PROGETTO SOL SUPERMARKET (a.a. 2019-2020)
ALESSIO LA GRECA 581578

Il progetto prevede di simulare, mediante la programmazione multi-threaded, un giornata di lavoro all'interno di un supermercato, dove dei clienti entrano per fare acquisti, vengono serviti dai cassieri e questi ultimi vengono coordinati dal direttore.
Per permettere ai vari thread di comunicare tra di loro vengono usate strutture dati condivise come variabili o array globali che possono essere acceduti in mutua esclusione mediante l'utilizzo delle mutex (o lock). Inoltre, per evitare attesa attiva, sono state usate anche alcune condition variables associate a variabili e lock. Di seguito saranno illustrate tali strutture dati con relative mutex e cv, così come il contesto in cui si inseriscono e che parte del progetto riguardano. Si noti che, per semplicità di programmazione (e si spera anche di lettura), ad ogni variabile condivisa "variabile_generica" è associata sempre una mutex chiamata "MUTEX_variabile_generica", e occasionalmente una cv chiamata "CV_variabile_generica". Benché non sempre questa nomenclatura renda immediata la relazione tra lock e strutture dati condivise (a volte un array condiviso ha ssociata una sola lock, mentre altre volte altri array condivisi hanno una lock per ogni loro posizione), ritengo che generalmente diminuisca il grado di confusione.
Inoltre, sempre per semplicità di lettura, grazie all'uso di due define è stato definito il tipo primitivo *bool* avente come possibili valori *true* e *false.*
Dopo questa panoramica sulle strutture dati condivise segue una descrizione del ciclo di vita di un thread cliente, cassiere e direttore.
Infine, definiamo brevemente i parametri a libera scelta:
V: tempo che ogni cassiere impiega per "scannerizzare" un prodotto (tempo di servizio lineare rispetto al numero di prodotti acquistati dal cliente attualmente servito).
D: ogni quanti millisecondi un cassiere deve informare il direttore del numero di clienti che ha in coda.
CasseAperteInit: quante casse vengono aperte all'apertura del supermercato
FileDiLog: nome del file di log della giornata di lavoro.

prodotti_acquistati_totale (MUTEX_prodotti_acquistati_totale)
clienti_serviti_totale (MUTEX_clienti_serviti_totale)
Variabili intere che permettono di contare il numero di clienti serviti in totale durante la giornata e il totale di prodotti acquistati. Vengono aggiornate da un cliente quando questo esce dal supermercato dopo essere stato servito da un cassiere.

id_cliente (MUTEX_id_cliente)
Variabile intera che determina l'id univoco del cliente (serve per distinguerlo dagli altri nel file di log). Ogni volta che un cliente esce, legge questa variabile per sapere quale è il suo id e poi la incrementa di uno, così da evitare ripetizioni.

clienti_con_zero_prodotti_in_coda (MUTEX_clienti_con_zero_prodotti_in_coda)
Variabile intera che permette di tenere conto di quanti clienti hanno fatto 0 acquisti e stanno aspettando il permesso del direttore per uscire. Quando un cliente con 0 acquisti vuole uscire dal supermercato, aumenta di uno il valore di questa variabile. Quando il direttore fa uscire questi clienti, li fa uscire tutti assieme, impostando questa variabile a 0.

array_cassieri_aperti (MUTEX_array_cassieri_aperti, una per ogni cassa*)*
Array di booleani lungo K dove ogni posizione corrisponde ad un cassiere. Se la cassa i è aperta, allora array_cassieri_aperti[i] = true, altrimenti array_cassieri_aperti[i] = false. Viene letta periodicamente dai cassieri così che capiscano se devono chiudere o meno, mentre viene scritta dal direttore quando decide di aprire o chiudere una cassa.

array_cassieri_numero_clienti_in_coda (MUTEX_array_cassieri_numero_clienti_in_coda, CV_array_cassieri_numero_clienti_in_coda, una mutex e una cv per ogni cassa*)*
Array di interi lungo K dove i cassieri tengono traccia del numero di clienti che hanno in coda. Quando un cliente si mette in coda alla cassa i, array_cassieri_numero_clienti_in_coda[i] aumenta di uno, mentre diminuisce di uno ogni volta che il cassiere i serve un cliente. La cv serve a evitare attesa attiva: se il cassiere i non ha clienti in coda ed è aperto, aspetta un segnale che può arrivare da un cliente o dal direttore. Se arriva

dal cliente, vuol dire che la cassa deve servire qualcuno. Se arriva dal direttore, vuol dire che la cassa deve chiudere.

