```
//------------------createFile----------------------------------------------------
int createFile(char* fileName, int permission){
    struct Block* directoryBlock = getCurrentDirectory();
    if(FileAlreadyExist(fileName)) return -1;
    struct iNode newINode = {permission, size = 0};
    if current cluster is full on data block || full on INode
        find nearest cluster with enough space (inode + one block);
        int iNodeIndex = allocateINodeInThatCluster(newINode);
        if allocate fails return -1;
    else
        int iNodeIndex = allocateINodeInCurrentCluster(newINode);
        if allocate fails return -1;
    // Do preallocation, all files will automatically have one block (4k) at creation
    allocateFirstDataBlock(iNodeIndex);
    if(isFull(directoryBlock))
        Append a newBlock after current directory blocks in its parent directory;
        newBlock.append({filename, iNodeIndex});
    else
        directoryBlock.append({filename, iNodeIndex});
    updateINodeInCache(); // Write changed directory to the disk
                // And load Inode table of this cluster in cache
    return 0;
}
//--------------openFile------------------------------------------------------
int openFile(char* fileName){
    if (file is already opened)
        return getFileDescriptor(fileName);

    if fileDescriptorTable is full return -1;
    int newFD = findEmptyEntry(fileDescriptorTable);

    int iNodeIndex = findINodeIndex(fileName);
    if iNode not in cache:
        cluster = find the cluster containing the INode in Disk;
        loadINodeFromDisk(cluster);
    if not a single cluster contain this INode
        // file does not exist yet. Try create first
        if (createFile(fileName) failed) return -1;

    fileDescriptorTable[newFD].iNodeIndex = iNodeIndex;
    fileDescriptorTable[newFD].readPtr = FILE_START;
    if(peekFilePermission(iNodeIndex) == READ_ONLY)
        fileDescriptorTable[newFD].writePtr = NULL;
    else
        fileDescriptorTable[newFD].writePtr = FILE_START;
    return newFD;
}

//----------------seek------------------------------------------------------------
int seek(int FDindex, int offset, char readOrWrite){
    // the seek updates the selected pointer by number specified in offset
    // if update is successful, this method return the pointer value prior to the update
    // (Analogy to sbrk())
    int pointerToChange;
    if(readOrWrite == "r")
        fileDescriptorTable[FDindex].readPtr = fileDescriptorTable[FDindex].readPtr + offset
        return pointerToChange;
    else if (readOrWrite == "w")
        fileDescriptorTable[FDindex].writePtr = fileDescriptorTable[FDindex].writePtr + offset
        return pointerToChange;
    else
        return -1;
}

//--------------------read----------------------------------------------------------
int read(int fdID, char* buf, int length){
    // check if file descriptor ID is valid
    if(fdID < 0 || fdID > Max_size_of_fdt){
        return -1;
    }

    int INodeID = fileDescriptorTable[fdID].INodeIndex
    int readPointer  = fileDescriptorTable[fdID].readPtr

    // unlikely to happen since the prior open() will bring INode to cache
    // But to cover extreme condition (like the file is opened but its iNode got replaced in cache)
    // we added the condition below. Same thing in write
    if (iNodeIdx exist in cache)
       curINode = get INode in cache;
   else
       curINode = find corresponding cluster and loadINodeFromDisk(cluster);

    // check if read outside of file
    if (readPointer+length > file_size){
        length = file_size - readPointer;    //if outside, change length to read to end of file
    }
   int read_end = readPointer+length;    //end position of read
   char* buf_tracker = buf       //track position in buf
   int byteAlreadyRead = 0;

   while(readPointer != read_end){
      int block_Position_In_INode = readPointer/BLOCK_SIZE;
      int bytesToRead = min(BLOCK_SIZE - readPointer % BLOCK_SIZE ,
length-byteAlreadyRead);
      if(isSingleIndirect(block_Position_In_INode)){
         Target_Block_toRead = single_indirect_pointer[block_Position_In_INode-12];
      }else if(isDoubleIndirect(block_Position_In_INode)){
