

Operating Systems

(CL-2006)

Complex Engineering Problem (CEP)

SPRING-2022



**Title: Multi-threaded Brute-force Password
Cracker**

Submitted by: Muhammad Zeeshan Ansar

National University of Computer and Emerging Sciences, Islamabad

Project Title: Multi-threaded Brute-force Password Cracker

Team Lead

Muhammad Zeeshan Ansar (i19-0772)

Project Members

Ahmad Azhar (i19-0832)

Ali Raza (i19-0774)

Date of Submission

Thu, 30 June, 2021

Introduction

Whenever sensitive data such as passwords is stored on a database or any other kind of datastore, it is hashed before saving it to a database. For the purpose of hashing we've many algorithms but SHA-256 algorithm is the most commonly used algorithm. So, we've designed a Linux based multi-threaded application which can effectively crack a SHA-256 hash. Hash cracking means recovering the original data from hashed data. This application is designed using POSIX threads to optimize the cracking speed and to take full advantage of multi-core processor.

Procedure/Design

The whole design of application can be broken down to three main parts which are as follow:

Pattern Generation

To crack a hash, we must have random string patterns of data which will be used in the cracking process. To do this task, we've designed an algorithm which can generate all the possible strings which can be used as passwords in the real world. A thread named *void *T_char_pattern_gen(void *arg)* is used for the sole purpose of generating random string patterns. After the generation of each pattern, that pattern is inserted into queue named as *queue<string> pattern_queue*.

SHA-256 Hash Generation (producer-consumer)

The pattern generator thread keeps on generating patterns and it inserts them into *pattern_queue*. *pattern_queue* is shared resource among pattern generator thread and hash generator threads. Threads named as *void *T_hash_gen(void *arg)* are used for generating SHA-256 hashes based on the patterns generated by the pattern generator thread. Hash generator threads pop the patterns after reading them from the *pattern_queue*. The pattern read from the *pattern_queue* by hash generator threads is used for generating a SHA-256 hash of that pattern. In this way, pattern generator keeps on pushing the data into the pattern as producer of data and hash generator threads consume that data from queue.

After the hash generator threads generate a hash, that hash is pushed into another queue named as *queue<passwd_n_hash> combined_queue*. *combined_queue* stores both the original pattern and its hash in it.

Hash Check

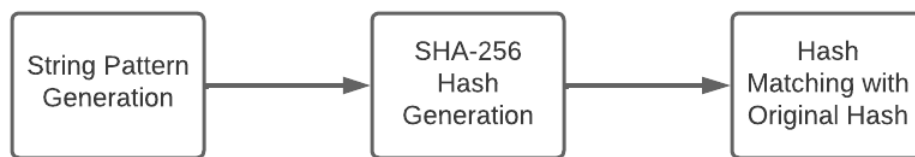
In the previous part, hash generator threads were used to generate hashes of random patterns. Now these hashes are compared with the original hash to check whether the correct hash has been found or not. To do this task, threads named as *void *T_hash_compare(void *arg)* are used. These comparer threads read hashes from *combined_queue* and match them against the original hash. If the program generated hash is correctly matched with the original, then it means the original hash has been

cracked and then program displays the original data which was used to generate that hash.

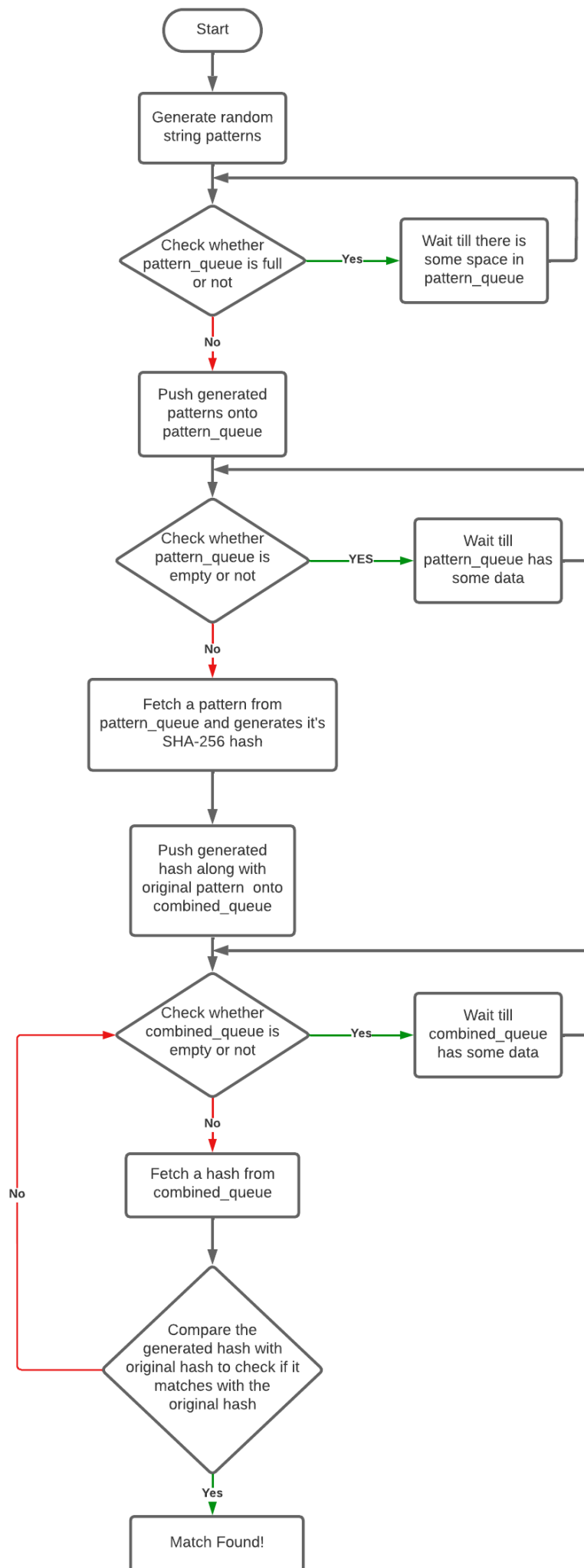
Thread Synchronization

This application has shared resources because of the producer-consumer nature of problem. So, to avoid any race conditions and data loss, we've used Mutex locks as a synchronization tool. These mutex locks are used for locking the critical sections of threads to synchronize data access and modification. Details about mutex lock implementation are provided in the code.

