Sistemi Operativi **(M. Cesati)**

Compito scritto del 13 febbraio 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. 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

Una matrice rettangolare di numeri di tipo `unsigned int` è memorizzata in un file con il seguente formato binario:

- All'inizio del file sono memorizzate le dimensioni N (numero di righe) e M (numero di colonne) della matrice come sequenza di byte corrispondenti al formato nativo del calcolatore.
- Di seguito sono memorizzati riga per riga gli $N \times M$ elementi della matrice, ciascuno come sequenza di byte nel formato nativo del calcolatore.

Ad esempio, se il calcolatore è little-endian e con interi `unsigned int` di 32 bit, la matrice $\begin{pmatrix} 100 & 200 \\ 300 & 400 \\ 500 & 600 \end{pmatrix}$ è memorizzata con la sequenza di byte nel file:

3 0 0 0 2 0 0 0 100 0 0 0 200 0 0 0 44 1 0 0 144 1 0 0 244 1 0 0 88 2 0 0

Scrivere una applicazione C/POSIX multithread che riceva come argomento il percorso di un file contenente la codifica appena descritta di una matrice, con dimensioni (non prefissate) $N \times M$. L'applicazione deve utilizzare $N + M$ thread, ciascuno dei quali calcolerà (in modo concorrente rispetto agli altri thread) la somma degli elementi di una riga o di una colonna della matrice. Al termine, l'applicazione dovrà scrivere in standard output il valore massimo tra tutte le somme delle righe e delle colonne (nell'esempio sopra riportato, il valore 1200). Si possono trascurare i problemi legati agli overflow delle somme. Si curino gli aspetti di sincronizzazione tra i vari thread dell'applicazione.

Esercizio 2

Con riferimento ad un moderno sistema operativo di tipo Unix, si descriva in modo conciso ma esauriente in cosa consiste la differenza tra una applicazione multiprocesso ed una applicazione multithread. Si descrivano inoltre i meccanismi interni di comunicazione e sincronizzazione più comunemente utilizzati in ciascuna delle due tipologie di applicazione.

Sistemi Operativi **(M. Cesati)**

Esempio di programma del compito scritto del 13 febbraio 2017

Esercizio 1

Descriviamo il programma utilizzando un approccio "top-down". La funzione `main()` apre il file di input e lo mappa in memoria, crea i thread "figli", aspetta la loro terminazione calcolando il valore massimo delle varie somme, ed infine stampa in standard output il risultato:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>

int main(int argc, char *argv[])
{
    unsigned int *map, rows, cols, max;
    struct thread_t *threads;
    abort_on_error(argc != 2,
                   "Wrong number of command line arguments");
    map = open_and_map_file(argv[1], &rows, &cols);
    threads = spawn_threads(map, rows, cols);
    max = wait_results(threads, rows+cols);
    printf("Max value: %u\n", max);
    return EXIT_SUCCESS;
}
```

La macro `abort_on_error()` controlla una condizione d'errore e se necessario termina prematuramente il programma:

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

Per aprire e mappare in memoria il file `main()` invoca `open_and_map_file()`, così definita:

```
unsigned int *open_and_map_file(const char *name,
                        unsigned int *pr, unsigned int *pc)
{
    unsigned int *map, rows, cols;
    size_t flen;
    int fd = open(name, O_RDONLY);
    abort_on_error(fd == -1, "Can't open input file");
    rows = read_uint(fd);
    cols = read_uint(fd);
    flen = (2+rows*cols)*sizeof(unsigned int);
    map = mmap(NULL, flen, PROT_READ, MAP_PRIVATE, fd, 0);
    abort_on_error(map==MAP_FAILED,"Can't map input file");
    *pr = rows;
    *pc = cols;
    return map+2;
}
```

Si osservi come la funzione inizialmente legga dal file il numero di righe e colonne della matrice. Sulla base di questa informazione calcola la dimensione occupata dalla matrice nel file e predispone il memory mapping in modo opportuno. La funzione restituisce l'indirizzo del primo elemento della matrice (saltando dunque i due interi iniziali del file che codificano la dimensione della matrice).

Per leggere un intero dal file viene utilizzata la funzione `read_uint()`:

