Scompattando il tarball (creato col comando **tar -cvzf**) si ottengono i seguenti files: **bin** (directory dove verrà creato l'eseguibile supermercato), **include** (directory dove sono presenti i file di header, ognuno dei quali contenente le clausule #ifndef #define #endif per evitare inclusioni multiple), **lib** (directory dove verrà creata staticamente la libreria per la gestione delle Linked Lists), **log** (directory di default dove verrà creato il file di log e altri file temporanei di appoggio da analizzare), **src** (directory contenente i file sorgente), **analisi.sh** (script bash che analizza il file di log generato dall'esecuzione), **config.txt** (file testuale di configurazione dal quale il programma attinge per impostare i propri parametri d'esecuzione), **Makefile** (Makefile per generare l'eseguibile "supermercato" ed eseguire il test), **Relazione.pdf** (la relazione del progetto).

Il file **Makefile** contiene il codice che permette di ottenere il file eseguibile **supermercato** nella cartella **bin** dalle proprie dipendenze, pulire i file generati ed esegure i tests. Ho assegnato una label a ogni directory del progetto per rendere i comandi più leggibili. Ogni flag di complazione ha la propria label a seconda del comando che deve essere eseguito, questo facilita la lettura e l'eventuale aggiunta di flags. Ho deciso di inserire 5 **phony targets**: **all** (per la generazione di **supermercato**), **clean** (per rimuovere ogni file creato dal **make** e dalle esecuzioni dei tests), **run** (che esegue il target dopo aver pulito la cartella log), **test** (che segue la specifica del progetto), **test1** (sulla falsariga del test1 descritto nella specifica del progetto).
Nel **Makefile** ho fatto le seguenti scelte implementative:

- Mi limito a linkare i file oggetto **cassiere** e **cliente** a **supermercato** perché non ritengo che sia necessario renderli una libreria (data la scarsa applicabilità ad altri usi).
- Ho optato per la creazione della libreria **linkedlist** in modo statico, per il nostro sistema non ho visto vantaggi nella crazione di una libreria dinamica.
- Nel phony **clean** rimuovo il **target**, tutto il contenuto della cartella **log**, tutti i file con estensione **.a** e **.o** a partire dalla directory corrente come da specifica.
- L'esecuzione di uno dei "phony di testing" elimina tutto il contenuto della cartella **log** prima di eseguire qualsiasi altro comando.
- Ho deciso di inviare i segnali (**SIGHUP** e **SIGQUIT**) con il comando **killall** con prima opzione il segnale e con seconda opzione il nome del processo (**supermercato**).
- Prima del newline nei phony **test** e **test1** ho inserito **punto e virgola** e **backslash** così da dividere solo fisicamente (e non logicamente) i comandi. Questo mi permette di avere tutti i comandi su un'unica sub-shell nella quale posso fare una wait su **!$** (**PID** dell'ultimo processo eseguito in background, che sarà **supermercato**) e lanciare **analisi.sh** solo quando il processo **supermercato** sarà terminato (quindi "alla sua chiusura").
- **test1** non è richiesto però mostra il funzionamento del progetto in caso di **SIGQUIT** (a differenza di quanto richiesto dalla specifica del **test1** non eseguo con Valgrind e lancio **analisi.sh** alla terminazione di **supermercato**).

Nell'implementazione del file di configurazione ho fatto le seguenti scelte:

- Ho optato per un file **.txt** per un'elevata compatibilità ed editabilità.
- Le costanti sono disposte per riga e il loro nome (che può essere modificato con uno più significativo, il parsing lo ignora) viene separato dal valore da un uguale.
- Ho introdotto le seguenti costanti oltre quelle espressamente rischieste dalla specifica: **K_INI** (intero che indica quante casse aprire prima dell'ingresso dei clienti), **TP** (intero che stabilisce il tempo in millisecondi richiesto per elaborare un prodotto da parte di un qualsiasi cliente), **S1** (intero che stabilisce la soglia delle casse con meno di due clienti per la quale il direttore ordina la chiusura di una cassa), **S2** (intero che stabilisce la soglia dei clienti in coda in almeno una cassa per la quale il direttore ordina l'apertura di una cassa), **I** (intero che stabilisce l'intervallo in millisecondi di comunicazione fra casse e direttore), **LOG** (stringa che stabilisce quale sia il file di log, se non specificato con opzione **-l** prima dell'esecuzione).
- Ho omesso la costante **S** perché non utilizzata nella versione semplificata.
- Il parsing di **config.txt** avviene nelle prime istruzioni del **main** dove viene aperto il file con **fopen** e vengono lette le righe escludendo la parte prima dell'uguale (compreso). Il valore letto viene salvato nel campo corrispondente della **struct_costanti** (struttura che non si limita a contenere le costanti).
- Contestualmente al parsing vengono effettuati controlli sulla consistenza e coesione dei valori.

Il file linkedlist.c e il corrispondente header contengono funzioni per la manipolazione di linked lists con elementi void *, il tutto in mutua esclusione per la possibilità di accesso concorrente (è lasciata comunque la possibilità di non utilizzare questi meccanismi di mutua esclusione passando **NULL** dove richiesto). Il file di header **linkedlist.h** è stato incluso in ogni file che richiedesse l'uso di una sua funzione.
Per questioni di leggibilità e pertinenza ho creato i file **.c** (e i relativi **.o**) a seconda dell'attore che li utilizzerà durante l'esecuzione. Ho scelto di tenere in **supermercato.c** le funzioni che verranno invocate alla creazione dei thread "attori" (**direttore**, **cassiere**, **cliente** e **threads di supporto**) per rendere meno dispersiva la lettura procedurale di comporamenti e interazioni. Essendo la versione semplificata del progetto il **direttore** e il **supermercato** saranno lo stesso processo.
In **supermercato.c** troviamo anche la dichiarazione e inizializzazione globale tramite macro delle **mutex** che saranno utilizzate per la manipolazione di risorse condivise (variabili e file di log). Dopo di che ho deciso di sfruttare la centralità di **supermercato.c** per dichiarare e inizializzare a 0 due variabili **volatile sig_atomic_t** che fungeranno da flag a toggle atomico in caso di ricezione di segnali (questo per mantenere breve il codice di handling dichiarato subito dopo).
Riutilizzo i comportamenti in caso della ricezione del **SIGHUP** anche in caso di **SIGQUIT** (in modo leggermente differente) quindi nell'handler di quest'ultimo setto entrambi i flags a 1. Prima delle funzioni dei threads possiamo trovare una funzione generica **attesa** la quale sfrutta la struttura **timespec** e la **nanosleep** per effettuare attese parametriche sull'argoento **amount**. Questa funzione è utilizzata da qualsiasi thread che abbia la necessita di attendere un determinato numero di millisendi, anche se per motivi differenti come può essere il servizio di un cliente o l'intevallo di comunicazione col direttore.
Ho sfruttato **supermercato.h** come header di utility inserendovi

- 2 **error handler** (**CHECK_PTR** e **CHECK_ERR**) per chiamate a funzioni esterne (ogni funzione che può dare origine a errore è opportunamente protetta da una delle due).
- strutture per threads ausiliari e per costanti (comprensive anche di variabili condivise).

Nelle prime righe del main avviene il controllo dell'opzione di esecuzione **-l** per specificare il file di **log**. Dopo di che apro il file di configurazione e ne effettuo il parsing (assumendo che segua lo standard da me definito) nel seguente modo: per ogni riga effettuo una **fscanf** con stream **config** considerando solo i dati che matchano le condizioni (cioè solo il valore che sta dopo l'uguale) e immagazzino il valore nel campo della **struct_costanti** corrispondente. Dopo ogni lettura segue un controllo di consistenza e coesione su ciò che è appena stato letto. Se è stato specificato un file di log col flag **-l** non viene effettuato il parsing dal file di configurazione. Completata la lettura di **config** viene creato e lanciato il thread direttore. Viene fatta una join su di esso in modo che il supermercato attenda la terminazione del direttore prima di "chiudere".

