University of Wolverhampton

School of Mathematics and Computer Science

Student Number: 2407710

Name: Dhadkan K.C.

6CS005 High Performance Computing Week 5 Workshop

Q1

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 1
#include <stdio.h>

#define N 3

int main() {
    int A[N][N] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int B[N][N] = {
        {1, 0, 1},
        {0, 1, 0},
        {1, 0, 1}
    };

    int C[N][N];

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) {

            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[r][k] * B[k][c];
            }
            C[r][c] = sum;
        }
    }

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) {
            printf("%d ", C[r][c]);
        }
        printf("\n");
    }

    return 0;
}


// For C[0][0] (first row, first column)

// Take row 0 of A → {1, 2, 3}
// Take column 0 of B → {1, 0, 1}

// Multiply corresponding elements:
//A[0][0] * B[0][0];    // 1 * 1 = 1
//A[0][1] * B[1][0];    // 2 * 0 = 0
//A[0][2] * B[2][0];    // 3 * 1 = 3

// Add them:
//1 + 0 + 3;            // = 4

// Final result:
//C[0][0] = 4;
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Output:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./1
4 2 4
10 5 10
16 8 16
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 1.c
#include <stdio.h>
```

Explaination:

This program multiplies two 3×3 matrices, A and B, by calculating each element of the result matrix C using the dot product of a row from A and a column from B. For every position C[r][c], the code multiplies corresponding elements from row *r* of A and column *c* of B, adds those products, and stores the sum in C. After computing all nine elements through nested loops, the program prints the final multiplied matrix.

Q2

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 2.c
#include <stdio.h>

#define N 3

int main() {
    int A[N][N], B[N][N], C[N][N];

    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            scanf("%d", &A[r][c]);

    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            scanf("%d", &B[r][c]);

    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++) {
            int sum = 0;
            for (int k = 0; k < N; k++)
                sum += A[r][k] * B[k][c];
            C[r][c] = sum;
        }

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++)
            printf("%d ", C[r][c]);
        printf("\n");
    }

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Output:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./2
./2
294912 8994816 0
0 0 0
7864320 239861760 0
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

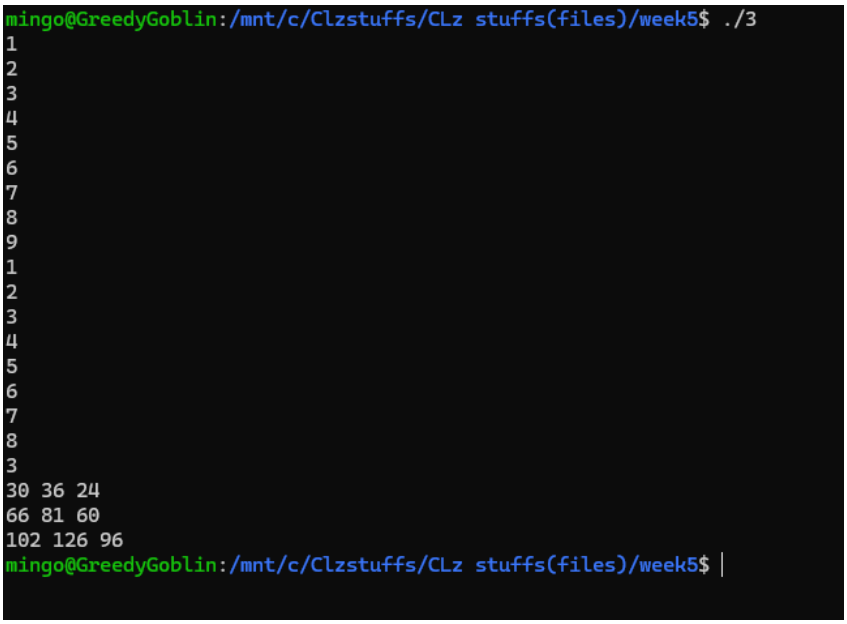
Explaination:

This program multiplies two 3×3 matrices entered by the user. It first reads all elements of matrix A, then all elements of matrix B. Using three nested loops, it computes each element of the result matrix C by taking the dot product of a row from A and a column from B—multiplying corresponding elements and adding them together. After completing all calculations, it prints the final 3×3 matrix C as the output.

Q3

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 3.c
#include <stdio.h>
#include <pthread.h>

#define N 3

int A[N][N], B[N][N], C[N][N];

void* work(void* arg) {
    int i = *(int*)arg;
    int r = i / N;
    int c = i % N;

    int sum = 0;
    for (int k = 0; k < N; k++)
        sum += A[r][k] * B[k][c];

    C[r][c] = sum;
    return NULL;
}

int main() {
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            scanf("%d", &A[r][c]);

    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            scanf("%d", &B[r][c]);

    pthread_t t[N*N];
    int idx[N*N];

    for (int i = 0; i < N*N; i++) {
        idx[i] = i;
        pthread_create(&t[i], NULL, work, &idx[i]);
    }

    for (int i = 0; i < N*N; i++)
        pthread_join(t[i], NULL);

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++)
            printf("%d ", C[r][c]);
        printf("\n");
    }

    return 0;
}
```

