Progetto di Laboratorio 2

Sistema di Gestione delle Emergenze con Multithreading e Message Queue in C11

a.a. 2024-25

Obiettivo

Progettare e implementare un sistema multithread nella versione C11, che sia in grado di gestire situazioni d'emergenza all'interno di una griglia bidimensionale, utilizzando (anche in modo non esclusivo) le chiamate di sistema POSIX. Le emergenze devono essere raccolte tramite una coda di messaggi POSIX e richiedono una distribuzione delle risorse in modo concorrente, quali forze dell'ordine, pompieri, ecc., tenendo in considerazione le priorità e rispettando rigorosi vincoli di tempo. Il sistema deve essere progettato per evitare situazioni di stallo (deadlock) e, nel caso di coloro che non abbiano superato le prove in itinere, deve gestire anche la prevenzione della starvation.

1 Descrizione delle entità del sistema

Il contesto di riferimento è costituito da una griglia rettangolare bidimensionale, in cui ciascuna cella è definita tramite coordinate (x, y). Ogni unità di soccorso dispone di un sistema di localizzazione ed è rappresentata attraverso un *gemello digitale*[1] che ne riflette il tipo (ad esempio, Pompieri), lo stato (disponibile o in servizio) e la posizione all'interno della griglia. Ogni emergenza rimossa dalla coda è caratterizzata da un tipo, una posizione definita sulla griglia, e l'istante in cui è stata registrata. L'obiettivo del sistema è di assegnare i soccorritori appropriati a ciascuna emergenza, ottimizzando la probabilità che le emergenze siano gestite rapidamente.

I tipi di emergenze e i soccorritori disponibili sono delineati tramite file di configurazione specifici: `rescuers.conf` e `emergency_types.conf`. La dimensione dell'ambiente è definita in un terzo file di configurazione, `env.conf`. Si può assumere che il contenuto di questi file non cambi e richieda di essere letto solo durante l'avvio del programma. I file devono essere prodotti dagli studenti seguendo le indicazioni sintattiche specificate in seguito. I file devono essere letti tramite parser dedicati: `parse_rescuers.c`, `parse_emergency_types.c` e `parse_env.c`. Anche i parser devono essere realizzati dagli studenti.

1.1 Tipi e istanze di Soccorritori

La sintassi usata per descrivere i tipi e le istanze di soccorritori disponibili, il loro numero e la loro velocità, sempre in formato BNF, è la seguente:

```
<config> ::= <entry> | <entry> "\n" <config>
<entry> ::= "[" <name> "]" "[" <num> "]" "[" <speed> "]" "[" <base> "]"
<base>   ::= <x_coord> ";" <y_coord>
```

Ogni tipo di soccorritore ha un nome identificativo (ad esempio, Pompieri), una cardinalità rappresentata da un numero intero che indica quanti mezzi può disporre, una velocità espressa sempre in numero intero, ovvero in "caselle al secondo", e un set di coordinate che specificano la

[1] `https://it.wikipedia.org/wiki/Gemello_digitale`

posizione della loro base (dove i mezzi possono essere trovati all'inizio del programma o dopo aver gestito l'intervento per un'emergenza). Ecco un esempio di file di configurazione appropriato per gestire tipi di emergenze:

```
[Pompieri][5][2][100;200]
[Ambulanza][25][4][150;250]
```

Dopo aver elaborato il file, è necessario generare le strutture dati che corrispondono ai tipi e alle istanze dei soccorritori disponibili, sfruttando i tipi di dati elencati in seguito. Per ciascun tipo di soccorritore, si deve creare una struttura del tipo `rescuer_type_t` ed istanziare un numero di gemelli digitali pari al totale dei mezzi a loro disposizione. Ogni digital twin ha uno stato specifico: `IDLE` (libero), `EN_ROUTE_TO_SCENE` (si sta dirigendo sulla scena dell'emergenza), `ON_SCENE` (si trova sulla scena dell'emergenza), `RETURNING_TO_BASE` (sta tornando alla base). Inizialmente assegnato a `IDLE`, cambia ogni volta che l'istanza del soccorritore cambia di stato.

```c
typedef enum {
    IDLE, EN_ROUTE_TO_SCENE, ON_SCENE, RETURNING_TO_BASE
} rescuer_status_t;

typedef struct {
    char * rescuer_type_name;
    int    speed;
    int    x;
    int    y;
} rescuer_type_t;

typedef struct {
    int                id;
    int                x;
    int                y;
    rescuer_type_t *   rescuer;
    rescuer_status_t   status;
} rescuer_digital_twin_t;
```