Il thread **direttore** riceve come argomenti la struct contenente le costanti e svolge le seguenti azioni:

- Inizializza i valori non costanti della **struct_costanti** a 0 (inizialmente non ho né casse né clienti).
- Installa gli handler dei segnali.
- Crea le casse.
  - Ho deciso di rappresentare la cassa e il cassiere come un'unica entità che immagazzina i dati nella propria struct così che siano persistenti fra le chiusure.
  - Il direttore avrà a disposizione un'**array** di **struct_cassiere** in modo da avere il pieno controllo sui **threads cassiere**.
  - Ho deciso di fare comunicare i cassieri e il direttore con una **pipe** per ciascun cassiere. La **pipe** viene creata prima della creazione del thread cassiere (la quale è un elemento della **struct_cassiere** e, in quanto tale, è facilmente reperibile dal direttore).
  - Scrivo contestualmente un valore iniziale negativo nella **pipe** così che, al passo successivo, possa far entrare i clienti solo quando le casse sono aperte e comunicanti (valore letto >=0).
- Attende che ogni cassa sia comunicante (
- Apre le prime **K_init** casse.
  - Ogni cassa creata effettua una **push** del proprio **cleanup_handler**, effettua una **cassiere_initialize** (dove vengono inizializzati e settati gli elementi della **struct_cassiere** che non erano stati manipolati dal direttore alla creazione) e crea il thread **gestore comunicazione** che gestisce la comunicazione col **direttore** a intervalli regolari (relativamente alla cassa, non in assoluto) del numero di clienti in coda (scrive nella propria **pipe** l'elemento **count** della linked list **in_coda**).
  - Ho deciso di rappresentare l'apertura e la chiusura di una cassa con un intero che ne denoti il suo **status** (0 se la cassa è chiusa, 1 se la cassa è aperta). Questo facilita il compito del direttore che per

aprirla si limiterà a settare il suo **status** a 1 e a fare una signal sulla sua variabile di condizione **empty**. La cassa stessa provvederà ad aggiornare il numero di casse attualmente aperte (**K_open**) e a mettersi in attesa dei clienti.

- Crea una lista che conterrà i clienti che non hanno effettuato acquisti e intendono uscire con il permesso del direttore.
  - Questo compito è delegato al thread **gestore_noprod** che, con politica **FIFO**, darà il permesso ai clienti di uscire (i quali saranno in **wait** sulla variabile di condizione **permesso_uscita**, elemento della **struct_cliente**).
  - La coda **noprod** appartiene alla **struct_cliente** e sarà visibile a tutti i clienti (come anche la mutex statica **mtxnoprod** che ne garantisce la mutua esclusione).
- Vengono creati ("vengono fatti entrare") i primi **C** clienti.
  - I clienti hanno fra gli elementi di **struct_clienti** anche **l'array delle casse**. Questo da loro la possibilità di verificare lo **status** di qualsiasi cassa e avere informazioni temporali.
  - Ogni thread cliente creato effettua una **pthread_detach** di se stesso (così da rilasciare le risorse alla sua terminazione), una **push** del proprio **cleanup_handler** e apre il file di **log** in **append**.
- Viene creato il thread **gestore_ingressi** che avrà il compito di scaglionare gli ingressi successivi in gruppi di **E** clienti (controllando costantemente il numero di clienti all'interno del supermercato).
  - Questo thread ha come argomento una **struct_ingressi**, la quale contiene i dati necessari da per costruire le nuove **struct_clienti** (ovvero **costanti**, **casse** e **noprod**)**.**
- Inizia un ciclo while fino a che non viene settato il flag **sighup_flag,** dove per ogni cassa aperta (anche se quelle non aperte continuano a comunicare) raccoglie i dati dalle sulle code ottenuti dalla **pipe** e li analizza. Infine controlla se i valori raccolti sono sufficienti a mettere in atto le politiche per l'apertura e chiusura di una cassa. Al termine del ciclo attende tanti millisecondi quanti quelli richiesti al **gestore_comunicazione** (non avrò mai un'assoluta sincronia) per permettere alle casse di comunicare nuovi dati.
  - Ho ipotizzato che il direttore, qualora entrambe le soglie siano soddisfatte, decida di non fare niente per evitare inutili aperture/chiusure (ho messo le condizioni in **XOR**).
  - Verosimilmente il direttore chiuderà la cassa che nell'analisi dei dati comunicati sia risultata (**al tempo della misurazione**) quella col minore numero di clienti in coda (questo per far spostare il minor numero possibile di clienti). A parità di numero minore il direttore sceglierà quella con l'**id** più grande. Il numero delle casse aperte sarà sempre almeno 1. Il direttore si occupa inoltre di svuotare la coda della cassa designata con la funzione **sposta** di **cassiere.c** (che sveglierà i clienti in coda con una signal su **being_serv** dopo aver settato loro il flag **cambio_coda**).
  - Per semplicità (e per mancanza di ragioni contrarie) il direttore aprirà la cassa chiusa con l'**id** più piccolo (sempe che **K_open** sia minore di **K**).