array_id_cliente_presente_nel_supermercato (MUTEX_array_id_cliente_presente_nel_supermercato, una sola lock per tutto l'array).
Array di interi lungo C dove ogni posizione corrisponde allo "stato" di uno dei C clienti che può trovarsi all'interno del supermercato. Ad ogni cliente può corrispondere uno dei seguenti sei valori, ciascuno dei quali ha un proprio significato e rappresenta appunto lo stato di un particolare cliente:
-2) il cliente è uscito e la sua memoria è stata liberata (il direttore ha fatto la join su di lui)
-1) il cliente è uscito ma la sua memoria deve essere liberata (il direttore deve fare la join su di lui)
0) il cliente è entrato nel supermercato e sta girando per gli scaffali (non si è ancora mai messo in coda)
1) il cliente si è messo in coda ad una cassa, ma ha almeno una persona davanti a lui
2) il cliente si è messo in coda ad una cassa ed è il prossimo ad essere servito
3) il cliente sta cercando una nuova cassa dopo che quella in cui si trovava è stata chiusa.
Si noti che d'ora in poi useremo sempre la parola "stato", relativamente a un cliente, per indicare il numero ad esso corrispondente in questo array.

array_tails (MUTEX_array_tails, CV_array_tails, una mutex e una cv per ogni coda).
Array lungo K di puntatori a elementi di una lista di tipo coda_cassa. Serve per tenere traccia dei clienti che si sono messi in coda ad una cassa. Le cv servono a evitare attesa attiva: i clienti in coda si mettono in attesa del loro turno, attendendo un segnale dal cassiere.

array_cassieri_numero_clienti_in_coda_direttore
(MUTEX_array_cassieri_numero_clienti_in_coda_direttore, una mutex per ogni cassiere)
Array di interi lungo K che viene aggiornato ogni D millisecondi da un cassiere, e che viene letto dal direttore. E' sulla base dei valori letti in questo array che il direttore decide se aprire o chiudere una cassa.

array_tails_tda (MUTEX_array_tails_tda, una mutex per ogni cassa*)*
Array lungo K di puntatori a elementi di una lista di tipo cc_tempi_attesa. Viene usata dai clienti per conservare le informazioni relative al loro tempo di attesa in coda, così che possano essere successivamente reperite dai cassieri.

direttore_tail (MUTEX_direttore_tail)
Puntatore a elementi di una lista di tipo clienti_attesa_direttore. Quando un cliente non acquista prodotti, si aggiunge mediante una funzione ad una lista puntata da questo puntatore, così che il direttore possa a un certo punto accedervi per far uscire i clienti in attesa.

MUTEX_scrittura_clienti
Mutex a cui non è associata nessuna variabile. Serve ad assicurare che la scrittura di un cliente sul file di log dei clienti avvenga in mutua esclusione.

si_chiude_sigquit (MUTEX_si_chiude_sigquit)
Variabile booleana che permette ai clienti, in caso di segnale sigquit ricevuto, di non rimettersi in coda ma di uscire immediatamente dal supermercato.

notifica_direttore_cambiamento (MUTEX_notifica_direttore_cambiamento, CV_notifica_direttore_cambiamento)
Variabile booleana che permette di evitare attesa attiva da parte del direttore. Quando è false, il direttore, dopo aver controllato lo stato attuale del supermercato, si mette in attesa. Viene risvegliato da un cassiere qualunque che abbia passato l'intervallo dei D millisecondi, settando tale variabile a true. Quando questo avviene, il direttore fa un altro controllo della situazione del supermercato.

sighup_numero_casse_concluse (MUTEX_sighup_numero_casse_concluse, CV_sighup_numero_casse_concluse)
Variabile intera che serve ai cassieri, in caso di segnale sighup ricevuto, per comunicare al direttore quando hanno finito di servire tutti i clienti che avevano in coda, così che il direttore possa (eventualmente) chiuderli. Serve inoltre ad evitare attesa attiva da parte del direttore.

uscito_un_cliente_dal_supermercato (MUTEX_uscito_un_cliente_dal_supermercato)
Variabile booleana che serve ai clienti, in caso di segnale sighup ricevuto, per comunicare al direttore che uno dei clienti rimasti è uscito. Anche questa serve a evitare l'attesa attiva del direttore.

