Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Student Name	
Student ID	

List of Experiments/Content

Lab #	Date	Objective	Obtained Marks	Signature
1		Executing some of the most frequently used Linux commands		
2		Managing Files & Directories in Linux		
3		Programming using Shell Scripting.		
4		Focusing on the usage of the test command and conditional statements.		
5		Focusing on the usage of iteration statements and functions in shell programming		
6		Understanding Process. Process creation in Linux using System Calls. • Fork() method. • Zombie and Orphan processes.		
7		Understanding Threads. • Threads vs. Processes. • Multithreaded Programming using Python and C.		
8		Simulation of FCFS CPU scheduling algorithm. Simulation of SJF CPU scheduling algorithm.		
9		Simulation of Round Robin CPU scheduling algorithm. Simulation of Priority CPU scheduling algorithm.		
10		Implementation of Semaphore Mechanism. Solving producer-consumer (Classical Problem) problem in Python using semaphores.		
11		Inter process communication (IPC) using Pipe. • Multiprocessing in Python. • Implement Pipe using *os* and *multiprocessing* module in Python.		
12		Inter process communication (IPC) using Shared Memory. • Implement Shared Memory using multiprocessing module in Python.		
13		Implementation of Deadlock Avoidance Mechanism (Banker's Algorithm).		
14		**Open Ended Lab** **Objective:** Write a GUI based Shell script that behaves like an Operating System.		

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #1

Objective:
Executing some of the most frequently used Linux commands
To understand and use basic utilities/Commands of UNIX/LINUX.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective:

To understand and use basic utilities/Commands of UNIX/LINUX.

THEORY

A Linux/UNIX command is any executable file. This means that any executable file added to the system becomes a new command on the system. A Linux command is a type of file that is designed to be run, as opposed to files containing data or configuration information.

Input and Output Redirection

Linux allows to "pipe" the output from a command so that it becomes another command's input. This is done by typing two or more commands separated by the **|** character. The **|** character means "Use the output from the previous command as the input for the next command." Therefore, typing command_1|command_2 does both commands, one after the other, before giving you the results. Another thing you can do in Linux is to send output to a file instead of the screen. To send output to a file, use the “>” symbol. There are many different reasons why you might want to do this. You might want to save a "snapshot" of a command's output as it was at a certain time, or you might want to save a command's output for further examination. You might also want to save the output from a command that takes a very long time to run, and so on.

Command Options & other parameters

You can use command options to fine-tune the actions of a Linux command. Instead of making you learn a second command, Linux lets you modify the basic, or default, actions of the command by using options. Linux commands often use parameters that are not actual command options. These parameters, such as filenames or directories, are not preceded by a dash.

Types of Commands

There are three types of commands in Linux

1. Simple: Execute only by name, *e.g*, date command in Windows
2. Complex: With which we have some arguments (actions on commands) or options (can change the behavior of command) *e.g*, dir, format, dir/w, copy
3. Compound: A mixture of two or more commands

[root@localhost root]# *Note: Every user has assigned a directory*

[UserName @ Machine Name Directory name]

Executing a Linux Command

From the command prompt simply type the name of the command:

$ *command*

Where $ is the prompt character for the Shell (Bourne shell).

Or if the command is not on your path type the complete path and name of the command such as:

$ /usr/bin/*command.*

Some of the frequently used Linux commands:

1. su

Description: "su" stands for "super user". Runs a new shell under different user and group IDs. If no user is specified, the new shell will run as the root user.

Syntax: su [-flmp] [-c command] [-s shell] [--login] [--fast] [--preserve-environment] [-command=command] [--shell=shell] [-] [user]

(See the man pages for the description of the flags and options by using man su)

```
$ su <username>   (to become another user) or
$ su              (to become the root user).
```

Adding a new user

```
#useradd user1
#passwd user1
<password>
```

2. pwd

Description: Displays the name of the current directory.pwd stands for present working directory. By typing this command, you are informed of which directory you are currently in.

Syntax: pwd

3. cd

Description: Changes the current directory to any accessible directory on the system.

Syntax: For instance, to change from /home/user1 to a subdirectory of user1 wordfiles use the following:

```
$ cd wordfiles
```

4. ls

Description: Displays the listing of files and directories. If no file or directory is specified, then the current directory's contents are displayed. By default, the contents are sorted alphabetically.

Syntax: To view the contents of user1 home directory use this:

```
$ ls
```

To list the contents of any directory on the system use:

```
$ ls /usr
```

Options: 2

-l Lists all the files, directories and their mode, Number of links, owner of the file, file size, Modified date and time and filename.
-t Lists in order of last modification time.
-a Lists all entries including hidden files.
-d Lists directory files instead of contents.
-p Puts slash at the end of each directory.
-u List in order of last access time.
-i Display inode information.

5. grep

Description: Searches files for lines matching a specific pattern and displays the lines

Grep stands for Global Regular Expression Parser. What grep does, essentially, is find and display lines that contain a pattern that you specify. There are two basic ways to use grep.

The first use of grep is to filter the output of other commands. The general syntax is <command> | grep

<pattern>. For instance, if we wanted to see every actively running process on the system, we would type ps -a | grep R. In this application, grep passes on only those lines that contain the pattern (in this case, the single letter) R. Note that if someone were running a program called Resting, it would show up even if its status were S for sleeping, because grep would match the R in Resting. An easy way around this problem is to type grep " R ", which explicitly tells grep to search for an R with a space on each side. You must use quotes whenever you search for a pattern that contains one or more blank spaces. The second use of grep is to search for lines that contain a specified pattern in a specified file. The syntax here is grep <pattern> <filename>. Be careful. It's easy to specify the filename first and the pattern second by mistake! Again, you should be as specific as you can with the pattern to be matched, in order to avoid "false" matches.

Syntax: Assuming that you are in your home directory, then the following command searches for the word “hello” in each file in your home directory and produces the results as follows: $ grep hello*

6. CLEAR

It is used to clear the screen.

Syntax: $clear

7. Date:

This command is used to display the current data and time.

Syntax: $date

Options: - a = Abbrevated weekday.
A = Full weekday.
b = Abbrevated month.
B = Full month.
c = Current day and time.
C = Display the century as a decimal number.
Y = Display the full year.
Z = Time zone .

8. echo:

echo command prints the given input string to standard output.

SYNTAX: echo [options...] [string]

Using man Pages

The man pages are manual pages provided in a standard format with most Linux software. Almost all the commands that ship with Red Hat Linux distribution include man pages. Using the man command in its most basic form, any existing man page can be read:

$ man command-name

The above displays the man page for the specified command and allows scrolling through it and searching it the same way as when using the less command to display text. If the specified man page cannot be found an error is displayed.

Lab Exercise(s):

1. Write Linux command to List all files (and subdirectories) in the home directory.

2. Write Linux command to Display the content of /etc/passwd file with as many lines at a time as the last digit of your roll number.

3. Write Linux command to Count all files in the current directory.

4. Use grep to search for the pattern: ‘The’ in a text file in the home directory

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #2

Objective:
Managing Files & Directories in Linux
To understand Linux directory structure, and file manipulation.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: To understand Linux directory structure, and file manipulation.

THEORY

Files & Filenames

The most basic concept of a file defines it as a distinct chunk of information that is found on the hard drive. Distinct means that there can be many different files, each with its own particular contents. To keep files from getting confused with each other, every file must have a unique identity. In Linux, you identify each file by its name and location. In each location or directory, there can be only one file by a particular name. Linux allows filenames to be up to 256 characters long. These characters can be lower- and uppercase letters, numbers, and other characters, usually the dash (-), the underscore (_), and the dot (.). They can't include reserved meta characters such as the asterisk, question mark, backslash, and space, because these all have meaning to the shell.

Directories

Linux, like many other computer systems, organizes files in directories. You can think of directories as file folders and their contents as the files. However, there is one absolutely crucial difference between the Linux file system and an office filing system. In the office, file folders usually don't contain other file folders. In Linux, file folders can contain other file folders. In fact, there is no Linux "filing cabinet"— just a huge file folder that holds some files and other folders. These folders contain files and possibly other folders in turn, and so on.

Naming Directories

Directories are named just like files, and they can contain upper- and lowercase letters, numbers, and characters such as -, ., and _. The slash (/) character is used to show files or directories within other directories.

Important Directories in the Linux File System

This section summarizes some of the more important directories on a Linux system.

/ This is the root directory. It holds the actual Linux program, as well as subdirectories. Do not clutter this directory with your files! This is the root directory. It holds the actual Linux program, as well as subdirectories. Do not clutter this directory with your files!

/home
This directory holds users' home directories. In other UNIX systems, this can be the /usr or /u directory.

/bin
This directory holds many of the basic Linux programs. bin stands for binaries, files that are executable and that hold text only computers could understand.

/usr
This directory holds many other user-oriented directories. Some of the most important are described in the following sections. Other directories found in /usr include

docs	Various documents, including useful Linux information
man	The man pages accessed by typing man <command>
games	The fun stuff!

/usr/bin
This directory holds user-oriented Linux programs.

sbin
This directory holds system files that are usually run automatically by the Linux system.
etc

This directory and its subdirectories hold many of the Linux configuration files. These files are usually text, and they can be edited to change the system's configuration (if you know what you're doing!).

Creating Files

Linux has many ways to create and delete files. In fact, some of the ways are so easy to perform that you have to be careful not to accidentally overwrite or erase files!
Return to your home directory by typing cd. Make sure you're in your /home/<user> directory by running pwd. A file can be created by typing ls -l /bin > test. Remember, the > symbol means "redirect all output to the following filename." Note that the file test didn't exist before you typed this command. When you redirect to a file, Linux automatically creates the file if it doesn't already exist.
What if you want to type text into a file, rather than some command's output? The quick way is to use the command cat.

The cat Command

The cat command is one of the simplest, yet most useful, commands in Linux. The cat command basically takes all its input and outputs it. By default, cat takes its input from the keyboard and outputs it to the screen. Type cat at the command line:

```
$ cat
```

The cursor moves down to the next line, but nothing else seems to happen. Now cat is waiting for some input:

```
Hello
```

Everything you type is repeated on-screen as soon as you press Enter.

