CSE 30 Spring 2022 Programming Assignment #5 - (Vers 1.2)
Due Saturday June 4, 2022 @ 11:59PM

Final day to submit for any credit is Friday June 10, 2022 @11:59PM

Assignment: Stack Frames, Bit Operations, Passing Args in Assembly

In this PA you will write a program called encrypter, a file encryption/decryption program that uses a variation of what is called a book cipher. The program will both encrypt and decrypt a file using a combination of bit manipulation and a text file (traditionally a real book) as the source of cipher keys.

To perform an encryption, encrypter reads a byte from standard input, reorders the bits in the byte, performs an eor (exclusive OR) with a byte read from the cipher file (called a book file), and writes it to the encrypted output file. Encrypter repeats these steps, one byte at a time until EOF is reached on standard input and the entire file has been encrypted.

To decrypt a file, the reverse of the encryption process is performed. Read a byte from the encrypted file, read a byte from the book file, perform an eor on these two bytes, reorder the bits in the result to get back the original encrypted byte and then write the decrypted byte to standard output.

Description of the Command Line Options

```
Arm version: ./encrypter (-d|-e) -b <bookfile> <encryption_file>
C version: ./Cencrypter (-d|-e) -b <bookfile> <encryption_file>
```

Flag	Description
`-d`	Sets the program to decrypt. **<u>Exactly</u> 1 of -d OR -e must be provided, but <u>not</u> both.**
`-e`	Sets the program to encrypt. **<u>Exactly</u> 1 of -d OR -e must be provided, but <u>not</u> both.**
`-b bookfile`	The path to the input bookfile (more info on this file later).
`encryption_file`	When **encrypting**, this is the path to the **output** file (overwrite whatever exists). The input file to be encrypted is read from stdin. (Highly recommend redirecting input from a file using "<"). When **decrypting**, this is the path to the **input** file (contains an encrypted file). The decrypted file is written to stdout. (Highly recommend redirecting output to a file using ">").

Output Examples

```
$ cat MESSAGE
Time in a bottle.
$ ./encrypter -e -b BOOK ENCRYPTED < MESSAGE
$ od -c MESSAGE
0000000   T   i   m   e       i   n       a       b   o   t   t   l   e
0000020  \n
0000021
$ od -c ENCRYPTED
0000000 021 376 263   v   C 362 220   g   x   v   S 204   "   4 346   9
0000020 306
0000021
$ ./encrypter -d -b BOOK ENCRYPTED > DECRYPT_MSG
$ od -c DECRYPT_MSG
0000000   T   i   m   e       i   n       a       b   o   t   t   l   e
0000020  \n
0000021
$ cat SHORTBOOK
AAA
$ ./encrypter -e -b SHORTBOOK ENCRYPTED < MESSAGE
rdbuf: Bookfile is too short for message
$
```

In the last line we see what happens when the BOOKFILE is shorter than the message, the program halts with an error. For the purposes of helping you debug your program, the error message starts with the name of the function generating the message (rdbuf() in this example). You **would not normally do this in real code**, as coding standards for command line programs require that the message would look like this:

```
./encrypter: Bookfile is too short for message
```

How the Program Works

The program is called with a decrypt or encrypt flag, a bookfile flag and a file path for a bookfile, and an encryption file path. Command line option handling is done for you in the supplied function setup() (written in C in the file subs.c)

Inputs:

- Decrypt flag OR encrypt flag
- Bookfile flag and file path of the bookfile

- Encryption file path

Outputs:

- When encrypting, nothing should be printed to `stdout`, only to the `encryption_file`
- When decrypting, the decrypted text should be printed to `stdout`
- If the decrypt flag and encrypt flag are both missing, print usage
- For all other misinputs, error strings are not tested, but should return `EXIT_FAILURE`

Obtaining the Key

If you do not know what a key in cryptography is (specifically, symmetric cryptography), it is essentially similar to how a physical key and a lock works. The key is used to close the lock (or encrypt a message), and the same key is required to unlock the lock (or decrypt the encrypted message). In practice, this almost always uses the exclusive-or operation, or XOR. This is because XOR has the wonderful identity and self-inverse properties, meaning that A ⊕ 0 = A (⊕ is the XOR symbol), and A ⊕ A = 0. Thus, if we have the message M and we XOR it with key K, we can XOR the key again to reobtain M.

