بسم الله الرحمن الرحيم

Introduction to Linux Operating System

Lab 01

اللهم علمنا ما ينفعنا ،،،   وانفعنا بما علمتنا ،،،   وزدنا علماً

Lab Objective

- To introduce some of the common Linux commands

Introduction

- *Linux* is a clone of the *Unix* operating system.
- Unix was developed in 1969 by Dennis Ritchie and Kevin Thompson at Bell Laboratories.
- Most of the Unix operating system is written in the high-level programming language C.
- A Unix operating system consists of a kernel and a set of common utility programs.
- The kernel is the core of the operating system, which manages the computer hardware, controls program executions, manages memory, etc.
- The utility programs provide user level commands, such as those to create and edit files.

Why Linux?

- Free, open source.
- Ubuntu is a complete Linux operating system
- At Ubuntu's heart is the Linux kernel
- Ubuntu has a graphical user interface (GUI), making it similar to other popular operating systems like Windows and Mac OS
- The OS represents applications as icons or menu choices that you can select using keyboard commands or a mouse

How to Use Ubuntu?

- Requirements:

1- VirtualBox (on your Windows or Mac computer)
https://www.wikihow.com/Install-VirtualBox

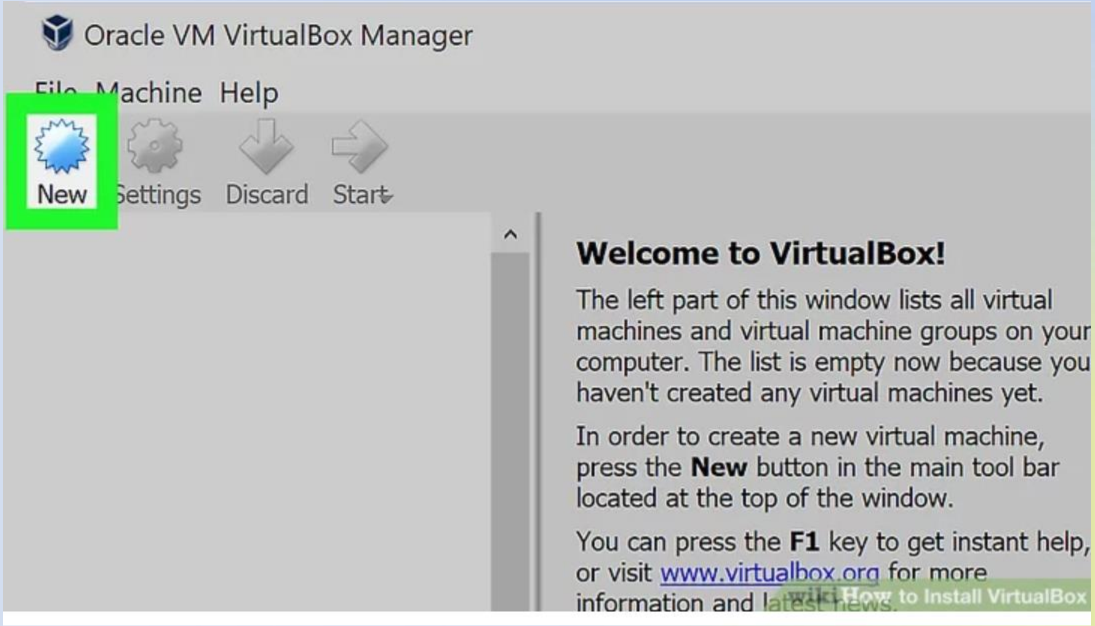
2- Ubuntu disk image (ISO File)
https://ubuntu.com/download/desktop

How to Use Ubuntu?

- Once you have download the VirtualBox:

  1- Install the Ubuntu operating system by using its ISO file on the Virtual Machine.

  2- Open VirtualBox and click on New tab.



Welcome to VirtualBox!

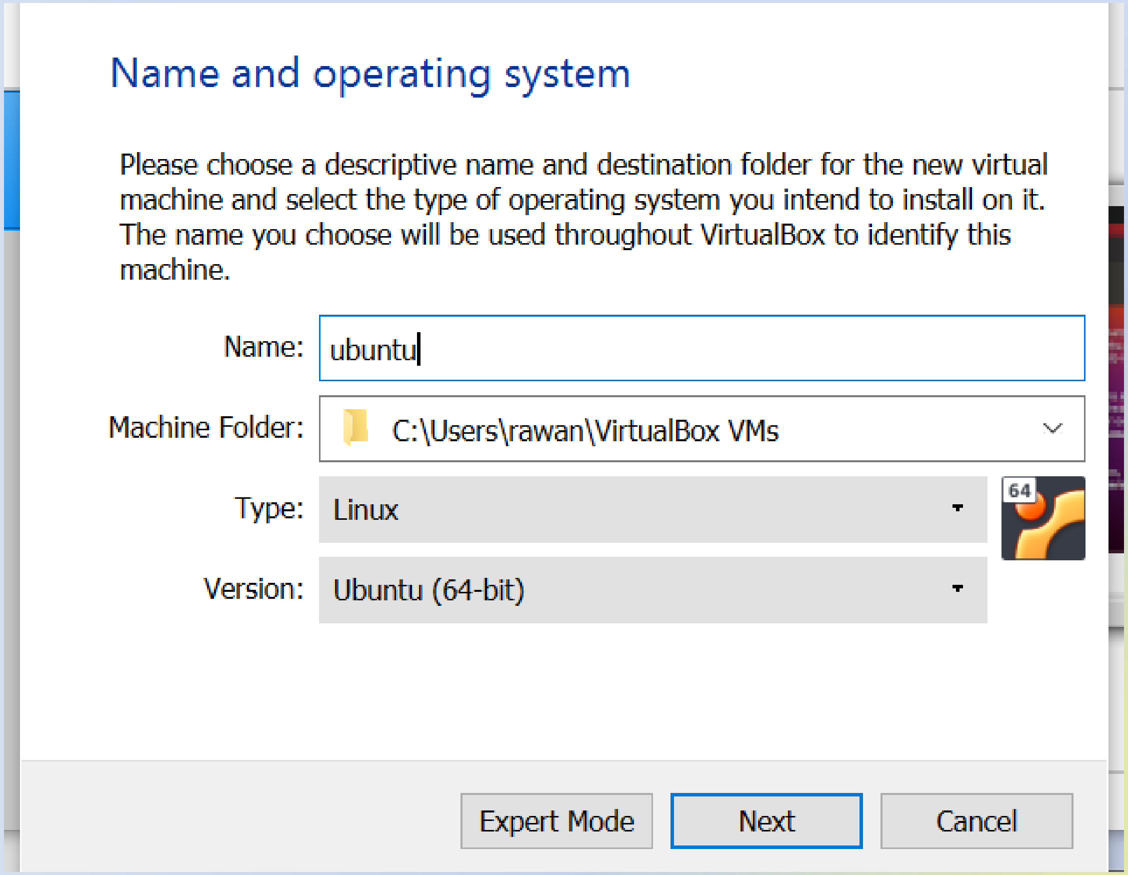
The left part of this window lists all virtual machines and virtual machine groups on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for more information and latest news.

How to Use Ubuntu?

3- Identify the operating system as following:



Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: ubuntu

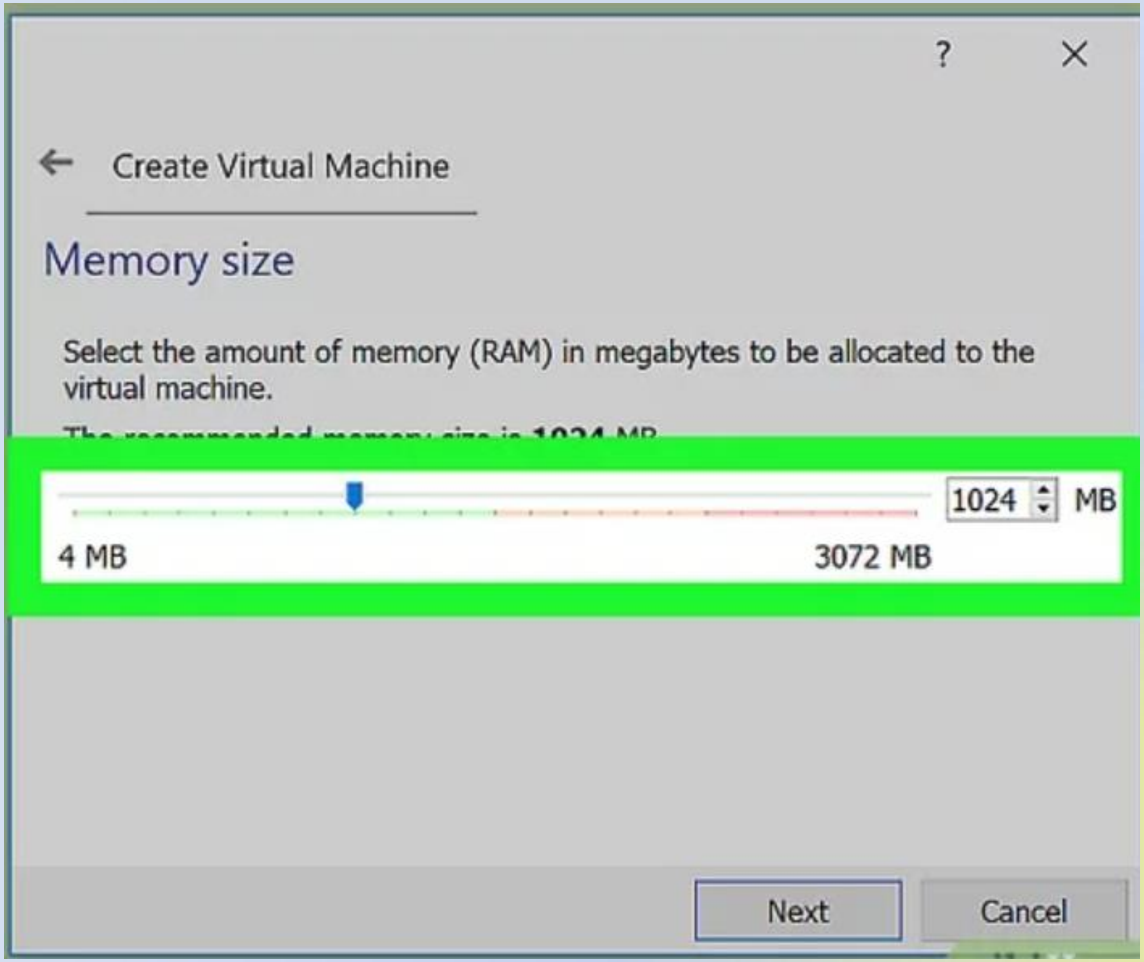
Machine Folder: C:\Users\rawan\VirtualBox VMs

Type: Linux

Version: Ubuntu (64-bit)

How to Use Ubuntu?

4- Set the amount of RAM as following:



Create Virtual Machine

Memory size

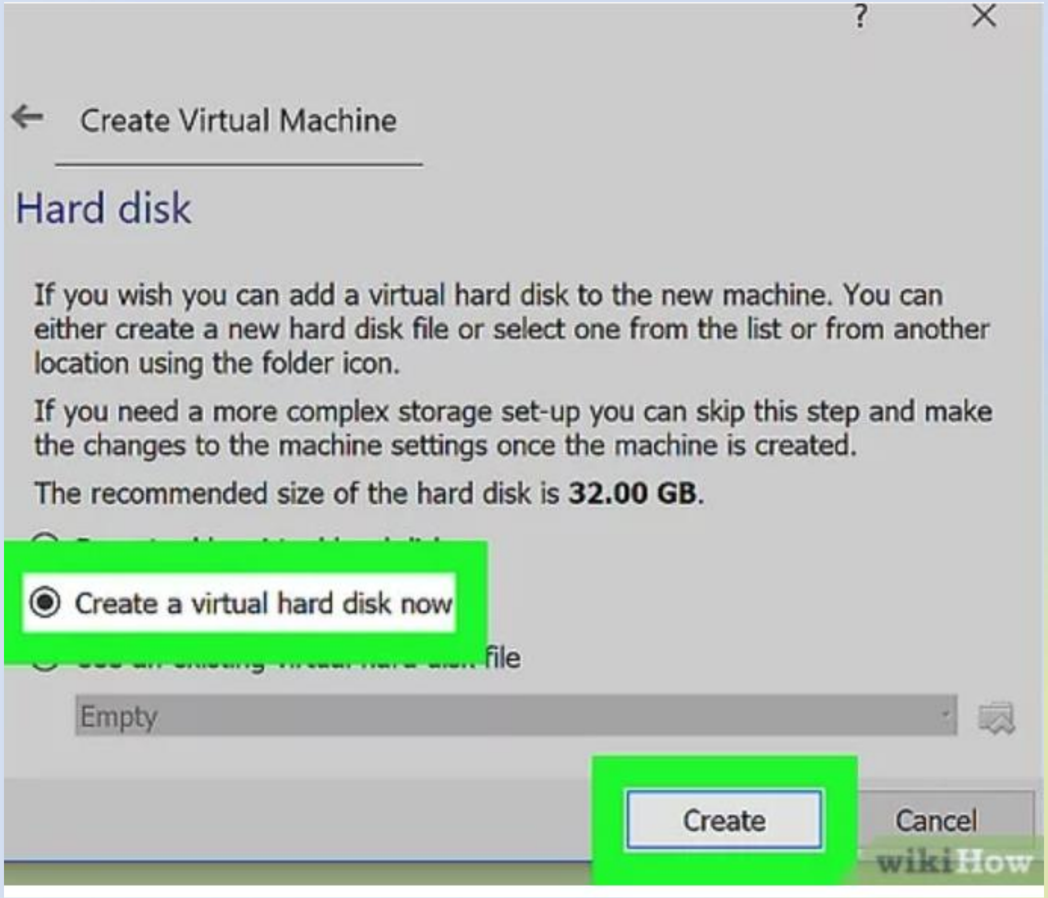
Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is 1024 MB.

1024 MB

How to Use Ubuntu?

5- Create a virtual hard drive as following:



Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard disk is **32.00 GB**.

○ Do not add a virtual hard disk

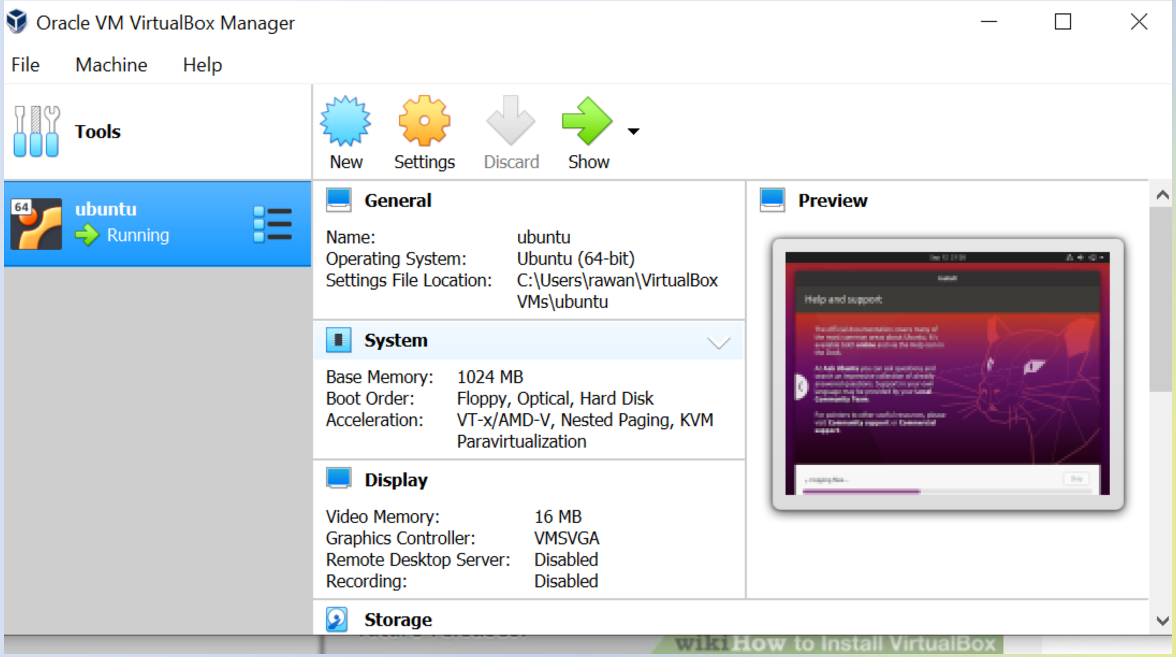
● Create a virtual hard disk now

○ Use an existing virtual hard disk file

Empty

How to Use Ubuntu?

6- Once the virtual machine has been configured, Start the operating system installation. Double-click your new machine (ubuntu) in the left menu, then browse through your computer for the installation image file



General

Name: ubuntu
Operating System: Ubuntu (64-bit)
Settings File Location: C:\Users\rawan\VirtualBox VMs\ubuntu

System

Base Memory: 1024 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization

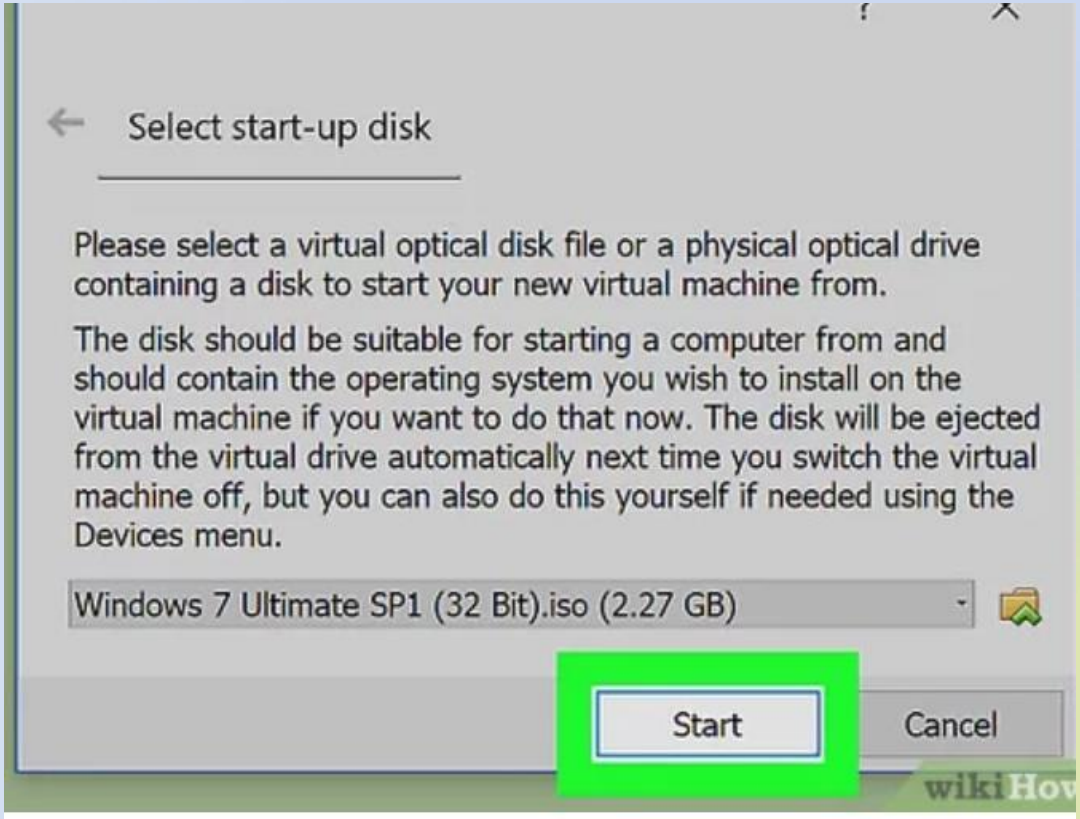
Display

Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage

How to Use Ubuntu?

7- Click Start to prompt VirtualBox to begin reading your ISO file.



Select start-up disk

Please select a virtual optical disk file or a physical optical drive containing a disk to start your new virtual machine from.