Un thread cliente che viene creato ("entra nel supermercato") effettua le azioni sopracitate, dopo di che attende attende un numero di millisecondi variabile con **scelta_prodotti** (quanto attendere viene stabilito con la funzione **get_delay_cliente** chiamata nella funzione **cassiere_initialize**, il quale valore viene salvato nel campo **delay** della **struct_cliente**).

- La scelta pseudo-randomica viene fatta utilizzando **rand_r** con seed **time(NULL)*id** del cliente (questo dovrebbe garantire una corretta pseudo-casualità) e modulando il risultato in modo che sia compreso fra 10 e il valore **T** preso dal file di configurazione.
- Questo metodo viene utilizzato anche per il numero dei prodotti acquistati, della cassa in cui inserirsi e dal cassiere per ottenere il tempo di servizio di un singolo cliente (diverso per ogni cassiere in un range tra 20 e 80 millisecondi).

Viene valutato il numero di prodotti acquistati (stabilito in modo pseudo-random nella **cliente_initialize** con la funzione **get_tobuy**): se è diverso da 0 (non è possibile ottenere quantità negative quindi sarà necessariamente >0) il cliente effettua **scelta_coda**, altrimenti si inserisce nella coda **noprod** in attesa del permesso di uscire.

- Una volta ottenuto viene ridotto il numero di clienti all'interno del supermercato. Se questo valore è 0 viene segnalato a chiunque stia aspettando che non ci siano clienti all'interno (il direttore che sta aspettando di chiudere il supermercato). Dopo di che viene scritto sul file di **log** (successivamente chiuso), eseguita una **pop** del **cleanup_handler** e terminato.

Un cliente che sceglie una coda ottiene un seed come detto precedentemente e lo utilizza per generare un numero compreso fra 0 e K, dopo di che controlla lo status di apertura di ogni cassa dell'array casse fino a

trovarne una aperta (a partire dalla cassa scelta pseudo-randomicamente) incrementando di 1 **chosen** a ogni verifica di apertura senza successo (modulando a ogni tentativo per **K**).
Se inserito correttamente nella coda di una cassa (svegliandola eventualmente se fosse in wait con una signal built-in all'**insert_tail** sulla variabile di condizione **empty** del cassiere) il cliente si salva **tempo_apertura** della cassa scelta e entra in un ciclo dove esegue una wait sulla variabile di condizione **being_serv**. Ogni variabile negata e messa in and nella guardia è settata a 0 inizialmente, questo protegge da wake up spurie.
Una volta svegliato da una signal su **being_serv** (non necessariamente perché è stato servito) esce sicuramente dal while (per almeno una delle 2 variabili non è più vera la guardia) e controlla se: deve cambiare coda (in questo caso aggiunge al proprio tempo all'interno del supermercato la differenza fra il precedente **tempo_apertura** e quello attuale e ricomincia il ciclo aggiungendo 1 alle casse visitate e genera un nuovo seed) o se è stato servito (in questo caso torna alla funione **fun_cliente**, controlla se è stato fatto uscire a seguito di una SIGQUIT, scrive sul file di log e esce).

