**Testing Document – Assignment 3 (100265937)**

**Note:** The following testing document uses examples from the copytestfiles.bat. These files have been included in the submission. Using the copytestfiles.bat will add these files to the img file.

**Design Philosophy:** Separated Disk\_Commands and Filesystem into two modules, this is to keep concerns separated. Also considered putting Disk\_Command directly into the Command.c, but felt like this would make command.c too long and too “unfocused”

Another important decision made was that I store the current working directory. There is another way to do this: Everytime I do a cd start from root and go up. The disk on bochs is quite slow, and disk reads in general can be quite slow. Therefore, I felt storing the current directory in memory would be beneficial.

**Note:** Only a NON-ROOT Directory is stored in memory, while there’s some small processing involved to check whether we’re a root directory it’s ultimately trivial, and it’s much better than having to store 7kb of working directories in memory. Instead we store 0.5kb.

**Testing Code:** Some of the testing code can be found commented out in kernel\_main.c.

Others were added to the individual methods, but were removed from the functions on submission.

**Testing Examples:** Submitted with the code is a copytestfiles.bat. The Makefile automatically calls this when creating. Any Necessary directories and files are included in the zip file.

**PART A:**

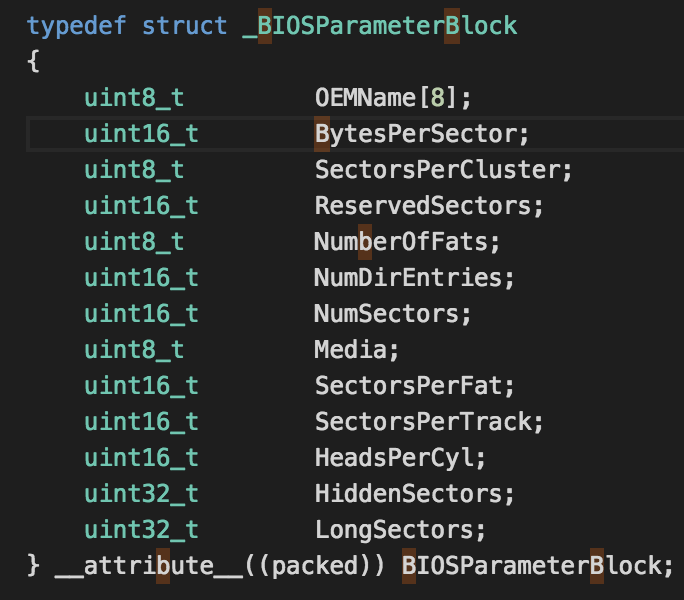
**void FsFat12\_Initialise():**

*The Kernel should do the following things:*

* ***Read the BIOS parameter block from the boot sector.***

The Boot Sector Exists in Sector 0 – Reading from this sector, and converting tp BPB allows us to get the BiosParameterBlock (Bpb).

We also know a few things about the Bpb.

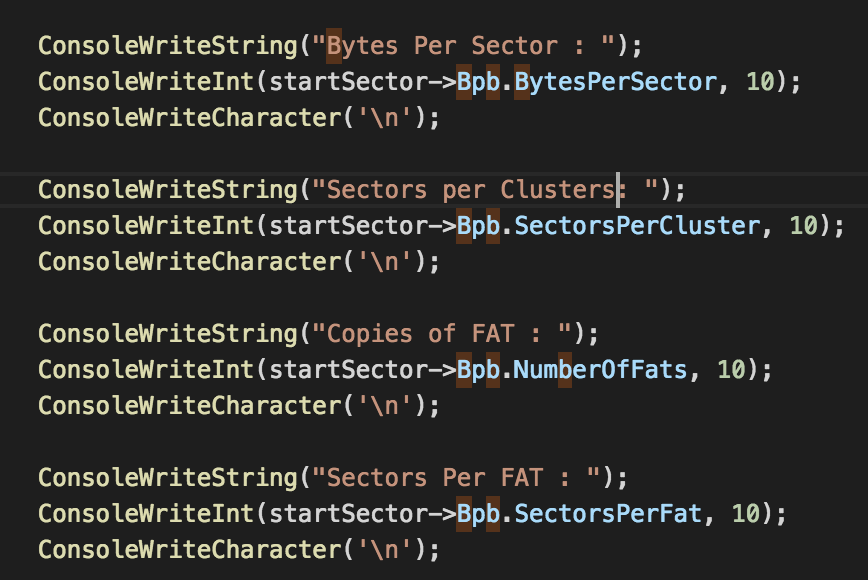


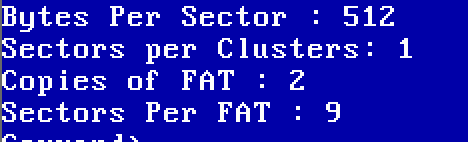
There’s a few things we know implicitly due to the FAT12 File System. We know, for instance, that there are:

**512** *Bytes Per Sector*

1. *Sector Per Cluster*

**2** FAT Copies.  
**9** SectorsPerFat.





This would be an incredible coincidence. We can be confident we’re copying the correct location.

