CSci 4061: Introduction to Operating Systems (Fall 2015)
Assignment 2: Encoding/Decoding Engine

Due
October 28th, 2015, 11:59 pm. Please work in groups of 2.

Purpose
This assignment will introduce you to file and directory operations in Unix. You will also become familiar with a character encoding/decoding scheme.

Description
Your main program *codec_4061* will be responsible for encoding/decoding all files present in the given input directory using the given scheme.

High-level view
The main program must first parse the command line and decide whether to encode or decode. The program must recursively traverse the input directory and encode/decode each file it encounters. The encoded/decoded files must be placed in the output directory, and must have the same name as the original file. Additionally, you must generate a report file for each run of your program.

Program invocation
The program will be invoked as shown below:
```
$ ./codec_4061 -[ed] <input_directory> <output_directory>
```

To encode all files in the 'encodetest01' directory and save the encoded files in 'output' directory:
```
$ ./codec_4061 -e encodetest01 output
```

To decode all files in the 'decodetest01' directory and save the decoded files in 'output' directory:
```
$ ./codec_4061 -d decodetest01 output
```

Report file
Your program must generate a report file each time it is run. The name of the report file should be `<name>_report.txt`, where `<name>` is the name of the input directory. This file should be generated and placed under the `output` directory.

Each line of the report file must be as shown below:
```
<filename>,<filetype>,<original size>,<current size>
```
where,
`filename`: Name of the input file. (This just needs to be the basename, and not the entire path)
`filetype`: Should be one of "`regular file`" or "`directory`"
`original size`: Size of the file before encoding/decoding
`current size`: Size of the file after encoding/decoding

For directories, `original size` and `current size` can be 0. Additionally, the entries in the report file must be sorted alphabetically. As an example, here are the contents of `encodetest02_report.txt`:

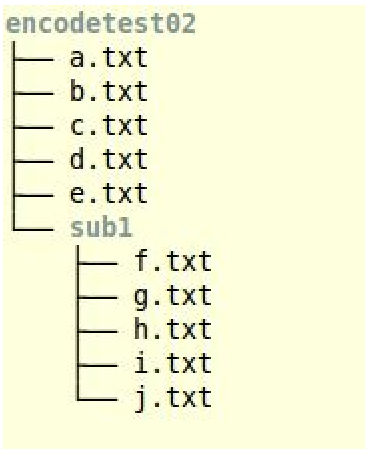
```
$ cat encodetest02_report.txt
a.txt, regular file, 0, 0
b.txt, regular file, 2, 5
c.txt, regular file, 3, 5
d.txt, regular file, 4, 9
e.txt, regular file, 66, 89
f.txt, regular file, 0, 0
g.txt, regular file, 2, 5
h.txt, regular file, 3, 5
i.txt, regular file, 4, 9
j.txt, regular file, 66, 89
sub1, directory, 0, 0
$
```

Directory structure

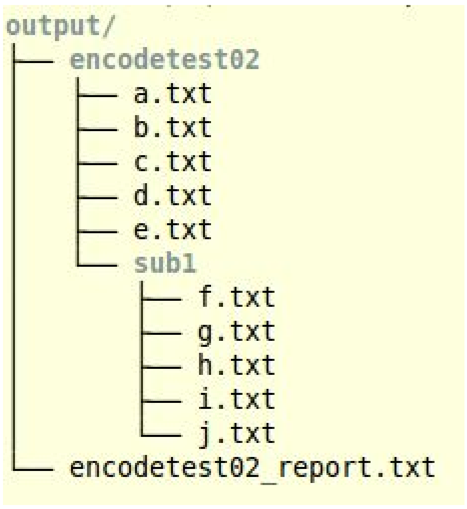
Given below is the structure of the `'encodetest02'` input directory. When executed, your program must:

1. Replicate the same structure of `encodetest02` under the `'output'` directory
2. Encode each file it finds under `encodetest02`
3. Generate a report file named `encodetest02_report.txt`

```
$ ./codec_4061 -e encodetest02 output
```



```
encodetest02
├── a.txt
├── b.txt
├── c.txt
├── d.txt
├── e.txt
└── sub1
    ├── f.txt
    ├── g.txt
    ├── h.txt
    ├── i.txt
    └── j.txt
```



```
output/
├── encodetest02
│   ├── a.txt
│   ├── b.txt
│   ├── c.txt
│   ├── d.txt
│   ├── e.txt
│   └── sub1
│       ├── f.txt
│       ├── g.txt
│       ├── h.txt
│       ├── i.txt
│       └── j.txt
└── encodetest02_report.txt
```

Encoding

You may use the provided `encode_block()` API for encoding. The function prototype is:

```
/* encode 3x 8-bit binary bytes as 4x '6-bit' characters */
size_t encode_block(uint8_t *inp, uint8_t *out, int len);
```

- `inp` should be an array of 3 characters/bytes
- `out` should be an array of 4 characters/bytes
- `len` should hold the length of the input array
- The function will return the length of the encoded block

To encode a file, you will need to read the file 3 bytes at a time, call this API to encode the block of 3 bytes and write the encoded bytes to a new file.
NOTE: You need to write a newline to the file when you have finished encoding it. The newline should be written to the file only if the file contains some encoded characters - In other words, if the file to be encoded is empty, then you should not write the newline to the encoded file (The encoded file should just be an empty file if the original file is empty). You can write a newline to the file by using the following snippet of code (fp is a file pointer to the file):

```
unsigned char ch = 0x0a;
fputc(ch, fp);
```

Decoding:
You may use the provided `decode_block()` API for decoding.
The function prototype is:

```
/* decode 4x '6-bit' characters into 3x 8-bit binary bytes */
size_t decode_block(uint8_t *inp, uint8_t *out);
```

- `inp` should be an array of 4 characters/bytes
- `out` should be an array of 3 characters/bytes
- The function will return the length of the decoded block.

