Manuale Gestione Emergenze

Lucchesi Paolo
690948
Laboratorio II A
p.lucchesi7@studenti.unipi.it

Introduzione

Il manuale è diviso in 3 sezioni: Architettura, Scelte Progettuali e Compilazione/Esecuzione.

La parte sull'architettura fornisce una breve introduzione sulla suddivisione dei compiti fra le varie funzioni per l'esecuzione del programma.

La parte più descrittiva riguarda le scelte progettuali, dove analizzeremo in dettaglio tutti i componenti del programma partendo dal main e successivamente le varie funzioni di parsing, gestione lista_emergenze, gestione della memoria, thread per l'inserimento e thread per lo svolgimento emergenze.

La sezione Makefile offre un breve excursus sul funzionamento dello stesso e su come viene gestito il logFile.txt, generato ad ogni esecuzione del programma.

Suddivisione progetto in folder:

- Progetto:
  - MakeFile
  - main.c
  - client.c
  - Conf:
    - env.conf
    - rescuer.conf
    - emergency_types.conf
  - Parser:
    - parser_emergency_types.c
    - parser_env.c
    - parser_rescuers.c
  - Utils:
    - Funzioni.c
    - Lista.c
    - Macro.h
    - Strutture.h
  - thrd_functions:
    - thrd_insert.c
    - thrd_operatori.c
    - rescuer_on_scene.c
    - rescuers_return.c

1. Architettura del Sistema

1.1. Parser

La sezione parser, dedita all'interpretazione dei file .conf e alla creazione di strutture dati, è così formata:

- parser_rescuer.c crea un array di tipo rescuer_type_t che raccoglie tutti i vari attributi relativi ai tipi di soccorritori;
- parser_emergency_types.c crea un array di tipo emergency_type_t con le informazioni relative ai tipi di emergenze;
- parser_env.c crea una struttura con gli attributi queue_name, width e height riguardanti l'ambiente.

1.2. Main

Il main funge da snodo centrale per gestire tutte le varie componenti del programma:

- Chiama le funzioni parser per ricavare le informazioni su emergenze e soccorritori;
- Inizializza i thread per la gestione delle emergenze;
- Inizializza i thread per l'inserimento di nuove emeregenze (ricevute dal client) nella lista per la gestione;
- Si occupa di terminare il programma non appena riceve dal client il messaggio di **STOP**;
- Prima di terminare il programma, si occupa di liberare la memoria da strutture allocate dinamicamente, chiudere memorie condivise, chiudere message_queue e terminare i vari thread ancora attivi.

1.3. Threads

Ci sono diversi tipi di threads nel programma:

- I thread di gestione emergenze vengono attivati all'inizio del programma e rimangono in attesa di nuove emergenze. Viene attesa la loro terminazione con una Join.
- I thrd per l'inserimento di nuove emergenze vengono inizializzati appena il client invia un messaggio (che non sia di terminazione) e subiscono immediatamente una detach.
- I thread soccorritori si occupano del soccorritore sulla scena dell'emergenza simulando un'attesa e, successivamente, inizializzano un nuovo thread per il ritorno del soccorritore alla base a cui fanno immediatamente una detach. Il motivo di questa scelta verrà specificato nella sezione delle scelte progettuali.
- I thread ritorno si occupano del ritorno alla base del soccorritore, simulando un'attesa e ripristinando le risorse, svegliando i thread in attesa con una broadcast.

1.4. Client

Il Client è un programma autonomo che non utilizza nessuna delle componenti sopra descritte. Una volta avviato attraverso linea di comando con i relativi argomenti, vi sono 4 esiti possibili:

- Se viene passata un'emergenza («**./client emergenza x y delay**»), il client invia quest'ultima al main, dove, in caso di validazione con successo, verrà inserita nella coda emergenze per poi essere processata;
- Se viene passato un file di testo («**./client -f file**»), il client invierà le emergenze descritte all'interno del file, riga per riga, al main per essere validate e processate.
- E' possibile terminare il programma attraverso il comando «**./client exit**»;
- In caso di qualsiasi altro input (numero di argomenti diversi o parole chiave non riconosciute), il client non riconoscerà il comando e terminerà la sua esecuzione senza intaccare il main.

