THE UNIVERSITY OF WESTERN AUSTRALIA

Department of Computer Science and Software Engineering

CITS2002 Systems Programming

Project 2 2021, see also: Project 2 clarifications

Background

In recent years, computer disk sizes and densities have increased dramatically, with costs dropping to 4c/gigabyte (HDD) and 14c/gigabyte (SSD). As a consequence, we store files on our computers' disks in very different ways and, because we typically have a huge amount of free space available, we end up having multiple copies of many files on our disks. This presents few problems, until we eventually run out of disk space, or need to transfer files to more expensive cloud-storage, perhaps over a network with limited bandwidth. At that time we'd like to locate all duplicate files, and only make one copy of them to our backup destination.

> The **goal** of this project is to design and develop a command-line utility program, named *duplicates*, to locate and report duplicate files in, and below, a named directory.
>
> Successful completion of the project will develop and enhance your understanding of advanced features of the C11 programming language, and core Linux operating system system-calls and POSIX function calls.

There are two parts to the project:

1. The *basic* version requires you to develop a working program with a restricted set of features. It is possible to receive full marks for the project by only completing the *basic* version.
2. 🌶 If you would like a greater challenge, you may like to attempt the *advanced* version of this project. Undertaking and completing significant parts of the advanced tasks provides the opportunity to *recover* any marks not awarded for the *basic* version of the project.

Program invocation

Your implementation of *duplicates* will be invoked with zero or more valid command-line options, and one directory name. With no command-line options (i.e. only a directory name is provided) *duplicates* will simply list 4 things (with just one integer per line):

1. the total number of files found,
2. the total size (in bytes) of all files found,
3. the total number of unique files (i.e. any duplicate is only counted once), and
4. the possible minimum total size of all files found (i.e. the sizes of duplicated files are only counted once).

Files and directories (other than the "starting" directory indicated on the command-line) which cannot be read should be silently ignored (no error messages should be printed).
For the *basic* project, the "starting" directory will only contain regular files and sub-directories. In particular, there will be no hard- or symbolic-links.
Your project is required to support the following command-line options and, if attempting the *advanced* version of the project, command-line options marked with a chili 🌶.

-a	By default, hidden and configuration files (conventionally those beginning with a '.' character) are not considered by *duplicates*. Providing the *-a* option requests that *all* files be considered. This is similar to the standard Linux utility *ls*.
-A	This option indicates if the program attempts the *advanced* version of the project. *duplicates -A* produces no output at all, simply terminating with *EXIT_SUCCESS* (for advanced) or with *EXIT_FAILURE* (for basic).
-f filename	find and list, one per line, the relative pathnames of all files whose SHA2 hash matches that of the indicated file. The name of the indicated file is *not* listed. *duplicates -f* terminates with *EXIT_SUCCESS* if any matching files are found, or with *EXIT_FAILURE* otherwise.
-h hash	find and list, one per line, the relative pathnames of all files with the indicated SHA2 hash. *duplicates -h* terminates with *EXIT_SUCCESS* if any matching files are found, or with *EXIT_FAILURE* otherwise.
-l	*duplicates* lists all duplicate files found. Each line of output consists of the relative pathnames of two or more files that are duplicates of each other. The pathnames of duplicate files (on the same line line) must be separated by the *TAB* character.
-m	🌶 *duplicates* minimizes the total number of bytes required to store all files' data by modifying the directory structure being considered.
-q	*duplicates* executes quietly, simply testing if the named directory contains any duplicate files. *duplicates -q* produces no output at all, simply terminating with *EXIT_SUCCESS* if there are no duplicates (i.e. storage is already minimized), or with *EXIT_FAILURE* otherwise.

Detecting duplicate files

- Two or more files are *defined to be duplicates* iff their *contents* are identical. Duplicate files will thus have the same size but, when determining if two files are duplicates of each other, their filenames and modification times are not considered.
- A file is *defined to be unique* iff no other file has the same contents.

