Operating Systems - Assignment 5.

File System Simulator

General Overview

The main goal of this assignment is to clarify most of the relevant mechanisms that are part of the implementation of a file system. These include structures, naming, access, use, protection, etc. The assignment is based on the UNIX file system organization. The central goal is to model a file system on a local machine and to allow users access to that file system.

The system you will construct will provide one local file system for all users. To be more specific, the program will service only one user at the same time

Simplifications:

The tree of files is simplified and includes up to four levels.
"/" (**root**) has only directories of users. Each user has up to two levels of directories the **higher** can contain files and directories and the lower can contain only files.

/ **(root)**
/user1
/<file1>
/<file2>
/<file3>
/user2
/user3
/user4

A file-name is always **ABSOLUTE**, e.g. **user1/file1**. You can have your own restrictions for file-name valid characters and length. A directory is a file whose data is a sequence of entries, each consisting of an **i-node** number and the **name of a file** contained in the directory.

How to Run the Simulator - External Interface

The user interface of the file system simulator is by a simple method that reads from the standard input commands of the following formats and simulates them. First the file system is initialized, creating the simulated disk, which includes the disk and its resources. This part has to simulate the persistence of a file system, i.e. after the 1st initialization it remains permanently on **FILE_SYS** (standard name for the file) and can be accessed at any time

After initialization, each user that runs the program issues first a login command with a specific **user ID**, then a list of commands. The program performs each command and displays the results one by one on the standard output. The last command of the user must be a logout.
The file system is on **FILE_SYS** during all of the user's sessions. The general sequence for a user session is:

login ID

commands

logout

Important Simplification:

The fact that the simulator supports only one user at the same time, avoids the need for a global open-file table. There will be only one open file table in the simulator's system. The entry number of the open file table will be returned as a file descriptor (like a user process table). The **i-node** number of the file will be stored in another entry of the same table. The user open file table will have multiple entries of the same file, if it was opened multiple times.

The commands for a single user session are:

- **login <user-id>** - Start the user session. The user-id (an integer) is used to identify the concrete user by the file system.
- **logout** - End the user session.
- **mkdir <dir-name> -** Make a directory with a given name.
- **create <file-**name> - create an empty file. Returns a file descriptor entry <filenum>.
- **open <file-name> < access-mode > -** Open file with a given path. The second parameter is 0 – read, 1 - write, 2 - read-write. Default access-mode for a new file is ReadWrite for owner only. Returns a file descriptor entry <filenum>.
- **close <filenum>** - Close the file that is identified by filenum
- **read <filenum> <num-bytes>** - Read num-bytes from the file that is identified by filenum to standard output. Returns the number of files that were read.
- **write <filenum> <string>** - Write string to the file. Returns the number of bytes that were written (the written bytes are added at the end of the existing file).
- **seek <filenum> <location> -** Change current file position to location that is measured in bytes from the beginning of the file. Return –1 if the file is smaller then 'location'.
- **rm <file> -** Remove a file with the given name
- **rmdir <dir-name> -** Remove the given directory. The directory must be empty.
- **ls -** List the current working directory. This should give the following attributes file-owner, size, i-node, file-name

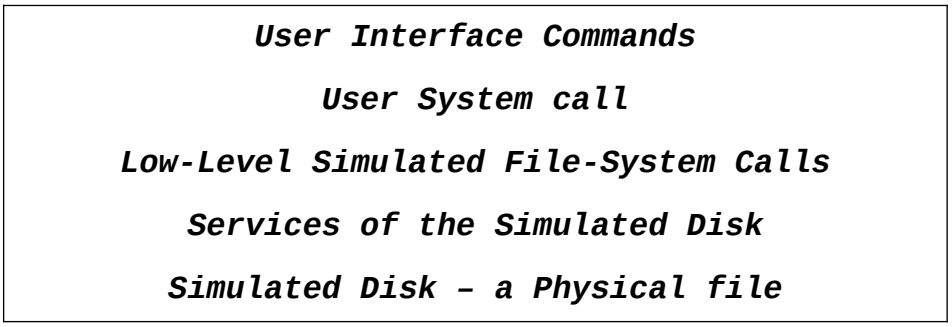
  copy <path1> <path2> - copy the file or directory represented in path1 to the directory represented in path2. 'path1' can be any existing path in the system. 'path2' must be an existing path of a directory, that begins with the home directory of the calling user.

  import - <global path> <path> - copy the file from the Unix-file system with the global path to the path in the simulator file system.