Block Diagram



Flow Chart



Code

```
/**
 * @file final.cpp
 * @author Zeeshan
 * @brief SHA-256 hash cracker
 * program takes input a SHA-256 hash of any string with lenth <= 25 and cracks it
 * Compile with:  g++ -pthread final.cpp -o a
 * Run as:      ./a
 *
 * @version 1
 * @date 2022-05-28
 * @example SHA-256 hash of string "abcd" is
 * 88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
 */

#include <iostream>
#include <string.h>
#include <string>
#include <pthread.h>
#include <unistd.h>
#include <queue>
#include <chrono>
#include <ctime>
using namespace std;

// SHA-256 defines
#define uchar unsigned char
#define uint unsigned int
```

```

#define DBL_INT_ADD(a, b, c)    \
    if (a > 0xffffffff - (c)) \
        ++b;                \
    a += c;

#define ROTLEFT(a, b) (((a) << (b)) | ((a) >> (32 - (b))))
#define ROTRIGHT(a, b) (((a) >> (b)) | ((a) << (32 - (b))))

#define CH(x, y, z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define EP0(x) (ROTRIGHT(x, 2) ^ ROTRIGHT(x, 13) ^ ROTRIGHT(x, 22))
#define EP1(x) (ROTRIGHT(x, 6) ^ ROTRIGHT(x, 11) ^ ROTRIGHT(x, 25))
#define SIG0(x) (ROTRIGHT(x, 7) ^ ROTRIGHT(x, 18) ^ ((x) >> 3))
#define SIG1(x) (ROTRIGHT(x, 17) ^ ROTRIGHT(x, 19) ^ ((x) >> 10))

typedef struct
{
    uchar data[64];
    uint datalen;
    uint bitlen[2];
    uint state[8];
} SHA256_CTX;

uint k[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4,
    0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
    0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc,

```

```
0x76f988da,  
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351,  
0x14292967,  
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e,  
0x92722c85,  
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585,  
0x106aa070,  
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,  
0x682e6ff3,  
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7,  
0xc67178f2};
```

```
// this structure is used for storing both the password and its hash  
// password is a pattern generated by pattern generator thread. This pattern  
// would be hashed using SHA-256 algorithm.
```

```
// the object of this structure will be pushed onto a queue which will be  
// used by another thread for matching against the user provided hash.
```

```
struct passwd_n_hash
```

```
{
```

```
    string passwd;
```

```
    string hash;
```

```
};
```

```
// this structure is a shared resource among threads
```

```
// pattern_queue: this queue is used for storing the random patterns generated by the
```

```
// pattern generator thread. this queue is shared among two types of threads, one is the
```

```
// pattern generator which generates random patterns and push them into this queue and the
```

```
// other ones are hash generator threads. These threads consume the patterns pushed into
```

```
// the pattern_queue.
```



```

// combined_queue is a queue of datatype passwd_n_hash. It is shared among hash generator threads
and
// comparer threads.
// hash generators obtain patterns from pattern_queue, generate their hashed and store them in
// combined_queue along with their sha-256 hashes.
// combined_queue is consumed by comparer threads. These threads compare the hashes generated by
// the program with the user provided hash.

// hash_to_be_cracked is used for storing the user provided hash string which will be compared with
// the program generated hashes.

struct pattern_and_hash_queues
{
    queue<string> pattern_queue;
    queue<passwd_n_hash> combined_queue;
    string hash_to_be_cracked;
};

// GLOBAL VARIABLES
//=====

// BASE_CHAR_ARRAY[] contains all the possible characters which are used in a password
// pattern generator thread will use this array for generating patterns to crack the
// user provided hash
const char BASE_CHAR_ARRAY[] =
"abcdefghijklmnopqrstuvwxyz1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ@#$$%^&*!()-._?
[]_`~:;+=\";

```

```
// total length of the base array
int S_LEN = strlen(BASE_CHAR_ARRAY);

// this variable indicates the total length of crackable password
// if a hashed password is longer than 25 characters than this program
// can't crack it
const int TOTAL_PASSW_LENGTH = 25;

// stores total number of hashes tested against the user provided hash
double TEST_COUNT = 0;

// no of hash generator threads
int hashers = 8;

// no of hash comparer threads
int comparers = 6;

// size limit of pattern_queue
// if the size of pattern_queue has reached 20000, then the pattern generator will
// halt the production of patterns until the size is less than 20000.
// reason for limiting the size of queue is that it will take a lot of memory
// if size is not limited
const int MAX_PATTERN_Q_SIZE = 20000;

// program sleep time is seconds
// if the CPU overheats, then the program will suspend itself for SLEEP_TIME seconds
// after waking up, the program will resume it's execution from the point where it left
```

```

// before suspension
const int SLEEP_TIME = 5;

// this limit is used for preventing CPU overheating
// it indicates the no of hash compares the program has done.
// if the program has done a specific no. of compares then it will suspend for SLEEP_TIME seconds
int sleep_limit_offset = 100000000;

//=====
// Boolean Flags
//=====
// true if all the patterns have been generated using the BASE_CHAR_ARRAY, otherwise
// it is false. It's value is only modified by pattern generator thread.
bool IS_P_LEN_RANGE_COMPLETE = false;

// this is an array of 8 elements, where each element represent a flag for each thread.
// since there are 8 hasher threads, so the size of this array is 8
// each thread will set this flag to true on exiting, which indicates that the
// thread has performed hashing on all the patterns and now it is exiting because there
// are no more patterns remaining in the pattern_queue
bool is_hashing_complete[8];

// this flag represents all the hash generator threads' status
// it's value will be set to true when all elements in is_hashing_complete[] will be true.
// which indicates that all the hash generator threads have done their task.
bool are_hash_thread_complete = false;

// this flag will be set to true upon sucessful hash match

```

```

// it will be used threads to determine whether the hash has been cracked or not.
// if hash has been cracked, this flag will be set to true, and all threads will terminate
// by checking this flag
bool hash_cracked = false;


// this flag for used for temporarily suspending the program execution.
// it will be set to true whenever the CPU overheats.
// all threads will check it's value, if it's value is true, then they will halt their
// themselves and wait for this flag to be set to false.
bool sleep_flag = false;


//=====
// Thread IDs
//=====
// these are the thread IDs
// pattern_gen_tid    TID of pattern generator thread
// hasher_tid[8]      TIDs of hash generator threads
// comparer_tid[8]    TIDs of hash comparer threads
pthread_t pattern_gen_tid;
pthread_t hasher_tid[8];
pthread_t comparer_tid[6];


//=====
// Mutex Locks
//=====
pthread_mutex_t LOCK1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t LOCK2 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t LOCK3 = PTHREAD_MUTEX_INITIALIZER;

```

```

pthread_mutex_t LOCK4 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t HASH_FLAG_SET_LOCK = PTHREAD_MUTEX_INITIALIZER;

//=====
// Function Prototypes
//=====
string conv_to_string(char arr[], int size);
string generate_sha256(string arg);
void SHA256Transform(SHA256_CTX *ctx, uchar data[]);
void SHA256Init(SHA256_CTX *ctx);
void SHA256Update(SHA256_CTX *ctx, uchar data[], uint len);
void SHA256Final(SHA256_CTX *ctx, uchar hash[]);
string SHA256(char *data);
string get_user_input_hash();

//=====
// Thread Functions
//=====
void *T_char_pattern_gen(void *arg);
void *T_hash_gen(void *arg);
void *T_hash_compare(void *arg);

// main
int main()
{
    // display title
    cout << "\033[93;1;5m-----\033[0m" << endl;
    cout << "\033[93;1;5m\t\tBrute-Force Password Cracker\t\t\033[0m" << endl;

```

```

cout << "\033[93;1;5m-----\033[0m" << endl;

string input = get_user_input_hash();

// initially, set all the is_hashing_complete flags to false.
// their value can only be set to true when a hasher thread has completed it's task
for (int i = 0; i < hashers; i++)
    is_hashing_complete[i] = false;

// start measuring the execution time of program from now
auto start = std::chrono::system_clock::now();

// this object contains both pattern queue and hash queue
// a pointer to this object is passed onto every thread
// it is a shared resource among threads and all the accesses are synchronized to
// avoid data loss in this object
pattern_and_hash_queues ph_object;

//ph_object.hash_to_be_cracked = generate_sha256("abcde");
ph_object.hash_to_be_cracked = input;
pthread_attr_t thread_at;
pthread_attr_init(&thread_at);

pthread_create(&pattern_gen_tid, &thread_at, T_char_pattern_gen, &ph_object);

for (int i = 0; i < hashers; i++)
    pthread_create(&hasher_tid[i], &thread_at, T_hash_gen, &ph_object);

