Progetto Lab II

- Oltre ai 3 Parser richiesti sono presenti due file.c/.h per la gestione delle emergenze, c**lient.c** e **server.c**, insieme a **functions.c** che contiene tre funzioni riservate ai parser ed una per l'intero sistema, **utils\_server.c** creato con l'obiettivo principale di avere un’organizzazione più pulita con le funzioni che servono al server. Infine, **debug\_server.c** che contiene funzioni non richieste/necessarie ma che ho usato per il debugging del programma (oltre al file **events.log**).

- Ogni **file.c** ha ovviamente un relativo **file.h** in cui inizialmente definisco delle macro con il nome del file in grassetto per non avere problemi di ridefinizioni multiple (ovviamente non a tutti i **file.h**)

- Tutte le **typedef struct** richieste le ho messe in un unico file **data\_structure.h**. Inoltre, ho aggiunto altre strutture per aiutarmi con gli eventuali output, e funzionamenti del programma, che sono presenti in dei **file.h** separati.

- Per ogni problema con l'allocazione della memoria, apertura di un file, o per errore di sintassi, viene esplicitato il tipo di errore riportando il nome del file e la linea di codice.

- Le uniche chiamate di sistema che ho utilizzato per le operazioni nel file.log sono protette da **MACRO**.

- La memoria viene liberata in un colpo solo nel **server.c,** dato che tutti i riferimenti passeranno ad essa.

- Per compilare tutto, digitare **make** per poi avviare il client e il server a parte, oppure digitare **make run** compilando tutto e avviando subito il server.

- Segnalo che ho utilizzato due funzioni che non abbiamo visto a lezione, tra cui: int snprintf(char \*str, size\_t size, const char \*format, ...); per sovrascrivere un buffer con una stringa formattata, semplificando molte righe di codice; struct tm \*<var> = localtime(&<current time>); di cui ne parlerà più avanti.

* Come ho gestito le emergenze

- Nella griglia può essere gestita più di un'emergenza nella stessa posizione (x;y).

- Gli unici motivi per cui può essere scartata un'emergenza:

1) non c'è nemmeno un’istanza di un tipo di soccorritore necessario per l'emergenza;

2) uno o più soccorritori non riescono ad arrivare in tempo al punto d'emergenza per la priorità;

- Se per un’emergenza mancano una o più istanze di un soccorritore necessario, allora, per i gemelli digitali di quel tipo, gli verrà assegnato del tempo in più per ricoprire il tempo che non possono impiegare le istanze mancanti.

ESEMPIO: ho [incendio][...]Pompieri:2,4;Polizia:3,5; nel file **emergency.conf** e [Pompieri][2][...][...], [Polizia][1][...][...] nel file **rescuers.conf**.

In questo caso ci sono tutti i pompieri, quindi ogni pompiere gestisce l'emergenza per 4 secondi. Per quanto riguarda la polizia mancano due istanze, perciò quell'unica istanza deve ricoprire il lavoro degli altri poliziotti, dato che il tempo totale è dato da <istanze richieste poliziotti> \* <tempo dedicato> = 3 \* 5 = 15, verrano assegnati, ad ogni gemello digitale di tipo poliziotto (<istanze richieste poliziotti> \* <tempo dedicato>) / (istanze disponibili al momento) = 15 / 1 = 15. Quindi quel poliziotto dovrà essere lì presente per 15 secondi.

- Se per un’emergenza ci sono 0 istanze di un tipo di soccorritore disponibili, non per preimpostazione del file **rescuers.conf**, ma perché sono occupati per un'altra emergenza, allora viene inserita nella coda d'attesa.

- La coda d'attesa contiene le emergenze in ordine decrescente, per priorità, partendo dall'indice 0.

- Ogni tipo di emergenza che riceve il server deve avere: un nome presente nel file **emergency.conf** e le coordinate che rientrano nelle dimensioni della griglia.

- Più avanti verranno approfondite le implementazioni del server che riguarderà principalmente la gestione delle emergenze.

* Per ciascun parser

- Inizializzo l'id, l'event, e un array di caratteri all'inizio che servono per la scrittura nel file .log.

