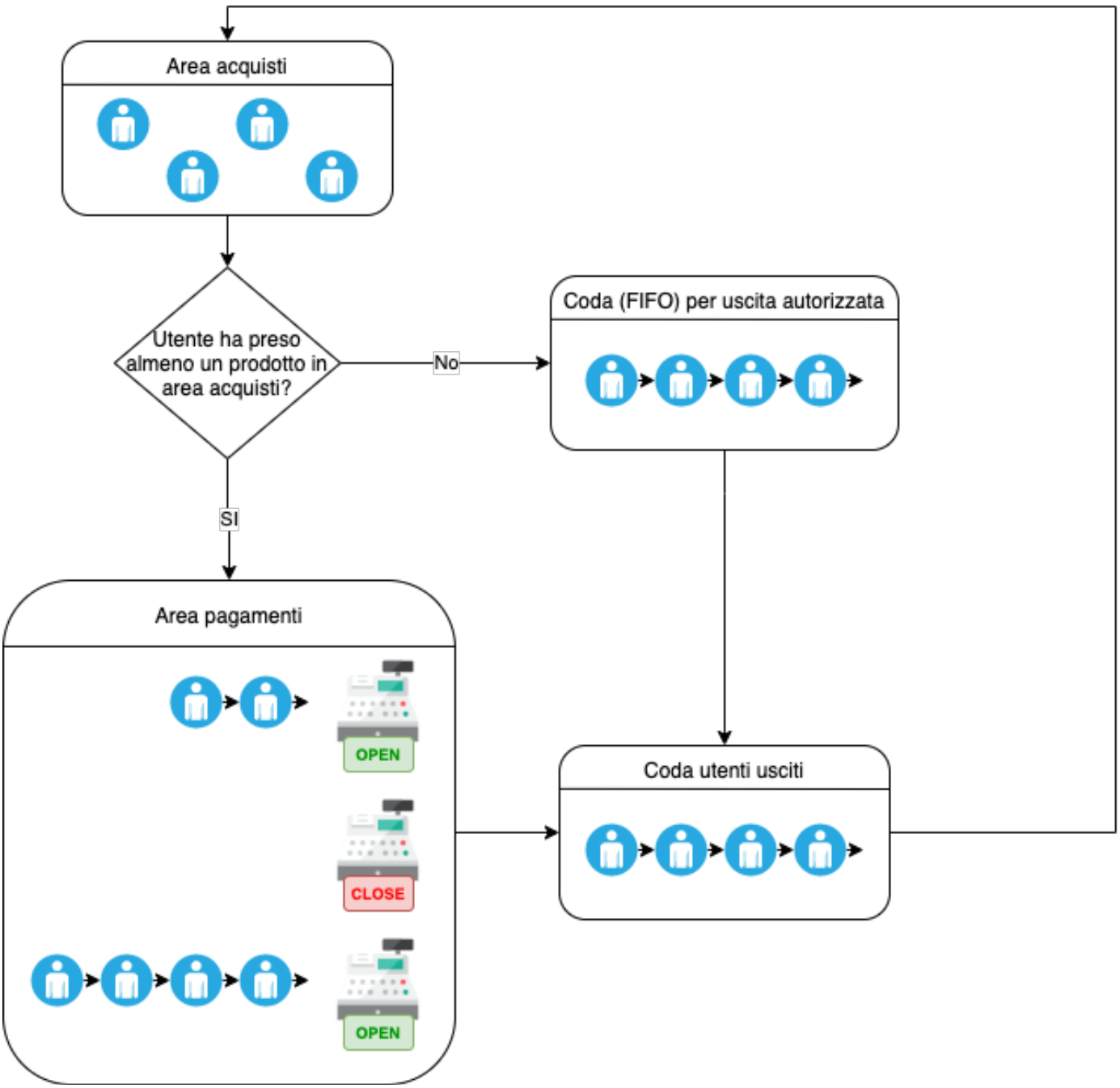
Progetto di laboratorio
Sistemi Operativi
a.a.2019/2020 - Corso A