* ***Calculate the Offsets (Ie: Number of Sectors) to the FAT, the offset to the root directory, and the offset to the data sectors.***
* ***Read the FAT into memory.***

Here I use a hex reader to check which sector a file lies in. Using this I can read an area and see whether, using 32 bit uint, whether the value is correct based on what we’re expecting.

To Accomplish this the first thing I did was put the current sector in the ls for debugging purposes.



Here we can see KERNEL.SYS exists in sector 2, Testing in sector 83, and HiTest.txt in Sector 95.

Therefore we can look at these sectors to get an idea of if we’ve copied to the right location.

32 bits there

FAT\_TABLE[1] =

Binary:



Hexadecimal:



// In uodos.img it looks like the follow. Highlighted to show the sequence.



If we convert this to same endianness we get the following bytes.



Also the fact that read works with more than one Sector (As shown in **the READ testing section**) helps to validate this.

**FILE FsFat12\_Open(const char\* filename)**

*Search the directory for the specified filename, which can be either a* ***complete path, including directories.***

*If the file exists on the disk, return a FILE structures set correctly, otherwise return a FS\_INVALID.*

***Testing:***

*Code can be found*

*Passing Empty Filename:*

This should return FILE with FS\_INVALID



*Passing NULL:*

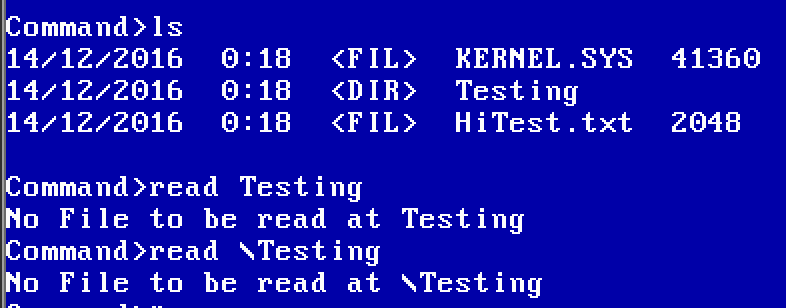
This should return FILE with FS\_INVALID.



*Opening a Directory:*

Opening a Directory and returning the FILE. We should see that the FS\_Directory Flag is set.

Testing with READ (Trying to read a directory should fail)



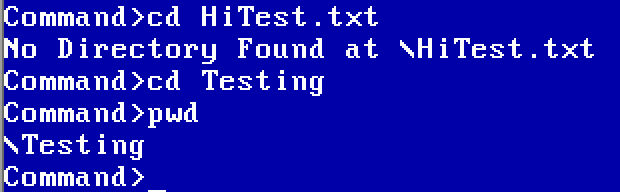
Test with Code.



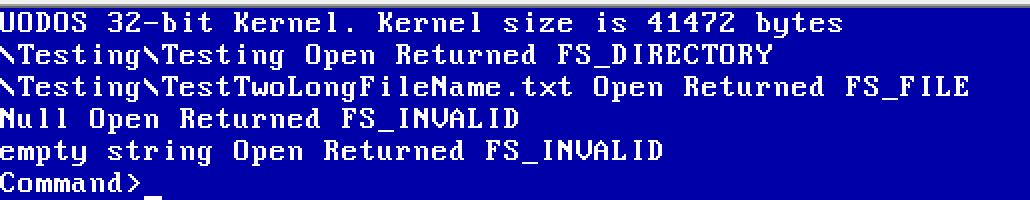
*Opening a File:*

Opening a File we should see that the returned FILE should have the FS\_FILE flag set

Test with Change Directory on a FILE (Should fail)



Test with Code



***For More Testing See Part B.***

**unsigned int FsFat12\_Read(PFILE file, unsigned char\* buffer, unsigned int length):**

*Reads the next block of data from the specified pointer to file. Reads position from the CurrentCluster and Position fields.   
  