```

```

for (int i = 0; i < comparers; i++)
    pthread_create(&comparer_tid[i], &thread_at, T_hash_compare, &ph_object);

pthread_join(pattern_gen_tid, NULL);

for (int i = 0; i < hashers; i++)
    pthread_join(hasher_tid[0], NULL);

for (int i = 0; i < comparers; i++)
    pthread_join(comparer_tid[i], NULL);

// store time at completion
auto end = std::chrono::system_clock::now();

chrono::duration<double> elapsed_seconds = end - start;
time_t end_time = chrono::system_clock::to_time_t(end);

cout << "\n\033[34;1;5mElapsed Time: " << elapsed_seconds.count() << " secs \033[0m" << endl;
}

string get_user_input_hash()
{
    string hash_str;
    cout << "\n\033[34;1;5mEnter a correct SHA256 hash to crack it\033[0m" << endl;
    do
    {

        cout << "\033[34;1;5mSHA256 Hash >> \033[0m";
    }
}

```

```

        cin >> hash_str;
        if (hash_str.length() != 64)
            cout << "\n\033[31;1;5m[Error]. Incorrect Hash, enter again... \033[0m" << endl;
        ;
    } while (!(hash_str.length() == 64));

    return hash_str;
}

void *T_char_pattern_gen(void *arg)
{
    int oldtype;
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);

    struct pattern_and_hash_queues *ptr;
    ptr = (struct pattern_and_hash_queues *)arg;
    string pattern = "";
    char temp[4];
    int p_len = 0;
    for (int counter = 0; counter < TOTAL_PASSW_LENGTH; counter++)
    {
        p_len++;
        for (int i = 0; i < S_LEN; i++)
        {
            temp[0] = BASE_CHAR_ARRAY[i];
            if (p_len == 1)
            {
                pattern = conv_to_string(temp, p_len);
            }
        }
    }
}

```



```

pthread_mutex_lock(&LOCK1);
ptr->pattern_queue.push(pattern);
pthread_mutex_unlock(&LOCK1);

while (true)
{
    if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)
        break;
}
if (sleep_flag == true)
{
    sleep(SLEEP_TIME);
}
}
else
{
    for (int j = 0; j < S_LEN; j++)
    {
        temp[j] = BASE_CHAR_ARRAY[j];
        if (p_len == 2)
        {
            pattern = conv_to_string(temp, p_len);

            pthread_mutex_lock(&LOCK1);
            ptr->pattern_queue.push(pattern);
            pthread_mutex_unlock(&LOCK1);
            while (true)
            {

```

```

        if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)
            break;
    }
    if (sleep_flag == true)
    {
        sleep(SLEEP_TIME);
    }
}
else
{
    for (int k = 0; k < S_LEN; k++)
    {
        temp[2] = BASE_CHAR_ARRAY[k];
        if (p_len == 3)
        {
            pattern = conv_to_string(temp, p_len);
            pthread_mutex_lock(&LOCK1);
            ptr->pattern_queue.push(pattern);

            pthread_mutex_unlock(&LOCK1);
            while (true)
            {
                if (ptr->pattern_queue.size() <
MAX_PATTERN_Q_SIZE)
                    break;
            }
            if (sleep_flag == true)
            {

```

```

        sleep(SLEEP_TIME);
    }
}
else
{
    for (int l = 0; l < S_LEN; l++)
    {
        temp[3] = BASE_CHAR_ARRAY[l];
        if (p_len == 4)
        {
            pattern = conv_to_string(temp, p_len);
            pthread_mutex_lock(&LOCK1);
            ptr->pattern_queue.push(pattern);

            pthread_mutex_unlock(&LOCK1);

            while (true)
            {
                if (ptr->pattern_queue.size() <
MAX_PATTERN_Q_SIZE)
                    break;
            }
            if (sleep_flag == true)
            {
                sleep(SLEEP_TIME);
            }
        }
    }
    else

```

```
{
    for (int i1 = 0; i1 < S_LEN; i1++)
    {
        temp[4] = BASE_CHAR_ARRAY[i1];
        if (p_len == 5)
        {
            pattern = conv_to_string(temp, p_len);
            pthread_mutex_lock(&LOCK1);
            ptr->pattern_queue.push(pattern);

            pthread_mutex_unlock(&LOCK1);

            while (true)
            {
                if (ptr->pattern_queue.size() <
MAX_PATTERN_Q_SIZE)
                    break;
            }

            if (sleep_flag == true)
            {
                sleep(SLEEP_TIME);
            }
        }
        else
        {
            for (int i2 = 0; i2 < S_LEN; i2++)
            {
```

```
BASE_CHAR_ARRAY[i2];
```

```
p_len);
```

```
pthread_mutex_unlock(&LOCK1);
```

```
< MAX_PATTERN_Q_SIZE)
```

```
temp[5] =
```

```
if (p_len == 6)
```

```
{
```

```
    pattern = conv_to_string(temp,
```

```
    pthread_mutex_lock(&LOCK1);
```

```
    ptr->pattern_queue.push(pattern);
```

```
while (true)
```

```
{
```

```
    if (ptr->pattern_queue.size()
```

```
        break;
```

```
}
```

```
if (sleep_flag == true)
```

```
{
```

```
    sleep(SLEEP_TIME);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
    for (int i3 = 0; i3 < S_LEN; i3++)
```

```
    {
```

```
        temp[6] =
```

```
BASE_CHAR_ARRAY[i3];
```

```
conv_to_string(temp, p_len);
```

```
pthread_mutex_lock(&LOCK1);
```

```
>pattern_queue.push(pattern);
```

```
pthread_mutex_unlock(&LOCK1);
```

```
>pattern_queue.size() < MAX_PATTERN_Q_SIZE)
```

```
sleep(SLEEP_TIME);
```

```
S_LEN; i4++)
```

```
if (p_len == 7)
```

```
{
```

```
    pattern =
```

```
    ptr-
```

```
while (true)
```

```
{
```

```
    if (ptr-
```

```
        break;
```

```
}
```

```
if (sleep_flag == true)
```

```
{
```

```
}
```

```
}
```

```
else
```

```
{
```

```
    for (int i4 = 0; i4 <
```

	{
	temp[7] =
BASE_CHAR_ARRAY[i4];	if (p_len == 8)
	{
	pattern =
conv_to_string(temp, p_len);	
pthread_mutex_lock(&LOCK1);	ptr-
>pattern_queue.push(pattern);	
pthread_mutex_unlock(&LOCK1);	
	while (true)
	{
	if (ptr-
>pattern_queue.size() < MAX_PATTERN_Q_SIZE)	
break;	}
	if (sleep_flag
== true)	{
sleep(SLEEP_TIME);	}
	}
	else

```

{
    for (int i5 =
0; i5 < S_LEN; i5++)
    {
        temp[8]
        = BASE_CHAR_ARRAY[i5];
        if (p_len
        == 9)
        {
            pattern = conv_to_string(temp, p_len);
            pthread_mutex_lock(&LOCK1);
            ptr->pattern_queue.push(pattern);
            pthread_mutex_unlock(&LOCK1);
        }
        while (true)
        {
            if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)
            {
                break;
            }
            if
            (sleep_flag == true)
            {

