Homework HWC2_IPC 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_IPC` *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.

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.

Non è invece necessario sviluppare tutti i test-case per tutti gli scenari riportati di seguito, ma la valutazione terrà conto della quantità e della qualità dei test-case realizzati.

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. Riprendere le specifiche riportate in `HWC1` ma utilizzare le chiamate di sistema di un sistema UNIX per realizzare i flussi di esecuzione mediante processi in C. Come strumento di sincronizzazione utilizzare i semafori per processi.

Viene esplicitamente richiesto di progettare una soluzione il più possibile *resistente* alla propagazione dei fallimenti degli utilizzatori: si consideri la presenza di produttori e consumatori che possano imprevedibilmente fallire, e cercare di evitare che il fallimento di un generico produttore/consumatore si ripercuota (ad esempio causando lo stallo) sugli altri flussi di esecuzione che insistono sul medesimo buffer. Al contrario, si ritiene assolutamente accettabile che il fallimento di un produttore/consumatore causi la perdita del relativo processo e/o dei soli messaggi che ha già prodotto/consumato e/o stava per produrre/consumare.

Viene esplicitamente richiesto di utilizzare le segnature elencate in `HWC1` e che ricordiamo per comodità nelle Figure 1 e 2.

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).

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.

Elenco di Scenari per i Test-Case

Di seguito viene riportato alcuni possibili scenari per il testing del codice prodotto, partendo da quelli già delineati per `HWC1`. Altri sono aggiunti con l'obiettivo di precisare ulteriormente che cosa si intende per codice resistente ai fallimenti degli utilizzatori.

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, `*` indica se vi possono essere dei fallimenti.

```
#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

```
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 seguenti scenari sono ben motivati sia per `HWC1` sia per il presente homework:

- (`P=1`; `C=0`; `N=1`) *Produzione di un solo messaggio in un buffer vuoto*
- (`P=0`; `C=1`; `N=1`) *Consumazione di un messaggio da un buffer pieno*
- ... *[Tutti gli scenari delineati in* `HWC1` *hanno senso anche qui]*
- (`P>1`; `C>1`; `N>1`) *Consumazioni e produzioni concorrenti di molteplici messaggi in un buffer*

I seguenti scenari non risulterebbero ben motivati nel contesto di `HWC1`, ma chiariscono che cosa si intende per resistenza ai fallimenti degli utilizzatori:

- (`P=0`; `C=1*`; `N=1`) *Consumazione di un solo messaggio da un buffer pieno, il consumatore può fallire*
- ... *[Altre varianti con fallimenti degli scenari delineati in* `HWC1` *]*
- (`P>1*`; `C>1*`; `N>1`) *Consumazioni e produzioni concorrenti di molteplici messaggi in un buffer, un consumatore fallisce fuori sezione critica*
- (`P>1*`; `C>1*`; `N>1`) *Consumazioni e produzioni concorrenti di molteplici messaggi in un buffer, un consumatore fallisce in sezione critica*

Suggerimento: valutare l'impiego di semafori System V per processi con l'uso dell'opzione `SEM_UNDO`.