CS431 Operating Systems
Programming Assignment 1 - POSIX System Calls and Error Handling

1 Overview

This programming project serves as an introduction or refresher to C programming and the use of basic system calls in Unix that we begin to discuss in Chapter 2 and will continue to use for the duration of the term. Conceptually, the requirements of this assignment are quite simple. All you are asked to do is write a C program that copies the contents of a file of any size or type from a source location to a destination, both specified as command line arguments. That objective is straightforwards enough, but a recurring theme in this course and in systems design in general is the handling of errors and unexpected conditions as they arise. Therefore, the intent is for you to write code that not only functions properly in the ideal case, but also can gracefully and correctly handle error conditions should they occur.

2 Requirements

Write a program called *mycopy* that opens a file whose name is given to it as an argument on the command line and copies its contents to another (newly created) file, also specified as a command line argument. The program will be invoked as follows:

```
$ ./mycopy SourceFile DestinationFile
```

Note that you should not assume the file to be copied is a small text file. The program will copy bytes, and it makes no difference what those bytes represent, whether ASCII text or something else. Therefore, your program should be able to correctly copy multimedia files, operating system images, executable programs, and other large binary files in addition to text files. Your program must create an identical copy of `SourceFile` under the new name `DestinationFile`, so do not hard code any file names or file name extensions in your program. When the program successfully completes the copy, it should output the total number of bytes copied, then exit with a status code of zero. For example:

```
copied 17856 bytes from file InFile.dat to OutFile.dat.
```

If your program encounters an error condition (to include user error such as specifying a source file that does not exist), it must produce an error message stating the operation that failed, whether open, read, write, or close, and the reason why the operation failed. Then the program must exit with a non-zero status code. For example:

```
$ ./mycopy file1.jpg file2.jpg
could not open file file1.jpg:  No such file or directory
```

or...

```
$ ./mycopy file1.jpg file2.jpg
could not write to file file2.jpg:  Permission Denied
```

If the user incorrectly invokes the program, then the program displays a helpful error message and terminates. For example:

```
$ ./mycopy file1.exe file2.exe file3.exe
mycopy:  invalid number of arguments
usage:  mycopy <sourcefile> <destinationfile>
```

3 Unix System Calls

This assignment will require you to become familiar with the following system calls:

`open`, `creat`, `read`, `write`, `close`

and the following C library functions:

`strerror`, `errno`, `exit`, `printf`

On any Unix system, manual pages (so-called "man pages" for short) provide the complete reference documentation for system calls. One thing to be aware of is that man pages are divided into sections, with Section 2 reserved for system calls and Section 3 for C library functions. Therefore, on any Unix machine simply type `man` with the section number and the system call or library function name to bring up the appropriate man page. For example, typing `man 2 open` gives you the man page for the `open()` system call. Specifying Section 2 avoids bringing up the wrong page in those cases where a system call and a system command have the same name, of which `write()` is one. You can also use any number of resources on the internet, including http://linux.die.net/man, to obtain the contents of the man pages and other documentation should you find that more convenient.

One caveat: man pages can be a little tricky to understand at first because they are not only fairly comprehensive and technical, but do not show how various system calls interrelate. As an example, the `man` page for `open()` contains quite a bit of information about it (more than 800 lines of text). However, certain parts will be more useful to you than others. Under the heading SYNOPSIS, for instance, it tells you what header files must be included in order to use `open()`:

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

This means any program that calls `open()` must have those three `#include` directives verbatim. In that same section, the man page also gives you the the function prototype, i.e., function declaration, for several variations of the system call:

```
int open(const char pathname, int flags);
int open(const char pathname, int flags, mode_t mode);
int creat(const char pathname, mode_t mode);
```

As mentioned earlier, the man page documentation is fairly comprehensive, so it contains quite a bit more information than you will need for this assignment. As a rule of thumb, you should generally read the first paragraph and then scroll through the rest skimming the content until you find what you need.

Important Note: Do NOT use library routines such as `fopen()`, `fgets()`, `fgetc()`, `fclose()`, etc. in this program for file I/O. One of the objectives of the assignment is to give you practice using POSIX system calls, which these are not. Doing so defeats one of the primary purposes of this assignment and will result in a grade penalty of 25 percent of the overall assignment grade.

4 Error Handling

This section describes the most important aspect of this assignment. The basic outline of your program will contain the following steps:

- open the source file using `open()`
- create the destination file using `creat()`
- loop {
  `read()` a chunk of data from the source file
  `write()` that same chunk of data to the destination file
  }
- `close()` both source and destination files
- print a message indicating success using `printf()`
- exit

Conceptually it is simple, but program correctness will prove to be a bit more complicated in practice because the program can encounter errors at nearly all of the above steps. Any operating system must be able to handle errors and unexpected conditions and programs should as well. As you are aware, users frequently attempt illegal or nonsensical operations, either intentionally or unintentionally. If you have taken CS465, you know you should check return values and take appropriate action when failures occur. Robustness is essential to any operating system, so when writing operating system code, the need to do so is even greater.

For system calls, success or failure is indicated by the return value of the function. Under the RETURN VALUE heading, the man page will tell you exactly how it indicates successful execution or a failure. As a convention, nearly every Unix system call that returns an integer, e.g., `open()`, `read()`, `write()`, `close()`, etc., does the following:

- If the call is successful, it returns an integer value greater than or equal to zero.

- If the call failed, it returns an integer value less than zero (typically -1), and sets the value of a global variable called `errno` to indicate the reason for the error.