Seguono alcune strutture dati condivise che non posseggono una mutex (di seguito è motivato anche il perché):
array_TSC: array di interi lungo K che contiene il Tempo di Servizio Costante di ogni cassiere. I valori delle K posizioni vengono generati randomicamente dal main all'inizio del programma, prima che qualche altro thread venga generato. Successivamente viene solo letto dai cassieri, ciascuno dei quali legge solo la posizione ad egli relativa. Dato che viene scritto una volta sola all'inizio e successivamente solo letto, ritengo superflua una mutex per questo array.
array_cassieri_numero_prodotti_elaborati_totale: array di interi lungo K
array_cassieri_numero_clienti_serviti_totale: array di interi lungo K
array_cassieri_numero_di_chiusure_totale: array di interi lungo K
array_cassieri_tempo_di_apertura_totale: array di long lungo K
Sono tutti array lunghi K posizioni che vengono aggiornati da un cassiere al momento della sua chiusura. I nomi dovrebbero essere abbastanza esplicativi: quando un cassiere chiude accede alla sua posizione relativa di ogni array, aggiornando il numero di prodotti che ha elaborato in totale, i clienti serviti, il numero di chiusure e il tempo di apertura della cassa, tutti dati che verranno usati alla fine del programma per produrre un sunto delle statistiche di ogni cassiere. Dato che, alla posizione i-esima di ogni array accede solo il cassiere i-esimo quando sta per chiudere, e che non possono esistere contemporaneamente due cassieri i-esimi, non ritengo necessario l'uso di lock per l'accesso a queste strutture dati.

Per prevenire il fenomeno di deadlock si è cercato, oltre a ridurre il più possibile le sezioni critiche, di evitare che si verificasse la condizione necessaria *circular waiting,* attesa circolare. Ovvero: se ad un certo punto del codice un thread entra in possesso della mutex "i" e prima di rilasciarla entra in possesso anche della mutex "i+k" (con i e k valori generici positivi), allora si può stare tranquilli che in nessun altro pezzo di codice accade mai che qualcuno acquisisce prima la mutex "i+k" e poi la mutex "i". Le mutex sono insomma ordinate in modo crescente tale che possano essere acquisite solo se si rispetta questa cardinalità.

Per la gestione dei segnali sono stati installati due gestori che si limitano a settare a 1 una variabile globale, una per il sighup e una per il sigquit. Quando queste variabili sono a 1, il comportamento del direttore, dei cassieri e dei clienti può variare di poco o di molto.

Thread main: Quando il programma comincia, viene letto il file di configurazione passato come argomento, e i dati letti vengono inseriti in una struct che contiene tutte le variabile di esecuzione (C, K, S1, S2…). Aprendo il file config.txt è possibile modificare tali valori. Successivamente, il main alloca tutte le strutture dati di cui ha bisogno, di cui verrà successivamente fatta la free prima della terminazione del programma. Questo include anche eventuali array di lock e cv la cui quantità dipende dai parametri di configurazione. Fatto ciò viene lanciato il thread direttore, e il main si mette in attesa della sua terminazione. Il direttore, inizialmente, aprirà il numero iniziale di casse specificato e genererà C clienti.

