Nicola Coltelli, 581087
Relazione progetto SOL

a.a. 2019/2020

FOLDERS

- La cartella '**bin**' contiene il programma 'supermarket' prodotto dalla compilazione tramite comando make.
- La cartella '**config**' è adibita a contenere i files di configurazione utilizzati dal programma, incluso quello di default richiesto dalla specifica.
- La cartella '**include**' contiene i file headers contenenti la specifica delle funzioni utilizzate dai rispettivi file .c ed alcune macro di utlilità.
- La cartella '**lib**' contiene la libreria dinamica 'libfifo_unbunded.so' prodotta dalla compilazione tramite comando make.
- La cartella '**logs**' contiene i files di log prodotti dall'esecuzione del programma e dagli scripts di testing.
- La cartella '**script**' contiene lo script analisi.sh richiesto dalla specifica del progetto e altri scripts usati in fase di testing del programma.
- La cartella '**src**' contiene i files .c, ossia l'implementazione del programma, il codice sorgente della libreria 'fifo_unbounded' e di alcune funzioni di utilità contenute in 'utils.c'. La cartella contiene inoltre i file oggetto prodotti dalla compilazione tramite comando make.

MAKEFILE

Il **makefile**, tramite il comando make, permette di creare il file eseguibile 'supermarket' all'interno della cartella 'bin', la shared library 'libfifo_unbounded.so' all'interno della cartella 'lib', e i files oggetto all'interno della cartella 'src'. Oltre ai phony **all**, **cleanall** (che svolge le funzionalità richieste a 'clean' nella specifica) e **test** (che corrisponde a test2 nella specifica, in quanto test1 non era richiesto), sono presenti i phony start, startandquit, memory, memoryquit, clean e cleanlogs. Sono di seguito descritti i phony non richiesti nella specifica.

start: avvia il supermercato e dopo 25 secondi invia un segnale di SIGHUP al supermercato.

startandquit: avvia il supermercato e dopo 25 secondi invia un segnale di SIGQUIT al supermercato.

memory: avvia Valgrind con all'interno il supermercato e dopo 25 secondi invia un segnale di SIGHUP a Valgrind. Valgrind è lanciato con il parametro --show-leak-kinds=all.

memoryquit: avvia Valgrind con all'interno il supermercato e dopo 25 secondi invia un segnale di SIGQUIT a Valgrind. Valgrind è lanciato con il parametro –show-leak-kinds=all.

clean: rimuove il file eseguibile supermarket e la shared library libfifo_unbounded.so. Il phony non coincide con il 'clean' richiesto dalla specifica, che è implementato in cleanall.

cleanlogs: rimuove tutti i files '.log' contenuti all'interno della cartella 'logs'.

FILE DI CONFIGURAZIONE

Il file di configurazione di default è 'config.ini', contenuto all'interno della cartella 'config'. È possibile specificare un file di configurazione personalizzato avviando il programma con l'opzione '-f' seguita da il path del file. Nel file possono essere inserite commenti preceduti da '#' e possono essere lasciate righe vuote. Il file contiene i seguenti parametri di configurazione.

- **C**: customers limit.
- **E**: customers threshold.
- **T**: maximum time a customer can spend shopping inside the supermarket.
- **P**: maximum number of products a customer can buy.
- **F**: frequency with which the cashiers will report to director in msec.
- **W**: if a cashier has <= customers in queue is considered below minimum.
- **X**: if a cashier has >= customers in queue is considered above maximum.
- **Y**: if the total of below minimum cashiers is >= we need to close a cash desk, if possible.
- **Z**: if the total of above maximum cashiers is >= we need to open a cash desk, if possible.
- **I**: path del file di log del supermercato.
- **L**: path del file di log dei cassieri.
- **M**: path del file di log dei clienti.
- **N**: path del file di log del direttore.

LOGGING
Il programma utilizza file di log aggiuntivi oltre al file richiesto dalla specifica. Il file richiesto dalla specifica è chiamato '**supermarket.log**', mentre viene fatto uso aggiuntivo dei files '**cashiers.log**', '**customers.log**', '**director.log**'. Tutti i file, ad eccezione del direttore che è utilizzato solamente dal thread direttore, sono protetti da mutex.

- Nel file '**cashiers.log**' viene registrato l'avvio di ogni thread cassa, l'operazione di servizio di un cliente, la chiusura e apertura della cassa, e la chiusura del thread.
- Il file '**customers.log**' registra l'avvio di ogni thread cliente, la cassa a cui si sta accodando un cliente o la richiesta del permesso di uscire al direttore nel caso in cui abbia 0 prodotti, la corretta ricezione del permesso di uscire, il corretto servizio da parte della cassa, la segnalazione da parte del direttore del cambio di cassa, e la terminazione del thread.
- Il file '**direttore.log**' non viene utilizzato, ma è stato comunque lasciato per future aggiunte.
- Il file '**supermarket.log**', anch'esso protetto da mutex, logga i dati richiesti dalla specifica. Il formato seguito è il seguente (i cambi sono separati da tab all'interno del file):
  Cliente (C=customer): C ID tempoInSupermercato tempoInCoda codeCambiate prodottiAcquistati
  Casssiere (K=cashier): K ID ClientiServiti ProdottiElaborati NumeroChiusure
  Cassiere che serve cliente (KC=cashier customer): KC IDCassiere IDCliente TempoServizio
  Turno del cassiere (KS=cashier shift): KS ID TempoApertura

SUPERMERCATO
Il thread main si occupa della fase di configurazione e di inizializzazione del supermercato. La prima operazione eseguita è la gestione dei segnali SIGHUP e SIGQUIT tramite variabili globali condivise all'interno di tutto il supermercato. Si noti che sighup_status e sigquit_status sono le uniche variabili globali all'interno del programma. La scelta di limitare l'uso di variabili globali è dovuta al tentativo di rendere il codice chiaro e di facile comprensione, in quanto lo scope di ogni funzione è limitato solamente ai suoi parametri.

Di seguito viene creata la cartella adibita al logging se non già presente ed un controllo verifica se con il parametro -f viene passato un file di config non di default.
Dal file vengono presi i parametri di configurazione. Il formato dei parametri all'interno del file è "ID=VALORE". L'ordine in cui i parametri sono scritti all'interno del file non è rilevante, possono essere aggiunti commenti se preceduti con '#' e le righe vuote sono ignorate dal programma.

Una volta presi in input i parametri vengono effettuate le inizializzazioni delle strutture dati e dei threads necessari al funzionamento del supermercato. La prima inizializzazione è quella dei files di log, che sono protetti da mutua esclusione per poter essere scritti da più thread concorrentemente. Per praticità, è stata creata un'apposita struct chiamata 'xlog' contenente il file pointer e la mutex.

In seguito vengono generati, nell'ordine, i cassieri, il direttore e un gruppo iniziale di clienti. Ognuno di questi thread viene generato da un'apposita funzione inizializzatrice che restituirà una struct contenente solo i dati essenziali che devono essere condivisi tra i threads.

Successivamente viene creato un thread 'entrata' di supporto che si occupa di attendere, utilizzando attesa passiva segnalata da ogni cliente al momento dell'uscita, che i clienti scendano sotto C-E per poterne reimmettere di nuovi e tornare ad avere C clienti. Durante il reinserimento di nuovi clienti potrebbe accadere che il numero complessivo di clienti scenda leggermente sotto la soglia indicata nella file di configurazione; questo è ammissibile da specifica. Il thread 'entrata' termina al momento della ricezione di uno dei due segnali.

Una volta che il thread 'entrata' è stato creato, le inizializzazioni sono terminate e il supermercato è operativo. Il thread main si fermerà in attesa sulla join del direttore, riprendendo la sua esecuzione al momento della chiusura.

Ripresa la sua esecuzione, il thread main procederà a joinare la casse, terminare il logging e chiudere i file utilizzati, per poi terminare.

DIRETTORE
Il thread direttore utilizza in parallelo un thread ausiliario (‘cashiers_handler’) avente il compito di gestire l’apartura e la chiusura delle casse, e la comunicazione con esse.
Contemporaneamente, il direttore aspetta passivamente richieste di uscita provenienti da clienti che hanno 0 prodotti. Una volta ricevuto uno dei due segnali, il direttore attende che tutti i clienti siano usciti prima di procedere. Nel caso di SIGHUP, il direttore continuerà a generare permessi di uscita per clienti che ne fanno richiesta. Nel momento in cui tutti i clienti sono usciti, il direttore joina il thread ‘cashiers_handler’, per poi terminare.

