Sistemi Operativi (M. Cesati)

Compito scritto del 16 luglio 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 un thread "master" e 26 thread "slave". Il thread "master" legge dallo standard input un testo, ossia una sequenza di parole separate da caratteri non alfabetici (ai fini dell'esercizio, qualunque carattere non alfabetico come '`1`', ' ', '`\n`' è considerato un separatore di parola). Ciascun thread "slave" è associato ad una differente lettera dell'alfabeto inglese, e gestisce una propria lista dinamica che, alla fine, conterrà tutte le parole del testo che iniziano con quella determinata lettera. La lista dovrà essere ordinata in modo lessicografico (funzione di libreria `strcmp`). Al termine della lettura del testo in standard input, ciascun thread "slave", nell'ordine appropriato, dovrà scrivere in standard output la propria lista di parole. Si ponga attenzione ai problemi di sincronizzazione e race condition dei thread.

Esercizio 2

Si descrivano in modo chiaro ed esauriente i principi di funzionamento degli algoritmi di tipo Round Robin utilizzati per la schedulazione dei processi nei moderni sistemi operativi. Discutere anche come vengono gestiti processi di diversa topologia (ad esempio, batch o interattivi) e processi aventi differenti priorità.

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 16 luglio 2015

Risolviamo l'esercizio con un approccio top-down. Il thread "master", ossia la funzione `main()`, deve creare i 26 thread "slave", poi leggere le parole dallo standard input e passarle al thread "slave" appropriato. Una volta terminato di leggere dallo standard input il thread "master" deve far stampare le liste ordinate a ciascun thread, e terminare l'esecuzione.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
int main()
{
    struct slave_thread_t slaves[26];
    create_threads(slaves);
    read_words_from_stdin(slaves);
    print_word_lists(slaves);
    pthread_exit(0);
}
```

Ciascuna struttura di tipo `slave_thread_t` include tutte le informazioni necessarie per svolgere l'attività di ciascun thread "slave" e assicurare la sincronizzazione con il thread "master".

```
struct slave_thread_t {
    pthread_t tid;
    pthread_mutex_t mtx;
    pthread_cond_t got_new_word;
    int turn;
    char *word;
    struct node_t *list_head;
};
```

Il campo `word` contiene l'indirizzo della stringa con la parola da inserire. Per la sincronizzazione tra slave e master si prevede di utilizzare un mutex `mtx`, una variabile condizione `got_new_word`, che serve a segnalare allo slave che il master ha una nuova parola da far inserire in lista, ed al master che lo slave ha terminato l'inserimento della parola; inoltre `turn` è impostato a 0 dallo slave se esso è in attesa di una parola, ed è impostato ad 1 dal master se esso ha inviato una parola. Il mutex, oltre a proteggere l'accesso ai campi della struttura `slave_thread_t`, serve anche ad evitare che il master possa modificare

il campo `word` mentre lo slave è ancora occupato a cercare la posizione della precedente parola nella lista. La testa della lista di parole è memorizzato nel campo `list_head` di tipo puntatore alla struttura `node_t`, che include l'indirizzo della parola e l'indirizzo dell'elemento successivo in lista:

```
struct node_t {
    char *word;
    struct node_t *next;
};
```

Per creare i thread slave definiamo la funzione `create_threads()`, che inizializza ciascun elemento di tipo `slave_thread_t` prima di invocare `pthread_create()`:

```
void create_threads(struct slave_thread_t *slaves)
{
    int k;
    struct slave_thread_t *s = slaves;
    for (k=0; k<26; ++k, ++s) {
        s->word = NULL;
        s->list_head = NULL;
        s->turn = 1;
        if (pthread_mutex_init(&s->mtx, NULL))
            error_exit("Error in pthread_mutex_init()\n");
        if (pthread_cond_init(&s->got_new_word, NULL))
            error_exit("Error in pthread_cond_init()\n");
        if (pthread_create(&s->tid, NULL, slave_fn, s))
            error_exit("Error in pthread_create()\n");
    }
}
```

La funzione `error_exit()` viene invocata in caso di errore restituito da una API e termina l'esecuzione del programma:

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

Per leggere dallo standard input il thread "master" invoca la funzione `read_words_from_stdin()`:

```
void read_words_from_stdin(struct slave_thread_t *slaves)
{
    char *word;
    int  slave;
    for (;;) {
        word = read_word();
        if (word == NULL)
            break;
        slave = to_lower(word[0]) - 'a';
        give_word_to_slave(word, &slaves[slave]);
    }
}
```

La funzione legge una parola dallo standard input. In caso di EOF sullo standard input la funzione termina. Altrimenti si considera il primo carattere della parola, lo si converte in minuscolo nel caso fosse maiuscolo, si calcola l'indice sequenziale del thread "slave" che deve gestire la parola, e si passa la stringa a questo thread.

Per leggere una parola dallo standard input si definisce la funzione `read_word()`. Analizziamola un po' per volta:

```
char * read_word(void)
{
    char *w;
    int c, k, maxlen = 32;
    w = malloc(sizeof(char)*maxlen);
    if (w == NULL)
        error_exit("Memory allocation error\n");
```

Si inizia allocando spazio per una parola lunga al più 32 caratteri.

```
    do {
        c = getchar();
        if (c == EOF)
            return NULL;
    } while (!is_a_letter(c));
```

Leggiamo dallo standard input fino a che non si ottiene un carattere alfabetico A-Z, sia maiuscolo che minuscolo. Nel caso in cui lo standard input finisca, la funziona termina restituendo `NULL`.