So how do you use cat to create a file? Simple! You redirect the output from cat to the desired filename:

```
$ cat > newfile
     Hello world
     Here's some text
```

Moving and Copying Files

You often need to move or copy files. The mv command moves files, and the cp command copies files. The mv command is much more efficient than the cp command. When you use mv, the file's contents are not moved at all; rather, Linux makes a note that the file is to be found elsewhere within the file system's structure of directories.
When you use cp, you are actually making a second physical copy of your file and placing it on your disk. This can be slower (although for small files, you won't notice any difference), and it causes a bit more wear and tear on your computer. Don't make copies of files when all you really want to do is move them! The syntax for the two commands is similar:

```
mv <source> <destination>  cp <source> <destination>
```

In the Linux environment renaming a file is just a special case of moving a file. To move a file to /tmp use this:

```
$ mv fileone /tmp
$ mv fileone /tmp/newfilename
```

Similarly to make a copy file in the /tmp directory use the following command:

```
$ cp thisfile /tmp
```

and if you want to copy this file to /tmp but give the new file a different name, enter

```
$ cp thisfile /tmp/newfilename
```

Also, to avoid overwriting a file accidentally use the –i flag of the cp command which forces the system to confirm any file it will overwrite when copying. Then a prompt like the following appears:

```
$ cp –i  thisfile newfile  cp: overwrite thisfile?
```

Copying Multiple Files in One Command

In DOS only one file or file expression can be copied at a time. To copy three separate files then three commands must be issued. The Linux cp command makes this a bit easier. The cp command can take more than two arguments. If more than two arguments are passed to the command then the last one is treated as the destination and all preceding files are copied to this destination.

For example to copy fileone ,filetwo and filethree in the current directory to /tmp then the following commands can be issued:

```
$ cp fileone /tmp
$ cp filetwo /tmp
$ cp filethree /tmp
```

All this can be bundled into one command like this:

```
$ cp fileone filetwo filethree /tmp
```

Creating a Directory

To create a new directory, use the mkdir command. The syntax is mkdir <name>, where <name> is replaced by whatever you want the directory to be called. This creates a subdirectory with the specified name in your current directory:

```
$ ls  anotherfile newfile thirdfile
$ mkdir newdir
```

Moving Directories

To move a directory, use the mv command. The syntax is mv <directory> <destination>. In the following example, you would move the newdir subdirectory found in your current directory to the /tmp directory:

```
$ mv newdir /tmp
$ cd /tmp
$ ls
/newdir
```

The directory newdir is now a subdirectory of /tmp.

Note: *When you move a directory, all its files and subdirectories go with it.*

Removing Files

To remove (or delete) a file, use the rm command found at /bin/rm. (rm is a very terse spelling of remove). The syntax is rm <filename>. For instance:

$ rm myfile removes the file myfile from your current directory.

$ rm /tmp/myfile removes the file myfile from the /tmp directory.

$ rm * removes all files from your current directory. (Be careful when using wildcards!)

$ rm /tmp/*files removes all files ending in “files” from the /tmp directory.

Note: As soon as a file is removed, it is gone! Always think about what you're doing before you remove a file.

Use the i (interactive) option with rm:

```
$ rm -i *files    rm: remove 'myfiles'? y    rm: remove 'newfiles'? n    rm: remove 'samefiles'? y
```

Note: When you use rm -i, the command goes through the list of files to be deleted one by one, prompting you for the OK to remove the file. If you type y or Y, rm removes the file. If you type any other character, rm does not remove it. The only disadvantage of using this interactive mode is that it can be very tedious when the list of files to be removed is long.

Removing Directories

The command normally used to remove (delete) directories is rmdir. The syntax is rmdir <directory>.
Before you can remove a directory, it must be empty (the directory can't hold any files or subdirectories).
Otherwise, you see

```
rmdir: <directory>: Directory not empty
```

File Permissions and Ownership

All Linux files and directories have ownership and permissions. You can change permissions, and sometimes ownership, to provide greater or lesser access to your files and directories. File permissions also determine whether a file can be executed as a command.
If you type ls -l or dir, you see entries that look like this:

```
-rw-r—r— 1 fido users 163 Dec 7 14:31 myfile
```

The -rw-r—r— represents the permissions for the file myfile. The file's ownership includes fido as the owner and users as the group.

File and Directory Ownership

When you create a file, you are that file's owner. Being the file's owner gives you the privilege of changing the file's permissions or ownership. Of course, once you change the ownership to another user, you can't change the ownership or permissions anymore!
File owners are set up by the system during installation. Linux system files are owned by IDs such as root, uucp, and bin. Do not change the ownership of these files.

The chown command

Use the chown (change ownership) command to change ownership of a file. The syntax is chown <owner> filename>. In the following example, you change the ownership of the file myfile to root:

```
$ ls -l myfile
$ chown root myfile
$ ls -l myfile
```

To make any further changes to the file myfile, or to chown it back to fido, you must use su or log in as root.

File Permissions

Linux lets you specify read, write, and execute permissions for each of the following: the owner, the group, and "others" (everyone else).

read permission enables you to look at the file. In the case of a directory, it lets you list the directory's contents using ls.

write permission enables you to modify (or delete!) the file. In the case of a directory, you must have write permission in order to create, move, or delete files in that directory.

execute permission enables you to execute the file by typing its name. With directories, execute permission enables you to cd into them.

For a concrete example, let's look at myfile again:

rw-r—r— 1 fido users 163 Dec 7 14:31 myfile

The first character of the permissions is -, which indicates that it's an ordinary file. If this were a directory, the first character would be d.

The next nine characters are broken into three groups of three, giving permissions for owner, group, and other. Each triplet gives read, write, and execute permissions, always in that order. Permission to read is signified by an r in the first position, permission to write is shown by a w in the second position, and permission to execute is shown by an x in the third position. If the particular permission is absent, its space is filled by -.

In the case of myfile, the owner has rw-, which means read

and write permissions. This file can't be executed by typing myfile at the Linux prompt.

The group permissions are r—, which means that members of the group "users" (by default, all ordinary users on the system) can read the file but not change it or execute it.   Likewise, the permissions for all others are r—: read-only.

Read permission	4
write permission	2
execute permission	1

File permissions are often given as a three-digit number—for instance, 751. It's important to understand how the numbering system works, because these numbers are used to change a file's permissions. Also, error messages that involve permissions use these numbers.

The first digit codes permissions for the owner, the second digit codes permissions for the group, and the third digit codes permissions for other (everyone else).

The individual digits are encoded by summing up all the "allowed" permissions for that particular user as follows:

Therefore, a file permission of 751 means that the owner has read, write, and execute permission (4+2+1=7), the group has read and execute permission (4+1=5), and others have execute permission (1). If you play with the numbers, you quickly see that the permission digits can range between 0 and 7, and that for each digit in that range there's only one possible combination of read, write, and execute permissions.

Changing File Permissions

To change file permissions, use the chmod (change [file] mode) command. The syntax is chmod specification> file.

There are two ways to write the permission specification. One is by using the numeric coding system for permissions:

Suppose that you are in the home directory.

```
$ ls -l myfile
```

```
-rw-r—r— 1 fido users 114 Dec 7 14:31 myfile
$ chmod 345 myfile
```

You can also use letter codes to change the existing permissions. To specify which of the permissions to change, type u (user), g (group), o (other), or a (all). This is followed by a + to add permissions or a - to remove them. This in turn is followed by the permissions to be added or removed. For example, to add execute permissions for the group and others, you would type:

```
$ chmod go+r myfile
```

Lab Exercise(s):

1. Write a command to copy all files of current directory to /home?

2. What is the difference between the permissions 777 and 775 of the chmod command?

3. Write a command to remove all files with name containing text ‘the’ ?

4. Draw Linux Directory Structure (Tree Like structure).

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #3

Objective:
Programming using Shell Scripting.
To Create, Execute and use basic Shell Scripting.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: To Create, Execute and using basic Shell Script.

THEORY

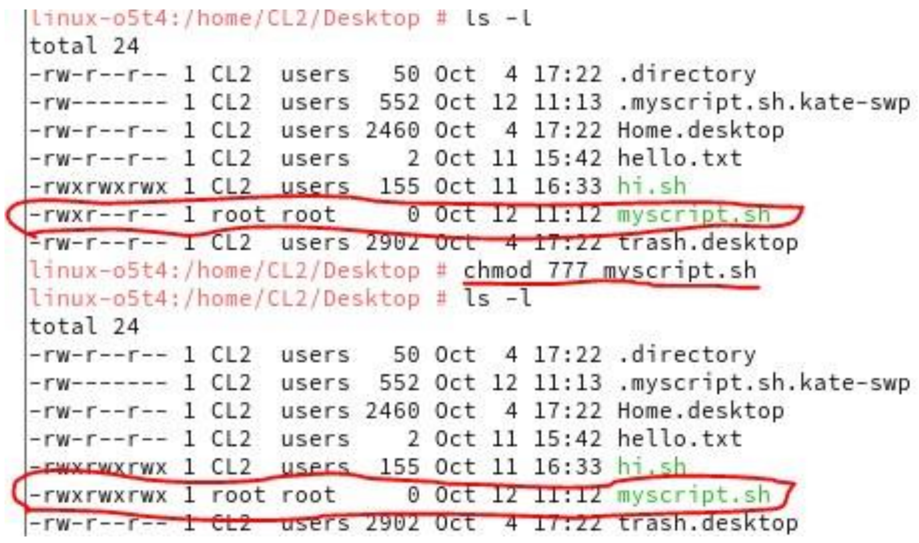
When a user enters commands from the command line, he is entering them one at a time and getting a response from the system. From time to time, it is required to execute more than one command, one after the other, and get the final result. This can be done with a ***shell program or a shell script***. A shell program is a series of Linux commands and utilities that have been put into a file by using a text editor. When a shell program is executed, the commands are interpreted and executed by Linux one after the other.

A shell program is like any other programming language and it has its own syntax. It allows the user to define variables, assign various values, and so on.

Creating and Executing a Shell Program

At the simplest level, shell programs are just files that contain one or more shell or Linux commands. These programs can be used to simplify repetitive tasks, to replace two or more commands that are always executed together with a single command, to automate the installation of other programs, and to write simple interactive applications.

