CSE 30 Spring 2022 Programming Assignment #4 - (Vers 1.3)
Due Saturday May 21, 2022 @ 11:59PM

Remember: 'Start Early, Start Often'

Assignment: Counting chars, words and line in C and ARM 32 Assembly

In this PA you will write a program called `cw` (in ARM32) and `ccw` (in C) that both does the following:

1. Reads a buffer at a time until EOF is reached, from either a file specified on the command line or from `stdin` if `cw` (or `ccw`) is run with no arguments.
2. Counts the number of newline characters `'\n'` in the file.
3. Counts the number of words in a file. A word is a non-zero length sequence of characters, delimited by either a space `' '`, a newline `'\n'`, or a tab `'\t'`. Words can start at the first character in the buffer.
4. Counts the number of characters in each word. Space, tabs, and newlines before, after, or between words are *not* counted.
5. After reaching EOF, prints a summary to `stdout`: `"lines:%8.8d words:%8.8d Chars:%8.8d\n"`

You can assume that only printable ASCII characters, including the space, tab, and newline, will be in the buffer. When the buffer is filled, it contains an array of characters. It is *not* a string, so it is not `'\0'` terminated.

The following starter files are the ones you need to edit and complete:

1. `ccw.c`: A starter file containing `main()` for a solution written in C, using only pointers (i.e. no array operators `[]`).
2. `cw.S`: A starter file containing `main()` for a solution written in ARM32 (ARM V6 only)
3. `result.S`: A starter file containing `result()`, used with both the C and ARM solutions

The following functions are provided to you in source for your reference.

1. `rd.S` reads a buffer from either the file specified or from `stdin`.
2. `setup.S` handles `argv` processing and opening of the file.
3. `cw.h` contains the common definition for the size of the buffer. **(Do not change this!)**

Description of the Command Line Options

`./cw [file]` or **`./ccw [file]`**

Flag	Description
`[file]`	Specifies the name of a file to be read where you will count the characters in each word, the number of words and the number of newlines. If the file is not supplied, the program will read

	from `stdin`.

The program counts words, lines and characters in a word. There are no null (`'\0'`) characters in the input, so you must depend on the return value from `rd()` (see below) to know how many characters to process in the buffer.

Here are some examples of how the code operates. A descending "$_b$" is a single blank, a descending "$_t$" is a single tab (`'\t'`) and a descending "$_n$" is a single newline (`'\n'`).

Buffer content	Return of `rd()`	charcount	wordcnt	linecnt
abc$_b$mno$_n$	8	6	2	1
abc$_b$mno$_n$abc$_b$mno$_n$abc$_b$mno$_n$	24	18	6	3
bbbbbabc$_b$mno$_{ttt}$mno$_{nnn}$abc	24	12	4	3
111111111111111111111111	24	24	1	0

Coding and Program Development Requirements

1. You will write your code only in C and ARM V6 assembly. It must run in a Linux environment on the pi-cluster.
2. Only edit the following files. These are the only files you will submit.
    a. `ccw.c`
    b. `cw.S`
    c. `result.S`
3. You cannot use any library functions in this PA, other than the ones supplied in the starter files.
4. When writing the C version of `main()` in `ccw.c`, you may only use pointers. **Do not use array index operators `[]`.**
5. When writing assembly language code, only use register base addressing when accessing memory in all the assembly code YOU write. **Do not use register or immediate offset addressing** when writing your code.
6. You must write all the assembly language by hand. The use of compiler generated (or other tool generated) assembly will NOT be accepted for grading. This is easy to detect and will **result in a zero (0)** for the PA.
7. You may **only use the ARM instructions** listed in the ARM ISA green card. The green card can be found on Canvas under Documents.

8. You must test your program on the pi cluster.

How to Approach This Programming Assignment

The starter files are contained in the `cs30sp22` public folder (`../public` relative to your home directory) in a tarball (a common way to distribute files in Linux). From your home directory, run the following to create the directory PA4 in your home directory:

```
tar -xvf ../public/PA4.tar.gz
```

Inside the directory at the top-level, the following files are of interest:

`Makefile:` The Makefile you will use to develop your program. **Please examine the Makefile** to understand the various targets. There are five targets of importance:

1. `make ccw` – compiles the c version of the program - **write this C version first!**
2. `make cw` – assembles and links the ARM assembly language version of the program
3. `make ccwtest` – runs the test harness on the C version of the program.
4. `make cwtest` – runs the test harness on the assembly language version of the program
5. `make` – same as running both make ccw and make cw

`setup.S:` Supplied code for basic command line argument handling in assembly. You will not have to modify this file, but you should study it to see how argv is accessed using assembly language. This file also demonstrates how to use trap instructions (svc 0) to make OS system calls to `open()` and `dup2()`. You can read about these Linux OS system calls from the Linux command line by doing a: `% man 2 open` and a `% man 2 dup2`. These are I/O calls to set up file descriptors (which are below `FILE *` pointers). This function is called by both the C and assembly language version of `main()`. This function returns a 0 on success and -1 on failure.