The disk should be suitable for starting a computer from and should contain the operating system you wish to install on the virtual machine if you want to do that now. The disk will be ejected from the virtual drive automatically next time you switch the virtual machine off, but you can also do this yourself if needed using the Devices menu.

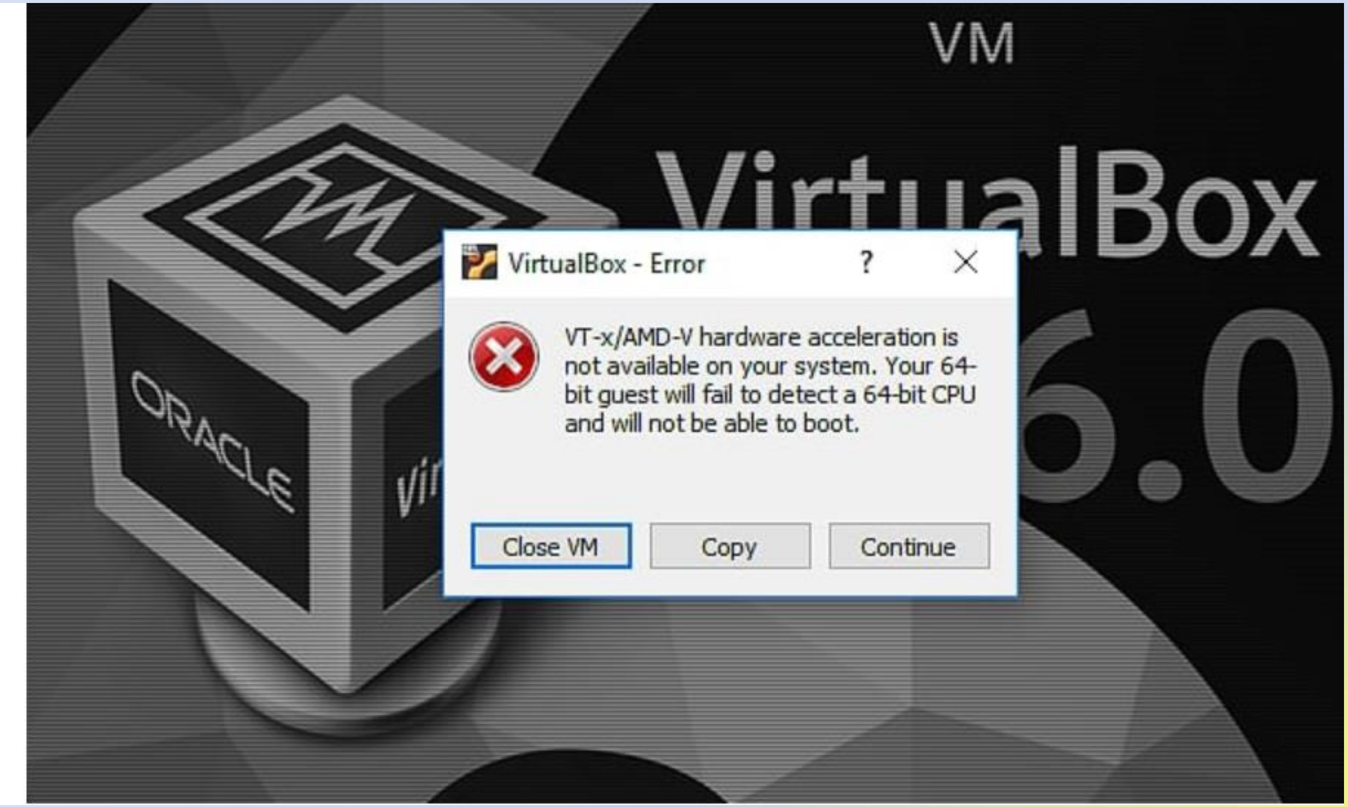
Windows 7 Ultimate SP1 (32 Bit).iso (2.27 GB)

How to Use Ubuntu?

8- Boot up your virtual machine. Once the operating system is installed, your virtual machine is ready to go. Simply double-click the name of your virtual machine in the left menu of the VirtualBox main page to start it up.

How to Use Ubuntu?

You may encounter an ERROR when you try to start up Ubuntu as this message shows:



VT-x/AMD-V hardware acceleration is not available on your system. Your 64-bit guest will fail to detect a 64-bit CPU and will not be able to boot.

How to Use Ubuntu?

To solve that problem, you have to enable Virtualization by Restarting your computer and booting. HOW?

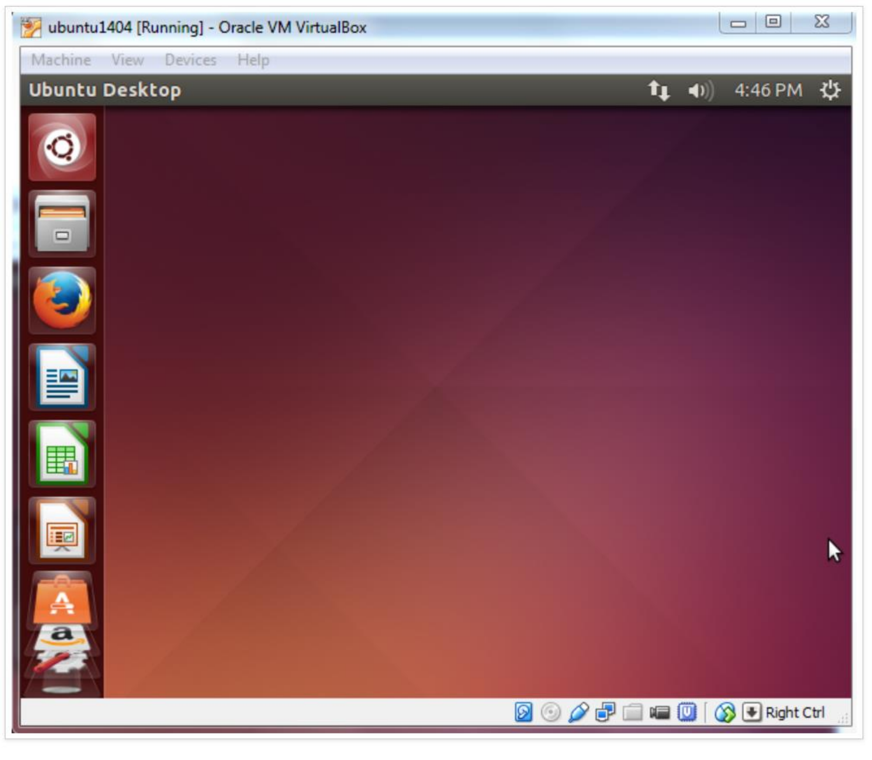
- By pressing F12 or F2 while starting the system to get the booting menu, then choose advanced setting and enable Virtualization or choose (VT-X / AMD V) from Virtualization menu.
- Save the changes and exit.

How to Use Ubuntu?

Once there is no problem, Install Ubuntu as described in the following link.

https://brb.nci.nih.gov/seqtools/installUbuntu.html#install

Ubuntu

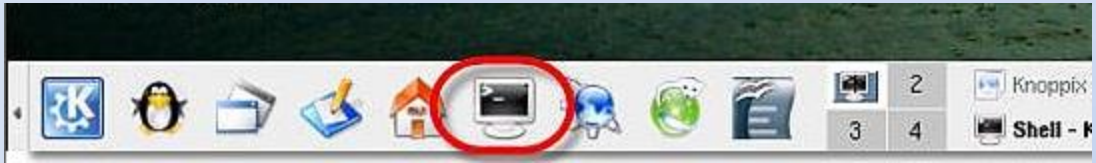


- This screenshot shows the Ubuntu desktop. A Web browser opens by default. You can minimize or close it to get it out of the way.

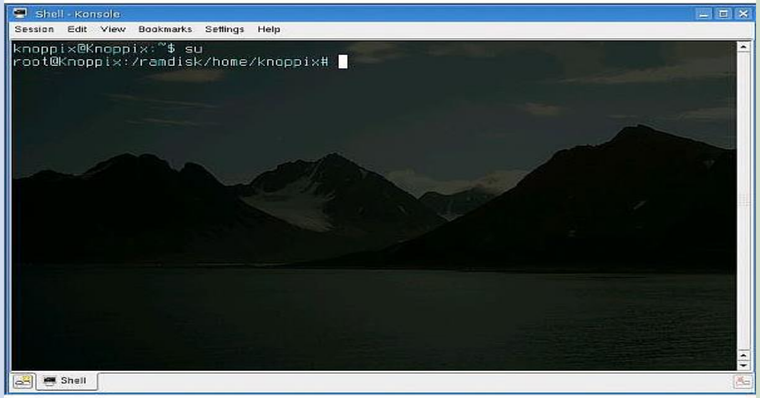
LINUX COMMANDS OVERVIEW

Starting an UNIX Terminal

- To open an UNIX terminal window, click on the "Terminal" icon in the lunch bar.



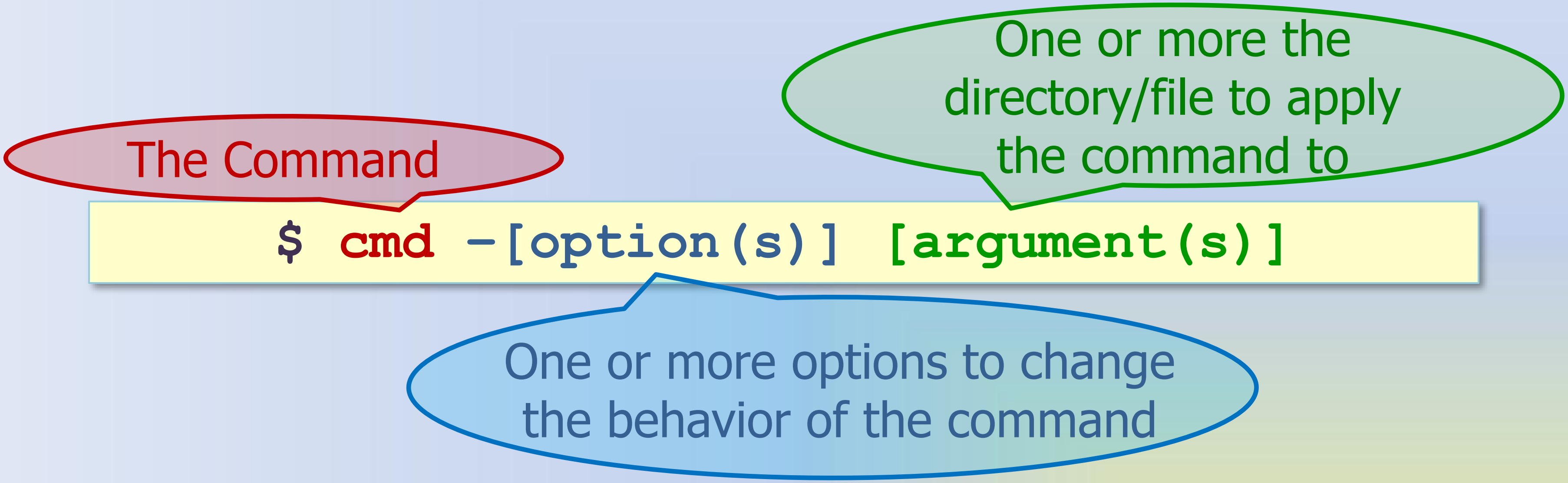
- An UNIX Terminal window will then appear with a **$** prompt, waiting for you to start entering commands.
- Unix Terminal is like Windows DOS



```
knoppix@Knoppix:~$ su
root@Knoppix:/ramdisk/home/knoppix# 
```

General Linux Command Format

- A little like DOS commands on windows with some differences



The Command

One or more the directory/file to apply the command to

```
$ cmd -[option(s)] [argument(s)]
```

One or more options to change the behavior of the command

- Notes:
  - Parts between [] packets are optional
  - Linux is CASE SENSITIVE

Getting Help

- In Linux, there are on-line manuals which gives information about most commands.
- `man` is used to read the manual page for a particular command one page at a time:

```
$ man cmd
```

- **Examples,**

```
$ man ls
```

→ Displays the manual pages of the command **ls**

```
$ man man
```

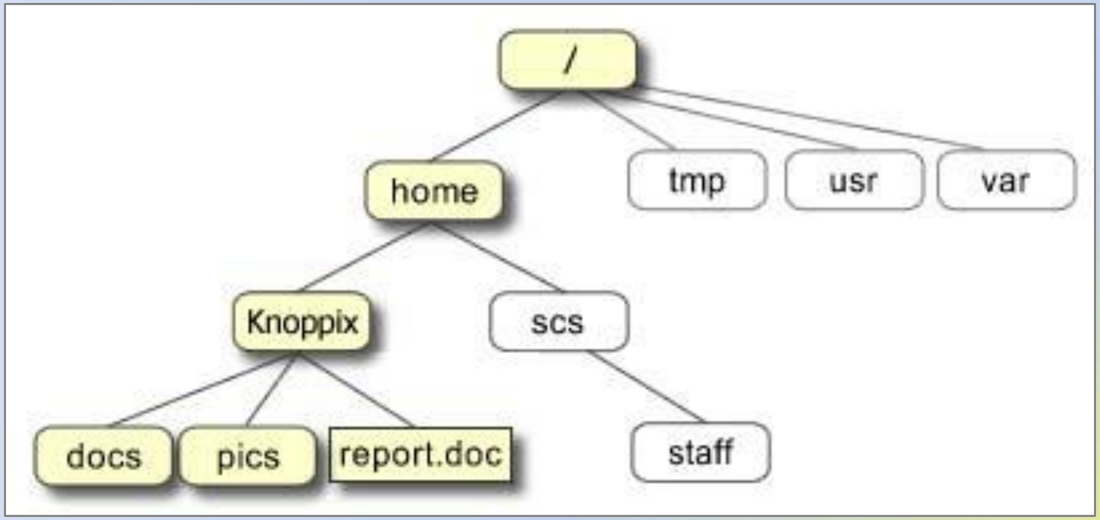
→ Displays the manual pages of the command **man**

- Use the following keys to go through the manual
  - Enter → one line forward
  - F → Forward one window OR
  - B → Backward one window OR screen
  - Q → Quits the manual

DIRECTORY COMMANDS

What is a Directory?

- In Linux, all the files are grouped together in the directory structure.
- The file-system is arranged in a hierarchical structure, like an inverted tree.
- The top of the hierarchy is called **root** (written as a slash **/**)



- In the diagram above, the full path to the file **report.doc** is: **/home/knoppix/report.doc**

Pathnames

- **pwd** (print working directory) is used to prints the current directory, type:

```
$ pwd
```

The full pathname will look something like this: /home/rawan

Making and Removing Directories

- **`mkdir`** and **`rmdir`** are used for making and removing directories.

```
$ mkdir dirname
```

Creates a new directory with name ***dirname*** in the current directory

```
$ rmdir dirname
```

Deletes the directory ***dirname*** from the current directory

  - **Note:** A directory must not contain any files when it is deleted, otherwise an error message is displayed.

- Examples:

```
$ mkdir dir1
```

Creates a new directory called **dir1**

```
$ rmdir dir3
```

Removes the directory **dir3** (if it exists)

Changing to a Different Directory

- `cd` (Change Directory) is used to change the working directory.

```
$ cd dirpath
```

Changes the current directory to the relative or absolute pathname of the directory ***dirpath***.

```
$ cd
```

If no directory is given, the command changes the current directory to the home directory.

```
$ cd ..
```

Changes to the parent directory.

- Examples:

```
$ cd
$ cd dir1
$ cd dir2
$ cd ..
$ cd dir2
$ cd /home/knoppix/dir1
```

Change to home-directory

Change to directory **dir1**

Error because **dir2** is not in **dir1**

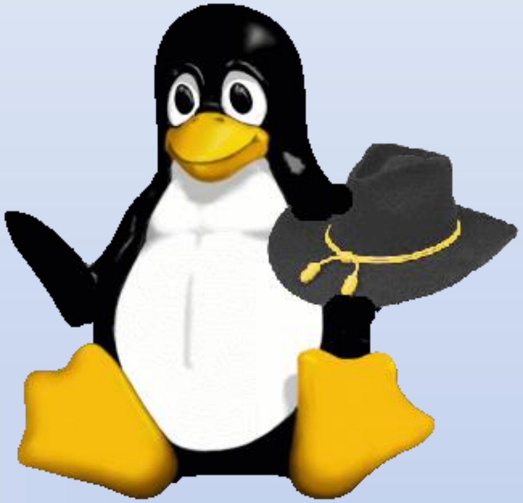
Change to parent directory **dir1**

Change to directory **dir2**

Change to directory **dir1**

Directory Commands Summary

Command	Meaning
`pwd`	display the path of the current directory
`mkdir` *dirname*	make a directory
`rmdir` *dirname*	remove a directory
`cd` *directory*	change to named directory
`cd`	change to home-directory
`cd ..`	change to parent directory



FILE COMMANDS

What is a file?

- A file is a collection of data.
- They are created by users using text editors, running compilers etc.
- Examples of files:
  - a document (report, essay etc.)
  - the text of a program written in some high-level programming language (like C or C++)

Listing files and directories

- **ls** (list) is used to list information about files and directories.

```
$ ls dirpath
```

If the command has a directory name as argument (i.e., dirpath), then the command lists the files in that directory.

```
$ ls
```

If no directory is given, then the command lists the files in the current directory.

```
$ ls -l
```

Includes extensive information on each file.

- Note: The **ls** command has several options. The most important is **ls -l**, which includes extensive information on each file, including, the access permissions, owner, file size, and the time when the file was last modified.

Moving and renaming Files

- **mv** is used to rename or move a file or a directory.

```
$ mv fname newfile
```

→ The file or directory ***fname*** is renamed as ***newfile***. If the destination file (***newfile***) exists, then the content of the file is overwritten, and the old content of ***newfile*** is lost.

```
$ mv fname dirname
```

→ If the first argument is a file name and the second argument is a directory name (*dirname*), the file is moved to the specified directory.

- Examples:

```
$ mv dir2 dir5
$ mv dir5 dir1
$ mv file2 dir1
```

→ Renames **dir2** to **dir5**

→ Moves **dir5** to **dir1**

→ Moves **file1.txt** to **dir1**

Copying and Removing Files

- **cp** (copy) and **rm** (remove) are used to copy and remove files:

```
$ cp fname newfile
```

Copies the content of file *fname* to *newfile*. If a file with name *newfile* exists the content of that file is overwritten.

```
$ cp fname dirname
```

If the second argument is a directory, then a copy of *fname* is created in directory *dirname.*

```
$ rm fname
```

Removes the file *fname* from the current directory

- Examples:

```
$ cp file1 dir1
$ cd dir1
$ cp file1 file2
$ rm file1
```

Copy **file1** to **dir1**

Copy to **file1** to **file2** overwriting its content

Removes **file1**

View and Modify Text Files

- `more` and `cat` are used to view and modify text files.

```
$ more fname
```

Displays the contents of file ***fname***, one page at a time.

```
$ cat fname
```

Similar to the more command, but the file is displayed without stopping at the end of each page

- Examples:

```
$ more file1
$ cat file1
```

Displays the contents of **file1**

Displays the contents of **file1**

File Commands Summary

Command	Meaning
`ls`	list files and directories in the current directory
`ls` *dirpath*	List files and directories in *dirpath*
`ls -l`	Includes extensive information on each file
`mv` *file1 file2*	rename *file1* to *file2*
`mv` *file1 dirpath*	move *file1* to *dirpath*
`cp` *file1 file2*	copy *file1* and call it *file2*
`cp` *file1 dirpath*	copy *file1* to *dirpath*
`rm` *file*	remove a file
`more` *file*	display a file
`cat` *file*	display a file

Redirecting Programs Output

- **>** and **>>** are used to redirect program output

```
$ cmd > fname
```

The output of ***cmd*** is written to file ***fname***. The file is created if it doesn’t already exist, and the contents is overwritten if the file exists.

```
$ cmd >> fname
```

Appends the output of command ***cmd*** to the end of file ***fname***.

- Examples:

```
$ ls > mylist
```

Writes a listing of the current directory in file ***mylist***

```
$ ls >> mylist
```

Appends a listing of the current directory to file ***mylist***

PROCESSES AND JOBS COMMANDS

Foreground and Background Processes

- A *process* is an executing program identified by a unique PID (process identifier).
- In Linux, each terminal window can run multiple commands at the same time.
- It is possible to stop a command temporarily and resume it at a later time.
- In each terminal window, one command can be run as a *foreground* process and multiple command can be run as *background* processes.