- Accedo a ciascun file con la **fopen()** verificando sempre se è andata a buon fine.

- Inizializzo degli array dinamici per la gestione della funzione, aiutandomi con l'eventuale output.

- Ogni riga del file la ottengo con la **fgets()** dove successivamente elimino gli spazi tra i caratteri della riga, semplificandomi il lavoro con la verifica della sintassi.

- Ad ogni ciclo utilizzo una **strtok()** con determinati delimitatori.

- Infine, l'output, in particolare per il **parse\_rescuers.c** e **parse\_env.c**, hanno un tipo **struct** aggiunto da me per avere a disposizione più di un elemento.

* functions.c

- Contiene 4 funzioni tra cui:

* char\* trim(char\* line): utilizzato per rimuovere gli spazi tra i caratteri semplificando il lavoro per la correzione della sintassi.
* void write\_log\_file(time\_t timestamp, char\* id, char\* evento, char\* desc): utilizzato per aggiornare il file.log.
* int format\_check\_rescuers(char\* line): verifico la sintassi della BNF del file **rescuers.conf.**
* int format\_check\_Emergency(char\* line): verifico la sintassi della BNF del file **emergency.conf**.
* int digits(int x): semplice funzione che utilizzo nel client per contare il numero di cifre di un numero.

* parse\_rescuer.c
* rescuer\_type\_t\* find\_rescuer(rescuer\_type\_t\*\* rescuers, char\* name, int num\_rescuers): Verifico se, tra i soccorritori disponibili in **rescuers.conf**, esiste un soccorritore utilizzando il parametro **char\* name.**
* void free\_rescuers\_digital\_twins(rescuer\_digital\_twin\_t\*\* rd\_twin, int n\_twins) e void free\_rescuers(rescuer\_type\_t\*\* rescuers, int num): Libero dall'heap l'array di gemelli digitali e l'array dei soccorritori disponibili.
* void print\_rescuers(rescuer\_type\_t\*\* rescuers, int num) e void print\_digitals\_twins(rescuer\_digital\_twin\_t\*\* rd\_twins, int num): funzioni che stampano tutti i campi dei soccorritori e dei gemelli digitali. In particolare, la funzione che stampa i gemelli digitali viene usata nel server per debugging tenendo traccia periodicamente dello stato de gemelli digitali, in particolare per la posizione e lo stato.
* result\_parser\_rescuers\* parse\_rescuers(char\* filename): funzione principale per ottenere i soccorritori, in particolare inserisco in una struttura che ho aggiunto (resul\_parser\_rescuer) che contiene l'array dei soccorritori e gemelli digitali disponibili con le loro grandezze.
* parse\_emergency\_types.c
* emergency\_type\_t\* parser\_emergency(char\* filename, int\* num\_emergency\_type): esegue il parsing del file \*\*emergency.conf\*\* restituendo un array contenente tutte le emergenze disponibili.
* void print\_emergencies(emergency\_type\_t\* emergencies, int num): stampa l'array delle emergenze disponibili.
* parse\_env.c
* env\_t\* parser\_env(char\* filename): restituisce un tipo di una struct aggiunta (\*\*env\_t\*\*) che contiene il nome della message queue e la grandezza della griglia. Per farsì che il parsing avvenga correttamente, ogni riga del file \*\*env.conf\*\* deve essere del tipo: queue=<string><matricola>, width=<int>, height=<int>. Le chiavi devono essere solamente quelle, come accennato dal testo del progetto.
* server.c

- Il server inizia prelevando i gemelli digitali disponibili, le possibili emergenze che si possono soddisfare e le variabili d'ambiente.

- Vengono avviati un numero di thread pari al numero di gemelli digitali, che infatti li rappresentano, e restano in esecuzione fino a quando la message queue è ancora operativa. Questi thread restano bloccati quando ancora non arriva qualcuno a chiamarli per un’emergenza.

- Quando il server riceve un’emergenza, verifica se è valida controllando se il nome dell'emergenza è tra quelle che si possono soddisfare e se le coordinate rientrano nella griglia.