1.5. SysCall & C std Library

Vengono utilizzate chiamate di sistema per gestire memoria condivisa, message_queue fra client e main.
La gestione di file .conf per i parser è stata fatta mediante funzioni di libreria C std.

2. Scelte Progettuali

Ogni accesso per acquisizioni/modifiche di risorse (**digital_twins** o contatori **soccorritori_liberi**) verrà regolato tramite **rescuer_mtx** e **rescuer_cnd** per l'attesa.

Ogni accesso alla **lista_emergenze** verrà regolato tramite **lista_mtx** e **lista_cnd** per l'attesa.

Ogni accesso di scrittura al **logFile.txt** verrà regolato tramite **log_mtx**

2.1. Macro.h

Header file contenente tutte le macro utilizzate per gestire diverse chiamate tra cui le SysCall.
ATTENZIONE: questo file contiene le macro che settano il numero totale di TIPI soccorritori (**RESCUER_TYPES**) e TIPI emergenze (**EMERGENCY_TYPES**). In caso di aggiunta o rimozione di uno di questi campi, bisognerà modificare opportunamente queste Macro per garantire il corretto funzionamento del programma.
Oltre a questo, il file contiene diversi paramatri, fra cui una macro che definisce il numero di thread che vengono inizializzati per gestire le emergenze (**THRD_OPERATIVI**), settabile a piacimento.

2.2. Strutture.h

Header file contenente le strutture descritte nelle specifiche da seguire per la realizzazione del proggetto e altre strutture utili per la gestione dati.

2.3. main.c

Si definiscono variabili globali nel file che vengono usate da tutte le funzioni relative ai thread, esportate in altri file, oltre a main.c, con la dicitura **extern**:

- **lista_emergenze**: lista doppia contenente emergenze;
- **soccorritori_liberi**: array che tiene conto dei soccorritori liberi;
- **mtx_t** e **cnd_t** per accesso a risorse, emergenze e scrittura del **logFile.txt**;
- Variabili Atomiche:
  - **keep_running** per mandare avanti il programma o terminarlo settandola a 0;
  - **id_emrg** per dare un identificativo numerico alle varie emergenze;
  - **thrd_attivi** per tenere sotto traccia i thread ancora in funzione a cui sono stati eseguiti thrd_detach;
  - **emrg_gestite** che tiene di conto dei thrd_operatori che hanno preso in carico un'emergenza.

Il **main** inizia chiamando le funzioni relative ai **parser** per generare le strutture dati contenenti le informazioni specificate nei file .conf

Successivamente, apre una memoria condivisa con un certo nome (salvato come macro nel file Macro.h) per condividere con il client il nome della messae_queue.

Dopo di che si inizializza la message_queue con i suoi attributi per la comunicazione delle emergenze.

Come ultime operazioni preliminari, si inizializzano le mutex, le condition variables, la lista delle emergenze e i thread (**thrd_operatori()**) che si occuperanno dello svolgimento delle emergenze.

Conclusa questa parte, si entra in un ciclo while(1) dove il programma riceve dal client stringhe contenenti richieste di emergenze e, per ciascuna di esse, avvia un thread (**thrd_insert()**) per validarle ed inserirle nella lista_emergenze.
Il programma termina se, dalla message_queue, viene letto il messaggio di stop, definito da una macro in Macro.h (di default è «exit»).

Ricevuto il messaggio di terminazione, si setta keep_running = 0 e si svegliano i thread in attesa sulla lista_emergenze; si aspetta la terminazione di tutti i thread (sia dei thrd_operative con una join, sia dei restanti a cui è stata fatta la detach) e, infine, si chiude la message_queue, la shared memory, i descrittori di file e si deallocano tutte le strutture dati.

