PA-2: C Programming on Linux

Total points: **100 pts** Due on **Friday Feb 21, 2020 at 11:59 PM**

Instructor: Endadul Hoque

Logistics

1. This assignment must be done **individually, not in groups**.
2. For this programming assignment (PA-2), you have to download an archive called `student-pa2.tar.gz` from Blackboard. Upon extraction, you will see a directory named `student/`, which contains 5 directories named `task-1`, `task-2`, and, so on – one for each task.
3. For each task, you have to complete a `C` program provided in the respective directory. For instance, for **Task-1**, you have to complete `solution-1.c` located in `task-1` directory.
4. Take a look at the accompanied **`README.md`** in each task directory for additional instructions, such as ***how to build/run/test your program***.
5. Unless stated otherwise, you **MUST NOT** modify/edit/rename/move other existing files under the `student/` directory.
6. For all the following tasks, you have to use the Linux server (`lcs-vc-cis486.syr.edu`) dedicated for this course. You **must** know how to remotely login to that machine, how to use `terminal` and how to copy files back and forth between your computer and the Linux server.
7. Some helpful links are listed in Appendix A. You may want to leverage them for solving PA-2.

Obtain the source code for this PA

Obtain `student-pa2.tar.gz` from `blackboard`. Unpack the package to find the skeleton code required for this PA. The extracted directory is named `student/`.

```
$ tar xzf student-pa2.tar.gz
```

Submission Instructions

1. Go to **each task** directory and clean the directory using `make clean`. For example,

   ```
   $ cd task-1/
   $ make clean
   ```

2. Change directory (`cd`) to your `student/` directory. Now `student/` should be your current working directory.
3. **Use your netid** to create a compressed archive (`<netid>-pa2.tar.gz`) of your solution directory. For instance, if your netid is `johndoe`, create a compressed archive as follows:

```
$ cd ..       # moving to the parent directory of student/
$ mv student/ johndoe-pa2/
$ tar czf johndoe-pa2.tar.gz johndoe-pa2/
```

4. Use **Blackboard** to submit your compressed archive file (*e.g.*, `johndoe-pa2.tar.gz`).

Grading

We will use an automated grading tool with pre-determined inputs to check your solution for correctness. ***Please make sure you follow the instructions provided throughout this handout; otherwise your assignment will not be graded at all by the auto grader.***

- You must not rename/move `solution-*.c` files under each `task-*/` directory.
- You must not modify/edit/rename/move other existing files/directories in `student/` or in its sub-directories.
- You must follow the naming convention for each task as directed, while creating files, directories, archives, and so on.
- Your solutions must execute on the Linux machine (`lcs-vc-cis486.syr.edu`), otherwise your solutions will not be considered for grading.
- Your must follow the submission instructions, or it will not be graded properly by the auto grader.

Task 1. File operations (20 pts)

You need to write a C program (location: `task-1/solution-1.c`) that will take a single command line argument. The command line argument will be the name of a file. The file will contain a sequence of Unix/Linux commands for file manipulations: one command in each line. Your program needs to execute each of these commands. For this task, you must not use `system()`; instead, you have to use `fork()`, `execvp()`, `wait()`/`waitpid()` functions.

For each command, your program will spawn a process using `fork()` which will carry out the given file operation using `exec()` (or one of its variants). You can use `execvp()`. See this link ([1]) for `exec()` man-page and this link ([2]) for a pictorial description of its family.

Assumptions: You can safely assume that each line will have a maximum of 1024 characters including the newline character at the end of the line. The file contains only two types of commands: (1) `cp <source-file-path> <destination-path>`, and (2) `mv <source-file-path> <destination-path>`.

Error checking: Your program should do appropriate error checking. In case of an error, your program must print the following error message: "`An error has occurred\n`" and then exit with the error code 1, *e.g.*, using `exit(1)`.

How to build/run/test your program?: See `task-1/README.md` file.

Task 2. Hexadecimal to Binary (10 pts)

You will write a C program (location: `task-2/solution-2.c`) that will take **two command line arguments**. The first command line argument will be the name of the **input file** and the second command line argument will be the name of the **output file**.