Shell script is just a simple text file with ".sh" extension, having executable permission. Use chmod command to change permission.



```
linux-o5t4:/home/CL2/Desktop # ls -l
total 24
-rw-r--r-- 1 CL2  users   50 Oct  4 17:22 .directory
-rw------- 1 CL2  users  552 Oct 12 11:13 .myscript.sh.kate-swp
-rw-r--r-- 1 CL2  users 2460 Oct  4 17:22 Home.desktop
-rw-r--r-- 1 CL2  users    2 Oct 11 15:42 hello.txt
-rwxrwxrwx 1 CL2  users  155 Oct 11 16:33 hi.sh
-rwxr--r-- 1 root root     0 Oct 12 11:12 myscript.sh
-rw-r--r-- 1 CL2  users 2902 Oct  4 17:22 trash.desktop
linux-o5t4:/home/CL2/Desktop # chmod 777 myscript.sh
linux-o5t4:/home/CL2/Desktop # ls -l
total 24
-rw-r--r-- 1 CL2  users   50 Oct  4 17:22 .directory
-rw------- 1 CL2  users  552 Oct 12 11:13 .myscript.sh.kate-swp
-rw-r--r-- 1 CL2  users 2460 Oct  4 17:22 Home.desktop
-rw-r--r-- 1 CL2  users    2 Oct 11 15:42 hello.txt
-rwxrwxrwx 1 CL2  users  155 Oct 11 16:33 hi.sh
-rwxrwxrwx 1 root root     0 Oct 12 11:12 myscript.sh
-rw-r--r-- 1 CL2  users 2902 Oct  4 17:22 trash.desktop
```

Process of writing and executing a script

•Open terminal.
•Navigate to the place where you want to create script using 'cd'command.
•Cd (enter) [This will bring the prompt at Your home Directory].
•touch hello.sh (Here we named the script as hello, remember the '.sh' extension is compulsory).
•vi hello.sh (nano hello.sh) [You can use your favourite editor, to edit the script].
•chmod 744 hello.sh (making the script executable).
•sh hello.sh or ./hello.sh or bash hello.sh (running the script)

The bang line
Bang line tells the kernel that a specific shell or language is to be used to interpret the contents of the file. This is the line which is written at the beginning of every program. It starts with the bang character (#!) and goes for example like this:
#!/bin/bash
This is the link to the Bourne again shell(bash). The instructions written after this bang line will be interpreted using the above-named shell.

Writing your First Script

```
#!/bin/bash # My first script

echo "Hello World!"
```

Using Variables
Linux shell programming is full-fledged programming language and, as such, supports various types of variables. Variables have three major types:

•**Environment variables** are part of the system environment, and they do not need to be defined. They can be used in shell programs. Some of them such as PATH can also be modified within the shell program.
•**Built-in variables** are provided by the system. Unlike environment variable they cannot be modified.
•**User variables** are defined by the user when he writes the script. They can be used and modified at will within the shell program.

Environment Variables
You can use any one of the following commands to display the environment variables and their values.
printenv

Built-in Variables
These are special variables that Linux provides that can be used to make decisions in a program. Their values cannot be modified.
Several other built-in shell variables are important to know about when you are doing a lot of shell programming. The following table lists these variables and gives a brief description of what each is used for.

Variable	**Use**
$#	Stores the number of command-line arguments that were passed to the shell program.
$?	Stores the exit value of the last command that was executed.
$0	Stores the first word of the entered command (the name of the shell program).
$*	Stores all the arguments that were entered on the command line ($1 $2 ...).

"$@"	Stores all the arguments that were entered on the command line, individually quoted ("$1" "$2" ...).

A list of the commonly used variables in Linux:

o $BASH
o $USER
o $BASH_VERSION
o $PATH
o $TERM etc.

User Variables

Assigning values to variables

A value is assigned to a variable simply by typing the variable name followed by an equal sign and the value that is to be assigned to the variable. For example, if you wanted to assign a value of 5 to the variable count, you would enter the following command in bash or pdksh:

```
count=5
```

With tcsh you would have to enter the following command to achieve the same results:

```
set count = 5
```

With the bash and pdksh syntax for setting a variable, you must make sure that there are no spaces on either side of the equal sign. With tcsh, it doesn't matter if there are spaces or not.

Notice that you do not have to declare the variable as you would if you were programming in C or Pascal. This is because the shell language is a non-typed interpretive language. This means that you can use the same variable to store character strings that you use to store integers. You would store a character string into a variable in the same way that you stored the integer into a variable. For example:

name=Garry - (for pdksh and bash)

set name = Garry - (for tcsh)

Accessing variable values

To access the value stored in a variable precede the variable name with a dollar sign ($). If you wanted to print the value stored in the count variable to the screen, you would do so by entering the following command:

```
echo $count
```

If you omitted the $ from the preceding command, the echo command would display the word count onscreen.

Shell Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	\`expr $a + $b\` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	\`expr $a - $b\` will give -10
* (Multiplication)	Multiplies values on either side of the operator	\`expr $a \\* $b\` will give 200
/ (Division)	Divides left hand operand by right hand operand	\`expr $b / $a\` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	\`expr $b % $a\` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

There are many options to perform arithmetic operations on the bash shell. Some of the options are given below that we can adopt to perform arithmetic operations:

Double Parentheses

Double parentheses is the easiest mechanism to perform basic arithmetic operations in the Bash shell. We can use this method by using double brackets with or without a leading $.

Example#1

```
Sum=$((10+3))
echo "Sum = $Sum"
```

Example#2

```
x=8
y=2
echo "x=8, y=2"
echo "Addition of x & y"
echo $(( $x + $y ))
```

expr command with backticks
Arithmetic expansion could be done using backticks and expr.

Example#3

```
a=10
b=3
# there must be spaces before/after the operator
sum=`expr $a + $b`
echo $sum
```

Lab Exercise(s):

1. Write a shell program that takes one parameter (your name) and displays it on the screen.

2. Write a shell program that takes a number parameters equal to the last digit of your roll number and displays the values of the built-in variables such as $#, $0, and $* on the screen.

3. Write a Shell script to perform addition on numbers provided by command line parameters.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #4

Objective:

Test Commands and Conditional Statements.

Focusing on the usage of the test commands and conditional statements

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Focusing on the usage of the test commands and conditional statements.

THEORY

In shell scripting a command called test is used to evaluate conditional expressions. Test command is used to evaluate a condition that is used in a conditional statement or to evaluate the entrance or exit criteria for an iteration statement.

Several built-in operators can be used with the test command. These operators can be classified into four groups: integer operators, string operators, file operators, and logical operators.

The shell integer operators perform similar functions to the string operators except that they act on integer arguments. The following table lists the test command's integer operators.

The Test Command's Integer Operators

Operator	Meaning
Int1 -eq int2	Returns True if int1 is equal to int2.
Int1 -ge int2	Returns True if int1 is greater than or equal to int2.
Int1 -gt int2	Returns True if int1 is greater than int2.
Int1 -le int2	Returns True if int1 is less than or equal to int2.
Int1 -lt int2	Returns True if int1 is less than int2.
Int1 -ne int2	Returns True if int1 is not equal to int2.

The string operators are used to evaluate string expressions. The table below lists the string operators that are supported by the three shell programming languages.

The Test Command's String Operators

Operator	Meaning
Str1 = str2	Returns True if str1 is identical to str2.
Str1 != str2	Returns True if str1 is not identical to str2.
str	Returns True if str is not null.
-n str	Returns True if the length of str is greater than zero.
-z str	Returns True if the length of str is equal to zero.

The test command's file operators are used to perform functions such as checking to see if a file exists and checking to see what kind of file is passed as an argument to the test command. The following is the list of the test command's file operators.

The Test Command's File Operators

Operator	Meaning
-d filename	Returns True if file, filename is a directory.
-f filename	Returns True if file, filename is an ordinary file.
-r filename	Returns True if file, filename can be read by the process.
-s filename	Returns True if file, filename has a nonzero length.
-w filename	Returns True if file, filename can be written by the process.
-x filename	Returns True if file, filename is executable.

The test command's logical operators are used to combine two or more of the integer, string, or file operators or to negate a single integer, string, or file operator. The table below lists the test command's logical operators.

The Test Command's Logical Operators

Command	Meaning
! expr	Returns True if expr is not true.
expr1 -a expr2	Returns True if expr1 and expr2 are true.
expr1 -o expr2	Returns True if expr1 or expr2 is true.

Conditional Statements

The bash, pdksh, and tcsh each have two forms of conditional statements. These are the if statement and the case statement. These statements are used to execute different parts of your shell program depending on whether certain conditions are true. As with most statements, the syntax for these statements is slightly different between the different shells.

The if Statement

All three shells support nested if...then...else statements. These statements provide you with a way of performing complicated conditional tests in your shell programs. The syntax of the if statement is the same for bash and pdksh and is shown here:

```
if [ expression ] then commands
 elif [ expression2 ] then
commands  else
commands  fi
```

The elif and else clauses are both optional parts of the if statement.

Example:

This statement checks to see if there is a file in the current directory:

```
if [ -f “sar.txt” ] then echo "There is a .txt file in the current directory."  else echo "Could not find
the .txt file."
fi
```

The case Statement

The case statement enables you to compare a pattern with several other patterns and execute a block of code if a match is found. Once again, the syntax for the case statement is identical for bash and pdksh and different for tcsh. The syntax for bash and pdksh is the following:

```
        case string1 in str1) commands;;  str2)
        commands;;
        *) commands;
;  esac
```

Example:

The following code is an example of a bash or pdksh case statement. This code checks to see if the first command-line option was -i or -e. If it was -i, the program counts the number of lines in the file specified by the second command-line option that begins with the letter i. If the first option was -e, the program counts the number of lines in the file specified by the second command-line option that begins with the letter e. If the first command-line option was not -i or -e, the program prints a brief error message to the screen.

```
        case $1 in
        -i) count='grep ^i $2 | wc
                -l'  echo "The number of lines in $2 that start with an i is $count"
                ;;
        -e) count='grep ^e $2 | wc -l' echo "The number of lines in $2 that start with an e is $count"
                ;;
        * ) echo "That option is not recognized"
       ;;

  esac
```

Lab Exercise(s):

1. Write a script that takes two strings as input compares them and depending upon the results of the comparison prints the results.
   The user may provide input to the Bash script using:
   read var
2. Write a script that takes a number (parameter) from 1-3 as input and uses case to display the name of corresponding month.
3. Write a script that takes command-line argument for age and marks, and decide whether student is eligible for admission or not.
   Eligibility Criteria:
   Age should be lesser than 18 and marks should be greater than 700

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #5

Objective:
Loops and Functions
Focusing on the usage of iteration statements and functions in shell programming.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Focusing on the usage of iteration statements and functions in shell programming

THEORY

The shell languages also provide several iteration or looping statements. The most commonly used of these is the for statement.

