

Operating Systems – COC 3071L

SE 5th A – Fall 2025

Lab 5: Introduction to Threads

1. Introduction to Threads

1.1 What is a Thread?

A **thread** is the smallest unit of execution within a process.

- A **process** can have multiple threads running concurrently
- All threads within a process share:
 - Memory space (code, data, heap)
 - File descriptors
 - Process ID
- Each thread has its own:
 - Thread ID (TID)
 - Stack
 - Program counter
 - Register set

Real-world analogy:

- **Process** = A restaurant kitchen
- **Threads** = Multiple cooks working together in the same kitchen, sharing ingredients and equipment

1.2 Threads vs Processes – Quick Comparison

Feature	Process	Thread
Memory	Separate memory space	Shared memory space
Creation	Expensive (fork)	Lightweight (pthread_create)
Communication	IPC needed (pipes, etc.)	Direct (shared variables)
Context Switch	Slower	Faster
Independence	Fully independent	Dependent on parent process

When to use threads?

- When tasks need to share data frequently
 - For parallel execution within the same application
 - When you need lightweight concurrency
-

2. POSIX Threads (pthreads) Library

In Linux, we use the **POSIX threads (pthreads)** library for thread programming.

2.1 Compilation Requirements

When compiling programs with threads, you **must** link the pthread library:

```
gcc program.c -o program -lpthread
```

The `-lpthread` flag links the pthread library.

3. C Programs with Threads

Program 1: Creating a Simple Thread

Objective: Create a thread and print messages from both main thread and new thread.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// Thread function - this will run in the new thread
void* thread_function(void* arg) {
    printf("Hello from the new thread!\n");
    printf("Thread ID: %lu\n", pthread_self());
    return NULL;
}

int main() {
    pthread_t thread_id;

    printf("Main thread starting...\n");
    printf("Main Thread ID: %lu\n", pthread_self());

    // Create a new thread
    pthread_create(&thread_id, NULL, thread_function, NULL);
```

```

// Wait for the thread to finish
pthread_join(thread_id, NULL);

printf("Main thread exiting...\n");
return 0;
}

```

Compile and run:

```

gcc thread1.c -o thread1 -lpthread
./thread1

```

```

// Task1.c
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 // Thread function - this will run in the new thread
5 void* thread_function(void* arg) {
6     printf("Hello from the new thread!\n");
7     printf("Thread ID: %lu\n", pthread_self());
8     return NULL;
9 }
10 int main() {
11     pthread_t thread_id;
12     printf("Main thread starting...\n");
13     printf("Main Thread ID: %lu\n", pthread_self());
14     // Create a new thread
15     pthread_create(&thread_id, NULL, thread_function, NULL);
16     // Wait for the thread to finish
17     pthread_join(thread_id, NULL);
18     printf("Main thread exiting...\n");
19     return 0;
20 }

