Part 4

1. We can implement commands successively using the system library function, which calls fork() to create a child process and excel() to execute the command via the child process. system() returns only after the command mentioned is completed.

   system(cmd1);

   system(cmd2);

   In order to execute 2 or multiple commands irrespective of the result of the previous command,
   we need to use ; to separates the commands or newline so that it has only one command per line and in the order we want it to be executed.

   (cd /XXXdirectory); (find . -name 'a*' -exec rm {} \ );(ls -la)

And, In case of executing the second command only when the first command is successful, we need to use && between the two commands.

cmd1 && cmd2

In this case only when cmd1 is successful cmd2 will be executed.

(cd /XXXdirectory) && (find . -name 'a*' -exec rm {} \ ) && (ls -la)

In this case also, we can use system() but with the appropriate condition.

If(system(cmd1)==0)

system(cmd2);

---

2. cmd1(cmd2)

layer 1

layer 2 - > sub process

layer 3 -> sub-sub process

1. Here , (……..) creates a subshell which is done using a subprocess. Cmd1 calls a fork () and wait().

a. In the subshell cmd2 is an external command which needs a subprocess of the subshell created. Thus the subshell calls fork () and wait ().

i. In the sub-sub process, executes the external command and finally terminates using the exit().

b. wait is completed in the subshell.

2. Wait is completed for the original process.

Example:-

```
echo "-$(sed 's/cat/lion/g' a.txt)" > hello.txt
```

echo is layer 1

creates a subshell for sed command- layer 2

executes sed command in the sub-sub process layer-3 (replaced all cats with lion in a.txt)

returns control to subshell – sub process

returns control to original process

echo into hello.txt file

---

3. Approach to implement background running commands supporting “&” and “wait”

When we create a child process using fork (), if it is created successfully, the parent usually waits for the child() and the child executes execvp() of the program. But here when we encounter ” &” , we want the program to run in the background, meaning that after the creation of the child, control should come back to the parent and proceed further while the child executes in the background.

For this functionality, We first call fork (), and in the child process we call setpgid(0, 0) to put the child process in another new process group. Set the background flag variable to be true. We have a condition to check if the background variable After this, the parent process continues on without waiting.

Wait % job_no. Can be used to wait until the child process which is being executed in the background is completed. It is also possible to use the pid of the child process instead of job_no.

Ex.

```
$ sleep 10 & sleep 13
$ wait %1
```

Sleep 10 happens in the background, as we have given the wait % 1 command. It will wait until sleep 10 process in done.

Part 1

Exec () functions replaces the current process image with a new process image.

char *const argv[] argument is an array of pointers to null - terminated strings that represent the argument list available to the new program. The first element of argv[] by default always points to filename of the file which we want to execute as part of the command.

v of the exec() family implies the arguments must be passed as array of strings(vector) to the main(). As this is what we are currently using we have to include v in our exec function.

p of the exec() uses the PATH variable to find the executable binary file. If the environment is not explicitly passed to the new process via the system call, and the environment of the current process is used.

Syntax:-

```
int execvp(const char *file, char *const argv[]);
```

Part 2

Open (file, flag, mode) is used to open the file with the required characteristics. In flags, we use O_RDONLY in the case of < for read-only access. In the case of > O_WRONLY for write-only access, O_CREAT for creating a new file if it does not exist, and O_TRUNC which will truncate the file if it already exists to length 0. In mode, we use S_IRUSR to give the user read permission to the file and S_IWUSR to give the use write permission to the file. These are the 2 required permissions in our use case and not giving all permissions to all users is more secure.

We then use dup() which allocates a new file descriptor that refers to the same open file descriptor. So, the fd in rcmd refers to the same file descriptor as newfd which is what was created in the open command.

Now we execute the command part of rcmd which is executed and then read or written into the file with the help of the fd.

Part 3 :

Pipe(p) creates a pipe that is used for inter-process communication. The array p is used to return two file descriptors referring to the ends of the pipe. p[0] refers to the read end of the pipe and p[1] refers to the write end of the pipe.

In the pipe command, the left part of the pipe executes a command which is read by and used in the right part of the pipe. It is a one – way communication.
When we fork() after creating a pipe, it can be used to communuiacte between parent and the child process.

This concept is leveraged for executing ‘|‘part here. One program writes into the pipe. The second process reads from the pipe once write in the pipe is completed.

So we first call dup() on the write end of the pipe and run the left command which is written with the help of fd. And then dup() on the read end of the pipe is called which reads information from the left side and right command is executed.