Output



Explanation:

This program performs matrix multiplication using **multithreading**, where each element of the result matrix C is computed by a separate thread. After reading the input matrices A and B, it creates 9 threads (one for each cell in a 3×3 matrix). Each thread receives an index, converts it into a row and column, computes the dot product of that row of A and column of B, and stores the result in C. After all threads finish (using pthread_join), the program prints the fully computed matrix C.

Q4

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 4.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>   // for usleep()

#define N 3

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

int B[N][N] = {
    {1,0,1},
    {0,1,0},
    {1,0,1}
};

int C[N][N];
int temp;

void* work(void* arg) {
    int i = *(int*)arg;
    int r = i / N;
    int c = i % N;

    temp = 0;

    for (int k = 0; k < N; k++) {
        temp += A[r][k] * B[k][c];
        //usleep(1000);
    }

    C[r][c] = temp;
    return NULL;
}

int main() {
    pthread_t t[N*N];
    int idx[N*N];

    for (int i = 0; i < N*N; i++) {
        idx[i] = i;
        pthread_create(&t[i], NULL, work, &idx[i]);
    }

    for (int i = 0; i < N*N; i++)
        pthread_join(t[i], NULL);

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++)
            printf("%d ", C[r][c]);
        printf("\n");
    }

    return 0;
}

mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Output:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./4
4 2 4
10 5 10
16 8 16
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 4.c
```

Explanation:

This program multiplies two 3×3 matrices using **multithreading**, but with a key issue: all threads share the same global variable temp. Each thread calculates one element of matrix C by taking a row from A and a column from B, looping through and adding the products. However, because temp is global and not thread-safe, threads may overwrite each other's values if they run simultaneously. After launching one thread per matrix cell (9 threads total), the program waits

for all threads to finish and then prints the final matrix C. This code demonstrates how race conditions can occur when multiple threads share writable global variables.

Q4.(sleep 1000)

```
mingo@GreedyGoblin:/mnt/c/Cizstuffs/CLz stuffs(files)/week5$ cat 4.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>   // for usleep()

#define N 3

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

int B[N][N] = {
    {1,0,1},
    {0,1,0},
    {1,0,1}
};

int C[N][N];
int temp;

void* work(void* arg) {
    int i = *(int*)arg;
    int r = i / N;
    int c = i % N;

    temp = 0;

    for (int k = 0; k < N; k++) {
        temp += A[r][k] * B[k][c];
        usleep(1000);
    }

    C[r][c] = temp;
    return NULL;
}

int main() {
    pthread_t t[N*N];
    int idx[N*N];

    for (int i = 0; i < N*N; i++) {
        idx[i] = i;
        pthread_create(&t[i], NULL, work, &idx[i]);
    }

    for (int i = 0; i < N*N; i++)
        pthread_join(t[i], NULL);

