ECSE 427 Assignment 4

Yujing Yang 260827923

Han Zhou 260763624

Helen Ren 260846901

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

4k Bytes.

According to the given information, most files are with size 1024 - 8k bytes. 4k bytes is reasonable as smaller block size will cause more splits and increasing allocation times, and larger size leaves more unused space. The largest file is 16M, which can fit in the inode system.

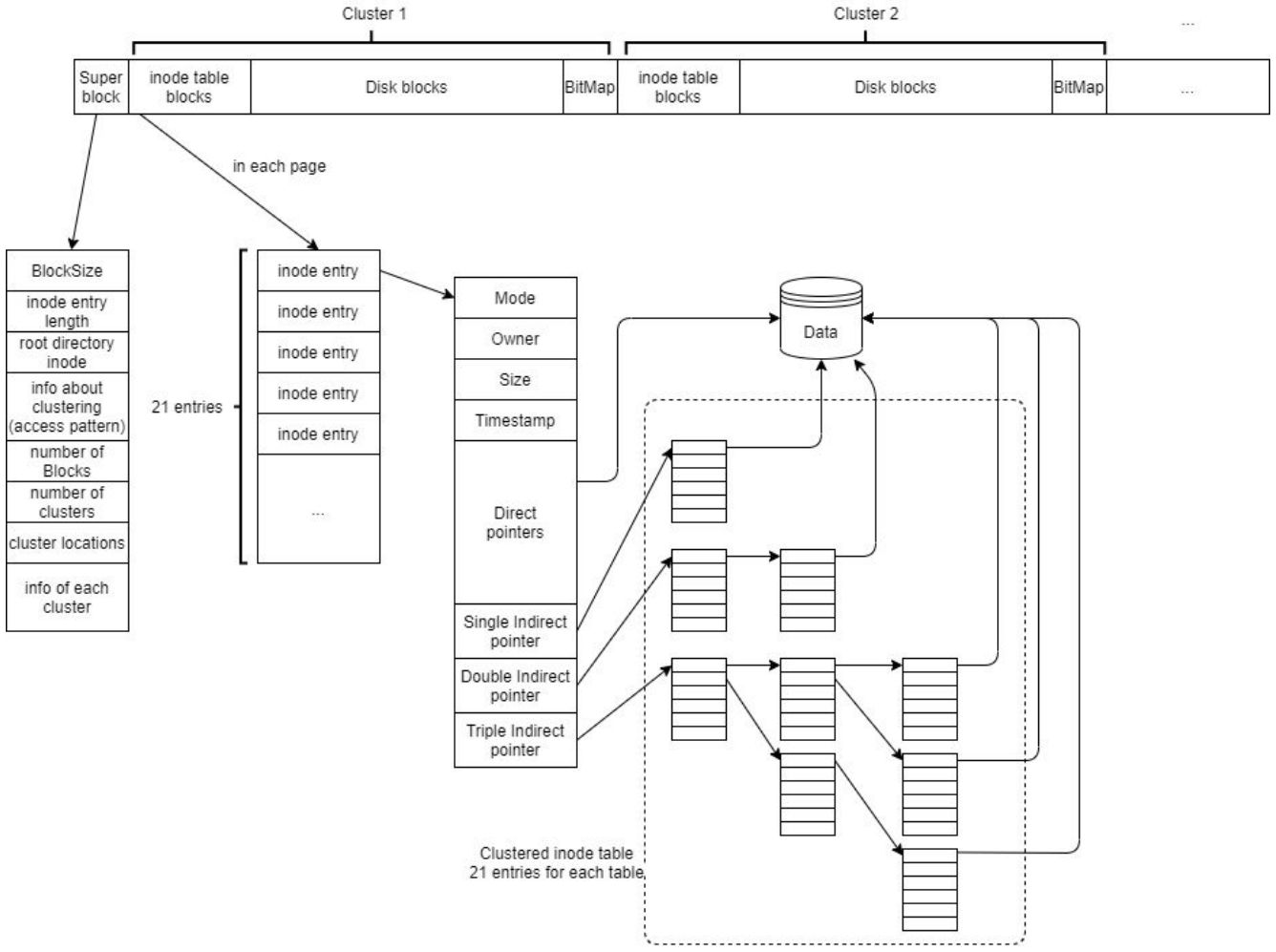
2. **What is the file system candidate you want to use? Why? (250 words)**

Cluster I-node file system.

Unlike the FAT system which directly represents the disk. The i-node file system replaces the FAT table with an i-node table. The FAT allocation scheme can result in a significant amount of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. In this design where the majority of activities is read and write, the FAT system is less efficient than the inode system.

Compared with inode table, uncluster i-node table would have a better performance, especially since we have almost all the files at the file system creation time and we know the access patterns of the file. With this information, we can group these files into clusters of files that are going to be accessed close to each other, thus further reducing overhead of modifying the read/write pointer. In this case, using a cluster file system can guarantee a fast access to target files.

