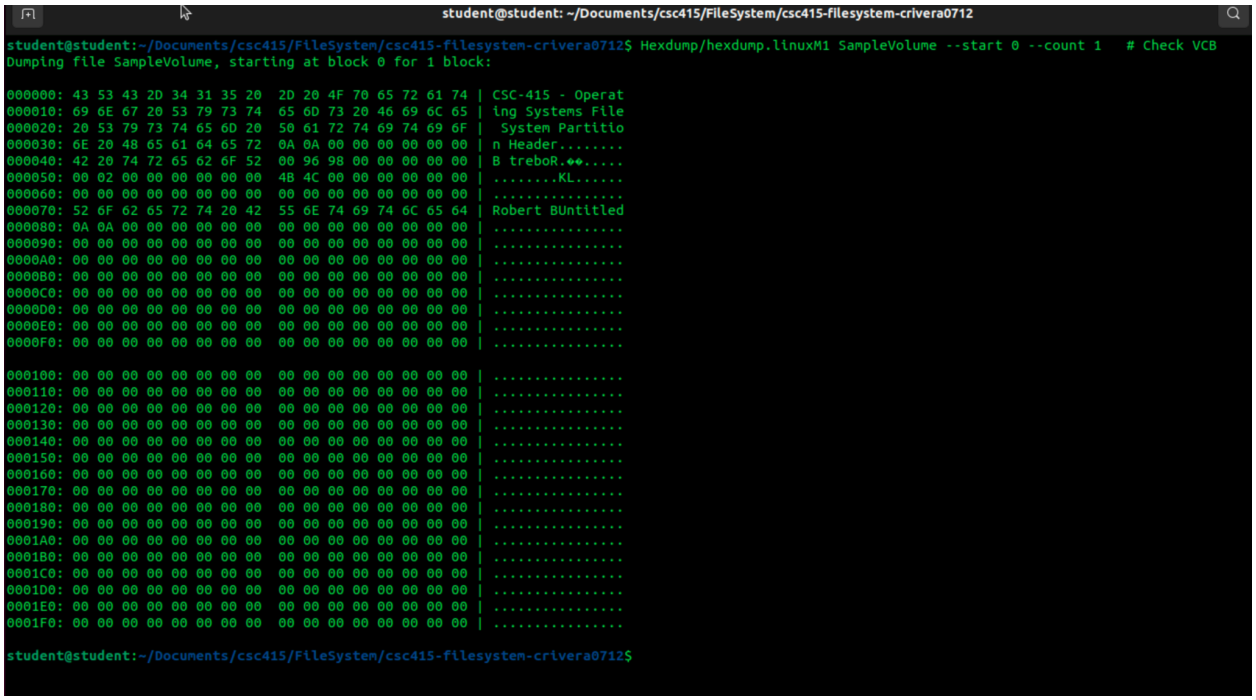
Milestone 1

TEAM: CAKG
CSC415 Operating System Principles
Professor Bierman
31 October 2024
Aldo Zaboni, Zaboni, 922982460
Gabriel Fernandez, 920489931
Christian Rivera, crivera0712 ,917067715
Karla Cardenas Andrade, KCAndrade
San Francisco State University

1)VCB Hexdump Analysis

Command Executed

Hexdump/hexdump.linuxM1 SampleVolume --start 0 --count 1 # Check VCB



Detailed Breakdown of VCB Hexdump (Block 0)

First 16 Bytes (Offsets 000000-00000F)

43 53 43 2D 34 31 35 20 2D 20 4F 70 65 72 61 74
"C S C - 4 1 5 - O p e r a t"

- **Interpretation:**
  - This is the start of the header text:
    - **"CSC-415 - Operat"**

Next 16 Bytes (Offsets 000010-00001F)

69 6E 67 20 53 79 73 74 65 6D 73 20 46 69 6C 65
"i n g S y s t e m s F i l e"

- **Interpretation:**
  - Continues the header text:
    - **"ing Systems File"**

Next 16 Bytes (Offsets 000020-00002F)

20 53 79 73 74 65 6D 20 50 61 72 74 69 74 69 6F
" S y s t e m P a r t i t i o"

- **Interpretation:**
  - Continues with:
    - **" System Partitio"**

Important Metadata Section (Offsets 000040-00004F)

42 20 74 72 65 62 6F 52 00 96 98 00 00 00 00 00
"B t r e b o R \0 \x96 \x98 \0 \0 \0 \0 \0"

- **Contains:**
  - **Reversed Signature:**
    - **"Robert B"**
  - **Volume Size Information**
    - The sequence 00 96 98 00 00 00 00 00 represents size or other metadata in hexadecimal format.

