UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing
Decriptazione password
Endterm

Massimiliano Mancini

Anno accademico 2019/20

1 Introduzione

Il presente elaborato mostra come utilizzare POSIX Threads in linguaggio C per sfruttare al meglio la capacità di elaborazione allo scopo di decriptare una password di 8 caratteri selezionati nell'alfabeto [./0-9A-Za-z] della quale conosciamo solo la codifica hash ottenuta con funzione DES. Si tratta di un semplice attacco di forza bruta con uno spazio delle chiavi di cardinalità pari a 64^8 oppure 2^{48} ovvero 281.474.976.710.656.

2 Approccio utilizzato

Il tipo di problema può essere risolto in diversi modi, per esempio, se fossimo in presenza di un file di probabili password, un approccio conveniente potrebbe essere quello di produttore-consumatore, dove uno o più produttori leggono il file delle password mentre i consumatori verificano la corrispondenza degli hash. Nel nostro caso, dove non è presente un file di password e la risoluzione del problema è affidata solo alla pura potenza di calcolo, dobbiamo privilegiare al massimo la parallelizzazione eliminando tutti i possibili accessi a risorse condivise, sezioni critiche e le serializzazioni in generale così che il tempo di risoluzione sia direttamente proporzionale alla capacità di elaborazione ossia al numero di core a nostra disposizione. Per questa ragione useremo un approccio di tipo loop level dove lo spazio delle chiavi viene suddiviso tra i diversi thread che eseguono tutti lo stesso programma. I thread sono generati in modalità fork/join dal main. In particolare questi generano e contestualmente verificano le password nello spazio assegnato. Il primo thread che individua la giusta password, accede in maniera esclusiva a una variabile globale in cui registrerà il risultato interrompendo così anche il lavoro degli altri thread. Il controllo viene quindi restituito al main che stampa il risultato.

3 Ambiente di sviluppo

Il codice è stato sviluppato su piattaforma google cloud con le seguenti caratteristiche: un socket, 4 cores ognuno in grado di eseguire due threads per un totale di 8 CPU (Listato 1).
L'accesso al cloud è garantito attraverso una connessione ssh utilizzata anche in modalità sftp per l'accesso ai file sorgenti al fine di elaborarli con l'editor notepad++.

Listing 1: Caratteristiche gCloud

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU @ 2.30GHz
Stepping:              0
CPU MHz:               2300.000
BogoMIPS:              4600.00
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              46080K
NUMA node0 CPU(s):     0-7
```

4 Compilazione dei sorgenti

Il file principale `main` è compilato con i seguenti parametri
`g++ -Wall -Wextra -O3 -lcrypt -lpthread par.c -o par`
oppure, in modalità non parallela
`g++ -Wall -Wextra -O3 -lcrypt seq.c -o seq`
In entrambi i casi la compilazione avviene senza errori né avvisi (warning).

5 Utilizzo dei programmi

Il programma `seq`, compilato senza includere la libreria pthread, accetta due parametri: la password (presunta segreta) e il numero tentativi massimo da condurre, 0 per l'intero spazio delle chiavi.
Il programma `par` accetta un ulteriore parametro che rappresenta il numero di thread da utilizzare.
L'esecuzione del programma, sia in versione sequenziale che in versione parallela stampa su standard output il tempo di esecuzione e il risultato che può essere

positivo oppure negativo nel caso in cui sia esplicitato il numero massimo di tentativi da condurre.

6 Dettagli su scelte di codice

Listing 2: Ciclo principale while

```
while ((i < ci->size) && (strcmp(pwdes, testdes) != 0) && (! found)) {
    index[7]++;
    if (index[7] == 64) {
        index[7] = 0;
        test[7] = '.';
        index[6]++;
        if (index[6] == 64) {
            index[6] = 0;
            test[6] = '.';
            index[5]++;
            ...
```

Listing 3: Condizione di uscita per password trovata