2.4. client.c

Il programma deve essere avviato successivamente al main (eseguibile principale), altrimenti darà un errore e terminerà.
Il client inizia accedendo alla memoria condivisa per poter ottenere il nome della message_queue. Successivamente, accede alla message_queue e inizia a lavorare con gli argomenti passati da riga di comando.
Grazie ad uno switch, si sceglie come operare in base al numero di argomenti presenti:

- argc = 2: si controlla che sia stato inserito il messaggio di stop e si invia al main;
- argc = 3: si utilizza una funzione «**da_file()**» che verifica la validità degli argomenti e, successivamente, inizia a leggere le emergenze riga per riga inviandole al main, dove verranno validate e processate;
- argc = 5: si crea una stringa concatenando i vari argomenti e si invia direttamente al main l'emergenza. Attenzione: il nome dell'emergenza deve contare come singolo argomento quindi, se è composta da più parole, vanno collegate da un trattino basso (Disastro_Ambientale).

Qualsiasi errore nei dati passati verra controllato successivamente dalle funzioni per la validazione.

2.5. parser_rescuers.c

Funzione che genera un array di tipo **rescuer_type_t** dove vengono salvate le informazioni relative ai soccorritori. L'ordine in cui vengono salvate è lo stesso ordine in cui vengono lette dal file .conf.
La funzione ritorna un puntatore alla struttura dati allocata dinamicamente.

Una volta creata la struttura (array) per salvare le informazioni, e terminato le operazioni preliminari di apertura file .conf e .log, si utilizza un ciclo while() per leggere, riga per riga, il file ed eseguire il parsing dei dati. Queste operazioni di parsing (e di scrittura su file log) vengono eseguite mediante funzioni di libreria standard C (**fgets()**, **sscanf()** etc.) e vengono protette da MACRO su misura che sfruttano i valori di ritorno delle funzioni per controllare la presenza di errori, lo stesso ciclo while(ret) utilizza il valore di ritorno di **fgets()** per terminare la lettura del file (ritorna **NULL** quando terminato). In queste operazioni preliminari, se vengono rilevati errori nei file .conf o durante l'esecuzione di queste operazioni di parsing, il programma termina riportando un messaggio di errore specifico.

Successivamente, il main chiamerà la funzione per la creazione di gemelli digitali, definita sempre in questo file. Questa funzione ritorna una matrice organizzata in questo modo:
i gemelli digitali appartenenti ad una tipologia di soccorritore, vengono salvati in un riga comune. Queste righe vengono organizzate seguendo lo stesso ordine dell'array delle tipologie di soccorritori (se la tipologia pompiere viene salvata nella cella numer 3 => i gemelli digitali pompieri verranno salvati nella riga numero 3). Lo stesso ordine viene utilizzato per la variabile globale **soccorritori_liberi** che tiene conto del numero di soccorritori disponibili.
Questo scorrimento ordinato della matrice (gemelli digitali) e dell'array (soccorritori_liberi) permette un accesso ordinato (non circolare) alle risorse e quindi permette di evitare deadlock.

2.6. parser_emergency_types.c

Funzione che genera un array di tipo **emergency_type_t** dove vengono salvate le informazioni realtive ai soccorritori.
La funzione ritorna un puntatore alla struttura dati allocata dinamicamente.

Una volta creato l'array per salvare le informazioni e terminate le operazioni preliminari per l'apertura del file .conf e .log, esattamente come il parser_rescuer, si utilizza un ciclo while(ret) con funzioni di libreria standard C per la lettura da file, parsing informazioni e scrittura su file log (ciascuna chiamata verrà protetta da una macro).
La complicazione aggiuntiva nel parsing delle emergenze, deriva dalla mancanza di vincoli sul numero di tipi soccorritori specificabili nel file .conf per ciascuna emergenza. Quindi, durante il parsing della linea che descrive il tipo di emergenza, bisogna separare in una stringa a parte la sezione con i tipi soccorritori, numero e tempo richiesto per svolgimento emergenza. Questa stringa dovrà essere tokenizzata in un ciclo while. Si salveranno così i tipi soccorritori richiesti in un array interno alla struttura emergency_type_t NELLO STESSO ORDINE dell'array contenente i tipi soccorritori.

2.7. parser_env.c

Funzione che genera una struttura di tipo environment_t (definita in Strutture.h) con campi queue_name, height e width.