3. Details of the selected architecture



note: Since the access pattern of all files are given, we define the cluster based on the abscess pattern, the files that are accessed closely are arranged in the same cluster.

4. C-like pseudo code for create

```
int create (char *fileName){
// create a file, return 0 if success, -1 otherwise
        if (strlen(fileName) >  MAX_FILE_NAME)  return -1    // return -1 if the file name > 32 bytes (missing requirement)
        for (int i = 0; i < nodeNumbers; i++){
                if (strcmp (iNodeList[i].name, fileName)  = = 0 ) return -1    // return -1 if the fileName exist or already
opened (missing requirement)
        }
        inode *newNode = (inode*) malloc (sizeof (inode))    // create a new inode
        inode->linkCount=0; inode->size=0; inode->bp=NULL
        In the create function, we first check the access pattern and create the file according to the target cluster of the file.
        n = get_inode_from_cluster(cluster,name)
        Allocate an I-Node on disk with inode Index, return -1 if fails
        block_ptr = get_block_ptr(n,inode_idx) // get direct/indirect block pointer block_ptr
        if (block_ptr has enough space){   // if only one block is needed
                block_ptr.append({fileName, inode->id})
        }
        else{
                allocate a new block new_block_ptr
                new_block_ptr.append({fileName, inode->id})
        }
```

```
        write_inode_to_disk( n.i_num, n.i_node))  //Flush I-Node cache to disk, flush the written directory block to disk
        return 0
}
```

5. **C-like pseudo code for open**

```
int open (char *name, fd_entry* cluster){
// Start open file from given file name and cluster. If success, return the fileDescriptor. Otherwise, return -1.
In the open function, we first check the access pattern and find the target cluster for the file.
        // cluster is an array contains the all clusters in sequence according to pattern
        n = get_inode_from_cluster(cluster,name)
        lif (n == NULL) return -1//if the inode cannot be found, return -1(missing requirement)
        fd = get_free_fd(cluster)
        if(fd==-1) return -1 //if the file descriptor cannot be found, return -1(missing requirement)
        // attach information of inode to the clustered inodes table
        cluster[fd] <- n
        return fd
}
```

6. **C-like pseudo code for read**

```
int read (int fileDescriptor, fd_entry* cluster, void *buf, int length){
// Start reading from the cursor position and store the read data into a buffer. If success, return the length. Otherwise, return
-1.
        n = cluster[fileDescriptor]
        if(n.status==1) or n.rw_pointer+length>MAX_FILE_SIZE // if the file is already closed or the rw_pointer after
modified would excess MAX_FILE_SIZE return -1 (missing requirement)
        if (n.rw_pointer+ length > n.i_node->size)      // adjust reading length to the rest of the inode block
                length =n.inode->size - n.rw_pointer
        block_idx= (n.rw_pointer) / BLOCK_SIZE      // find target block index and occupied size
        entry_idx=(n.rw_pointer) % BLOCK_SIZE
        if ((entry_idx+ length ) <= BLOCK_SIZE){      // if only one block is needed
                buf <- read_block(n , block_idx)
                n.rw_pointer += length
                return length
        }
        last_block_idx=((entry_idx+length) / BLOCK_SIZE) + block_idx   // if not, read the last block index
        last_entry_idx=(entry_idx+length) % BLOCK_SIZE
        buf <- read_blocks(block_idx)
        buf += BLOCK_SIZE - occupied
        block_idx++
        while(block_idx <= last_block_idx) {              // write to buf until the pointer reach the last block index
              buf <- read_block(n, block_idx)             // read from the disk (by block index) to the buffer
              buf += BLOCK_SIZE
              block_idx++
        }
        n.rw_pointer += length
        return length
}
```

7. **C-like pseudo code for write**

```
int write (int fileDescriptor, fd_entry* cluster, void *buf, int length){
// Start writing the data from  buffer to the cursor position If success, return the length. Otherwise, return -1.
        n = cluster[fileDescriptor]
        if(n.status==1) or n.rw_pointer+length>MAX_FILE_SIZE// if the file is already closed or the rw_pointer after
modified would exceed MAX_FILE_SIZE return -1 (missing requirement)
                return -1
        if (n.status == 1) return -1   // if the file is already closed, return -1(missing requirment)
        if (inodes[fileDescriptor]->rw_pointer + length > MAX_FILE_SIZE) return -1 // if exceed max file size, return -1
        block_idx= (n.rw_pointer) / BLOCK_SIZE
        entry_idx=(n.rw_pointer) % BLOCK_SIZE
        if ((entry_idx+ length ) <= BLOCK_SIZE){   // if only one block is needed
                write_block(n,buf,block_idx,entry_idx)
                n.rw_pointer += length
                return length
        }
        last_block_idx=((entry_idx + length) / BLOCK_SIZE) +block_idx   // if need multiple block, read the last block index
        last_entry_idx=(entry_idx + length)% BLOCK_SIZE
        write_block(n,buf,block_idx,entry_idx)
        buf += BLOCK_SIZE - entry_idx
```

```
        block_idx++
        while(block_idx<last_block_idx){
                write_block(n,buf,block_idx,0)
                buf+= BLOCK_SIZE
                block_idx++
        }
        n.rw_pointer=rw_pointer+length
        return length
}
```

