OS LAB ASSIGNMENT-1

Submitted by: KS Ashin Shanly - 20250019

Question 01: Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., `100`). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?

OUTPUT

```
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ ./a.out
Parent process!!
The final value of x is =4
Child process!!
The final value of x is =6
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$
```

> The fork() call makes a near duplicate of the current process, identical in almost every way (not *everything* is copied over, for example, resource limits in some implementations, but the idea is to create as close a copy as possible). Only one process *calls* fork() but *two* processes return from that call. The new process (called the child) gets a different process ID (PID) and has the PID of the old process (the parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they need to be able to tell which is which - the return code of fork() provides this information - the child gets 0, the parent gets the PID of the child (if the fork() fails, no child is created and the parent gets an error code).

The value in the child process is the same as the value in the mainprocess. Here x = 5 initially, the parent process decrements x, whereas the child process increments x. Now the child and the parent process will print out different values of x as they have separately changed their copies.

Question 02: Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?

OUTPUT

```
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ ./a.out
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ cat sample.txt
Parent process!!
Child process!!
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$
```

Yes, both the child and parent can access the file descriptor. They can access the file concurrently, but the OS decides which one of the parent and child writes first to the file here. This can vary from system to system.

Question 03: Write another program using `fork()`. The child process should print “hello”; the parent process should print “goodbye”. You should try to ensure that the child process always prints first; can you do this *without* calling **wait()** in the parent?

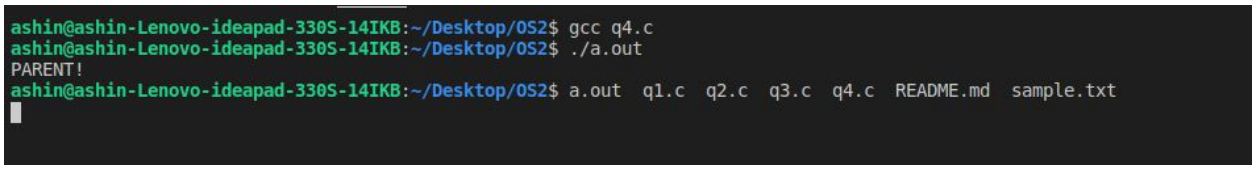
OUTPUT

```
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ ./a.out
HELLO!
GOODBYEE!
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$
```

> To ensure the child process prints first, we need a kind of synchronization in between both processes, and there are a lot of system primitives that have a semantic of "communication" between processes (for example locks, semaphores, signals, etc). I doubt one of these is to be used here. Instead of wait(), a **waitpid()** can be used to do this as shown in the corresponding program. Otherwise, a simple **sleep(1)** can also be used instead of wait() in the parent process to ensure that the child prints first.

Question 04: Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including (on Linux) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvpe()`. Why do you think there are so many variants of the same basic call?

OUTPUT



> The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file unistd.h. When the file q4.c is compiled, as soon as the statement execvp(args[0],args) is executed, this very program is replaced by the program /bin/ls. Any statements after this statement are not printed, because as soon as the execvp() function is called, this program is replaced by the program /bin/ls.
> All the functions of family exec essentially do the same thing: loading a new program into the current process, and providing it with arguments and environment variables. The differences are in how the program is found, how the arguments are specified, and where the environment comes from.

Question 05: Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?

OUTPUT

```
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ ./a.out
CHILD!
PARENT!
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$
```

> wait() returns process ID of the terminated child process. If more than one child processes are terminated, then wait() returns any ***arbitrary child*** process ID.
If any process has no child process then wait() returns “-1” immediately. Therefore, if we use a wait() inside a child, and if it has no child of it’s own, then wait() returns a “-1”.

Question 06: Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?

OUTPUT

```
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ ./a.out
CHILD!
PARENT!
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$
```

> wait() blocks the caller until a child process terminates. waitpid() can be either blocking or non-blocking. If *options* is 0, then it is blocking, if *options* is WNOHANG, then is it non-blocking.
if more than one child is running then wait() returns the PID of first child to exit. waitpid() is more useful in this case as, if *pid* == *-1*, it waits for any child process. In this respect, waitpid is equivalent to wait. if *pid* > *0*, it waits for the child whose process ID equals pid ,if *pid* == *0*, it waits for any child whose process group ID equals that of the calling process if *pid* < *-1*, it waits for any child whose process group ID equals that absolute value of pid.

Question 07: Write a program that creates a child process, and then in the child closes standard output (`STDOUT_FILENO`). What happens if the child calls `printf()` to print some output after closing the descriptor?

OUTPUT

```c
#include <stdlib.h>

int main(){
    int a = fork();
    if(a<0){
        printf("Error!");
        exit(1);
    }
    else if(a==0){
        close(STDOUT_FILENO);
        printf("Statement after closing descriptor in child!\n");
    }
    else{
        printf("Parent!!\n");
    }

    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$ ./a.out
Parent!!
ashin@ashin-Lenovo-ideapad-330S-14IKB:~/Desktop/OS2$
```

> After closing the standard output in the child, nothing will be printed. As it can been seen, only the parent's printf() is clearly printed out here, this is because the standard output file descriptor is the basis for using printf().

Question 08: Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the `pipe()` system call.

> Pipe is one-way communication only i.e we can use a pipe such that one process writes to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a ***“virtual file”***. The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it. If a process tries to read before something is written to the pipe, the process is suspended until something is written. The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and write ends of the pipe.

When we use a fork in any process, file descriptors remain open across the child process and parent process. If we call a fork after creating a pipe, then the parent and child can communicate via the pipe.