CMPSCI 311 Lecture 18.1/18.2

Pipes and Threads

Scribe Notes for 2/24/16 by Steven Tu, Tiwari Vedant, Wei Wang

Table of Contents

Overview

This lecture begins by reviewing some key interprocess communication (IPC) concepts that are key to understanding pipes and FIFOs. We will then explore pipes, their usage, and how to implement them. We will briefly touch on FIFOs and shared memory mechanisms, but it will mostly be left for you to research on your own if interested. Next, we will introduce threads, motivation for using them.

Recall

Interprocess communication (IPC) is a set of mechanisms provided by the system that allows its concurrent processes to share information and data. The following table outlines some of the methods the operating system can provide for interprocess communication that we went over in class.

Despite these many methods, IPC (and really any type of communication that exists) fundamentally boils down to two primary modes of communication: Message Passing (MP) and Shared Memory (SM). Recall that the primary difference lies in how information is shared. Message passing relies on sending messages between processes while shared memory allows for a portion of memory to be accessed from by multiple processes (for reference we use `shmget` in POSIX`)`. We will now begin exploring pipes, an essential message passing tool for processes to use to solve problems.

Pipes

Pipes are an important mechanism in IPC that chains processes in a way such that the standard output stream of one process can be fed into the standard input stream of the next process. We learned earlier that pipes were an example of shared memory, however, this is not true. When the pipe function is called, a communication buffer is created that can only be accessed through the file descriptors `filedes[0]` and `filedes[1]`. Data written to `filedes[1]` can only be read from `filedes[0]` on a FIFO basis. As a result, one cannot seek and access data at some arbitrary place in the pipe, only at the front and end by the file descriptors. In addition, because a pipe has no external/permanent name, and can only be accessed through it file descriptors, pipes can only be used by the process that created it, as well as its descendants (inherited on `fork`). Because this memory buffer lives in the operating system's address space, and not the process' address space, it requires system calls to read and write to. This sending and receiving of information is what distinguishes pipes as MP and not SM.

Implementation:

```
#include <unistd.h>
int pipe(int fildes[2]);
```

This line of code and its library uses the system call `pipe()` to create the communication buffer that can be accessed through the file descriptors. If successful, `pipe` returns 0, else `pipe` returns -1 and sets the `errno`. It is useful to use `perror` to indicate a pipe failure. Let's analyze a more complete implementation of pipes. The following program mimics
ls -l | sort -n +4.

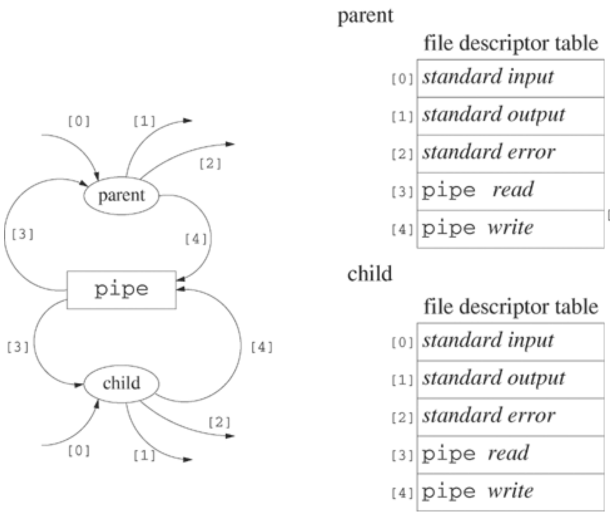
Program 6.3 - pg. 236

```
1     #include <errno.h>
2     #include <stdio.h>
3     #include <unistd.h>
4     #include <sys/types.h>
5     int main(void) {
6           pid_t childpid;
7           int fd[2];
8           if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {
9                 perror("Failed to setup pipeline");
10                return 1;
11          }
12          if (childpid == 0) { /* ls is the child */
13                if (dup2(fd[1], STDOUT_FILENO) == -1)
14                      perror("Failed to redirect stdout of ls");
15                else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
16                      perror("Failed to close extra pipe descriptors on ls");
17                else {
18                      execl("/bin/ls", "ls", "-l", NULL);
19                      perror("Failed to exec ls");
20                }
21                return 1;
22          }
23          if (dup2(fd[0], STDIN_FILENO) == -1) /* sort is the parent */
24                perror("Failed to redirect stdin of sort");
25          else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
26                perror("Failed to close extra pipe file descriptors on sort");
27          else {
28                execl("/bin/sort", "sort", "-n", "+4", NULL);
29                perror("Failed to exec sort");
30          }
31          return 1;
32    }
```

In lines 8-10, the program sets up the pipe and creates a child process, immediately returning an error and exiting the program if either creation fails. Upon success, child processes inherit a copy of the file descriptor table of the parent process. With the pipe set up, fd[0] is now set for reading, and fd[1] is set for writing.

Line 12 verifies that we are currently in the child process, and line 13 uses the system call `dup2` to copy/redirect the write end of the pipe to the standard output of `ls`. Once `execl` completes the `ls` command, we can now redirect (using `dup2` again) the read end of the pipe to the standard input file descriptor of our parent process: `sort` (line 23). .

Pre-dup2
(after fork)



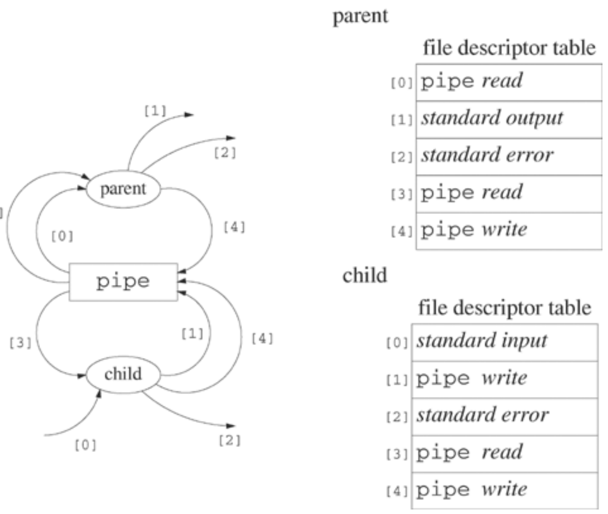
parent file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

child file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

Post-dup2
(both dup2)



parent file descriptor table

[0]	pipe *read*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

child file descriptor table

[0]	*standard input*
[1]	pipe *write*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

The previous graphics shows how file descriptors of `ls` (the child process) goes from being a copy of the parent's file descriptors, to having its output changed to the write end of the pipe and `sort`'s standard input is duplicated from the read end of the pipe.

FIFOs

Although we are not going over FIFOs in detail (we will be learning it in a later course and can read about it in our textbooks), we can quickly discuss the idea of them. In general, FIFOs, or named pipes, are special files that unlike normal pipes, persist even after all processes have closed them. This allows for unrelated processes to use the same pipe. FIFOs can be created using the command or system call `mkfifo`.

Implementation:

In order to use the mkfifo function mentioned earlier, it is important to include the stat header file

```
#include <sys/stat.h>
```

The function used to create FIFO is mkfifo with two parameters

```
    int mkfifo(const char *path, mode_t mode);
```