**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:

The

-

lpthre

a

d

flag links the pthread library.

g

cc

progr

a

m

.

c

-

o

progr

a

m

-

lpthre

a

d

## 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.



//

Wa

it

for

the

thre

a

d

to

finish

pthre

a

d

\_

join

(

thre

a

d

\_

id

,

NULL

)

;

printf

(

"

Ma

in

thre

a

d

exiting

...\

n

"

)

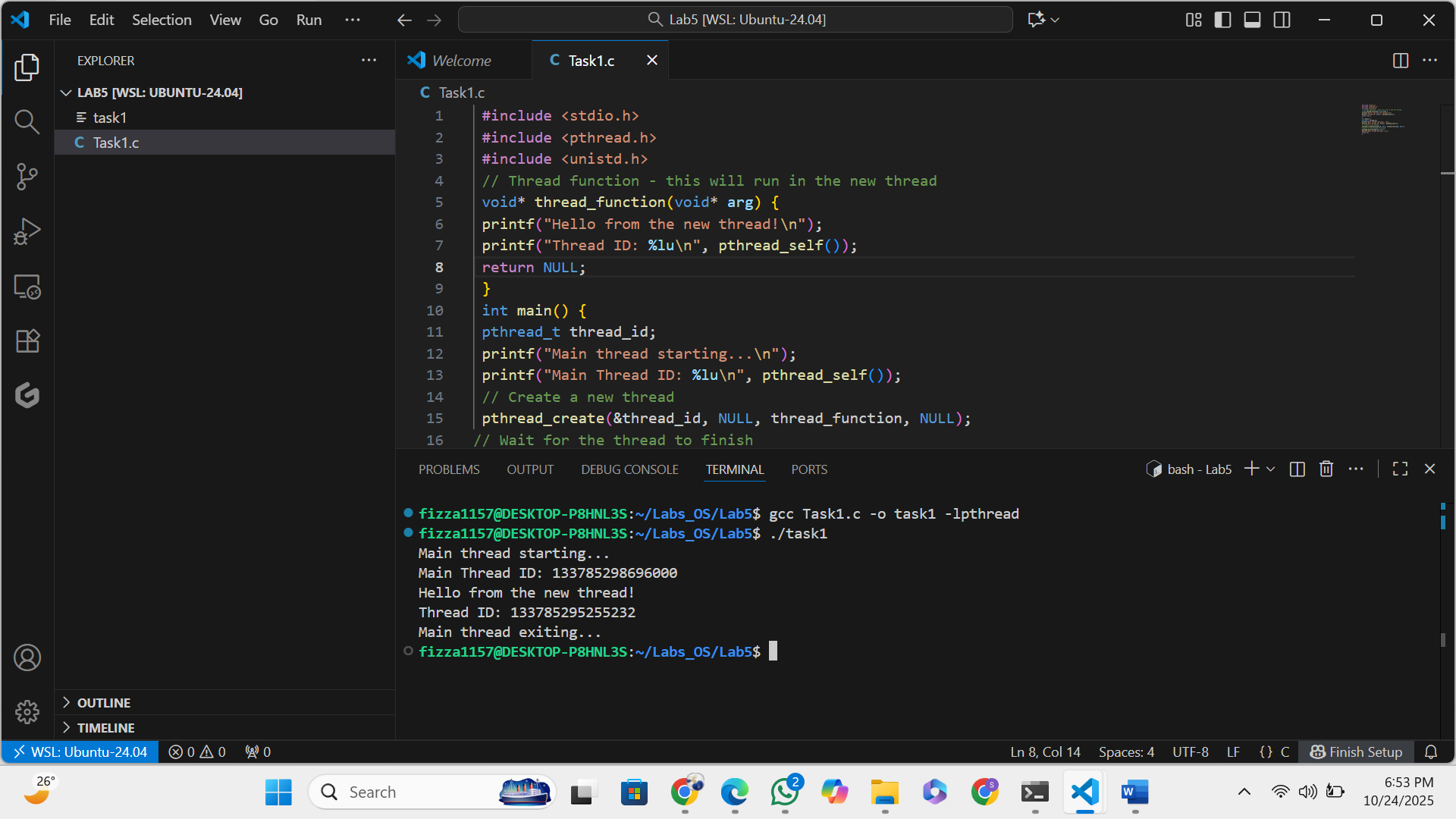
;

return

0

;