Andrea Del Corto
matricola: 561446

1 Analisi del problema e scelte progettuali

1.1 Schema supermercato

La prima fase del progetto è stata quelle di definire chiaramente le entità coinvolte nel progetto. A tal proposito è stato realizzato il seguente schema che da una visione del supermercato incentrata sullo spostamento degli utenti durante la simulazione:



1.2 Thread

Successivamente è stato definito l'insieme dei thread e i loro rispettivi compiti:

1.2.1 Market thread

Il programma ha un unico thread di questo tipo che gestisce i seguenti aspetti della simulazione:

1. Inizializza tutte le strutture dati necessarie per avviare la simulazione di un supermercato.
2. Ogni volta che un utente arriva in "Coda usciti" le sue informazioni vengono registrate nel file di log e subito dopo vengono resettate per una futura simulazione.
3. Quando in "Coda usciti" arrivano E utenti, essi vengono spostati nuovamente in "Area acquisti" per una nuova simulazione. Questo riutilizzo delle strutture dati permette di ridurre sensibilmente il numero di chiamate malloc e pthread_create, con una conseguente riduzione del consumo delle risorse computazionali.
4. Quando un segnale di chiusura (SIGHUP o SIGQUIT) viene ricevuto, questo thread smette di reintrodurre nuovi utenti nel supermercato e si occupa di gestire la chiusura facendo terminare in modo controllato tutti gli altri thread. In particolare la chiusura viene gestita in questo modo:
    a. Viene inviato un segnale a tutti i CashDesk service thread che eseguono la loro procedura di chiusura.
    b. Dopo che tutti i CashDesk service thread sono terminati viene inviato un segnale anche ai thread: Director desk handling e Director authorization queue che a loro volta eseguono la loro procedura di chiusura.
    c. Una volta terminati anche i thread Director desk handling e Director authorization queue, tutti gli utenti presenti in "Coda usciti" vengono informati della chiusura del supermercato in modo da far partire la loro procedura di chiusura che porta alla terminazione del loro thread utente.
    d. Le strutture dati allocate dinamicamente durante la simulazione, vengono deallocate.

1.2.2 User thread

Il programma ha C thread di questo tipo e ognuno di essi gestisce i seguenti aspetti della simulazione:

1. Gestisce una struttura dati "User" che contiene le informazioni relative ad un utente.
2. Attende un periodo di tempo, definito in modo casuale compreso fra 10 e T millisecondi per simulare tempo di shopping.
3. La struttura dati "User" gestita dal thread viene spostata in una cassa aperta di "Area pagamenti" scelta casualmente se è stato preso almeno un prodotto, altrimenti viene spostata in "Coda (FIFO) per uscita autorizzata". A questo punto il thread utente rimane in attesa di procedere con un nuovo ciclo di esecuzione. Ciò avverrà solo quando il thread Market metterà nuovamente l'utente in "Area acquisti" per una nuova simulazione, oppure quando il thread sarà informato che il supermecato sta chiudendo. Nel secondo caso il thread User termina.

1.2.3 CashDesk service thread

Il programma ha K thread di questo tipo e ognuno di essi gestisce i seguenti aspetti della simulazione:

1. Gestisce una struttura dati “CashDesk” che contiene le informazioni relative ad una cassa.
2. Monitora i cambi di stato della cassa (aperta/chiusa) per tenere traccia del tempo totale di apertura e del numero di chiusure.
3. Ogni utente che viene inserito nella coda della cassa viene servito simulando un’attesa di servizio pari al seguente tempo:

T(n) = c1 + n*NP n: numero di prodotti presi dal ciente c1: tempo costante diverso per ogni cassiere compreso fra 20-80 ms (viene determinato in modo casuale alla creazione della struttura dati CashDesk) NP: è un parametro del file di configurazione che definisce il tempo in millisecondi necessario per processare un prodotto.

