Sistemi Operativi (M. Cesati)

Compito scritto del 4 settembre 2015

Nome: _____ **Cognome:** _____

Matricola: _____ **Corso di laurea:** _____

Crediti da conseguire: 5 6 9

Scrivere i dati richiesti in stampatello. Al termine consegnare tutti i fogli. Per conseguire un voto sufficiente è necessario ottenere un voto non gravemente insufficiente in entrambi gli esercizi. Tempo a disposizione: 2,5 ore.

Esercizio 1

Scrivere una applicazione C/POSIX multithread costituita da tre thread che implementano le seguenti procedure:

T1) Il primo thread inserisce in una lista dinamica ordinata per valori crescenti tutti i valori interi `x` maggiori di 1, ossia 2, 3, 4, 5, ecc.

T2) Contemporaneamente, il secondo thread rimuove dalla stessa lista dinamica i valori interi che possono essere divisi esattamente da uno dei valori precedenti nella lista. Ad esempio, il thread lascerà in lista i valori 2 e 3, ma rimuoverà il valore 4, perché è divisibile per 2.

T3) Contemporaneamente, il terzo thread accede alla stessa lista dinamica e scrive in standard output i valori che *non* sono stati (e *non* dovranno essere) cancellati dal secondo thread.

I thread debbono eseguire il più possibile in parallelo tra loro, ma si deve evitare ogni "race condition" dovuta agli accessi alla lista dinamica condivisa. Si ponga attenzione alla sincronizzazione tra i thread. Ad esempio, T3 non deve stampare valori in lista che dovrebbero essere rimossi da T2.

Esercizio 2

Si descrivano in modo chiaro ed esauriente quali soluzioni possono essere adottate nei moderni sistemi operativi per prevenire gli stalli dovuti all'utilizzo delle primitive di sincronizzazione (*deadlock*).

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 4 settembre 2015

La questione più importante da risolvere in questo esercizio è la sincronizzazione tra i tre thread nell'accesso alla lista dinamica di valori interi. Riassumiamo brevemente la situazione:

T1 Il thread T1 deve semplicemente aggiungere elementi alla lista, in coda. Non modificherà mai alcun elemento della lista che non sia l'ultimo elemento.

T2 Il thread T2 deve rimuovere elementi dalla lista. Nei confronti di T1, potrebbe creare un problema se cancellasse l'ultimo elemento della lista, in quanto è quello modificato da T1 per aggiungere elementi. Possiamo dunque progettare l'applicazione in modo che T2 possa modificare qualunque elemento della lista tranne l'ultimo.

T3 Il thread T3 non modifica mai alcun elemento della lista. D'altra parte, non deve processare alcun elemento della lista che non sia stato preventivamente analizzato dal thread T2.

Per sincronizzare i thread T1 e T2 utilizziamo una variabile puntatore `ptr_last` che contiene l'indirizzo dell'ultimo elemento della lista. Ad essa sono associati un mutex `mtx_last` ed una variabile condizione `cond_last`.

In modo analogo, per sincronizzare i thread T2 e T3 utilizziamo una variabile puntatore `ptr_safe`, che contiene l'indirizzo dell'ultimo elemento della lista che è stato analizzato da T2 ed è quindi possibile stampare in T3. A questo puntatore sono associati un altro mutex `mtx_safe` ed un'altra variabile condizione `cond_safe`.

Presentiamo il programma seguendo un approccio "top-down".

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct node_t {
    unsigned long long value;
    struct node_t *next;
};

struct node_t * head;

struct node_t * ptr_last;
pthread_mutex_t mtx_last;
pthread_cond_t  cond_last;
```

```
struct node_t * ptr_safe;
pthread_mutex_t mtx_safe;
pthread_cond_t  cond_safe;
```

Ovviamente la struttura `node_t` rappresenta un elemento della lista, e la variabile `head` contiene l'indirizzo del primo elemento della lista.

```
int main()
{
    initialize();
    T1_job();
    return 0;
}
```

Il thread originale dell'applicazione svolgerà il ruolo di T1. Quando T1 termina, l'intera applicazione viene chiusa. (Questo caso in pratica non si realizza mai: la lista dinamica alla fine occupa così tanta RAM da causare la terminazione anomala del programma.)

La funzione `initialize()` si occupa di inizializzare tutte le strutture di dati utilizzate dall'applicazione:

```
void initialize(void)
{
    pthread_t tid1, tid2;
    if (pthread_mutex_init(&mtx_last, NULL) ||
        pthread_mutex_init(&mtx_safe, NULL))
        error_exit("Error in pthread_mutex_init()\n");
    if (pthread_cond_init(&cond_last, NULL) ||
        pthread_cond_init(&cond_safe, NULL))
        error_exit("Error in pthread_cond_init()\n");
    head = NULL;
    insert_value(2, &head);
    ptr_last = ptr_safe = head;
    if (pthread_create(&tid1, NULL, T2_job, NULL) ||
        pthread_create(&tid2, NULL, T3_job, NULL))
        error_exit("Error in pthread_create()\n");
}
```

La funzione `error_exit()` serve a terminare il programma segnalando una condizione d'errore:

```
void error_exit(const char *msg)
{
    fputs(msg, stderr);
    exit(EXIT_FAILURE);
}
```

Il primo elemento della lista dinamica (corrispondente al valore 2) viene inserito direttamente da `initialize()`. A tale scopo si utilizza la funzione `insert_value()`:

```
void insert_value(unsigned long long value,
                  struct node_t **pnode)
{
    struct node_t *new = malloc(sizeof(struct node_t));
    if (new == NULL)
        error_exit("Memory allocation error\n");
    new->value = value;
    new->next = *pnode;
    *pnode = new;
}
```

Presentiamo ora la funzione `T1_job()`, che implementa la procedura svolta specificatamente dal thread T1.

```
void T1_job(void)
{
    unsigned long long value;
    struct node_t *p = head;
    for (value=3; value>2; ++value) {
        insert_value(value, &p->next);
        lock(&mtx_last);
        ptr_last = p->next;
        awake(&cond_last);
        unlock(&mtx_last);
        p = p->next;
    }
}
```

La funzione esegue un ciclo che termina solo quando la variabile `value` va in overflow. Poiché tipicamente `value` è costituita da 64 bit, questo caso in pratica non si verifica mai. Dopo aver inserito un nuovo elemento in coda alla lista, si acquisisce il mutex `mtx_last`, si aggiorna il puntatore `ptr_last`, si fa in modo di risvegliare T2 se esso sta dormendo sulla variabile condizione `cond_last`, ed infine si rilascia il mutex. Tutto ciò viene realizzato con le seguenti funzioni di servizio:

```
void lock(pthread_mutex_t *mtx)
{
    if (pthread_mutex_lock(mtx))
        error_exit("Error in pthread_mutex_lock()\n");
}
void unlock(pthread_mutex_t *mtx)
{
    if (pthread_mutex_unlock(mtx))
        error_exit("Error in pthread_mutex_unlock()\n");
}
void awake(pthread_cond_t *cnd)
{
    if (pthread_cond_signal(cnd))
        error_exit("Error in pthread_cond_signal()\n");
}
```

La funzione `T2_job()` realizza invece la procedura del thread T2:

```
void *T2_job(void *v)
{
    struct node_t *p = head, *prev = NULL;
    v = v; /* v is unused */
    for(;;) {
        lock(&mtx_last);
        if (p == ptr_last)
            wait(&cond_last, &mtx_last);
        unlock(&mtx_last);
        if (is_divisible(p->value, prev)) {
            p = p->next;
            remove_next_node(prev);
        } else {
            if (ptr_safe != p) {
                lock(&mtx_safe);
                ptr_safe = p;
                awake(&cond_safe);
                unlock(&mtx_safe);
            }
            prev = p;
            p = p->next;
        }
    }
}
```

Innanzi tutto la funzione acquisisce il mutex `mtx_last`, poi confronta l'indirizzo dell'elemento da controllare con il valore in tale puntatore. Se coincidono, T2 deve attendere che T1 aggiunga un altro elemento in lista, perciò blocca sulla variabile condizione `cond_last` utilizzando la funzione `wait()`:

```
void wait(pthread_cond_t *cnd, pthread_mutex_t *mtx)
{
    if (pthread_cond_wait(cnd, mtx))
        error_exit("Error in pthread_cond_wait()\n");
}
```

Per controllare la divisibilità di un elemento `T2_job()` utilizza la funzione `is_divisible()`. Si noti che `prev` contiene l'indirizzo dell'elemento in lista che precede quello attualmente analizzato, ovvero `NULL` se l'elemento da analizzare coincide con `head`.

```
int is_divisible(unsigned long long value, struct node_t *
    limit)
{
    struct node_t *p = head;
    if (limit == NULL)
        return 0;
    while (p != limit) {
        if (value % p->value == 0)
            return 1;
        p = p->next;
    }
    return 0;
}
```

Ovviamente se `limit` (ossia `prev`) è nullo non vi è nulla da fare. Altrimenti si scandiscono tutti gli elementi della lista, e nel caso se ne trovi uno che divide esattamente il valore dato si restituisce 1 (true). Altrimenti si restituisce 0.

Se `T2_job()` determina che l'elemento è divisibile, lo rimuove dalla lista utilizzando la funzione `remove_next_node()`:

```
void remove_next_node(struct node_t *prev)
{
    struct node_t *toberemoved = prev->next;
    prev->next = toberemoved->next;
    free(toberemoved);
}
```

Se invece l'elemento deve restare in lista è necessario aggiornare il valore del puntatore `ptr_safe`. Al solito, si acquisisce il rispettivo mutex, si aggiorna il puntatore, si risveglia l'eventuale thread T3 che aspetta sulla variabile condizione `cond_safe`, e si rilascia il mutex.

Infine, la funzione `T3_job()` che implementa la procedura eseguita dal thread T3:

```
void *T3_job(void *v)
{
    struct node_t *p = head;
    v = v; /* v is unused */
    for (;;) {
        lock(&mtx_safe);
        if (p == ptr_safe)
            wait(&cond_safe, &mtx_safe);
        unlock(&mtx_safe);
        printf("%llu\n", p->value);
        p = p->next;
    }
}
```