**Operating Systems – Lab 08**

**Name:** Hafsa Salman

**Roll no.** 22K-5161

Code Workout 02

**Part A**

Code:

#include<stdio.h>

#include<pthread.h>

static volatile int counter = 0;

void \*mythread(void \*arg)

{

printf("%s: Begin\n", (char \*)arg);

int i;

//int counter = 0;

for (i=0; i<1e7; i++)

{

counter = counter + 1;

}

printf("%s: Done. Counter = %d\n", (char \*)arg, counter);

return NULL;

}

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

{

pthread\_t p1, p2;

printf("Main: Begin (Counter = %d)\n", counter);

pthread\_create(&p1, NULL, mythread, "A");

pthread\_create(&p2, NULL, mythread, "B");

pthread\_join(p1, NULL);

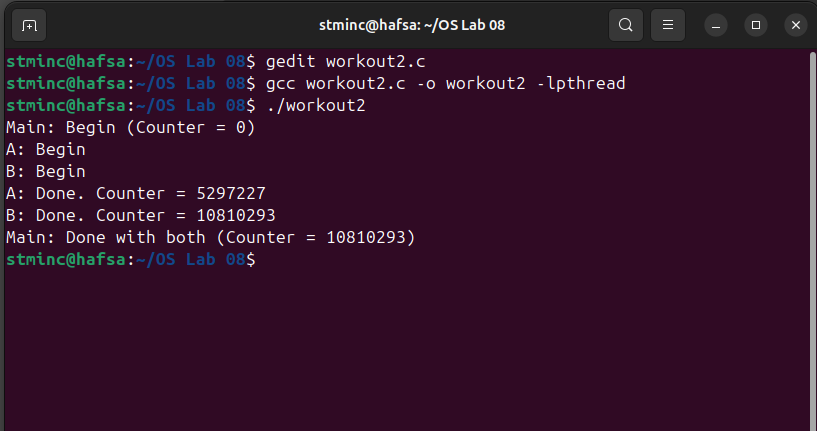
pthread\_join(p2, NULL);

printf("Main: Done with both (Counter = %d)\n", counter);

return 0;

}

Output:



**Part B**

Both threads show different values for counter because they share the same global variable counter.

**Part C**

Code:

#include<stdio.h>

#include<pthread.h>

static volatile int counter = 0;

void \*mythread(void \*arg)

{

printf("%s: Begin\n", (char \*)arg);

int i;

int counter = 0;

for (i=0; i<1e7; i++)

{

counter = counter + 1;

}

printf("%s: Done. Counter = %d\n", (char \*)arg, counter);

return NULL;

}

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

{

pthread\_t p1, p2;

printf("Main: Begin (Counter = %d)\n", counter);

pthread\_create(&p1, NULL, mythread, "A");

pthread\_create(&p2, NULL, mythread, "B");

pthread\_join(p1, NULL);

pthread\_join(p2, NULL);

printf("Main: Done with both (Counter = %d)\n", counter);

return 0;

}

Output:



Explanation:

The counter in this code is being used as a local variable. Each thread gets their own local variable resulting in the same answer.

In-Lab Questions

**Question no. 01: Part A**

Code:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 10000000

int main()

{

float \*A = malloc(N \* sizeof(float));

float \*B = malloc(N \* sizeof(float));

float \*C = malloc(N \* sizeof(float));

if (A == NULL || B == NULL || C == NULL)

{

fprintf(stderr, "Memory allocation failed\n");

return 1;

}

for (int i = 0; i < N; i++)

{

A[i] = i \* 0.5f;

B[i] = i \* 0.2f;

}

clock\_t start = clock();

for (int i = 0; i < N; i++)

{

C[i] = A[i] + B[i];

}

clock\_t end = clock();

double time\_taken = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("C = A + B for %d elements.\n", N);

printf("Time taken: %.3f seconds\n", time\_taken);

printf("Verification (first 5 values):\n");

for (int i = 0; i < 5; i++)

