COMP310/ECSE427 Assignment 4

Violet Wei 260767913
Glen Xu 260767363
Guoshuai Shi 260836752
Cynthia Shi 260860344

1. What is the block size you want to use? Why did you select the block size value? (50 words)

   Block sizes of 4KB are used. Since the file size of 4KB has the highest popularity, a block size greater than 4KB would lead to more internal fragmentation and a block size less than 4KB would increase the disk movement.

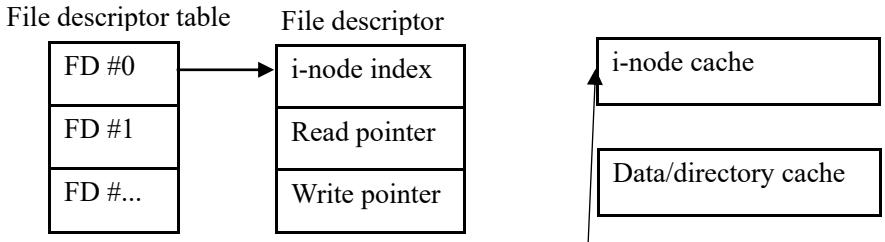
2. What is the file system candidate you want to use? Why? (250 words)

   We choose the clustered i-node file system. In a normal i-node file system, the average distance between the i-node block and the block it is pointing to is half the disk size, which is very long in our case (1TB). However, in a clustered i-node file system, the files are grouped into clusters based on access patterns, and each cluster has their own i-node table. Therefore, the largest average distance would be half the size of a largest cluster, which could be much shorter than 1TB, hence the seek time would decrease. Also, since the files are grouped according to access pattern, a block being accessed now will have a neighbour being accessed next, which will also increase performance. Furthermore, an i-node table for a cluster could even be small enough to be cached into 5% of the memory; even if it is not, caching part of it will still do the work, since the order of i-nodes is related to the order of access. Compared to the FAT file system, it is more stable since a single block corruption will not affect the whole file, and it takes much less space and can avoid external fragmentation.

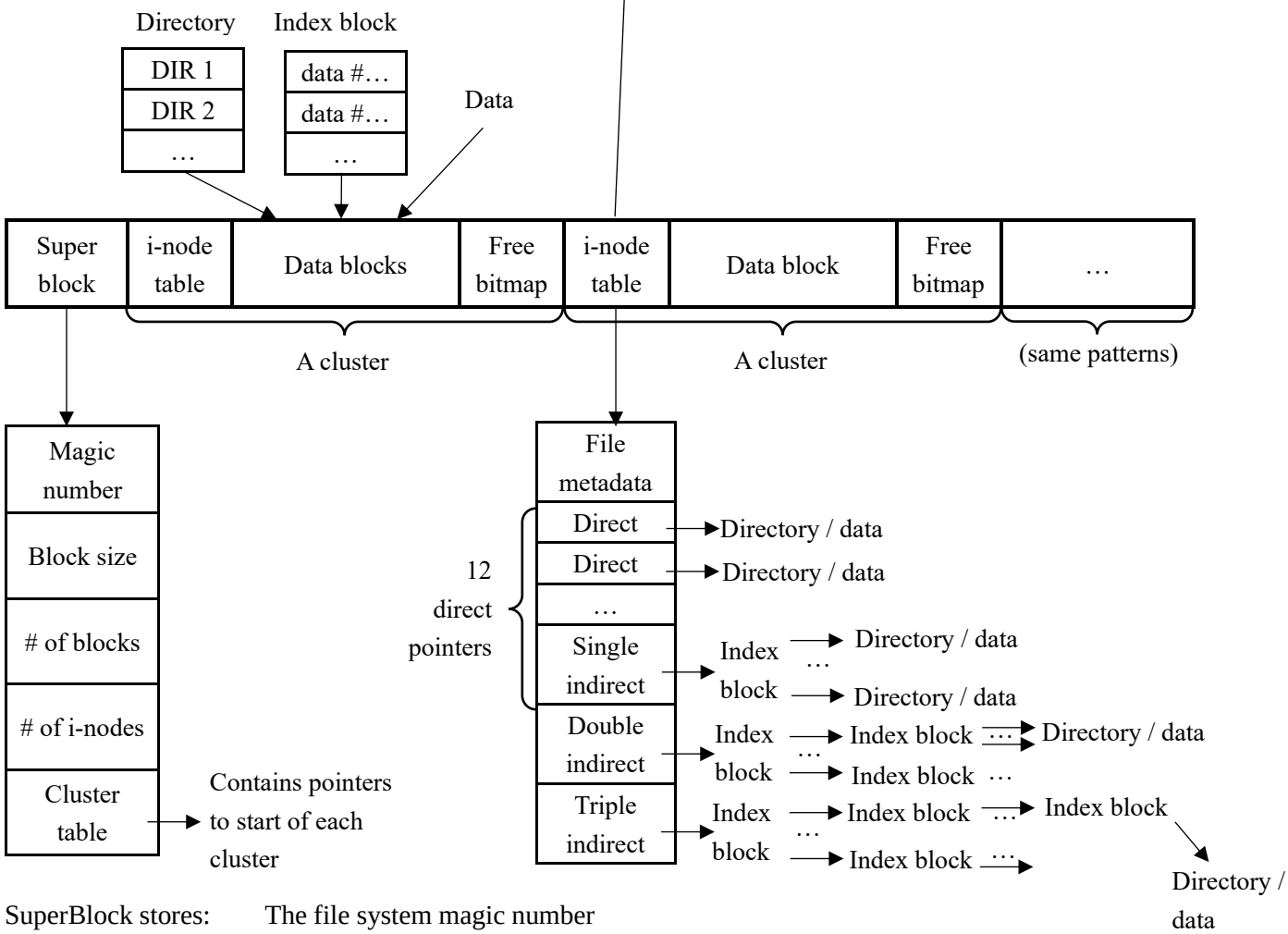
   (Diagram is on the next page since it could not fit in this empty space)