Thread cliente: Quando un cliente viene creato (e quindi il suo stato è a 0) inizializza una nuova struct che conterrà i dati di suo interesse: T, P, V, K e C. Inoltre, il direttore gli passa come argomento un intero che va da 0 a K-1, e che serve a distinguerlo dagli altri clienti presenti nel supermercato. Il cliente dunque genera randomicamente un numero p (i prodotti che acquista) tra 0 e P, e un numero t (il tempo che trascorre nel supermercato facendo acquisti) tra 10 e T. Se p==0, si mette nella coda di attesa dei clienti che stanno aspettando il permesso del direttore per uscire. Altrimenti fa una nanosleep che dura t millisecondi, e poi con una clock_gettime si segna il momento iniziale in cui si è messo in coda. A questo punto il cliente sceglie randomicamente una cassa in cui mettersi in coda. Se, tuttavia, è arrivato il segnale di sighup, il cliente setta il suo stato a -1, e esce senza venire servito da alcun cassiere. Se invece la cassa scelta è chiusa, il cliente ne sceglie un'altra. Trovata una cassa in cui può mettersi in coda, si inserisce in essa mediante la funzione insert, che aggiorna il suo stato da 0 a 1 oppure 2. Se lo stato diventa 1, il cliente si mette in attesa su una cv aspettando di essere risvegliato dal cassiere, che eventualmente manderà una broadcast alla sua coda dopo aver servito un cliente. Se lo stato del cliente al momento del risveglio è ancora 1 torna in attesa. Se è 2, vuol dire che il cliente è stato servito. In tal caso, il cliente fa una seconda clock_gettime per poi fare la differenza

tra i due istanti di tempo (da quando si è messo in coda a quando è stato servito), e si mette a scrivere nel file di log dei clienti le informazioni ad esso relative. Inserisce inoltre in una lista il suo tempo di attesa in coda, che servirà poi al cassiere per fare un calcolo delle sue statistiche di servizio. Se invece lo stato al momento del risveglio è 3, il cliente cerca una nuova cassa in cui mettersi in coda per poi rimettersi in attesa fino a che lo stato non è aggiornato a 2. In caso di sighup, prima di terminare, il cliente informa il direttore che è uscito. In ogni caso, quando un cliente termina, setta il suo stato a -1, così che il direttore possa rendersi conto che quel cliente è uscito, e che quindi la sua memoria può essere liberata.

Thread cassiere: Quando un cassiere viene aperto, entra in un ciclo che non termina fino a quando la cassa non viene chiusa dal direttore. E' al suo interno che si svolge la vita di un cassiere. Inizialmente, il cassiere verifica se ha mandato, nell'ultima esecuzione del ciclo, un aggiornamento del numero di clienti in coda alla cassa al direttore. Se l'ha fatto, fa una clock_gettime. Poi controlla se è arrivato il segnale di sighup. Se è arrivato, serve tutti i clienti presenti in coda, e informa il direttore non appena li ha serviti tutti. Se invece tale segnale non è arrivato, controlla due cose: se ha clienti in coda e se il direttore gli ha detto di chiudere. Se il cassiere ha zero clienti in coda e deve ancora rimanere aperto, si mette in attesa di un segnale. Se questo segnale arriva dal direttore, allora il cassiere si segna che deve chiudere, e eviterà di servire i clienti. Se invece questo segnale arriva da un cliente è perché c'è qualcuno che si è messo in coda e sta aspettando di essere servito. In tal caso, il cassiere lo serve facendo una nanosleep un numero di secondi pari a: il suo tempo di servizio costante + il numero di prodotti acquistati dal cliente * V. Poi, dopo aver impostato a 2 lo stato del cliente subito successivo a quello appena servito, il cassiere risveglia tutti gli altri eventuali clienti in coda con una broadcast. Tra questi, però, solo quello il cui stato è cambiato da 1 a 2 proseguirà l'esecuzione. Gli altri torneranno in attesa. Fatto ciò, il cassiere controlla se nel frattempo il direttore gli ha richiesto di chiudere. Se, a questo punto, il cassiere deve chiudere, chiama una funzione (close_cashier) che notifica a tutti i clienti in attesa alla cassa di cambiare coda, impostando il loro stato a 3. Arrivati a questo punto, il cassiere fa una seconda clock_gettime e calcola il tempo trascorso tra le due. Se è maggiore o uguale ai D millisecondi specificati nel file di configurazione e deve ancora restare aperto (il direttore non gli ha detto di chiudere), informa il direttore del numero di clienti che ha attualmente in coda. La comunicazione del numero di clienti funziona così: quando un cliente si mette in coda alla cassa i, aumenta di uno il valore di array_cassieri_numero_clienti_in_coda[i], mentre quando il cassiere serve qualcuno diminuisce di uno questo valore (diventa 0 quando la cassa chiude). Ogni D millisecondi, il cassiere i copia il valore attuale di array_cassieri_numero_clienti_in_coda[i] in array_cassieri_numero_clienti_in_coda_direttore[i]. Sarà quest'ultimo l'array che il direttore controllerà quando dovrà decidere se aprire o chiudere una cassa. Inoltre il cassiere manda un segnale al direttore per informarlo del fatto che è avvenuto un cambiamento nel supermercato (il numero di clienti in questa cassa è ora più aggiornato). Infine, se deve chiudere, esce dal ciclo while per poi aggiornare i dati globali della sua cassa (clienti serviti, prodotti scannerizzati, numero di volte che ha chiuso e tempo di apertura).