The amount to read is specified in the length parameter, and we should read to the specified buffer.*

*Reading 32 Bytes:*



*Once we have finished reading we set the EOF flag to 1.* ***Reading to the end of the file should set the flag, added the following to the read code***

*if (file->Position == file->FileLength)*

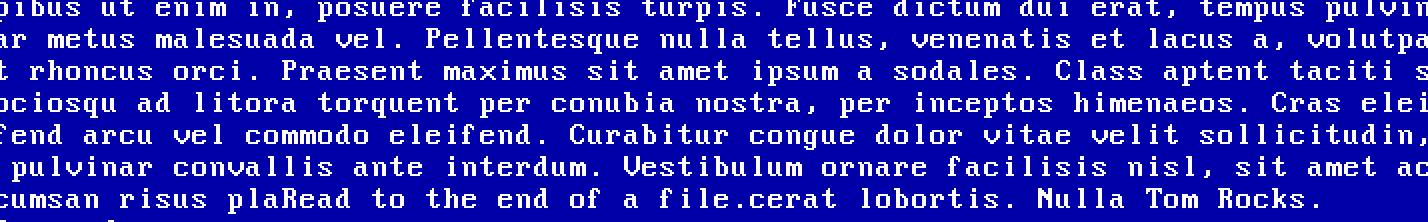
*{*

*ConsoleWriteString("Read to the end of a file.");*

*file->Eof = 1;*

*}*

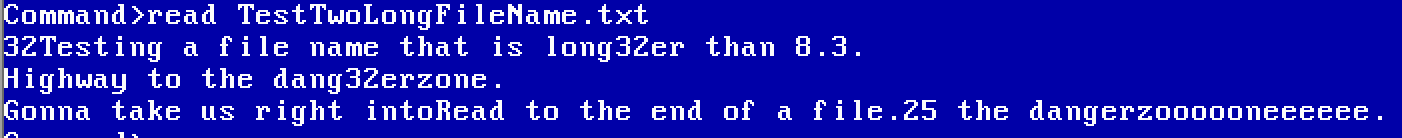
This therefore will print when we’ve read to the end: Example:



If you look before cerat Lobrotis. Nulla Tom Rocks. You’ll see the ‘Read to the end of File’ printed out. This is since the printing is performed after the read.

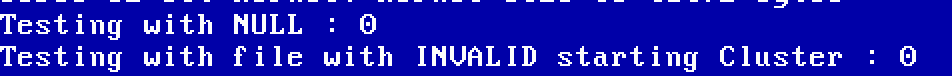
*The function returns the actual number of bytes read*

Here we can see it’s 32 (The limit we passed in) until the end, where it’s 25 (The amount left over)

**

*Testing NULL // Testing FILE where CurrentCluster is set to 0*

*If we pass in null we should handle this by returning 0 and doing nothing. We ConsoleWriteInt to confirm this.*

**

***SEE READ filename in part B for other tests.***

**void FsFat12\_Close(PFILE file)**

*Sets the Eof field in the FILE to 1.*

This is relatively simple to test – I simply called FsFat12\_Close on the pointer to the file, and then, outside of the function check that it the eof flag has been set.

I also called this on a null pointer, to ensure that it correctly handles NULL files.

Tested with the following code:

FILE file;

file.Eof = 0;

FsFat12\_Close(&file);

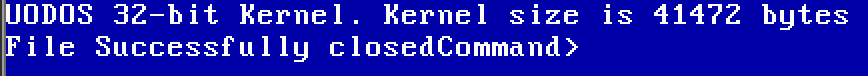
if (file.Eof)

{

ConsoleWriteString("File Successfully closed");

}

FsFat12\_Close(NULL);



**PART B:**

**READ filename**

*Reads the entire file at 32 bytes at a time, hitting enter continues reading the next 32 bytes. CTRL+C will cancel the read. Can pass a FULL filepath, or pass a filepath relative to the present working directory:*

*For instance:*

* ***read \filepath\file.****txt will read the file*
* ***cd filepath******-> read file.txt*** *will read the file.*

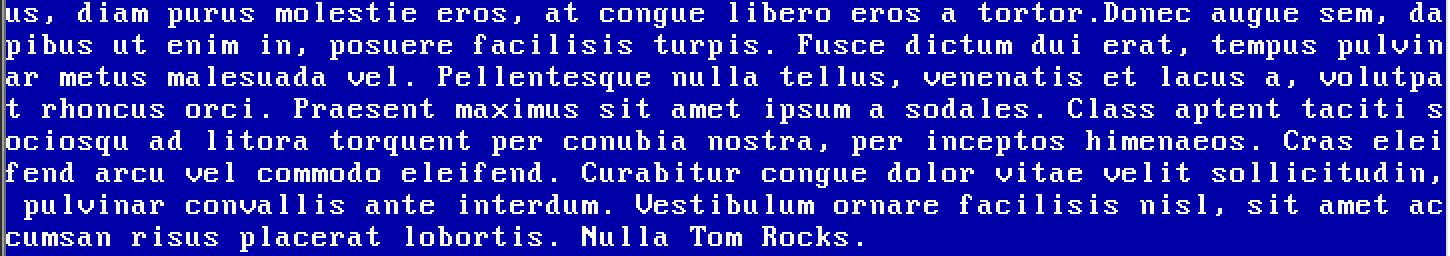
*Testing (Note we are using the HiTest.txt and \Testing\Testing\One\Two\Three\Hear.txt in the root directory for this test):*