The for Statement

The ‘for’ statement executes the commands that are contained within it a specified number of times. The ‘bash’ and ‘pdksh’ have two variations of the for statement.

The first form of the for statement that bash and pdksh support has the following syntax:

```
for var1 in list do commands
done
```

In this form, the for statement executes once for each item in the list. This list can be a variable that contains several words separated by spaces, or it can be a list of values that is typed directly into the statement. Each time through the loop, the variable var1 is assigned the current item in the list, until the last one is reached.
The second form of for statement has the following syntax:

```
for var1 do statements
done
```

In this form, the for statement executes once for each item in the variable var1. When this syntax of the for statement is used, the shell program assumes that the var1 variable contains all the positional parameters that were passed in to the shell program on the command line.
Typically, this form of for statement is the equivalent of writing the following for statement:

```
for var1 in "$@" do statements
done
```

The equivalent of the for statement in tcsh is called the foreach statement. It behaves in the same manner as the bash and pdksh for statement. The syntax of the foreach statement is the following:

```
foreach name (list) commands
end
```

The while Statement

Another iteration statement offered by the shell programming language is the while statement. This statement causes a block of code to be executed while a provided conditional expression is true. The syntax for the while statement in bash and pdksh is the following:

```
while expression do statements
done
```

The syntax for the while statement in tcsh is the following:

```
while (expression) statements
end
```

Care must be taken with the while statements because the loop will never terminate if the specifies condition never evaluates to false.

BASH while Loop Example

```
#!/bin/bash
 c=1
while [ $c -le 5 ]
do
 echo "Welcone $c times"
((counter++))
done
```

The shift Command

bash, pdksh, and tcsh all support a command called shift. The shift command moves the current values stored in the positional parameters to the left one position. For example, if the values of the current positional parameters are

```
$1 = -r $2 = file1 $3 = file2
```

and you executed the shift command

```
shift
```

the resulting positional parameters would be as follows:

```
$1 = file1 $2 = file2
```

The break Statement

The break statement can be used to terminate an iteration loop, such as a for, until or repeat command.

```
#!/bin/bash #breaking a loop num=1
while [ $num –lt 10 ] do
if [ $num –eq 5 ] then break
fi
done echo “Loop is complete”
```

The exit Statement

The exit statement can be used to exit a shell program. A number can be optionally used after exit. If the current shell program has been called by another shell program, the calling program can check for the code and make a decision accordingly.

Functions

The shell languages enable you to define your own functions. These functions behave in much the same way as functions you define in C or other programming languages. The main advantage of using functions as opposed to writing all of your shell code in line is for organizational purposes. Code written using functions tends to be much easier to read and maintain and also tends to be smaller, because you can group common code into functions instead of putting it everywhere it is needed. The syntax for creating a function in bash and pdksh is the following:

```
fname ()
{ shell commands
}
```

Once you have defined your function, you can invoke it by entering the following command:

```
fname [parm1 parm2 parm3 ...]
```

Shell functions have their own command line argument.
Use variable $1, $2..$n to access argument passed to the function.

The syntax is as follows:

```
name()
{   arg1=$1
    arg2=$2
}
```

To invoke the function, use the following syntax:
name hello dear
Where,
name = function name.
hello = Argument # 1 passed to the function (positional parameter # 1). *dear* = Argument # 2 passed to the function.

Lab Exercise(s):

1. Write a script that creates a backup version of each file in your home directory to a subdirectory called backup using **for** statement. If the operation fails an error message is to be displayed.

2. Write a script that calculates the average of all even numbers less than or equal to your roll number and prints the result.

3. Write a function that displays the name of the week days starting from Sunday if the user passes a day number. If a number provided is not between 1 and 7 an error message is displayed.

4. Write scripts that displays the parameters passed along with the parameter number using while and until statements.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #6

Objective:
Processes in Operating Systems
Understanding Process, Process Creation, Zombie and Orphan Processes

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Understanding Process, Process Creation, Zombie and Orphan Processes.

THEORY

Everything that runs on a Linux system is a process—every user task, every system daemon— is a process. Knowing how to manage the processes running on your Linux system is an important (indeed even critical) aspect of system administration.

There are several definitions of the term process, including:

- A program in execution.
- An instance of Program running on a computer.
- The entity that can be assigned to and executed on a processor.
- A unit of activity with an associated set of system resources.

Using ps command

The easiest method of finding out what processes are running on your system is to use the ps (process status) command. The ps command has a number of options and arguments, although most system administrators use only a couple of common command-line formats. We can start by looking at the basic usage of the ps command, and then examine some of the useful options. The ps command is available to all system users, as well as root, although the output changes a little depending on whether you are logged in as root when you issue the command.

For example, you might see the following output when you issue the command:

```
$ ps
PID TTY STAT TIME COMMAND
41   v01      S        0:00 -bash
134 v01      R        0:00 ps
```

Result contains four columns of information.
Where,
PID – the unique process ID
TTY – terminal type that the user is logged into
TIME – amount of CPU in minutes and seconds that the process has been running
CMD – name of the command that launched the process.

To obtain much more complete information about the processes currently on the system.
ps -aux

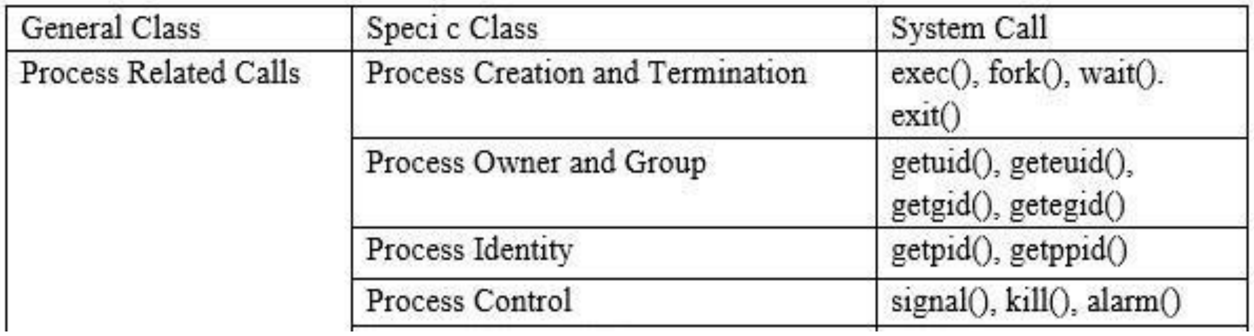
-To display a tree of processes*.* ***pstree***

-To see currently running processes and other information like memory and CPU usage with real time updates. ***top***
The -e option generates a list of information about every process currently running.
The -f option generates a listing that contains fewer items of information for each process than the -l option. - ps -ef | less

Note: For more understanding and to know about more options for ps command go to man page.
"man ps"
Process Creation

A “parent process" is a process that has created one or more child processes. In UNIX, every process except process 0 is created when another process executes the fork system call. The process that invoked fork is the parent process and the newly created process is the “child process". Every process except process 0) has one parent process, but can have many child processes. The kernel identifies each process by its process identifier (PID). Process 0 is a special process that is created when the system boots. Process 1, known as init, is the ancestor of every other process in the system.

When a child process terminates execution, either by calling the exit system call, causing a fatal execution error, or receiving a terminating signal, an exit status is returned to the operating system. A parent will typically retrieve its child's exit status by calling the wait system call. However, if a parent does not do so, the child process becomes a “zombie process." Following table illustrates various process system calls.



General Class	Speci c Class	System Call
Process Related Calls	Process Creation and Termination	exec(), fork(), wait(). exit()
	Process Owner and Group	getuid(), geteuid(), getgid(), getegid()
	Process Identity	getpid(), getppid()
	Process Control	signal(), kill(), alarm()

The fork() System Call

Fork system call use for creates a new process, which is called *child process*, which runs concurrently with process (which process called system call fork) and this process is called *parent process*. After a new child process 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.

fork() creates a new process by duplicating the calling process. The new process, referred to as t child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group.
- The child's parent process ID is the same as the parent's process ID.

Predict the Output of the following programs to understand fork() Program1:

```
# importing os module import os

os.fork()
print("I am  process:")
```

Program2:

```
# importing os module import os

os.fork()
os.fork()
os.fork()
print("I am  process:")
```

Wait() Method Of Python Os Module

The wait() method of os module in Python enables a parent process to synchronize with the child process. i.e, to wait till the child process exits and then proceed.

Example Program:

```
# import the os module of Python

import os
# Create a child process retVal = os.fork()
   if retVal is 0:
         print("Child process Running………..”)
                 print("Child process Running………..”)
                 print("Child process Running………..”)
                 print("Child process %d exiting"%(os.getpid()))
  else:
   # Parent process waiting
       childProcExitInfo = os.wait()
       print("Child process %d exited"%(childProcExitInfo[0]))
       print("Now Parent process Running….")
```

Orphan and Zombie Process State

Zombie Processes

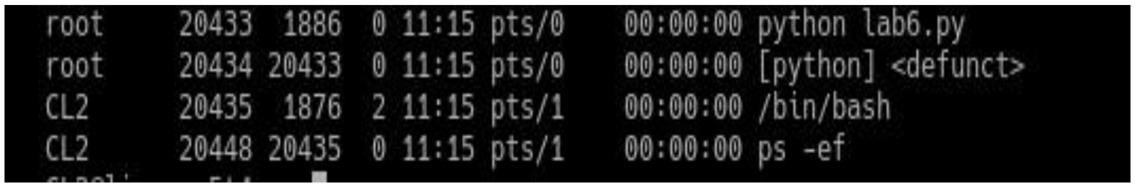
On Unix and Unix-like computer operating systems, a \zombie process" or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie an undead person. When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, whereupon the zombie is removed.

