

Is

**FAKULTI TEKNOLOGI KEJURUTERAAN
 ELEKTRIK DAN ELEKTRONIK
 UNIVERSITI TEKNIKAL MALAYSIA MELAKA**

OPERATING SYSTEMS

BEEC3453

SEMESTER 1

SESI 2020/2021

LAB 1: PROCESS & THREADS

NO.	STUDENTS' NAME	MATRIC. NO.
1.	RAHMAN KAZI ASHIKUR	B081910450
2.		
3.		
PROGRAMME	BEEC	
SECTION / GROUP	BEEC-S1/3	
DATE	1/2/2021	
NAME OF INSTRUCTOR(S)	1. NOOR MOHD ARIFF BIN BRAHIN	
	2.	

EXAMINER'S COMMENT(S)	TOTAL MARKS

Rev. No.	Date	Author(s)	Description
1.0	30 JAN 2019	1. Shamsul Fakhar 2. Noor Mohd Ariff	1. Update to new UTeM logo 2. Update faculty's name 3. Change "course" to "programme" 4. Remove verification stamp

1. LEARNING OUTCOMES

1. Differentiate the functionality among various kinds of OS components.
2. Manipulate OS theories to solve basic functional problems.
3. Perform lab and present technical report in writing.

2. REQUIREMENTS

1. PC with Linux Ubuntu installed (or any other POSIX-compliant system)
2. Knowledge of C programming language

3. SYNOPSIS & THEORY

What is a Process?

An executing instance of a program is called a process. For example, when you double-click the Microsoft Word icon, you start a process that runs Word. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

What is a Thread?

A thread is the entity within a process that can be scheduled for execution. When you start Word, the operating system creates a process and begins executing the primary thread of that process. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

4. PROCEDURE

PART 1: Introduction to Linux

The Linux ecosystem gives the user a choice between interacting in GUI (Graphical User Interface) or CLI (Command Line Interface). While it is sufficient for the average computer users to just use the GUI especially since the inception of Microsoft Windows, the geeks and the more advanced computer users prefer the command line, just because you have to know what you're doing to use it. You have to know the exact commands. You can't hunt and peck like you do with a GUI.

1. Start your computer and once Ubuntu has been loaded, press Ctrl+Alt+T to invoke the command line interface, also known as the terminal.
2. Type **pwd** (print working directory) and hit Enter.
3. The **pwd** command will allow you to know in which directory you're located (**pwd** stands for "print working directory"). Example: "**pwd**" in the Desktop directory will show "~/Desktop".
4. Type **ls** (directory listings) and hit Enter.
5. The **ls** command will show you the files in your current directory. Used with certain options, you can see sizes of files, when files were made, and permissions of files. Example: "**ls ~**" will show you the files that are in your home directory.
6. Type **clear** and hit Enter.

7. The **clear** command will clear the screen from previous outputs.
8. To change directory into another subfolder, type **cd <subfolder name>** and hit Enter.
9. To go up one level to the previous folder, type **cd ..** and hit Enter.
10. You can also use the command **cd -** to go back to the previous directory.
11. If you have a command that you want to use but you're not quite sure how to use it, you can always refer to the command's help page by following this format:

```
<command name> --help
```

```

ashik@ashik:~$ pwd
/home/ashik
ashik@ashik:~$ ls
a.out      Documents  Music  Pictures  r.c  Templates  Videos
Desktop  Downloads  o1.c   Public    snap  Untitled-1.cpp  w.c
ashik@ashik:~$ cd Desktop/
ashik@ashik:~/Desktop$ cd ashik\kazi\b081910450/
bash: cd: ashikkazib081910450/: No such file or directory
ashik@ashik:~/Desktop$ cd ..
ashik@ashik:~$ cd --help
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Change the shell working directory.

Change the current directory to DIR.  The default DIR is the value of the
HOME shell variable.

The variable CDPATH defines the search path for the directory containing
DIR.  Alternative directory names in CDPATH are separated by a colon (:).
A null directory name is the same as the current directory.  If DIR begins
with a slash (/), then CDPATH is not used.

If the directory is not found, and the shell option 'cdable_vars' is set,
the word is assumed to be a variable name.  If that variable has a value,
its value is used for DIR.

Options:
  -L      force symbolic links to be followed: resolve symbolic
          links in DIR after processing instances of '..'
  -P      use the physical directory structure without following
          symbolic links: resolve symbolic links in DIR before
          processing instances of '..'
  -e      if the -P option is supplied, and the current working
          directory cannot be determined successfully, exit with
          a non-zero status
  -@      on systems that support it, present a file with extended
          attributes as a directory containing the file attributes

The default is to follow symbolic links, as if '-L' were specified.
'..' is processed by removing the immediately previous pathname component
back to a slash or the beginning of DIR.

Exit Status:
Returns 0 if the directory is changed, and if $PWD is set successfully when
-P is used; non-zero otherwise.
ashik@ashik:~$ █