```



```
sleep(SLEEP_TIME);

    }

}

else
{
    for
    {

temp[9] = BASE_CHAR_ARRAY[i6];

if (p_len == 10)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;
```

```
}
```

```
if (sleep_flag == true)
```

```
{
```

```
sleep(SLEEP_TIME);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
for (int i7 = 0; i7 < S_LEN; i7++)
```

```
{
```

```
temp[10] = BASE_CHAR_ARRAY[i7];
```

```
if (p_len == 11)
```

```
{
```

```
pattern = conv_to_string(temp, p_len);
```

```
pthread_mutex_lock(&LOCK1);
```

```
ptr->pattern_queue.push(pattern);
```

```
pthread_mutex_unlock(&LOCK1);
```

```
while (true)
```

```
{
```

```
if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)
```

```
break;
```

```
}
```

```
if (sleep_flag == true)
```

```
{
```

```
sleep(SLEEP_TIME);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
for (int i8 = 0; i8 < S_LEN; i8++)
```

```
{
```

```
temp[11] = BASE_CHAR_ARRAY[i8];

if (p_len == 12)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);


while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}


if (sleep_flag == true)

{
```

```
sleep(SLEEP_TIME);

}

}

else

{

for (int i9 = 0; i9 < S_LEN; i9++)

{

temp[12] = BASE_CHAR_ARRAY[i9];

if (p_len == 13)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)
```

```
{  
  
if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)  
  
break;  
  
}
```

```
if (sleep_flag == true)
```

```
{  
  
sleep(SLEEP_TIME);  
  
}
```

```
}
```

```
else
```

```
{
```

```
for (int i10 = 0; i10 < S_LEN; i10++)
```

```
{
```

```
temp[13] = BASE_CHAR_ARRAY[i10];
```

```
if (p_len == 14)
```

```
{
```

```
pattern = conv_to_string(temp, p_len);
```

```
pthread_mutex_lock(&LOCK1);
```

```
ptr->pattern_queue.push(pattern);
```

```
pthread_mutex_unlock(&LOCK1);
```

```
while (true)
```

```
{
```

```
if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)
```

```
break;
```

```
}
```

```
if (sleep_flag == true)
```

```
{
```

```
sleep(SLEEP_TIME);
```

```
}
```

```
}
```

```
else
```

```
{

for (int i11 = 0; i11 < S_LEN; i11++)

{

temp[14] = BASE_CHAR_ARRAY[i11];

if (p_len == 15)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;
```



```
}

if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

else

{

for (int i12 = 0; i12 < S_LEN; i12++)

{

temp[15] = BASE_CHAR_ARRAY[i12];

if (p_len == 16)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);
```

```
pthread_mutex_unlock(&LOCK1);
```

```
while (true)
```

```
{
```

```
if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)
```

```
break;
```

```
}
```

```
if (sleep_flag == true)
```

```
{
```

```
sleep(SLEEP_TIME);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
for (int i13 = 0; i13 < S_LEN; i13++)
```

```
{

temp[16] = BASE_CHAR_ARRAY[i13];

if (p_len == 17)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);


while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}

if (sleep_flag == true)
```

```
{  
  
sleep(SLEEP_TIME);  
  
}  
  
}  
  
else  
  
{  
  
for (int i14 = 0; i14 < S_LEN; i14++)  
  
{  
  
temp[17] = BASE_CHAR_ARRAY[i14];  
  
if (p_len == 18)  
  
{  
  
pattern = conv_to_string(temp, p_len);  
  
pthread_mutex_lock(&LOCK1);  
  
ptr->pattern_queue.push(pattern);  
  
pthread_mutex_unlock(&LOCK1);  
  
while (true)
```

```
{  
  
if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)  
  
break;  
  
}  
  
if (sleep_flag == true)  
  
{  
  
sleep(SLEEP_TIME);  
  
}  
  
}  
  
else  
  
{  
  
for (int i15 = 0; i15 < S_LEN; i15++)  
  
{  
  
temp[18] = BASE_CHAR_ARRAY[i15];  
  
if (p_len == 19)  
  
{
```

```
pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}

if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

else
```

```
{

for (int i16 = 0; i16 < S_LEN; i16++)

{

temp[19] = BASE_CHAR_ARRAY[i16];

if (p_len == 20)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}
```

```
if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

else

{

for (int i17 = 0; i17 < S_LEN; i17++)

{

temp[20] = BASE_CHAR_ARRAY[i17];

if (p_len == 21)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);
```



```
while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}


if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

else

{

for (int i18 = 0; i18 < S_LEN; i18++)

{

temp[21] = BASE_CHAR_ARRAY[i18];

if (p_len == 22)
```

```
{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}

if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}
```

```
else

{

for (int i19 = 0; i19 < S_LEN; i19++)

{

temp[22] = BASE_CHAR_ARRAY[i19];

if (p_len == 23)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}
```

```
if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

else

{

for (int i20 = 0; i20 < S_LEN; i20++)

{

temp[23] = BASE_CHAR_ARRAY[i20];

if (p_len == 24)

{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);
```

```
while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}


if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

else

{

for (int i21 = 0; i21 < S_LEN; i21++)

{

temp[24] = BASE_CHAR_ARRAY[i21];

if (p_len == 25)
```

```
{

pattern = conv_to_string(temp, p_len);

pthread_mutex_lock(&LOCK1);

ptr->pattern_queue.push(pattern);

pthread_mutex_unlock(&LOCK1);

while (true)

{

if (ptr->pattern_queue.size() < MAX_PATTERN_Q_SIZE)

break;

}

if (sleep_flag == true)

{

sleep(SLEEP_TIME);

}

}

}
```

