CS3841- Fun with File Systems

1 Background

A mass storage device such as a disk drive, solid state drive, or USB "stick" can be considered a large contiguous block of storage like memory. Like memory, mass storage devices are accessed via an address, however mass storage devices address are for 'blocks' not bytes. A read from a block device reads the entire block and a write writes an entire block. For mass storage devices, a block is typically 512 bytes. The OS creates an abstraction of 'files' and 'directories' (folder) over the top of mass storage devices to give the user (and applications) the appearance of a file system. A contemporary operating system such as Linux contains drivers for many different file systems, with many being quite complex.

In this lab you will experiment with creating different file systems in Linux as well as interpret a file system architecture for a brand-new file system, teeny tiny file system (TTFS).

In Linux the file system is organized as a tree of files and directories starting at the root directory (/). The "mount" command allows a system administrator to hang (mount) file systems from a branch of the tree. Kind of like how you might hang an ornament on a tree during the holidays. To mount a file system, you first need to create what is known as a "mount point" i.e. an empty directory that will serve as the entry point for the file system.

2 Create and Mount a RAM disk

A RAM disk creates a functional file system directly in random access memory (RAM). Reads and writes to this file system will be an order of magnitude faster than mass storage device operations. This fast access speed is a huge advantage.

What are the drawbacks? 1) Size – all file systems are limited based on the size of their backing media. For a RAM disk that backing media is RAM which is on the order of a few gigabytes. 2) Persistence – When the system is shutdown or rebooted, the data stored in RAM is lost. For a RAM disk this means that any files stored the file system will also be lost.

Why use a RAM disk then? Temporary files – many applications use short lived files, either for inter process communication, storing intermediate results, or logging status. For example, gcc runs through several phases to build an executable program: 1) running the processor, 2) generating assembly, 3) building object code 4) linking the final executable. The output from each of these phases is one or more temporary files that are ultimately thrown away when the final executable is built. Storing these intermediate files in a RAM disk may be able to increase compile speeds. RAM disks are also used during bootup to place a minimal system in memory which can then start loading the actual system.

To start, see how much free memory you have.

```
user@pc:~$ free -h
              total        used        free      shared  buff/cache   available
Mem:           3.9G        263M        3.1G        7.8M        469M        3.4G
Swap:          4.3G          0B        4.3G
```

You should see two rows in the output. One for physical memory and one for swap (the disk space reserved for the kernel to use for 'kicking out' pages from memory).

For a description of what each field in the table means you can run: `man free`

There are two file system drivers that can be used for RAM disks, ramfs and tmpfs. Large tmpfs disks can actually be swapped out to physical disk, which sort of defeats the purpose, but it is a little more full-featured than ramfs.

Now, make a mount point for the RAM disk. /media is a common location for mount points. To do this you will need administrator (root) authority. The 'sudo' command allows you to temporarily escalate your system privileges to admin. You'll need to type your password to escalate your privileges.

```
user@pc:/$ sudo mkdir /media/ramdisk
user@pc:/$
```

Now we can mount the drive with the mount command and proper options. The following will create a 512 MB RAM disk with file system type tmpfs in /media/ramdisk. See `man mount` for full details.

```
user@pc:/$ sudo mount -t tmpfs -o size=512m tmpfs /media/ramdisk
user@pc:/$
```

The disk is now usable.

Because the disk was made transiently, it will not be recreated on a reboot. This is because Linux does not keep a record of file systems that were mounted when it shuts down. There is, however, a file that Linux uses determine which file systems to mount at boot up. This file is the file system table (fstab) and is locate in the /etc directory. You can have a look opening /etc/fstab or just use the "cat" command to print it out. You can have a RAM disk be created on every boot by adding an entry to /etc/fstab. If you do this, the ram disk will be there on every boot, but any files present will be gone.

To complete this activity, perform the following and record your answers to the questions:

1. View the man page for free with "man free". What does each column mean? In your own words, what is the difference between free memory and available memory?
2. View the man page for mount with "man mount". Use the information in the man page to record which file system types your Linux installation supports.
3. Pick one of the file systems supported by our Linux installation and learn more about how it works (use google to help). Give a brief summary of the file system. Document your resources.
4. Create the RAM disk as outlined above. What do each of the parameters to "mount" mean?
5. Record the output of `free -h` after creation of the RAM disk. What is different? Why?
6. Based on the observation of the free command before adding the ram disk, and after adding it and adding files to it, in which category of memory does the ram drive appear to reside?
7. Record the status of the RAM disk with the command `df -h`. What can you learn about the file system by running df?
8. Copy or download a large file into the RAM disk. I suggest a large pdf file (maybe around 25M). Interact with this file (such as opening a pdf with a viewer) in the RAM disk as well as a copy on the normal file system. What differences do you notice (e.g. performance, etc.)?
9. Record the output of `free -h` and `df -h` now that the RAM disk has a file in it. How does the output compare to what you recorded before? Does the output make sense? Why?
10. Record the contents /etc/fstab. What does each entry mean? Consult "man fstab" for help.
11. What would an entry in /etc/fstab need to look like in order to create the 512MB tmpfs RAM disk when the system boots?

