Sistemi Operativi **(M. Cesati)**

Compito scritto del 8 settembre 2017

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.

Esercizio

Scrivere una applicazione C/POSIX multithread che riceve sulla linea comando un numero arbitrario n di stringhe $s_1, \ldots, s_n$. L'applicazione è costituita da $n + 1$ thread concorrenti.

Il primo thread T_0 dell'applicazione legge dallo standard input righe di testo (terminate dal carattere "`\n`") ed inserisce i puntatori a tali righe di testo in un buffer circolare avente come capacità massima 100 puntatori.

Ciascuno degli altri thread T_i $(1 \leq i \leq n)$ esamina tutte le righe di testo inserite nel buffer circolare per ricercare la stringa s_i. Se la stringa s_i è presente nella riga, il thread T_i stampa l'intera riga sullo standard output.

Le righe di testo devono essere memorizzate fino al momento in cui sono state esaminate da tutti i thread $T_1, \ldots T_n$; successivamente devono essere eliminate, in modo da liberare le posizioni nel buffer circolare e recuperare la memoria occupata. Tutti i thread devono lavorare in modo concorrente, ma si devono curare gli aspetti di sincronizzazione. Ad esempio, si deve gestire la situazione in cui il buffer circolare è pieno (thread T_0 bloccato), oppure vuoto (thread $T_1, \ldots, T_n$ bloccati), ed evitare le race condition sulle strutture di dati condivise.

Sistemi Operativi **(M. Cesati)**

Esempio di programma del compito scritto del 8 settembre 2017

Seguendo l'approccio "top-down", cominciamo a definire le strutture di dati principali del programma. Il buffer circolare è realizzato dalla struttura globale `cb`:

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

#define CIRC_BUF_SIZE (100+1)

struct {
    char *lines[CIRC_BUF_SIZE];
    unsigned int counts[CIRC_BUF_SIZE];
    int write_idx;
    pthread_mutex_t mtx;
    pthread_cond_t newentry;
    pthread_cond_t getfree;
} cb;
```

Per semplificare la gestione del buffer circolare lasceremo sempre libera almeno una posizione, quindi il buffer conterrà il numero di posizioni richieste dal testo dell'esercizio più uno. Il vettore `lines` memorizza i puntatori alle righe di testo, mentre il vettore `counts` contiene, per ciascuna posizione, il numero di thread che debbono ancora cercare la propria parola nella riga corrispondente; di conseguenza, il valore zero può essere utilizzato per indicare che la posizione del buffer è libera. Il campo `write_idx` rappresenta la posizione in cui il thread T_0 memorizzerà la successiva riga letta dallo standard input. Il campo `mtx` serve a serializzare l'accesso agli altri campi della struttura. Infine le variabili condizione `newentry` e `getfree` servono a segnalare, rispettivamente, che T_0 ha inserito una nuova riga e che una posizione nel buffer è stata liberata.

La funzione `main()` al solito è molto semplice, ed utilizza `abort_on_error()` per gestire l'uscita anticipata dal programma in caso di errori:

```
#define abort_on_error(cond, msg) do { \
    if (cond) { \
        fprintf(stderr,
                "%s (errno=%d [%m])\n ", msg, errno); \
        exit(EXIT_FAILURE); \
    } \
} while (0)
```

```
int main(int argc, char *argv[])
{
    abort_on_error(argc<2,
        "Wrong number of command line arguments");
    initialize_circ_buf();
    create_threads(argc-1, argv+1);
    read_lines_from_stdin(argc-1);
    finish();
}
```

Dopo aver controllato che sia stato passato almeno un argomento sulla linea comando, `main()` invoca `initialize_circ_buf()` per inizializzare la variabile globale `cb`:

```
#define abort_on_pthread_error(cond, msg) \
    abort_on_error((errno=cond), msg)

void initialize_circ_buf(void)
{
    memset(&cb, 0, sizeof(cb));
    abort_on_pthread_error(
            pthread_mutex_init(&cb.mtx, NULL),
            "Cannot initialize the mutex");
    abort_on_pthread_error(
            pthread_cond_init(&cb.newentry, NULL),
            "Cannot initialize the condition");
    abort_on_pthread_error(
            pthread_cond_init(&cb.getfree, NULL),
            "Cannot initialize the condition");
}
```

Tutti i campi della struttura sono inizializzati a zero. Poi vengono inizializzate le variabili specifiche per la gestione dei POSIX thread. La macro `abort_on_pthread_error()` semplifica la gestione degli errori impostando la variabile `errno` e terminando il programma se il valore assegnato è diverso da zero.

La funzione `main()` continua invocando `create_threads()`. Il primo argomento è il numero di parole passate sulla linea comando, e quindi è il numero di thread da creare. Il secondo argomento è il vettore di puntatori alle parole stesse.