Processes and Jobs Commands

Command	Meaning
Ctrl+C	Terminates the command running in the foreground
Ctrl+Z	Stops (suspend) the commands in the foreground.
cmd&	Executes the command *cmd* in the background
bg	background the suspended job
jobs	Lists all background and stopped commands of the current user, and assigns a number to each command.
fg %n	Resume suspended job number **n** in the foreground, and make it the current job. The numbers are as displayed by the jobs command.
bg %n	Resumes suspended job number n in the background, as if it had been started with &.
ps -all	Lists all current processes and their assigned ID (pid)
kill *pid*	Terminates the process with the specified ID: ***pid***, where ***pid*** is as displayed by the command ***ps***

Exercise

- List all the content of the home directory then remove any subdirectory in it
- Go to the home directory then make 3 new subdirectory called (pics, docs, backup)
- Make a subdirectory in (pics), call it (babies)
- Rename the (backup) directory to (bup) then move it to the (docs) directory
- Write a listing of the current directory in a file called (*list_a*)
- Copy the file (*list_a*) to the (docs) directory
- Make a copy of (*list_a*) and call it (*list_b*) then move (*list_b*) to (bup) directory
- Run the command that displays the manual of the (passwd) command in the background
- Terminate all the background process

??? ANY QUESTIONS ???

☺

بسم الله الرحمن الرحيم

Compiling java Programs + Compiling c++ program in Linux

Lab 02

Lab Objective

- To practice writing and compiling java programs in Linux
- Ton Learn how to Compile c++ program

Compiling Java

- Three things are necessary for creating java programs:
- a *text editor,*
- a compiler
- a *java standard library* if you use Java

A text editor

- A text editor is all that is needed to create the *source code* for a program in java or in any other language.
- A text editor is a program for writing and editing plain text.
- It differs from a word processor in that it does not manage document formatting (e.g., typefaces, fonts, margins and italics) or other features commonly used in desktop publishing.

A text editor

- java programs can be written using any of the many text editors that are available for Linux, such as *vi*, *gedit*, *kedit* or *emacs*.
- At least one text editor is built into every Unix-like operating system, and most such systems contain several.

A text editor

- To see if a specific text editor exists on the system, all that is necessary is to type its name on the *command line* (i.e., the all-text user interface) and then press the ENTER key.
- If it exists, the editor will appear in the existing window if it is a command line editor, such as vi.

A text editor

- It will open in a new window if it is a GUI (graphical user interface) editor such as gedit.
- For example, to see if vi is on the system (it or some variation of it almost always is), all that is necessary is to type the following command and press the ENTER key: vi

A compiler

- A compiler is a specialized program that converts source code into *machine language* (also called *object code* or *machine code*) so that it can be understood directly by a CPU (central processing unit).
- An excellent java compiler is included in the *Java Compiler* (javac), one of the most important components of most modern Linux distributions.

A compiler

- GNU is an on-going project by the Free Software Foundation (FSF) to create a complete, Unix-compatible, high performance and freely distributable computing environment.
- All that is necessary to see if the javac is already installed and ready to use is to type the following command and press the ENTER key: javac

java library

- A library is a collection of subprograms that any programmer can employ to reduce the amount of complex and repetitive source code that has to be written for individual programs.
- Every Unix-like operating system requires a C library.

Practice …

- Write the following program using any text editor and save it in a file called **`HelloWorld.java`**

```
public class HelloWorld {
   public static void main(String[]
args) {
      System.out.println("Hello,
World");
   }
}
```

Note: The text file name should be the same as the class name.

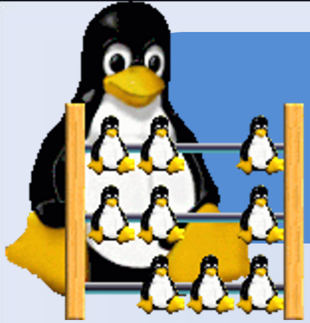
… Practice

- The standard way to compile this program is with the following command:

```
$ javac HelloWorld.java
```

- This command compiles **`HelloWorld.java`** into an executable program called **`HelloWorld.class`** that you run by typing the following at the command line:

```
$ java HelloWorld
```



Exercise

1) Execute the previous program
2) Write and compile another program, name it `forloop.java` that only has an infinite loop like the following:

```
for(;;);
```

3) Execute the `loop` program in the ***background***.
4) List all current processes and their assigned ID (PID). Write down the PID of the `loop` program.
5) Kill the `loop` program.

Solution:

```
public class forloop
{ public static void main(String[] args)
    { for (;;)
        { System.out.println("hello world"); } }}
```

....

```
$ javac forloop.java
$ java forloop
$ ps –all
$ kill 7351
```

Compiling C and C++

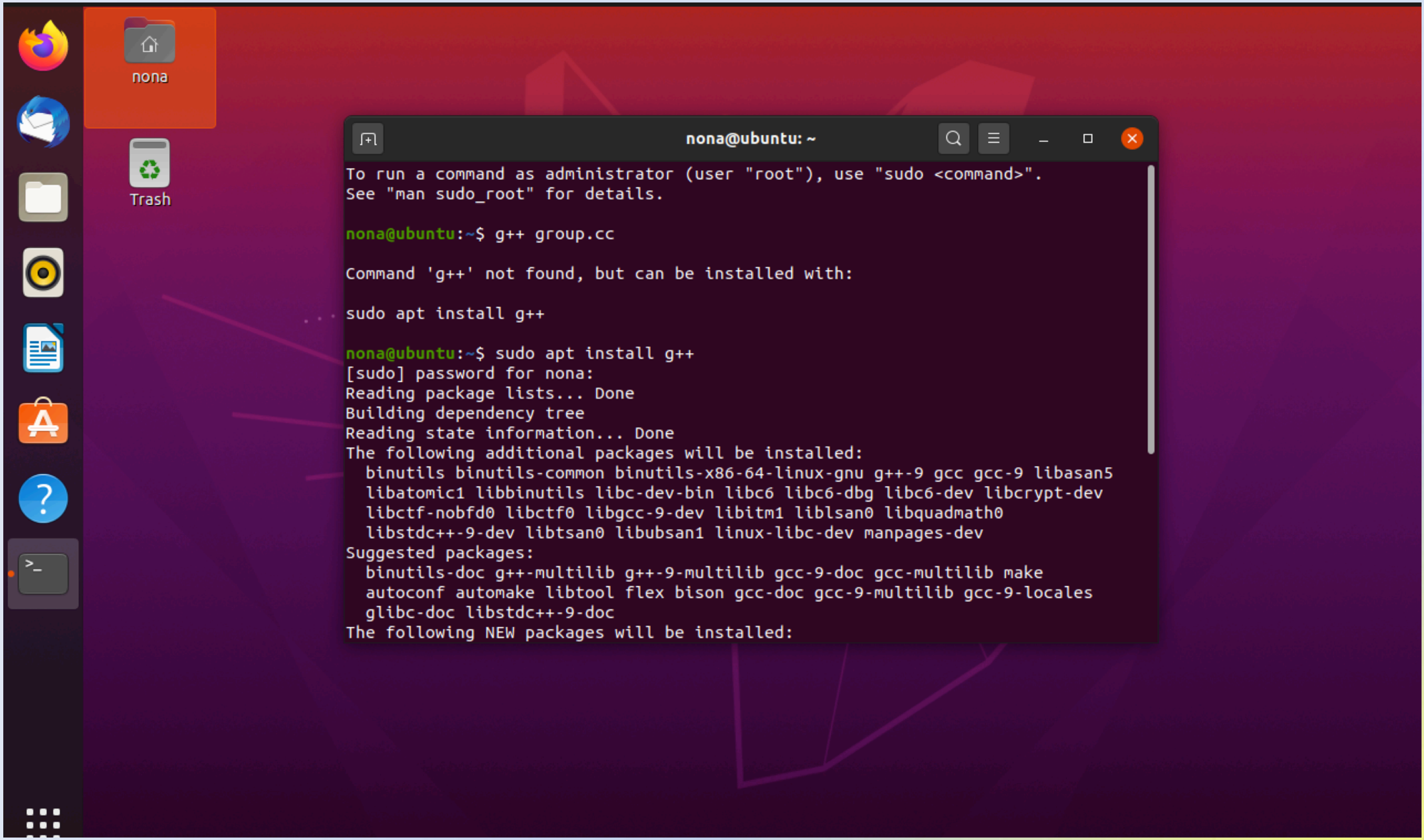
- **C++** programs are saved with extensions **.cc** whereas **c** program saved with extensions **.c**
- If you are using g++ compiler:

```
g++ program.cc
```

To execute and see the output of program: (Run)

```
./a.out
```

You have to install g++ Compiler



```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

nona@ubuntu:~$ g++ group.cc

Command 'g++' not found, but can be installed with:

sudo apt install g++

nona@ubuntu:~$ sudo apt install g++
[sudo] password for nona:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-common binutils-x86-64-linux-gnu g++-9 gcc gcc-9 libasan5
  libatomic1 libbinutils libc-dev-bin libc6 libc6-dbg libc6-dev libcrypt-dev
  libctf-nobfd0 libctf0 libgcc-9-dev libitm1 liblsan0 libquadmath0
  libstdc++-9-dev libtsan0 libubsan1 linux-libc-dev manpages-dev
Suggested packages:
  binutils-doc g++-multilib g++-9-multilib gcc-9-doc gcc-multilib make
  autoconf automake libtool flex bison gcc-doc gcc-9-multilib gcc-9-locales
  glibc-doc libstdc++-9-doc
The following NEW packages will be installed:
```

You have to install g++ Compiler

- So, you must write the command to enable installing g++ compiler.

Important commands

❖ Ps command

- Ps stands for "Process Status", it is used to display the currently running processes on Unix/Linux systems.

```
ps ux
```

❖ Kill command

- If you want to terminate any process you would look up the process idenifier (PID).

```
kill PID
```

Important commands

❖ **Script utility**

- Records everything printed on your screen. The record is recorded to the filename.

```
script filename
```

??? ANY QUESTIONS ???

☺

بسم الله الرحمن الرحيم

Processes

Lab 03

Lab Objective

- To practice creating child process using `fork()`.

The `fork` Function

- In computing, when a process forks, it creates a copy of itself, which is called a "***child process***." The original process is then called the "***parent process***“ .
- The `fork()` function is used from a “*parent*” process to create a **duplicate** process, the “*child*”.
- The parent and the child processes can tell each other apart by examining the return value of the `fork()` system call

The `fork` Function

```
pid_t fork(void);
```

- If successful, the `fork` function returns twice:

Parent

returns PID of the newly-created child process

Child

returns 0

- On failure, the `fork` function returns once:

Parent

returns -1

Parent and Child

- A child inherits its parent's permissions, working-directory, root-directory, open files, etc.
- All descriptors that were open in the parent before the call to `fork` are shared with the child after the `fork` returns.

More Info

- The child process inherits the following attributes from the parent process:
  - Real and effective user and group IDs
  - Environment settings
  - Signal handling settings
  - Attached shared memory segments
  - Memory mapped segments
  - Process group ID
  - Current working directory
  - File mode creation mask
  - Controlling terminal
  - nice value

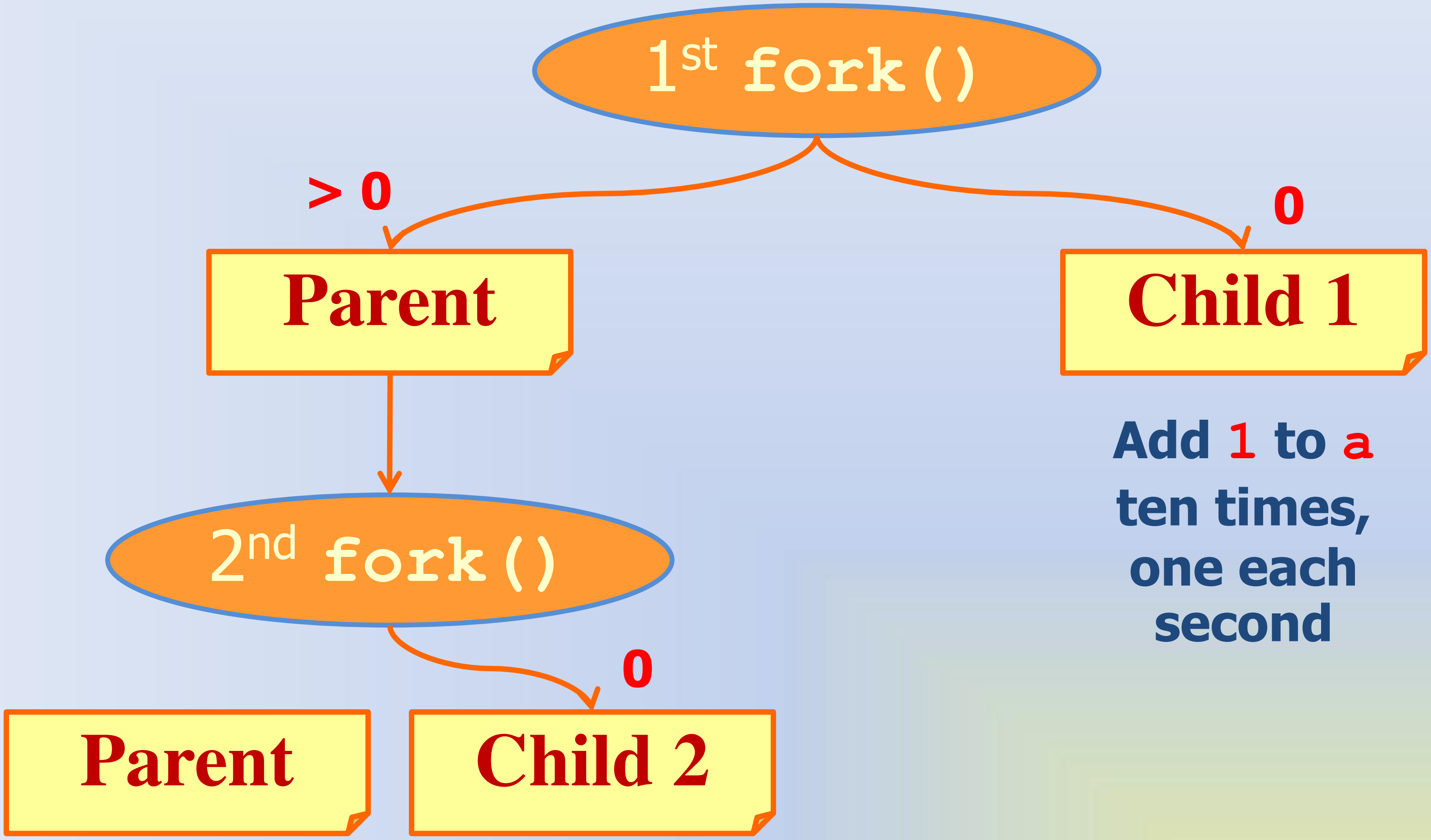
More Info

- The child process differs from the parent process in the following ways:
  - The child process has a unique process ID, which also does not match any active process group ID.
  - The child process has a different parent process ID (that is, the process ID of the process that called fork()).
  - The child process has its own copy of the parent's file descriptors Each of the child's file descriptors refers to the same open file structures as the file descriptor of the parent.
  - The child process has its own copy of the parent's open directory streams.
  - The child process' process execution times (as returned by times()) are set to zero.
  - Pending alarms are cleared for the child.
  - All semaphore adjustment values are cleared.
  - File locks set by the parent process are not inherited by the child process.
  - The set of signals pending for the child process is cleared.
  - Interval timers are reset.
  - The new process has a single thread. If a multi-threaded process calls fork(), the new process contains a replica of the calling thread and its entire address space, including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal safe operations until such time as one of the exec() functions is called. Fork handlers may be established using the pthread_atfork() function to maintain application invariants across fork() calls.

Practice

Ex1:

- In the following C++ program, the main process forks two children.
- Every child repeats adding the value 1 to the variable “a” ten times.
- Write, compile and run the program in Linux.



1st fork()

> 0

Parent

0

Child 1

2nd fork()

Parent

0

Child 2

Add 1 to a ten times, one each second

Wait for child termination

Add 1 to a ten times, one each second

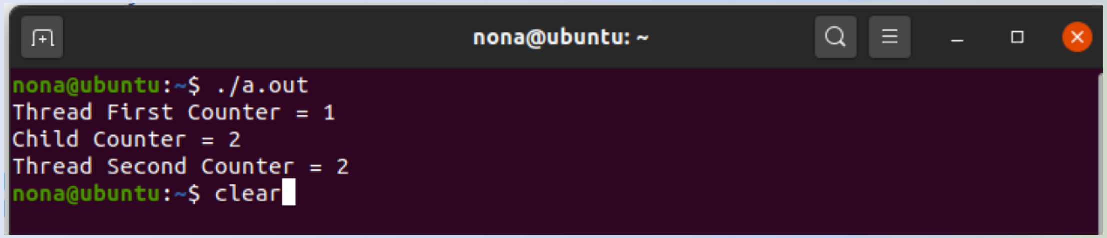
Notes

- **wait()** System Call

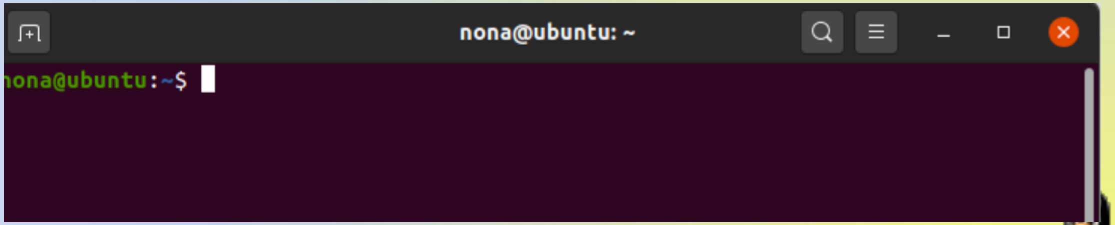
This function blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process.

- The **main()** should be declared as int , because when you declare it as **void**, it causes an error.