```
unsigned int read_uint(int fd)
{
    unsigned int v;
    int c = read(fd, &v, sizeof(v));
    abort_on_error(c != sizeof(v),
                   "Unexpected error reading from file");
    return v;
}
```

Si osservi che poiché il file codifica i numeri nel formato nativo del calcolatore, è sufficiente leggere un numero di byte corrispondenti alla dimensione dell'intero in byte, e non occorre operare alcuna conversione. Per semplicità, e considerato il ridotto numero di byte che codificano un intero, si è scelto di considerare le letture "corte" (interrotte da un segnale o altro evento esterno) come errori di lettura.

La funzione `main()` continua invocando `spawn_threads()` per creare i thread secondari dell'applicazione:

```
struct thread_t *spawn_threads(unsigned int *map,
                               unsigned int r, unsigned int c)
{
    struct thread_t *ts, *t;
    unsigned int i;
    int rc;
    ts = malloc((r+c)*sizeof(struct thread_t));
    abort_on_error(ts == NULL, "Memory allocation error");
    for (i=0, t=ts; i<r; ++i, ++t) {
        t->start = map+(i*c);
        t->skip = 1;
        t->dim = c;
        rc = pthread_create(&t->tid, NULL, compute_sum, t);
        abort_on_error((errno=rc), "Can't spawn a thread");
    }
    for (i=0; i<c; ++i, ++t) {
        t->start = map+i;
        t->skip = c;
        t->dim = r;
        rc = pthread_create(&t->tid, NULL, compute_sum, t);
        abort_on_error((errno=rc), "Can't spawn a thread");
    }
    return ts;
}
```

Inizialmente la funzione alloca un vettore con $N + M$ elementi di tipo `struct thread_t`, ciascuno dei quali descrive il lavoro che deve essere svolto da un singolo thread:

```
struct thread_t {
    pthread_t tid;
    unsigned int *start;
    unsigned int skip;
    unsigned int dim;
    unsigned int sum;
};
```

Il significato dei campi della struttura è il seguente:

`tid` Il descrittore POSIX del thread.

`start` L'indirizzo del primo elemento considerato dal thread.

`skip` La distanza in memoria tra due elementi consecutivi considerati dal thread. Poiché le matrici sono memorizzate riga per riga, nel caso di un thread che analizza una riga è pari a 1, mentre nel caso di un thread che analizza una colonna è pari a M (numero di colonne).

`dim` Il numero di elementi da analizzare per il thread. Nel caso di un thread "riga" è il numero di colonne M, mentre nel caso di un thread "colonna" è il numero di righe N.

`sum` La somma di riga o colonna calcolata dal thread.

La funzione `spawn_threads()` procede poi a creare e lanciare gli $N + M$ thread secondari dell'applicazione, inizializzando opportunamente ciascun elemento del vettore di descrittori. Si osservi che per poter utilizzare correttamente la macro `abort_on_error()` con le API POSIX per i thread, assegniamo il valore restituito da `pthread_create()` alla variabile `errno`. Il valore restituito da questa assegnazione è esattamente quello restituito dalla API, pertanto se esso è diverso da zero (errore) l'esecuzione del programma viene terminata prematuramente.

Infine `spawn_threads()` termina restituendo l'indirizzo del vettore di descrittori.

Ciascun thread esegue la funzione `compute_sum()`:

```
void *compute_sum(void *arg)
{
    struct thread_t *t = (struct thread_t *) arg;
    unsigned int i, sum=0, *p;
    p = t->start;
    for (i=0; i<t->dim; ++i) {
        sum += *p;
        p += t->skip;
    }
    t->sum = sum;
    pthread_exit(NULL);
}
```

Dopo aver ricavato l'indirizzo del proprio descrittore dall'argomento passato alla funzione, si esegue un semplice ciclo in cui si somma il valore di tutti gli elementi della riga o colonna della matrice.

Successivamente la funzione `main()` invoca `wait_results()` per attendere la terminazione di tutti i thread:

```
unsigned int wait_results(struct thread_t *ts,
                          unsigned int n)
{
    unsigned int i, max=0;
    int rc;
    for (i=0; i<n; ++i) {
        rc = pthread_join(ts[i].tid, NULL);
        abort_on_error((errno=rc), "Can't join a thread");
```

```
        if (ts[i].sum > max)
            max = ts[i].sum;
    }
    return max;
}
```

Man mano che ciascun thread termina, la funzione controlla se il valore della somma da esso calcolato è maggiore del valore massimo finora osservato. Alla fine viene restituito il valore massimo tra le somme di tutte le righe e colonne della matrice, che viene poi scritto in standard output dalla funzione `main()`.