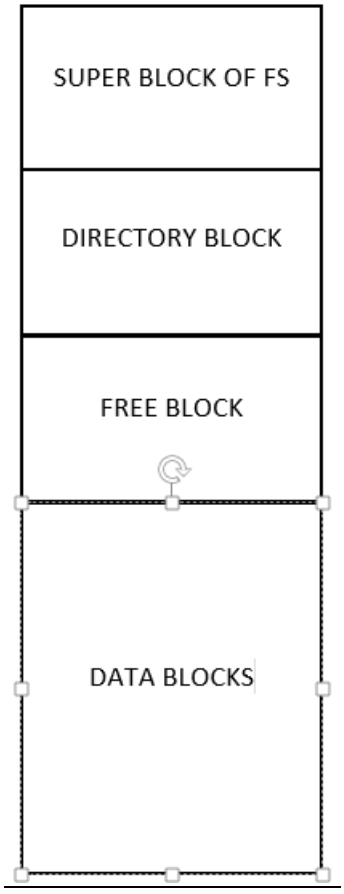
CSCE 611: MP7: DESIGN DOCUMENT
AUTHOR: SRIVIDHYA BALAJI
UIN: 827007169

PROBLEM STATEMENT:

1) To implement a simple File System
2) [BONUS 1] : Design of Thread safe File system

IMPLEMENTATION DESIGN :

Two classes File System and File are used together in order to implement the Simple File System. The file system controls the mapping from the file name space to files, handles file allocation, free block management and other issues. The File contains interfaces for reading a file, writing a file, resetting the file pointer and other file operations. The disk is initialized with three reserved blocks. One for Super Block of the File system which contains details about the File system, ie, the number of files, size of the File system, first free block in the file system. The second block is the directory block, which maps each file inode with the filename, and the third block is the free block information. Remaining blocks from the fourth block are data blocks till the end of the disk. Each disk block is 512 bytes.



Initially, the first three reserved blocks are initialized, then the freeblocknumber which maintains the first freeblock available is stored, and the entire disk with the remaining blocks is initialized as free blocks as the disk is initially empty. The file system maintains a linked list of the files being allotted in the disk and updates the FreeBlockNumber to the Next free block number that is available. While reading and writing a file, the file accesses its startblock information and with the help of its size, reads from the disk. Similarly while writing a file, its writes to the disk and updates the corresponding size as its file size and maintains its necessary blocks details. Similarly when a file is deleted, its entry in the linked list is removed, and its blocks are freed and the freeblockNumber is correspondingly updated. The next sections talk about the various interfaces in File System and File class.

INTERFACES IN FILE SYSTEM :

1) **FileSystem::Format(SimpleDisk * _disk, unsigned int _size)**
   This function allocates the _size as the size of the File system. It also uses the _disk object to initialize the blocks in the disk. As mentioned above, it initializes the reserved blocks 0,1,2 and then initializes the other data blocks as free blocks as none of them are allocated to files currently.

   PSUEDO CODE:

```
Bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size)
{
  //Create a new buffer.
  unsigned char *buf = new unsigned char[512];
   for i= 1 to RESERVEDBLOCKSIZE
        disk_write(i,buff)

       /*for data blocks each block consists the last 4 bytes as that which points to next free data,
         which is the next block as the file system is implemented and allocated in a sequential
         manner* /
 For I = RESERVEDBLOCKSIZE upto  _size/512 -1
       Save the last 4 bytes of buff= i+1
       disk.write(I,buff)

/*Allocate the last block's last 4 bytes as -1 which indicates the end of the file system blocks*/
     Last 4 bytes of buff = -1
     Disk.write(I,buff)

/*Initializing the first super block */
      First 4 bytes = _size
      Next 4 bytes = RESERVEDBLOCKSIZE // which indicates the first free block
      Next 4 bytes = 0 // number of files
      Initialize the remaining 500 bytes to 0
     Disk.write(0,buff) //writing this initialized buffer to the first block in the disk which is the
super block
Return true }
```

2) **FileSystem::Mount(SimpleDisk * _disk)**

This function reads the disk and initializes the FILE SYSTEM variables with the content in the disk and mainly allocates the disk object as the FILE SYSTEM's disk.

PSEUDO CODE:

```
Bool FileSystem::Mount(SimpleDisk * _disk)
{
    disk = _disk //file system's disk is initializes with the system disk

     disk->read(0,buf) //reading the super block from the disk.
     Size = first 4 bytes of the buffer //represents the FS size
     freeBlock = next 4 bytes of the buffer // this represents the first free block in the FS
     fileCount = next 4 bytes of the buffer //  this represents the total number of files in the FS.
Return true
}
```

3) **bool FileSystem::CreateFile(int _file_id)**

This function creates a file, with the _file_id, ie. It first extracts the available free block in the File System, uses it as the Info block for the file, and from the last 4 bytes of this info block, it gets the next sequentially available free block. These two blocks are assigned as the startinginfoblock and startingdatablock for the new file and are used to create a File object which holds these parameters local to its object which represents each file. Further, we also maintain a linked list of the files that are created in the File system and allocate this file object in the Linkedlist too.

