**Operating Systems – COC 3071L**

**SE 5th A – Fall 2025**

# Introduction

A **process** is simply a program in execution.

When you type a command in Linux (like ls ), the OS creates a process for it.

Every process has:

**PID (Process ID)** → unique number for each process.

**PPID (Parent Process ID)** → ID of the process that created it.

**State** → running, sleeping, stopped, zombie, etc.

In this lab, you will:

1. Learn Linux commands to monitor and manage processes.
2. Write C programs to create and observe processes.

# Linux Process Commands

## Viewing Processes

**ps**

**→ Process Status**

Output example:

Shows processes in the current terminal session.

ps

PID

TTY

TIME

CMD

1234

pts

/

0

00

:

00

:

00

ba

sh

1256

pts

/

0

00

:

00

:

00

ps

**PID**

→ Process ID

**TTY**

→ terminal

**TIME**

→ CPU time used

**CMD**

→ command name

**ps**

**-**

**ef**

**→ Full list of all processes**

ps

-

ef

-

e

→ show all processes (not just yours).

-

f

→ full format with UID, PPID, etc.

Try:

This finds all processes related to the

ba

sh

shell.

ps

-

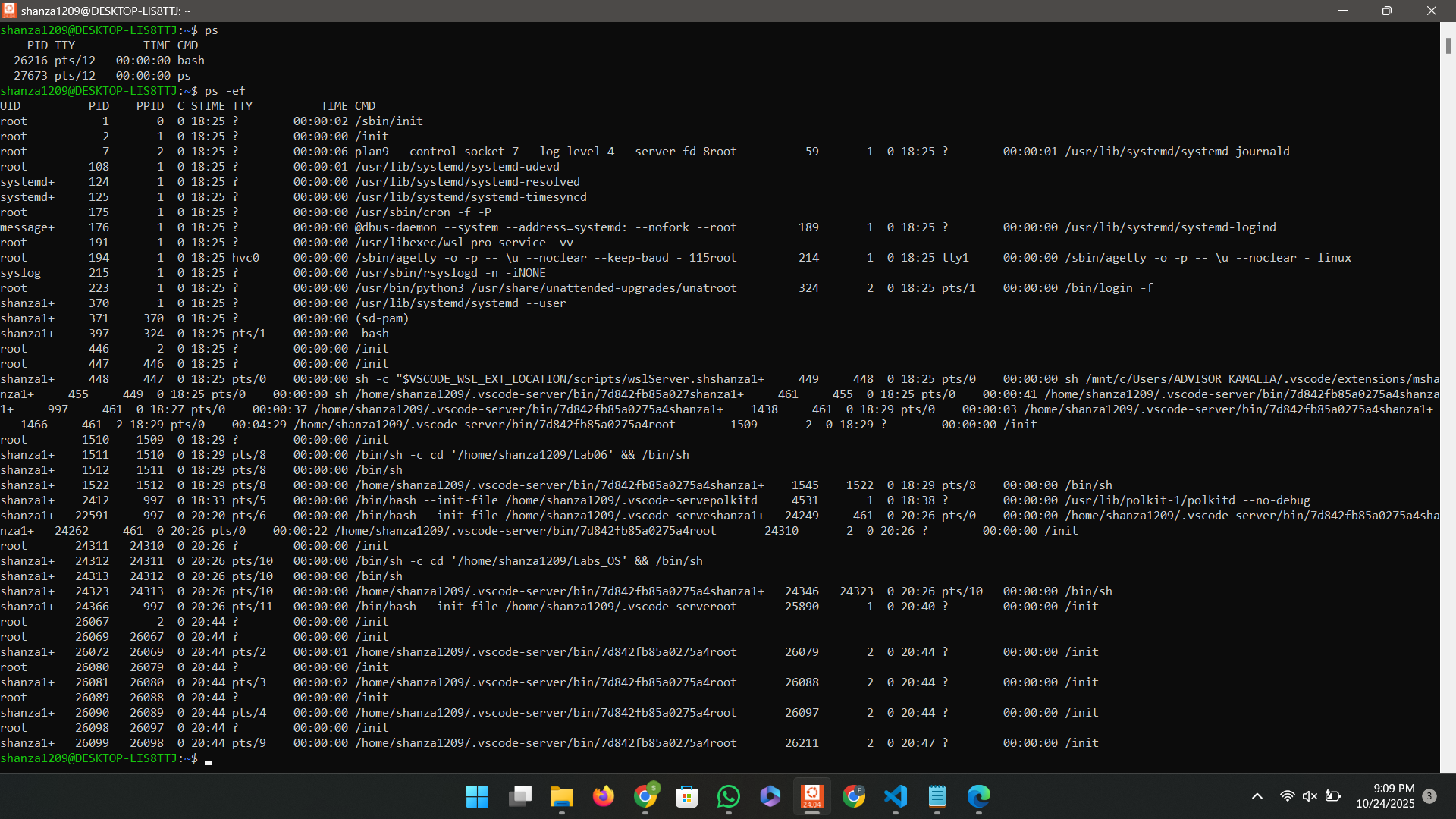
ef

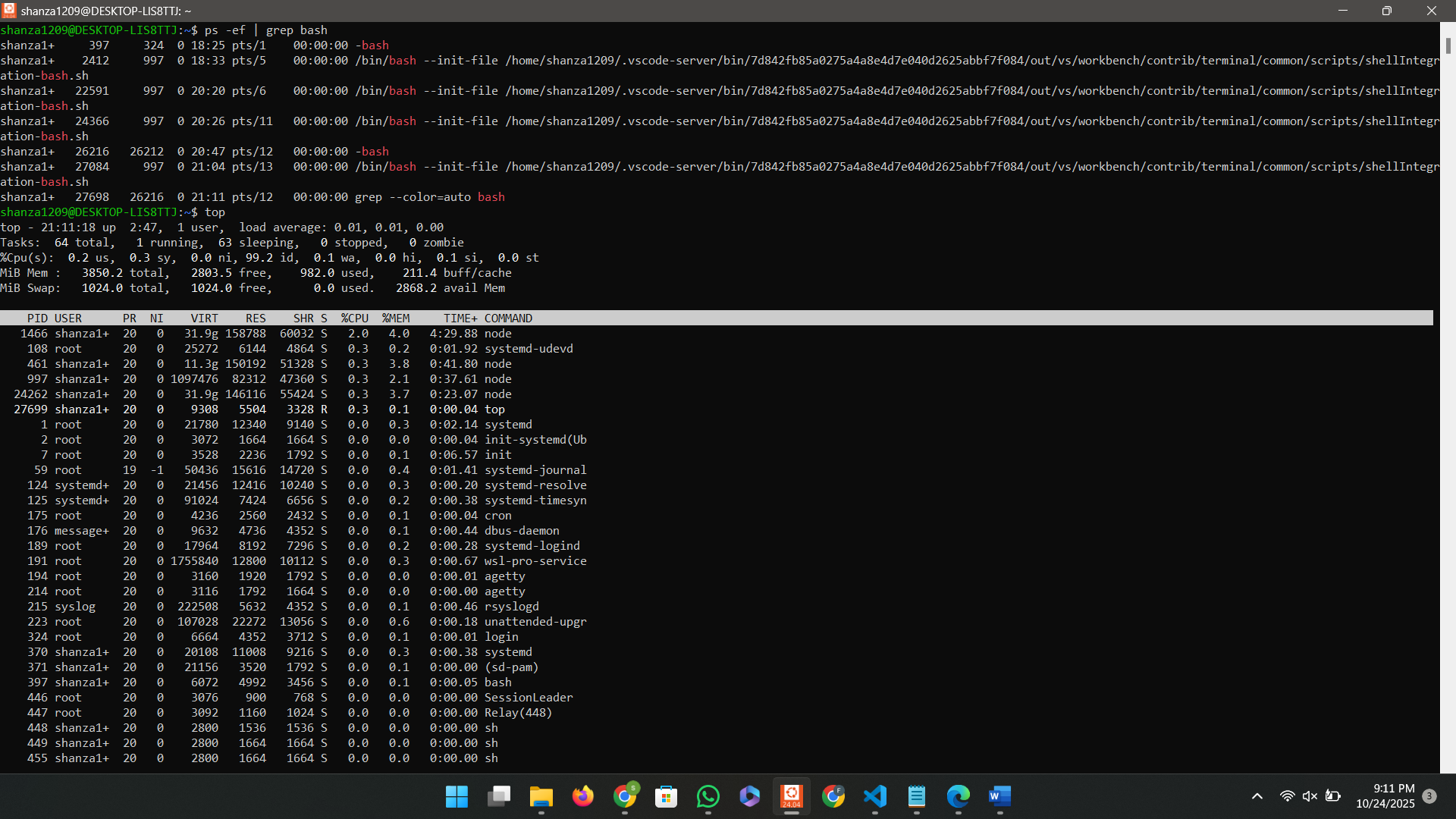
|

grep

ba

sh





## Monitoring Processes Interactively

**top**

**→ Dynamic process viewer**

top

Displays running processes with CPU and memory usage.

