CIS 5480
Recitation 02

Friday, September 11, 2026

Penn Engineering
University of Pennsylvania

Agenda

- File Descriptor Refresher
- Redirection Refresher
- Worksheet Problem #1
- Pipelines & Pipes
- Worksheet Problem #2
- FDs with Forking
- Worksheet Problems #3,4

File Descriptors

A **file descriptor** is a non-negative integer that refers to an open file or I/O resource.

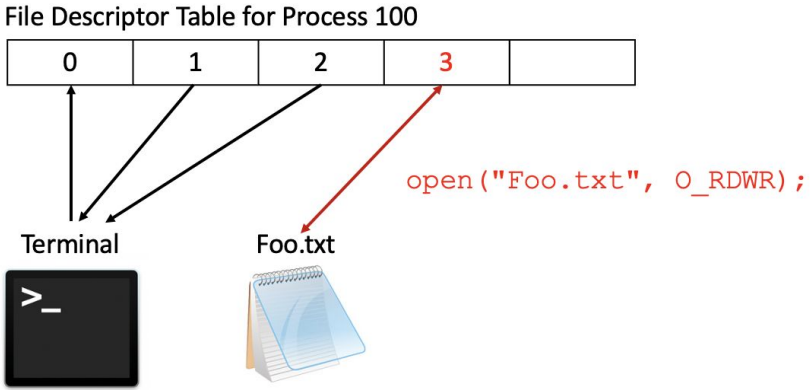
Examples of resources:

- A literal file
- Terminal input/output
- Pipes

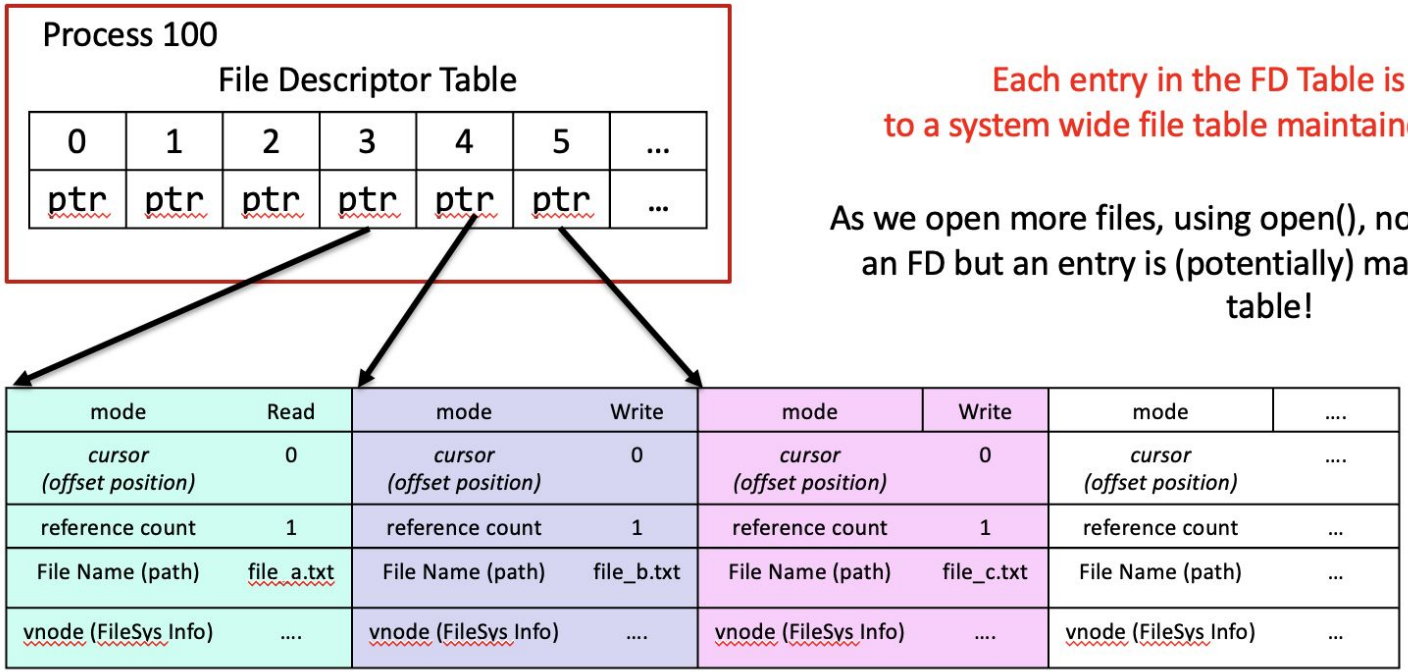
Per-Process File Descriptor Table

Each process has its own **FD table** managed by the OS. A file descriptor indexes into this table.

The process fd table keeps track of what resources are being used by a process.



The Open File Table (OFT) (System-Wide)



Each entry in the FD Table is a pointer to a system wide file table maintained by the kernel!

As we open more files, using open(), not only do we receive an FD but an entry is (potentially) made in the open file table!

Redirection

- We can manipulate the File Table so that a FD Table entry is associated with another file.
- `int dup2(int oldfd, int newfd);`
  - The file descriptor *newfd* is adjusted so that it now refers to the same open file description as *oldfd*. (newfd is closed silently…shh)

```
int dup2(int redirect_here, STDOUT_FILENO);
```

  - In this example, STDOUT_FILENO, no longer refers to the terminal, but rather the FILE associated with *redirect_here*

Worksheet #1A

1 Silencing Program Output

One practical use of file-descriptor redirection on Unix systems is to silence a program's outputs. For example, a debugging build might log messages to the terminal, while a release build would opt to suppress them.

Unix provides a special device file, /dev/null, that discards any data written to it. By redirecting both standard output (stdout) and standard error (stderr) to /dev/null, we can prevent a program from displaying any output through either streams.

Please complete the function below using dup and/or dup2 such that after silence_program() returns, anything written to stdout or stderr is discarded. You may assume that the call to open() succeeds.

```
// Recall the standard file descriptors are:
// STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
void silence_program() {
   int devnull_fd = open("/dev/null", O_WRONLY);



   close(devnull_fd);
}
```

Worksheet #1A Solution

```
// Recall the standard file descriptors are:
// STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
void silence_program() {
        int devnull_fd = open("/dev/null", O_WRONLY);

        dup2(devnull_fd, STDOUT_FILENO);
        dup2(devnull_fd, STDERR_FILENO);

        close(devnull_fd);
}
```

Worksheet #1B

```
// Recall the standard file descriptors are:
// STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
void silence_program() {
        int devnull_fd = open("/dev/null", O_WRONLY);

        dup2(devnull_fd, STDOUT_FILENO);
        dup2(devnull_fd, STDERR_FILENO);

        close(devnull_fd);
}
```

Why is it safe to close `devnull_fd` at the end of the function?

Worksheet #1B Solution

Each file descriptor is an index into the process' file descriptor table, whose entries point to entries in the Open File Table (OFT).
`dup2(devnull_fd, STDOUT_FILENO)` and
`dup2(devnull_fd, STDOUT_FILENO)`
copy the OFT pointer from `/dev/null` into the `stdout` and `stderr` indices, so all three descriptors refer to the same open file.

Closing `devnull_fd` only clears one FD-table entry on the process' fd table. `stdout`, and `stderr` still point to `/dev/null` in the OFT .