To observe the creation of Zombie Process, Python sleep() call will be used to simulate a delay in the parent process.

Example Program:

```
# Create a child process import os import time retval=os.fork()

if retval == 0 :

    print("Child process Runnin")
         print("Child process Running")
         print("Child process Running")
         print("Child process exiting")
else:
    # Parent process did not use wait()
    time.sleep(60)
    print("Now Parent process Running")
```

Zombie or defunct process can be seen in another instance of the terminal by using ps -ef, just after running the above program.


```
root     20433  1886  0 11:15 pts/0    00:00:00 python lab6.py
root     20434 20433  0 11:15 pts/0    00:00:00 [python] <defunct>
CL2      20435  1876  2 11:15 pts/1    00:00:00 /bin/bash
CL2      20448 20435  0 11:15 pts/1    00:00:00 ps -ef
```

Note: *The Process with status as Z or marked as <defunct> i.e., de-functioning is ZOMBIE Process.*

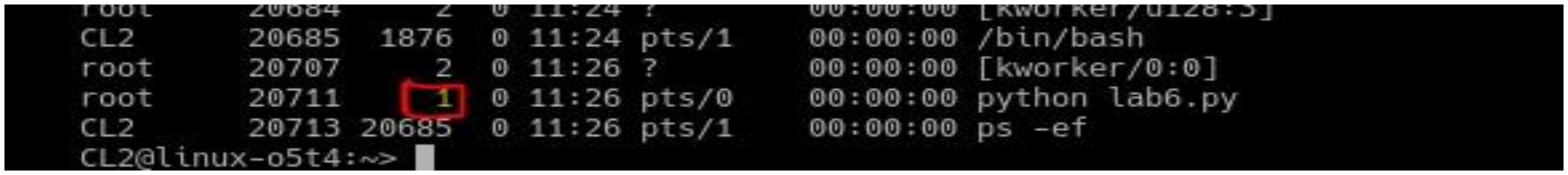
Orphan Processes

An “orphan process" is a computer process whose parent process has finished or terminated, though it remains running itself. In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called reparenting and occurs automatically. Even though technically the process has the \init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists. A process can be orphaned unintentionally, such as when the parent process terminates or crashes.

Example Program:

```
import os import time

retval=os.fork()

if retval == 0 :

   print("Child process Runnin")

        print("Child process Running")

        print("Child process Running")

        print("Child process Going To Sleep for 60 seconds")

        time.sleep(60)

        print("Child process Running after a nap ")

   print("Child process exiting")
else:
   # Parent process Running
   print("Parent process Running and exiting.....")
```

To observe the creation of Orphan Process, Python sleep() call will be used to simulate a delay in the child process. Parent id of this orphan process i.e 1 , can be seen in another instance of the terminal by using ps -ef, just after running the above program.


```
    root      20684      2  0 11:24 ?        00:00:00 [kworker/u128:3]
    CL2       20685   1876  0 11:24 pts/1    00:00:00 /bin/bash
    root      20707      2  0 11:26 ?        00:00:00 [kworker/0:0]
    root      20711      1  0 11:26 pts/0    00:00:00 python lab6.py
    CL2       20713 20685  0 11:26 pts/1    00:00:00 ps -ef
    CL2@linux-o5t4:~>
```

Compile and Run Python program

```
python lab6.py
```

Lab Exercise(s):

1. Using either a Linux system, write a program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds.

2. Write a program that creates a child process which further creates its two child processes. Store the process id of each process in an array called Created Processes. Also display the process id of the terminated child to understand the hierarchy of termination of each child process.

3. Write a program in which a parent process will initialize an array, and child process will sort this array. Use wait() and sleep() methods to achieve the synchronization such that parent process should run first.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #7

Objective:
Threads in Operating Systems
Understanding Threads, Threads Creation, Multithreaded Programming using Python

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Understanding Threads, Threads Creation, Multithreaded Programming using Python & C.

THEORY

Thread is the smallest unit of processing. It is scheduled by an OS. In general, it is contained in a process. So, multiple threads can exist within the same process. It shares the resources with the process. Each thread belongs to exactly one process and no thread can exist outside a process.

A thread uses the same address space of a process. A process can have multiple threads. A key difference between processes and threads is that multiple threads share parts of their state. Typically, multiple threads can read from and write to the same memory (no process can directly access the memory of another process). However, each thread still has its own stack of activation records and its own copy of CPU registers, including the stack pointer and the program counter, which together describe the state of the thread's execution. A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads. Processes are independent while thread is within a process. Processes have separate address spaces while threads share their address spaces. Multithreading has some advantages over multiple processes. Threads require less overhead to manage than processes, and intra-process thread communication is less expensive than inter-process communication.

Types of Threads:

User Level thread (ULT) –

User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

Kernel Level Thread (KLT) –

Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

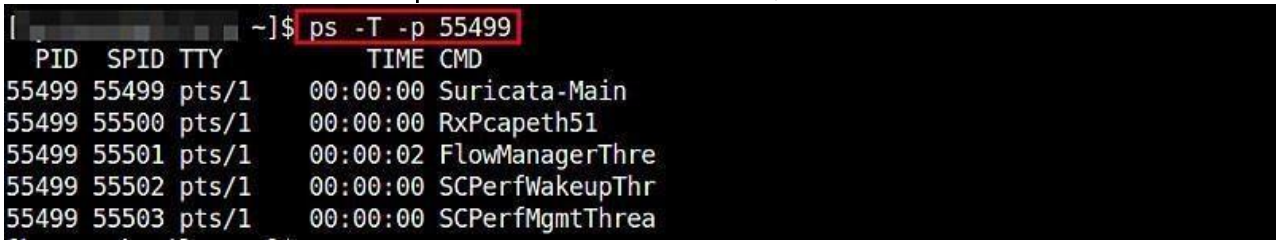
Viewing threads of a process/processes on Linux

Ps Command

In ps command, "-T" option enables thread views. The following command list all threads created by a process with <pid>.

```
ps -T -p <pid>
```

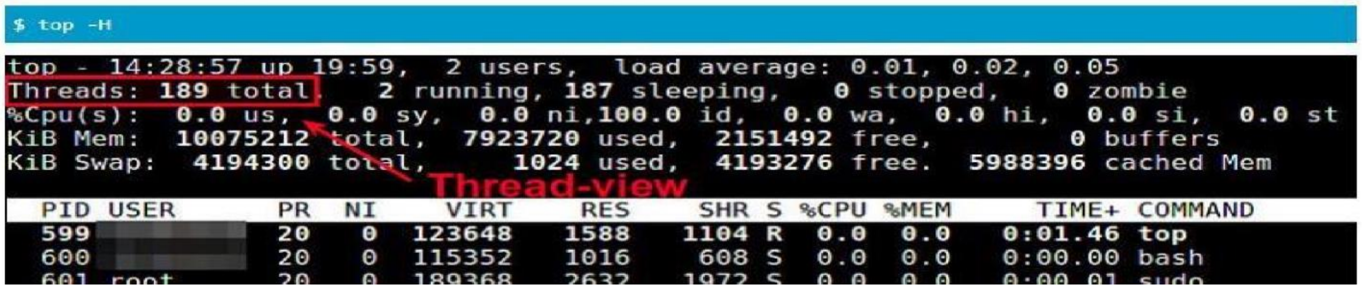
The "SID" column represents thread IDs, and "CMD" column shows thread names.



```
[[redacted] ~]$ ps -T -p 55499
  PID  SPID TTY          TIME CMD
55499 55499 pts/1    00:00:00 Suricata-Main
55499 55500 pts/1    00:00:00 RxPcapeth51
55499 55501 pts/1    00:00:02 FlowManagerThre
55499 55502 pts/1    00:00:00 SCPerfWakeupThr
55499 55503 pts/1    00:00:00 SCPerfMgmtThrea
```

Top Command

The top command can show a real-time view of individual threads. To enable thread views in the top output, invoke top with "-H" option. This will list all Linux threads. You can also toggle on or off thread view mode while top is running, by pressing 'H' key.



```
$ top -H

top - 14:28:57 up 19:59,  2 users,  load average: 0.01, 0.02, 0.05
Threads: 189 total,   2 running, 187 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  10075212 total,  7923720 used,  2151492 free,        0 buffers
KiB Swap:  4194300 total,     1024 used,  4193276 free.  5988396 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  599 [redacted] 20   0  123648   1588   1104 R  0.0  0.0   0:01.46 top
  600 [redacted] 20   0  115352   1016    608 S  0.0  0.0   0:00.00 bash
  601 root      20   0  189368   2632   1972 S  0.0  0.0   0:00.01 sudo
```

***Command-line tools such as ps or top, which display process-level information by default, can be instructed in many ways to display thread-level information.**

THREAD CREATION in Python- (Multithreaded Programming):

There are two modules which support the usage of threads in Python3 − _thread , Threading

The Threading Module

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module

the threading module has the Thread class that implements threading. The methods provided by the Thread class are as follows –

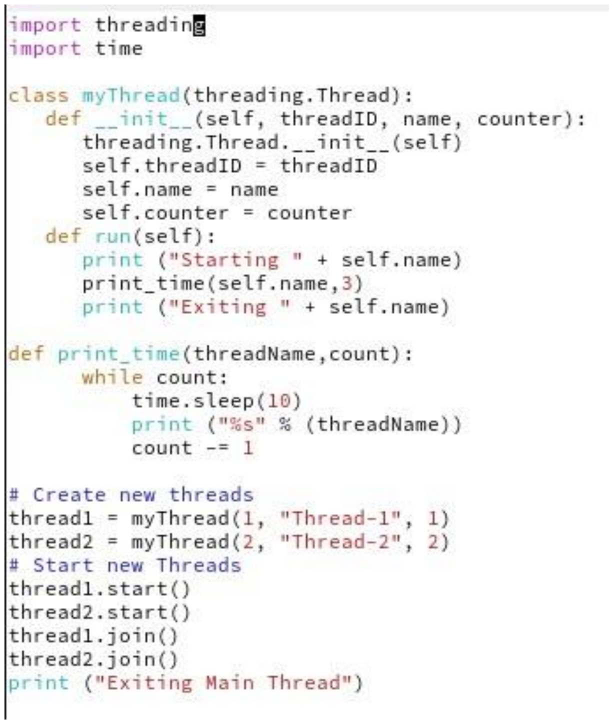
- run() − The run() method is the entry point for a thread.
- start() − The start() method starts a thread by calling the run method.
- join([time]) − The join() waits for threads to terminate.
- isAlive() − The isAlive() method checks whether a thread is still executing.
- getName() − The getName() method returns the name of a thread.
- setName() − The setName() method sets the name of a thread.