4. Dopo essere stato servito l’utente viene spostato in “Coda utenti usciti”, in modo da permettere al Market thread di utilizzarlo per una nuova simulazione.
5. Quando il thread riceve il segnale da parte del Market thread che lo informa della chiusura del supermercato possono esserci due possibili esiti:
   a. Se il supermercato sta chiudendo a causa di un segnale SIGHUP (chiusura normale) tutti gli utenti in coda vengono serviti normalmente.
   b. Se il supermercato sta chiudendo a causa di un segnale SIGQUIT (chiusura normale) tutti gli utenti in coda vengono fatti uscire senza pagare.

1.2.4 CashDesk notification thread

Ogni “CashDesk service thread” istanzia un thread ausiliario dedicato all’invio delle notifiche al thread direttore (id della cassa, numero utenti in coda nella cassa, stato della cassa). Ogni TD millisecondi (TD è un parametro del file di configurazione), questo thread inserisce una apposita struttura dati contenente le informazioni di stato della cassa in una coda gestita dal thread direttore. Quando il thread riceve un segnale di chiusura si limita a terminare.

1.2.5 Director desk handling thread

Il programma ha un unico thread di questo tipo e si occupa di gestire i seguenti aspetti della simulazione:

1. Gestisce le notifiche ricevute da i thread ”CashDesk notification thread”:
   Questo thread ha un vettore di notifiche chiamato “lastReceivedMsg” di K elementi il cui scopo è quello di tenere traccia delle notifiche più recenti relative a ciascuna cassa. Inizialmente gli elementi di questo vettore hanno valore “NULL” e ogni volta che viene ricevuta una notifica viene messa nel vettore “lastReceivedMsg” nella posizione relativa alla cassa che ha generato la notifica. Quando il vettore non contiene più elementi uguali a NULL significa che tutte le casse hanno inviato almeno una notifica ed è quindi il momento di decidere se aprire o chiudere una cassa.
   Questa decisione viene presa come descritto nel testo del progetto e la cassa da chiudere/aprire viene scelta in modo casuale. Gli utenti che si trovano in una cassa che viene chiusa vengono ridistribuiti in modo random fra le casse aperte.
2. Quando il thread riceve un segnale di chiusura si limita a terminare.

1.2.6 Director authorization queue thread

Il thread “Director desk handling thread” istanzia un thread ausiliario dedicato alla gestione della coda degli utenti in attesa di ricevere l’autorizzazione ad uscire. Questo thread si limita semplicemente a spostare gli utenti nella coda di uscita, in modo da permettere al Market thread di utilizzarli per una nuova simulazione. Quando il thread riceve un segnale di chiusura si limita a terminare.

1.2.7 Signal handler thread

Il programma ha un unico thread di questo tipo e si occupa di intercettare i segnali di chiusura del supermercato: SIGHUP (chiusura normale) e SIGQUIT (chiusura rapida) inviati al processo. Affinché il programma funzioni correttamente solo il “Signal handler thread” deve poter intercettare questi segnali, in questo modo i thread possono essere informati della chiusura in modo ordinato. A tal proposito il Signal handler thread rimane in attesa dei segnali utilizzando la funzione “sigwait”e gli altri thread vengono creati solo dopo aver inizializzato la maschera dei segnali del main() thread in modo che possano ignorare i seganli SIGHUP e SIGQUIT (i thread creati ereditano la maschera dei segnali del thread padre).
Quando un segnale di chiusura viene intercettato, viene inviato un segnale al Market thread in modo da dare il via alla chiusura.

1.3 Sincornizzazione dei thread

La maggior parte dei thread descritti nel capitolo precedente hanno in comune la seguente struttura:

```
1.  while(1) {
2.      Lock(&<lock>);
3.      while(!<cond_news>)
4.          pthread_cond_wait(&<cond_var>, &<lock>);
5.      Unlock(&<lock>);
6.      //….Resto del codice….
7.  }
```

<lock>: variabile di tipo pthread_mutex_t associata al thread corrente.
<cond_var>: condition variable (pthread_con_t) associata alla condizione <cond_news>.
<Lock> e <Unlock>: sono funzioni wrapper di pthread_mutex_lock e pthread_mutex_unlock che includono la gestione degli errori.