}

}

}

}

}

}

}

}

}

}

}

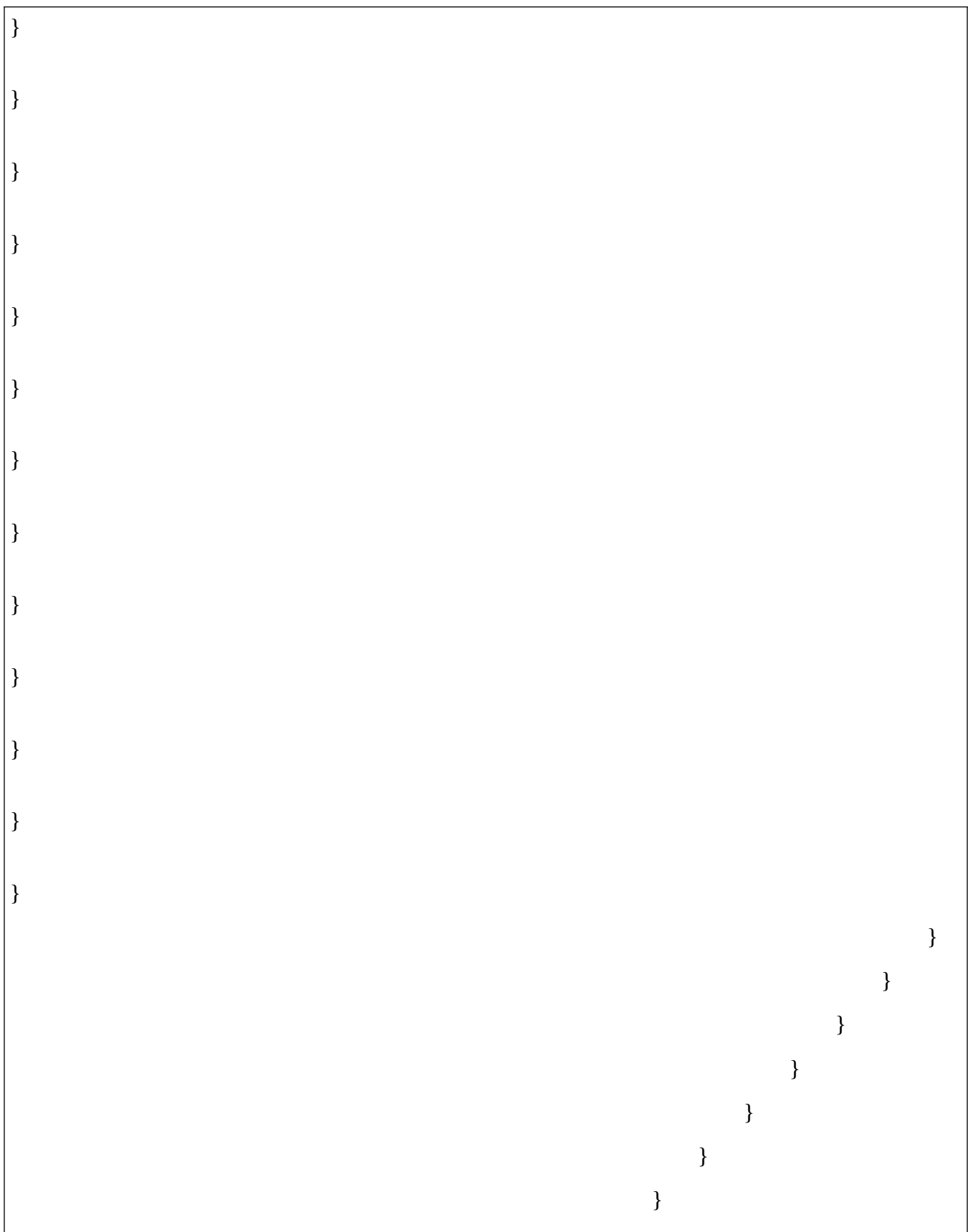
}

}

}

}

}




```

    }
}

IS_P_LEN_RANGE_COMPLETE = true; // indicates that all the patterns have been generated
pthread_exit(NULL);
}

// this function takes char array as input and returns a string of that array
string conv_to_string(char arr[], int size)
{
    string tmp = "";
    for (int i = 0; i < size; i++)
        tmp = tmp + arr[i];
    return tmp;
}

/**
 * @brief This function generates SHA-256 hashes based on the random patterns generated by

```

T_char_pattern_gen thread

*

* @param arg argument is a pointer to ph_object

* @return void* this function returns NULL

*/

void *T_hash_gen(void *arg)

{

 // set the cancellation type to asynchronous

 int oldtype;

 pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);

 // type casting argument pointer

 struct pattern_and_hash_queues *ptr;

 ptr = (struct pattern_and_hash_queues *)arg;

 // pulled data from pattern_queue will be stored in temp_str

 string temp_str;

 // generated SHA-256 hash of a string stored in temp_str will be stored in hash_str

 string hash_str;

 // object used for storing both the temp_str and hash_str in struct_obj

 passwd_n_hash struct_obj;

 /**

 * this loop will terminate when two conditions are true:

 * pattern_queue is empty and IS_P_LEN_RANGE_COMPLETE is true

```

*
* IS_P_LEN_RANGE_COMPLETE flag indicates whether the pattern generation is complete or
not
* if this flag is false, then it means that the pattern generator thread is still busy producing
* the patterns so the hash generator threads must keep while loop running. because if patterns are
hash
* still produced then the hash generators must wait for them to be available in pattern_queue to
* them.
*
* if both of the above mentioned flags are true, then it means that the pattern_queue is empty and
there
* are no more patterns left to be produced.
*/

while (!(ptr->pattern_queue.empty() & IS_P_LEN_RANGE_COMPLETE))
{
    /**
    * check again whether the queue is empty or not
    * if it is empty, then don't perform any operations on it as it will cause an error
    */
    if (!(ptr->pattern_queue.empty()))
    {
        /**
        * @brief Critical Section CS-1
        * Critical Section CS-1 covers the modification operations on pattern_queue
        * as the pattern_queue is shared among pattern generator thread and hash generator
threads
        * so the access to this queue must be synchronized.
        * LOCK1 variable is used for synchronizing all the access to pattern_queue

```

generated

```
*
* here the hash generator threads are data consumers which are consuming pattern data
* by pattern generator thread
*/

// START OF CRITICAL SECTION CS-1
pthread_mutex_lock(&LOCK1);
if (!(ptr->pattern_queue.empty()))
{
    temp_str = ptr->pattern_queue.front();
    ptr->pattern_queue.pop();
}
pthread_mutex_unlock(&LOCK1);
// END OF CRITICAL SECTION CS-1

// call function to generate hash
hash_str = generate_sha256(temp_str);

// store original string and hashed string into combined queue
struct_obj.passwd = temp_str;
struct_obj.hash = hash_str;

/**
* @brief Critical Section CS-2
* Critical Section CS-2 covers all the operations on combined_queue
* combined_queue is shared among hash generator threads and comparer threads so
* the access to this queue has been synchronized through mutex LOCK2
*/
```

```

// START OF SECOND CRITICAL SECTION CS-2

pthread_mutex_lock(&LOCK2);
ptr->combined_queue.push(struct_obj);
pthread_mutex_unlock(&LOCK2);
// END OF SECOND CRITICAL SECTION CS-2


/**
 * @brief Terminate thread if hash_cracked is true
 * hash_cracked flag indicates whether the user provided hash has been
 * cracked or not. if it's value is true, then it means the hash has
 * been cracked else it has not been cracked yet
 *
 * if the hash_cracked flag is true, then only the thread with TID of
 * hash_tid[0] will cancel the pattern generator thread and then it will
 * exit with NULL pointer. All other threads will not try to cancel pattern
 * generator thread. They will only terminate themselves with NULL pointers.
 *
 */
if (hash_cracked == true)
{
    if (pthread_equal(hasher_tid[0], pthread_self()) > 0)
        pthread_cancel(pattern_gen_tid);

    pthread_exit(NULL);
}


/**
 * the program waits here when the sleep_flag is true

```

```

        * otherwise the program comes out of while loop using break; statement
        * continue it's normal execution
    */
    while (true)
    {
        if (sleep_flag != true)
        {
            break;
        }
        else
            sleep(2);
    }
}

// when a hash generator thread has completed it's task, then set it's
// is_hashing_complete[i] to true
for (int i = 0; i < hashers; i++)
{
    if (pthread_equal(hasher_tid[i], pthread_self()) > 0)
        is_hashing_complete[i] = true;
}

/**
 * @brief Critical Section HASH-FLAG
 * are_hash_thread_complete flag indicates whether the hash generators have completed
 * their tasks or they are still doing it.
 * this flag will be true only when all the hash generator have set their individual