`rd.S:` Supplied code that performs the low level read system calls in assembly using traps. This function fills the global variable buffer up to BUFSZ bytes after reading from the command line argument file or stdin and returns the number of bytes stored in the buffer for each read in register r0 (just like all function calls). When EOF is reached, it returns). If an error occurs, it returns -1. You should treat a return of <= 0 as ending your loop. You do not have to handle -1 error returns. You can read about these Linux OS systems calls from the Linux command line by doing a: `% man 2 read`

`result.S:` This is a partial assembly language file which provides an interface to `printf`. This is just for this PA until we cover function calls. (You would normally not use a helper file for `printf`.) In this file, you need to write the commands to read the values in the three global variables `cntchar`, `cntword` and `cntline` into the specified registers before the call to `printf`. You will not need to edit anything else in this file. Both the C and assembly language versions of the program call this function.

`ccw.c:` **Write this first!** This contains the starter code for the C version of main() in this assignment. You will write the C version of `main` here. The program is short so no additional helper functions should be necessary.

You will call `int setup(argc, argv)` to parse argv and set up the open file descriptors, `int rd(void)` to fill the buffer until EOF and `void result(void)` to print the results to stdout. You will place the count of lines in the global integer variable `cntline`, the count of words in the global variable integer `cntword`, and the count of characters in the words in the global integer variable `cntchar`. You will return `EXIT_SUCCESS`, unless `setup()` fails, in which case you would return `EXIT_FAILURE`.

Until you get `result.S` working, you can use `printf` in `ccw.c`. At the bottom of the starter code in the file `ccw.c`, uncomment the following line (to use the result.S solution code):

```
//#define _PA4DBG_ to look like #define _PA4DBG_
```

This will use `printf`. When you complete `result.S`, restore the line to look like `//#define _PA4DBG_`

`cw.S:` This is where you will write the assembly language version of `main()`. All the requirements for ccw.c apply here. Be aware that you can use `' '` as a blank as an instruction constant. For example: `cmp r1, ' '`. You can use the characters `'\t'` and a `'\n'` the same way.

Return values from function calls are always in register `r0`. You will note that the supplied code handles the return values for you. Make sure you understand this, as you will have to write this for yourself in PA5. **You can use only register base addressing mode like `ldr r5, [r6]`. Remember this is an array of characters/bytes in the global variable buffer.**

You can use any of the scratch registers `r0-r3`. Remember that each call to `rd()` may change the contents of `r0-r3`, as they are scratch registers. `rd()` returns the char `cnt` in `r0`. You should treat EOF as a return value of `<= 0`. You can use the preserved registers `r4-r7` safely. Do not use any other preserved registers in this PA. We will cover this in more detail in class before PA5.

`cw.h:` This is the common header file. It is only used to contain the size of buffer BUFSZ.

Submitting your code

Comment and check your code follows the CSE30_C_Style document and the CSE30_ARM_Style document

To get all the style points, read over the style document and clean up your code by following the guidelines. In PA4, style will be more enforced than in PA3.

Submit to Gradescope under the assignment titled "A4: counting words"

After you pass all the MVP tests in the test harness and only then, will you submit to Gradescope the following files:

```
ccw.c      cw.S       results.S
```

You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all the files and upload the .zip to the assignment. Ensure that the files you submit are not in a nested folder.

After submitting, the autograder will

1. Check all files were submitted.
2. Check that the program compiles without errors.
3. Rerun the public tests and then the private tests.

Make sure to check the autograder's output while submitting!

Grading
The assignment will be based on 60 points:

5 points on programming style and comments
35 points on passing the MVP public tests **(stdout data only; stderr output will not be checked)**
20 points on passing the Field Trial private tests

Note: The coding style is expected to be followed and will be evaluated stricter than PA3.

The Test Harness

This test harness is basically the same as previous PA's. (It has the same hierarchy of files). There are two scripts in the tests directory: `ccwruntests` for testing the C version, and `cwruntests` for testing the assembly version. As supplied there are only two tests, test1 and test2. **You are expected to write your own tests.**