To detect duplicate files we'll employ a *cryptographic hash function* named *SHA2* (pronounced 'shar-2') [Wikipedia] and [Youtube]. SHA2 examines the contents of a file and produces a fixed-length summary of its contents. Cryptographic hash functions are designed by mathematicians and those developing encryption and security software.

Here is an implementation of the function strSHA2.c which you may (should) use without attribution.

Two or more files are considered identical if their cryptographic hashes are identical. For this project, we'll use a C11 string to store this representation, and two files will be considered identical if their SHA2 string representations are identical. The function *strSHA2()*, with the following prototype:

```
char *strSHA2(char *filename);
```

is provided for this project (it is not a standard C11 function). If *strSHA2* can read the indicated file, it will return a dynamically allocated string holding the SHA2 string representation of the file's contents. If the indicated file cannot be read, *strSHA2* will return *NULL*. Note that you *do not* have to understand the *SHA2* algorithm or its *strSHA2()* implementation for this project.

Getting started

There is no *required* sequence of steps to undertake the project, and no sequence of steps will guarantee success. However, the following sequence is strongly recommended (and this is how the sample solution was built). It is assumed (considered essential for success!) that each step:

- is extensively tested before you proceed to the next step, and
- reports any errors to *stderr* as they are found. Serious errors may require the whole program to terminate.

The recommended steps:

0. **find a project partner**.
1. determine what activities are to be performed by the program. Determine what data needs to be stored during the execution of the program. Design one of more data types and data structures to store this long-term data, and create a new *duplicates.h* header file to define these constructs.
2. break up the activities into fairly independent sets. Write a number of "empty" C files, each of which includes your *duplicates.h* header file, to implement the activities. Ensure that your file implementing the *main()* function is named *duplicates.c*.
3. write a *Makefile* to compile and link your C files, each of which will be dependent on the header file.
4. write the *main()* function, whose initial task is simply to receive and validate the command-line options and arguments. Write a *usage()* function to report the program's synopsis, should command-line processing detect any errors.
5. ensure that the "real" command-line argument is the name of a directory that you can read.
6. open and read each directory, locating all regular files in the directory. At this stage (as a debugging aid) just print each filename as it is found. When directories are located within directories, you should process those directories *recursively*.
7. add support for the *-a* command-line option when finding files.
8. store the name (location) and size of each file found in each directory.
9. having finished reading the directories, identify all duplicate files.
10. if the *-q* command-line option is provided, perform its role and terminate.
11. if no command-line options are provided, print out the required "statistics" about the files found and terminate.
12. if the *-l* command-line option is provided, perform its role and terminate.
13. armed with a fully tested program, and overflowing with confidence, consider of the *advanced* tasks (described below).

It is anticipated that a successful project will use (some of) the following Linux system-calls, and standard C11 & POSIX functions:

getopt(), malloc(), realloc(), free(), opendir(), readdir(), closedir(), stat(), and perror(); link()✒, and unlink();✒, strdup(), and strSHA2() (neither are part of the C11 standard).

Advanced tasks

If you would like a greater challenge, you may like to attempt to an advanced version of this project. Undertaking and completing significant parts of the advanced tasks provides the opportunity to *recover* any marks not awarded for the *basic* version of the project.
There are 3 additional tasks in the advanced section:

1. ✒ support duplicate detection in and below two or more directories provided on the command-line. For example, if four directory names are provided, then *all files* in *all four directories* should be considered.

2. ✒ support the presence of hard-links (but not symbolic-links) in the directories being considered. When counting the total number of bytes occupied by all files, the "contents" of all files hard-linked together are counted only once.

3. ✒ support the *-m* command-line option.
   This task requires you to identify duplicate files and to then store only one instance of each (with possibly different names). The Linux *link()* system call (see *man 2 link*) provides this facility for us, by creating *hard-links* between two or more files. The actual file *contents* will only be stored only once, and multiple relative pathnames will refer to that single copy.

   WARNING: until you are very confident that your *duplicates* program is working correctly, you are strongly advised NOT to use your *duplicates* program with its *-m* option to minimize the storage of *important* files and directories! Test with an unimportant temporary directory.

Project requirements

1. Your project **must** be developed in multiple C11 source files and (at least one) header file.

2. Your submission will be compiled with the C11 compiler arguments **-std=c11 -Wall -Werror -pedantic**, and marks will not be awarded if your submission does not compile.

3. Compilation and linking of your project **must** be supported by a *Makefile*, containing approriate variable definitions and automatic variables.

4. The default target of your *Makefile* **must** be named *duplicates*, and invocation of the *make* command must produce an executable program named *duplicates*.

5. Your project **must** employ sound programming practices, including the use of meaningful comments, well chosen identifier names; appropriate choice of basic data-structures, data-types, and functions; and appropriate choice of control-flow constructs.

Assessment

The project is due at **11:59pm Friday 22nd October (end of week 12)**, and is worth **25% of your final mark** for CITS2002.

It will be marked out of 50. The project may be completed **individually or in teams of two** (but not teams of three). The choice of project partners is up to you - you will not be automatically assigned a project partner.

You are **strongly** advised to work with another student who is around the same level of understanding and motivation as yourself. This will enable you to discuss your initial design together, and to assist each other to develop and debug your joint solution. Work together - do not attempt to split the project into two equal parts, and then plan to meet near the deadline to join your parts together.

25 of the possible 50 marks will come from the correctness of your solution. The remaining 25 marks will come from your programming style, including your use of meaningful comments; well chosen identifier names; appropriate choice of basic data-structures, data-types and functions; and appropriate choice of control-flow constructs.

During the marking, attention will obviously be given to the correctness of your solution. However, a correct and efficient solution should not be considered as the perfect, nor necessarily desirable, form of solution. Preference will be given to well presented, well documented solutions that use the appropriate features of the language to complete tasks in an easy to understand and easy to follow manner. That is, do not expect to receive full marks for your project simply because it works correctly. Remember, a computer program should not only convey a message to the computer, but also to other human programmers.

Submission requirements

1. The deadline for the project is **11:59pm Friday 22nd October (end of week 12)**.

2. **You must submit your project electronically using *cssubmit*.**

   You should submit ALL C11 source-code (*.c) and header (*.h) files and a *Makefile* that specify the steps required to compile and link your application. You **do not need to submit** any data files/directories or testing scripts that you used while developing and testing your project. You can submit multiple files in one submission by first archiving them with *zip* or *tar*.

   The *cssubmit* program will display a receipt of your submission. You should print and retain this receipt in case of any dispute. Note also that *cssubmit* does not archive submissions and will simply overwrite any previous submission with your latest submission.

3. **At least one** of your submission's C11 source files **must** contain C11 block comment:

```
//  CITS2002 Project 2 2021
//  Name(s):             student-name1 (, student-name2)
//  Student number(s):   student-number1 (, student-number2)
```

4. If working as a team, only one team member should make the team's submission.

5. Your submission will be examined, compiled, and tested on a contemporary **Ubuntu Linux system**. Your submission should work as expected on this platform. While you may develop your project on other computers, excuses such as *"it worked at home, just not when you tested it!"* will not be accepted.

6. This project is subject to UWA's Policy on Assessment - particularly §5.3 *Principles of submission and penalty for late submission*. In accordance with this policy, you may *discuss* with other students the general principles required to understand this project, but the work you submit must be the result of your own efforts. All projects will be compared using software that detects significant similarities between source

code files. Students suspected of plagiarism will be interviewed and will be required to demonstrate their full understanding of their project submission.

Clarifications

Please post requests for clarification about any aspect of the project to *help2002* so that all students may remain equally informed. If necessary, the main project page will be updated to clarify any English in the project's description. Make reference to the online copy, and not a (dated) paper copy.

Good luck!

Amitava Datta and Chris McDonald.