Operating Systems 1: Project D - File indexing

Table of Contents

1 Overview

The aim of the project is to create a program that: traverses all files in a given directory and its subdirectories, creates a data structure containing the requested information about the files and then waits for user input. User inputs commands that query data gathered in the data structure. To avoid repetitive scanning of the directory, the data structure is written to a file and read when the program is run again.

From now on we will call the data structure *index*.

The assignment states that some operations must be run in separate thread. This threads must be created an used. In cases where it is not specified the solution can use any number of additional threads (including none at all).

2 Invocation and arguments

The project should provide a single executable called `mole`. The program has the following commandline arguments:

-d path a path to a directory that will be traversed, if the option is not present a path set in an environment variable `$MOLE_DIR` is used. If the environment variable is not set the program end with an error. The effective value of the parameter is denoted as **path-d**.

-f path a path to a file where *index* is stored. If the option is not present, the value from environment variable `$MOLE_INDEX_PATH` is used. If the variable is not set, the default value of file '.mole-index' in user's home directory is used. Effective value of the parameter is denoted as **path-f**.

-t n where n is an integer from the range \([30,7200]\). n denotes a time between subsequent rebuilds of *index*. This parameter is optional. If it is not present, the periodic re-indexing is disabled. The value of the option will be referred as **t** in latter paragraphs.

3 Program specification

When stated, the program tries to open a file pointed by **path-f** and if the file exists *index* from the file is read otherwise the program starts indexing procedure described later. After that program starts waiting for user's input on `stdin`.

3.1 Indexing procedure

Index stores the information about the following file types:

- directories

- JPEG images
- PNG images
- gzip compressed files
- zip compressed files (including any files based on zip format like docx, odt, ...).

A file type recognition must be based on a file signature (a so called magic number) not a file name extension. Any file types other than the above are excluded from *index*.

Index stores the following information about each file:

- file name
- a full (absolute) path to a file
- size
- owner's uid
- type (one of the above).

The indexing procedure works as follows: a single thread is started. The thread creates a new *index* by traversing all files in **path-d** and its subdirectories. For each file a file type is checked and if the type is one of the indexed types, the required data is stored in *index*. Once traversal is complete, the index structure is written to **path-f**.

Once (run in background) indexing is finished the stdout notification about finished indexing is printed.

3.2 Available commands

A command processing works parallel to a potential re-indexing process. The commands must be processed even if indexing is in progress. As the new index structure is not ready an old version may be used to provide user with the answers.

The program reads subsequent lines from `stdin`. Each line should contain the one of the following commands. If the read line is not a command an error message is printed and the program waits for the next line.

Commands:

exit starts a termination procedure – the program stops reading commands from stdin. If an indexing is currently in progress, the program waits for it to finish (including writing the result to the file) and then the program ends

exit! quick termination – the program stops reading commands from stdin. If any indexing is in progress it is canceled. If the result of the indexing if currently being written to a file, the program waits for it to be finished (it is required that after the program termination the index file is not broken in any way, for instance by unfinished writes).

index if there is no currently running indexing operation a new indexing is stated in background and the program immediately starts waiting for the next command. If there is currently running indexing operation a warining message is printed and no additional tasks are performed.

count calculates the counts of each file type in *index* and prints them to stdout.

largerthan x x is the requested file size. Prints full path, size and type of all files in index that have size larger than x.

namepart y y is a part of a filename, it may contain spaces. Prints the same information as previous command about all files that contain y in the name.

owner uid uid is owner's identifier. Same as the previous one but prints information about all files that owner is uid.

3.2.1 Pagination If any of the commands: largerthan, namepart, owner is going to print more than 3 records as a result, the user needs an ability to scroll through results. This capability has to be implemented with use of value of `$PAGER` environment variable. If `$PAGER` is not set or the number of records is less than 3 the data is just printed to `stdout`. Otherwise the procedure is as follows: start a program provided by `$PAGER` variable with `popen` function. Print the data to a stream provided by the function (the other end of the provided stream is `stdin` of the child process running `$PAGER`) and wait for a child process termination with `pclose`.

`popen` and `pclose` were not introduced during the lab, you have to find out the needed information in the manual.

An example pager (which you can set `$PAGER` variable to) is `less` program.

3.3 Data structure

There are no requirements about data structure used to store index. It can use anything including arrays, lists and trees. The design of the structure should incorporate the need for reading and writing it to a file. For instance use of pointers may complicate reading and writing the structure.

There may be an arbitrary limit on lengths of file names and paths. In such case the indexing operation must print warning messages when the file name/path is too long (and continue indexing). Limits must be easy to configure during project compilations.

3.4 Reading and writing index file

An index file must be read and written using low level POSIX IO. Moreover there should not be any unnecessary conversion to strings. If data can be written in binary format it should be done this way. The files do not need to be portable between architectures (so you not need to care about byte order, different sizes of types or structure padding).

3.5 Re-indexing in background

If the parameter **t** s present, the program starts a thread that runs indexing process when the index is older than **t** seconds. A time is counted from either last re-indexing on timeout or a manual re-index whichever is later. If the index was read from a file the last indexing time is set to the file modification time (this may trigger an immediate re-indexing after reading an old file).

4 Other constraints

Only low level POSIX IO can be used for file read/write. The exception is writing/reading stdout/err/in and results of `popen`.

Created: 2020-12-07 pon 12:23

Validate