*Sistemi Operativi* **(M. Cesati)**

**Compito scritto del 14 luglio 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. Tempo a disposizione: 2 ore.

**Esercizio**

La firma di controllo di un file è lunga 32 bit, ed è costruita considerando il file suddiviso in blocchi da 4096 byte. La firma di controllo a 32 bit di ciascun blocco è determinata così:

- I 16 bit meno significativi sono i 16 bit meno significativi della somma $\sum_i b_i$ ove $b_i$ sono i singoli byte del blocco ($0 \leq i \leq 4095$).
- I 16 bit più significativi sono i 16 bit meno significativi della somma $\sum_i (4096 - i) \cdot b_i$ ove $b_i$ sono i singoli byte del blocco ($0 \leq i \leq 4095$).

(Se l'ultimo blocco di un file è lungo meno di 4096 byte, considerare come se i byte mancanti avessero valore zero.)

La firma di controllo a 32 bit dell'intero file è lo XOR delle firme di controllo di ciascuno dei suoi blocchi.

Scrivere una applicazione C/POSIX multiprocesso _e_ multithread che riceve sulla linea comando un insieme arbitrario di nomi di file. L'applicazione crea un processo per ciascuno dei file indicati che lavora in modo concorrente rispetto agli altri processi.

Ciascun processo calcola in parallelo la somma di controllo a 32 bit del file utilizzando un thread per ciascun blocco di 4096 byte del file, e scrive in standard output il nome del file seguito dal valore della somma di controllo in esadecimale.

Tenere presente gli aspetti relativi alla sincronizzazione tra i flussi di esecuzione, ove necessario.

*Sistemi Operativi* (M. Cesati)

**Esempio di programma del compito scritto del 14 luglio 2017**

Svolgiamo l'esercizio seguendo un approccio "top-down".

```
#include <stdio.h>
#include <stdint.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, const char *argv[])
{
    abort_on_error(argc < 2,
                   "Wrong number of command line arguments");
    create_processes(argc-1, argv+1);
    return EXIT_SUCCESS;
}
```

La funzione `main()` controlla che sulla linea comando sia presente almeno un nome di file. Al solito, per gestire gli errori utilizziamo una macro per produrre un messaggio d'errore ed interrompere l'esecuzione:

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

La funzione `create_processes()` crea un diverso processo per ciascun nome di file sulla linea comando:

```
void create_processes(int numfile, const char *filenames[])
{
    int i;

    for (i=0; i<numfile; ++i) {
        pid_t pid = fork();
```

```
        abort_on_error(pid == -1, "Cannot fork a process");
        if (pid == 0) {
            compute_checksum(filenames[i]);
            return;
        }
    }
}
```

I processi lavorano in modo concorrente, ma poiché ciascuno opera su un file differente non è necessario sincronizzarli tra loro.

La funzione `compute_checksum()` riceve il nome di un file, calcola il checksum e lo stampa in standard output:

```
void compute_checksum(const char *filename)
{
    size_t filesize;
    unsigned char *mmbuf;
    uint32_t crc;

    mmbuf = open_and_map_file(filename, &filesize);
    crc = compute_checksum_in_parallel(mmbuf, filesize);
    printf("0x%08x %s\n", crc, filename);
}
```

Per prima cosa la funzione invoca `open_and_map_file()` per controllare se il file indicato è apribile in lettura e creare un file memory mapping per il suo contenuto:

```
unsigned char *open_and_map_file(const char *name,
                                 size_t *plength)
{
    unsigned char *map;
    off_t flen;

    int fd = open(name, O_RDONLY);
    abort_on_error(fd == -1, "Can't open input file");
    flen = lseek(fd, 0, SEEK_END);
    abort_on_error(flen == (off_t) -1,
                   "Can't seek to end of file");
    map = mmap(NULL, flen, PROT_READ, MAP_PRIVATE, fd, 0);
    abort_on_error(map == MAP_FAILED,
                   "Can't map input file");
    *plength = flen;
    abort_on_error(close(fd) == -1,
                   "Can't close input file");
    return map;
}
```

La chiamata di sistema `mmap()` richiede la lunghezza del blocco di dati del file da mappare. Per semplicità, la soluzione proposta si ricava la lunghezza totale del file e lo mappa in memoria nella sua interezza. Ovviamente è possibile implementare diversi approcci alternativi a questo.

