Sistemi Operativi (M. Cesati)

Compito scritto del 13 luglio 2016 (Traccia A)

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 multithread C/POSIX per una architettura di tipo "little-endian" a 32 bit. Sulla riga comando dell'applicazione vengono passati un numero arbitrario n di valori interi $v_1, \ldots, v_n$ in formato testo decimale. L'applicazione è costituita da $n+1$ thread. Il primo thread T_0 legge valori numerici positivi o nulli codificati entro un file chiamato `input.dat` in formato da 6 byte "little-endian". Il thread T_0 pone i valori letti dal file in un buffer circolare (di dimensione pari ad una pagina di memoria). Ciascuno dei restanti thread T_i (con $1 \leq i \leq n$) cerca all'interno del buffer circolare le occorrenze del valore v_i. Al termine, ciascuno dei thread T_i ($1 \leq i \leq n$) scrive in standard output il numero di occorrenze del valore v_i. Si presti attenzione alla sincronizzazione tra i processi ed alla gestione del buffer circolare.

Esercizio 2

Si descrivano lo scopo, il principio di funzionamento ed alcune possibili implementazioni della "allocazione indicizzata" utilizzata nei filesystem dei moderni sistemi operativi.

Sistemi Operativi **(M. Cesati)**

Compito scritto del 13 luglio 2016 (Traccia B)

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 multithread C/POSIX per una architettura di tipo "little-endian" a 32 bit. Sulla riga comando dell'applicazione vengono passati un numero arbitrario n di valori interi $v_1, \ldots, v_n$ in formato testo decimale. L'applicazione è costituita da $n + 1$ thread. Il primo thread T_0 legge i valori numerici codificati entro un file chiamato `input.dat` in formato da 4 byte "big-endian". Il thread T_0 pone i valori letti dal file in un buffer circolare (di dimensione pari ad una pagina di memoria). Ciascuno dei restanti thread T_i (con $1 \leq i \leq n$) cerca all'interno del buffer circolare le occorrenze del valore v_i. Al termine, ciascuno dei thread T_i ($1 \leq i \leq n$) scrive in standard output il numero di occorrenze trovate. Si presti attenzione alla sincronizzazione tra i processi ed alla gestione del buffer circolare.

Esercizio 2

Si descrivano lo scopo, il principio di funzionamento ed una possibile implementazione della primitiva "semaforo" utilizzata nel kernel di un moderno sistema operativo.

Sistemi Operativi (M. Cesati)

Esempi dei programmi del compito scritto del 13 luglio 2016

Esercizio 1, traccia A

La difficoltà principale dell'esercizio consiste nel coordinare l'accesso dei vari thread al buffer circolare. Il thread T_0 è l'unico produttore, ed aggiorna un indice E che identifica la prima locazione libera del buffer. I restanti thread sono consumatori, e ciascuno di loro deve aggiornare un proprio indice S_i che identifica la prima locazione del buffer ancora da leggere.

La condizione "buffer vuoto" per un dato thread T_i corrisponde al caso in cui E `==` S_i, con la solita clausola che in ogni istante vi è sempre almeno una locazione libera nel buffer.

La condizione "buffer pieno" è invece più complessa: innanzi tutto deve essere verificata dal thread produttore T_0 che, prima di sovrascrivere una locazione del buffer, deve controllare che ciascuno dei thread T_i consumatori abbia già letto il dato in esso precedentemente memorizzato. Un semplice ragionamento sui tre possibili casi in cui l'ultima locazione libera del buffer può cadere (nella prima posizione, nell'ultima posizione, od in mezzo al buffer) porta a concludere che la condizione "buffer pieno" si verifica quando E' `==` m, ove E' è l'indice successivo ad E e

$$m = \begin{cases} \min_{1 \le i \le n} \{S_i\} & \text{se } \forall i, S_i \le E \\ \min_{1 \le i \le n} \{S_i \,:\, S_i > E\} & \text{altrimenti} \end{cases}$$

Svolgiamo l'esercizio seguendo un approccio "top-down". La funzione `main()` alloca ed inizializza il buffer circolare, crea i thread T_i ($i > 0$), poi passa ad eseguire il lavoro del thread T_0:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <pthread.h>
int main(int argc, char *argv[])
{
    int n = argc - 1;
    struct circular_buffer *cb = create_circular_buffer(n);
    spawn_threads(n, cb, argv);
    work_T0(cb);
    pthread_exit(0);
}
```

Il numero `n` di thread da creare è indicato, come si desume dalla traccia dell'esercizio, dal numero di stringhe passate come argomento sulla linea comando.

La struttura `struct circular_buffer` descrive lo stato del buffer circolare:

```
typedef long long value_t;

struct circular_buffer {
    pthread_mutex_t mtx;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
    value_t *buf;
    int n;
    unsigned int dimen;
    unsigned int E;
    unsigned int *vS;
    unsigned int minS;
};
```

Il buffer circolare contiene, nel vettore puntato dal campo `buf`, dati di tipo `value_t`, in questo caso un numero intero abbastanza grande da poter rappresentare valori a 48 bit (6 byte). In una architettura a 32 bit il tipo di dati C `long long` ha generalmente dimensione 64 bit.

La struttura di dati contiene anche un mutex e due variabili condizione per sincronizzare l'accesso agli indici del buffer circolare. Il campo `vS` punta al vettore con gli indici S_i, mentre il campo `minS` memorizza il valore di m discusso in precedenza.

La funzione `create_circular_buffer()` alloca ed inizializza il buffer circolare:

```
struct circular_buffer * create_circular_buffer(int n)
{
    int rc;
    struct circular_buffer *cb;
    long pagesize = sysconf(_SC_PAGESIZE);
    abort_on_error(pagesize == -1, "Error in sysconf()");
    cb = malloc(sizeof(struct circular_buffer));
    abort_on_error(cb==NULL, "Cannot allocate memory");
    cb->vS = malloc(n*sizeof(cb->vS[0]));
    abort_on_error(cb->vS==NULL, "Cannot allocate memory");
    cb->buf = malloc(pagesize);
    abort_on_error(cb->buf==NULL,"Cannot allocate memory");
    cb->dimen = pagesize / sizeof(cb->buf[0]);
    cb->E = cb->minS = 0;
    memset(cb->vS, 0, n*sizeof(cb->vS[0]));
    cb->n = n;
    rc = pthread_mutex_init(&cb->mtx, NULL);
    abort_on_error(rc != 0 && ((errno = rc)),
```

```
                    "Cannot initialize pthread mutex");
    rc = pthread_cond_init(&cb->not_full, NULL);
    abort_on_error(rc != 0 && ((errno = rc)),
                    "Cannot initialize pthread condvar");
    rc = pthread_cond_init(&cb->not_empty, NULL);
    abort_on_error(rc != 0 && ((errno = rc)),
                    "Cannot initialize pthread condvar");
    return cb;
}
```

Seguendo la traccia dell'esercizio, la dimensione del buffer circolare è pari ad una pagina di memoria, ed è ricavata usando `sysconf()`. La macro `abort_on_error` verifica una condizione d'errore ed eventualmente provoca la terminazione dell'intera applicazione:

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

Si osservi come la macro venga utilizzata per controllare i codici di errore delle funzioni della libreria PThreads, che non impostano la variabile d'errore `errno` ma restituiscono direttamente il codice d'errore (diverso da zero).

La funzione `main()` invoca la funzione `spawn_threads()` per creare i thread consumatori:

```
void spawn_threads(int n, struct circular_buffer *cb,
                    char *argv[])
{
    int i, rc;
    struct thread_work_t *tws =
                     malloc(n*sizeof(struct thread_work_t));
    abort_on_error(tws == NULL, "Cannot allocate memory");
    for (i=0; i<n; ++i) {
        pthread_t tid;
        tws[i].idx = i;
        tws[i].arg = argv[i+1];
        tws[i].cb = cb;
        rc = pthread_create(&tid, NULL, work_Ti, &tws[i]);
        abort_on_error(rc != 0 && ((errno = rc)),
                        "Cannot create thread");
    }
```

```
}
```

La struttura di dati `thread_work_t` descrive il lavoro che deve essere svolto da un thread consumatore: In pratica, la struttura conterrà l'indice $i-1$ del thread, la stringa passata sulla linea comando che codifica il valore da cercare entro il buffer circolare, ed il puntatore al buffer circolare stesso:

```
struct thread_work_t {
    int idx;
    char *arg;
    struct circular_buffer *cb;
};
```

Infine la funzione `main()` invoca `work_T0()` per svolgere il lavoro affidato al thread T_0:

```
int T0_terminated = 0;
void work_T0(struct circular_buffer *cb)
{
    FILE *f;
    int rc;
    f = fopen("input.dat", "r");
    abort_on_error(f==NULL, "Cannot open input.dat file ");
    while (!feof(f)) {
        value_t l = 0;
        if (fread(&l, 6, 1, f) == 1)
            write_to_cb(cb, l);
        else
            abort_on_error(ferror(f),
                           "Error reading from 'input.dat'");
    }
    abort_on_error(fclose(f),
                   "Cannot close input.dat file");
    get_cb_lock(cb);
    T0_terminated = 1;
    rc = pthread_cond_broadcast(&cb->not_empty);
    abort_on_error(rc != 0 && ((errno = rc)),
                   "Error in broadcasting pthread condvar");
    release_cb_lock(cb);
}
```

Dopo aver aperto il file `input.dat` in sola lettura, `work_T0()` entra in un ciclo in cui legge i dati nel file a gruppi di 6 byte alla volta. Nella traccia dell'esercizio è specificato che il file memorizza i valori in formato "little-endian" da 6 byte, mentre l'architettura del calcolatore che esegue il programma è little-endian da

4 byte. Inoltre possiamo assumere che la dimensione della variabile `l` di tipo `value_t` è di almeno 8 byte. Se pertanto azzeriamo la variabile `l` e la usiamo come buffer destinazione in cui scrivere la sequenza di 6 byte del valore letto dal file, i byte nulli più significativi di `l` non verranno modificati, e quindi alla fine `l` conterrà esattamente il valore richiesto, ma codificato in 8 byte anziché 6.

Per scrivere il valore letto dal file nel buffer circolare viene utilizzata la funzione `write_to_cb()`:

```
void write_to_cb(struct circular_buffer *cb, value_t v)
{
    int rc;
    unsigned int nextE = cb->E + 1;
    if (nextE == cb->dimen)
        nextE = 0;
    get_cb_lock(cb);
    while (nextE == cb->minS) {
        rc = pthread_cond_wait(&cb->not_full, &cb->mtx);
        abort_on_error(rc != 0 && ((errno = rc)),
                       "Error waiting on pthread condvar");
    }
    cb->buf[cb->E] = v;
    cb->E = nextE;
    cb_recompute_min(cb);
    rc = pthread_cond_broadcast(&cb->not_empty);
    abort_on_error(rc != 0 && ((errno = rc)),
                   "Error in broadcasting pthread condvar");
    release_cb_lock(cb);
}
```

Le funzioni `get_cb_lock()` e `release_cb_lock()` sono utilizzate per ottenere e rilasciare il mutex del buffer circolare:

```
void get_cb_lock(struct circular_buffer *cb)
{
    int rc = pthread_mutex_lock(&cb->mtx);
    abort_on_error(rc != 0 && ((errno = rc)),
                                "Cannot get pthread mutex");
}
void release_cb_lock(struct circular_buffer *cb)
{
    int rc = pthread_mutex_unlock(&cb->mtx);
    abort_on_error(rc != 0 && ((errno = rc)),
                              "Cannot release pthread mutex");
}
```

Nel caso in cui il buffer risulti pieno il thread T_0 dorme in attesa sulla variabile condizione `cb->not_full`. Specularmente, quando T_0 scrive un nuovo

valore entro il buffer risveglia tutti i thread in attesa sulla variabile condizione `cb->not_empty` utilizzando `pthread_cond_broadcast()`.

Ogni aggiornamento dell'indice E richiede di aggiornare il valore dell'espressione m, che dipende anche da E. Ciò viene svolto eseguendo la funzione `cb_recompute_min()`:

```
void cb_recompute_min(struct circular_buffer *cb)
{
    int j;
    unsigned int min, minE, E = cb->E;
    min = minE = cb->dimen;
    for (j=0; j<cb->n; ++j) {
        unsigned int idx = cb->vS[j];
        if (idx < min)
            min = idx;
        if (idx > E && idx < minE)
            minE = idx;
    }
    if (minE == cb->dimen)
        minE = min;
    cb->minS = minE;
}
```

Al termine della lettura del file di input, `work_T0()` imposta ad 1 la variabile globale `T0_terminated` e risveglia ogni thread che possa essere ancora in attesa sulla variabile condizione `not_empty`.

Analizziamo ora la funzione `work_Ti()` eseguita dai vari thread consumatori:

```
void * work_Ti(void *arg)
{
    struct thread_work_t *tw= (struct thread_work_t *) arg;
    struct circular_buffer *cb = tw->cb;
    int idx = tw->idx;
    value_t v, target;
    unsigned long found = 0;
    target = ascii_to_value_t(tw->arg);
    for (;;) {
        int finish = 0;
        v = read_from_cb(cb, tw->idx, &finish);
        if (finish)
            break;
        if (v == target)
            ++found;
    }
    printf("T%d: found %lu values (%lld)\n",
            idx+1, found, target);
    pthread_exit(0);
```

```
}
```

Come prima operazione, il thread converte il valore in formato testo decimale passato sulla linea comando in un numero memorizzato in una variabile `value_t` utilizzando la funzione `ascii_to_value_t()`:

```
value_t ascii_to_value_t(const char *arg)
{
    value_t v;
    char *p;
    errno = 0;
    v = (value_t) strtoll(arg, &p, 10);
    abort_on_error(errno!=0 || *p!='\0', "Invalid input");
    return v;
}
```

Successivamente il thread entra in un ciclo. In ogni iterazione viene letto un valore dal buffer circolare utilizzando la funzione `read_from_cb()`. Questa funzione in alternativa restituisce un nuovo valore da controllare oppure imposta il flag `finish` a 1 per segnalare che T_0 ha terminato di leggere il file e contemporaneamente che il buffer non contiene più alcun valore da leggere per il thread.

```
value_t read_from_cb(struct circular_buffer *cb, int i,
                     int *term_flag)
{
    int rc;
    value_t v;
    get_cb_lock(cb);
    while (cb->E == cb->vS[i]) {
        if (T0_terminated) {
            *term_flag = 1;
            release_cb_lock(cb);
            return 0;
        }
        rc = pthread_cond_wait(&cb->not_empty, &cb->mtx);
        abort_on_error(rc != 0 && ((errno = rc)),
                    "Error in waiting on pthread condvar");
    }
    v = cb->buf[cb->vS[i]];
    increment_cb_idx(cb, i);
    rc = pthread_cond_signal(&cb->not_full);
    abort_on_error(rc != 0 && ((errno = rc)),
                       "Error signaling pthread condvar");
    release_cb_lock(cb);
    return v;
}
```

Infine, per aggiornare il valore dell'indice S_i del thread viene invocata la funzione `increment_cb_idx()`:

```
void increment_cb_idx(struct circular_buffer *cb, int i)
{
    cb->vS[i]++;
    if (cb->vS[i] == cb->dimen)
        cb->vS[i] = 0;
    cb_recompute_min(cb);
}
```

Esercizio 1, traccia B

La traccia di questo esercizio è sostanzialmente analoga a quella precedente. L'unica differenza è che questa volta il formato dei dati nel file di ingresso `input.dat` è "big-endian" a 4 byte. Ci limitiamo pertanto a descrivere i punti del codice differenti rispetto a quanto già mostrato.

Innanzi tutto, la dimensione dei dati nel buffer circolare può essere ridotta, poiché 4 byte sono sufficienti per memorizzare tutti i valori:

```
typedef long value_t;
```

La funzione `work_T0()` deve convertire dal formato "big-endian" del file al formato "little-endian" del calcolatore:

```
void work_T0(struct circular_buffer *cb)
{
    FILE *f;
    int rc;

    f = fopen("input.dat", "r");
    abort_on_error(f==NULL,
                   "Cannot open file 'input.dat'");
    while (!feof(f)) {
        value_t l;
        if (fread(&l, 4, 1, f) == 1) {
            l = big_to_little_endian(l);
            write_to_cb(cb, l);
        } else
            abort_on_error(ferror(f),
                   "Error reading from 'input.dat'");
    }
    abort_on_error(fclose(f),
```

```
                                "Cannot close input.dat file");
    get_cb_lock(cb);
    T0_terminated = 1;
    rc = pthread_cond_broadcast(&cb->not_empty);
    abort_on_error(rc != 0 && ((errno = rc)),
                   "Error in broadcasting pthread condvar");
    release_cb_lock(cb);
}
```

La conversione vera e propria è effettuata dalla funzione `big_to_little_endian()`:

```
value_t big_to_little_endian(value_t be)
{
    value_t le;
    char *pb = (char *) &be;
    char *pl = (char *) &le;
    pl[0] = pb[3];
    pl[1] = pb[2];
    pl[2] = pb[1];
    pl[3] = pb[0];
    return le;
}
```