GESTORE DELLE CASSE
Il thread cashiers_handler fa uso di rand_r, per cui viene immediatamente creato un nuovo seed, diverso da quello usato dal thread supermercato. Per facilitare la gestione delle casse, viene creata una mappa binaria indicante quali sono le casse aperte e quali quelle chiuse.
Successivamente, per ogni cassiere viene letto il numero di clienti in coda dalla memoria condivisa con le casse. Il valore è sempre aggiornato perché i cassieri sovrascrivono il valore precedente. Se il numero di clienti in una cassa è al di sotto di un certo valore soglia (‘director_too_few_customers’), la cassa viene contaggiata come ‘below_min’. Analogamente, se il numero di clienti in una cassa è al di sopra di un altro valore soglia (‘director_too_many_customers’), la cassa viene contaggiata come ‘above_max’.
Per rendere il supermercato più dinamico, in ogni turno il cashiers_handler può solamente o chiudere o aprire una cassa. Non sarebbe razionale chiudere ed aprire una cassa nello stesso momento.

È data priorità all’apertura di una nuova cassa, perciò se il numero di casse sopra il valore soglia è maggiore del numero di casse sotto l’altro relativo valore soglia, viene valutatata l’apertura di una cassa. La cassa è aperta se il numero di casse oltre il limite è maggiore di director_above_max_limit, e se non sono già tutte aperte.
Una nuova cassa è aperta anche se ci sono meno casse del valore richiesto (‘director_above_max_limit’), ma una almeno una è above max, e il numero totale di casse aperte è minore di ‘director_above_max_limit’. La cassa è scelta in modo casuale, e nel caso in cui la cassa selezionata sia già aperta, si guarda la successiva fino a che non ne viene trovata una chiusa.

Nel caso in cui il numero di casse sotto il valore soglia è maggiore del numero di casse sopra l’altro relativo valore soglia, viene chiusa una cassa. La cassa viene scelta in modo randomico con la stessa politica descritta per l’apertura di una cassa. Dopo aver chiuso la cassa, la sua coda viene svuotata ed eventuali clienti in coda vengono avvertiti dello spostamento e segnalati.

Per vedere l’algoritmo “in azione” è sufficiente compilare definendo DEBUG con un valore >= 2. In questo modo viene stampato su stdout un elenco di tutte le casse affiancate del rispettivo numero di clienti in coda, o la parola CLOSE se la cassa è chiusa. Inoltre, sotto l’elenco viene stampato il valore di ‘above_max’ e ‘below_min’.

CASSIERI
Le casse utilizzano un thread ausiliario chiamato ‘report_to_director’ che ha il solo compito di scrivere su una variabile condivisa il numero di clienti attualmente presenti nella propria coda, e di segnalare il thread ‘cashiers_handler’ che sta potenzialmente aspettando la ricezione di questo valore. Il valore precedente viene sovrascritto in quanto non più necessario. La decisione di creare un thread distinto da ‘cashiers’ è dovuta al tentare di ottenere una migliore leggibilità del codice del cassiere, e al fatto che è stato stimato un numero di cassieri relativamente basso che permette di raddoppiare il numero di thread relativi alle casse senza avere un degrado delle prestazioni.
Le cassse possono essere inizializzate con stato OPEN o CLOSE a seconda del valore a cui è impostata la variabile O ('initial open cashiers') nel file di configurazione. Le casse ciclano in un while fintanto che non ci sono clienti rimanenti all interno del supermercato e si è ricevuto uno dei due segnali. In caso si riceva SIGHUP le casse continuano a servire i clienti rimanenti, altrimenti inviano semplicemente una risposta che indica la ricezione di un segnale di uscita. L'azione di servire clienti avviene in un ciclo interno la cui guardia è vera se lo stato della cassa è aperto. La cassa può uscire dal ciclo interno anche se si verifica la condizione del ciclo esterno.
Se si è usciti dal ciclo interno perché lo stato della cassa è CLOSE, significa che siamo stati chiusi dal direttore e dobbiamo aspettare che quest'ultimo ci segnali la riapertura.