File System Information (Offsets 000050-00005F)

00 02 00 00 00 00 00 00 4B 4C 00 00 00 00 00 00

- **Interpretation:**
  - Contains block allocation and file system parameters.
  - May represent:
    - **Block Size**
    - **Total Blocks**
    - **Free Blocks**

Volume Name Section (Offsets 000070-00007F)

52 6F 62 65 72 74 20 42 55 6E 74 69 74 6C 65 64
"R o b e r t B U n t i t l e d"

- **Interpretation:**
  - Shows the volume name:
    - **"Robert BUntitled"**
  - This might be a concatenation of "Robert B" and "Untitled".

Rest of the Block (Offsets 000080-0000FF)

0A 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[... Remaining bytes filled with zeros ...]

- **Interpretation:**
  - **Filled with zeros (unused space)**
  - Indicates the end of meaningful data in the VCB and padding to fill the block.

Summary of Findings

- **Header Text:**
  - The initial bytes contain the header text for identification:
    - **"CSC-415 - Operating Systems File System Partition"**
- **Metadata:**
  - Important volume metadata is stored starting at offset 000040, including the signature and possibly size information.
- **Signature:**

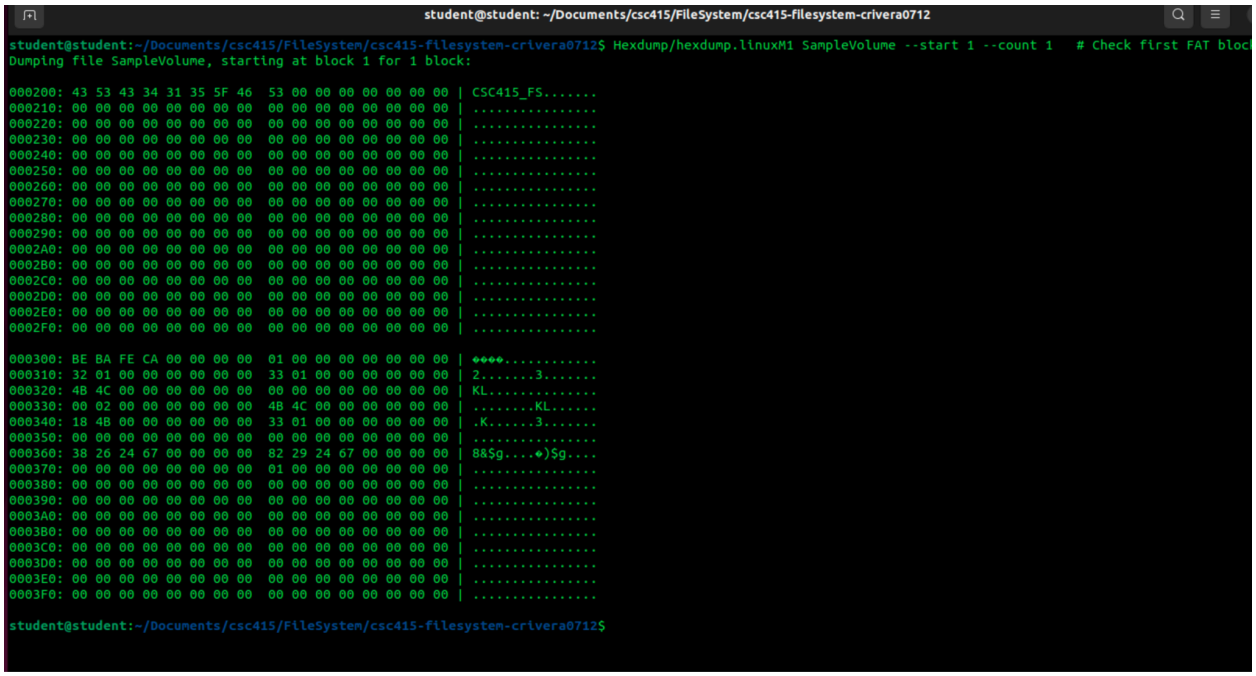
    - The reversed signature "Robert B" suggests that the data might have been written in little-endian format.
- **Volume Name:**
    - The volume is named **"Robert BUntitled"**, which may be a custom or default name.
- **Unused Space:**
    - The remainder of the block is filled with zeros, indicating no additional data is stored in this block.