```

12. Try to read the help file on how to use the commands below and try out the commands yourself:

• cp • rm • mkdir • rmdir

```
ashik@ashik:~$ ls
a.out  Documents  learn.txt  o1.c  Public  snap  Untitled-1.cpp  w.
Desktop Downloads Music  Pictures  r.c  Templates  Videos
ashik@ashik:~$ cd desktop/
bash: cd: desktop/: No such file or directory
ashik@ashik:~$ cd Desktop/
ashik@ashik:~/Desktop$ ls
0s-lab1  ashik_lab1  my
ashik@ashik:~/Desktop$ my
my: command not found
ashik@ashik:~/Desktop$ cd my
ashik@ashik:~/Desktop/my$ my ..
my: command not found
ashik@ashik:~/Desktop/my$ cd my ..
bash: cd: too many arguments
ashik@ashik:~/Desktop/my$ cd ..
ashik@ashik:~/Desktop$ cd os-lab1
bash: cd: os-lab1: No such file or directory
ashik@ashik:~/Desktop$ mkdir subfolder
ashik@ashik:~/Desktop$ ls
0s-lab1  ashik_lab1  my  subfolder
ashik@ashik:~/Desktop$ rmdir subfolder
ashik@ashik:~/Desktop$ ls
0s-lab1  ashik_lab1  my
ashik@ashik:~/Desktop$
```

PART 2: Hello World

It is always a good idea to run a simple program in the beginning of an experiment to ensure that the compiler and the environment has been properly set up.

1. In terminal, go to Desktop and create a directory “BEEC3453_lab1”. **note: Remember to delete the directory after you have completed the lab session.*
2. Type “vi” and hit Enter to open the Unix visual editor.
3. **Save** the file as "helloWorld.c" on your desktop.
4. Write the code below into "helloWorld.c":

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");

    return 0;
}
```

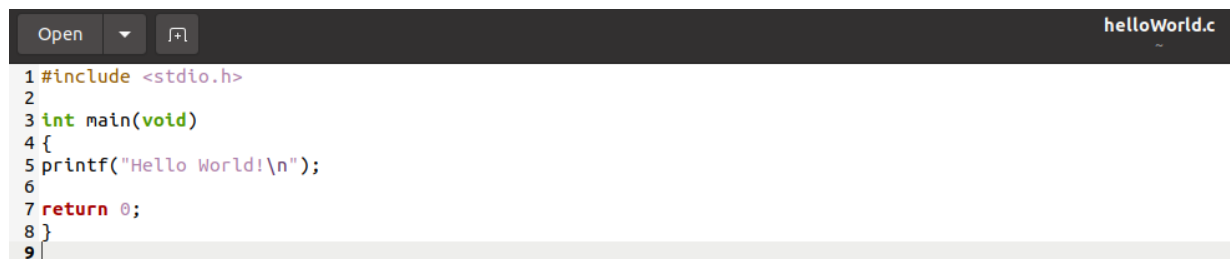
5. To **compile** the program, open the command line editor. Browse to the directory where "helloWorld.c" is located and enter the following command:

```
gcc -o myOutput helloWorld.c
```

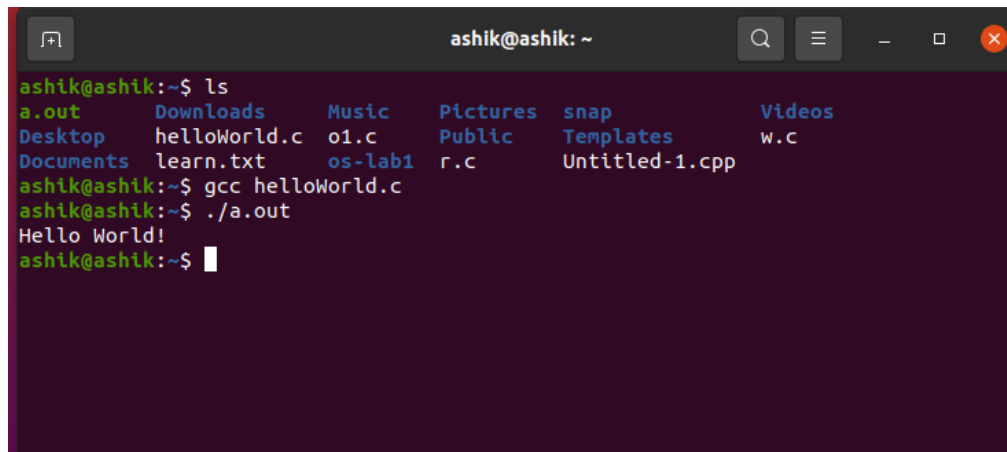
6. If everything compiles without any errors, an **output file** named "myOutput" will be created on the desktop.
7. To **run** the output file, enter the following command:

```
./myOutput
```

8. You should see "Hello World!" displayed on the command line as the program is executed.
9. Now that you know how to create a new file, code, compile and run the program, we can now move on to more complex codes.



