CS3026 Assessment 2015-2016

CONTENTS

1. CGS D3-D1

1.1 Implementation (D3_D1)

void format ( )

The *format*( ) function in filesys.c:80 prepares the virtual disk:

94-97: Fills a temporary block with null bytes, copies the virtual disk general information ("CS3026 Operating Sytsems Assessment") and writes this block to block 0 of the virtual disk.

102: Clears the FAT array in memory.

105-108: Dynamically initialises the FAT array. FAT[0] should always be ENDOFCHAIN, FAT[1] to FAT[fatblocksneeded - 1] should each be set to the next concurrent FAT block, and FAT[blocksneeded] should always be ENDOFCHAIN. FAT[blocksneeded + 1] always represents the root directory, so should be ENDOFCHAIN also.

This allows the constant MAXBLOCKS to be redefined as up to 64 times larger than 1024 blocks.

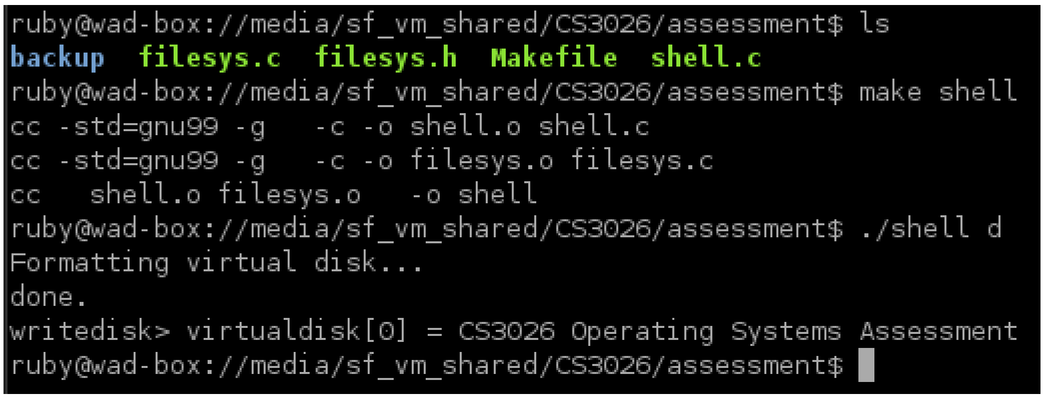
109: Function *copyFAT*( ) is called to write the contents of the FAT memory array to the virtual disk.

114-118: Prepares root directory in a temporary block. Parent of root is set to 0 to indicate we cannot go up from the root during function calls which process a path. All entries in the root must be initialised to *.unused* = 1 to mark them as free. Temp block is written to virtual disk at root directory block index. Global var *rootDirIndex* is set for later reference when processing absolute paths.

121: Global var *currentDirIndex* is set (equals root dir) for later reference when processing relative paths.

1.2 Instructions (D3_D1)

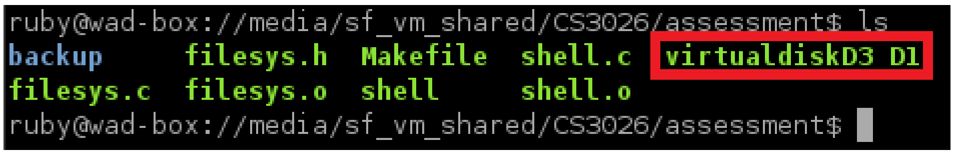
1. Compile with `>make shell`
2. The shell program takes one argument which should be the CGS band. Run `>./shell d`



```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ls
backup  filesys.c  filesys.h  Makefile  shell.c
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ make shell
cc -std=gnu99 -g   -c -o shell.o shell.c
cc -std=gnu99 -g   -c -o filesys.o filesys.c
cc   shell.o filesys.o   -o shell
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ./shell d
Formatting virtual disk...
done.
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ 
```

Fig 1.0 Compiling and running the shell program

The disk image virtualdiskD3_D1 should be created in the current directory.



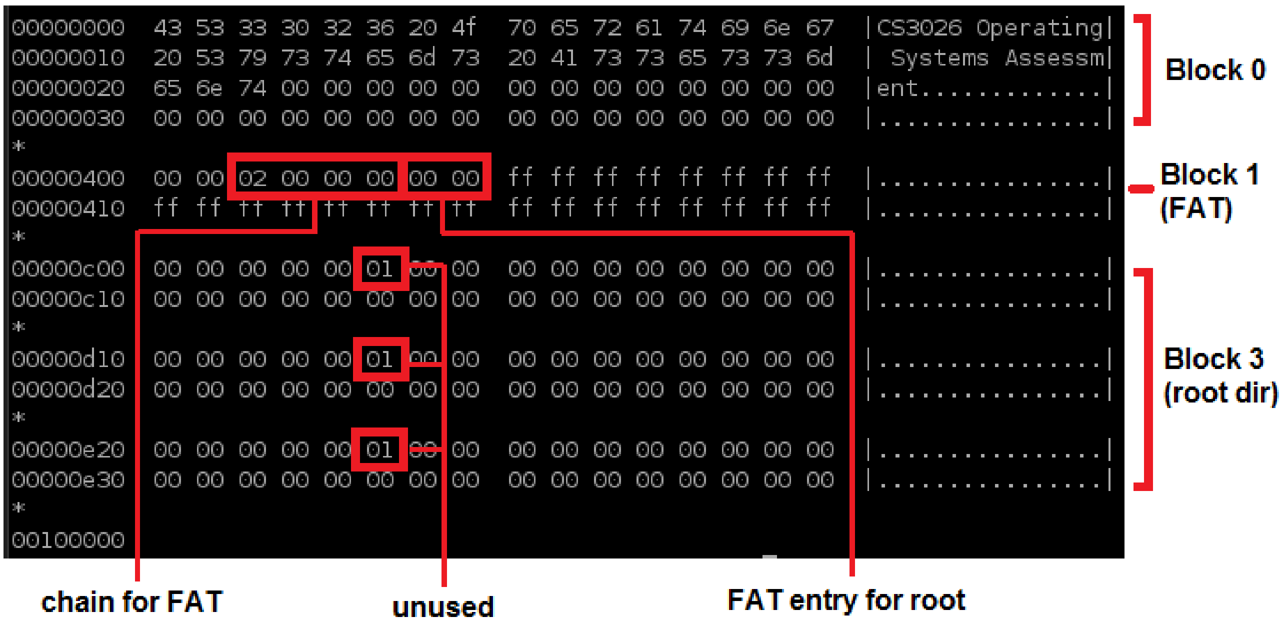
```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ls
backup     filesys.h  Makefile  shell.c  virtualdiskD3_D1
filesys.c  filesys.o  shell     shell.o
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ 
```

Fig 2.0 Directory contents after execution with argument 'd'

3. Run `>hexdump -C virtualdiskD3_D1`

1.3 Analysis (D3_D1)

The hexdump of the disk image file should appear as below.



```
00000000  43 53 33 30 32 36 20 4f  70 65 72 61 74 69 6e 67  |CS3026 Operating|
00000010  20 53 79 73 74 65 6d 73  20 41 73 73 65 73 73 6d  | Systems Assessm|
00000020  65 6e 74 00 00 00 00 00  00 00 00 00 00 00 00 00  |ent.............|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  00 00 02 00 00 00 00 00  ff ff ff ff ff ff ff ff  |................|
00000410  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000c00  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00000c10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000d10  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00000d20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000e20  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00000e30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00100000
```

Fig 3.0 Hexdump of virtualdiskD3_D1

The virtual disk has (if initialised to manage 1024 blocks) 4 blocks populated after formatting.

Block 0:

Reserved for file system information and end-of-chain handling.

