

**Lab 3 Report**

**Report Subject: OS Experiment - Lab 3**

**Student ID: 2018380039**

**Student Name: Dikshya Kafle**

**Submission Date: 2021/11/18**

**Computer Operating System Experiment**

**Laboratory 3**

**Inter-Process Communication**

1. **Objective:**

Internalize a couple of important facts about IPC in Linux. Including:

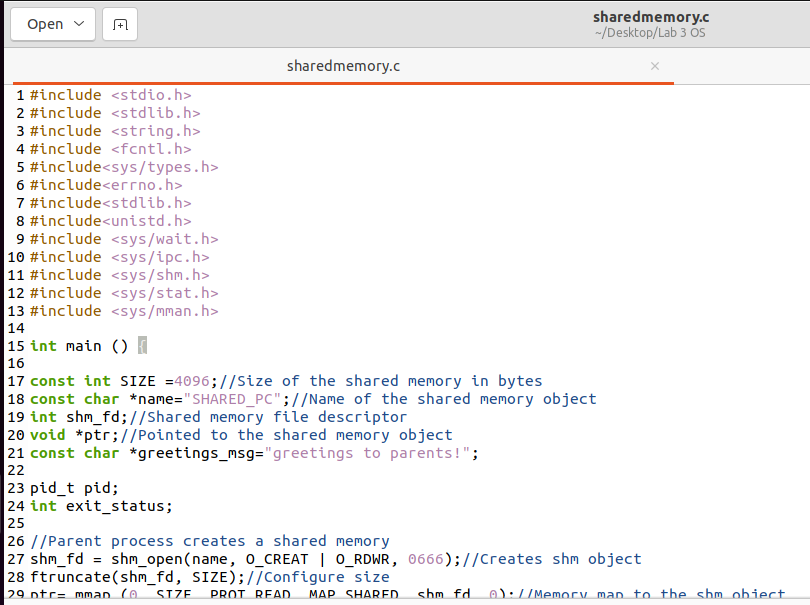
* Shared memory
* Ordinary Pipes
* Named Pipes

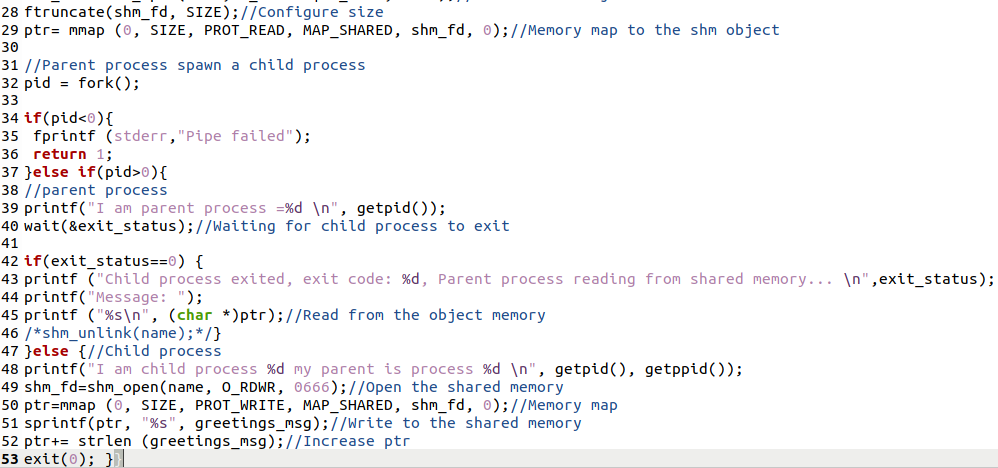
1. **Equipment:**

* VirtualBox with Ubuntu Linux

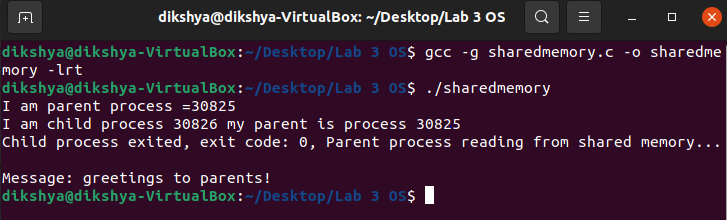
1. **Experiments:**
2. **Experiment 1: shared memory communication**

* The code:





* The results:



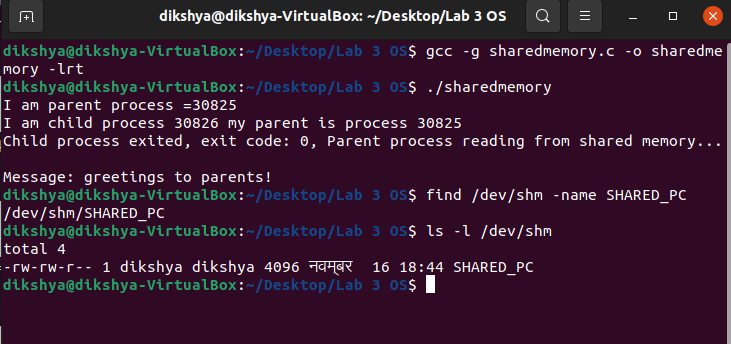
* Explanation:

The Parent process creates a shared memory then it spawns the child process. The Child process then runs and write a “greeting to parent!” message to shared memory.The Parent process waits child process to terminate, if the exit code == 0, then read the message from shared memory.

As the parent process will only read the shared memory we use PROT\_READ (We can also use PROT\_WRITE cause we opened it using both read and write permissions). As the child process will write the shared memory we use PROT\_WRITE (which implies PROT\_READ as specified in the man page).

* Questions: On Linux, all shared memory objects can be found in /dev/shm. Can you find the shared memory objects that?

Answer: First, as we have used shm\_unlink(name), this will delete the shared memory region given a file descriptor (name). So to answer this question, I commented that line than used the find command to find the shared memory object that we named SHARED\_PC (or I could also run ls -l /dev/shm and try to find it by myself by looking for it)

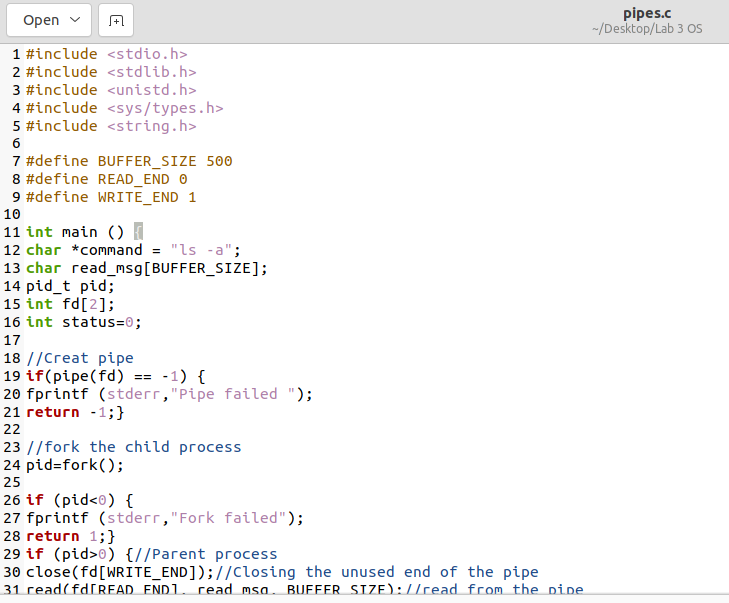


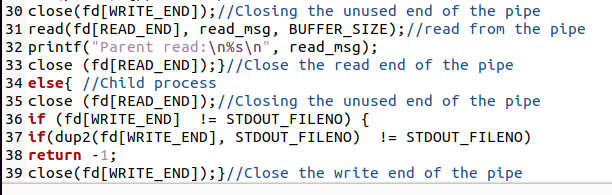
1. **Experiment 2: ordinary pipe communication**
2. **Write an ordinary pipe between parent process and child process.**

The program first creates a pipe, and then creates a child process of the current process through fork(). Then each process closes the file descriptors that are not needed for the read and write pipes. The child process executes the "ls -a" command under the current path, and writes the command execution output to the pipe by copying the pipe write descriptor fd[WRITE\_END] to standard output; the parent process reads the pipe data and displays it through fd[READ\_END].

As the standard output becomes fd[WRITE\_END], the output of the command “ls -a” is written in the write end of the ordinary pipe. As we can see as a result, the parent process reads the names of files in the same directory.

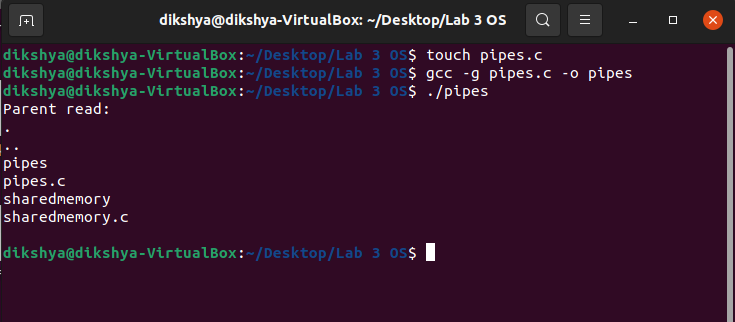
* The code:

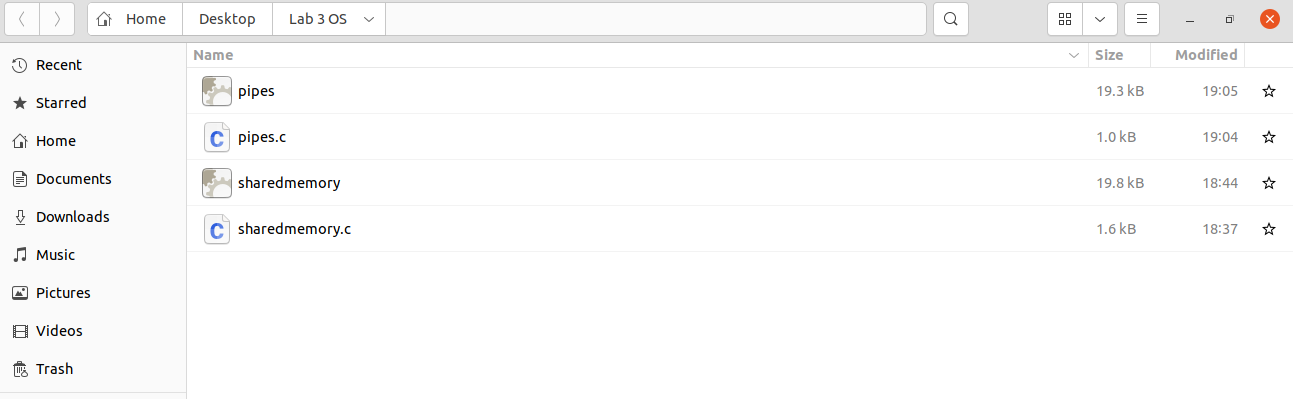




* The result:

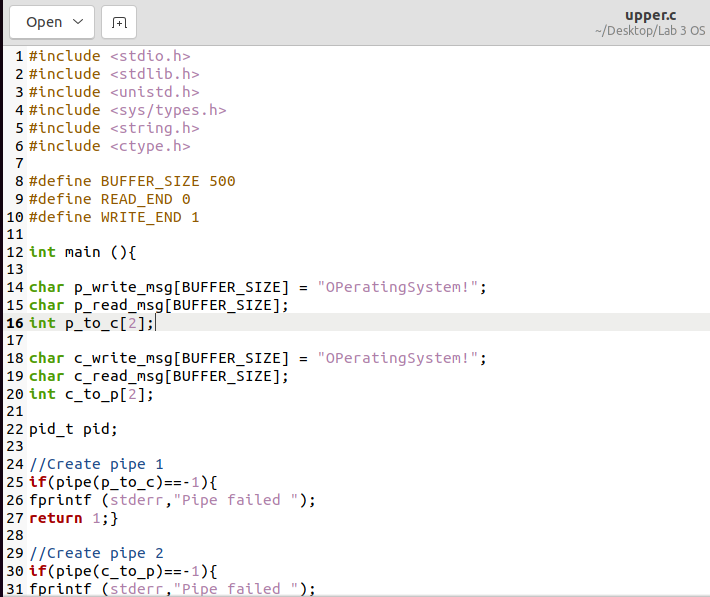
As we can see, the content of the directory matches what the parent process has read.

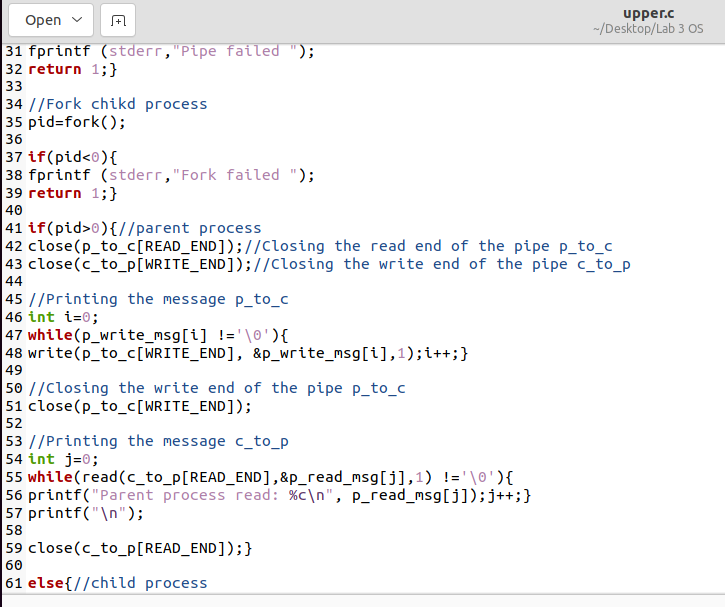


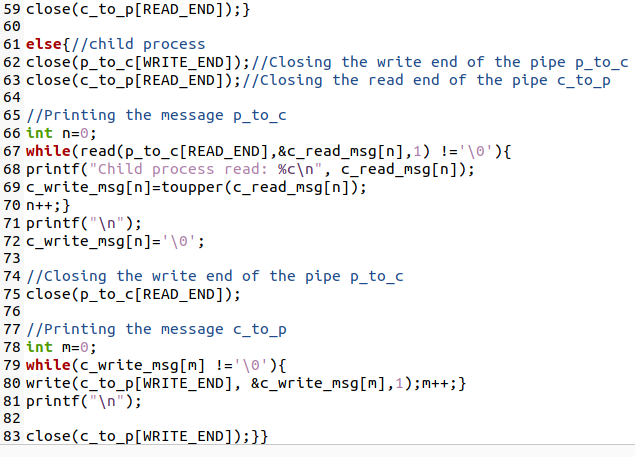


1. **Write a two-way ordinary pipe:**

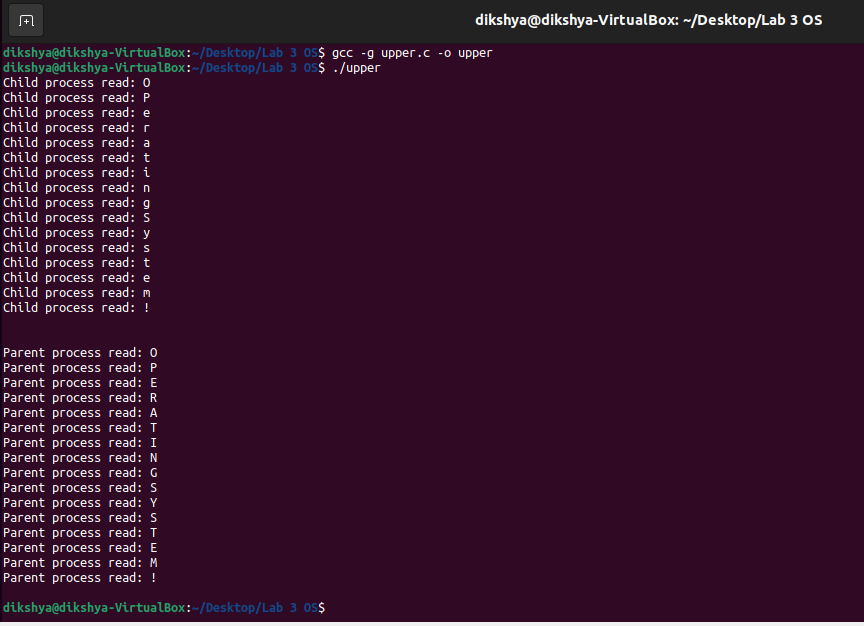
* The code:







* The results:



* Questions:

1. Does the pipe allow bidirectional communication, or is communication unidirectional in an ordinary pipe?

Answer:

Ordinary pipes are unidirectional, allowing only one-way communication.

1. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

Answer:

Two-way communication is not allowed when working with ordinary pipes.

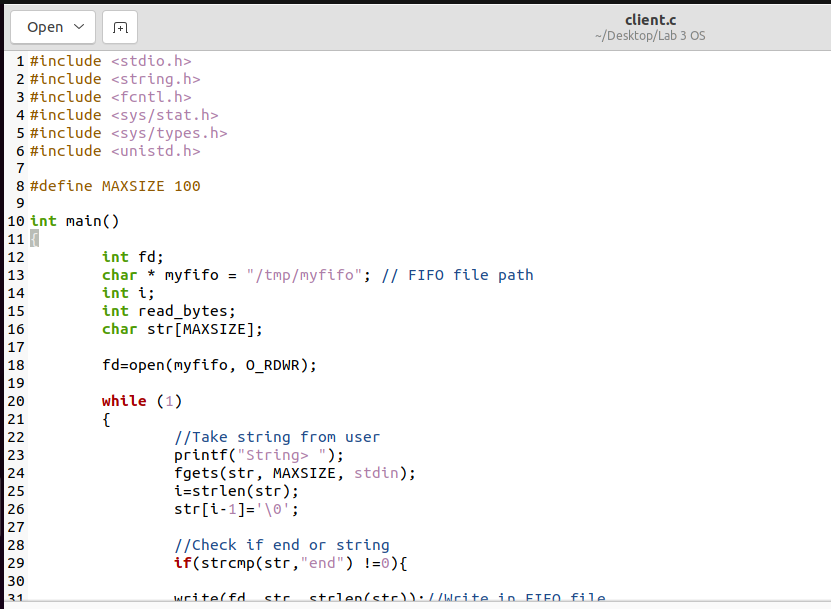
1. Must a relationship (such as parent–child) exist between the communicating processes in an ordinary pipe?

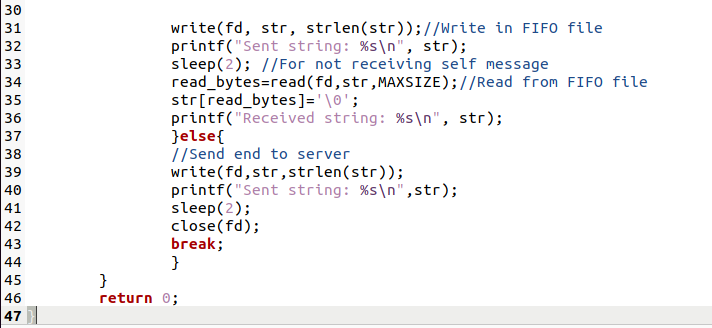
Answer:

Ordinary pipes require a parent-child relationship between the communicating processes. They cannot be accessed from outside the process that created it. A parent process creates a pipe and uses it to communicate with a child process that it created.

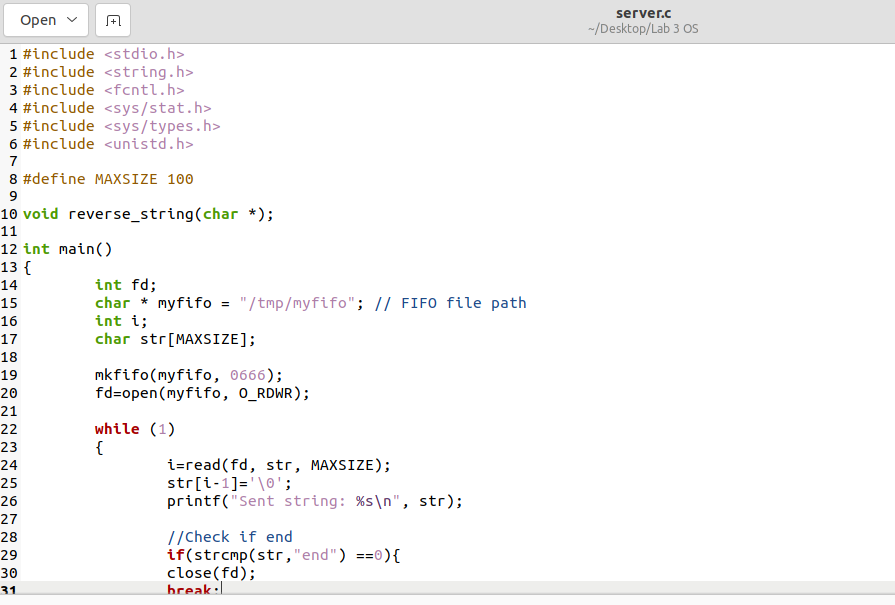
1. **Experiment 3: Named pipe communication**