When done, drive should be unmounted to tell Linux that it's no longer needed:

```
user@pc:/media$ sudo umount /media/ramdisk
user@pc:/media$
```

Remember, that since this is a RAM disk, unmounting the file system will destroy the contents of the file system.

3 Mount a File as a File System

Linux represents all block devices as a file in a special file system directory '/dev'. This directory contains a representation of devices on the system.

List the contents of '/dev' on your Linux VM, you should see several 'files' sdaX (X will be a number). These represent the disk partitions of the disk on your system. Partitioning a disk is a way to create logical sub-disks that can each contain a file system that Linux can mount into the directory tree. The 'file' in '/dev' will contain the binary representation of the file system.

The entire disk (all partitions) is represented by the file '/dev/sda'. View the partitions created on the disk by running the following:

```
user@pc:/$ sudo fdisk -l /dev/sda
```

Run the following to view the binary data by reading from the file:

```
user@pc:/$ sudo hexdump -C /dev/sda1
```

This will print out the contents of the device '/dev/sda1' in hexadecimal.

Run 'mount' (with no arguments) to see all the file systems currently mounted on your Linux VM. Several of them are RAM disks used for temporary file systems. You should be able to see where your sdaX partitions are mounted.

While a file system will typically be backed by a physical device represented in '/dev', it doesn't have to be. A file system can also be created using an ordinary file. This could be used to create an image to be written to an optical disk or a removable flash drive, or to simply explore file systems without having to install and partition a physical hard drive.

Run the following to create a FAT16 image file with 5000 blocks (a block is 1024 bytes), and then mount it.

```
user@pc:~$ sudo mkdir /media/myimage
user@pc:~$ sudo mkdosfs -C -s 1 -S 512 -F 16 flash.img 5000
mkfs.fat 4.1 (2017-01-24)
user@pc:~$ sudo mount flash.img /media/myimage
user@pc:~$ cd /media/myimage/
user@pc:/media/myimage$ ls -al
total 20
drwxr-xr-x 2 root 0 16384 Dec 31  1969 .
drwxr-xr-x 8 root 0  4096 Nov  4 19:21 ..
user@pc:/media/myimage$
```

When doing this, also look at the actual file flash.img and its size. As with the previous activity try creating and/or copying files onto the drive. Hint, the default owner is root, so your user may not have access. Figure out a way to modify the owner of the mounted drive. Look at the size of the image file after adding files to the drive.

Specifically, add four text files, and delete two of them. Also create at least one file with a long filename (longer that 8 characters).

As before, umount when done.

1. Where are the mounted partitions of '/dev/sda1' mounted? Why do suspect that two partitions are used? What is an advantage of partitioning the disk in this way?
2. Look at the image file with hexdump (`hexdump –C flash.img | more`). Can you find your files? How about the ones you deleted? How did you find them or why couldn't you?

3. Show a capture of the hexdump of the directory entries for the files you added to the image as well as the ones you added and deleted. What is different?
4. Show a capture of the hexdump of a portion of the file contents within the image of a file you deleted. What do you observe?
5. Change directory to the directory that contains the file. Try to unmount the file system with "umount /media/myimage". Did it work successfully? Why or why not?