Press

q

to quit.

Press

k

inside

top

to kill a process (enter PID).

Press

h

for help.

## Foreground and Background Jobs

**Foreground**

:

A process that takes control of the terminal until it finishes.

sleep

30

→ You cannot type new commands until it finishes.

**Background**: Add & to run without blocking.

sleep

30

&

→ Terminal is free while the command runs.

**Check background jobs**:

jo

b

s

**Bring a job to foreground**

:

%

1

means job number 1 (from

jo

b

s

output).

fg

%

1

**Suspend a job**

:

Press

**Ctrl + Z**

while it runs.

**Resume suspended job in background**

:

b

g

%

1

## Process Identification

**Get PID of a process by name**

:

Example output:

3421

(PID of sleep command).

pidof

sleep

**Search using**

**ps**

**and**

**grep**

:

ps

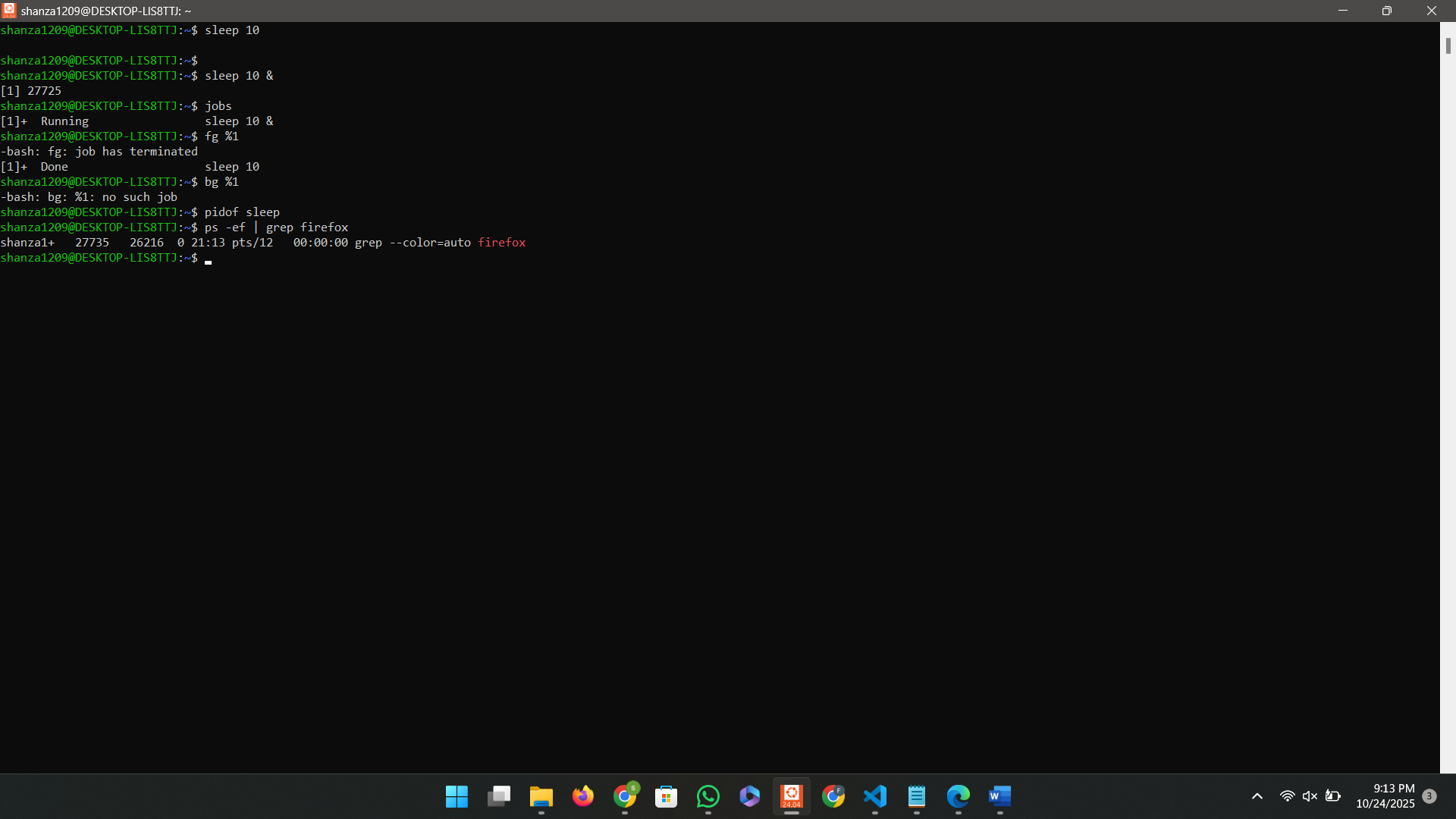
-

ef

|

grep

firefox



## Killing Processes

**Practice Task**

:

**Kill by PID**

:

kill

-

9

3421

-

9

→ force kill (SIGKILL).

**Kill all processes by name**

:

kill

a

ll

sleep

1. Run an infinite process:

(

yes

prints “y” forever; redirected to

/

dev

/

null

to hide output).