         int first_level_indirect_location = (i-12-1024)/1024
         int second_level_indirect_location = (i-12-1024)%1024
         Target_Block_toRead =
double_indirect_pointer[first_level_indirect_location][second_level_indirect_location];
      }else{
         Target_Block_toRead = direct_pointer[block_Position_In_INode];
      }

      memcpy(Target_Block_toRead + readPointer % BLOCK_SIZE, buf_tracker,
bytesToRead);

      readPointer += bytesToRead;    //update pointers
      buf_tracker += bytesToRead;
      byteAlreadyRead += bytesToRead;
   }

   updateReadpointer();    //update read pinter in I-node

   return length;
}

//----------------------writeFile-----------------------------------------------------------
int writeFile(int fd, const char* buf, int len){
   // Note every file "borns with one block (4k)"
   // this is to reduce the time of updating/syncing inode
   bool inodeHasChanged = false;
   char bufferForOneBlock [BLOCK_SIZE];
   memset(buffer, 0 ,BLOCK_SIZE); //intialize buffer
   int byteAlreadyWritten = 0;
   iNodeIdx = fileDescriptorTable[fd].iNodeIndex;
   if iNodeIdx is null or invalid
```

```
        return -1; // there is error in opening the file
    writePtr = fileDescriptorTable[fd].writePtr;

    if (iNodeIdx exist in cache)
        INode = get INode in cache;
    else
        find corresponding cluster holding INode and loadINodeFromDisk(cluster);
    int original_File_Size_Before_Write = INode.filesize;

    while(byteAlreadyWritten < len){
        int block_Position_In_INode = ptr/BLOCK_SIZE;
        int block_idx;
        if(is_In_12_DirectBlocks(block_Position_In_INode)){
            // One of the 12 direct node
            block_idx = directBlocksInINode[block_Position_In_INode];
            if(getBlock(block_idx) is null){
                block_idx = allocate a new data block and link to inode else break;
                inodeHasChanged = true;
            }
        }else if (is_In_Single_Indirect_Region(block_Position_In_INode)){
            // writeptr is in single indir region
            if (single indirect index block has not existed yet){
                allocate a new index block and link to INode else break;
                indexBlock_idx = get index of the index block;
                inodeHasChanged = true;
            }
            loadIndexBoxIntoBuffer(indexBlock_idx, bufferForOneBlock);
            indexBlock_SingleIndir = (int[1024])bufferForOneBlock; // now this block buffer holds 1024 entry for block
            block_idx = indexBlock_SingleIndir[block_Position_In_INode -12];
            if block_idx points to null
                // Need to allocate the target block
                block_idx = allocate a new data block else break;
                indexBlock_SingleIndir[block_Position_In_INode -12] = block_idx;
        }
        else{
            // based on block_Position_In_INode, the writePtr is targeting at the double indirect pointer
            // 1. make sure single block has exist, if not, create a single indir index block first
            if(single indirect index block has not existed yet){
                allocate a new index block and link to INode else break;
                inodeHasChanged = true;
            }
            // after making sure there are no sparse space in the single indir region in INode
            // we start touching the double indir pointer
            if(double indirect index block has not existed yet){
                // if there is no double indir block yet, allocate one
                indexBlock_idx = allocate a index block fo double indir in available cluster else break;
                inodeHasChanged = true;
            }
            loadIndexBoxIntoBuffer(indexBlock_idx, bufferForOneBlock);
            indexBlock_DoubleIndir = (int[1024])bufferForOneBlock;
            // every entry of double indir index block should be a index block (as tutorial, B-->B, not A/C)
            // the corresponding double indir index is calculated through (block_Position_In_INode - 12 - 1024)/1024
            // (block_Position_In_INode - 12 - 1024) is current block index. One entry on double indir holds 1024 blocks.