Thread direttore: Il direttore, all'inizio, inizializza una propria struct contenente le informazioni di suo interesse (S1, S2, C, E, K, D e CasseAperteInit). Poi crea due array di pthread_t: uno lungo K per i cassieri e uno lungo C per i clienti. Dopodiché crea un numero di thread cassieri pari a CasseAperteInit e C clienti (in questo ordine), settando lo stato dei clienti a 0. Inizia poi un ciclo while che rappresenta la sua vita quando ancora non sono arrivati segnali di chiusura. Innanzitutto, se il direttore ha già dato uno sguardo alle casse (ovvero, ha già valutato se fosse il caso di chiuderne o aprirne qualcuna), fa un clock_gettime per segnarsi l'istante corrente. Subito dopo, controlla se ci sono clienti che non hanno fatto acquisti in attesa del suo permesso per uscire. Se ci sono, li fa uscire, e setta il loro stato a -1. A questo punto il direttore controlla quanti clienti sono presenti nel supermercato (ovvero, quanti hanno uno stato pari a 0, 1, 2 o 3). Se ce ne sono meno di C-E, "riempie" il supermercato, così che tornino ad essercene 50. I clienti non sono creati a caso: quelli che avevano come stato -1, ovvero che erano usciti ma di cui non era ancora stata liberata la memoria, vengono joinati. Il loro stato viene messo quindi a -2, solo per poi essere subito modificato a 0 con conseguente creazione di un nuovo thread (con lo stesso id (% C) di quello appena joinato) . Questo doppio assegnamento serve solo a rendere chiaro il fatto che prima la memoria di un cliente terminato viene liberata, e poi ne viene creato uno nuovo. Dato che ogni cliente, dopo essere stato joinato, viene subito ri-creato, ci si potrebbe chiedere se non sia superfluo avere lo stato -2 per i clienti. La sua utilità si manifesterà al momento della trattazione della chiusura del supermercato.
Dopo aver verificato la presenza di sufficienti clienti nel supermercato, il direttore fa una seconda clock_gettime, per poi fare la differenza tra questa e la prima. Se il risultato, in millisecondi, è superiore a D, va a controllare la situazione alle casse. Innanzitutto fa un primo controllo delle casse, segnandosi quante

