CSE 410: Systems Programming

Pipes and Redirection

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Interprocess Communication

UNIX pipes are a form of interprocess communication (IPC).

IPC provides a mechanism for processes to cooperate.

There are many forms of IPC on POSIX systems:

- Pipes
- Sockets
- Shared memory
- Signals
- Process return values
- Environment variables
- …

UNIX Pipes

The UNIX pipe was introduced as early as 1972 [2].

Pipes provide a file-like abstraction for IPC:

- Data written into one end of a pipe can be read at the other.
- Reading and writing on pipes uses UNIX I/O functions.
- Pipes are represented as file descriptors.

Pipes and Standard I/O

A pipe can be connected to any file descriptor.

However, pipes on the standard I/O file descriptors (0-2), are particularly useful.

Many UNIX utilities:

- read from standard input (file descriptor 0)
- write to standard output (file descriptor 1)

By placing a pipe on these file descriptors, the output of one process becomes the input of another.

Creating a Pipe

```
int pipefd[2];

if (pipe(pipefd) < 0) {
    perror("pipe");
}
```

The `pipe()` system call creates a pipe as a pair of file descriptors.

The first is read-only, and the second is write-only.[1]
(This is the same order as standard input and standard output.)

---

[1] Some systems may have bidirectional pipes.

Simple Usage

```
int pipefd[2], rval, wval = 42;

pipe(pipefd);

write(pipefd[1], &wval, sizeof(wval));
read(pipefd[0], &rval, sizeof(rval));

printf("%d\n", rval);
```

Output:

```
42
```

Simple Usage

```c
int pipefd[2], rval, wval = 42;

pipe(pipefd);

write(pipefd[1], &wval, sizeof(wval));
read(pipefd[0], &rval, sizeof(rval));

printf("%d\n", rval);
```

Create a pipe on the fd array.

Output:

```
42
```

Simple Usage

```
int pipefd[2], rval, wval = 42;

pipe(pipefd);

write(pipefd[1], &wval, sizeof(wval));
read(pipefd[0], &rval, sizeof(rval));

printf("%d\n", rval);
```

Write an integer into the pipe.

Output:

```
42
```

Simple Usage

```
int pipefd[2], rval, wval = 42;

pipe(pipefd);

write(pipefd[1], &wval, sizeof(wval));
read(pipefd[0], &rval, sizeof(rval));

printf("%d\n", rval);
```

Read the integer from the pipe.

Output:

```
42
```

Simple Usage

```c
int pipefd[2], rval, wval = 42;

pipe(pipefd);

write(pipefd[1], &wval, sizeof(wval));
read(pipefd[0], &rval, sizeof(rval));

printf("%d\n", rval);
```

`rval = wval`

Output:

```
42
```

Mechanism

Each pipe is a kernel buffer accessed through file descriptors.

pipe fd 0

Kernel buffer

pipe fd 1

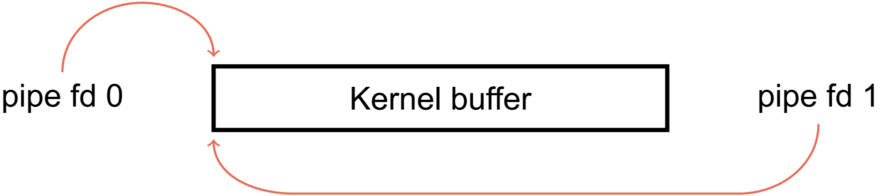
Mechanism

Each pipe is a kernel buffer accessed through file descriptors.

The read file descriptor has a read pointer.

Mechanism

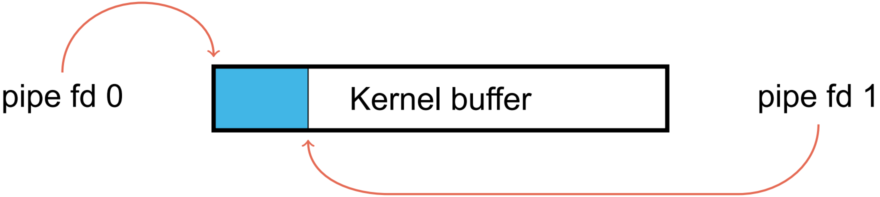
Each pipe is a kernel buffer accessed through file descriptors.



