Sistemi Operativi (M. Cesati)

Compito scritto del 24 settembre 2013 — Turno 1

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.
Tenere presente che saranno valutati anche l'ordine e la chiarezza dell'esposizione.

Esercizio 1. Una matrice rettangolare di numeri di tipo `float` 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, nel formato nativo del calcolatore.
- Di seguito sono memorizzati riga per riga gli $N \times M$ elementi della matrice, nel formato nativo del calcolatore.

Realizzare una applicazione parallela multithread costituita da $M + 1$ thread. Ogni colonna della matrice è associata ad un diverso thread, che determina il valore massimo $V_{\max}$ ed il valore minimo $V_{\min}$ tra gli elementi della colonna, e restituisce come risultato la differenza $V_{\max} - V_{\min}$. Un altro thread calcola la somma di tutti i valori restituiti dagli altri thread e stampa il risultato in standard output.

Esercizio 2. Si descriva in modo esauriente lo scopo ed il funzionamento dell'algoritmo *Buddy system*.

Sistemi Operativi **(M. Cesati)**

Compito scritto del 24 settembre 2013 — Turno 2

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.
Tenere presente che saranno valutati anche l'ordine e la chiarezza dell'esposizione.

Esercizio 1. Scrivere una applicazione C/POSIX costituita da 2 processi P_1 e P_2 collegati tra loro mediante 2 pipe A e B (aventi diverse direzioni del flusso di dati).

- Il processo P_1 continuamente legge linee di testo dallo standard input; per ciascuna linea letta:
  - Rovescia i caratteri (il primo diventa ultimo, il secondo diventa penultimo, ecc.)
  - Scrive la linea rovesciata sulla pipe A, inviandola così al processo P_2
  - Legge una linea dalla pipe B, ne rovescia i caratteri e scrive il risultato sullo standard output
- Il processo P_2 continuamente legge linee di testo dalla pipe A; per ciascuna linea letta:
  - Modifica la linea in modo da anteporre tutti i caratteri in posizione dispari a quelli in posizione pari (si veda l'esempio sotto)
  - Scrive la linea così modificata sulla pipe B.

Ad esempio, se P_1 legge dallo standard input la linea "`abcdefghijklmn`", invia a P_2 la linea "`nmlkjihgfedcba`", riceve da P_2 la linea "`nljhfdbmkigeca`", e scrive in standard output la linea "`acegikmbdfhjln`".

Esercizio 2. Si descriva in modo esauriente lo scopo ed il funzionamento generale del meccanismo di annotazione di un *journaling file system*.

Sistemi Operativi (M. Cesati)

Esempio di programmi del compito scritto del 24 settembre 2013

Esercizio 1 — Turno 1

Svolgiamo l'esercizio secondo un approccio "top-down". Decidiamo preliminarmente di mappare in memoria il file contenente la matrice. Pertanto possiamo considerare le operazioni principali del programma: (1) apertura e mappatura del file, (2) creazione delle strutture di dati e dei thread, (3) attesa dei risultati e stampa del valore finale.

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

int main(int argc, char *argv[])
{
    float sum, *map;
    struct job_t *jobs;
    unsigned int nrow, ncol;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return EXIT_FAILURE;
    }

    map = map_file(argv[1], &nrow, &ncol);
    jobs = malloc(sizeof(struct job_t)*ncol);
    if (jobs == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        return EXIT_FAILURE;
    }
    spawn_threads(jobs, nrow, ncol, map);
    sum = join_threads(jobs, ncol);
    printf("%f\n", sum);
    return EXIT_SUCCESS;
}
```

La struttura di dati `struct job_t` descrive il lavoro che deve svolgere un singolo thread:

```
struct job_t {
    pthread_t tid;
    unsigned int ncol;
    unsigned int nrow;
    float *vect;
    float res;
};
```

La funzione `main()` alloca un vettore di strutture `job_t`, una per ciascun thread da lanciare.

La funzione `map_file()` riceve in ingresso il nome del file contenente la matrice letto dalla linea comando, e restituisce il numero di righe, il numero di colonne e l'inizio della zona di memoria contenente i valori della matrice.

