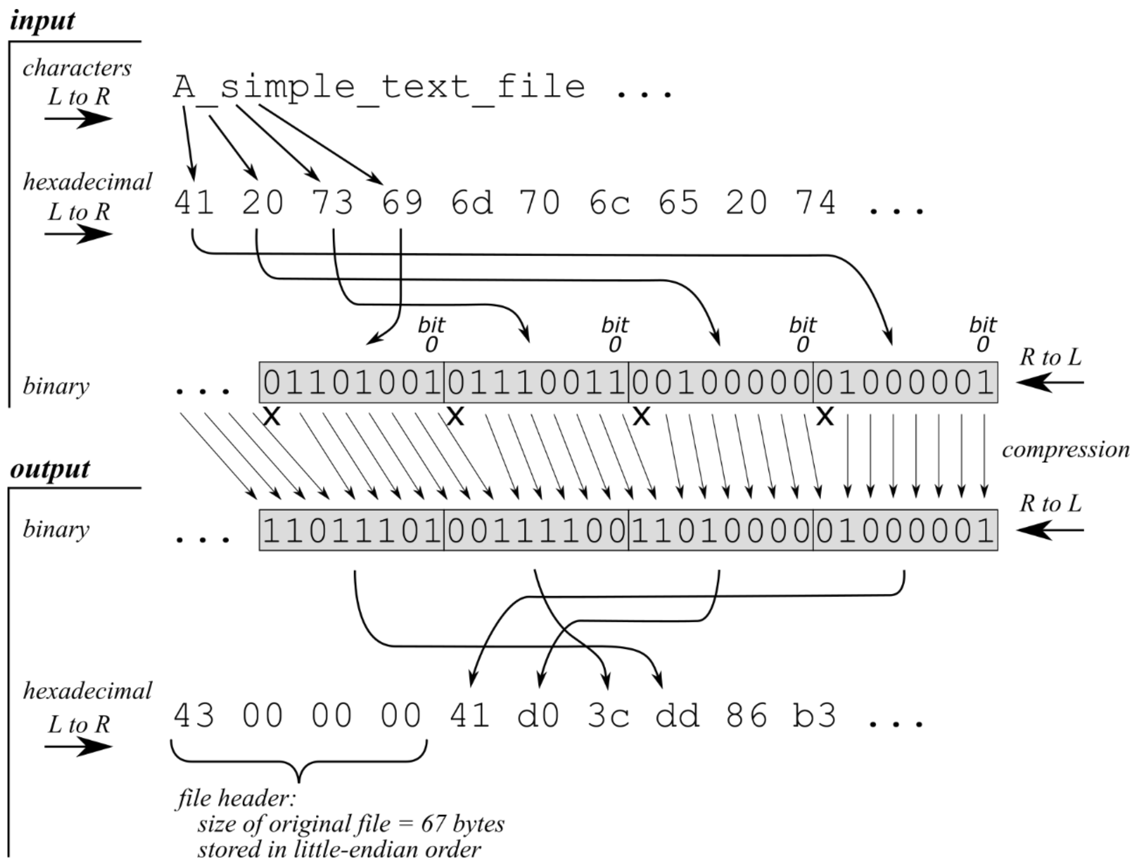
COS 350: Program #2: z827 compression

Objectives: Working with strings and command line arguments. Working with **binary files** (open, close, read, write, creat, lseek, unlink.) Working with machine representations of data and using bitwise operations. Using English writing skills to write a manual page.

You will write a simple compression algorithm that takes a standard 8-bit ASCII text file and converts it to a compacted 7-bit encoding. This utilizes the fact that in normal ASCII text the top bit is always zero, and thus is not necessary. The diagram below shows the process.



At the top of the diagram the input file is shown in three formats: as a stream of characters, as the hexadecimal encodings of those characters, and as a stream of binary bits. This data is from the example file short.txt. There is a Linux utility program od (stands for octal dump) that can display files in a variety of formats. The command od –t x1 short.txt will display the file in hexadecimal chunks of a size 1 byte, and thus display the values as shown on the second line. The third line is important for understanding this assignment. The bytes are showing in binary from Right to Left because this is the order that we, as humans, think of the digits of a number. The top bit of each byte is marked with an **x** indicating that it is dropped during compression.