}



**Compile and run:**

g

cc

thre

a

d

1

.

c

-

o

thre

a

d

1

-

lpthre

a

d

./

thre

a

d

1

**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** |
| &thread | pthread\_t\* | Where the new thread ID will be stored |
| NULL | pthread\_attr\_t\* | 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 |

pthre

a

d

\_

join

()

→ Waits for thread to finish (like

w

a

it

()

for processes)

pthre

a

d

\_

self

()

→ Returns the thread ID of calling thread

### Program 2: Passing Arguments to Threads

**Objective:** Pass data to a thread function.

#

in

c

lude

<

stdio

.

h

>

#

in

c

lude

<

pthre

a

d

.

h

>

void

\*

print

\_

num

b

er

(

void

\*

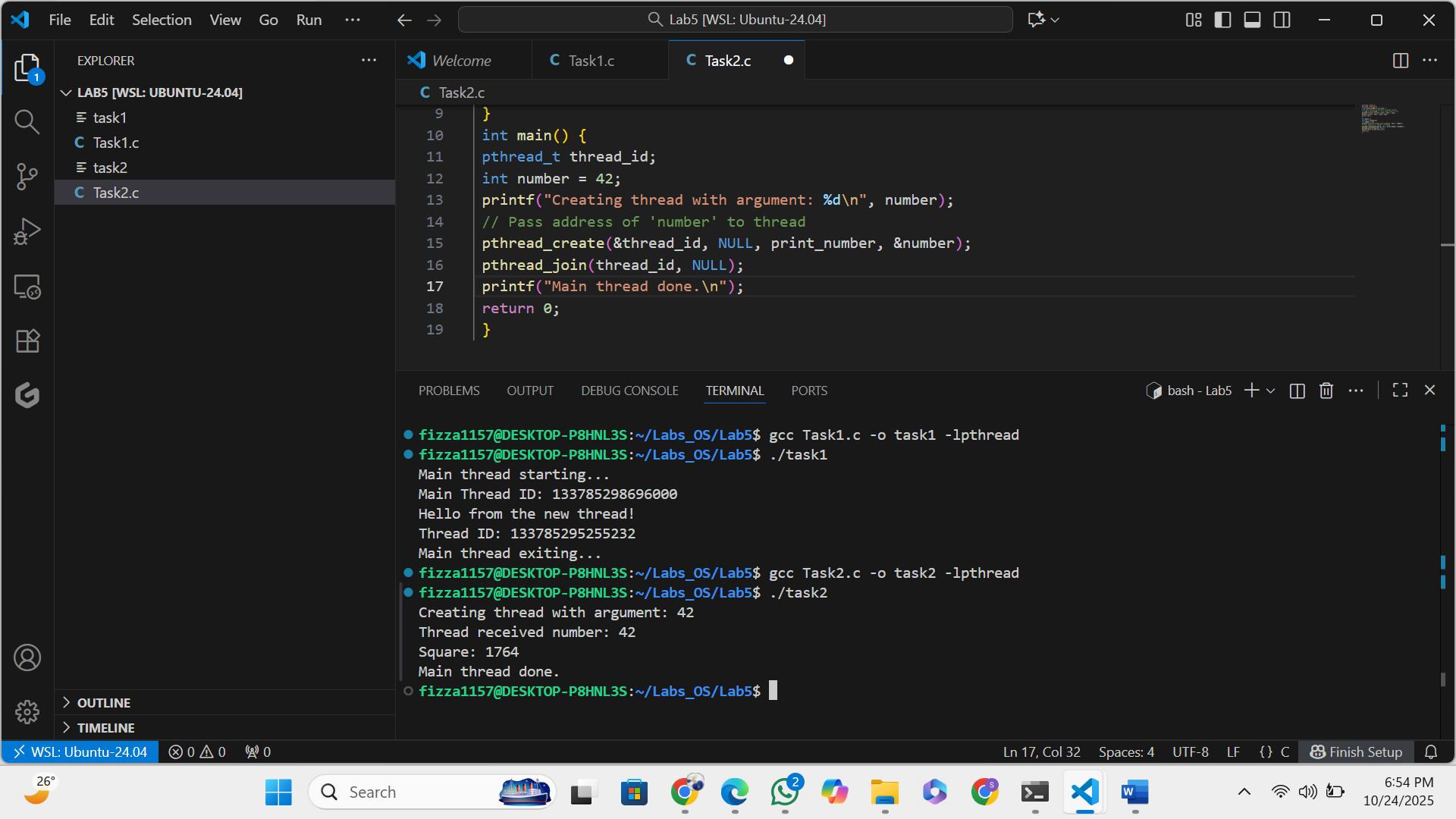
a

rg

)

{





**Compile and run:**

g

cc

thre

a

d

2

.

c

-

o

thre

a

d

2

-

lpthre

a

d

./

thre

a

d

2

**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:

**Why use**

**void**

**\***

int

v

a

lue

=

\*

(

int

\*

)

a

rg

;

1

.