```
float *map_file(const char *path, unsigned int *row,
                unsigned int *col)
{
    char *map;
    int N, M, fd = open_file(path);
    N = read_int(fd);
    M = read_int(fd);
    if (N <= 0 || M <= 0) {
        fprintf(stderr, "Invalid matrix dimensions\n");
        exit(EXIT_FAILURE);
    }
    *row = N;
    *col = M;
    map = mmap(NULL, 2*sizeof(int)+(N*M)*sizeof(float),
            PROT_READ, MAP_PRIVATE, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    /* skip the dimensions */
    return (float *) (map+2*sizeof(int));
}
```

Il cuore della funzione è la chiamata di sistema `mmap()`, che mappa in memoria il contenuto dell'intero file. La funzione `open_file()` apre il file in lettura, mentre la funzione `read_int()` legge un valore di tipo `int` in formato nativo dal file:

```
int open_file(const char *path)
{
    int fd = open(path, O_RDONLY);
    if (fd == -1) {
        perror(path);
```

```
        exit(EXIT_FAILURE);
    }
    return fd;
}

int read_int(int fd)
{
    int v, c = read(fd, &v, sizeof(v));
    if (c == sizeof(v))
        return v;
    if (c == 0)
        fprintf(stderr, "Unexpected end of file\n");
    else if (c == -1 && errno != EINTR)
        perror("Read from input file");
    else
        fprintf(stderr, "Read operation interrupted,
           aborting\n");
    exit(EXIT_FAILURE);
}
```

Tornando alla descrizione della funzione `main()`, i thread vengono lanciati invocando la funzione `spawn_thread()`:

```
void spawn_threads(struct job_t *jobs, unsigned int nrow,
                   unsigned int ncol, float *map)
{
    unsigned int i;

    for (i=0; i<ncol; ++i) {
        jobs[i].nrow = nrow;
        jobs[i].ncol = ncol;
        jobs[i].vect = map+i;
        if (pthread_create(&jobs[i].tid, NULL, thread_job,
                           jobs+i) != 0) {
            fprintf(stderr, "Error in pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
}
```

Per attendere che i thread completino il lavoro, `main()` invoca la funzione `join_threads()`:

```
float join_threads(struct job_t *jobs, unsigned int num)
{
    float sum = 0.0;
    unsigned int i;
```

```
    for (i=0; i<num; ++i) {
        if (pthread_join(jobs[i].tid, NULL) != 0) {
            fprintf(stderr, "Error in pthread_join()\n");
            exit(EXIT_FAILURE);
        }
        sum += jobs[i].res;
    }
    return sum;
}
```

Si noti come è stato scelto di memorizzare il risultato calcolato da ciascun thread in un apposito campo della struttura `job_t`. Sarebbe stato possibile in questo caso particolare anche restituire il valore direttamente tramite i parametri di `pthread_exit()`/`pthread_join()`.

Infine la procedura eseguita dagli M thread lanciati da `main()` è implementata dalla funzione `thread_job()`:

```
void *thread_job(void *p)
{
    struct job_t *job = (struct job_t *) p;
    unsigned int i;
    float Vmin, Vmax;
    float *v = job->vect;

    Vmin = Vmax = *v;
    for (i=1; i<job->nrow; ++i) {
        v += job->ncol;
        if (*v < Vmin)
            Vmin = *v;
        if (*v > Vmax)
            Vmax = *v;
    }
    job->res = Vmax - Vmin;
    pthread_exit(NULL);
}
```

Esercizio 1 — Turno 2

Svolgiamo l'esercizio secondo un approccio "top-down". La funzione `main()` del programma esegue le tre operazioni principali dell'applicazione: (1) creazione delle pipe di comunicazione, (2) creazione del processo figlio P_2 e (3) esecuzione del lavoro del processo P_1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int pipeA[2]; /* data dir: from P1 to P2 */
int pipeB[2]; /* data dir: from P2 to P1 */

int main(int argc, char *argv[])
{
    if (argc != 1) {
        fprintf(stderr, "Usage: %s (no arguments)\n",
                argv[0]);
        return EXIT_FAILURE;
    }
    make_pipes(pipeA, pipeB);
    spawn_child();
    do_P1_work(pipeA, pipeB);
    return EXIT_SUCCESS;
}
```

La funzione `make_pipes()` crea le due pipe di comunicazione A e B:

```c
void make_pipes(int pipeA[2], int pipeB[2])
{
    if (pipe(pipeA) || pipe(pipeB)) {
        perror("creating pipes");
        exit(EXIT_FAILURE);
    }
}
```

La funzione `spawn_child()` crea il processo P_2:

```c
void spawn_child(void)
{
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
```

```
    if (pid == 0)
        do_P2_work(pipeA, pipeB);
}
```

Rimangono da analizzare le due funzioni specifiche dei processi P_1 e P_2, ossia rispettivamente `do_P1_work()` e `do_P2_work()`.

```
void do_P1_work(int pipeA[2], int pipeB[2])
{
    /* pipeA is write-end, pipeB is read-end */
    if (close(pipeA[0]) || close(pipeB[1])) {
        perror("close");
        exit(EXIT_FAILURE);
    }
    process_lines_from_stdin(pipeA[1], pipeB[0]);
}
```

