Sistemi Operativi (M. Cesati)

Compito scritto del 16 febbraio 2016

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 due thread T_1 e T_2. Il thread T_1 legge il contenuto di un file il cui nome è passato sulla linea comando. Il file contiene un numero finito e variabile di strutture del seguente tipo:

```
struct record {
    int key;
    char name[32];
};
```

Il formato dei dati è nativo per l'architettura del calcolatore; in altre parole, la sequenza di byte nel file corrisponde alla rappresentazione in RAM delle varie strutture record, una dopo l'altra. I record entro il file non sono ordinati in alcun modo particolare.

Il thread T_1 legge dal file un record alla volta e lo inserisce in un buffer circolare. Il thread T_2 legge i record dal buffer circolare e li inserisce in una lista ordinata in base al valore del campo `key`.

Al termine, il thread T_1 scrive in standard output i record ordinati nella lista con il seguente formato: una riga di testo per ciascun differente valore di `key` contenente il valore di `key` e tutte le stringhe `name` associate a quel valore.

Si assuma che il campo `name` di ogni record contenga sempre una stringa di lunghezza variabile terminata da `'\0'`.

Si considerino gli eventuali problemi legati alle race condition ed alla sincronizzazione tra i due thread.

Esercizio 2

Si descrivano in modo esauriente il ruolo, le differenze e le relazioni esistenti tra i concetti di "chiamata di sistema", "API" e "funzioni di libreria".

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 16 febbraio 2016

Esercizio 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo col definire le strutture di dati principali utilizzate dal programma:

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

struct record {
    int key;
    char name[32];
};

#define ringbuf_size 16

struct ringbuf {
    int E, S;
    pthread_mutex_t mtx;
    pthread_cond_t no_full, no_empty;
    struct record rec[ringbuf_size];
};

struct list_t {
    struct record rec;
    struct list_t *next;
};
```

La struttura `record` è come definita nel testo dell'esercizio. La struttura `ringbuf` implementa il buffer circolare: i campi `E` e `S` indicano la prima locazione vuota e la prima locazione ancora da leggere, rispettivamente. Il mutex `mtx` e le variabili condizioni `no_full` e `no_empty` consentono di sincronizzare gli accessi al buffer circolare da parte dei due thread.

La funzione `main()` è la seguente:

```
int main(int argc, char *argv[])
{
    struct ringbuf rb;
    struct list_t *lh = NULL;
    struct thread_data td = { &rb, &lh };
```

```
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return EXIT_FAILURE;
    }
    initialize_ringbuf(&rb);
    spawn_thread(&td);
    read_file(argv[1], &rb);
    wait_until_rb_empty(&rb);
    print_sorted_list(lh);
    return EXIT_SUCCESS;
}
```

Il thread T_1 è il thread iniziale dell'applicazione, ossia quello che esegue la funzione `main()`. Poiché l'applicazione deve terminare quando si conclude l'esecuzione del thread T_1, è possibile allocare le strutture di dati condivise dai thread sullo stack di T_1, ossia come variabili automatiche di `main()`.

La funzione inizializza il buffer circolare invocando `initialize_ringbuf()`:

```
void initialize_ringbuf(struct ringbuf *r)
{
    r->E = r->S = 0;
    chk_pthread_err(pthread_mutex_init(&r->mtx, NULL));
    chk_pthread_err(pthread_cond_init(&r->no_full, NULL));
    chk_pthread_err(pthread_cond_init(&r->no_empty, NULL));
}
```

La funzione `chk_pthread_err()` verifica che la funzione "pthread" non restituisca un errore:

```
void chk_pthread_err(int rc)
{
    if (rc == 0)
        return;
    errno = rc;
    perror(NULL);
    exit(EXIT_FAILURE);
}
```

Si ricordi che le API dei POSIX thread restituiscono direttamente il codice di errore che nelle altre API di sistema viene scritto nella variabile `errno`.

Successivamente `main()` invoca `spawn_thread()` per creare il thread T_2. La struttura di dati `thread_data` serve a passare a T_2 gli indirizzi del buffer circolare `rb` e della testa della lista ordinata `lh`.

```
struct thread_data {
    struct ringbuf *r;
    struct list_t **lh;
};

void spawn_thread(struct thread_data *ptd)
{
    pthread_t tid;
    chk_pthread_err(pthread_create(&tid,NULL,T2_work,ptd));
}
```

Esamineremo in seguito la funzione `T2_work()` eseguita dal thread T_2.

La funzione `main()` comincia ad eseguire il lavoro del thread T_1 invocando la funzione `read_file()`:

```
void read_file(const char *fname, struct ringbuf *rb)
{
    struct record rec;
    FILE *f = fopen(fname, "r");
    if (f == NULL) {
        perror(fname);
        exit(EXIT_FAILURE);
    }
    while (!feof(f)) {
        if (get_record(f, &rec))
            ringbuf_in(&rec, rb);
        if (ferror(f)) {
            perror(fname);
            exit(EXIT_FAILURE);
        }
    }
}
```

Dopo aver aperto il file, `read_file()` legge un record alla volta invocando `get_record()`:

```
int get_record(FILE *f, struct record *buf)
{
    size_t rc = fread(buf, sizeof(struct record), 1, f);
    return (rc == 1);
}
```

Il record viene inserito nel buffer circolare con la funzione `ringbuf_in()`:

```
void ringbuf_in(struct record *rec, struct ringbuf *rb)
{
    int nE;
    chk_pthread_err(pthread_mutex_lock(&rb->mtx));
    nE = wait_until_rb_not_full(rb);
    rb->rec[rb->E] = *rec;
    rb->E = nE;
    chk_pthread_err(pthread_cond_signal(&rb->no_empty));
    chk_pthread_err(pthread_mutex_unlock(&rb->mtx));
}
```

Dopo aver acquisito il mutex, la funzione controlla che il buffer non sia completamente pieno, inserisce il nuovo record nel buffer, e segnala al thread T_2 che il buffer è certamente non più vuoto. Infine viene rilasciato il mutex. La funzione `wait_until_rb_not_full()` è la seguente:

```
int wait_until_rb_not_full(struct ringbuf *rb)
{
    int nE = (rb->E + 1) % ringbuf_size;
    while (nE == rb->S)
        chk_pthread_err(pthread_cond_wait(&rb->no_full,
                                          &rb->mtx));
    return nE;
}
```

Dopo aver terminato la lettura del file il thread T_1 deve stampare in standard output il contenuto della lista ordinata costruita dal thread T_2. Prima però T_1 deve attendere che T_2 estragga dal buffer circolare i restanti record e li inserisca nella lista. A tale scopo, attende che il buffer circolare si svuoti completamente invocando la funzione `wait_until_rb_empty()`:

```
void wait_until_rb_empty(struct ringbuf *rb)
{
    chk_pthread_err(pthread_mutex_lock(&rb->mtx));
    while (rb->E != rb->S)
        chk_pthread_err(pthread_cond_wait(&rb->no_full,
                                          &rb->mtx));
    chk_pthread_err(pthread_mutex_unlock(&rb->mtx));
}
```

Infine `main()` invoca la funzione `print_sorted_list()` per stampare in standard output il contenuto della lista ordinata col formato richiesto dal testo dell'esercizio:

```
void print_sorted_list(struct list_t *h)
{
    while (h != NULL) {
        int key = h->rec.key;
        printf("%d: %s", key, h->rec.name);
        for (;;) {
            h = h->next;
            if (h == NULL || key != h->rec.key)
                break;
            printf(" %s", h->rec.name);
        }
        printf("\n");
    }
}
```

Il thread T_2 inizia l'esecuzione dalla funzione `T2_work()`:

```
void * T2_work(void *arg)
{
    struct thread_data *td = (struct thread_data *) arg;
    struct record rec;
    for (;;) {
        ringbuf_out(&rec, td->r);
        insert_in_ordered_list(&rec, td->lh);
        chk_pthread_err(pthread_cond_signal(&rb->no_full));
    }
}
```

Il thread T_2 esegue un ciclo senza fine trasferendo record dal buffer circolare alla lista ordinata. La funzione `ringbuf_out()` estrae un elemento dal buffer circolare:

```
void ringbuf_out(struct record *rec, struct ringbuf *rb)
{
    chk_pthread_err(pthread_mutex_lock(&rb->mtx));
    wait_until_rb_not_empty(rb);
    *rec = rb->rec[rb->S];
    rb->S = (rb->S + 1) % ringbuf_size;
    chk_pthread_err(pthread_mutex_unlock(&rb->mtx));
}
```

La funzione è simile a `ringbuf_in()`, ed invoca `wait_until_rb_not_empty()` per verificare che il buffer circolare non sia vuoto:

```
void wait_until_rb_not_empty(struct ringbuf *rb)
{
    while (rb->E == rb->S)
        chk_pthread_err(pthread_cond_wait(&rb->no_empty,
                                          &rb->mtx));
}
```

Per inserire l'elemento nella lista ordinata `T2_work()` invoca la funzione `insert_in_ordered_list()`:

```
void insert_in_ordered_list(struct record *rec,
                            struct list_t **phead)
{
    int key = rec->key;
    struct list_t *new = malloc(sizeof(struct list_t));
    if (new == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    new->rec = *rec;
    new->next = NULL;
    while (*phead != NULL && key >= (*phead)->rec.key)
        phead = &(*phead)->next;
    insert_after_node(new, phead);
}
```

È necessario allocare un nuovo elemento della lista ordinata, e percorrere la lista fino a determinare la posizione in base al valore del campo `key`. Come al solito, la funzione di manipolazione della lista `insert_after_node()` è basata sull'indirizzo del campo `next` dei vari elementi in modo da gestire in modo uniforme anche il caso della lista vuota:

```
void insert_after_node(struct list_t *new,
                       struct list_t **pnext)
{
    new->next = *pnext;
    *pnext = new;
}
```

Infine, la funzione `T2_work()` segnala al thread T_1 che ha estratto un elemento dal buffer circolare, e quindi questo è certamente non più vuoto. Si noti però che T_2 ritarda questa "segnalazione" fino all'avvenuto inserimento dell'elemento nella lista ordinata. In caso contrario si avrebbe una race condition: il thread T_1 potrebbe stampare la lista ordinata prima che il thread T_2 abbia inserito in essa l'ultimo elemento estratto.