1.2 Tipi di Emergenze

La sintassi usata per descrivere i tipi di emergenza, in formato BNF, è la seguente:

```
<config> ::= <entry> | <entry> "\n" <config>
<entry> ::= "[" <name> "]" "[" <priority> "]" <rescuers>
<rescuers> ::= <rescuer> | <rescuer> <rescuers>
<rescuer> ::= <rescuertype> ":" <number>,<time_in_secs>;
```

Ogni tipo di emergenza è identificato da un nome univoco (come, ad esempio, Allagamento), e ha una priorità che può assumere 3 diversi valori: 0 (bassa), 1 (media) e 2 (alta). Inoltre, include un elenco dei tipi di soccorritori necessari, specificando il numero di istanze richieste per ciascuno di essi e il tempo che l'unità dedicherà alla gestione dell'emergenza. Un file di configurazione rappresentativo per i tipi di emergenza potrebbe essere strutturato nel seguente modo:

```
[Allagamento] [1] Pompieri:1,5;Ambulanza:1,2;
[Incendio] [2] Pompieri:2,8;Ambulanza:3,1;
[Sommossa] [2] Polizia:3,8;Ambulanza:2,4;
```

Una volta processato il file, è fondamentale costruire le strutture dati che rappresentano i tipi di emergenze consentiti. La priorità è indicata dal tipo `short`, mentre per il nome si utilizza

un puntatore a carattere. È necessario anche un array di entità `rescuer_request_t`, ognuna delle quali comprende un puntatore al tipo di soccorritore `rescuer_type_t`, la loro quantità, espressa dal campo intero `required_count`, e il tempo stimato (in secondi) per gestire quella emergenza.

```
typedef struct {
    short                   priority;
    char *                  emergency_desc;
    rescuer_request_t *     rescuers;
    int                     rescuers_req_number;
} emergency_type_t;

typedef struct {
    rescuer_type_t *        type;
    int                     required_count;
    int                     time_to_manage;
} rescuer_request_t;
```

1.3 Ambiente

La sintassi del file di configurazione dell'ambiente risulta più semplice e si limita ad indicare il nome della coda e le dimensioni della griglia considerata nel seguente modo:

```
queue=emergenze123456
height=300
width=400
```

In questo caso si tratterebbe, ad esempio, di una griglia larga 400 e alta 300 e di una coda denominata emergenze123456.

1.4 Istanze di Emergenza

Ognuna delle richieste di emergenza viene prelevata dalla coda dei messaggi e elaborata. Ciascuna contiene il nome che indica il tipo di emergenza (ad esempio, Incendio), le coordinate del luogo dell'evento e l'ora in cui è accaduto. Dopo l'estrazione, è necessario verificare la validità di nome, coordinate e timestamp. Una richiesta errata deve essere scartata e l'evento registrato nel file di log (si vedano in seguito i dettagli). In caso di validità, viene creata una nuova istanza di emergenza del tipo `emergency_t`.

```
#define EMERGENCY_NAME_LENGTH 64

typedef enum {
WAITING, ASSIGNED, IN_PROGRESS, PAUSED, COMPLETED, CANCELED, TIMEOUT
} emergency_status_t;

typedef struct {
    char  emergency_name[EMERGENCY_NAME_LENGTH];
    int   x;
    int   y;
    time_t timestamp;
} emergency_request_t;

typedef struct {
    emergency_type_t          type;
    emergency_status_t        status;
    int                       x;
```

```
    int                         y;
    time_t                      time;
    int                         rescuer_count;
    rescuer_digital_twin_t*     rescuers_dt;
} emergency_t;
```

2 Gestione delle Emergenze

Le emergenze devono essere gestite in modo concorrente, garantendo correttezza e sicurezza mediante adeguati meccanismi di sincronizzazione. Non è accettabile un progetto in cui le emergenze vengano gestite in modo sequenziale da un unico thread. La priorità è definita dal tempo massimo entro cui un'emergenza deve essere affrontata: priorità 0 indica nessun limite, priorità 1 equivale a 30 secondi e priorità 2 a 10 secondi. Se il gestore rileva l'impossibilità di affrontare un'emergenza tempestivamente (ovvero avere tutti i mezzi necessari collocati sul luogo dell'emergenza, che a quel punto passa nello stato IN_PROGRESS) a causa di carenze di risorse o distanza, ciò deve essere immediatamente segnalato nel file di log e lo stato dell'emergenza deve essere impostato su TIMEOUT. Coloro i quali non hanno superato i compitini devono ideare una soluzione per garantire che le emergenze con priorità 0 vengano gestite. Ogni modifica dello stato di un'emergenza o di un mezzo di soccorso deve essere registrata nel file di log (dettagli specificati in seguito).