Il processo P_1 chiude i descrittori dei terminali delle pipe a cui non è interessato, poi invoca la funzione `process_lines_from_stdin()`:

```
#define max_line 1024

void process_lines_from_stdin(int wfd, int rfd)
{
    char line[max_line];
    for (;;) {
        if (fgets(line, max_line, stdin) == NULL) {
            if (feof(stdin))
                break;
            fprintf(stderr, "Error reading from stdin\n");
            exit(EXIT_FAILURE);
        }
        /* remove trailing new-line */
        if (line[strlen(line)-1]=='\n')
            line[strlen(line)-1]='\0';
        reverse_line(line);
        snd_line(line, wfd);
        if (rcv_line(line, rfd)) {
            fprintf(stderr,
                    "Process P2 has closed pipeB\n");
            exit(EXIT_FAILURE);
        }
        reverse_line(line);
        fputs(line, stdout);
        fputc('\n', stdout);
    }
}
```

La funzione continuamente legge linee dallo standard input, rimuove l'eventuale terminatore '`\n`' della linea, rovescia la linea e la invia tramite la pipe A al processo P_2. Poi attende una linea di risposta dalla pipe B, rovescia la linea ricevuta, e stampa il risultato sullo standard output.

La funzione `reverse_line()` rovescia la stringa di caratteri passata come argomento. Si appoggia ad una funzione `swap()` che scambia di posto due caratteri della stringa:

```
void swap(char *line, int i, int j)
{
    char c = line[i];
    line[i] = line[j];
    line[j] = c;
}
void reverse_line(char *line)
{
    int i, k = strlen(line)-1;
    for (i=0; i<k; i++, k--)
        swap(line, i, k);
}
```

Per inviare la stringa su di una pipe viene utilizzata la funzione `snd_line()`. Oltre ai caratteri stampabili della stringa viene anche inviato il terminatore di stringa, ossia il carattere '`\0`', che potrà essere usato dall'altro processo per riconoscere la fine della stringa.

```
void snd_line(char *line, int fd)
{
    int len = strlen(line)+1; /* include '\0' */

    while (len > 0) {
        int rc = write(fd, line, len);
        if (rc == -1) {
            perror("write-end of pipe");
            exit(EXIT_FAILURE);
        }
        len -= rc;
        line += rc;
    }
}
```

Per ricevere una stringa da una pipe viene utilizzata la funzione `rcv_line()`. Poiché si è scelto di non inviare i caratteri new-line '`\n`' e di delimitare le stringhe con il terminatore '`\0`', il modo più semplice (anche se non il più efficiente) di implementare la funzione consiste nel leggere un carattere alla volta dalla pipe.

```
int rcv_line(char *line, int fd)
{
    char c;
    int i, rc;
    for (i=0; i<max_line-2; ++i) {
        rc = read(fd, &c, sizeof(char));
        if (rc == -1) {
            perror("read-end of pipe");
            exit(EXIT_FAILURE);
        }
        if (rc == 0 && i == 0)
            return 1;
        if (rc == 0 || c == '\0')
            break;
        line[i] = c;
    }
    line[i] = '\0';
    return 0;
}
```

La funzione `rcv_line()` restituisce il valore 1 se si rileva la condizione EOF sulla pipe e non si è ancora letto alcun carattere della nuova stringa. Altrimenti viene restituito il valore 0.

Passiamo ora a descrivere la funzione `do_P2_work()`, che è del tutto analoga a `do_P1_work()`:

```
void do_P2_work(int pipeA[2], int pipeB[2])
{
    /* pipeB is write-end, pipeA is read-end */
    if (close(pipeA[1]) || close(pipeB[0])) {
        perror("close");
        exit(EXIT_FAILURE);
    }
    process_lines_from_pipeA(pipeA[0], pipeB[1]);
}
```

La funzione `process_lines_from_pipeA()` continuamente legge stringhe di testo delimitate da '\0' dalla pipe A, le modifica, e spedisce le nuove stringhe sulla pipe B:

```
void process_lines_from_pipeA(int rfd, int wfd)
{
    char line[max_line];
    for (;;) {
        if (rcv_line(line, rfd))
            break; /* no more lines from P1 */
        oddeven_line(line);
```

```
        snd_line(line, wfd);
    }
}
```

Infine, la funzione `oddeven_line()` modifica una stringa passata come argomento anteponendo tutti caratteri in posizione dispari a quelli in posizione pari:

```
void oddeven_line(char *line)
{
    char app[max_line];
    int i, j, l = strlen(line);
    for (i=j=0; i<l; i+=2, ++j)
        app[j] = line[i];
    for (i=1; i<l; i+=2, ++j)
        app[j] = line[i];
    app[j] = '\0';
    strncpy(line, app, max_line);
}
```