Il cassiere ha anche il compito di loggare il tempo dei suoi turni di lavoro (da quando apre a quando chiude) e per ogni cliente servito il tempo impiegato a servirlo. Al momento della chiusura del supermercato, viene inoltre effettuato un log di informazioni generali riguardo al cassiere, come il numero di clienti serviti, di prodotti elaborati e il numero di chiusure relative ad una.

CLIENTI
Il thread customer si detacha appena creato, in quanto la sua chiusura è asincrona rispetto all' esecuzione del supermercato. Porzioni differenti del codice del cliente vengono eseguite a seconda del numero di prodotti che il cliente intende acquistare. Nel caso in cui non effetti alcun acquisto (numero di prodotti uguale a 0), il cliente invia al direttore una richiesta di uscita, e resta in attesa fintanto che non viene segnalato da esso.
Nel caso in cui il numero di prodotti sia maggiore di 0, il cliente sceglie in modo randomico una cassa e ne attende una risposta. Se la risposta è -2, significa che è stato ricevuto un segnale di sigquit e che il supermercato deve essere chiuso immediatamente, pertanto il cliente esce immediatamente. Se la risposta è -1, significa che non è stata ricevuta ancora nessuna risposta, e il cliente resta in attesa. Se la risposta è 0, significa che la cassa è stata chiusa dal direttore e che il cliente deve cambiare cassa, ripetendo l'algoritmo di scelta cassa. Se la risposta è 1, significa che il cliente è stato servito correttamente e può uscire dal supermercato.
Come richiesto dalla specifica, il cliente logga il tempo speso nel supermercato, il tempo speso in coda (che sarà 0 nel caso in cui sia stato interrotto da un SIGQUIT), il numero di volte che ha cambiato coda (che sarà 0 nel caso in cui abbia acquistato 0 prodotti) e il numero di prodotti acquistati (anch'esso 0 nel caso in cui abbia acquistato 0 prodotti).

FIFO_UNBOUNDED
La libreria fifo_unbounded, il cui codice è stato scritto da me, permette l'utilizzo del tipo fifo_unbounded_t, ossia una coda illimitata nella quale un elemento viene pushato nella coda e poppato dalla testa. L'utilizzo di un puntatore alla testa ed un puntatore alla coda rendono queste due operazioni asintoticamente costanti. Le operazioni di push e di pop possono far uso di mutex e variabile di condizionamento se si desidera utilizzare la coda in mutua esclusione ed usare attesa passiva nel caso in cui non possa essere poppato alcun elemento. Nel caso in cui non si voglia far uso di questo meccanismo, questi due campi possono essere lasciati NULL. La coda mantiene aggiornato anche un contatore che tiene traccia del numero di elementi presenti all'interno di essa, ed una funzione che permette di ottenere questo valore (anche questa funzione può far uso di mutua esclusione). La coda mantiene un puntatore agli elementi, e non una copia di essi, al fine di essere generale ed utilizzabile da componenti diverse. Per questo motivo gli elementi di ogni nodo sono dichiarati void*. Un' ultima funzione è presente per svuotare e deallocare gli elementi all'interno della coda.

UTILITÀ
L'header 'utils.h' contiene numerose macro di utilità che permettono di fare error checking in modo built-in, garantendo un migliore pulizia del codice. È inoltre presente la macro 'ERROR_AT' che permette di stampare il nome del file e il numero della riga in cui è avvenuto l'errore. Alcune di queste macro permettono di inserire una post-condizione personalizzata in caso non si desideri uscire all'occorrenza di un errore "non fatale".
È anche definito DEBUG, che se >=2 permette di vedere il funzionamento dell'algoritmo di apertura e chiusura delle casse, stampato su stdout. In caso di DEBUG=1, sarebbe opportuno stampare su stdout alcuni commenti verbosi riguardo l'esecuzione del programma, ma questo non è stato implementato.
Sono, infine, defininite quattro funzioni di utilità:

- **xmalloc**: una malloc con check dell'errore built-in, che effettua un "lazy exit" in caso di errore.
- **my_strtoi**: converte una stringa in un integer, ritornandolo, ed ha il controllo dell'errore built-in.
- **nanotimer**: attende un certo numero di microsecondi specificato come parametro utilizzando la funzione nanosleep.
- **timespec_diff**: effettua la differenza tra due struct timespec, restituendola in un'altra struct timespec passata come parametro.