Example: M ⊕ K ⊕ K = M ⊕ (K ⊕ K) = M ⊕ (0) = M

The bookfile is simply a file to obtain keys from. This is based on book ciphers, in which the plaintext of a book is used as a key to a cipher. This is more convenient than carrying around specific keys, as books are public and easily accessible. In the starter code, the bookfile is just a plaintext file of The Adventures of Sherlock Holmes by Arthur Conan Doyle.

How the program obtains keys from this bookfile is simply by opening it and reading characters from the file. The first key should be the very first character of the file, which in the case of the starter code is `'T'` (the first line of the file being `"The Adventures of Sherlock Holmes"`). However, the next time we obtain a key, we will increment the location by one byte, meaning that the next key would be `'h'`. To encrypt the file, for each byte from the input a corresponding byte is read from the bookfile to be used at the one-time key (each byte from the bookfile is only used to encrypt/or decrypt one byte from the input file).

Encryption Algorithm

There are two main steps to the encryption algorithm: swapping halves and XORing the setup key. Note that the encryption algorithm is performed on each byte at a time..
This is equivalent to a single byte, or 8 bits.

Step 1: Swapping Upper Half (4-bits) and lower half (4-bits) of a byte

Let's begin with the example of encrypting the letter `'a'`. In ASCII, `'a'` has the decimal value of 97, or hexadecimal value of 61. This means in binary it is the following (6 = 0110, 1 = 0001):

0	1	1	0	0	0	0	1

The first step of the algorithm is to switch (exchange) the first four bits with the last four bits, keeping the order of these bits. After this procedure, the result will be the following:

0	0	0	1	0	1	1	0

This process can be done any number of ways. More notes later in the implementation section. However, one thing to note is that at this point in the process, the encryption would result in the ASCII character of value 22, which is not a printable character. This approach will work with any byte value (not just characters).

Step 2: XORing the Key

Once we have performed the first step, a single bitwise operation needs to be done. Using a key (how to obtain the key is described above), the next step of the algorithm is to exclusive-or (XOR) it with the result from the first step.

For example, let's use the letter `'T'` as the key. In ASCII, `'T'` has the decimal value of 84, or hexadecimal value of 54. This means in binary it is the following (5 = 0101, 4 = 0100):

0	1	0	1	0	1	0	0

XOR 0101 0100 with 0001 0110, and obtain the final result:

0	1	0	0	0	0	1	0

This gives hexadecimal 42, or decimal value 66. The corresponding character in ASCII is `'B'`. Thus the outputted encrypted character would be `'B'`.

You will implement the encryption and decryption program in two separate functions.

Starter Code Overview

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 PA5 in your home directory:

```
    tar -xvf ../public/PA5_new.tar.gz
```

In the starter code you are supplied the source for the entire program for encrypter in C, except for the encrypt and decrypt functions. Your task is to recode the C files into readable, well formatted (no excessive or unnecessary branches), commented assembly code that you write yourself in assembly. No machine generated assembly will be accepted. The C source for the two encryption functions encrypt and decrypt are not provided, but working versions of these two functions are contained in the supplied library file libpa5.a.

The makefile generates two versions of the program: Cencrypter (the C version) and encrypter (the assembly version). The assembly functions encrypt(), and decrypt() are used in both versions. The C functions setup() and cleanup() are used in both versions (**you do not write assembly versions for these two functions**). All other functions have corresponding C and assembly versions that you complete for this PA with the same functionality and calling interfaces (same parameter lists).

File	Function name	Description
Cmain.c	main	The main function written in C. This allocates space for local variables and calls the function setup(), rdbuf(), wrbuf(), encrypt() decrypt() and cleanup()
main.S	main	The main function version is written in assembly based on the C version above. Starter code in assembly is provided. You have to finish the setup of the stack frame design for local variables (matching the C version exactly) and implement the code as shown in the C version in assembly.
subs.c	setup	Written in C (you do not have to rewrite this in assembly). Handles the command line options and opens the files required by each mode (encrypting or decrypting).
subs.c	cleanup	Written in C (you do not have to rewrite this in assembly). Handles the closing of files at the end of the program (called by main() right before it exits).
Crdbuf.c	rdbuf	The C version of the function rdbuf. Rdbuf reads up to a block of bytes (1024) from the input (FILE * FPIN) into the IOBUF with a single call to

