Systems Programming Home Work Assignment-2

Saiprakash Nalubolu

B01037579

I.1

1. Opens file x.dat with flags O_WRONLY | O_CREAT resulting in file descriptor fd1

x.dat is created (since it does not already exist) and opened for writing. Initial content of x.dat would be empty.

2. Opens file x.dat with flags O_WRONLY | O_CREAT resulting in file descriptor fd2

x.dat is opened again for writing using a new file descriptor fd2. Content of x.dat would still be empty.

3. Writes 6 ‘A’ characters to fd1

AAAAAA is written to x.dat using fd1. Hence content of x.dat is “AAAAAA”.

4. Writes 3 ‘B’ characters to fd2

BBB is written to x.dat using fd2. Since fd2 was opened separately, its file position is at the start.

Content of x.dat is “BBBAAA”.

5. Does a dup2(fd1, fd2)

fd2 is duplicated to fd1. This means fd2 now shares the same file descriptor as fd1, including the file position. Content of x.dat is BBBAAA (no change yet).

6. Writes 3 ‘B’ characters to fd2

Since fd2 is now the same as fd1 and fd1's file position is after the 6 'A' characters, this write will start from the 7th byte. Content of x.dat: “BBBAAABBB”

7. Closes descriptors fd1 and fd2

Both file descriptors are closed. Content of x.dat is “BBBAAABBB” (no change).

Overall, at step 1 and 2; both file descriptors fd1 and fd2 are opened separately, pointing to the beginning of x.dat. At step 3, writing 6 'A's with fd1 moves its file position to the end of 'AAAAAA'. Step-4, writing 3 'B's with fd2 starts from the beginning, overwriting the initial 3 'A's, resulting in BBBAAA. At step-5 dup2(fd1, fd2) makes fd2 equivalent to fd1, including the file position. At Step-6 writing 3 'B's using fd2 (which is now fd1 after dup2) starts from where fd1 left off, resulting in BBBAAABBB. Finally at step-7 closing the file descriptors does not change the file contents.

So the final content of x.dat is “BBBAAABBB”.

I.2

When you run the program as "./t", it reads from the keyboard and prints each character right away because it uses line-buffering. This means it processes each line as soon as you press Enter.

When the program is executed as "./t | ./t", the output of the first program is sent to the second program through a pipe. Pipes use block-buffering, which means the second program waits until it has a larger chunk of data before it processes and prints it. This is why one would see the output in batches instead of the lines echoing immediately.

In short, "./t" gives immediate output due to line-buffering. "./t | ./t" produces a delayed output in batches due to block-buffering because of the use of pipe(|).

I.3

1. At the start of the code, The parameter const char *dir has the same name as the DIR *dir variable, causing a conflict. They should have different names. Using the same name for both the parameter and the local variable creates a conflict, making it unclear which variable is being referred to at different points in the code. This can lead to confusion and bugs, as the local variable will shadow the parameter.
2. for (dirent dP = readdir(dir); dP; dP = readdir(dir)) is incorrect. struct dirent *dP should be used instead of dirent dP. The readdir function returns a pointer to a struct dirent. The correct declaration should therefore be struct dirent *dP
3. struct stat *statP; should be struct stat statP; because stat expects a pointer to struct stat.
4. The buffer char dir_name[strlen(dir) + 1 + strlen(name)]; does not account for the null terminator. It should be char dir_name[strlen(dir) + 1 + strlen(name) + 1];.
5. sprintf(dir_name, "%s/%s", dir, name); should be checked for buffer overflows or can be replaced with a safer function like snprintf.
6. The stat function is called with stat(dir_name, statP);, but it should be stat(dir_name, &statP); because stat expects a pointer to struct stat.
7. The directory stream opened by opendir is not closed with closedir.
8. The loop for (dirent dP = readdir(dir); dP; dP = readdir(dir)) should properly handle the end of the directory stream by checking for NULL.
9. const char *name = dP->d_name; does not need const since dP->d_name is already a const string.
10. Given the aim of the program to return number of symlinks in a directory could not be achieved because of the current use of stat instead of lstat.

