# Progetto Sistemi Operativi A.A. 2019/2020

Roberto Lorenzetti

## Indice



## 1 Introduzione

La cartella del progetto contiene in totale dodici file, in questa introduzione verrano rapidamente elencati per poi andarne a spiegare le funzionalità in dettaglio nei capitoli successivi.

Vi sono:

- Sette file sorgente .c: cassa.c, cliente.c, direttore.c, gestore.c, parsing.c, queue.c, supermercato.c.
- Tre script .sh: analisi.sh, test1.sh, test2.sh.
- Un Makefile e un util.h.

## 2 Supermercato

Il supermercato è il file .c contenente il main. Appena la funzione main viene invocata effettua immediatamente il parsing del file config.txt, tramite il quale assegna i valori alle variabili globali : k, c, e, t, p, s, x, y, m, l, z.
Le prime sei rispecchiano i ruoli descritti nella specifica, le restanti cinque, chiamate con le lettere dell'alfabeto in coerenza con la notazione della consegna, hanno il seguente significato:

- x: il valore s1 ovvero "definisce il numero di clienti in coda in almeno una cassa".
- y: il valore s2 dunque "il numero di clienti in coda in almeno una cassa".
- m: il numero di casse inizialmente aperte, prima che il numero venga variato dalle scelte del direttore.

- l: “il tempo di gestione di un singolo prodotto”.
- z: l’intervallo di tempo tramite il quale il cassiere aggiorna regolarmente il direttore sul numero di clienti in coda nella propria cassa.

Successivamente, il main crea il thread di gestione dei segnali la cui funzione è presente in supermercato.c . Il compito di quest’ultimo é quello standard, ovvero aspetta tramite una sigwait uno tra i segnali SIGQUIT e SIGHUP.
Dopodiché c’è la fase di inizializzazione che posso schematizzare in 3 parti ben distinte.
La prima riguarda le mutex. Vengono inizializzate cinque mutex singole rispettivamente per vedere il numero di casse aperte, per aggiornare il numero di clienti in coda, per il numero totale di clienti all’interno del supermercato, per chiedere il permesso al direttore di uscire nel caso in cui non si acquistano prodotti e una mutex per scrivere sul file log.txt. Infine ci sono le mutex “concorrenti” relative alle code delle casse implementate tramite un vettore lungo k. Ogni cassa possiede la propria variabile mutex tramite la quale un cliente può inserirsi in coda.
La seconda implementa ed inizializza tutti gli array necessari per memorizzare i dati delle casse (aperte/chiuse, numero di clienti in coda, clienti serviti, prodotti, chiusure e quant’altro) e gli argomenti dei thread (direttore, cassieri iniziali, gestore) La terza e ultima parte riguarda la struct core. La struct core contiene tutti i dati necessari ai vari thread per lavorare correttamente, come ad esempio il vettore delle mutex per le casse, il fp per scrivere sul file di log, la mutex del direttore ecc.
Finalmente avvia i thread gestore dei clienti, il thread direttore e si mette in attesa che quest’ultimo termini la sua esecuzione. Una volta terminato, l’esecuzione del main riparte facendo un sunto delle statistiche riportate negli array appositamente creati. La sua terminazione avviene dopo aver liberato la memoria. La terminazione di tutto il processo la spiegherò alla fine con l’intento di essere il più chiaro possibile.

