Assignment-3

CS342: Operating System Lab

General Instruction

• Assignments should be completed and evaluated in the Lab session.

• Markings will be based on the correctness and soundness of the outputs. Marks will be deducted in case of plagiarism.

• The assignments must be written in C language. Proper indentation & appropriate comments (if necessary) are mandatory in the code.

In today's lab session, the goal is to use system calls related to process creation and usage. After completing this lab you will find yourself closer to building multi-process programs.

A set of system calls that we will use in this lab are listed below. You should look up the appropriate man pages for details of each.

- **fork**: This system call is used to create a new process. After the process is created, both parent and child processes continue execution from the point after fork(). The return value is different in child and parent processes: zero in the child and the process-id(pid) number of the child in the parent. In case of error, -1 is returned and new process is not created.
- **wait**: This call makes the calling process wait till a child process terminates and reclaims the resources associated to it. The return value is the pid of the child process. A variant of this, waitpid waits for a process with a given pid.
- **exec**: This call is used to run a new executable by replacing calling process's code with the executable program's code. The code after exec() in the original process is executed only if exec fails. If it succeeds, execution continues from the first line of the executable. Check out the several variants of this in the man page.
- **open**: This call is used to open a file or create one. The return value is the file descriptor of the opened file. This call assigns the smallest unused non-negative integer as the file descriptor.
- **close**: This call loses a file descriptor. The resources associated with the open file are freed.
- **read**: This call is used to read a specified number of bytes from a file descriptor into a buffer. It does not necessarily read the requested number of bytes as the file may have lesser number of bytes. The number of bytes read is returned.
- **write**: This call is used to write a specified number of bytes from a buffer to a file descriptor. The number of bytes written is returned
- **getpid**: This returns the process ID of the calling process and other process ID related calls (get parend pid etc.).

The default first three file descriptors, which are opened and available for every process, are: 0: Standard input (stdin); 1: Standard output (stdout); 2: Standard error (stderr)

Task 1: Baby steps to forking

Write a program p1.c that forks a child and prints the following (in the parent and child process),

Parent : My process ID is: 12345
Parent : The child process ID is: 15644

and the child process prints
Child : My process ID is: 15644
Child : The parent process ID is: 12345.

Task 2: Write another program p2.c that does exactly same as in previous exercise, but the parent process prints it's messages only after the child process has printed its messages and exited. Parent process **waits** for the child process to exit, and then prints its messages and a child exit message,

Parent : The child with process ID 12345 has terminated.

File Descriptors, Fork, and Exec

A file descriptor (a.k.a fd) is an abstract indicator (handle) used to access a file or other input/output resources, such as a pipes, network sockets, disks, terminals etc. A file descriptor is referenced as a non-negative integer index in a descriptor table. When a fork operation occurs, the file descriptor table is copied for the child process, which allows the child process access to the files opened by the parent process.

Note: Only file descriptor entries are copied, system wide file information along with current offsets etc. are common and can be manipulated by each process.

The exec system call is used to load an executable in a process by overwriting the content of the existing process with contents of the executable. Typically, in Unix systems, as fork is the primary way of creating processes, a fork+exec combination is used to spawn new programs. In this technique, to run an executable, the parent process forks a new process and the child process uses exec or it's variants to load and run the executable.

Task 3: A program mycat.c that reads input from stdin and writes output to stdout. Further, write a program p3.c that executes the binary program mycat (compiled from mycat.c) as a child proces of p3.

Task 4a: Writing to a file without opening it

Write a program p4a.c which takes a file name as command line argument. Parent opens file and forks a child process. Both processes write to the file, "hello world! I am the parent" and "hello world! I am the child". Verify that the child can write to the file without opening it. The parent process should wait for the child process to exit, and it should display and child exit message.

Task 4b: Input file re-direction magic

Write a program p4b.c which takes a file name as a command line argument. The program should print content of the file to stdout from a child process. The child process should execute the mycat program.

Cannot use any library functions like printf, scanf, cin, read, write in the parent of child process.

Hint: close of a file results in it's descriptor to be reused on a subsequent open.

Note: Do not make any modifications in mycat.c