Computer Architecture and Technology Area

Universidad Carlos III de Madrid



OPERATING SYSTEMS

Lab 1. System Calls

**BACHELOR'S DEGREE IN COMPUTER SCIENCE AND**

**ENGINEERING**

Year 2016/2017

AUTHORS:

Ryan Cohane (100370386)

Carine Torres (100370405)

**Index**

1. Description of Code ………………………………………………………………………………… 3
   1. mycat ………………………………………………………………………………… 3
   2. myls …………………………………...……………………………………………… 4
   3. mysize ……………………………………………..………………………………… 5
2. Description of Tests …...…………………………………………………………………………… 6
   1. mycat ………………………………………………………………………………… 6
   2. myls …………………………………...……………………………………………… 8
   3. mysize ……………………………………………..………………………………..10
3. Conclusions …...……………………………………………………………..………………………. 12

**1. Description of Code**

**mycat**

**myls**

**mysize**

**1. Description of Tests**

**mycat**

The first program, **mycat**, was intended to open a file specified by argument and show the “whole contents” of the file through standard output using the calls *open, read, write,* and *close*.

*Test 1: Basic Functionality*

Evidently, **mycat** should thus accomplish the same functionality as the built-in Linux function **cat**. Thus, our first test of **mycat** included a comparison of the output of running the program on f1.txt, a small file provided in the p1\_tests directory, with the output of running **cat** on f1.txt. We would expect the output to be the same, and indeed, the output was the same. This thus proved that **mycat** indeed reads files with content and writes such content to standard output because that is what **cat** does.

*Test 2: Functionality with Empty Files*

One edge case to test is whether or not **mycat** knows how to handle empty files. This is important because having 0 bytes of content in the file should function no differently than having >0B of content as the “read” function should still return a non-negative number for such a file (it would return 0 to be precise) such that the read and write loop still functions correctly, treating a 0B content file the same way as a 7B content file. The only way “reading” would fail is if <0 is returned from the read function, which should not be the case if 0 is returned. Thus, we tested whether running **mycat** on f3.txt, an empty file added to the p1\_tests directory, resulted in no error, nor any output as was our expectation. Because it resulted in neither, we are confident that **mycat** can correctly read and write the contents of a 0-byte file.

*Test 3: Functionality with Large Files*

Another edge case to test is on the opposite end—whether or not **mycat** knows how to handle large files. This is important because even if the number of bytes in the file surpasses the buffer size, the buffer should have no problem reading and writing a set amount of bytes (1024 in this case), and then reading and writing in the next set amount of 1024B in the file, and so on for however many bytes are in the file. Thus, no matter what, the entire contents of a file should be written out and processed because all that will happen is that 1024B are read at a time, until the end of the file is reached. We thus created f4.txt, a file of 75KB that we added to the p1\_tests directory, and on which we ran **mycat**, expecting no errors, and the entirety of the file output. Doing so resulted in no errors, and every character of f4.txt was actually output, so we know **mycat** is successful on larger files.

*Test 4. Functionality with Files with no-read permission*

Another edge case to test is to ensure **mycat** results in an error when a file cannot be read. We thus added file f5.txt, a file with no read-write-execute permissions (changed by the command **chmod 000 f5.txt**), and tested **mycat** on it, expecting permission errors when ran because reading it should return -1, indicating program failure. Indeed, we received an “Error opening file: Permission Denied” error when ran, successfully guaranteeing that **mycat** understands file permissions and cannot read the contents of a non-readable file, and thus cannot output its contents.

**myls**

The second program, **myls**, was intended to open a directory passed as a parameter (or the current directory if none specified), and print on separate lines, all entries in the directory using the commands *opendir, readdir, printf,* and *closedir*.

*Test 1. Basic Functionality*

Evidently, **myls** should list the same directory entries, in the same order that are listed when **ls –f –l** is called (which lists directory contents unsorted). This is because **myls** was intended to have the same functionality as **ls,** which reads the contents of a directory and lists all of them. Thus, our first test of **myls** included a comparison of the output of **myls** and **ls –f -l** both on the current directory (by specifying no directory). And indeed, the output corresponded, with both commands listing the existence of the same 15 files/directories in the same order. The lack of errors with no directory specified also affirms the correct default functionality of **myls** in listing current directory entries if a specific directory is not specified.

