Sistemi Operativi **(M. Cesati)**

Compito scritto del 19 settembre 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 C/POSIX multithread costituita da 12 thread concorrenti (da T_1 a T_{12}). L'applicazione legge un file di input tramite *memory mapping* con blocchi di dimensione massima di 4096 byte. Il nome del file è specificato sulla linea comando, e si assume che esso contenga valori numerici di 8 bit in formato nativo del calcolatore. Il thread T_i (per i da 1 a 12) analizza le porzioni contigue di 2^i byte del file e scrive in standard output la somma dei valori da 8 bit di ciascuna porzione. Quindi, per esempio, il thread T_1 scriverà in standard output la somma del primo e secondo byte del file, poi la somma del terzo e quarto byte del file, eccetera; il thread T_2 scriverà in standard output la somma dei primi 4 byte del file, poi la somma dei successivi 4 byte, e così via. L'applicazione deve funzionare con file di qualunque lunghezza, ma si possono trascurare i problemi di overflow quando si superano i limiti della dimensione nativa del calcolatore. Curare gli aspetti di sincronizzazione tra i thread, ove necessario.

Esercizio 2

Si descriva in modo esauriente in che modo è tipicamente organizzata la memoria utilizzata in User Mode da un processo Unix.

Sistemi Operativi **(M. Cesati)**

Compito scritto del 19 settembre 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 C/POSIX multithread che legge un file di input il cui nome è specificato come primo argomento della linea comando. Il file di input deve essere letto utilizzando la chiamata di sistema a basso livello `read()`. Si assuma che il file contenga valori numerici di 8 bit in formato nativo del calcolatore. L'applicazione è costituita da 256 thread concorrenti (da T_0 a T_{255}). Il thread T_i (per ogni i da 0 a 255) dovrà scrivere in standard output il numero di occorrenze di byte con valore i all'interno del file di input. L'applicazione deve funzionare con file di qualunque lunghezza, ma si possono trascurare i problemi di overflow quando si superano i limiti della dimensione nativa del calcolatore. Curare gli aspetti di sincronizzazione tra i thread, ove necessario.

Esercizio 2

In riferimento ad un generico sistema operativo moderno, si definisca cosa si intende per "processo" e si descriva come il kernel tipicamente memorizza le informazioni correlate a tutti i processi esistenti nel sistema.

Sistemi Operativi (M. Cesati)

Esempi dei programmi del compito scritto del 19 settembre 2016

Esercizio 1, traccia A

Descriviamo il programma utilizzando un approccio "top-down". Poiché dobbiamo realizzare una applicazione multithread, risulta conveniente definire alcune variabili globali per mezzo delle quali i thread possono condividere informazioni:

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

int g_fd;
unsigned char * g_buf;
size_t g_len;
pthread_barrier_t g_barrier;
```

`g_fd` è il descrittore del file di input, mentre `g_len` contiene la sua lunghezza; `g_buf` contiene l'indirizzo di un buffer contenente un blocco di dati del file; infine `g_barrier` consente ai vari thread di sincronizzarsi.

La funzione `main()` inizialmente controlla che sia stato passato il nome di un file sulla linea comando:

```
#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)

int main(int argc, char *argv[])
{
    abort_on_error(argc<2,
          "Missing file name in command line");
    g_fd = open_file(argv[1]);
```

```
    g_len = get_file_length(g_fd);
    g_buf = mmap_at_offset(g_fd, 0);
    init_barrier(&g_barrier);
    spawn_threads();
    pthread_exit(0);
}
```

`main()` apre il file specificato sulla linea comando con `open_file()` e ne calcola la lunghezza con `get_file_length()`:

```
int open_file(const char *name)
{
    int fd = open(name, O_RDONLY);
    abort_on_error(fd==-1, "Cannot open input file");
    return fd;
}

size_t get_file_length(int fd)
{
    off_t v = lseek(fd, 0, SEEK_END);
    abort_on_error(v==(off_t)-1, "Error in lseek()");
    return (size_t) v;
}
```

Successivamente `main()` mappa il primo blocco di `BLOCKSIZE` (ossia 4096) byte del file invocando la funzione `mmap_at_offset()`:

```
#define BLOCKSIZE 4096

unsigned char * mmap_at_offset(int fd, off_t offset)
{
    void *p = mmap(NULL, BLOCKSIZE, PROT_READ, MAP_PRIVATE,
                   fd, offset);
    abort_on_error(p==MAP_FAILED, "Error in mmap()");
    return p;
}
```

Si noti che l'indirizzo in memoria del blocco di dati del file è salvato da `main()` nella variabile globale `g_buf`.

Poi `main()` inizializza la barriera di sincronizzazione invocando `init_barrier()`:

```
void init_barrier(pthread_barrier_t *pb)
{
    int rc = pthread_barrier_init(pb, NULL, 12);
    abort_on_error(rc!=0 && ((errno=rc)),
```

```
                    "Cannot initialize barrier");
}
```

Si osservi come, in caso di errore, il codice di errore venga copiato esplicitamente in `errno`; le funzioni POSIX Thread, infatti, non impostano la variabile globale `errno`.

Infine, prima di terminare il thread originale, `main()` invoca `spawn_threads()` per creare i 12 thread richiesti:

```
void spawn_threads(void)
{
    int i, rc, size;
    pthread_t tid;

    for (i=1, size=2; i<=12; ++i, size*=2) {
        rc = pthread_create(&tid, NULL, thread_worker,
                            (void *)size);
        abort_on_error(rc!=0 && ((errno=rc)),
                       "Thread creation failed");
    }
}
```

Si osservi come si è utilizzato il puntatore passato alla funzione eseguita dai thread per codificare il valore 2^i che ciascun thread deve considerare.

Il cuore dell'applicazione è costituito ovviamente dalla funzione `thread_worker()`, eseguita da ciascuno dei thread:

```
void *thread_worker(void *arg)
{
    int size = (int) arg, selected;
    size_t curoff = 0;

    for (;;) {
        compute_sums(size);
        selected = wait_on_barrier();
        curoff += BLOCKSIZE;
        if (curoff >= g_len)
            break;
        if (selected) {
            int rc = munmap(g_buf, BLOCKSIZE);
            abort_on_error(rc==-1, "Error in munmap()");
            g_buf = mmap_at_offset(g_fd, curoff);
        }
        wait_on_barrier();
    }
    pthread_exit(0);
```

```
}
```

La funzione `thread_worker()` calcola le somme richieste invocando la funzione `compute_sums()`. Prima di proseguire si pone in attesa che tutti gli altri 11 thread completino questa operazione. A tale scopo invoca `wait_on_barrier()`:

```
int wait_on_barrier(void)
{
    int rc = pthread_barrier_wait(&g_barrier);
    if (rc == 0 || rc == PTHREAD_BARRIER_SERIAL_THREAD)
        return rc;
    abort_on_error(((errno=rc)),
                   "Error in pthread_barrier_wait()");
    return 0; /* NEVER REACHED */
}
```

Si deve ricordare che la funzione `pthread_barrier_wait()` prosegue l'esecuzione solo quando tutti i 12 thread hanno raggiunto la barriera; 11 di essi avranno come valore restituito zero, mentre uno solo di essi avrà il valore `PTHREAD_BARRIER_SERIAL_THREAD`.

La funzione `thread_worker()` controlla se l'analisi del file è completata; in tal caso, interrompe il ciclo `for`. Poi il thread selezionato dalla funzione `pthread_barrier_wait()` rimuove il memory map del blocco del file appena analizzato, e crea un nuovo memory map per il successivo blocco. Nel frattempo, tutti gli altri thread attendono ancora una volta sulla barriera, perché devono aspettare che il nuovo memory map sia completato prima di eseguire una nuova iterazione del ciclo.

Infine resta da analizzare la funzione `compute_sums()`, che calcola le somme dei gruppi di byte di dimensione 2^i e ne stampa il valore sullo standard output:

```
void compute_sums(int size)
{
    unsigned long value;
    int p, q;

    for (p=0; p<BLOCKSIZE; p+=size) {
        value = 0;
        for (q=0; q<size; ++q)
            value += *(g_buf + p + q);
        printf("%d-block sum: %lu\n", size, value);
    }
}
```

Esercizio 1, traccia B

L'esercizio è analogo a quello della traccia A, e la soluzione utilizza la stessa idea fondamentale: tutti i thread utilizzano una barriera per sincronizzarsi tra loro; uno di essi, scelto dal sistema in modo automatico, aggiorna il buffer contenente i dati del file prima di procedere con la successiva iterazione.

Anche in questo caso è conveniente utilizzare alcune variabili globali per i dati condivisi dei thread:

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

#define BLOCKSIZE 4096

unsigned char g_buf[BLOCKSIZE];
int g_fd;
size_t g_len;
pthread_barrier_t g_barrier;
```

`g_buf` è il buffer contenente un blocco di `BLOCKSIZE` (4096) byte del file; `g_fd` è il descrittore del file, mentre `g_len` è la quantità di dati effettivamente presente in `g_buf`. Infine `g_barrier` è la barriera di sincronizzazione dei thread.

La funzione `main()` è analoga a quella della traccia A, con la differenza che non è necessario ricavare in anticipo la dimensione del file:

```
#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)

int main(int argc, char *argv[])
{
    abort_on_error(argc<2,
                   "Missing file name in command line");
    g_fd = open_file(argv[1]);
    g_len = read_next_block(g_fd, g_buf);
    init_barrier(&g_barrier);
```

```
    spawn_threads();
    pthread_exit(0);
}
```

La funzione `open_file()` è identica a quella utilizzata nel programma della traccia A. La funzione `read_next_block()` legge un nuovo blocco di dati dal file, memorizzandolo in `g_buf`; la lunghezza effettiva di tale blocco è salvata in `g_len`:

```
size_t read_next_block(int fd, unsigned char *buf)
{
    int toread = BLOCKSIZE;
    while (toread > 0) {
        int rc = read(fd, buf, toread);
        abort_on_error(rc==-1, "Error in read()");
        if (!rc)
            break;
        buf += rc;
        toread -= rc;
    }
    return BLOCKSIZE-toread;
}
```

Si noti che la funzione tenta di leggere `BLOCKSIZE` file, e "si arrende" solo in caso di errore di lettura oppure fine del file. La funzione termina l'esecuzione dell'intera applicazione (in caso di errore) oppure restituisce il numero di byte effettivamente letti.

La funzione `init_barrier()` è analoga a quella utilizzata nella traccia A, ma questa volta il numero di thread richiesto per proseguire l'esecuzione è pari a 256:

```
void init_barrier(pthread_barrier_t *pb)
{
    int rc = pthread_barrier_init(pb, NULL, 256);
    abort_on_error(rc!=0 && ((errno=rc)),
                   "Cannot initialize barrier");
}
```

Anche la funzione `spawn_threads()` è analoga a quella già vista:

```
void spawn_threads(void)
{
    int i, rc;
    pthread_t tid;
```

```
    for (i=0; i<256; ++i) {
        rc = pthread_create(&tid, NULL, thread_worker,
                            (void *)i);
        abort_on_error(rc!=0 && ((errno=rc)),
                       "Thread creation failed");

    }
}
```

In questo caso i thread ricevono come argomento di `thread_worker()` il proprio indice i, ossia il valore che essi devono ricercare all'interno dei blocchi del file.

La funzione `thread_worker()` è molto simile a quella omonima già considerata:

```
void *thread_worker(void *arg)
{
    unsigned char target = (int) arg;
    int selected;
    unsigned long sum = 0;

    while (g_len > 0) {
        sum += count_values(target, g_len);
        selected = wait_on_barrier();
        if (selected)
            g_len = read_next_block(g_fd, g_buf);
        wait_on_barrier();
    }
    printf("%d-values: %lu\n", target, sum);
    pthread_exit(0);
}
```

Questa volta la fine del ciclo `for` è segnalata dal valore zero presente in `g_len`, ossia dal fatto che la lettura del file non ha trovato nuovi byte da inserire nel buffer `g_buf`. La funzione `wait_on_buffer()` è identica a quella utilizzata nella traccia A. Infine, per contare le occorrenze del valore da ricercare ciascun thread invoca la funzione `count_values()`:

```
unsigned long count_values(unsigned char target, int bsize)
{
    int i;
    unsigned long t=0;

    for (i=0; i<bsize; ++i)
        if (g_buf[i] == target)
            ++t;
    return t;
}
```