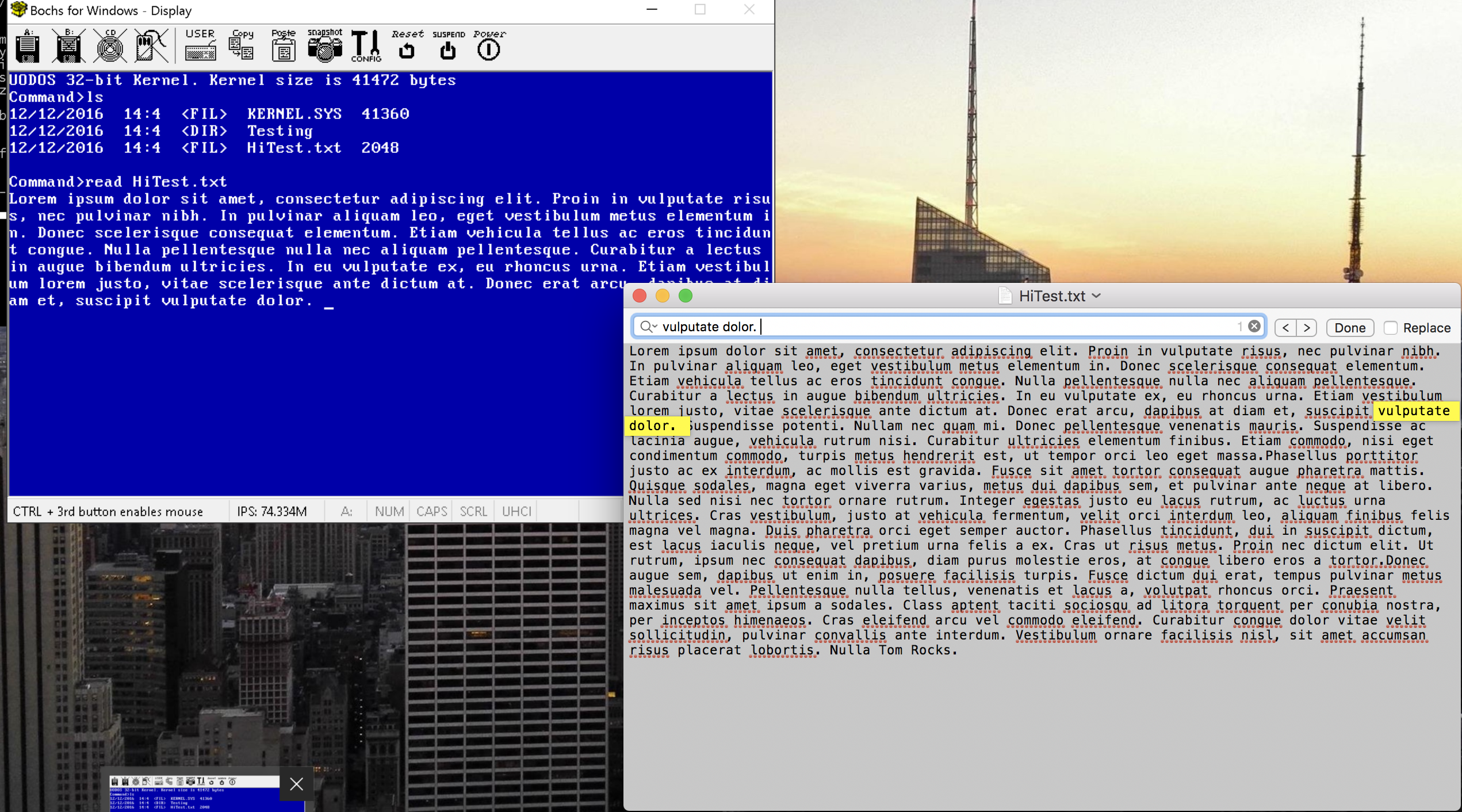
* First we ensure that we cannot read a NULLPTR. This is important, as we would get an exception otherwise.
* Test Reading LESS than a sector, in this case 32 bytes

