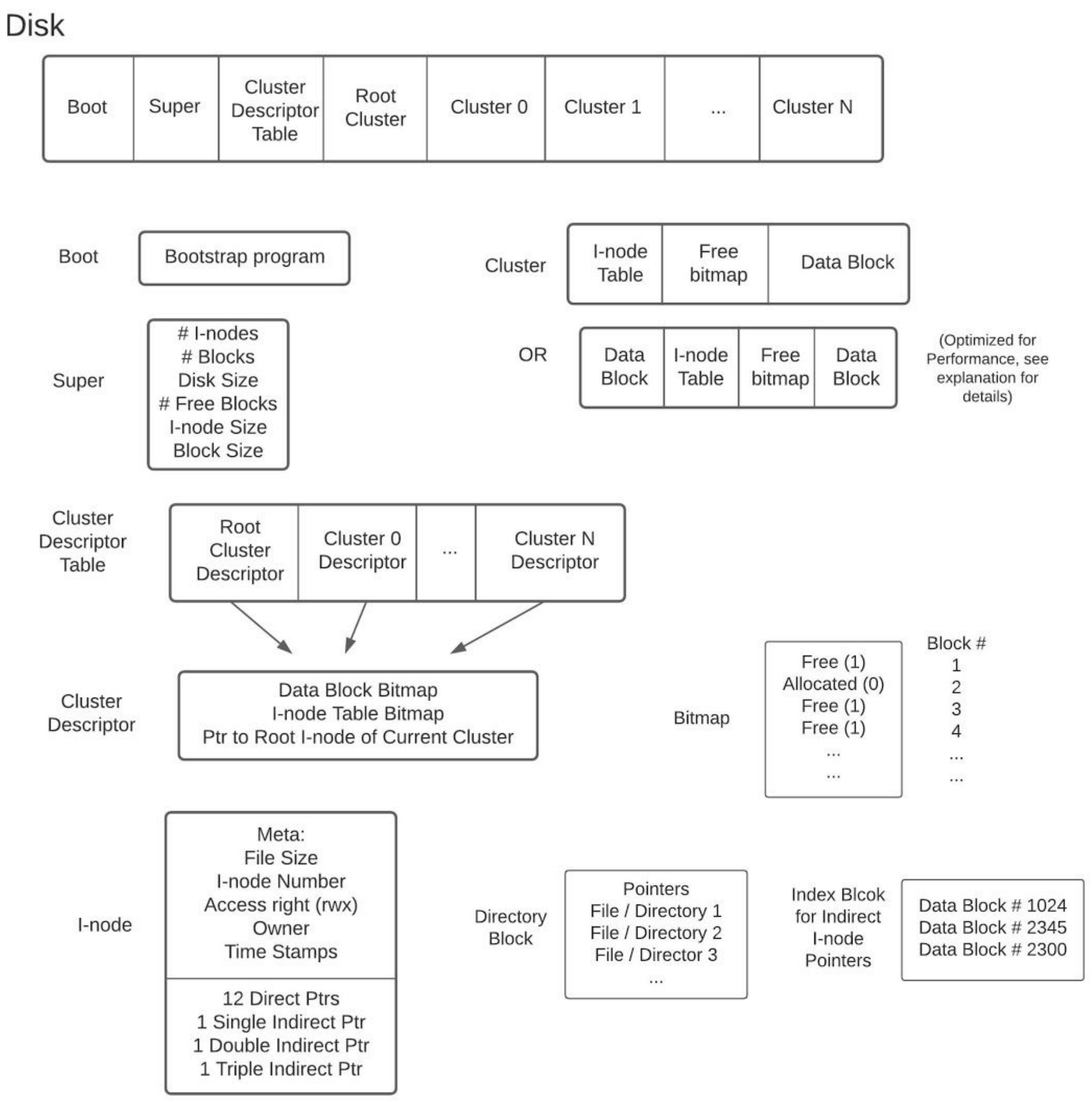
**What is the block size you want to use? Why did you select the block size value?**
2KB per block.
With 12 direct pointers in each i-node and 2KB block size, the file system can directly access files no larger than 24KB, i.e., 74% of the files on disk. This ensures performance. 2KB block size restricts max internal fragmentation per file to 2KB as well.