SCRIPTS
Nella cartella 'script' è presente lo script richiesto dalla specifica 'analisi.sh' e altri sei scripts utilizzati per testare il programma. Ne segue la descrizione per ognuno di essi.

- **analisi.sh**: lo script prende in input il file './log/supermarket.log' prodotto durante l'esecuzione del supermercato. Il file deve essere formattato come descritto in precedenza. Al fine di garantire la compatibilità tra sistemi usanti geolocalizzazioni diverse, LC_NUMERIC è settato localmente a 'en_US.UTF-8'.
  In questo modo viene utilizzato il punto come separatore decimale all'interno delle printf, e questo garantisce, inoltre, piena compatibilità con l'input di 'bc -l'. Le righe vengono lette dal file usando read con IFS settato su '\n', tramite redirezione dell'input. Gli id di ogni riga vengono discriminati attraverso l'uso di un 'case', e la tokenizzazione della riga (separata da tab) viene effettuata dalla read con IFS settato su '\t', che prende tramite 'here-string' la linea letta in precedenza. In seguito vengono effettuati gli opportuni calcoli, memorizzazioni e, per quanto riguarda i clienti, le stampe (in quanto non richiedo elaborazione aggiuntiva).
  Infine, vengono effettuate ulteriori elaborazioni per il cassiere, per poi stamparne i risultati. Nel caso in cui il cassiere non abbia servito alcun cliente, non viene calcolato il tempo medio di servizio per evitare di dividere per 0 (ed ottenere, ovviamente, un errore da bc). Sfruttando la monospaziatura del font del terminale, i risultati dovrebbero essere visualizzati in forma tabulare con titolo di colonna.
- **hupmultinovg.sh**: Per utilizzare questo script è necessario eseguire "./script/hupmultinovg.sh #numeroditests". Lo scopo di questo script è l'individuazione di deadlocks nel caso di ricezione del segnale SIGHUP. Il programma viene eseguito senza Valgrind per cui permette di effettuare un elevato numero di test contemporaneamente senza appesantire eccessivamente le risorse del sistema.
- **quitmultinovg.sh**: Per utilizzare questo script è necessario eseguire "./script/quitmultinovg.sh #numeroditests". Lo scopo e il funzionamento sono analoghi a hupmultinovg.sh, ad eccezione che il programma riceve il segnale SIGQUIT.
- **hupmulti.sh**: Per utilizzare questo script è necessario eseguire "./script/hupmulti.sh #numeroditests". Lo scopo di questo script è l'individuazione di memory leaks o puntatori a memoria ancora reachable non liberati dal programma. Il programma viene eseguito all'interno di Valgrind, per cui può risultare pesante sulle risorse del sistema se troppe istanze dello script vengono eseguite contemporaneamente. Il file descriptor come file di log per Valgrind è utilizzato per forzare Valgrind a scrivere sul file in mutua esclusione con altre eventuali istanze di testing. L'uso del fd è analogo per gli scripts a seguire.
- **quitmulti.sh**: Per utilizzare questo script è necessario eseguire "./script/quitmulti.sh #numeroditests". Lo scopo e il funzionamento sono analoghi a hupmulti.sh, ad eccezione che il programma riceve il segnale SIGQUIT.
- **hupsingle.sh**: Per utilizzare questo script è necessario eseguire "./script/hupmulti.sh #numeroditests coreid". Lo scopo di questo script è verificare il comportamento del programma su un singolo core. Il programma viene eseguito all'interno di Valgrind, e riceve il segnale di SIGQUIT.
- **quitsingle.sh**: Per utilizzare questo script è necessario eseguire "./script/quitsingle.sh #numeroditests coreid". Lo scopo e il funzionamento sono analoghi a hupsingle.sh, ad eccezione che il programma riceve il segnale SIGQUIT.

NOTA AGGIUNTIVA
Eseguendo il programma in Valgrind con parametri "--leak-check=full –show-leak-kinds=all" può accadere che risultino "possibly lost" 272 bytes. Questo comportamento è più comune se il test viene eseguito utilizzando un solo core. Questo è dovuto al fatto che glibc riusa lo stack dei threads, talvolta non liberandolo al momento della chiusura del thread al fine di migliorare le prestazioni. Non è un memory leak in quanto il kernel lo libera automaticamente alla chiusura del main.