Homework HWC2_THREAD
di Programmazione Concorrente
1 dicembre 2016 — anno accademico 2016/2017

Modalità di consegna

L'homework va consegnato entro le ore 20:00 di giovedì 22 dicembre 2016 inviando al docente una mail con subject "`PC: HWC2_THREAD` *Nome Cognome Matricola*" ed allegando in formato `.tar.gz` o `.zip` tutti i sorgenti (codice e testo) prodotti. Non inserire nell'archivio eseguibili ed in generale documenti che non siano sorgenti a meno che non facciano parte di librerie esterne necessarie per compilare il codice consegnato.

L'homework si compone di una parte di codice principale da sviluppare secondo le specifiche riportate di seguito, di una serie di test tesi a verificare il corretto comportamento del codice principale, di un documento di testo (usare direttamente file testuali `.txt`, non usare formati proprietari) che descrivi *astrattamente* la propria soluzione in meno di una pagina, a complemento ma non in sostituzione delle informazioni deducibili direttamente dal codice.

Per la consegna sono necessarie le seguenti condizioni: il codice principale deve compilare e funzionare completamente oppure i malfunzionamenti devono essere documentati dai test; il codice dei test deve compilare; il codice dei test deve andare in esecuzione sollecitando il codice principale che si comporta esattamente come previsto dai test oppure il test deve evidenziare chiaramente il problema esistente; il codice dei test deve essere strutturato come specificato di seguito; il documento di testo deve contenere una descrizione della soluzione fedelmente allineata al codice principale sviluppato. Per la scrittura dei test è richiesto di utilizzare il framework CUnit per la scrittura di test di unità in C (http://cunit.sourceforge.net).

La valutazione terrà conto di quanti, di quali test-case sono stati correttamente realizzati, e di come sono stati organizzati.

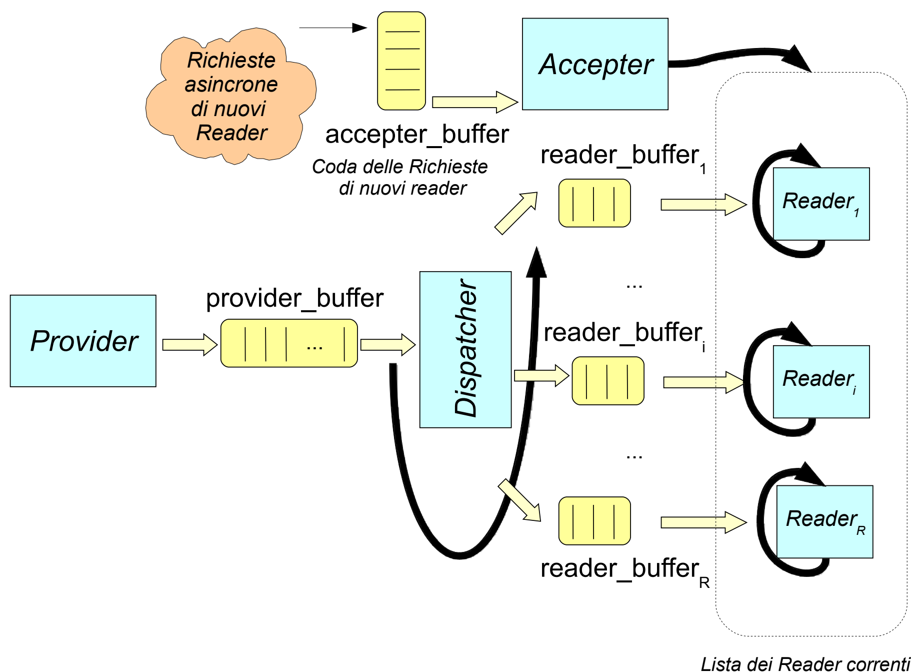

Figura 1: Schema di Soluzione

Specifiche

Riutilizzando il codice prodotto e testato nell'ambito dell'homework `HWC1`, realizzare un *dispatcher*, ovvero un flusso di esecuzione il cui compito è quello di raccogliere una sequenza finita di messaggi provenienti da un altro flusso *provider* e smistarne celermente delle sue copie verso dei flussi *reader* secondo lo schema proposto in Figura 1.

Il numero di flussi *reader* non è noto a priori e può variare dinamicamente: infatti la loro immissione nel sistema è responsabilità di un apposito flusso *accepter* che riceve richieste asincrone dall'esterno. L'*accepter* si occupa di evadere le richieste creando per ciascuna un flusso *reader* corrispondente, e quindi di collocarlo all'interno della *Lista dei* reader correnti a cui a sua volta accederà il *dispatcher*.

