Sistemi Operativi (M. Cesati)

Compito scritto del 6 settembre 2013

Nome: _____ **Cognome:** _____

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

Crediti da conseguire: 5 6 9

Scrivere i dati richiesti in stampatello. Al termine consegnare tutti i fogli. Tempo a disposizione: 2 ore.
Tenere presente che saranno valutati anche l'ordine e la chiarezza dell'esposizione.

Esercizio 1. Scrivere una applicazione C/POSIX multithread costituita da tre thread che implementano le seguenti procedure:

T_1) Il primo thread inserisce in un buffer circolare B1 i valori interi `x-(x*x*x)%8+1` per `x` uguale a 0, 1, 2, ecc. Il buffer B1 può contenere al massimo 1024 numeri interi, quindi se necessario il thread T_1 deve bloccare in attesa che venga liberato spazio in esso.

T_2) Il secondo thread inserisce in un buffer circolare B2 i valori interi `x+(x*x)%7` per `x` uguale a 0, 1, 2, ecc. Il buffer B2 può contenere al massimo 1024 numeri interi, quindi se necessario il thread T_2 deve bloccare in attesa che venga liberato spazio in esso.

T_3) Il terzo thread scandisce e confronta gli elementi contenuti nei due buffer B1 e B2 in modo sincrono (il primo elemento di B1 con il primo elemento di B2, il secondo con il secondo, ecc.). Se necessario T_3 deve bloccare in attesa che venga inserito un elemento in uno dei buffer. Se per un certo valore di `x` il valore risultante nei due buffer è identico, T_3 stampa tale valore in standard output. Dopo ogni confronto T_3 rimuove gli elementi corrispondenti dai due buffer.

I thread debbono poter eseguire il più possibile in parallelo tra loro, ma si deve evitare ogni possibile "race condition" dovuta agli accessi ai buffer circolari condivisi.

A titolo di esempio, i primi valori stampati dal thread T_3 dovrebbero essere

$$7, 9, 21, 23, 35, 37, 49, 49, 51, 63, 65, 77, 79, 91, 93, \ldots$$

corrispondenti ai valori di `x`

$$6, 8, 20, 22, 34, 36, 48, 49, 50, 62, 64, 76, 78, 90, 92, \ldots$$

Esercizio 2. In riferimento ad un generico sistema operativo moderno, si descrivano in modo esauriente i possibili stati di un processo e gli eventi che causano le transizioni tra tali stati.

Sistemi Operativi (M. Cesati)

Esempio di programmi del compito scritto del 6 settembre 2013

Esercizio 1 (prima soluzione)

Le strutture di dati principali del programma sono due buffer circolari, ciascuno dei quali è acceduto esclusivamente da due thread, uno in lettura ed uno in scrittura. Un buffer circolare acceduto unicamente da un lettore e da uno scrittore non presenta particolari problemi di "race condition". Pertanto, è possibile realizzare una soluzione che non fa uso di alcuna primitiva di sincronizzazione.

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo con la definizione della struttura di dati necessaria per implementare i due buffer circolari:

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

#define cbuf_size 1025
struct double_circular_buffer {
    int E1;
    int E2;
    int S;
    unsigned int buf1[cbuf_size];
    unsigned int buf2[cbuf_size];
};
struct double_circular_buffer db;
```

Si noti che la dimensione dei buffer è pari ad un elemento in più di quanto richiesto; infatti, per semplificare la logica di controllo e distinguere facilmente i casi "buffer pieno" e "buffer vuoto" dovrà sempre esserci almeno una posizione del buffer libera.

I campi `E1` e `E2` rappresentano la fine dei dati rispettivamente nel primo e nel secondo buffer. Il campo `S` rappresenta l'inizio dei dati in entrambi i buffer: infatti le letture sono realizzate dal thread T_3 e coinvolgono sempre la stessa posizione relativa nei due buffer.

Un'altra struttura di dati necessaria per il programma è il descrittore del lavoro che deve essere compiuto dai thread T_1 e T_2:

```
struct job_t {
    unsigned int *buf;
    int *pE, *pS;
    unsigned int (*expr)(unsigned int);
};
struct job_t job1, job2;
```

La struttura `job_t` contiene puntatori al buffer circolare su cui dovrà operare il thread; inoltre contiene il puntatore `expr` ad una funzione che implementa la diversa espressione matematica utilizzata dai thread T_1 e T_2.

La funzione `main()` deve creare i due thread T_1 e T_2, poi prosegue eseguendo il lavoro affidato al thread T_3:

```
int main(void)
{
    int s;
    pthread_t t;

    db.S = db.E1 = db.E2 = 0;