```

```

    * flags in is_hashing_complete[] array to true.
    */

    pthread_mutex_lock(&HASH_FLAG_SET_LOCK);
    for (int i = 0; i < hashers; i++)
    {
        if (is_hashing_complete[i] == true)
        {
            are_hash_thread_complete = true;
        }
        else
        {
            are_hash_thread_complete = false;
            pthread_exit(NULL);
        }
    }
    pthread_mutex_unlock(&HASH_FLAG_SET_LOCK);

    pthread_exit(NULL);
}

void *T_hash_compare(void *arg)
{
    // cout << "Comparer workign\n";
    struct pattern_and_hash_queues *ptr;
    ptr = (struct pattern_and_hash_queues *)arg;

    passwd_n_hash temp_obj;

```

```

string hashed_passwd;

string temp;

int prev_val = 0;

int sleep_lim = 0;

while (!(ptr->combined_queue.empty() & are_hash_thread_complete))
{
    if (!(ptr->combined_queue.empty()))
    {
        // START OF SECOND CRITICAL SECTION CS-2
        pthread_mutex_lock(&LOCK2);
        if (!(ptr->combined_queue.empty()))
        {
            temp_obj = ptr->combined_queue.front();
            ptr->combined_queue.pop();
        }
        pthread_mutex_unlock(&LOCK2);
        // END OF SECOND CRITICAL SECTION CS-2

        hashed_passwd = temp_obj.hash;
        if (hashed_passwd == ptr->hash_to_be_cracked)
        {
            cout << "\n\033[94;1;4mHash Cracked\033[0m\n";
            cout << "\n\033[94;1;5mPassword: \033[0m";
            cout << "\033[92;1;5m" << temp_obj.passwd << "\033[0m" << endl;
            pthread_mutex_lock(&LOCK4);
            hash_cracked = true;
            pthread_mutex_unlock(&LOCK4);
        }
    }
}

```



```

    }

    if (hash_cracked == true )
    {
        pthread_exit(NULL);
    }

    while (true)
    {
        if (sleep_flag != true)
        {
            break;
        }
        else
        {
            sleep(10);
            if (pthread_equal(comparer_tid[0], pthread_self()) > 0)
            {
                sleep(SLEEP_TIME);
                pthread_mutex_lock(&LOCK4);
                sleep_flag = false;
                pthread_mutex_unlock(&LOCK4);
            }
        }
    }

    if (pthread_equal(comparer_tid[0], pthread_self()) > 0)
    {

```

```

        pthread_mutex_lock(&LOCK3);
        TEST_COUNT++;
        if (TEST_COUNT > (prev_val + 37805))
        {
            cout << "\033[92;21;24mTotal patterns checked: " << TEST_COUNT << "\033[0m" << endl;
            prev_val = TEST_COUNT;
        }
        if (TEST_COUNT > sleep_lim + sleep_limit_offset)
        {
            cout << "\n\033[31;1;4mWARNING: CPU Overheating:\033[0m ";
            cout << "\033[31;1;24mSuspending for almost " << SLEEP_TIME << "Seconds\033[0m\n";
            pthread_mutex_lock(&LOCK4);
            sleep_flag = true;
            pthread_mutex_unlock(&LOCK4);

            sleep_lim = TEST_COUNT;
        }
        pthread_mutex_unlock(&LOCK3);
    }
}

pthread_exit(NULL);
}

```

```

string generate_sha256(string arg)

```

```

{

    // assigning value to string s
    string s = arg;

    int n = s.length();

    // declaring character array
    char pass_array[n + 1];

    // copying the contents of the
    // string to char array
    strcpy(pass_array, s.c_str());

    string hash = SHA256(pass_array);
    return hash;
}

//=====
// SHA-256 Algorithm
//=====

void SHA256Transform(SHA256_CTX *ctx, uchar data[])
{
    uint a, b, c, d, e, f, g, h, i, j, t1, t2, m[64];

    for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);
}

```

```
for (; i < 64; ++i)
    m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];
```

```
a = ctx->state[0];
```

```
b = ctx->state[1];
```

```
c = ctx->state[2];
```

```
d = ctx->state[3];
```

```
e = ctx->state[4];
```

```
f = ctx->state[5];
```

```
g = ctx->state[6];
```

```
h = ctx->state[7];
```

```
for (i = 0; i < 64; ++i)
```

```
{
```

```
    t1 = h + EP1(e) + CH(e, f, g) + k[i] + m[i];
```

```
    t2 = EP0(a) + MAJ(a, b, c);
```

```
    h = g;
```

```
    g = f;
```

```
    f = e;
```

```
    e = d + t1;
```

```
    d = c;
```

```
    c = b;
```

```
    b = a;
```

```
    a = t1 + t2;
```

```
}
```

```
ctx->state[0] += a;
```

```
ctx->state[1] += b;
```

```
    ctx->state[2] += c;
    ctx->state[3] += d;
    ctx->state[4] += e;
    ctx->state[5] += f;
    ctx->state[6] += g;
    ctx->state[7] += h;
}

void SHA256Init(SHA256_CTX *ctx)
{
    ctx->datalen = 0;
    ctx->bitlen[0] = 0;
    ctx->bitlen[1] = 0;
    ctx->state[0] = 0x6a09e667;
    ctx->state[1] = 0xbb67ae85;
    ctx->state[2] = 0x3c6ef372;
    ctx->state[3] = 0xa54ff53a;
    ctx->state[4] = 0x510e527f;
    ctx->state[5] = 0x9b05688c;
    ctx->state[6] = 0x1f83d9ab;
    ctx->state[7] = 0x5be0cd19;
}

void SHA256Update(SHA256_CTX *ctx, uchar data[], uint len)
{
    for (uint i = 0; i < len; ++i)
    {
        ctx->data[ctx->datalen] = data[i];
```

```

        ctx->datalen++;
        if (ctx->datalen == 64)
        {
            SHA256Transform(ctx, ctx->data);
            DBL_INT_ADD(ctx->bitlen[0], ctx->bitlen[1], 512);
            ctx->datalen = 0;
        }
    }
}

```

```

void SHA256Final(SHA256_CTX *ctx, uchar hash[])

```

```

{
    uint i = ctx->datalen;

    if (ctx->datalen < 56)
    {
        ctx->data[i++] = 0x80;

        while (i < 56)
            ctx->data[i++] = 0x00;
    }
    else
    {
        ctx->data[i++] = 0x80;

        while (i < 64)
            ctx->data[i++] = 0x00;
    }
}

```

```

    SHA256Transform(ctx, ctx->data);

    memset(ctx->data, 0, 56);
}

DBL_INT_ADD(ctx->bitlen[0], ctx->bitlen[1], ctx->datalen * 8);
ctx->data[63] = ctx->bitlen[0];
ctx->data[62] = ctx->bitlen[0] >> 8;
ctx->data[61] = ctx->bitlen[0] >> 16;
ctx->data[60] = ctx->bitlen[0] >> 24;
ctx->data[59] = ctx->bitlen[1];
ctx->data[58] = ctx->bitlen[1] >> 8;
ctx->data[57] = ctx->bitlen[1] >> 16;
ctx->data[56] = ctx->bitlen[1] >> 24;
SHA256Transform(ctx, ctx->data);

for (i = 0; i < 4; ++i)
{
    hash[i] = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 4] = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 8] = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 20] = (ctx->state[5] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 24] = (ctx->state[6] >> (24 - i * 8)) & 0x000000ff;
    hash[i + 28] = (ctx->state[7] >> (24 - i * 8)) & 0x000000ff;
}
}

