

# Operating Systems

## CSCI 5806

Spring Semester 2020 — CRN 21176 / 26762

---

Term Project — Step 0 — Preparation

Target completion date: Friday, January 17, 2020

### Goals

- Learn about the five basic low-level UNIX file access functions
- Create two functions to display memory regions

### Details

UNIX has five basic file access system calls:

- **int open(char \*fn)**  
Open the file whose name is given. The filename can be either a relative or absolute path. The function returns a nonnegative integer — the *file handle* — if the function succeeds, or  $-1$  if the function fails.
- **void close(int fd)**  
Close the file whose handle is given.
- **ssize\_t read(int fd, void \*buf, size\_t count)**  
Reads the given number of bytes from the given open file, placing the bytes in the given buffer. The location of the first byte read is given by the file's *cursor*; subsequent bytes are read sequentially and the cursor is advanced to the end of the block that is read.
- **ssize\_t write(int fd, void \*buf, size\_t count)**  
Writes the given number of bytes to the given file, starting at the cursor. Bytes are written sequentially and the cursor is advanced to the end of the written block. Bytes to be written are located in the given buffer.
- **off\_t lseek(int fd, off\_t offset, int anchor)**  
Move the cursor of the given file to the given location, based on the offset and anchor values.

Look up these functions; I recommend emulating this set of functions, as they will simplify the process of accessing the proper bytes from the filesystem.

You should also write two functions:

- **void displayBufferPage(uint8\_t \*buf, uint32\_t count, uint32\_t start, uint64\_t offset)**  
Displays up to 256 bytes from the given buffer.  
Parameters:
  - **buf** — buffer to be displayed
  - **count** — number of bytes to be displayed, max of 256
  - **start** — location within a 256-byte block of the first byte of the buffer (see examples)
  - **offset** — location within the disk structure of the start of the buffer.

- **void displayBuffer(uint8\_t \*buf, uint32\_t count, uint64\_t offset)**  
Displays **count** bytes in the given buffer. The **offset** parameter is used to display the location of the buffer within the disk structure. This should just call **displayBufferPage()** repeatedly to display all of the bytes in the buffer.

The functions should display each 256-byte block in two forms: hexadecimal and character. A byte with offset *off* within the 256-byte block should be displayed in hexadecimal if  $start \leq off < start + count$  and should also be displayed as a character if it is printable (**isprint()** returns true).

These display functions are for your use in building your program. They aren't part of the final product, but they may prove helpful in debugging steps along the way.

Format the output to make it easily readable. Here are some examples; your code doesn't have to replicate this format exactly, this is just to give you an idea of what the output looks like.

### ►Example 1

This is the VDI header from a sample VDI file. It's a 400-byte structure found at the start of a VDI file. A 400-byte buffer was read and displayed using the function call

```
displayBuffer(headerBuf,400,0);
```

---

```

1  Offset: 0x0
2      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
3      +-----+-----+
4  00|3c 3c 3c 20 4f 72 61 63 6c 65 20 56 4d 20 56 69|00|<<< Oracle VM Vi|
5  10|72 74 75 61 6c 42 6f 78 20 44 69 73 6b 20 49 6d|10|rtualBox Disk Im|
6  20|61 67 65 20 3e 3e 3e 0a 00 00 00 00 00 00 00 20|20|age >>>
7  30|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 30|
8  40|7f 10 da be 01 00 01 00 90 01 00 00 02 00 00 00|40|
9  50|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 50|
10 60|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60|
11 70|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 70|
12 80|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80|
13 90|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 90|
14 a0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 a0|
15 b0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 b0|
16 c0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 c0|
17 d0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 d0|
18 e0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 e0|
19 f0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0|
20      +-----+-----+
21
22 Offset: 0x100
23      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f    0...4...8...c...
24      +-----+-----+
25 00|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|
26 10|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10|
27 20|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20|
28 30|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 30|
29 40|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40|
30 50|00 00 00 00 00 00 00 10 00 00 00 20 00 00 00 00 50|
31 60|00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 60|
32 70|00 00 00 08 00 00 00 00 00 00 10 00 00 00 00 00 70|
33 80|80 00 00 00 80 00 00 00 07 07 d8 e6 24 6b c1 46|80|          $k F
34 90|
35 a0|
36 b0|
37 c0|
38 d0|
39 e0|
40 f0|
41      +-----+-----+

```

---

## ►Example 2

This is the first 256 bytes of the VDI file's translation map. There is a field in the VDI header that determines the map's location within the VDI file; in this case the map is  $1\,048\,576 = (10\,000)_{16}$  bytes from the start. It was called via

```
displayBufferPage(pageBuf, sizeof(pageBuf), 0, f->header->mapOffset);
```

---

```

1 Offset: 0x100000
2   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f   0...4...8...c...
3   +-----+-----+-----+-----+-----+-----+
4 00|00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00|00|
5 10|04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00|10|
6 20|08 00 00 00 09 00 00 00 0a 00 00 00 0b 00 00 00|20|
7 30|0c 00 00 00 0d 00 00 00 0e 00 00 00 0f 00 00 00|30|
8 40|10 00 00 00 11 00 00 00 12 00 00 00 13 00 00 00|40|
9 50|14 00 00 00 15 00 00 00 16 00 00 00 17 00 00 00|50|
10 60|18 00 00 00 19 00 00 00 1a 00 00 00 1b 00 00 00|60|
11 70|1c 00 00 00 1d 00 00 00 1e 00 00 00 1f 00 00 00|70|
12 80|20 00 00 00 21 00 00 00 22 00 00 00 23 00 00 00|80|   !   "   #
13 90|24 00 00 00 25 00 00 00 26 00 00 00 27 00 00 00|90|$   %   &   '
14 a0|28 00 00 00 29 00 00 00 2a 00 00 00 2b 00 00 00|a0|(   )   *   +
15 b0|2c 00 00 00 2d 00 00 00 2e 00 00 00 2f 00 00 00|b0|,   -   .   /
16 c0|30 00 00 00 31 00 00 00 32 00 00 00 33 00 00 00|c0|0   1   2   3
17 d0|34 00 00 00 35 00 00 00 36 00 00 00 37 00 00 00|d0|4   5   6   7
18 e0|38 00 00 00 39 00 00 00 3a 00 00 00 3b 00 00 00|e0|8   9   :   ;
19 f0|3c 00 00 00 3d 00 00 00 3e 00 00 00 3f 00 00 00|f0|<  =  >  ?
20   +-----+-----+-----+-----+-----+-----+

```

---

### ►Example 3

The third example shows the partition table from the sample VDI file. The table — at least for MBR-based disks — is always in the first 512-byte sector at offset 446, or 190 bytes into the second 256-byte block. This example illustrates only showing bytes from *start* through *start + count*, as the first 190 bytes are skipped as are the last two. The function call used was

```
displayBufferPage(pageBuf,64,190,256);
```

---

```

1 Offset: 0x100
2   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f   0...4...8...c...
3   +-----+-----+-----+-----+-----+-----+
4 00|                                     |00|
5 10|                                     |10|
6 20|                                     |20|
7 30|                                     |30|
8 40|                                     |40|
9 50|                                     |50|
10 60|                                     |60|
11 70|                                     |70|
12 80|                                     |80|
13 90|                                     |90|
14 a0|                                     |a0|
15 b0|                                     |b0|
16 c0|21 00 83 51 01 10 00 08 00 00 00 00 f8 03 00 00 00 00|c0|!  Q
17 d0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|d0|
18 e0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|e0|
19 f0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|f0|
20   +-----+-----+-----+-----+-----+-----+

```

---