Questa struttura permette ai thread di andare in esecuzione solamente quando ci sono novità per loro ovvero quando la condizione <cond_var> è vera, evitando inutili sprechi di risorse.
Rispetto a questa struttura comune, fanno eccezione i seguenti thread: Signal handler thread e CashDesk notification thread in quanto essi devono sempre essere in esecuzione e non solo quando particolari eventi si verificano.

1.4 Seed per rand_r

I numeri casuali vengono generati utilizzando la funzione rand_r per la quale è necessario utilizzare un seed differente per ciascun thread. A tal proposito si presta bene la keyword **_Thread_local** disponibile a partire dallo standard c11. Questa keyword permette di definire una variabile per ogni thread, in questo modo è sufficiente definire una sola variabile globale g_seed da usare come seed.

2 Come compilare/eseguire il programma

2.1 Makefile

Il progetto è dotato di un Makefile il cui scopo è quello di semplificare il processo di compilazione ed esecuzione del programma. Il make file può essere usato come segue:

1. **make (oppure make all):** compila i file sorgente del progetto producendo 3 diversi eseguibili che vengono messi nella cartella "bin": main (è il programma supermercato), Test_config e test_squeue (sono programmi creati per verificare il funzionamento degli oggetti SQueue e delle funzioni che gestiscono i file di configurazione). Nota bene: per poter eseguire correttamente questo comando è necessaria la presenza della cartella obj con le stesse sotto cartelle di "src" (per esserne certi eseguire in sequenza make clean e make dir).
2. **make dir:** crea cartelle "obj" e "bin". La cartella "obj" viene creata con le stesse sottocartelle della cartella "scr".
3. **make doc:** esegue comando doxygen che, utilizzando il file "Doxyfile", genera la documentazione del progetto (maggiori dettagli nel capitolo 3.2).
4. **make clean:** rimuove cartelle "bin" e "obj", tutti i file da cartella "log"e file con estensione .PID.
5. **make docker_build:** viene creata l'immagine market (maggiori dettagli nel capitolo 3.1).
6. **make docker_run:** viene messa in esecuzione l'immagine market (maggiori dettagli nel capitolo 3.1).
7. **make test:** esegue il test richiesto per la versione semplificata del progetto. Il file di configurazione utilizzato è: configFiles/config_test.txt, mentre il file di log viene creato nel seguente percorso logFiles/log_test.txt. L'esecuzione di questo comando provoca la creazione di un file chiamato "main.PID" il cui unico scopo è quello di memorizzare il PID del processo supermercato che viene messo in esecuzione.
8. **make test_1:** uguale al comando "make test" con l'unica differenza che il segnale SIGHUP viene inviato dopo 5 secondi anziché 25.
9. **make test_2:** uguale al comando "make test" con l'unica differenza che il segnale che viene inviato è SIGQUIT e viene inviato dopo 5 secondi anziché 25.

2.2 Eseguire il programma

Il programma può essere eseguito nel seguente modo:

main <config_file_path.txt> <log_file_path.txt>

<config_file_path.txt>: path relativo ad un file di configurazione valido.
<log_file_path.txt>: path dove salvare log di simulazione.

Se esiste già un file in <log_file_path.txt> il programma chiederà se si intende proseguire sovrascrivendo il file oppure no. Dopodiché, se il file di configurazione <config_file_path.txt> esiste e contiene tutti i parametri necessari per configurare il supermercato, la simulazione comincia.

2.3 File di configurazione

Un file di configurazione valido deve contenere i seguenti parametri con questo formato <nome_parametro>=<valore_parametro>:

1. **K:** numero di casse presenti nel supermercato.
2. **KS:** numero di casse inizialmente aperte.
3. **C:** massimo numero di clienti presenti all'interno del supermercato.
4. **E:** numero di utenti che devono uscire dal supermercato affinché ne possano entrare altri E.
5. **T:** massimo tempo espresso in millisecondi che un cliente può spendere facendo acquisti
6. **P:** massimo numero di prodotti acquistabili da un cliente.
7. **S:** intervallo di tempo espresso in millisecondi seguito da ogni cliente per valutare se cambiare coda oppure no (in questa versione del progetto questo parametro non viene utilizzato).
8. **S1:** numero di casse con al più un cliente in coda necessario per poter chiudere una cassa aperta.
9. **S2:** numero di clienti che devono esserci in almeno una cassa per poter aprire una nuova cassa.
10. **NP:** numero di millisecondi impiegati da un cassiere per processare un prodotto.
11. **TD:** intervallo di tempo espresso in millisecondi che definisce la cadenza con cui ogni cassa deve informare il direttore del suo stato corrente.