```
if ((strcmp(pwdes, testdes) == 0)) {
    pthread_mutex_lock(&lock);
    found = 1;
    strcpy(found_pw, test);
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
```

Listing 4: Creazione dei thread

```
for (int i = 0; i < thread_idx; i++) {
    ci[i].start = size * i;
    ci[i].size = size;
    pthread_create(&tid[i], NULL, worker, (void *) &ci[i]);
}

// if n_threads is odd, we catch remainder
ci[thread_idx].start = size * (thread_idx);
ci[thread_idx].size = n - ci[thread_idx].start;

pthread_create(&tid[thread_idx], NULL, worker, (void *) &ci[thread_idx]);
```

Nel codice prodotto, sono state effettuate alcune scelte implementative che meritano una breve spiegazione.
Nel Listato 2 è riportato una parte del metodo utilizzato per la generazione di password. In particolare sono utilizzate 7 istruzioni `if` in cascata al fine di minimizzare il numero di operazioni per ogni ciclo. Una soluzione più elegante poteva implementare il calcolo della password in base a un indirizzo lineare, proprio come fatto nella sezione immediatamente precedente a quella in esame. Questa soluzione, però avrebbe costo superiore perché il numero di operazioni da compiere in ogni ciclo è maggiore. Si noti inoltre come venga costantemente controllata la variabile condivisa `found`. Essendo un controllo in sola lettura, non si generano problemi di accesso o race conditions.
Nel Listato 3 è riportato l'utilizzo del mutex `lock` per l'accesso esclusivo alle due variabili condivise `found` e `found_pw`. L'accesso esclusivo è necessario per mantenere una certa generalità di approccio anche in caso di hash collision. L'intera esecuzione non viene comunque appesantita in quanto il lock avviene una sola, per l'intero insieme di thread, subito dopo aver trovato la giusta password.
Nel Listato 4 si osserva che la creazione dei thread viene fatta in due passaggi: un primo ciclo `for` per i primi n - 1 thread e un secondo set di istruzioni per l'ultimo thread. Questo risolve due problemi: in caso di un numero thread dispari, l'ultimo thread ci assicura che tutte le chiavi saranno comunque esaminate, inoltre in caso di un solo thread il ciclo for viene direttamente ignorato generando un solo thread attraverso le istruzioni immediatamente successive.

Listing 5: Output di TOP