3. Give a diagram to illustrate your file system design. Mark all important components there.

Memory:



Disk:



SuperBlock stores: The file system magic number
Block size
Number of blocks in the file system
Number of i-nodes in file system
Cluster table

Each Inode stores: File metadata (file size, type, etc)
direct_pointers[12]
single_indirect_pointer
double_indirect_pointer
triple_indirect_pointer

Each Cluster stores: i-node table
data blocks
free bitmap

Each file descriptor stores: i-node index
readPtr
writePtr

4. Give the pseudo code for the following operations: create, open, read, write, close, seek (should not exceed 300 lines in total). Pseudo code must be self explanatory. If you add comments to explain concepts, they count towards the pseudo code lines.

```
int createFile(filename){
      if (filename already exists) return –1;
      clusterIndex = belongToCluster(filename);
      cluster = superblock.clusterTable[clusterIndex]
      inode = initialize new inode;
      inode.size = 0;
      inode.type = file.type;
      iNodeIndex = allocate inode in cluster, if failed return -1;
      if(iNodeIndex = -1) return -1;
      block b = find directory block in cluster, if failed return -1;
      if (b == -1) return -1;
      if (Block b has enough space)
            b.append({fileName, iNodeIndex});
      else
            block n = allocate a new directory block in parent directory
      i-node;
        n.append({fileName, iNodeIndex});
      Flush I-Node cache to disk, flush the written directory block to
      disk;
      return 0;
}

int open(filename){
      newFileID = -1;
      if (filename file does not exist)
               return -1;
      inodeIndex = get inode index of filename;

      for i from 1 to (fdTable.length) do
            if (fdTable[i] not empty && fdTable[i].inode == inodeIndex)
                  return i;
      for j from 1 to (fdTable.length) do
```

```
            if (fdTable[j] is empty)
                  newFileID = j;
                  break;

      if (newFileID  == -1)
            newFileID = fdTable.length + 1;
            Double the size of file descriptor table;

      fdTable[newFileID] = allocate a new file descriptor;
      fdTable[newFileID].readPtr = 0;
      fdTable[newFileID].writePtr = size of file;
      fdTable[newFileID].inode = inodeIndex;

      Flush i-node table of the cluster the file belongs to cache, if the
      table in cache is not already this particular one;

      return newFileID;
}

int read(fileID, buf, length) {

      if (length < 0 || buf == NULL || fileID >= fdTable.length ||
      fdTable[fileID] == NULL)
            return -1;

      if (fdTable[fileID].readPtr + length) > (size of file)
            length = (size of file) - fdTable[fileID].readPtr;

      temp_buf[(# blocks to read) * blockSize];
      Flush the block after the first block to read to data cache;

      for i from first block to read to last block to read do{
            blockNumber = (inode of file).(i-th block of data);
            if (blockNumber == data block number in data cache){
                  Fetch data from cache
            else {
                  Fetch data from disk
            }
            Flush next block after blockNumber to data cache;
      }

      // copy 'length' many bytes from temp_buf to buf
      memcpy(buf, temp_buf + (fdTable[fileID].readPtr % blockSize),
      length);
```

```
        seek(fileID, fdTable[fileID].readPtr + length - 1);
        return length;
}

int write(fileID, buf, length) {
        writePtr = fdTable[fileID].writePtr;
        inode_no = fdTable[fileID].inode;
        add_blocks = 0;

        temp_buf[(# blocks to write) * blockSize];

        for i from first block to last block to write do {
              if (file occupies i many blocks) {
                    blockNumber = (inode of file).(i-th block of data);
                    if (blockNumber == -1) return -1;
                    Append data in blockNumber to end of temp_buf;
              } else {
                    blockNumber = allocate i-th block for file with inode_no
                    closest to the (i-1)-th block;
                    if (blockNumber == -1) return -1;
                    added_blocks = 1;
              }
        }

        // copy the buffer into the block
        memcpy(temp_buf + (writePtr % blockSize), buf, length);

        if writePtr + length > (size of file) {
              size of file = writePtr + length;
              Flush inode table;
              if (added_blocks) {
                    Flush free bitmap;
              }
        }

        for i from first block to last block to write do {
              blockNumber = (inode of file).(i-th block of data);

              if first block = 0 {
                    buffer = temp_buf + i * block size;
              } else {
                    buffer = temp_buf + (i % first_block) * block size;
              }
```

```
            write_blocks(starting from: blockNumber, number of blocks to
            write: 1, content: buffer);
        }

        seek(fileID, writePtr + length - 1);

        return length;
}

int closeFile(fileID){
      if (fileID <= fdTable.length) {
        Set fdTable[fileID] empty;
        return 0;
      }
      return -1;
}

int seek(int fileID, int loc) {
      if(fileID > fdTable.length || loc < 0 || loc >= size of file
      indicated by fileID) return -1;
      fd = fileDescriptorTable[fileID];
      fd.readPtr = loc;
      fd.writePtr = loc + size of file;

      return 1;
}
```

