First Assignment OS

Feras Fadi Mustafa **Naeem Al-Oudat**

3.1: Process Table

Solution:

1) In the report, include the relevant lines from **"ps -l"** for your programs, and point out the following items:
   - Output:
   - process name:

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1000	12117	12110	0	80	0	-	2654	Do_wai	Pts/0	00:00:00	bash
0	T	1000	12230	12117	0	80	0	-	624	Do_sig	Pts/0	00:00:00	test
0	R	1000	12358	12007	0	80	0	-	2854	-	Pts/0	00:00:00	ps

   using **“`$ ps -aux`”** `command OR` **“`ps -l`”**
   `look at CMD column.`

   - process state (decode the letter!)?
     using **“ps a”** command and see “STAT” column
   - process ID (PID):
     using **“ps -l”** command OR **getpid()** function in c language
     Process ID is: 12553
   - parent process ID (PPID):
     using **“ps -l”** command OR **getpid()** function in c language
     Parent process ID is: 12117

2) Repeat this experiment and observe what changes and doesn't change.
   - Process ID(PID) and Parent process ID(PPID) **they change**
     They will be Process ID is:
     Process ID is: 12603
     Parent process ID is: 12117

3) **Find out the name of the process that started your programs. What is it, and what does it do?**
   Using **“ps axgf”** command this command print process as tree
   The first row this is a first process.

Summary:

- **In First test stand out error:**

implicit declaration of function 'getpid' Warning?

After searching solution the problem “if you do not include this header file, to fix the warning include <unistd.h> header file in the program” To solve this install Cygwin

- **Underestimated sleep time to facilitation**

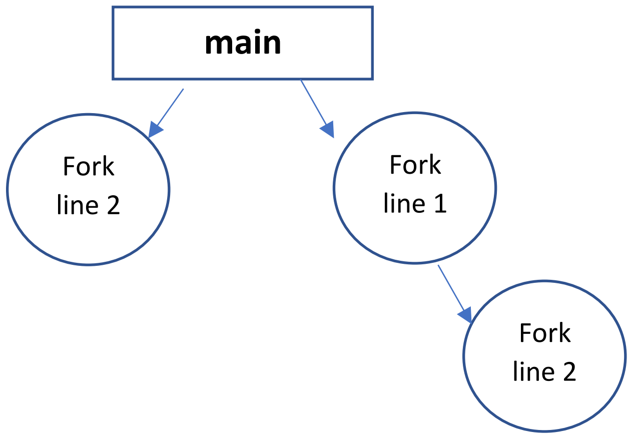
3.2 The fork() system call:

- **Include the output from the program.**

```
Process 15068's parent process ID is 15053

Process 15070's parent process ID is 15068

Process 15069's parent process ID is 15068

Process 15071's parent process ID is 15069
```

- **Draw the process tree (label processes with PIDs)**

2 ^ n child



- **Explain how the tree was built in terms of the program code**
  - When will built a program the first version will be created (main function)
  - When execution first line (fork() in line one) will create a copy of program
  - Last copy program will be create another copy (when execution fork() in line two)
  - Last main function will be execution fork() in line two.
  - Fork() in line two will be created another copy.
- **Explain what happens when the sleep statement is removed.**

  Sleep function in order to wait for a current thread for a specified time

- this is meaning we will remove this function will happen overlap (nip up) in threads **the output will be as a**

```
Process 15388's parent process ID is 15053

fersabarahmeh@ubuntu:~/Desktop/Lap$ Process 15389's
parent process ID I3546

Process 15390's parent process ID is 3546
```

3.3 The fork() syscall, continued:

- **Include the (completed) program and its output**

```
#include <unistd.h>
#include <stdio.h>
 #include <sys/types.h>
 #include <sys/wait.h>
int main() {
        int ret;
        ret = fork();
        if (ret == 0) {
        /* this is the child process */
        printf("The child process ID is %d\n", getpid());
        printf("The child's parent process ID is %d\n", getppid());
        } else {
        /* this is the parent process */
        printf("The parent process ID is %d\n", getpid());
        printf("The parent's parent process ID is %d\n", getppid());
        }
        sleep(2);
        return 0;
} // The solution:  condition in if statement "ret == 0".
```

- **Speculate why it might be useful to have fork return di_erent values to the parent and child. What advantage does returning the child's PID have?**

When you fork(), the code that's running finds itself running in two processes (assuming the fork is successful): one process is the parent, the other the child. fork() returns 0 in the child process, and the child pid in the parent process: it's entirely deterministic.

This is how you can determine, after the fork(), whether you're running in the parent or the child. (And also how the parent knows the child pid — it needs to wait on it at some point.)