```
massimiliano1_mancini@instance-1:~/endterm$ top -H -p 7440
top - 10:25:22 up  3:12,  5 users,  load average: 7.83, 3.13, 1.64
Threads:  17 total,  16 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 20565092 total, 19916944 free,   357024 used,   291124 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 19906428 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7451 massimi+  20   0 1197220   2536   2184 R 50.3  0.0   0:19.66 par
 7452 massimi+  20   0 1197220   2536   2184 R 50.3  0.0   0:19.65 par
 7441 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.74 par
 7442 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.73 par
 7443 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.67 par
 7444 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.65 par
 7446 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.64 par
 7447 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.72 par
 7449 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.62 par
 7450 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.62 par
 7454 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.63 par
 7455 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.65 par
 7456 massimi+  20   0 1197220   2536   2184 R 50.0  0.0   0:19.61 par
 7445 massimi+  20   0 1197220   2536   2184 R 49.7  0.0   0:19.62 par
 7448 massimi+  20   0 1197220   2536   2184 R 49.7  0.0   0:19.71 par
 7453 massimi+  20   0 1197220   2536   2184 R 49.7  0.0   0:19.63 par
 7440 massimi+  20   0 1197220   2536   2184 S  0.0  0.0   0:00.00 par
```

Durante i vari test sono state catturate alcune schermate del comando `top -H -p <pid>` allo scopo di verificare che effettivamente tutti i thread fossero stati avviati come richiesto. Una schermata esemplificativa catturata durante l'esecuzione con 16 thread, è riportata in Listato 5

7 Metodo di rilevazione dei tempi di esecuzione

I tempi di esecuzione del codice sono rilevati attraverso `clock_gettime` sostanzialmente su tutta l'esecuzione della procedura, sia in versione sequenziale che in versione parallela.

8 Analisi dei risultati

Una prima misura rivela che sia la versione sequenziale che quella parallela del programma hanno un comportamento in termini di tempo, esattamente lineare rispetto al numero di chiavi da verificare. In altre parole il tempo impiegato per analizzare n chiavi è direttamente proporzionale a n come si vede in Figura 1

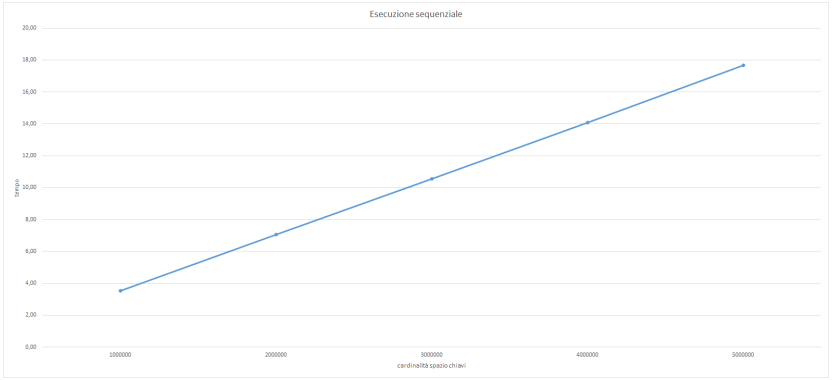

Figura 1: Linearità

Ricordando la formula dello speedup

$$S_P = \frac{t_s}{t_P}$$

dove t_s è il tempo di esecuzione sequenziale e t_P è il tempo di esecuzione parallela e P il numero di processori. In Figura 2 è riportato lo speedup ottenuto analizzando un campione di 5 milioni di chiavi. Questi ha un andamento lineare fino a 8 thread, per poi stabilizzarsi a un livello fisso di circa 4.15 anche in presenza di un numero maggiore di thread. Tale effetto di stabilizzazione e quindi di perdita di efficienza è mostrato in Figura 3

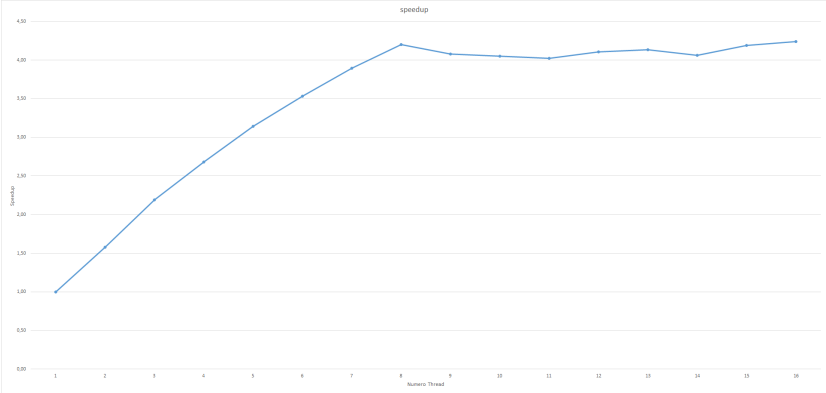

Figura 2: Speedup

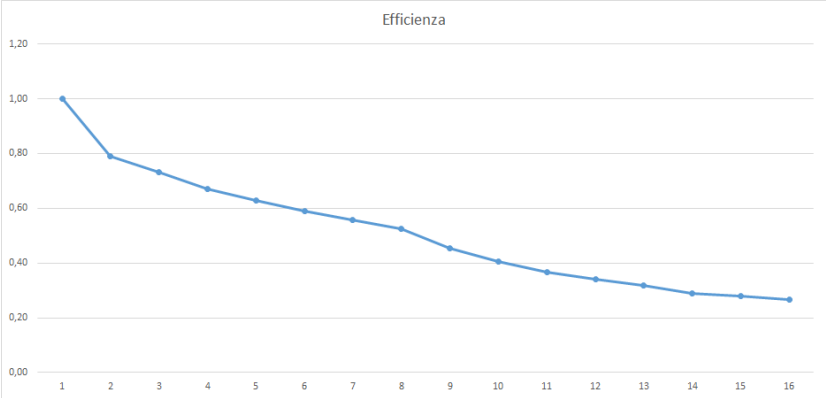

Figura 3: Efficienza

Codici

Listing 6: par