		fread(), and then reads exactly the same number of bytes into the BOOKBUF from the book file (FILE *FPBOOK). So if the read (using stdio fread()) returns 100 bytes, this function using fread() then reads exactly 100 bytes from the bookfile.
rdbuf.S	rdbuf	The rdbuf function version is written in assembly based on the C version above. Starter code in assembly is provided. You have to implement the code as shown in the C version in assembly.
Cwrbuf.c	wrbuf	The C version of the function wrbuf. Wrbuf takes the contents of the IOBUF (given the starting address and the number bytes to write from the IOBUF) and writes to the output file (FILE *FPOUT).
wrbuf.S	wrbuf	The wrbuf function version is written in assembly based on the C version above. Starter code in assembly is provided. You have to implement the code as shown in the C version in assembly.
Cerrmsg.c	errmsg	The C version of the function errmsg. Errmsg writes a string pointer passed to it to stderr.
errmsg.S	errmsg	The errmsg function version is written in assembly based on the C version above. Starter code in assembly is provided. You have to implement the code as shown in the C version in assembly.
decrypt.S	decrypt	The decrypt function is written in assembly by you and is used in both the C and assembly versions of the program. Be aware that if you do not uncomment the #define MYCODE at the top of the file, it will use the version in the library file libpa5.a. Decrypt is passed two buffers that contain the same number of bytes. It decrypts the bytes in IOBUF one byte at a time until all the bytes are processed. Each byte is processed first by performing an eor (bitwise exclusive or) with the corresponding byte (same index offset) in the second buffer BOOKBUF. Then it exchanges the upper and lower 4-bits in the byte. FInally It stores the result of the bit exchange back into IOBUF (overwriting the original encrypted version of the byte.
encrypt.S	encrypt	The encrypt function is written in assembly by you and is used in both the C and assembly versions of the program. Be aware that if you do not uncomment the #define MYCODE at the top of the file, it will use the version in the library file libpa5.a. Encrypt is passed two buffers that contain the same number of bytes. It encrypts the bytes in IOBUF one byte at a time until all the bytes are processed. Each byte is processed first by exchanging the upper and lower 4-bits. The exchanged byte in IOBUF is then eor (bitwise exclusive or) with the corresponding byte (same index offset) in the second buffer BOOKBUF. Finally it stores the result of the eor back into IOBUF, overwriting the original unencrypted version of the byte.

libpa5.a	Encrypt & decrypt	Contains working versions of the functions: encrypt and decrypt, for you to use until you get your assembly language versions written. These versions are included independently when you link your .o files together (make does this for you) with the library and you have not uncommented the #define MYCODE at the top of encrypt.S and decrypt.S
encrypter.h	Function prototypes and common definitions	Used by all files, assembly or C to maintain consistency on several return values used by the functions as well as containing the function prototypes for all the functions in a format used by C.

Using the Makefile

There are several targets in the Makefile that you may find useful to use.
These are called with % make target (%make A for example)

Target	What it does
A	Creates the assembly language version of the program encrypter. This will likely not be functional until you write the assembly language versions of the C source files
encrypter	Same as A
C	Creates the C version of the program Cencrypter (using the C source files provided). This will work right away without you writing any code.
Cencrypter	Same as C
Atest	Runs the public tests on the assembly language version of the program encrypter
Ctest	Runs the public tests on the C source language version of the program Cencrypter
test	Same as make Atest
clean	Removes all the object files and executables for both the C and assembly language versions. Also calls make clean in the test directory, removing all test output files from the tests/out directory
	WIth no arguments to make, it runs make A