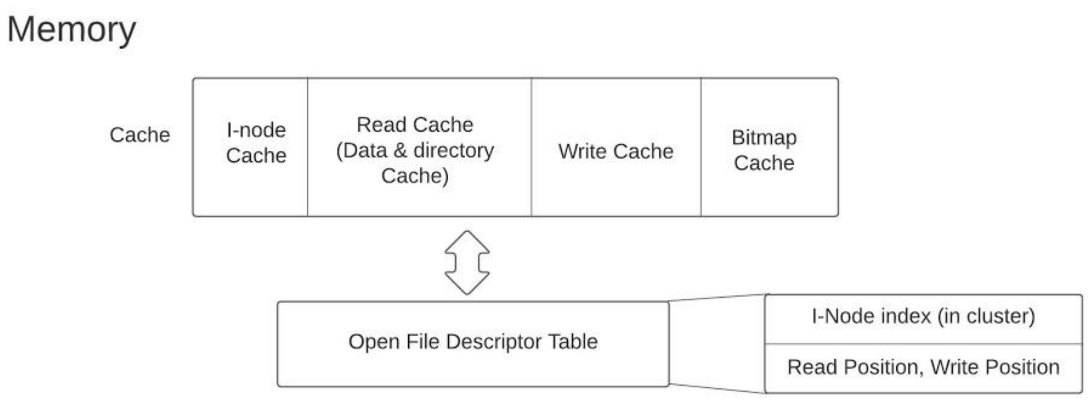
**What is the file system you want to use? Why?**
We would like to choose the clustered i-node file system. Our arguments are as follows:

1. In the FAT file system, the file allocation table needs to be cached in the memory. However, there will be many file seeks if the entire FAT is not stored in the memory. For example, in a 2TB (2^41 bytes) disk with each file size of 2KB (worst case: each block is a file), FAT has 2^30 entries. Suppose 4 bytes are used per entry -> 4GB (2^32) of main memory are required, which is a sizable overhead. Given the constraints in the assignment handout, only 0.8GB (16GB*5%) can be used for caching. Therefore, we cannot store the entire FAT in the main memory. This will lead to extra seek time.
   FAT cannot utilize the access pattern provided. This is another disadvantage.

2. Compared to the simple i-node file system, a clustered one can scatter the whole i-node table in different locations on the disk according to the locality. When we want to query a particular file, we only need to look for part of the whole disk. Thus this clustered i-node file system can improve the performance.

**Details of the selected architecture (diagram)**




Memory
Cache
I-node Cache
Read Cache (Data & directory Cache)
Write Cache
Bitmap Cache
Open File Descriptor Table
I-Node index (in cluster)
Read Position, Write Position

