# CSCI 4100
# Assignment 4
# The POSIX File Interface

## Learning Outcomes

Write a Linux utility program using the POSIX file interface.

## Required Reading

Saltzer & Kashoek 2.2-2.3

## Instructions

For this programming assignment you are going to implement a simple C version of the UNIX `cp` program called `monkey`. The `cp` program is used to copy files and has many command line options. The `monkey` program will just copy a file without all of the fancy options. The correct usage of your program should be to execute it on the command line with two command line arguments: the file to be copied (the source file) and the new file to be created (the destination file.) However, your program should also respond well when it is used incorrectly.

Here is what your program will need to do:

1. Create a buffer to store the data to be copied. This should be an array of 4096 characters (the size of a file block.)
2. Check to see if the program was given the correct number of arguments. If not, display a usage message and exit the program.
3. Open the source file as read-only. If there is an error opening the source file, display an error message and exit the program.
4. Create the destination file. If there is an error creating the file, display an error message and exit the program.
5. Do the following until the entire file to be copied has been read:
   (a) Read the next 4096 bytes from the source file into the buffer.
   (b) Write however many bytes were actually read into the buffer to the destination file.
6. Close both of the files.

## The POSIX File Interface

To use the POSIX file interface you must add two more include statements to the ones used in Assignment 2:

```
#include <unistd.h>
#include <fcntl.h>
```

To create a new file you must use the `creat` function:

```
fd = creat(filename, mode);
```

- The first argument is a C-style string representing the name of the file to be created.
- The second argument is an octal representation of the mode or permissions for the file (for example: `0644` means that the file will be readable by everyone but only writable by the user.)
- The return value is a file descriptor number (an `int`), which must be used when reading from, writing to, or closing the file. If there was an error creating the file, the return value will be -1.

To open an existing file you must use the `open` function:

```
fd = open(filename, how);
```

- The first argument is a C-style string representing the name of the file to be opened.
- The second argument is a flag indicating how the filed should be opened. The options for this argument are `O_RDONLY` for read-only, `O_WRONLY` for write-only, and `O_RDWR` for read and write.
- The return value is the same as the return value for `creat`.

To read from a file you must use the `read` function:

```
chars_read = read(fd, buffer, num_bytes);
```

- The first argument is the file descriptor returned by `creat` or `open`.
- The second argument is a character array where the contents of the file will be stored.
- The third argument is the number of bytes to attempt to read from the file.
- The return value is the number of bytes actually read, or -1 if there was an error reading the file. If the end of the file has been reached, the return value will be 0.

To write to a file you must use the `write` function:

```
chars_written = write(fd, buffer, num_bytes);
```

- The first argument is the file descriptor returned by `creat` or `open`.
- The second argument is a character array containing the data to be written.
- The third argument is the number of bytes to attempt to write to the file.
- The return value is the number of bytes actually written, or -1 if there was an error writing the file.

To close a file you must use the `close` function:

```
result = close(fd);
```

- The argument is the file descriptor returned by `creat` or `open`.
- The return value is 0 if closing the file was successful and -1 if there was an error.

## Displaying Error Messages

In Assignment 2 we used `fputs` to display error messages. This is still the best way to generate a usage message:

```
if(argc != 3) {
  fputs("Usage: monkey source destination\n", stderr);
  exit(1);
}
```

When it comes to the other error messages, C provides a better way to generate errors through the `perror` function:

```
in_fd = open(argv[1], O_RDONLY);
if(in_fd == -1) {
  perror(argv[1]);
  exit(1);
}
```

This will display the name of the file provided along with a description of the most recent error encountered:

```
$ ./monkey nosuchfile.txt nosuchfile_copy.txt
nosuchfile.txt: No such file or directory
```

## Compiling and Running Your Code

Your source file should be called `yourlastnameAssign4.c` except with your actual last name. To compile this code you should use the `gcc` compiler on the Linux server:

```
gcc -o monkey yourlastnameAssign4.c
```

If your program compiled, you should see an entry for `monkey` when you execute the `ls` command.

When you run your code, try it using both text files and executable files. You can use `diff` to check if the copied text files are correct, and `cmp` to test if the copied executable files are correct. Both of these commands will display no output if the files are the same. Here are some examples of what several runs should look like:

```
$ ./monkey myfile.txt myfile_copy.txt
$ diff myfile.txt myfile_copy.txt
$ ./monkey monkey monkey_copy
$ cmp monkey monkey_copy
$ ./monkey myfile.txt
Usage: monkey source destination
$ ./monkey nosuchfile.txt nosuchfile_copy.txt
nosuchfile.txt: No such file or directory
$ ./monkey myfile.txt .
.: Is a directory
```

## What to Hand In

Your source file should have comments at the top listing your name, CSCI 4100, Assignment 4, and a brief explanation of what the program does. Download the source file to your local machine, then upload it to D2L in the dropbox called Assignment 4.