CMPUT 379 – Operating System Concepts **Fall 2019**

Assignment 3 – A Trivial UNIX File System

This assignment will be due **December 6, 2019 at 11:59pm** Mountain Time.

Objective

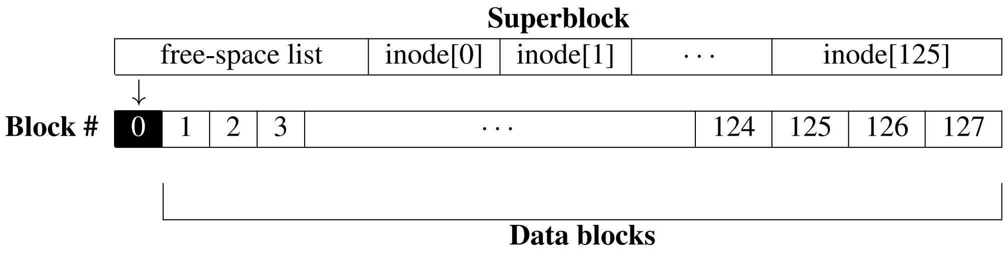
This programming assignment is intended to give you experience in writing a simple UNIX-like file system.

The UNIX-like File System

A file system stores data files persistently on disk, and provides a unified way to access and retrieve these files. It keeps a fair amount of information about each file in a data structure which is essential to isolate and identify data. In this assignment, you will write a program in C or C++ (whichever you prefer) which simulates a trivial file system. This program mounts a file system that resides on a virtual disk (i.e., a file acting as the disk) and checks if it is consistent. It subsequently reads commands from an input file and executes them by simulating the file system operations.

Disk Layout

The file system you will develop in this assignment resides on a disk that is 128 KB in size. The size of each block is 1 KB (1024 bytes). The first block is called the *superblock* and contains the list of free blocks (*free-space list*) and 126 index nodes (*inodes*). The remaining 127 blocks store data blocks of the files on your file system. A file consists of one or multiple blocks, hence its size can only be multiples of a kilobyte. Users can create or delete files and directories, and arbitrarily change the size of a file provided that the disk or the inodes are not full.



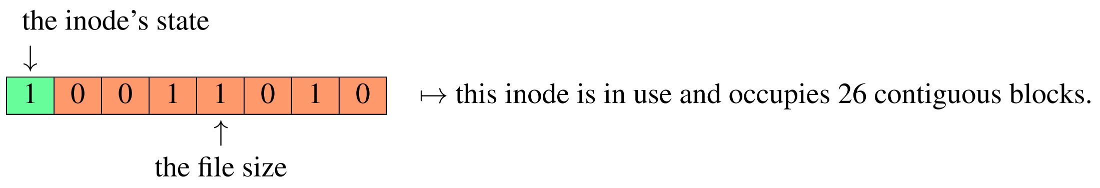
The free-space list is implemented as a bit vector occupying the first 128 bits (16 bytes) in the superblock. Each block is represented by a bit which indicates whether the corresponding block is free or not; 0 indicates that the block is free, and 1 indicates that the block is not available to use. The first bit represents the state of the first block, i.e., the superblock, and the last bit represents the state of the last data block.

Each inode contains information about a file or a directory. In this file system, an inode contains six pieces of information, namely the name of the file (or directory), the size of the file, the index of the first block allocated to the file, the state of the inode, the type of the inode, and the index of the inode of the parent directory. The inode structure is defined below:

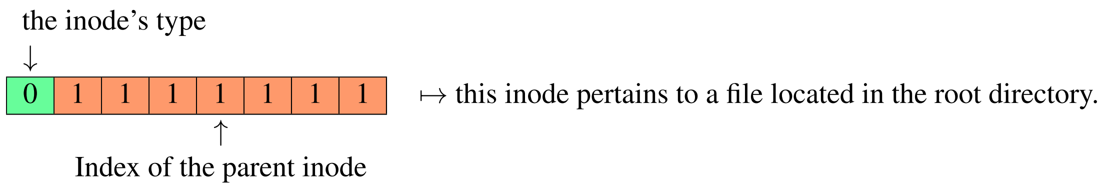
```c
typedef struct {
    char name[5];          // Name of the file/directory, not null terminated
    uint8_t used_size;     // Inode's state and size of the file/directory
    uint8_t start_block;   // Index of the start file block
    uint8_t dir_parent;    // Inode's type and index of the parent inode
} Inode;
```