Dopo aver creato la struttura per le informazioni e aver aperto i file .conf (da analizzare) e log, si fanno 3 operazioni di lettura + parsing senza bisogno di cicli.
Si utilizza anche in questo caso funzione libreria C standard per gestione file.
Al nome della coda estratto si deve aggiungere come prefisso «/».
Terminate le tre operazioni di Lettura + Parsing, si documenta sul file log i dati ottenuti, si chiudono i descrittori file e si ritorna il puntatore alla scrittura dati

2.8. thrd_insert.c

La funzione ha il compito di validare una stringa rappresentante una richiesta di emergenza e, in caso di successo, inserirla nella lista_emergenze.
Per fare ciò, la funzione prende un puntatore (a cui fare il casting) a una struttura dati **thrd_data_insert** (Strutture.h) allocata dinamicamente (che dovrà deallocare prima di terminare) che contiene dati forniti dal main: la stringa con l'emergenza, descrittore log file, puntatore ad array con tipi emergenza, puntatore a struttura dati ambiente.

Il thread sfrutta 3 funzioni per svolgere il suo compito:

1. **analisi_richiesta()**: prende la stringa con l'emergenza come argomento, alloca dinamicamente spazio per la struttura **emergency_request_t**. Il timestamp dell'emergenza diventa il tempo corrente dell'inserimento, più il delay specificato durante l'invio dell'emergenza da parte dell'utente. In caso di errori nei dati forniti, la funzione ritorna un puntatore a **NULL** e il thrd, dopo aver deallocato la struttura argomenti, termina.
2. **validazione_richiesta()**: passato il puntatore a struttura emergency_request_t, controlla che i dati inseriti siano validi e ritorna un puntatore ad una struttura allocata dinamicamente di tipo **emergency_t**. In caso di errori nei dati forniti, la funzione ritorna un puntatore a **NULL** e il thrd, dopo aver deallocato la struttura emergency_request_t e argomenti, termina.
3. **add_emrg() + lista_mtx**: grazie ad una mutex, per un accesso ordinato che evita sovrapposizioni, si può inserire la nuova emergenza nella lista_emergenze a seconda della sua priorità e del tempo rimanente prima del TIMEOUT. In caso il delay fosse maggiore di zero, si esegue una **thrd_sleep()** per attendere il tempo specificato prima dell'inserimento.

Terminati questi passi, si utilizza una **broadcast** su **lista_cnd** per svegliare i thread in attesa di una nuova emergenza da prendere in carico, si dealloca lo spazio degli argomenti, della struttura emergency_request_t, NON si dealloca la struttura emergency_t (che verrà deallocata successivamente da altre funzioni), e si termina il thread.

2.9. thrd_operatori.c

La funzione ha il compito di prendere in carico un'emergenza, gestirla a seconda delle condizioni, deallocare lo spazio delle strutture relative l'emergenza una volta esaurito il loro utilizzo e, una volta terminato, prendere in carico un'altra emergenza e continuare fino alla terminazione del programma.
Per fare ciò, la funzione prendere un puntatore (a cui fare il casting) a una struttura dati **thrd_data_operative** (Strutture.h) allocata dinamicamente (che dovrà deallocare prima di terminare) che contiene dati forniti dal main: l'id del thread (per il debug), descrittore file log, puntatore ad array tipi soccorritori, puntatore ad array tipi emergenze, puntatore a matrice gemelli digitali.

La funzione entra in un while() regolato dalla variabile atomica keep_running e, attraverso la funzione **estrai_nodo()** (che ritorna un puntatore a struttura allocata dinamicamente) e la mutex **lista_mtx**, estrae un'emergenza da prendere in carico. Se la lista_emergenze è vuota, il thread si metti in wait su **lista_cnd** finché non viene inserita una nuova emergenza.

Vengono copiati i dati dell'emergenza estratta su una struttura locale e viene deallocata la struttura sullo heap.

