CS 218 – Assignment #11, Part A

Purpose: Become more familiar with operating system interaction, file input/output operations, and file I/O buffering.
Points: 250 (grading will include functionality, documentation, and coding style)

<u>**Assignment:**</u>

Write an assembly language program that will read a file, add lines numbers in ASCII/Base-13 format, and write the new line (line number with the original line) to another file. The program should read both file names from the command line. The program should allow the arguments to be entered in either order (input file or output file first). The program must perform error checking. If no command line input was entered, invalid or incorrect file names were entered, or no file matching the specification was found an appropriate error message should be generated and the program terminated. An example command line is:

"It was terrible. My life flashed before my eyes. Then it started buffering."

```
 ed-vm% ./addlines -i file.txt  -o nfile.txt
```

The line numbers must be BASE13_STR_LEN characters long which includes the base-13 string with leading 0's, terminated in a colon "**`:`**" followed by one space. For example, if the original file line was "**`Hello World`**", the new line to be written into the new file would be "**`00000001: Hello World`**".

To ensure efficiency, the program ***must perform buffered input***. The buffer size should be set to BUFFSIZE which is declared as a constant. *Note*, the buffer size will be changed in the next assignment, so all code should reference the buffer size constant (not hard code the number).

The provided main program calls the following routines:

- A value returning function, **getFileDescriptors()**, that reads and checks the command line arguments. The command line arguments must be in the format of: **`-i <inputFile> -o <outputFile>`** ***in either order***. To check the file name, attempt to open the file. If the files open, the routine should return the file descriptors. If there is an error, an appropriate error message should be displayed (see examples). The function should return a boolean result (FALSE or TRUE).

- A value returning function, **getLine()**, that should return a single line of text. In order to perform this efficiently, the routine must perform input buffering. As necessary, data from the file should be read into a primary buffer (BUFFSIZE). If the line is longer than the passed maximum length, MAX_LINE_LEN (which includes the NULL), the over limit bolean should be set to TRUE and set to FALSE otherwise. If the line limit is reached, a line feed should be added to the line being return (MAX_LINE-LEN-2 characters, LF, and a NULL). The remaining characters must be read but not returned (and are thus lost). When the primary buffer is empty, the routine should fill the primary buffer by reading the file. The function must handle any unexpected read errors. The function should return FALSE if there are not more characters and TRUE otherwise.

- A void function, **addLineNumber()**, that will add the line number (in base-13). A new line will be created from the current line (max lenght is passed) and the ASCII/Base-13 line number (max length is passed). The integer to ASCII/base-13 conversion function, **cvtint2B13()**, will be called by this function. The integer and current line address are passed, and the maximum line length are passed to the **cvtint2B13()** function. The new line index must be updated by the passed maximum line length to skip over the line number in the new line.

- Void function **cvtint2B13()** to convert the passed integer into a base-13 string. This function will be called by the **addLineNumber()** function. The integer and the starting address of the string should be passed

- A value returning function, **writeNewLine()**, to write the new text line (with line number) to the output file. The function must handle any unexpected write errors by displaying a message (pre-defined). The function should return a boolean result (FALSE or TRUE).

A main program and a functions template will be provided. All functions must be in a separate source file, ***a11procs.asm***, that is independently assembled. Only the functions file, not the provided main, will be submitted on-line. As such, you must not change the provided main! You may use static variables as needed (no requirement to create stack-dynamic locals).

Submission:

- All source files must assemble and execute on Ubuntu and assemble with **`yasm`**.
- Submit source files
  - Submit a copy of the program source file via the on-line submission.
  - Note, only the functions file (**`a11procs.asm`**) will be submitted.
- Once you submit, the system will score the project and provide feedback.
  - If you do not get full score, you can (and should) correct and resubmit.
  - You can re-submit an unlimited number of times before the due date/time.
- Late submissions will be accepted for a period of 24 hours after the due date/time for any given lab. Late submissions will be subject to a ~2% reduction in points per an hour late. If you submit 1 minute - 1 hour late -2%, 1-2 hours late -4%, … , 23-24 hours late -50%. This means after 24 hours late submissions will receive an automatic 0.

