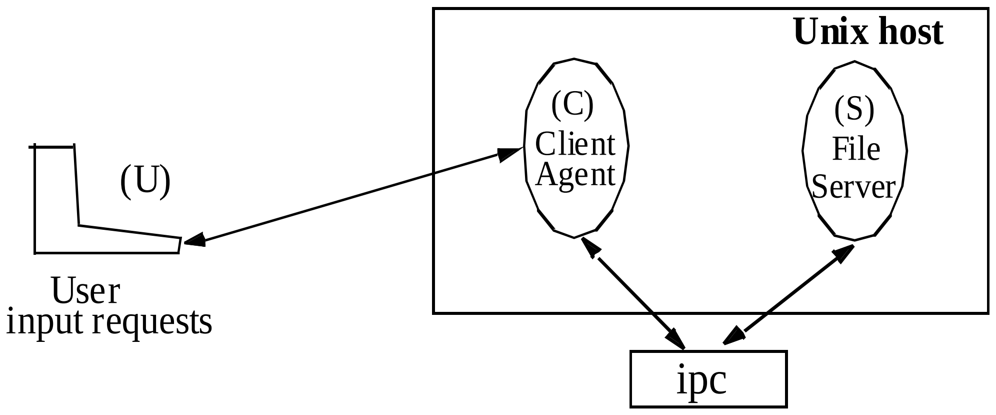
Santa Clara University

Department of Computer Engineering
(COEN 236)

Project-1

Project Overview:

The goal of this project is to implement a primitive mechanism/capability for two UNIX processes on the same host to communicate through the UNIX file system. This mechanism is similar to what is known as UNIX pipes. The purpose of this project is to deal with synchronization issues and exercise your skills in UNIX programming rather than developing a new usable IPC mechanism.



Overview:

In this project, we would like to build a simple **File Server (S)** and a **Client** agent (**C**) processes as client and server respectively. The client agent process reads input commands from the user (**U**), passes the operation required to the **file server (S)** through our IPC mechanism, gets the result back and pass it back to the User (**U**). The interactions between the **User**, **Client**, and the **File Server** are synchronous, i.e., the user has to wait to receive the result of an operation before requesting another operation, etc.

<u>Functional Requirements</u>:

1. The client agent (**C**) process reads file-based operations, one command at a time, from the input device (terminal) as a string <**Opcode** filename>.
2. (**C**) writes the command to a disk file in a format understood by both (**C**) and (**S**).
3. (**C**) signals (**S**) and goes to sleep.
4. Once (**S**) detects the signal, it reads the Opcode and possibly a filename from the shared file, executes the required operation, writes the result back to the disk file, signals (**C**) and goes back to sleep.
5. (**C**) Catches the signal, read the result from the file and writes it to the output device, etc.
6. The agent needs to recognize an exit command when user is done. (**C**) needs to do cleaning up including killing (**S**), etc.

<u>Programming Model</u>:

1. (**C**) forks and the child execs the File Server executable.
2. Both (**C**) and (**S**) share a file as the logical IPC medium.
3. Both (**C**) and the (**S**) have to use **Posix reliable signals**. You cannot afford loosing signals!
4. Necessary commands to be implemented include: (read, delete) file operations, and the exit commands.
5. You can choose any command syntax at the user interface level as well as the message format between the agent and the server processes.

<u>Programming Hints</u>:

1. Your agent process (C) should be structured into four main segments:
   - *<u>Initialization segment</u>*: setting up signals, opening the shared file, creating the server process, etc.....
   - *<u>Command input segment</u>*: read a line command <Opcode filename> where the supported Opcodes include (EXIT, READ, DELETE) from the

standard input, check legal syntax, and branch to execute legitimate commands.

- *Command execution segment*: if Non Exit command, forward the file operation to the file server and waits for the result. If Exit do cleanup, including the server, and terminates.
- *Output segment*: write the operation result on the standard output, and asks for next command.

2. Include one/two pages description of your project implementation. Remember to have good comments in your code. Also, include a hard copy of user interaction with your file server application. Finally, write clearly on the Front page the **absolute Unix pathname** for your application to the grader to try (making sure RWX permissions for others).

3. Bring to class two complete copy sets of your project (one for the grader and the second for myself).