FIT2100 Assignment #1
Building a file utility with C Programming
Semester 2 2020

Mr Muhammed Tawfiqul Islam
Lecturer, Faculty of IT.
Email: `tawfiq.islam@monash.edu`

August 21, 2020

Revision Status

```
version 1: August 2020 by Muhammed Tawfiqul Islam
```

Contents

1 Introduction

In this assignment, you will build a multipurpose file utility, which combines the **simplified** features of multiple Linux utilities such as `cat, cp and mv`.

This document constitutes the requirement specification for this assignment. You will be assessed on your ability to both comprehend and comply with the requirements as specified herein.

Due: 11th September 2020 (Friday) 5pm.

Late submissions: A late submission penalty of 5% of the total available marks per day will apply.

This assignment is worth 15% of the total marks for this unit.

2 Caveats

To complete these tasks, you are allowed to use any of the standard C library functions found on your virtual machine environment[1]. Except the following:

You are *NOT* allowed to use any functions available in `<stdio.h>`. This means you cannot use `printf()` to produce output. (For example: to print output in the terminal, you will need to write to *standard output* directly, using appropriate file system calls.) In addition, you are *NOT* allowed to use the `system()` library function, and any library functions which spawns new process(s) from your program. This makes the assignment more challenging, but also means you are interacting with services of the operating system directly.

Your main C source file should be named with your student ID as: `fileutil-123456789.c`, where 123456789 is your student ID.[2]

[1]For example, you might choose to use `getopt` or other advanced libraries according to your preference, but use of any particular library is neither expected nor required, except as required to make system calls.

[2]If your completed program contains multiple source files, you may name other source and header files as you wish.

3 If you require extra help

This assignment is an independent learning and assessment exercise.

You may utilise the Ed Discussion Forum to ask questions and obtain clarification, however you may not share details or code in your implementation with other students, nor may you show your code to the teaching team prior to submission. This is an assessment task: tutors and lecturers are not permitted to help you debug your assignment code *directly* (you are expected to debug and test your own code), but can help with more general queries, such as queries related to C programming syntax, concepts and debugging tips.

You may make use of online references with appropriate citation in accordance with academic integrity policies, however your work must be your own.

3.1 References

The following resources are available on the FIT2100 Unit Information page.

- Curry, David, *UNIX Systems Programming for SVR4*, Chapter 3: 'Low-Level I/O Routines'.
- Curry, David, *Using C on the UNIX System*, Chapter 3: 'Low-Level I/O'.
- He, Jialong, *LINUX System Call Quick Reference*.

4 Task 1: File content viewing functionality with hard-coded options

Write a utility called `fileutil` ('File Utility') which does the following:

1. Opens a file named logfile.txt in the current working directory.
2. Outputs the entire file contents (to *standard output*—usually the terminal). Note that, when the program completes normally, no other output should be produced.

Your program should always exit 'cleanly.' This means closing any open files and freeing any other resources you may have allocated before termination.

If there is a problem accessing the file (e.g. file does not exist), your program should display an appropriate and sensible error message (you should output this to the program's `standard error` stream rather than `standard output`) and exit cleanly with a return value of 1. On successful completion, your program should return an error code of 0.

Write up an instruction manual (user documentation) in a plain text file, explaining how to compile your program and how to use it.

You may use your user documentation itself or any other text file to test your program.

5 Support Command-line Arguments

Now extend the functionalities of the same program (from Task 1) to allow the user to run it with command-line arguments as follows:

5.1 Task 2: Functionality to support the use of a different source file

Allow the user to specify a different `filename` as a source (instead of `logfile.txt`) by putting the filename in a command-line argument. You can assume that the user will provide the *absolute path*[3] for the file. Note that, if a source file argument is specified, it **must** be the first argument. If a source file argument is not specified, the logfile.txt should be used by default.

5.2 Task 3: Copy file functionality

Instead of showing the contents of the file, allow the user to specify a `-d` argument at the command line in order to specify a destination directory where a copy of the source file will be created. When the `-d` option is used, the argument immediately following it should be given as the path of the destination directory. If it is not, the program arguments are invalid (see below).