2.1 Distanza di Manhattan

Per stimare il tempo di arrivo di un soccorritore:

$$d\big((x_1, y_1), (x_2, y_2)\big) = |x_1 - x_2| + |y_1 - y_2|.$$

Dividendo d per la velocità del soccorritore si ottiene un'approssimazione del tempo di percorrenza.

3 Requisiti Tecnici

3.1 Compatibilità

È fondamentale che il codice sviluppato operi senza problemi su Linux Ubuntu 24.04. In modo specifico, il sistema deve essere testato sulla macchina laboratorio2, la quale utilizza Ubuntu 24.04.1 LTS, prima della sua consegna.

3.2 Indentazione e stile

È importante che il codice venga adeguatamente indentato e commentato, seguendo uno stile di programmazione che ne ottimizzi la leggibilità.

3.3 Protezione chiamate di sistema

Come spiegato in diverse occasioni durante le lezioni, ogni volta che si utilizza una chiamata di sistema, è fondamentale che essa venga adeguatamente "protetta" tramite l'uso di macro. Qualora l'uso delle macro non fosse praticabile, è comunque cruciale eseguire i necessari controlli di errore.

3.4 Concorrenza

Utilizzare le primitive di sincronizzazione viste nel corso (mutex, condition variables, semafori, ecc.), motivando le scelte implementative.

3.5 Message Queue

Le emergenze devono essere aggiunte alla coda mediante un'applicazione separata (client). Il client invia le richieste alla coda POSIX denominata `"emergenze"` seguita dal numero di matricola dello studente, ad esempio, emergenze123456. Ogni messaggio include il tipo, le coordinate e il timestamp dell'emergenza. Il client deve consentire la generazione di una richiesta di emergenza utilizzando la seguente sintassi:

```
./client <nome_emergenza> <coord_x> <coord_y> <delay_in_secs>
```

Il nome dell'emergenza, insieme alle coordinate (x e y), oltre al ritardo per l'inserimento di quest'ultima. Deve essere inoltre implementata la possibilità che il client sia eseguito con l'opzione "-f" ("leggi da file"). In questo caso il file passato come argomento sarà in formato testo e per ogni riga avrà la seguente struttura:

```
<nome_emergenza> <coord_x> <coord_y> <delay_in_secs>
```

3.6 Log di Esecuzione

Il sistema di gestione delle emergenze dovrà registrare tutte le operazioni significative in un file di log. Ogni riga del log seguirà il formato: `[Timestamp] [ID] [Evento] Messaggio`.

- `Timestamp` indicherà l'ora dell'evento usando un long.
- `ID` sarà un identificatore univoco associato all'emergenza o all'operazione specifica (es., l'ID dell'emergenza, l'ID della richiesta di parsing). Se il log non è specificamente legato a un'emergenza o operazione, questo campo potrà contenere un valore convenzionale come "N/A" (Non Applicabile).
- `Evento` specificherà la categoria generale dell'azione registrata (es., `FILE_PARSING`, `MESSAGE_QUEUE`, `EMERGENCY_STATUS`, `RESCUER_STATUS`). È compito dello studente definire le categorie necessarie.
- `Messaggio` conterrà una descrizione dell'evento.

Nello specifico, il log conterrà informazioni riguardanti:

- **Parsing dei file di configurazione** (`FILE_PARSING`): apertura dei file (`rescuer.conf`, `emergency_types.conf`, `env.conf`), lettura di ogni riga, valori estratti per soccorritori, tipi di emergenza e ambiente, eventuali errori di formato e completamento del parsing. L'ID in questo caso potrebbe essere l'ID del file stesso o un ID di operazione di parsing.
- **Interazioni con la coda di messaggi** (`MESSAGE_QUEUE`): ricezione di nuove richieste di emergenza, contenuto delle richieste (tipo, coordinate, timestamp), esito della validazione delle richieste e motivo dello scarto per eventuali richieste non valide. L'ID qui sarà l'ID univoco dell'emergenza ricevuta dalla coda.
- **Modifiche allo stato delle emergenze** (`EMERGENCY_STATUS`): transizioni di stato di un'emergenza (da `WAITING` a `ASSIGNED`, da `IN_PROGRESS` a `COMPLETED`, ecc.), inclusa l'ora della modifica e i dettagli dell'emergenza coinvolta. L'ID sarà l'ID dell'emergenza di cui si sta modificando lo stato.
- **Modifiche allo stato dei soccorritori** (`RESCUER_STATUS`): cambiamenti nello stato di un'unità di soccorso (da `IDLE` a `EN_ROUTE_TO_SCENE`, ecc.), inclusa l'ora della modifica e i dettagli del soccorritore. Anche qui, si potrebbe usare un ID del soccorritore.

