CMSC 421 – Operating Systems
Krishna Sivalingam and Dennis Frey
Spring 2006
Project 3: File System
Due Date: **May 16, 9PM**; On-Line Submission via *submit* command
NO LATE SUBMISSIONS

1 Project description

The objective of this project is to implement a simulated file system. Your simulated file system will be fully contained in a single file called "Diskfile". The Diskfile will contain directory disk information, free block information, file data blocks, etc. The name of the diskfile and the size of the disk file (in KB) will be input as command line parameters. Your project will accept system commands from the user as described in the sample session. One system command ("load") will require that you read and execute directory/file commands from a data file (see "Input Executable data files").

The project will be implemented in C/C++ that will run in the GL Linux environment using the /usr/local/bin/g++ or /usr/local/bin/gcc compilers with the -Wall and -ansi flags set.

Input values: Diskfile, Disk size (D in KB), one or more executable files. The size of each disk block is fixed to be 128 bytes. Thus, $D = 10$, indicates an 10 KB virtual disk with block size of 128 bytes, and there are 80 blocks in the disk. The disk block address size is 32 bits.

Diskfile information:

1. If *Diskfile* already exists in the current directory, the value of D is ignored and Diskfile will serve as the virtual disk.
2. If Diskfile does not exist in the current directory, create this file with specified size in the current directory. This file will serve as the virtual disk. Note that between different executions of your project, the virtual disk contents are maintained.

Disk organization: The first part of the disk will contain a set of blocks dedicated to maintain disk directory information. The number of blocks reserved for this information is left to the implementer. The free space maintenance data structure to be used is the *Free Bit Vector* explained in the book. Note that this vector should also be stored on the virtual disk.

The disk will contain a list of files and directories, with a maximum of **two levels** in the directory tree. The root of the disk is denoted as '/'.

For example, the disk can have contents: `/f1, /f2, /d1/f3, /d2/f4`

where the above denotes two files f1 and f2 and two directories /d1 and /d2 containing one file each.

File Information: A file is viewed as a sequence of bytes, with the first byte numbered 0. File is allocated space one block at a time. All references to files in this project will be by their fully expanded names (i.e. including complete directory tree information), as in /dir1/file2.h. For filenames, extensions are optional and can include up to three characters.

The **inode** structure for each file contains the following information:

1. Directory name or Filename: Up to 16 bytes total for a fully expanded filename (including the two '/' characters). The directory name can have up to 4 bytes.

Thus, di4, dir3 are valid directory name, while direc3 is not; Likewise, /dir3/filename.c is valid since it contains 16 total characters.

2. Type: Directory (value = 1) or File (value = 0): Use 1 byte to store this value.
3. File size: recorded using 4 bytes. Theoretical maximum file size is 2^{32} bytes, but in practice, the file sizes for this project will not exceed 4,480 Bytes.
4. Date Created: Output of *date* command of Linux - requires a 28-byte string.
5. Date Last Modified: Output of *date* command of Linux - requires a 28-byte string.
6. Direct block addresses: 3 4-byte integers, that will point to the first three data blocks of the file.
7. Index block address: 1 4-byte integer, that will point to one index block, which can contain up to 32 block addresses (Since each block size is 128 bytes and each block address needs 4 bytes). These addresses in the index block point to blocks 4, 5, 6, etc. of the file.

   Thus, the maximum possible file size = $(3 + 32) = 35 \times 128 = 4,480\text{Bytes}$.

Note that the inode information for each file is stored on the disk in the disk directory part of the disk. Assume that only one inode is stored per disk block.

You must implement a *linear disk directory* structure, that will store information about all files on the directory and the inodes for the files. The specific inode structure and data maintained for directory level information is left open to the implementer.

System commands: On startup, the system reads the disk directory contents (if the disk has any) and waits for commands from user. That is, it is an interactive program that executes the user's commands, which will be from the following list:

▷ `load filename`

Execute the sequence of instructions contained in the file. If the file does not exist, output an error message and continue to accept the next command.

▷ `printinode filename`

Print the inode contents for the specified file.

For example, assuming a block size of 128 bytes and file size of 600 bytes (5 blocks):

```
**********************
Filename: /file1.c
Size:  bytes
Date Created: Thu Apr 13 17:59:55 EDT 2006
Date Last Modified: Thu Apr 20 16:45:10 EDT 2006
Direct block values: 23 51 67
Index block is stored in: 88
Index block contents: 34 12
**********************
```

▷ `exit`

Exit the system, but leave the virtual disk intact.

Input Executable data files: The executable files will consist of a set of commands in the format explained below. All command names and file names are case sensitive.

