CS-518 - WRITE ONCE

Ashwin Patil (aap327) | Sammed Admuthe (ssa180)

1 Introduction

To the operating system, a disk is nothing but a block of memory cells. All the other functionalities such as file systems to read/write/create/delete files and directories are nothing but policies imposed on the disk. Creating a new file in the file system and writing to it is nothing but reserving necessary memory blocks on disk and storing the data there. In Linux, the file system is implemented by mapping the files to respective memory blocks on the disk. In Linux, this is a mapping between the filename and i-node. What are i-nodes? I-nodes are information nodes containing detailed information about the file such as owner, type, last-modified time, last accessed time, last i-node modified time, access permissions, number of links to the file, total size, etc. The file system is implemented assuming there are 4 types of memory blocks. First is the Boot Block which stores information about where the Super Block is stored and also the data/code which boots the OS. Second is the Super Block which holds information about the disk such as the size of the file system, number of free memory blocks, index of the next free block, size of i-node list, list of free i-nodes, and index of the next free i-node. Third is the i-node list block. This block stores the list of i-nodes, each i-node in this list contains information about the file as mentioned before. Fourth block is the Data Block. This block contains all the memory cells where actual file data is stored.

2 Procedures

2.1 wo_mount

In this section, we have explained our own “mount” function called “wo_mount” which is a virtualization of mounting a disk on a computer.

Every time we call wo_mount, the very first thing we check is if the disk is initialized or if the disk is broken. If the disk is not initialized then we initialize the disk that we want to manage. If the disk is broken, we initialize the disk with proper structures in it.

Input to wo_mount is char *filename and void *address. The first input is the filename of the file that is virtualizing our disk. This mount function tries to open this file. If the file does not exist, then we create a new file with its name as the filename and initialize the necessary structures of the disk. If the disk already exists but it does not satisfy our structure criteria then we report the disk as broken and return an error. The second argument to the wo_mount is the memory address where we are supposed to read the entire “disk” into.

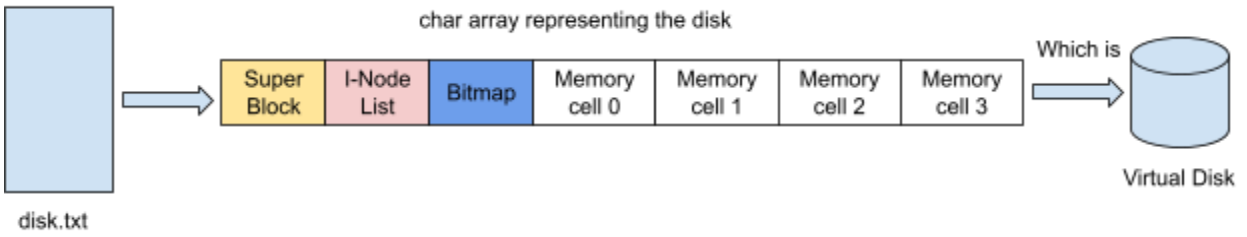

Fig : wo_mount function invocation

2.2 wo_unmount

In this section, we have explained our own “unmount” function called “wo_unmount” which is a virtualization of unmounting a disk from the computer.

Since we are maintaining all the data of the disk on a file, every time we call the wo_unmount function we have to write down all the contents of the “disk” from the memory address to the disk file “disk.txt”.

Input to the wo_unmount function is a memory address where all the disk content is currently present. This wo_unmount function will read the entire disk data from this input address and write it to the actual file we created during the wo_mount call.

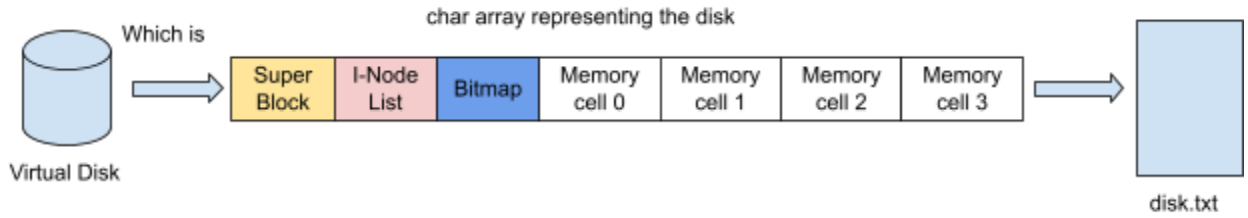

Fig : wo_unmount function invocation

2.3 wo_create

In this section, we have explained the wo_create function which creates a new file if it does not already exist.

The input to the wo_create function is as given below :

1. char* filename - name the file we wish to create.
2. <flags> - WO_RDONLY , WO_WRONLY , WO_RDWR , WO_CREAT

When we invoke the we_create function with valid input, we first check if any file with the same filename already exists. If yes we throw an error, else we create a new file with the input

filename and set the permissions for the file. This metadata about the file is stored in i-node. By default, we create a file in WO_RDWR mode.

Note : wo_create will also open the file.

2.4 wo_open

In this section, we have explained the wo_open function which is our own implementation of the “open” function in Linux.

The input to the wo_open function is as given below :

1. char* filename - The name of the file we wish to open.
2. <flags> - WO_RDONLY , WO_WRONLY , WO_RDWR

When we invoke the wo_open function with valid input, we first check if the file is already open. If the file is already open then we return an error. If the file is not already open then we open the file in the requested mode.

For example, if we open the file in WO_RDONLY mode then we will only be able to perform the read operation. If we try to perform a write operation while we have opened the file in read-only mode then we will return an error.

2.5 wo_close

In this section, we have explained the wo_close function which is our own implementation of the “close” function in Linux.

The input to the wo_open function is as given below :

1. int fd - a valid file-descriptor.

When we invoke the wo_close function with valid input, we first check if the file is already closed. If the file is already closed then we return an error. If the file is not already closed then we set the flag of that file to “closed”.

2.6 wo_read

In this section, we have explained the wo_read function which is our own implementation of the “read” function in Linux.

The input to the wo_read function is as given below :

1. int fd - a valid file descriptor for our filesystem.
2. void* buffer - a memory location where we have to read the data into.
3. int bytes - total number of bytes to be read into the buffer.

When we invoke the wo_read function with valid input, it reads the data from the “file” into the buffer and returns the total number of bytes read.

For example, let’s assume we have a file named “Apple.txt” in our filesystem. If we invoke the wo_read function to read 2 blocks of data from “Apple.txt” then the buffer we passed as input will be loaded with the data from the first two blocks of “Apple.txt”.

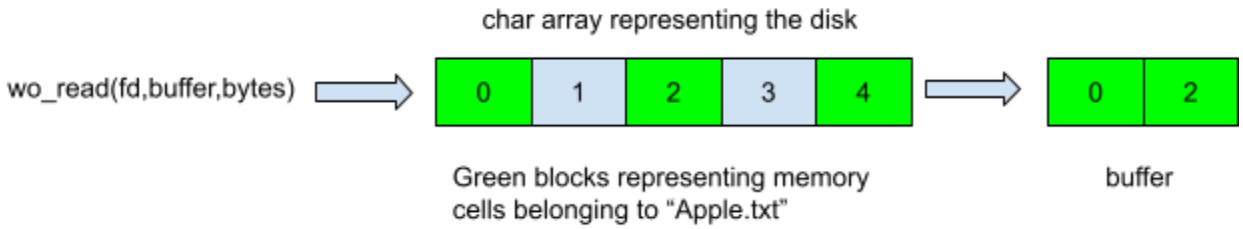

Fig : wo_read function invocation input and output

Note : Now that we have read some data into a buffer, we modify the i-node indicating how many bytes of data we have already read (read_offset). Next time when we invoke the read function, we start reading after this offset.

2.7 wo_write

In this section, we have explained the wo_write function which is our own implementation of the “write” function in Linux.

The input to the wo_write function is as given below :

1. int fd - a valid file descriptor for our file system.
2. void* buffer - a memory location from where we take the data and put it into “disk”.
3. int bytes - total number of bytes to be written into the buffer.

When we invoke the wo_write function with valid input, it reads the data from the buffer and writes into the memory, and returns the total number of bytes read.

For example, let’s assume we have a file named “Apple.txt” in our filesystem. Let’s assume that there is some data already written into this file and it occupies 2 memory blocks. If we invoke the wo_write function to write 1 block of data to “Apple.txt” then the buffer we passed as input will be written in the next free memory cell available to be written.