Terminal Output:
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab5$ gcc Task1.c -o task1 -lpthread
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab5$ ./task1
Main thread starting...
Main Thread ID: 123908765701952
Hello from the new thread!
Thread ID: 123908762105536
Main thread exiting...
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab5$

```

Explanation:

`pthread_t thread_id`

This creates a **variable** to hold the thread's ID (like a file descriptor or process ID). It's just a handle the OS uses to manage the thread.

`pthread_create(&thread_id, NULL, thread_function, NULL)`

Let's decode the four parameters:

Parameter	Type	Meaning
<code>&thread</code>	<code>pthread_t*</code>	Where the new thread ID will be stored
<code>NULL</code>	<code>pthread_attr_t*</code>	Thread attributes (priority, stack size, etc.) — NULL means default

myThread	void* (*start_routine) (void*)	Function to run in the new thread
NULL	void*	Pointer passed to the function for data

- `pthread_join()` → Waits for thread to finish (like `wait()` for processes)
- `pthread_self()` → Returns the thread ID of calling thread

Program 2: Passing Arguments to Threads

Objective: Pass data to a thread function.

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

void* print_number(void* arg) {
    // We know that we've passed an integer pointer
    int num = *(int*)arg; // Cast void* back to int*
    printf("Thread received number: %d\n", num);
    printf("Square: %d\n", num * num);
    return NULL;
}

int main() {
    pthread_t thread_id;
    int number = 42;

    printf("Creating thread with argument: %d\n", number);

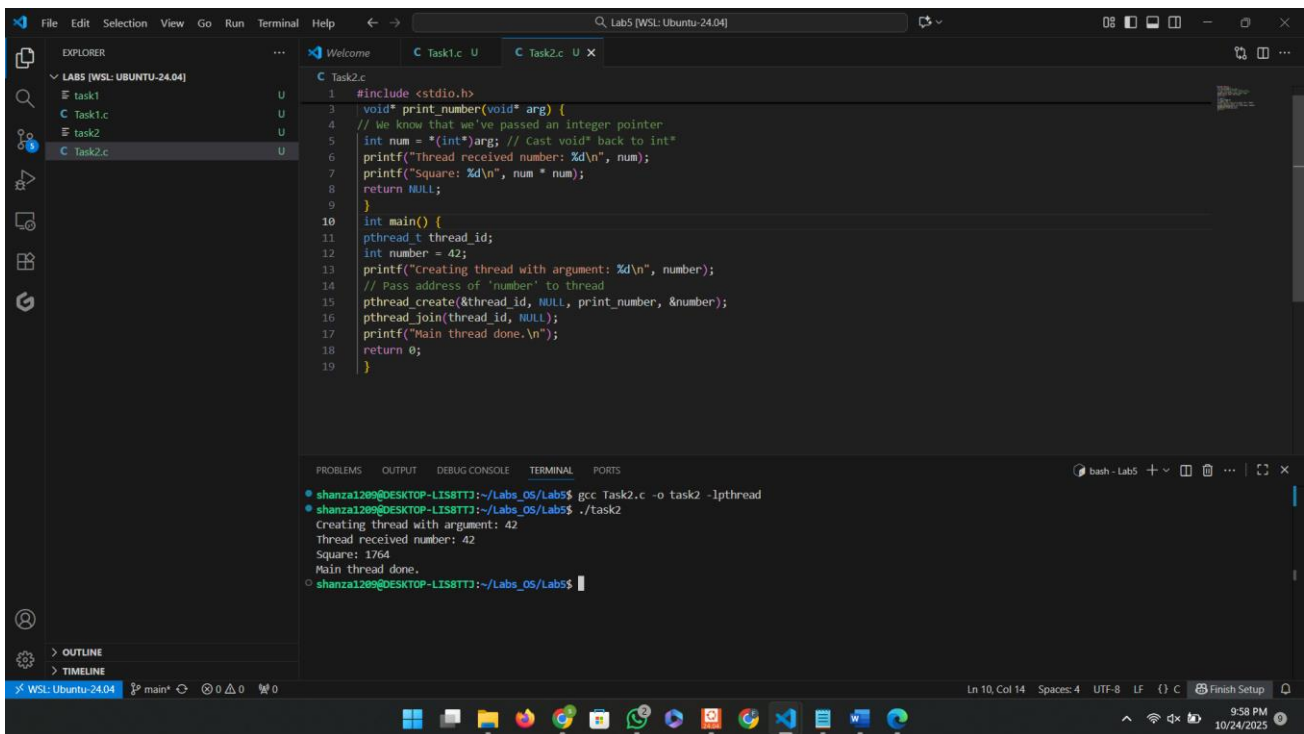
    // Pass address of 'number' to thread
    pthread_create(&thread_id, NULL, print_number, &number);

    pthread_join(thread_id, NULL);

    printf("Main thread done.\n");
    return 0;
}
```