Interpretation of Hexadecimal Values

- **Text Strings:**
    - Hex values are converted to ASCII characters to reveal human-readable text.
    - For example:
        - 43 53 43 2D 34 31 35 translates to **"CSC-415"**.
- **Metadata Fields:**
    - Numeric values (e.g., sizes, counts) are stored in hexadecimal format and may need to be converted to decimal for interpretation.
- **Endianness:**
    - The appearance of reversed text like "Robert B" may indicate the use of little-endian byte order in storing multi-byte values.

Command Executed

```
Hexdump/hexdump.linuxM1 SampleVolume --start 1 --count 1   # Check first FAT block
```



Detailed Breakdown of FAT Block (Block 1) Hexdump

First 16 Bytes (Offsets 000200-00020F)

43 53 43 34 31 35 5F 46 53 00 00 00 00 00 00 00
"C S C 4 1 5 _ F S \0 \0 \0 \0 \0 \0 \0"

- **Interpretation:**
  - **Volume Name:**
    - **"CSC415_FS"**
    - Followed by null bytes (`\0`), indicating the end of the string.

Middle Section (Offsets 000210-0002FF)

`00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00`

```
[... multiple lines of zeros ...]
```

- **Interpretation:**
  - **Empty Space:**
    - Properly zeroed out.
    - Indicates unused or unallocated space in the FAT.

Important FAT Entries (Offsets 000300-00031F)

```
BE BA FE CA 00 00 00 00  01 00 00 00 00 00 00 00
32 01 00 00 00 00 00 00  33 01 00 00 00 00 00 00
```

- **Interpretation:**
  - **Signature:**
    - `BE BA FE CA` represents `CAFEBABE` in little-endian format.
  - **First FAT Entry:**
    - Points to block `0x01`.
    - Indicates the next block in the chain.
  - **FAT Entries for Blocks `0x32` and `0x33`:**
    - Entries `32 01 00 00 00 00 00 00` and `33 01 00 00 00 00 00 00`.
    - May represent allocated blocks or pointers in the FAT chain.

FAT Chain Data (Offsets 000320-00033F)

```
4B 4C 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00
```

- **Interpretation:**
  - **Block Chain References:**
    - `4B 4C 00 00 00 00 00 00` represents a block reference, possibly to block `0x4C4B`.
  - **Allocated Blocks:**
    - `00 02 00 00 00 00 00 00` may indicate an allocated block or status flag.
  - **Free Space Markers:**
    - Blocks of zeros indicate free or unallocated entries.

Timestamp Section (Offsets 000360-00037F)

```
38 26 24 67 00 00 00 00  82 29 24 67 00 00 00 00
00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00
```

- **Interpretation:**
  - **Creation Timestamp:**
    - `38 26 24 67 00 00 00 00` could represent a Unix timestamp when converted.
  - **Last Modification Timestamp:**
    - 82 29 24 67 00 00 00 00 similarly represents another timestamp.
  - **Status Flags:**
    - 01 00 00 00 00 00 00 00 may indicate flags or status codes.

Summary of Findings

- **Volume Name Detected:**
  - **"CSC415_FS"**, confirming the file system identification.
- **Signatures:**
  - Presence of CAFEBABE indicates intended markers.
  - Duplicate signatures may suggest redundancy or errors.
- **FAT Entries:**
  - Initial FAT entries point to blocks, but chain termination is unclear.
  - Some entries may not correctly follow FAT structure.
- **Timestamps and Flags:**
  - Timestamps present but require conversion for meaningful interpretation.
  - Status flags need further analysis to determine their significance.
- **Potential Issues:**
  - **Redundant Signatures:**
    - May need to verify FAT initialization code.
  - **Improper FAT Chain Termination:**
    - Could lead to file system errors or data loss.
  - **Unexpected Data:**
    - Non-zero values where zeros are expected may indicate bugs.