yes

>

/

dev

/

null

&

1. Find it with:
2. Kill it with:

kill

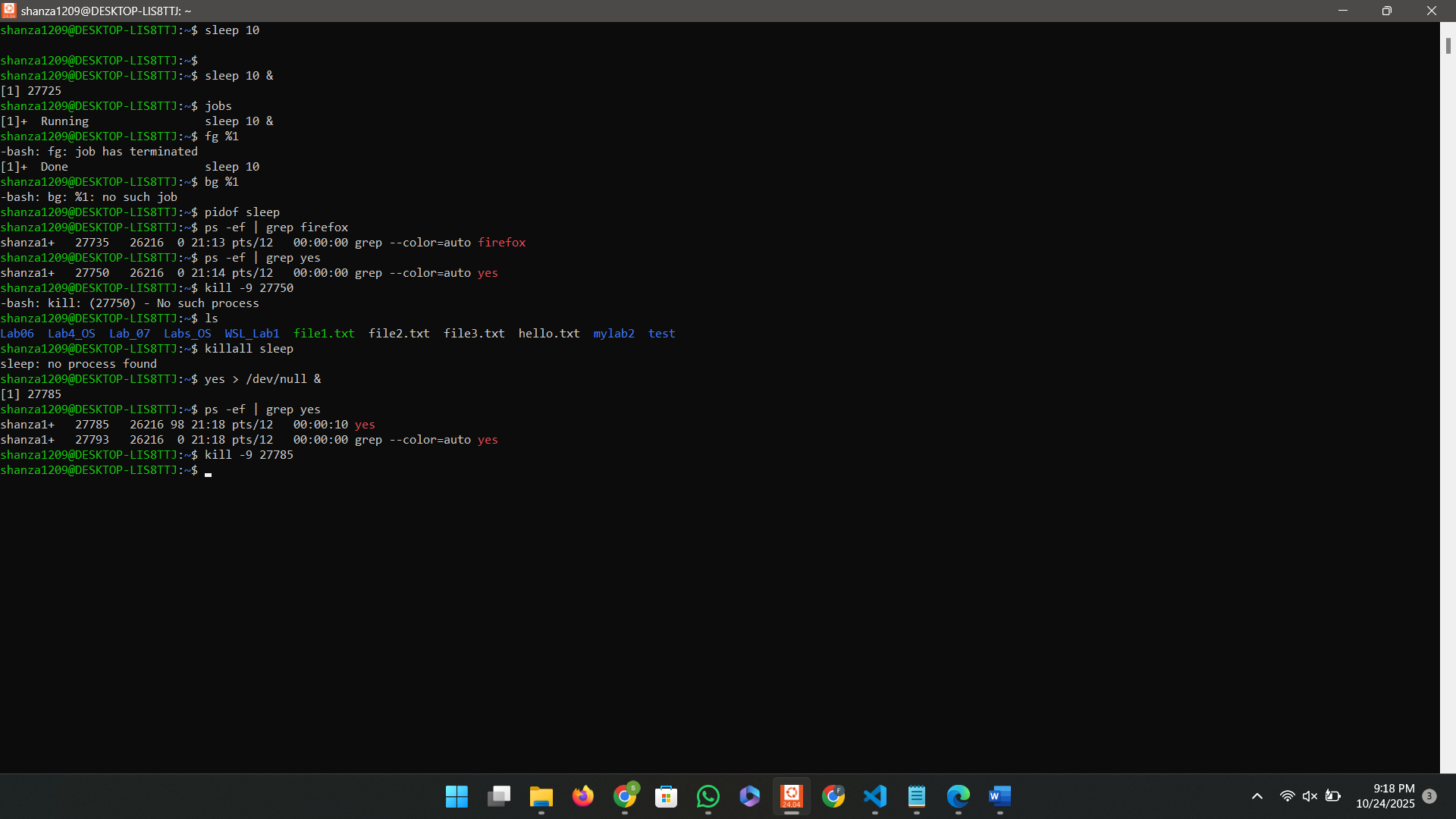
-

9

<

PID

>



# C Programs on Processes

## Program 1: Print PID and PPID

ps

-

ef

|

grep

yes

#

in

c

lude

<

stdio

.

h

>

#

in

c

lude

<

unistd

.

h

>

int

m

a

in

(

)

{

printf

(

"

M

y

PID

:

%

d

\

n

"

,

getpid

(

)

)

;

printf

(

"

M

y

Pa

rent

PID

:

%

d

\

n

"

,

getppid

(

)

)

;

return

0

;

}

#

in

c

lude

<

unistd

.

h

>

→ contains process-related functions like

getpid

()

and

getppid

()

.

getpid

()

→ returns the unique

**process ID**

of the current process.

getppid

()

→ returns the

**parent’s PID**

.

Every process in Linux has a parent (except the very first process, usually

init

or

systemd

).