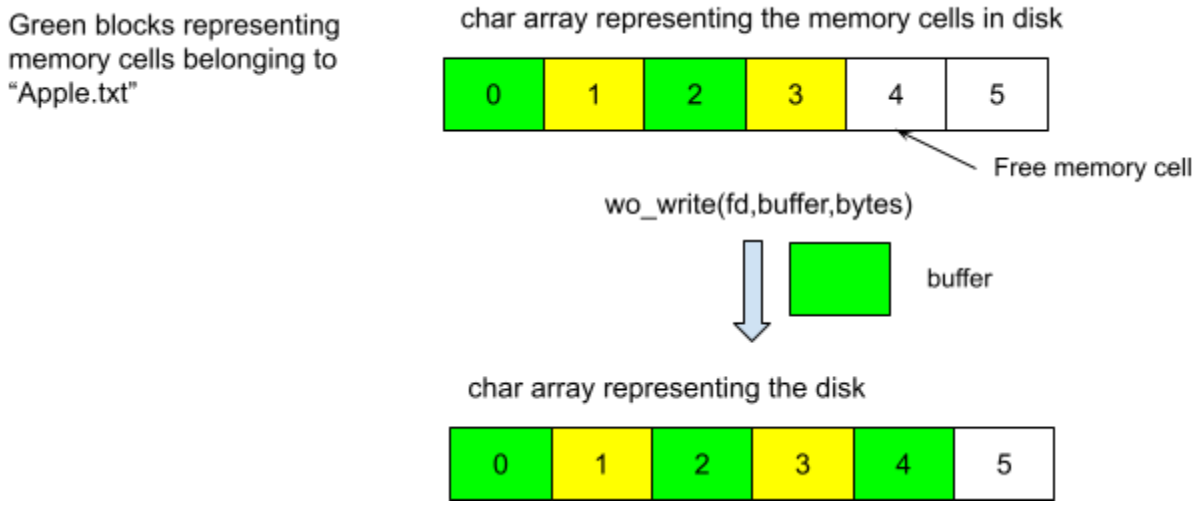

Fig : wo_write function invocation input and output

3 Finding the next free memory cell in the disk

In this section, we have explained the logic behind finding the next free memory cell to write to. When we invoke the wo_write function with valid input and we have to write data to the disk, there could be 2 conditions.

1. There is no data written to the file and the file is empty. In this case, we directly look into the bitmap and find out the first free block of memory cell available and write the data to that cell.
2. There is some data already written in the file. In this case, we traverse through all the memory cells that this particular file occupies and stop at the last memory cell. Then we analyze if this last memory cell is completely occupied or partially occupied. If the memory is partially occupied then we write the incoming data (buffer) into this last memory cell. If this last memory cell becomes full then we find the next free memory cell using bitmap and start writing the remaining buffer into it.

In both the conditions mentioned above, we need to find the next free memory cell in the disk. Below we have explained the structure of the disk.
In the beginning, when we initialize the disk, all the memory cells are empty, so we initialize all the values of bitmap to 'n' indicating that all the memory cells are empty and not occupied.
The following figure explains the initial structure of the disk.

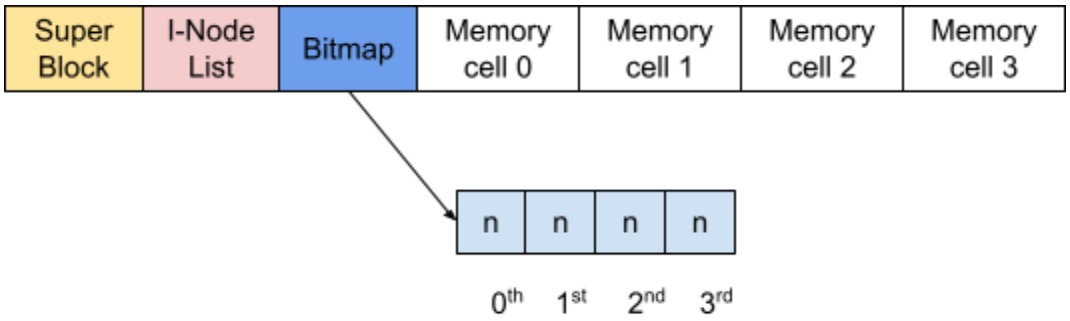

Fig : Initial structure of the disk when the disk is empty

After some wo_write operation, let's say memory block 0 gets occupied, then we get a disk structure as shown in the figure below. Now 0^{th} location of the bitmap contains 'y' indicating that the 0^{th} memory block is now occupied.

