogni thread. Ad ogni mutex é associata una condition variable tranne che per le ultime due elencate le quali non ne hanno bisogno.
- Nella stampa del file di log non é facile distinguere l'ID delle casse dall'ID dei clienti, quindi ho

pensato di fare una stampa alternativa in cui abbiamo “C:” prima dei clienti e “K:” prima delle casse. Peró, visto che non rispettava il formato di output richiesto dal progetto, ho pensato di lasciarlo come commento appena dopo la stampa effettiva. É possibile cambiare il commento con l'altra fprintf per vedere l'output personalizzato. In ogni caso le casse vengono stampate a fine file di log, quindi si troveranno nelle ultime righe.

- Per accedere alle informazioni di un solo cliente o di una sola cassa basta l'indice del vettore. Quindi é per questo che ho implementato il codice comunicando tra i vari thread solamente con l'uso dell'indice dell'elemento a cui si vuole accedere (cliente o cassa che sia). Ad esempio, alle push non passo il cliente e la cassa in cui effettuare l'inserimento ma solo gli indici di questi.
- Dall'output ottenuto dalla stampa del file di log, possiamo notare che il numero di chiusure delle casse é elevato solo nelle ultime due (talvolta anche solo una delle due). Peró questo é normale perché é una conseguenza logica dell'implementazione della chiusura/apertura delle casse. Infatti la chiusura avviene sulla cassa “S1esima” con dimensione di coda inferiore a 2. Quindi é molto raro che accada tra le prime casse in quanto aperte per prime e giá colme di clienti. L'apertura avviene invece sulla prima cassa chiusa da 0 a K-1. Di conseguenza l'apertura e la chiusura delle casse ricadrá quasi sempre sulle stesse (ovvero quelle con indice maggiore). Non ho risolto questo “problema” perché andrebbe contro le specifiche indicate dal progetto e perché comunque non influisce sul funzionamento del programma.
- É possibile attivare o disattivare le printf nei thread cambiando il valore della macro “M_PRINTF” nel file “shared_data.h”. Se la macro ha valore 1 allora verranno effettuate le stampe, mentre se ha valore 0 no.