- Se l'emergenza è valida viene avviato un thread per gestirla, impostando il tipo **emergency\_t** e verificando, dopo che gli sono stati assegnati tutti i tipi di gemelli digitali che servono, se può essere gestita in base alla presenza dei soccorritori.

- Un’emergenza può essere gestita solamente se esiste almeno un’istanza per ogni tipo di soccorritore (ad esempio, se ho [Incendio][<int>]Pompieri:2,3; e [Pompieri][1][<int>][//] l'emergenza può essere gestita lo stesso anche se non ci sono tutte le istanze) ma, ovviamente, ad ogni gemello, per un determinato tipo di soccorritore, gli deve essere distribuito del tempo in più per poter ricoprire il tempo che avrebbero dovuto impiegare i soccorritori mancanti (nel caso di istanze mancanti).

- Una volta aver completato il settaggio dell'emergenza e aver verificato che si può soddisfare, viene avviata la funzione **start\_emergency** in cui, attraverso un ciclo poco dopo l'inizio della funzione, vengono iterati tutte le istanze disponibili dei gemelli per ogni tipo di soccorritore necessario e, se per un tipo di soccorritore non c'è nemmeno un’istanza disponibile al momento, l'emergenza va nella coda d'attesa.

- per l'occupazione dei gemelli digitali ho aggiunto una struttura **rescuer\_t** che serve per ricordare gli id dei gemelli che occupo al momento, ci sono due possibili casi:

1) se ci sono 0 istanze disponibili per un soccorritore, oltre ad aggiungere l'emergenza nella coda di attesa, libero le lock che avevo occupato dei gemelli digitali precedenti, evitando così il deadlock con altre emergenze a venire;

2) se per ogni soccorritore necessario esiste almeno un’istanza disponibile, allora l'emergenza può essere avviata dopo aver occupato tutti i lock dei gemelli digitali che servono. Se per un determinato tipo di soccorritore ci sono tutte le istanze richieste, allora per ogni gemello, sempre di quel tipo, verrà assegnato un tot tempo pari a quello specificato nel file \*\*emergency.conf\*\*, altrimenti ad ogni gemello di quel tipo verrà aggiunto del tempo in più per coprire il tempo dei soccorritori assenti;

- per quanto riguarda le emergenze nella coda di attesa, c'è un thread avviato nel main che si occupa di gestire le emergenze in attesa, bloccato inizialmente da un semaforo, definito globalmente, che può procedere quando c'è almeno un’emergenza in attesa. In alcune parti del programma viene fatto un controllo per verificare che continui a sorvegliare la coda d'attesa quando c'è almeno un elemento. Il procedimento viene fatto controllando il valore attuale del counter del semaforo e, se è uguale a 0 mentre c'è almeno un elemento in coda, allora viene portato a 1.

- Tutte le operazioni che si fanno sull'array delle emergenze in generale, e sull'array delle emergenze in attesa, vengono fatte in modo sicuro evitando fault e problemi vari per via della protezione da parte di due lock distinte: uno per la coda d'attesa e l'altro per le emergenze globali.

- Ogni soccorritore, per essere chiamato per un’emergenza, deve essere necessariamente alla sua base.

- Infine, ho voluto creare due funzioni per l'ottimizzazione del sistema, che sono **control\_waiting\_queue** e **check\_priority\_waiting\_queue** di cui se ne parlerà più avanti.

- Ho dovuto aggiungere delle strutture utili principalmente per la sincronizzazione e per l'organizzazione dell'intero programma.

