CO327 : Lab 02

E/14/158 : Gihan Jayatilaka

13-11-2018

Exercise 1.1

(a). Explain what flags *O_WRONLY, O_APPEND and O_CREATE* do.

All these are flags.

- **O_WRONLY:**
  Open for writing only.

- **O_APPEND:**
  The new data that is written to the file will be written to the end of the file. This is done by setting the offset to the end of the file.

- **O_CREATE**
  A new file is created by the file name is the file is not existing. If the file exists, nothing happens. If a new file is created, it will be owned by the user of the process in which open() is called.

(b). Explain what the modes *S_IRUSR, S_IWUSR* do.
All these are modes.

- **S_IRUSR**
  This is the symbolic flag of the bit which dictates the user read permission.

- **S_IWUSR**
  This is the symbolic flag of the bit which dictates the user write permission.

Exercise 1.2

(a). Write a program called mycat which reads a text file and writes the output to the standard output

mycat.c

(b). Write a program called mycopy using open(), read(), write() and close() which takes two arguments, viz. source and target file names, and copy the content of the source file into the target file. If the target file exists, just overwrite the file.

mycopy.c

Exercise 2.1
(a). What does write(STDOUT_FILENO, &buff, count); do?

This is the lower level system call that can be used to print some characters to the standard output. This can be done on a few levels.

1. printf("%s",buff)
   This is the layman method of printing a string in buff without specifying the length you need to print. The length is calcuylated as the number of characters until a NULL termination.

2. ("%.*s",count,buff);
   This is the way to specify the length to be printed in the printf function.

3. fprintf(stdout,"%s",buff);
   fprintf function provides more functionality than printf. Here, we can print to any FILE*. The standard output is also considered as FILE* stdout.

4. fprintf(stdout,"%.*s",count,buff);
   Same as above function but can specify the length of the string to be printed.

5. write(STDOUT_FILENO, &buff, count);
   The low level system call that enables all the above functions. This function is wrappped inside all of them.
   This function accepts a integer file descripter instead of a FILE*. STDOUT_FILENO is the constant integer corresponding to the standard output. This can be manually obtained as
   int fileDescriptorStdOut = fileno(stdout);

(b). Can you use a pipe for bidirectional communication? Why (not)?

The pipes cannot be used for bi directional communication because once a person writes to the write end of the pipe, it is received by the read end of the pipe for everyone. Because of this a pipe can be used only for unidirectional communication.

(c). Why cannot unnamed pipes be used to communicate between unrelated processes?

The unnamed pipes are accessed by file descriptors. There is no way to copy the file descriptor other than forking the process after assigning the file descriptors in memory.

(d). Now write a program where the parent reads a string from the user and send it to the child and the child capitalizes each letter and sends back the string to parent and parent displays it. You'll need two pipes to communicate both ways.

ex21d.c

Exercise3.1
Write a program that uses fork() and exec() to create a process of ls and get the result of ls back to the parent process and print it from the parent using pipes. If you cannot do this, explain why.

This cannot be done since exec will replace everything in the parent process by the new shell command's process.

Exercise3.2
(a). What does 1 in the line dup2(out,1); in the above program stands for?
1 is the file descriptor of the standard output.

b. The following questions are based on the example3.2.c
i. Compare and contrast the usage of dup() and dup2(). Do you think both functions are necessary? If yes, identify use cases for each function. If not, explain why.

dup() and dup2() takes in a file descriptor, assigns another file descriptor to it and returns it. The difference comes in what the new file descriptor is.

- dup(int oldFileDescriptor)
  This function assigns the smallest numbered file descriptor that is not being used.
- dup2(int oldFileDescriptor, int newFileDescriptor)
  The function tries to assign the oldFileDescriptor to the newFileDescriptor.

If the newFileDescriptor is being already used, it is closed and used for the function.
If the newFileDescriptor is invalid, the function aborts.
If both file descriptors are being used for the same, the function completes without doing anything.

Yes. Both dup() and dup2() have their own usecases.
dup() is useful if we want to get a new file descriptor without overlaping with the file descriptors that are in use at the moment.
dup2() is useful if we want to assign a file descriptor to a known file descriptor (usually to modify an existing file descriptor)

ii. There's one glaring error in this code (if you find more than one, let me know!). Can you identify what that is (hint: look at the output)?

On appearent reason is that calling
close()
dup()
in that order on two asynchronous child and parent processes will have a race condition where a process may use the file descriptor from another process.

Please note: This does not happen because file descriptors are in the process and this race condition does not actually occur.

iii. Modify the code to rectify the error you have identified above.
ex32b.c

(c) ex32c.c

Exercise4.1:
a. Comment out the line "mkfifo(fifo,0666);" in the reader and recompile the program. Test the programs by alternating which program is invoked first. Now, reset the reader to the original, comment the same line in the writer and repeat the test. What did you observe? Why do you think this happens? Explain how such an omission (i.e., leaving out mkfifo()function call in this case) can make debugging a nightmare.

- When mkfifo is removed in the reader:
  - If the reader is run first, it will wait until the writer is run and the

message is written from writer. Then the reader will display the message and both programs will close.

  - If the writer is run first, it will wait until the reader is run and reads the message. Then the reader will display the message and both programs will close.

- When mkfifo is removed in the writer:
  - If the reader is run first, it will wait untl the writer is run and the message is written from the writer. Then the reader will display the message and both programs will close.
  - If the writer is run first, it will not wait until the reader reads the message. The message will be lost.

mkfifo makes a special file which needs to be open on both ends (reading and writing) for a process to write to it. This makes read() write() functions blocking until open() is called in the other process.

The if mkfifo is omitted, it will be difficult to debug since the error is a race condition, which cannot be seen all the time.

b. Write two programs: one, which takes a string from the user and sends it to the other process, and the other, which takes a string from the first program, capitalizes the letters and send it back to the first process. The first process should then print the line out. Use the built in command tr() to convert the string to uppercase.

ex42user.c
ex42capitalizer.c