{

printf("A[%d] + B[%d] = %.2f + %.2f = %.2f\n", i, i, A[i], B[i], C[i]);

}

free(A);

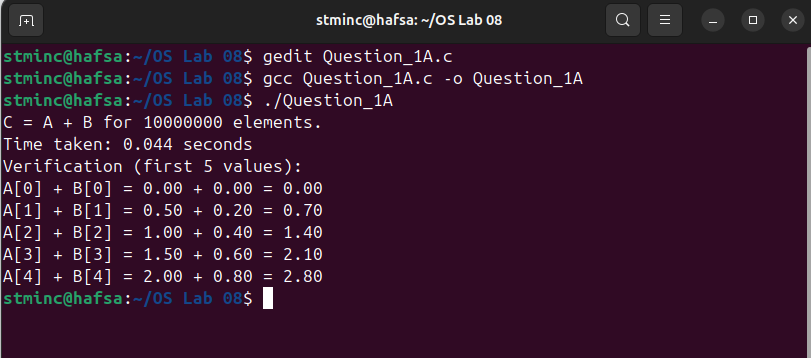
free(B);

free(C);

return 0;

}

Output:



**Question no. 01: Part B**

Code:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#define N 10000000

#define THREADS 10

float \*A, \*B, \*C;

typedef struct

{

int start;

int end;

} ThreadData;

void\* add\_arrays(void\* arg)

{

ThreadData\* data = (ThreadData\*) arg;

for (int i = data->start; i < data->end; i++)

{

C[i] = A[i] + B[i];

}

pthread\_exit(0);

}

int main()

{

pthread\_t threads[THREADS];

ThreadData thread\_data[THREADS];

A = malloc(N \* sizeof(float));

B = malloc(N \* sizeof(float));

C = malloc(N \* sizeof(float));

if (A == NULL || B == NULL || C == NULL)

{

fprintf(stderr, "Memory allocation failed\n");

return 1;

}

for (int i = 0; i < N; i++)

{

A[i] = i \* 0.5f;

B[i] = i \* 0.2f;

}

clock\_t start = clock();

int chunk\_size = N / THREADS;

for (int i = 0; i < THREADS; i++)

{

thread\_data[i].start = i \* chunk\_size;

thread\_data[i].end = (i == THREADS - 1) ? N : (i + 1) \* chunk\_size;

pthread\_create(&threads[i], NULL, add\_arrays, &thread\_data[i]);

}

for (int i = 0; i < THREADS; i++)

{

pthread\_join(threads[i], NULL);

}

clock\_t end = clock();

double time\_taken = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("Computed C = A + B for %d elements using %d threads.\n", N, THREADS);

printf("Time taken: %.3f seconds\n", time\_taken);

printf("Verification (first 5 values):\n");

for (int i = 0; i < 5; i++)

{

printf("A[%d] + B[%d] = %.2f + %.2f = %.2f\n", i, i, A[i], B[i], C[i]);

}

free(A);

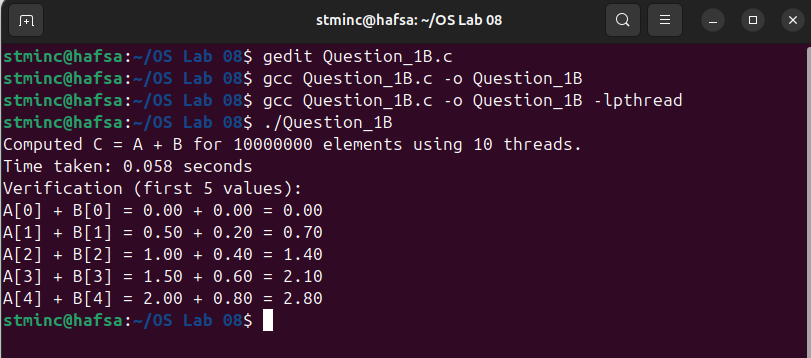
free(B);

free(C);

return 0;