In a little more detail:

- the future parent process calls fork();
- the kernel creates a new process, which is the child, and sets various things up appropriately — but both processes are running *the same code* and are "waiting" for a return from the same function;
- *both processes continue running* (not necessarily straight away, and not necessarily simultaneously, but that's besides the point):

- fork() returns 0 to the child process, which continues and uses that information to determine that it's the child;
- fork() returns the child pid to the parent process, which continues and uses that information to determine that it's the parent.

Summary:

The first speculated if fork() return id this id as pointer to childe this is help to be connection parent to child and vice versa. This first reason the second reason if will child and parent process at the same time

3.4 Time Slicing

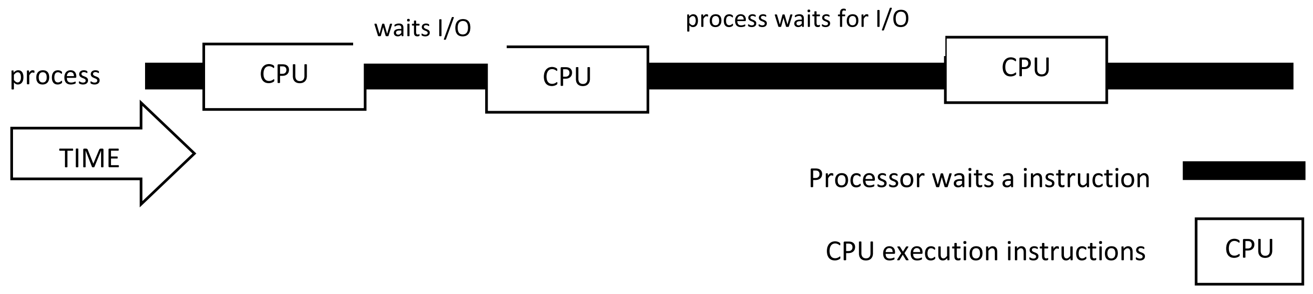
- **Include small (but relevant) sections of the output**

  After reduction repeated the output is.

```
Child: 0
Child: 1
Child: 2
Child: 3
Child: 4
Parent: 0
Parent: 1
Parent: 2
Parent: 3
Parent: 4
```

- **Make some observations about time slicing**

  I will explain that in a simple figure



Time slicing:

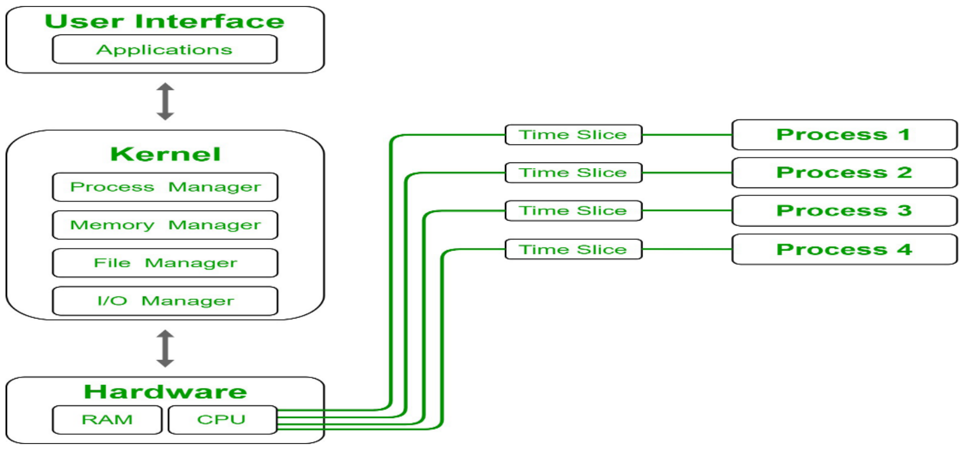
Before time slicing:

CPU execution instructions and waits in these cases:

- When a process waits for I/O
- When a process has got its quota at time.
- When a previously started I/O has completed.

Time slicing when any one in these cases happened the advantage CPU execution instruction for all times

We can definition time: A short interval of time allotted to each user or program in a multitasking or timesharing. Time slices are typically in milliseconds.



Summary:

- I reduced the repetition to be able to explain the code

- ## 3.5 Process synchronization using wait():

- **Explain the major difference between this experiment and experiment 4:**

wait() function :

Using to stops parent executing the operation until the operation is executed child.

