Homework HWC1 di Programmazione Concorrente
17 novembre 2017 — anno accademico 2017/2018

Modalità di consegna

L'homework va consegnato entro le ore 20:00 di domenica, 3 di dicembre, inviando al docente una mail con subject "`PC: HWC1` *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.

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 mezza pagina, e si limiti a descrivere i soli aspetti che non siano più facilmente apprezzabili direttamente dal codice sorgente. Per finire l'homework prevede una seconda parte facoltativa.

Per la consegna si invita a produrre elaboborati che rispettino tutte le seguenti condizioni: il codice principale deve compilare e funzionare completamente oppure i malfunzionamenti devono essere documentati da opportuni test (più semplici sono, meglio è); 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.

Non è invece necessario sviluppare tutti i test-case per tutti gli scenari riportati di seguito: la valutazione terrà conto dei criteri di scelta, della qualità dei test-case realizzati, e della loro *minimalità*.

Specifiche

Implementare una possibile soluzione al problema classico dei produttori/consumatori nella variante in cui esistono `C` consumatori, `P` produttori ed un buffer intermedio capace di ospitare un numero finito di `N` generici messaggi non nulli. Utilizzare 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).

Prevedere la possibilità che le operazioni di inserimento e di estrazione possano entrambe essere effettuate utilizzando sia funzioni bloccanti che funzioni non bloccanti. In particolare, le estrazioni bloccanti seguiranno la tradizionale semantica secondo la quale un flusso di esecuzione che estrae da un buffer vuoto verrà sospeso in attesa che qualche produttore vi deponga nuovi messaggi; gli inserimenti bloccanti comporteranno la sospensione di un produttore che inserisce un messaggio in un buffer già pieno in attesa che qualche consumatore liberi delle posizioni.

Le funzioni non bloccanti restituiscono immediatamente il controllo al flusso di esecuzione invocante senza comportare sospensioni e restituendo un codice di ritorno se impossibilitate a dare seguito immediato all'operazione richiesta: pertanto un inserimento non bloccante su un buffer pieno restituisce immediatamente il controllo al flusso invocante restituendo un codice di ritorno prestabilito, e parimenti un'estrazione non bloccante da un buffer vuoto restituisce immediatamente il controllo al flusso invocante con un valore di ritorno prestabilito.

Per garantire una sufficiente genericità della soluzione prodotta, da doversi riutilizzare in successivi homework, viene esplicitamente richiesto di utilizzare le segnature riportate in Figura 1, adottando l'ipotesi che i buffer siano modellati con il tipo di dato `buffer_t` da definirsi nel dettaglio, che i generici messaggi siano modellati con il tipo `msg_t` riportato in Figura 2, e che i codici restituiti dalle funzioni siano definite con la costante `BUFFER_ERROR`.

Il tipo di dato `buffer_t` è una struttura dati da definirsi in funzione della soluzione adottata, ma che in essenza vuole modellare tutto il necessario per gestire il buffer intermedio su cui operano le primitive di cui sopra; ad es. può contenere almeno gli indici di accesso (estrazione e inserimento) ed un array che ospiterà i messaggi scambiati. Notare anche che la funzione di inserimento esclude la possibilità di inserire messaggi *nulli*: questa scelta permette una più agevole distinzione, ad es. in fase di estrazione, dei messaggi utilizzati per comunicare il codice di errore `BUFFER_ERROR==NULL`.

Il tipo di dato `msg_t` consente di rappresentare un generico messaggio da poter inserire ed estrarre in buffer di tipo `buffer_t` con le funzioni oggetto del presente homework. La struttura dati in questione, oltre a rappresentare genericamente il contenuto del messaggio con un campo `content` di tipo

```
#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);
```

Figura 1: Segnature delle funzioni da implementare

`void *`, raccoglie anche tre puntatori a funzioni: `msg_init`, `msg_destroy`, e `msg_copy`. Queste funzioni servono a gestire esplicitamente il ciclo di vita dei messaggi e consentono rispettivamente di:

`msg_t *msg_init(void *content)` creare ed allocare un nuovo messaggio ospitante un generico contenuto `content` di tipo `void*`

`void msg_destroy(msg_t *msg)` deallocare un messaggio

`msg_t* msg_copy(msg_t* msg)` creare ed allocare un nuovo messaggio ospitante il medesimo contenuto di un messaggio dato, similarmente ai classici costruttori di copia

Ad esempio, è possibile utilizzare il tipo `msg_t` per ospitare delle stringhe utilizzando le definizioni mostrate in Figura 3.