**Pseudo code for create,open, read, write, close, seek**

```
Int create(fileName) {
    If (file exists || file opened already) return -1;
    I_node = empty_inode();
    Find the i-node for fileName’s parent directory, call it p;
    Find the directory block of p, name the block dir_block; // we assume that the locality access is
within directories
    Find in which cluster does p locate, name the cluster c, return -1 if fail;
    Allocate an I-Node on cluster c with iNodeIndex, return -1 if fail; // fileName i-node
    Assign fileName i-node with necessary meta, i.e., name, access rights, …;
    If (BlockHasEnoughSpace(dir_block))  dir_block.append({fileName, iNodeIndex});
    Else if (ClusterHasEnoughSpace(c)) {
        allocate a new dir block in parent directory I-Node, call the dir block b1;
        b1.append{fileName, iNodeIndex};
        Link b1 to p;
    }
    Else { // Neither the original dir block nor the cluster has enough room
        Go to the nearest cluster that has enough data block room, call this cluster C;
        Create a new dir block on C, call the block b2;
        b2.append{fileName, iNodeIndex};
        Link b2 to p;
    }
    Flush I-Node cache to disk, flush the written directory block to disk;
    Flush free bitmap to disk;
    Return 0;
}


int open(fileName, accessRight) {
    if (FileOpened(fileName))
        return GetIndex(openFileTable, fileName);
    Find an empty slot in openFileTable,
        with index newFD, else return -1;
    if (fileName is in cachedCluster): // If already preloaded cache
        Find the I-Node of fileName in INode cache,
            with I-Node Id iNodeId;
    Else {
        if (fileName doesn't exist in the disk){
            return -1;
        }
        // (Performance optimization helper) pre-load the cluster that the fileName belongs to
        read_from_disk(fileName);
        Find the I-Node of fileName in INode cache, with I-Node Id iNodeId;
    }
    // check if access request matches the access right stored in the meta
    if (accessRight is read only AND access right in I-Node meta does not allow read){
        return -1;
    } else if (accessRight is write only AND access right in I-Node meta does not allow write ) {
        return -1;
    } else if (accessRight is read and write AND access right in i-Node does not allow write and
    read) {
        return -1;
    }
    openFileTable[newFD].readPtr = 0; //first direct ptr of i-node iNodeId;
    openFileTable[newFD].writePtr = 0;
    openFileTable[newFD].iNodeId = iNodeId;
    return newFD;
}

Int read(int fd, void *buf, size_t nbyte) {
    if (openFileTable[fd] does not exist)  return -1;
    inodeId = openFileTable[fd].iNodeId;
    if (inodeId<0)  return -1;
    inode=get_inode_from_id(inodeId);
    readPtr = openFileTable[fd].readPtr;
    currentDlockIndex= readPtr/blockSize; //picking which block to read from in inode
    Initialize blockToRead;
    if (currentDataBlockIdx<12) { // If is direct blocks
        blockToRead=inode[currDataBlockIdx];
        If (blockToRead is null) return -1;
    } else if(currentDataBlockIdx < 12+2048/sizeof(block ptr)) { // If is single indirect blocks
        if (INodeSingleIndirectEmpty()) return -1;
        while(directBlock not found) {
            Repeat searching in the indirect blocks;
        }
        blockToRead=newly found directBlock;
    } else if (currentDataBlockIdx < 12+2048/sizeof(block ptr)+(2048/sizeof(block ptr))^2){
        // If is double indirect blocks
        if (INodeDoubleIndirectEmpty()) return -1;
        while(directBlock not found) {
            Repeat searching in the indirect blocks;
        }
        blockToRead=newly found directBlock;
    } else {
        // Else is triple indirect blocks
        if (INodeTripleIndirectEmpty()) return -1;
        while(directBlock not found) {
            Repeat searching in the indirect blocks;
        }
        blockToRead=newly found directBlock;
    }
    if(length of rest data < nbyte) { // Cannot read nbytes of file b/c file is not that long
            Actual_len = length of rest data;
    } else
        actual_len = nbyte;
    rest_bytes _to_read = actual_len;
    If (check need to read more than one block == false) {
```

```
        //if read doesn't go beyond block
            Copy current reading portion into buffer;
            readPtr+=actual_len;
        } else {   // the material to read is contained in more than one block
            While (still have more blocks to read, i.e., rest_bytes_to_read > 0) {
                Read rest data from current block into buffer;
                Rest_bytes_to_read -= read data size from current block;
                Go to next data block, update readPtr;
            }
        }
        return actual_len;
}


Int write(int fd, void *buf, int len) {
        if (openFileTable[fd] does not exist)  return -1;
        inodeId = openFileTable[fd].iNodeId;
        if (inodeId<0)  return -1;
        inode=get_inode_from_id(inodeId);
        writePtr = openFileTable[fd].writePtr;
        currDataBlockIdx = writePtr / blockSize; //picking which block in inode to write
        int blockToWrite;
        if (currentDataBlockIdx<12) { //blockToWrite is a direct block
            blockToWrite=inode[currDataBlockIdx];
            If (blockToWrite is null) {
                blockToWrite = allocate new block;
                inode[currDataBlockIdx] = blockToWrite;
            }
        } else if(currentDataBlockIdx < 12+2048/sizeof(block ptr)) { // If is single indirect blocks
            if (INodeSingleIndirectEmpty()) return -1;
            while(directBlock not found) {
                Repeat searching in the indirect blocks;
            }
            blockToWrite=newly found directBlock;
        } else if (currentDataBlockIdx < 12+2048/sizeof(block ptr)+(2048/sizeof(block ptr))^2){
            // If is double indirect blocks
            if (INodeDoubleIndirectEmpty()) return -1;
            while(directBlock not found) {
                Repeat searching in the indirect blocks;
            }
            blockToWrite=newly found directBlock;
        } else {
            // Else is triple indirect blocks
            if (INodeTripleIndirectEmpty()) return -1;
            while(directBlock not found) {
                Repeat searching in the indirect blocks;
            }
            blockToWrite=newly found directBlock;
        }
        If (check need to write only one block== true) {
            //if we’re only writing to one single block
            //Write the full buffer into disk block (using performance optimization helper)
            write_to_disk(buffer);
            Update writePtr;
        } else if (have enough room in write in current cluster = true) {
            //if we need to write to more than one block
            // split the whole write data into multiples chunks, write one chunk to one block per time

            write_to_disk(firstPartOfBuffer); //first part of buffer has length length_firstWrite

            // some manipulation with the write length and buffer
            buffer+=length_firstWrite;
            len-=length_firstWrite;
            writePtr+=len_firstWrite;

            While (still have data to write)
                continue writing the remainder of buffer to inode[++currDataBlockIdx] in a
similar fashion, using write_to_disk();
        } else { //need to write outside the current cluster
            Search the nearest neighboring cluster(s), find enough free data blocks to assign to
blockToWrite; // Make sure the data is physically close
            if(data block not found) return -1; // Disk is full
            Write data to these new found data blocks using write_to_disk(); // Performance
optimization
            Link data blocks with i-node;
        }
        return len;
}

Int close(int fd) {
        Flush the cached blocks to disk;
        if (fd<OpenFileTableLength && openFileTable[fd].iNodeId){
            openFileTable[fd] = OpenFileObjectUsed; // OpenFileObjectUsed is a null object,
explained in the extra explanation section
            return 0;
        }
        return -1; // no file descriptor entry for the given fd
}




void seek(fd,offset) {
        // update read & write pointer inside openFileTable[fd]
        writePtr = openFileTable[fd].writePtr;
        readPtr = openFileTable[fd].readPtr;
        writePtr+=offset; //advance write pointer by the desired distance
        writePtr=min(writePtr,fileSize); //prevent pointer from moving outside the inode
        readPtr+=offset; //advance the read pointer by the desired distance
        readPtr=min(readPtr,fileSize); //prevent pointer from moving outside the inode
}
```