}

Output:



**Question no. 02**

Code:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

typedef struct

{

int \*numbers;

int count;

}

ThreadData;

double average;

int minimum;

int maximum;

void\* calc\_average(void\* param)

{

ThreadData\* data = (ThreadData\*) param;

int sum = 0;

for (int i = 0; i < data->count; i++)

{

sum += data->numbers[i];

}

average = (double) sum / data->count;

pthread\_exit(0);

}

void\* calc\_minimum(void\* param)

{

ThreadData\* data = (ThreadData\*) param;

minimum = data->numbers[0];

for (int i = 1; i < data->count; i++)

{

if (data->numbers[i] < minimum)

{

minimum = data->numbers[i];

}

}

pthread\_exit(0);

}

void\* calc\_maximum(void\* param)

{

ThreadData\* data = (ThreadData\*) param;

maximum = data->numbers[0];

for (int i = 1; i < data->count; i++)

{

if (data->numbers[i] > maximum)

{

maximum = data->numbers[i];

}

}

pthread\_exit(0);

}

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

{

if (argc < 2)

{

fprintf(stderr, "Usage: %s <list of integers>\n", argv[0]);

return 1;

}

int count = argc - 1;

int \*numbers = malloc(count \* sizeof(int));

for (int i = 0; i < count; i++)

{

numbers[i] = atoi(argv[i + 1]);

}

ThreadData data = { numbers, count };

pthread\_t tid\_avg, tid\_min, tid\_max;

pthread\_create(&tid\_avg, NULL, calc\_average, &data);

pthread\_create(&tid\_min, NULL, calc\_minimum, &data);

pthread\_create(&tid\_max, NULL, calc\_maximum, &data);

pthread\_join(tid\_avg, NULL);

pthread\_join(tid\_min, NULL);

pthread\_join(tid\_max, NULL);

printf("The average value is %.0f.\n", average);

printf("The minimum value is %d.\n", minimum);

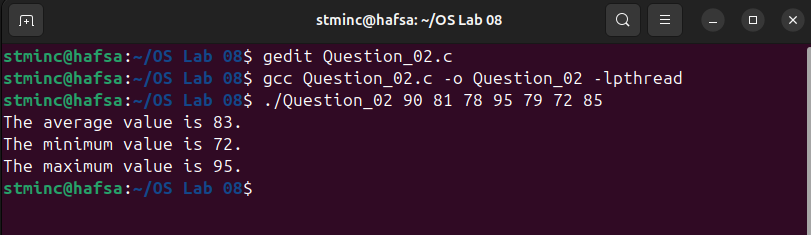
printf("The maximum value is %d.\n", maximum);

free(numbers);

return 0;

}

Output:



Lab Exercises

**Task no. 01**

Code:

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

void\* thread\_func (void\* arg)

{

int num = \*(int\*)arg;

printf("Thread %d started\n", num);

pthread\_exit((void\*)(long)num);

}

int main()

{

pthread\_t threads[3];

int thread\_num[3] = {1, 2, 3};

void\* retval;

for (int i=0; i<3; i++)

{

int status = pthread\_create(&threads[i], NULL, thread\_func, &thread\_num[i]);

if (status != 0)

{

printf("Error creating thread %d\n", thread\_num[i]);

}

else

{

pthread\_join(threads[i], &retval);

printf("Thread %d terminated with return value %ld\n", thread\_num[i], (long)retval);

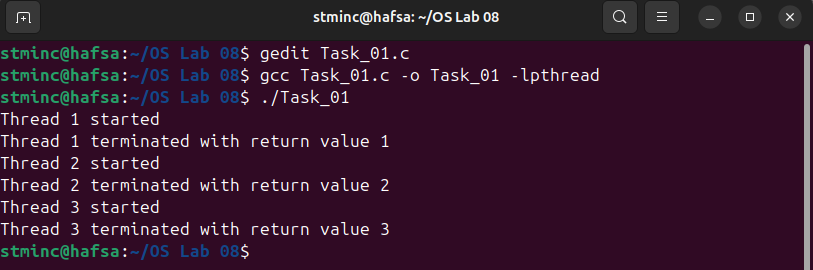
}

}

return 0;