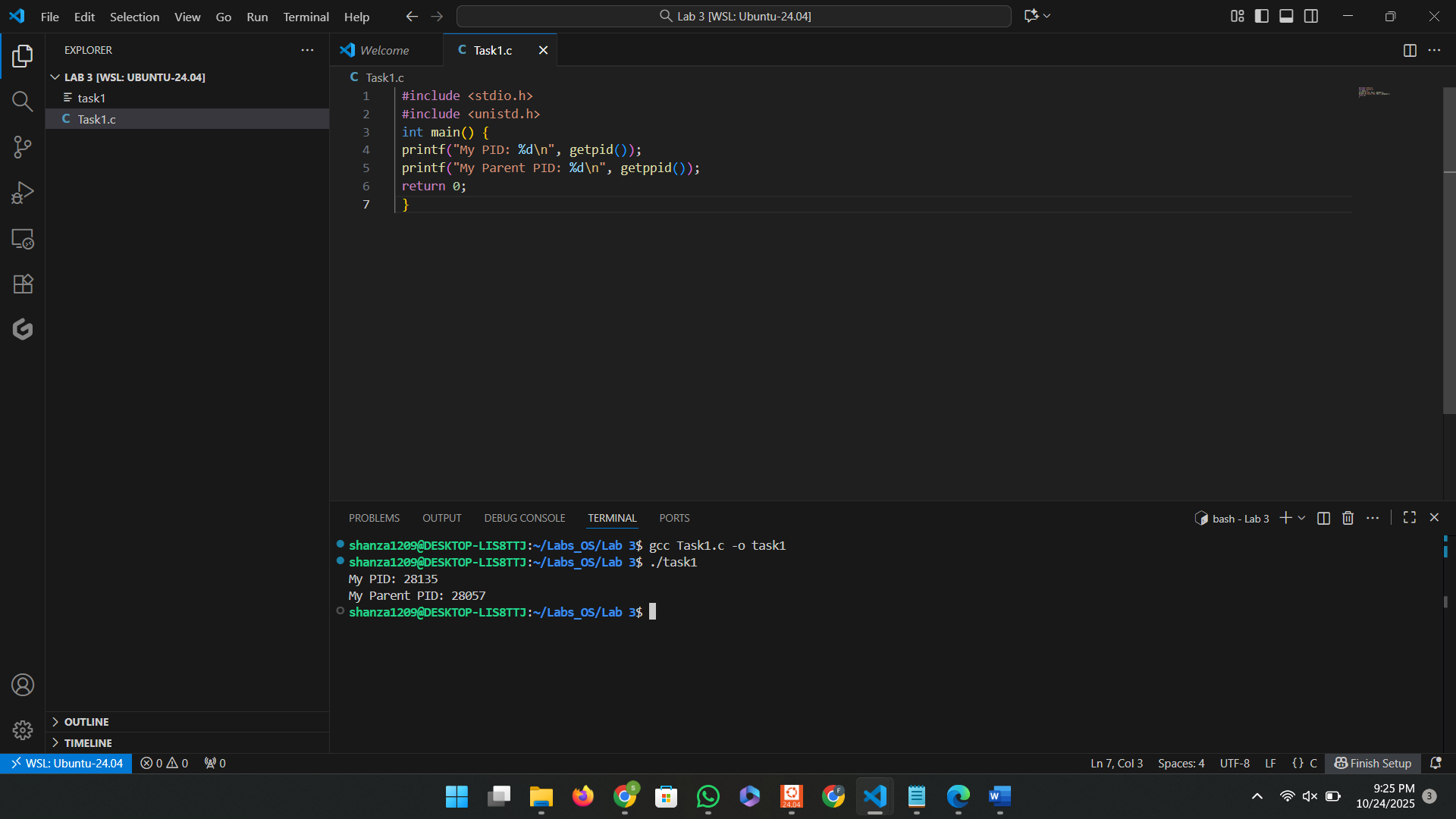
Run and compare with

ps

-

ef

.



## Program 2: Fork – Creating Child Process

#

in

c

lude

<

stdio

.

h

>

#

in

c

lude

<

unistd

.

h

>

int

m

a

in

(

)

{

pid

\_

t

pid

=

fork

(

)

;

if

(

pid

==

0

)

{

//

T

his

b

lo

c

k

runs

in

the

c

hild

pro

c

ess

printf

(

"

C

hild

:

PID

=

%

d

,

Pa

rent

=

%

d

\

n

"

,

getpid

(

)

,

getppid

(

)

)

;

}

else

{

//

T

his

b

lo

c

k

runs

in

the

p

a

rent

pro

c

ess

printf

(

"

Pa

rent

:

PID

=

%

d

,

C

hild

=

%

d

\

n

"

,

getpid

(

)

,

pid

)

;

}

return

0

;

}

fork

()

creates a new process by duplicating the current one.

Return value of

fork

()

:

0

→ you are inside the

**child**

process.

Positive number (child PID) → you are in the

**parent**

process.

After

fork

()