I parametri sopracitati devono rispettare i seguenti vincoli:

1. K >= 1
2. 0 < KS <= K
3. C >= 1
4. 0 < E <= C
5. T > 10
6. P > 0
7. S > 0
8. 0 < S1 <= K
9. 0 < S2 <= C
10. NP > 0
11. TD > 0

2.4 Analisi.sh

Lo script "analisi.sh" mostra un sunto della simulazione utilizzando un file di log generato dall'esecuzione del programma.
Lo script deve essere utilizzato in questo modo:

analisi.sh <log_file_path.txt>

<log_file_path.txt>: path relativo ad un file di log di simulazione.

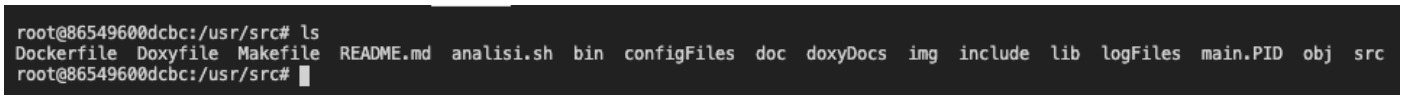
3 Parti supplementari

Questo capitolo descrive brevemente le parti del progetto non richieste dal testo che sono state fatte per approfondire argomenti reputati interessanti.

3.1 Docker

Il progetto è stato interamente sviluppato su una macchina virtuale con Ubuntu 16.04, ma il programma potrebbe essere valutato anche su Debian. Essendo entrambi sistemi UNIX non dovrebbero esserci problemi nel compilare ed eseguire il progetto su entrambi, ma per esserne più sicuri sono state fatte delle prove anche su Debian. Per poter effettuare le prove su Debian è stata utilizzata un'apposita immagine Docker derivata dall'ultima release stabile di Debian il cui nome è: market.
All'interno del progetto è stato definito un file chiamato: "Dockerfile" il cui scopo è quello di definire l'immagine market. L'immagine market viene configurata in modo che abbia i seguenti tool necessari per compilare ed eseguire il progetto: gcc, valgrind e make.
Utilizzando il comando "make docker_build" l'immagine verrà creata e potrà essere messa in esecuzione con il comando "make docker_run" (questi comandi del makefile del progetto sono utilizzabili solamente se Docker è installato). Dopo aver eseguito il comando "make docker_run" nel terminale dovrebbe presentarsi una situazione di questo tipo:

Questo significa che l'immagine market è stata messa in esecuzione correttamente, pertanto da questo momento in poi tutti i comandi che saranno digitati saranno eseguiti in ambiente Debian. Il terminale del container (immagine in esecuzione prende questo nome secondo la terminologia Docker) market di default viene posizionato nella cartella "usr/src" che contiene tutti i file del progetto che si trovano sull'host. Infatti eseguendo il comando ls si vedono i file del progetto presenti sull'host.



Per terminare l'esecuzione del container market è sufficiente eseguire il comando exit.

3.2 Doxygen

Per questo progetto è stato utilizzato Doxygen: un tool che permette di generare automaticamente la documentazione di un progetto scritto in C++, C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL o in D (supportato solo in parte).
I commenti presenti nei file sorgenti sono stati definiti seguendo lo standard Doxygen descritto sulla pagina ufficiale: https://www.doxygen.nl/manual/starting.html#step3 in modo da poter generare la documentazione con doxygen.
La documentazione generata con Doxygen, sotto forma di pagine html, si trova nella seguente cartella: "doxyDocs" (index.html è la home page).