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++)
            printf("%d ", C[r][c]);
        printf("\n");
    }

    return 0;
}
```

Output:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ gcc -o 4 4.c
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./4
12 55 55
55 55 55
55 55 55
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./4
51 51 51
51 51 51
51 51 51
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./4
58 58 58
58 58 58
58 58 58
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./4
58 58 58
```

Explanation:

This program multiplies two 3×3 matrices using multithreading, but the output changes on every run because all threads share the same global variable temp. Each of the 9 threads is supposed to compute one element of matrix **C**, but since they run at the same time and all read and write to temp simultaneously, they overwrite each other's calculations. As a result, the final matrix contains inconsistent and repeated values depending on which thread last modified temp. This inconsistent output clearly demonstrates a race condition caused by unsynchronized access to shared global data.

Q5.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 5.c
#include <stdio.h>
#include <pthread.h>

int counter = 0;

static void counter_func() {
    for (int i = 0; i < 100000; i++) {

        counter++;
    }
}

void* thread_one(void* arg) {
    counter_func();
    return NULL;
}

void* thread_two(void* arg) {
    counter_func();
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, thread_one, NULL);
    pthread_create(&t2, NULL, thread_two, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("%d\n", counter);
    return 0;
}
```

Output:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ gcc -o 5 5.c
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./5
129555
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explaination:

This program creates two threads, and each thread runs counter_func(), which increments the shared global variable counter 100,000 times. Ideally, the final output should be 200,000, but because both threads modify the same variable simultaneously without any locking or

synchronization, a race condition occurs. This means the increment operations overlap and interfere with each other, causing some updates to be lost. As a result, the printed value is usually less than 200,000, demonstrating how unsynchronized multithreaded access to shared data leads to incorrect results

Q6

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 6.c
#include <stdio.h>
#include <pthread.h>

int counter = 0;
pthread_mutex_t lock;

static void counter_func() {
    for (int i = 0; i < 100000; i++) {

        pthread_mutex_lock(&lock);   // critical section START
        counter++;
        pthread_mutex_unlock(&lock); // critical section END
    }
}

void* thread_one(void* arg) {
    counter_func();
    return NULL;
}

void* thread_two(void* arg) {
    counter_func();
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);

    pthread_create(&t1, NULL, thread_one, NULL);
    pthread_create(&t2, NULL, thread_two, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);

    printf("%d\n", counter);
    return 0;
}
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ gcc -o 6 6.c
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./6
200000
```

Explanation:

This program fixes the race condition from the previous version by using a mutex lock to protect the shared variable counter. Two threads are created, and each thread increments the counter 100,000 times inside counter_func(). Before each increment, the thread locks the mutex, performs the update, and then unlocks it, ensuring that only one thread can modify counter at a time. This prevents interference and lost updates. As a result, the final printed value will reliably be 200,000, demonstrating safe multithreaded access to shared data using mutual exclusion.

Q7

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 7.c
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>

int value = 0;
pthread_mutex_t lock;

void* reader(void* arg) {
    for (int i = 0; i < 50000; i++) {
        pthread_mutex_lock(&lock);
        int v = value;
        pthread_mutex_unlock(&lock);
        (void)v;
    }
    return NULL;
}

void* writer(void* arg) {
    for (int i = 0; i < 50000; i++) {
        pthread_mutex_lock(&lock);
        value++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t r[8], w[2];
    pthread_mutex_init(&lock, NULL);

    struct timeval start, end;
    gettimeofday(&start, NULL);

    for (int i = 0; i < 8; i++)
        pthread_create(&r[i], NULL, reader, NULL);

    for (int i = 0; i < 2; i++)
        pthread_create(&w[i], NULL, writer, NULL);

    for (int i = 0; i < 8; i++)
        pthread_join(r[i], NULL);

    for (int i = 0; i < 2; i++)
        pthread_join(w[i], NULL);

    gettimeofday(&end, NULL);

    long micro = (end.tv_sec - start.tv_sec) * 1000000L +
                 (end.tv_usec - start.tv_usec);

    double seconds = micro / 1000000.0;

    printf("Final value: %d\n", value);
    printf("Execution time (seconds): %.6f\n", seconds);

    pthread_mutex_destroy(&lock);
    return 0;
}
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./7
Final value: 100000
Execution time (seconds): 0.044508
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explanation:

This program measures how long it takes multiple threads to repeatedly read and write a shared variable using a mutex for synchronization. It creates 8 reader threads that repeatedly lock, read, and unlock the shared variable, and 2 writer threads that lock, increment, and unlock it. Because all accesses—both reading and writing—use the same mutex, every thread must wait for others, causing contention. The program uses gettimeofday() to record the start and end times, then calculates the total execution duration. Finally, it prints the final value (which equals the total number of writer increments) and the time taken, demonstrating how mutex locking affects performance when many threads compete for a shared resource.

Qn 8

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 8.c
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

int value = 0;
pthread_rwlock_t rwlock;

void* reader(void* arg) {
    for (int i = 0; i < 50000; i++) {
        pthread_rwlock_rdlock(&rwlock);
        int v = value;
        pthread_rwlock_unlock(&rwlock);
        (void)v;
    }
    return NULL;
}

void* writer(void* arg) {
    for (int i = 0; i < 50000; i++) {
        pthread_rwlock_wrlock(&rwlock);
        value++;
        pthread_rwlock_unlock(&rwlock);
    }
    return NULL;
}

int main() {
    pthread_t r[8], w[2];
    pthread_rwlock_init(&rwlock, NULL);

    struct timeval start, end;
    gettimeofday(&start, NULL);

    for (int i = 0; i < 8; i++)
        pthread_create(&r[i], NULL, reader, NULL);

    for (int i = 0; i < 2; i++)
        pthread_create(&w[i], NULL, writer, NULL);

    for (int i = 0; i < 8; i++)
        pthread_join(r[i], NULL);

    for (int i = 0; i < 2; i++)
        pthread_join(w[i], NULL);

    gettimeofday(&end, NULL);

    long micro = (end.tv_sec - start.tv_sec) * 1000000L +
                 (end.tv_usec - start.tv_usec);

    double seconds = micro / 1000000.0;

    printf("Final value: %d\n", value);
    printf("Execution time (seconds): %.6f\n", seconds);

    pthread_rwlock_destroy(&rwlock);
    return 0;
}
```

output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./8
Final value: 100000
Execution time (seconds): 0.035635
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explanation:

This program is used to test the efficiency of using a read write lock (pthread Rwlock) to coordinate access to a shared variable among a large number of reader threads and a small number of writer threads. Eight reader threads continuously take the read lock and read the value many times and release the read lock,and more than eight reader threads can concurrently execute. The write lock is obtained and the shared variable is updated and released by two writer threads--they have exclusive access whenever they are writing. The read locks can be shared and hence the readers do not block one another; this makes read locks better than a standard mutex. The program measures the whole operation with the help of gettimeofday() and writes the final value (equivalent to total increment by writers) and the time taken to execute it and proves the efficiency advantages of read write locks in conditions of significant reads.

Q9

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 9.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define N 4

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

int count = 0;

void broken_barrier()
{
    pthread_mutex_lock(&m);
    count++;
    if (count == N)
        pthread_cond_broadcast(&c);
    else
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

void* work(void* a)
{
    printf("T%d reached barrier 1\n", (int)(long)a);
    broken_barrier();
    printf("T%d passed barrier 1\n", (int)(long)a);

    sleep(1);

    printf("T%d reached barrier 2\n", (int)(long)a);
    broken_barrier();
    printf("T%d passed barrier 2 (broken)\n", (int)(long)a);

    return NULL;
}

int main()
{
    pthread_t t[N];
    for (long i = 0; i < N; i++)
        pthread_create(&t[i], NULL, work, (void*)i);

    for (int i = 0; i < N; i++)
        pthread_join(t[i], NULL);

    return 0;
}
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./9
T1 reached barrier 1
T0 reached barrier 1
T2 reached barrier 1
T3 reached barrier 1
T3 passed barrier 1
T2 passed barrier 1
T0 passed barrier 1
T1 passed barrier 1
T0 reached barrier 2
T2 reached barrier 2
T3 reached barrier 2
T1 reached barrier 2
```