Si controlla che i soccorsi possano arrivare in tempo confrontando il tempo rimanente dell'emergenza (**tempo_rimanente()**) e il tempo di attesa richiesto per l'arrivo dei soccorsi (**tempo_arrivo_soccorsi()**). Se il tempo rimanente non basta ad attendere l'arrivo dei

soccorsi, il thread documenta sul file log l'impossibilità di intervenire e passa all'emergenza successiva.
Se il tempo è «virtualmente» sufficiente, il thread prende in carico l'emergenza e inizializza un ciclo FOR che scorre tutti i tipi di soccorritori nell'ordine specificato precedentemente (ordine stabilito precisamente per evitare deadlock in questa porzione di codice) per ottenere le risorse necessarie alla risoluzione dell'emergenza. L'iterazione riguardante una certa risorsa viene saltata se l'emergenza corrente non richiede quello specifico tipo di soccorritore.
In una di queste iterazioni, riguardante uno specifico tipo di soccorritore (richiesto dalla nostra emergenza), si alloca dinamicamente la riga nella matrice di gemelli digitali della nostra emergenza (organizzando le righe di soccorritori nello stesso ordine sopra citato). Ammesso che i soccorritori siano disponibili, con l'ausilio di una mutex (**rescuer_mtx**), si salvano i riferimenti dei gemelli digitali IDLE nella nostra matrice, si cambia loro lo status e si sottrae il numero di soccorritori richiesto dall'array di contatory **soccorritori_liberi**. Se, invece, i soccorritori non sono disponibili, si esegue un'attesa sulla condition variable **rescuer_cnd**.

A questo punto si esegue un ultimo controllo sul tempo rimanente, per vedere se l'attesa per il reperimento delle risorse ha inciso sul tempo rimanente dell'emergenza. In caso di successo il thread inizia lo svolgimento dell'emergenza. Altrimenti, con un ciclo simile a quello per l'acquisizione delle risorse, si ripristina lo stato IDLE dei gemelli digitali, si deallocano le righe della matrice con i gemelli e si ripristina il contatore del dato tipo in **soccorritori_liberi** (tutto con l'ausiglio di una mutex). Infine si esegue una broadcast su **rescuer_cnd**.

L'attesa delle risorse viene simulata attraverrso una semplice **thrd_sleep()** prendendo il tempo di attesa maggiore fra i tipi di soccorritore richiesti.
Si esegue, tramite mutex, una modifica allo stato e coordinate dei gemelli digitali e si inizia a svolgere l'emergenza.

L'emergenza viene svolta grazie all'avvio di diversi thread (ciascuno per ogni gemello digitale) con una struttura argomenti passata, che simulerà l'attesa per lo svolgimento.
Terminati questi con una **thrd_join()**, setto lo stato dell'emergenza come COMPLETED e dealloco le righe della matrice di gemelli digitali. Dopo aver documentato la fine della gestione emergenza, il thread ricomincia un altro ciclo prendendo in carico un'altra emergenza (sempre che **keep_running** non sia settato a 0).

2.10. thrd_on_scene.c

Questa funzione per thread ha il compito di simulare l'attesa per lo svolgimento di un emergenza da parte di un dato soccorritore.

Per fare questo, l'emergenza sfrutturà un puntatore (a cui fare il casting) ad una struttura dati **thrd_data_socc** che contiene: un riferimento al gemello digitale relativo al soccorritore, un puntatore all'array contenente le tipologie di soccorritore, il descrittore del file log e il tempo richiesto per lo svolgimento dell'emergenza.

Il thread esegue l'attesa per lo svolgimento emergenza e, successivamente, attraverso l'uso di una mutex (**rescuer_mtx**) modifica lo stato dei soccorritori. Prima di terminare il suo svolgimento, si inizializza un nuovo thread (**rescuers_return()**) a cui vengono passati gli stessi argomenti del thread corrente e a cui viene fatta una **detach**, solo dopo il thread termina. Questo perché il thrd_operatori() attende che tutti i thread **rescuer_on_scene** terminino (ovvero lo svolgimento dell'emergenza) per concludere l'emergenza. Non bisogna attendere anche il ritorno dei soccorritori alla base. In questo modo, i thread possono prendere in carico nuove emergenze e ridurre il rischio di TIMEOUT.

2.11. rescuers_return.c

Questa funzione ha il compito di simulare l'attesa per il ritorno alla base di un soccorritore, cambiare il suo status e coordinate, ripristinare il contatore dei soccorritori_liberi e svegliare i thread in attesa di risorse.