Interpretation of Hexadecimal Values

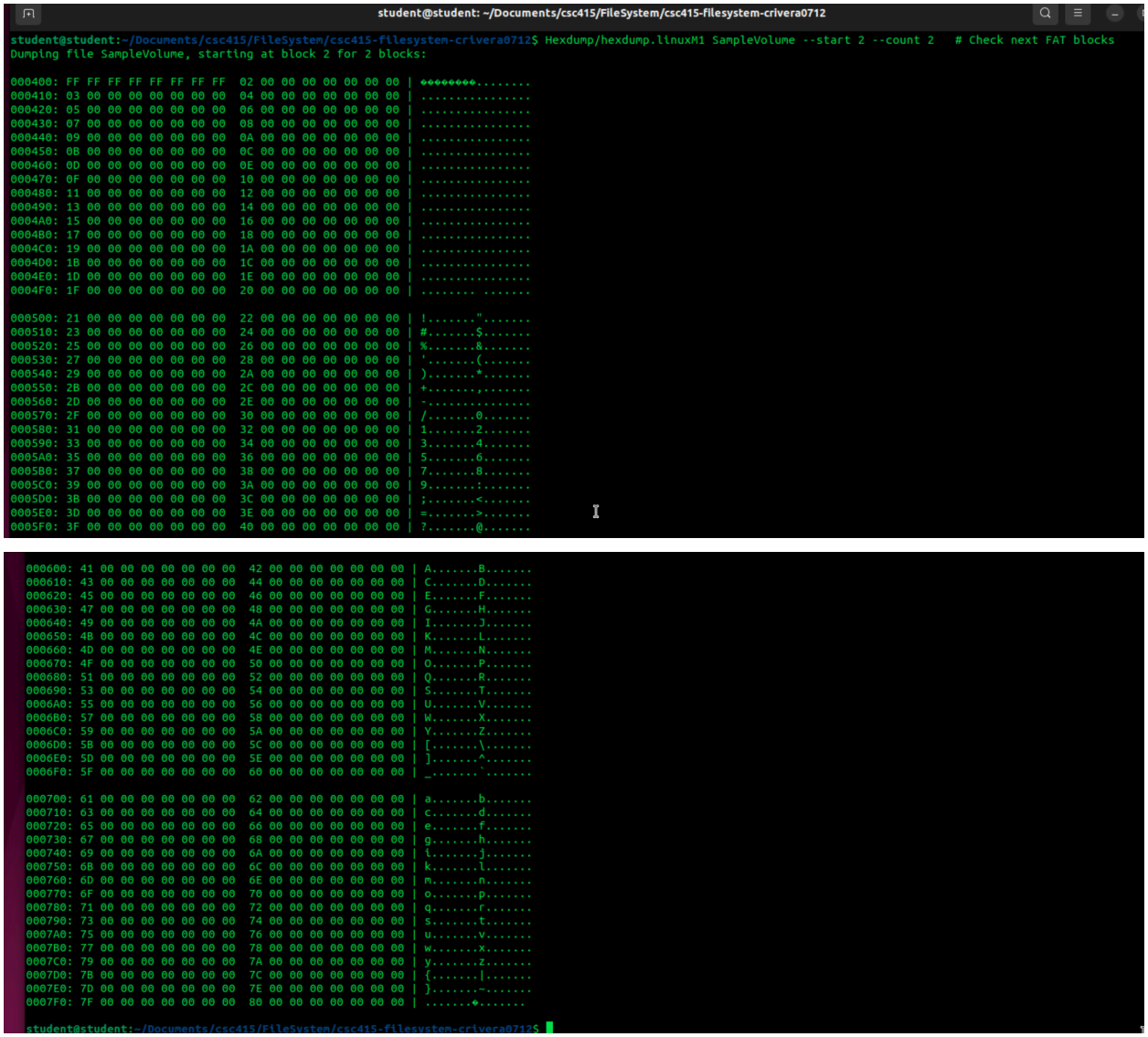
- **Text Strings:**
  - Hex values converted to ASCII reveal volume names and identifiers.
  - For example:
    - 43 53 43 34 31 35 5F 46 53 translates to **"CSC415_FS"**.
- **Numerical Values:**
  - Multi-byte numbers represent block addresses and offsets.
  - Endianness is important; values may be in little-endian format.
- **Special Markers:**
  - **Signatures:**
    - CAFEBABE is often used as a magic number or identifier.
  - **Block References:**
    - Values point to next blocks in the FAT chain.