}

Output:



**Task no. 02**

Code:

#include<stdio.h>

#include<pthread.h>

void\* table(void\* arg)

{

int n = \*(int\*)arg;

for (int i=1; i\*n<=1000; i++)

{

printf("%d x %d = %d\n", n, i, n\*i);

}

pthread\_exit(0);

}

int main()

{

pthread\_t t[4];

int nums[4] = {5, 6, 7, 8};

for (int i=0; i<4; i++)

{

pthread\_create(&t[i], NULL, table, &nums[i]);

}

for (int i=0; i<4; i++)

{

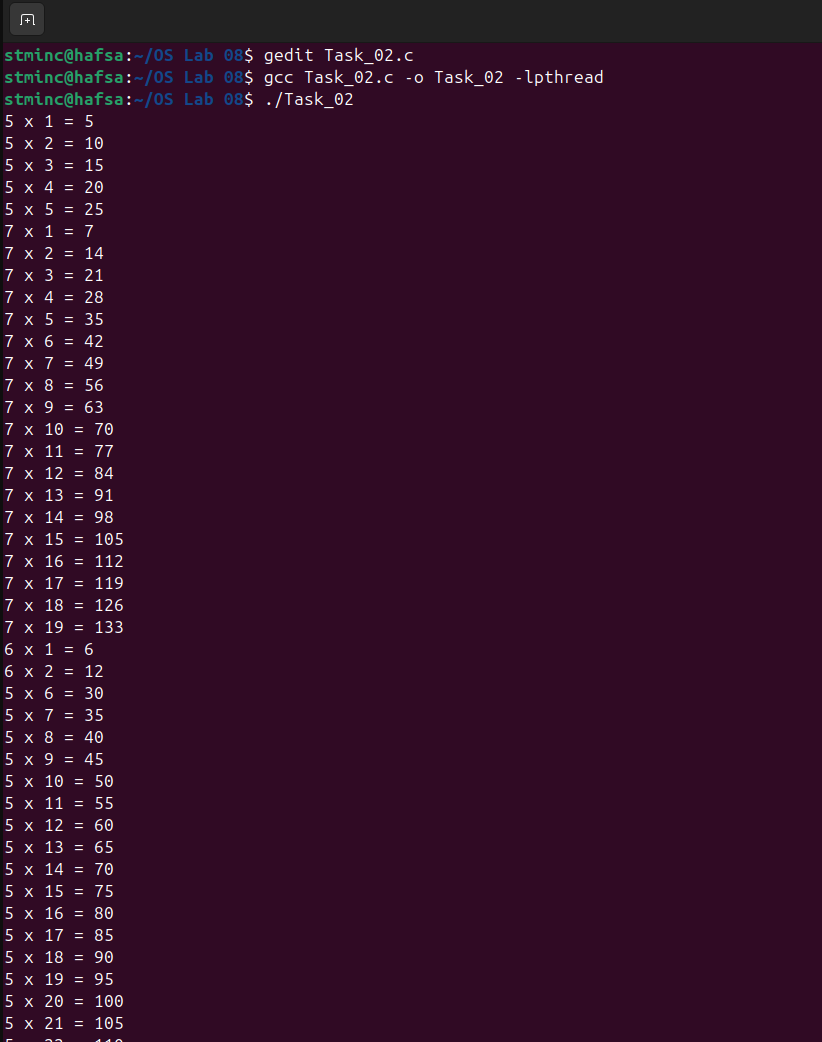
pthread\_join(t[i], NULL);