Block 1-2:

File Allocation Table. We only see part of the first block since all bytes past the cutoff repeat and are set to UNUSED (ff).

Bytes 0x0402 - 05 represent the chain for the FAT itself: FAT[1] indicates FAT[2], which in turn is the end of the chain.

The last entry in the FAT indicates that the root directory is also end-of-chain.

Block 3:

Root directory. Contains three empty entries which have been marked with a byte to denote that they are not currently in use.

2. CGS C3-C1

2.1 Implementation (C3_C1)

MyFILE * myfopen (const char * path)

The *myfopen*( ) function in filesys.c:255 returns a filedescriptor and creates new files in write-mode:

260-262: Ensures proper mode usage, rejects any arguments that aren't read or write.

266-272: Allocates memory for the filedescriptor and initialises common variables: position within the current file block and total bytes read/written to track absolute position within the file, and access mode.

276-280: Prepares for navigation to the desired directory. Absolute paths will begin with '/' and should be tracked from the root.

283-285: Prepares for tokenization of the path. Copies the path string to an allocated memory location which can be manipulated by *strtok_r*( ), since we are not able to do this with string literals in protected memory.

286-308: Tokenisation of path. We prevent the filename from being lost by ensuring the while-loop exits when the remaining path has been exhausted, so when the loop exits successfully the filename is pointed to by *char * token*.

> **288**: Instances of '..' in the path indicate we need to go up one level. Each directory block stores the index of its parent directory which makes this trivial to handle.
>
> **296**: Instances of '.' in the path indicate we continue from the current directory, the loop then continues without needing to update the location tracker variable *loc*.
>
> **299**: The location tracker is updated with the block number and entry index of the token within the current directory by *getdirentryloc*( ) (filesys.c:131).
>
> **302-304**: Handle situations where the directory path doesn't exist.

311-334: Check for the file within the final directory. Create a new file directory entry ONLY IF file does not exist AND we are in write-mode (313-328). When this section finishes successfully, the location tracker variable should indicate the first block of the file contents.

> **314-316**: Handle non-existing files accessed in read-mode.
>
> **319**: Function *getfreeentry*( ) (filesys.c:221) iterates through the given directory to return the first unused entry. In this implementation, directories may be extended in the FAT and be distributed over multiple disk blocks. If the directory is full, *getfreeentry*( ) will attempt to create another directory block and modify the FAT chain.
>
> **321-323**: Handles situations where there are no unused entries in the directory and the virtual disk is full.
>
> **325**: Function *makenewfile*( ) (filesys.c:167) initalises a given directory entry with a filename, modtime and index of the first block of file contents. File length is cleared and the entry is flagged as being in use. The FAT is modified to reflect that the first block of the file

contents is the end of a chain.

326-328: Handles situations where there is space for the directory entry, but no available blocks left for file data.

332: If the directory entry for the file already exists, we simply need to get the data from *.firstblock* to find where the file contents start on the virtual disk.

334: free the memory allocated to store the path.

337-339: Finish initializing the filedescriptor with the current block it should point to, and the location of the file's directory entry in the virtual disk for when it needs to be updated.

343-346: Error handling. Ensures we free up any allocated memory if things go wrong.

void myfclose (MyFILE * stream)

The *myfclose*( ) function (filesys.c:349) ensures that the stream buffer is written to the virtual disk, that the file directory entry is updated with the correct meta data and that the memory allocated for the filedescriptor is freed:

360: Writes buffer contents over current block.

361: Updates the file modification time.

364-365: Compares the total bytes read and written by the stream to the previously known length of the file. If this total is greater it means we have appended extra data and the file length needs to be updated (there is no seeking functionality in this implementation).

void myfputc (int b, MyFILE * stream)

The *myfputc*( ) function (filesys.c:371) writes a single byte to the file stream buffer. If the buffer is full, it's contents are written to the virtual disk and the next block is loaded into the buffer first:

379-397: Handle cases where we try to write past the end of the block.

380: If the position of the stream within a block is greater than the blocksize, we need to check whether the current block is the end of a FAT chain.

383: Function *getfirstunusedblock*( ) (filesys.c:155) returns the index of the first virtual disk block not in use, else 0 if they are all in use.

384-388: Handles cases where there are no unused blocks to extend the file.

391-397: Clears the contents of the block used to extend the file since it may contain leftover data from removed files, then extends and updates the FAT chain.

402-407: Now that we can be sure there is another block in the chain, write the buffer contents out to the virtual disk and read the next block in the chain. Position needs to be reset because we are now at the first byte of the block.

410-412: Finally, write the byte into the buffer and increment the position and total bytes.

int myfgetc (MyFILE * stream)

The *myfgetc*( ) function (filesys.c:415) returns a single byte from the file stream buffer. If the end of the buffer is reached, it's contents are written to the virtual disk and the next block is loaded into the buffer first. Returns EOF when the end of the file is reached:

418: If we are attempting to read past the end of a file, EOF is returned immediately.

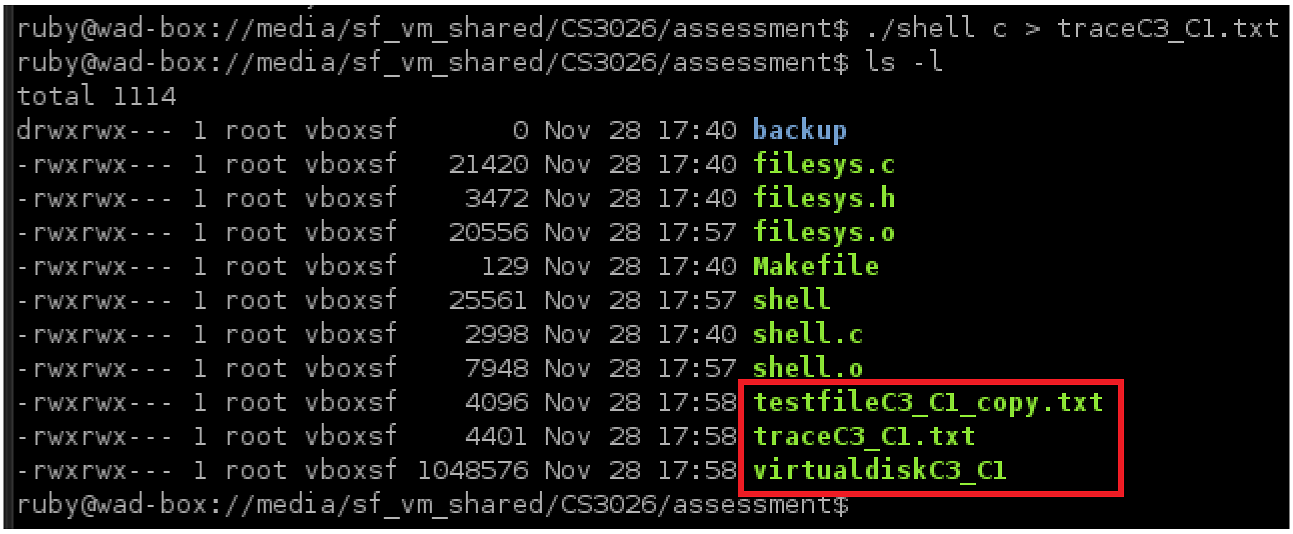
421-431: Handle cases where we try to read past the end of a block. 423 prevents us from reading into the contents of another file accidently. The rest of the section writes the buffer content, in case we made changes to it with *myfputc*( ) earlier, then updates the filedescriptor block index and head position and reads the next block of the file into the buffer.

436-438: Read a byte from the buffer and update the head position and total bytes covered.

2.2 Instructions (C3_C1)

1. Compile with `>make shell` if not already done
2. Run `>./shell c > traceC3_C1.txt`

The following should be created in the current directory: the virtual disk image, a trace of the program execution and a copy of the contents of a text file written on the virtual disk.

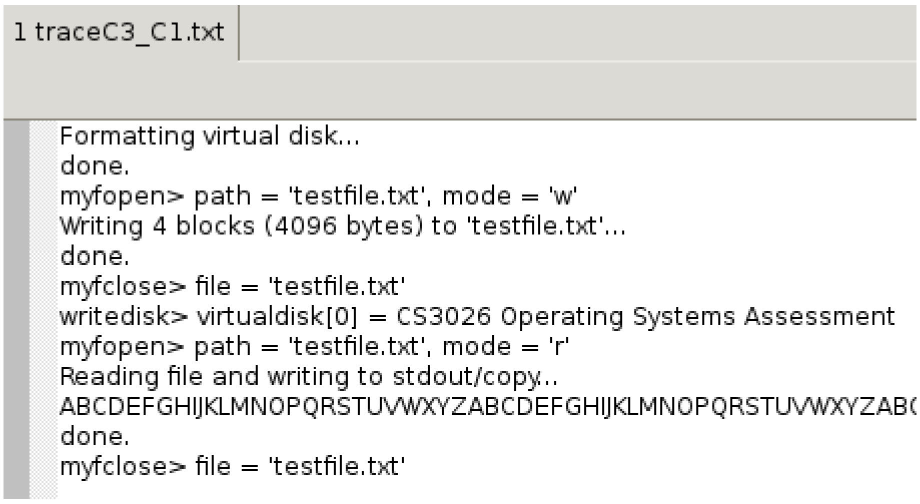


```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ./shell c > traceC3_C1.txt
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ls -l
total 1114
drwxrwx--- 1 root vboxsf       0 Nov 28 17:40 backup
-rwxrwx--- 1 root vboxsf   21420 Nov 28 17:40 filesys.c
-rwxrwx--- 1 root vboxsf    3472 Nov 28 17:40 filesys.h
-rwxrwx--- 1 root vboxsf   20556 Nov 28 17:57 filesys.o
-rwxrwx--- 1 root vboxsf     129 Nov 28 17:40 Makefile
-rwxrwx--- 1 root vboxsf   25561 Nov 28 17:57 shell
-rwxrwx--- 1 root vboxsf    2998 Nov 28 17:40 shell.c
-rwxrwx--- 1 root vboxsf    7948 Nov 28 17:57 shell.o
-rwxrwx--- 1 root vboxsf    4096 Nov 28 17:58 testfileC3_C1_copy.txt
-rwxrwx--- 1 root vboxsf    4401 Nov 28 17:58 traceC3_C1.txt
-rwxrwx--- 1 root vboxsf 1048576 Nov 28 17:58 virtualdiskC3_C1
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$
```

Fig 4.0 Directory contents after execution with argument 'c'

2.3 Analysis (C3_C1)

The shell program performs the functions specified in the assessment requirements for C3_C1. The trace file shows what the shell has done:

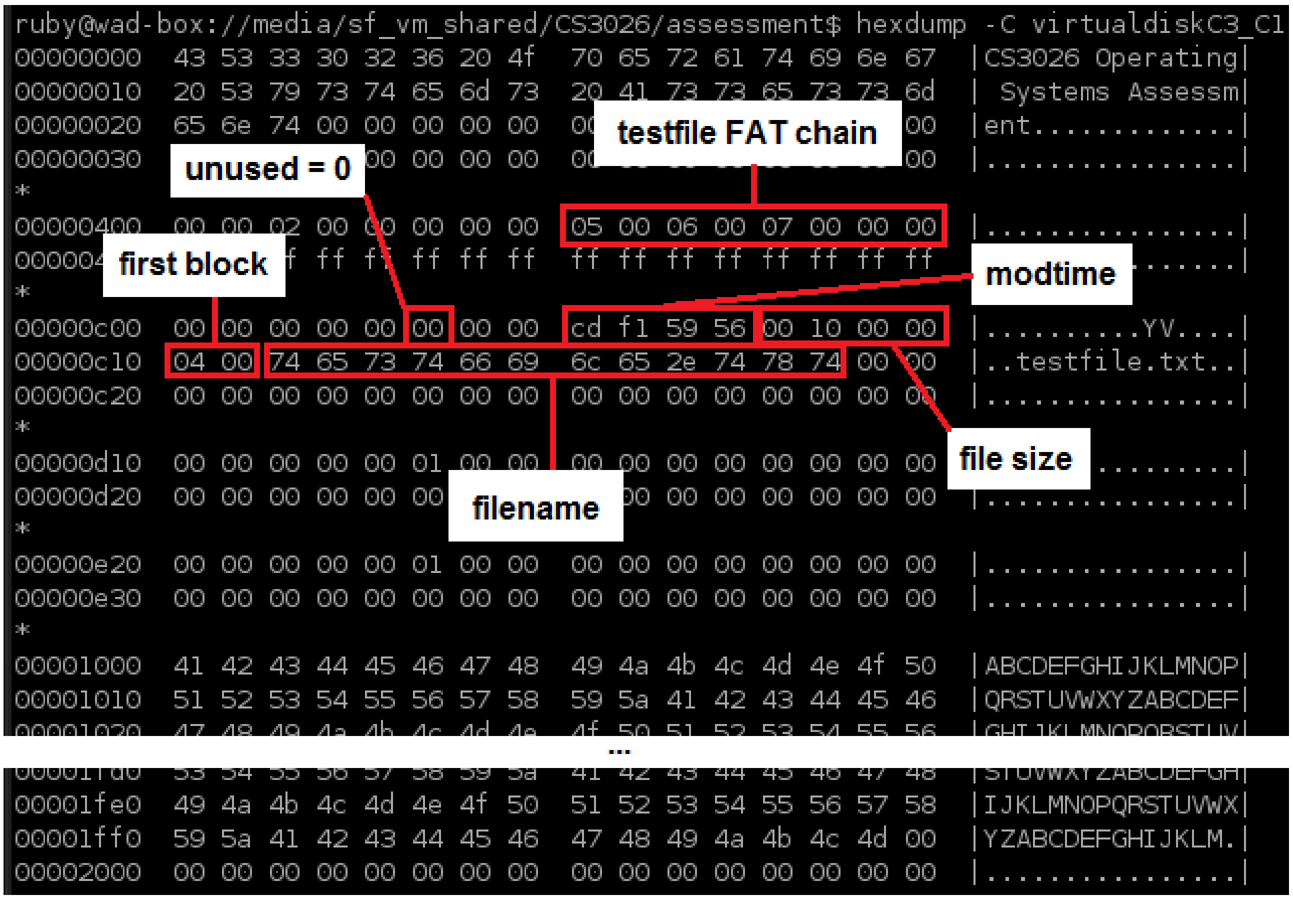


```
1 traceC3_C1.txt

Formatting virtual disk...
done.
myfopen> path = 'testfile.txt', mode = 'w'
Writing 4 blocks (4096 bytes) to 'testfile.txt'...
done.
myfclose> file = 'testfile.txt'
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
myfopen> path = 'testfile.txt', mode = 'r'
Reading file and writing to stdout/copy...
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZAB(
done.
myfclose> file = 'testfile.txt'
```