- **clear** command uses to Clear Linux Terminal.



```
nona@ubuntu:~$ ./a.out
Thread First Counter = 1
Child Counter = 2
Thread Second Counter = 2
nona@ubuntu:~$ clear
```



```
nona@ubuntu:~$
```

```
#include <iostream>
#include <stdlib.h>     /* exit() */
#include <unistd.h>     /* fork() */
#include <sys/types.h>  /* pid_t */
#include <sys/wait.h>   /* wait() */
using std::cout; // it is a predefined variable which
allow to send data to the console to be printed as
text. It stands for "character output"

int main()
{
        pid_t pid1, pid2, cpid;
        int i, j, a, status;
        a = 0;

        pid1 = fork();  //fork child 1 process
        if (pid1 < 0)   //error occurred
        {
                cout<< "First Fork Failed\n";
                exit(-1);
        }/end if
        else if (pid1 == 0)     //child 1 process
        {  for (i=0; i<10; i++)
                {  a++;
                        cout<< "Child1: a =
"<<a<<"\n";
                        sleep(1);
                }//end for
        }//end else if
```

The main process forks child 1

Error

When `fork()` returns a negative number, an error happened

Child 1

When `fork()` returns 0, we are in the child 1 process

Here child 1 Add 1 to 'a' ten times

a = 10

```
    else     //parent process
    {
        pid2 = fork();   //fork child 2 process
        if (pid2 < 0)    //error occurred
        {
        cout<< “Second Fork Failed\n";
        exit(-1);
        }//end if
        else if (pid2 == 0) //child 2 process
        {
            for (j=0; j<10; j++)
            {
                a++;
                cout<< "Child2: a = "
                  <<a<<"\n";
                sleep(1);
            }//end for
        }//end else if
        else     //parent process
        {
            cpid = wait(&status);
            cout<< "\n*****Parent is
Closing*****\n";
            exit(0);
        }
    }//end else
}//end main
```

Parent

The parent forks another child

Error

When `fork()` returns a negative number, an error happened

Child 2

When `fork()` returns 0, we are in the child 2 process

Here child 2 Add 1 to ‘a’ ten times

a = 10

Parent

When `fork()` returns a positive number, we are in the parent process

The parent waits for children termination

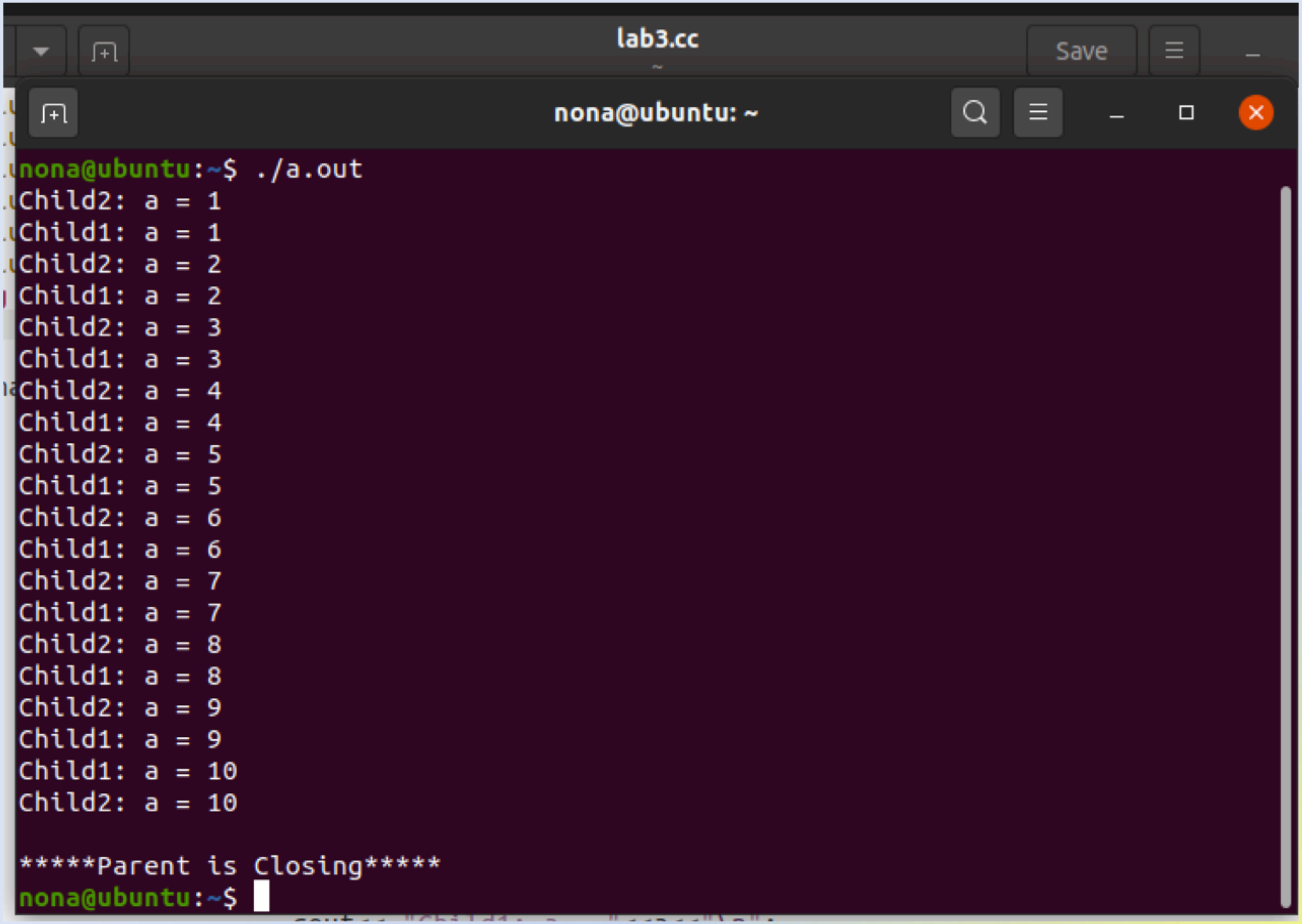
on Ubuntu

```
#include <iostream>
#include <stdlib.h>          /* exit() */
#include <unistd.h>          /* fork() */
#include <sys/types.h>       /* pid_t */
#include <sys/wait.h>        /* wait() */
#include <stdio.h>
using std::cout;
int main()
{
        pid_t pid1, pid2, cpid;
        int i, j, a, status;
        a = 0;

        pid1 = fork();       //fork child 1 process
        if (pid1 < 0)        //error occurred
        {
                cout<<"First Fork Failed\n";
                exit(-1);
        }//end if
```

```
else if (pid1 == 0) //child 1 process
{
        for (i=0; i<10; i++)
        {
                a++;
                cout<< "Child1: a = "<<a<<"\n";
                sleep(1);
        }//end for
}//end else if
else    //parent process
{
        pid2 = fork();          //fork child 2 process
        if (pid2 < 0)           //error occurred
        {
        cout<< "Second Fork Failed\n";
        exit(-1);
        }//end if
```

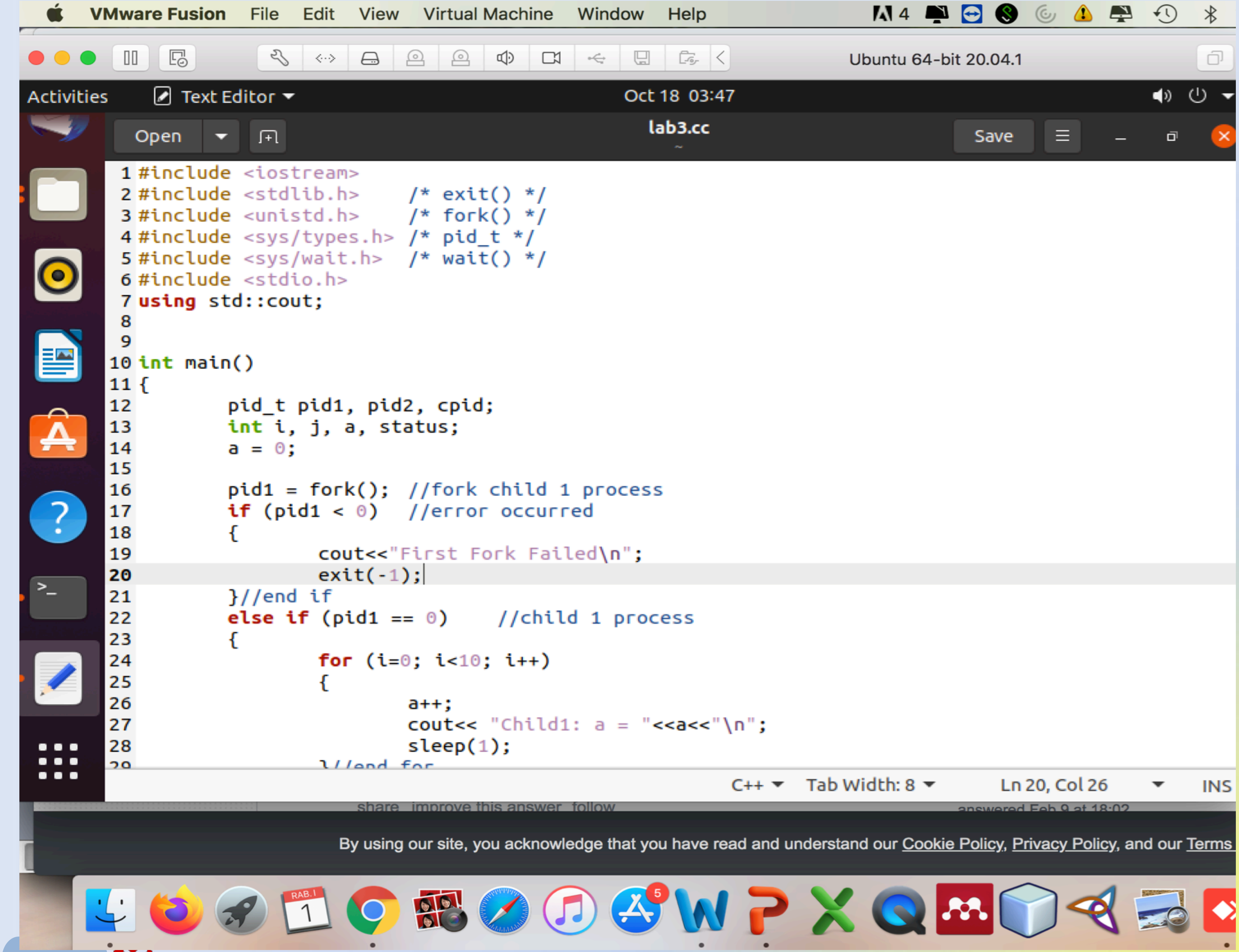
```
                        else if (pid2 == 0) //child 2 process
                {
                        for (j=0; j<10; j++)
                        {  a++;
                                cout<< "Child2: a = "
                                  <<a<<"\n";
                                sleep(1);
                        }//end for
                }//end else if
                else    //parent process
                {
                        cpid = wait(&status);
                        cout<< "\n*****Parent is Closing*****\n";
                        exit(0);
                }
        }//end else

}//end main
```

Output



```
nona@ubuntu:~$ ./a.out
Child2: a = 1
Child1: a = 1
Child2: a = 2
Child1: a = 2
Child2: a = 3
Child1: a = 3
Child2: a = 4
Child1: a = 4
Child2: a = 5
Child1: a = 5
Child2: a = 6
Child1: a = 6
Child2: a = 7
Child1: a = 7
Child2: a = 8
Child1: a = 8
Child2: a = 9
Child1: a = 9
Child1: a = 10
Child2: a = 10

*****Parent is Closing*****
nona@ubuntu:~$ 
```

Output



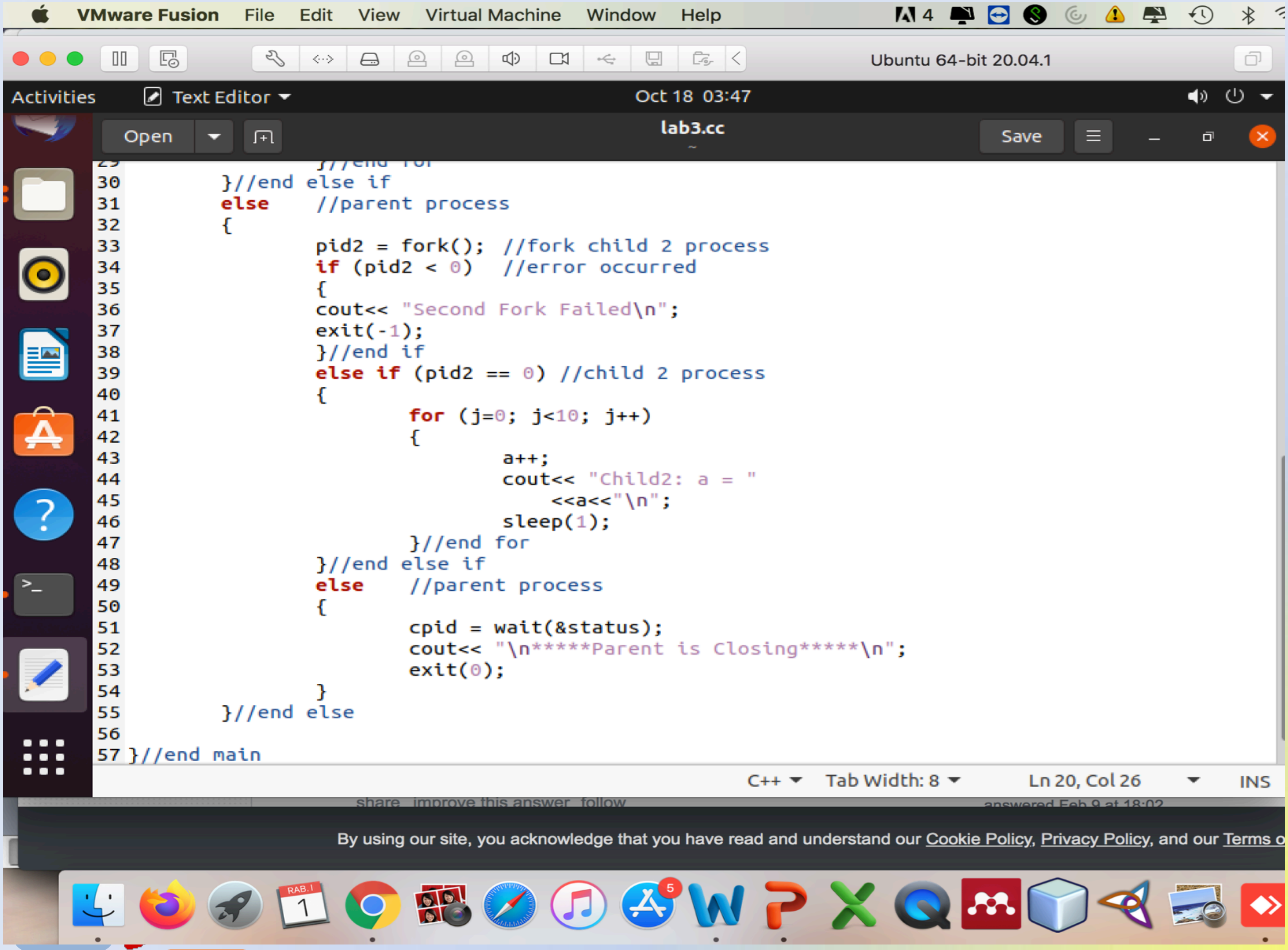
```
#include <iostream>
#include <stdlib.h>     /* exit() */
#include <unistd.h>     /* fork() */
#include <sys/types.h>  /* pid_t */
#include <sys/wait.h>   /* wait() */
#include <stdio.h>
using std::cout;


int main()
{
        pid_t pid1, pid2, cpid;
        int i, j, a, status;
        a = 0;

        pid1 = fork();  //fork child 1 process
        if (pid1 < 0)   //error occurred
        {
                cout<<"First Fork Failed\n";
                exit(-1);
        }//end if
        else if (pid1 == 0)     //child 1 process
        {
                for (i=0; i<10; i++)
                {
                        a++;
                        cout<< "Child1: a = "<<a<<"\n";
                        sleep(1);
                }//end for
```

share improve this answer follow

answered Feb 9 at 18:02

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms

Output



```
                }//end for
        }//end else if
        else    //parent process
        {
                pid2 = fork();  //fork child 2 process
                if (pid2 < 0)   //error occurred
                {
                cout<< "Second Fork Failed\n";
                exit(-1);
                }//end if
                else if (pid2 == 0) //child 2 process
                {
                        for (j=0; j<10; j++)
                        {
                                a++;
                                cout<< "Child2: a = "
                                   <<a<<"\n";
                                sleep(1);
                        }//end for
                }//end else if
                else    //parent process
                {
                        cpid = wait(&status);
                        cout<< "\n*****Parent is Closing*****\n";
                        exit(0);
                }
        }//end else

}//end main
```

share improve this answer follow

answered Feb 9 at 18:02

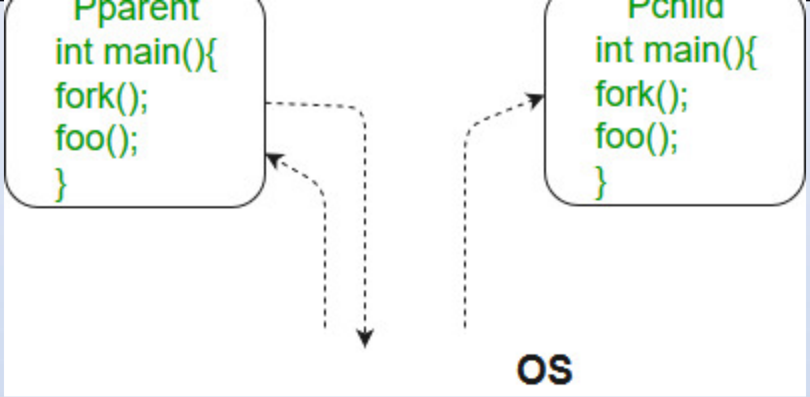
By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms o

fork() in C

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process **uses the same pc(program counter), same CPU registers**, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.



Pparent

```
int main(){
fork();
foo();
}
```

Pchild

```
int main(){
fork();
foo();
}
```

OS

Ex2:

```
//predict the out put of following programe
#include <stdio.h> // For dealling with Inuput and Output
#include <sys/types.h> //This library defines different data types such as pid_t (used for process IDs)
#include <unistd.h> // Enabling POSIX API
int main()
{
    // make two process (Parent & Child)  which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Ex3:

Zombie and Orphan Processes in C

Zombie Process:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

In the following code, the child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn’t call wait() and the child process’s entry still exists in the process table.

```
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

Ex4: Orphan Process:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process

```
// A C program to demonstrate Orphan Process.
// Parent process finishes execution while the
// child process is running. The child process
// becomes orphan.
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // Create a child process
    int pid = fork();

    if (pid > 0)
        printf("in parent process");

    // Note that pid is 0 in child process
    // and negative if fork() fails
else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }

    return 0;
}
```

Ex5:

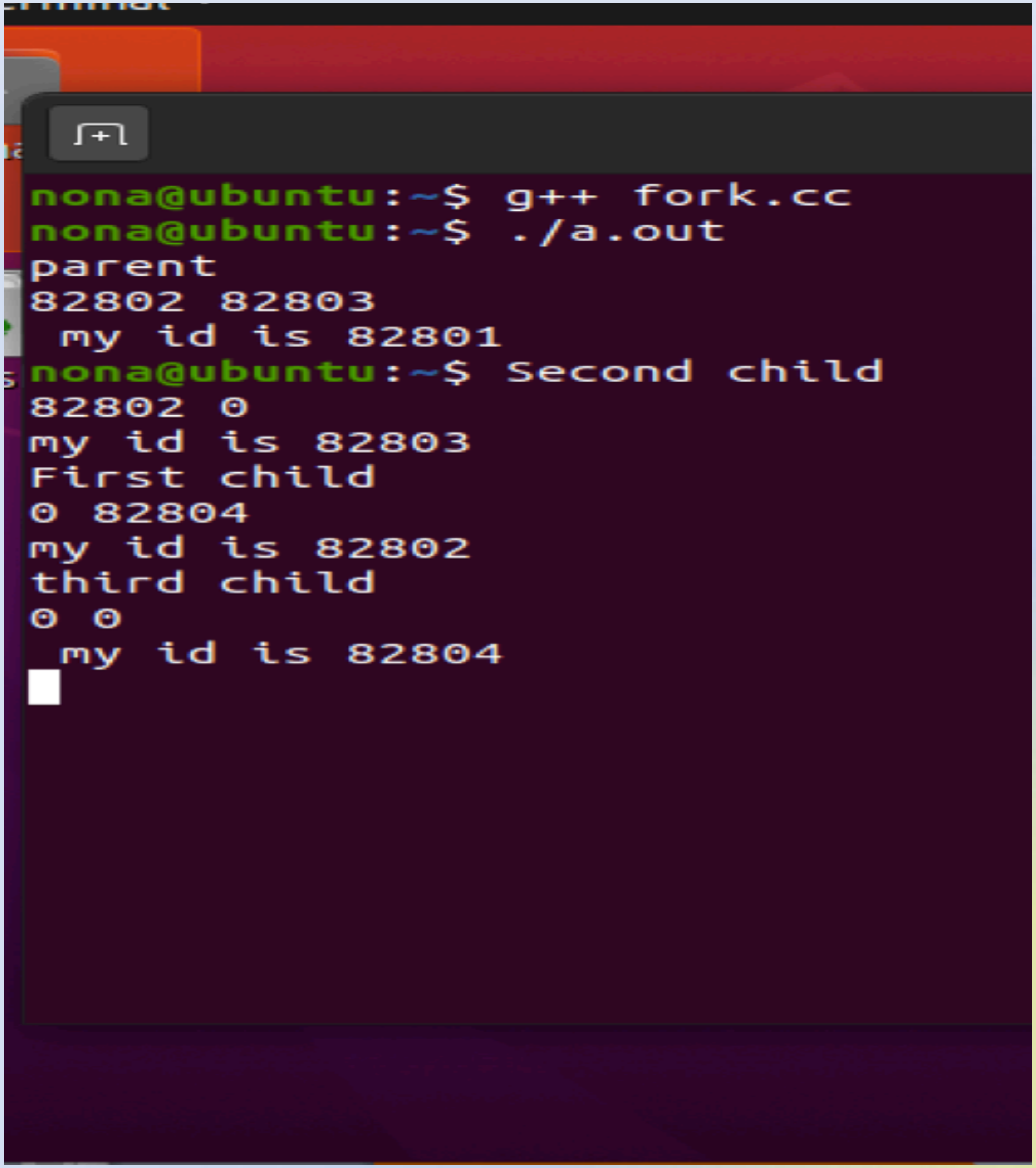
/ C++ program to demonstrate creating process (**three children)** using fork()