Input: The input file will contain a sequence of hexadecimal numbers: one in each line followed by a terminating newline character (*i.e.*, `'\n'`). Each hexadecimal number will have digits from the following set: $D = \{0, 1, 2, \ldots, 9, a, b, c, d, e, f\}$. Each input line will have the digits of an hexadecimal number printed in ASCII format. For instance, digit 0 will be character `'0'`, digit 1 will be `'1'`, and similarly, digit a will be character `'a'` and so on. Note that the input hexadecimal number can have leading zeros.

Assumptions: You can safely assume that each input line will have a maximum of 1024 characters including the newline character at the end of the line.

Output: For each hexadecimal number in the input file, you should print a line in the output file. The line should contain the corresponding binary representation of the hexadecimal number followed by a newline character (*i.e.*, `'\n'`). The binary number should be printed using ASCII characters: `'0'`s and `'1'`s. You should not remove the leading zeros from the binary representation of the hexadecimal number while printing to the output file.

Error checking: Your program should do appropriate error checking. In case of an error, your program must print the following error message: "`An error has occurred\n`" and then exit with the error code 1, *e.g.*, using `exit(1)`.

How to build/run/test your program?: See `task-2/README.md` file.

Task 3. Binary to Hexadecimal (10 pts)

You will write a C program (location: `task-3/solution-3.c`) that will take **two command line arguments**. The first command line argument will be the name of the **input file** and the second command line argument will be the name of the **output file**.

Input: The input file will contain a sequence of binary numbers: one in each line. Each binary number is a string of ASCII characters `'0'`s and `'1'`s, followed by a newline character (*i.e.*, `\n`). Note that the input binary number can have leading zeros.

Assumptions: You can safely assume that (a) the number of digits in a binary number will be a multiple of 4; and (b) each input line will have a maximum of 1024 characters including the newline character at the end of the line.

Output: For each binary number in the input file, you should print a line in the output file. The line should contain the corresponding hexadecimal representation of that number (in ASCII format) followed by a newline character (i.e., `\n`). Each hexadecimal number should be represented with digits from the following set: $D = \{0, 1, 2, \ldots, 9, a, b, c, d, e, f\}$. You should not remove the leading zeros from the hexadecimal representation of the input binary number while printing to the output file.

Error checking: Your program should do appropriate error checking. In case of an error, your program must print the following error message: "`An error has occurred\n`" and then exit with the error code 1, *e.g.*, using `exit(1)`.

How to build/run/test your program?: See `task-3/README.md` file.

Task 4. Command parsing (20 pts)

You will write an interactive C program (location: `task-4/solution-4.c`) which will continuously wait for a user to type in *command strings* on the standard input (`stdin`). For each command string, your program should print out the tokens in the given string on the standard output (*i.e.*, `stdout`): one token in each line.

Input: A command string can contain any printable ASCII characters except the newline character (*i.e.*, `'\n'`). The newline character is used as the terminator of a command string.

Output: For each command string, your program should print the command followed by its arguments (in the same order as they appear on the input) in separate lines on the standard output. Your program should terminate without printing anything on the standard output whenever it observes the `exit` command after parsing.

Assumptions: You can safely assume that each input line will have a maximum of 1024 characters including the newline character at the end of the line.

> **Hint:** Once a command string has been entered by the user – determined by observing a newline character – your program should use `space` and `tab` characters as delimiters to tokenize the command string. Once a command string has been tokenized, the first token is considered to be the "command" whereas the rest of the following tokens are considered to be "arguments to the command".
>
> You may want to utilize `getline()`/`fgets()` and `strtok()` library functions for this task.

Error checking: Your program should do appropriate error checking. In case of an error, your program must print the following error message: "`An error has occurred\n`" and then exit with the error code 1, *e.g.*, using `exit(1)`.

How to build/run/test your program?: See `task-4/README.md` file.

Task 5. Sorting binary objects (40 pts)

You will write a simple C program (location: `task-5/solution-5.c`) which sorts given binary objects. This program will be compiled and invoked in the following way:

```
$ gcc -Werror -Wall -o fastsort solution-5.c      # Compilation
$ ./fastsort inputfile outputfile                 # Execution
```

The built executable (*i.e.*, `fastsort`) takes two command line arguments: an *input file* (named `inputfile`) containing the binary objects to be sorted and an *output file* (named `outputfile`) to store the sorted results.