NOTICE: All system-calls that a return value was not specified for them return 1 if the call succeeds and 0 otherwise of course the returned value has no meaning when the system call is called from the shell .

System Implementation:

The following is a layer scheme of the simulator without the backup file system:



Each Layer uses the services of the layer under it!!!

The File System

The file system consists of a simulator that is part of the system level. Users access data in a file by a byte (char) offset. All operations use byte counts and view a file as a stream of bytes starting at byte address 0 and going up to the size of the file. The system level converts the user view of bytes into a view of blocks: The file starts at logical block 0 and continues to a logical block number that corresponds to file size. The "*Services of the Simulated Disk*" layer accesses the i-node and converts the logical file block into the appropriate disk block. The *Disk-Physical file* is the collection of blocks that are accessed.

The File System level is composed of 2 parts:

1. The simulated disk, which includes the disk and its resources. This part has to simulate the persistence of a file system, i.e. after the 1st initialization it remains permanently on FILE_SYS (standard name for the file) and can be accessed at any time
2. The low-level system calls layer, which uses the simulated disk and displays a logic structure of a file system to the user. It includes the i-nodes table, maintaince of free blocks, etc.

The Simulated Disk

The entire simulated disk is contained in one Unix file called: FILE_SYS. Like a real disk FILE_SYS is divided into blocks, each is 1K in size, and each is addressed by its physical number. FILE_SYS is composed of 5 parts (first three parts in the super-block):

1. A superblock which is physical blocks 0-4 for the whole disk, containing:
    1. The size of the file system (in blocks)
    2. Blocks Control:
        1. The number of free blocks in the file system
        2. A limited size bit-map (at most 4 blocks) of free blocks numbers. (Note 4 blocks byte-map give you access to maximum total size of file system: 4Mbytes).

3. The index of the next free block in the bit-map

(Your implementation for 2,3 will be a bitmap. This is not a requirement and can be implemented with byte-map)

3. I-Node Control
    1. The size of the i-node array
    2. The number of free i-nodes in the file system
    3. A limited size bit-map of free i-nodes
    4. The index of the next free i-node in the free i-node list

A bitmap implementation for 3,4. This is not a requirement and can be implemented with byte-map)

Note, the main simplification here is that all the allocation/freeing of both data-blocks and i-nodes is done within the Super-block only!

The rest of the disk is composed of the two parts below:

4. An array of i-nodes (the i-nodes table)
5. An array of blocks
6. Optional: A flag indicating whether the super block has been modified.

Limitations and Fixed sizes:

1. The disk size is 4Mbytes.
2. Maximal number of files in the disk is 1024 files.
3. The size of addresses of blocks is 2 bytes.
4. An i-node includes 10 direct block numbers, 1 single indirect and 1 double indirect.
5. The size of the i-node array is 1050 blocks.

The overview of the disk/file-system is an array of blocks and looks like this:

Super block-s	***The I-nodes array***	***The blocks array***
Size of the file system Blocks Control I-node Control	2-bytes per block number (note, even though an inode takes only 64 bytes, it is allocated in complete block)	(rest of disk…)

Note, for simplification it will be possible to use Byte-map instead of bit-map.

In case you are implementing a byte map the first four blocks after the super-block (blocks 1 – 4) will be used for the byte-map and the i-node array will start after them (in block 5).

Services of the Simulated Disk

The disk layer must include the following services:

- l_initialize() - Initialize the disk by freeing all blocks and i-nodes
- i_alloc() - Allocate an i-node, returning its number
- i_free(i-node number) - Free an i-node + all the blocks associated with it
- i_read(i-node number, buffer) - Read the specified i-node into the buffer
- i_write(i-node number, buffer) - Write the i-node from the buffer
- b_alloc() - Allocate a block, returning its number
- b_free(block number) - Free that block
- b_read(block number, buffer) - Read the specified block into the buffer
- b_write(block number, buffer) - Write that block from the buffer

Every service prints a message to the stdout when used!!!

Low-Level Simulated File-System Calls

Low-level calls use i-node number instead of the file-descriptor of user-level calls. The i-node is stored in the open-file table of the local client in response to the user commands. We list below the system calls used in translation to user level calls (in the next section):

- u_initialize() – a system call used to trigger the lower level initialization service and construct the root directory.
- f_create(path, access-mode) - Creates a file and returns an i-node number for it. Error is returned if the parent directory does not exist, or there is no write permission, or the path leads to an existing file.
- d_create(path, access-mode) - Creates a directory and returns an i-node number. Error is returned if the parent directory does not exist, or there is no write permission, or the path leads to an existing directory.
- f_read(i-node, buffer, n_offset, number_of_bytes) - Reads number_of_bytes bytes from file of i-node number, into the specified buffer. Returns the number of bytes actually read.
- f_write(i-node, buffer, n_offset, number_of_bytes) - Writes number_of_bytes bytes from the specified buffer, into the file in i-node number. Returns the number of bytes, which were actually written
- d_read(i-node, buffer) - Reads a complete directory into the buffer
- d_write(i-node, buffer) - Writes a complete directory from the buffer

- f_delete(i_node) - Deletes that i-node. Error is returned if the i-node does not exist or if it is associated with a directory, which is not empty or if the directory or any of its descendents is open on a client.

The program can be run multiple times, each time it should be able to continue as if it never stopped. That means that the initialization services may not be called every running session. Otherwise it will erase the entire disk (external file) and will be unable to recover. The initialization locates the disk file and recovers the last state of the file system. If FILE_SYS does not exist, this means that the program is running for the first time.

The System Calls Layer / Backup Commands

The list of required **user** system calls is given below in the form of a C++ class.

```
class SCalls
{
public:
      SCalls (int UserID);
      ~ SCalls ();

      int ChMod(int Mod, char* file-name); //changes the 'file-name' mode to Mod.
      int MakeDir(char * dir-name); //creates a new directory named 'dir-name'.
      int ChDir(char * dir-name); //changes the location directory to 'dir-name'
      int RmDir(char * dir-name); //removes the directory 'dir-name'
      int RmFile(char * file-name); //removes the file 'file-name'
      int ls(dir-name); //lists the content of the location-directory.
      int Open(char * file-name, int Mode); //open the file 'file-name' in the given
                                            // access mode.
      int Close(int fileID); //close the entry of the file in the file table.
      int Seek(int fileID, int location); //move the file curser to 'location'.
      int Read ( int fileID, int nBytes, char * Buffer);//read nBytes from the received
                                                        // fileID's file into the buffer.
      int Write ( int fileID, int nBytes,char * Buffer);// write nBytes from buffer
                                                        // into the received fileID's file.
private:
      int UserID;
. . .
}
```

The Caching feature

Reading and writing to and from the disk should be done through a cache buffer.

The cache buffer size will be three blocks. You can choose the switching algorithm.

You must implement a dirty-bit mechanism.

Output in Monitoring Mode

You must provide a monitoring mode in your system. In that mode all the calls to the various layers must be shown,)e,g, a command such as:

Read(...)

May cause a sequence of calls, first to the high-level file system, then to the cache, then to the low-level file system, then to the simulated disk...

You need to add to the interface two commands **monitor** and **no monitor** that will allow us to activate the monitor mode
which will generate notices for each call of a function.

Bonus:

The bonus is divided in to two parts:

- Protection Modes**(2 points)**:

  A mode for a file or a directory which specifies permissions for the owner and others. Two bits for the owner and two bits for the other (0,1,2,3 – nothing, read, write, all)

  example:

  chmod 31 example.c

  all permissions for the user, read permissions for other

  The root directory has permission mode 33, so any user can create a subdirectory of root. The way to implement a set of valid users is under your discretion.

  chmod <mode> <file-name> - changes the files mode to the received mode.

  chmod command will allow the owner to control the protection level of his files and directories.

  NOTE: If you wish to make your directory accessible to others, you must give others appropriate permissions for the directory.

- An extension on the File System including**(3 points)**:
  - Implementing a 6-block cache with an efficient LRU switching algorithm based on a linked-list and/or a hash-table.
  - Unbounded path depth.
  - Relative path and a ‘cd’ user system call to pass between directories.
  - Hard and soft links that behave exactly like in Unix.

You can implement the first without the second **BUT NOT THE SECOND WITHOUT THE FIRST!!!**

Good Luck.

:c)