(

int

\*)

a

rg

— cast

void

\*

back to

int

\*

.

.

2

\*(

int

\*)

a

rg

— dereference the pointer to get the integer value it points to.

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

void

\*

fun

c

tion

\_

n

a

me

(

void

\*

a

rg

)

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

stru

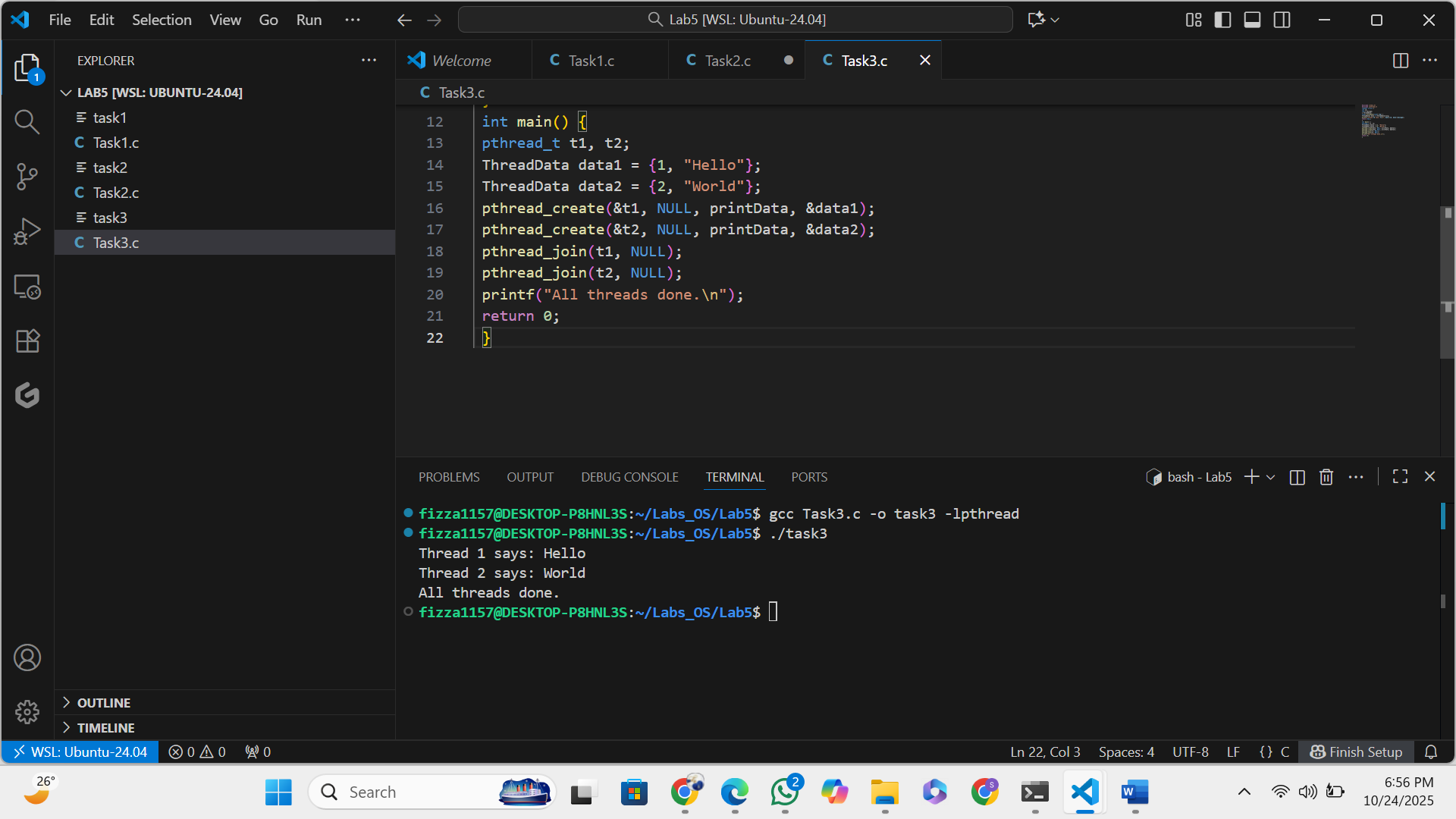
c

t

and pass a pointer to it.

