**ASSIGNMENT-1**

**Question1:** Consider the C function printf() on UNIX. Is printf() implemented by the OS, or by an application-level library? What system call does printf() make internally?

**Answer:** printf() is implemented by the application library of C. It uses the write() system call. The write() system call is actually responsible for sending data to the output.

**Question2:** Why are mkdir, ln and rm implemented as separate user-level programs, while cd is implemented as a built-in command?

**Answer:** cd command changes the directory of the current program in which the user is executing. If we use it as a user level program, then the child process will automatically copy the directory of the parent process. Hence there will be no change in the child directory and so using cd command as user level program will be useless. That’s why it is used as a built-in function.

**Question3:** What is the total number of processes at the end of the execution of the following program? Assume there is one process in the beginning that starts running at main. Also, assume that all system calls succeed.

main()

{ fork();

fork();

fork(); }

Explain.

**Answer:** There will be total eight processes.

The first fork() creates a parent p1 and a child c1, hence two processes are created. The second fork() creates a child of c1(c2) and p1(c3) . So, four processes are created till now. Now, the third fork() call creates child of p1(c4), c1(c5), c2(c6) and c3(c7). Therefore, total eight processes – p1, c1, c2, c3, c4, c5, c6, c7 – will be there.

**Question4:** Consider the following program:

main()

{ int fd;

fd = open(outfile, O\_RDWR)

fork();

write(fd, Hello, 5);

exit(); }

Assume all system calls finish successfully on a uniprocessor system. Also, assume that a system call cannot be interrupted in the middle of its execution. What will be the contents of the outfile file, after all processes have successfully exited? Explain briefly.

**Answer:** The contents of the output file will be:

Hello

Hello

REASON: The function fork() creates a copy of itself, thus, two processes will be there- one parent and the other child. Since two simultaneous processes are created, so output will be given two times.

**Question5:** In UNIX, a child process may terminate before a parent calls wait(). When the parent calls wait() eventually, it still expects to read the correct exit code that the child returned. To support this functionality, UNIX does not completely remove the process till it’s parent has called wait() on it. Such processes that have completed execution but still have an entry in the process table are called zombie processes. Usually, the presence of zombie processes in the system for a long time indicates a bug in the program (it is a common error).

UNIX also provides the SIGCHLD signal, which is received by the parent process whenever one of its children exits. In class we discussed that the shell implements “&” functionality by not calling wait() immediately. Should the shell never call wait()? When should it call wait()? Answer by providing short pseudo-code. (Hint: you may want to use the SIGCHLD signal).

**Answer:** The shell should call wait() at a time as if zombie processes are not destroyed, then it may cause resource leak or may create an error.

PSEUDOCODE:

if pid>0 :

while SIGCHLD != 1:

wait()

else if pid==0:

…. //execute the child process//

SIGCHLD generates if one of the processes created by parent process has exited. Thus, when child is a process, parent needs to wait() and when the child has terminated, SIGCHLD will be true and the parent need not to wait in that case.