La funzione `compute_checksum_in_parallel()` realizza il calcolo del checksum dell'intero file:

```
#define BLOCKSIZE (4096)

struct pthread_param {
    pthread_t tid;
    unsigned char *block;
    int block_size;
    uint32_t bcrc;
};

uint32_t compute_checksum_in_parallel(unsigned char *mmbuf,
                                      size_t size)
{
    uint32_t crc = 0;
    struct pthread_param *ptpars;
    int i, rc, nblocks, last_block_size;

    nblocks = (size+BLOCKSIZE-1) / BLOCKSIZE;
    last_block_size = size - (nblocks-1)*BLOCKSIZE;
    ptpars = malloc(nblocks*sizeof(struct pthread_param));
    abort_on_error(ptpars == NULL,
                   "Memory allocation failure");
    for (i=0; i<nblocks; ++i) {
        ptpars[i].block = mmbuf + i * BLOCKSIZE;
        ptpars[i].block_size = (i < nblocks-1 ?
                                BLOCKSIZE : last_block_size);
        rc = pthread_create(&ptpars[i].tid, NULL,
                    block_checksum, (void *)(ptpars+i));
        abort_on_error((errno=rc),
                       "Thread creation failure");
    }
    for (i=0; i<nblocks; ++i) {
        rc = pthread_join(ptpars[i].tid, NULL);
        crc ^= ptpars[i].bcrc;
    }
    free(ptpars);
    return crc;
}
```

Innanzi tutto viene calcolato il numero di blocchi del file; la dimensione prefissata di ogni blocco è indicata dalla macro `BLOCKSIZE`. Poi viene allocato un vettore di strutture di tipo `pthread_param`, ciascuna delle quali contiene i parametri relativi ad un singolo thread: `tid` è l'identificatore del thread, `block`

punta all'inizio del blocco di dati, `block_size` è la dimensione del blocco (sempre pari a `BLOCKSIZE` tranne eventualmente per l'ultimo blocco del file), e `bcrc` conterrà la somma di controllo del blocco calcolata dal thread.

Successivamente, viene eseguito un primo ciclo `for` che lancia l'esecuzione di un thread per ciascun blocco del file. Ciascun thread eseguirà la funzione `block_checksum()` con argomento il puntatore alla struttura `pthread_param` del thread.

Infine viene eseguito un secondo ciclo `for` in cui si attende la terminazione di tutti i thread lanciati precedentemente. Man mano che ciascun thread termina, il risultato della sua elaborazione, ossia la somma di controllo del blocco, viene utilizzato per l'operazione di XOR che aggiorna la somma di controllo del file.

Non rimane che esaminare la funzione `block_checksum()` che calcola la somma di controllo di un singolo blocco:

```
void * block_checksum(void *arg)
{
    struct pthread_param *param =
                        (struct pthread_param *)arg;
    unsigned char *p = param->block;
    int i, msum = 0, lsum = 0;

    for (i=0; i<param->block_size; ++i, ++p) {
        lsum += (*p);
        msum += (*p)*(BLOCKSIZE-i);
    }
    param->bcrc = lsum & 0xffff;
    param->bcrc |= (msum & 0xffff) << 16;
    pthread_exit(NULL);
}
```

Si osservi che `msum` non è presumibilmente in grado di memorizzare l'esatto valore di $\sum_i [b_i \times (4096 - i)]$, ma ciò non costituisce un problema in quanto quel che interessa sono i 16 bit meno significativi di tale valore.