```
void create_threads(int n, char *argv[])
{
    int i, rc;
    for (i=0; i<n; ++i) {
        pthread_t tid;
        char *p = strdup(argv[i]);
```

```
        abort_on_error(!p, "String duplication failure");
        rc = pthread_create(&tid, NULL, thread_search, p);
        abort_on_pthread_error(rc,
                               "Thread creation failure");
    }
}
```

La funzione `strdup()` copia una stringa in memoria allocata dinamicamente. Ciascun thread riceve come parametro l'indirizzo della nuova stringa così allocata.

Tralasciamo per il momento il lavoro svolto dai thread $T_1, \ldots, T_n$ e continuiamo l'esame di `main()`. Viene invocata la funzione `read_lines_from_stdin()`, passando come argomento il numero totale di parole da ricercare. Perciò, T_0 è realizzato dal thread originale della applicazione:

```
void read_lines_from_stdin(int n)
{
    char *p, locbuf[1024];
    for (;;) {
        p = fgets(locbuf, 1024, stdin);
        if (p == NULL) {
            abort_on_error(!feof(stdin),
                           "Error reading from stdin");
            return;
        }
        p = strdup(locbuf);
        insert_line_in_cb(p, n);
    }
}
```

La funzione legge una riga di testo terminata da "`\n`" dallo standard input in un buffer locale. Nel caso in cui la lettura fallisca, si controlla se lo standard input sia terminato con `feof()`. In questo caso la funzione ritorna, altrimenti il programma termina prematuramente.

Se invece la riga è stata letta correttamente, essa viene duplicata con `strdup` (in alternativa per duplicare la riga si possono utilizzare opportunamente le funzioni `strlen()`, `malloc()` e `strcpy()`).

Infine `read_lines_from_stdin()` invoca `insert_line_in_cb()` per inserire la riga nel buffer circolare:

```
void insert_line_in_cb(char *line, int n)
{
    int rc, widx;
    rc = pthread_mutex_lock(&cb.mtx);
    abort_on_pthread_error(rc, "Cannot lock the mutex");
```

```
    widx = cb.write_idx;
    while (cb.counts[widx] > 0) {
        rc = pthread_cond_wait(&cb.getfree, &cb.mtx);
        abort_on_pthread_error(rc,
                               "Cannot wait for condition");
    }
```

Per prima cosa la funzione acquisisce il mutex che protegge i campi del buffer circolare. Poi legge la posizione della successiva posizione in cui scrivere (`cb.write_idx`). Se per tale posizione il corrispondente valore nel vettore `cb.counts` è positivo, la posizione non è libera; di conseguenza, T_0 si pone in attesa che la variabile condizione `cb.getfree` segnali l'avvenuta liberazione di una posizione nel buffer, e riesegue il controllo.

```
    cb.lines[widx] = line;
    cb.counts[widx] = n;
    rc = pthread_cond_broadcast(&cb.newentry);
    abort_on_pthread_error(rc,
                           "Cannot broadcast the condition");
```

`insert_line_in_cb()` poi aggiorna il vettore `cb.lines()` con l'indirizzo della riga, ed il vettore `cb.counts` con il numero di thread che dovranno cercare la propria parola nella riga. Successivamente, la funzione segnala a tutti gli altri thread che una nuova posizione nel buffer è stata riempita utilizzando la variabile condizione `cb.newentry`.

