Reading Directories

In this project you will write your non-trivial first assembly language program from scratch. The program will use library calls to open the given Linux directory (or the current directory if no command line argument is given) for reading. It will read every directory entry, printing out each file's inode number, type and name.

Samples

Output when no command line argument is given

```
perryk@ROCI pk_dirent % ./a.out
40905818             0x04 .
7943328              0x04 ..
40905889             0x08 README.md
40908053             0x08 a.out
40906048             0x08 main.c
40905888             0x08 .gitignore
40905819             0x04 .git
40907182             0x04 .vscode
perryk@ROCI pk_dirent %
```

The first column is the named file's inode number. Think of an inode number as a file's unique (per file system) serial number. You must print it left justified in a field of 20 digits using `printf` - since it is possible you've never used `printf` before, I will supply the correct (for `C`) statement:

```
printf("%-20llu 0x%02x %s\n", de->d_ino, de->d_type, de->d_name);
```

In my code, `de` is defined:

```
struct dirent * de;
```

The second column is the file's type printed as a single byte's worth of hex.

The third column is the file's name.

Output when a command line argument is given

```
perryk@ROCI pk_dirent % ./a.out /
2                    0x04 .
1                    0x04 ..
1152921504606781440  0x0a home
1152921500312809862  0x04 usr
1152921500312809678  0x04 bin
1152921500312809756  0x04 sbin
1152921500311879697  0x08 .file
1152921500312809753  0x0a etc
1152921500312845202  0x0a var
```

```
1152921500311879700  0x04 Library
1152921500311879701  0x04 System
1152921500311879696  0x0a .VolumeIcon.icns
1152921500312809755  0x04 private
1152921500311879698  0x04 .vol
1152921500312809676  0x04 Users
1152921500311879699  0x04 Applications
1152921500312809754  0x04 opt
1152921500312809752  0x04 dev
1152921500312809677  0x04 Volumes
1152921500312809861  0x0a tmp
1152921500312809751  0x04 cores
perryk@ROCI pk_dirent %
```

Output when a bad command line argument is given

```
perryk@ROCI pk_dirent % ./a.out fooble
fooble: No such file or directory
perryk@ROCI pk_dirent %
```

The error string is produced by `perror()`.

man

Here is where you will get documentation for `perror()`, `opendir()`, `closedir()`, and `readdir()`. The man page for `readdir()` also describes `struct dirent`.

```
man perror
man opendir
man closedir
man readdir
```

`man` is your friend, though of course in the 21st century it should be called `person`. To learn more about `man`, do the obvious thing:

```
man man
```

"Just" 439 lines.

DON'T DO THIS FROM A MAC TERMINAL – WHY? STEVE JOBS THAT'S WHY.

It will be equally pointless to try the above Linux shell commands from a Windows command prompt but hey - give it a try. So where should you read these `man` pages? In your ARM Linux VM, of course.

The reason to not read the `man` pages on the Mac is that everything beyond the name of the functions will be different. You know, "Think Different."

`opendir()`

This function takes a NULL terminated `C-string` and attempts to open it as a directory. Get the details from the `man` page. If you get an error return, pass the attempted directory name to `perror()` to get the right error message.

`closedir()`

Call this function to close a successfully opened directory. Get the details from the `man` page.

`readdir()`

Call this function to be given a pointer to the next `dirent` or `NULL` if there are no more (or there is an error). Pay attention to the `man` page to distinguish between no more `dirent` structures and an error. In short, `errno` should be initialized to 0 then checked once you've gotten a `NULL` back from `readdir()`.

Source code to a `C` version

At the beginning of this document I said:

In this project you will write your first assembly language program from scratch.

but here's the source code to my `C` version because you may be just getting started with `C` and Linux programming. And because I'm a wonderful pushover of a professor.

```c
#include <stdio.h>                                                              /* 1 */
#include <errno.h>                                                              /* 2 */
#include <dirent.h>                                                             /* 3 */
                                                                                /* 4 */
int main(int argc, char ** argv) {                                              /* 5 */
    int retval = 1;                                                             /* 6 */
    char * dirname = ".";                                                       /* 7 */
                                                                                /* 8 */
    if (argc > 1)                                                               /* 9 */
        dirname = argv[1];                                                      /* 10 */
                                                                                /* 11 */
    DIR * dir = opendir(dirname);                                               /* 12 */
    if (dir) {                                                                  /* 13 */
        struct dirent * de;                                                     /* 14 */
        errno = 0;                                                              /* 15 */
        while ((de = readdir(dir)) != NULL)                                     /* 16 */
            printf("%-20llu 0x%02x %s\n", de->d_ino, de->d_type, de->d_name);   /* 17 */
        if (errno != 0)                                                         /* 18 */
            perror("readdir() failed");                                         /* 19 */
        closedir(dir);                                                          /* 20 */
```

```
        retval = (errno != 0); // force error return to be 1          /* 21 */
    }                                                                 /* 22 */
    else                                                              /* 23 */
        perror(dirname);                                              /* 24 */
    return retval;                                                    /* 25 */
}                                                                     /* 26 */
```

Line 6 and Line 21

Command line programs return 0 to who called them when all is well. A non-zero return value signifies and error.

Lines 7, 9 and 10

Notice how the program is made to default to the current directory (`"."`) which can be overridden if a command line argument is supplied.

Line 15

`errno` is initialized to 0 and then quizzed to see if it turned non-zero when `readdir()` finally returns `NULL`.

Line 17

Implementing this line is where you will need to calculate the correct offsets to each data member.

See the book chapter on `struct`.

Line 18

The error condition is distinguished from the end of the directory by looking at `errno`.

Getting the address of `errno`

`errno` is an `extern`. To store anything into it (or query its contents), you must have its address. For reasons which will be explained, getting its address is accomplished by calling a library function.

Remember to properly set the return value of `main()`

If all ends well, zero should be returned from `main()`. If any error is found, a value of 1 should be returned.

Check in this way:

```
pk_dirent > ./a.out
4327                0x04 .
```

```
4328                0x04 ..
4329                0x08 main.s
4330                0x08 README.md
4331                0x08 a.out
4332                0x08 main.c
4333                0x08 .gitignore
4334                0x08 project.s
4335                0x04 .git
4336                0x04 .vscode
pk_dirent > echo $?
0
pk_dirent > ./a.out main.s
main.s: Not a directory
pk_dirent > echo $?
1
pk_dirent >
```

`$?` is a shell variable that contains the value returned from the last program run by the shell.

Likely source of error

If you're printing garbage, double check your calculations of offsets within the `dirent`. While this isn't the only explanation, it is a likely explanation.

Setting expectations

I provide the following not as a challenge, but to set your expectations.

My assembly language solution is about 60 lines plus comments. If you find yourself writing much more than this, you're doing it wrong.