            if (indexBlock_DoubleIndir[(block_Position_In_INode - 12 - 1024)/1024] is not an index block) break;

            // targeted_index_block is what the current entry of double Indir block pointing to
            targeted_index_block = indexBlock_DoubleIndir[(block_Position_In_INode - 12 - 1024)/1024];

            // check if the target single indir block has not existed yet
            else if( targeted_index_block is null){
                targeted_index_block = allocate an index block else break;
                indexBlock_DoubleIndir[(block_Position_In_INode - 12 - 1024)/1024] = targeted_index_block;
            }
            // The two lines below access one level down the index blocks and get the data block
            indexBlock_SingleIndir = getIndexBlock(targeted_index_block)
            block_idx = indexBlock_SingleIndir[(block_Position_In_INode - 12 - 1024)%1024];

            if(block_idx points to null){
                block_idx = allocate a data block for write else break;
                targeted_index_block[(block_Position_In_INode - 12 - 1024)%1024] = block_idx;
            }
        }
        bufferForOneBlock = getWriteTargetBlock(block_idx);
        // determine how many bytes to write in this iteration based on the leftover space in block
        int bytesToWrite = = min(BLOCK_SIZE - writePtr % BLOCK_SIZE , len-byteAlreadyWritten);
        // Keep writing from the input buf to the disk data block buffer
        memcpy(bufferForOneBlock + writePtr % BLOCK_SIZE, buf + byteAlreadyWritten,
bytesToWrite );
        byteAlreadyWritten += bytesToWrite;
        writeptr += bytesToWrite;
    }

    if(inodeHasChanged == true) updateINodeInCache();
    if updateClusterToDisk(original_File_Size_Before_Write, writeptr) == -1
        return -1; // Update to disk failed, the write is not good
    return byteAlreadyWritten;
}

//---------------close-----------------------------
int CloseFile(fd){
    if(IsFileOpen(fd)){
        openFileTable[fd] = OpenFD_unused;
        return 0;
    }
    return -1;
}

//-----------------Helper Method----------------------------------------------------------------------------------

// To improve performance, our implementation follows below principles syncing cache to disk:
// 1. we are using clustered INode system, thus everytime we always load complete INode table
from one cluster to make use of locality
// 2. We update INode in cache only when there is a change. The changed INode is synced
back to disk when it is evicted from cache
// 3. To improve performance, we keep all updating of index blocks at the end of writeFile()
using helper method updateClusterToDisk()
//    please check updateClusterToDisk() for detail.
int updateClusterToDisk(original_File_Size_Before_Write, writeptr){
    // Our design choose clustered INode system
    // Principle 1 is one file must have all its data blocks in the same cluster
    // Principle 2 is that all files are dense (sparse space will be filled with data blocks of all zero,
    // also empty entries on index blocks will be filled)

    // This method first checks how many extra blocks needed to fill sparse region due to write
    // then based on the extra space, it determines whether to allocate all those blocks in current
cluster
    // or to move the entire file with all data block and INode to a new cluster (since very unlikely,
but current cluster is nearly full)

    dense_Block_Position_In_INode = original_File_Size_Before_Write/BLOCK_SIZE;
    end_Block_Position_In_INode = writeptr/BLOCK_SIZE;

    numberOfAdditionBlockNeeded = end_Block_Position_In_INode - dense_Block_Position_In_INode;
    if (numberOfAdditionBlockNeeded * BLOCK_SIZE > CLUSTER_SIZE - currentCluster.size){
        // current cluster is full and can not be write directly back to disk
        newCluster = find clostest cluster with enough size and bring to cache else return -1;
        copy current file I-node and data blocks there;
        fillSparseSpaceAt(newCluster, writeptr, numberOfAdditionBlockNeeded, fileInode);
    }else{
        fillSparseSpaceAt(currentCluster, writeptr, numberOfAdditionBlockNeeded,fileInode);
    }
    syncDataBlocksInCacheToDisk();
    return 0;
}

void fillSparseSpaceAt(selectedCluster, writeptr, numberOfAdditionBlockNeeded){
    newIntermediateBlocks = allocate numberOfAdditionBlockNeeded of data blocks in selectedCluster;
    if(isIn_12Direct_Block(writeptr)){
        initialize newIntermediateBlocks and link to fileInode;
    }else if(is_In_Single_Indirect_Region(writeptr)){
        initialize newIntermediateBlocks and link to fileInode and single indir index block;
    }else{
        // in second indir region
        for every single indir index block linked by the double indie index block:
            initialize newIntermediateBlocks and link to fileInode and single indir index block;
    }
}

void loadINodeFromDisk(newlyEnteringCluster){
    // according to our memory structure, the cache has enough space to hold
    // over 600 cluster's complete INode tables
    // EveryTime a new cluster is called, its inodes will be brought to cache
    // The cache replace cluster's INode table using LRU principle
    if cache is full on cluster INode table entries:
        replacedOne = find the cluster which has not been called for longest time;
        Write INode content in replaceOne back to disk;
        Evict replaceOne;
        Load newlyEnteringCluster.INodeTable in cache;
    else:
        // Cache is not full, need not to replace
        Load newlyEnteringCluster.INodeTable in cache;
```

```
}
```