Comp517 Midterm Report

Tom Pan, August Pokorak

1 Introduction

Processes, as defined in the seminal introduction of UNIX, are one of the key abstractions of execution on computers [**?**]. In essence, they are comprised of program code and a current execution environment (such as register values, stored data, etc). Since processes have separate virtual address spaces and do not share non-code data, it is more difficult to allow separate processes to collaborate than it is for separate threads of execution running within the same process. Many different forms of inter-process communication (IPC) have thus been developed over the years. The most basic is perhaps the hierarchical file system, one of the first examples of which was the Multics system. In Multics, all memory was part of the file system (no temporary process memory at all) [**?**]. In the Unix approach which has seen far more widespread use, permanent files are instead separate from process virtual memory, and each running process has an associated set of file descriptors corresponding to currently open files [**?**].

One particularly desirable form of IPC is for the output of one process to be used as the input into another process. Of course, a file system could be used to provide this by having processes always write their outputs to stored files, which can then be read by other processes. However, this is clearly inefficient in terms of space usage, for example if an output is immediately used by only one other process. An alternative approach once again popularized by Unix is the concept of a (anonymous) pipe [**?**]. A Unix pipe is a temporary memory construct, associated with pair of file descriptors (one for reading and the other for writing) which are returned by the pipe() system call. These file descriptors can be used to connect the output of one child process to the input of another child. Due to their definition, programmers can continue using the established file system API when working with pipes. The concept of piping is also closely tied to the Unix shell, and greatly improves its usability.

Our primary objective is to implement a simple version of the Unix pipe() system call, in order to better comprehend a form of inter-process communication. To allow us to evaluate our implementation's functionality and time performance across different workloads, we also plan to create a basic shell to operate within the Linux environment. Given that Unix-style piping is intricately linked with the fork/exec method for new processes, we believe implementing the shell in languages other than C may prove challenging due to the intricacies of working with certain system calls.

While we aim to maximize attributes like simplicity, performance, security, programmability, and reliability, it is a given that not all can be optimized equally. Our exploration therefore revolves around understanding different approaches, implementing a system that illustrates necessary tradeoffs, and validating our hypothesis regarding design decision effects. Our success will primarily hinge on functionality—our design must operate as expected. Furthermore, we will utilize both benchmarking performance indicators and qualitative measures, such as programmability, to determine if our approach effectively validates our design rationale.

Through this endeavor, we anticipate shedding light on the intricacies of crafting a pipeline-able shell and contrasting the efficiency of our naive pipes against Linux's current implementation. We hope to be able to elucidate the challenges of implementing OS/systems constructs and the repercussions of various implementation decisions on time metrics.

2 Background

The concept of piping different programs was perhaps first mentioned by Doug McIlroy in a brief document from 1964 - "We should have some ways of coupling programs like garden hose–screw in another segment when it becomes when it becomes necessary to massage data in another way" [**?**]. Also, in 1967 the Dartmouth Time-Sharing System included so-called "communication files", which we would recognize as substantially similar to Unix pipes. Although they were more general, and with some substantial differences including the need for a parent process to manage one end and the notable lack of the "|" operator. Communication files do, however, share the key characteristics of being based on files and being

able to be passed to other processes. Although communication files required management from the parent, they could be treated as normal files by the child, mirroring the usage of Unix pipes as normal files [**?**].

A few years later, Unix would make pipes one of the key features of its shell/command line system. Unix pipes build on Unix files to connect arbitrary processes (without either process knowing that the corresponding file descriptors are not associated with "true" files) while maintaining process isolation. In Unix and many subsequent systems, the "|" character is used to specify a pipe between programs in the shell, and an arbitrary number of programs can be chained together in this manner, forming what is known as a pipeline. Also, many command line programs provided by Unix and its derivatives (e.g. grep) are designed to be relatively simple, and to take input from "standard input", apply some transformation, and output to "standard output", i.e. from and to the shell respectively [**?**]. These small programs (known as filters) can easily be placed within a larger pipeline that performs a complex operation by concatenating smaller ones - at each step, a pipe replaces standard input for the next process and standard output for the previous one.

The piping approach introduced by the Unix system and carried on in Linux has been in widespread use ever since, without substantial changes to the interface. Mac OS, being based on BSD, also makes use of pipe() system calls [**?**], while Windows provides a CreatePipe function [**?**]. In both cases, as expected the pipe has an associated "read" end and "write" end, and standard file system operations are used to interact with the pipe.