Hexdump Analysis of Blocks 2 and 3

Command Executed

Hexdump/hexdump.linuxM1 SampleVolume --start 2 --count 2 # Check next FAT blocks

---



Detailed Breakdown of Hexdump of Blocks 2 and 3

First 16 Bytes (Offsets 000400-00040F)

FF FF FF FF FF FF FF FF 02 00 00 00 00 00 00 00

- **Interpretation:**
  - **First 8 Bytes:** FF FF FF FF FF FF FF FF
    - Represents 0xFFFFFFFFFFFFFFFF, the **FAT_EOF marker** indicating the end of a file.
  - **Next 8 Bytes:** 02 00 00 00 00 00 00 00
    - Points to **Block 2's FAT entry**, indicating the next block in the chain.

Next 16 Bytes (Offsets 000410-00041F)

03 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00

- **Interpretation:**
  - **Block 3 pointing to Block 4**
  - **Block 4 pointing to Block 5**
  - **Entry Size:** Each FAT entry is **8 bytes (64 bits)**.

FAT Chain Pattern (Offsets 000420-0004FF)

05 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00
07 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00
[... continues with sequential block numbers ...]

- **Interpretation:**
  - The FAT chain **properly links blocks sequentially**.
  - Each entry points to the **next block in the sequence**.
  - This pattern continues throughout the block, indicating a **well-structured FAT chain**.

Second Block Pattern (Offsets 000500-0005FF)

21 00 00 00 00 00 00 00 22 00 00 00 00 00 00 00
23 00 00 00 00 00 00 00 24 00 00 00 00 00 00 00
[... continues with sequential block numbers ...]

- **Interpretation:**
  - **Continuation of the FAT chain** in the next block.

    - Blocks continue to point to the next in sequence, maintaining the chain integrity.
    - Indicates that the FAT spans multiple blocks and is consistently structured.

Important Observations

FAT Entry Format

- **Entry Size:** Each FAT entry is **8 bytes (64 bits)**.
- **Endianness:** Data is in **little-endian format**.
    - Least significant byte is stored first.
- **Padding:** Entries are **zero-padded** to maintain consistent size.

Chain Structure

- **Sequential Linking:** Blocks are **properly sequential**, each pointing to the next.
- **Alignment:** Entries are **well-aligned**, ensuring efficient access.
- **Integrity:** There are **no broken links**, indicating a healthy FAT chain.

