CSI 402 – Systems Programming – Spring 2014

Programming Assignment V

Date given: Apr. 17, 2014 **Due date:** May 5, 2014
Weightage: 10%

The deadline for this assignment is **11 PM, Monday, May 5, 2014**. *There is no two-day grace period for this assignment.* Thus, the assignment *won't* be accepted after 11 PM, Monday, May 5, 2014.

Very important: There are two parts in this assignment. For Part (a), your submission may consist of a single C source file. For Part (b), the source program must be split into two or more C files in a meaningful fashion. Note, in particular, that your submission must have only ONE `makefile` that can create executables for both the parts. All the source files for both the parts (along with the `makefile`) must be submitted together using only one `turnin-csi402` command. Additional specifications regarding the `makefile` will be included in the `README` file for this assignment.

The total grade for the assignment is 100 points, with 35 points for Part (a) and 65 points for Part (b).

Description of Part (a):

In Unix, file names starting with the character `'.'` are referred to as **hidden** files since the `ls` command normally does not show such files. For example, the home directory of each user may contain hidden files such as `".bash_profile"`, `".emacs"`, etc. Also, the convention in Unix is to use the file name `"."` to refer to the current directory and the file name `".."` to refer to the parent of the current directory.

The executable version of your program for Part (a) must be named `p5a`. It will be executed by a command line of the following form:

`p5a` *PathName* *OptionalFlag*

Here, *PathName* gives the full path name of a directory. As the name suggests, the last argument *OptionalFlag* is **optional**; if it is specified, the only acceptable flag is `"-s"`.

Your program should go through the files in the specified directory. For each hidden file in the directory, except `"."` and `".."`, your program must print to `stdout`, the file name, the (logical) size of the file in bytes and the date (month, day and year) of last modification of the file. When the optional argument is *not* specified, your program may print the output in any order. However, when the flag `"-s"` is specified, the output must be printed in non-decreasing order of logical file size.

Your program for Part (a) should detect the following errors.

(1) The number of command line arguments is not equal to two or three. This is a fatal error. In this case, your program should produce a suitable error message to `stderr` and stop.

(2) The number of command line arguments is equal to three but the flag specified is not `"-s"`. This is also a fatal error. Here also, your program should produce a suitable error message to `stderr` and stop.

(3) The directory specified on the command line cannot be opened. This is yet another fatal error, and your program should produce a suitable error message to `stderr` and stop.

(4) The program is unable to obtain information about a specific file using the `stat` function. This is *not* a fatal error. In this case, your program should print a suitable message to `stdout` giving the name of the file for which the call to `stat` failed. However, the program should *not* stop; instead, it should continue to examine the other files in the specified directory.

In writing the program for Part (a), the following material from the text by Haviland, Gray and Salama will be helpful.

(i) Pages 53 through 57 of Section 3.3 (**Obtaining file information: `stat` and `fstat`**): In particular, you must understand the specifications of the library function `stat` and the components of the `stat` structure (pages 53–54).

(ii) Pages 67 through 71 of Section 4.4 (**Programming with directories**): In particular, you must understand the specifications of the functions such as `opendir`, `closedir`, `chdir` and `readdir`. The two program examples given on pages 70 and 71 will also be helpful.

(iii) Page 318 of Section 12.4 (**Time**): This section discusses how the time values stored in the `stat` structure can be converted into conventional date/time specifications.

Description of Part (b): In this part, you will be implementing a basic version of the Unix command `tar`. This program (whose name is an abbreviation of the phrase "tape archive") combines several files into a single file (called an archive) in such a way that the individual files can be "extracted" from the archive.

The executable version of your program must be named `p5b`. It will be executed using a command line which has one of the following three forms:

`p5b -c` *archive infile1 infile2 ... infilek*
`p5b -x` *archive*
`p5b -p` *prefix archive*

In the above commands, *archive* denotes the name of a file created by combining several files. The arguments *infile1*, *infile2*, ..., *infilek* denote the names of the files which must be combined to create an archive. The meanings of the commands are as follows.

(i) The first form of the command, which has the flag `"-c"`, *creates* the specified archive from the input files specified on the command line. Note that this command should leave the input files intact.

(ii) The second form of the command, which has the flag `"-x"`, *extracts* the individual files from the specified archive. This command should leave the archive file intact.

(iii) The third form of the command has the flag `"-p"`. Here, *prefix* is a string. This command should *write* to `stdout` the names and sizes of the files in the archive whose names start with or fully match the string specified by *prefix*. If there are no such files in the archive, the command should print an error message of the form "`No matching files found.`" to `stdout`. Note that this command should *not* extract the files from the archive. Further, it should leave the archive file intact.

For the `"-c"` option, assume that the input files are in the current directory and that the archive is to be created in the current directory. For the `"-x"` option, assume that the archive file is in the current directory and that the extracted files must also be in the current directory. For the `"-p"` option, assume that the archive file is in the current directory.

The number of input files specified in the command may vary from 1 to 255. Some of the input files may be text files and others may be binary files. Your program should work correctly regardless of the types of input files. The size of each input file is at most $2^{32} - 1$ bytes. (Thus, the size of each file can be stored in a variable of type `unsigned int`.)

Format of the archive: The archive is an *unformatted* (i.e., binary) file. The format of this file is shown below.

The first byte of the archive (denoted by `N`) stores the number of files which were combined to form this archive. (Recall that the number of input files may vary from 1 to 255.) This is followed by the information about the individual files (denoted by `I1` through `IK` in the figure). In turn, this is followed by the actual bytes of the individual files (denoted by `B1` through `BK` in the figure). The information about each individual file is organized as shown in the figure below.

In the above figure, the first byte (represented by `L`) gives the number of characters in the name of the file; this count does *not* include the byte for the `'\0'` character. Suppose the decimal value stored in the byte is ℓ. The next ℓ bytes contain the name of the file (denoted by `S` in the above figure). The name is followed by 4 bytes (denoted by `Z` in the above figure) which give the size of the file in bytes.

Once you understand the format of the archive file, you should be able to see how the archive can be created from the individual files and how the individual files can be extracted from an archive.

Errors to be detected: Your program for Part (b) needs to detect only the usual command line errors: wrong number of arguments, wrong command line flag or a file specified on the command line can't be opened. In each case, the program must print an appropriate error message to `stderr` and stop.

Suggestion: In writing the program for Part (a), the following material from the text "Unix System Programming" by Haviland, Gray and Salama is likely to be helpful.

(i) Functions `open`, `close`, `read`, `write` and `lseek` from Chapter 2 of the above text.

(ii) Pages 53 through 57 of Section 3.3 (**Obtaining file information: `stat` and `fstat`**): In particular, you can get the size of a file from the `stat` structure returned by these functions; see pages 53–54 of the above text.

Information about `README` file: The `README` file for this assignment will will be available by 10 PM on Tuesday, April 22, 2014. The name of the file will be `prog5.README` and it will be in the directory `~csi402/public/prog5` on `itsunix.albany.edu`.