Per fare questo, l'emergenza sfrutturà un puntatore (a cui fare il casting) ad una struttura dati **thrd_data_socc**, utilizzata prima da **thrd_data_socc** che contiene: un riferimento al gemello digitale relativo al soccorritore, un puntatore all'array contenente le tipologie di soccorritore, il descrittore del file log e il tempo richiesto per lo svolgimento dell'emergenza.

La funzione calcola il tempo in funzione della distanza Manhattan e della velocità del proprio soccorritore, dopo di che esegue un attesa.

Attraverso l'ausilio di una mutex (**rescuer_mtx**), si cambiano lo stato e le coordinate del soccorritore, si ripristina il contatore **soccorritori_liberi**, si documentano gli eventi sul logFile.txt e si esegue una broadcast su **rescuers_cnd**.

Prima di terminare il thread, si esegue una funzione **controllo_situazione()** che, in caso di emergenze terminate e che tutti i soccorritori siano tornati alla base, stampa un messaggio a schermo che indica l'esaurimento di task da eseguire.

A questo punto, il thread dealloca lo spazio per gli argomenti e termina.

2.12. Lista.c

Questo file contiene una serie di funzioni utilizzate per gestire la **lista_emergenze**. in questa sezione, vi si fornisce una breve descrizione:

- **lista_init()**: genera la struttura iniziale **lista_t** e ritorna un puntatore ad essa.
- **destroy_lista()**: funzione che dealloca lo spazio per la struttura lista_t e, in caso di terminazione anticipata del prograrmma, dealloca anche nodi interni contenenti emergenze.
- **add_emrg()**: crea un nuovo nodo della lista contenente un emergenza. La funzione, per inserire il nuovo nodo, utilizza una funzione **confronto()** che permette di trovare la posizione adeguata per la nuova emergenza. Infine, incrementa il contatore delle emergenze in lista di 1.
- **rimuovi_timeout()**: funzione invocata da un thread prima dell'estrazione di una nuova emergenza dalla lista. La funzione scorre tutta la lista partendo dalla coda (emergenza a priorità più alta) e rimuove tutti i nodi contenenti emergenze con TIMEOUT (tempo scaduto), documentando tutte le rimozioni sul logFile.txt.
- **estrai_nodo()**: funzione che estrae un nodo dalla coda della lista (priorità maggiore), dealloca la struttura nodo (non dealloca l'emergenza) e ritorna un puntatore alla struttura **emergency_t** estratta.

2.13. Funzioni.c

Questo file contiene un insieme di funzioni utilizzate in diverse parti del programma. Si cercherà di fornire una breve descrizione di ciascuna di esse:

- **analisi_richiesta()**: funzione che prende come argomento una stringa di testo (inviata dal client) ed esegue il parsing per estrarre informazioni relative all'emergenza. Se vi è qualche errore di formato, si documenta grazie alla funzione **emergenza_errata()** e si restituisce un puntatore a **NULL**. In caso di successo, si alloca spazio per una struttura di tipo **emergency_request_t** con i relativi dati e si ritorna un puntatore a questa struttura.
- **validazione_richiesta()**:funzione che prende come argomento un puntatore a struttura **emergency_request_t** e controlla che i dati inseriti al suo interno siano corretti. In caso di

qualche errore, si documenta questo grazie alla funzione **emergenza_errata()** e si ritorna un puntatore a **NULL**. In caso di successo, si alloca spazio per una struttura **emergency_t** con i relativi dati e si ritorna un puntatore a qeusta struttura

