# CIS 5480 Recitation 2 - File Descriptors, Pipes, and Shell

*Welcome to recitation!!!* 🦀

**Name: _____** **PennID: _____**

## 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);
}
```

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

# 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`**

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

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

# 3 Piling on the Pipeline

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;
}
```

## 3.1 FD and Open File Table

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

## 3.2 Wait what was that?

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

# 4 To pipe2 or not to pipe2?

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]
}
```

## 4.1 What is incorrect about your friend’s code and how would you fix it?

## 4.2 Does it make sense to use pipe2 for this program? Why or why not?