Pseudo inode-based File System

The inode-based file system is one of the most prevalent file systems and has been in use for over fifty-years. The inode-based file system is used for Linux, UNIX, BSD and many other files system.

Your goal is to implement an in-core (RAM) file system using a simplified version of inodes. The file system is largely an implementation of the file system discussed in your author's textbook.

Your file system is not required to support access control, permissions or other attributes. You only need to support, at a minimum a single directory (the root directory), file creation, reading, writing and listing the directory contents.

DISK

The geometry of the file-system (disk) is a linear collection of Blocks.

- Each **Block** is 1024 bytes in size
- The disk has a total of 1091 Blocks
- The 1st block is reserved and contains an identifying string only
- The remaining blocks are:
    - One inode-bitmap block
    - One data-bitmap block
    - Sixty-four (64) Inode blocks
    - 1024 Data blocks

    (similar to authors diagram in text chapter 40, page 4)

Inode-bitmap

- Used to track inode-entries

- Assume 64 (16-bit words, i.e. an unsigned short)
- Each word (16-bits) tracks 16 inode-entries
- Use bit-operations to determine next available inode-entry

Data-Block-bitmap

- Used to track available data-blocks.
- We have 1024 data-blocks
- Use bit-operations to determine next available inode-entry

Inode-Blocks

- Used to store inodes. Inodes is a structure that stores meta-data about files.
- Each Inode-block stores 16 inode-entries.
- Your file system should have 64 Inode-blocks. This will allow us to have a maximum of 1024 files (64 inode-blocks x 16 inode-entries == 1024)

Data-Blocks

- Contains raw data for files or directories
- There is at least one data-block used for the **root** directory
    - The root directory initially has two entries
        - "."
        - ".."

Supported Operations

- void **pdos_mkdisk()**
    - creates disk storage for the in-memory file system (FS)
- void **pdos_mkfs(char * ID)**

- lays out the structure for the file system and initially populates the administration (super block, inode-bitmap block, data-bitmap blocks, etc.)
- Creates a single root directory and updates the administration blocks appropriately.
- The above represents an empty file system

- PDOS_FILE ***pdos_open**(const char *fname, const char *mode)

  - creates a new file or open an existing file

- int **pdos_fgetc**(PDOS_FILE *)

  - read a byte of data from file. returns **EOF** if end of file

- void **pdos_fputc**(int b, PDOS_FILE*)

  - writes a byte to file. Note can cause file to grow in size

- void **pdos_fclose**(PDOS_FILE *)

  - closes file, causes any buffers to be written to disk

- char ****pdos_dir**()

  - Returns a list containing the contents (files) in current directory. Last element is NULL

- void **pdos_mkdir**(char *dir)

  Creates a new sub directory of the root directory

User Commands

- **pdos_mkdisk**
  - **a wrapper program that just calls your pdos_mkdisk function.**
- **pdos_mkfs**
  - **a wrapper program that just calls your pdos_mkfs function.**
- **pdos_dir** *directory_name*
  - **list contents of directory. default directory is *"/", the root directory***

1. Create a static library for your file system.
2. write a set of small C programs – linked with your library – that executes each function
   a. one that creates a disk file system
      - show it is the correct size

   b. one that formats and initializes the disk by writing the FS structure
      - super block (with identifying string)
      - inode-bitmap block
      - data-bitmap block
      - update inode-block (entry zero) for root Directory
      - one data-block with appropriate information for root directory

        1 should have two entries

        2 "." And ".."

   c. one that uses the above FS and creates a test file(file1.txt) in the root directory
      - writes out **1023** bytes of A-Z repeatedly

d. one that reads in the data from above file and prints it out ensuring it matches

e. Repeat (c,d), this time create a second file file2.txt

f. Create a new test file that

- opens file1.txt and writes an additional 1024 bytes. (this should case file1 to expand and consume 2-data blocks)
- test to ensure file1.txt and file2.txt remain intact and the file-system has not been corrupted.

g. Repeat (f), but this time for file2.txt

h. Print out the contents of file1.txt and file2.txt

i. one that does directory listing of the root directory

Short and concise design document 1-2 pages that describes your design.
Well commented code

Example output showing typical results to

follow.

NOTES:

shm_open()/shm_unlink()	used to create file-backed shared memory. File is created in /dev/shm directory. Must link with **-lrt**
ftruncate()	used to set size of file-backed shared memory.
mmap()/mumap()	convenient means of mapping a file into RAM
ar	used to create a library (an archive)

ranlib	generates an index for archive
hexdump –C FS	hex dump of file system. Shows content of FS

- **You can only write or read a single BLOCK** of information to the pseudo file system. Remember you are emulating a disk!
- The file-backed shared memory (which is our File System) is persistent unless:
    - you reboot your machine
    - you unlink file
- Your C test programs cannot make any assumptions about the state of the file system. Further, multiple processes could access the file system. However, you are not required to provide synchronization framework to guard against race conditions.
- You must create a static library that provides the functions for your FS. Much the same way that a regular C program links against “libc” to access fopen()/fwrite()/fread() etc.

Examples showing possible output to follow: