Sistemi Operativi **(M. Cesati)**

Compito scritto del 26 giugno 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

Un file cifrato contiene una sequenza di byte c_i ($0 \leq i < L$) che sono il risultato di una operazione di XOR (operatore C "^") tra i caratteri di un testo sconosciuto p_i ($0 \leq i < L$) ed una password segreta s di lunghezza al più 32 byte. La password segreta in generale è più corta del contenuto del file, quindi viene ripetuta ciclicamente:

$$c_i = p_i \text{ XOR } s_j, \quad 0 \leq i < L, \quad j = i \bmod \texttt{strlen}(s)$$

Si assuma che la password segreta sia costituita esclusivamente da caratteri alfabetici minuscoli.

Scrivere una applicazione C/POSIX multithread costituita da 128 thread che cerca esaustivamente la password di un file cifrato il cui nome è specificato sulla linea comando. L'applicazione deve cercare tra tutti i possibili valori della password segreta quelli che decodificano il file cifrato in modo tale che la percentuale di caratteri alfabetici (maiuscoli e minuscoli) tra quelli decodificati sia maggiore del 90%. I valori ammissibili della password devono essere scritti sullo standard output.

Il lavoro di ricerca deve essere suddiviso tra i 128 thread, che debbono lavorare in parallelo. Si presti attenzione ai possibili problemi di sincronizzazione.

Sistemi Operativi **(M. Cesati)**

Esempio di programma del compito scritto del 26 giugno 2017

Svolgiamo come al solito l'esercizio seguendo un approccio "top-down". Per prima cosa è necessario decidere come suddividere le stringhe (password candidate) tra i 128 thread dell'applicazione.

Una soluzione potrebbe essere quella di suddividere lo spazio di ricerca (costituito da 26^{32} stringhe) in modo statico tra i 128 thread. Per i primi due caratteri della password vi sono in totale $26^2 = 676$ possibilità. Poiché $5 < 676/128 < 6$, il primo thread dovrebbe analizzare tutte le stringhe comincianti con le lettere "`aa`", "`ab`", ..., "`af`", il secondo thread dovrebbe analizzare tutte le stringhe comincianti con le lettere da "`ag`" a "`al`", e così via.

Un'altra soluzione, quella qui implementata, consiste invece nel fare in modo di memorizzare in un vettore globale l'ultima stringa analizzata dall'applicazione. Ciascun thread genera la stringa successiva all'ultima analizzata ed aggiorna il vettore globale. Ovviamente questa soluzione richiede la sincronizzazione dei thread per evitare che il vettore globale sia corrotto da accessi concorrenti.

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

#define MAX_PASS_LEN 32

char last_pw[MAX_PASS_LEN+1];
size_t length;
char *map;
int finished;
pthread_mutex_t mtx;
```

L'applicazione fa uso di alcune variabili globali utilizzate da tutti i thread. Il vettore `last_pw` contiene l'ultima stringa analizzata dalla applicazione. La variabile `map` punta ad una regione di memoria in cui è mappato il file cifrato, mentre `length` è la lunghezza di tale file. Il flag `finished` è utilizzato per gestire la terminazione dell'applicazione, mentre il mutex `mtx` serve per sincronizzare i thread in modo da evitare accessi concorrenti a `last_pw`.

La funzione `main()` controlla il numero di argomenti passati sulla linea comando, mappa il file cifrato in memoria, inizializza le altre strutture di dati, e crea i thread. L'esecuzione del thread originale poi si conclude:

```
int main(int argc, char *argv[])
{
    abort_on_error(argc != 2,
        "Wrong number of command line arguments");
    map = open_and_map_file(argv[1], &length);
    init_data_structures();
    spawn_threads();
    pthread_exit(NULL);
}
```

Al solito, per controllare gli errori e terminare prematuramente l'applicazione definiamo la macro `abort_on_error`:

```
#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 `open_and_map_file()` mappa il file cifrato in memoria:

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

La funzione `mmap()` richiede la conoscenza della lunghezza della porzione di file da mappare, perciò viene utilizzata la funzione `lseek()` per ricavare la lunghezza totale del file cifrato.

La funzione `init_data_structures()` completa l'inizializzazione delle altre variabili globali del programma:

```
void init_data_structures(void)
{
    int i, rc;
    for (i=0; i<MAX_PASS_LEN+1; ++i)
        last_pw[i] = '\0';
    rc = pthread_mutex_init(&mtx, NULL);
    abort_on_error((errno=rc),
                   "Can't initialize the mutex");
    finished = 0;
}
```

Si osservi che per riempire di zeri il vettore `last_pw` avremmo potuto usare le funzioni di libreria `memset()` o `bzero()`.