```

```
string SHA256(char *data)
{
    int strLen = strlen(data);
    SHA256_CTX ctx;
    unsigned char hash[32];
    string hashStr = "";

    SHA256Init(&ctx);
    SHA256Update(&ctx, (unsigned char *)data, strLen);
    SHA256Final(&ctx, hash);

    char s[3];
    for (int i = 0; i < 32; i++)
    {
        sprintf(s, "%02x", hash[i]);
        hashStr += s;
    }

    return hashStr;
}
```

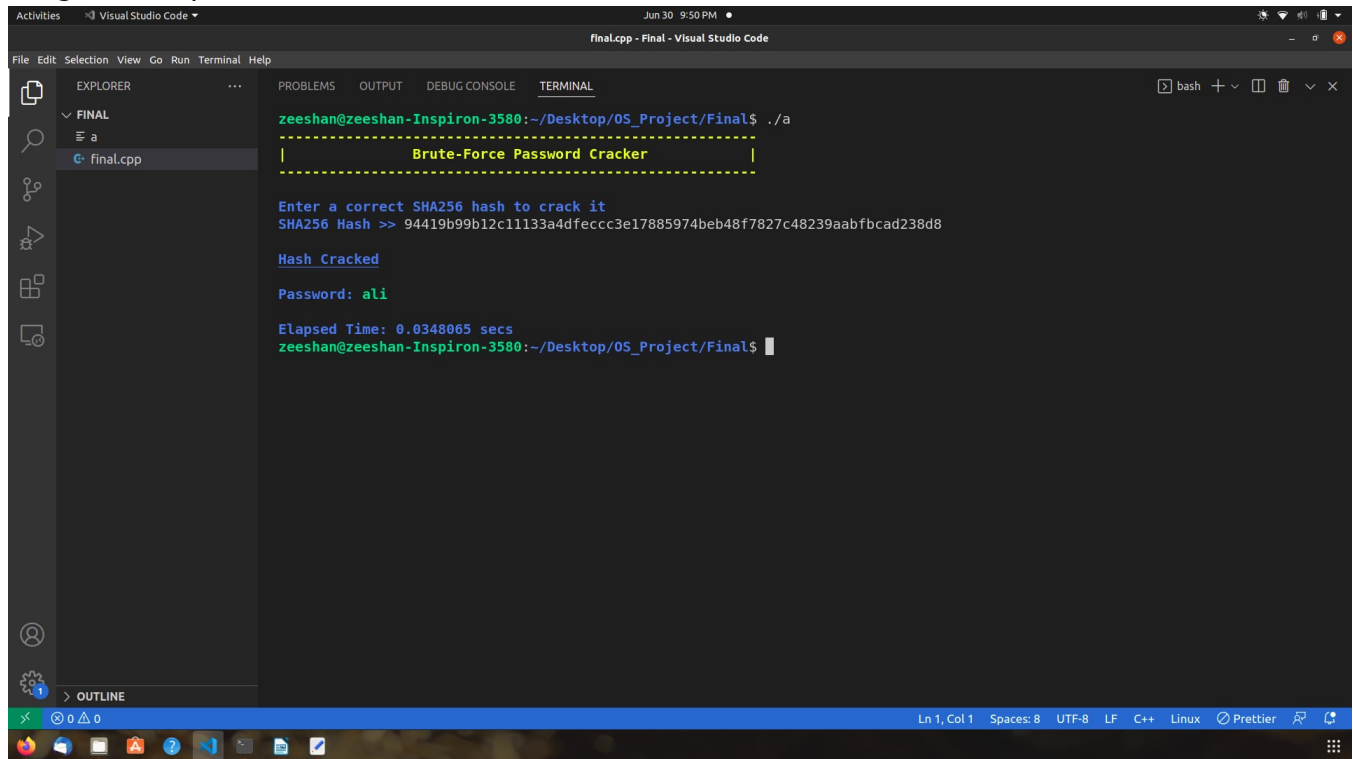

Results

Test Run - 1

Hash of string "ali":

94419b99b12c11133a4dfec3e17885974beb48f7827c48239aabfbcad238d8

Program output



The screenshot shows the Visual Studio Code interface with a terminal window open. The terminal displays the output of a program named 'final.cpp'. The program prompts the user to enter a correct SHA256 hash to crack it. The user enters the hash '94419b99b12c11133a4dfec3e17885974beb48f7827c48239aabfbcad238d8'. The program then outputs 'Hash Cracked' and 'Password: ali'. The elapsed time is shown as 0.0348065 secs. The terminal window title is 'final.cpp - Final - Visual Studio Code'. The status bar at the bottom shows 'Ln 1, Col 1', 'Spaces: 8', 'UTF-8', 'LF', 'C++', 'Linux', and 'Prettier'.

```
zeeshan@zeeshan-Inspiron-3580:~/Desktop/OS_Project/Final$ ./a
-----
          Brute-Force Password Cracker
-----

Enter a correct SHA256 hash to crack it
SHA256 Hash >> 94419b99b12c11133a4dfec3e17885974beb48f7827c48239aabfbcad238d8

Hash Cracked

Password: ali

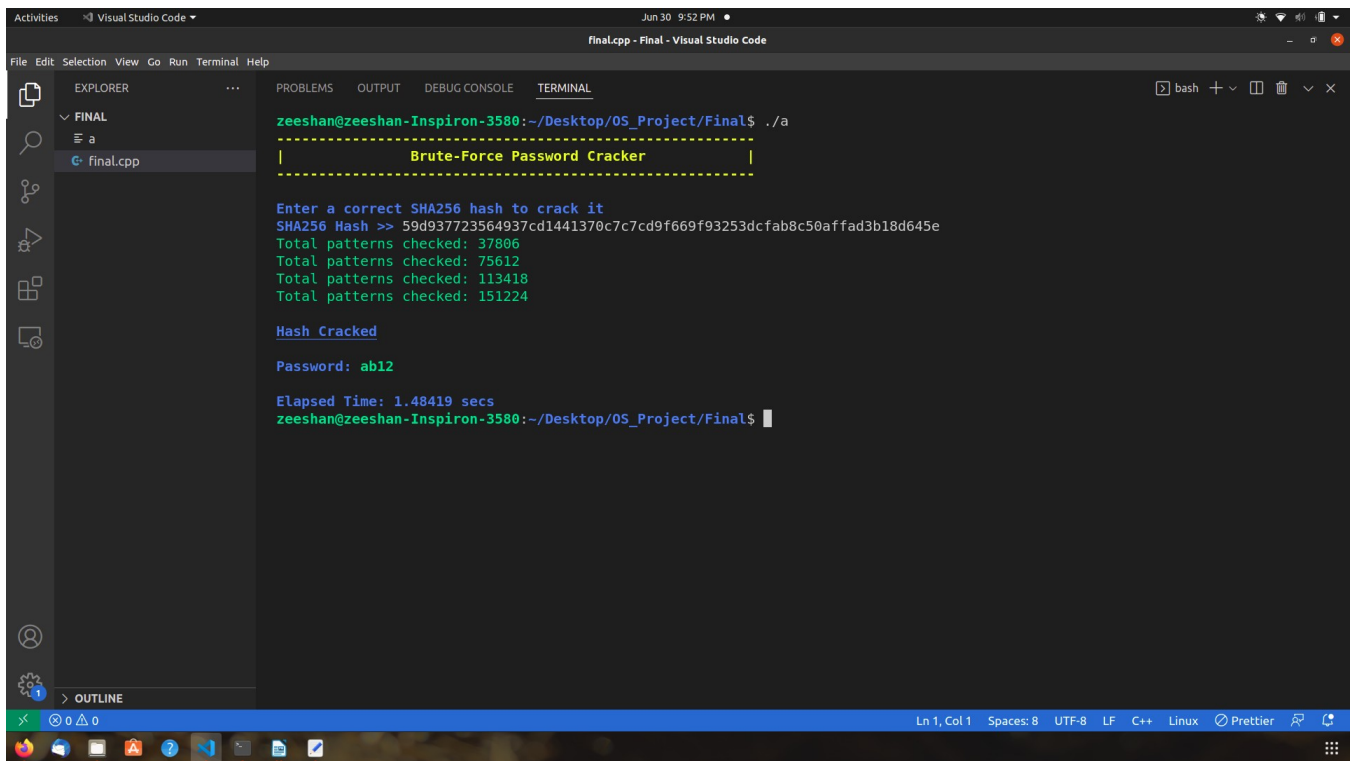
Elapsed Time: 0.0348065 secs
zeeshan@zeeshan-Inspiron-3580:~/Desktop/OS_Project/Final$
```