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    // Creating first child
    int n1 = fork();
      // Creating second child. First child
    // also executes this line and creates
    // grandchild.
    int n2 = fork();
      if (n1 > 0 && n2 > 0) {
        printf("parent\n");
        printf("%d %d \n", n1, n2);
        printf(" my id is %d \n", getpid());
    }
    else if (n1 == 0 && n2 > 0)
    {
        printf("First child\n");
        printf("%d %d \n", n1, n2);
        printf("my id is %d  \n", getpid());
    }
    else if (n1 > 0 && n2 == 0)
  {              printf("Second child\n");
                printf("%d %d \n", n1, n2);
                 printf("my id is %d  \n", getpid());  }
else {  printf("third child\n");// gradechild
        printf("%d %d \n", n1, n2);
        printf(" my id is %d \n", getpid());
    }
    return 0;
}
```

Output- 1

```
parent
28808 28809
 my id is 28807
First child
0 28810
my id is 28808
Second child
28808 0
my id is 28809
third child
0 0
my id is 28810
```

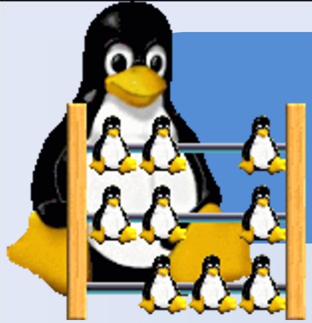
Output- 2



```
nona@ubuntu:~$ g++ fork.cc
nona@ubuntu:~$ ./a.out
parent
82802 82803
 my id is 82801
nona@ubuntu:~$ Second child
82802 0
my id is 82803
First child
0 82804
my id is 82802
third child
0 0
 my id is 82804
```

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates.



Check Off on Ex1

1) Why the final value of `a` is 10 and not 20?

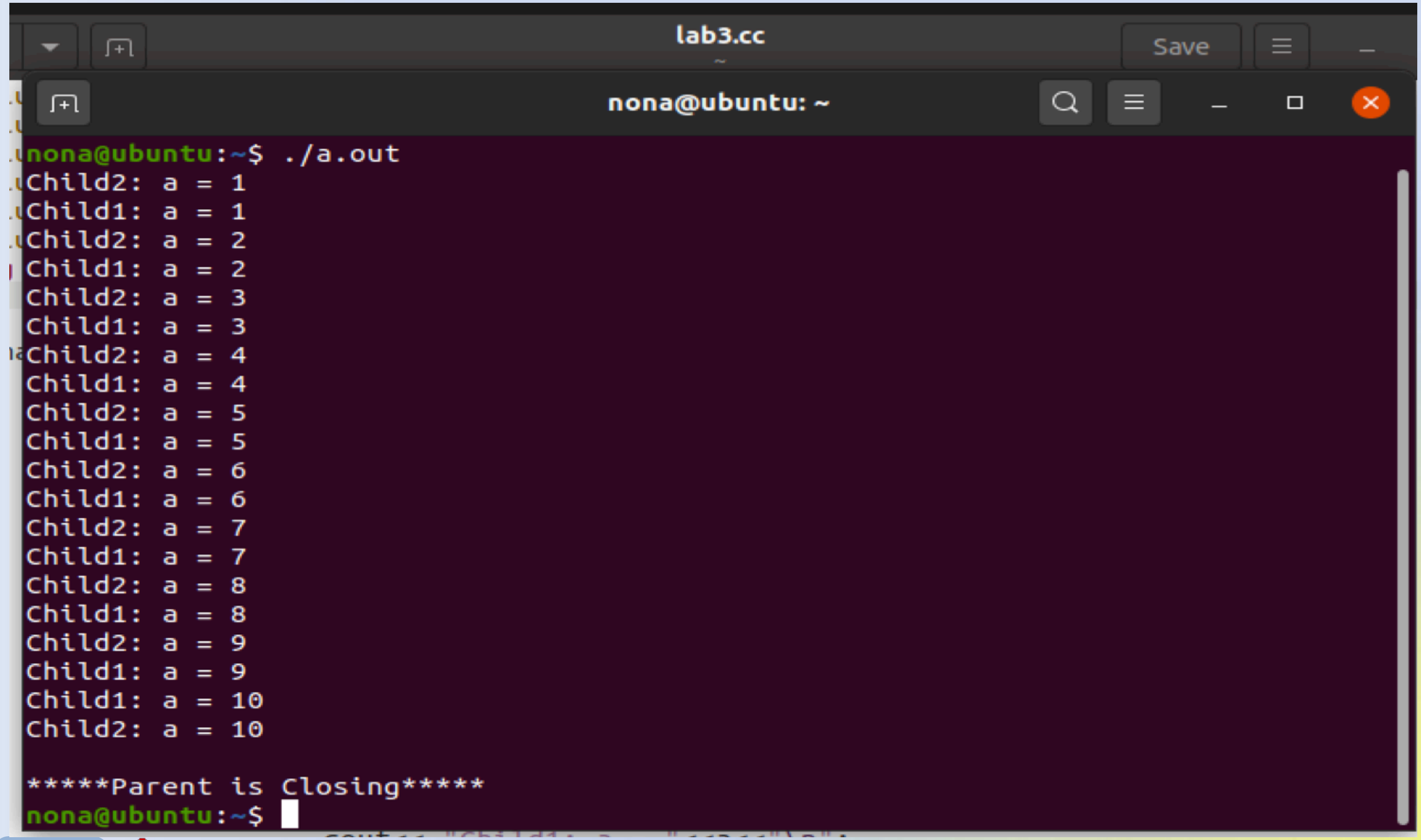
2) Use the command `ps -all` in a separate window while the above program is running. Write down the `PID` of the processes related to the program.

3) Kill child 1 and then child 2 while the program is running. Briefly explain what will happen.

4) Kill the main process while the program is running. Briefly explain what will happen.

Solution

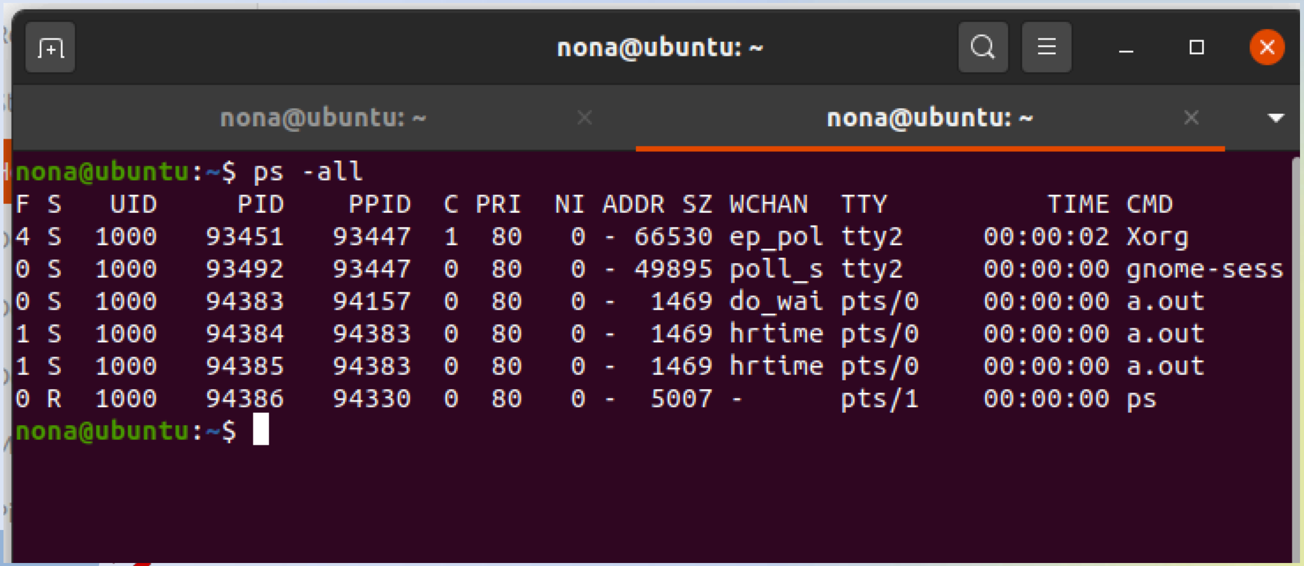
1) a is 10 because each process has its own variable.



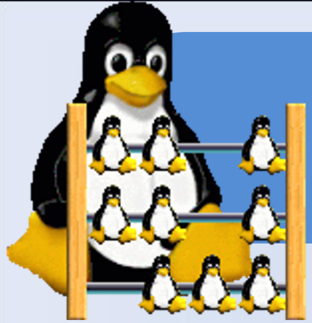
```
nona@ubuntu:~$ ./a.out
Child2: a = 1
Child1: a = 1
Child2: a = 2
Child1: a = 2
Child2: a = 3
Child1: a = 3
Child2: a = 4
Child1: a = 4
Child2: a = 5
Child1: a = 5
Child2: a = 6
Child1: a = 6
Child2: a = 7
Child1: a = 7
Child2: a = 8
Child1: a = 8
Child2: a = 9
Child1: a = 9
Child1: a = 10
Child2: a = 10

*****Parent is Closing*****
nona@ubuntu:~$ 
```

Solution

2) You have to write the command ps –all in a separate window while the program is running (so you will have 2 separates windows), the PID of the processes will be displayed. [change sleep(5);]



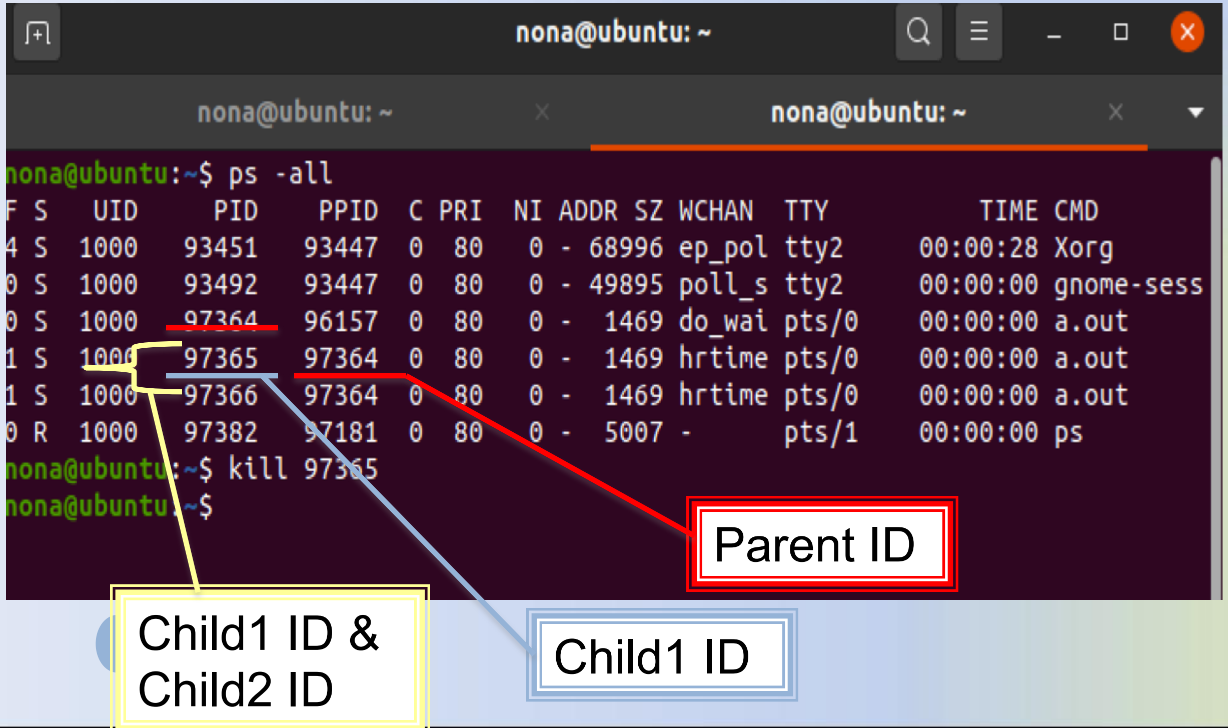
```
nona@ubuntu:~$ ps -all
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000   93451   93447  1  80   0 - 66530 ep_pol tty2     00:00:02 Xorg
0 S  1000   93492   93447  0  80   0 - 49895 poll_s tty2     00:00:00 gnome-sess
0 S  1000   94383   94157  0  80   0 -  1469 do_wai pts/0    00:00:00 a.out
1 S  1000   94384   94383  0  80   0 -  1469 hrtime pts/0    00:00:00 a.out
1 S  1000   94385   94383  0  80   0 -  1469 hrtime pts/0    00:00:00 a.out
0 R  1000   94386   94330  0  80   0 -  5007 -      pts/1    00:00:00 ps
nona@ubuntu:~$ 
```



Solution

3) To kill the child, you have to write the command ---> kill PID of child, e.g. kill 1234 (when you kill child 1 and child 2 they will be terminated)

In the following we try to kill child1:



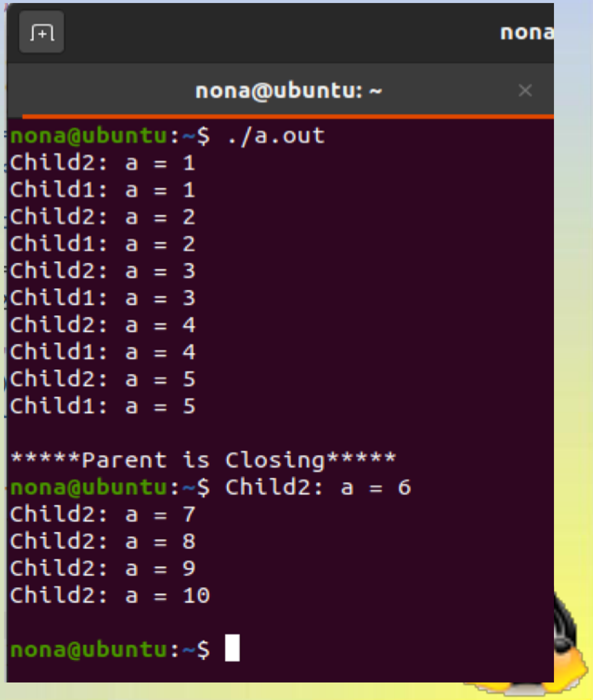
```
nona@ubuntu:~$ ps -all
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000   93451   93447  0  80   0 - 68996 ep_pol tty2     00:00:28 Xorg
0 S  1000   93492   93447  0  80   0 - 49895 poll_s tty2     00:00:00 gnome-sess
0 S  1000   97364   96157  0  80   0 -  1469 do_wai pts/0    00:00:00 a.out
1 S  1000   97365   97364  0  80   0 -  1469 hrtime pts/0    00:00:00 a.out
1 S  1000   97366   97364  0  80   0 -  1469 hrtime pts/0    00:00:00 a.out
0 R  1000   97382   97181  0  80   0 -  5007 -      pts/1    00:00:00 ps
nona@ubuntu:~$ kill 97365
nona@ubuntu:~$
```

Parent ID

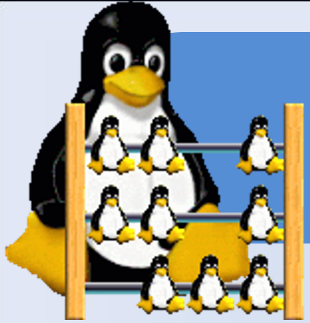
Child1 ID & Child2 ID

Child1 ID

Result:

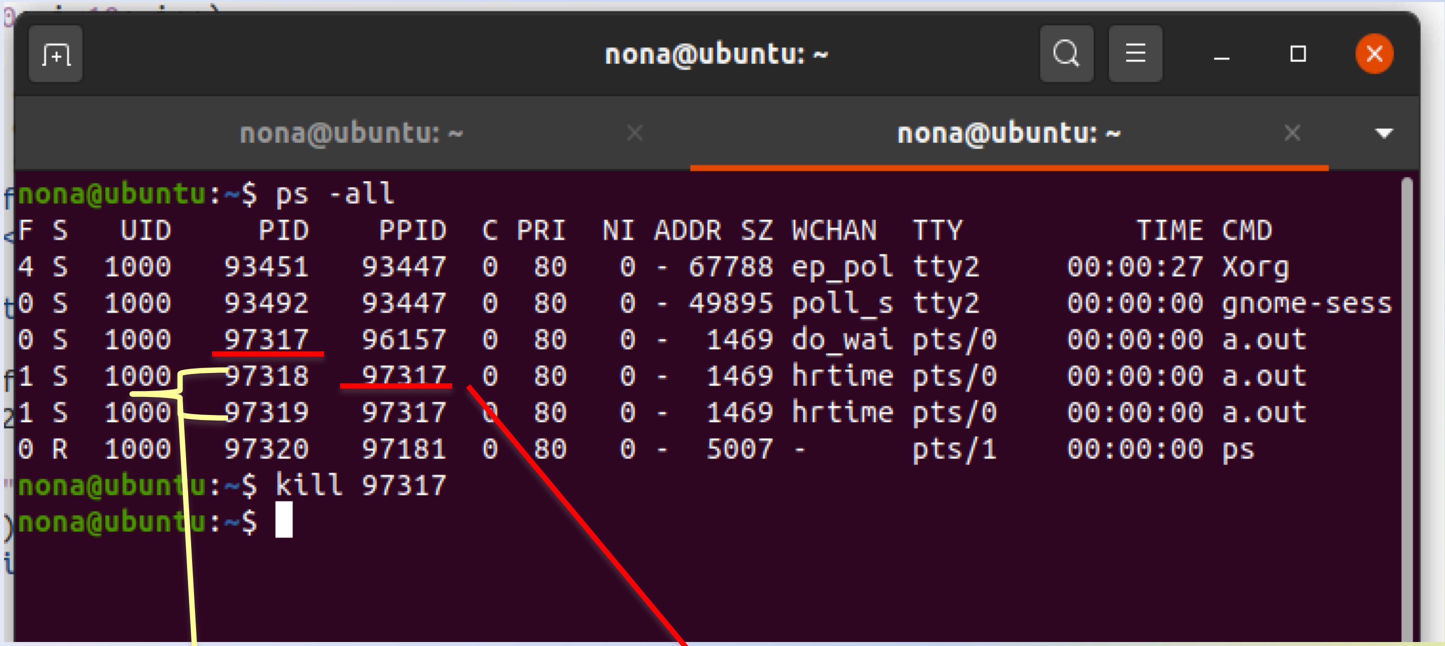


```
nona@ubuntu:~$ ./a.out
Child2: a = 1
Child1: a = 1
Child2: a = 2
Child1: a = 2
Child2: a = 3
Child1: a = 3
Child2: a = 4
Child1: a = 4
Child2: a = 5
Child1: a = 5