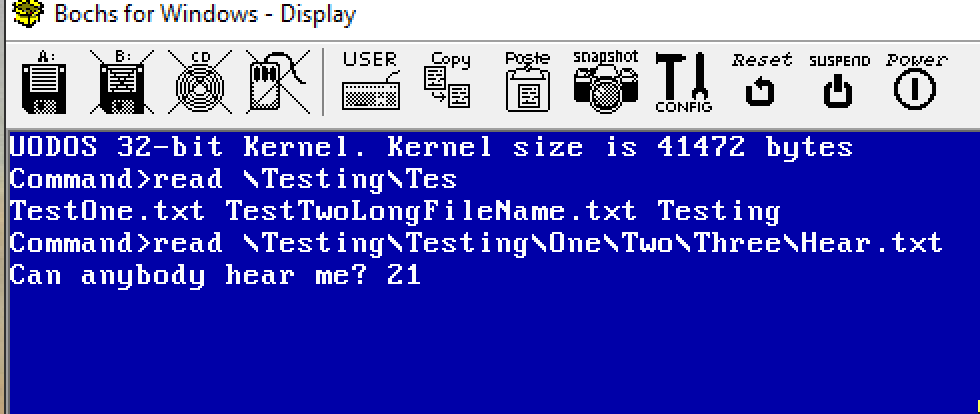
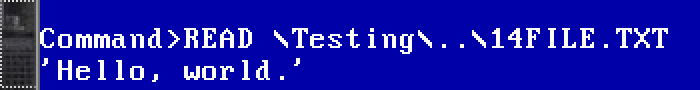
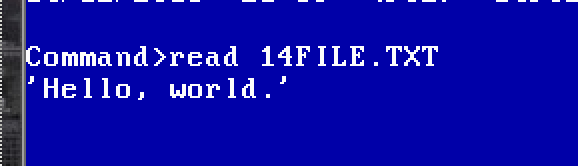
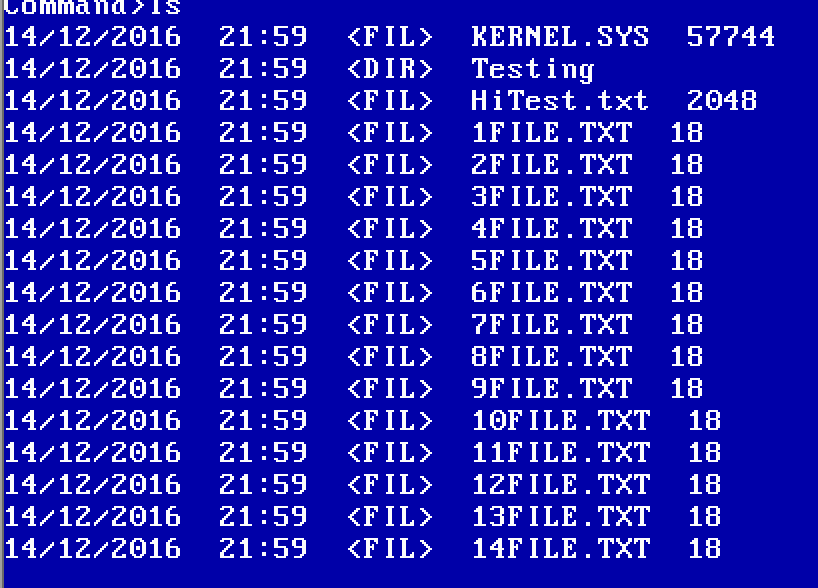


* Then continue to read, another 32 bytes, to ensure continuity



* We then ensure that we can read to the end of the file and we correctly identify and end at the end of the file. 
* You can tell this is the end of the file as it ends with “Nulla Tom Rocks.” The other thing to note here is that the final bytes are not % 32, so we ensure that we can read arbitrary length files. Also note that this file is greater than 512 bytes (a sector) proving we can navigate sectors correctly.
* Next we attempt to Read using a buffer of 513 bytes. One more than a sector usually allows.