To implement a new thread using the threading module, you have to do the following − Define a new subclass of the Thread class.

Override the __init__(self [,args]) method to add additional arguments.

Then, override the run(self [,args]) method to implement what the thread should do when started. Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the start(), which in turn calls the run() method.

Example Program:



```python
import threading
import time

class myThread(threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print ("Starting " + self.name)
      print_time(self.name,3)
      print ("Exiting " + self.name)

def print_time(threadName,count):
      while count:
          time.sleep(10)
          print ("%s" % (threadName))
          count -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("Exiting Main Thread")
```

Limitations and difficulty in using Python for Multithreading

GIL (global interpreter lock), the mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. Due to the GIL only one thread can be executed at a time. Therefore, the above code is concurrent but not parallel. Multi-Threading in python can be considered as an appropriate model only if you want to run multiple I/O-bound tasks simultaneously.

Using the threading module in Python or any other interpreted language with a GIL can actually result in reduced performance. If your code is performing a CPU bound task, using the threading module will result in a slower execution time. For CPU bound tasks and truly parallel execution, we can use the multiprocessing module instead of multithreading in Python.

Thread Creation in C

Normally when a program starts up and becomes a process, it starts with a default thread.
So, we can say that every process has at least one thread of control.
C does not contain any built-in support for multithreaded applications, so POSIX (pthreads) can be used to write multi-threaded C program. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

The following routine is used to create a POSIX thread –

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

Parameter	Description
thread	An opaque, unique identifier (pthread_t type address) for the new thread returned by the subroutine. So the first argument will hold the thread ID of the newly created thread.
attr	An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
start_routine	The third argument is a function pointer. The C routine that the thread will execute once it is created. This is something to keep in mind that each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread f rom.
arg	A single argument that may be passed to start_routine. As the function (whose address is passed in the third argument above) may accept some arguments also so we can pass these arguments in form of a pointer to a void type. Now, why a void type was chosen? This was because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

Example C program:

```
#include <stdio.h>
#include <pthread.h> #include <stdlib.h>
void *print_message_function( void *ptr );  int main()
{
 int status;
 char *msg1 = "Thread 1";
char *msg2 = "Thread 2";
```

```
        pthread_t tid1,tid2;
         pthread_create(&tid1,NULL,myfunc,(void*)msg1);
         pthread_create(&tid2,NULL,myfunc,(void*)msg2);
         pthread_join(tid1,NULL);   pthread_join(tid2,NULL);
         return 0;
        }
        void *myfunc ( void *ptr )
        {
        char *message;  message = (char *) ptr;   for
        (int i=0; i<10;i++)
        {   printf("%s  %d\n", message,i);
        Sleep(1); }
        Return 0;
}
```

Compile C 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.

gcc -o hello hello.c -lpthread

This command will invoke the GNU C compiler to compile the file hello.c and output (-o) the result to an executable called hello **Execute the program.**

Type the command

```
./hello
```

This should result in the output

```
Hello World
```

Lab Exercise(s):

1. Modify Example 1 to display strings via two independent threads:
   thread1: "Hello ! StudentName___", thread 2: "Student roll no is :__________"

2. Create threads message as many times as user wants to create threads by using array of threads and loop. Threads should display message that is passed through argument.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #8

Objective:
CPU Scheduling – FCFS and SJF
Implementation of CPU scheduling algorithms; FCFS and SJF.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Implementation of CPU scheduling algorithms; FCFS and SJF.

THEORY

FCFS CPU SCHEDULING ALGORITHM

First Come First Serve is a Non-preemptive Scheduling algorithm where each process is executed according to its arrival time.

Step 1 : Input the number of processes required to be scheduled using FCFS, burst time for each process and its arrival time.

Step 2 : Using enhanced bubble sort technique, sort the all given processes in ascending order according to arrival time in a ready queue.

Step 3 : Calculate the Finish Time, Turn Around Time and Waiting Time for each process which in turn help to calculate Average Waiting Time and Average Turn Around Time required by CPU to schedule given set of process using FCFS.

Step 4 : Process with less arrival time comes first and gets scheduled first by the CPU.

Step 5 : Calculate the Average Waiting Time and Average Turn Around Time.

Step 6 : Stop.

Sample Run:

```
Enter total number of processes (maximum 20): 3
Enter Process Arrival Time
  P[1]:0
  P[2]:0
  P[3]:0
Enter Process Burst Time
P[1]:24
P[2]:3
P[3]:3
        Process      Burst Time    Waiting Time  Turnaround Time
        P[1]    24     0      24
        P[2]    3      24     27
        P[3]    3      27     30
  Average Waiting Time:17
Average Turnaround Time:27
```

SJF CPU SCHEDULING ALGORITHM

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

Step 1 : Input the number of processes required to be scheduled using SJF, burst time for each process and its arrival time.

Step 2 : Using selection sort technique, sort the all given processes in ascending order according to burst time in ascending order.

Step 3 : Calculate the Finish Time, Turn Around Time and Waiting Time for each process which in turn help to calculate Average Waiting Time and Average Turn Around Time required by CPU to schedule given set of process

Step 4 : Process with less Burst and arrival time comes first and gets scheduled first by the CPU.

Step 5 : Calculate the Average Waiting Time and Average Turn Around Time.

Step 6 : Stop

Sample Run:

```
Enter number of process: 4
Enter Burst Time:  p1:4  p2:8  p3:3 p4:7
Enter Process Arrival Time
   P[1]:0
   P[2]:0
   P[3]:0
P[4]:0

Process         Burst Time    Waiting Time  Turnaround Time
p3      3                                0           3
p1      4                                3           7
p4      7                                7          14
p2      8                               14          22

Average Waiting Time=6.000000
Average Turnaround Time=11.500000
```

Terms and formulas used in above scheduling algorithms:

Completion Time: Time at which process completes its execution.

Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): Time Difference between turnaround time and burst time.

Waiting Time = Turn Around Time – Burst Time

Lab Exercise(s):

1. Write a Python program to implement and simulate the FCFS Algorithm.
2. Write a Python program to implement and simulate the SJF Algorithm. Modify both algorithms for the different arrival time.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #9

Objective:

CPU Scheduling – Round Robin and Priority

: Implementation of CPU scheduling algorithms; Priority and Round Robin.

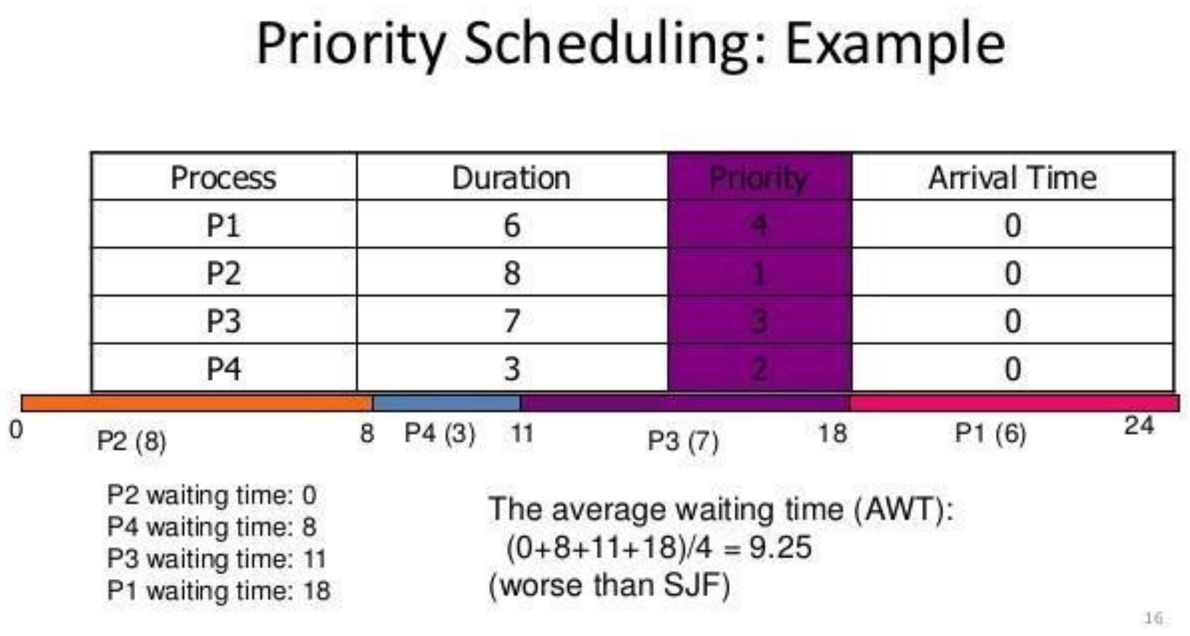
Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Implementation of CPU scheduling algorithms; Priority and Round Robin.

THEORY

PRIORITY SCHEDULING ALGORITHM

In priority scheduling algorithm each process has a priority associated with it and as each process hits the queue, it is stored in based on its priority so that process with higher priority is dealt first. It should be noted that equal priority processes are scheduled in FCFS order.