* Descrizione delle struct aggiuntive

- emergency\_id\_t: ho voluto creare questa struttura per identificare le emergenze in modo univoco con un intero, creando un array dinamico di questo tipo includendo tutte le emergenze che sono state inviate (VALIDE) per avere una panoramica completa di quello che è successo (potendo vedere quali emergenze sono state soddisfatte e non). Oltre all'id ci sono le variabili: **int num\_twins** intero che identifica tutti i gemelli digitali che sono stati salvati nell'array **rescuers\_dt** nella variabile di tipo **emergency\_t**; **bool was\_in\_waiting\_queue** variabile che, inizialmente falsa, mi tiene conto se l'emergenza in questione è passata nella coda di attesa, utile per il debugging e per la rimozione dalla coda; **bool in\_loading** altra booleana utilizzata nella funzione che gestisce la coda d'attesa per verificare che, l'emergenza prelevata dalla coda d'attesa, non sia già in esecuzione; **mtx\_t lock\_emergency** e **cnd\_t cnd\_emergency** servono per la barrier. Dato che nel testo è stato specificato "ovvero avere tutti i mezzi necessari collocati sul luogo dell’emergenza, che a quel punto passa nello stato IN\_PROGRESS" ho voluto farsi che un’emergenza per essere eseguita, i soccorritori devono cominciare a lavorare insieme e successivamente agiscono indipendentemente per conto loro. Per cui, grazie alla **struct rescuers\_data\_t** che possiede l'id dell'emergenza da soddisfare, può accedere alla lock e cv, condivisa con altri soccorritori della stessa emergenza, per sincronizzarsi con la barrier; la variabile atomica **rescuers\_finished** indica il numero corrente dei soccorritori che sono arrivati alla barrier, mentre **tot\_rescuers\_finished** indica il numero di soccorritori che devono essere presenti in quell'istante (anche queste due ovviamente servono per la barrier).

- waiting\_queue\_t: struttura che possiede l'id, priorità e descrizione dell'emergenza che è attualmente in attesa. Il campo fondamentale è l'id per identificare l'emergenza nell'array globale per le emergenze, potendo identificare correttamente l'emergenza che mi serve.

- params\_handler\_queue\_t: struttura che contiene le variabili d'ambiente e l'array dei gemelli digitale e dei soccorritori, che verrà passata al thread che gestisce la message queue.

- rescuer\_data\_t: struttura da supporto per ogni gemello digitale, contiene: la lock e la cv per la sincronizzazione; le coordinate del punto d'emergenza (x,y); la descrizione dell'emergenza (aggiunta per comodità); il tempo che deve dedicare il soccorritore per il suo lavoro (time\_to\_manage); id\_current\_emergency è l'id dell'emergenza di cui si sta occupando, utilissima per poter prelevare informazioni di questa emergenza tramite l'array globale delle emergenze; infine **rescuer\_digital\_twin\_t\* twin** in cui sono salvate le informazioni del gemello digitale, inizializzate nel main.

- params\_handler\_emergency\_t e params\_control\_waiting\_queue\_t: strutture d'appoggio per il passaggio dei parametri a due funzioni eseguite da thread.