Per creare i thread la funzione `main()` esegue `spawn_threads()`:

```
#define Nthreads 128

void spawn_threads(void)
{
    int i, rc;
    for (i=0; i<Nthreads; ++i) {
        pthread_t tid;
        rc = pthread_create(&tid, NULL, search_passwords,
                            NULL);
        abort_on_error((errno=rc), "Can't spawn a thread");
    }
}
```

Considerata la natura dell'algoritmo e l'uso di variabili globali, non è necessario passare alcun dato specifico ai vari thread: tutti quanti possono svolgere esattamente lo stesso lavoro, eseguendo la funzione `search_passwords()`.

```
void *search_passwords(void *arg)
{
    char my_pw[MAX_PASS_LEN+1];
    arg = arg; /* avoid gcc's warning */
    for (;;) {
        get_lock();
        if (!get_next_password()) {
            release_lock();
            pthread_exit(NULL);
        }
        strncpy(my_pw, last_pw, MAX_PASS_LEN+1);
        release_lock();
```

```
            try_password(my_pw);
        }
    }
```

Ciascun thread utilizza un vettore locale allocato sullo stack, `my_pw`, per memorizzare la stringa su cui deve lavorare. La funzione ottiene il lock e genera una nuova stringa da analizzare invocando `get_next_password()`. Se tale funzione restituisce il valore zero, allora la ricerca è stata già conclusa, quindi il thread rilascia il lock e termina. Altrimenti, la stringa in `last_pw` viene copiata in `my_pw`, il lock viene rilasciato, e viene invocata la funzione `try_password()` per analizzare la possibile password.

Il mutex viene manipolato tramite le funzioni di servizio `get_lock()` e `release_lock()`:

```
void get_lock(void)
{
    int rc = pthread_mutex_lock(&mtx);
    abort_on_error((errno=rc), "Can't get mutex\n");
}

void release_lock(void)
{
    int rc = pthread_mutex_unlock(&mtx);
    abort_on_error((errno=rc), "Can't release mutex\n");
}
```

La funzione `get_next_password()`, eseguita in modo esclusivo da ciascun thread, considera la stringa memorizzata in `last_pw` e genera la stringa successiva. L'ordine delle stringhe è definito innanzi tutto dalla lunghezza della stringa, e poi dall'ordine lessicografico.

```
int get_next_password(void)
{
    int l, pl;
    if (finished == 1)
        return 0;
    l = pl = strlen(last_pw);
    while (l > 0 && last_pw[l-1] == 'z') {
        last_pw[l-1] = 'a';
        --l;
    }
    if (l > 0) {
        last_pw[l-1]++;
        return 1;
    }
    if (pl == MAX_PASS_LEN) {
        finished = 1;
```

```
            return 0;
        }
        last_pw[pl] = 'a';
        return 1;
}
```

All'inizio si controlla subito il flag `finished`: se settato, la funzione restituisce zero per far terminare il thread.

Successivamente viene determinata la lunghezza della stringa in `last_pw`. Il ciclo `while` comincia dall'ultimo carattere della stringa e procede verso il primo, sostituendo ogni carattere "`z`" con "`a`". Il ciclo si interrompe incontrando un carattere diverso da "`z`" oppure oltrepassando il primo carattere della stringa.

Subito dopo il ciclo `while` viene controllata la condizione d'uscita dal ciclo: se il ciclo si è interrotto perché è stato trovato un carattere diverso da "`z`", allora è sufficiente sostituire tale carattere con quello successivo nell'ordine alfabetico. Così, ad esempio, la stringa successiva a "`bczz`" sarà "`bdaa`".

Se invece il ciclo `while` è stato interrotto perché è stata oltrepassato l'inizio della stringa, la stringa successiva ha un carattere in più della precedente. Se la stringa precedente era già di 32 caratteri, il lavoro è concluso: si imposta `finished` a uno e si termina. Altrimenti, si aggiunge un nuovo carattere "`a`" in coda alla stringa. Così, ad esempio, la stringa successiva a "`zzzz`" sarà "`aaaaa`".

Infine, la funzione `try_password()` decodifica il file cifrato con la password candidata, e stampa la password se il numero di caratteri alfabetici risultati è maggiore del 90% della dimensione del file:

```
void try_password(char *pw)
{
    char *p = map;
    size_t f, nalp = 0;
    int j=0, pwlen = strlen(pw);
    for (p=map, f=0; f<length; ++p, ++f) {
        char c = *p ^ pw[j];
        if ((c >= 'a' && c <= 'z') ||
            (c >= 'A' && c <= 'Z'))
            ++nalp;
        if (++j == pwlen)
            j = 0;
    }
    if (nalp*10 > length*9)
        printf("%s\n", pw);
}
```