We describe the file attributes stored in this structure below:

- **`name`**: represents the name of the file or directory, limited to 5 characters only. Hence, it is possible to have a name like `"image"`, which leaves no space for the `'\0'`, although filenames under 5 characters must end with a null terminator `'\0'`. Moreover, the name of the file must never start or end with a white space, and all leading and ending spaces must be ignored (for example, `"  docs  "` should be changed to `"docs"` which is a valid name).
- **`used_size`**: represents the state of the inode followed by the size of the file. The first bit of this variable indicates the state of the inode, where 1 means the current inode is in use and 0 means it is free. The other 7 bits represent the file size (in blocks). If the inode pertains to a directory, the size would be zero.



- **`start_block`**: represents the index of the first block on the disk allocated to the file. In this file system, each file must be allocated a number of *contiguous blocks*. If the inode pertains to a directory, then this variable would be zero.
- **`dir_parent`**: represents two pieces of information, the inode's type and parent. The first bit of this variable indicates the type of the inode, where 1 means that it pertains to a directory and 0 means that it pertains to a file. The other 7 bits indicate the index of the parent inode in the superblock (which can be between 0 and 125 inclusive). The index is set to 127 for files and directories located in the root directory. Note that the index cannot be set to 126.


the inode's type
0 1 1 1 1 1 1 1
↦ this inode pertains to a file located in the root directory.
Index of the parent inode

Simulating Basic Operations

In this assignment, you will simulate a number of basic operations on your file system. Your program must load the superblock into the memory without loading the file blocks, unless it is necessary. You will have to write the superblock and the file blocks back to the file that emulates the disk immediately after running a command.

Your program must read commands from a file and perform the corresponding file system operations (described in the next section). If an undefined command is detected in the input file, it should print an error message to `stderr` with the name of the input file and the line number where the error occurred (see the example below), then continue running the command in the next line. The format of the error message is:

```
Command Error: <input file name>, <line number>
```

For example

```
Command Error: command_file, 10
```

suggests that the command in Line 10 of the input file `command_file` is not recognized or cannot be parsed. More specifically, this error message must be printed in the following cases: invalid command, wrong number of arguments passed to the command, unexpected arguments (e.g., when the length of the filename is greater than 5), and invalid index of the block number (when it is outside [1, 127]).

The file system will use a *buffer* of 1 KB (1024 bytes) to hold data that is read from or must be written to a block within a file. This buffer needs to be initialized to zero when your program starts. The program must also keep track of the current working directory.

The commands that may appear in the input file are as follows, noting that all filenames must be found in the current working directory:

- `M` - Mount the file system residing on the disk (results in the invocation of `fs_mount`)
  Usage: `M <disk name>`
  Description: Mounts the file system for the given virtual disk (file) and sets the current working directory to root.
- `C` - Create file (results in the invocation of `fs_create`)
  Usage: `C <file name> <file size>`
  Description: Creates a new file with the provided name in the current working directory.
- `D` - Delete file (results in the invocation of `fs_delete`)
  Usage: `D <file name>`
  Description: Deletes the specified file from the current working directory.
- `R` - Read file (results in the invocation of `fs_read`)
  Usage: `R <file name> <block number>`
  Description: Reads the block number-th block from the specified file into the buffer.
- `W` - Write file (results in the invocation of `fs_write`)
  Usage: `W <file name> <block number>`
  Description: Writes the data in the buffer to the block number-th block of the specified file.
- `B` - Update buffer (results in the invocation of `fs_buff`)
  Usage: `B <new buffer characters>`
  Description: Updates the buffer with the provided characters. Up to 1024 characters can be provided. If fewer characters are provided, the remaining bytes (at the end of the buffer) are set to 0.
- `L` - List files (results in the invocation of `fs_ls`)
  Usage: `L`
  Description: Lists all the files/directories in the current working directory, including `.` and `..`
- `E` - Change files size (results in the invocation of `fs_resize`)
  Usage: `E <file name> <new size>`
  Description: Changes the size of the given file. If the file size is reduced, the extra blocks must be deleted (zeroed out).
- `O` - Defragment the disk (results in the invocation of `fs_defrag`)
  Usage: `O`
  Description: Defragments the disk, moving used blocks toward the superblock while maintaining the file data. As a result of performing defragmentation, contiguous free blocks can be created.
- `Y` - Change the current working directory (results in the invocation of `fs_cd`)
  Usage: `Y <directory name>`
  Description: Updates the current working directory to the provided directory. This new directory can be either a subdirectory in the current working directory or the parent of the current working directory.