Pipelines

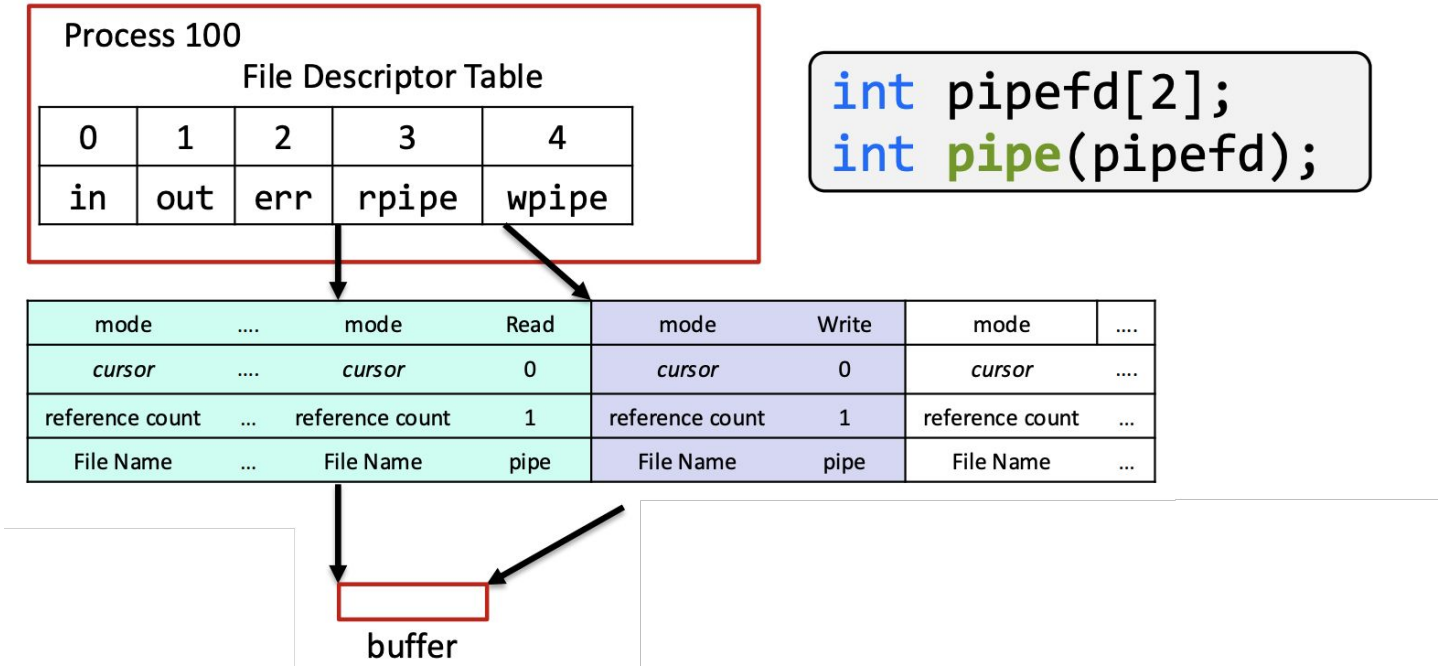
What is a pipe?

- Kernel buffer with two file descriptors - one for each end
- Push from/pull from buffer from write and read end respectively
- Used for inter-process communication (IPC)

Pipelined functions: i.e. cat log | grep brains | wc -l

- What does this do? Output log → grep for lines that start with “brains” → count number of such matching lines

Pipes Visualization



Worksheet #2A

2 Redirecting to Concatenation Station

`cat` is a command that reads data from a list of provided files and writes their concatenation to standard output. If no files are provided it instead reads from standard input. For the provided commands, describe the following:

- The output produced by running the command
- Where cat is reading from
- Where cat is writing to
- The system calls needed to implement each command in penn-shell

Assume the file **`file1.txt`** exists and contains the text "`Not all who wander are lost... but I sure am!`" and `file2.txt` exists and contains the text "`Off I go into the wild pale yonder`". Be explicit about which FD is being used (where you can) and what specific file it refers to.

Questions:

- **`cat file1.txt file2.txt`**

Worksheet #2A Solution

`cat file1.txt file2.txt`

User-Visible Output: "Not all who wander are lost... but I sure am! Off I go into the wild pale yonder"

Cat Reading From: cat's stdin (FD=0) is still the terminal, but it doesn't use it b/c filenames are given

For each filename:

cat calls open("file1.txt", O_RDONLY) → get some FD back

cat calls open("file2.txt", O_RDONLY) →

FD 3 → file1.txt

FD 4 → file2.txt

Cat Writing To: cat writes to stdout (FD 1)

System Calls:

- fork()
- execvp("cat", ["cat", "file1.txt", "file2.txt", NULL])

parent →
- waitpid(child-pid, ...)