PSUEDOCODE:

```
bool FileSystem::CreateFile(int _file_id)
{
  startInfoBlock = freeBlock
  disk->read(freeBlock, buff)

/*    The last 4 bytes of this buffer contains data regarding the next block in the disk which is assigned as the startdatablock.*/

startDataBlock = last 4 bytes from freeBlock number in the disk

disk->read(startdatablock, buff)

/*Last 4 bytes of this buffer which points to the next block, is updated as the next available free block*/
freeBlock = last 4 bytes from startblock number in the disk

File * newCreatedFile = new File(&startInfoBlock, &startdatablock) //function explained while explaining class File
```

```
addToFileList(newCreatedFile, _file_id)  //function to add to the Linked List

file_count = file_count+1
Return true
}
```

Function to add to the Linkedlist :

The File Node is of file_blockNode type which contains parameters such as file ID, file, next pointer.

```
void FileSystem::addToFileList(File *newFile,unsigned int newfileID)
{
   file_blockNode *node = new file_blockNode();
   node->fileID = newfileID
   node->file = newFile
   node->next = NULL

   if(filecount == 0)
       fileListHead = fileListTail = node
  else
       fileListTail->next = node
       fileListTail = node

}
```

4) **File * FileSystem::LookupFile(int _file_id) :**

This function is used to Lookup if a file with the given FILE ID exists, in the File system. We do this by checking for the file ID in the Linked List of the files we store to represent the file List in the file system. If we find the node with the input file ID, we return the file corresponding to that node.

PSEUDO CODE:

```
File * FileSystem::LookupFile(int _file_id)
{
   file_blockNode *newNode = fileListHead;

    if(newNode == NULL)
        return NULL
    else
    {
          For I  = 0 till file_count
                If(newNode->fileID == _file_id
                    Return newNode->file
          newNode = newNode->next
     } }
```

5) bool FileSystem::DeleteFile(int _file_id) :

In this function, we delete the file with the input file ID, ie we de-allocate the memory allocated to it as info and data blocks and also remove its entry from the FILE SYSTEM's LINKEDLIST.

PSEUDO CODE:

```
bool FileSystem::DeleteFile(int _file_id)
{
removeFromFileList(_file_id);
filecount = filecount -1
RETURN true
}

void FileSystem::removeFromFileList(unsigned int fileID)
{
 Prev = NULL ; curr = fileListHead
For I = 0 to file_count
    If(curr->fileID == _file_ID)
          Deallocating assigned fileinfo block and file datablock
           Updating the new freeBlock as the filenfo block
           Break
Prev = curr; curr = curr->next

/*If node to be removed is the last node in the list*/
 if(curr->next == NULL)
    if(prev != NULL)
       prev->next = NULL;
   fileListTail = prev;
 else
  if(prev != NULL)
    prev->next = curr->next ; // current node reference is removed from the LL.
```

```
  else
        fileListHead = curr->next;
}
```

INTERFACES IN THE FILE CLASS:

1) **int File::Read(unsigned int _n, char * _buf) :**
   This function reads _n bytes from the given file based on the current position and the position of bytes being read everytime.

   PSEUDO CODE :

```
int File::Read(unsigned int _n, char * _buf)
{
     Readposition = 0;

    while(read_position < _n)
{
    FILE_SYSTEM->disk->read(current_block,diskReadBuff);
   readBytesRemaining = _n -read_position
   blockBytesRemaining = 508 - current_pos

   if(readBytesRemaining <= blockBytesRemaining)
    {
          _buf + readpostion = readBytesRemaining from diskReadBuff+current_pos
           read_position = read_position + readBytesRemaining;
           current_pos = current_pos + readBytesRemaining;

           if(read_position == _n)
                return _n;

    }
Else
{
          _buf + readpostion = blockBytesRemaining from diskReadBuff+current_pos
          read_position = read_position + blockBytesRemaining;
           current_pos = 0;

            current_block = next block number obtained from last 4 bytes of diskReadbuff
}
}
```

Return _n //indicating the number of bytes that was read.

}

2) **void File::Write(unsigned int _n, const char * _buf) :**
This function writes _n bytes to the file in the disk. While writing , again it maintains a set of counters to track the current position in the file etc.