### Program 3: Passing Multiple Data





**Compile and run:**

g

cc

thre

a

d

3

.

c

-

o

thre

a

d

3

-

lpthre

a

d

./

thre

a

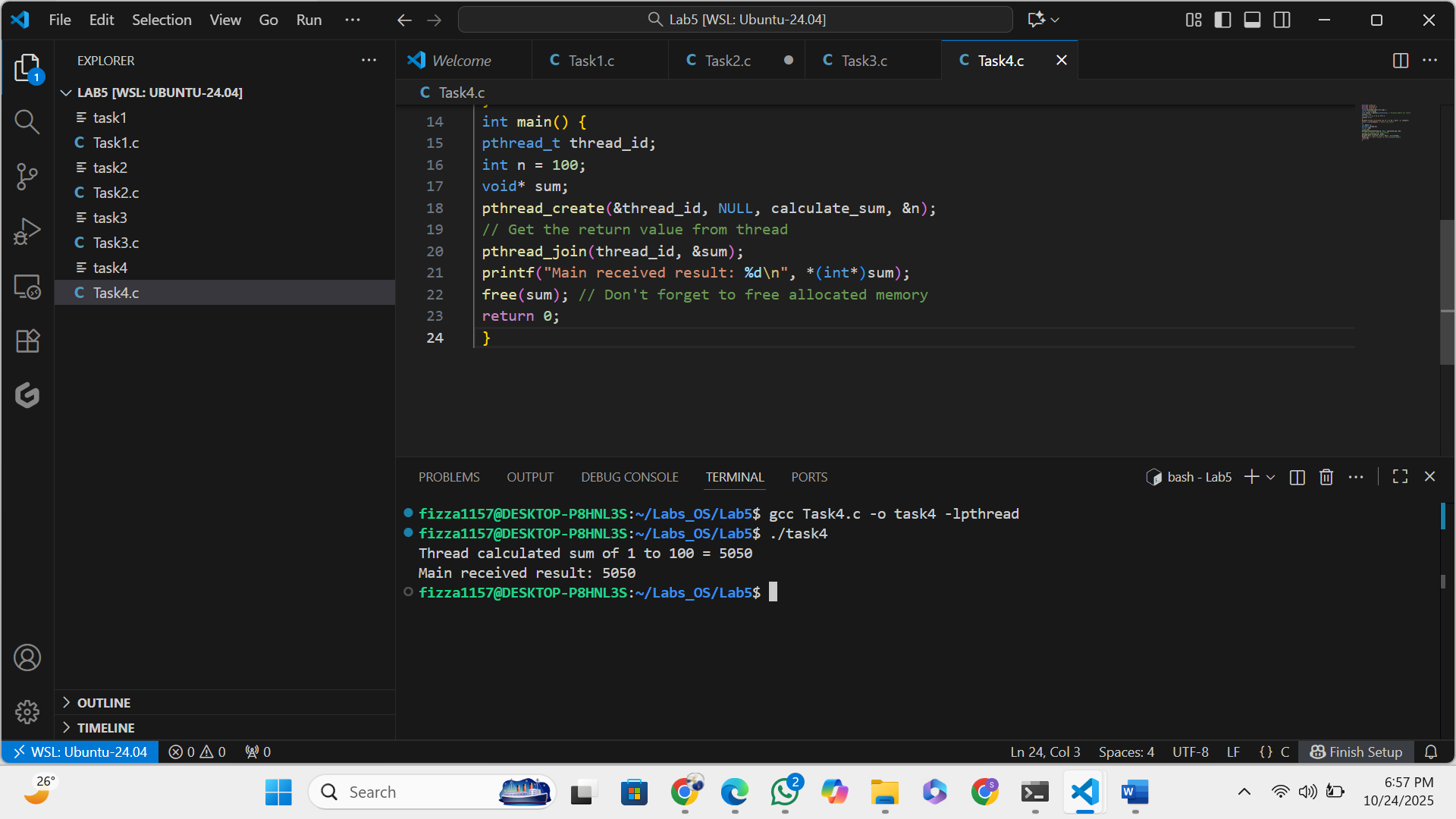
d

3

### Program 4: Thread Return Values

**Objective:** Get return values from threads.





**Compile and run:**

g

cc

thre

a

d

5

.