File System Implementation

You will implement the following functions to handle the commands described in the previous section. Errors must be checked and reported in the same order as they are listed for each function.

- **`void fs_mount(char *name)`**

  Mounts the file system residing on the virtual disk with the specified name. The mounting process involves loading the superblock of the file system, but before doing this, you should check if there exists a file (i.e., a virtual disk) with the given name in the current working directory. Print the following error to `stderr` if this file does not exist:

  ```
  Error: Cannot find disk <disk name>
  ```

  The next step is to check the consistency of the file system. We say a file system is inconsistent when an arbitrary number of bits in its superblock are changed accidentally. Your program should read through the file system and perform the following consistency checks:

  1. Blocks that are marked free in the free-space list cannot be allocated to any file. Similarly, blocks marked in use in the free-space list must be allocated to *exactly* one file.
  2. The name of every file/directory must be unique in each directory.
  3. If the state of an inode is free, all bits in this inode must be zero. Otherwise, the `name` attribute stored in the inode must have at least one bit that is not zero.
  4. The `start_block` of every inode that is marked as a file must have a value between 1 and 127 inclusive.
  5. The `size` and `start_block` of an inode that is marked as a directory must be zero.
  6. For every inode, the index of its parent inode cannot be 126. Moreover, if the index of the parent inode is between 0 and 125 inclusive, then the parent inode must be in use and marked as a directory.

  If the file system is inconsistent, you must print the following error to `stderr`:

  ```
  Error: File system in <disk name> is inconsistent (error code: <number>)
  ```

  where `<number>` identifies the type of inconsistency (as listed above) and can be from 1 to 6. Note that you must check the file system inconsistency in the same order as above, and only report the first inconsistency you detect. Hence, if a file system is inconsistent for several different reasons, you only print the smallest error code.

  Your program should not mount a file system that fails the consistency check. In this case, you should continue to use the last file system that was successfully mounted for running the next commands. If no file system was successfully mounted before, you should print the following error to `stderr` for each command that comes after the `mount` command until you read another `mount` command that can mount a file system successfully:

  ```
  Error: No file system is mounted
  ```

  Note that you should report this error only if the command is valid but no file system is mounted. If the command is not valid, you should just print the command error (described earlier).

  If the disk exists and the residing file system is consistent, you can proceed to loading the superblock and set the current working directory to the root directory. Do not flush the buffer when mounting a file system.

- **`void fs_create(char name[5], int size)`**

  Creates a new file or directory in the current working directory with the given name and the given number of blocks, and stores the attributes in the first available inode. A size of zero means that the user is creating a directory. If no inode is available, you must print the following error to `stderr`:

```
Error: Superblock in disk <disk name> is full, cannot create <file name>
```

You must make sure the new file or directory have a unique name within the current working directory, and the name cannot be `.` or `..` which is reserved. You can have multiple files with the same name only if they are located in different directories. If there already exists a file or directory with the same name under the current working directory, then print the following error to `stderr`:

```
Error: File or directory <file name> already exists
```

Recall that the file must be allocated a number of contiguous blocks, so your program must find the first set of contiguous blocks that can be allocated to the file by scanning data blocks from 1 to 127. Print the following error to `stderr` if there is not enough contiguous free blocks to allocate to the new file:

```
Error: Cannot allocate <file size> on <disk name>
```

You should check the availability of a free inode, then the uniqueness of the file/directory name, and finally the availability of enough contiguous free blocks to fit the new file's data blocks.