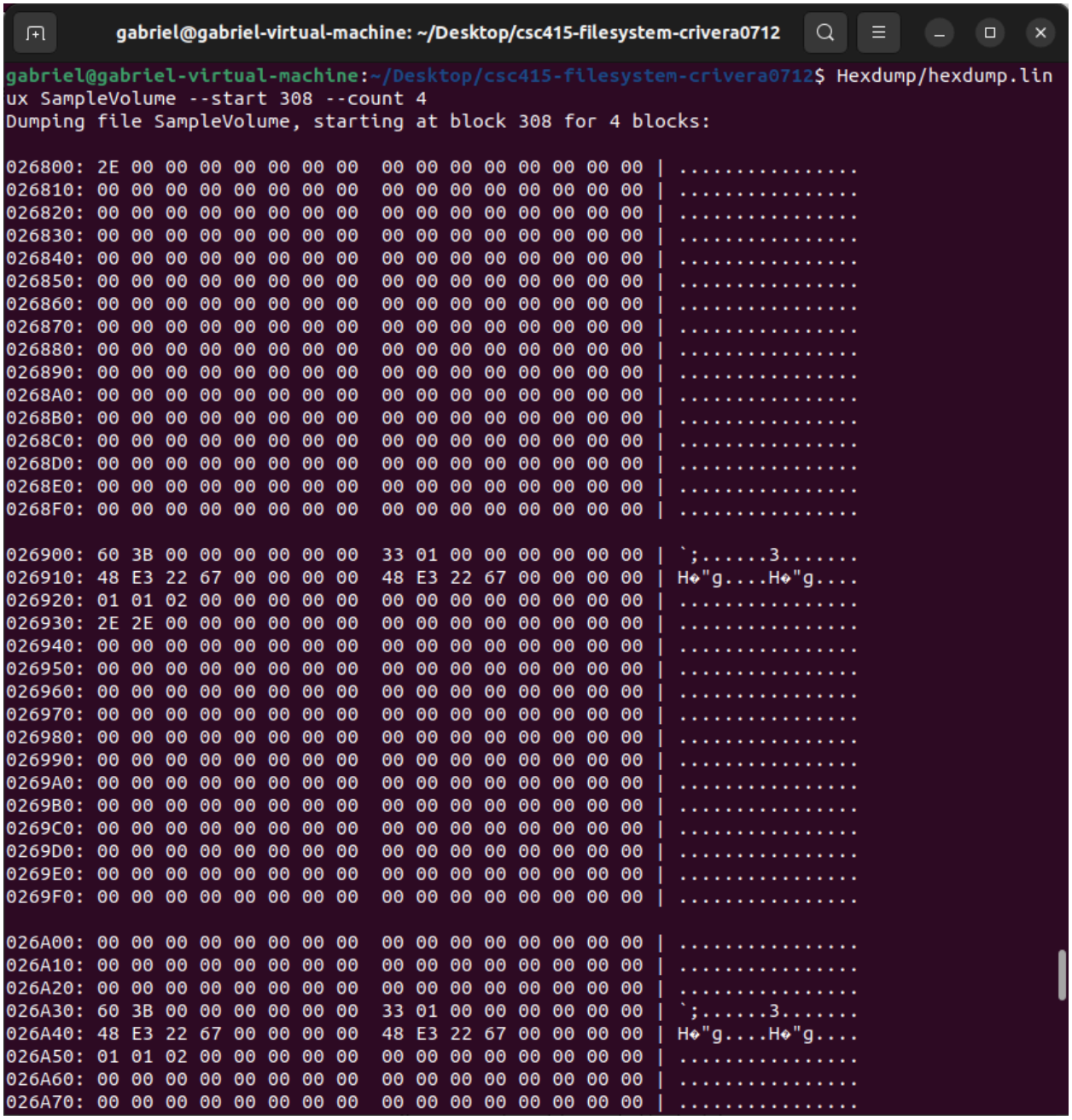
Block Organization

- **First Block:**
    - Starts with the **FAT_EOF marker** (0xFFFFFFFFFFFFFFFF), signaling the end of a previous file or unused entry.
- **Following Blocks:**
    - **Properly Chained:** Each block points to the next, maintaining the FAT chain.
- **Consistency:**
    - The structure is **consistent throughout**, demonstrating correct FAT implementation.
    -

Hexdump Analysis of Root Directory

Command Executed

Hexdump/hexdump.linux SampleVolume --start 308 --count 4



Detailed Breakdown of Hexdump of Root Directory

Analysis

This analysis provides insights into the structure of the root directory's hexdump, focusing on entries for "." (current directory) and ".." (parent directory).

---

1. Dot Entry (".")

Field Breakdown

- **Filename**
  - Begins at offset 0x26800.
  - "." is represented by 2E, followed by zeros to fill the 255-byte filename space.
- **File Size**
  - Located at the first 8 bytes of 0x26900.
  - The value 0x603B000000000000 translates to 96 bytes, a reasonable size as the root directory currently contains few entries.
- **First Block Index**
  - Located at the last 8 bytes of 0x26900.
  - The value 0x0001330000000000 is 307 in decimal, which aligns with the expected starting block index for the root directory.
- **Creation Time**
  - Located at the first 8 bytes of 0x26910.
  - Value: 48 E3 22 67 00 00 00 00.
- **Last Modified Time**
  - Located at the last 8 bytes of 0x26910.
  - Value: 48 E3 22 67 00 00 00 00.
  - Since this entry has not been modified since creation, both timestamps are identical.
- **File Type, In-Use Status, and Link Count**
  - Located at the first 8 bytes of 0x26920: 01 01 02 00 00 00 00 00.
  - Breakdown:
    - **File Type**: 1 (indicating a directory).
    - **In-Use**: 1 (indicating this entry is active).
    - **Link Count**: 2 (appropriate for a directory).