Organizzazione dei Test

Per la scrittura dei test utilizzare il framework CUnit per la scrittura di test di unità in C (http://cunit.sourceforge.net).

```
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: Tipo `msg_t` inclusivo di puntatori a funzioni per operare sui messaggi

I test-case devono verificare il corretto funzionamento delle funzioni richieste in diversi scenari di utilizzo. Notare che non bisogna testare il codice di ipotetici produttori/consumatori: oggetto del presente homework sono le quattro funzioni di cui sopra ma non viene affatto richiesto di sviluppare e consegnare il codice dei produttori/consumatori.

L'insieme di test-case e di scenari da considerare deve essere definito cercando di raggiungere un ottimale compromesso tra le esigenze contrapposte che naturalmente nascono in fase di test. In particolare bisogna verificare per primi gli scenari più semplici e minimali, in quanto utili per rilevare in un contesto semplificato i più gravi errori di progettazione. Solo dopo conviene verificare gli scenari più rappresentativi, in quanto utili a validare la soluzione prodotta prima di mandarla in produzione. Di seguito verrano elencati un insieme di scenari minimali per il testing: lo studente è libero di ampliarlo o modificarlo.

Ogni test-case si articola in tre passi:

set-up si prepara uno scenario di utilizzo del codice principale

sollecitazione si invoca il codice principale sollecitando uno degli aspetti di cui si vuol verificare il funzionamento

verifica si verifica che l'esecuzione sia andata come atteso, ad esempio controllando che lo stato raggiunto a causa della sollecitazione sia proprio quello atteso

Per chiarezza si svolgono due esempi. Il primo test-case è particolarmente semplice in quanto non richiede la creazione di più flussi di esecuzione e riguarda la produzione non bloccante in un buffer già pieno. Il secondo test-case è più articolato, in quanto si prefigge di verificare il corretto funzionamento nello scenario in cui un solo consumatore si blocca in seguito ad una estrazione bloccante da un buffer vuoto, e pertanto necessita della creazione di più flussi di esecuzione da sincronizzarsi opportunamente.

```
#include <stdlib.h>
#include <string.h>

msg_t* msg_init_string(void* content) {
    //viene creata una copia "privata" della stringa
    msg_t* new_msg = (msg_t*)malloc( sizeof(msg_t) );
    char* string = (char*)content;
    char* new_content = (char*)malloc(strlen(string)+1); // +1 per \0 finale
    strcpy(new_content, string);

    new_msg->content     = new_content;
    new_msg->msg_init    = msg_init_string;
    new_msg->msg_destroy = msg_destroy_string;
    new_msg->msg_copy    = msg_copy_string;

    return new_msg;
}

void msg_destroy_string(msg_t* msg) {
    free(msg->content); // free copia privata
    free(msg);          // free struct
}

msg_t* msg_copy_string(msg_t* msg) {
    return msg->msg_init_string( msg->content );
}
```

Figura 3: Esempio di utilizzo del tipo `msg_t` per gestire stringhe

Produzione non bloccante in un buffer già pieno

set-up si crea un buffer pieno `buffer_t *buffer_pieno` di dimensione unitaria che ospita l'intero `EXPECTED_MSG`

sollecitazione si effettua un'unica produzione su `buffer_pieno` invocando `put_non_bloccante(buffer_pieno, MSG)` con `MSG!=EXPECTED_MSG`

verifica si verifica che la chiamata `put_non_bloccante()` abbia restituito `BUFFER_ERROR`, e che il buffer risulti tuttora pieno e contenente `EXPECTED_MSG`

Consumazione bloccante da un buffer inizialmente vuoto

set-up si crea un buffer vuoto `buffer_t *buffer_vuoto` ed un apposito tipo di consumatore che effettuerà una sola consumazione bloccante; in questo modo il consumatore sarà inizialmente sospeso

sollecitazione appurato che il consumatore è bloccato, lo si sblocca depositando un messaggio `GO_MSG` in `buffer_vuoto` e lo si lascia libero di effettuare la sua unica consumazione

verifica si verifica che il consumatore abbia effettuato la sua unica consumazione e che abbia ricevuto il messaggio `GO_MSG`; si verifica che `buffer_vuoto` sia tornato ad assere vuoto

Elenco di Scenari per i Test-Case

Di seguito viene riportato un elenco di possibili scenari per il testing del codice prodotto. L'elenco risulta approssimativamente ordinato per complessità dello scenario delineato. `C` indica il numero di consumatori, `P` indica il numero di produttori, `N` è la dimensione del buffer. L'elenco non pretende, in nessun modo, di essere esaustivo, ma consente di evidenziare come sia possibile creare molteplici test-case differenziati per il numero e la tipologia dei flussi coinvolti, per la dimensione del buffer, per la particolare sequenza di interleaving che sollecitano, ed infine, per la scelta della funzione del codice principale che viene sollecitata. I test migliori sono brevi, semplici, leggibili, auto-contenuti e l'aspetto che intendono verificare risulta unitariamente e nitidamente circoscritto. In una parola, i test migliori sono *minimali*:

- (`P=1`; `C=0`; `N=1`) *Produzione di un solo messaggio in un buffer vuoto*
- (`P=0`; `C=1`; `N=1`) *Consumazione di un solo messaggio da un buffer pieno*
- (`P=1`; `C=0`; `N=1`) *Produzione in un buffer pieno*
- (`P=0`; `C=1`; `N=1`) *Consumazione da un buffer vuoto*
- (`P=1`; `C=1`; `N=1`) *Consumazione e produzione concorrente di un messaggio da un buffer unitario; prima il consumatore*
- (`P=1`; `C=1`; `N=1`) *Consumazione e produzione concorrente di un messaggio in un buffer unitario; prima il produttore*
- (`P>1`; `C=0`; `N=1`) *Produzione concorrente di molteplici messaggi in un buffer unitario vuoto*
- (`P=0`; `C>1`; `N=1`) *Consumazione concorrente di molteplici messaggi da un buffer unitario pieno*
- (`P>1`; `C=0`; `N>1`) *Produzione concorrente di molteplici messaggi in un buffer vuoto; il buffer non si riempe*
- (`P>1`; `C=0`; `N>1`) *Produzione concorrente di molteplici messaggi in un buffer pieno; il buffer è già saturo*
- (`P>1`; `C=0`; `N>1`) *Produzione concorrente di molteplici messaggi in un buffer vuoto; il buffer si satura in corso*

- (`P=0`; `C>1`; `N>1`) *Consumazione concorrente di molteplici messaggi da un buffer pieno*
- (`P>1`; `C>1`; `N=1`) *Consumazioni e produzioni concorrenti di molteplici messaggi in un buffer unitario*
- (`P>1`; `C>1`; `N>1`) *Consumazioni e produzioni concorrenti di molteplici messaggi in un buffer*

Suggerimenti

È controproducente scrivere tutti i test-case solo alla fine, dopo aver completato il codice principale: al contrario è molto più proficuo intervallare la scrittura di una parte del codice principale con la scrittura dei relativi test-case, per poi tornare a lavorare sul completamento del codice già scritto od anche su alcune estensioni e/o miglioramenti (ad es. del livello di parallelismo). Seguendo questa metodologia di sviluppo, non si avrà mai troppo codice principale non testato, e come tale inaffidabile. Pertanto in seguito a modifiche del codice che danno luogo a fallimenti nei test, si massimizzerà la probabilità che gli errori siano localizzati proprio nelle ultime modifiche effettuate. Inoltre il patrimonio di test-case accumulato, se di qualità, deve permettere di localizzare gli errori con uno sforzo contenuto.

Per aumentare la qualità dei test-case prodotti (as esempio la loro *leggibilità* e la loro *località*) si suggerisce di creare contestualmente a ciascun test-case dei produttori/consumatori semplificati, ovvero versioni *usa-e-getta* di un produttore o di un consumatore (ad esempio che inseriscono/estraggono un solo valore e poi terminano) che servono unicamente a quel test-case per creare e verificare il particolare scenario che si vuole riprodurre. Questi produttori/consumatori specializzati non entranno a far parte del codice principale e non ha senso cercare di generalizzarli oltre misura per riciclarli tra test-case distinti, in quanto all'aumentare della loro complessità diminuisce la località del test, ovvero la sua capacità di fallire per colpa di errori presenti specificatamente nel corpo delle funzioni testate.

Altri produttori/consumatori *usa-e-getta* plausibili per il testing di codice concorrente potrebbero limitarsi ad effettuare una sequenza di tre passi: (*i*) sospensione in attesa di un segnale di partenza; (*ii*) singola produzione/consumazione rispett.; (*iii*) segnalazione di terminazione e comunicazione del valore restituito dalla `put()` e `get()` rispett. In questa maniera il flusso che svolge il ruolo di *driver*[1] dei test può decidere se creare altri flussi di queste tipologie per realizzare il proprio scenario di testing. Il *driver* guiderà appropriatamente, e direttamente nel corpo del codice del test-case, questi flussi specializzati di modo da realizzare alcune particolari sequenze di interleaving di interesse per il test-case in oggetto.

[1] Questo è il flusso che viene creato dal framework di testing per eseguire i test-case.

Back-Pressure (*Facoltativo*)

Un scenario sempre più comune per il crescente interesse verso l'elaborazione di enormi quantità di dati, prevede che diversi nodi di elaborazione comunichino i propri risultati parziali utilizzando dei buffer che occupano una quantità di memoria *finita* per disaccoppiare la produzione dei semilavorati in output ad un nodo di computazione e di input ad un successivo nodo di computazione. Con un'opportuna progettazione dei nodi di elaborazione, attività ormai supportata da un numero sempre maggiore di librerie, se non addirittura da interi paradigmi di programmazione (come quello ad *attori*), è possibile ottenere un buon speed-up semplicemente impiegando diversi thread di esecuzione per ciascun nodo computazionale così distribuendo il carico computazionale su tutti i processori fisici disponibili. I nodi di computazione, in un certo senso, si configurano come *consumatori* dei semi-lavorati in ingresso e *produttori* di quelli in uscita.

Il principale problema che si riscontra è se i nodi in questione sono separati da un *asynchronous boundary*, come accade in uno degli scenari di maggiore interesse, ovvero distribuendo la computazione su più macchine, e non solo su tutti i processori di una stessa macchina fisica. Infatti, mentre se i nodi risiedono sulla stessa macchina la natura bloccante stessa del buffer evita la sovraproduzione di un nodo produttore rispetto ad un consumatore (semplicemente, se uno di questi buffer si riempe, i produttori che vi insistono sopra sono costretti a fermarsi in attesa dei consumatori, che finiscono per acquisire maggiori risorse computazionali), in uno scenario distribuito la sovraproduzione potrebbe significare un sotto-utilizzo delle risorse di una macchina, oppure l'utilizzo di buffer di memoria illimitata, soluzione nella pratica ritenuta inaccettabile.

Progettare una soluzione a questo problema di sempre maggiore importanza inserendo un meccanismo di *back-pressure* in cui i consumatori sono abilitati a comunicare "all'indietro", verso i produttori, l'esistenza di un problema di eccesso di produzione sicché questi ultimi possano rallentare e rilasciare risorse di calcolo, che comunque risulterebbero inutilizzate, oppure, peggio ancora, porterebbere all'overflow di uno dei buffer di disaccoppiamento.

Modificare il codice prodotto ed implementare il meccanismo di back-pressure sopra descritto per supportare uno scenario in cui i consumatori ed i produttori, attraverso la condivisione del buffer, aderiscono ad un protocollo comune per coordinare il bilanciamento del carico, rendendo possibile l'impiego di soli buffer di dimensione finita, come sempre accade nella pratica.