Running the Tests - What Passing the tests Looks like

```
$ make test
(cd ./tests; make Atest)
make[1]: Entering directory '/home/kmuller/PA5/tests'
./Aruntests
##########
Assembly version Running Public Tests
----- Starting test sequence 1 -----
Running encryption: ../encrypter -e -b BOOK out/ENout1 <in/ENtest1 2> out/ENerr1
***** Encryption test 1 passed *****
Running decryption: ../encrypter -d -b BOOK in/DEtest1 >out/DEout1 2> out/DEerr1
***** Decryption test 1 passed *****
----- Ending   test sequence 1 -----
----- Starting test sequence 2 -----
Running encryption: ../encrypter -e -b BOOK out/ENout2 <in/ENtest2 2> out/ENerr2
***** Encryption test 2 passed *****
Running decryption: ../encrypter -d -b BOOK in/DEtest2 >out/DEout2 2> out/DEerr2
***** Decryption test 2 passed *****
----- Ending   test sequence 2 -----
----- Starting test short book -----
Running: ../encrypter -e -b SHORTBOOK out/shortbook <in/ENtest2 2> out/errshortbook
***** Short book test  passed *****
----- Ending   test short book -----
----- Starting test binary     -----
Running encryption: ../Cencrypter -e -b BOOK out/ENbinary <in/ENbinary 2> out/ENerrbinary
***** Binary encrypt   passed *****
Running decryption: ../Cencrypter -d -b BOOK in/DEbinary >out/DEbinary 2> out/DEerrbinary
***** Binary decrypt   passed *****
----- Ending   test binary     -----
Assembly version Tests Complete
##########
make[1]: Leaving directory '/home/kmuller/PA5/tests'
```

```
$ make Ctest
(cd ./tests; make Ctest)
make[1]: Entering directory '/home/kmuller/PA5/tests'
./Cruntests
##########
C version Running Public Tests
----- Starting test sequence 1 -----
Running encryption: ../Cencrypter -e -b BOOK out/ENout1 <in/ENtest1 2> out/ENerr1
***** Encryption test 1 passed *****
Running decryption: ../Cencrypter -d -b BOOK in/DEtest1 >out/DEout1 2> out/DEerr1
***** Decryption test 1 passed *****
----- Ending   test sequence 1 -----
----- Starting test sequence 2 -----
Running encryption: ../Cencrypter -e -b BOOK out/ENout2 <in/ENtest2 2> out/ENerr2
***** Encryption test 2 passed *****
Running decryption: ../Cencrypter -d -b BOOK in/DEtest2 >out/DEout2 2> out/DEerr2
***** Decryption test 2 passed *****
----- Ending   test sequence 2 -----
----- Starting test short book -----
Running: ../Cencrypter -e -b SHORTBOOK out/shortbook <in/ENtest2 2> out/errshortbook
***** Short book test  passed *****
----- Ending   test short book -----
----- Starting test binary     -----
Running encryption: ../Cencrypter -e -b BOOK out/ENbinary <in/ENbinary 2> out/ENerrbinary
***** Binary encrypt   passed *****
Running decryption: ../Cencrypter -d -b BOOK in/DEbinary >out/DEbinary 2> out/DEerrbinary
***** Binary decrypt   passed *****
----- Ending   test binary     -----
C version Tests Complete
##########
make[1]: Leaving directory '/home/kmuller/PA5/tests'
```

Coding and Program Development Requirements

1. You will write your code only in ARM V6 assembly. It must run in a Linux environment on the pi-cluster.
2. 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.
3. You may **only use the ARM instructions** listed in the ARM ISA green card. The green card can be found on Canvas under Documents.

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

How to Approach This Programming Assignment

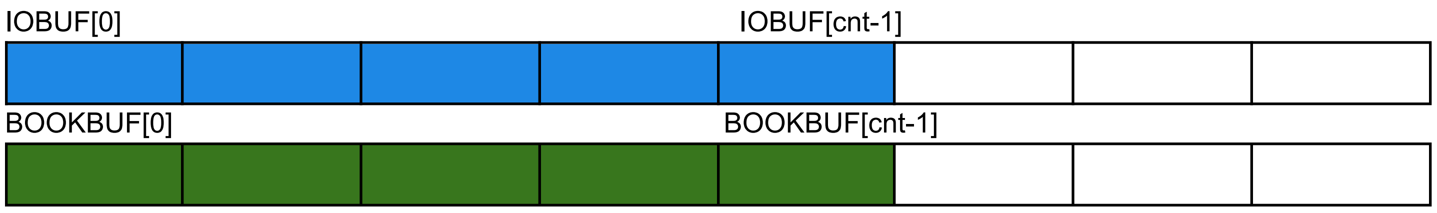
Read over the supplied C versions and focus on how the parameters are passed between the various functions. Take notice of which arguments are output parameters (the function is passed a pointer to a memory function as it modifies it) and what values you currently have in registers r0-r3 at the point you are making the function call.

When designing your assembly functions, decide which registers you are going to use for each local variable first. If the function you are working on does not call any function that has output parameters, you can put all the variables in registers (combination of preserved and scratch registers). Try to use the scratch registers first, remembering that function calls can change the values in r0-r3. For example in main.S you see a table of suggested uses for the preserved registers. You do not have to follow these suggestions, but this is how the solution code used the preserved registers. All of the starter files for the assembler functions you need to write have suggested push and pop lists for preserved registers. This is what the solution code used and should be sufficient for the code you write.