- **Assegnazione di soccorritori alle emergenze:** registrazione di quali soccorritori sono stati assegnati a quale emergenza e l'ora dell'assegnazione. L'ID in questo caso potrebbe essere l'ID dell'emergenza a cui si sta assegnando o un ID dell'operazione di assegnazione.
- **Timeout delle emergenze:** casi in cui un'emergenza non può essere gestita (leggasi: ricevere tutti i mezzi necessari sul luogo dell'emergenza in tempo, passando nello stato IN_PROGRESS) entro il tempo limite previsto, con indicazione del motivo (carenza di risorse, distanza). L'ID sarà l'ID dell'emergenza che ha subito il timeout.

Questo log dettagliato e arricchito con ID fornirà una cronologia completa e correlabile delle operazioni del sistema, essenziale per il debug, il monitoraggio e l'analisi delle prestazioni.

4 Deadlock [Solo per consegne successive al 19/05/2025]

Nel contesto di un sistema concorrente, si parla di deadlock (o stallo) quando due o più thread o processi rimangono permanentemente in attesa di risorse detenute l'uno dall'altro, impedendo a ciascuno di proseguire l'esecuzione. In pratica, si verificano le quattro condizioni di Coffman—mutua esclusione, attesa di risorse già allocate, non preemption e attesa circolare—che, tutte presenti simultaneamente, garantiscono l'insorgere dello stallo. Dal punto di vista progettuale, il deadlock richiede meccanismi di rilevamento e risoluzione, oppure di prevenzione.

I progetti consegnati successivamente al primo appello utile (19/05/2025) dovranno implementare la gestione del deadlock.

5 Estensioni per chi non ha superato le prove in itinere

Per coloro i quali non avessero superato almeno 3 prove in itinere, il progetto richiede un insieme aggiuntivo di funzionalità, riportate in seguito:

- **Preemption**: interruzione di interventi in corso per gestire emergenze più urgenti. Il sistema deve essere in grado di sospendere interventi la cui gestione è in corso, qualora si palesino richieste a più alta priorità, se l'invio delle risorse allocate ad altre emergenze a minore priorità consenta la loro gestione in tempi utili. I mezzi di soccorso possono essere interrotti solo se nello stato: EN_ROUTE_TO_SCENE oppure ON_SCENE. Un'emergenza in corso di gestione che fosse privata di un mezzo di soccorso, passerebbe allo stato "PAUSED" e il timeout per la sua gestione torna a scorrere.
- **Priorità dinamica (aging)**: meccanismi per evitare starvation. Vi ricordo che la starvation si verifica quando un thread rimane in attesa indefinita di accedere a una risorsa perché altri thread, a più alta priorità o con tempistiche di richiesta più frequenti, ne monopolizzano continuamente l'uso; la starvation non blocca globalmente il sistema, ma impedisce a singoli attori di fare progressi. La starvation si affronta garantendo politiche di scheduling eque (fairness) o invecchiamento ("aging") delle richieste, in modo da evitare che richieste a bassa priorità restino per sempre inevase. Nel caso del progetto vanno implementate quelle funzionalità che possano ridurre la probabilità di starvation per le richieste a bassa priorità facendo in modo che, dopo un certo tempo, passino a priorità media (non oltre).

Anche in questo caso i vari eventi andranno registrati nel file di log.

6 Consegna

- **Pacchetto zip**: il progetto deve essere consegnato come unico file zip contenente il codice sorgente, il makefile e la documentazione.
- **Codice Sorgente**: file `.c`, `.h`, `.conf`, ecc.
- **Makefile**: compilazione ed esecuzione con `make, make run`.
- **Documentazione**: report di max 10 pagine contenente:
  - Architettura del sistema.
  - Scelte progettuali.
  - Istruzioni per compilare ed eseguire.