- **`void fs_delete(char name[5])`**

  Deletes the file or directory with the given name in the current working directory. If the name represents a directory, your program should recursively delete all files and directories within this directory. For every file or directory that is deleted, you must zero out the corresponding inode and data block. Do not shift other inodes or file data blocks after deletion. If the specified file or directory is not found in the current working directory, print the following error to `stderr`:

  ```
  Error: File or directory <file name> does not exist
  ```

- **`void fs_read(char name[5], int block_num)`**

  Opens the file with the given name and reads the `block_num`-th block of the file into the buffer. If no such file exists or the given name corresponds to a directory under the current working directory, the following error must be printed to `stderr`:

  ```
  Error: File <file name> does not exist
  ```

  If the `block_num` is not in the range of [0, `size`-1], where `size` is the number of blocks allocated to the file, print the following error to `stderr`:

  ```
  Error: <file name> does not have block <block_num>
  ```

- **`void fs_write(char name[5], int block_num)`**

  Opens the file with the given name and writes the content of the buffer to the `block_num`-th block of the file. If no such file exists or the given name corresponds to a directory under the current working directory, the following error must be printed to `stderr`:

  ```
  Error: File <file name> does not exist
  ```

  If the `block_num` is not in the range of [0, `size`-1], where `size` is the number of blocks allocated to the file, print the following error to `stderr`:

  ```
  Error: <file name> does not have block <block_num>
  ```

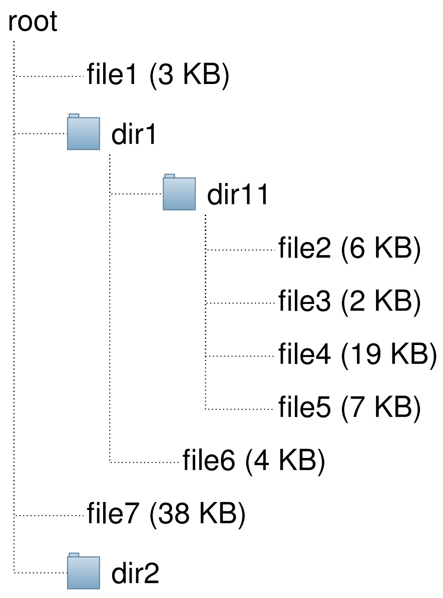
- **`void fs_buff(uint8_t buff[1024])`**

  Flushes the buffer by setting it to zero and writes the new bytes into the buffer. No errors must be handled in this function.

- **`void fs_ls(void)`**

  Lists all files and directories that exist in the current directory, including special directories `.` and `..` which represent the current working directory and the parent directory of the current working directory, respectively. You must print the result to `stdout`. Always print `.` and `..` as the first and second rows of the result, respectively, and then all files and directories according to the indices of their corresponding inodes. For files, you must print the size of the file as well. For directories, you must show the number of files and directories that exist in this directory (do not do this recursively). If the current directory is the root directory of the virtual disk, then `..` and `.` both represent the current directory.

Suppose the disk has the following structure:



If the current working directory is `root`, then your program should output:

```
.        6
..       6
file1    3 KB
dir1     4
file7   38 KB
dir2     2
```

where each output row is formatted as follows if it is for a file:

```
printf("%-5s %3d KB\n", name, size);
```

and as follows if it is for a directory:

```
printf("%-5s %3d\n", name, num_of_children);
```

In the above example, `dir1` contains 4 items because there are 3 directories, namely `dir11`, ., . ., and 1 file which is `file6`. If the current working directory is `dir1`, then your program should output:

```
.        4
..       6
dir11    6
file6    4 KB
```

You do not need to handle any errors in this function.

- **`void fs_resize(char name[5], int new_size)`**

  Changes the size of the file with the given name to `new_size`. If no such file exists in the current working directory or the name corresponds to a directory rather than a file, your program should print the following error message to `stderr`:

  ```
  Error: File <file name> does not exist
  ```

If the `new_size` is greater than the current `size` of the file, you need to allocated more blocks to this file. You must keep the `start_block` fixed and add new data blocks to the end such that the file's data blocks are still contiguous. If there are enough free blocks after the last block of this file, change the `size` in the inode to `new_size`. Otherwise, you must try to move the file so that you can increase the file size to the new size, and copy all data in the previous blocks to the new blocks. The new starting block must be at the first available one that has enough space for the resized file. If there are not enough contiguous free blocks on the disk to fit the file with its new size, print the following error to `stderr` and do not update the file size:

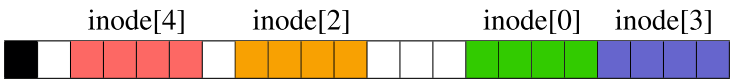
```
Error: File <file name> cannot expand to size <new size>
```

If the `new_size` is smaller than the current `size`, then delete and zero out blocks from the tail of the block sequence allocated to this file. The starting block is not moved when decreasing a files size. Finally, change the `size` attribute in the inode to the `new_size`.

