# Assignment 4 – Design and Analysis of a File System

COMP 310 / ECSE 427 – Operating Systems

Fall 2020


Alex Gruenwald 260783506

Sean Tan 260761889

Yoan Poulmarc'k 260773983

Colton Campbell 260777576

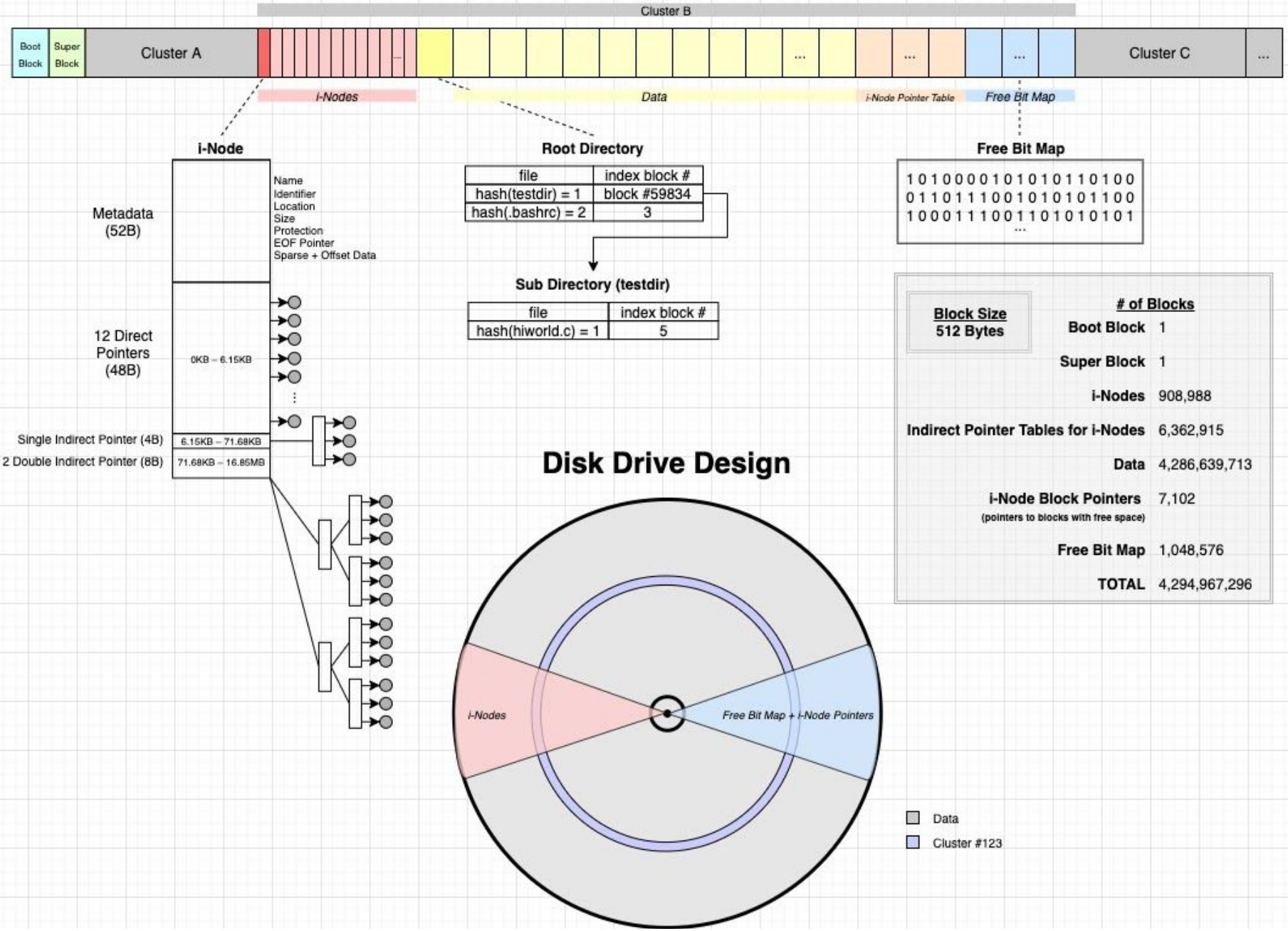
I. Block Size

**512B.** The average file size is 483KB. 512B blocks ensure minimal internal fragmentation and increase IOPP to optimize CPU utilization. With this size, there are 5 i-nodes/block, with 2 bytes of internal fragmentation. FCB, indirect pointer tables, and FBM table/cluster are 503.686KB of the 800KB cache. Increase: 94 blocks/year/file.

II. File System Candidate

Clustered i-node implementation is used since it's given that files that are grouped together are accessed close to each other. Each cluster is allocated its own i-node table, FBM and parent directory for quick lookups and minimal disk movement as each cluster is stored on their own track. Block accesses are quicker in comparison to FAT and single i-node tables because searching for the block requires a linear search through all table entries, which is especially slow in those implementations for the chosen block size. Clusters are segmented by track for minimal R/W head movement, something that FAT and i-node tables can't be grouped by. Under the assumption that files in the same directory are accessed together, clusters will be placed on a track proportional to the size of its directories. Furthermore, growth in directory size over time is negligible (10% over 12 months), so the risk of one cluster reaching max capacity and needing to place a file in another directory, is unlikely. This is more of a concern for the FAT as it handles all pointers in a single table, and with growth, increases the number of entries, thus increasing lookup speed with each year for all data lookups. The downside to this implementation is that the SuperBlock on the disk will have to store pointers for cluster locations resulting in additional overhead; however, this architecture does not grow with time as the number of clusters is static and is not impacted by the block size.

# File System Design




| Block Size 512 Bytes | # of Blocks |
|---|---|
| Boot Block | 1 |
| Super Block | 1 |
| i-Nodes | 908,988 |
| Indirect Pointer Tables for i-Nodes | 6,362,915 |
| Data | 4,286,639,713 |
| i-Node Block Pointers (pointers to blocks with free space) | 7,102 |
| Free Bit Map | 1,048,576 |
| TOTAL | 4,294,967,296 |

# Disk Drive Design

## III. Diagrams

A. There are 1024 clusters where each cluster has its own FBM, i-node table, parent directory, i-node pointer table and data blocks (files only accessible in the cluster).

B. Each i-node has 102B of data with 52B for metadata (name, ... , sparse + offset data). 5 i-nodes can be stored within each block (2 bytes of internal fragmentation). To remember which blocks have space to store future i-nodes, an i-node pointer table is allocated for each cluster to point to free space. There are 3 indirect pointers - 1x single, 2x double - to access 16.85MB of data.

C. To optimize lookup in directory tables, linked list hash tables are used where each table uses a hash to the pointer to another directory/file. Collisions are dealt by simply incrementing from the original collision area in order to have files close together for locality.

D. Disk arrangement: each cluster is assigned to 1 track on creation (all clusters are created and assigned by pooling and grabbed when new clusters are needed). Each cluster grows proportional to the data of the assigned track (inner: smallest | outer: largest). There are two sectors: I-Nodes and FBM + i-node pointers. The sectors are on opposite sides to prevent corruption of the file and to reduce wait time to read information (half a rotation).

E. The Super Block stores information on where clusters are and where directories are.

F. Read/Write pathing: SuperNode -> Cluster -> parent directory -> i-nodes -> data blocks

## IV. Pseudo code

### i) Create (5 points)

```
Create(NewFileURI, NewSetupSize)
   if (NewFileURI == empty or NewSetupSize <= 0) return -1
// Check for permission to create a file in the current directory
   If (!permissionToCreateInCurrentDirectory(NewFileURI)) return -1
   cluster = findDirectoryCluster(NewFileURI) //Recursively go through directory I-Nodes to find immediate parent directory based off URI
   if( getAvailableSpace(cluster) < NewSetupSize ) return -1
   if( clusterHasFile(cluster, NewFileName) ) return -1

   // getPtrToNewFileFreeSpace is responsible for saving the FBM from the disk to the cache, iterating through the bitmap for 1, and returning a list of pointers that correspond to these free block spaces. The FBM is stored back into the disk afterwards and cleared from the cache
   ptr = getPtrToNewFileFreeSpace(cluster, hashFunction(NewFileName), NewSetupSize)
   // addNewINode loads the i-node pointer table into cache to check for blocks that have space to save a node to. The node is found in the pointer directory and loads the block into memory where the specific open space is found and allocated to the node. The parameters in the i-node are set and a pointer to it is returned
   node = addNewINode(Ptr, NewFileName, NewSetupSize)

   // In our implementation, each directory I-Node has its own list representing blocks with enough free space to add additional I-Nodes (our blocks size means we can hold 5 I-Nodes exactly in one), which this function will take into consideration when adding an entry
   // Also updates the directory with the new I-Node with file name
   addINodeToDirectory(cluster, node)

   updateClusterBitMap(cluster, node)

   //Updates:
   // the permissions (chmod structure -rwx------)
   // pointer for read, write and EOF
   // sparse files: list of pointers -> (start, end, offset)
   // update creation date
   // set owner of file, group, ACL, last modified, created
   // update file size
   // pointers to the blocks that store the data
   finalizeINode(cluster, node)

//cluster tells the disk which track to read from

   save()

  return NewFileURI
```

### ii) Open (10 points)

```
Open(fileUri, ACCESS_TYPE)
   if (fileUri == empty) return -1
   // Check open-file table to find if file opened by another process
   if ( isFileCurrentlyOpened(fileUri) ) return -2
   // Read FCB permission code regarding current user
   if( ! userHasPermissionToOpen(node, ACCESS_TYPE) ) return -3
   // File descriptor points to per-process file table entry which points to sys-wide file table entry so we are creating an entry or
incrementing how many processes are reading it
   if( fileOpenedInSystemWideFileTable(fileUri) )
            sysWidePtr = getPtrToSystemWideFileTableEntry(fileUri)
            incrementEntryAccesses(sysWidePtr)
   else
            sysWidePtr = createPtrToSystemWideFileTableEntry(fileUri)
            getAndLoadFCBIntoSystemWideFileTable(sysWidePtr, fileUri)
   //A pointer is created in the per-process open-file table with a pointer to the entry in the system-wide open-file table. Other
pointers are also created pointing to the location in the file where the read and write operations are. File is in open mode
   ptr = addPtrInPerProcessOpenFileTable(sysWidePtr)
   File_descriptor = addPtrToFileDescriptor(ptr)
return File_descriptor
```

### iii) Read (15 points)

```
Read(fd, buffer, size)
   if ( size < 1 ) return -1
   if ( !isValidFileDescriptor(fd) ) return -2
// Using the pointer from the fd table, get the pointer to the per process entry
   ptr = getPerProcessOpenFileTableFile(fd)
  // Check permissions for read
   if ( ! hasReadPermissions(ptr) ) return -3
// The FCB block should be stored in memory but if it is not, then load it from the disk
   ptr = getSystemWideOpenFileTableFile(ptr)
// If FCB is not loaded into cache, then load into cache
   if (ptr == null)
           fcb = loadFcb()
           ptr =  getFCBPtr(fcb)
   else fcb = getFCBFromCache(ptr)
   / Ensure only 1 process is accessing the resources at any given time. Uses busy wait to ensure no race conditions
   // getSystemWideProcessConnections checks the number of processes
            if (getSystemWideProcessConnections(ptr) != 1)
            while( isResourceLocked(ptr) ) //Wait for resource to be free
            lock(mutex)
            readPtr = getReadPtrAtCurrentLocation(ptr)
            holes = getSparseDataWithOffsets(fd)
            //Shift the read pointer using read_next() -> discussed below using scan nearest next scan
          blocksRequested = ceil(size / blockSize)
            // If data that I am trying to access is in indirect pointers, load those blocks into memory and access those pointers to
the required blocks
            Ptrs = getBlockPtrsFromFCB(ptr)
            foreach(blockPtr in ptrs)
                //for each read operation, looking to compare the current read pointer to the holes stored in the metadata to see if
there needs to be any 0s read.
                writeToBuffer(buffer, size, readPtr, holes)
```

```
            resetReadPointer(readPtr)
            clearLocalMemoryOfDatablocks(Ptrs)
            updateFCBStoredInCache(fcb)
    return buffer
```

### iv) Write (15 points)

```
Write(fd, buffer, inputSize)
    if ( !isValidFileDescriptor(fd) ) return -1
// Using the pointer from the fd table, get the pointer to the per process entry
    ptr = getPerProcessOpenFileTableFile(fd)
   // Check permissions for write
    if ( ! hasWritePermissions(ptr) ) return -2
// The FCB block should be stored in memory but if it is not, then load it from the disk
    ptr = getSystemWideOpenFileTableFile(ptr)
// If FCB is not loaded into cache, then load into cache
    If (ptr == null)
            fcb = loadFcb()
            ptr =  getFCBPtr(fcb)
    Else fcb = getFcbFromCache(ptr)
    // Ensure only 1 process is accessing the resources at any given time. Uses busy wait to ensure no race conditions
    // getSystemWideProcessConnections checks the number of processes
            if (getSystemWideProcessConnections(ptr) != 1)
            while( isResourceLocked(ptr) ) //Wait for resource to be free
            lock(mutex)
            writePtr = getWritePtr(ptr)
           // Check how many blocks are needed to write to the file with the given input
            if ( writePtr == EoF )
               bytesNeeded = inputSize - getFreeBytesInCurrentBlock()
            else
               bytesNeeded = inputSize - getFreeBytesInCurrentBlock() - getBytesFromOffsets(ptr->FCB->metadata)
            blocksNeeded = ceil(bytesNeeded/BlockSize)
            // This function loads the FBM into cache from the disk to check for blocks that are free to use and write to     and
updates the bits in the cache. If the blocks that are free cannot be reached by the currently loaded i-node pointers, then load the
appropriate tables in memory for the single and double indirect pointers to save later. If there are no more free blocks left then it
returns a null pointer
            ptrs = getNFreeBlocksFromFBM(fbm, blocksNeeded)
            If (ptrs == null) return -3
            // Save the assigned pointers to the FCB and all associated, loaded tables
            insertPointersToFCB(fd, ptrs)
            save(fcb)
           unlock(mutex)
            // Write to each block until complete
            foreach(ptr in ptrs)
               buffer = write(buffer) //Writes as much of buffer as possible then returns where it stopped if too much space is added
for the block. Automatically shifts the write pointer within each block. Ptr is updated each loop
               save(ptr)
     Return buffer
```

### v) Close (5 points)

```
Close(fd)
   if ( !isValidFileDescriptor(fd) ) return -1
   process = getRequestingProcess()
   pointerToSystem = removeEntryFromPerProcessFileTable(process, fd)
// Check to ensure other processes are not using the system wide open table
   If (numberOfProcessesUsingEntry(pointerToSystem) == 0)
            deleteEntry(pointerToSystem)
            clearFCBCache(pointerToSystem)
   Else
         decrementCount(pointerToSystem)
   saveFCBIntMemory(getCurrentFCB().Address)
   Return success
```

### vi) Seek (5 points)

```
Seek(fd, offset, SEEK_SET || SEEK_END)
   if ( !isValidFileDescriptor(fd) ) return -1
// Using the pointer from the fd table, get the pointer to the per process entry
   ptr = getPerProcessOpenFileTableFile(fd)
// Check permissions for seek
   if ( ! hasPermissions(ptr) ) return -2
// The FCB block should be stored in memory but if it is not, then load it from the disk
   ptr = getSystemWideOpenFileTableFile(ptr)
// Ensure only 1 process is accessing the resources at any given time. Uses busy wait to ensure no race conditions
// getSystemWideProcessConnections checks the number of processes that are currently accessing the entry in the system wide
open table
   if (getSystemWideProcessConnections(ptr) != 1)
   while( isResourceLocked(ptr) ) //Wait for resource to be free
   lock(mutex)
   seekPtr = getResourceWritePointer(ptr)
   if ( SEEK_SET )
            resetSeekPtr(seekPtr)
            save(seekPtr) //Save to cached FCB
   if ( SEEK_END )
            setSeekPtrToEoF(seekPtr)
           // Save the pointer location into metadata of the FCB for sparse files later. Doesn't actually increment the seek pointer
as all the information is stored in metadata instead of physically
            updateOffsetForFile(ptr, seekPtr, offset)
            save(seekPtr)
   unlock(mutex)
   return seekPtr
```

V. Discussion of Performance Optimization (Combined with PART VII, based on 100 lines count)

**Disk Structure** – By separating each cluster into its own track, head movement is minimized while reading different blocks until that request is complete. Given by the prompt, files that are in the same directory are accessed together, thus the head can stay on the same track. This reduces time spent on the head moving, potentially missing out on reading blocks due to the rotation of the disk. The i-Node and FBM sectors are positioned on opposite sides of the disk for corruption prevention and to reduce the reading time from one full rotation to half of one. We follow a FCFS scanning technique for process requests with each block request using the shortest service time first scan. The head changes tracks for requests with different structures but sticks with the track for requests from the same cluster. Upon completion of a request, the next request will be decided using FCFS, ensuring no starvation. Read and write operations will be faster due to the decrease in R/W head movements on the disk since files are going to be accessed within the same cluster.

**Block Size** – 512B block size has been chosen for minimizing internal fragmentation for i-Node and data storage per file, and for the increase in IOPP. This reduces the time spent on each read/write operation which maximizes CPU utilization (more intervals for IO bound processes). With our clustered i-Node system and 512B block size, each cluster, having their own FCBs & FBM, will require 503KB of space to store all information in the 800KB local cache. Since files in the same directory, and consequently our clusters, are likely to be accessed together, the storing of the FBM and parent directory in the cache (unique to each cluster) from previous file requests, will eliminate the need for future file requests (from the same cluster) to read directly from the disk. With this information present in memory, read and write requests will be much faster as FCB locations can be found through the cached directory table and free blocks within the cluster can be found in the stored FBM. Once again, as from the prompt, files in similar directories/clusters will be accessed together will ensure this information is used for future requests. Create and write operations will not have to access the disk in many cases, because all information related to FBM, i-Node tables and parent directory can be stored in the cache.

**i-Node Pointer Table** – This table keeps track of blocks that have not reached max-capacity for i-Node storage. The table contains pointers to the blocks, so that when new files are created, it is easy to locate which blocks have free space to store the FCB, reducing the overall time needed to create a file. The create operation can quickly find the exact block in the disk that has space for the new i-Node to be added to the block, instead of having to search all i-Node blocks in the disk, to find free space.

**Caching Algorithm** – The caching algorithm is the Least Recently Used (LRU) algorithm. The important information being stored in the cache is the current cluster FBM, the currently open FCBs and the parent directory with all i-Node pointers. All this information takes up 503KB of memory which will need to be replaced for every new cluster call. Because files in the same directory are accessed together, we can assume that once a cluster has changed to access a new file in another cluster, the previous cluster will not be accessed for some time, thus making the clusters previously cached information useless for future requests. From this model, upon the request of a new file in a new cluster, we expect extremely high page fault rates through disk reads, but once the information is stored much like the working set model, it is expected that the page faults will be minimized for the remaining requests in the cluster. With this

information stored in the cache, R/W operations will not require frequent disk access and will only require accessing the disk to actually read the data blocks.

## VI. Missing Requirements

Note: In the above section, we added a lot of additional information that can be repeated in this section. To avoid reiterating our points with different wording, please reference the above points. Additional points have been made below.

A hash function is used to create a distributed hash table for each directory. These hash tables will hold all file and directory names, as well as pointers to their locations for their respective parent directory. The function prioritizes placing files together (clustered hash function) as there will be a lot more reads than writes (given from instructions); therefore, having them next to each other would be optimal for read speed.

Number of clusters chosen: Research shows ~1024 tracks per HDD. With 2TB of disk space, on average, the tracks can hold 2GB of data. Considering all that is written in the design of our file system, 0.2% of the data on each track will be used for non-data purposes. If we assign 1 track per cluster, there will be 1024 clusters, with on average, 2GB of space for the centermost cluster. The outermost cluster will be the largest and innermost, the smallest. As it is expected that blocks won't fit perfectly in each track, with space that cannot be used, the small blocks will ensure that minimal space will be lost as at most, per cluster, there will be 511B of lost space.

The structure chosen for the directories starting from the parent in each cluster, is the tree structure. The files are considered the leaves and all internal nodes are the directories. This tree-like structure is chosen to reduce the lookup time for deeply nested files and their allocated blocks.

We are using the extension file type for the files that we are saving rather than the fingerprint.

Sparse files are files with a lot of holes in their memory due to seeking operations; these holes are usually assigned 0s. To prevent having to store these 0s in physical memory - wasting space - and having to store it in logical memory, we decided to implement a pointer + offset list in the metadata of the FCB. This information will tell the read() function when there are expected holes in memory, but will ensure the holes aren't actually stored in memory. This ensures we store all files efficiently and prevent any internal fragmentation resulting from the offsets.

Additional Scenarios:
If a file exceeds the size of a cluster, return that the disk space is empty. By organizing our files system the way we did, it ensures each cluster grows at a steady rate and will reach max capacity at the same time (10% growth per year). Thus, if one file is too big, then all future files will be too large.

If the file exists in another cluster/directory, there are no issues as we are using a tree structure to separate the files. If the file already exists, then we return an error message saying: "cannot have multiple files in the same directory with the same name".

If a user is trying to create a file in a directory they don't have permission to create a file in, then an error message is returned saying “you do not have permission to create a file in this directory”.

If a file is trying to be accessed without the correct permissions, an error message will be returned saying: “Incorrect permissions to access this file”.

If a file opened with read has a write command, an error message will be returned saying: “You are only allowed to read from this file”.