    /* Thread T1 acts on buffer b1 and expression exp1 */
    job1.buf = db.buf1;
    job1.pE = &db.E1;
    job1.pS = &db.S;
    job1.expr = expr1;
    s = pthread_create(&t, NULL, fill_buffer, &job1);
    if (s != 0) {
        fprintf(stderr, "Error in pthread_create\n");
        exit(EXIT_FAILURE);
    }

    /* Thread T2 acts on buffer b2 and expression exp2 */
    job2.buf = db.buf2;
    job2.pE = &db.E2;
    job2.pS = &db.S;
    job2.expr = expr2;
    s = pthread_create(&t, NULL, fill_buffer, &job2);
    if (s != 0) {
        fprintf(stderr, "Error in pthread_create\n");
        exit(EXIT_FAILURE);
    }

    /* The initial thread plays as T3 */
    scan_buffers(&db);
    return 0; /* never reached */
}
```

Le funzioni che implementano le espressioni matematiche dei thread T_1 e T_2 sono molto semplici:

```
unsigned int expr1(unsigned int x)
{
    return x-(x*x*x)%8+1;
}
unsigned int expr2(unsigned int x)
{
    return x+(x*x)%7;
}
```

La funzione `fill_buffer()` è comune ad entrambi i thread creati entro `main()` (T_1 e T_2). Ha il compito di riempire un buffer circolare con i dati calcolati per mezzo dell'espressione matematica indicata:

```
void *fill_buffer(void *p)
{
    struct job_t *job = (struct job_t *) p;
    unsigned int *buf = job->buf;
    int *pE = job->pE;
    volatile int *pS = job->pS;
    unsigned int x, v;

    for (x=0; ; ++x) {
        int nE = (*pE + 1) % cbuf_size;
        v = job->expr(x);
        /* controllare se buffer pieno */
        while (nE == *pS)
            ; /* busy wait */
        buf[*pE] = v;
        *pE = nE;
    }
    /* NOT REACHED */
}
```

Nel caso in cui il buffer circolare sia pieno (ossia vi sia una sola posizione libera), la funzione entra in un ciclo "busy wait" in cui attende che il thread di lettura renda libera una nuova posizione del buffer incrementando la variabile puntata da `job->pS`. A tale riguardo si noti come questo puntatore venga assegnato ad una variabile locale di tipo `volatile int *`. La parola chiave `volatile` informa il compilatore che il contenuto della cella di memoria referenziata dal puntatore può essere modificato da un altro flusso di esecuzione esterno alla funzione. In assenza di `volatile`, il compilatore potrebbe generare codice ottimizzato che non rileva accessi alla cella e quindi evita di rileggere ad ogni ciclo il valore nella cella di memoria, con conseguente funzionamento erroneo del programma.

Infine la funzione `scan_buffers()` implementa il lavoro affidato al thread T_3: legge gli elementi di entrambi i buffer, controllando i valori coincidenti, e contemporaneamente libera lo spazio occupato:

```
void scan_buffers(struct double_circular_buffer *db)
{
    volatile int *pE1 = &db->E1;
    volatile int *pE2 = &db->E2;
    for (;;) {
        int S = db->S;
        while (S == *pE1 || S == *pE2)
            ; /* busy wait: a circular buffer is empty */
        /* check if the two values are equal */
        if (db->buf1[S] == db->buf2[S])
            printf("%u\n", db->buf1[S]);
        /* remove the elements from the buffers */
        db->S = (S + 1) % cbuf_size;
    }
}
```

Anche in questo caso si invoca un "busy wait" nel caso in cui uno od entrambi i buffer circolari sia vuoto, e si utilizza la parola chiave `volatile` per accedere ai campi che sono aggiornati in modo asincrono dagli altri due thread del programma.

Esercizio 1 (seconda soluzione)

Una implementazione alternativa dell'esercizio consiste nell'evitare l'utilizzo dei "busy wait", facendo in modo di sospendere il thread in attesa che qualche elemento venga aggiunto o rimosso da un buffer circolare.

In pratica è possibile utilizzare ad esempio variabili condizione di tipo `pthread_cond_t`. Ciò implica anche utilizzare opportuni mutex per sincronizzare l'accesso alle variabili condizione. Il codice è pertanto più complesso della soluzione con "busy wait", ma anche considerevolmente più efficiente.

Il codice di questa implementazione alternativa, senza ulteriori commenti, è il seguente:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#define cbuf_size 1025
```

```
struct circular_buffer {
    pthread_mutex_t mtx;
    pthread_cond_t cnd_not_empty;
    pthread_cond_t cnd_not_full;
    int S;
    int E;
    unsigned int buf[cbuf_size];
};
struct circular_buffer b1, b2;

struct job_t {
    struct circular_buffer *buf;
    unsigned int (*expr)(unsigned int);
};
struct job_t job1, job2;

void scan_buffers(struct circular_buffer *b1,
                  struct circular_buffer *b2)
{
    for (;;) {
        if (pthread_mutex_lock(&b1->mtx) != 0) {
            perror("pthread_mutex_lock");
            exit(EXIT_FAILURE);
        }
        while (b1->S == b1->E) {
            /* circular buffer b1 empty */ ;
            if (pthread_cond_wait(&b1->cnd_not_empty,
                                  &b1->mtx) != 0) {
                perror("pthread_cond_wait");
                exit(EXIT_FAILURE);
            }
        }
        if (pthread_mutex_unlock(&b1->mtx) != 0) {
            perror("pthread_mutex_unlock");
            exit(EXIT_FAILURE);
        }
        if (pthread_mutex_lock(&b2->mtx) != 0) {
            perror("pthread_mutex_lock");
            exit(EXIT_FAILURE);
        }
        while (b2->S == b2->E) {
            /* circular buffer b2 empty */ ;
            if (pthread_cond_wait(&b2->cnd_not_empty,
                                  &b2->mtx) != 0) {
                perror("pthread_cond_wait");
                exit(EXIT_FAILURE);
            }
        }
        if (pthread_mutex_unlock(&b2->mtx) != 0) {
            perror("pthread_mutex_unlock");
            exit(EXIT_FAILURE);
        }
        /* Check if the two values are equal */
```

```
        if (b1->buf[b1->S] == b2->buf[b2->S])
            printf("%u\n", b1->buf[b1->S]);
        /* remove the elements from the buffers */
        b1->S = (b1->S + 1) % cbuf_size;
        b2->S = b1->S;
        if (pthread_cond_signal(&b1->cnd_not_full) != 0) {
            perror("pthread_cond_signal");
            exit(EXIT_FAILURE);
        }
        if (pthread_cond_signal(&b2->cnd_not_full) != 0) {
            perror("pthread_cond_signal");
            exit(EXIT_FAILURE);
        }
    }
}

void *fill_buffer(void *p)
{
    struct job_t *job = (struct job_t *) p;
    struct circular_buffer *cb = job->buf;
    unsigned int x, v;
    int nE;

    for (x=0; ; ++x) {
        v = job->expr(x);
        if (pthread_mutex_lock(&cb->mtx) != 0) {
            perror("pthread_mutex_lock");
            exit(EXIT_FAILURE);
        }
        nE = (cb->E + 1) % cbuf_size;
        /* controllare se buffer pieno */
        while (nE == cb->S) {
            /* buffer pieno */
            if (pthread_cond_wait(&cb->cnd_not_full,
                                  &cb->mtx) != 0) {
                perror("pthread_cond_wait");
                exit(EXIT_FAILURE);
            }
        }
        if (pthread_mutex_unlock(&cb->mtx) != 0) {
            perror("pthread_mutex_unlock");
            exit(EXIT_FAILURE);
        }
        cb->buf[cb->E] = v;
        cb->E = nE;
        if (pthread_cond_signal(&cb->cnd_not_empty) != 0) {
            perror("pthread_cond_signal");
            exit(EXIT_FAILURE);
        }
    }
    /* NOT REACHED */
}
```

```c
void initialize_circular_buffer(struct circular_buffer *cb)
{
    cb->S = cb->E = 0;
    if (pthread_mutex_init(&cb->mtx,NULL) != 0) {
        fprintf(stderr, "Error in pthread_mutex_init\n");
        exit(EXIT_FAILURE);
    }
    if (pthread_cond_init(&cb->cnd_not_empty, NULL) != 0 ||
        pthread_cond_init(&cb->cnd_not_full,  NULL) != 0) {
        fprintf(stderr, "Error in pthread_cond_init\n");
        exit(EXIT_FAILURE);
    }
}

unsigned int expr1(unsigned int x)
{
    return x-(x*x*x)%8+1;
}
unsigned int expr2(unsigned int x)
{
    return x+(x*x)%7;
}

int main(void)
{
    int s;
    pthread_t t;

    initialize_circular_buffer(&b1);
    initialize_circular_buffer(&b2);
    /* Thread T1 acts on buffer b1 and expression exp1 */
    job1.buf = &b1;
    job1.expr = expr1;
    s = pthread_create(&t, NULL, fill_buffer, &job1);
    if (s != 0) {
        fprintf(stderr, "Error in pthread_create\n");
        exit(EXIT_FAILURE);
    }
    /* Thread T2 acts on buffer b2 and expression exp2 */
    job2.buf = &b2;
    job2.expr = expr2;
    s = pthread_create(&t, NULL, fill_buffer, &job2);
    if (s != 0) {
        fprintf(stderr, "Error in pthread_create\n");
        exit(EXIT_FAILURE);
    }
    /* The initial thread plays as T3 */
    scan_buffers(&b1, &b2);
    return 0; /* never reached */
}
```