This process done by calling from the programmer **but** time slicing done by OS

And wait function: If some higher priority process arrive then preemptive scheduling have ability to stop executing process and start executing this newly arrived higher priority process

Time slicing scheduling: process execute for particular assigned slice of time of that process and stop them and start another process and after that it will execute according to priority

- ## 3.6 Signals using kill():

- **The program appears to have an infinite loop. Why does it stop?**

  We are know pid of child equal 0 and parent greater than 0
  In this code the condition in if statement will be true the condition of while loop true always this is an infinite loop.

- **From the definitions of sleep and usleep, what do you expect the child's count to be just before it ends?**

  Using #include <time.h> library and this structure ➡

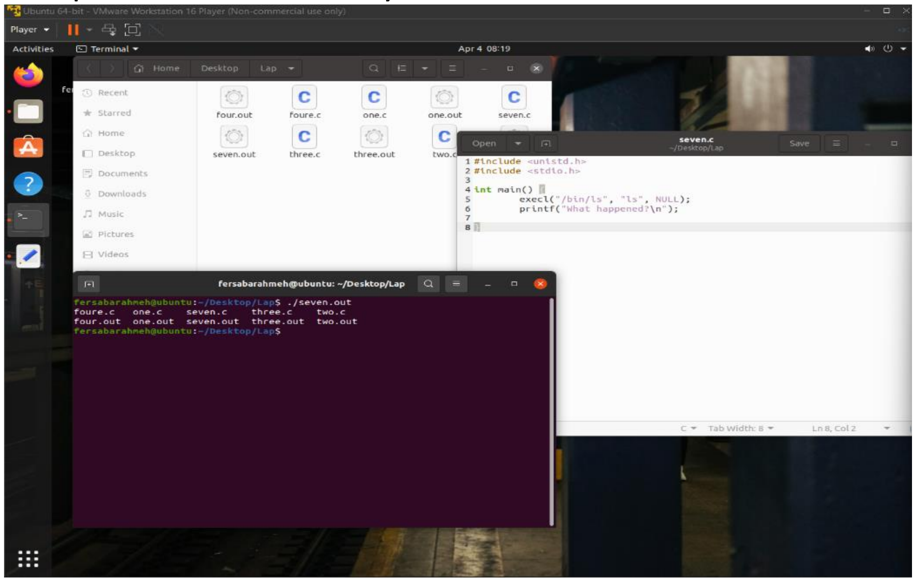
```
 time_t begin;
 time(&begin);
            // Somethings
time_t end;
time(&end);
printf("Execution time %f\n", difftime(end, begin));
return (0);
```

- **Why doesn't the child reach this count before it terminates?**

  Because after each process be stopped 0.001 sec

- ## 3.7 The execve() family of functions:
  - ### The output:

    Will print files in directory



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

int main() {
        execl("/bin/ls", "ls", NULL);
        printf("What happened?\n");

}
```

```
fersabarahmeh@ubuntu:~/Desktop/Lap$ ./seven.out
foure.c   one.c    seven.c    three.c    two.c
four.out  one.out  seven.out  three.out  two.out
fersabarahmeh@ubuntu:~/Desktop/Lap$
```

- **Under what conditions is the printf statement executed? Why isn't it always executed?**
  - **Under what conditions is the printf statement executed?**

    The exec() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for execve(2). (See the manual page for execve(2) for further details about the replacement of the current process image.)
    The initial argument for these functions is the name of a file that is to be executed.
    The functions can be grouped based on the letters following the "exec" prefix.
  - **Why isn't it always executed?**

    When the compiler execution first line (**execl("/bin/ls", "ls", NULL);)**
    Will get override for this process and execution the process in parameter

- ## 3.8 The return value of main()

  - ### What is the range of values returned from the child process?
    - Child exited with status 174 **(Note: 174 random number in each run the number change)**
  - **What is the range of values sent by child process and captured by the parent process?**
    - 0 To RAND_MAX/4

- ## 3.9 Zombie process
  - ### What is a zombie process?

    - is a process that has completed execution (via the exit system call) but still has an entry in the process table. This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status
- **Write a program that creates a zombie process and describe how to verify that the zombie process has been created.**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
   pid_t pid;
   int status;

   if ((pid = fork()) < 0) {
      perror("fork");
      exit(1);
   }

   /* Child */
   if (pid == 0)
      exit(0);

   /* Parent
    * Gives you time to observe the zombie using ps(1) ... */
   sleep(100);

   /* ... and after that, parent wait(2)s its child's
    * exit status, and prints a relevant message. */
   pid = wait(&status);
   if (WIFEXITED(status))
      fprintf(stderr, "\n\t[%d]\tProcess %d exited with status %d.\n",
            (int) getpid(), pid, WEXITSTATUS(status));

   return 0;
}
```