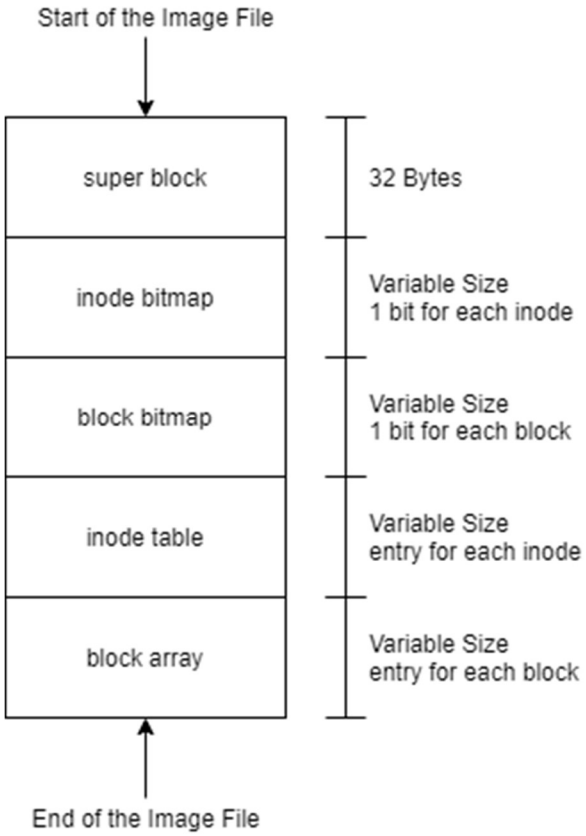
4 Interpreting a File System - TTFS

Linux supports many different file system types. Each has their own intended use along with advantages and disadvantages. New file systems are being developed all the time to fit the needs of newer backing storage and application purposes. In this activity you will be given the format for a new file system (teeny tiny file system – TTFS). Study the file system format and use the information to interpret a binary file containing a TTFS file system image.

TTFS stores files and directories of varying sizes. Each file is allocated one or more blocks of data. An index node (inode) stores information about a file or directory in the file system. A TTFS file system image is a binary file divided into the following regions:

1. File system super block – contains information about the entire file system
2. inode bitmap – a bit map that indicates which inodes are free and which are used to store file information
3. block bitmap – a bit map that indicates which blocks are free and which hare used to store file data
4. inode table – an array of inodes for files and directories in the file system
5. block array – the array of data blocks for files and directories in the file system

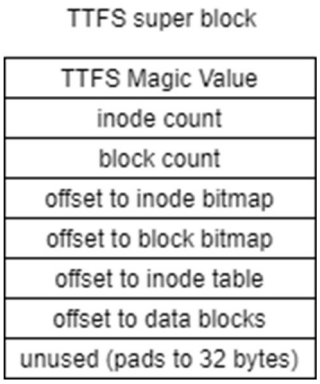
An image for TTFS would look like this:



Regions of the file system follow immediately after each other. The super block contains offset values that can be used to find where regions begin. Data blocks are 512 bytes long and are accessed via their block index. The block index is zero based so the first block is block 0, the second is block 1, etc.

4.1 TTFS Super Block

Each field in the super block is 4 bytes. The offset numbers indicate where the region begins from the ***start of the file system image***. Here is a diagram of each of the fields contained in the super block



- TTFS Magic Value – an eye catcher – must contain the string value "TTFS"
- inode count – The total number of index nodes in the file system (used or unused)
- block count – the total number of blocks in the file system (used or not)
- offset to inode bitmap – the offset (in bytes) from the start of the file system image to the start of the inode bitmap
- offset to block bitmap – the offset (in bytes) from the start of the file system image to the start of the block bitmap
- offset to inode table – the offset (in bytes) from the start of the file system image to the start of the inode table
- offset to data blocks – the offset (in bytes) from the start of the file system image to the start of the data block array

The last field "unused" is just a padding to make the super block 64 bytes (makes it easier to read when printed via a hex dump).

To be a valid TTFS file system the image must begin with the characters "TTFS", this is a common way for operating systems to know which driver to load to interpret file a file system.

The inode count represents the total number of index nodes in the file system. Each file or directory in the file system needs to have an index node. This count tells how many files or directories the file system can support.

The block count represents the total number of data blocks that are in the file system. Each file or directory needs to be allocated at least one block to store data. This count tells how much space is available in the file system.

The inode and block bitmap are arrays of bit representing which inodes and blocks are used for files and which are not. For example, if a file system had 16 inodes and 2 were used for files, then the remaining inodes would be free. Used inodes are represented with a bit value of 1 and free inodes have a bit value of 0. So, for this example the inode bitmap would contain 2 bytes (8 bits per byte and 16 inodes requires 2 bytes) and would look like this:

```
             Byte 0   Byte 1
     Binary: 00000011 00000000
Hexadecimal: 03       00
```

Notice that the byte ordering goes from low to high from left to the right, but the bit ordering goes from low to high from right to left.