**Performance improvement helper functions:**

```
Void read_from_disk(file_location) {
        // The following code updates the cache
        Open the cluster descriptor table;
        Find the cluster in which file_location resides, call this cluster C;

        If (cluster C has been pre-loaded) {
                Load the i-node and data blocks corresponding to ‘file_location’ into cache;
        }
        Else { // Cluster C is not preloaded, we need to load some portions of cluster into cache
                // N%: percent of cache we want to do pre-load
                Evict N% of the blocks inside cache that does not belong to cluster C;
                Load these N% of the cache blocks with cluster C’s blocks;
                // Now cache stores at least N% of the cluster C’s blocks
                // Later load the rest blocks with demand (similar to demand paging)
        }
}
```

```
Void write_to_disk(data) {
        Int data_size = size_of(data);
        Int cache_data_size = size_of(data_inside_write_cache);
        If (data_size + cache_data_size >= K * 2048)
                // K is a global integer value
                // We propose a min threshold for writing to disk
                Flush data and data_inside_write_cache into disk;
        Else { // Not enough data to write
                Append data to write cache, along with the data’s writing location;
        }
}
```

**Performance optimization:**

1. The file system tries to preload N% of cache whenever a user opens a file from a cluster. Users tend to access files in the same cluster by access pattern. Thus, we decrease the time to do other file operations. By picking not large N%, we avoid some penalty when a user does not use the pre-load data.
2. According to write_to_disk(), data will be written to disk only if the total written data's size passes a minimum threshold. This helper function avoids the disk to be occupied with the write operation.
3. Clusters are grouped by the user's access pattern. Data inside each cluster is also physically close to each other. Thus, the temporal locality is met.

**Missing requirements :**

1. Our file system is for single user and single process only. Conflicts happen if more than one user/process tries to modify i-node data or data block at the same time.
2. There will be only one **File Descriptor Table (FDT)** in memory. We implemented the FDT as an array with each entry as an OpenFileObject. An OpenFileObject has three attributes: iNodeID, readPtr, and writePtr. The OpenFileObjectUsed is a pre-defined null object whose iNodeID equals 0, and two pointers point to NULL.
3. We assume that the **locality access is within directories:** files and subdirectories in the same directory will be accessed together. This was confirmed by the professor. Hence, each cluster is simply a big directory. For example, if a file ABC.txt is created in directory D, ABC.txt will go to the same cluster as D.
4. The create() function assumes it can find a free i-node slot inside the cluster where its parent directory i-node resides. The create call fails if it cannot find a free slot in this specific cluster. Clustering by locality is violated if we allow newly created i-nodes to reside in other clusters because sooner or later, every cluster will have some i-nodes which belong to another access pattern group if we do so.

**Explanation of the second cluster architecture:**

The I-node table and free bitmap are placed together to reduce the time of finding these two data. We partition the data block into two parts and put them onto two ends of a cluster. The i-node table and free bitmap are allocated in the center of a cluster. This design improves performance because the disk head will always be in the center of a cluster. When the disk head goes to the data block with max distance d in a standard cluster structure, its travel distance is reduced by half to (d/2) in our optimized setting because 1) the disk head only need to go to one of the two splitted data blocks, and 2) each splitted data block has max travel distance (d/2).