```
    ++widx;
    if (widx == CIRC_BUF_SIZE)
        widx = 0;
    while (cb.counts[widx] > 0) {
        rc = pthread_cond_wait(&cb.getfree, &cb.mtx);
        abort_on_pthread_error(rc,
                               "Cannot wait for condition");
    }
    cb.write_idx = widx;
    rc = pthread_mutex_unlock(&cb.mtx);
    abort_on_pthread_error(rc, "Cannot unlock the mutex");
}
```

Ora è necessario aggiornare l'indice contenente la successiva posizione del buffer da scrivere. Dopo aver incrementato l'indice precedente (modulo la dimensione del buffer), la funzione attende che il corrispondente valore nel vettore `cb.counts` si annulli. In tal modo, in ogni istante una posizione del buffer circolare rimarrà sempre libera, e la condizione "buffer vuoto" non potrà essere fraintesa con quella di "buffer pieno". Per l'attesa viene utilizzata la variabile

condizione `cb.getfree`. Infine il campo `cb.write_idx` viene aggiornato ed il mutex rilasciato.

Analizziamo ora il lavoro svolto dai thread $T_1, \ldots, T_n$ implementato dalla funzione `thread_search()`:

```
int master_thread_finished = 0;

void *thread_search(void *arg)
{
    char *r, *line, *word = (char *) arg;
    int rc, read_idx = 0;
    rc = pthread_mutex_lock(&cb.mtx);
    abort_on_pthread_error(rc, "Cannot lock the mutex");
    for (;;) {
        while (cb.write_idx == read_idx) {
            if (master_thread_finished) {
                rc = pthread_mutex_unlock(&cb.mtx);
                abort_on_pthread_error(rc,
                                "Cannot unlock the mutex");
                return NULL;
            }
            rc = pthread_cond_wait(&cb.newentry, &cb.mtx);
            abort_on_pthread_error(rc,
                            "Cannot wait for condition");
        }
        rc = pthread_mutex_unlock(&cb.mtx);
        abort_on_pthread_error(rc,
                            "Cannot unlock the mutex");
```

All'inizio di ogni iterazione del ciclo `for` il thread ha già acquisito il mutex. Per verificare che vi sia qualcosa da leggere il thread confronta l'indice di scrittura globale `cb.write_idx` con l'indice di lettura locale del thread (variabile `read_idx`). Se le due variabili hanno lo stesso valore, il buffer non contiene alcun elemento che il thread non abbia già esaminato. Pertanto il thread aspetta che T_0 scriva un nuovo elemento per mezzo della variabile condizione `cb.newentry`.

La variabile globale `master_thread_finished` è impostata ad uno da T_0 subito prima che esso termini. Controllare il valore della variabile prima di bloccare permette di far terminare tutti i thread dell'applicazione in modo ordinato dopo che ciascuno di essi ha finito di esaminare tutte le righe rimaste nel buffer circolare.

```
        line = cb.lines[read_idx];
        r = strstr(line, word);
        if (r != NULL)
            printf("[%s]: %s", word, line);
```

Dopo aver determinato una posizione del buffer con una riga ancora da analizzare, `thread_search()` controlla la presenza della parola in `word` utilizzando la funzione di libreria `strstr()`. Tale funzione restituisce l'indirizzo iniziale della sottostringa, se essa è presente, oppure il valore `NULL`. In caso di successo, l'intera riga viene scritta sullo standard output.

```
        rc = pthread_mutex_lock(&cb.mtx);
        abort_on_pthread_error(rc,"Cannot lock the mutex");
        --cb.counts[read_idx];
        if (cb.counts[read_idx] == 0) {
            free(line);
            cb.lines[read_idx] = NULL;
            rc = pthread_cond_signal(&cb.getfree);
            abort_on_pthread_error(rc,
                                "Cannot signal the condition");
        }
        read_idx++;
        if (read_idx == CIRC_BUF_SIZE)
            read_idx = 0;
    }
}
```

Poi è necessario decrementare il contatore corrispondente alla riga appena esaminata, naturalmente dopo aver acquisito nuovamente il mutex per evitare le race condition dovute agli accessi concorrenti da parte degli altri thread. Se il contatore si azzera, la riga è stata esaminata da tutti i thread, quindi lo spazio occupato dalla stringa è recuperato tramite `free()` e viene segnalato l'evento tramite la variabile condizione `cb.getfree`. Infine viene aggiornato l'indice locale della posizione da leggere nel buffer circolare.

Quando il thread T_0 termina la lettura dello standard input, `main()` invoca la funzione `finish()`:

```
void finish(void)
{
    master_thread_finished = 1;
    abort_on_pthread_error(
            pthread_cond_broadcast(&cb.newentry),
            "Cannot broadcast the condition");
    pthread_exit(NULL);
}
```

La funzione imposta ad uno `master_thread_finished` e sblocca ogni thread eventualmente in attesa sulla variabile condizione `cb.newentry`. Infine il thread T_0 termina.