▷ open `filename`

Semantics: If the file does not exist, creates the file with name specified by filename. All filenames are fully specified including the parent directory, as in /dir2/abc.mp3. Upon creation, a file has size of 0 bytes. The file is opened for both reading and writing. Set the read and write pointers for this file to 0.

If the file already exists, open the file for reading and writing. Set the read and write pointers for this file to 0.

Thus, there are two possible filename formats: format /file1.c and /dir1/file2.h

If the specified sub-directory does not exist, report an error and process next command.

Output: Echo the command to the screen and the result, as in:

COMMAND: create /file1.c ; RESULT: Created /file1.c
COMMAND: create /dir1/file2a.c ; RESULT: Created /dir1/file2a.c
The above assumes that /dir1 exists.

COMMAND: create /dir3/file9.c ; RESULT: Error - /dir3/ does not exist
The above assumes that /dir3/ does not exist

▷ createdir `dirname`

Semantics: Creates the directory with name specified. If the directory already exists, print an error message and process next command.

Output: Echo the command to the screen and the result, as in:

COMMAND: createdir /dir4 ; RESULT: /dir4 has been created.

▷ seekread `filename` pos

Semantics: Move the read pointer for the specified file to byte number pos. Remember that the first byte of a file is numbered 0.

If specified filename does not exist, print an error message and process next command in file.

If pos points to a location beyond the last valid character, print an error message and process next command in file.

Output: Echo the command to the screen and the result, as in:

COMMAND: seekread /file1.c 35; RESULT: Read pointer set to byte 35
COMMAND: seekread /dir1/file2c.c 35; RESULT: File does not exist
COMMAND: seekread /dir1/file2a.c 100; RESULT: File length is exceeded.

▷ seekwrite `filename` pos

Semantics: Move the write pointer for the specified file to byte number pos. Remember that the first byte of a file is numbered 0.

If specified filename does not exist, print an error message and process next command in file.

If pos points to a location beyond the last valid character, move the pointer to just past the last valid byte.

Output: Echo the command to the screen and the result, as in:

COMMAND: seekwrite /file1.c 35; RESULT: Write pointer set to byte 35
COMMAND: seekwrite /dir1/file2c.c 35; RESULT: File does not exist
COMMAND: seekwrite /dir1/file2a.c 100; RESULT: Write pointer set to byte 90

In above example, assume that the file contained 90 bytes when command was issued.

▷ seekwriteend `filename`

Semantics: Move the write pointer for the specified file to past the last valid byte in the file. For instance, if the file has 10 bytes, then the write pointer will point to position 11.

If specified filename does not exist, print an error message and process next command in file.

Output: Echo the command to the screen and the result, as in:

COMMAND: seekwriteend /file1.c; RESULT: Write pointer set to byte 11

▷ read `filename` length

Semantics: Print, to screen, the characters starting from readpointer to Minimum(filelength - 1, readpointer+length-1). If file does not exist, print an error message and proceed to next message in command.

Output: Echo the command to the screen and the result, as below. Assume that readpointer is 10:

COMMAND: read /file1.c 10; RESULT:

```
************************
Filename: /file1.c; Bytes read: 10 - 19
String stored: klmnopqrst
************************
```

As another example, assume that readpointer is 5 and file length is 10 bytes:

COMMAND: read /dir1/file2a.c 10; RESULT:

```
************************
Filename: /dir1/file2a.c; Bytes read: 5 - 9
String stored: fghij
************************
```

▷ write `filename` string

Semantics: Write to file, the string in locations starting from writepointer to writepointer+length-1; where length is the length of the string (in bytes). The file length will be updated if the write goes past the previous write byte. The string will contain only readable characters from the set $\{[a-z], [A-Z], [0-9], \#, ?, \$, !, \&\}$ and will contain no spaces.

If file does not exist, print an error message and proceed to next message in command.

Output: Echo the command to the screen and the result, as below. Assume that writepointer is 10:

COMMAND: write /file1.c ; RESULT: File written and length is: 25 bytes

▷ delete `filename`

Semantics: Delete the file, if it exists. If it does not exist, print an error message and proceed to next command in the file.

Output: Echo the command to the screen and the result of the addition, as in:

COMMAND: delete /file1.c; RESULT: /file1.c deleted COMMAND: delete /file39.c; RESULT: /file39.c does not exist

▷ deletedir `dirname`

Semantics: Delete the directory, if it exists AND if there are no files under this directory. Else, print an error message and proceed to next command in the file. Note: The root (/) directory CANNOT be deleted.

Output: Echo the command to the screen and the result of the addition, as in:

COMMAND: deletedir /dir1; RESULT: /dir1 has files and not deleted
COMMAND: deletedir /dir2; RESULT: /dir2 is deleted
COMMAND: delete /; RESULT: Quit kidding - will not delete it!

- ls `dirname`

  Semantics: List any directories, filenames and filesizes under the specified directory – Recursive listing is not needed. List one directory and one filename per file

  Output: Echo the command to the screen, as in:

  COMMAND: ls /dir11; RESULT: /dir11 does not exist

  COMMAND: ls /; RESULT:

```
******************************
Directories:
/dir1
/dir2
Files:
/file1.c  25
/song.mp3 128
/dir1/file2.c 64
******************************
```

Note: For all appropriate commands, if there is no space available on the virtual disk, print an error message and proceed to the next command.

2 Sample Session

Assume that you have created the necessary files and the corresponding executables in your PROJECT3 directory.

Assume that $D = 64$; and that the mydisk.fil does not exist. Also, there is one executable file as described below, with **file1** having following contents:

```
ls /
open /file1.c
open /dir1/file2.c
createdir /dir1
open /dir1/file2.c
seekread /file1.c 10
```

```
write /file1.c HelloWorld?HowAreYouDoing?
read /file1.c 10
write /dir1/file2.c EducationIsforLife!NotMerelyALiving!
read /dir1/file2.c 15
ls /
delete /dir2/file3.c
deletedir /dir2
```

The sample session is as follows.

```
% cd PROJECT3

% make proj3

% ./proj3 mydisk.fil -D 64
<Command Please> load file1
<Command Please> printinode /file1.c
<Command Please> printinode /dir3/file2.c
<Command Please> printinode /dir1/file2.c
<Command Please> exit
```

Now, when you invoke the program again with same virtual disk file, all disk contents should be intact as left off from the previous run. Also, there is one executable file as described below, with **file2** having following contents:

```
ls /
open /file1.c
seekread /file1.c 15
seekwrite /file1.c 25
write /file1.c This#Is#A#Sample#Sentence#The#Accused#Got#Ten#Years!
read /file1.c 15
ls /
delete /dir1/file2.c
deletedir /dir1
```

The second run is as follows:

```
% ./proj3 mydisk.fil -D 64
<Command Please> load file2
<Command Please> printinode /file1.c
<Command Please> printinode /dir1/file2.c
<Command Please> exit
```

3 What to Submit

Log into one of the campus GL machines. Change to your project directory. Make sure that ONLY files related to this project are in this directory - we do not want any temporary files, music files, etc.

Submit your project using the *submit* command. For help with submitting, click on the *Miscellaneous* link on the course homepage.

```
submit cs421 proj3 <list of files>
```

You can test the program execution after submission using *submitmake* and *submitrun* routines that are provided in Mr. Frey's public directory (`/afs/umbc.edu/users/d/e/dennis/pub/CMSC421`).

Submit the following files:

- ▷ Source Files
- ▷ Makefile
  Typing command 'make' at the Linux command prompt **MUST** generate the required executable. The executable **MUST** be named **proj3** as in the samples above. Make must use either the /usr/local/bin/g++ or the /usr/local/bin/gcc compiler as well as the -Wall and -ansi flags.
- ▷ A Script file obtained by running UNIX command *script* which will record the way you have finally tested your program. The script file will show the execution of the system demonstrating at least the sample session above. You can include additional runs of the system.
- ▷ A README file for the TA. The README should document known error cases and weaknesses with the program. You should also document if any code used in your submission has been obtained/modified from any other source, including those found on the web. If you helped any other 421 student and/or took help from/discussed with any other student, please describe it here.
- ▷ A COMMENTS file which describes your experience with the project, suggestions for change, and anything else you may wish to say regarding this project. This is your opportunity for feedback, and will be very helpful.

4 Help

1. WARNING ABOUT ACADEMIC DISHONESTY: Do not share or discuss your work with anyone else. The work YOU submit SHOULD be the result of YOUR efforts. The academic conduct code violation policy and penalties, as discussed in the class website, will be applied.
2. Ask questions EARLY. Do not wait until the week before. This project is quite time-consuming.
3. Implement the solutions, step by step. Trying to write the entire program in one shot, and compiling the program will lead to frustration, more than anything else.

5 Grading

- ▷ Basic disk organization and command interpretation: 15 points

- ▷ open, seekread, seekwrite: 15 points
- ▷ read, write: 15 + 35 = 50 points
- ▷ delete, deletedir: 15 points
- ▷ printinode: 5 points

No README/COMMENTS: -5 points; No Script File: -10 points; Incomplete Compilation: -10 points