Sistemi Operativi **(M. Cesati)**

Compito scritto del 19 febbraio 2013

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 1. In un file di testo è memorizzata una sequenza di lunghezza indefinita di numeri interi di tipo `unsigned int` separati tra loro da spazi bianchi e/o caratteri di fine riga (`\n`). Scrivere un programma C/POSIX che riceve il nome del file di testo da linea comando; esso deve leggere i numeri dal file e riscrivere in standard output solo i numeri che sono divisibili esattamente per

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47.$$

Ad esempio, se nel file di testo appare il numero 109134 esso dovrà essere scritto sullo standard output (infatti $109134 = 2 \cdot 3^3 \cdot 43 \cdot 47$). Al contrario, se nel file di testo appare il numero 109130, esso non dovrà apparire in standard ouput (infatti $109130 = 2 \cdot 5 \cdot 7 \cdot 1559$).

Esercizio 2. Modificare il programma dell'esercizio precedente in modo tale esso utilizzi 8 thread POSIX, ciascuno dei quali esamina un diverso numero letto dal file. (Si noti che in generale il file contiene più di 8 numeri, perciò ciascun thread, dopo aver processato un numero, deve leggere il successivo numero da analizzare dal file.) Si raccomanda di curare gli aspetti di sincronizzazione relativi alla lettura dal file ed alla scrittura sullo standard output.

Esercizio 3. Si descriva in modo esauriente il concetto di "memoria virtuale", e se discutano i potenziali vantaggi per la semplicità e l'efficienza del sistema.

Sistemi Operativi (M. Cesati)

Esempio dei programmi del compito scritto del 19 febbraio 2013

Esercizio 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Le operazioni principali del programma sono (1) l'apertura del file di input, e (2) il ciclo di processamento dei valori entro il file:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *inf;

    if (argc != 2) {
        fprintf(stderr,"Usage: %s <inputfile>\n", argv[0]);
        return EXIT_FAILURE;
    }
    inf = open_input_file(argv[1]);
    process_values_in_file(inf);
    if (fclose(inf) != 0)
        perror(argv[1]);
    return EXIT_SUCCESS;
}
```

La funzione `open_input_file()` apre il file di ingresso:

```
FILE * open_input_file(const char *filename)
{
    FILE *f;

    f = fopen(filename, "r");
    if (f == NULL) {
        perror(filename);
        exit(EXIT_FAILURE);
    }
    return f;
}
```

Il cuore del programma è realizzato dalla funzione `process_values_in_file()`:

```
void process_values_in_file(FILE *inf)
{
    int rc;
    unsigned int v;

    do {
        rc = fscanf(inf, "%u", &v);
        if (rc == 1) {
            if (analyze_value(v))
                printf("%u\n", v);
        } else
            if (!feof(inf))
                fprintf(stderr,
                        "Error reading from input file\n");
    } while (!feof(inf));
}
```

La funzione `analyze_value()` deve restituire 1 se e solo se il valore passato come argomento è esattamente divisibile per i divisori indicati nel testo. Un semplice algoritmo per realizzare tale funzione consiste nel dividere ripetutamente per i vari divisori, passando da un divisore al successivo quando si scopre che il resto della divisione è diverso da zero. Se il quoziente ad un certo punto si riduce ad uno, allora il valore iniziale era divisibile e la funzione restituisce 1. Al contrario, se i divisori finiscono ed il quoziente non è ridotto ad uno, allora il valore iniziale non era divisibile e la funzione restituisce 0.

```
#define num_divisors 15
static const unsigned int divisors[num_divisors] =
    {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47};

int analyze_value(unsigned int v)
{
    unsigned int i, d, q;

    if (v < 2)
        return 0;
    for (i=0; i<num_divisors; ++i) {
        d = divisors[i];
        for (;;) {
            if (v == 1)
                return 1;
            q = v / d;
            if (d * q != v)
                break;
            v = q;
        }
    }
    return 0;
}
```

Esercizio 2

Il programma dell'esercizio precedente deve essere modificato per effettuare l'elaborazione utilizzando 8 thread POSIX.

È necessario includere l'header file per i thread POSIX, e definire una struttura di dati che contenga le informazioni necessarie a ciascun thread:

```
#include <pthread.h>

#define num_threads 8

struct thread_data {
    pthread_t tid;
    FILE *inf;
    pthread_mutex_t *pmtx;
};
```

Nella struttura sono memorizzati l'identificatore del thread, il puntatore alla struttura `FILE` per l'accesso al file di ingresso, e un puntatore al mutex necessario per sincronizzare gli accessi di ingresso e uscita dei thread.

Nella funzione `main()` vengono allocati il vettore di strutture `thread_data` ed il mutex:

```
int main(int argc, char *argv[])
{
    FILE *inf;
    struct thread_data td[num_threads];
    pthread_mutex_t mtx;
    int i;
```

Dopo l'apertura del file di ingresso, realizzata dalla stessa funzione `open_input_file()` dell'esercizio precedente, si inizializza il mutex:

```
    if (argc != 2) {
        fprintf(stderr,
                    "Usage: %s <inputfile>\n", argv[0]);
        return EXIT_FAILURE;
    }
    inf = open_input_file(argv[1]);
    if (pthread_mutex_init(&mtx, NULL) != 0) {
        fprintf(stderr, "Error in pthread_mutex_init()\n");
        return EXIT_FAILURE;
    }
```

Poi vengono inizializzati gli elementi del vettore di strutture `thread_data`:

```
    for (i=0; i<num_threads; ++i) {
        td[i].inf = inf;
        td[i].pmtx = &mtx;
    }
```

Si creano 7 thread POSIX (l'ottavo è costituito dal processo iniziale):

```
    td[0].tid = pthread_self();
    create_threads(td+1, num_threads-1);
```

Subito dopo la loro creazione tutti i thread iniziano ad eseguire la funzione `process_values_in_file()`. Terminata la creazione dei thread anche il processo principale passa ad eseguire tale funzione:

```
    process_values_in_file(td+0);
```

Quando il thread master ritorna dalla funzione `process_values_in_file()`, aspetta la terminazione di tutti gli altri thread, poi conclude l'esecuzione del programa:

```
    join_all_threads(td+1, num_threads-1);
    if (fclose(inf) != 0)
        perror(argv[1]);
    return EXIT_SUCCESS;
} /* end of function main() */
```

La creazione dei thread è affidata alla funzione `create_threads()`:

```
void create_threads(struct thread_data *td, int num)
{
    int i;

    for (i=0; i<num; ++i)
        if (pthread_create(&td[i].tid, NULL,
                  process_values_in_file, td+i) != 0) {
            fprintf(stderr,
                  "Error returned by pthread_create()\n");
            exit(EXIT_FAILURE);
        }
}
```

La funzione `process_values_in_file()` deve essere modificata rispetto all'esercizio precedente: deve ricevere come argomento l'indirizzo della struttura `thread_data` contenente le informazioni necessarie al thread; inoltre deve proteggere le funzioni di ingresso e uscita per mezzo del mutex.

```
void *process_values_in_file(void *arg)
{
    int rc;
    unsigned int v;
    struct thread_data *td = (struct thread_data *) arg;

    do {
        get_mutex(td->pmtx);
        rc = fscanf(td->inf, "%u", &v);
        put_mutex(td->pmtx);
        if (rc == 1) {
            if (analyze_value(v) == 1) {
                get_mutex(td->pmtx);
                printf("%u\n", v);
                put_mutex(td->pmtx);
            }
        } else
            if (!feof(td->inf))
                fprintf(stderr,
                    "Error reading from input file\n");
    } while (!feof(td->inf));
    return NULL;
}
```

La funzione `analyze_value()` è la stessa dell'esercizio precedente.

Per acquisire e rilasciare il mutex sono utilizzate due semplici funzioni:

```
void get_mutex(pthread_mutex_t *pmtx)
{
    if (pthread_mutex_lock(pmtx) != 0) {
        fprintf(stderr, "Error in pthread_mutex_lock()\n");
        exit(EXIT_FAILURE);
    }
}

void put_mutex(pthread_mutex_t *pmtx)
{
    if (pthread_mutex_unlock(pmtx) != 0) {
        fprintf(stderr, "Error in pthread_mutex_lock()\n");
        exit(EXIT_FAILURE);
    }
}
```

Infine l'attesa per la terminazione dei thread è affidata alla funzione `join_all_threads()`:

```
void join_all_threads(struct thread_data *td, int num)
{
    int i;

    for (i=0; i<num; ++i)
        if (pthread_join(td[i].tid, NULL) != 0) {
            fprintf(stderr,
                    "Error returned by pthread_join()\n");
            exit(EXIT_FAILURE);
        }
}
```