CS 0449 – Shared Objects

Background

We are going to go above and beyond our normal activities this week by writing a compression / decompression utility! Don't worry… you aren't writing the algorithm itself. For that, we will make use of an existing software library called zlib to do this for us.

For this worksheet, we will make use of the zlib library to create our main program. Along the way, we will deal with command-line arguments and, somewhat more devious, developer documentation of zlib. You should open those links up in browser tabs and refer to them as needed.

Procedure

1. Login via SSH to `thoth.cs.pitt.edu`

2. Change directories to one you created for the work for this class.

3. Then, create a directory for this worksheet and navigate to it:

```
mkdir shared
cd shared
```

4. We will need some files I will provide to help you test your program. Refer to the resources page on the course website for information about how to get these files. Only proceed once you've downloaded them to your local directory.

5. The first step will be to create our main C file (using the terminal editor of your choice):

```
nano compressor.c
```

6. Now enter our initial code: (we now have two arguments to main: argc which is the number of terms on the command-line, and argv is an array of strings for each term.)

```
int main(int argc, char** argv) {
    return 0;
}
```

7. Save the file

8. We now need to compile this. You can make a Makefile if you want! You may find it helpful.

```
gcc -o compressor compressor.c
```

9. We can run it with `./compressor` however, we know it does not do anything right now. We need to have it react to the command-line arguments. We want this behavior:

```
./compressor -c file        - Compress the given file
./compressor -d file        - Decompress the given file
```

If we give it less than the necessary arguments (`argc` is less than 3) or the flag is not a `-c` or `-d`, then we want the program to end as a failure and tell the user what went wrong. Update your program to do the following:

NOTE: `fprintf` prints to a `FILE`. `stderr` is a special `FILE` that prints non-output to the screen. We will make use of this to avoid mixing informational messages with our output data later!

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv) {
    if (argc < 3) {
        fprintf(stderr, "error: not enough arguments provided.\n");
        fprintf(stderr, "       use -c or -d followed by a path.\n")
        return 1;
    }
    return 0;
}
```

10. Recall that `argv`, albeit a scary `char**`, is just an array of `char*` elements. It is an array of strings, much like what we expect from Java. How do we check a command-line argument? (**NOTE**: argv[0] is the first term specified on the command-line: a string containing the executable name)

Add this code to your main function after checking for the number of arguments.

```c
if (!strcmp(argv[1], "-c")) {
    fprintf(stderr, "compressing...\n")
}
```

11. Let's focus on compressing right now using the `-c` flag. How do we interact with the zlib library? For now, we will dynamically link to the library. To do this, we need to include the library routines by including the necessary header file, `zlib.h` at the top of our file:

```
#include <zlib.h>
```

12. This gives us access to all of the routines provided by the zlib library. Refer to the documentation link in the introduction section. We will be looking at three different functions: `compressBound`, `compress`, and `uncompress`.

    When we look at `compress()`, we see it takes a pointer to a buffer with the source data and the length of the buffer. This means we need to read the entire file to a buffer before we make use of the `compress()` function.

    How do we learn how big a file is? We can use the `fseek()` function to go to the end of a file and then use the `ftell()` function to tell us what byte offset we are currently at within the file (the end.)

```
FILE* f = fopen(argv[2], "rb"); // open the file in binary mode
fseek(f, 0, SEEK_END);          // go to end of file
long size = ftell(f);           // get file position (the end of the file)
fseek(f, 0, SEEK_SET);          // go back to beginning of file
```

13. Using the code above, add code within the block for compressing the file to create a buffer large enough for the entire file and use fread to read in the file contents.

```
if (!strcmp(argv[1], "-c")) {
    FILE* f = fopen(argv[2], "rb"); // open the file in binary mode
    fseek(f, 0, SEEK_END);          // go to end of file
    long size = ftell(f);           // get file position (the end of the file)
    fseek(f, 0, SEEK_SET);          // go back to beginning of file

    char* source = malloc(sizeof(char) * size);
    fread(source, size, 1, f);      // read the entire file into the buffer
}
```

14. Now, the `compress()` function also takes a buffer for the compressed data. We have no idea how big this might be!! We know the peril of the buffer overflow, so we should use `compressBound()` to determine how big that buffer should be:

```
    long destSize = compressBound(size);
```

15. Add the above after the file read and try to compile… oh no!

```
/tmp/ccjDxmxg.o: In function `main':
compressor.c:(.text+0xe6): undefined reference to `compressBound'
```

This happened because the code for compressBound does not exist! Our linker could not resolve that symbol. Hmm. We need to tell the linker where that function is implemented.

Thankfully, it is implemented in the libz.so shared object file which is already on our thoth machine. It is in a standard place: `/usr/lib64/libz.so` which we can just add on to our gcc command line:

```
gcc -o compressor compressor.c /usr/lib64/libz.so
```

However, to simplify things and have the flexibility to have the libraries in different directories, our linker already knows where the library files generally are (the /usr/lib and /usr/lib64 directories, on this system) so you can specify a common shorthand flag:

```
gcc -o compressor compressor.c -lz
```

Note that we drop the "lib" prefix and the ".so" suffix along with the absolute directory. These are always implied when the linker goes in search of the needed `*.so` file. (a Makefile is becoming more and more necessary!)

Your program should now compile without a problem. Let's actually compress!

16. Once we have the upper-bound for the destination buffer, we can allocate it and pass it to the `compress()` function: (don't forget to add the free calls to the bottom of the block!)

    After the `compress()` call completes, `dest` will be filled with the compressed data. Since we passed the `destSize` variable by-reference, it will be updated with the total size of the compressed data. It will likely be less that the upper-bound, after all.

    We will then write out the uncompressed size (which is used when decompressing) and the compressed data directly to the screen using the special `FILE` called `stdout`.

```
    char* dest = malloc(sizeof(char) * destSize);
    compress(dest, &destSize, source, size);
    fwrite(&size, sizeof(long), 1, stdout);  // write uncompressed size
    fwrite(dest, destSize, 1, stdout);       // write compressed data
```

17. And then re-compile our executable and run it be compressing the given file you downloaded earlier:

```
./compressor -c img1.bmp > img1-compressed.bin
```

NOTE: we are redirecting the program's standard output to a file because the raw compressed data is not really human-readable and would really mess up our terminal. Think Raiders of the Lost Ark where they open the Ark and the bad guys all get vaporized. That kind of thing happens. Here, the file img1-compressed.bin is created with the compressed data.

SUPER NOTE: Things printed with "stderr," however, are not redirected and you will still see them printed on the screen. Make sure your prints are using fprintf and stderr so they don't End up in the compressed data stream!!

Your turn…

1. Update the program to also decompress by using the `uncompress()` function provided by zlib. Make note of the documentation for the function and, from what you already know, piece together what you need to do

```
./compressor -d img1-compressed.bin > img1-copy.bmp
```

2. You can determine if two files are identical by running the diff program and seeing if there is absolutely no output. (It only tells you when there are differences.)

```
diff img1.bmp img1-copy.bmp
```

3. Happy coding!

Hints

1. Reading a long from a `FILE` is just as easy as writing it:

```
    fread(&destSize, sizeof(long), 1, f);  // read uncompressed size
```

2. This is a useful number… because it tells us how much to allocate for our decompressed data stream we will pass to `uncompress()`! If you got the size of the file, remember to subtract the size of a long from that when you read in the rest of that file.