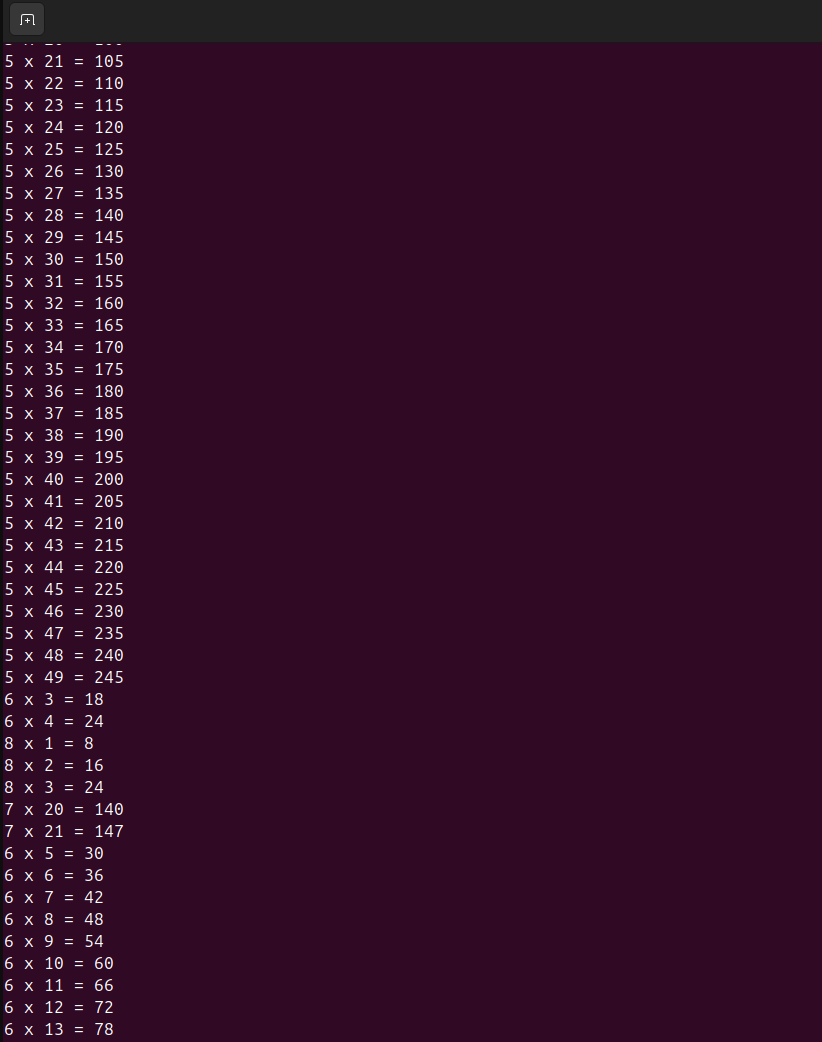
}

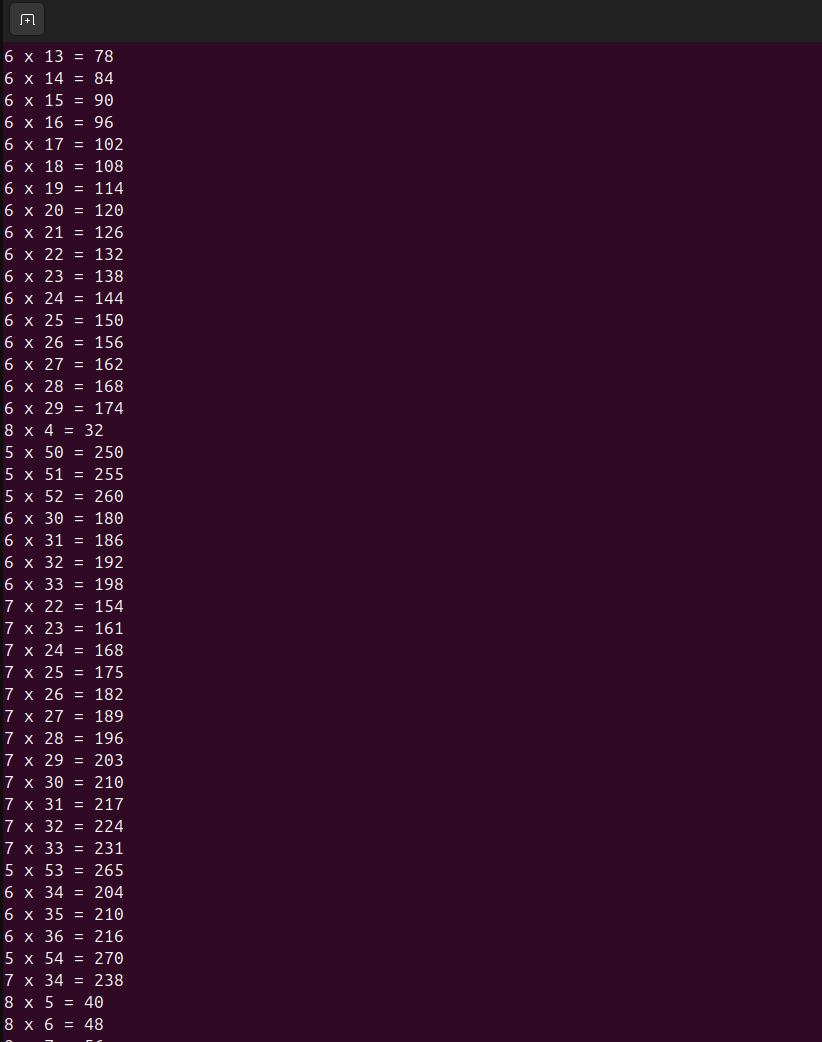
return 0;

}

Output:







The output is very long, but it is fulfilling the task requirement.

**Task no. 03**

Code:

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<unistd.h>

void\* thread\_func(void\* arg)

{

int tid = \*(int\*)arg;

printf("Hello from thread %d\n", tid);

pthread\_exit(0);

}

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

{

if (argc != 2)

{

printf("Usage: %s <number\_of\_threads>\n", argv[0]);

return 1;

}