Compile and run:

```
gcc thread2.c -o thread2 -lpthread
./thread2
```



```
1 #include <stdio.h>
2
3 void* print_number(void* arg) {
4     // We know that we've passed an integer pointer
5     int num = *(int*)arg; // Cast void* back to int*
6     printf("Thread received number: %d\n", num);
7     printf("Square: %d\n", num * num);
8     return NULL;
9 }
10
11 int main() {
12     pthread_t thread_id;
13     int number = 42;
14     printf("creating thread with argument: %d\n", number);
15     // Pass address of 'number' to thread
16     pthread_create(&thread_id, NULL, print_number, &number);
17     pthread_join(thread_id, NULL);
18     printf("Main thread done.\n");
19     return 0;
20 }
```

```
shanza1209@DESKTOP-LIS8T73:~/Labs_05/Lab5$ gcc Task2.c -o task2 -lpthread
shanza1209@DESKTOP-LIS8T73:~/Labs_05/Lab5$ ./task2
Creating thread with argument: 42
Thread received number: 42
Square: 1764
Main thread done.
```

Important Notes:

- The 4th argument of `pthread_create()` is passed to the thread function
- It's a `void*` pointer, so you can pass any data type
- Remember to cast it properly inside the thread function

Here's what happens step by step:

```
int value = *(int*)arg;
```

1. `(int*)arg` — cast `void*` back to `int*`.
2. `*(int*)arg` — dereference the pointer to get the integer value it points to.

Why use `void*`

The thread function must have the **standard signature**:

```
void* function_name(void* arg)
```

That's because threads can accept *any* data type — integers, structs, arrays, etc.

`void*` acts like a universal pointer type.

If you need to pass multiple variables, you wrap them in a `struct` and pass a pointer to it.

Program 3: Passing Multiple Data

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

typedef struct {
    int id;
    char* message;
} ThreadData;

void* printData(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    printf("Thread %d says: %s\n", data->id, data->message);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    ThreadData data1 = {1, "Hello"};
    ThreadData data2 = {2, "World"};

    pthread_create(&t1, NULL, printData, &data1);
    pthread_create(&t2, NULL, printData, &data2);

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

    printf("All threads done.\n");
    return 0;
}
```

Compile and run:

```
gcc thread3.c -o thread3 -lpthread
./thread3
```

```
5 char* message;
6 } ThreadData;
7 void* printData(void* arg) {
8     ThreadData* data = (ThreadData*)arg;
9     printf("Thread %d says: %s\n", data->id, data->message);
10    return NULL;
11 }
12 int main() {
13     pthread_t t1, t2;
14     ThreadData data1 = {1, "Hello"};
15     ThreadData data2 = {2, "World"};
16     pthread_create(&t1, NULL, printData, &data1);
17     pthread_create(&t2, NULL, printData, &data2);
18     pthread_join(t1, NULL);
19     pthread_join(t2, NULL);
20     printf("All threads done.\n");
21     return 0;
22 }
```

```
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab5$ gcc Task3.c -o task3 -lpthread
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab5$ ./task3
Thread 1 says: Hello
Thread 2 says: World
All threads done.
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab5$
```

Program 4: Thread Return Values

Objective: Get return values from threads.

```

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* calculate_sum(void* arg) {
    int n = *(int*)arg;
    int* result = malloc(sizeof(int)); // Allocate memory for result

    *result = 0;
    for (int i = 1; i <= n; i++) {
        *result += i;
    }

    printf("Thread calculated sum of 1 to %d = %d\n", n, *result);
    return (void*)result; // Return the result
}

int main() {
    pthread_t thread_id;
    int n = 100;
    void* sum;

    pthread_create(&thread_id, NULL, calculate_sum, &n);

    // Get the return value from thread
    pthread_join(thread_id, &sum);

    printf("Main received result: %d\n", *(int*)sum);