```
    w[0] = c;
    k = 1;
    for (;;) {
        for (; k<maxlen; ++k) {
            c = getchar();
            if (!is_a_letter(c)) {
                w[k] = '\0';
                break;
            }
            w[k] = c;
        }
        if (k < maxlen)
            break;
        maxlen += maxlen;
        w = realloc(w, sizeof(char)*maxlen);
        if (w == NULL)
            error_exit("Memory reallocation error\n");
    }
```

In `c` vi è il primo carattere di una nuova parola: salviamolo nella prima posizione del buffer `w`, poi entriamo in un ciclo per leggere tutti gli altri caratteri. In effetti il ciclo è doppio perché viene gestito anche il caso in cui la parola abbia lunghezza maggiore di 32 caratteri, raddoppiando ogni volta la dimensione del buffer. Si noti che nel caso in cui lo standard input termini mentra si sta leggendo una parola, la parola stessa viene comunque restituita al programma chiamante. La condizione EOF verrà riconosciuta come tale all'invocazione successiva.

```
    w = realloc(w, sizeof(char)*(strlen(w)+1));
    if (w == NULL)
        error_exit("Memory reallocation error\n");
    return w;
}
```

Infine la funzione aggiusta la dimensione del buffer alla reale dimensione della parola letta, e si restituisce l'indirizzo iniziale del buffer.

Per riconoscere se il carattere letto è una lettera dell'alfabeto utilizziamo la funzione `is_a_letter()`:

```
int is_a_letter(int c)
{
    return ((c >= 'a' && c <= 'z') ||
            (c >= 'A' && c <= 'Z'));
}
```

Anche la funzione per convertire una lettera da maiuscola a minuscola è semplicissima. (In effetti la libreria C mette a disposizione la funzione `tolower()` per questa operazione.)

```
int to_lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        c += 'a' - 'A';
    return c;
}
```

Siamo giunti ora ad analizzare la funzione `give_word_to_slave()` utilizzata dal master per segnalare al thread "slave" la nuova parola da inserire in lista:

```
void give_word_to_slave(char *word,
                        struct slave_thread_t *slave)
{
    lock(&slave->mtx);
    if (slave->turn != 0)
        wait(&slave->got_new_word, &slave->mtx);
    slave->turn = 1;
    slave->word = word;
    awake(&slave->got_new_word);
    unlock(&slave->mtx);
}
```

Essenzialmente la funzione acquisisce il mutex dello slave, controlla il valore di `turn` bloccando il master se necessario, aggiorna il campo `word`, risveglia il thread, e rilascia il mutex. Definiamo le funzioni `lock()` e `unlock()` per operare sul mutex:

```
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");
}
```

Invece per operare sulla variabile condizione definiamo le funzioni `awake()` e `wait()`:

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

Quando la lettura dallo standard input è terminata, il thread "master" forza i thread "slave" a stampare in standard output ciascuno la propria lista di parole, in ordine. I thread "slave" riconoscono questa condizione determinando che il campo `word` della struttura è un puntatore nullo:

```
void print_word_lists(struct slave_thread_t *slaves)
{
    int k;
    for (k=0; k<26; ++k, slaves++) {
        give_word_to_slave(NULL, slaves);
        pthread_join(slaves->tid, NULL);
    }
}
```

I thread "slave" terminano l'esecuzione dopo aver stampato la propria lista, quindi il master può utilizzare `pthread_join()` per sequenzializzare le stampe delle liste in standard output.

Abbiamo terminato di analizzare il codice eseguito dal thread "master". Il codice eseguito dal thread "slave" prende l'avvio dalla funzione `slave_fn()` passata a `pthread_create()`:

```
void *slave_fn(void *v)
{
    struct slave_thread_t *s = (struct slave_thread_t *) v;
    lock(&s->mtx);
    for (;;) {
        awake(&s->got_new_word);
        s->turn = 0;
        wait(&s->got_new_word, &s->mtx);
        if (s->word == NULL)
            break;
        insert_word(s->word, &s->list_head);
    }
    print_list(s->list_head);
    unlock(&s->mtx);
```

```
    pthread_exit(0);
}
```

La funzione acquisce il proprio mutex, sblocca il master, e poi si mette in attesa sulla variabile condizione. Come sappiamo, durante l'attesa il mutex viene rilasciato, ed automaticamente riacquisito quando la condizione si verifica. Perciò il ciclo continua con il mutex acquisito: viene controllato se `word` è un puntatore nullo. Se non lo è, la parola viene inserita nella lista. Altrimenti, la lista viene stampata in standard output, il mutex viene rilasciato, ed il thread termina.

La funzione `insert_word()` inserisce la parola nella corretta posizione della lista:

```
void insert_word(char *word, struct node_t **pnode)
{
    int rc;
    while (*pnode != NULL) {
        rc = strcmp(word, (*pnode)->word);
        if (rc <= 0)
            break;
        pnode = &(*pnode)->next;
    }
    insert_node(word, pnode);
}
```

Per controllare l'ordinamento lessicografico tra due parole utilizziamo come suggerito nel testo la funzione di libreria `strcmp()`, che restituisce un valore minore di zero, uguale a zero, oppure maggiore di zero se la prima stringa è rispettivamente minore, uguale o maggiore della seconda stringa.

Una volta che è stata determinata la posizione giusta nella lista si invoca la funzione `insert_node()` per allocare ed inserire un nuovo elemento di lista:

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

Infine, per stampare la lista di parole il thread “slave” utilizza la funzione `print_list()`:

```
void print_list(struct node_t *node)
{
    while (node != NULL) {
        printf("%s\n", node->word);
        node = node->next;
    }
}
```