NP Assignment-1 Design Document

P1. Custom Shell

Extracting Commands:

For parsing the commands we are tokenizing the entered command in two parts.
(i) Tokenizing based on Special Symbol
(ii)Tokenizing based on Command Arguments

We save the special symbol to take pipelining decisions while executing the commands. And the command arguments are passed to exec() after parsing it into an array of arguments.

Job Control:

1. **Foreground Jobs:**
   While a job is running in the foreground it has the control of the shell which is given using tcsetpgrp() function which also allows the signal to be passed to the running job only rather than passing it to the shell. Using Ctrl + Z a running job can be stopped. When this happens the shell makes the entry of this process in a queue structure maintaining all the background tasks. And the control of the terminal is given back to the shell.

2. **Background Jobs:**
   Using & operator jobs can be run in the background which automatically adds the job to the background queue. When any task running in the background finishes execution the shell gets the SIGCHLD signal, and if the child successfully executes it is removed from the background queue.
   "*bg*" command only resumes a process running in the background using the SIGCONT signal and doesn't alter the background queue.
   While *fg* command resumes a process and also removes it from the background queue and give that process control of the terminal.

Pipelining Commands:

The shell supports the pipelining of commands wherein the output of one command is sent as input to the next command and so on in a cascading manner.

cmd1 OP cmd2 OP cmd3 … so on

The **OP** represents one of the three IPC mechanisms to achieve the desired effect.

1. Using Pipe
2. Using Message Queue
3. Using Shared Mem

1. **Pipeline Using Pipes:**
   The design of the custom shell dictates that for the first (*N - 1)* commands, children are spawned which executes them in a sequential manner, meanwhile, the parent forwards the read file descriptor to the next command using which the next command receives its input. This goes on for *(N-1)* commands.

   The last command is executed by the parent as it is, and the output is received on the STDOUT.

2. **Pipeline Using Message Queue:**

   MESSAGE QUEUE for Normal Usage ***: cmd1 # cmd2 #….***

   The message queue based pipelining follows a different design concept because of its inherent incapability to support File Descriptors.

   Our Custom shell uses PIPE + MESSAGE QUEUE to achieve the desired functionality. The basic iteration remains the same, i.e. consisting of (N-1) spawned children for first (N-1) commands. However, for communicating, the processes output to a pipe which is read and written to a message queue created by the parent. The parent, in turn, sends the message to the next spawned child. The child reads the message received and writes it into another pipe from where the process uses it as STDIN while duplicating the *write fd* of the pipe created by the parent as STDOUT.
   This write fd can be forwarded either directly or in the above manner according to the upcoming operator in the chain.

   MESSAGE QUEUE for MULTIPLE REPLICATION: ***cmd1 ## cmd2, cmd3***

   The multiple replications of output from the 1st command to subsequent commands is achieved by saving the intermediate output after cmd1 for the cmd3 and letting the cmd2 output on STDOUT as it is.

3. **Pipeline Using Shared Memory:**

   Pipeline using Shared Memory follows the same design structure as that of Message Queues. The only difference being the use of shared memory.

   SHARED MEMORY for MULTIPLE REPLICATION: ***cmd1 SS cmd2, cmd3***

   The inherent property of shared memory eliminates the need for saving any intermediate output since the data is always there even after reading it.

NOTE: The aforementioned operators can be used in any allowed combination along with Redirection operators. This is achieved by using a pipe's FD as an interface among various IPCs throughout the chain of commands and forwarding appropriate descriptors for the next command in the chain.

<u>Daemonize:</u>

For daemonizing the task we use *setsid()* to separate the job from the controlling terminal we also close its std sockets. But we did not change the working directory to "/" so that we can execute the local programs in the directory.

<u>Redirection:</u>

`INPUT REDIRECTION:` ***`cmd1 < file`*** *`OP cmd3 OP cmd4…`*

The input redirection mode in the shell expects an existing file on the right side of the "<" operator. ***`cmd1 < file`*** is considered as a single command and is executed by a single child which first opens the file and duplicates its file descriptor to STDIN. Subsequently, upon executing the command, the input is taken from the file and written to an existing file descriptor which is forwarded to the remaining pipeline just like in any of the aforementioned cases by the parent.

`OUTPUT REDIRECTION :` ***`cmd*OP* > file`*** *`OP cmd3 OP cmd4…`*
`(cmd*op* : chain of commands)`

The ">" operator writes the output of the processed pipeline of commands to the file on the right side of the operator. This is done simply by opening a file and duplicating its fd to STDOUT before executing the previous command. We now skip the "file" in our commands array since it has been taken care of along with the previous command.

The subsequent commands are processed as a completely new pipeline.

Additional Attempts:
Two functions useSharedMemory() and useMessageQueue() are part of code but not used in it follows a different design in which parent just manages the messages and n child executes the pipeline of commands and each of the function can only handle pipeline of one type only. We later used a more generic approach.