The write file descriptor has a write pointer.

Mechanism

Each pipe is a kernel buffer accessed through file descriptors.

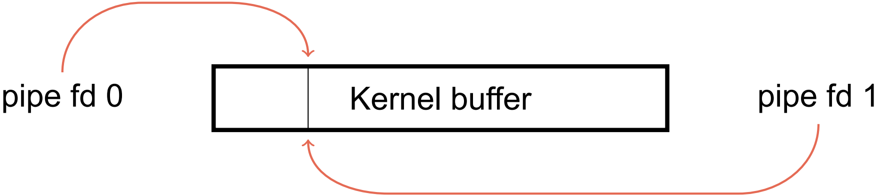


Writing to the write file descriptor fills the buffer:

```
write(pipefd[1], &wval, sizeof(wval));
```

Mechanism

Each pipe is a kernel buffer accessed through file descriptors.



Reading from the read file dscriptor drains the buffer:

```
read(pipefd[0], &rval, sizeof(rval));
```

Buffer Capacity

If the buffer becomes full, writes block.

If the buffer is empty, reads block.

This makes pipe communication within a single process susceptible to deadlock:

- The process writes $>$ buffer size bytes and blocks

Buffer Capacity

If the buffer becomes full, writes block.

If the buffer is empty, reads block.

This makes pipe communication within a single process susceptible to deadlock:

- The process writes $>$ buffer size bytes and blocks
- No read is available to drain the buffer

Buffer Capacity

If the buffer becomes full, writes block.

If the buffer is empty, reads block.

This makes pipe communication within a single process susceptible to deadlock:

- The process writes $>$ buffer size bytes and blocks
- No read is available to drain the buffer
- …

Fork and pipe

Reading between processes does not have this problem.

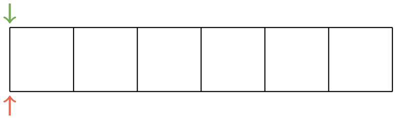
```
pipe(pipefd);
if ((pid = fork()) == 0) {
    write(pipefd[1], "Hello", 6);
} else {
    read(pipefd[0], &buf, 6);
    puts(buf);
}

Hello
```

Fork and Pipe in Action

```
pipe(pipefd);
if ((pid = fork()) == 0) {
    write(pipefd[1], "Hello", 6);
} else {
    read(pipefd[0], &buf, 6);
    puts(buf);
}
```

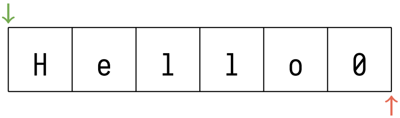
Kernel buffer:



Child process begins write, parent process blocks on read.

Fork and Pipe in Action

```
pipe(pipefd);
if ((pid = fork()) == 0) {
    write(pipefd[1], "Hello", 6);
} else {
    read(pipefd[0], &buf, 6);
    puts(buf);
}
```



Kernel buffer:

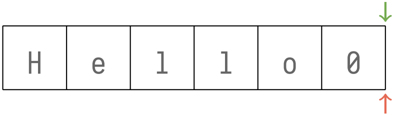
H	e	l	l	o	0

Child process writes `"Hello"`.

Fork and Pipe in Action

```
pipe(pipefd);
if ((pid = fork()) == 0) {
    write(pipefd[1], "Hello", 6);
} else {
    read(pipefd[0], &buf, 6);
    puts(buf);
}
```

Kernel buffer:



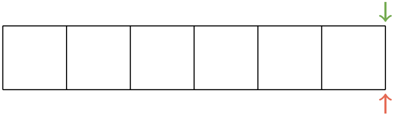
H	e	l	l	o	0

Parent process read continues.

Fork and Pipe in Action

```
pipe(pipefd);
if ((pid = fork()) == 0) {
    write(pipefd[1], "Hello", 6);
} else {
    read(pipefd[0], &buf, 6);
    puts(buf);
}
```

Kernel buffer:



Parent process puts buffer.

More on File Descriptors

Recall that a file descriptor is a small integer.

