COMPSCI 210 Assignment 2

Due date: 23:59 2nd June 2024
Total marks: 60

This assignment aims to give you some experience on writing simple C programs.

Important Notes

- There are subtle differences between various C compilers. We will use the GNU compiler gcc on the Linux sever at FlexIT for marking. Therefore, you MUST ensure that your submissions can be compiled and run on the FlexIT server. Submissions that fail to compile or run on the server will attract NO marks.
- Markers will use files and password that are different from the examples given in the specifications when testing your programs.
- The files containing the examples can be downloaded from Canvas and unpacked on server with the command below:
    - `tar xvf A2Examples.tar.gz`
- As we need to return the assignment marks before the exam of this course, there is NO possibility to extend the deadline for this assignment.

Academic Honesty

Do NOT copy other people's code (this includes the code that you find on the Internet).

We will use Stanford's MOSS tool to check all submissions. The tool is very "smart". Changing the names of the variables and shuffling the statements around will not fool the tool. In previous years, quite a few students had been caught by the tool; and, they were dealt with according to the university's rules at https://www.auckland.ac.nz/en/about/learning-and-teaching/policies-guidelines-and-procedures/academic-integrity-info-for-students.html

Part 1 (20 marks)

The example program crypt.c discussed in lecture hardcoded the password in the program. The first part of this assignment requires you to rewrite the program to allow the password to be provided as one of the command line arguments. The detailed requirements are:

- Name the C program as "part1.c".
- When the program is run, apart from the name of the program, the program should take two command line arguments, i.e. the name of the file to be encrypted/decrypted using the XOR operation and the password for encrypting/decrypting the file.

For example, if the name of the file to be encrypted/decrypted is “auckland.jpg” and the password is “abcd1234”, the command below is used to run the program (“$” is the command window prompt):

```
$ ./part1 auckland.jpg abcd1234
```

- The result of the encryption/decryption should be stored in a file. The name of the file is the name of the original file with a “new-“ prefix. For example, if the name of the original file is “auckland.jpg”, the name of the file containing the result of the encryption/decryption operation should be “new-auckland.jpg”.
- It should be assumed that the name of the file to be encrypted/decrypted consists of at most 15 characters.
- In this part, the program does not need to implement the encryption/decryption operation. When the program is run, it just prints out the name of the new file and the length of the password.
- An example of the command for executing the program and the corresponding outputs of the program is given below. The outputs of the program are highlighted in red. (NOTE: “auckland.jpg” is the exact name of the file.).

```
$ ./part1 auckland.jpg abcd1234
newFileName = new-auckland.jpg
password length = 8
```

- **Note**: Markers will probably use a file with a different name and a different password.

Part 2 (20 marks)

This part is based on Part 1.

- Name this program as part2.c
- Expand the functionality of the program in part 1 to check the validity of the password. A password is valid if it consists of at least 8 characters and at least one digit (a digit is also counted as one character).
- You can assume that the marker will only enter letters and digits for the password and the length of the password is at most 15.
- It is possible that a password consists of digits only.
- In this part, the program does not need to implement the encryption/decryption operation.
- When the program is run, if the password is valid, the output of the program is the same as in part1. If the password is invalid, the output of the program should also include the information indicating the problem(s) with the password.
- Here are some examples of running the program. The outputs of the program are highlighted in red.

  Example 1 (the password is valid):

```
$ ./part2 auckland.jpg abcd1234
newFileName = new-auckland.jpg
password length = 8
```

Example 2 (the password is invalid):

```
$ ./part2 auckland.jpg abcd123
newFileName = new-auckland.jpg
password length = 7
The password needs to have at least 8 characters.
```

Example 3 (the password is invalid):

```
$ ./part2 auckland.jpg abcdefgh
newFileName = new-auckland.jpg
password length = 8
The password needs to contain at least one digit.
```

Example 4 (the password is invalid):

```
$ ./part2 auckland.jpg abcdefg
newFileName = new-auckland.jpg
password length = 7
The password needs to have at least 8 characters.
The password needs to contain at least one digit.
```

- **Note**: Markers will probably use a file with a different name and a different password.

Part 3 (10 marks)

This part is based on Part 2.

- Name this program as part3.c
- Expand the functionality of the program in part 2 to encrypt/decrypt the file using the same method as the program crypt.c discussed in the lecture. The details are as below:
    - You need to divide the file into blocks. The size of each block (except the last block) must equal to the length of the password.
    - Apply the XOR operation to each block and the password as what program crypt.c does.
    - Save the results of the XOR operations in a file. The name of the file should be the name of the original file with a "new-" prefix. For example, if the name of the original file is "auckland.jpg", the name of the file containing the results of the XOR operations should be "new-auckland.jpg".
    - The outputs of part3.c are stored in a file. It does not display any outputs in the command window. For example, if the command below is used to run the program,

      ```
      $ ./part3 auckland.jpg abcd1234
      ```

The outputs of the program are stored in file “new-auckland.jpg”.

- A couple of sample files, “sample1” and “sample2”, are given for you to check whether your file has been encrypted correctly. The commands for checking the correctness of your implementation are as below:
  Check 1:
  ```
  $ ./part3 auckland.jpg abcde1234
  $ cmp new-auckland.jpg sample1
  ```

  If the execution of command “cmp” does not produce any output, it means the two files, new-auckland.jpg and sample1, are identical. This means you have implemented the encryption/decryption correctly. If the execution of command “cmp” shows message like “new-auckland.jpg sample1 differ: char 1, line 1” or something similar, it means your program has NOT implemented the encryption/decryption correctly.