- Ho ritenuto corretto distinguere i clienti fatti uscire forzatamente da quelli usciti normalmente facendo scrivere **CQ** invece di **C** nel file di log, questo renderà più facile il parsing con **analisi.sh**

Una cassa aperta, finché non le viene ordinato di chiudere, serve i clienti della sua coda in ordine FIFO (sfrutta la **wait** built-in nella **pop_head** per attendere se la coda è vuota).
Con la funzione **serve** il cassiere ottiene il primo elemento della sua lista **in_coda** (il primo cliente) e controlla se è un membro dello **staff**.

- Ho deciso di utilizzare dei **clienti fittizi** con flag **staff** attivo per chiudere le casse in caso di segnali. Nel caso in cui il cassiere ne incontri uno (sia in **serve** che in **fa_uscire)** setta il proprio flag **ordine_chiusura** a 1 e non effettua la procedura di servizio di un cliente standard (attesa del tempo di servizio e aggiornamento delle informazioni elaborati/serviti).

Se il cliente non è dello **staff** la cassa aggiunge il proprio tempo di servizio al numero di prodotti per il tempo di gestione di ogni prodotto (questa costante **TP** è presa dal file di configurazione) e attende per il totale (aggiornando il tempo speso dal cliente servito basandosi sull'attuale **tempo_apertura,** al quale sommerà poi la nuova attesa). Vengono anche aggiornati il totale dei clienti serviti e i prodotti elaborati.
Al cliente viene settato il flag **servito** (sia che sia fittizio o standard) e viene effettuata una **signal** su **being_serv** per farlo uscire dalla **wait**. Proseguendo sulla funzione **fun_cassiere** se non è stato servito un cliente fittizio (flag **ordine_chiusura**=0) viene decrementato il numero di **C_in**.
Se la cassa riceve l'ordine di chiusura alla prossima iterazione non servirà i clienti ma si preparerà per chiudere settando il proprio status a 0, scrivendo sul file di **log**, chiudendo i files apeti (**log** e **pipe**) e facendo la **pop** del **cleanup_handler**.

- I tempi (sia delle casse che dei clienti), essendo rappresentati in millisecondi, ho deciso di immagazzinarli in variabili intere e divideri per 1000 per ottenere i secondi (che verranno scritti sul file di **log** come floats con 3 cifre decimali).

In caso di ricezione di segnali gli **handlers** settano i **flags** come decritto sopra. Le reazioni variano a seconda del thread:

- Il **gestore_ingressi** in ogni caso cessa il suo funzionamento ed termina.
- Le casse reagiscono solo al **SIGQUIT** grazie al quale, se aperte, eseguono **fa_uscire** che si comporta in modo simile a **serve** ma non effettua attese, non elabora prodotti e setta il flag **quit** del cliente.
  - L'ultimo cliente che "farà uscire" la cassa sarà quello **fittizio**, il quale le ordinerà di chiudere.
- Il direttore uscirà dal ciclo di valutazione dei dati comunicatigli dalle casse e attenderà che tutti i clienti siano usciti (sia serviti che forzatamente) dal supermercato effettuando una wait su **noclienti** (variabile di condizione segnalata ogni volta che, facendo uscire un cliente, il valore di **C_in** è 0).
  - Successivamente, per ogni cassa, il direttore verifica se la cassa è aperta (se non lo fosse la apre) e crea un thread **cliente fittizio** con lo scopo di andare a chiudere la cassa (questo permette di "avvertire" anche casse già aperte che sono in attesa dell'arrivo di un cliente). Le casse che sono state aperte in questo modo verranno svegliate dall'ingresso in coda del **cliente fittizio** tramite la funzione **avvisa_cassiere** (dal funzionamento analogo a **scelta_coda** di un cliente standard ma con la differenza che la scelta della coda non è casuale ma dettata dalla variabile **tokill**, non viene considerata nessuna componente temporale e non è possibile cambiare coda).

Il direttore attenderà con una join la terminazione di tutte le casse prima di passare alla cancellazione del **gestore_noprod** (il quale si trova in wait di clienti con 0 prodotti che, per costruzione, non arriveranno mai). Dopo di che uscirà e sarà joinato nel main, facendo così terminare il processo.

Ho predisposto fra i define un flag **DEBUG** che, se = 1, accompagna l'esecuzione con log sintetici sulla situazione in tempo reale del supemercato.
Ho cercato di liberare tutto lo spazio allocato dinamicamente con **free** opportune ma continuano ad essere presenti memory leaks.

Per il file di log mi sono attenuto al formato richiesto dalla specifica del **test2**. Questo mi ha facilitato nel parsing grazie ai separatori '|'. Ho ritenuto corretto aggiungere **K**, **C** e **CQ** prima del campo **id** per identificare l'enità alla quale appartengonon i dati.
Di default il file di log è "**./log/log.txt**", se modificato tramite **config.txt** o tramite opzione **-l** è importante tenere presente che lo script **analisi.sh** è programmato per fare il parsing di **./log/*** quindi darà errori di divisione per 0 se la cartella log è vuota o errori impredicibili se i file al suo interno non rispettano lo standard da me definito.

- I comandi **make**, **run**, **test** e **test1** puliscono completamente la cartella **log** per non sovrapporre risultati di diverse esecuzioni (sia i cassieri che i clienti aprono il file di **log** in **append**).

Lo script **analisi.sh** si apre col fissare i valori iniziali delle variabili a 0.
Successivamente metto in pipe i seguenti comandi (questo procedimento verrà usato per i clienti, i clienti usciti forzatamente e i cassieri):

- Eseguo il comando **cat** su tutto il contenuto della cartella log, metto in pipe il risultato col **pattern matching** desiderato (**C|**, **CQ|** o **K|** a seconda dell'entità interessata), effettuo un **sort** e salvo il contenuto in un file temporaneo (che verrà eliminato con l'ultima riga dello script).
- Eseguo una **exec** su un file descriptor libero del file temporaneo appena creato (sarebbe possibile usare sempre 3 ma, in caso di modifiche del sistema, preferisco mantenere **fd** separati).
- In un ciclo while leggo una riga alla volta dal file descriptor (opzione **-u** di **read**) preso in considerazione e, a seconda dell'entità considerata, effettuo operazioni differenti.

Durate il parsing delle righe effettuo **echo ${riga}** per stampare a schermo come richiesto nella specifica.
Utilizzo una combinazione di pattern matching per isolare un campo alla volta di ogni riga letta fino all'esaurimento di quest'ultima. Avanzo con **riga=${riga#*|}** (elimino dalla riga la più corta occorrenza iniziale che termina con '|') e isolo un campo con **=${riga%%|*}** (elimino dalla riga la più lunga occorrenza finale che inizia con '|').
Per eseguire operazioni aritmetiche metto in pipe la stampa con **echo** e **bc** (con **scale=3** per far fede alla specifica che richiede 3 cifre decimali). Questo appesantisce il parsing di **log** ma mi permette di fare operazioni con i float e aggregare i dati in modo da ottenere informazioni user-friendly che stampo alla fine dello script (avendo cura di considerare tutti i possibili casi).
Analisi.sh pulisce i file temporanei con l'ultima istruzione.

Duarante lo sviluppo del progetto ho utilizzato codice conforme allo standard **C99** e **POSIX 199309L** (ho notato che questa combinazione che mi permetteva di riconoscere tutte le funzioni esterne utilizzate).