Sistemi Operativi **(M. Cesati)**

Compito scritto del 18 luglio 2012

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. Una matrice rettangolare di numeri di tipo `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 `int` di 32 bit, la matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ è memorizzata con la sequenza di byte nel file:

3 0 0 0 2 0 0 0 1 0 0 0 2 0 0 0 3 0 0 0 4 0 0 0 5 0 0 0 6 0 0 0

a) Realizzare un programma C/POSIX che riceva come argomento tre percorsi di file. I primi due file sono di ingresso e contengono ciascuno una matrice rettangolare con la codifica appena descritta. Il terzo file è di uscita: il programma scrive nel file con la stessa codifica la matrice corrispondente al prodotto "riga-colonna" delle due matrici di ingresso:

$$c_{i,j} = \sum_{k=1}^{M} a_{i,k} \cdot b_{k,j}, \qquad i = 1, \ldots N, \quad j = 1, \ldots P$$

Se le dimensioni delle matrici rettangolari di ingresso non consentono di effettuare il prodotto (ossia il numero di colonne della prima matrice non coincide con il numero di righe della seconda matrice) il programma deve terminare segnalando un errore.

b) Modificare il programma precedente in modo che il calcolo della matrice prodotto sia effettuato in parallelo con thread POSIX.

Esercizio 2. Descrivere in modo esauriente la problematica e le soluzioni relative alla gestione della memoria fisica nel nucleo di un sistema operativo.

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 18 luglio 2012

Esercizio 1-a

Svolgiamo l'esercizio seguendo un approccio "top-down". Decidiamo preliminarmente di mappare i file in memoria per accedere alle matrici di ingresso e uscita. Pertanto possiamo considerare le operazioni principali del programma: (1) apertura e mappatura dei file di ingresso, (2) creazione e mappatura del file di uscita, (3) calcolo e scrittura degli elementi della matrice prodotto.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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[])
{
    int fdA, fdB;  /* file descriptors */
    int rA, rB, cA, cB; /* dimensions of input matrices */
    int *mA, *mB, *mC;

    if (argc != 4) {
        fprintf(stderr,"Usage: %s <filename> <filename> <filename>\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    fdA = open_file(argv[1]);
    map_matrix(fdA, &mA, &rA, &cA);
    if (close(fdA) == -1)
        perror(argv[1]);  /* non fatal error */

    fdB = open_file(argv[2]);
    map_matrix(fdB, &mB, &rB, &cB);
    if (close(fdB) == -1)
        perror(argv[2]);  /* non fatal error */

    if (cA != rB) {
        fprintf(stderr,
                "Input matrices have wrong dimensions, aborting\n");
        exit(EXIT_FAILURE);
    }
```

```
    mC = create_output_matrix(argv[3], rA, cB);
    matrix_product(mC, mA, mB, rA, cA, cB);
    return EXIT_SUCCESS;
}
```

Per aprire il file in lettura controllando gli eventuali errori:

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

La funzione `map_matrix()` legge le dimensioni della matrice dal file e mappa la matrice in memoria:

```
void map_matrix(int fd, int **matr, int *row, int *col)
{
    int r, c;
    int *m;

    /* read dimensions of the matrix */

    r = read_int(fd);
    c = read_int(fd);

    if (r <= 0 || c <= 0) {
        fprintf(stderr,
                "Non positive dimension in input file, aborting\n");
        exit(EXIT_FAILURE);
    }

    *row = r;
    *col = c;

    /* create a memory mapping of the matrix */

    m = mmap(NULL, (2+r*c)*sizeof(int), PROT_READ,MAP_PRIVATE, fd,0);
    if (m == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    *matr = m+2; /* skip the matrix dimensions */
```

```
}
```

Per leggere un intero dal file utilizziamo la funzione `read_int()`. Poiché gli interi sono memorizzati nel file nel formato nativo dell'architettura, è sufficiente copiare i byte che costituiscono l'intero in una variabile di tipo `int`, esattamente nell'ordine in cui si trovano nel file.

```
int read_int(int fd)
{
    int v, c;

    c = read(fd, &v, sizeof(v));
    if (c == sizeof(v))
        return v;

    if (c == -1 && errno != EINTR) {
        perror("Read from input file");
        exit(EXIT_FAILURE);
    }

    if (c == 0) {
        fprintf(stderr, "Unexpected end of file\n");
        exit(EXIT_FAILURE);
    }

    /* a short read: either a signal, or a partial read from a FIFO...
        */

    fprintf(stderr, "Read operation interrupted, aborting\n");
    exit(EXIT_FAILURE);
}
```

Per creare il file di uscita e mapparlo in memoria utilizziamo la funzione `create_output_matrix()`. L'unica complicazione è che è necessario estendere il file fino alla sua dimensione finale per poterlo mappare in memoria. Ciò non costituisce un problema in quanto le dimensioni della matrice risultato sono già note.

```
int * create_output_matrix(const char *path, int row, int col)
{
    int *m;
    int fd, size = (row*col+2)*sizeof(int);

    /* create the output file */

    fd = open(path, O_RDWR|O_CREAT, 0644);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
```

```
    /* write the dimensions */

    write_int(fd, row);
    write_int(fd, col);

    /* extend the file size */

    if (lseek(fd, size-sizeof(int), SEEK_SET) == -1) {
        perror("lseek");
        exit(EXIT_FAILURE);
    }

    write_int(fd, fd); /* actual data does not matter */

    m = mmap(NULL, size, PROT_WRITE, MAP_SHARED, fd, 0);
    if (m == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    if (close(fd) == -1)
        perror("close"); /* non fatal error */

    return m+2;
}
```

Per scrivere un intero nel file utilizziamo la funzione `write_int()`, analoga alla `read_int()` già definita:

```
void write_int(int fd, int v)
{
    int c;

    c = write(fd, &v, sizeof(v));
    if (c == sizeof(v))
        return;

    if (c == -1 && errno != EINTR) {
        perror("Write to output file");
        exit(EXIT_FAILURE);
    }

    fprintf(stderr, "Write operation interrupted, aborting\n");
    exit(EXIT_FAILURE);
}
```

Finalmente per calcolare la matrice prodotto definiamo la funzione `matrix_product()`. Qui la complicazione sta nel fatto che il numero di colonne delle matrici non è costante (noto

al momento della compilazione), perciò non è possibile far riferimento ad un elemento della matrice `m` con la notazione naturale `m[i][j]`:

```
void matrix_product(int *C, int *A, int *B, int rA, int cA, int cB)
{
    int i, j, k;

    for (i=0; i<rA; ++i) {
        for (j=0; j<cB; ++j) {
            int a, b, v = 0;
            for (k=0; k<cA; ++k) {
                a = *(A+(i*cA+k)); /* A[i][k] */
                b = *(B+(k*cB+j)); /* B[k][j] */
                v += a*b;
            }
            *(C+(i*cB+j)) = v; /* C[i][j] */
        }
    }
}
```

Esercizio 1-b

Il calcolo della matrice prodotto deve essere svolto in parallelo. Decidiamo di creare un thread per ciascun elemento della matrice risultato. Adattiamo il programma sviluppato per il punto precedente. È innanzi tutto necessario includere un nuovo header file:

```
#include <pthread.h>
```

Possiamo limitarci a modificare la funzione `matrix_product()`.

Per tenere traccia del lavoro svolto da ciascun thread definiamo una struttura di dati apposita. Ciascun thread esegue il prodotto di una riga per una colonna, perciò vengono passati l'indirizzo della riga, l'indirizzo del primo elemento della colonna, e l'indirizzo dell'elemento che memorizza il risultato. Inoltre è necessario avere il numero di colonne della due matrici in ingresso.

```
struct thread_job {
    pthread_t tid;
    int *row;
    int *col;
    int colAsize;
    int colBsize;
    int *dest;
};
```

La procedura `matrix_product()` alloca dinamicamente un vettore di strutture `thread_job`; si noti che i thread non saranno in alcun modo sincronizzati, perciò non è possibile allocare sullo stack. Poi per ciascun elemento della matrice risultato la procedura crea il thread.

```
void matrix_product(int *C, int *A, int *B, int rA, int cA, int cB)
{
    int i, j;
    struct thread_job *jobs;

    jobs = malloc(rA*cB*sizeof(struct thread_job));
    if (jobs == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    for (i=0; i<rA; ++i)
        for (j=0; j<cB; ++j) {
            jobs->row = A + i*cA;
            jobs->col = B+j;
            jobs->colAsize= cA;
            jobs->colBsize= cB;
            jobs->dest = C+(i*cB+j);
            if (pthread_create(&jobs->tid,NULL,prod_rowcol,jobs)!=0) {
                fprintf(stderr, "Error in pthread_create\n");
                exit(EXIT_FAILURE);
            }
            ++jobs;
        }
}
```

Infine la procedura `prod_rowcol()` è eseguita da ciascuno dei thread per calcolare il prodotto riga-colonna. I parametri su cui operare vengono letti dal puntatore alla struttura di dati `thread_job` il cui valore è ricavato, con un opportuno *cast*, dall'argomento passato alla procedura.

```
void *prod_rowcol(void *p)
{
    int k, a, b, v;
    struct thread_job *job = (struct thread_job *)p;

    for (k=v=0; k < job->colAsize; ++k) {
        a = *(job->row +k);
        b = *(job->col +(k*job->colBsize));
        v += a*b;
    }
    *job->dest = v;
    return NULL;
}
```