sono aperte, quante hanno 0 o 1 cliente in coda e se esiste una cassa con più di S2 clienti. Successivamente decide se aprirne o chiuderne una. Attenzione che le due cose si escludono a vicenda: se una cassa viene aperta, non se ne può chiudere subito un'altra. Questo perché, se il direttore decidesse di aprire una cassa per poi chiuderne subito un'altra, non avremmo grandi risultati se non dei clienti costretti (forse, dato che la cassa chiusa potrebbe anche avere 0 clienti in coda) a cambiare cassa. Il primo controllo è in positivo: il direttore valuta se è il caso di aprire una cassa o meno. Se ci sono meno di K casse aperte e è stata rilevata una cassa che ha almeno S2 clienti in coda, viene aperta una nuova cassa. Se la cassa non viene aperta, il direttore controlla se è il caso di chiuderne una. Se ci sono almeno due casse aperte (vogliamo che almeno una cassa sia sempre aperta) e almeno S1 casse hanno o 0 o 1 cliente in coda, allora una di queste casse poco frequentate viene chiusa. Nel frattempo, un cassiere i generico potrebbe aver appena aggiornato array_cassieri_numero_clienti_in_coda_direttore[i], settando a true la variabile notifica_direttore_cambiamento e mandando un segnale al direttore. Se questo è vero, il direttore ripete il ciclo while. Altrimenti, si mette in attesa sulla cv CV_notifica_direttore_cambiamento, aspettando un segnale da un cassiere. Quando questo arriva, il direttore ricomincia con il ciclo while. Questa cv serve a evitare l'attesa attiva del direttore: quest'ultimo controlla la situazione dei clienti e dei cassieri solo dopo che ha ricevuto un segnale di cambiamento da un cassiere. Altrimenti il direttore rischierebbe di controllare molte volte di fila la stessa "fotografia" del supermercato. Il ciclo while termina qualora arrivi un segnale di sighup o di sigquit.
Segnale di sighup: il direttore deve prima far uscire tutti i clienti e poi far terminare i cassieri.
Il direttore controlla quante casse sono aperte, così da sapere quante andranno chiuse. Controlla poi se ci sono casse aperte che non hanno clienti in coda, e fintanto che almeno una cassa rimane aperta, chiude tali casse, joinandole per liberarne la memoria. Questo perché, in caso di sighup, i cassieri servono tutti i clienti e poi mandano un segnale al direttore per informarlo che hanno servito tutti i clienti che avevano in coda. Ma questo non succede se i cassieri non hanno clienti in coda (praticamente i cassieri sarebbero in attesa di un segnale dal direttore, che però può inviarglielo solo dopo che ne ha ricevuto uno dai cassieri stessi, e avremmo un attesa infinita). Dopo aver chiuso queste casse, abbiamo un numero casse_ancora_aperte_con_almeno_un_cliente_in_coda>=1 di casse che stanno ancora servendo dei clienti. Il direttore aspetta dunque che queste casse abbiano servito tutti i clienti che avevano in coda mettendosi in attesa su una cv. Ogni volta che una di queste casse serve tutti i suoi clienti, incrementa di uno la variabile sighup_numero_casse_concluse e invia un segnale al direttore. Quando il numero di casse concluse equivale a quello delle casse che stavamo aspettando, il direttore si risveglia. Controlla poi se ci sono clienti in attesa del suo permesso di uscire, e nel caso li fa uscire. A questo punto il direttore controlla lo stato di tutti i clienti. Quelli con stato pari a -1 vengono joinati dal direttore, che setta poi il loro valore a -2 così da sapere che sono usciti e la loro memoria è stata liberata (non vanno più guardati insomma). Se dopo questo controllo risulta che tutti i clienti sono usciti e la loro memoria è stata liberata, il direttore può terminare. Altrimenti si mette in attesa di questi clienti, che non appena escono (il loro stato diventa -1) mandano un segnale al direttore, il quale ricomincia il ciclo while per essenzialmente liberare la memoria di questi clienti e ricontare quelli ancora rimasti. Questo si ripete fino a che lo stato di tutti i clienti non è a -2. Per questo era essenziale assicurarsi che almeno una cassa rimanesse aperta: potrebbero infatti esserci clienti "ritardatari", cioè che stavano ancora vagando tra gli scaffali quando è arrivato il sighup. Lasciando una cassa aperta questi clienti sono potuti uscire dal supermercato senza problemi. Prima di terminare, il direttore chiude le ultime casse rimaste aperte, liberandone la memoria.
Segnale di sigquit: il direttore deve chiudere le casse senza che i clienti presenti vengano serviti. Pertanto, come prima cosa, chiude tutte le casse, settando inoltre a true la variabile si_chiude_sigquit. Così facendo, i clienti che si metto a cercare una nuova cassa a seguito della chiusura di quella precedente entrano in un ramo condizionale dove impostano a -1 il proprio stato, per poi terminare senza scrivere nulla sul file di log dei clienti. Il direttore, contemporaneamente, si mette a contare il numero di clienti ancora presenti nel supermercato, liberando la memoria di quelli con stato pari a -1 (in modo analogo al sighup praticamente). Quando tutti hanno stato -2, il direttore termina, liberando la memoria allocata dinamicamente.

E' disponibile una debug mode. Basta cambiare lo 0 in un 1 alla decima riga del programma (#define DEBUG 0)

Per la generazione dei numeri casuali mi sono ispirato al codice presente a questo link:
https://stackoverflow.com/questions/43151361/how-to-create-thread-safe-random-number-generator-in-c-using-rand-r