The super block for a TTFS image consisting of 20 inodes and 100 data blocks, would look like this in a hexadecimal dump. NOTE: numbers in an x86 created binary image are little endian

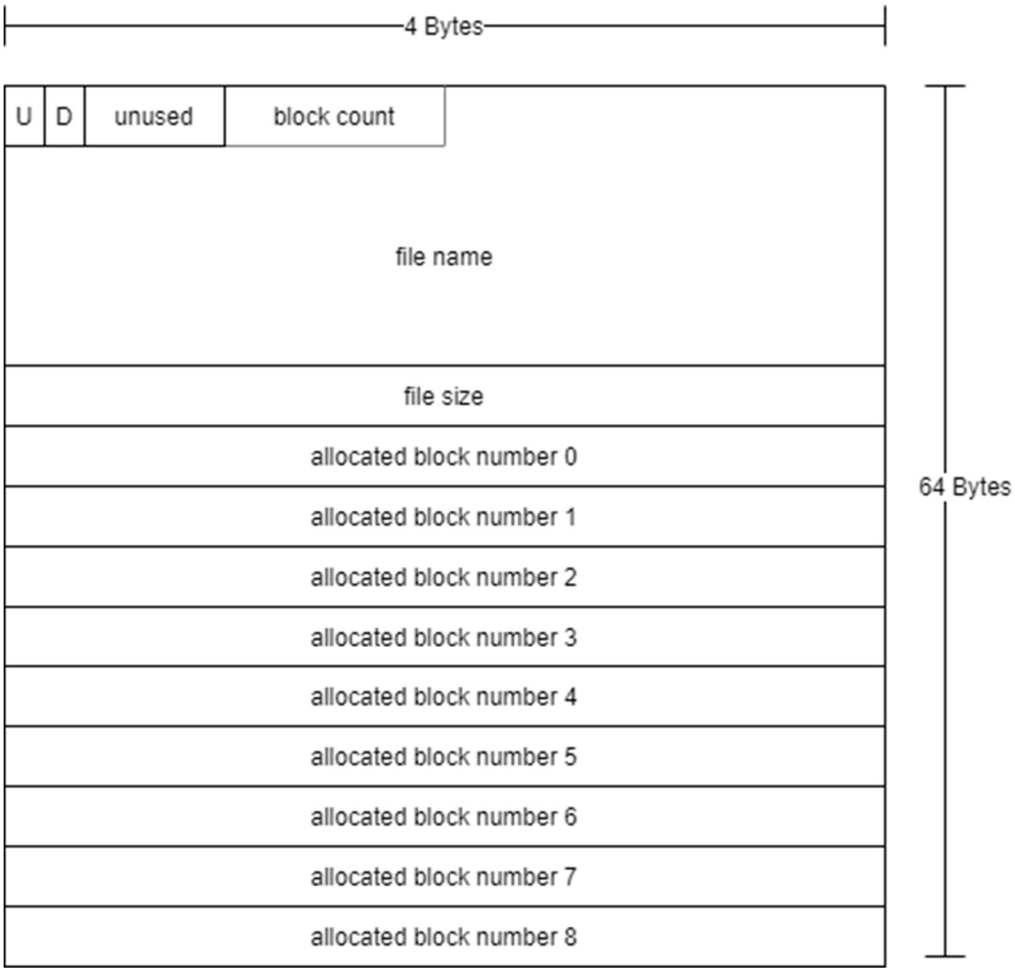
```
54 54 46 53 14 00 00 00  64 00 00 00 20 00 00 00  |TTFS....d... ...|
23 00 00 00 30 00 00 00  30 05 00 00 00 00 00 00  |#...0...0.......|
```

`54 54 46 53` is the TTFS magic value
`14 00 00 00` is the number of inodes – hex 00 00 00 14 -> 20 in decimal
`64 00 00 00` is the number of blocks – hex 00 00 00 64 -> 100 in decimal
`20 00 00 00` is the offset to the inode bitmap – hex 00 00 00 20 -> 32 in decimal
`23 00 00 00` is the offset to the block bitmap – hex 00 00 00 23 -> 35 in decimal
`30 00 00 00` is the offset to the inode table – hex 00 00 00 30 -> 48 in decimal
`30 05 00 00` is the offset to the block array – hex 00 00 05 30 -> 1328 in decimal

4.2 TTFS Index Node

An index node contains meta data (data about the data) for a file or directory. The inode table contains an array of inodes, each inode is 64 bytes long and contains several fields.

The structure of an inode is as follows:



- U – used bit – a single bit that indicates if this inode is used for a file (or directory) or not
- D – directory bit – a single bit that indicates if this inode is for a regular file or a directory
- block count – (1 byte) the total number of data blocks this file is using
- file name – (22 bytes) a string of characters representing the name of the file (null terminated). The file name in TTFS cannot be longer than 21 characters to make room for the null terminator
- file size – (4 bytes – little endian) represents the total size of the file in bytes. Block are 512 bytes long and a file might not end exactly at the end of a block
- allocated block table – (up to 9 entries 4 bytes each little endian) represents the block numbers that are allocated to this file. The block count indicates which entries in this table are valid. NOTE: Entries in the table indicate the block number for the allocated block number, not an offset in bytes.