```
Open ▼ [icon] helloWorld.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
6
7     return 0;
8 }
9 |
```


A terminal window titled 'ashik@ashik: ~' with standard window controls. The terminal shows the following commands and output:

```
ashik@ashik:~$ ls
a.out      Downloads  Music      Pictures   snap       Videos
Desktop    helloWorld.c  o1.c      Public    Templates  w.c
Documents  learn.txt    os-lab1    r.c       Untitled-1.cpp

ashik@ashik:~$ gcc helloWorld.c
ashik@ashik:~$ ./a.out
Hello World!
ashik@ashik:~$
```

PART 3: The fork() Command

When you come to metaphorical "fork in the road" you generally have two options to take, and your decision effects your future. Computer programs reach this fork in the road when they hit the `fork()` system call.

At this point, the operating system will create a new process that is exactly the same as the parent process. This means all the state is copied, including open files, register state and all memory allocations, which includes the program code.

1. Open text editor and create a new file.
2. **Save** the file as "fork.c" on your desktop.
3. Write the code below into "fork.c":

```

#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int childProcess; int count1
    = 0; int count2 = 0;

    printf("Starting fork...\n");

    childProcess = fork();

    if(childProcess==0)
    {
        while(count1 <= 10)
        {
            printf("Execute child %d\n", count1);
            sleep(1);    count1++;
        }
    } else {
        while(count2 <= 10)
        {
            printf("Execute parent %d\n", count2);
            sleep(1);    count2++;
        }
    }

    return 0;
}

```

4. **Compile** and **run** the program.

```
1
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main(void)
6 {
7     int childProcess;
8     int count1 = 0;
9     int count2 = 0;
10
11     printf("Starting ... \n");
12
13     while (count1 <= 10)
14     {
15         printf ("Execute child %d\n", count1);
16         sleep (1);
17         count1++;
18     }
19
20     while (count2 <= 10)
21     {
22         printf ("Execute parent %d\n", count2);
23         sleep (1);
24         count2++;
25     }
26
27     return 0;
28 }
29
```

```
ashik@ashik:~$ ls
a.out  Desktop  Documents  Downloads  helloWorld
ashik@ashik:~$ gcc nofork.c
ashik@ashik:~$ ./a.out
Starting ...
Execute child 0
Execute child 1
Execute child 2
Execute child 3
Execute child 4
Execute child 5
Execute child 6
Execute child 7
Execute child 8
Execute child 9
Execute child 10
Execute parent 0
Execute parent 1
Execute parent 2
Execute parent 3
Execute parent 4
Execute parent 5
Execute parent 6
Execute parent 7
Execute parent 8
Execute parent 9
Execute parent 10
ashik@ashik:~$
```

5. Observe and note your findings.

First Create child than create parent so it take more time.

6. Now **modify** the program to wait for 10 seconds before doing the fork.
7. **Compile** and **run** the program. At the same time, open another terminal window and observe the current running processes from terminal by typing the command:

```
ps -a | grep myfork
```

8. Note how the same program executes 2 separate processes when forking is called.


```
ashik@ashik:~$ ./fork
Starting fork...
Execute parent 0
Execute child 0
Execute parent 1
Execute child 1
Execute parent 2
Execute child 2
Execute parent 3
Execute child 3
Execute parent 4
Execute child 4
Execute parent 5
Execute child 5
Execute parent 6
Execute child 6
Execute parent 7
Execute child 7
Execute parent 8
Execute child 8
Execute parent 9
Execute child 9
Execute parent 10
Execute child 10
ashik@ashik:~$
```

Here we can see parent and child create same time so it takes small time to treat parent and child

5. QUESTIONS

1. List down the command you need to enter in terminal in order to create a folder named “My Folder” on your desktop.

mkdir 'My Folder'

2. List down the command you need to enter in terminal in order to create a folder named “Another Folder” located inside “My Folder”.

cd 'My Folder'

mkdir 'Another Folder'

3. List down the command you need to enter in terminal in order to delete “My Folder” and it’s contents.

rmdir -r My Folder

4. What are the differences between running the similar code in Part 3 by using fork and without fork? Explain your observation.

Fork() code runs both the parent and child processes at the same time, whereas nofork() code runs the child process first and then the parent process. As a result, code with fork will run faster than code without fork.

5. What is the advantage of having multiple processes running simultaneously?

Multiple processes running simultaneously has the benefit of allowing multiple tasks to be completed at the same time. Furthermore, it can improve the consistency of work and save time.

6. Give an example where you, as a normal computer user, benefit from this advantage.

It lets us to use many applications at the same time without any problems, and it allows the user to view multiple tasks at once

7. What are the disadvantages? How can this be exploited?

Despite the fact that it has its own memory and file system, it nonetheless runs the same code. Furthermore, it necessitates a huge amount of memory and has a bigger memory footprint. Because there are so many memories allocated, it will be more prone to exploits, such as a denial of service (DoS) attack known as a fork bomb.

8. Consider the code below. By looking at the code, what would be the predicted output? Explain in simple terms how the code is executed by the processor.

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

int main(void)
{
    int processId;

    while(1)
    {
        processId = fork();
        printf("Running process %d\n", processId);
    }
}
```