8. **C-like pseudo code for close**

```
int close (int fileDescriptor, fd_entry* cluster){
// close the file and make the file descriptor available , 0 for success, -1 otherwise
        n = cluster[fileDescriptor]
        if (n.status == 1) return -1   // if the file is already closed, return -1 (missing requirement)
        write_inode_to_disk(n.i_num,n.i_node)
        free(n.inode)       // free inode from memory
        n.status = 1        // mark as closeds
        return 0

}
```

9. **C-like pseudo code for seek**

```
int seek (int fileDescriptor, fd_entry* cluster, int offset){
// modify the position of the read and write pointer of a file, 0 if success, -1 otherwise
        n = cluster[fileDescriptor]
        if (n.status == 1) return -1   // if the file is already closed, return -1 (missing requirement)
        if (n.i_node->size <= offset) offset = n.i_node->size  // if size of inode is lesser than offset, readjust offset measure
        n.rw_pointer = offset  // set the pointer with offset
        return 0
}
```

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

```
int read_block(inode* n, int block_idx){
        block_ptr = get_block_ptr(n,inode_idx) // get direct/indirect block pointer
        if(block_ptr >MAX_BLOCK) return -1  //missing requirement
        fseek(fp, block_ptr *BLOCK_SIZE, SEEK_SET)  //go to the data requested from the disk
        buf <- fread(tmp, BLOCK_SIZE,1,fp)
        return buf
}

int write_block(inode* n,char* buf,int block_idx,int entry_idx){
        block_ptr = get_block_ptr(n,inode_idx) // get direct/indirect block pointer
        if(block_ptr >MAX_BLOCK) return -1 //if the block is full missing requirement
        content = (entry_idx != 0)? read_block(n,block_idx) : empty string // if entry_idx is not 0, copy blocks into content
        content += buf
         // go to where the data is to be written on the disk
        fseek(fp, block_ptr*BLOCK_SIZE, SEEK_SET)
        fwrite(content , BLOCK_SIZE, 1, fp)
}

int get_free_fd(inode** cluster){
        for(i=0;i<MAX_OPEN_FILES; i++){
                if(cluster[ i ].status==1){
                        cluster[i].status=0
                        return i
                }
        }
        return -1
}
```

```
int write_inode_to_disk( int i_num, inode i_node ){   // return -1 if fail to write to disk, return 0 if success
        block_idx= (n.rw_pointer) / BLOCK_SIZE
        entry_idx=(n.rw_pointer) % BLOCK_SIZE
        read_block(INODE_BEGIN_NEW+block_idx, 1, buf)
        inode_array = (inode*)buf
        memcpy(&inode_array[entry_idx], inode, sizeof(inode_t))
        write_block(INODE_BEGIN_NEW+block_idx, 1, buf)
}
```

Other explanations (500 words).

1. **How we are able to handle the largest files in our cluster inode file system.**

The largest file size is 16M Byte in our system, the block size is 4k Byte, and we are using the default inode size of 128 Byte. Using triple indirect pointers, the largest file size we can handle will be:

$$(4096/128)^3 \times 4096 \;=\; 134M\ Byte$$

While using 2k Byte for block size would handle 8M Byte file maximum, we picked the lowest possible page size in multiple of 1024.

In addition, according to the distribution of file sizes, the majority of files are smaller than 32K and can be handled by single indirect pointers. Only a few files may need triple indirect pointers. 4k Bytes block size is reasonable as smaller block size will cause more splits on file thus increasing allocation times, and larger block size leaves more unused space in each block.

In this case, the chosen block size and file system is effective for given distribution of file sizes.

2. **The struct we used for this cluster inode system is**

```
inode{
        int size;
        int id;
        block_pointer bp;
        int linkCount;
}

fd_entry{
        int status;
        int i_num;
        inode i_node;
        int rw_pointer;
}

block_pointer {
        int[] direct;
        int indirect;
}
```

3. **Optimization of clustered inode method**

We have implemented a series of design patterns to ensure the optimized operating efficiency of our designed program.
In terms of clustering, we keep the sequential information in terms of access near to each other, which will further more improves the access efficiency from the inode file system.
We choose 4k Bytes as the block size. As we concluded from the diagram, most of the file sizes fall in the range of 1024 - 8k bytes. 4k bytes block size then becomes a reasonable choice for page size, as smaller block size will cause more splits and can’t handle large files, and larger block size leaves more unused space in each block.
Moreover, caching can help to gain performance. By loading the i-nodes table, data/directory table to memory cache evenly, accessing the inode entry information can be even faster.