Input: The `inputfile` contains binary data[1] that needs to be sorted. The binary data consists of a series of **100-byte binary objects** (aka blobs). Each object is a key-value pair, where the first four bytes represent an unsigned integer **key** and the remaining 96 bytes is the corresponding value. Each value (*i.e.*, the remaining 96 bytes) actually represents a **record** consisting of 24 unsigned integers. Therefore, each binary object looks something as follows (where each letter represents 4 bytes):

kRRRRRRRRRRRRRRRRRRRRRRRR

[1]The binary data may not be viewed properly using a traditional IDE/text editor. Instead, you can use `xxd`, `hexdump` or the provided `dump.c` to view/display the content of the file.

How to generate such input files?: Input files are generated by a given program called `task-5/generate.c`, which can be complied as follows:

```
$ gcc -Wall -Werror -o generate generate.c
$ ./generate -h
usage: ./generate [-s random_seed] [-n number_of_objects] [-o generated_filename]
```

The `generate` program takes three optional arguments:

- `-s`: This option specifies a `random_seed` that this program will use as the seed for a new sequence of (pseudo-)random integers. This allows you to generate input files with different data while testing your `fastsort`.
- `-n`: This option specifies how many binary objects to write to the `generated_filename` file. Each object is of 100 bytes.
- `-o`: This options specifies the filename where the `number_of_objects` binary objects will be stored.

A sample execution of `generate` is shown below. **Note that the generated file (`test.in`) can be used as the input file to your `fastsort`.**

```
$ ./generate -s 0 -n 100 -o test.in
```

Output: Your sorting program (named `fastsort`) must take in one of these generated files and sorts it based on the 4-byte key (the remainder of the object (*i.e.*, the record) should be kept with the same key). The output is written to the specified output file. While writing the output, you should only write the sorted binary objects, no additional characters (*e.g.*, `'\n'`).

> **NOTE:** All these programs related to this task need the header file `sort.h` during their compilation. This header is located at `task-5/sort.h`.

Assumptions:

- *32-bit integer*: You may assume all the integers are 32-bit. In addition, the keys are unsigned 32-bit integers.
- *File length*: May be pretty long! However, there is no need to implement a fancy two-pass sort or anything like that; the data from a file will fit into memory.

Error checking: Your program should do appropriate error checking.

- *Invalid files:* If the user specifies an input or output file that your program cannot open (for whatever reason), your program should EXACTLY print: `"Error: Cannot open file %s\n"`, where `%s` refers to the filename and then exit with `exit(1)`.
- *Too few or many arguments passed to program*: If the user runs your program without any arguments, or with few/many arguments, or in some other way passes incorrect options, then you must print the appropriate `usage` string of `fastsort` and exit with `exit(1)`.
- For all other errors, your program must print the following error message: "`An error has occurred\n`" and then exit with the error code 1, *e.g.*, using `exit(1)`.

How to build/run/test your program?: See `task-5/README.md` file.

How to display/view such files?: You may use `xxd` or `hexdump` available on the server machine. However, `task-5/` directory contains a useful program named `dump.c`. This program can be used to dump the contents of a file generated by `generate` or by your `fastsort`.

```
$ gcc -Wall -Werror -o dump dump.c
$ ./dump
usage:  ./dump binary_filename
```

Hints

Studying the source code in `dump.c` and `generate.c` can make your life extremely easy for this task. You can find how to read/write the binary data from/to a file.

Reading inputs and writing outputs: In your program, you should use `open()`, `read()`, `write()`, and `close()` to access files. See the code in `generate.c` or `dump.c` for examples.

Dynamic memory allocation: If you want to figure out how big an input file is before reading it in, use the `stat()` or `fstat()` functions. The function `malloc()` is useful for memory allocation. Make sure to exit gracefully if `malloc()` fails!

Sorting: To sort the data, use any sort algorithm that you'd like to use. An easy way to go is to use the library routine `qsort()` in `stdlib.h`.

Exiting: To exit, call `exit()` with a single argument. This argument to `exit()` is then available to the user to see if the program returned an error (*i.e.*, return 1 by calling `exit(1)`) or exited without any error (*i.e.*, return 0 by calling `exit(0)`).

If you dont know how to use these functions, use the `man` pages. For example, typing `man malloc` will provide you information about how to use the malloc function.

Appendix A: Some Useful Links

1. `exec()` man page – `http://man7.org/linux/man-pages/man3/exec.3.html`
2. `exec.md` – `https://gist.github.com/fffaraz/8a250f896a2297db06c4`
3. C Preprocessor – `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`