Lab 2

Interprocess Communication

Due date

Oct 6, 2025 11:59 PM CST

Keywords

Interprocess communication, unnamed pipes, named pipes, file descriptors

Introduction

In this lab, you will modify the existing banking system to use Unix pipes and named pipes (FIFOs) for inter-process communication. The previous implementation used a different communication mechanism in RequestChannel. You will implement both standard pipes and named pipes as options for IPC between the client and server processes.

Starter Code

You are given a source directory with the following files:

- RequestChannel, PipeChannel, and FIFOChannel classes (channel.h and channel.cpp): channel.h declares these three classes, which you will implement with IPC to communicate between the server and client. RequestChannel is an abstract class, while PipeChannel and FIFOChannel are concrete classes (can be instantiated). You must implement the methods for each class in channel.cpp. PipeChannel and FIFOChannel inherit from RequestChannel. As their names suggest, they are implemented using unnamed pipes and named pipes respectively.
- Makefile (Makefile): This file compiles and builds the source files when you type the make command in the terminal. You can use ‘make’ to compile, ‘make clean’ to remove executables, and ‘make distclean’ to remove both the executables and files the executables generate.
- Server programs (finance.cpp, file.cpp, logging.cpp) : These files have the same functionality as in lab 1. However, you will now modify them to use the new RequestChannel.
- Client program (client.cpp): As in lab 1, the client will be responsible for creating new server processes and communicating with them via the RequestChannel class as well as printing process information, which you will implement. However, you will update it to now support communication with the server through PipeChannel and FIFOChannel
- Utilities (common.h): This file contains useful classes and functions shared between the server and the client such as Request and Response definitions. An enum for different types of requests (e.g., a login request) is also defined here. Do not modify these files.
- The storage directory will contain any files managed by the File Management program (file.cpp). To start, it contains the following:
    - example_execution.txt: this is an example of the program running. Use this as a reference for the expected behavior

    - example.log: this is an example of how the log file should look at the end of example_execution.

Objectives

By the end of this lab, you should be familiar with:

- Using unnamed pipes for process communication, including using pipe(), close() and dup2()
- Using named pipes for process communication, including mkfifo() and open()

Getting Started

Go to the assignment's GitHub classroom: https://classroom.github.com/a/oaO7q9KS

Read through the code. There have been changes made to all files, but the overall structure and function are the same.

Differences from Lab 1

As every lab will build on the same project, there are not many changes from lab 1. The changes that were made are listed below.

Channel.h, channel.cpp

This is the largest change from lab 1. You can now get an idea of how RequestChannel was implemented, and implement it yourself.

RequestChannel

```
class RequestChannel {
protected:
    int rfd;
    int wfd;
    RequestChannel();

public:
    enum Side {SERVER_SIDE, CLIENT_SIDE};

    virtual ~RequestChannel() {}

    int cread(void* msgbuf, int msgsize);
    int cwrite(void* msgbuf, int msgsize);

    Response send_request(const Request& req);
    Request receive_request();
    bool send_response(const Response& resp);
};
```

RequestChannel is now an abstract base class; you cannot create an object of RequestChannel since the constructor is protected. Instead, it simply declares two attributes (write and read fds) and a few methods that you will need to define in channel.cpp.

cread() and cwrite() directly send/receive data through the file descriptors by calling read() and write() syscalls. They should read/write a message msgbuf with size msgsize bytes into the appropriate fd and return the number of bytes read/written.

It also declares send_request(), send_response(), and receive_request(), which should use cread() and cwrite() instead of read() and write() directly. These three methods should be used the same way as in lab 1.

- If send_request fails, you should return a Response with s=false
- If receive_request fails, you should return a Request with type FAILURE
- If send_response fails, you should return false

Two channels inherit from RequestChannel: PipeChannel and FIFOChannel. These two channels should be implemented with their respective IPC type: unnamed pipes and named pipes.

PipeChannel

```
class PipeChannel : public RequestChannel {
public:
    PipeChannel(int _rfd, int _wfd);

    ~PipeChannel();
};
```

The PipeChannel class doesn't declare any new attributes or member functions. Because of the nature of pipe, we cannot create the pipe within the PipeChannel object; the fds need to be part of a pipe before the child is created. The constructor will take in these fds as arguments and handle the closure of the pipe gracefully in the destructor.

FIFOChannel

```
class FIFOChannel : public RequestChannel {
private:
    std::string pipe1, pipe2;

    int open_pipe(std::string _pipe_name, int mode);

public:
    FIFOChannel(const std::string& name, const Side _side);

    ~FIFOChannel();
};
```

The FIFOChannel class declares a new member function “open_pipe” and two new attributes, pipe_1 and pipe_2 which are strings.

The open_pipe() function will create a fifo from a _pipe_name parameter using mkfifo and then open it according to the mode parameter. It should return the corresponding fd. For example, if mode was set to O_WRONLY (this can be considered an int), the fifo should be opened with O_WRONLY.

The constructor should assign the appropriate fd’s depending on the channel side using open_pipe. The names will already be supplied for you in pipe1 and pipe2.

The destructor should handle the closure of the fifo including anything created in the constructor/open_pipe() gracefully.