I.4

	data1 R	data1 W	data2 R	data2 W
u1 runs exec1	Y	Y	N	Y
u2 runs exec1	Y	Y	N	Y
u1 runs exec2	Y	Y	Y	Y
u2 runs exec2	Y	Y	Y	Y

User u1 belongs to primary group g1 and supplementary group g2.
User u2 belongs only to primary group g2. Neither u1 nor u2 are super-users.
-rwxr-xr-x 1 u2 g2 14096 Sep 8 22:38 exec1
-rwsr--r-x 1 u1 g1 44096 Sep 25 01:36 exec2
-rw-r--rw- 1 u2 g2 4012 Sep 23 01:12 data1
-rw-r---w- 1 u1 g1 8222 Sep 8 17:13 data2

From the ls -l we can know that **exec1** has owner u2 with permissions Read, Write, Execute and group g2 with permissions Read and write only and others has permissions Read and execute only.
From the ls -l we can know that **exec2** has owner u1 with permissions Read, Write, Execute and group g1 with permissions Read only and others has permissions Read and execute only. But since setuid is set irrespective of the user it is executed with user u1 permissions.
From the ls -l we can know that **data1** has owner u2 with permissions Read and Write, group g2 with permission Read only and others has permissions Read and write only.
From the ls -l we can know that **data2** has owner u1 with permissions Read, Write and group g1 with permissions Read only and others has permissions Read write only.

Filling the table:
u1 runs exec1;
data1 R: Yes (Read as others or group g2)
data1 W: Yes (Write as others)
data2 R: No (No read permission for others)
data2 W: Yes (Write as others)

u2 runs exec1:
data1 R: Yes (Owner)
data1 W: Yes (Owner)
data2 R: No (No read permission for others)
data2 W: Yes (Write as others)

u1 runs exec2:
data1 R: Yes (u1 can read data1 as "others" and also as a supplementary group g2 member)
data1 W: Yes (u1 can write as "others")

data2 R: Yes (u1, as owner, can read)
data2 W: Yes (u1, as owner, can write)

u2 runs exec2:
Running as u1 due to setuid:
data1 R: Yes (u1, as owner or "others" or group g2 member, can read)
data1 W: Yes (u1 can write as "others")
data2 R: Yes (u1, as owner, can read)
data2 W: Yes (u1, as owner, can write)

I.5

1. cd (changes the current working directory): Must be a built-in command.

This command changes the shell's current working directory. If it were an external program, it would only change the directory for its own process environment and terminate without affecting the shell's environment. Therefore, to affect the shell directly, cd must be a built-in command.

2. pwd (prints the current working directory): Can be an external program, but is typically built-in.

This command prints the directory in which the user is currently working. Although pwd can technically be an external program since it simply queries and returns the current directory, it is typically built-in for efficiency and is universally available as such in shells.

3. exec (replaces the shell by another program): Must be a built-in command.

This command replaces the current shell process with a new program. If exec were implemented as an external program, it would run in a new process, unable to replace the shell process from which it was called. Hence, exec needs to be built-in to perform its intended function of replacing the shell process with another program.

4. exit (exits the shell): Must be a built-in command.

This command is used to terminate the shell. If it were external, the shell would need to wait for the external program to finish executing to terminate, which contradicts the immediate nature of exiting a shell. Therefore, exit must be built-in to effectively and immediately terminate the shell process.

I.6

1. Writing non-empty data to a file descriptor will always change the file pointer associated with the file descriptor.

This is not a valid statement. This statement would be incorrect due to the existence of functions like pwrite(), which allow data to be written at specified offsets without changing the current file pointer. This function is particularly used for applications that need to write data to multiple parts of a file concurrently without interfering with each other's file pointers.

2. If a user with a particular user id creates a file and the permissions/ownership on that file are never changed, then that user can also remove the file.

This is a potentially invalid statement. While it might seem intuitive that the creator of a file can also delete it, this action actually depends on the write permissions of the directory containing the file. If the user has sufficient permissions on the directory, they can delete the file. However, directory permissions might restrict this ability, even if the user owns the file.

3. The user and group ownership of a new file is always set to the effective uid and gid of the process which created it.

This is an invalid statement. This statement doesn't always hold true due to system-specific behaviors, such as directories with the setgid bit set, which cause new files created within them to inherit the directory's gid instead of the creating process's effective gid. Thus, the actual behavior can vary based on directory properties and system configuration.

4. Moving a file from one directory to another is done by copying the file and removing the original.

This statement is invalid. When moving files within the same file system. In such cases, the move operation typically involves changing the file's directory entry without physically copying the file data. However, if the move is between different mounted file systems, then it involves copying the file to the new file system and deleting it from the original location.