## 3 Cliente

cliente.c contiene la funzione cliente che rappresenta il thread. Subito si mette in modalità detach.
Subito dopo effettua il parsing della struct passatagli per argomento e crea immediatamente una seconda struct per salvare tutti i dati relativi alla sua visita nel supermercato. Ulteriori struct di tipo “timeb” vengono utilizzate per misurare il tempo in coda e il tempo totale nel supermercato.
Tra i vari dati vi è il numero di prodotti acquistati e il tempo degli acquisti. Come richiesto tali valori vengono calcolati casualmente tramite la funzione rand_r, il cui seme viene inizializzato come risultato della somma del tempo di sistema (time(NULL)) e del proprio id cliente. In questo modo mi assicuro che ogni thread abbia un seme diverso grazie al quale posso garantire l’effettiva pseudo casualità dei risultati calcolati dalla rand_r.
Il numero di prodotti acquistati determina una ramificazione significativa. Infatti se il numero di prodotti é uguale a zero il cliente si inserisce in coda dal direttore che provvederà successivamente a fornirgli l’autorizzazione per uscire dal supermercato. Viceversa, se il numero di prodotti è diverso da zero, il cliente entra in un while(1) che gli consente di inserirsi in coda ad una cassa (ciclicamente nel caso in cui venisse chiusa), scelta, analogamente a quanto detto sopra, in maniera casuale. Possiede solo due modalità per uscire dal ciclo: o viene servito da un cassiere, oppure il supermercato viene chiuso per emergenza (SIGQUIT). Nel primo caso, prima di terminare stampa nel file di log tutti i dati significativi e in aggiunta aggiorna il numero di clienti totali all’interno del supermercato. Ancora, nel caso in cui il numero di clienti nel supermercato sia uguale a C - E, invia un segnale al gestore dei clienti che provvederà a fare entrare altri E clienti nel supermercato. Nel caso in cui sia uguale a 0 avvisa analogamente il gestore per terminare l’esecuzione del processo correttamente.
Alternativamente, se non rientriamo in nessuna delle due situazioni precedenti, si limita a terminare la propria esecuzione.

## 4 Gestore

Anch'esso contiene una sola funzione principale che descrive il thread. Dopo il parsing della struct argomento, fa partire subito i primi C thread clienti.
Fatto ciò entra in un ciclo while(1) che gli consente di effettuare la gestione delle entrate dei clienti. All'inizio del ciclo controlla se sm_emerg é uguale a uno (che corrisponde al segnale SIGQUIT), in tal caso termina dopo aver mandato un cliente speciale di terminazione (rappresentato da dataclose) per avvisare il direttore che il supermercato deve chiudere immediatamente. Se invece non c'é nessuna emergenza, il gestore controlla se siamo nel secondo caso di terminazione, ovvero se sm_open é uguale a zero(SIGHUP). Se cosi fosse, attende che tutti i clienti siano usciti dal supermercato tramite una classica wait. Una volta arrivato il segnale sulla relativa variabile di condizione, avvisa il direttore del fatto che bisogna chiudere immediatamente come nel caso precedente.
Superati questi due controlli, quindi se siamo in una situazione normale (non é arrivato nessun segnale) il gestore può occuparsi dei clienti, in particolare si mette in attesa su una variabile di condizione, tramite la quale scopre quando il numero di clienti è esattamente C - E ed agisce di conseguenza facendo entrare altri E clienti.

## 5 Direttore

Il direttore é composto da due thread distinti. Questa scelta è stata forzata dalla necessità di dover, sia gestire i clienti con zero prodotti sia applicare l'algoritmo di decisione per aprire/chiudere le casse. Il primo thread effettua come di consueto il parsing della struct argomento, dopodiché fa partire subito il secondo thread. Il primo continua la sua esecuzione entrando in un ciclo while(1) grazie al quale si mette in attesa di clienti con zero prodotti e, di volta in volta quando viene risvegliato dalla presenza di un cliente lo lascia uscire dal supermercato inviandogli un segnale. Eccezion fatta per il cliente di terminazione, mandatogli nell'eventualità dal gestore, che lo notifica del fatto che il supermercato deve essere chiuso il prima possibile. Per fare ciò aspetta, prima il thread gestore, successivamente imposta un flag di terminazione per il thread che applica l'algoritmo e lo attende prima di terminare. Il secondo thread effettua periodicamente l'algoritmo di decisione tramite i valori S1 e S2 (nel nostro caso x e y). La lunghezza dell'intervallo che descrive il periodo di azione dell'algoritmo è uguale a quella che utilizzano le casse per aggiornare il numero di clienti in coda. Sono consapevole del fatto che la sincronizzazione in quest'ottica non può essere strutturalmente garantita, tuttavia mi è sembrata la scelta più sensata. Dunque l'intervallo è uguale a z. Quando il flag di terminazione cambia valore si occupa di terminare tutte le casse aperte garantendo la corretta chiusura.