Priority Scheduling: Example

Process	Duration	Priority	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

P2 waiting time: 0
P4 waiting time: 8
P3 waiting time: 11
P1 waiting time: 18

The average waiting time (AWT):
(0+8+11+18)/4 = 9.25
(worse than SJF)

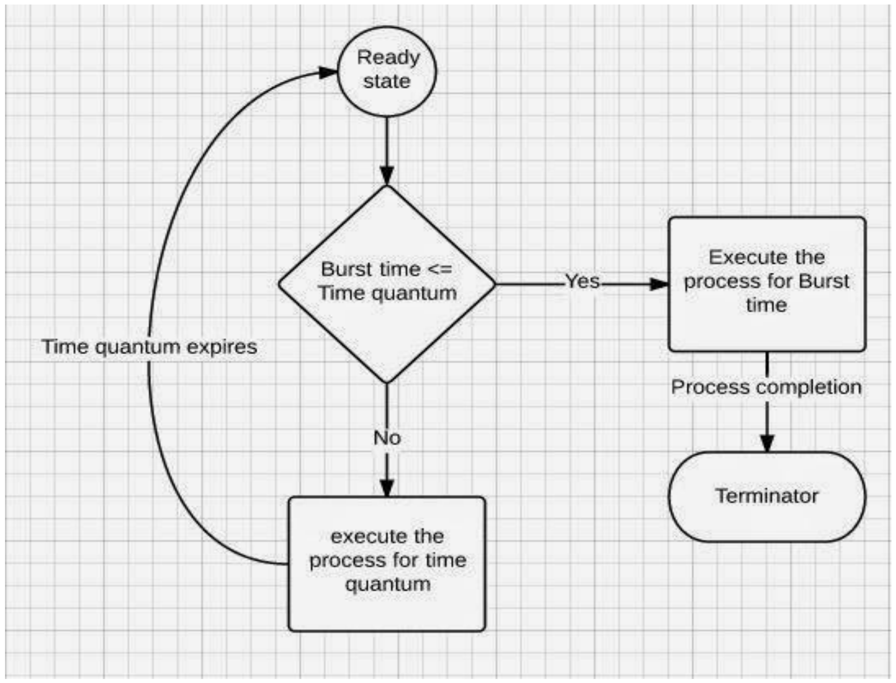
Implementation –

1. First input the processes with their arrival time, burst time and priority.
2. Sort the processes, according to arrival time if two process arrival time is same then sort according process priority if two process priority are same then sort according to process number.
3. Now simply apply FCFS algorithm.

   Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

Round Robin Scheduling Algorithm

Round Robin scheduling algorithm is one of the most popular scheduling algorithms which can actually be implemented in most of the operating systems. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a **cyclic way**. A certain time slice is defined in the system which is called time **quantum**. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time, then the process will **terminate** else the process will go back to the **ready queue** and waits for the next turn to complete the execution.


Ready state
Burst time <= Time quantum
Yes
Execute the process for Burst time
Process completion
Terminator
No
execute the process for time quantum
Time quantum expires

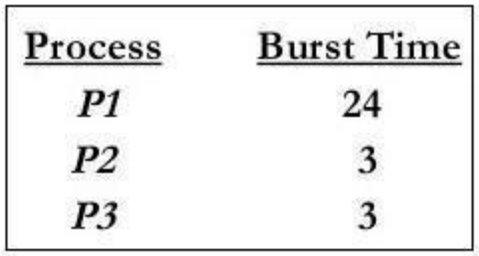
Implementation

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

Steps to find Completion times of all processes:

```
1-   Create an array rem_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)
2-   Create another array ct[] to store completion times of processes. Initialize this array as 0.
3-   Initialize time : t = 0
4-   Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.
     a- If rem_bt[i] > quantum
        (i)   t = t + quantum
        (ii)  bt_rem[i] -= quantum;
     c- Else // Last cycle for this process
        (i)   t = t + bt_rem[i];
        (ii)  ct[i]=t;
        (iii) bt_rem[i] = 0; // This process is over
```

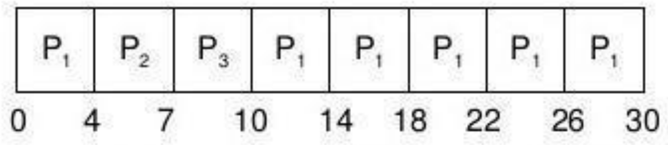
Round Robin Scheduling Example



Process	Burst Time
P1	24
P2	3
P3	3

Quantum time = 4 milliseconds

The Gantt chart is:



Average waiting time = {[0+(10-4)]+4+7}/3 = 5.6

Lab Exercise(s):

1. Write a Python program to implement and simulate the Priority Algorithm.
2. Write a Python program to implement and simulate the Round Robin Algorithm.
3. Modify both algorithms for the different arrival time.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #10

Objective:

Semaphore Mechanism in Operating Systems

To Implementation of a Classical problem (Producer-Consumer) using semaphores.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Implementation of a Classical problem (Producer-Consumer) using semaphores.

THEORY

Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi-processing environment.
The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only. A semaphore can only be accessed using the following operations: wait() and signal().

```
wait(Semaphore s) {
    while (s==0); /* wait until s>0 */
    s=s-1;
}
signal(Semaphore s){
    s=s+1;
}
```

[***In python, acquire() and release() provide wait() and signal() functionality, respectively]***

Python's Simple Lock *using class threading.Lock*
A simple mutual exclusion lock used to limit access to one thread. This is a semaphore with s = 1. **acquire()**
Obtain a lock. The process is blocked until the lock is available.
release()
Release the lock and if another thread is waiting for the lock, wake up that thread.

Python's Semaphore
using class threading.Semaphore(s)
acquire()
Obtain a semaphore. The process is blocked until the semaphore is available.
release()
Release the semaphore and if another thread is waiting for the semaphore, wake up that thread.

Python: Producer-Consumer Solution using Semaphores

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n (n =5 in this example); the semaphore full is initialized to the value 0.

```
import threading import random
import time

buf = []
empty = threading.Semaphore(5)
full = threading.Semaphore(0)
```

```
mutex = threading.Lock()

def producer():
    nums = range(5)
    global buf

    num = random.choice(nums)
    empty.acquire()
    mutex.acquire()        # added
    buf.append(num)
    print("Produced", num, buf)
    mutex.release()        # added
    full.release()
    time.sleep(1)

def consumer():

    global buf
    full.acquire()
    mutex.acquire()        # added
    num = buf.pop(0)
    print("Consumed", num, buf)
    mutex.release()        # added
    empty.release()
    time.sleep(2)

producerThread1 = threading.Thread(target=producer)
consumerThread1 = threading.Thread(target=consumer)
producerThread2 = threading.Thread(target=producer)
consumerThread2 = threading.Thread(target=consumer)

consumerThread1.start()
consumerThread2.start()
producerThread1.start()
producerThread2.start()
```

Lab Exercise(s):

1. Write a python program that demonstrates the synchronization of Readers and Writer Problem using semaphores.
2. Write a python program that demonstrates the synchronization of Consumer producer Bounded Buffer Problem using semaphores.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #11

Objective:
Inter process communication (IPC) using Pipe
Multiprocessing in Python, Implement Pipe using ***os*** and ***multiprocessing*** module

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Multiprocessing in Python, Implement Pipe using os and multiprocessing module in Python.

THEORY

Inter process communication (IPC)

Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions.
There are several different ways to implement IPC, for example pipe, shared memory, message passing. IPC is set of programming interfaces, used by programs to communicate between series of processes.

Pipe

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe.

Two-way communication using pipes

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child need to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.
os.pipe() method in Python is used to create a pipe. A pipe is a method to pass information from one process to another process. It offers only one-way communication and the passed information is held by the system until it is read by the receiving process. os.pipe(), Return a pair of file descriptors (r, w) usable for reading and writing, respectively.

Just like other programming languages, Python also supports the implementation of pipes.

```
import os
 r, w = os.pipe()
pid = os.fork()
if pid > 0:

    os.close(r)
    print("Parent process is writing")
    text = "Hello child process"
    os.write(w, text)

else:
     os.close(w)
    # Read the text written by parent process
     print("\nChild Process is reading")
     r = os.fdopen(r)
    print("Read text:", r.read())
```

os.fdopen(fd, *args, **kwargs)

Return an open file object connected to the file descriptor **fd.**

A file descriptor (FD) is a small non-negative integer that helps in identifying an open file within a process while using input/output resources like network sockets or pipes.

Python method write() writes the string str to file descriptor fd. Return the number of bytes actually written.

Syntax

Following is the syntax for write() method : **os.write(fd, str)**

Multiprocessing

Multiprocessing is the use of two or more central processing units (CPUs) within a single computer

system”. Multiprocessing can significantly increase the performance of the program.

In Python , multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

Exchanging objects between processes

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use Pipe() (for a connection between two processes) or a queue (which allows multiple producers and consumers).

multiprocessing.Pipe([duplex])

Returns a pair (conn1, conn2) of Connection objects representing the ends of a pipe.

If duplex is True (the default) then the pipe is bidirectional. If duplex is False then the pipe is unidirectional

The Pipe() function returns a pair of connection objects connected by a pipe which by default is duplex (two-way).

For example:

```
from multiprocessing import Process, Pipe

def f(conn):

conn.send([30, None, 'hello Students'])

conn.close()

if __name__ == '__main__':

parent_conn, child_conn = Pipe()

p = Process(target=f, args=(child_conn,))

p.start()

print(parent_conn.recv())

p.join()
```

The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() and recv() methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the same end of the pipe at the same time. Of course, there is no risk of corruption from processes using different ends of the pipe at the same time.

Lab Exercise(s):

1. Modify Example code to pass your name from parent to child process and display it by concatenating with your roll number and department. Take your name as user input and pass it as command line argument.

2. Write a Python program in which child process put the square of each number from 1 to 5 (exclusive) on pipe. After the child process has finished its execution, we get the data from the pipe in parent process.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #12

Objective:
Inter process communication (IPC) using Shared Memory
To Implement Shared Memory using multiprocessing module in Python

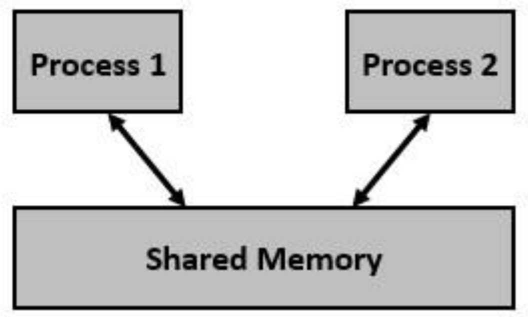
Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: Implement Shared Memory using multiprocessing module in Python.