*****Parent is Closing*****
nona@ubuntu:~$ Child2: a = 6
Child2: a = 7
Child2: a = 8
Child2: a = 9
Child2: a = 10

nona@ubuntu:~$ █
```



Solution

4) When you kill the main process, all children processes will be terminated. Main process (Parent)

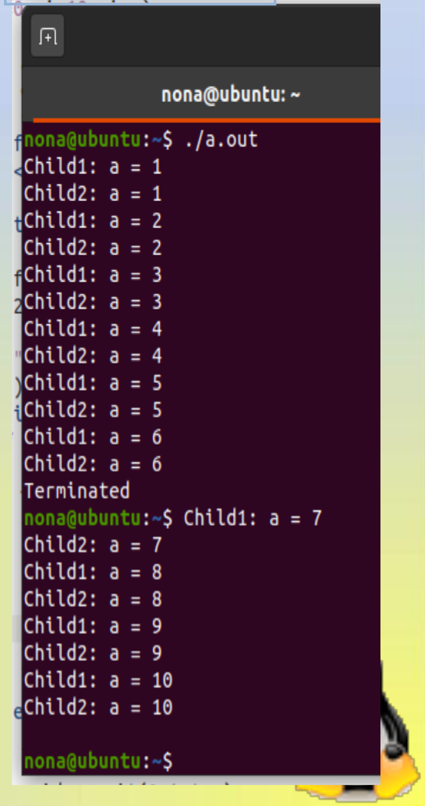


```
nona@ubuntu:~$ ps -all
F S   UID      PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000    93451   93447  0  80   0 - 67788 ep_pol tty2     00:00:27 Xorg
0 S  1000    93492   93447  0  80   0 - 49895 poll_s tty2     00:00:00 gnome-sess
0 S  1000    97317   96157  0  80   0 -  1469 do_wai pts/0    00:00:00 a.out
1 S  1000    97318   97317  0  80   0 -  1469 hrtime pts/0    00:00:00 a.out
1 S  1000    97319   97317  0  80   0 -  1469 hrtime pts/0    00:00:00 a.out
0 R  1000    97320   97181  0  80   0 -  5007 -      pts/1    00:00:00 ps
nona@ubuntu:~$ kill 97317
nona@ubuntu:~$ 
```

Child1 ID & Child2 ID

Parent ID

Result:



```
nona@ubuntu:~$ ./a.out
Child1: a = 1
Child2: a = 1
Child1: a = 2
Child2: a = 2
Child1: a = 3
Child2: a = 3
Child1: a = 4
Child2: a = 4
Child1: a = 5
Child2: a = 5
Child1: a = 6
Child2: a = 6
Terminated
nona@ubuntu:~$ Child1: a = 7
Child2: a = 7
Child1: a = 8
Child2: a = 8
Child1: a = 9
Child2: a = 9
Child1: a = 10
Child2: a = 10

nona@ubuntu:~$
```

Important

- **The ps Command**

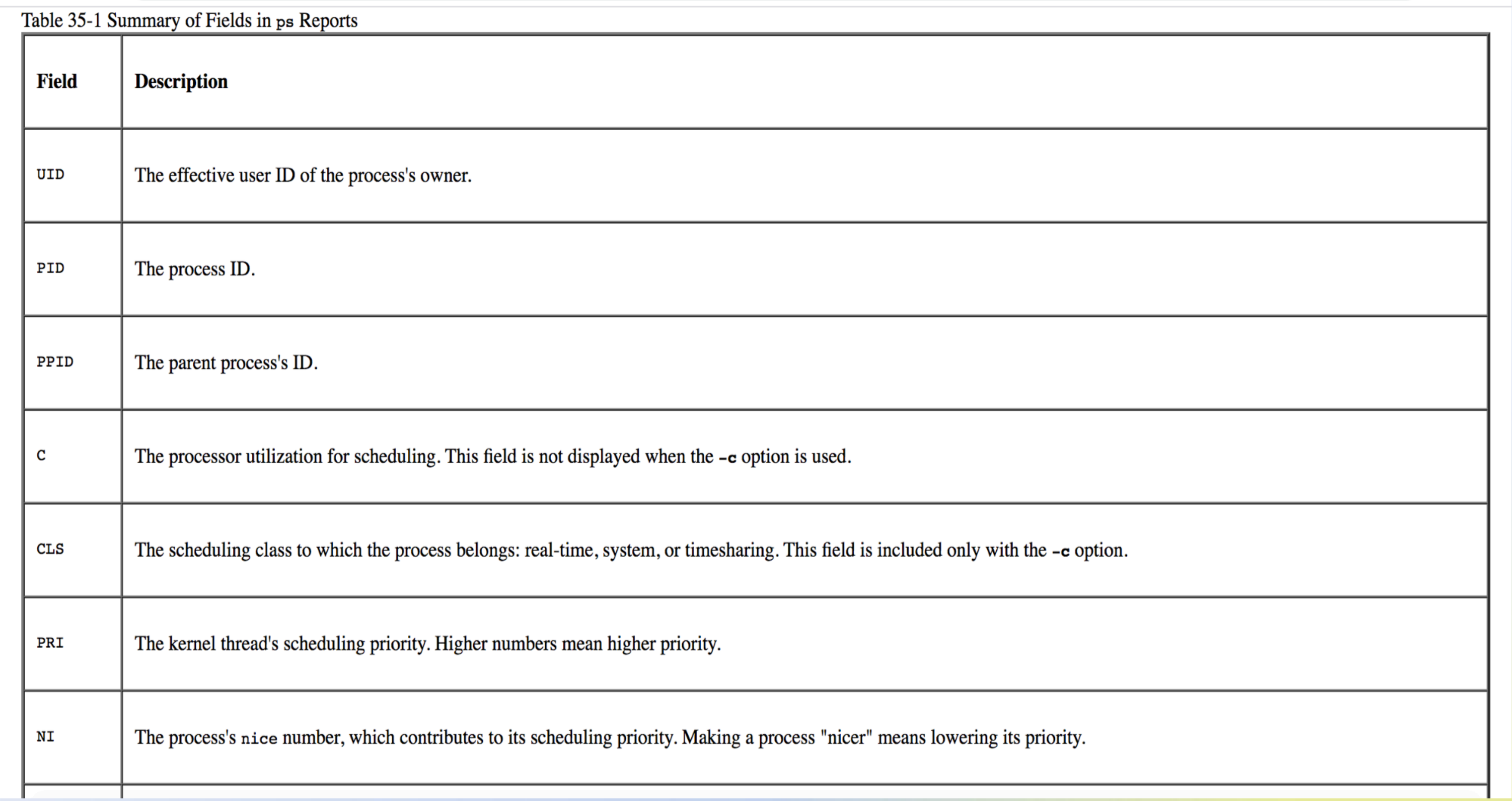


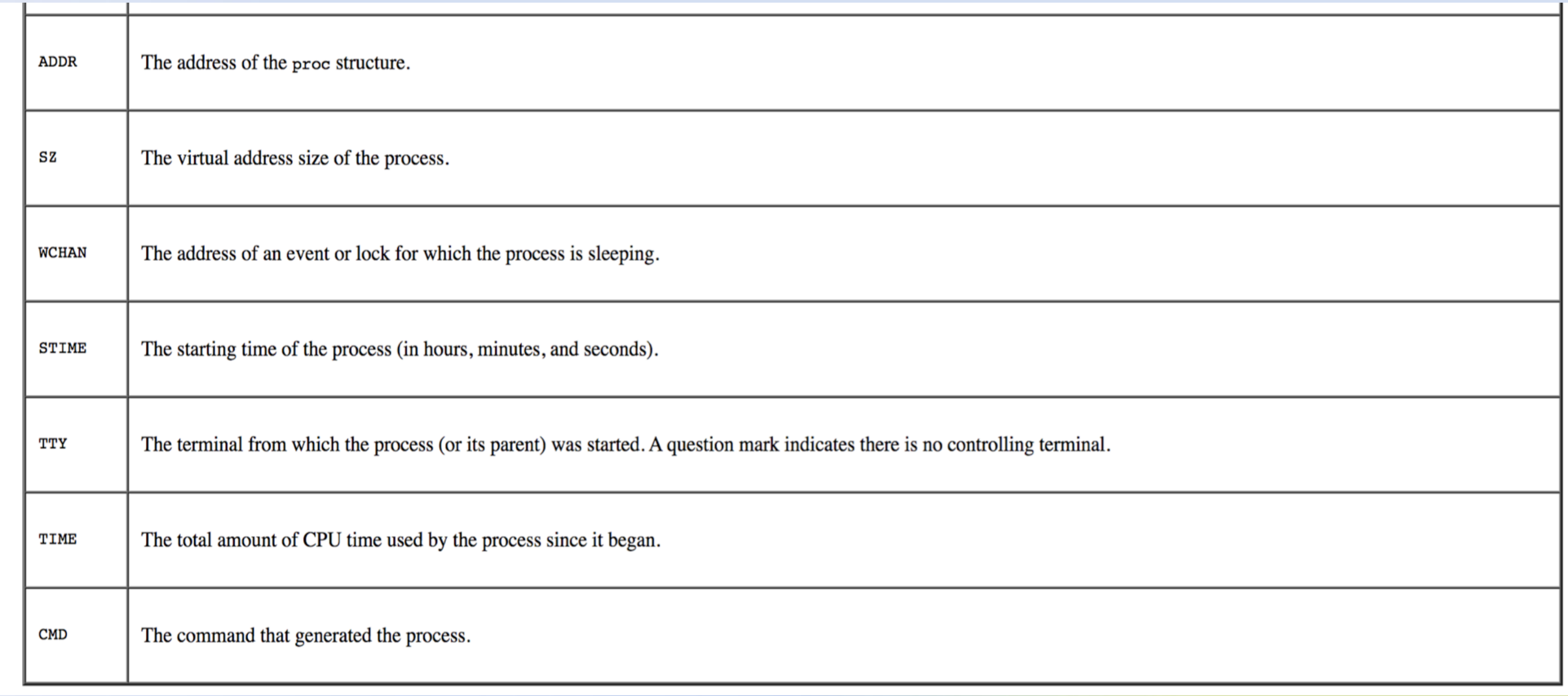
Table 35-1 Summary of Fields in ps Reports

Field	Description
UID	The effective user ID of the process's owner.
PID	The process ID.
PPID	The parent process's ID.
C	The processor utilization for scheduling. This field is not displayed when the -c option is used.
CLS	The scheduling class to which the process belongs: real-time, system, or timesharing. This field is included only with the -c option.
PRI	The kernel thread's scheduling priority. Higher numbers mean higher priority.
NI	The process's nice number, which contributes to its scheduling priority. Making a process "nicer" means lowering its priority.

Source: https://docs.oracle.com/cd/E19455-01/805-7229/6j6q8svgp/index.html

Important

- **The ps Command**



ADDR	The address of the `proc` structure.
SZ	The virtual address size of the process.
WCHAN	The address of an event or lock for which the process is sleeping.
STIME	The starting time of the process (in hours, minutes, and seconds).
TTY	The terminal from which the process (or its parent) was started. A question mark indicates there is no controlling terminal.
TIME	The total amount of CPU time used by the process since it began.
CMD	The command that generated the process.

Source: https://docs.oracle.com/cd/E19455-01/805-7229/6j6q8svgp/index.html

??? ANY QUESTIONS ???

☺

Threads

Lab 04

اللهم علمنا ما ينفعنا،، وانفعنا بما علمتنا،، وزدنا علماً

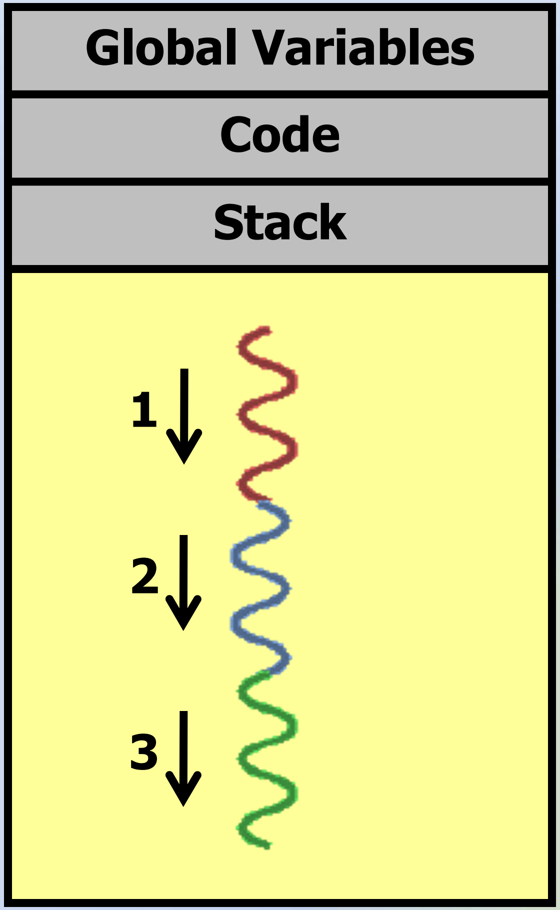
Lab Objective

- To practice using threads.

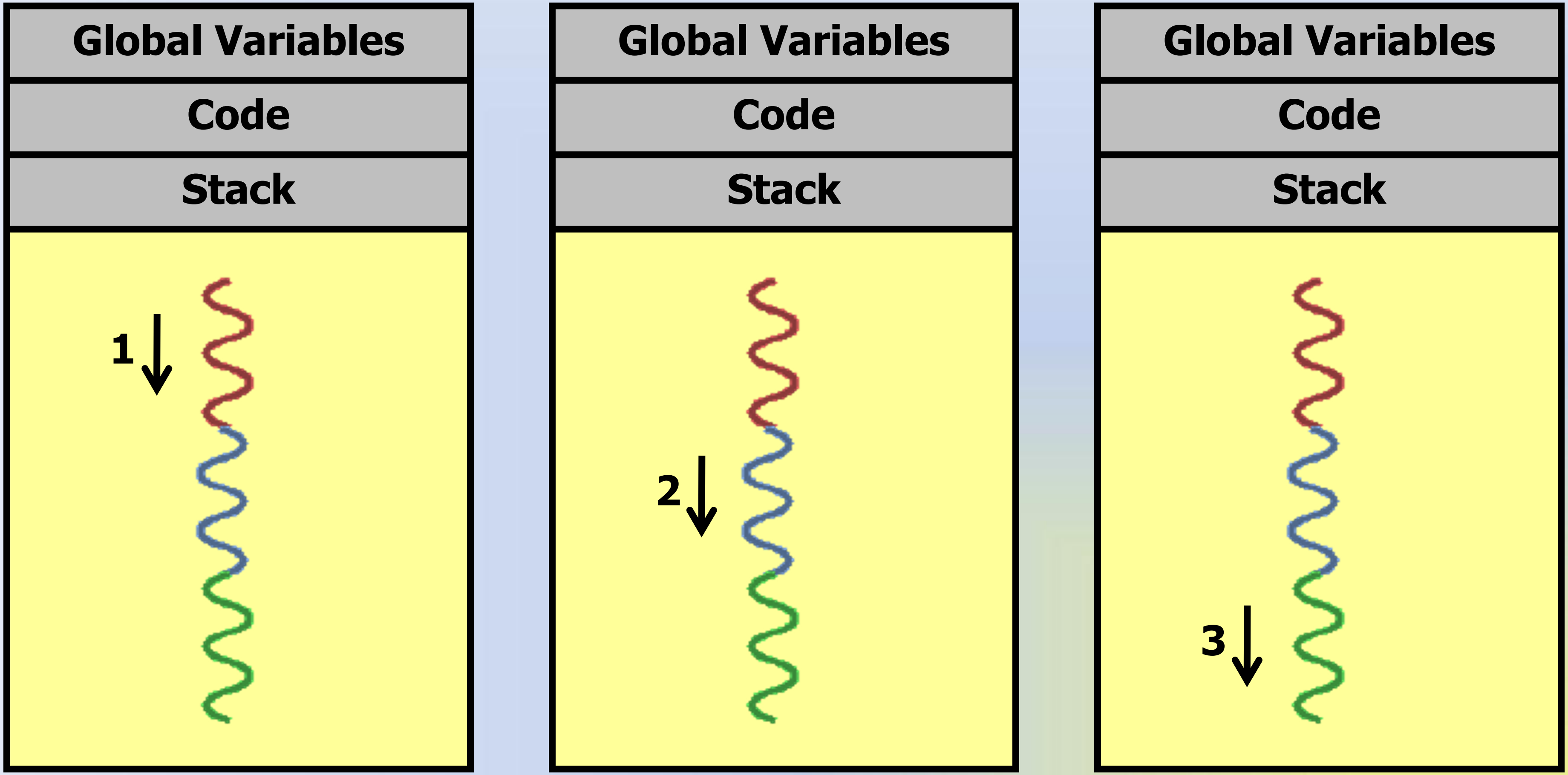
Threads are Fun!!

THREADS VS. PROCESSES

Single Process



Multiple Processes using `fork()`



Global Variables

Code

Stack

Global Variables

Code

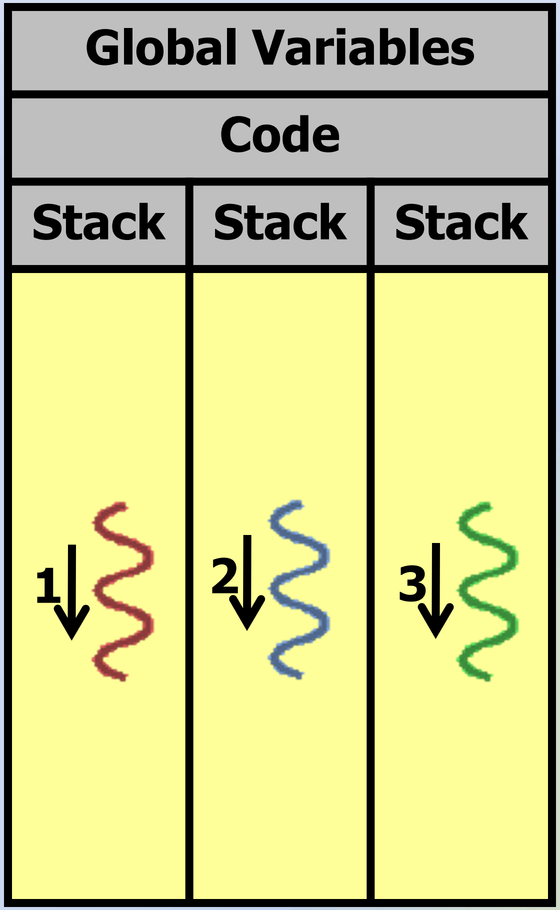
Stack

Global Variables

Code

Stack

Single Process with Multiple Threads



Thread Creation

- When a program is started, a single thread is created, called the *initial thread* or *main thread*.
- Additional threads are created by:

```
int pthread_create (pthread_t * tid,①
                    const pthread_attr_t *attr,②
                    void *(*func) (void*),③
                    void *arg);④
```

- Returns 0 if OK, positive Exxx value on error

① `tid` → The newly-created thread ID

② `attr` → the new thread attributes, use NULL to get system default

③ `func` → Pointer to a function to execute when the thread starts

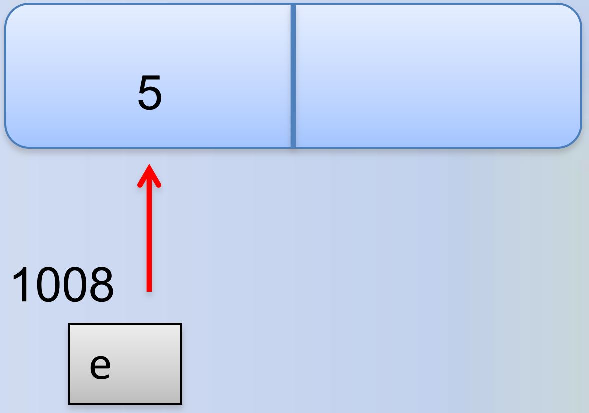
④ `arg` → Pointer to `func` argument (multiple arguments can be passed by creating a structure and passing the address of the structure)

Example of structre

```
struct arg {
Char x[10];
Int d;
Float salary;
};
```

Note: Pointer

```
int *e;
count<< e; //1008
Count<< *e; //5
```



```
Count<< &e; //1008
```

Thread Management

- Each thread has a unique ID, a thread can find out its ID by calling:

```
pthread_t pthread_self();
```

- A thread can be terminated by calling:

```
void pthread_exit();
```

- The main thread can wait for a thread to terminate by calling:

```
int pthread_join(pthread_t tid, void **status);
```

  - **Note:** with `pthread_join` with we must specify the `tid` of the thread.

Very Important Note

- Use the option `-pthread` or `-lpthread` with the compilation command to enable the support of multithreading with the pthread library.
- Your command line should look something like this:

```
$ g++ lab4.CC -pthread
```

A simple C program to demonstrate use of pthread basic functions Please note that the below program may compile only with C compilers with pthread library.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>
// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{ sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}
int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);  }
```

How to compile above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

Output:

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Before Thread
Printing GeeksQuiz from Thread
After Thread
gfg@ubuntu:~/$
```

A C program to show multiple threads with global and static variables

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

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Store the value argument passed to this thread
    int *myid = (int *)vargp;

    // Let us create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    ++s; ++g;

    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}
```

```
int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);

    pthread_exit(NULL);
    return 0;
}
```

Output:

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Thread ID: 3, Static: 2, Global: 2
Thread ID: 3, Static: 4, Global: 4
Thread ID: 3, Static: 6, Global: 6
gfg@ubuntu:~/$
```

References:
http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

Practice

- In the following C++ program, the main process creates two threads of the function `doit`
- The function has a loop to increment the global variable `counter` by 1 for 10 times.
- Within every iteration of the loop, the function prints out the ID of the thread that is running and the current value of `counter`
- Write, compile and run the program in Linux then answer the questions in the check-off section.