* As you can see the highlighted text in the text file, and the text in TomOs. The 513th byte is the Space proceeding “dolor.”
* Next thing to try is to try to pass in a length greater than the size of the file. Here we test with a file that is 21 bytes long.
* 
* **The 21 Here is a ConsoleWriteInt(totalRead, 10). It indicates only 21 bytes were read.** We passed in a buffer of 513, meaning that we do not read OVER any file.
* **Notice on the above two screenshots:**
  + **We do reading directly from the root directory (Ie: no \ in the filepath) and reading using a full file path**
* **We must ALSO be able to read any file in the root directory over the 16th entry, as the root can store 224 files. Below 14FILE.txt is the 17th Entry in the RootDirectory. Both the commands in the below screenshot work.**
* ******

**Change Directory**

*Changes Directory. Can pass both a FULL filepath and one relative to the present working directory. This uses Cygwins / Powershell / Windows CMD case sensitivity.*

*For instance:*

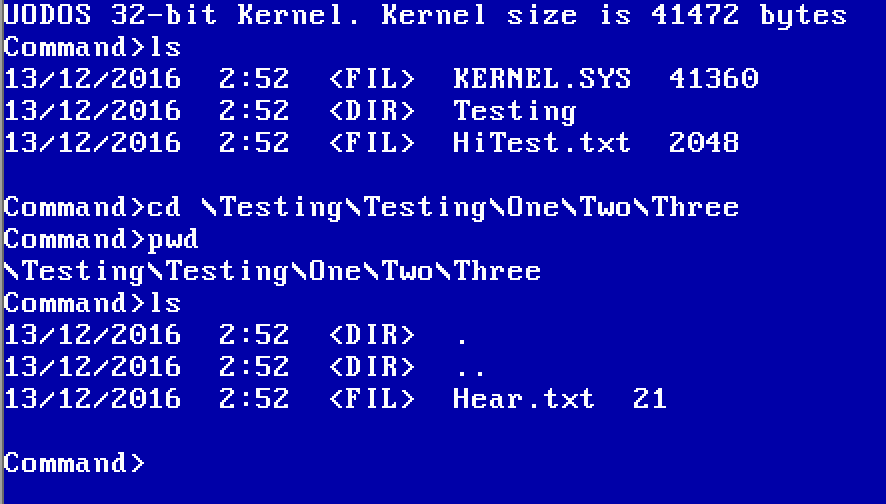
***cd \Testing\Testing\One\Two\Three***

***cd Testing*** *->* ***cd Testing*** *->* ***cd One*** *->* ***cd Two*** *->* ***cd Three***

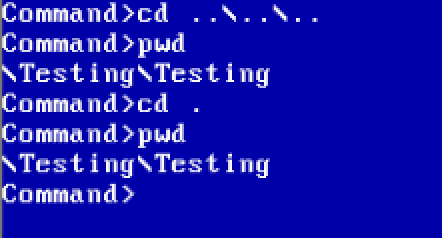
*typing pwd after both will be \Testing\Testing\One\Two\Three*

*Testing:*

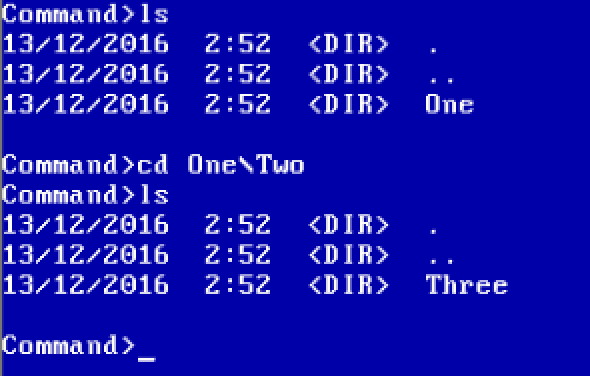
* First we check changing into a path from root (“\Testing\Testing\One\Two\Three”). This demonstrates we can change from root.



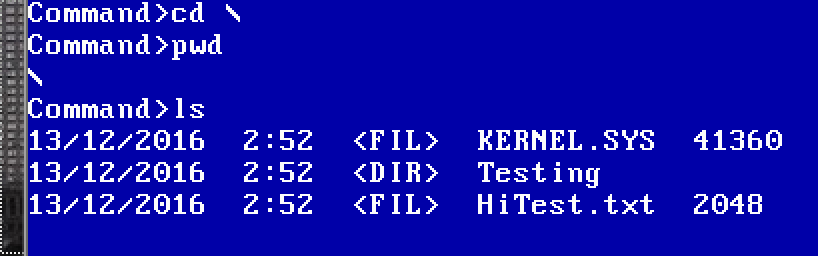
* Next we attempt to not only address paths relative to the PWD, which in this case is \Testing\Testing\One\Two\Three, but also demonstrate that we can use the .. directory to move to the parent.