Fig 5.0 traceC3_C1.txt

We may inspect the virtual disk image with hexdump:



```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ hexdump -C virtualdiskC3_C1
00000000  43 53 33 30 32 36 20 4f  70 65 72 61 74 69 6e 67  |CS3026 Operating|
00000010  20 53 79 73 74 65 6d 73  20 41 73 73 65 73 73 6d  | Systems Assessm|
00000020  65 6e 74 00 00 00 00 00  00                   00  |ent.............|
00000030                 00 00 00  00                   00  |................|
*
00000400  00 00 02 00 00 00 00 00  05 00 06 00 07 00 00 00  |................|
000004           f ff ff ff ff ff  ff ff ff ff ff ff ff ff          ......|
*
00000c00  00 00 00 00 00 00 00 00  cd f1 59 56 00 10 00 00  |..........YV....|
00000c10  04 00 74 65 73 74 66 69  6c 65 2e 74 78 74 00 00  |..testfile.txt..|
00000c20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000d10  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00            .........|
00000d20  00 00 00 00 00 00          00 00 00 00 00 00 00  |................|
*
00000e20  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00000e30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000  41 42 43 44 45 46 47 48  49 4a 4b 4c 4d 4e 4f 50  |ABCDEFGHIJKLMNOP|
00001010  51 52 53 54 55 56 57 58  59 5a 41 42 43 44 45 46  |QRSTUVWXYZABCDEF|
00001020  47 48 49 4a 4b 4c 4d 4e  4f 50 51 52 53 54 55 56  |GHIJKLMNOPQRSTUV|
...
00001fd0  53 54 55 56 57 58 59 5a  41 42 43 44 45 46 47 48  |STUVWXYZABCDEFGH|
00001fe0  49 4a 4b 4c 4d 4e 4f 50  51 52 53 54 55 56 57 58  |IJKLMNOPQRSTUVWX|
00001ff0  59 5a 41 42 43 44 45 46  47 48 49 4a 4b 4c 4d 00  |YZABCDEFGHIJKLM.|
00002000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Fig 6.0 Hexdump of virtualdiskC3_C1 with annotiation

We see that testfile.txt begins at block 4 and is 1000h (4096d) bytes long. Detailed directory list printout in *Fig 4.0* confirms the copy that was made of testfile.txt contains 4096 bytes.

3. CGS B3-B1

3.1 Implementation (B3_B1)

void mymkdir (char * path)

The *mymkdir*( ) function in filesys.c:446 creates new directories along a path. The implementation is similar to the way *myfopen*( ) navigates to a directory and creates a file there, however for each step of the navigation *mymkdir*( ) will also create new directories if they do not already exist:

450-454: Determine whether the path is relative or absolute

457-460: Prepare for tokenization of path.

462: We need to remember the current directory starting block so that newly made directories can store their parent's index.

463-474: Handle instances of '..' and '.' along the path.

477-487: If the directory doesn't exist then it needs to be made. On line 484 the function *makenewdir*( ) (filesys.c:189) functions in a similar way to *makenewfile*( ) by initializing a given directory entry with a name and index of the first block of the directory. The entry is also flagged as being in use. The FAT is modified to show that the first block of the directory is the end of a chain.

491-493: Move our location to the next directory and continue tokenization of the path.

char ** mylistdir (char * path)

The mylistdir( ) function in filesys.c:506 returns the contents of a directory as a list of strings in allocated memory. The last element of the list is always NULL. Returns NULL if the path is not found:

511-542: Navigation and tokenization process very similar to *mymkdir*( ) without any directory manipulation, since the path must exist for the function to return successfully.

545: Prepare a pointer to an array of strings for storing the contents of the directory.

546: Keep track of how many strings we've copied into memory.

548-551: For each directory entry in the current block that is in use:

> **550**: Extend the memory allocated to our list by one string pointer
>
> **551**: Copy the name of the directory entry into allocated memory and store the pointer to that memory in our list

554-559: If we have reached the end of the directory (in this implementation directories can be extended over more than one block):

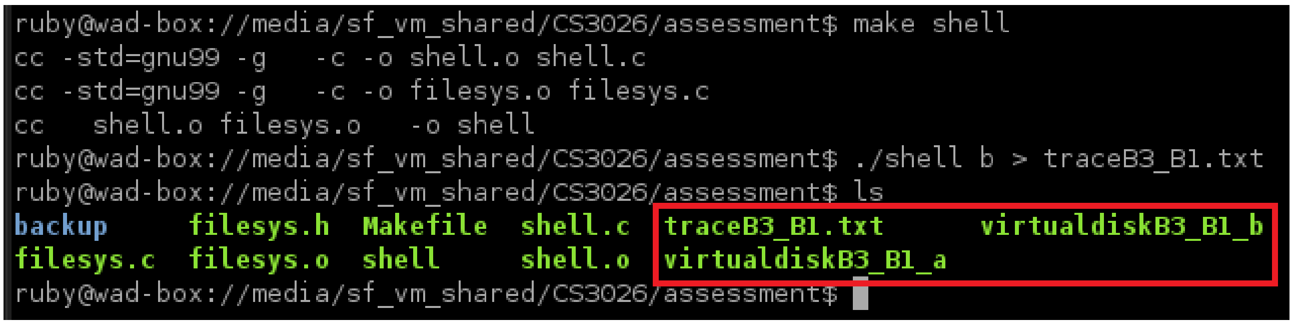
> **556-557**: Allocate space for one more string pointer in the list and have it point to NULL.

561: Else we continue and look at the next block in the FAT chain for the directory.

3.2 Instructions (B3_B1)

1. Compile with `>make shell` if not already done
2. Run with `>./shell b > traceB3_B1.txt`

The following should be created in the current directory: the virtual disk image at two different stages of execution, and a trace of the program execution:

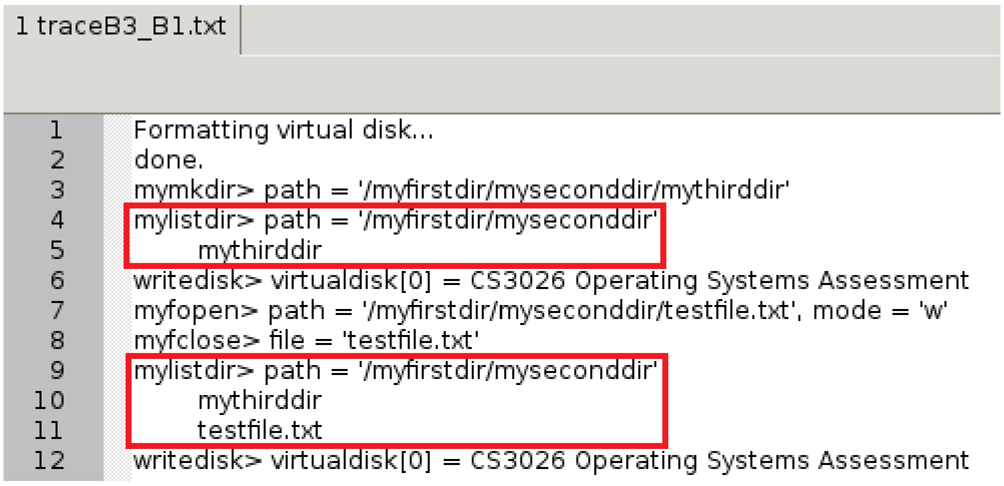