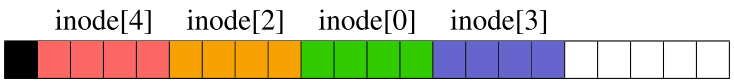
- **`void fs_defrag(void)`**

  Re-organizes the file blocks such that there is no free block between the used blocks, and between the superblock and the used blocks. To this end, starting with the file that has the smallest `start_block`, the `start_block` of every file must be moved over to the smallest numbered data block that is free.

  For example, if the blocks are in this state before defragmentation:



  they will be in the following state after this operation:



  When moving blocks, the data must be copied over to the new blocks. Hence, files will contain the same data as before. You do not need to handle any errors in this function.

- **`void fs_cd(char name[5])`**

  Changes the current working directory to a directory with the specified name in the current working directory. This directory can be `.`, `..`, or any directory the user created on the disk. If the specified directory does not exist in the current working directory (i.e., the name does not exist or it points to a file rather than a directory), print the following error message to `stderr`:

```
Error: Directory <directory name> does not exist
```

  You can assume that the given name has no slash at the end.

Starter Code

Download `a3-starter-code.zip` from eClass; it contains the starter code for this assignment. The starter code includes a program to create and format a virtual disk, a header file, and a small number of input files with the expected `stdout` and `stderr` outputs and resulting disk images. You might modify the header file provided with the starter code.

To create a file that emulates the disk and format it, run the following command:

```
$ ./create_fs disk0
```

This command will create a 128 KB file in your current directory which acts as the disk for your file system. The name of this file is `disk0` in the above example. This program also "formats" your disk by initializing the superblock and inodes.

Note that your file system must store data persistently on disk, making sure that when you request the data again, you get what you put there in the first place. Thus, if you terminate your program and then run it at a later time, all files in your file system must be intact.

Input file

Your program should read commands from an input file, run them one at a time, printing out the result of each command. The input file follows a strict format: each line contains a command and a number of arguments.

The name of this input file is passed to your program, `fs`, as a command line argument:

```
$ ./fs <input_file>
```

Deliverables

Submit your assignment as a single compressed archive file (`fs-sim.zip` or `fs-sim.tar.gz`) containing:

1. A custom Makefile with at least these targets:
   (a) the main target `fs` which simply links all object files and creates an executable file called `fs`.
   (b) the target `compile` which compiles your code and produces the object file(s)
   (c) the target `clean` which removes the objects and executable files
   (d) the target `compress` which creates the compressed archive for submission, this should not compress and binary (executable or object) files
2. A plain text or markdown document, called `readme.md`, which explains your design choices, lists the system calls you used to implement each function, and elaborates on how you tested your implementation. Make sure you cite all sources that contributed to your assignment in this file. You may use a markup language, such as Markdown, to format this plain text file.
3. All files required to build your project including the starter code.

Misc. Notes

- This assignment must be completed **individually** without consultation with anyone besides the instructor and TAs.
- You can write the file system simulator program in C or C++. You must compile the program using `gcc` or `g++`. No warnings should be returned when your code is compiled with `-Wall` flags. Also your code must not print anything extra for debugging.
- Check your code for memory leaks. You may use the Valgrind tool suite:

  ```
  valgrind --tool=memcheck --leak-check=yes ./fs
  ```
- You can use your own machine to write the code, but you must make sure that it compiles and runs on the Linux lab machines (e.g., ugXX.cs.ualberta.ca where XX is a number between 00 and 34).
- When developing and testing your program, make sure that you clean up all processes before you logout of a workstation. You can use the `ps` command to list the processes you created and the `kill` command to terminate them.