c

-

o

thre

a

d

5

-

lpthre

a

d

./

thre

a

d

5

**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.



pthre

a

d

\_

join

(

thre

a

ds

[

i

]

,

NULL

)

;

}

printf

(

"

Ma

in

thre

a

d

:

A

ll

thre

a

ds

h

a

ve

finished

.\

n

"

)

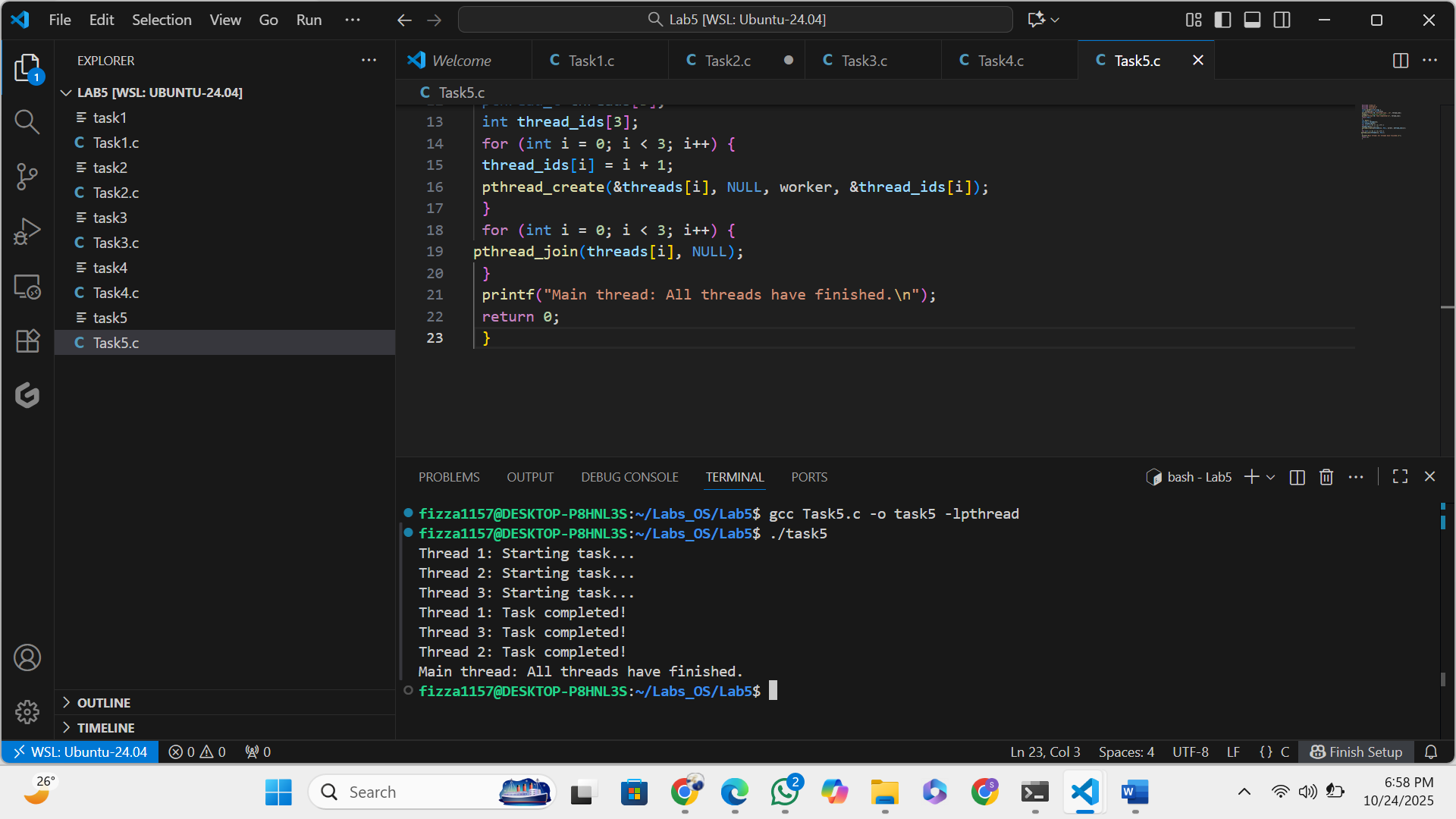
;

return

0

;

