# Struttura

Typedef enum statoarticoloT {

STORAGE, //magazzino

TOBEDELIVERED, //da consegnare

DELIVERED, // consegnato

COLLECTED // ritirato

} status;

Typedef struct listT {

char codice\_articolo[32]; // 000000000

char descrizione\_articolo[256];

char indirizzo\_destinazione[256];

status stato\_articolo;

pthread\_mutex\_t sem;

} PackageList;

//esempio di codice articolo basato su tempo

codice\_articolo:

#include <time.h>

struct timeval tv;

gettimeofday(&tv,NULL);

//timestamp

tv.tv\_sec // seconds

tv.tv\_usec // microseconds

//192.168.100.100.45.2000

# Client

* 2 argv: IP server, porta SERVER
* Connessione al server
  + Riceve al massimo n elementi K, li salva nella lista linkata L
  + Stampa a video la lista degli elementi
* Comandi Client:
  + elenca (distingui in “consegna” da “ritirato”)
  + elencaserver
  + consegnato#codice
  + ritirato#codice#descrizione#indirizzo
  + smista#codice
* il server risponde e il client elenca a video
* quando la coda L locale termina, il client esce
* elencaserver elenca tutta la coda originaria del client o tutti gli articoli del server?
* Posso usare gli enum?

# Funzioni

* Funzione di controllo input argc, argv
* Funzione di salvataggio degli elementi su struttura locale
* Stampa elenco locale
* Consegnato#codice: modifica da “in consegna” a “consegnato” in remoto. Cancella in locale. Gestione errori­: correttezza del codice
* Ritirato#codice#descrizione#indirizzo: aggiunge a elenco locale degli articoli. Viene caricato su server con Smista#codice
* Smista#codice: passa da “ritirato” a “in magazzino” , lo carica sul server e lo elimina dalla lista locale. Eccezione: non chiudere il programma prima che la lista locale sia vuota
* Funzione di riconoscimento codice
* Funzione di riconoscimento comando
* Funzione di invio messaggio
* Controllo connessione server

# Server

* 3 argv:
* N operatori
* K oggetti per operatore
* File S con oggetti da consegnare
* File S:
* Codice#Descrizione#indirizzo

# Funzioni

* Check input argv
* Check nome file in ingresso
* Check coerenza file di testo?
* Lettura file di testo
  + Salvataggio su lista
  + Aggiornamento della lista
* Scrittura su file di testo alla chiusura?
* 3D:

**Master Thread**

* + Master thread prende parametri, legge file, crea il socket, effettua tutti i controlli.

Passa il socket al 3d gestore delle connessioni

**Thread figli**

* + 1 thread gestisce le accept dei client
  + 1 thread gestisce il singolo client: ricezione messaggi, aggiornamento della struttura dati condivisa che impiega i mutex.
  + Pthread join per impedire che il Master Thread termini prima dei figli
  + Allocazione dinamica del mutex nella struttura: if (pthread\_mutex\_init(&fp->f\_lock, NULL) != 0)

# Protocollo di comunicazione

Lista vuota o terminata: NULL#

Caso 1: Ricezione lista iniziale

* Id#descrizione#indirizzo su k stringhe
  + Il server all’atto dell’invio modifica lo stato dell’oggetto nel suo database in “in consegna”.
  + Il client imposta lo stato nella lista locale dell’oggetto in “in consegna”.

Caso 2: elencaserver

Il server invia tutte le stringhe in memoria:

* Elencaserver#id#descrizione#indirizzo#stato

Caso 3: ritirato

* Ritirato#codice#descrizione #indirizzo
* Aggiunge l’oggetto in locale e sul server

Caso 4: smista

* Smista#codice
* Codice nel server passa da “ritirato” a “in magazzino”
* In locale l’oggetto viene eliminato

# Link utili

/\*\*

Master thread + thread per client

\*/

<https://gist.github.com/silv3rm00n/5821760>

/\*\*

Guida facile su socket

\*/

<http://www.binarytides.com/socket-programming-c-linux-tutorial/>

# Dubbi

* Per ogni thread creato è richiesto il salvataggio dell’t\_id specifico?
* Potrebbe essere necessario creare un nuovo comando a basso livello, EXIT:

Quando il client termina il suo lavoro, invia automaticamente un comando EXIT al suo Thread di riferimento, in modo da gestirne la chiusura. Potrebbe essere inutile in quanto il Thread cicla fintanto che il socket resta aperto.

# Funzione lettura file di testo

|  |
| --- |
| Parametri in ingresso:  * File descriptor di un file di testo * Puntatore alla lista di oggetti PackageList * Numero di token per linea del file di testo  Valore restituito:  * Puntatore alla lista di oggetti PackageList |

La funzione riceve in ingresso una struttura dati contenente:

* File descriptor del file di testo
* Puntatore a lista oggetti

La funzione crea la lista PackageList composta da tutti gli elementi del file di testo.