int N = atoi(argv[1]);

pthread\_t threads[N];

int thread\_ids[N];

for (int i=0; i<N; i++)

{

thread\_ids[i] = i+1;

pthread\_create(&threads[i], NULL, thread\_func, &thread\_ids[i]);

if ((i+1)%4 == 0)

{

sleep(1);

}

}

for (int i=0; i<N; i++)

{

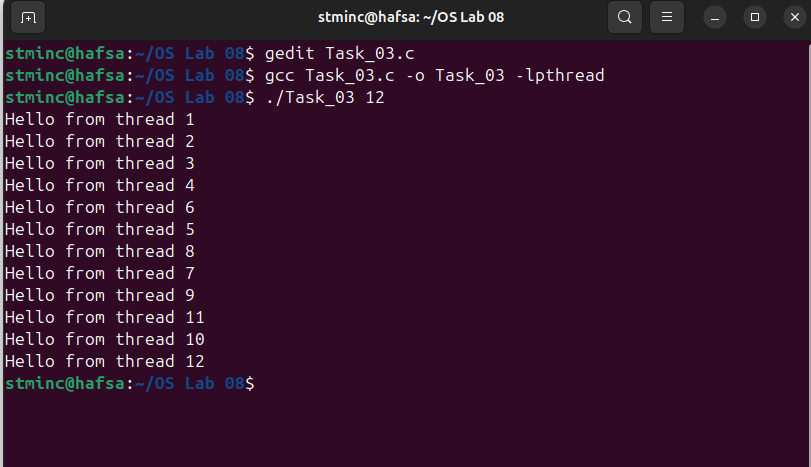
pthread\_join(threads[i], NULL);

}

return 0;

}

Output:



**Task no. 04**

Code:

#include<stdio.h>

#include<pthread.h>

int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int sum\_1 = 0, sum\_2 = 0;

void\* sum\_half\_1 (void\* arg)

