E/15/156
Jayathilaka H.A.D.T.T.

CO327
Operating Systems
Lab 02 : IPC

Exercise1.1:

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

O_WRONLY : Open for writing only
O_APPEND : The O_APPEND mode causes all write actions to happen at the end of the file.
O_CREAT : O_CREAT is set and either the path prefix does not exist or the path argument points to an empty string.

b. Explain what the modes S_IRUSR, S_IWUSR do.

S_IRUSR : Read permission bit for the owner of the file.
S_IWUSR : Write permission bit for the owner of the file.

Exercise1.2:

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

Code: 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.

Code: mycopy.c

Exercise2.1: Look at the code in example2.2.c and answer the following questions.

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

The *write*() function shall attempt to write *count* bytes from the buffer pointed to by *buff* to the file associated with the open file descriptor, *STDOUT_FILENO.*

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

No, because everyone can receive the message that one person writes to the write end of the pipe, from the read end of the pipe. So that a pipe cannot be used for bidirectional communication.

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

Unnamed pipes can only be used for communications between processes on the same machine and they exist only when the processors are communicating with each other. So, an unnamed pipe

cannot be accessed outside the process which created it. Because of that unnamed pipes cannot be used to communicate between unrelated processes

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.**
   Code: exercise2_1_d.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.

We can't create a process of ls and get the result of ls back to the parent process and print it from the parent using pipes because exec will replace everything in the parent process when creating 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 stands for 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.

The dup() system call creates a copy of a file descriptor. It uses the lowest-numbered unused descriptor for the new descriptor.

The dup2() system call is similar to dup() but the basic difference between them is that instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified by the user.

Yes, I think both are necessary. dup() is useful if we want a new file descriptor without overlapping with the file descriptors which are in use. dup2() is useful if we want to assign a file descriptor to a known 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)?

We can't call close() function and duo() function in this order because this will create a confusion in between child and parent process. Due to this different file descriptors can be might be use in some processes.

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

Code: exercise3_2_b.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.

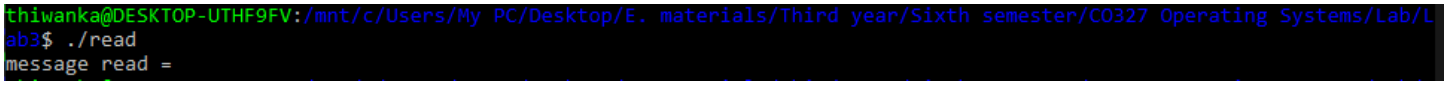
Code: exercise3_2_c.c

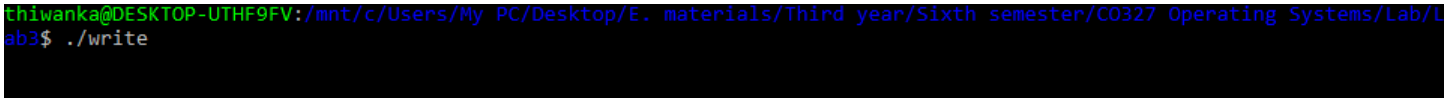
Exercise4.1:

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.

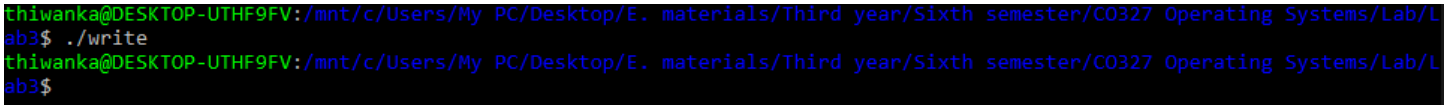
Task 1

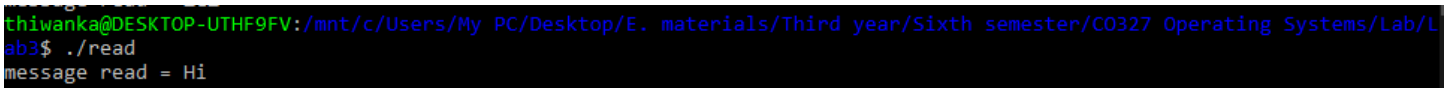
When reader is invoked first, Reader looks for an value and since no value is given, return nothing. Writer writes the message and waits for reader to read it.