Penn

Worksheet #2B

`cat` is a command that reads data from a list of provided files and writes their concatenation to standard output. If no files are provided it instead reads from standard input. For the provided commands, describe the following:

- The output produced by running the command
- Where `cat` is reading from
- Where `cat` is writing to
- The system calls needed to implement each command in `penn-shell`

- `echo "Hello?" | cat | wc -l`

`echo "Hello?" | cat | wc -l`

User-visible o/p: echo "Hello?" produces Hello\n

cat copies its input to its output

wc -l counts lines

Final o/p: 1

Where cat is reading from: cat has no filename arguments, so it reads from stdin

↳ stdin has been changed by shell

FD 0 → p1

Where cat is writing to: In cat, FD 1 has been changed to write end of pipe2

p2 ← FD 1

System calls for penn-shell: Creates 2 pipes

```
int p1[2]
int p2[2]
pipe(p1)
pipe(p2)
```

fork for echo hello:

Child 1: echo - connect stdout to pipe 1 write end
close unused ends
execvp("echo"...

fork for cat:

child 2: cat - connect stdin to p1 read
connect stdout to p2 write
closed unused ends
execvp("cat"...)

fork for wc -l:

child 3: wc - connect stdin to p2 read
close all pipe fds
execvp("wc"...)

parent (shell): close all pipe fds
waitpid(c1....)
waitpid(c2...)
waitpid(c3...)

Penn

Worksheet #2C

`cat` is a command that reads data from a list of provided files and writes their concatenation to standard output. If no files are provided it instead reads from standard input. For the provided commands, describe the following:

- The output produced by running the command
- Where `cat` is reading from
- Where `cat` is writing to
- The system calls needed to implement each command in `penn-shell`

- `cat < file1.txt > file3.txt`

Worksheet #2C Solution

`cat < file1.txt > file3.txt`

User-Visible O/p: NO OUTPUT TO TERMINAL

new file file3.txt created, its content becomes exactly that of file1.txt

Where cat is Reading From: invoked w/ no filename args, so reads from stdin

↳ redirected to file1.txt

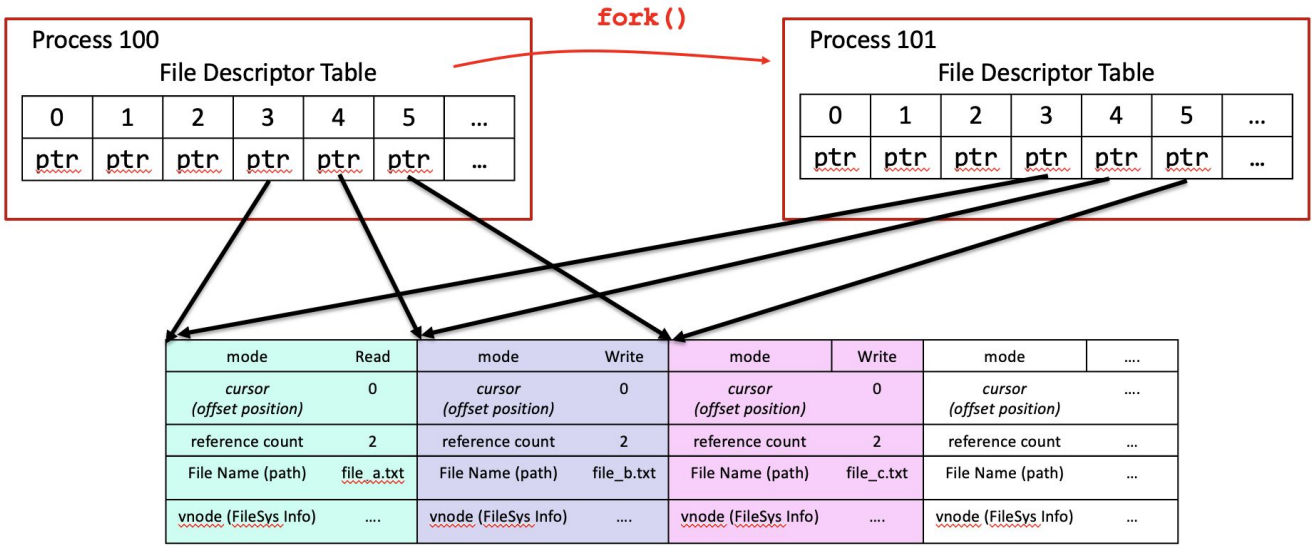
Where cat is reading from: stdout redirected to file3.txt
fd=1