```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ make shell
cc -std=gnu99 -g   -c -o shell.o shell.c
cc -std=gnu99 -g   -c -o filesys.o filesys.c
cc   shell.o filesys.o   -o shell
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ./shell b > traceB3_B1.txt
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ls
backup     filesys.h  Makefile  shell.c  traceB3_B1.txt      virtualdiskB3_B1_b
filesys.c  filesys.o  shell     shell.o  virtualdiskB3_B1_a
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$
```

Fig 7.0 Directory contents after execution with arg 'b'

3.3 Analysis (B3_B1)

The shell program performs the functions specified in the assessment requirements for B3_B1. The trace file shows what the shell has done:

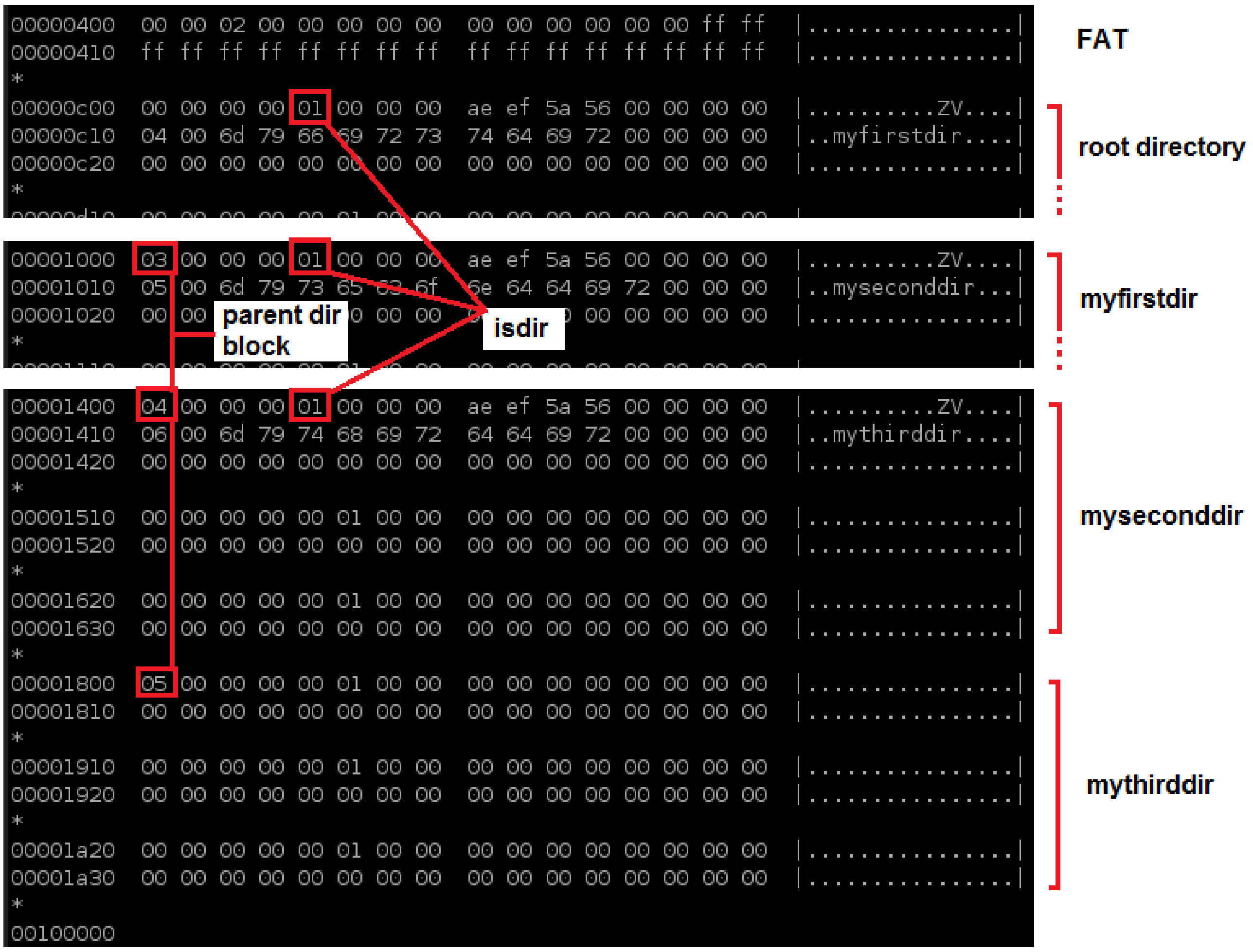


```
Formatting virtual disk...
done.
mymkdir> path = '/myfirstdir/myseconddir/mythirddir'
mylistdir> path = '/myfirstdir/myseconddir'
        mythirddir
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
myfopen> path = '/myfirstdir/myseconddir/testfile.txt', mode = 'w'
myfclose> file = 'testfile.txt'
mylistdir> path = '/myfirstdir/myseconddir'
        mythirddir
        testfile.txt
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
```

Fig 8.0 traceB3_B1.txt with highlighted list results

We can see the result of calling *mylistdir*( ) on /myfirstdir/myseconddir is consistent with the creation of the subdirectory mythirddir and the file testfile.txt.

Here is the image of the virtual disk before testfile.txt is created:



```
00000400  00 00 02 00 00 00 00 00  00 00 00 00 00 00 ff ff  |................|
00000410  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000c00  00 00 00 00 01 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00000c10  04 00 6d 79 66 69 72 73  74 64 69 72 00 00 00 00  |..myfirstdir....|
00000c20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000d10  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|

00001000  03 00 00 00 01 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00001010  05 00 6d 79 73 65 63 6f  6e 64 64 69 72 00 00 00  |..myseconddir...|
00001020  00 00             0 00 00           0 00 00 00 00 00  |................|
*
00001110  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|

00001400  04 00 00 00 01 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00001410  06 00 6d 79 74 68 69 72  64 64 69 72 00 00 00 00  |..mythirddir....|
00001420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001510  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001520  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001620  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001630  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001800  05 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001810  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001910  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001920  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001a20  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001a30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00100000
```

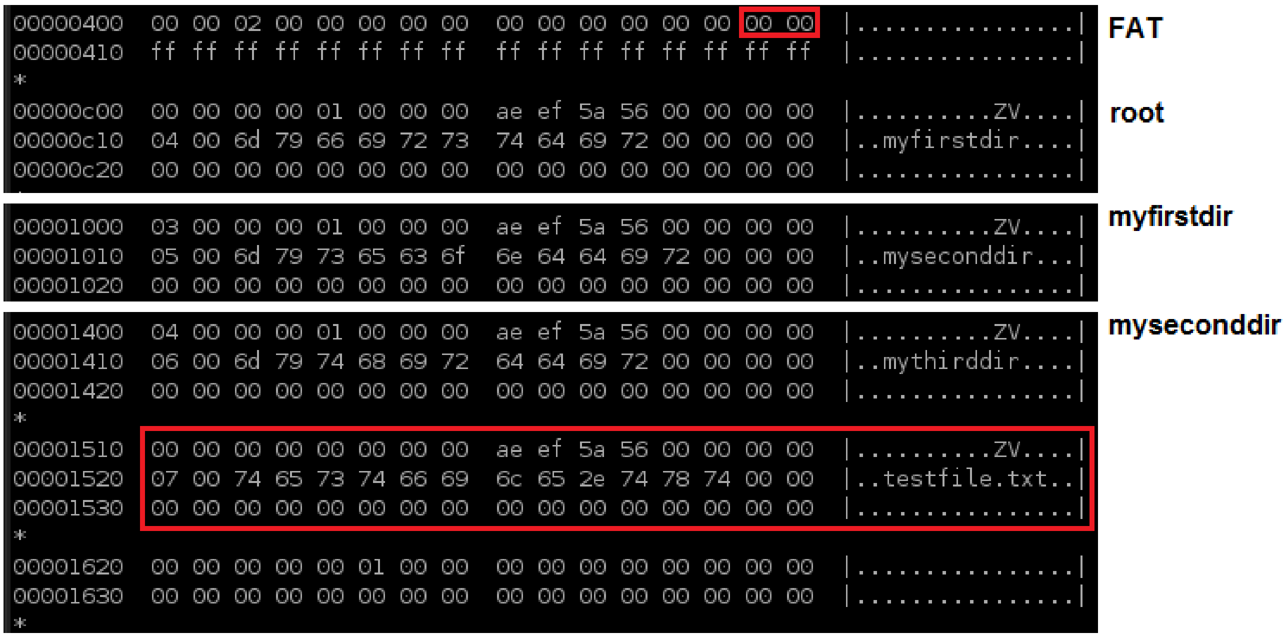
Fig 9.0 Hexdump of virtualdiskB3_B1_a with labelled directories

We see that the root contains a directory entry named myfirstdir starting at block 4, /myfirstdir contains myseconddir starting at block 5, and that /myfirstdir/myseconddir contains mythirddir starting at block 6.

This last directory contains three unused entries.

The file allocation table tells us that none of these directories occupies more than one block; all directory blocks are end-of-chain in the FAT section.

Now we look at the second image of the virtual disk taken after testfile.txt was created in /myfirstdir/myseconddir/mythirddir:



```
00000400  00 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000410  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000c00  00 00 00 00 01 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00000c10  04 00 6d 79 66 69 72 73  74 64 69 72 00 00 00 00  |..myfirstdir....|
00000c20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|

00001000  03 00 00 00 01 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00001010  05 00 6d 79 73 65 63 6f  6e 64 64 69 72 00 00 00  |..myseconddir...|
00001020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|

00001400  04 00 00 00 01 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00001410  06 00 6d 79 74 68 69 72  64 64 69 72 00 00 00 00  |..mythirddir....|
00001420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001510  00 00 00 00 00 00 00 00  ae ef 5a 56 00 00 00 00  |..........ZV....|
00001520  07 00 74 65 73 74 66 69  6c 65 2e 74 78 74 00 00  |..testfile.txt..|
00001530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001620  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001630  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

Fig 10.0 Hexdump of virtualdiskB3_B1_b with changes highlighted

The second entry in myseconddir contains the information for testfile.txt and the FAT has allocated a block for this file.

4. CGS A5-A1

4.1 Implementation (A5_A1)

void mychdir (const char * path)

The *mychdir*( ) function in filesys.c:573 changes the current directory global variable currentDirIndex, using a path. Respects absolute and relative notation ('/', '..', '.'):

577-611: Duplicates implementation from previous function that navigate to a directory using *strtok_r*( )

613: Global var currentDirIndex is updated if the function has not reported an error.

void myremove (const char * path)

The *myremove*( ) function in filesys.c:655 removes an existing file using a path. When a file is removed, a call is made to a helper function which ensures empty directory blocks are removed from the FAT chain of that directory:

660-692: Duplicates navigation functionality.

695: We need to save the location of the starting block of the directory containing the file, in case we have to alter the FAT chain.

696-702: Get the location of the directory entry for the file, abort if not found.

706-707: Flag the entry as unused so that it may be written over. Updating the modtime allows us to keep track of when it was removed if we want to try file-recovery.

710: Call to *clearFAT*( ) (filesys.c:645) recurses on the FAT chain of the file and sets all FAT entries to UNUSED.

711: Call to *tidyFAT*( ) (filesys.c:622) on the first block of the directory. This is a tail-recursive function that modifies the FAT chain of the directory to remove any unused directory blocks. Without this function, directories would grow to accommodate new entries but never shrink.

712: FAT is written to the virtual disk.

void myrmdir (cont char * path)

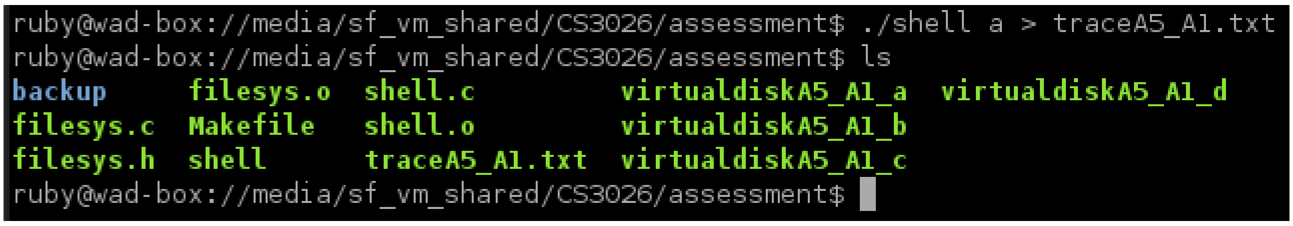
The *myrmdir*( ) function in filesys.c:719 removes an existing directory using a path. It will abort if the directory is not empty. Although directories may span multiple blocks in this implementation, part of the file removal function is to prune unused dirblocks. This makes the job of *myrmdir*( ) easier because we never have to inspect the entry array of more than one block to determine if there are any files in the directory.

Implementation is very similar to *myremove*( ) however the function iterates over the first block of the directory to ensure there are no entries in use before marking the entry of the removed directory as unused.

4.2 Instructions (A5_A1)

1. Compile with `>make shell` if not already done.
2. Run with `>./shell a > traceA5_A1.txt`

The following should be created in the current directory: four images of the virtual disk at different stages of program execution, and a trace file of the program execution:

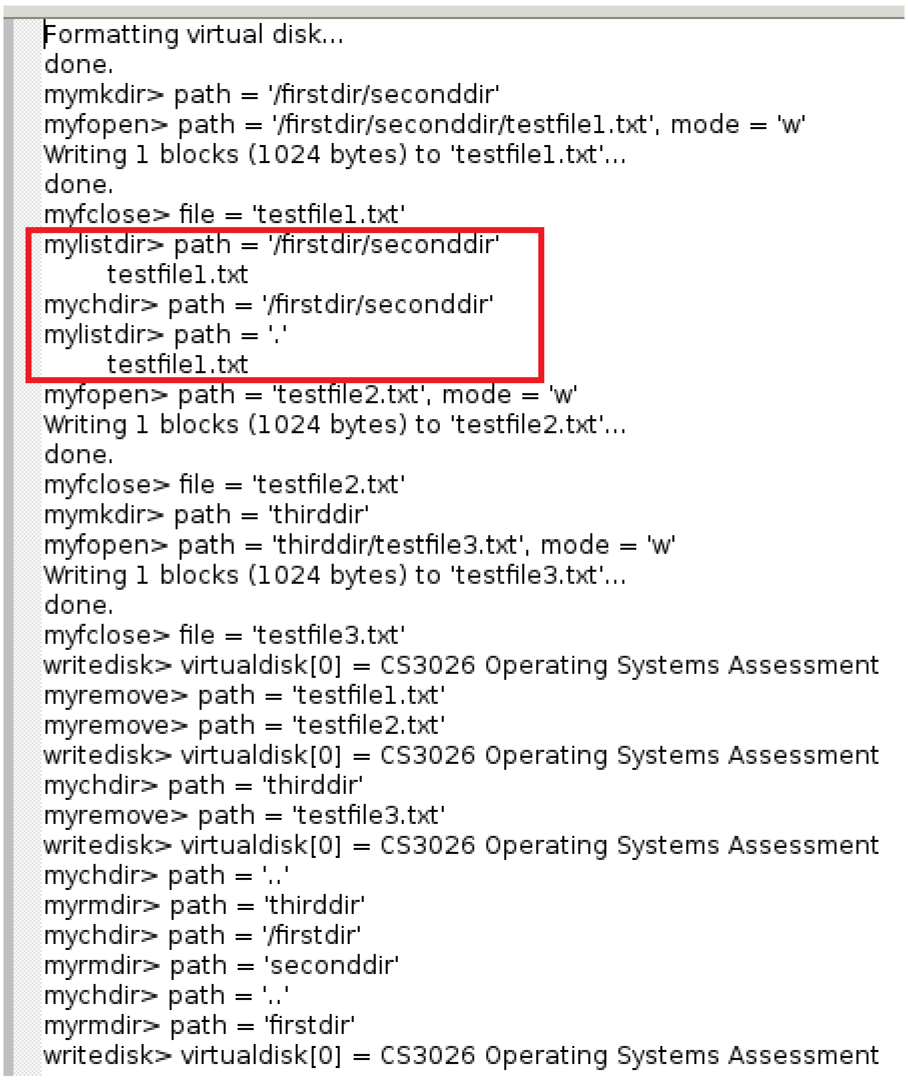


```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ./shell a > traceA5_A1.txt
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ ls
backup     filesys.o  shell.c         virtualdiskA5_A1_a  virtualdiskA5_A1_d
filesys.c  Makefile   shell.o         virtualdiskA5_A1_b
filesys.h  shell      traceA5_A1.txt  virtualdiskA5_A1_c
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ 
```