* Funzioni
* handler\_queue\_emergency(void\* args): Si occupa della gestione della message queue ricevendo le emergenze attraverso la struct \*\*emergency\_request\_t\*\*, verifica se è valida in termini di nome e coordinate e, se così fosse, verrebbe mandato in esecuzione un thread che gestisce il seguito.
* int handler\_emergency(void\* args): si occupa del settaggio dell'emergenza ed esegue una funzione (rescuers\_is\_avalaible) che verifica se è possibile gestire l'emergenza in termini di soccorritori, se non esiste nessuna istanza di un determinato soccorritore, non può essere gestita.
* int start\_emergency(emergency\_id\_t\* current\_emergency): si occupa di provare ad avviare l'emergenza. Inizialmente vengono iterati i gemelli digitali richiesti per quell'emergenza e se c'è almeno un’istanza disponibile per ogni soccorritore allora si può avviare. Prima di avviarla viene verificata se esiste un soccorritore che non può arrivare in tempo per gestire l'emergenza, in base alla priorità, in quel caso viene scartata passando allo stato TIMEOUT (ovviamente se la priorità è diversa da 0), se sono in grado di arrivare in tempo l'emergenza viene avviata risvegliando ogni gemello digitale assegnato per l'emergenza settando tutti i campi del tipo **rescuers\_data\_t** per ogni gemello.
* void free\_locks\_rescuers(rescuers\_t\*\* id\_locks, int count): questa funzione viene chiamata per rilasciare i lock attualmente acquisiti dei gemelli digitali quando un’emergenza non può essere avviata a causa di distanze del soccorritore, oppure per mancanza di tutte le istanze di un soccorritore. Funzione fondamentale per evitare il deadlock.
* int handle\_rescuer(void\* args): permette di gestire indipendentemente ogni gemello digitale, resta in esecuzione fino a quando la message queue è operativa.
* int handler\_waiting\_queue(void\* args): funzione che viene avviata da un thread nel main che si occupa di gestire le emergenze in attesa. È gestita da un semaforo definito globalmente dove in alcune zone del programma viene verificato il valore attuale del counter del semaforo e, se è uguale a 0 mentre ci sono delle emergenze in attesa, allora viene incrementato ad 1. Quest'operazione viene protetta dalla lock **lock\_operation\_on\_waiting\_queue** in modo da non far andare ad un valore maggiore di 1 il valore del counter.
* int control\_waiting\_queue(void\* args): funzione opzionale di ottimizzazione. Resta in esecuzione fino a quando la message queue è operativa e controlla periodicamente, dopo un tot di secondi, se le emergenze in attesa possono essere soddisfatte in termini di distanza di qualche soccorritore, in caso contrario vengono eliminate dalla coda d'attesa prima ancora che vengano prelevate. L'idea è quella di mantenere pulita e leggera la coda d'attesa senza tenere emergenze inutili che non possono essere più soddisfatte.
* Utils\_server.c

- file.c pensato per tenere un’organizzazione più pulita che contiene delle funzioni necessarie per il funzionamento del server.

* Funzioni
* int barrier\_rescuers(emergency\_id\_t\* current ,atomic\_int\* count, atomic\_int\* tot\_rescuers\_required, mtx\_t\* mtx, cnd\_t\* cnd, bool is\_active): barrier che serve per far cominciare a lavorare i gemelli nello stesso momento. Inizialmente, il programma era implementato in modo che ogni gemello appena arrivato al posto d'emergenza, indipendentemente dagli altri gemelli, iniziava a lavorare, ma siccome nel testo viene accennato che lo stato dell'emergenza passa nello stato IN\_PROGRESS nel momento in cui tutti sono collocati nella zona d'emergenza, allora ho implementato questa barrier. Dopo la barrier i gemelli si comportano in modo indipendente.
  + int get\_priority\_limit(int priority): restituisce il tempo massimo entro cui bisogna cominciare a gestire l'emergenza in base alla priorità.
* int distance\_manhattan(int x1, int x2, int y1, int y2, int speed): tempo di arrivo stimato di un soccorritore.
* int rescuers\_is\_avalaible(rescuer\_digital\_twin\_t\*\* rd\_twins, int num\_twins, rescuer\_request\_t\* requests, int req\_number, char\* desc): verifica se sono disponibili i soccorritori per gestire l'emergenza. Può restituire 1,0 e -1 a seconda dei casi.

1) (-1) se non esiste nessuna istanza per un determinato tipo di soccorritore, in questo caso l'emergenza va scartata. (ESEMPIO: [Incedio][...]Pompieri:4,2 - [Pompieri][0][...][...;...])

2) 0 se esiste almeno un’istanza per ogni soccorritore necessario, ma non tutte quelle richieste per uno o più soccorritori (ESEMPIO: [Incedio][...]Pompieri:4,2 - [Pompieri][2][...][...;...]).

3) 1 se esistono tutte le istanze richieste per ogni tipo di soccorritore (ESEMPIO: [Incedio][...]Pompieri:4,2 - [Pompieri][4][...][...;...]).

