Will Keleher wpk28

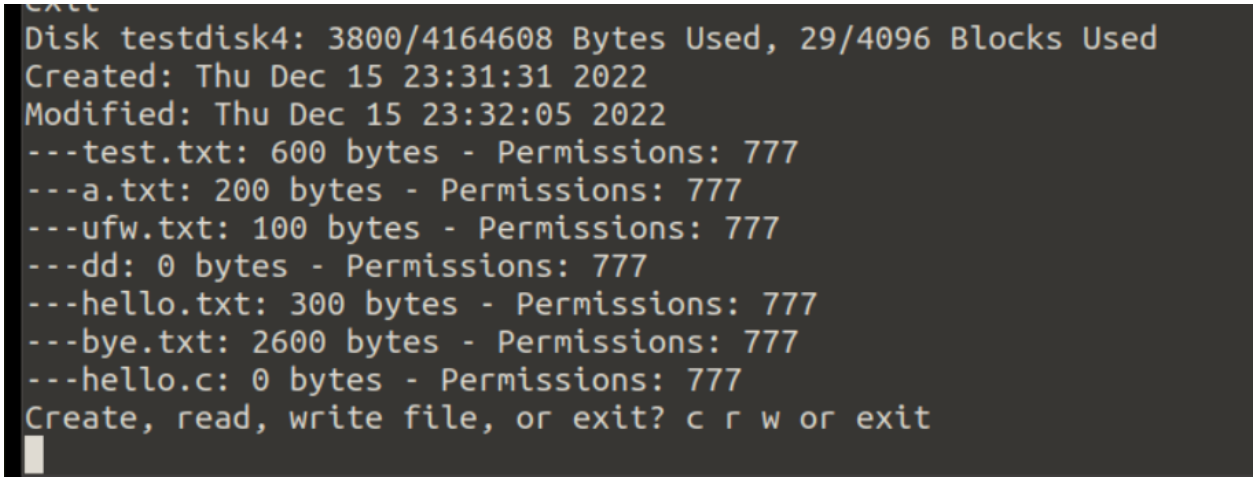
Harshal Keswani hk770

How to run

1. Run make in the directory A2 to initialize object files and libraries
2. Cd into “testing” directory
3. Rename any testing files to ”diskgrinder.c”
4. Run make all
5. The executable line will be ./diskgrinder

BONUS!

We elected to include an interactive demo should you run make and with no file selected. If you simply run “make” the executable will be initialized to a file that takes input from the user and offers them the ability to access their own virtual disk! Text can also be piped in from terminal.



Functions

Core Functions

Wo_mount:

This function initializes a ‘disk’ and prepares it for any operations to come. It handles any errors associated with the selected ‘disk’ and in the case of a blank ‘disk’ space, will give the space the appropriate properties for our implantation.

Wo_unmount:

When invoked, this function will attempt to write out everything prepared by wo_write into the appropriate ‘diskfile’ space.

Wo_open:

On call, this function will search the ‘disk’ space for the file requested. If a file exists within the space but is incompatible with this implantation it will return an error. This function can be set to ‘create’ mode in which it will take a file name and check if there exists a file of that name within the space and if not then it will create the file.

Wo_read:

Wo_read will take a file descriptor and move to the buffer specified and find whatever data is stored at that location.

Wo_write:

This function determines the data to write to the ‘disk’. First, it determines if the file descriptor passed is valid and returns an error if it is not. Wo_write makes sure that there is enough space remaining to perform the write and allocates blocks as needed.

wo_close:

On invocation, this function will evaluate the given file descriptor. If its valid and open, the function will remove the file from the working set of files.

Important return values

Wo_mount:

With no errors, wo_mount returns 0

On an error, the function will return one of the following depending on the error:

> “Disk space data structure not supported” (Not formatted correctly)
>
> “Insufficient space for disk”

Wo_unmount:

Unmount returns 0

Wo_open:

On success, the function will return a file pointer to the file selected.

On an error, the function will return one of the following:

```
-25 = Not enough space, can't be created

-15 = File already exists, can't be created

-1 = File doesn't exist
```

Wo_read:

On success, the function will return an int of the size of the bytes read.

On an error, the function will return one of the following:

```
-10 = Invalid FD

-5 = Cannot read a write only file

-20 = Not enough bytes to be read in file
```

Wo_write:

On success, the function will return an int of the size of the bytes written.