The bottom of the diagram shows the output. First it shows the collected binary bits and how they are regrouped into bytes. Then it shows these bytes encoded in hexadecimal as might be seen by using od on the output file. A 32 bit integer (4 bytes) is written at the start of the output file. It contains the length of the original input file. This is needed to correctly decompress the file because there may be up to 7 extra unused bits at the end of the encoding.

Requirements:

1. Use binary files. (open, close, read, write, creat, lseek, unlink) These manual pages are all in section 2. E.g. man 2 read Use unlink to remove a file.
2. Use buffers to read and write the file in large chunks. I suggest 8 kilobytes. Do not read or write 1 byte at a time; that would make it much slower!
3. Your program should take one file name on the command line.
4. If it is an ordinary file it should compress it and append the extension .z827 to the file name. E.g. short.txt goes to short.txt.z827
5. If it is already a compressed file (based on the extension), it should decompress it into a file with the original name.
6. After compressing or decompressing, the original file should be removed. Be careful and never remove a file until you are sure that the result file has been successfully written and closed. (Users get very upset if you lose their file!)
7. If the file to be compressed has any characters that have an eighth bit set, you can simply report that the file is not compressible and exit.
8. You should create a document describing your program and how to use it. Write this document on a word processor of your choice and organize and format it in the style of a man page. For an example see man gzip. **Do a careful job of this. I will be grading it for completeness, grammar, spelling, and formatting.**

Suggestions:

1. Make a new directory for just this project.
2. Copy the 2 example files short.txt & shake.txt from the course directory /usr/class/cos350/prog2
3. Keep it simple! The core of the compression algorithm can just be a simple loop that processes 1 byte at time. Mine used a single unsigned int as a bit-buffer, and a second integer to count bits. Each new input character added 7 bits onto the top of the bit-buffer, and whenever the bit-buffer grew to 8 or more bits, the bottom 8 bits were removed and sent to the output buffer.
4. Review C's bitwise operators: >> right shift, << left shift, | bitwise OR, & bitwise AND. You will need to use these to shift bits around, merge them together, and regroup them into new bytes.
5. Use types unsigned char and unsigned int to avoid possible problems that can happen when bit shifting signed types.
6. If 8K buffering seems complicated, just use large buffers that can hold the entire file.
7. You can make simplifying assumptions about the length of file names and the length of the files. (It should certainly at least accommodate the 2 example files.)
8. Work in stages. Get small pieces working, and then gradually add more functionality.
9. Work on compression first and get it completely working. Decompression is very similar and you can reuse much of your code, plus decompression is only worth 20% of the total points, so if you run out of time, you can skip it and still get most of the credit.
10. Always check the return values of library calls and print an error message if something goes wrong. This will be a big help in debugging your code if a system call is failing.
11. Use the debugger to step through your program and see what is going on. In gdb/ddd you can print out the value of a variable in binary. E.g. p /t bitbuf or graph display /t bitbuf
12. Writing an integer to a binary file for the header can be done with: write(fd, &yourVariable, 4);

What to turn in:

Written report:

1. Your name(s) and, if a team, who did the electronic submission.
2. A discussion of any incomplete parts, known bugs, deviations from the specification, or extra features in your program.
3. Your manual page.
4. Your code.
5. A log of the following testing results. (I recommend using **script** to capture it. Then use **scriptCleaner** to clean up any typos.)

```
cp short.txt short.orig             save copies of the original files
cp shake.txt shake.orig
z827 short.txt                      compress the files
z827 shake.txt
ls -l                               check that the original files have been removed
od -t x1 short.txt.z827             print a hex byte dump of the short file only
z827 short.txt.z827                 uncompress the files
z827 shake.txt.z827
ls -l                               verify the compressed files have been removed
cmp short.txt short.orig            verify we got back our original files
cmp shake.txt shake.orig              note: cmp prints nothing if the files match
z827 badfilename                    does it gracefully handle a bad file name
z827                                does it gracefully handle a missing file name
```

Electronic submission:

- Before you submit your program, compile and test it on a **Linux** machine in the lab.
- From a machine in the lab run the program "submit" to submit your files.
- Submit your source code (z827.c) and your executable (z827) to a directory named: **prog2**
- **Do not in any way combine, compress, zip, or tar your files!**

Grading:

10 points: Manual page
10 points: Reading and writing the input file using read() and write() and large buffers
10 points: Creating the correctly named output file
30 points: Correctly compressed data
 5 points: Correct file size at the start of the file
 5 points: Removing the old file after processing
20 points: Decompression
10 points: error handling