```
#include <stdio.h>
#include <crypt.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <chrono>
#include <ctime>

const char alphabet[64] = {'.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                           'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
                           'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
                           'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
                           'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'};


long long int n;


char *pwdes;
char found_pw[9] = "";
int found = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

struct chunk_info {
    long long int start;
    long long int size;
};

void* worker (void *input) {

    // struct to get args
    struct chunk_info *ci;
    ci = (chunk_info *) input;

    long long int i = 0;

    // used in reentrant crypt func
    struct crypt_data cryptdata;
    cryptdata.initialized = 0;

    // calc starting password (reverse linear address to string)
    char test[] = ".......";

    for (int j = 7; j >= 0; j--) {
        test[j] = alphabet[ci->start % 64];
        ci->start = ci->start / 64;
    }

    char *testdes = (char *) malloc(14);
    strncpy(testdes, crypt_r(test, "00", &cryptdata),13);

    int index[8] = {0};

    while ((i < ci->size) && (strcmp(pwdes, testdes) != 0) && (! found)) {
        index[7]++;
        if (index[7] == 64) {
            index[7] = 0;
            test[7] = '.';
            index[6]++;
            if (index[6] == 64) {
                index[6] = 0;
                test[6] = '.';
                index[5]++;
                if (index[5] == 64) {
                    index[5] = 0;
                    test[5] = '.';
                    index[4]++;
                    if (index[4] == 64) {
                        index[4] = 0;
                        test[4] = '.';
                        index[3]++;
                        if (index[3] == 64) {
                            index[3] = 0;
                            test[3] = '.';
                            index[2]++;
                            if (index[2] == 64) {
                                index[2] = 0;
                                test[2] = '.';
                                index[1]++;
                                if (index[1] == 64) {
                                    index[1] = 0;
                                    test[1] = '.';
                                    index[0]++;
                                } else {
```

```
                                        test[1] = alphabet[index[1]];
                                    }
                                } else {
                                    test[2] = alphabet[index[2]];
                                }
                            } else {
                                test[3] = alphabet[index[3]];
                            }
                        } else {
                            test[4] = alphabet[index[4]];
                        }
                    } else {
                        test[5] = alphabet[index[5]];
                    }
                } else {
                    test[6] = alphabet[index[6]];
                }
            } else {
                test[7] = alphabet[index[7]];
            }

            i++;
            strncpy(testdes, crypt(test, "00"),13);
        }

        // if pw is found, mutex is used to access shared variables
        if (strcmp(pwdes, testdes) == 0) {
            pthread_mutex_lock(&lock);
            found = 1;
            strcpy(found_pw, test);
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }
        return NULL;
    }

    int main(int argc, char* argv[]) {

        if (argc < 3) {
            printf("Usage: %s <password> <trials * 10^6> <threads>\n", argv[0]);
            return 1;
        }

        struct timespec t1, t2;
        double time_span;

        // get start time
        clock_gettime(CLOCK_REALTIME, &t1);

        pwdes = (char *) malloc (14);
        strncpy(pwdes, crypt(argv[1], "00"), 13);

        // number of keys
        if (atoi(argv[2]) == 0) {
            n = 281474976710656; // 64^8 or 2^48
        } else {
            n = atoi(argv[2]) * 1000000;
        }

        // number of threads to use
        int n_threads = atoi(argv[3]);
        int thread_idx = n_threads - 1;

        pthread_t tid[n_threads];
        struct chunk_info ci[n_threads];