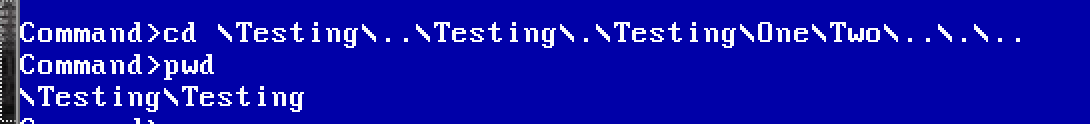
* From here we demonstrate that we can continue to move up to children directories addressing from relative paths.

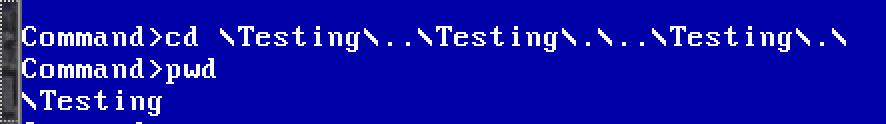


* Then we try going back to the root directory. (cd \)

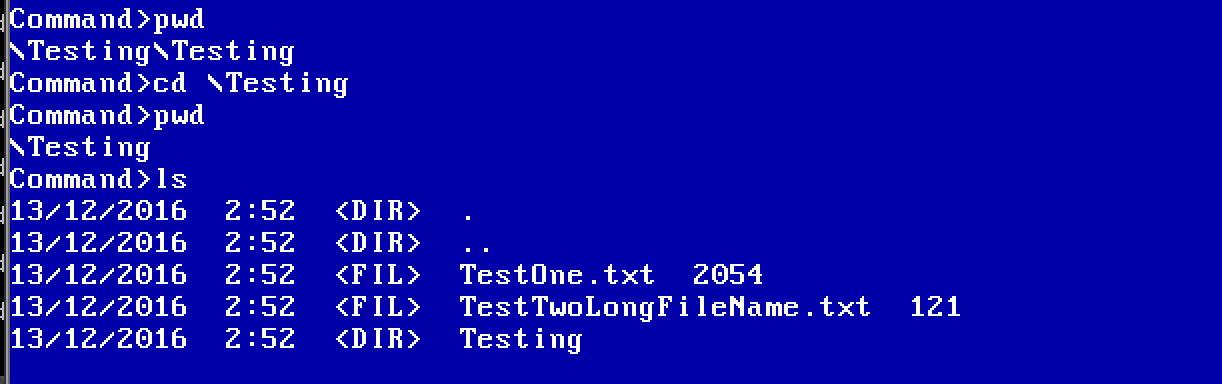


* Then we try a complicated change directory path utilizing a lot of .. and . – This doubles as a test for pwd, as we can see after changing the directory we are still in a valid directory path

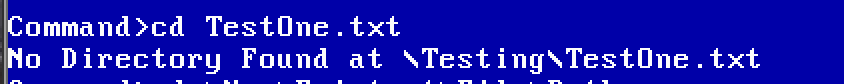




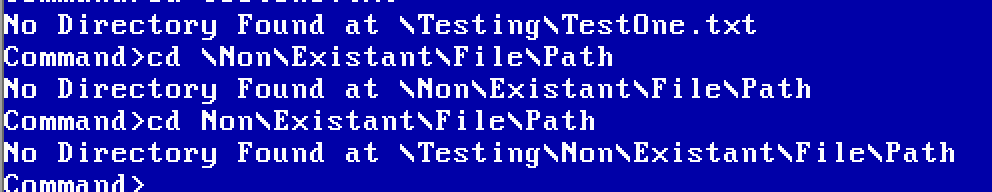
* Then we attempt to change to a directory relative to root from a non-root directory (from \Testing\Testing to \Testing)



* Then we test we cannot change directory into a file path



* Then check that we cannot change into a non-existent file path.



**PresentWorkingDirectory**

*Set Present Working Directory:*

As shown in the above tests the pwd handles file paths with . and ..

Here are some more examples of that.

*Get Present Working Directory:*

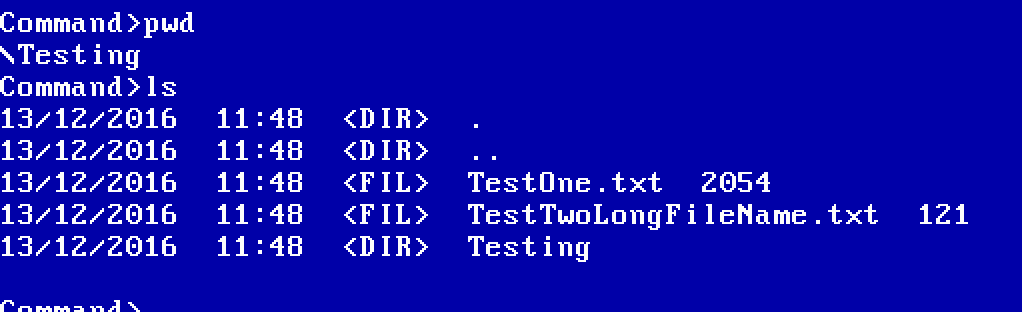
We can only test this with the pwd command in the disk command. However, we can be sure that it works because of the above tests.