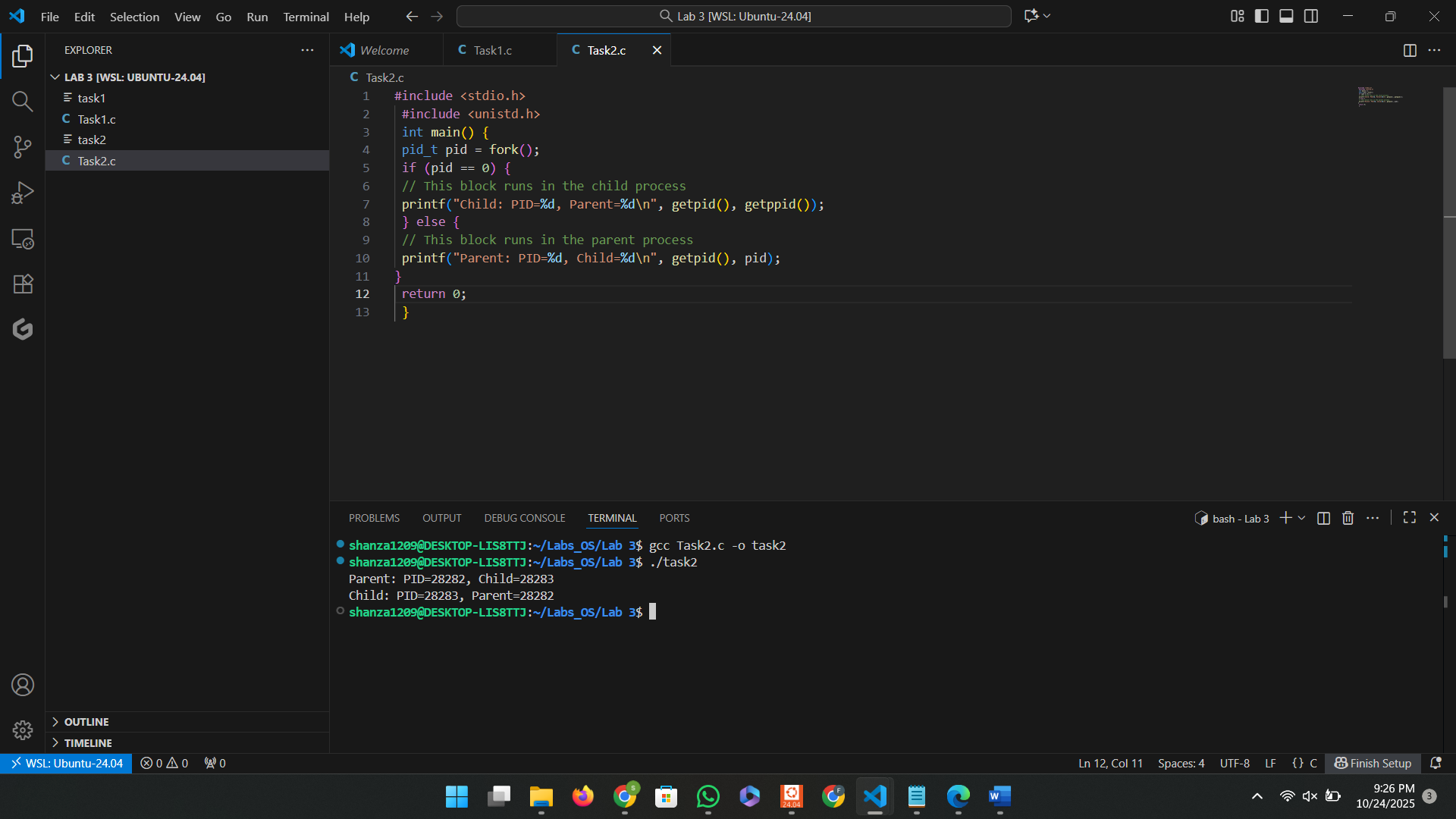
, both parent and child run

**the same code**

, but in different branches of the

if

.



## Program 3: Execl – Replacing a Process

After exec() , the child **no longer runs our C code** – it becomes ls .

#

in

c

lude

<

stdio

.

h

>

#

in

c

lude

<

unistd

.

h

>

int

m

a

in

(

)

{

pid

\_

t

pid

=

fork

(

)

;

if

(

pid

==

0

)

{

exe

c

lp

(

"

ls

"

,

"

ls

"

,

"-

l

"

,

NULL

)

;

printf

(

"

T

his

will

not

print

if

exe

c

su

cc

eeds

.\

n

"

)

;

}

else

{

printf

(

"

Pa

rent

still

running

...\

n

"

)

;

}

return

0

;

}

fork

()

→ creates child.

In the child:

exe

c

lp

"

(

ls

", "

ls

", "-

l

",

NULL

)

;

Replaces the

**current process image**

with the

ls

program.

First

"

ls

"

= name of the program, second

"

ls

"

= argument 0 (how program

sees itself).

"-

l

"

