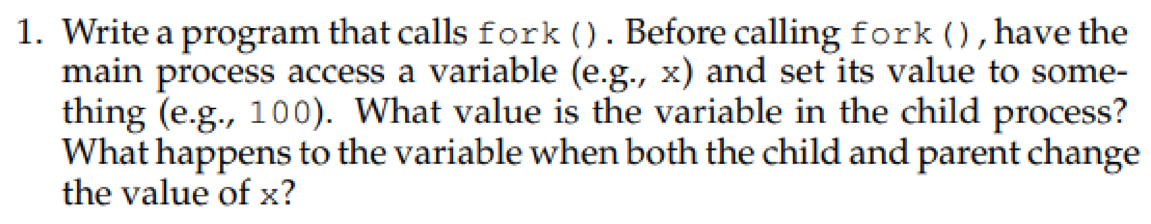
【OS】Day9(3)

【Ch2】Process API Homework(2)

All of the codes for this homework will be posted on my Github: KingArthur0205

Question 1



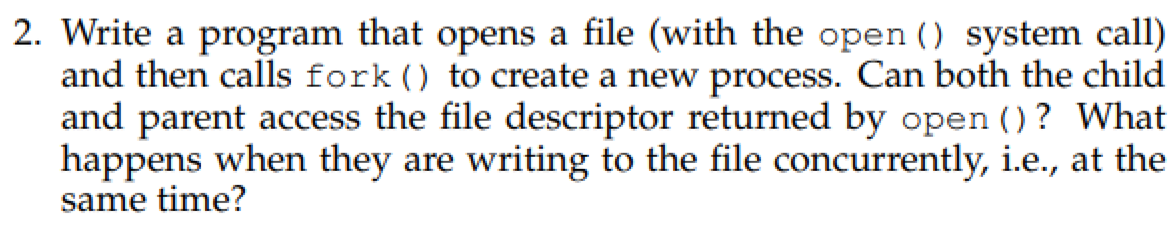
1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., `100`). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?

The varaible in the child process is also 100, an exact copy of the variable in the parent process.

Changing the variable in the child process will not change the variable in the parent process becausethe child process is an exact copy of the parent process other than the PID. It has its own stack and register values.

See Q1.c for code

Question 2



2. 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?

To use the open function, we need to incluce the following header:

```
#include <unistd.h> //have the definition of open inside
#include <sys/types.h> //have the definition type pid_t and size_t
#include <fcntl.h>
```

```
int open(const char *pathname, int flags);
```

The following flags are used:

- `O_RDONLY`: Have the file descriptor set to be read-only.
- `O_WRONLY`: Have the file descriptor set to be write-only

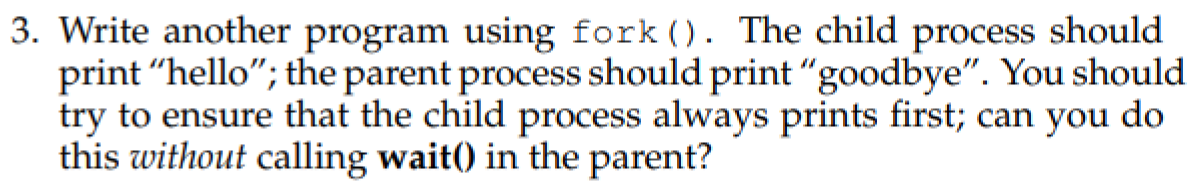
We also used function read and write in the program:

```
#include <unistd.h>

//fd is the file descriptor returned by open
//count is the number of bytes that we want to read
//buf is the destination that we want to read to
ssize_t read(int fd, void *buf, size_t count);

//buf is the source where we get our data from
//count is the number of bytes that we want to write
ssize_t write(int fd, void *buf, size_t count);
```

Have the parent process write the same file will overwrite the file. Since the `fd` in the parent and child process are two different objects, the parent process will overwrite the file from the beginning even if we set `O_APPEND` flag on.