Common utils

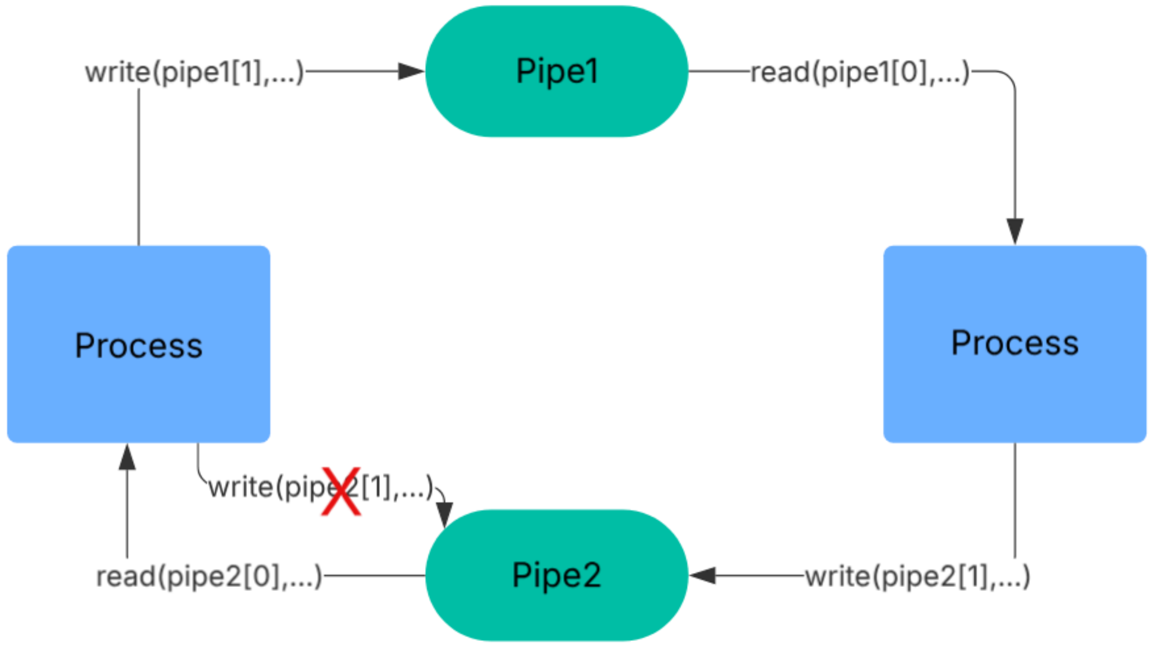
Common.h is now the only common file, since we have removed the need to parse requests as we did in lab 1. Now, Request and Response structs’ strings have been changed to char arrays so that pointers to them properly act as buffers.

```
Request example_req;
cread(&example_req, /*size*/); // pseudocode, this will not work directly
```

Client and Servers

There aren’t many changes here. The only changes were made to be able to implement the new version of IPC, most of which will be implemented by you, and should be obvious if you look through the files.

Pipes



Pipes are a form of interprocess communication. You can think of them as buffers with a limited amount of free space. They are unnamed, meaning they can not be accessed by a process that has not inherited it or created it. They are unidirectional, meaning if you want bidirectional communication between pipes, you need to use multiple pipes. Pipes have blocking behavior, so you should handle each end of every pipe carefully.

Named pipes have a few distinct differences. First, like the name implies, they are named. This means they can be accessed by a process that is unrelated to the process that created it. They also survive beyond the lifecycle of the process that creates it.

Refer to the slides, readings, or online resources such as this to learn more.

Tasks

Implement RequestChannel (25 pts)

- cread(), cwrite() must be implemented with read() and write() syscalls, and return the size of what was written/read in a ssize_t variable
- send_request(), receive_request(), and send_response() should be implemented using cwrite() and cread()
  - You should not use read() and write() syscalls directly to implement these 3 methods
  - You should handle errors if cwrite() or cread() do not write/read the appropriate number of bytes according to the specifications above

Implement PipeChannel (25 pts)

- The constructor should take in the read fd and the write fd, already created earlier in the program, and set the attributes of the channel
- The destructor should gracefully handle the end of the channel. The ends of the pipe on the channel's side should not be open after the destructor.

Implement FIFOChannel (25 pts)

- open_pipe() should create a named pipe with the same name as _pipe_name and open it with "mode" flags.
- The constructor should take in the read fd and the write fd, already created earlier in the program, and set the attributes of the channel
- The destructor should gracefully handle the end of the channel. The ends of the pipe on the channel's side should not be open after the destructor. The FIFO should not persist in the filesystem after the destructor.

Update Client to Use New Channels (15 pts)

- Two pipes should be created for each connection (e.g. two for the client-fileserver connection). You can use the fd's we set up for you around line 57.
- The ends of the pipes should be appropriately closed and dup2'ed to perform safe communication.
    - Remember, for bidirectional communication using pipes, two pipes must be used. If the unused ends are left open, you may get undefined or blocking behavior.Create the PipeChannel and FIFOChannel objects according to the IPC mode selected. Do this by calling the constructors implemented in channel.cpp.
- Create the PipeChannel and FIFOChannel according to the IPC mode selected. Call the constructors implemented in channel.cpp.
- Clean up the channels after exiting the program

Update Servers to Use New Channels (10 pts)

- Create the PipeChannel and FIFOChannel according to the IPC mode selected. Call the constructors implemented in channel.cpp.
- Carefully consider how to get the fd's when using pipe
    - Hint: After exec, one of the few things a process keeps are its standard file descriptors, stdin, stdout, and stderr.

Testing

As always, you are encouraged to develop your skills as a programmer by testing your code yourself, using a debugger gdb, and other methods of independently testing your code.

For this lab, you will primarily check your progress through the autograder on github, but we may add some additional tests as well. These tests expose some solution code, so we cannot expose them to you. You will be provided a test script in your repository for black box testing, which will attempt to run your whole program. Note that this local autograder is not representative of your final score, nor is it to be used for debugging purposes.

Deliverables (Important)

To receive a grade, push your code to your github repository created by accepting the assignment in the github classroom. The autograder score will be your score for the assignment.

Issues reporting

If you encounter a problem with the lab, please alert the course staff.