= argument for

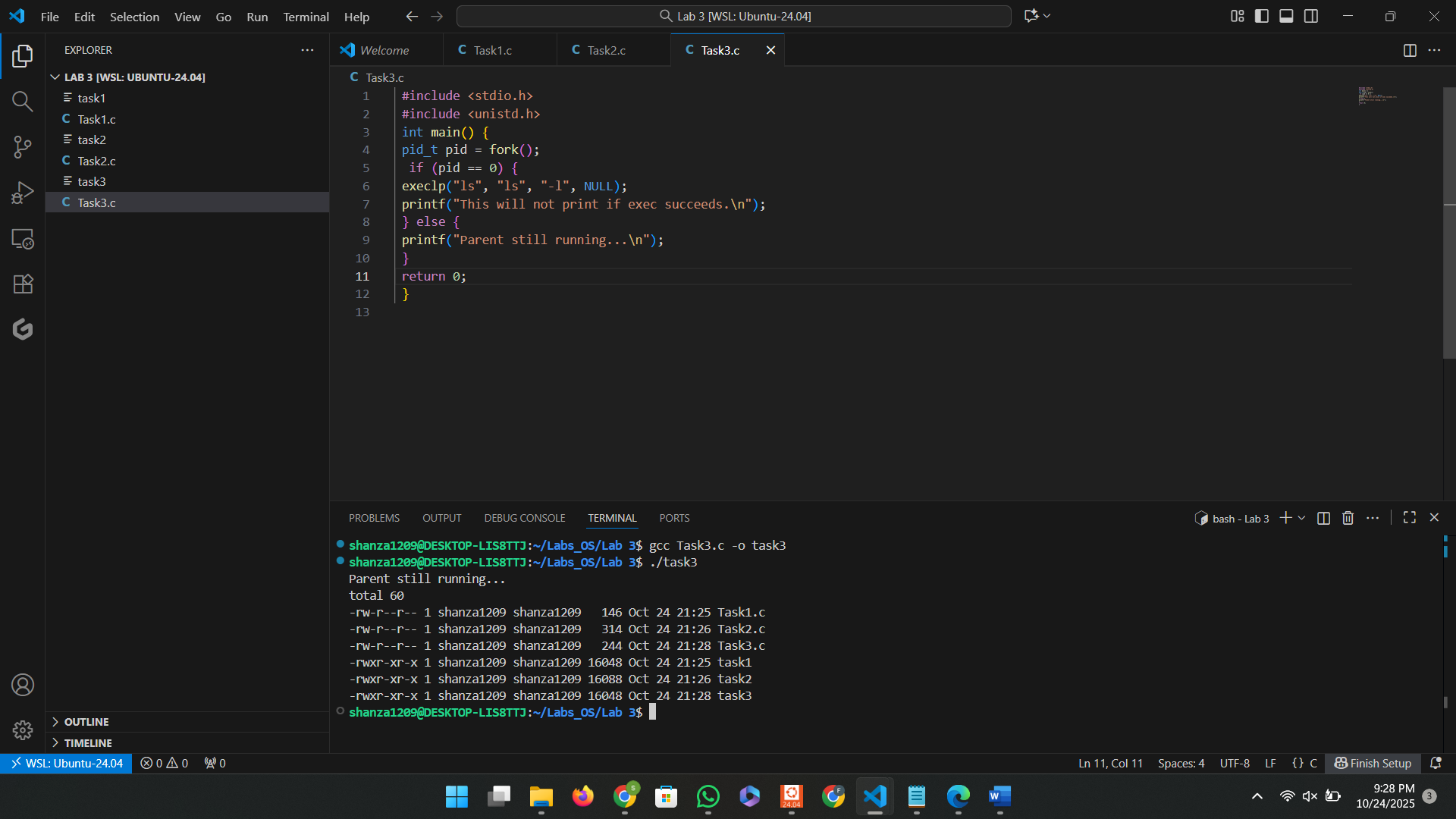
ls

.

NULL

marks end of arguments.

Parent is unaffected and continues normally.



## Program 4: Wait – Synchronization

#

in

c

lude

<

stdio

.

h

>

#

in

c

lude

<

unistd

.

h

>

#

in

c

lude

<

sys

/

w

a

it

.

h

>

int

m

a

in

(

)

{

pid

\_

t

pid

=

fork

(

)

;

if

(

pid

==

0

)

{

exe

c

lp

(

"

ls

"

,

"

ls

"

,

"-

l

"

,

NULL

)

;

printf

(

"

T

his

will

not

print

if

exe

c

su

cc

eeds

.\

n

"

)

;

}

else

{

w

a

itpid

(

pid

,

NULL

,

0

)

;

//

Wa

it

for

the

c

hild

pro

c

ess

to

finish

printf

(

"

Pa

rent

still

running

...\

n

"

)

;

}

return

0

;

}

fork

()

→ creates child.

sleep

(

3

)

→ child "works" for 3 seconds.

w

a

it

(

NULL

)

→ parent pauses until child exits.

Without

w

a

it

()

, parent may finish early and child could become a

**zombie process**

.