Test Run - 2

Hash of string "ab12":

59d937723564937cd1441370c7c7cd9f669f93253dcfab8c50affad3b18d645e

Program Output



The screenshot shows the Visual Studio Code interface with a terminal window open. The terminal displays the output of a program named 'Brute-Force Password Cracker'. The program prompts the user to enter a correct SHA256 hash to crack it. The user enters the hash '59d937723564937cd1441370c7c7cd9f669f93253dcfab8c50affad3b18d645e'. The program then checks a series of patterns, displaying the total patterns checked at each step: 37806, 75612, 113418, and finally 151224. The program successfully cracks the hash and displays the password 'ab12'. The elapsed time for the process is 1.48419 seconds. The terminal window is titled 'final.cpp - Final - Visual Studio Code' and the file explorer on the left shows the file 'final.cpp'.

```
zeeshan@zeeshan-Inspiron-3580:~/Desktop/OS_Project/Final$ ./a
-----
                Brute-Force Password Cracker
                -----

Enter a correct SHA256 hash to crack it
SHA256 Hash >> 59d937723564937cd1441370c7c7cd9f669f93253dcfab8c50affad3b18d645e
Total patterns checked: 37806
Total patterns checked: 75612
Total patterns checked: 113418
Total patterns checked: 151224

Hash Cracked

Password: ab12

Elapsed Time: 1.48419 secs
zeeshan@zeeshan-Inspiron-3580:~/Desktop/OS_Project/Final$
```

Test Run – 3

Hash of string “abbas”:

66cc6f6a245ee3a70ff54c9e96319aa85e6d57deba6a67929d5977dc30a0e12

Program Output

Activities Jun 30 9:56 PM

finalLcpp - Final - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

FINAL

a

finalLcpp

```
zeeshan@zeeshan-Inspiron-3580:~/Desktop/OS_Project/Finals$ ./a
-----
      Brute-Force Password Cracker
-----

Enter a correct SHA256 hash to crack it
SHA256 Hash >> 66cc6f6a245ee3a70ff54c9e96319aa85e6d57debaf6a67929d5977dc30a0e12
Total patterns checked: 37806
Total patterns checked: 75612
Total patterns checked: 113418
Total patterns checked: 151224
Total patterns checked: 189030
Total patterns checked: 226836
Total patterns checked: 264642
Total patterns checked: 302448
Total patterns checked: 340254
Total patterns checked: 378060
Total patterns checked: 415866
Total patterns checked: 453672
Total patterns checked: 491478
Total patterns checked: 529284
Total patterns checked: 567090
Total patterns checked: 604896
Total patterns checked: 642702
Total patterns checked: 680508
Total patterns checked: 718314
Total patterns checked: 756120
Total patterns checked: 793926
Total patterns checked: 831732
Total patterns checked: 869538
Total patterns checked: 907344
Total patterns checked: 945150
Total patterns checked: 982956
Total patterns checked: 1.02076e+06
Total patterns checked: 1.05857e+06
```

Ln 1, Col 1 Spaces: 8 UTF-8 LF C++ Linux Prettier

Activities Jun 30 10:00 PM

finalLcpp - Final - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

FINAL

a

finalLcpp

```
Total patterns checked: 1.18711e+07
Total patterns checked: 1.19009e+07
Total patterns checked: 1.19467e+07
Total patterns checked: 1.19845e+07
Total patterns checked: 1.20223e+07
Total patterns checked: 1.20601e+07
Total patterns checked: 1.20979e+07
Total patterns checked: 1.21357e+07
Total patterns checked: 1.21735e+07
Total patterns checked: 1.22113e+07
Total patterns checked: 1.22491e+07
Total patterns checked: 1.2287e+07
Total patterns checked: 1.23248e+07
Total patterns checked: 1.23626e+07
Total patterns checked: 1.24004e+07
Total patterns checked: 1.24382e+07
Total patterns checked: 1.2476e+07
Total patterns checked: 1.25138e+07
Total patterns checked: 1.25516e+07
Total patterns checked: 1.25894e+07
Total patterns checked: 1.26272e+07
Total patterns checked: 1.2665e+07
Total patterns checked: 1.27028e+07
Total patterns checked: 1.27406e+07
Total patterns checked: 1.27784e+07
Total patterns checked: 1.28162e+07
Total patterns checked: 1.2854e+07
Total patterns checked: 1.28918e+07
Total patterns checked: 1.29297e+07
Total patterns checked: 1.29675e+07
Total patterns checked: 1.30053e+07
Total patterns checked: 1.30431e+07
Total patterns checked: 1.30809e+07
Total patterns checked: 1.31187e+07

Hash Cracked
Password: abbas

Elapsed Time: 159.752 secs
zeeshan@zeeshan-Inspiron-3580:~/Desktop/OS_Project/Finals$
```

Ln 1, Col 1 Spaces: 8 UTF-8 LF C++ Linux Prettier

Discussion of Results

Password Cracker program takes a SHA-256 hash as input and starts brute-forcing it to crack it. Hashes of five or more than five character long strings takes more time because a large number of pattern is required to brute-force it. This number is usually in millions. So, the program tries millions of hashes against the original hash to crack it.

Conclusion

Problems

We faced the race condition problems in our program during the early development stages. As we added more functionality in our application, more race condition problems arosed. The reason for these problems were shared resources such as queues and producer-consumer relationship among the threads. Because there were around 15 threads in the program, the major problem was the synchronization among them.

Solution

To solve the synchronization problem, we identified all the critical sections in the program and then used Mutex Locks to prevent race conditions. We've used around four mutex locks in the whole program.

To solve the producer-consumer problem, first, we pointed out the code where the producer-consumer relationship existed and then we used programming logic to solve this problem.

Attachments

Program test runs are provided in the *Results* section

Flow chart