Most C library calls that return a pointer, e.g., `malloc()`, `fopen()`, etc., use a somewhat different convention:

- If the call is successful, it returns a pointer with a non-null, i.e., non-zero, value.
- If the call failed, it returns a null pointer and sets the global variable `errno` to indicate the cause of the error condition.

The `errno` variable is nothing more than an global integer variable that is set when certain error conditions occur, and its value indicates the particular condition. Each error condition value is defined by a name that is relatively easy to remember, like `EPERM` for permission denied. All available error types are specified in the header file `/usr/include/asm/errno.h` included with every C compiler. These error number constants are already defined for you, so please do not attempt to redefine them in your program! A few of the more commonly occurring ones include:

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
...
```

So how do you use these values? One (not recommended) way you can do it is by checking for a specific kind of error and then taking an appropriate action:

```
fd = open(filename, O_RDONLY, 0);   // open file for read only

if( fd < 0 )                        // open() indicated an error condition
{
  if(errno == EPERM)
  {
    fprintf(stderr, "Error: Permission Denied\n");
  }
  else
  {
    fprintf(stderr, "Error: Unknown Error\n");
    ...
  }
  exit(1);
}
```

One complication with this approach is the 133 distinct `errno` values, so checking for each one separately would quickly become a very tedious programming exercise. To simplify things, use the `strerror()` library function declared in the header file `<string.h>` to convert the `errno` integer value into a string, then use it to print a descriptive error message:

```
fd = open(filename, O_RDONLY, 0);   // open file for read only

if( fd < 0 )                        // open() indicated an error condition
{
  fprintf(stderr, "Unable to open %s:  %s\n", filename, strerror(errno));
  exit(1);
}
```

With the `read()` and `write()` system calls things can get a little tricky, and they require additional care. Consider an attempt to read `count` bytes of data as such:

```
result = read(fd, buffer, count);
```

In this instance. several outcomes could occur. If `read()` was able to access some amount of data, the return value will be the number of bytes of data actually read and saved in `buffer`. Here the return value assigned to the (int) variable `result` will be some value between 1 and `count`, depending on the number of bytes actually read. It could be smaller than `count` if, for instance, the number of bytes remaining in the file to be read was less than the requested amount. If a read is attempted with nothing left in the file, the return value of `read()` will be zero. Any error condition will produce a return value of -1, as mentioned previously. The `write()` system call behaves in a very similar manner.

Expanding on the previous outline, your overall program should now be structured something like this:

- open the source file or exit with an error message using `open()`
- create the destination file using `creat()` or exit with an error message
- loop {
  `read()` a chunk of data from the source file
  if there was an error reading data, exit the program with an error message
  if there are no more data left to read, exit the loop but continue the program
  `write()` the same chunk of data, i.e., write only the number of bytes read in the previous step and no more, to the destination file
  if not all the data was written, exit the program with an error message
  }
- `close()` both source and destination files
- print a message indicating success using `printf()`
- exit

5 Command Line Arguments and Exit Status

How does your program access command line arguments? In a C or C++ program, the `main()` function is typically defined with two arguments: the first is an integer called `argc`, which is the

number of command line arguments, and `argv`, which is the list of those arguments (with each item in the list represented as an array of type `char`. For example, consider the following program:

```
int main(int argc, char** argv)
{
   printf("You entered %d arguments:\n", argc);
   for(int i = 0; i < argc; i++)
     printf("%d:  %s\n"), i, argv[i]);
   return 0;
}
```

produces output like the following:

```
$ ./myprogram lions tigers bears
You entered 4 arguments:
myprogram
lions
tigers
bears
```

The `int` return type of `main()` is used to pass exit status codes to the operating system shell that can be used to determine if your program executed correctly or encountered a failure. By Unix convention, returning a non-negative value (most commonly zero) indicates success and a negative value (most commonly -1) indicates failure. These values can be passed and the program terminated with a `return` statement in `main()`, as shown above, or with the `exit()` library routine.

6 Testing

Be sure you test your program under a variety of conditions, e.g., both large and small files. Also, make sure your program handles error conditions such as permission denied, file doesn't exist, etc. Finally, make sure you check to see the copy worked correctly. One way to do that is to use the program `md5sum` (or any hashing program such as `sha512sum`, etc.) to take a checksum of both files. Only if the content of both files is identical will the checksums match:

```
$ md5sum SourceFile
b92891465b9617ae76dfff2f1096fc97 SourceFile
$ md5sum DestinationFile
b92891465b9617ae76dfff2f1096fc97 DestinationFile
```

7 Requirements

Correctness of your program requires the following conditions be satisfied:

- You may build and test your program on any POSIX-compliant platform, e.g., Linux, Mac OS X, Solaris, etc.

- Appropriate error checking must be done after attempting to open or create a file, after each read or write operation, and when closing a file, with a suitable error message displayed and program termination, if necessary.
- Your program must be reasonably structured with suitable variable names and helpful comments where appropriate. **Be sure to include a header block of comments with your name and assignment information**.
- Turn in **only** the plain-text source file, i.e., *mycopy.c*, for your program.

8 Deliverables

Submit the C code source file for your program through the WorldClass assignment dropbox no later than the due date specified in the weekly assignment.

9 Academic Integrity

I recognize that there are only so many ways to write code that meets the requirements of this assignment, and you may consult with other students in this class on the general approach to a solution. Having said that, sharing code, pseudocode, or other implementation details is not permitted, and I expect all work to be your own as evidenced by:

- Program structure, control flow, variable and function names (aside from system calls and C library functions, of course), etc. should be manifestly distinct from others' work.
- Your program should be commented in a manner sufficient for me to ascertain that you understand what the code you submit does.