Fig 11.0 Directory contents after execution with argument 'a'

4.3 Analysis (A5_A1)

The shell program performs the functions specified in the assessment requirements for A5_A1. The trace file shows what the shell has done:



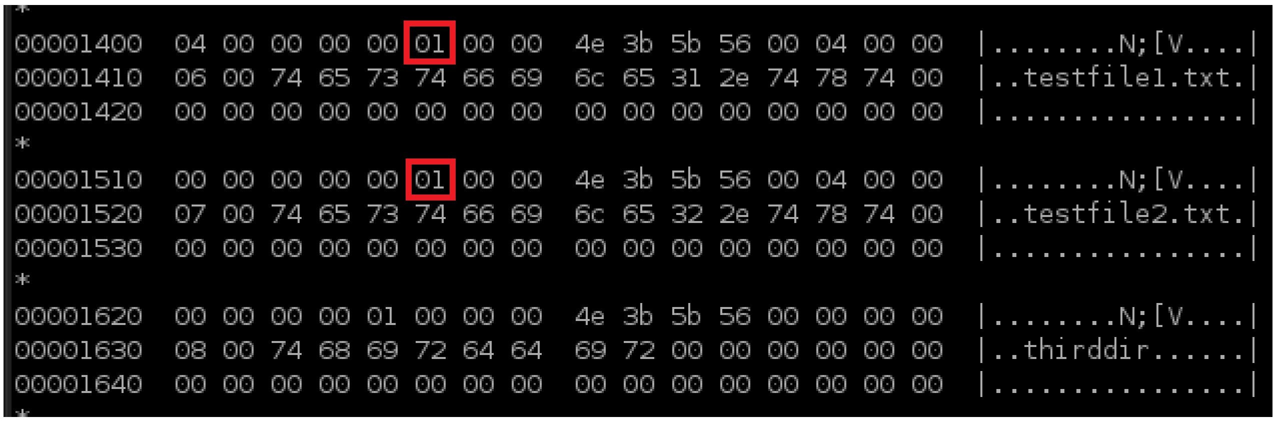
```
Formatting virtual disk...
done.
mymkdir> path = '/firstdir/seconddir'
myfopen> path = '/firstdir/seconddir/testfile1.txt', mode = 'w'
Writing 1 blocks (1024 bytes) to 'testfile1.txt'...
done.
myfclose> file = 'testfile1.txt'
mylistdir> path = '/firstdir/seconddir'
        testfile1.txt
mychdir> path = '/firstdir/seconddir'
mylistdir> path = '.'
        testfile1.txt
myfopen> path = 'testfile2.txt', mode = 'w'
Writing 1 blocks (1024 bytes) to 'testfile2.txt'...
done.
myfclose> file = 'testfile2.txt'
mymkdir> path = 'thirddir'
myfopen> path = 'thirddir/testfile3.txt', mode = 'w'
Writing 1 blocks (1024 bytes) to 'testfile3.txt'...
done.
myfclose> file = 'testfile3.txt'
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
myremove> path = 'testfile1.txt'
myremove> path = 'testfile2.txt'
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
mychdir> path = 'thirddir'
myremove> path = 'testfile3.txt'
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
mychdir> path = '..'
myrmdir> path = 'thirddir'
mychdir> path = '/firstdir'
myrmdir> path = 'seconddir'
mychdir> path = '..'
myrmdir> path = 'firstdir'
writedisk> virtualdisk[0] = CS3026 Operating Systems Assessment
```

Fig 12.0 contents of trace_A5_A1.txt

In the highlighted section we see that the result of calling *mylistdir*( ) is consistent with the change in directory by the call to *mychdir*( ).

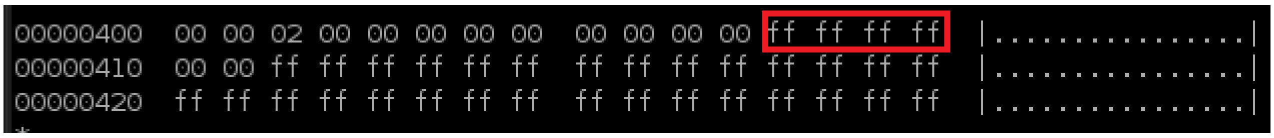
When remove functions are called in this implementation, the data contained in files and directory entries are not cleared, instead the relevant blocks are marked as unused in the FAT and the corresponding directory entries are also flagged as unused, allowing them to be overwritten by future calls to *mymkdir*( ), *myfopen*( ) and *myfputc*( ).

Consider the hexdump of the virtual disk state after testfile1.txt and testfile2.txt have been removed (virtualdiskA5_A1_b):



```
00001400  04 00 00 00 00 01 00 00  4e 3b 5b 56 00 04 00 00  |........N;[V....|
00001410  06 00 74 65 73 74 66 69  6c 65 31 2e 74 78 74 00  |..testfile1.txt.|
00001420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001510  00 00 00 00 00 01 00 00  4e 3b 5b 56 00 04 00 00  |........N;[V....|
00001520  07 00 74 65 73 74 66 69  6c 65 32 2e 74 78 74 00  |..testfile2.txt.|
00001530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001620  00 00 00 00 01 00 00 00  4e 3b 5b 56 00 00 00 00  |........N;[V....|
00001630  08 00 74 68 69 72 64 64  69 72 00 00 00 00 00 00  |..thirddir......|
00001640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Fig 13.1 Disk area corresponding to seconddir in virtualdiskA5_A1_b

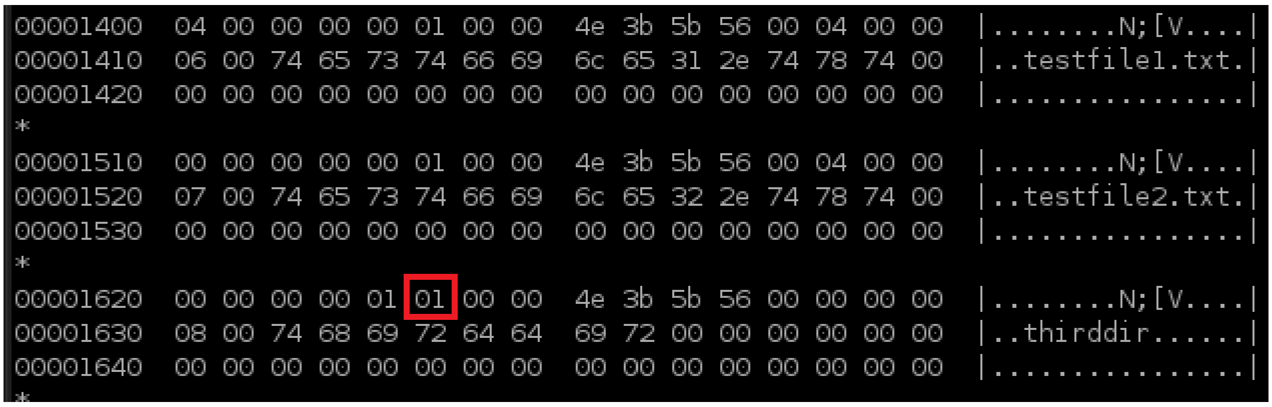