}



**Compile and run:**

g

cc

multithre

a

d

\_

ba

si

c

.

c

-

o

multithre

a

d

\_

ba

si

c

-

lpthre

a

d

./

multithre

a

d

\_

ba

si

c

**Explanation:**

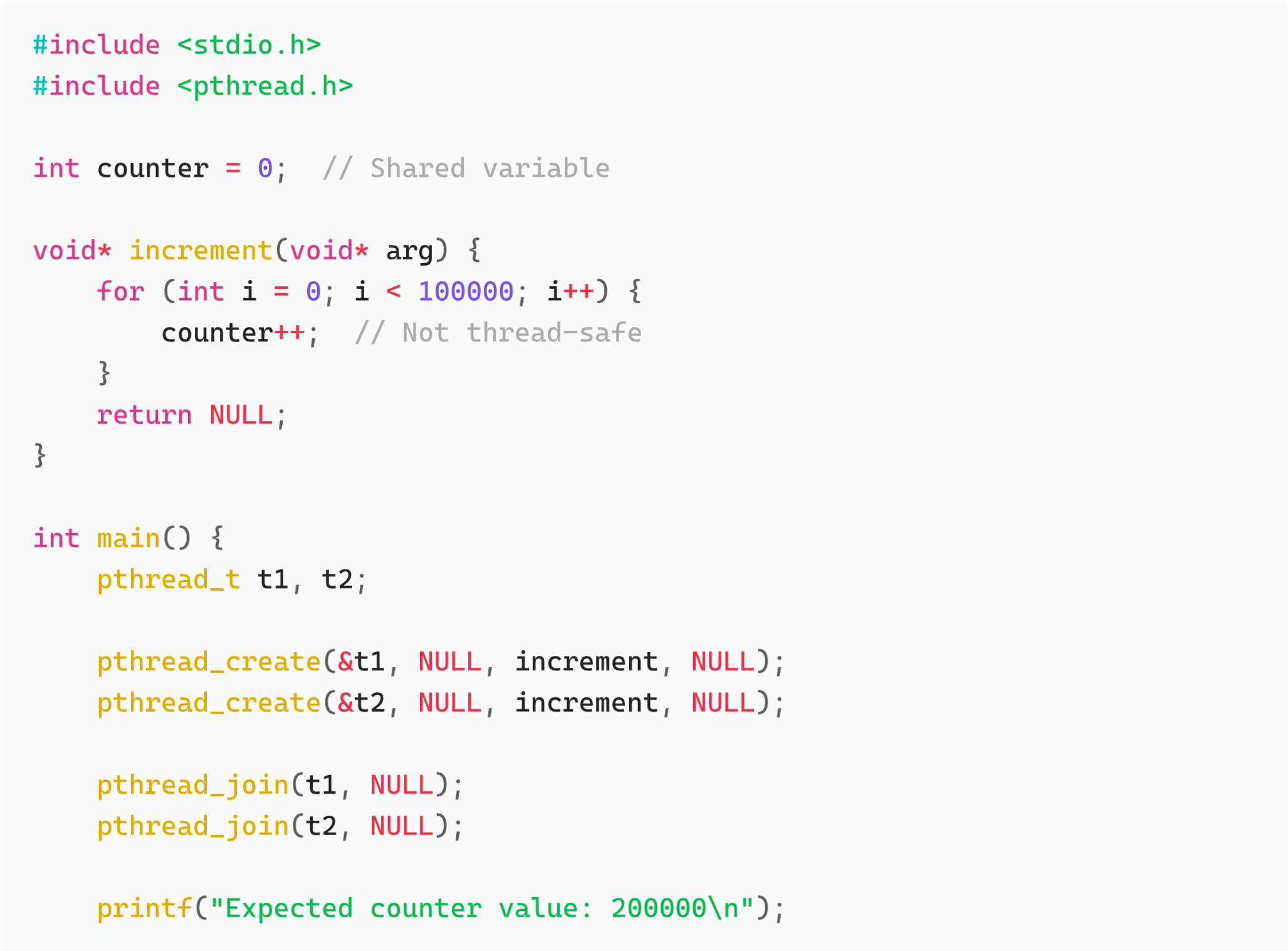
Three threads execute the same function concurrently.