THEORY

Inter process communication (IPC)
Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions.
There are several different ways to implement IPC, for example pipe, shared memory, message passing. IPC is set of programming interfaces, used by programs to communicate between series of processes.

Shared memory
Shared memory is a memory shared between two or more processes.
Shared memory is the fastest inter-process communication mechanism. The operating system maps a memory segment in the address space of several processes, so that several processes can read and write in that memory segment without calling operating system functions.



Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.

Implementation of Shared memory using Python
It is possible to create shared objects using shared memory which can be inherited by child processes. Data can be stored in a shared memory map using Value or Array.

multiprocessing.Value(typecode_or_type, *args, lock=True)

Return a **ctypes object allocated from shared memory. By default, the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the value attribute of a Value. typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module. *args is passed on to the constructor for the type.

If lock is True (the default) then a new recursive lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

multiprocessing.Array(typecode_or_type, size_or_initializer, *, lock=True)
Return a **ctypes array allocated from shared memory. By default, the return value is actually a synchronized wrapper for the array.

If *size_or_initializer* is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

** ctypes is *a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.*

Example#1: Consider following example code to store data in shared memory using multiprocessing.Value:

```
from multiprocessing import Process, Value

def f(n):
    n.value = 3.1415927

if __name__ == '__main__':
    num = Value('d', 0.0)
   p   =   Process(target=f,   args=(num,))
   p.start()
    p.join()
print(num.value)
```

Example#2: Consider following example code to store data in shared memory using multiprocessing.Array:

```
from    multiprocessing   import   Process,
Array
def f(a):
for  i in range(len(a)):

        a[i] = -a[i]

if __name__ == '__main__':
arr = Array('i', range(10))
p = Process(target=f, args=(arr))
p.start()
p.join()
print(arr[:])
```

The 'd' and 'i' arguments used when creating num and arr are typecodes of the kind used by the array module: 'd' indicates a double precision float and 'i' indicates a signed integer.

Lab Exercise(s):

1. Start five processes using multiprocessing.Process objects , each process will update shared memory Value object using their own target function (callable object to be invoked by the run() method). After execution of all child processes, parent process should display the value of the object.

2. Generate 10 random numbers between 0 and 10, and calculate square of each number such that process#1 calculates square of first five numbers and process#2 calculates square of remaining five numbers, Store the square results in an array (shared memory region) using multiprocessing module.

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #13

Objective:
Deadlock Avoidance
To Implement the Bankers algorithm for the purpose of deadlock avoidance.

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: To Implement the Bankers algorithm for the purpose of deadlock avoidance.

THEORY

Deadlock

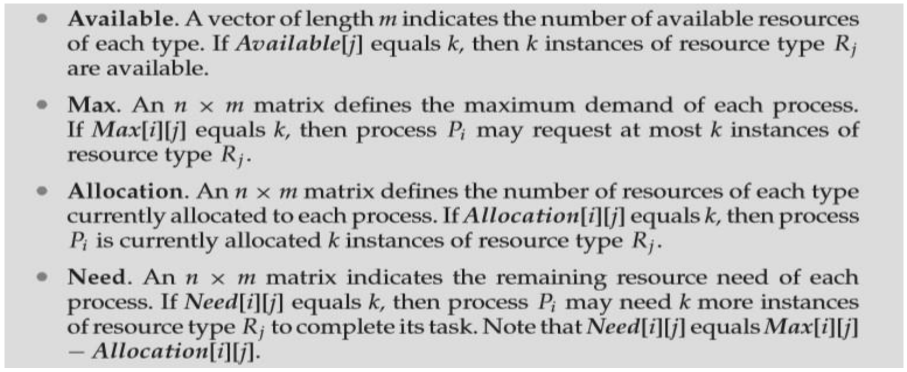
Deadlock (DL) can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. In a multiprogramming environment, several processes may compete for a finite number of resources.
A process requests resources; if the resources are not available at that time, the process enters a wait state. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the event is ever triggered.

Deadlock Avoidance

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:



- **Available**. A vector of length m indicates the number of available resources of each type. If $Available[j]$ equals k, then k instances of resource type R_j are available.
- **Max**. An $n \times m$ matrix defines the maximum demand of each process. If $Max[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.
- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i][j]$ equals $Max[i][j] - Allocation[i][j]$.

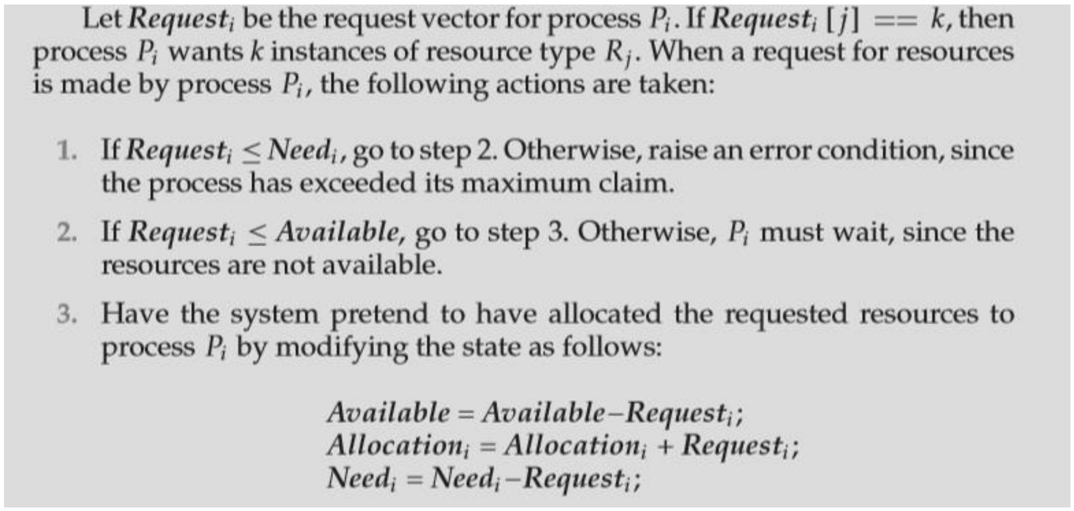
Safety Algorithm:

Algorithm for finding out whether or not a system is in a safe state Let Work and Finish be vectors of length m and n, respectively.
Initialize Work =Available and Finish[i]=false for i=0,1,...,n−1.
Find an index i such that both

1. Finish[i] ==false
2. Needi ≤Work If no such i exists, goto step4.
3. Work =Work + Allocationi , Finish[i]=true , Go to step 2.
4. If Finish[i] ==true for all i, then the system is in a safe state.

Resource-Request Algorithm:

The algorithm for determining whether requests can be safely granted.



Let $Request_i$ be the request vector for process P_i. If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Request i, and the old resource allocation state is restored.

Lab Exercise(s):

1- Write a Python Program to implement Banker's Algorithm (Safety Algorithm) for a snapshot (as per the inputs):

Inputs

- Number of processes
- Number of resources
- Available quantity of each resource after allocation of resources.
- Allocated quantity of each resource assigned to each process.
- Available resources or total resources

Output

Identify the state of system (safe/un-safe).

2- Write a Bash script that takes user input for number of processes and number of resources and passes them into above python script (required in Task#1) to find the safety sequence. Assuming that other inputs are provided inside Python Program.

3. Write a Python Program to implement Banker's Algorithm (Resource Request Algorithm) to determine whether requests can be safely granted or not?

Usman Institute of Technology

Department of Computer Science

Fall 2022

CS312 – Operating Systems

Lab #14

Objective:
Open Ended LAB
To construct GUI based script that behaves like an Operating System

Name of Student	
Student ID	
Date of Lab Conducted	
Marks Obtained	
Remarks	
Signature	

Objective: To construct GUI based script that behaves like an Operating System.

Requirements:

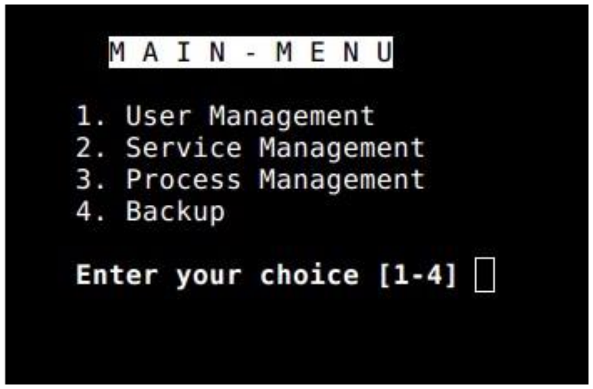
Your Operating System should have the following properties.

1. It allows you to create folders and files.
2. It allows you to change rights of your files.
3. It can help you in searching files.
4. It allows you to create processes and threads that perform specific tasks
   For example
   -Create a process that sorts array in ascending order.
   -Create multiple threads that may help in solving matrix operations etc.
5. It allows you to display processes like a task manager in Windows and should allow to kill any selected process.
6. Allows to open applications like Firefox, Image viewer etc.
7. Allows to share data between processes such that output of one process becomes input of another. Provide sub menu to select Process1 and Process2 that gives input or input or vice versa.
8. Allows to write Linux C/Python programs to create your own process that can perform any desirable task. Provide options in user sub-menu to execute and delete that program.

Use dialog boxes, file browsers, and other common "windowing" controls and techniques to interact with your users for more natural conversations so that "*You present information, ask for a response, and react accordingly* 😉"

Your script should have Main menu to select operations mentioned above. Main menu items may have further sub-menu for more natural and friendlier UI.

For Example:



```
M A I N - M E N U

1. User Management
2. Service Management
3. Process Management
4. Backup

Enter your choice [1-4]
```

Your script (**OS**) should use C/Python programs when creating processes, threads and killing processes.

-You can add further options into your script to make your OS more useful.
-Make necessary assumptions to design your OS that implements all the required options.