In the early implementation of Unix pipes, the memory construct that truly underlies a pipe is just "essentially a section of shared memory that could be accessed by two processes," [**?**] with one writing to the pipe while the other reads from it. The kernel manages the data in the buffer, ensuring synchronization between processes. An interesting detail about the early implementation is that the buffer size per pipe is defined as 4096 bytes, which suggests the value of using a fixed buffer size in our implementation. In line with their usage as normal file descriptors, behind the hood "ancient pipes used a file to provide the backing storage!" [**?**] The pipe system call code snippet demonstrates the allocation of an inode on the root device and two file structures to assemble the pipe with appropriate flags. The pipe() function adjusts file descriptor numbers, flags, and inode references to establish the two ends of the pipe. In the writep() function, mechanisms are laid out to handle various scenarios like writing to a full or closed pipe and managing the data write cycle within the defined buffer size. The inode's i_mode is repurposed to indicate waiting read/write operations on the pipe, marking a shift from its usual role of holding read/write/execute permissions. [**?**]

3 Methods and Plan

As stated, we will attempt to implement pipe creation and in-shell usage in a programming language such as C. Although we would like to use an even higher-level language in order to explore the implications of such a language choice, it may not be feasible to do so in anything other than C or closely derived languages.

One particular property of Unix pipes is that reading from the "read" file descriptor associated with a pipe should block or wait until another process has written to the "write" file descriptor of that pipe; this behavior is particularly tightly coupled to Unix's specific usage of files to represent I/O. We foresee one particularly troublesome aspect of this project being how this behavior can be enforced at the level of the pipe() system call implementation. One workaround solution to this would be to use condition variables or other synchronization constructs to provide the desired behavior in our shell implementation, but perhaps that could be considered too "easy" of a solution. And as in early implementations, another option could be to provide two new system calls, for reading from and writing to pipes, although this will perhaps break the illusion of pipes being treated the same as regular files. We will also need to see if allowing pipelines of more than two processes is feasible, or if we can only implement a single pipe between a pair of "processes".

Evaluating the accuracy of our pipe() implementation should be straightforward as the output of pipelines is known (from simply entering them in the Linux shell). Evaluating performance overheads compared to the Linux implementation may be more difficult; it may require additional effort to enable and run objective performance benchmarks. We may need to either make our shell implementation work in some fashion with the time wrapper command provided by Linux, or hard-code in various pipelines in our "shell" instead of accepting user input. Especially important is ensuring that our metrics are unbiased either towards or against our implementation. To complete this project, we do not anticipate that we need any resources beyond a Linux shell and a particular programming language and associated compiler/interpreter.

4 Evaluation and Results

So far, our main accomplishments have been threefold:

1. Verified the viability of our project
2. Created a program capable of running various pipe implementations on various pipelines
3. Implemented a basic functional pipe

These are all deeply connected, so we will describe them all together. The main controller program, implemented in `ourpipe.c`, meanwhile our first pipe is implemented in

`filepipe.c`, we also have a file `truepipe.c` that wraps a real linux pipe for comparison. The controller takes either command line arguments or prompts the user to select a pipe and pipeline to test on. For testing a pipeline a preset may be selected or any string may be input and tried to be used as a pipe. If the pipeline involves more than one stage, e.g. `ls -l | grep ourpipe | wc` then the controller changes the right hand side of the pipe to recursively call itself, taking the example above if we were using the pipe with index 1 we would change `grep ourpipe | wc` to be `./ourpipe -pipe 1 -pipeline "grep ourpipe | wc"`. This allows for arbitrary length pipes. Our current working custom pipe implementation is based on creating an actual file, which is then given to the child processes to read from and to, allowing communication through the normal filesystem. So far, our project has been successful.

5 Discussion and Lessons Learned

We have made substantial progress in implementing and experimenting with pipe creation and in-shell usage in a programming language, notably C. The endeavor has been challenging yet enlightening in several ways. Firstly, our choice to use C, motivated by its low-level capabilities and closeness to Unix systems, proved to be apt. It allowed us to delve into the intricacies of Unix pipes and their integration with the file I/O system. We though about using a higher-level language, which might simplify certain aspects but could obscure the essential details that C laid bare, so we are happy with our choice of C. Writing a basic functional pipe using actual files was a critical step. It demonstrated the feasibility of our approach and provided a foundation for further experimentation. This aspect of the project highlighted the importance of iterative development and testing in achieving functional goals. The implementation of blocking behavior and currency in pipes remains significant challenge. We are working on concurrent pipes, but this has proved much more difficult than linear pipes. Our decision to create a program capable of running various pipe implementations offered a pragmatic and flexible approach to testing and comparison. It emphasized the importance of designing adaptable and modular software, especially in a research context where requirements can rapidly evolve. Moving forward, the controller should be augmented with benchmarking capabilities to compare pipes statistically.