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**.



printf

(

"

Ac

tu

a

l

c

ounter

v

a

lue

:

%

d

\

n

"

,

c

ounter

)

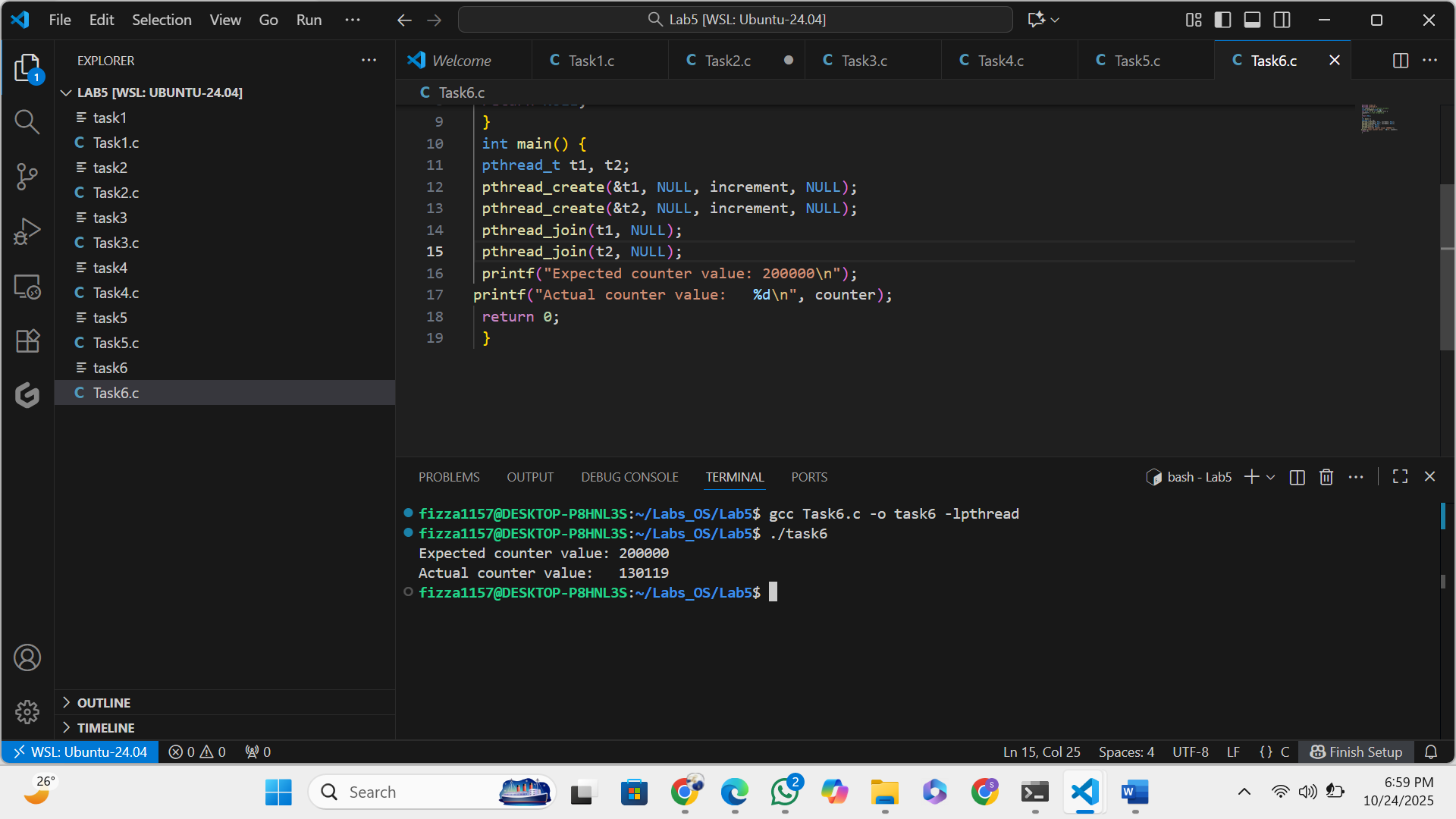
;

return

0

;

}



**Compile and run:**

g

cc

r

ac

e

\_

c

ondition

.

c

-

o

r

ac

e

\_

c

ondition

-

lpthre

a

d

./

r

ac

e

\_

c

ondition

**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.