Sys calls for Penn-Shell:

- open redirection files
- fork()
  - ↳ in child, redirect stdin to point to file1.txt
    redirect stdout to point to file2.txt
    execvp("cat" ...)
  - in parent, close new fds & waitpid

FDs with Forking

- When a process forks, the child inherits an identical FD Table!



Forking **clones** the FD table. Any modifications made within a process does not modify the per-process FD table of another process (they are independent).

Worksheet #3A

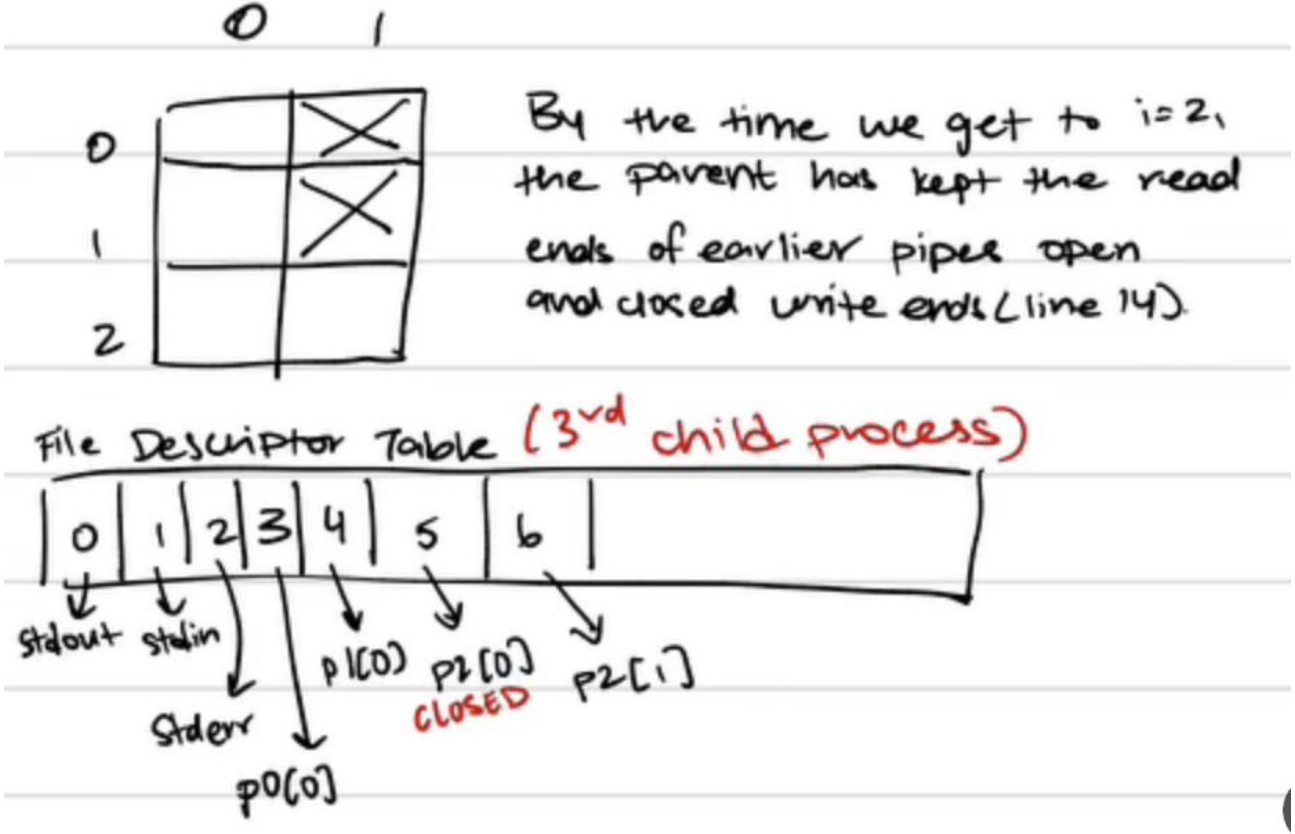
The following code forks three child processes, each of which has a pipe connecting it to the parent. Each child will read the contents of `"very_large_file.txt"` and write this content back to the parent using its pipe. The parent waits for a given child to finish before attempting to read from the associated pipe. Assume `read_and_write_file` is implemented correctly and that `"very_large_file.txt"` is larger than a page size.

```
int main() {
    char *file = "very_large_file.txt";
    int pipe_fds[3][2]; // Array of int arrays

    for (int i = 0; i < 3; i++) {
        pipe(pipe_fds[i]);
        pid_t pid = fork();
        if (pid == 0) {
            close(pipe_fds[i][0]); // Close read end of newly created pipe
            read_and_write_file(pipe_fds[i][1], file);
            return EXIT_SUCCESS;
        }
        close(pipe_fds[i][1]); // Close write end of newly created pipe

        wait(NULL);

        char str[BUF_SIZE];
        ssize_t chars_read = read(pipe_fds[i][0], str, BUF_SIZE);
        while (chars_read > 0) {
            write(STDOUT_FILENO, str, BUF_SIZE);
            chars_read = read(pipe_fds[i][0], str, BUF_SIZE);
        }
    }
    return EXIT_SUCCESS;
}
```