* int compare(const void \*a, const void \*b): funzione utilizzata per riordinare la coda d'attesa, dopo l'inserimento di una nuova emergenza, in base alla priorità.
* void add\_waiting\_queue(emergency\_id\_t\* id, waiting\_queue\_t\*\*\* waiting\_queue, int\* waiting\_queue\_len): funzione per aggiungere, in fondo, un’emergenza nella coda d'attesa.
* void remove\_from\_waiting\_queue(emergency\_id\_t\* emergency, waiting\_queue\_t\*\*\* waiting\_queue, int\* waiting\_queue\_len): funzione per rimuovere un'emergenza dalla coda d'attesa.
* emergency\_t\* set\_new\_emergency(params\_handler\_emergency\_t\* params\_emergency, int\* twins): settaggio dell'emergenza corrente, in particolare imposto il tipo\*\*emergency\_t\*\*.
* emergency\_id\_t\* add\_emergency(int\* id, emergency\_t\* emergency, emergency\_id\_t\*\*\* queue\_emergencies): aggiungo un’emergenza nell'array delle emergenze totali (valide).
  + int is\_valid\_request(emergency\_request\_t\* request, int width, int height, char\* problem, int\* index\_type\_emergency, emergency\_type\_t\* emergency\_avalaible, int num\_emergency\_avalaible): verifico che l'emergenza appena arrivata sia valida in termini di posizione e di disponibilità dell'emergenza. Nel caso in cui non sia valida, viene restituito 0 scrivendo il problema nella variabile \*\*char\* problem\*\*. Se è valida restituisce 1 impostando il valore dell'indice delle emergenze potendo identificare che tipo di emergenza devo gestire nell'array delle emergenze.
  + int check\_priority\_waiting\_queue(int current\_index, waiting\_queue\_t\*\*\* waiting\_queue, int waiting\_queue\_len, emergency\_id\_t\*\*\* queue\_emergencies): questa è un'altra funzione di ottimizzazione (opzionale). Durante l'esecuzione, se ci sono più elementi nella coda d'attesa è probabile che ci siano delle emergenze diverse ma con la stessa priorità che potrebbero essere eseguite. Quindi questa funzione è pensata per visionare delle emergenze diverse in attesa, con la stessa priorità di quella al primo posto, che potrebbero essere avviate restituendo l'indice dell'emergenza nell'array globale. Permette di alleggerire la coda.

- Il resto delle funzioni serve per liberare la memoria.

* + debug\_server.c

- file opzionale che contiene delle funzioni che utilizzo per il debbugging del programma (oltre al file.log). In particolare, contiene una funzione per stampare le emergenze in attesa e un'altra per stampare lo stato attuale dei gemelli digitali, principalmente per tenere traccia della loro posizione e del loro status.

* + client.c

- È possibile inviare da terminale una singola emergenza o più emergenze contenute in un file di estensione .txt o .conf.

- Alternativamente, il programma può essere avviato senza alcun tipo di parametro, semplicemente finisce in un ciclo in cui viene fornito un menù che permette di fare le stesse cose che ho citato in precedenza, con la differenza che il client rimane in esecuzione fino a quando non viene digitato un certo valore.

- È dal client che è possibile far terminare la message queue.

* Funzioni
  + int send\_queue(char\* filename, emergency\_request\_t\* msg): permette di inviare l'emergenza al server. \*\*filename\*\* è il nome del file \*\*parse\_env.c\*\* per il nome della message queue, mentre \*\*msg\*\* racchiude il nome e le coordinate dell'emergenza.
  + void stop\_message\_queue(): permette di far terminare la message queue del server.
  + int send\_with\_parameters(char\*\* parameters): permette di settare l'emergenza da inviare tramite i parametri da terminale.
  + int send\_with\_file(char\* file\_emergency): permette di inviare le emergenze da un file.{conf,text}.
  + ## Istruzioni per l'invio delle emergenze

- Sintassi inviare una singola emergenza da terminale: ./<nome file client> <nome emergenza> <x> <y> <delay> (Esempio: ./client Incendio 140 130 4).

- Sintassi per inviare più emergenze contenute in un file: ./<nome file client> -f <nome file>.{txt o conf} (Esempio: ./client -f f.conf).

- Se si vuole inviare le emergenze facendo rimanere in esecuzione il client, basta avviare il programma digitando solamente ./<nome file client> seguendo le istruzioni del menù.

- Se si vuole interrompere la message queue da terminare, digitare ./<nome file client> STOP (./client STOP).