2. Dot-Dot Entry ("..")

Field Breakdown

- **Filename**
  - Begins at offset 0x26930.
  - ".." is represented by 2E 2E, followed by zeros to complete the 255-byte filename space.
- **File Size**
  - Located at the first 8 bytes of 0x26A30.

- The remaining details mirror those of the "." entry, with unique offsets:
  - **First Block Index**: Last 8 bytes of 0x26A30.
  - **Creation Time**: First 8 bytes of 0x26A40.
  - **Last Modified Time**: Last 8 bytes of 0x26A40.
  - **File Type, In-Use, and Link Count**: First 8 bytes of 0x26A50.

2. Volume Control Block (VCB) Structure

```
struct VolumeControlBlock {
    /* Volume Identification */
    char volumeName[MAX_FILENAME_LENGTH];
    uint64_t signature;                 // 0xCAFEBABE
```

```
/* FAT Management */
uint64_t fatStart;
uint64_t fatBlocks;
uint64_t dataStart;
uint64_t fatEntryCount;

/* Block Management */
uint64_t blockSize;
uint64_t totalBlocks;
uint64_t freeBlocks;

/* Directory Management */
uint64_t rootDirectory;
uint64_t maxFilenameLength;
uint64_t maxFileSize;

/* Usage Statistics */
time_t creationTime;
time_t lastMountedTime;
time_t lastWriteTime;
uint32_t mountCount;

/* Metadata */
```

```
    uint64_t metadataLocation;
    uint32_t fsVersion;
    unsigned char reserved[64];
};
```

The Volume Control Block (VCB) is a central component within our file system, it is responsible for managing and identifying the volume. It contains fields like `volumeName` and `signature` that verify the file system type. The VCB plays a crucial role in storage allocation through FAT related fields such as `fatStart`, `fatBlocks`, and `dataStart`, along with block management fields (`blockSize`, `totalBlocks`, `freeBlocks`) to monitor available storage and manage file data. The VCB also organizes files and directories with fields like `rootDirectory`, `maxFilenameLength`, and `maxFileSize`, enabling efficient file structure management.

---

3. Free Space Management Structure

```
struct FATEntry {
    uint64_t nextBlock;    // Next block in chain or special value
};


#define FAT_EOF 0xFFFFFFFFFFFFFFFF  // End of file marker
#define FAT_FREE 0x0000000000000000 // Free block marker


// Free space allocation function
uint64_t allocateBlocks(uint64_t numBlocks) {
    if (vcb->freeBlocks < numBlocks) return 0;

    uint64_t firstBlock = 0;
    uint64_t prevBlock = 0;
    uint64_t blocksFound = 0;
```

```
    for (uint64_t i = vcb->dataStart; i < vcb->totalBlocks; i++) {
        if (fat[i].nextBlock == FAT_FREE) {
            if (firstBlock == 0) firstBlock = i;
            if (prevBlock != 0) fat[prevBlock].nextBlock = i;
            prevBlock = i;
            blocksFound++;

            if (blocksFound == numBlocks) {
                fat[prevBlock].nextBlock = FAT_EOF;
                vcb->freeBlocks -= numBlocks;
                LBAwrite(fat, vcb->fatBlocks, vcb->fatStart);
                return firstBlock;
            }
        }
    }
    return 0;
}
```

This free space management system is how the file system keeps track of free space on the disk, which is important for creating new files and directories. The file system uses a File Allocation Table (FAT) to manage free space. The FAT has three states, free, Allocated and End-of-file. The ‘next Block Number” is once of the most important part of the system and returns the block number of the next block in the chain. Another important aspect of FAT is the status which can be Free or Allocated.