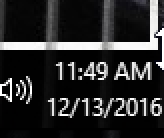
**LS / DIR**

List files and Directory works very similarly the cmd line dir. It shows Date of creation, Time of Creation, Whether it’s a file or a directory, the long file name and the file size.

*List Files*

*Below demonstrates that, the picture*

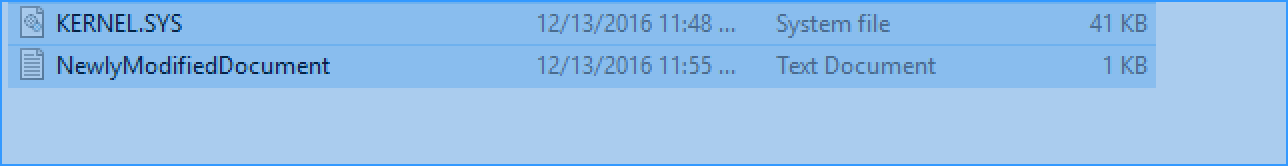
**

**

*I don’t know how to fix the Date in the VM to be DD/MM/YYYY so it’s currently MM/DD/YYYY.*

As you can see from the above all the files were created from the make process at the same time, 11:48 on the 13th December.

Adding a new file manually demonstrates the following (Note, windows is showing it in US date standards):

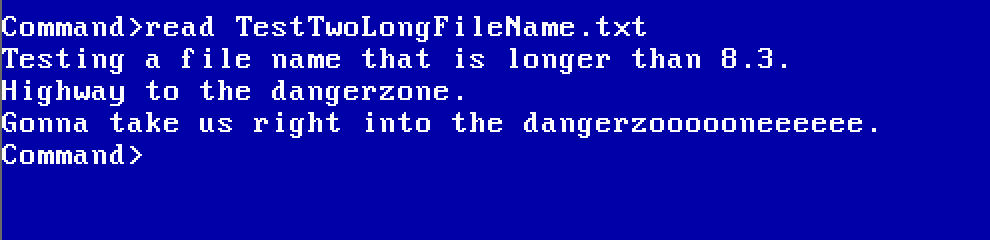


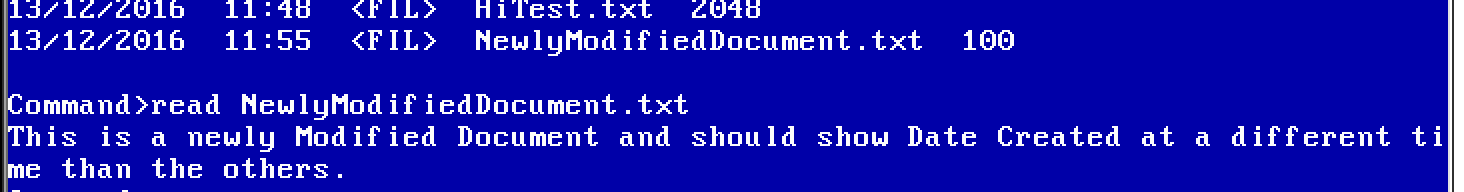


*Root Directory:*

The Root Directory can store more than 16 entries. Our code should handle that.

*Long File Names:*

LS handles long filenames. To prove we can use these long filenames I will demonstrate a read on the long file name “TestTwoLongFileName.txt”  


**

This tests that Long File Names are usable in the system.

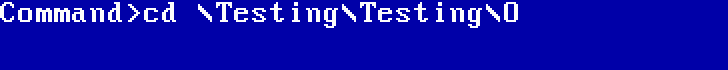
**Additional Features:**

**Tab to autocomplete:**

*Tapping on the Tab Key will autocomplete the path typed in*

*For example:*

*cd \Testing\Testing\O <TAB>  
will autocomplete to cd \Testing\Testing\One*

**

*<TAB>*

**

*cd \Testing\Tes<TAB>*

*Will show a list of files that could autocomplete: TestTwoLongFileName.txt TestOne.txt Testing*

**

*Autocomplete on RootDirectory on files greater than the 16th entry.*

***Note: Despite being called 14FILE this is actually the 17th***

**

*<Tab>*

**