```
#include <iostream>
#include <unistd.h>// important for using sleep()
#include "pthread.h”
using std::cout;
using std::dec;  //To display numbers in decimal format
using std::endl; //Output a new line
#define NLOOP 10 //Constant value
int counter = 0;
void * doit(void *);
int main()
{
        pthread_t tidA, tidB;
        pthread_create(&tidA, NULL, doit, NULL);
        pthread_create(&tidB, NULL, doit, NULL);
        pthread_join(tidA, NULL);
        pthread_join(tidB, NULL);
        exit(0);
}//end main
void * doit(void *vprt)
{
        int i, val;
        for( i = 0; i<NLOOP; i++) {
                val = counter;
                cout<<"Thread = "<<pthread_self();
                cout<<" Counter = "<<dec<<counter<<endl;
                sleep(2);
                counter = val+1;}
        return (NULL);
} //end doit function
```

Global variable incremented by the threads

Create two threads to run the function `doit`

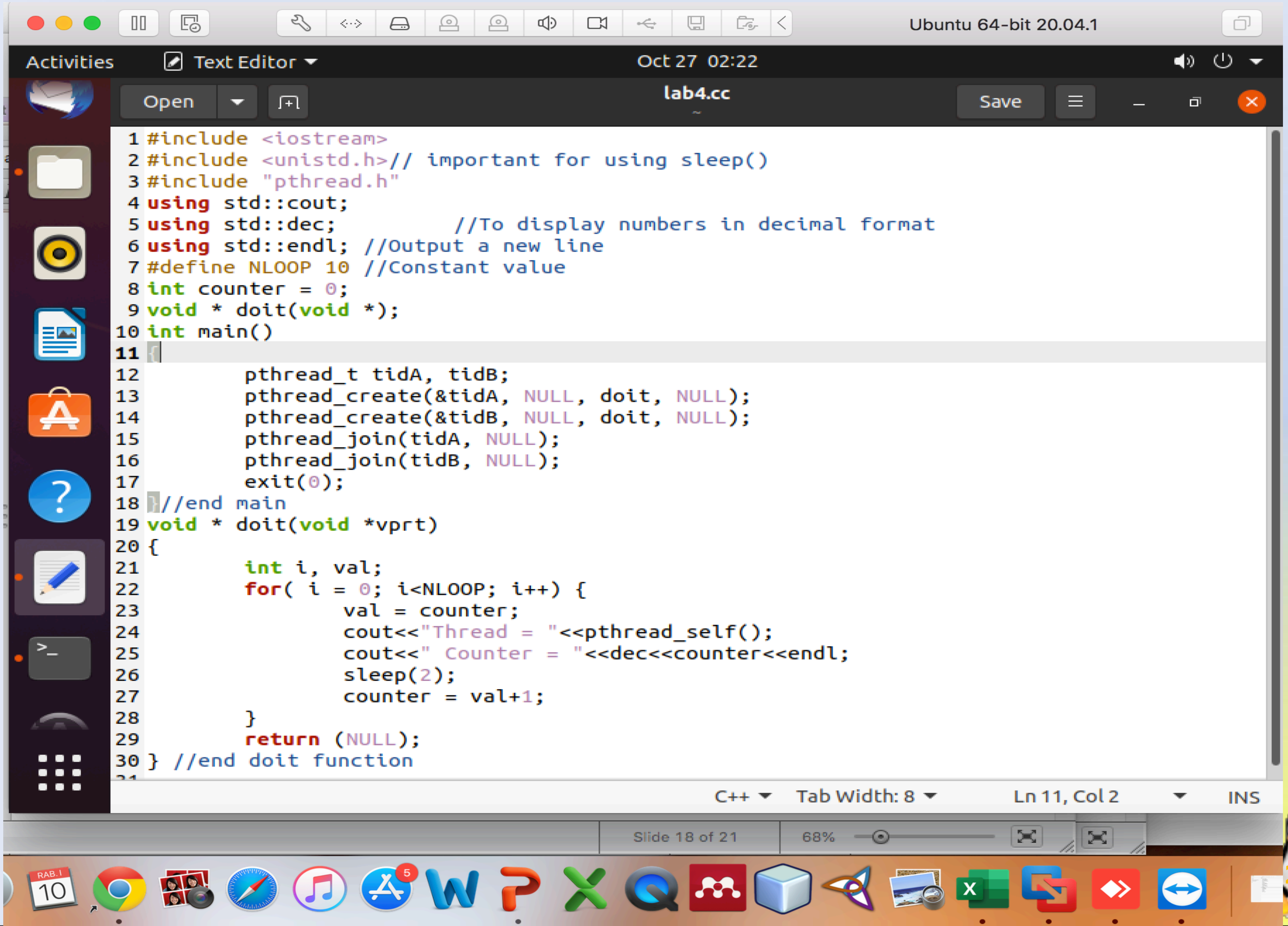
Wait for both threads to terminate

Each thread increments the global variable `counter` by 1 for 10 times

on Ubuntu

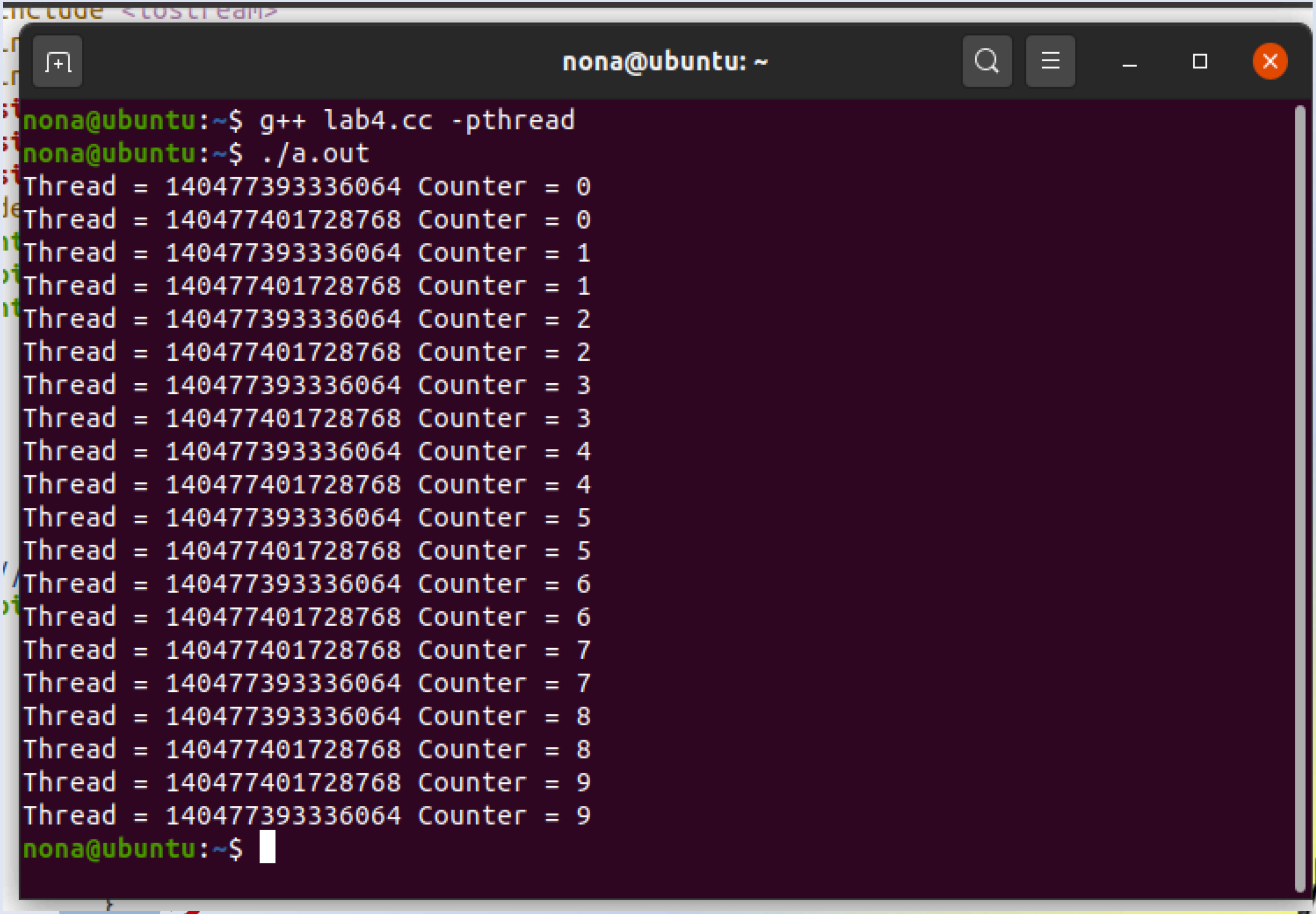
```
#include <iostream>
#include <unistd.h>// important for using sleep()
#include "pthread.h"
using std::cout;
using std::dec;        //To display numbers in decimal format
using std::endl; //Output a new line
#define NLOOP 10 //Constant value
int counter = 0;
void * doit(void *);
int main()
{
        pthread_t tidA, tidB;
        pthread_create(&tidA, NULL, doit, NULL);
        pthread_create(&tidB, NULL, doit, NULL);
        pthread_join(tidA, NULL);
        pthread_join(tidB, NULL);
        exit(0);
}//end main
void * doit(void *vprt)
{
        int i, val;
        for( i = 0; i<NLOOP; i++) {
                val = counter;
                cout<<"Thread = "<<pthread_self();
                cout<<" Counter = "<<dec<<counter<<endl;
                sleep(2);
                counter = val+1;
        }
        return (NULL);
} //end doit function
```

on Ubuntu



```cpp
#include <iostream>
#include <unistd.h>// important for using sleep()
#include "pthread.h"
using std::cout;
using std::dec;          //To display numbers in decimal format
using std::endl; //Output a new line
#define NLOOP 10 //Constant value
int counter = 0;
void * doit(void *);
int main()
{
        pthread_t tidA, tidB;
        pthread_create(&tidA, NULL, doit, NULL);
        pthread_create(&tidB, NULL, doit, NULL);
        pthread_join(tidA, NULL);
        pthread_join(tidB, NULL);
        exit(0);
}//end main
void * doit(void *vprt)
{
        int i, val;
        for( i = 0; i<NLOOP; i++) {
                val = counter;
                cout<<"Thread = "<<pthread_self();
                cout<<" Counter = "<<dec<<counter<<endl;
                sleep(2);
                counter = val+1;
        }
        return (NULL);
} //end doit function
```

Output



```
nona@ubuntu:~$ g++ lab4.cc -pthread
nona@ubuntu:~$ ./a.out
Thread = 140477393336064 Counter = 0
Thread = 140477401728768 Counter = 0
Thread = 140477393336064 Counter = 1
Thread = 140477401728768 Counter = 1
Thread = 140477393336064 Counter = 2
Thread = 140477401728768 Counter = 2
Thread = 140477393336064 Counter = 3
Thread = 140477401728768 Counter = 3
Thread = 140477393336064 Counter = 4
Thread = 140477401728768 Counter = 4
Thread = 140477393336064 Counter = 5
Thread = 140477401728768 Counter = 5
Thread = 140477393336064 Counter = 6
Thread = 140477401728768 Counter = 6
Thread = 140477401728768 Counter = 7
Thread = 140477393336064 Counter = 7
Thread = 140477393336064 Counter = 8
Thread = 140477401728768 Counter = 8
Thread = 140477401728768 Counter = 9
Thread = 140477393336064 Counter = 9
nona@ubuntu:~$ 
```

Check Off

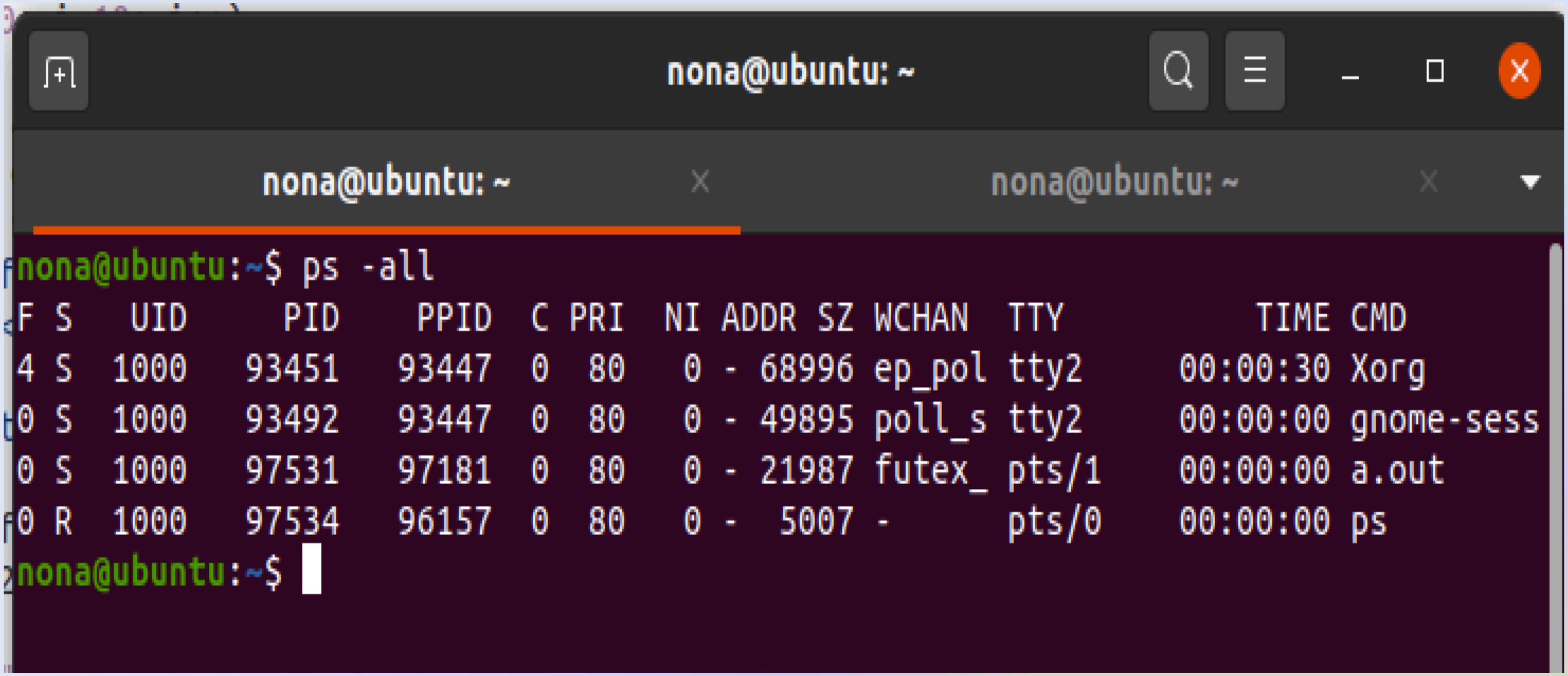
1) Why the final value of **counter** is 10 and not 20?
2) Run the program again. while it’ running, use the command **ps –all** in a separate window. Write down the **PID** of the process(es) related to the program. Explain the difference between this program and the program you had in the previous lab in terms of number of PIDs.
3) modify the loop in the **doit** function to be as follows:

```
for( i = 0; i<NLOOP; i++) {
        cout<<"Thread = "<<pthread_self();
        cout<<" Counter = "<<dec<<counter<<endl;
        counter++;
```

Recompile the program and run it. what is the maximum value of **counter** ?

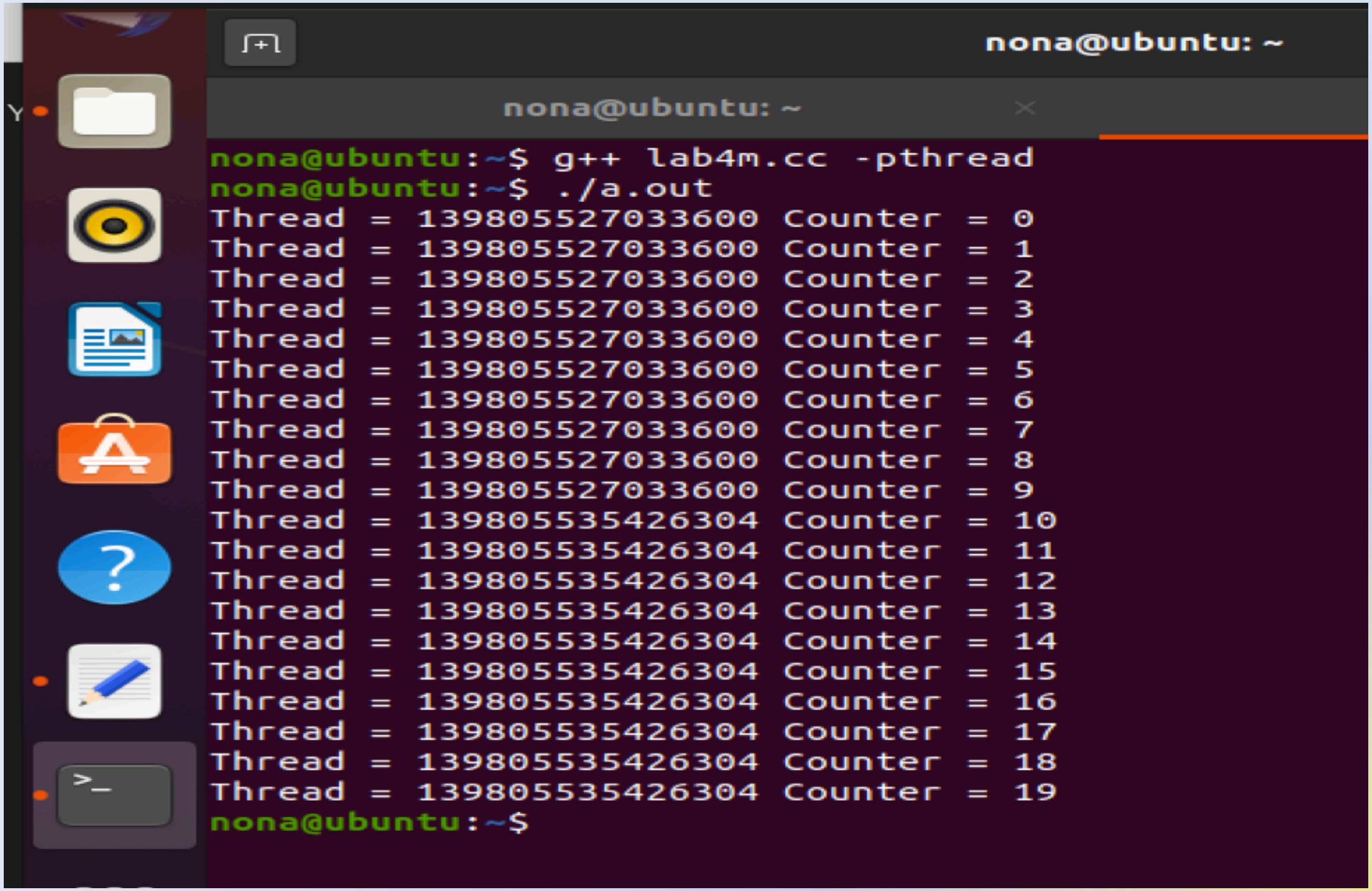
4) Briefly explain the behavior of the program based on the results you obtain from the previous questions.

Answer for question 2:



```
nona@ubuntu:~$ ps -all
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000   93451   93447  0  80   0 - 68996 ep_pol tty2     00:00:30 Xorg
0 S  1000   93492   93447  0  80   0 - 49895 poll_s tty2     00:00:00 gnome-sess
0 S  1000   97531   97181  0  80   0 - 21987 futex_ pts/1    00:00:00 a.out
0 R  1000   97534   96157  0  80   0 -  5007 -      pts/0    00:00:00 ps
nona@ubuntu:~$ 
```

Answer for question 3:



```
nona@ubuntu:~$ g++ lab4m.cc -pthread
nona@ubuntu:~$ ./a.out
Thread = 139805527033600 Counter = 0
Thread = 139805527033600 Counter = 1
Thread = 139805527033600 Counter = 2
Thread = 139805527033600 Counter = 3
Thread = 139805527033600 Counter = 4
Thread = 139805527033600 Counter = 5
Thread = 139805527033600 Counter = 6
Thread = 139805527033600 Counter = 7
Thread = 139805527033600 Counter = 8
Thread = 139805527033600 Counter = 9
Thread = 139805535426304 Counter = 10
Thread = 139805535426304 Counter = 11
Thread = 139805535426304 Counter = 12
Thread = 139805535426304 Counter = 13
Thread = 139805535426304 Counter = 14
Thread = 139805535426304 Counter = 15
Thread = 139805535426304 Counter = 16
Thread = 139805535426304 Counter = 17
Thread = 139805535426304 Counter = 18
Thread = 139805535426304 Counter = 19
nona@ubuntu:~$
```

??? ANY QUESTIONS ???

☺

Fork VS. Threads

Lab 05

اللهم علمنا ما ينفعنا،، وانفعنا بما علمتنا،، وزدنا علماً

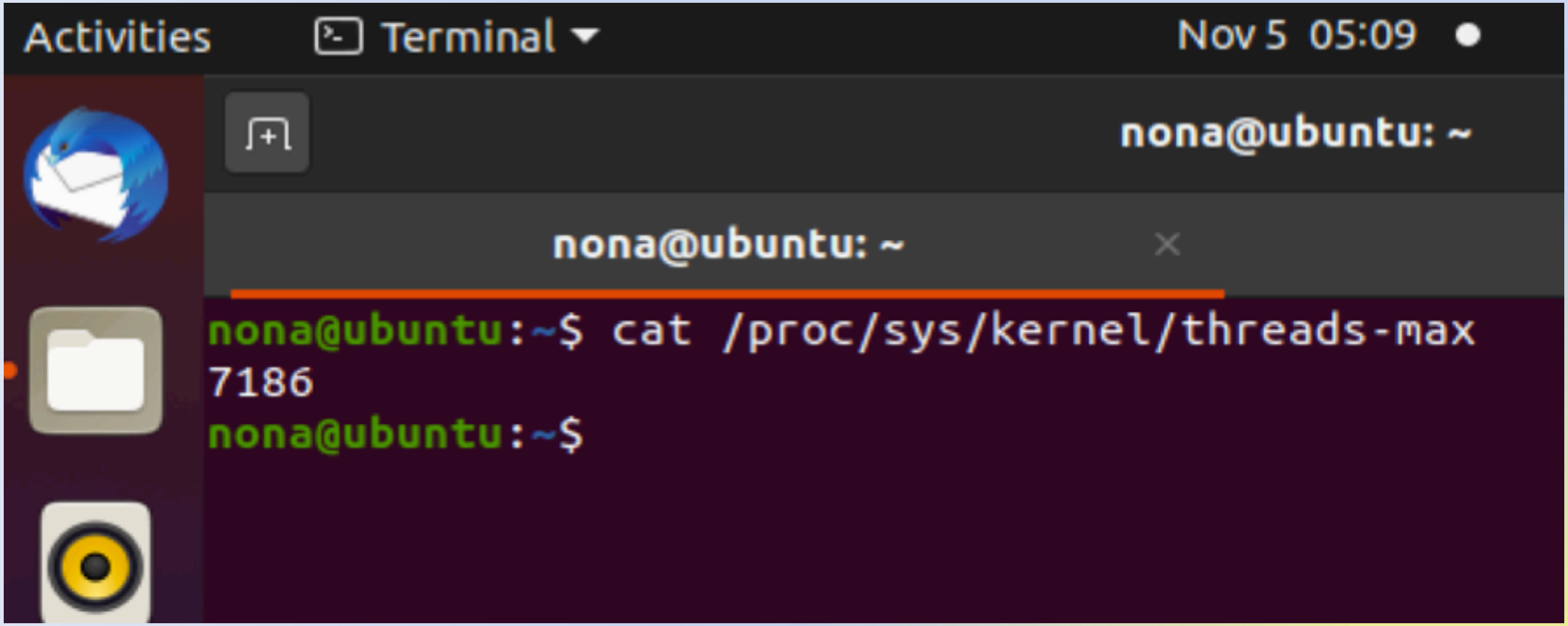
Lab Objective

- To understand the deference between thread and fork.

Maximum number of threads that can be created within a process in C:

- **Maximum number of threads can be seen is ubuntu by using command:**

cat /proc/sys/kernel/threads-max

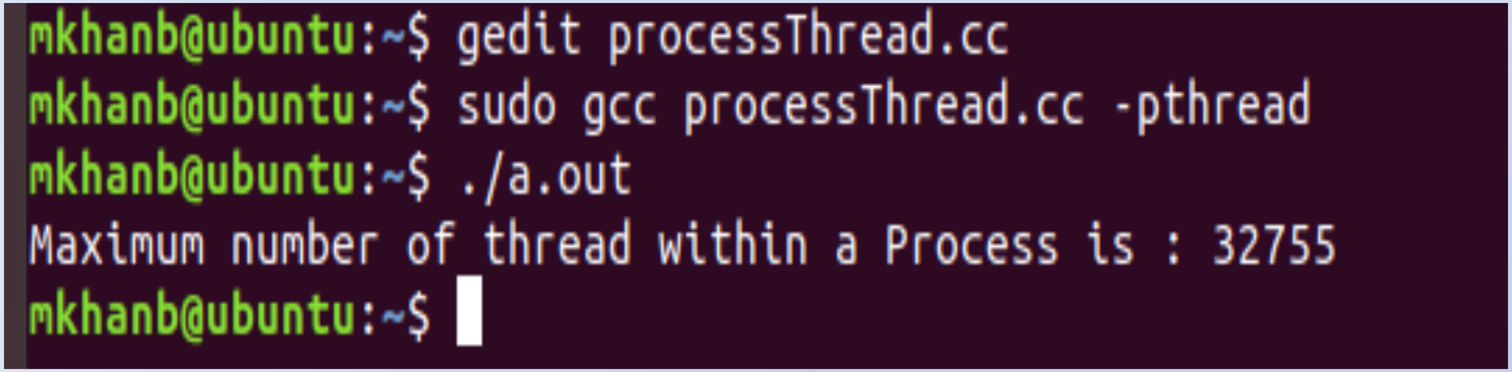


```
nona@ubuntu:~$ cat /proc/sys/kernel/threads-max
7186
nona@ubuntu:~$
```

```
// C program to find maximum number of thread within
// a process
#include<stdio.h>
#include<pthread.h>

// This function demonstrates the work of thread
// which is of no use here, So left blank
void *thread ( void *vargp){     }
 int main()
{
   int err = 0, count = 0;
   pthread_t tid;

   // on success, pthread_create returns 0 and
   // on Error, it returns error number
   // So, while loop is iterated until return value is 0
   while (err == 0)
   {
      err = pthread_create (&tid, NULL, thread, NULL);
      count++;
   }
   printf("Maximum number of thread within a Process"
                                   " is : %d\n", count);
}
```

Maximum number of threads that can be created within a process in C:

Use following commond to compile and run filename is **processThread.cc**



```
mkhanb@ubuntu:~$ gedit processThread.cc
mkhanb@ubuntu:~$ sudo gcc processThread.cc -pthread
mkhanb@ubuntu:~$ ./a.out
Maximum number of thread within a Process is : 32755
mkhanb@ubuntu:~$ 
```

- **Opening file using the command:**

gedit filename

Practice

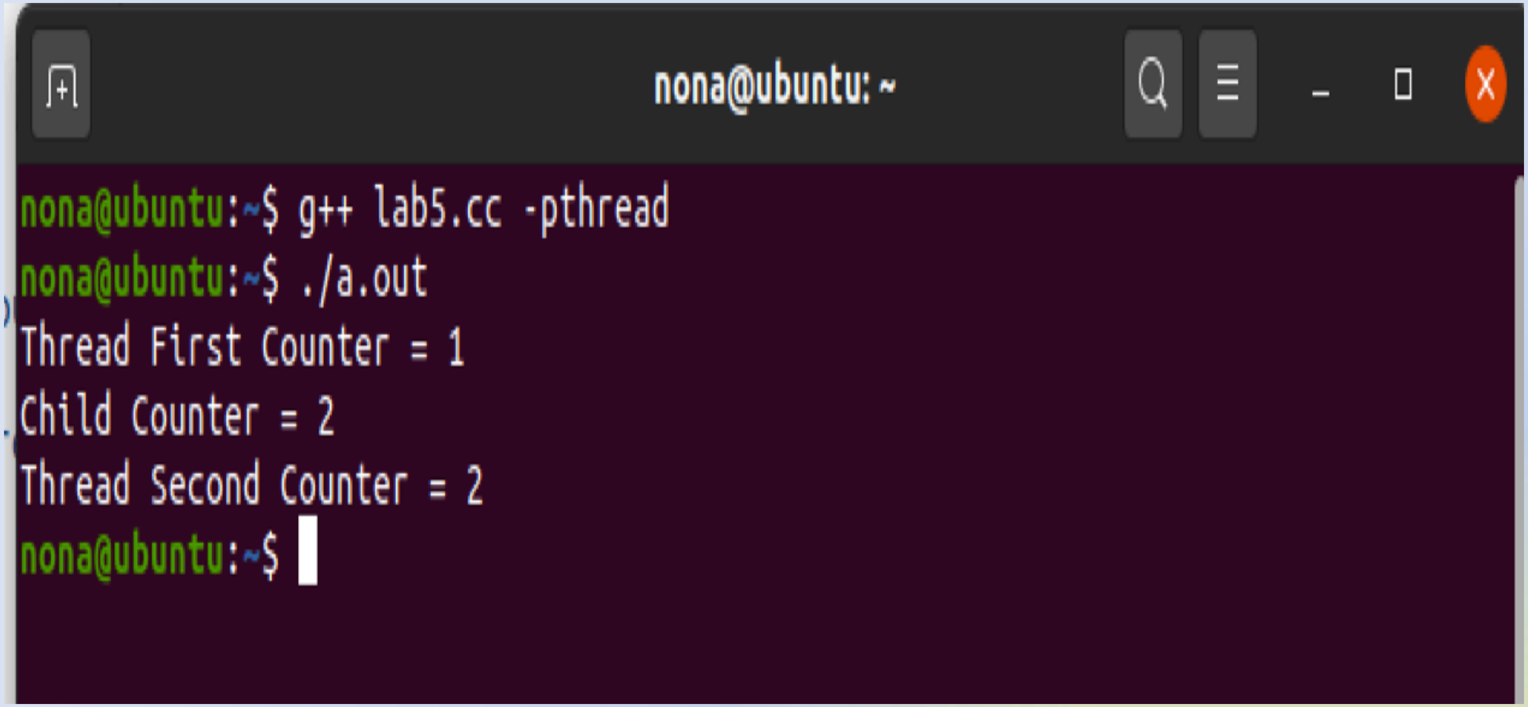
- In the following C++ program, the main process creates one thread of the function `doit` and forks one child.
- Both the `doit` function and the child code increment and display the global variable `counter`.
- Write, compile, and run the program.

```
#include <iostream>
#include <stdlib.h>  /* exit() */
#include <unistd.h>  /* fork() */
#include <sys/types.h>      /* pid_t */
#include <sys/wait.h>/* wait() */
#include "pthread.h"
using std::cout;
using std::endl; //Output a new line

int counter = 0; //Incremented by the threads and child
void * doit(void *);
int main()
{
    pthread_t tid;
    pid_t pid, cpid;
    int status;
    // Start the thread
    pthread_create(&tid, NULL, doit, NULL);
    //Delay between starting the thread and forking the child
    sleep(2);
    pid = fork(); //Fork the Child
if (pid < 0){
       cout<<“Fork Failed\n”;
       exit(-1);
    }
else if (pid == 0) { // child process
       sleep(2);
       counter++;
       cout << "Child Counter = " << counter << endl;
    }
    else // parent process
     // parent will wait for the child to complete
     cpid = wait(&status);
     // wait for the thread to terminate
     pthread_join(tid, NULL);
     exit(0);
} // End main
void * doit(void *vptr)
{
        sleep(1);
        cout << "Thread First Counter = " <<
              ++counter << endl;
        sleep(5);
        cout << "Thread Second Counter = " <<
              ++counter << endl;
       return(NULL);
}
```

```
#include <iostream>
#include <stdlib.h> /* exit() */
#include <unistd.h> /* fork() */
#include <sys/types.h>     /* pid_t */
#include <sys/wait.h>      /* wait() */
#include "pthread.h"
using std::cout;
using std::endl; //Output a new line

int counter = 0; //Incremented by the threads and child
void * doit(void *);
int main()
{
pthread_t tid;
pid_t pid, cpid;
    int status;
    // Start the thread
    pthread_create(&tid, NULL, doit, NULL);
//Delay between starting the thread and forking the child
    sleep(2);
    pid = fork(); //Fork the Child
if (pid < 0){
       cout<<"Fork Failed\n";
       exit(-1);
    }
else if (pid == 0) { // child process
       sleep(2);
       counter++;
       cout << "Child Counter = " << counter << endl;
    }
    else // parent process
     // parent will wait for the child to complete
     cpid = wait(&status);
     // wait for the thread to terminate
     pthread_join(tid, NULL);
     exit(0);
} // End main
```

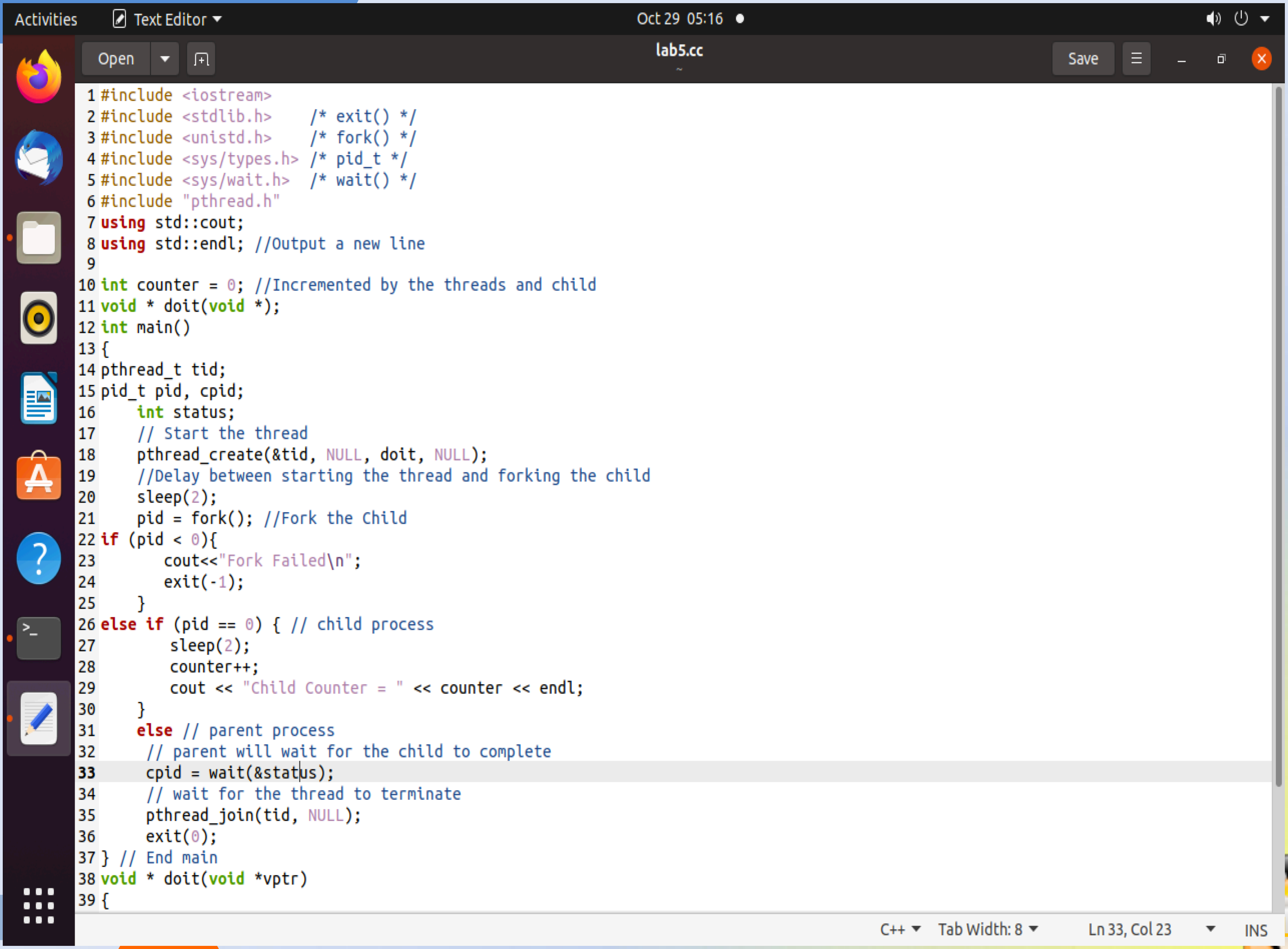
```
void * doit(void *vptr)
{
        sleep(1);
        cout << "Thread First Counter = " <<
             ++counter << endl;
        sleep(5);
        cout << "Thread Second Counter = " <<
             ++counter << endl;
       return(NULL);
}
```

Output



```
nona@ubuntu:~$ g++ lab5.cc -pthread
nona@ubuntu:~$ ./a.out
Thread First Counter = 1
Child Counter = 2
Thread Second Counter = 2
nona@ubuntu:~$ 
```

on Ubuntu



```
#include <iostream>
#include <stdlib.h>    /* exit() */
#include <unistd.h>    /* fork() */
#include <sys/types.h> /* pid_t */
#include <sys/wait.h>  /* wait() */
#include "pthread.h"
using std::cout;
using std::endl; //Output a new line

int counter = 0; //Incremented by the threads and child
void * doit(void *);
int main()
{
pthread_t tid;
pid_t pid, cpid;
    int status;
    // Start the thread
    pthread_create(&tid, NULL, doit, NULL);
    //Delay between starting the thread and forking the child
    sleep(2);
    pid = fork(); //Fork the Child
if (pid < 0){
       cout<<"Fork Failed\n";
       exit(-1);
    }
else if (pid == 0) { // child process
        sleep(2);
        counter++;
        cout << "Child Counter = " << counter << endl;
    }
    else // parent process
     // parent will wait for the child to complete
     cpid = wait(&status);
     // wait for the thread to terminate
     pthread_join(tid, NULL);
     exit(0);
} // End main
void * doit(void *vptr)
{
```

Check Off

1) What are the printed values of `counter`? Explain why `counter` gets these values from the child and the thread.
2) Remove the `sleep(2)` line that delays between starting the thread and forking the child. Recompile the program and run it. What are the new printed values of `counter`? Explain why `counter` gets these values from the child and the thread.

Check Off

1) Thread 1st counter = 1

Child counter = 2

Thread 2nd counter =2

2) Thread 1st counter = 1

Child counter = 1

Thread 2nd counter =2

??? ANY QUESTIONS ???

☺