Draw out the state of the file descriptor table for the third child process and the open file table.

Penn Engineering

Worksheet #3A Solution



Worksheet #3B

There is a bug in this code. Describe what the issue is and how it could be fixed.

Worksheet #3B Solution

2.2. The parent waits for the child to finish before reading, the file is larger than a page size.

Bug: Deadlock

- Child writes very_large_file.txt into the pipe
- Pipes have a limited buffer
- Child eventually fills pipe buffer and blocks on write
- parent does wait on child before it starts reading

Nobody can proceed: Deadlock

Fix #1:

Currently, we wait before reading. If we start reading then wait, the child's writes won't block indefinitely.

Worksheet #4A

A friend has modified the `pipe_poll.c` lecture code such that both the parent and child process can read and write messages to each other. They state that by using `pipe2` they do not need to worry any more about calling `close` on any of the pipe file descriptors. Part of their implementation is provided below.

```
int p_to_c[2];
int c_to_p[2];

pipe2(p_to_c, O_CLOEXEC);
pipe2(c_to_p, O_CLOEXEC);

pid_t pid = fork();
if (pid == 0) {
    // Read from parent one byte at a time
    char str[strlen(p_message) + 1];
    int i = 0;
    ssize_t chars_read = read(p_to_c[0], str, sizeof(char));
    while (chars_read > 0) {
        i++;
        chars_read = read(p_to_c[0], str + i, sizeof(char));
    }
    printf("%s", str);
    // Write to parent
    write(c_to_p[1], c_message, strlen(c_message) + 1);
    return EXIT_SUCCESS;
} else {
    // Assume parent's code does the reverse of the child, i.e., writes to p_to_c[1] and reads from
    ↪ c_to_p[0]
}
```

Question: What is incorrect about your friend's code and how would you fix it?

Worksheet #4A Solution

3. O_CLOEXEC does not close file descriptors on close. It closes them on exec().

fork()	child inherits all open FDs
exec()	FDs with O_CLOEXEC are closed
normal	FDs remain open

This means unused ends are never closed. So EOF is never delivered.

Fix: Explicitly close unused pipe ends in each process!

For every child c and parent p, close:

In child:

```
close(p_to_c[1])
.....
close(c_to_p[0])
```

In parent:

```
close(p_to_c[0])
close(c_to_p[1])
```

* remember: pipes establish one-way communication

Worksheet #4B

Question: Does it make sense to use pipe2 for this program? Why or why not?

Worksheet #4B Solution

No. pipe2(O_CLOEXEC) is useful if you fork then immediately exec. Since we don't exec, there is no benefit. This doesn't replace close.