4. Directory Entry Structure & Root Directory

```
struct DirectoryEntry {
    char filename[MAX_FILENAME_LENGTH];
    uint64_t fileSize;
    uint64_t firstBlockIndex;
    time_t creationTime;
    time_t lastModifiedTime;
    uint8_t fileType;
    uint8_t inUse;
    uint16_t linkCount;
    char padding[6];
};

// Root directory initialization
void initializeRootDirectory() {
    const int NUM_DE = 50;
    uint64_t dirSize = sizeof(struct DirectoryEntry) * NUM_DE;
    uint64_t dirBlocks = (dirSize + blockSize - 1) / blockSize;

    struct DirectoryEntry* rootDir = malloc(dirBlocks * blockSize);
    memset(rootDir, 0, dirBlocks * blockSize);

    // Initialize "." entry
    strcpy(rootDir[0].filename, ".");
```

```
    rootDir[0].fileSize = dirSize;
    rootDir[0].firstBlockIndex = startBlock;
    rootDir[0].creationTime = time(NULL);
    rootDir[0].lastModifiedTime = rootDir[0].creationTime;
    rootDir[0].fileType = 1;
    rootDir[0].inUse = 1;
    rootDir[0].linkCount = 2;

    // Initialize ".." entry (same as "." for root)
    memcpy(&rootDir[1], &rootDir[0], sizeof(struct DirectoryEntry));
    strcpy(rootDir[1].filename, "..");
}
```

The Directory System in our file system uses a DirectoryEntry struct to store metadata for files and subdirectories. Our system organizes files hierarchically, helps with navigation, and efficiently manages space and references. The root directory, initialized with entries for the current and parent directories, serves as the top level directory, allowing access to all other files and directories within our file system. Through fields like filename, firstBlockIndex, inUse, and timestamps, our directory system can manage files, track usage, and maintain directory structure efficiently.

The root directory is the top level directory. Meaning that it is the starting point, the “/”. Where the rest of the file path starts. The root (just like any other directory) contains the “.” and “..” entries. This is important because it provides a way of accessing the files in the directory.

5. Task Distribution

Component	Team Members
VCB Implementation	Aldo Zaboni
Free Space Management	Christian Rivera
Root Directory	Gabriel Fernandez
Hexdump Analysis	Karla Cardenas Andrade

6) How did we work together, how often we met, how did we meet, how did we divide up the tasks:

Collaboration strategy / task distribution: We started this project by breaking up the assignment into the three main pieces that needed to be written (VCB, Free space map, and Root Directory) with the fourth group member helping out on each part. We used the same approach for creating the writeup, breaking the assignment down into each of the 7 steps and assigning them (democratically) to whoever wanted said task.

Meeting frequency and format: We met a total of 3 times while working on milestone 1. The first of which was in person in the library where we discussed the assignment and assigned the initial tasks. We then met twice after then over discord to discuss how the project was coming along, what areas were still yet to be completed, as well as challenges we were facing. In addition to these 3 meetings we were constantly sending messages back and forth in Discord to ensure we were all on the same page and completing things in a timely manner.

---

7) Implementation Challenges and Solutions

1. Memory Management Issues

Problems Encountered:

- Buffer overflow risks in FAT allocation
- Memory leaks in directory initialization
- Inefficient memory usage patterns

Solutions:

```
// Use malloc for zero-initialized memory
fat = (struct FATEntry*)calloc(fatBlocks * blockSize, 1);

// Proper memory cleanup
if (writeResult != blocksToWrite) {
    free(fat);
    fat = NULL;
    return -1;
}
```

---

2. Disk I/O Optimization

Challenges:

- Large FAT writes causing performance issues
- Risk of partial writes
- Verification needed for data integrity

Solutions Implemented:

```
// Chunked writing implementation
const uint64_t CHUNK_SIZE = 8;
while (blocksWritten < fatBlocks) {
    uint64_t blocksToWrite = min(CHUNK_SIZE, fatBlocks - blocksWritten);
    // Write chunks}
```

---

3. Data Integrity

Verification Procedures:

```
if (LBAread(verifyBuffer, 1, vcb->fatStart) == 1) {
   // Verify first block contents
}

// Test FAT initialization
printf("FAT Initialization Results:\n");
printf("Total blocks: %lu\n", totalBlocks);
printf("FAT entries: %lu\n", fatEntries);
```