Una volta accettati, i *reader* ricevono tramite il *dispatcher* la sequenza di messaggi proveniente dal *provider* a partire dal primo ricevuto successivamente alla sua accettazione. In generale i *reader* possono avere velocità di lettura diverse ed è responsabilità del *dispatcher* assicurarsi che i rallentamenti dovuti ad uno di loro non si propaghino verso tutti gli altri interessati.

I seguenti punti chiariscono le principali interazioni che avvengono tra il

dispatcher, il *provider*, l'*accepter* e le varie istanze dei flussi *reader*:

- il *provider* spedisce una sequenza finita di messaggi al *dispatcher*; la sequenza è sempre terminata da una *poison pill* e dopo il suo invio il *provider* termina spontaneamente
- il *dispatcher* inoltra tutti i messaggi ricevuti dal *provider* verso tutti i flussi *reader* correntemente accettati ed organizzati in una *Lista dei reader*
- i *reader* sono creati a seguito di esplicite richieste asincrone presentate al flusso *accepter* e quindi collocati all'interno della *Lista dei reader*; solo dopo l'evasione della richiesta possono essere presi in considerazione dal *dispatcher*
- il *dispatcher* si impegna a fornire a ciascun *reader* tutti i messaggi ricevuti dal *provider* a cominciare dal primo ricevuto successivamente alla sua accettazione

I seguenti punti dettagliano il comportamento del *dispatcher*:

- il *dispatcher* si impegna ad inoltrare i messaggi che riceve dal *provider* verso i *reader* cercando di minimizzare il ritardo tra la ricezione di ciascun messaggio dal *provider* e la spedizione dello stesso verso i *reader* correnti contenuti nella *Lista dei reader*[1]
- al termine della sequenza di messaggi inviati dal *provider*, il *dispatcher* comunica la fine della comunicazione ai *reader* inoltrandogli la *poison pill* che lui stesso ha ricevuto dal *provider* per marcare la fine della sequenza
- il *dispatcher* inoltra i messaggi rispettando rigidamente l'ordine di ricezione dal *provider* e quindi nel flusso di messaggi che spedisce a ciascun *reader* non ci saranno messaggi che risultano invertiti di ordine e/o non inviati e/o inviati più di una volta
- i nuovi messaggi sono smistati il più celermente possibile verso ogni *reader* ma compatibilmente con le letture che questo ha già effettuato per non alternarne il naturale sequenziamento
- eventuali rallentamenti nella lettura dei messaggi da parte di un *reader* non causeranno ritardi nella spedizione dei messaggi verso gli altri *reader*

[1] In caso di una *Lista dei reader* vuota, il *dispatcher* si limita a consumare il messaggio ricevuto dal *provider* senza inoltrarlo ad altri.

- per evitare la propagazione dei rallentamenti, il *dispatcher* è libero di rimuovere i *reader* che giudica troppo lenti anticipandogli l'invio di opportune *poison pill* anche prima della naturale terminazione della sequenza
- il *dispatcher* è tenuto a terminare spontaneamente alla lettura di una *poison pill* proveniente dal *provider*; tuttavia, prima di finire, disciplina la terminazione di tutti i flussi (di tipo *reader*) coinvolti nel sistema duplicando ed inoltrando le opportune *poison pill*

I seguenti punti dettagliano il funzionamento dell'*accepter*:

- il flusso *accepter* accetta richieste di creazione di nuovi flussi *reader*; le richieste vanno inserite in una apposita *Coda delle richieste* di nuovi *reader*
- le richieste arrivano in maniera asincrona dall'esterno del sistema e pertanto è possibile che molteplici richieste pervengano concorrentemente
- i *reader* saranno creati dall'*accepter* che evade le richieste secondo una politica *FIFO*
- si tiene traccia dei flussi *reader* creati collocando una struttura dati che li rappresenta in una apposita *Lista dei reader* correnti
- anche l'*accepter* termina spontaneamente alla lettura di una *poison pill* dalla *Coda delle richieste* di nuovi *reader*: in questo caso eventuali altre richieste ancora presenti sulla coda saranno scartate e rimarranno inevase

I seguenti punti dettagliano il comportamento dei flussi di tipo *reader*:

- le velocità di lettura dei *reader* possono essere anche sensibilmente diverse da *reader* a *reader*
- nessun *reader* sarà mai completamente fermo: tutti i *reader* consumano i propri messaggi in un tempo finito
- i *reader* sono tenuti a terminare spontaneamente alla lettura di una *poison pill*
- non appena un *reader* termina, la struttura dati che lo rappresenta viene rimossa dalla *Lista dei reader* correnti
- i buffer a cui accedono i *reader* sono di lunghezza fissa, costante, uguale per tutti i *reader*, e nota a priori

Per finire dettagliamo la terminazione dell'applicazione, citando l'esistenza di un ulteriore flusso di esecuzione, che chiameremo *main*. Questo flusso ha la responsabilità di creare il sistema di flussi già descritto, e di gestire attivamente il transitorio di chiusura finale. In particolare *main* partecipa attivamente affinché si verifichino questi eventi:[2]

- il *provider* decide di terminare emettendo una *poison pill*
- *main*, non appena il *provider* termina, invia la *poison pill* all'*accepter*
- il *dispatcher* riceve la *poison pill*, ne smista una copia a tutti i *reader*, e quindi termina
- i *reader* terminano non appena ricevono, una ciascuno, la *poison pill*
- l'*accepter* riceve una *poison pill* e termina
- *main* aspetta la terminazione dell'*accepter* per inviare la *poison pill* ai *reader* ancora esistenti
- *main* termina

Implementare una possibile soluzione al problema proposto utilizzando i POSIX thread in C sotto Linux scegliendo liberamente gli strumenti di sincronizzazione più comodi allo scopo (in particolare mutex, condizioni, o semafori per thread).

Viene esplicitamente richiesto di rispettare le segnature ideate per l'homework `HWC1`. Per comodità vengono riportate, in Figura 2, le segnature delle relative funzioni. Si precisa che la *poison pill* è codificata con il messaggio costante `POISON_PILL` come chiarito nelle relative dichiarazioni di Figura 4 e nell'unità di compilazione di Figura 5.

La gestione della *Lista dei reader* può avvenire utilizzando una semplice libreria (`hwc2list.h`) per la gestione di una lista di elementi che viene fornita in allegato. Le segnature delle funzioni che compongono la libreria sono mostrate in Figura 3. Si precisa che la libreria fornita *non* è tuttavia *thread-safe*, e bisogna gestire opportunamente il suo utilizzo in un contesto multi-thread.

La sottomissione di nuove richieste di creazione di flussi *reader* all'*accepter* avviene invocando una apposita funzione `submitRequest()` di cui si può liberamente fissare una segnatura. Ad esempio:

```
// richiedi la creaz. di un reader
void submitRequest(buffer_t *requests, char name[]);
```

riceve come primo parametro la *Coda delle richieste* e come secondo parametro un nome che si vuole assegnare al nuovo *reader*, nome da utilizzarsi a scopi puramente informativi.

---

[2] N.B.: La sequenza elencata non vuole implicare un ordinamento cronologico totale degli eventi citati.

```
#define BUFFER_ERROR  (msg_t *) NULL
/* allocazione / deallocazione  buffer */
// creazione di un buffer vuoto di dim. max nota
buffer_t* buffer_init(unsigned int maxsize);

// deallocazione di un buffer
void buffer_destroy(buffer_t* buffer);

/* operazioni sul buffer */
// inserimento bloccante: sospende se pieno, quindi
// effettua l'inserimento non appena si libera dello spazio
// restituisce il messaggio inserito; N.B.: msg!=null
msg_t* put_bloccante(buffer_t* buffer, msg_t* msg);

// inserimento non bloccante: restituisce BUFFER_ERROR se pieno,
// altrimenti effettua l'inserimento e restituisce il messaggio
// inserito; N.B.: msg!=null
msg_t* put_non_bloccante(buffer_t* buffer, msg_t* msg);

// estrazione bloccante: sospende se vuoto, quindi
// restituisce il valore estratto non appena disponibile
msg_t* get_bloccante(buffer_t* buffer);

// estrazione non bloccante: restituisce BUFFER_ERROR se vuoto
// ed il valore estratto in caso contrario
msg_t* get_non_bloccante(buffer_t* buffer);

/* --- msg_t --- */

typedef struct msg {

    void* content;   // generico contenuto del messaggio

    struct msg * (*msg_init)(void*);        // creazione msg
    void (*msg_destroy)(struct msg *);      // deallocazione msg
    struct msg * (*msg_copy)(struct msg *); // creazione/copia msg

} msg_t;
```

Figura 2: Segnature delle funzioni per la gestione di buffer implementate nell'homework `HWC1`

```
//Gestione lista di elementi tipati puntatori a void (void *)
//Gli elementi non possono essere NULL
list_t*  list_init();               // crea una lista vuota
void list_destroy(list_t *list);    // dealloca una lista
int size(list_t* list);             // restituisce il numero di elementi
int isEmpty(list_t* list);          // restituisce vero sse vuota
void addElement(list_t *list, void *element);   // aggiunge in fondo
int removeElement(list_t *list, void *element); // rimuove il primo elemento
// il cui contenuto ha indirizzo element; restituisce falso se inesistente
iterator_t* iterator_init(list_t *list); // crea un iteratore sulla lista
void iterator_destroy(iterator_t *it);   // dealloca un iteratore
int hasNext(iterator_t *it);          // scansione finita?
void *next(iterator_t *it);           // prossimo elemento, NULL se finiti
void removeLastReturned(iterator_t *it); // rimuove ultimo elemento restituito
```

Figura 3: Segnature delle funzioni per la gestione di una lista di elementi di tipo `void *` nella libreria `hwc2list.h`

```
#define POISON_PILL ( (msg_t*)&POISON_PILL_MSG )

msg_t* msg_init_pill(void *);
msg_t* msg_copy_pill(msg_t *);
void msg_destroy_pill(msg_t *);

extern const msg_t POISON_PILL_MSG;
```

Figura 4: Dichiarazioni relative ad un messaggio singleton che svolga il ruolo di *poison pill*: il file `poison_pill.h`

Organizzazione dei Test di Unità

Ferme restanti tutte le considerazioni già fatte per l'homework `HWC1`,[3] il testing di unità in presenza di una architettura a più componenti ed un sistema di flussi più articolato permette considerazioni aggiuntive.

I test di unità devono verificare il corretto funzionamento di tutte le componenti del sistema considerandole il più possibile *isolatamente*. Oltre alla gestione dei buffer, già oggetto dell'homework `HWC1`, bisogna testare queste altre componenti: il *reader*, il *provider*, l'*accepter*, il *main* e per finire il componente più articolato ed anche più difficile da testare: il *dispatcher*.

I test di unità migliori sono quelli che riescono a testare un componente del sistema *isolatamente* da tutti gli altri allo scopo di verificare il corretto funzionamento del componente in assenza di interazioni complesse con le altre componenti.[4]

[3] Consultare: http://crescenzi.dia.uniroma3.it/didattica/aa2016-2017/PC/unit-testing.html

[4] Esistono altre tipologie di test espressamente dedicate a verificare il corretto svolgimento dei dialoghi tra componenti distinte. Tuttavia non si possono chiamare test di unità in senso stretto.

```
const msg_t POISON_PILL_MSG = {
        NULL,
        msg_init_pill,
        msg_destroy_pill,
        msg_copy_pill
};

msg_t* msg_init_pill(void *content) { return POISON_PILL; }

msg_t* msg_copy_pill(msg_t *msg)    { return POISON_PILL; }

void msg_destroy_pill(msg_t *msg)   { /* do nothing */    }
```

Figura 5: Un messaggio singleton e relative funzioni per la gestione di *poison pill*: il file `poison_pill.c`

Questa operazione dovrebbe risultare semplice per le strutture dati utilizzate: ad esempio il generico buffer per produttori/consumatori già testato nel precedente homework. Relativamente ai flussi, il testing isolato dovrebbe risultare agevole per il *provider* e per i *reader*. Al contrario risulta meno agevole per l'*accepter*, per il *main* e per il *dispatcher* che per loro natura interagiscono con altri flussi.

Vale la pena di ribadire che conviene intervallare la scrittura dei test con la scrittura dei componenti ed evitare di relegare la scrittura dei test solo in una fase finale successiva alla scrittura del codice principale. Inoltre è conveniente cominciare dalla scrittura (e dal relativo testing) delle componenti più semplici e più isolate. Ad esempio, conviene scrivere e testare prima la la gestione dei buffer e solo dopo il *provider* ed il *reader*; conviene scrivere i test dell'*accepter* e del *dispatcher* per ultimi.

In questa maniera è ragionevole assumere che durante la scrittura dei test del *dispatcher* si possa fare affidamento su una versione dei *reader* e dei *provider* sufficiente testati da poterli considerare corretti, e quindi ricercare la colpa dei fallimenti dei test del *dispatcher* all'interno del codice dello stesso e non altrove.

Resta valido il suggerimento di realizzare versioni semplificate e specializzate di ogni flusso collaboratore che pur non essendo il diretto oggetto delle verifiche di un test-case in corso di scrittura, risulti indispensabile per eseguire il codice testato. Ad esempio è possibile creare delle versioni di *reader lento* e/o *veloce* utili ai soli scopi di testing del *dispatcher* e non facenti parte del codice in produzione. Allo stesso modo è consigliabile creare versioni del *provider* che emettano sequenze di messaggi predeterminate.