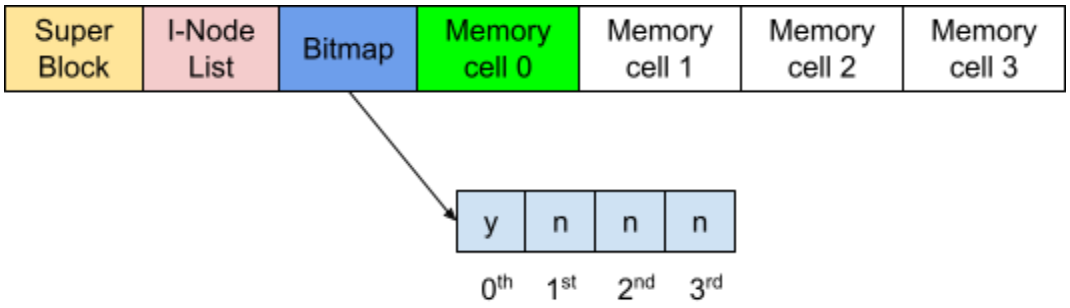

Fig : Disk structure after wo_write operation

To find the next free memory cell, all we have to do is start traversing the bitmap from the beginning and find out which index of the bitmap contains 'n' (meaning the index of the memory cell that is not occupied).

If we are not able to find any memory cell that is free, that means all the memory cells are occupied and the disk is full.

4 Structs

In this section, we have explained the structs that we have used to store all the information about i-nodes, super blocks, and data blocks.

4.1 Super-Block

```
typedef struct
{
   int signature;          //unique signature of the disk
   int no_inodes;          //total number of i-nodes the disk can support
   int no_data_block;  //total number of memory cells that are occupied
   int latest_inode;       //latest i-node
} super_block;
```

4.1 I-node

```
typedef struct
{
   int id;                      //unique id of the i-node
   char filename[200];   //filename
   int total_size_of_file; //total size of the file in bytes
   int data_block_start; //index of the first memory cell that stores the data of this file
   char inUse;              //flag indicating if the i-node already represents a file
   char isOpen;             //flag indicating if the file is open or not
   int permission;          //permission of the file
   int read_offset;         //how many bytes did we read the last time we invoked read
} inode;
```

4.1 Data Block (Memory cell)

```
typedef struct
{
   short next;             //index of the next memory cells that stores the data of the file
   char data[1020];     //actual file data stored in chunks of 1020 bytes

} data_block;
```

5 Test Case Evaluation

In this section, we have evaluated various test cases.

5.1 Is the disk broken?

Initially, the whole disk is empty. The very first time when we mount a disk, we initialize the disk to have some metadata. After that, we perform some operations and maintain the changes in the disk in our memory. When we unmount the disk, we actually write all the data back to the disk (disk file). Next time when we mount the disk, we first check if all the metadata is present or not. We check the structure that the disk holds and if it is not as expected, we return an error.

5.2 Maximum size of a single file we can create

In this test, we know there are a total of N memory cells available to hold data. When we mount the disk for the very first time, we know all the memory cells are empty. So we create a new file using wo_open in CREATE mode and then write data to it. In a while loop, we go on writing 1020 bytes of data until we get an error. Each memory cell can hold 1020 bytes of data. As a result, we will have a file that spans over N memory cells and whose size is N*1020.

Note : At any point in the execution of this test case, we try to write data that is larger than the total memory available. We expect that the wo_write function will write as much data as it can and return the number of bytes it was able to write.

5.3 Maximum number of files we can create

In this test, we perform the wo_open function in CREATE mode. We want to create as many files as we can. In our disk, we will be able to create as many files as many i-nodes we have. So if we have N number of i-nodes in our disk, we can only create a maximum N number of nodes.

Note : In our disk, we are supporting 60 i-nodes, implying we can create a maximum of 60 different files.

6 How to RUN

Step 1 : make
Step 2 : gcc -w benchmark.c -L./ -l writeonceFS -o benchmark
Step 3 : ./benchmark

7 Analysis

We have implemented a custom file system that supports file operations like read, write, create, open, close mount, and unmount functionalities. The file system supports over 60 files and 3.9 MB of user data. This implementation gave us an idea of how basic file systems work. This implementation tries to mimic the actual file system in the closest way possible.