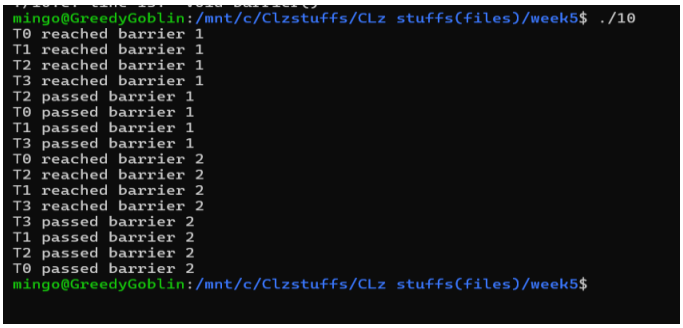
Explanation:

This program demonstrates a broken barrier implementation using a mutex and condition variable, showing why a simple barrier cannot be reused safely. Four threads run the work() function, reach “barrier 1,” and call broken_barrier(). The first three threads wait, and the fourth thread triggers pthread_cond_broadcast(), allowing all threads to proceed past barrier 1. However, the same barrier is then reused for “barrier 2” without resetting the shared variable count. Since count is already at 4 from the first barrier, every thread immediately sees the barrier as “complete” and passes through without waiting for others. This results in an incorrect, non-synchronized second barrier, illustrating why proper barriers must reset state or use dedicated barrier primitives like pthread_barrier_t.

Q10

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 10.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define N 4

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

int count = 0;
int generation = 0;

void barrier()
{
    pthread_mutex_lock(&m);

    int gen = generation;
    count++;

    if (count == N) {
        generation++;
        count = 0;
        pthread_cond_broadcast(&c);
    } else {
        while (gen == generation)
            pthread_cond_wait(&c, &m);
    }

    pthread_mutex_unlock(&m);
}

void* work(void* a)
{
    printf("T%d reached barrier 1\n", (int)(long)a);
    barrier();
    printf("T%d passed barrier 1\n", (int)(long)a);

    sleep(1);

    printf("T%d reached barrier 2\n", (int)(long)a);
    barrier();
    printf("T%d passed barrier 2\n", (int)(long)a);

    return NULL;
}

int main()
{
    pthread_t t[N];

    for (long i = 0; i < N; i++)
        pthread_create(&t[i], NULL, work, (void*)i);

    for (int i = 0; i < N; i++)
        pthread_join(t[i], NULL);

    return 0;
}
```

Output:



Explanation:

This program implements a reusable barrier using a mutex and a condition variable. Four threads run the work() function and call barrier() at two points. Each thread increments a shared count when it reaches the barrier. When all threads reach it (count == N), the generation counter is incremented, count is reset, and pthread_cond_broadcast() wakes all waiting threads. Threads that arrive earlier wait in a loop until the generation changes, ensuring proper synchronization. Unlike the broken barrier, this version allows the barrier to be safely reused, so all threads correctly wait at both barrier 1 and barrier 2 before proceeding.

Qn11

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 11.c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "I am the best student of HCK";
    char* token = strtok(str, " ");

    while (token != NULL) {
        printf("%s\n", token);
        token = strtok(NULL, " ");
    }

    return 0;
}
```

Output:

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./11
I
am
the
best
student
of
HCK
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explanation:

This program demonstrates tokenizing a string using strtok() in C. It takes the string "I am the best student of HCK" and splits it into words based on the space character " ". The first call to strtok(str, " ") returns the first word "I", and subsequent calls with strtok(NULL, " ") return the remaining words one by one. The while loop prints each word on a new line, resulting in each word of the original string being displayed separately.

Q12

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 12.c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void* thread_func(void* arg) {
    char* text = (char*)arg;
    char* token = strtok(text, " ");

    while (token != NULL) {
        printf("%s\n", token);
        for (volatile int i = 0; i < 1000000; i++);
        token = strtok(NULL, " ");
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    char str1[] = "I am the best student of HCK";
    char str2[] = "Thread unsafe strtok example demo";

    pthread_create(&t1, NULL, thread_func, str1);
    pthread_create(&t2, NULL, thread_func, str2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./12
I
Thread
unsafe
strtok
example
demo
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explanation:

This program demonstrates a thread-unsafe usage of strtok() in multithreading. Two threads are created, each calling thread_func() with a different string. Inside the function, strtok() is used to split the string into words. However, strtok() uses a global static buffer internally, which is shared across threads, so if multiple threads call it at the same time, they can interfere with each other's tokenization. This can lead to mixed or corrupted output, where words from both strings may appear incorrectly. The program shows why strtok_r() (the reentrant version) should be used in multithreaded programs.

Q13

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 13.c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void* thread_func(void* arg) {
    char* text = (char*)arg;
    char* saveptr;
    char* token = strtok_r(text, " ", &saveptr);

    while (token != NULL) {
        printf("%s\n", token);
        token = strtok_r(NULL, " ", &saveptr);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    char str1[] = "I am the best student of HCK";
    char str2[] = "Thread safe strtok_r example demo";

    pthread_create(&t1, NULL, thread_func, str1);
    pthread_create(&t2, NULL, thread_func, str2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./13
I
am
the
best
student
of
HCK
Thread
safe
strtok_r
example
demo
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explanation:

This program demonstrates thread-safe string tokenization using strtok_r(). Two threads are created, each processing a separate string. Unlike strtok(), strtok_r() uses a user-provided

saveptr to maintain context, so each thread keeps its own state and does not interfere with the other. Inside thread_func(), strtok_r() splits the string into words and prints each word on a new line. This ensures correct and independent tokenization in a multithreaded environment, avoiding the race conditions and output corruption that occur with the thread-unsafe strtok().

Qn 14

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ cat 14.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int flag = 0;

void* thread_one(void* arg) {
    printf("Thread 1: waiting for flag to become 1...\n");
    while (flag == 0);
    printf("Thread 1: detected flag = 1\n");
    return NULL;
}

void* thread_two(void* arg) {
    printf("Thread 2: sleeping for 2 seconds...\n");
    sleep(2);
    flag = 1;
    printf("Thread 2: flag set to 1\n");
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, thread_one, NULL);
    pthread_create(&t2, NULL, thread_two, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$ ./14
Thread 1: waiting for flag to become 1...
Thread 2: sleeping for 2 seconds...
Thread 1: detected flag = 1
Thread 2: flag set to 1
mingo@GreedyGoblin:/mnt/c/Clzstuffs/CLz stuffs(files)/week5$
```

Explanation:

This program demonstrates a **busy-wait loop** and highlights a **thread-safety issue** with shared variables. Thread 1 continuously checks the global variable flag in a while (flag == 0) loop, waiting for it to become 1. Thread 2 sleeps for 2 seconds and then sets flag = 1. Ideally, Thread 1 detects the change and prints the message. However, since flag is not declared volatile or protected by a mutex, the compiler or CPU may optimize the busy-wait loop, potentially causing **Thread 1 to never see the update**. This illustrates the need for proper synchronization (e.g., volatile, mutexes, or condition variables) when sharing data between threads.