On an error, the function will return one of the following:

```
-3 = Invalid FD

-7 = Cannot write a read only file

-12 = Not enough space on disk.
```

Wo_close:

With no errors, wo_unmount returns 0

On an error, the function will return one of the following depending on the error:

"Invald FD"

Program components

These are some of the components that were implemented to make our core functions run as seamlessly as possible.

Block/chunk:

Coming off of the custom malloc implantation, we elected to implement a similar design in this program. To write memory to the disk, our program creates a block every time there is a new write call. We modeled our implantation after file systems that automatically allocate extra bytes when a small file is initialized. This implantation allows for more data to be written to a single block and

File data (inode) Contains name, attributes,	Disk block (1024)	Disk block (1024)

create/modify times, and bitmap for block allocation And pointer to other inodes ->>>>>>>>		

|----------| |----------| |---------- |

MAX = 700 bytes 1024 bytes

Diskinfo:

The start of every file will contain metadata about the file structured in a similar way to an inode. We elected to call this metadata implantation diskinfo as it contains important information about the disk space.

Super block Disk name, size, free blocks, attributes, date, and checksum value to check for corruption	Inode / Data blocks / Inode series …….

|----------| |----------|

Width of 1024 bytes (4MB - 1024B)

Helper functions:

Print disk:

This function is rather handy for detailing the data in use in the disk space. It prints the name and size data of the disk called upon, as well as the time of creation and last access.

Create empty disk:

For our implantation, this function will create the data structure for our file system at the memory address specified if one does not already exist.

set/get block:

These functions help to implement the bitmapping throughout our disk space. When invoked they determine the next free space within our system and assign data there.

Getfirstfreeblock:

This function iterates through our data space and is able to retrieve the next space to put data in order to remain as contiguous as possible.

Build details

Process

Initaly, our implantation employed a series of linked lists to provide abstraction for handling large files. As we progressed we were drawn to a bit of a simpler approach, bitmapping. With this architecture, we have been able to vastly improve run times in our system without compromising on accuracy. Our methodology does take up a bit more space than the previous systems but as it falls well within the acceptable parameters of the requirements, we believe that the priority shift to speed was more than worth the space.

Data

filenode:

The implantation of our node structure has undergone many iterations. As of completion, our file node has the following properties:

- Name - a string of length 16 max that will be unique to the file in the disk space
- Size - unsigned int stored for immediate access on the size of the node
- Permissions - another unsigned int to control what parameters the file can be viewed in
- Creation time- time_t meta data on the time that the node was initially created
- Modified time- time_t meta data on the last time that the node was last accessed
- Next file - an unsigned int that serves as a “pointer” to the next file
- Blocktable - unsigned int that serves to communicate memory location and blocks in use

With our implantation, we elected to employ a bitmap of the storage space. This abstraction is easily manipulated to quickly find available room, the next empty space, remaining space, and so much more. In an earlier iteration, we elected to use “real” pointers but found that they were often more troublesome when it came to gathering this kind of information quickly. The bitmap allows for O(N) access times for data lookup while reducing the number of calculations needed. We employ a bit shifting system to accommodate files within our system and allow for slotting data where it needs to go when required.

diskinfo:

Our diskinfo houses important metadata for easy access and is very useful when initializing a disk space. As of completion, the diskinfo struct is as follows:

- Disk name - this char parameter houses the name of the disk in order to create many distinct disks if needed

- Used space - another unsigned int that is used in the calculation of the amount of actual space taken up on the disk
- Used blocks - in contrast, this parameter keeps track of the blocks of data assigned to file nodes within the system to create a running tally for calculations
- Num blocks - an unsigned int that determines the amount of total blocks in the disk space
- Create time - time_t meta data on the time that the node was initially created
- Modified time - time_t meta data on the last time that the node was last accessed
- First file - an unsigned int that once again functions as a "pointer" to the first data block
- Block table - this unsigned int another reference to our bitmap that allows for instant data access when examining a disk

Bitmap:

Our bitmap system is an abstraction of the data in the disk space. Each 1 represents a used block of data from our data total while each 0 represents an open slot. Since we elected to use a static data block size, that means we are quickly able to find available space on our disk for file accommodation if need be. Furthermore, if a file is to be moved from one location on the disk space to another, the bitmap will handle the abstraction, while using little space (all the info for a single file and inode fit within one 1024 byte block).