```
00000400  00 00 02 00 00 00 00 00  00 00 00 00 ff ff ff ff  |................|
00000410  00 00 ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000420  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

Fig 13.2 Disk area corresponding to the FAT in virtualdiskA5_A1_b

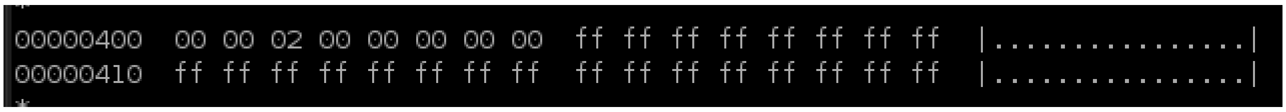
In the seconddir section we can see that both files have been flagged as unused. FAT[6] and FAT[7] are also unused. These files would now be overwritten if any directory or file was made in seconddir. *NB*: the FAT entry at index 8 corresponds to testfile3.txt which has not yet been removed.

Looking at state d, when testfile3.txt and all subdirectories have been removed, we see that the relevant disk areas have been modified appropriately:



```
00001400  04 00 00 00 00 01 00 00  4e 3b 5b 56 00 04 00 00  |........N;[V....|
00001410  06 00 74 65 73 74 66 69  6c 65 31 2e 74 78 74 00  |..testfile1.txt.|
00001420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001510  00 00 00 00 00 01 00 00  4e 3b 5b 56 00 04 00 00  |........N;[V....|
00001520  07 00 74 65 73 74 66 69  6c 65 32 2e 74 78 74 00  |..testfile2.txt.|
00001530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001620  00 00 00 00 01 01 00 00  4e 3b 5b 56 00 00 00 00  |........N;[V....|
00001630  08 00 74 68 69 72 64 64  69 72 00 00 00 00 00 00  |..thirddir......|
00001640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Fig 14.1 Area corresponding to seconddir in virtualdiskA5_A1_d



```
00000400  00 00 02 00 00 00 00 00  ff ff ff ff ff ff ff ff  |................|
00000410  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

Fig 14.2 Area corresponding to the FAT in virtualdiskA5_A1_d

5. Additional

This section will briefly discuss features of the file management program that are not specified or tested for in the assessment criteria.

Dynamic directory size

The current implementation allows directories to extend and shrink when files are added or removed.

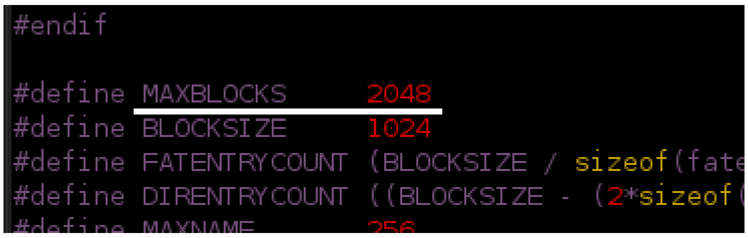
Running ./shell with argument 'x' will make enough files to cause the root directory to be extended over three blocks, then remove them.

The virtual disk image is saved at three points to show how the FAT is managed during file removal.

The drawbacks of the current implementation

Variable virtual disk size

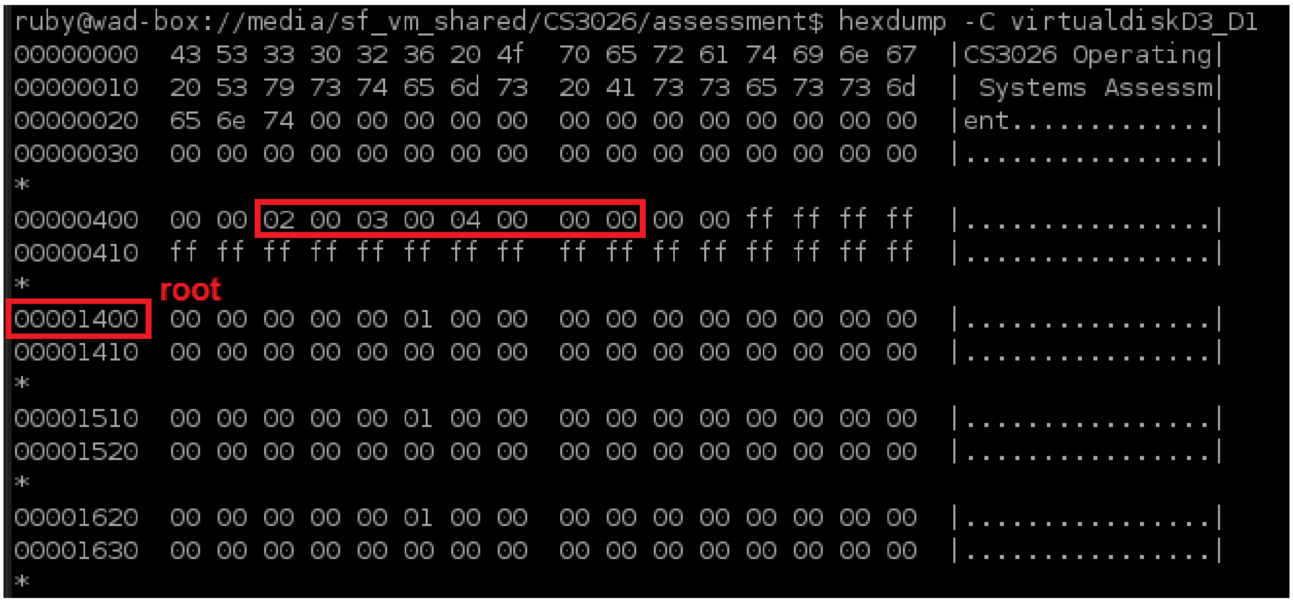
All functions were implemented to handle a virtual disk space of greater than just 1024*1024 bytes. If we change the constant MAXBLOCKS to double the size in the assessment specifications, the disk is still formatted correctly:



```
#endif

#define MAXBLOCKS       2048
#define BLOCKSIZE       1024
#define FATENTRYCOUNT   (BLOCKSIZE / sizeof(fate
#define DIRENTRYCOUNT   ((BLOCKSIZE - (2*sizeof(
#define MAXNAME         256
```

Fig 15.0



```
ruby@wad-box://media/sf_vm_shared/CS3026/assessment$ hexdump -C virtualdiskD3_D1
00000000  43 53 33 30 32 36 20 4f  70 65 72 61 74 69 6e 67  |CS3026 Operating|
00000010  20 53 79 73 74 65 6d 73  20 41 73 73 65 73 73 6d  | Systems Assessm|
00000020  65 6e 74 00 00 00 00 00  00 00 00 00 00 00 00 00  |ent.............|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  00 00 02 00 03 00 04 00  00 00 00 00 ff ff ff ff  |................|
00000410  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00001400  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001510  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001520  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001620  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00  |................|
00001630  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

Fig 16.0

We see the chain for the FAT itself spans 4 blocks now to be able to address all 2048 blocks, and that the root begins at 0x1400 which corresponds to block 5.