  Check 2:
  ```
  $ ./part3 auckland.jpg abcde12345
  $ cmp new-auckland.jpg sample2
  ```

  If the execution of command “cmp” does not produce any output, it means the two files are identical. This means you have implemented the encryption/decryption correctly. If the execution of command “cmp” shows message like “new-auckland.jpg sample2 differ: char 10, line 1” or something similar, it means your program has NOT implemented the encryption/decryption correctly.
- **Note**: Markers will probably use a file with a different name and a different password.

Part 4 (10 marks)

This part is based on Part 3.

- Name this program as part4.c
- Expand the functionality of the program in part 3 to print out the values of the first five bytes in the file that stores the results of the XOR operations.
- The value of each byte must be shown as a 2-digit hexadecimal number.
- In the output, each line shows the value of one byte.
- The letter digits “a” to “f” must be shown as lowercase letters.
- Here are some examples of executing the program. The outputs are highlighted in red.
  Example 1:
  ```
  $ ./part4 auckland.jpg abcd1234
  9e
  ba
  9c
  ```

```
84
31
```

Example 2:

```
$ ./part4 auckland.jpg xyz0123456
87
a1
85
d0
31
```

- **Note**: Markers will probably use a file with a different name and a different password.

Submission

1. You **MUST** thoroughly test your program on the server at FlexIT before submission. Programs that cannot be compiled or run on the server will **NOT** get any mark.
2. Use command "tar cvzf A2.tar.gz part1.c part2.c part3.c part4.c" to pack the **SOURCE** code of your completed C programs to file A2.tar.gz. [Note: You MUST use the tar command on the server to pack the files as files packed using tools on PC cannot be unpacked on the server. You will NOT get any mark if your file cannot be unpacked on the server.]
3. Submit A2.tar.gz through Canvas. The markers will only mark your latest submission.
4. **NO** email submission will be accepted.

Resource

- The three files, auckland.jpg, sample1 and sample2, used in the examples of the specifications are packed in file A2Examples.tar.gz that can be download from Canvas.
- Follow the steps below to extract the files in A2Examples.tar.gz
    - Put A2Examples.tar.gz in the directory in which your programs are stored.
    - Use command "tar xvf A2Examples.tar.gz" to unpack A2Examples.tar.gz.

Debugging Tips

1. Debugging is a skill that you are expected to acquire. Once you start working, you are paid to write and debug programs. Nobody is going to help you with debugging. So, you should acquire the skill now. **You can only acquire it by practicing.**
2. If you get a "segmentation faults" while running a program, the best way to locate the statement that causes the bug is to insert "printf" into your program.
3. If you can see the output of the "printf" statement, it means the bug is caused by a statement that appears somewhere after the "printf" statement. In this case, you should move the "printf" statement forward. Repeat this process until you cannot see the output of the "printf" statement.
4. If you cannot see the output of the "printf" statement, it means the bug is caused by a statement that appears somewhere before the "printf" statement.
5. Combining step 3 and 4, you should be able to identify the statement that causes the "segmentation faults".
6. Once you identify the statement that causes the "segmentation faults", you can analyse the cause of bug, e.g. whether the variables have the expected values.

Some Hints

You are strongly encouraged to attempt each part first without reading these hints.
There are many ways to write the programs that satisfy the specifications. The hints given here are just some of the possible implementations. You can implement your programs differently as long as the outputs satisfy the specifications.
These are hints rather than a detailed step by step descriptions on how to implement the assignment. So, you still need to bridge the gaps that are not explicitly stated.

Part 1

1. How to obtain the name of the original file?
   The name of the original file is given as a command line argument. Read program argv1.c discussed in the lecture on how to process command line argument.
2. How to generate the name of the new file?
   The name of the new file can be obtained by concatenating "new-" and the name of the original file. C has a function "strcat" for concatenating two strings. To using the function, you need to include the "<string.h>" header file in your program. The details of the function can be found at https://www.freebsd.org/cgi/man.cgi?query=strcat&apropos=0&sektion=0&manpath=FreeBSD+12.0-stable&arch=default&format=html
3. How to find out the length of the password?
   Password is given as a command line argument. When the operating system stores a command line argument in the memory, it stores a null character '\0' at the end of the argument. The value of this null character is 0. It is used to signify the end of a command line argument. You just need to count the number of characters from the beginning to the end of the command line argument that corresponds to the password.
   Example argv2.c discussed in the lecture shows how to go through each character in a command line argument. If you understand that example, you should know how to find out the length of the password.

Part 2

4. How to find out whether there are digits in the password?
   The password is a sequence of characters. You can use the ASCII code of each character to determine whether the character is a digit character. The ASCII code of a digit character is between 0x30 and 0x39.

Part 3

5. The implementation is the same as the example program crypt.c. You just need to make sure that the size of each block (except the last block) is the same as the length of the password.

Part 4

6. How to print the value of a byte as a 2-digit hexadecimal number?
   To print out the value of a byte as a 2-digit hexadecimal number, you can use “printf("%02x”, byte)” where “byte” is a variable of “unsigned char” type (the size of a “unsigned char” type variable is one byte).
7. Example program crypt.c shows how to read bytes from a file.