

# Operating Systems – COC 3071L

SE 5th A – Fall 2025

---

## Lab 4: 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/Lab 4$ gcc Task1.c -o task1 -lpthread
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab 4$ ./task1
Main thread starting...
Main Thread ID: 132116684797760
Hello from the new thread!
Thread ID: 132116681586368
Main thread exiting...
shanza1209@DESKTOP-LIS8TTJ:~/Labs_05/Lab 4$

```

## 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>&amp;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);

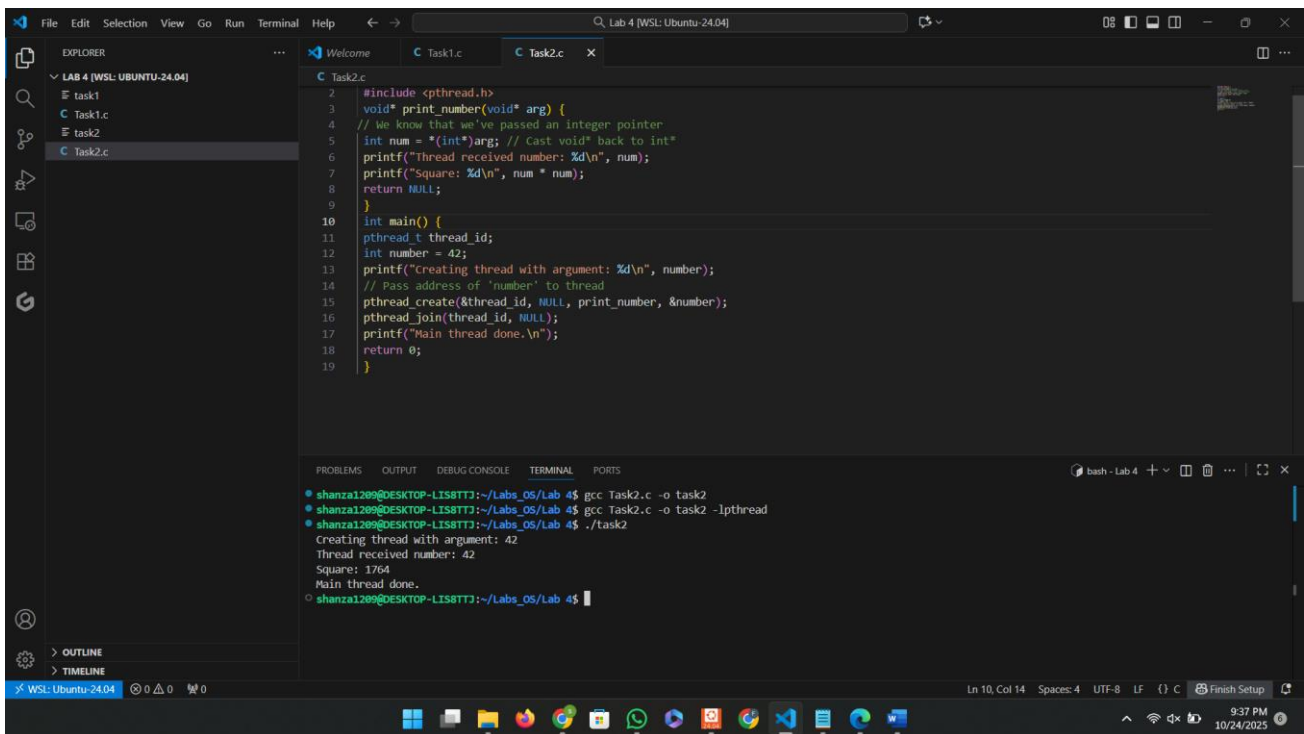