PSEUDO CODE:

```
void File::Write(unsigned int _n, const char * _buf)
{

    write_pos = 0
     while(write_pos < _n)
     {
          FILE_SYSTEM->disk->read(current_block,diskWriteBuff);

          writeBytesRemaining = _n-write_pos;
          blockBytesRemaining = 508 - current_pos;

           if(writeBytesRemaining < blockBytesRemaining)
            {
                  diskWriteBuff+current_pos = writeBytesRemaining FROM _buf+write
               FILE_SYSTEM->disk->write(current_block,diskWriteBuff);
                write_pos += writeBytesRemaining;
                current_pos += writeBytesRemaining;

               if(current_pos == _n)
                        Console::puts("WRITE COMPLETE \n");
              }
            else
              {
                    diskWriteBuff+current_pos = blockBytesRemaining FROM _buf+write
                   FILE_SYSTEM->disk->write(current_block,diskWriteBuff);

                   write_pos += blockBytesRemaining;
                    current_pos = 0;

                   current_block = FILE_SYSTEM->freeBlock;
                     FILE_SYSTEM->disk->read(FILE_SYSTEM->freeBlock,diskWriteBuff);

                      FILE_SYSTEM-> Freeblock =  last 4 bytes from diskWritebuff //storing the
next available block as free block.
dataBlockCount += 1;
```

```
            }


        }
            if(((dataBlockCount - 1)*512 +current_pos ) > fileSize)
              {
                    /*INCREASE THE FILE SIZE TO THE NEXT BLOCK*/
                       fileSize = dataBlockCount*512 + current_pos;
                       fileEndBlock = current_block;


               }
    }
```

3) void File::Reset() :

This function resets the current position of the file to the start of the file.

PSEUDOCODE:

```
Void File::Reset()
{
current_pos = 0;
current_block = startDataBlock;
}
```

4) void File::Rewrite() :

This function erases the contents of the file and updates the freeblock value and the start block value of the file.

PSEUDOCODE:

```
Void File::Rewrite()
{
temp_curr_block = startDataBlock
temp_next_block = 0

for I = 0 to dataBlockCount
      if(temp_next_block == -1)
                  break; // Last block in the FS

     FILE_SYSTEM->disk->read(temp_curr_block,diskReadBuff);
     Temp_next_block = last 4 bytes of the diskReadBuff
```

```
        diskWriteBuff+508 = FILE_SYSTEM->freeBLOCK
        if(startDataBlock != temp_curr_block)
                FILE_SYSTEM->freeBlock = temp_curr_block;

        temp_curr_block = temp_next_block;
/*Resetting the file counters once its contents are deleted*/
  fileSize = 512;
  dataBlockCount = 1;
 current_block = startDataBlock;
 }
```

5) bool File::EoF() : This function is used to tell if End Of file is reached.

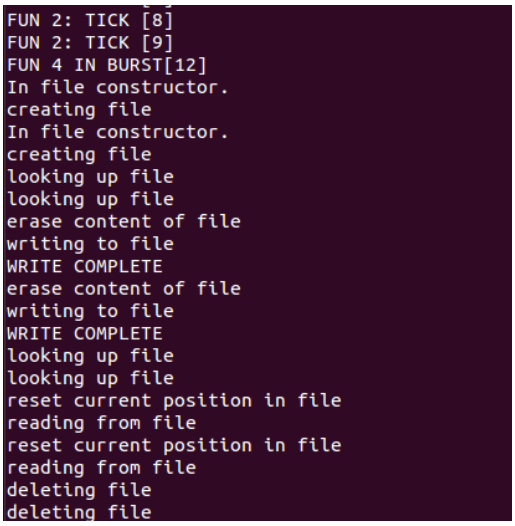
PSUEDO CODE:

```
bool File::EoF()
{
return (dataBlockCount*512 + current_pos ) == fileSize

}
```

TEST CASES :

The above implementation was tested using the exercise_file_system() code in Kernel. C. Under this all the file system implementations for both the files were executed and the code did not assert. Screenshot of the exercise_file_system() output for a particular burst is as below.



[BONUS :1] THREAD SAFE FILE SYSTEM : DESIGN:

In order to implement a thread safe File system, it is necessary to ensure that when the disk is being accessed by one of the File or File System functions of a particular thread, the disk access code in those functions are the critical code section and needs to be locked while it is being used by one thread. It is unlocked when the disk functions and the pointers of the file system is correspondingly updated.

This can be illustrated as below,

```
File * FileSystem::LookupFile(int _file_id)
{
    while(! filesystem_locked)
        filesystem_locked = true
       /*LOOK UP FUNCTIONS OF ACCESSING THE DISK*/
        filesystem_locked = false
       return the file
}
```

As seen when the first thread accesses this function filesystem_locked which is initially false is checked and the thread enters the critical section making it true. So when another thread pre-empts this and tries to enter the critical code section, it wouldn't as filesystem_locked is set as true and hence it waits for thread 1 to complete the Look up function and reset filesystem_locked variable to false again. This ensures only one thread can access the disk ,file and file system sections at a time, thus making the FILE SYSTEM a THREAD safe system.