## 6 Cassa

Anche la cassa é composta da due thread entrambi in cassa.c, entrambi ricreati ogni volta che viene chiusa e poi riaperta. Questa scelta é stata una delle più difficili. Inizialmente ero orientato su un thread singolo, il cui funzionamento era gestito tramite operazioni aritmetiche sul tempo di servizio del cliente. In particolare avevo intenzione di controllare, una volta calcolato il tempo di servizio di ogni cliente, se avessi fatto in tempo a servirlo prima dello scadere dell'intervallo oppure no. In caso affermativo lo avrei servito tranquillamente, in caso negativo avrei sottratto al tempo di servizio quello che mancava per raggiungere l'intervallo, quindi avrei aggiornato, in ordine, il numero di clienti in coda e poi servito il cliente per il tempo restante. Non ho optato per questa scelta perché sinceramente non mi sembrava realistica nell'ottica di una realizzazione di un supermercato su larga scala e inoltre volevo garantire più precisione e pulizia possibile.
Chiusa questa parentesi, il thread principale, dopo aver avviato il secondo thread(timer) entra in un while(1) in cui attende l'arrivo dei clienti da servire. Il cliente viene servito aspettando il periodo necessario determinato dal tempo di servizio costante (stavolta il seme é determinato da time(NULL) + id cassa + id thread in modo tale da garantire la casualità anche in caso di riaperture ravvicinate. La terminazione é gestita similmente a quanto descritto sopra, c'è un cliente speciale che fa capire

alla cassa che deve chiudere. Prima di terminare, la cassa scrive sul file di log tutti i vari dati relativi all'attuale finestra di apertura.

# 7 Parsing

Il parsing è una funzione invocata nel main, la sua descrizione è nel file parsing.c. Effettua il parsing del file config.txt con cui riesce ad assegnare i valori alle variabili globali elencate sopra. Ho utilizzato la funzione getopt. Nel caso in cui alcuni valori siano poco significativi o addirittura nulli viene chiuso il processo e viene richiesta una nuova esecuzione del programma.

# 8 queue.c

Il file queue.c contiene tutte le funzioni relative alla coda, quindi le classiche push back, push front, del, ecc. Inizialmente volevo creare una struttura più generale possibile, in modo tale da rendere flessibili le funzioni. Mi riferisco in particolare alla possibilità di lavorare con generici nodi (puntatori a void) . Ho abbandonato questa idea quando mi sono reso conto che nella mia architettura del progetto, tutte le code erano formate da struct di "tipo" cliente, quindi aveva poco senso complicare ulteriormente il codice al solo scopo di guadagnare eleganza.

# 9 Terminazione

Ci sono ovviamente due casi: SIGHUP: in caso di SIGHUP il thread gestore dei segnali setta la variabile sm_open = 0. Grazie a ciò il gestore capisce che non deve far entrare più clienti e aspetta che l'ultimo cliente esca dal supermercato. Quando il numero dei clienti nel supermercato è uguale e zero avvisa il direttore mandandogli un cliente di terminazione in coda, il direttore di conseguenza capisce che deve chiudere il supermercato. Dunque cambia valore ad un flag di terminazione e aspetta che il suo secondo thread termini. Il secondo thread del direttore, a sua volta, si occupa della chiusura di tutte le casse mandando dei clienti speciali. Le casse a loro volta attendo la terminazione del proprio thread timer. Ora il thread direttore può terminare e, il thread main, che era in attesa sul direttore, può terminare tranquillamente l'esecuzione del processo.
SIGQUIT: Il gestore vede che la variabile sm_emerg é impostata ad 1 e avvisa immediatamente il direttore. Lui agisce ugualmente a quanto scritto sopra. I clienti stavolta si rendono conto che il supermercato è chiuso per emergenza quindi appena possono escono dal supermercato senza accodarsi alle casse perché sono state tutte chiuse dal direttore. C'é la possibilità che qualche thread cliente termini successivamente al thread main perché durante la terminazione era in fase di acquisti. Per evitare questo ho messo una sleep di pochi secondi alla fine del programma. Sono consapevole del fatto che non è una bella soluzione, l'alternativa era fare una coda con i thread id dei clienti e forzare la loro terminazione in caso di emergenza. Ovviamente tutti i clienti che terminano si rimuovono autonomamente in modo da velocizzare il processo di chiusura. Purtroppo non ho avuto il tempo materiale per implementarla.