    // Pass address of 'number' to thread
    pthread_create(&thread_id, NULL, print_number, &number);

    pthread_join(thread_id, NULL);

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

**Compile and run:**

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



## 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;
}

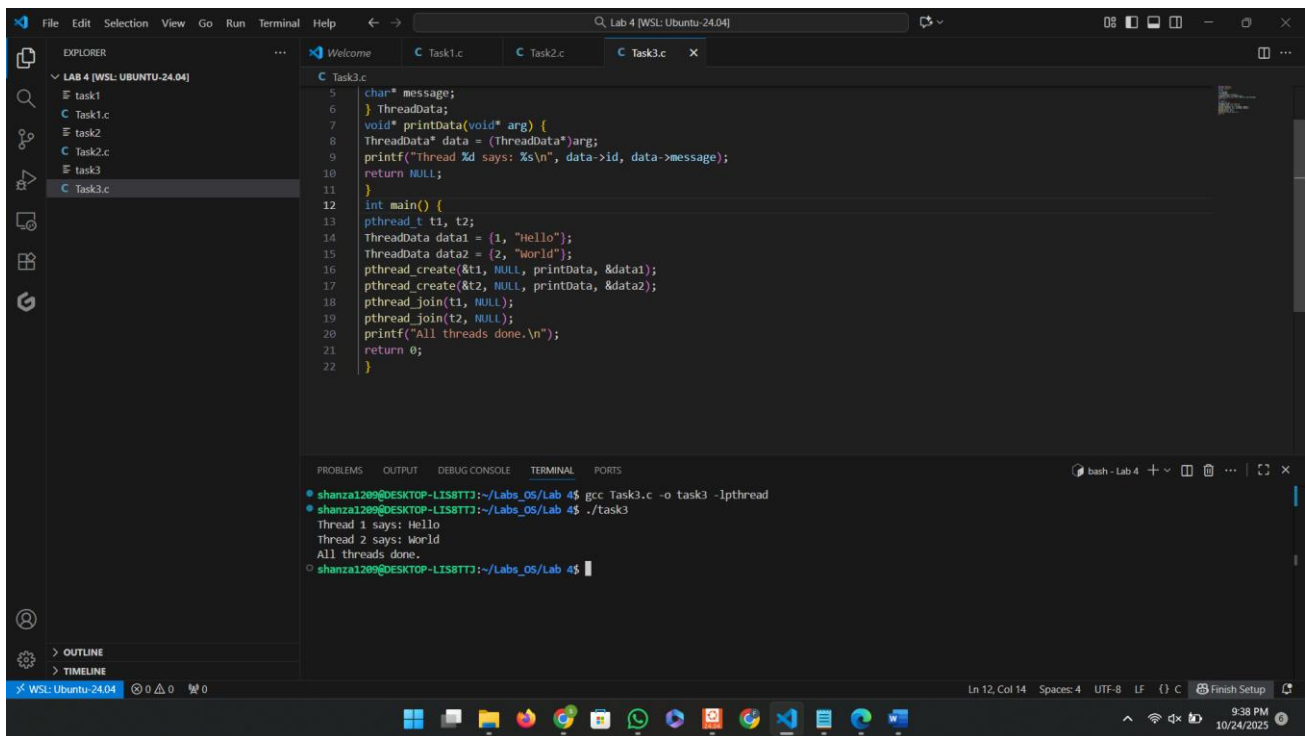
int main() {
    pthread_t t1, t2;

    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;
}
```



## Program 4: Multiple Threads

**Objective:** Create multiple threads executing the same function.

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

```

#include <unistd.h>

void* worker_thread(void* arg) {
    int thread_num = *(int*)arg;

    printf("Thread %d: Starting work...\n", thread_num);
    sleep(1); // Simulate some work
    printf("Thread %d: Work completed!\n", thread_num);

    return NULL;
}

int main() {
    pthread_t threads[5];
    int thread_args[5];

    // Create 5 threads
    for (int i = 0; i < 5; i++) {
        thread_args[i] = i + 1;
        printf("Main: Creating thread %d\n", i + 1);
        pthread_create(&threads[i], NULL, worker_thread, &thread_args[i]);
    }

    // Wait for all threads to complete
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
        printf("Main: Thread %d has finished\n", i + 1);
    }

    printf("All threads completed!\n");
    return 0;
}

```

### Compile and run:

```

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

```



The screenshot shows the Visual Studio Code editor with a C program named `Task4.c` open. The program creates 5 threads, each performing a simple task (printing their ID and 'Starting work...'). The main thread prints 'Main: Creating thread X' for each thread. The threads are created in a loop, and the main thread waits for all of them to complete using `pthread_join`. The terminal output shows the execution of the program, demonstrating that the threads do not necessarily execute in the order they were created, but all complete before the main thread finishes.

```
10 }
11 int main() {
12     pthread_t threads[5];
13     int thread_args[5];
14     // Create 5 threads
15     for (int i = 0; i < 5; i++) {
16         thread_args[i] = i + 1;
17         printf("Main: Creating thread %d\n", i + 1);
18         pthread_create(&threads[i], NULL, worker_thread, &thread_args[i]);
19     }
20     // Wait for all threads to complete
21     for (int i = 0; i < 5; i++) {
22         pthread_join(threads[i], NULL);
23         printf("Main: Thread %d has finished\n", i + 1);
24     }
25     printf("All threads completed!\n");
26     return 0;
}
```

Terminal Output:

```
shanza1209@DESKTOP-LI8BTJ3:~/Labs_05/Lab 4$ gcc Task4.c -o task4 -lpthread
shanza1209@DESKTOP-LI8BTJ3:~/Labs_05/Lab 4$ ./task4
Main: Creating thread 1
Main: Creating thread 2
Thread 1: Starting work...
Main: Creating thread 3
Thread 2: Starting work...
Main: Creating thread 4
Thread 3: Starting work...
Main: Creating thread 5
Thread 4: Starting work...
Thread 5: Starting work...
Thread 5: Work completed!
Thread 1: Work completed!
Thread 2: Work completed!
Thread 3: Work completed!
Main: Thread 1 has finished
Main: Thread 2 has finished
Main: Thread 3 has finished
```

## Observation:

- Notice how threads may not execute in order
- All threads run concurrently
- `pthread_join()` ensures we wait for all threads

## Program 5: 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 thread4.c -o thread4 -lpthread
./thread4

```

The screenshot shows the Visual Studio Code editor with a C file named `Task5.c`. The code defines a `main` function that creates a thread to calculate the sum of integers from 1 to 100. The thread function `calculate_sum` returns the sum. The main function prints the result and frees the memory. The terminal output shows the compilation and execution of the program, resulting in the output: `Thread calculated sum of 1 to 100 = 5050` and `Main received result: 5050`.

```
13 }
14 int main() {
15     pthread_t thread_id;
16     int n = 100;
17     void* sum;
18     pthread_create(&thread_id, NULL, calculate_sum, &n);
19     // Get the return value from thread
20     pthread_join(thread_id, &sum);
21     printf("Main received result: %d\n", *(int*)sum);
22     free(sum); // Don't forget to free allocated memory
23     return 0;
24 }
```

```
shanza1209@DESKTOP-LIS8TT3:~/Labs_05/Lab 4$ gcc Task5.c -o task5 -lpthread
shanza1209@DESKTOP-LIS8TT3:~/Labs_05/Lab 4$ ./task5
Thread calculated sum of 1 to 100 = 5050
Main received result: 5050
shanza1209@DESKTOP-LIS8TT3:~/Labs_05/Lab 4$
```

## Key Points:

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

## 5. Hands-on Practice Exercises

### Exercise 1: Thread Basics

Write a program that:

1. Creates 3 threads
2. Each thread prints its thread ID and a unique message
3. Main thread waits for all threads to complete

### Exercise 2: Prime Number Checker

Write a program that:

1. Takes a number as input
2. Creates a thread that checks if the number is prime
3. Returns the result to the main thread
4. Main thread prints whether the number is prime or not