5. Optionally give the pseudo code for the performance improvement helper functions that could be used in the above implementation (100 lines in total).

```
clusterIndex belongToCluster(fileID){
      for clusterIndex i in cluster table{  // stored in super block
            if (filename belongs to cluster i) return i;
      }
      return -1;
}
```

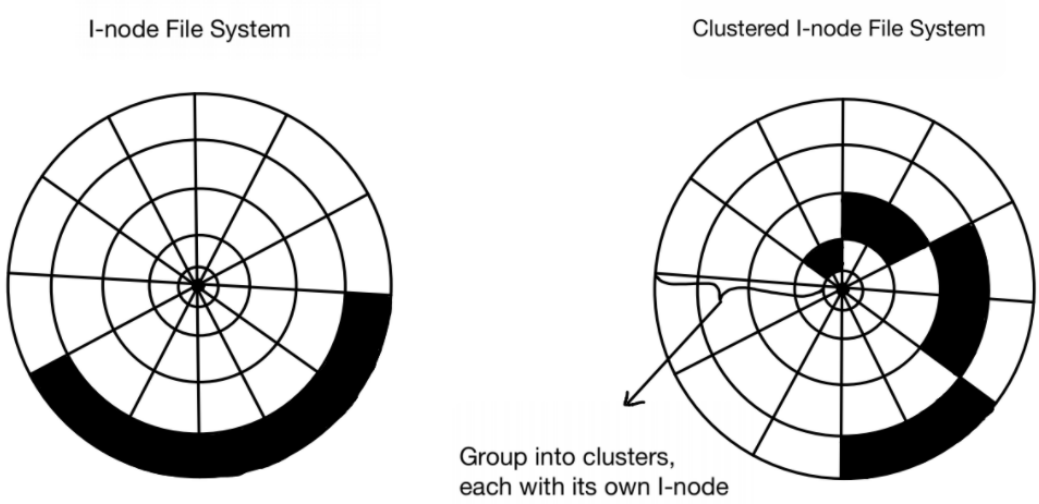
6. Other explanations (500 words).

We will discuss how our file system performance is optimized, and after this some missing requirements.

Optimization

1. Since the access patterns of files are given in advance, we grouped the files into clusters of files that are going to be accessed close to each other and assign an i-node table to each cluster of files to reduce disk arm

motion. The diagram below shows how the seek time in our design gets reduced compared with normal i-node file system architecture on the left, where i-node is located at the start of the disk. We have already explained the details in question two, but the basic idea is the average distance between i-node and the data block we want to locate is half the distance between the start and end of a sector (the whole disk in the normal i-node file system and the size of the (largest) cluster in the clustered i-node system).



2. Caching: the permanence can be optimized by reducing disk access time through block cache or buffer cache. Cache is defined as a collection of items of the same type stored in a hidden or inaccessible place. Therefore, if a disk access is initiated, the cache is checked first to see if the disk block is present. If there is a hit, then the read request can be satisfied without a disk access else the disk block is copied to cache first and then the read request is processed.

3. Free space allocation: since create is barely used, we will not discuss it a lot, but basically our create function guarantees that a new file is put into a directory with free space. In our write, although it is mentioned that a file growth will not exceed 10%, we still make sure that a if a new block is allocated for writing purpose, it is as close to the last block of the file as it should be. That way we can reduce future reading and writing seek time.

4. Block Read Ahead: We could improve the file system's performance by trying to get blocks into the cache before they are needed to increase the hit rate. However, this works only when files are read sequentially. When a file system is asked for block '*k*' in the file it does that and then also checks beforehand if '*k*+1' is available if not it schedules a read for the block '*k*+1' thinking that it might be of use later. Since files are already grouped into clusters at file creation time, we assume that the file blocks are also arranged sequentially, thus this method can indeed improve the performance.

Missing Requirements

1. create call would replace the file when a file is already there.
2. When the current file descriptor table is full, the file descriptor table is copied to the new file descriptor table where the size is doubled.