*Test 2. Functionality with Empty Directories*

An edge case to test is whether **myls** works on directories that have no explicit file or directory contents. It should work, with your expectations being that the only things that would get listed with **myls** are . and .., references to the current directory and parent directory, but not explicitly listing any other contents since there are none. Even with an empty directory, **myls**, should still be able to function in reading the directory since the only time “readdir” fails (i.e. returns NULL, which signals the program to return -1) is if the directory cannot be accessed, which is not the case with empty directories (unless set to be). And indeed, when running **myls** on a created empty directory with read-write-execute access, “empty,” there are no errors, and only . and .. listed as entries.

*Test 3. Functionality with Non-Directories*

Another edge case to test is that **myls** does not work when the argument specified is a file, not a directory. We would expect that when **myls** is run on a file, that “opendir” will return NULL, because there is no directory stream to which it can point. Returning NULL would lead to the program returning -1, indicating a failure, and the error, which we expect to be a compatibility error, would be printed. And indeed, when we ran **myls** on mycat.c, a file, we could a “couldn’t open directory: Not a directory” error, indicating that “opendir” had correctly failed to stop the completion of **myls**, which is necessary because a file does not have directory entries to list since it is not a directory.

*Test 4. Functionality with Directories with no-read permission*

Another edge case to test is that **myls** does not work when the directory specified as the argument has no read permissions. This is because “opendir” would fail (i.e. return NULL) if a directory is not accessible, to signal the program should return -1. Thus, you would expect that running **myls** on a directory with no read permissions would return a permission error. Indeed, running **myls** on “noread,” a directory created with no read-write-execute permissions (by the command **chmod 000 noread**), returned a “couldn’t open directory: Permission denied” error, and inhibited the completion of **myls**, which is correct because it should not be able to read and list the contents of a directory it does not have access to.

**mysize**

The third program, **mysize**, was intended to obtain the current directory and list all the regular files that it contains along with their sizes in bytes using the commands *opendir, readdir, open, lseek, close, printf,* and *closedir*.

*Test 1. Basic Functionality*

To the above effect, we would expect that after running **mysize** from within a directory like p1\_tests, with .txt files of certain sizes, we could compare the output to the output of running **ls –f –l** within that same p1\_tests directory and be able to manually verify that all the regular files listed with the latter appear in the former run. We could also compare the sizes listed to see that they match. Thus, for the original p1\_tests directory provided, which contained f1.txt and f2.txt, we would expect **mysize** run from within p1\_tests to produce an output of only one line of “f1.txt 87” and another of “f2.txt 87”because running **ls –f –l** indicates that p1\_tests only contains those two regular files, and references to two directories (. and ..) which should be ignored because they are *not* regular files. It also indicates that their sizes are both 87. And indeed, our expectations were met when running **mysize** on the original p1\_tests.

*Test 2. Functionality with Empty Directories*

An edge case would be to see how **mysize** handles current directories that are empty. To that effect, we ran **mysize** from empty, an empty directory, expecting that no errors nor no output would result. There should be no errors because **mysize** should have no problem reading empty directories. All that would happen is that when an empty directory, dirp, is read, dp, a pointer to a dirent structure referencing the next entry in the dirp would merely only ever point to the references to . and .. that exist within every directory. Because those are not regular dirent structure types, they would merely be skipped over (bypassing any attempts to “open” them) until readdir recognizes there are no other entries exist in the directory, thereby setting dp = NULL and closing out dirp with nothing ever having been printed (i.e. no output). Indeed, our expectations were met when we ran **mysize** from empty.

*Test 3. Functionality with no-read Files*

An edge case would be to see how **mysize** handles current directories that have files that lack read permissions. To that effect, we ran **mysize** from p1\_tests, having added f5.txt, a file with no-read permission. We expected that until that point, any files read (i.e. f1.txt 🡪 f4.txt) would have their data outputted with their associated file size, but when f5.txt would be read, “open” on that file would return a file descriptor value <0, so an open error would be printed and the program would exit because -1 would be returned to indicate the error inhibiting completion. Indeed, when we ran **mysize** on this updated p1\_tests directory, files f1.txt to f4.txt were read and sizes output, but when the program reached f5.txt, the open error “Error opening the file: Permission denied” was printed and the program exited upon return.

*Test 4. Functionality with no-read Directories*

An edge case would be to see how **mysize** handles current directories that lack read permissions. To that effect, we ran **mysize** from noread, a directory we created with no read-write-execute permissions (using the command **chmod 000 noread**). We expected the output to be non-existent, with only a permission error printed. This is because an attempt to “opendir” on noread should have resulted in a NULL pointer reference for a directory stream, signaling that a permission error would be printed and -1 would be returned to exit the program. Indeed, running **mysize** on noread, we got the error “couldn’t open directory: Permission denied,” and no output.