Program Header Block

All source files must include your name, section number, assignment, NSHE number, and program description. The required format is as follows:

```
;  Name: <your name>
;  NSHE ID: <your id>
;  Section: <section>
;  Assignment: <assignment number>
;  Description: <short description of program goes here>
```

Failure to include your name in this format will result in a loss of up to 10%.

Scoring Rubric
Scoring will include functionality, code quality, and documentation. Below is a summary of the scoring rubric for this assignment.

Criteria	**Weight**	**Summary**
Assemble	-	Failure to assemble will result in a score of 0.
Program Header	3%	Must include header block in the required format (see above).
General Comments	7%	Must include an appropriate level of program documentation.
Program Functionality (and on-time)	90%	Program must meet the functional requirements as outlined in the assignment. Must be submitted on time for full score.

Debugging -> Command Line Arguments
When debugging a program that uses command line arguments, the arguments must be entered after the debugger has been started. The debugger is started normally (ddd <program>) and once the debugger comes up, the initial breakpoint can be set. Then, when you are ready to run the program, enter the command line arguments. This can be done either from the menu (Propgram -> Run) or on the GDB Console Window (at bottom) by typing **`run <commandLineArguments>`** at the (gdb) prompt.

Example Executions:
The following will read file "a11File1.txt" and create a file named "tmp1.txt".

```
ed-vm% ./addlines  -i a11f1.txt  -o tmp1.txt
```

The following will read file "a11f1.txt" and create a file named "tmp1.txt" that includes the line numbers. As such, the output file will be larger. The following are some examples of error handling:

```
ed-vm% ./addliness
Usage: addlines -i <inputFile> -o <outputFile>
ed-vm%
ed-vm% ./addlines -z a11f2.txt -of tmp2.txt
Error, invalid specifier.
ed-vm%
ed-vm% ./addlines  -i none.txt -o tmp1.txt
Error, opening input file.
ed-vm%
ed-vm% ./addlines  -i a11f3.txt   -t tmp3.txt
Error, invalid specifier.
ed-vm%
ed-vm% ./addlines -i a11f2.txt -o ~
Error, opening output file.
ed-vm%
ed-vm% ./addlines -i a11f5.txt -o outFile5.txt
Warning, line 5737284 truncated.
ed-vm%
ed-vm% ./addlines -o outFile5.txt -i a11f5.txt
Warning, line 5737284 truncated.
ed-vm%
```

Example Input/Output Files:
Given an example input file containing:

```
Line Number -> 000001
Line Number -> 000002
Line Number -> 000003
Line Number -> 000004
Line Number -> 000005
very long line very long line very long line very long line very
long line very long line very long line very long line very long
line very long line very long line very long line very long line
very long line very long line very long line very long line very
long line very long line very long line very long line very long
line very long line very long line very long line very long line
very long line very long line very long line very long line
Line Number -> 000007
Line Number -> 000008
Line Number -> 000009
Line Number -> 000010
Line Number -> 000011

Line Number -> 000012
Line Number -> 000013
```

And, an execution as follows:

```
ed-vm$ ./addlines -o example.txt -o tmp.txt
Warning, line 5 truncated.
ed-vm$
```

The output file would contain:

```
00000000: Line Number -> 000001
00000001: Line Number -> 000002
00000002: Line Number -> 000003
00000003: Line Number -> 000004
00000004: Line Number -> 000005
00000005: very long line very long line very long line very long
line very long line very long line very long line very long line
very long line very long line very long line very long line very
long line very long line very long line very long line very lon
00000006: Line Number -> 000007
00000007: Line Number -> 000008
00000008: Line Number -> 000009
00000009: Line Number -> 000010
0000000A: Line Number -> 000011
0000000B:
0000000C: Line Number -> 000012
00000010: Line Number -> 000013
```

Note that the blank line was not counted correctly originally, but was counted in the output.
Additionally note that the long line was truncated.