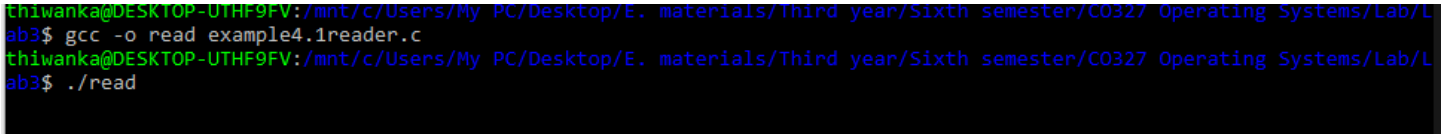
When writer invoked first, then reader, Writer writes the message and returns the message for reader to read it. Then reader read the message.

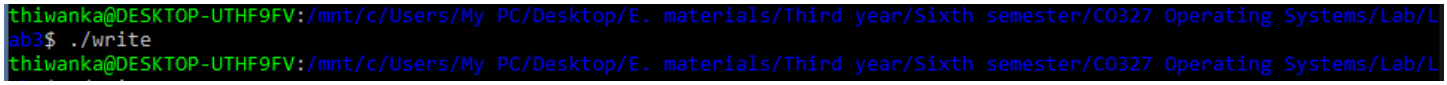


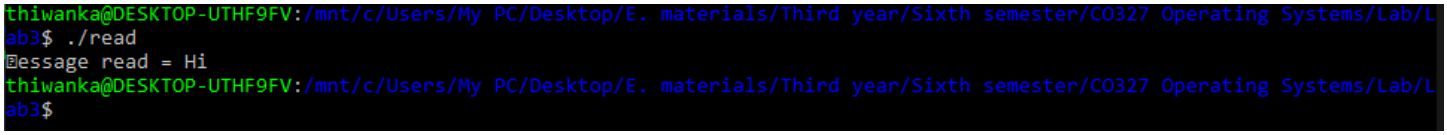


Task 2

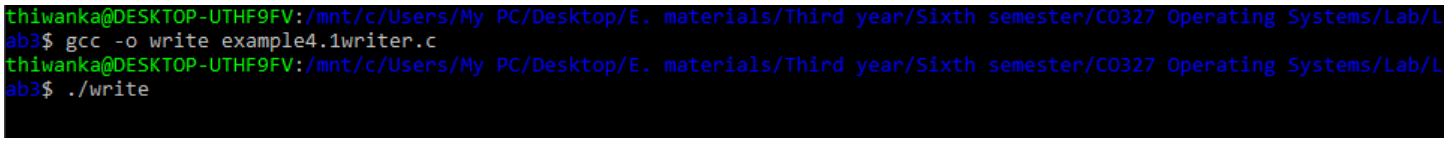
When reader is invoked first, reader waits while a value is written by writer. Writer writes the message and reader read it and exit.

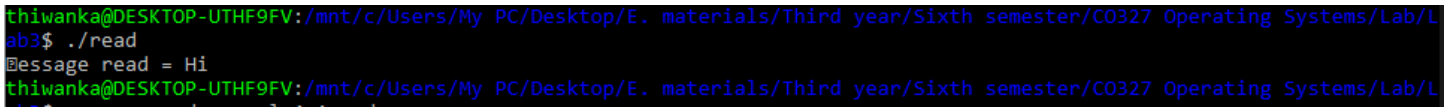






When writer invoked first, Writer writes the message and wait for reader to read it. Then reader read the message and exit.





mkfifo cannot be omitted because that makes a special file which needs to be open on reading end and writing end for a process to write to it. So that until open() is called, read() write() function are blocked. To debug this we need mkfifo.

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.

Code: exercise4_2_writer.c

Code: exercise4_2_reader.c