E/15/366

S.THINESH

Interprocess Communication

Exercise1.1:

a. Explain what the flags O_WRONLY, O_APPEND and O_CREAT do.

- O_WRONLY - Open for writing only.
- O_APPEND - If set, the file offset will be set to the end of the file prior to each write.
- O_CREAT - If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file is created, the user ID of the file is set to the effective user ID of the process.

b. Explain what the modes S_IRUSR, S_IWUSR do.

- S_IRUSR - Read permission bit for the owner of the file. On many systems this bit is 0400
- S_IWUSR - Write permission bit for the owner of the file. Usually 0200.

Exercise1.2:

a. Write a program called mycat which reads a text file and writes the output to the standard output.
Attached as 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.
Attached as mycopy.c

Exercise2.1:

a. What does write (STDOUT_FILENO, &buff, count); do?
write() writes up to *count* bytes from the buffer starting at *buff* to the file referred to by the file descriptor *STDOUT_FILENO*.

b. Can you use a pipe for bidirectional communication? Why (not)?
You can use a pipe for bidirectional communication if both processes keep both ends of the pipe open. You need to define a protocol for whose turn it is to talk. This is highly impractical.

c. Why cannot unnamed pipes be used to communicate between unrelated processes?
Unnamed or anonymous pipes provide a means of one-to-one, one-way inter process communication between different processes that are related by either a parent-child relationship. Therefore if a processes are unrelated then the unnamed pipes can't be used.

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.
Attached as capitalize.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.

A program cannot be written. The relation of parent and children is lost while executing fork and exec() to create a process of ls and get the result of ls back to the parent process because original code was edited. Unnamed pipes cannot be used to communicate between unrelated processes.

Exercise3.2:

a. What does 1 in the line dup2(out,1); in the above program stands for?
dup2 is a system call similar to dup in that it duplicates one file descriptor, making them aliases, and then deleting the old file descriptor. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the old file descriptor, performing the redirection in one elegant command.

b. The following questions are based on the example3.2.

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: int dup(int oldfd); uses the lowest-numbered unused descriptor for the new descriptor.

dup2 : int dup2(int oldfd, int newfd); makes newfd be the copy of oldfd. dup2 use newfd but dup use only the lowest numbered unused descriptor.

So, For copying the descriptor which is not the lowest-numbered use dup2
Moreover dup2 can be always used instead of dup

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)?
Child is waiting for more inputs but parent exist before the child

iii. Modify the code to rectify the error you have identified above.
Attached as exercise3.2.c

c. Write a program that executes "cat fixtures | grep | cut -b 1-9" command. A skeleton code for this is provided as exercise3.2.c_skel.c. You can use this as your starting point, if necessary.

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.

   Comment out mkfifo() in reader
   Run reader after that writer : no message received
   Run writer after that reader : original message received

   this is because mkfifo() function declares a named pipe. If a file is created as a named pipe then reader won't wait till the fifo gets filled. empty file will be read. This doesn't occur when writer runs at first. Because reader will read after writer wrote something.

   Comment out mkfifo() in writer
   Run reader after that writer : original message received
   Run writer after that reader : no output

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.

   Attached as exercise4.1.reader.c and exercise4.1.writer.c