    free(sum); // Don't forget to free allocated memory
    return 0;
}

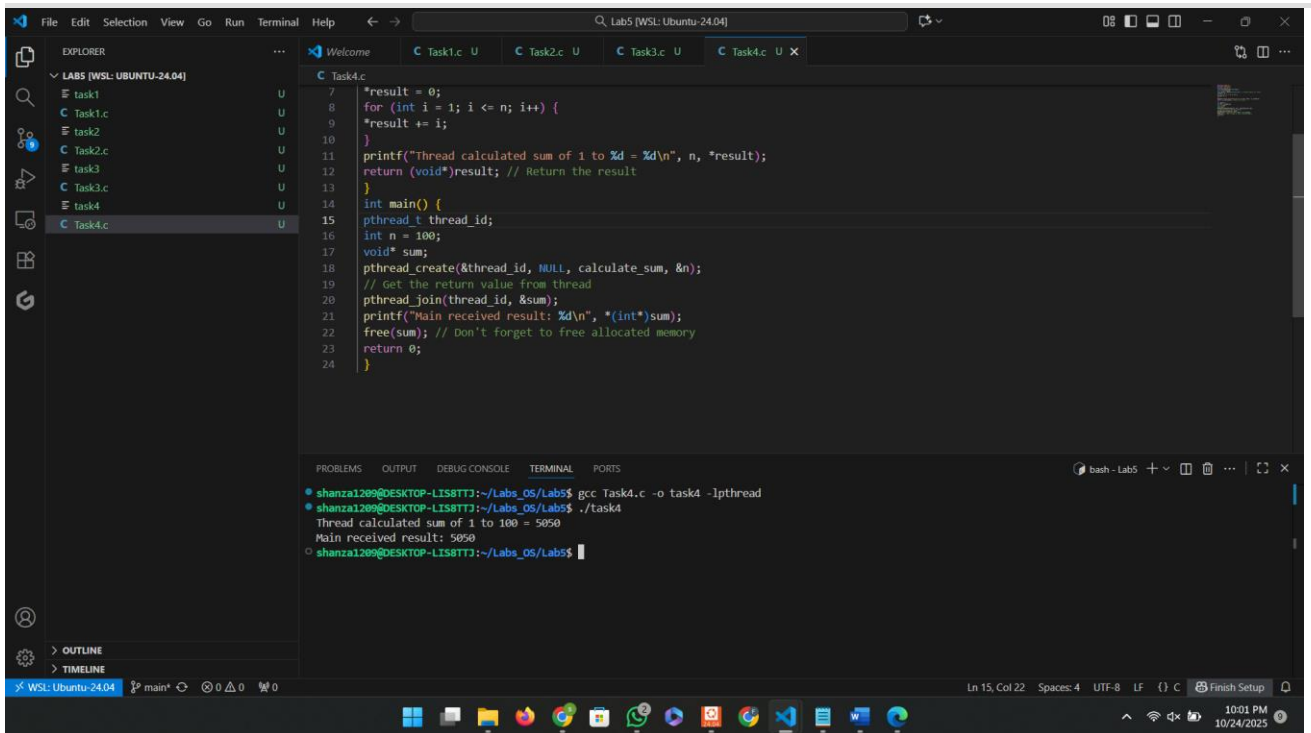
```

Compile and run:

```

gcc thread5.c -o thread5 -lpthread
./thread5

```

```
7  *result = 0;
8  for (int i = 1; i <= n; i++) {
9      *result += i;
10 }
11 printf("Thread calculated sum of 1 to %d = %d\n", n, *result);
12 return (void*)result; // Return the result
13 }
14
15 int main() {
16     pthread_t thread_id;
17     int n = 100;
18     void* sum;
19     pthread_create(&thread_id, NULL, calculate_sum, &n);
20     // Get the return value from thread
21     pthread_join(thread_id, &sum);
22     printf("Main received result: %d\n", *(int*)sum);
23     free(sum); // Don't forget to free allocated memory
24     return 0;
25 }
```

```
shanza1209@DESKTOP-LI58T73:~/Lab5_05/Lab5$ gcc Task4.c -o task4 -lpthread
shanza1209@DESKTOP-LI58T73:~/Lab5_05/Lab5$ ./task4
Thread calculated sum of 1 to 100 = 5050
Main received result: 5050
shanza1209@DESKTOP-LI58T73:~/Lab5_05/Lab5$
```

Key Points:

- Thread functions return `void*`
- Use `pthread_join()` to retrieve the return value
- Remember to free any dynamically allocated memory

4. Basic Multithreading

4.1 What is Multithreading?

- **Multithreading** means running multiple threads *concurrently* to perform different tasks within the same process.
- It allows:
 - Faster program execution on multi-core CPUs
 - Better resource utilization
 - Improved responsiveness (e.g., in GUIs or servers)

Example use cases:

- A web server handling multiple client requests simultaneously
- A program performing computation and I/O in parallel

Program 1: Creating and Running Multiple Threads

Objective:

Create multiple threads that execute independently and print messages concurrently.

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void* worker(void* arg) {
    int thread_num = *(int*)arg;
    printf("Thread %d: Starting task...\n", thread_num);
    sleep(1); // Simulate some work
    printf("Thread %d: Task completed!\n", thread_num);
    return NULL;
}

int main() {
    pthread_t threads[3];
    int thread_ids[3];

    for (int i = 0; i < 3; i++) {
        thread_ids[i] = i + 1;
        pthread_create(&threads[i], NULL, worker, &thread_ids[i]);
    }

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

        pthread_join(threads[i], NULL);
    }

    printf("Main thread: All threads have finished.\n");
    return 0;
}