If the destination directory already has a file with the same name as the source file or if the destination directory does not exist, your program should output an appropriate error message to standard error then exit cleanly with an error code of 2.

[3] An absolute path is defined as the **full** path beginning with /

Upon successful completion of the program with this argument, there will be **2 copies** of the same file (one in the source directory, the other one in the destination directory). Also, you should output the message "Copy successful" to standard output.

5.3 Task 4: Move file functionality

Allow the user to specify a `-M` argument at the command line to indicate that the file should be moved from the source directory to the destination directory. Note that, when the argument `-M` is provided, the `-d` argument is **compulsory** to provide a destination path where the source file should be relocated.

If the destination directory already has a file with the same name as the source file or if the destination directory does not exist, your program should output an appropriate error message to standard error then exit cleanly with an error code of 2.

Upon successful completion of the program with this argument, there will be only **1 copy** of the file (in the destination directory). Also, you should output the message "Move successful" to standard output.

5.4 Task 5: Force copy/move functionality

Allow the user to specify a `-F` argument at the command line to indicate that the file must be moved or copied from the source directory to the destination directory even if a file with the same name exists in the destination directory. Therefore, if the destination directory has a file which has the same name as the source file, it should be replaced by the the source file. Also, you should output the message of the functionality (copy/move), which was forced by this argument (as before, the appropriate success or error messages should be outputted).

Extend your user documentation to include **all the added usage functionalities.**

A summary of all the command-line arguments and their functionalities are shown in the table below:

Table 1: Summary of functionalities for different command-line arguments.

Command-line Argument	Functionality
sourcefile	Allows to input a path for the sourcefile to display/copy/move. sourcefile must appear immediately after the command (e.g., ./fileutil sourcefile).
-d destdir	Allows to input a path for the destination directory for copying or moving the source file. (destdir must appear immediately after -d)
-M	Changes the behaviour of the -d argument to move the file instead of copying. Only valid when used in conjunction with the -d ... argument.
-F	Makes the program to overwrite any file which has same name as the source file.

Example commands:

- `$ ./fileutil`
  Displays all the contents of the logfile.txt file from the current directory
- `$ ./fileutil /home/student/dir1/a.txt`
  Displays all the contents of the a.txt file
- `$ ./fileutil /home/student/dir1/a.txt -d /home/student/dir2/`
  Copy a.txt to dir2
- `$ ./fileutil /home/student/dir1/a.txt -d /home/student/dir2/ -F`
  Force the copying of a.txt from dir1 to dir2
- `$ ./fileutil /home/student/dir1/a.txt -d /home/student/dir2/ -M`
  Move a.txt from dir1 to dir2
- `$ ./fileutil /home/student/dir1/a.txt -d /home/student/dir2 -M -F`
  Force the move of a.txt from dir1 to dir2
- `$ ./fileutil -d /home/student/dir2`
  Copy logfile.txt from the current directory to dir2
- `$ ./fileutil /home/student/dir1/a.txt -M`
  Invalid argument, no destination to move the file!
- `$ ./fileutil /home/student/dir1/a.txt -M -F`
  Invalid argument, no destination to move the file!
- `$ ./fileutil /home/student/dir1/a.txt -F`
  Invalid argument, -F is redundant as nothing to force here!
- `$ ./fileutil /home/student/dir1/a.txt -d -F /home/student/dir2`
  Invalid argument: immediately after -d, a directory path was expected

- `$ ./fileutil /home/student/dir1/a.txt -F -M -d /home/student/dir2/`
  Force the move of a.txt from dir1 to dir2
- `$ ./fileutil /home/student/dir1/a.txt -d /home/student/dir2 -F`
  Force the copy of a.txt from dir1 to dir2

Some command-line arguments have **dependency** on other arguments (as mentioned above).

The position of the sourcefile argument is fixed. If it appears, it must be the first argument. However, the ordering of the rest of the arguments is NOT fixed and can vary. (For example, check the last 2 example commands.)

Do not collect these options from standard input, or prompt the user to enter them. They should be specified as *program arguments*.

5.5 Important: commenting is required

Commenting your code is essential as part of the assessment criteria (refer to Section 5.6). All program code should include three types of comments: (a) File header comments at the beginning of your program file, which specify your name, your Student ID, the start date and the last modified date of the program, as well as with a high-level description of the program. (b) Function header comments at the beginning of each function should describe the function, arguments and interpretation of return value. (c) In-line comments within the program are also part of the required documentation.

5.6 Marking Criteria

Each task is worth an equal share of the total. The same marking criteria will be applied on all tasks:

- 50% for working functionality according to specification.
- 20% for code architecture (algorithms, use of functions for clarity, appropriate use of libraries, correct use of pointers, etc. in your implementations of all the tasks.)
- 10% for general coding style (clarity in variable names, function names, blocks of code clearly indented, etc.)

- 20% for documentation (user documentation describes functionality for the relevant task, code is well-commented.)

5.7 Hints and tips

Do...	Don't... (!)
+ read up on low-level I/O system calls	– use fancy C `<stdio.h>` library functions that don't demonstrate your understanding of making system calls to the OS directly.
+ write normal program output to *standard output*	– write error messages to standard output (use `standard error` instead)
+ close open files before exiting and free any manually-allocated memory	– terminate without cleaning up first
+ check return values and handle error conditions	– act in undefined ways when a parameter is invalid or a file can't be opened
+ access and modify valid memory	– attempt to set values in undefined pointers
+ read up on `argc` and `argv`	– prompt the user to enter options after your program has started
+ write user documentation in a plain text file	– submit user documentation in a Word document or PDF
+ in user documentation: explain all functionality to those who might use your program	– expect your users to understand the C code inside your program (users are not always programmers!)
+ use descriptive, self-explanatory variable names	– use vague single-character variable names
+ break your program logic into multiple functions	– stuff everything into a single `main` function
+ use consistent code indentation for clarity	– use inconsistent indentation or fail to indent nested code blocks
+ comment your code with file header, function header and inline comments	– write uncommented code or add comments at the last minute
+ test your program in the specified VM environment	– forget to run your finished program through `valgrind` to test for memory access bugs
+ use the Ed Discussion Forum	**– implement each task as a separate program**
+ read the assignment specification carefully and thoroughly	**– treat this table as a substitute for reading the spec carefully.**

6 Submission

There will be NO hard copy submission required for this assignment. You are required to archive and compress all your deliverables into a single `.tar.gz` file named with your Student ID, for submission. For example, if your Student ID is `12345678`, you would submit a zipped file named `12345678_A1.tar.gz`.

Your submission is via the assignment submission link on the FIT2100 Moodle site by the deadline specified in Section 1, i.e. **11th September 2020 (Friday) by 5:00pm**.

Note: You must ensure you complete the entire Moodle submission process (do not simply leave your assignment in draft status) to signify your acceptance of academic integrity requirements.

6.1 Deliverables

Your submission should be archived and compressed into a single `.tar.gz` file containing the following documents:

- Electronic copies of ALL your files (i.e., C source file(s)) that are needed to compile and run your program. (Note that your program must run in the Linux Virtual Machine environment which has been provided for this unit. Any implementation that does not run at all in this environment will receive no marks.)
- A user documentation file (not more than 80 lines) in plain `.txt` format with clear and complete instructions on how to compile and run your program.

Marks will be deducted for any of these requirements that are not strictly complied with.

6.2 Academic Integrity: Plagiarism and Collusion

Plagiarism Plagiarism means to take and use another person's ideas and or manner of expressing them and to pass them off as your own by failing to give appropriate acknowledgement. This includes materials sourced from the Internet, staff, other students, and from published and unpublished works.

Collusion Collusion means unauthorised collaboration on assessable work (written, oral, or practical) with other people. This occurs when you present group work as your own or as the work of another person. Collusion may be with another Monash student or with people or students external to the University. This applies to work assessed by Monash or another university.

It is your responsibility to make yourself familiar with the University's policies and procedures in the event of suspected breaches of academic integrity. (Note: Students will be asked to attend an interview should such a situation is detected.)

The University's policies are available at: `http://www.monash.edu/students/academic/policies/academic-integrity`