It is an index into a kernel table of open files for the process.

Every process has its own file descriptor table.

The entries in this table point to a global table of open files.

Open File Table

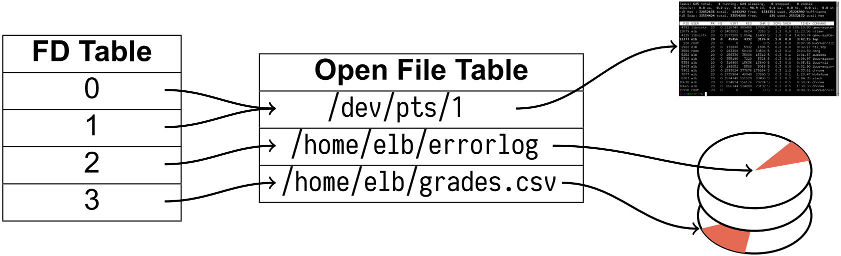
The global open file table maintains metadata for open files:

- The current position in the file for read/write
- The filesystem and disk location (or device status) of the file
- The permissions and mode of the open file (read, write, *etc.*)

The metadata in this table allows access to open files with minimal overhead.
(*E.g.*, no permission checks, no disk indexing, …)

File Descriptor Indirection

File descriptors therefore provide indirect access to open files.



FD Table
0
1
2
3

Open File Table
`/dev/pts/1`
`/home/elb/errorlog`
`/home/elb/grades.csv`

Fork and File Descriptors

As discussed, `fork()` duplicates the file descriptor table.

Since the descriptors point into global open files, the file metadata is the same in the new table.

This means that some things affect both descriptors:

- Changes to the file position
- Changes to other open file properties

Pipe File Descriptor Gotcha

The read end of a pipe returns EOF when the write end is closed.

Pipe file descriptors are cloned on fork.

A file table entry stays open if any file descriptor is open.

A pipe read end will never return EOF if any write end file descriptor remains open.

Safe Pipe and Fork

```
int pipefd[2], pid;
char buf[6];

pipe(pipefd);
if ((pid = fork()) == 0) {
    close(pipefd[0]);
    write(pipefd[1], "Hello", 6);
} else {
    close(pipefd[1]);
    read(pipefd[0], &buf, 6);
}
```

The child process closes the pipe output, and the parent process closes the pipe input.

Safe Pipe and Fork

```
int pipefd[2], pid;
char buf[6];

pipe(pipefd);
if ((pid = fork()) == 0) {
    close(pipefd[0]);
    write(pipefd[1], "Hello", 6);
} else {
    close(pipefd[1]);
    read(pipefd[0], &buf, 6);
}
```

The parent read will then reliably signal EOF.

Copying File Descriptors

File descriptors can be explicitly copied.

This creates two file descriptors pointing to the same open file table entry.

This is how standard input and output are redirected.

By placing a chosen file on a chosen file descriptor.
(*E.g.*, 0, 1, or 2.)

Copying a Descriptor

```
#include <unistd.h>

int dup(int fd);
int dup2(int oldfd, int newfd);
```

The `dup()` system call copies a file descriptor.

It accepts an open descriptor and returns a copy on a new descriptor.

The `dup2()` system call does the same — except it allows the destination fd to be specified.

Redirecting Standard Output

```
int fd;

fd = open("output.txt", O_WRONLY|O_CREAT, 0666);

dup2(fd, 1);
close(fd);

puts("Redirected output!");
```

Summary

- Pipes form a UNIX IPC mechanism.
- They are a kernel communication channel that provides file semantics.
- Pipes have finite buffer space.
- File descriptors are indirect pointers to open files.
- Fork copies file descriptors and thus open file state.
- File descriptors can be explicitly copied with `dup()` and `dup2()`.

Next Time …

- …

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 10: 10.8, 10.9. Pearson, 2016.

Optional Readings

[2] Dennis M. Ritchie. "The Evolution of the Unix Time-Sharing System". In: *Proceedings of the Symposium on Language Design and Programming Methodology*. `https://link.springer.com/content/pdf/10.1007%2F3-540-09745-7_2.pdf`. Sept. 1979, pp. 25–35.

License