Carefully look at how return values are used from the functions in the C code. This is important in controlling the loops and getting the return value from main is correct.

Each .S file has the #define MYCODE commented out at the top. When commented out, when you run make to create encrypter, it will use the solution version of the function from libpa5.a. When you want to test your version, remove the // from in front of the #define MYCODE in the file you want to test. Comment of #define MYCODE to get the solution version.

int encrypt(char *IOBUF, char *BOOKBUF, int cnt)
encrypt() is passed pointers to two buffers, the IOBUF which contains the bytes to be encrypted and BOOKBUF the cipher key buffer. Each buffer contains the exact same number of bytes cnt) in them. You need to write a loop that processes each byte in the buffers until all cnt bytes have been processed. Starting with byte 0 in IOBUF, you need to write the assembly language instructions to load it into a register and then exchange the upper and lower 4-bits in the byte. Next you load byte 0 in BOOKBUF into a register and eor the two registers. Finally you store the result back into byte 0 of IOBUF. Repeat this for each byte in the two buffers. Return the value of cnt.



int decrypt(char *IOBUF, char *BOOKBUF, int cnt)
decrypt() is passed pointers to two buffers, the IOBUF which contains the bytes to be decrypted and BOOKBUF the cipher key buffer. Each buffer contains the exact same number of bytes cnt) in them. You need to write a loop that processes each byte in the buffers until all cnt bytes have been processed. Starting with byte 0, you load a byte from each buffer into registers. Next you eor the two registers together. You then exchange the upper and lower 4-bits of the byte.. FInally you store the result back into byte 0 of IOBUF. Repeat this for each byte in the two buffers. Return the value of cnt.
Encrypt and decrypt should require less than 60 lines of assembly.

Int main(int argc, char **argv)
Main needs to have the layout of it's stack frame finished. Used the process that we did in lecture to determine the offset locations in the stack and how much space is needed for local variables and the stack frame parameters needed for setup() and rdbuf(). The call to setup is tricky as not only does it have six parameters, the last four arguments 3-6 inclusive are output parameters, so you are passing the address of local variables in main's stack frame to setup to modify. Be aware that the first four are passed on registers and the last two are passed on the stack. You may want to look over the function setup(), it is located in the file subs.c.

Main repeatedly calls rdbuf() to fill the IOBUF and BOOKBUF until the EOF is reached on the input file. If the BOOKBUF is too short (it hits EOF before the input file, the program exits. It then processes the buffers by either encrypting or decrypting them and call wrbuf to write out the processed buffer. At the end of the program, it calls cleanup (passing it the exit status) which closes off the files that setup() opens. Main returns EXIT_SUCCESS if the data was processed without an error or EXIT_FAILURE otherwise.
Main should be under 150 lines of assembly.

Int setup(int argc, char *argv[], int *mode, FILE **book, FILE **input, FILE **output)
This function is supplied in source form and you do not have to write an assembly language version of it. You do need to call it from your assembly language version of main. It is important to note that the first two arguments are argc and argv, and are passed unchanged from main. The remainder of the arguments are all output parameters pointing at memory location located in main's stack frame. Also be aware that there are six (6) arguments to this function, so two of them are passed on the stack.

Comment and check your code follows 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.

Submit to Gradescope under the assignment titled "A5 encrypter"

After you pass all the public MVP tests in the test harness and only then, will you submit to Gradescope the following files you have written yourself in assembly. THere are no other files that need to be turned in:

1. decrypt.S
2. encrypt.S
3. errmsg.S
4. main.S
5. rdbuf.S
6. wrbuf.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:

35 points on passing the MVP public tests
5 points for coding style and comments
20 points on passing the Field Trial private tests

We strongly encourage you to turn in your PA by 11:59 PM on Saturday June 4.

Submissions made after Tuesday June 7 at 11:59 PM will not receive credit for style and comments (so the maximum points possible will be reduced by 5 points after this deadline).

Submissions made after Tuesday June 7 11:59 PM will lose 10 points per day.

BE AWARE that normal tutor and TA help will end on Friday June 3. Plan on **extremely** limited help on Saturday June 4 and no TA/tutor help after June 4. Edstem support for PA's ends after Saturday June 4.

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: `Cruntests` for testing the C version, and `Aruntests` for testing the assembly version. **You are expected to write your own tests.**