The inode for a file consisting of 1 data block, that is using 10 bytes might look like this in a hexadecimal dump:

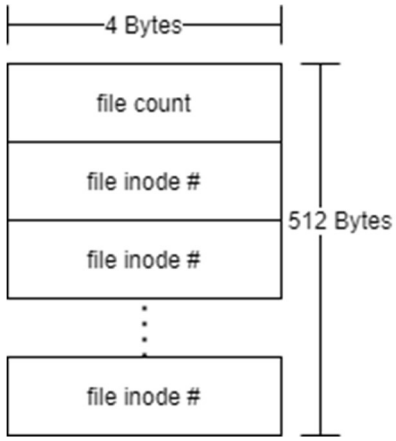
```
01 01 74 65 73 74 2e 74  78 74 00 00 00 00 00 00  |..test.txt......|
00 00 00 00 00 00 00 00  0a 00 00 00 01 00 00 00  |................|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
..... rest of the inode is zeros .....
```

01 shows the used bit is on but the directory bit is not
01 indicates the is 1 used block in the file

`74 65 73 74 2e 74 78 74` is the name of the file
`0a 00 00 00` is the size of the file in bytes – hex 00 00 00 0a -> 10 in decimal
`01 00 00 00` is the index of the first (and only) data block – hex 00 00 00 01 -> 1 in decimal

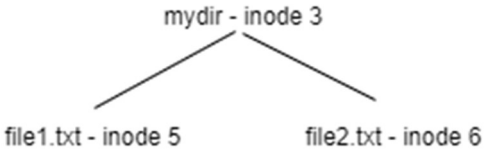
4.3 TTFS Directory

Directories (folders) in TTFS are special files. The inode for a directory has the directory bit flipped to indicate that is a directory and not an ordinary file. The data blocks for a directory entry to not contain file data. Instead they indicate the inodes for the files (or directories) that are contained in the directory. The structure for a directory entry data block is:



The first 4 bytes indicates an integer number (little endian) of the number of file inode numbers contained in the block. The rest of the data block contains 4 byte integers (little endian) for the inodes for files.

For example, consider a directory ‘mydir’ that contains two files ‘file1.txt’ and ‘file2.txt’



The inode for mydir (inode 3) would contain a single data block. The directory data block would look like

```
02 00 00 00 05 00 00 00  06 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
..... rest of the block is zeros .....
```

`02 00 00 00` is the number of files in the block
`05 00 00 00` is the inode number for file1.txt – hex 00 00 00 05 -> 5 in decimal
`06 00 00 00` is the inode number for file2.txt – hex 00 00 00 06 -> 6 in decimal

TTFS always contains a single special directory with no name (the file name is all zeros). This directory represents the root directory of the file system. All files and directories are contained in the root directory. The root director will always use inode 0.

Find the file ttfs.img attached to the lab assignment. This contains a binary image for a TTFS file system. Using the output from hexdump -C and the specification for TTFS answer the following questions:

NOTE: All integers are stored as little endian on x86. So make sure you read the values correctly

1. What is the largest file that TTFS can support? How did you compute this number?
2. How may inodes and data blocks are in the file system?
3. How many inodes are free (not used by files or directories?

4. Draw a diagram of the directory hierarchy, include this diagram with your submission.
5. What are the names and sizes of each file in the file system?
6. How many blocks is each file in the file system?
7. For each file, what is the offset (from the start of the file system) for each data block? How did you compute this?
8. Find one of the text files in the file system. Write the data (as a string) for the file contents. How were you able to find the data?

5 Extra Credit – TTFS Dump

Using the specification from section 4 and the include header file (ttfs.h) that defines the file system structures, write program that will read in a TTFS file system image and prints the directory structure and contents for each file. For example:

```
user@pc:~$ ttfsdump ./ttfstest.img
/file2.txt – 10 bytes
(the contents of file2.txt printed here)
/dir1/file1.txt – 100 bytes
(the contents of file1.txt printed here)
/dir1/dir2/file3.txt – 20 bytes
(the contents of file3.txt printed here)
(continue for all files)
```

Deliverables

Turn in a document per instructor's directions with the recorded output, answers to the questions for each activity, and the source code for the extra credit. Include an introduction describing the lab in your own words and a conclusion discussing what you learned as well as what you liked about the lab and what could be improved. Be sure to include documentation of any external resources (man pages, stackoverflow, etc.) that you used to complete the lab.