- **emergenza_errata()**: Questa funzione, prende come argomenti il descrittore al logFile.txt e una stringa da scrivere sul file. Serve a documentare un'emergenza scartata per un qualunque tipo di errore (formattazione, dati etc.).
- **confronto()**: prende come argomenti 2 puntatori a struttura **emergency_t** e restituisce un numero a seconda della loro priorià e del tempo rimanente: l'emergenza con priorità più alta viene spostata verso la coda; in caso di parità viene considerato il tempo rimanente più basso come priorità maggiore.
- **tempo_rimanente()**: funzione che prende come argomento un puntatore a struttura **emergency_t** e, a seconda della priorità e del suo timestamp, restituisce il tempo rimanente prima del TIMEOUT. Per le emergenze di priorità 0, viene sempre restituito 1000 dato che non hanno scadenza.
- **tempo_corrente()**: preso come argomento un puntatore a char, scrive in esso data e tempo corrente utilizzando **time(NULL)** e **ctime()**.
- **tempo_arrivo_soccorsi()**: funzione che prende come argomento un puntatore a **emergency_t** e calcola il tempo di attesa massimo per l'arrivo dei soccorsi. Inizializza una variabile di attesa a 0 e scorre i vari tipi di soccorritori richiesti dall'emergenza tramite un ciclo FOR. Ad ogni iterazione, calcola il tempo impiegato da quel soccorritore in funzione della distanza Manhattan e della sua velocità e lo confronta con il tempo corrente, prendendo il massimo.
- **controllo_situazione()**: funzione chiamata ogni volta che dei soccorritori tornano alla base (**rescuer_return**) o che delle risorse vengono rilasciate o che delle emergenze vengano terminate per TIMEOUT. Analizza la situazione globale del programma, controllando che non ci siano più emergenze nella lista_emergenze, che tutti i soccorritori siano tornati alla base e che non ci sia più alcun thread con un'emergenza presa in carico. Se tutte queste condizioni sono rispettate, la funzione scrive un messaggio a schermo, indicando che il programma ha terminato tutte le sue task da eseguire.

2.14. TERMINAZIONE ANTICIPITA

Il programma termina con l'inserimento del messaggio di STOP tramite client.
Se viene terminato il programma dopo la fine dello svolgimento delle emergenze, viene chiuso immediatamente. Se invece si termina il programma durante lo svolgimento delle emergenze, questo setterà la variabile **keep_running** a 0, sveglierà i thread in attesa con una broadcast su **lista_cnd**, impedirà l'inserimento di nuove emergenze nella lista e aspetterà che i thread terminino la loro esecuzione. Si attende la terminazione dei thread a cui è stato fatto detach tramite l'utilizzo della variabili atomica **thrd_attivi**, questo perché si incrementa di 1 ogni volta che si fa detach ad un thread e si decrementa di uno ogni volta che uno di questi thread termina.
Questo non richiederà troppo tempo perché i thread, prima di iniziare una qualsiasi attesa, controllano sempre la variabile **keep_running**, quindi potremmo evitare diverse attese per la simulazione di eventi. Inoltre, prima di iniziare lo svolgimento di un'emergenza, i thread controllano lo stato del programma e, in caso di terminazione, rilasciano tutte le risorse e terminano.

3. Compilare ed Eseguire

Per la compilazione, usare il comando **make**.
Per lanciare il programma, usare il comando **make run**.

3.1. MakeFile

CC = gcc
Compilatore.

CFLAGS = -O3 -Wall -pedantic -std=c11
flags di compilazione e ottimizzazione.

LIBS = -lpthread
Libreria threads.

OBJS = $(NAME).o parser/parser_rescuers.o etc...
Viene elencato ogni file oggetto ottenuto dal corrispettivo file .c

OBJCLIENT = client.o
file oggetto del client

LOGFILE = logFile.txt
file di testo con la cronologia dell'esecuzione del programma

.PHONY: default clear run
Phony targets

default: $(NAME) client
Obbiettivi da generare di default

%.o: %.c %.h
    $(CC) -c $(CFLAGS) $< -o $@
comando per generare file oggetto aventi come dipendenze file sorgente .c e header file .h

$(NAME): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^ $(LIBS)
Comando per generare l'eseguibile main

client: $(OBJCLIENT)
    $(CC) $(CFLAGS) -o $@ $^ $(LIBS)
Comando per generare l'eseguibile client

run: $(NAME)
    @if [-f $(LOGFILE)]; then
    rm $(LOGFILE);
    fi
    ./$(NAME)
Prima di far partire il programma, si eseguono dei comandi di shell per eliminare eventuali logFile.txt presenti, dato che il programma gli apre in modalità append.
Si utilizza «@» prima del comando di shell in modo da non farlo stampare a schermo prima dell'esecuzione del programma.

clear:
    rm -f $(NAME) $(OBJS) $(OBJCLIENT) client logFile.txt
Comando di rimozione eseguibili, file oggetto e logFile.txt