```

Compile and run:

```

gcc multithread_basic.c -o multithread_basic -lpthread
./multithread_basic

```

The screenshot shows a Visual Studio Code editor with a C program named `Task5.c` open. The program creates 5 threads, each of which prints a message, sleeps for 1 second, and then prints another message. The main thread prints a message after all other threads have finished. The output in the terminal shows the execution of the program, demonstrating that the threads execute concurrently and the output order is non-deterministic.

```
6 printf("Thread %d: Starting task...\n", thread_num);
7 sleep(1); // Simulate some work
8 printf("Thread %d: Task completed!\n", thread_num);
9 return NULL;
10 }
11 int main() {
12     pthread_t threads[3];
13     int thread_ids[3];
14     for (int i = 0; i < 3; i++) {
15         thread_ids[i] = i + 1;
16         pthread_create(&threads[i], NULL, worker, &thread_ids[i]);
17     }
18     for (int i = 0; i < 3; i++) {
19         pthread_join(threads[i], NULL);
20     }
21     printf("Main thread: All threads have finished.\n");
22     return 0;
23 }
```

Terminal Output:

```
shanza1209@DESKTOP-LIS8TT7:~/Labs_OS/Lab5$ gcc Task5.c -o task5 -lpthread
shanza1209@DESKTOP-LIS8TT7:~/Labs_OS/Lab5$ ./task5
Thread 1: Starting task...
Thread 2: Starting task...
Thread 3: Starting task...
Thread 1: Task completed!
Thread 2: Task completed!
Thread 3: Task completed!
Main thread: All threads have finished.
```

Explanation:

- Three threads execute the same function concurrently.
- Output order may vary because threads run in parallel.
- Demonstrates *basic multithreading behavior* and *non-deterministic execution order*.

Program 2: Demonstrating a Race Condition

Objective: What happens when multiple threads modify a shared variable **without synchronization**.

```

#include <stdio.h>
#include <pthread.h>

int counter = 0; // Shared variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++; // Not thread-safe
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

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

    printf("Expected counter value: 200000\n");

    printf("Actual counter value:  %d\n", counter);

    return 0;
}

```

Compile and run:

```

gcc race_condition.c -o race_condition -lpthread
./race_condition

```

The screenshot shows a Visual Studio Code editor window with a C file named `Task6.c`. The code defines a shared counter and two threads that increment it. The terminal output shows the program was compiled and run, but the actual counter value (136540) is significantly lower than the expected value (200000).

```
#include <pthread.h>
int counter = 0; // Shared variable
void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++; // Not thread-safe
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Expected counter value: 200000\n");
    printf("Actual counter value:  %d\n", counter);
    return 0;
}
```

```
shanza1209@DESKTOP-LI88TJ:~/Labs_OS/Lab5$ gcc Task6.c -o task6 -lpthread
shanza1209@DESKTOP-LI88TJ:~/Labs_OS/Lab5$ ./task6
Expected counter value: 200000
Actual counter value: 136540
shanza1209@DESKTOP-LI88TJ:~/Labs_OS/Lab5$
```

Observation:

- The final counter is often **less than 200000**.
- This happens because both threads read and write counter simultaneously — a **race condition**.

Concept introduced:

When multiple threads access shared data without control, results become unpredictable.

Synchronization will be used to solve this.