To decode a file, you will need to read four (4) **VALID** bytes from the file at a time, call this API to decode the block of 4 bytes and then write the decoded bytes to a new file. You can use the `is_valid_char()` API to determine a byte's validity.
NOTE: You DO NOT need to write a newline to end of the decoded file. The `decode_block()` API will take care of adding the newline to the end of the file.

Byte/character validation (Required for decoding)
You may use the provided `is_valid_char()` API to determine whether a particular byte/character from a file is valid or not. The API will return `1` if '`val`' is a valid byte or `0` if invalid.

```
/* Returns 1 if 'val' is a valid char under the encoding scheme */
int is_valid_char(uint8_t val);
```

Creating a directory
You can use the mkdir() system call to create a directory. Check the man pages for mkdir:

```
$ man 2 mkdir
```

You can use the following snippet of code to create a directory called "`new_dir`" under the `output` folder in the current working directory.

```
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>

int retval = mkdir("output/new_dir", 0755);
/* If retval is 0, then directory was created successfully */
if (retval == -1) {
     fprintf(stderr, "mkdir() failed: %s\n". strerror(errno));
     exit(EXIT_FAILURE);
}
```

Failure to pass in a valid mode argument (0755) to the mkdir API will cause you to run into issues like not being able to open the directory or create files in the directory.

Useful System Calls & Functions:
It is highly suggested that you make use of the following system calls or library functions:
`opendir, readdir, closedir, stat, lstat, fopen, fread, fwrite, fgetc, fputc, feof, fflush, mkdir`
You may copy and cite code from the text.

Testing
As you might have already noticed, your program can both encode and decode files. To test the correctness of your program, you may wish to first encode a file and then decode the encoded file. You can then use a tool like diff to compare the contents of the decoded file with the contents of the original file before encoding.

Simplifying assumptions:
- The specified input directory will exist and will not be empty.
- The specified input directory will consist of only text files and subfolders.
- Subfolders might be nested to any depth and will contain text files too.
- There will be a maximum of 64 files under any input directory. (All the 64 files might be in a single directory or they can be spread across multiple subdirectories)
- Each file/directory will have a unique filename.
- Files can be of any size.
- The specified output directory will exist.

Error handling:
You are expected to check the return value of all system calls that you use in your program to check for error conditions. Also, your main program should check to make sure the proper number of arguments is used when it is executed. If your program encounters an error, a useful error message should be printed to the screen. Your program should be robust; it should try to recover if possible. If the error prevents your program from functioning, then it should exit after printing the error message. (The use of perror function for printing error messages is encouraged)

Documentation:
You must include a readme file, named “README” without any extensions, which describes your program. It needs to contain the following:
- The purpose of your program
- How to compile your program
- How to use the program from the shell (syntax)
- What exactly your program does
- Your x500 and the x500 of your partner
- The CSELabs machine on which you tested the code

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion. Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of the README file and all other files you submit, please include the following comment:

```
/*
 * name: <full_name1>, <full_name2>
 * x500: <x500_student1>, <x500_student2>
 * CSELabs machine: <machine1>
 */
```

Deliverables:

You are required to submit online, a single file (.zip or .tar or .tar.gz) containing:

- A README file
- Source code: .c and .h files
- A Makefile that will compile your code and produce a program called `codec_4061`.
- Do NOT include the executable.
- Do NOT include the test cases.

Note: This makefile will be used by us to compile your program with the make utility. If your provided makefile doesn't work then you will not receive full credit for the test cases. See the references page on the course site for links on creating makefiles. All files should be submitted using Moodle. This is your official submission that we will grade. Please note that future submissions under the same assignment title will OVERWRITE previous submissions; we can only grade the most recent submission. Make sure you have tested the code you upload to moodle.

Grading:

5% - Readme file
15% - Coding style (indentation, readability of code, use of defined constants rather than numbers), documentation within code, Makefile
80% - Test cases (correctness, error handling, meeting the specifications)

- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read. It will not receive full points if your readme file lacks the correct name ("README") or does not include your (and your partner's) x500.
- You will be given most test cases up front. You are also encouraged to make up your own tests. If there is anything that is not clear to you, you should ask for clarification.
- We will use the GCC compiler installed on the CSELabs machines to compile your code. Make sure your code compiles and runs on the CSELabs machines.

Extra credit:

For extra credit (10 points), your program must handle hard links and soft/symbolic links. The folders `extracredit_encode01/` and `extracredit_decode01/` will be used for testing. Your program must identify hard links and soft/symbolic links, and take the following action:

1. **Soft links:** Ignore the soft/symbolic links and add the following line to the report file-
   `<filename>, sym link, 0, 0`

2. **Hard links:** Your program must process and encode/decode only one file corresponding to this hard link. All the other files which correspond to this hard link must be ignored, and the following line should be added to the report file-

```
    <filename>, hard link, 0, 0
```

Note: Symbolic links can be present for both files and directories. Hard links will be present only for regular files - however, the files may not be present in the same directory, and may reside in different subdirectories.