The predicted output would be like "**Running Process 1**" but currently the code outputs the result infinitely because the set up parameters in our while loop is static integer and does not know when to stop the process.

An exploit is a piece of code that takes advantage of a software defect or vulnerability. It's written by security researchers as a proof-of-concept threat or by malicious actors for use in their attacks.


```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4
5 void forkexample()
6 {
7 //return calue zero so child presecc
8 if(fork() == 0)
9 print("Hellow from child!\n");
10
11 else
12 printf("hellow from parent!\n");
13 // return non-zero. so parent process
14
15 }
16
17 int main(){
18 int processID;
19 int count = 0;
20 while(1){
21 processID = fork();
22 printf("Runnig process %d\n", processID);
23 count++;
24 }
25
26 //fork();
27 //printf("Hellow world\n");
28 //forkexample();
29 return 0;
30 }

```

. out put If the number return is 0, the process is called as a child process. If the number is greater than 0, then the process is called parent process. It will also loop fork in infinite loop and cause the fork bomb.

9. How can we prevent exploits such as in the code above?

To avoid a fork bomb, avoid running fork in an infinite loop. Because fork loop consumes CPU and memory, system resources will be provided until the operating system hits the maximum authorised process. It will create a self-replicating child process, preventing legal programmes from functioning and preventing the formation of new processes. As a result of this condition, the kernel will be unable to cope and will crash..

To Prevent exploitation such code above, We need to know when to stop the process by setting conditions in while loop parameter example is, we set processId < 5 inside the parameter of the while loop assuming the fork() method returns an integer then once the condition result as false, the loop will automatically terminated see the code below:

```
int processId = 0;

while(processId < 5)
{
    processId = fork();
    printf("Running process %d\n", processId);
}
```

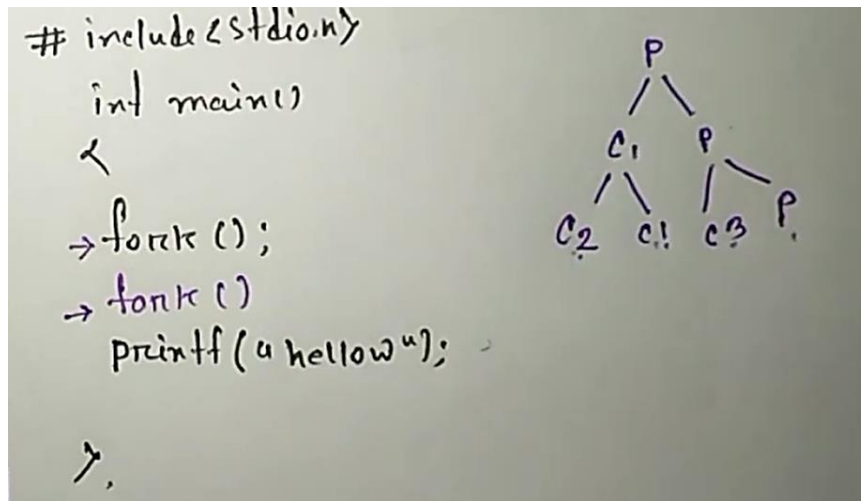
6. DISCUSSIONS & CONCLUSION

(What have you learned from this experiment?)

Fork()

Fork system call is used for creating a new process. Which is call child process. There child process will run concurrently with its parent process.

Fork() returns an integer value



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

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

Take a step-up from those "Hello World" programs. Learn to implement data structures like Heap, Stacks, Linked List and many more! Check out our Data Structures in C course to start learning today.

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.

Conclusion: - In this lab I learn Differentiate the functionality among various kinds of OS components. Manipulate OS theories to solve basic functional problems. Perform lab and present technical report in writing.