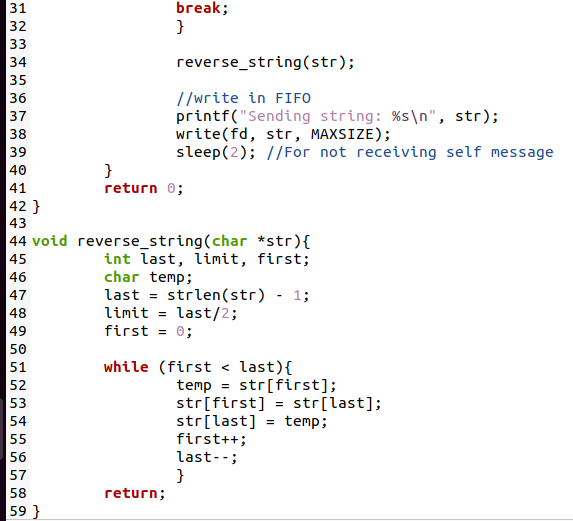
* client.c**:**



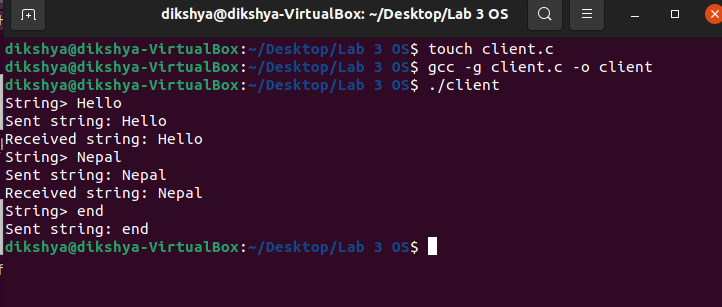


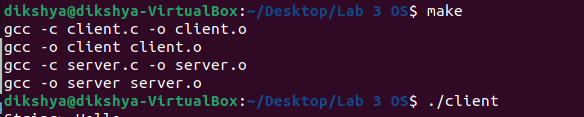
* server.c:

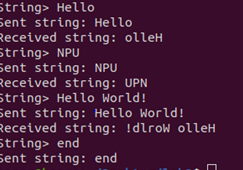


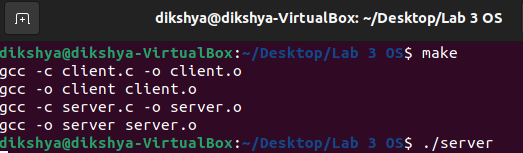


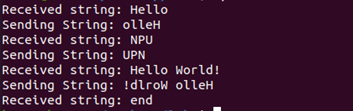
* Results:











The server creates a named pipe with permissions of read and write using library function mkfifo() with name “fifo” in /tmp directory, if not created. It then Opens the named pipe for read and write purposes and waits infinitely for a message from the client.

If the message received from the client is not “end”, it prints the message and reverses the string. The reversed string is sent back to the client. If the message is “end”, closes the fifo and ends the process.

The client process opens the named pipe for read and writes purposes. It takes an iput string from the user and checks if the user enters “end” or other than “end”. Either way, it sends a message to the server. If the string is “end”, this closes the FIFO and also ends the process. If the message is sent as not “end”, it waits for the message (reversed string) from the server and prints the reversed string. This is Repeated infinitely until the user enters the string “end”.

Note, we should take care of the EOF otherwise we get some wrong results. Also if we do not put process in sleep after writing in FIFO file, it will read its own message.

* Question:

What are the advantages of using named pipe?

Answer:

A Named pipe that can be used for two-way communication and supports bi-directional communication. Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required.

Also, Multiple users can send requests through the same named pipe and each request is removed from the pipe as it is received. In addition, the message handler procedure can loop indefinitely looking for input because it blocks (waits) until there is something to read. Finally, output through named pipes is more efficient than writing a complete response to an ordinary file, closing the file, and then informing the recipient that the results are available. The receiving process can read the result through a named pipe as soon as it is written.