La funzione scandisce del file di testo

Esempio di riga:

|  |  |  |
| --- | --- | --- |
| codice | descrizione | indirizzo |
| PKG0472 | pkg\_desc0472 | Piazza Martiri della Libertà - 10077 San Maurizio Canavese (TO) |

* Legge un carattere alla volta dal file di testo, salvando in maniera sequenziale i char all’interno di 3 stringhe di buffer: quando la funzione di scansione incontra un “#” o un “\n”, passa alla stringa di buffer successiva.
* Poi la funzione Controlla che le 3 stringhe non siano vuote e a quel punto, può passarle alla funzione di inserimento in coda (Accoda).
* Nel caso ci sia una stringa incorretta o vuota si invia un warning a schermo, si skippa alla riga successiva e si ricominica la lettura.
* passa le stringhe buffer alla funzione di inserimento in coda (Accoda) della lista PackageList. A questo punto, la funzione Accoda si occupa di caricare i campi della struttura. Ogni elemento accodato ovviamente inizializzato in modo che lo stato dell’articolo corrisponda a “In magazzino”. Il mutex viene inizializzato dinamicamente e viene “spento”.
* La funzione interrompe la scansione quando trova EOF e chiude il file descriptor del file di testo.

Questa funzione viene invocata all’avvio del programma (se passa i controlli sui parametri e sull’esistenza del file di testo) . mentre scrive la lista puntata deve bloccare il semaforo o può procedere senza curarsene?

Funzioni invocate: Inserisci in coda.

La funzione restituisce il puntatore alla lista allocata.

# Procedura di salvataggio su file

|  |
| --- |
| Parametri in ingresso:  * File descriptor di un file di testo * Puntatore alla lista di oggetti PackageList  Valore restituito:  * Procedura void |

La procedura prende in ingresso il file descriptor del file di testo in cui salvare lo stato corrente del “magazzino” e il puntatore alla lista degli oggetti.

La procedura scorre tutta la lista e salva i campi nel file di testo impiegando la codifica già analizzata:

codice\_articolo#descrizione#indirizzo\n

La procedura prende il controllo del semaforo e blocca la struttura.

La procedura può essere invocata in caso di errore di sistema ed utilizzata come una eccezione.

La procedura può essere invocata anche prima di una chiusura normale.

# Procedura di Aggiornamento della lista

|  |
| --- |
| Parametri in ingresso:  * Puntatore alla lista di oggetti PackageList * Struttura temporanea di tipo PackageList  Valore restituito:  * Procedura void |

Procedura del server che viene invocata quando un corriere deve **smistare un articolo.**

Prende in ingresso il puntatore alla Lista centrale e il puntatore ad una struttura temporanea in cui è presente l’articolo da inserire.

La procedura deve prendere il controllo della lista, quindi blocca il semaforo ed esegue le sue subroutines.

La procedura chiama una funzione di tipo Accoda che scorre la Lista e accoda l’articolo da inserire.

# Funzioni di libreria standard

* Aggiungi in coda (ricorsiva)
* Cancella elemento
* Cancella\_lista (ricorsiva)

# client Thread

pthread\_create( &thread\_id , NULL , connection\_handler , (void\*) &client\_sock)

|  |
| --- |
| Parametri in ingresso: Struttura dati void \* contenente:   * Puntatore alla lista di oggetti PackageList * Socket del client * Numero di oggetti da passare al client  Valore restituito:  * Funzione void \* |

**devo creare al più N thread per gestione del client.**

* Una volta avviato il clientThread, questo blocca la Lista PackageList e ne legge il contenuto per K volte.
* Spacchetta la struttura PackageList e utilizzando 3 stringhe di buffer ne salva localmente il contenuto.
* Cambia lo stato degli articoli letti da “in magazzino” a “in consegna”
* Inizia la comunicazione con il client: invia i 3 dati (codice#descrizione#indirizzo) ripetuti per K volte.
* Il clientThread si mette in ascolto dei comandi dal client, fintanto che il socket è aperto o fintanto che non riceve un comando EXIT dal client.
* In un ciclo while che termina quando la connessione è assente, il thread legge dal client una stringa salvata in buffer.
* La stringa viene spacchettata in una struttura temporanea di tipo PackageList e viene associato un numero intero per ogni genere di comando inviato dal client.
* Uno switch legge l’intero associato al comando, si porta sul caso corrispondente, **blocca il mutex** e chiama la subroutine necessaria alla risoluzione del comando. Terminato il comando sblocca il mutex e si ripete il ciclo.