See Q2.c for code

Question 3

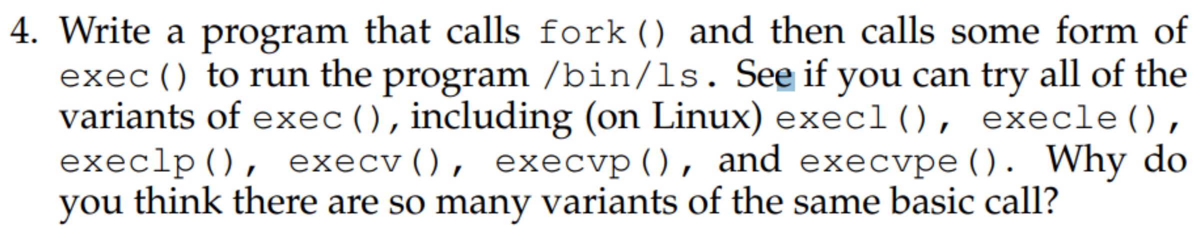


3. Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this *without* calling **wait()** in the parent?

The answer to the second question is "No", at least for now. Later we will learn a lot more advanced and low level knowledge to be able to do this.

See Q3.c for code

Question 4


4. 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?

```
#include <unistd.h>

int execl(const char *filepath, const char* arg1, const char* arg2,...)
int execlp(const char *filename, const char* arg1, const char* arg2,...)
int execle(const char *filepath, const char* arg1, const char* arg2,..., char* const envp[])
int execv(const char* filepath, char*argv[])
int execvp(const char* filename, char *argv[])
int execve(const char* filepath, char* argv[], char* const envp[])
```

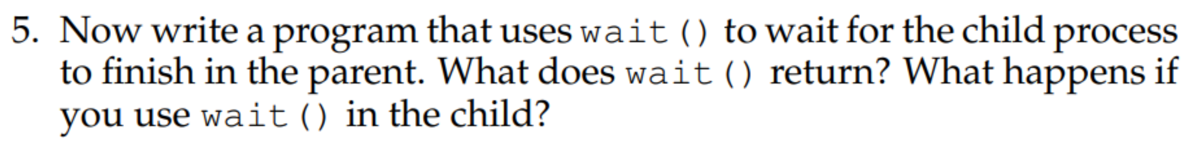
Why do we need these many variants?

1. Functions having l in their name need argument seperated by commas
2. Functions having v in their name need argument as an element in the array
3. Function have e in their name need enviornment argument

- `execl()` and `execv()` can execute any file given the absolute path
- `execlp()` and `execvp()` can execute only commands of the PATH enviornment given the file name
- `execle()` and `execvpe()` can execute command in the given enviornment

See Q4.c for code

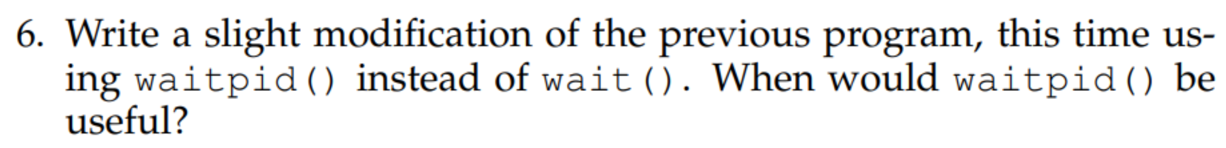
Question 5


5. 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?

`wait()` returns the pid of the terminated child; on error, -1 is returned

If we use `wait()` in child, then `wait()` returns -1

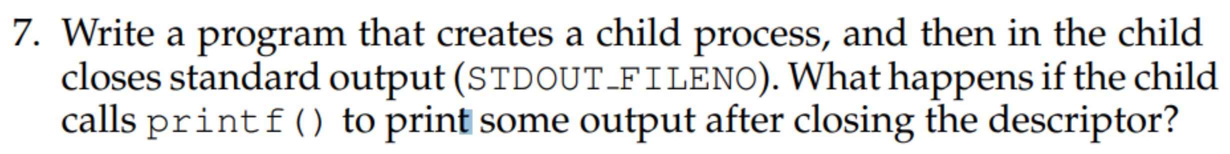
Question 6



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

`waitpid()` is used when we want to wait for a specific child process rather than aiting for all child processes to exit.

Question 7



7. 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?

If we close `STDOUT_FILENO` in the child process, we won't be able to print anything out to the screen.

However, we can still print in the parent process.

See Q7.c for code