        // number of keys per thread
        long long int size = n/n_threads;

        for (int i = 0; i < thread_idx; i++) {
            ci[i].start = size * i;
            ci[i].size = size;
            pthread_create(&tid[i], NULL, worker, (void *) &ci[i]);
        }

        // if n_threads is odd, we catch remainder
        ci[thread_idx].start = size * (thread_idx);
        ci[thread_idx].size = n - ci[thread_idx].start;

        pthread_create(&tid[thread_idx], NULL, worker, (void *) &ci[thread_idx]);

        // join all threads
        for (int i = 0; i < n_threads; i++) {
            pthread_join(tid[i], NULL);
        }

        // get stop time
        clock_gettime(CLOCK_REALTIME, &t2);
        time_span = (t2.tv_sec - t1.tv_sec) + (double)((t2.tv_nsec - t1.tv_nsec))/1000000000;

        if (found) {
```

```
        printf("Found pw: %s in %lf secs\n", found_pw, time_span);
    } else {
        printf("Not found, tested %llu in %lf secs\n", n, time_span);
    }
    return 0;
}
```

Listing 7: seq

```
#include <stdio.h>
#include <crypt.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

const char alphabet[64] = {'.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                           'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
                           'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
                           'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
                           'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'};
long long int n;


int main(int argc, char *argv[]) {

    if (argc < 2) {
        printf("Usage: %s <password> <trials * 10^6>\n", argv[0]);
        return 1;
    }

    // Number of keys
    if (atoi(argv[2]) == 0) {
        n = 281474976710656; // 64^8 or 2^48
    } else {
        n = atoi(argv[2]) * 1000000;
    }

    struct timespec t1, t2;
    double time_span;

    // get start time
    clock_gettime(CLOCK_REALTIME, &t1);

    char salt[] = "00";
    char *pwdes = (char *) malloc (14);
    char *testdes = (char *) malloc(14);

    long long int i = 0;

    // initial password
    char test[] = ".......";
    int index[8] = {0};

    strncpy(pwdes, crypt(argv[1], salt), 13);
    strncpy(testdes, crypt(test, salt),13);

    // test until number of trials is reached or pw is found
    while ((i < n) && (strcmp(pwdes, testdes) != 0)) {
        index[7]++;
        if (index[7] == 64) {
            index[7] = 0;
            test[7] = '.';
            index[6]++;
            if (index[6] == 64) {
                index[6] = 0;
                test[6] = '.';
                index[5]++;
                if (index[5] == 64) {
                    index[5] = 0;
                    test[5] = '.';
                    index[4]++;
                    if (index[4] == 64) {
                        index[4] = 0;
                        test[4] = '.';
                        index[3]++;
                        if (index[3] == 64) {
                            index[3] = 0;
                            test[3] = '.';
                            index[2]++;
                            if (index[2] == 64) {
                                index[2] = 0;
                                test[2] = '.';
                                index[1]++;
                                if (index[1] == 64) {
                                    index[1] = 0;
                                    test[1] = '.';
                                    index[0]++;
                                } else {
                                    test[1] = alphabet[index[1]];
                                }
                            } else {
                                test[2] = alphabet[index[2]];
                            }
                        } else {
                            test[3] = alphabet[index[3]];
                        }
                    } else {
```

```
                        test[4] = alphabet[index[4]];
                    }
                } else {
                    test[5] = alphabet[index[5]];
                }
            } else {
                test[6] = alphabet[index[6]];
            }
        } else {
            test[7] = alphabet[index[7]];
        }

        i++;
        strncpy(testdes, crypt(test, salt),13);
    }

    // get stop time and calc time_span
    clock_gettime(CLOCK_REALTIME, &t2);
    time_span = (t2.tv_sec - t1.tv_sec) + (double)((t2.tv_nsec - t1.tv_nsec))/1000000000;

    if (strcmp(pwdes, testdes) == 0) {
        printf("Found password %s in %lf secs\n", test, time_span);
    } else {
        printf("Not found, tested %llu in %lf secs\n", n, time_span);
    }
}
```