{

for (int i=0; i<5; i++)

{

sum\_1 += arr[i];

}

pthread\_exit(0);

}

void\* sum\_half\_2 (void\* arg)

{

for (int i=5; i<10; i++)

{

sum\_2 += arr[i];

}

pthread\_exit(0);

}

int main()

{

pthread\_t t1, t2;

pthread\_create(&t1, NULL, sum\_half\_1, NULL);

pthread\_create(&t2, NULL, sum\_half\_2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

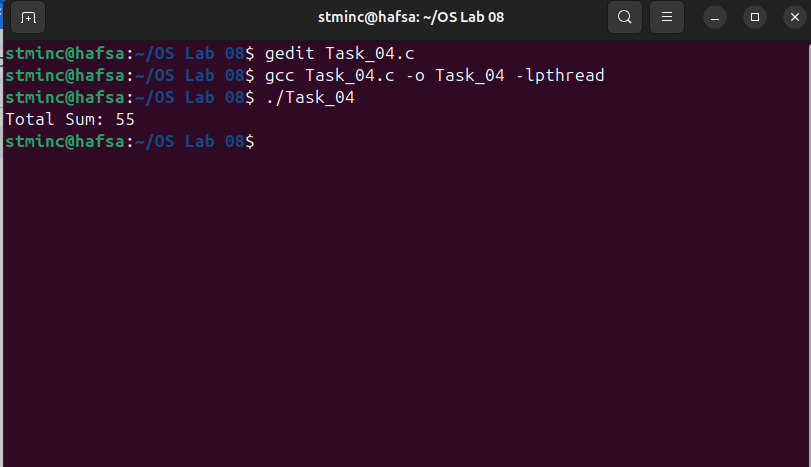
int total = sum\_1 + sum\_2;

printf("Total Sum: %d\n", total);

return 0;

}

Output:



**Task no. 05**

Code:

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PARTICIPANTS 10

pthread\_mutex\_t reg\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t ann\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t sponsor\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t query\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void registration(int id)

{

pthread\_mutex\_lock(&reg\_mutex);

printf("Participant %d is being registered by Volunteer 1.\n", id);

sleep(1);

pthread\_mutex\_unlock(&reg\_mutex);

}

void announcements(int id)

{

pthread\_mutex\_lock(&ann\_mutex);

printf("Participant %d receives announcements from Volunteer 2.\n", id);

sleep(1);

pthread\_mutex\_unlock(&ann\_mutex);

}

void sponsors(int id)

{

pthread\_mutex\_lock(&sponsor\_mutex);

printf("Participant %d talks to sponsors (Volunteer 3).\n", id);

sleep(1);

pthread\_mutex\_unlock(&sponsor\_mutex);

}

void queries(int id)

{

pthread\_mutex\_lock(&query\_mutex);

printf("Participant %d has queries resolved by Volunteer 4.\n", id);

sleep(1);

pthread\_mutex\_unlock(&query\_mutex);

}

void\* participant(void\* arg)

{

int id = \*((int\*)arg);

registration(id);

announcements(id);

sponsors(id);

queries(id);

printf("Participant %d completed all tasks.\n\n", id);

return NULL;

}

int main()

{

pthread\_t threads[NUM\_PARTICIPANTS];

int ids[NUM\_PARTICIPANTS];

for (int i = 0; i < NUM\_PARTICIPANTS; i++)

{

ids[i] = i + 1;

pthread\_create(&threads[i], NULL, participant, &ids[i]);

}

for (int i = 0; i < NUM\_PARTICIPANTS; i++)

{

pthread\_join(threads[i], NULL);

}

printf("\nAll participants have been attended.\n");

pthread\_mutex\_destroy(&reg\_mutex);

pthread\_mutex\_destroy(&ann\_mutex);

pthread\_mutex\_destroy(&sponsor\_mutex);

pthread\_mutex\_destroy(&query\_mutex);

return 0;

}

Output:

