CIS657 Spring 2019

Assignment 3

Submitted by –

Name – Anish Nesarkar

SUID – 368122582

Subject – Operating Systems

Assignment 3

CIS657 Spring 2019

Assignment Disclosure Form

Assignment #: 3

Name: Anish Nesarkar

1. Did you consult with anyone other than instructor or TA/grader on parts of this assignment?

2. Did you consult an outside source such as an Internet forum or a book on parts of this assignment?

I assert that, to the best of my knowledge, the information on this sheet is true.

Signature:_____Anish Nesarkar_____ Date : 5/6/2019

Design and Implementation:

This Assignment consists of implementing Nachos basic file system which uses Nachos simulated disk. The Nachos file system uses the following files:

a. filesys.h and filesys.cc -> They are the top level interface to the file system.
b. directory.h and directory.cc -> They translate the file names to disk file headers. The directory data structure is stored as a file.
c. filehdr.h and filehdr.cc -> They manage the data structure representing the layout of a file's data on disk.
d. openfile.h and openfile.cc -> They translate the file reads and writes to disk sector read and writes.
e. synchdisk.h and synchdisk.cc -> They provide synchronous access to the asynchronous physical disk, so that thread block until their requests have completed.
f. disk.h and disk.cc -> They emulate a physical disk, by sending requests to read and write disk blocks to a UNIX file and then generating an interrupt after some period of time.

The task 1 consists of implementing basic nachos file system and performing the following sub-tasks –

a. Synchronization to allow multiple threads to use file system concurrently i.e. the same file can be read or written by multiple threads concurrently. Each thread will give its own private seek position to the file. Thus, two threads read sequentially through same file without interfering.
b. All the file system operations should be atomic and serializable. If one thread is in middle of a file write, a thread concurrently reading the file will see either all of change or none of it. If write operation finishes before the call to read is started, the read must reflect the modified version of file.
c. When a file is deleted, threads with file open will continue to read and write the file until file is closed. Deleting a file should prevent further opens on the file, but the disk blocks for the file cannot be reclaimed until file has been closed by all the threads.
d. The file system should allow the maximum size of file to be as large as the disk i.e. 128Kbytes.
e. The file should be dynamically extensible. The files should expand as and when some data is written to the file.

The following files were added to make the file extensible:

i.e. indirect and doubly direct blocks were added as shown.

blockfile.h

```
#include "copyright.h"
#ifndef FILE_BLOCK_H
#define FILE_BLOCK_H

#include "disk.h"
#include "pbitmap.h"

#define MAX_BLOCKS (int) (SectorSize  / sizeof(int))
#define EMPTY_BLOCK -1


class IndirectBlock {
public:
    IndirectBlock();

    int Allocate(PersistentBitmap *bitMap, int numSectors);// Initialize a
indirect block

    void Deallocate(PersistentBitmap *bitMap);          // De-allocate this
file's
                        //  data blocks

    void FetchFrom(int sectorNumber);   // Initialize file header from disk
    void WriteBack(int sectorNumber);   // Write modifications to file header
                    //  back to disk

    int ByteToSector(int offset);   // Convert a byte offset into the file
                    // to the disk sector containing
                    // the byte

    // int FileLength();            // Return the length of the file
                    // in bytes

    void Print();           // Print the contents of the file.

private:
    int dataSectors[MAX_BLOCKS];
};



class DoublyIndirectBlock {
public:
    DoublyIndirectBlock();
```

```
    int Allocate(PersistentBitmap *bitMap, int numSectors);// Initialize a
indirect block

    void Deallocate(PersistentBitmap *bitMap);          // De-allocate this
file's
                        //  data blocks

    void FetchFrom(int sectorNumber);   // Initialize file header from disk
    void WriteBack(int sectorNumber);   // Write modifications to file header
                    //  back to disk

    int ByteToSector(int offset);   // Convert a byte offset into the file
                    // to the disk sector containing
                    // the byte

    // int FileLength();            // Return the length of the file
                    // in bytes

    void Print();           // Print the contents of the file.

private:
    int dataSectors[MAX_BLOCKS];
};




#endif  // FILE_BLOCK_H
```

blockfile.cc

```
#include "copyright.h"


#include "filehdr.h"
#include "debug.h"
#include "synchdisk.h"
#include "main.h"
#include "blockfile.h"
#include <string>

IndirectBlock::IndirectBlock() {
    for(int i = 0; i < MAX_BLOCKS; ++i)
        dataSectors[i] = EMPTY_BLOCK;
```

```
}

int
IndirectBlock::Allocate(PersistentBitmap *freeMap, int numSectors) { //
Initialize a file header,
    DEBUG('e', "starting single indirect allocation\n");
    if(numSectors < 0)
        return -1;

    if(freeMap->NumClear() < numSectors)                    // failure if not
enough free sectors on disk
        return -1;

    DEBUG('e', "enough space for single indirect allocation\n");
    int allocated = 0;
    for(int i = 0; i < MAX_BLOCKS && allocated < numSectors; ++i) {     //
allocate space for all blocks
        if(dataSectors[i] != EMPTY_BLOCK)
            continue;
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] != EMPTY_BLOCK);
        ++allocated;
    }

    DEBUG('e', "single indirect allocated\n");
    return allocated;
}

void
IndirectBlock::Deallocate(PersistentBitmap *freeMap) {
    DEBUG('r', "beginning indirect block deallocation\n");
    for(int i = 0, sector; i < MAX_BLOCKS; ++i) {       // deallocate all sectors
        sector = dataSectors[i];
        if(sector == EMPTY_BLOCK)
            continue;
        ASSERT(freeMap->Test(sector));                  // assert that sector
to be cleared is in use
        freeMap->Clear(sector);
    }
    DEBUG('r', "finished indirect block deallocation\n");
}

void
IndirectBlock::WriteBack(int sector) {
    kernel->synchDisk->WriteSector(sector, (char *)this);
```

```
}

void
IndirectBlock::FetchFrom(int sector) {
    kernel->synchDisk->ReadSector(sector, (char *)this);
}

int
IndirectBlock::ByteToSector(int offset) {
    int vBlock = offset / SectorSize;
    ASSERT(vBlock < MAX_BLOCKS);                // assert that it is a valid
virtual block
    int pBlock = dataSectors[vBlock];
    ASSERT(pBlock >= 0 && pBlock < NumSectors);
    return pBlock;
}


//#########################################################################################
##############################//
DoublyIndirectBlock::DoublyIndirectBlock() {
    for(int i = 0; i < MAX_BLOCKS; ++i)
        dataSectors[i] = EMPTY_BLOCK;
}

int
DoublyIndirectBlock::Allocate(PersistentBitmap *freeMap, int numSectors) { //
Initialize a file header,
    IndirectBlock *iblock;

    DEBUG('e', "starting doublyindirect allocation\n");
    // printf("numSectors requested dblock allocation: %d\n", numSectors);
    if(numSectors < 0)
        return -1;
    if(freeMap->NumClear() < numSectors)                    // failure if not
enough free sectors on disk
        return -1;

    DEBUG('e', "enough space for doublyindirect allocation\n");
    int allocated = 0;
    for(int i = 0; i < MAX_BLOCKS && allocated < numSectors; ++i)   {   //
allocate space for all indirect blocks
        iblock = new(std::nothrow) IndirectBlock();
        if(dataSectors[i] == EMPTY_BLOCK)
```

```
            dataSectors[i] = freeMap->FindAndSet();                         //
allocate block for indirect block
        else
            iblock->FetchFrom(dataSectors[i]);
        ASSERT(dataSectors[i] != EMPTY_BLOCK);
        int result = iblock->Allocate(freeMap, numSectors - allocated);

        ASSERT(result >= 0);
        iblock->WriteBack(dataSectors[i]);                          // write
indirect block hdr back to disk
        allocated += result;
        delete iblock;
    }

    DEBUG('e', "doubly indirect block allocated\n");
    return allocated;
}

void
DoublyIndirectBlock::Deallocate(PersistentBitmap *freeMap) {
    DEBUG('r', "beginning doublyindirect deallocation\n");
    IndirectBlock *iblock;
    for(int i = 0, sector; i < MAX_BLOCKS; ++i) {       // deallocate all blocks
        sector = dataSectors[i];
        if(sector == EMPTY_BLOCK)                       // skip empty block
            continue;
        ASSERT(freeMap->Test(sector));                  // assert that the sector
we are deallocating is in use
        iblock = new(std::nothrow) IndirectBlock();
        iblock->FetchFrom(sector);                      // load up filehdr
        iblock->Deallocate(freeMap);                        // deallocate filehdr
        ASSERT(freeMap->Test(sector));                  // just to be sure
nothing weird happened
        freeMap->Clear(sector);
        delete iblock;
    }
    DEBUG('r', "finished doubly indirect deallocation\n");
}

void
DoublyIndirectBlock::WriteBack(int sector) {
    kernel->synchDisk->WriteSector(sector, (char *)this);
}

void
```

```
DoublyIndirectBlock::FetchFrom(int sector) {
    kernel->synchDisk->ReadSector(sector, (char *)this);
}

int
DoublyIndirectBlock::ByteToSector(int offset) {
    int vBlock = offset /
SectorSize;                                        // calc virtual block we
want
    IndirectBlock *iblock = new(std::nothrow) IndirectBlock();
    iblock->FetchFrom(dataSectors[vBlock /
MAX_BLOCKS]);                            // load up indirect block hdr that
contains the virtual block we want
    int pBlock = iblock->ByteToSector((vBlock % MAX_BLOCKS) *
SectorSize);      // find the corresponding physical block
    delete iblock;
    // printf("doublyindirect ByteToSector: %d\n", pBlock);
    ASSERT(pBlock >= 0 && pBlock < NumSectors);
    return pBlock;
}
```

openfile.cc

The write function was modified as shown below for extendable file requirement

```
OpenFile::Write(char *into, int numBytes)
{

if(seekPosition + numBytes > hdr->FileLength()) {           // need to expand
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
        freeMap->FetchFrom(kernel->fileSystem->freeMapFile);                  //
fetch freemap
        ASSERT(hdr->Allocate(freeMap, numBytes));               // expand file
        hdr->WriteBack(hdrSector);                              // write header
back
        freeMap->WriteBack(kernel->fileSystem->freeMapFile);                  //
write freemap back
        delete freeMap;
    }

   int result = WriteAt(into, numBytes, seekPosition);
   seekPosition += result;

   return result;
```

```
}
```

The ReadAt and WriteAt function was modified as below to have multiple threads access the file to read and write using lock:

```
int
OpenFile::ReadAt(char *into, int numBytes, int position)
{
    bool release = false;


    if(!fileName.empty()){
    if(!kernel->filelock[fileName]->IsHeldByCurrentThread()) {
        release = true;
        kernel->filelock[fileName]->Acquire();
    }
    }

    int fileLength = hdr->FileLength();
    int i, firstSector, lastSector, numSectors;
    char *buf;

    if ((numBytes <= 0) || (position >= fileLength)){

        return 0;               // check request
    }
    //cout << "position " << position << endl;
    if ((position + numBytes) > fileLength)
    numBytes = fileLength - position;
    DEBUG(dbgFile, "Reading " << numBytes << " bytes at " << position << " from
file of length " << fileLength);
    firstSector = divRoundDown(position, SectorSize);
    lastSector = divRoundDown(position + numBytes - 1, SectorSize);
    numSectors = 1 + lastSector - firstSector;

    // read in all the full and partial sectors that we need
    buf = new char[numSectors * SectorSize];
    for (i = firstSector; i <= lastSector; i++)
        kernel->synchDisk->ReadSector(hdr->ByteToSector(i * SectorSize),
                    &buf[(i - firstSector) * SectorSize]);

    // copy the part we want

    bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
```

```
    if(release)
        {
            kernel->filelock[fileName]->Release();
            }
    delete [] buf;
    return numBytes;
}

int
OpenFile::WriteAt(char *from, int numBytes, int position)
{

    if(!fileName.empty())
        kernel->filelock[fileName]->Acquire();

    int fileLength = hdr->FileLength();
    int i, firstSector, lastSector, numSectors;
    bool firstAligned, lastAligned;
    char *buf;

    if ((numBytes <= 0) || (position >= fileLength)){
    kernel->filelock[fileName]->Release();
    return 0;
    }
                // check request
    if ((position + numBytes) > fileLength)
    numBytes = fileLength - position;
    DEBUG(dbgFile, "Writing " << numBytes << " bytes at " << position << " from
file of length " << fileLength);

    firstSector = divRoundDown(position, SectorSize);
    lastSector = divRoundDown(position + numBytes - 1, SectorSize);
    numSectors = 1 + lastSector - firstSector;

    buf = new char[numSectors * SectorSize];

    firstAligned = (position == (firstSector * SectorSize));
    lastAligned = ((position + numBytes) == ((lastSector + 1) * SectorSize));

// read in first and last sector, if they are to be partially modified
    if (!firstAligned)
    {

        ReadAt(buf, SectorSize, firstSector * SectorSize);
```

```
    }
    if (!lastAligned && ((firstSector != lastSector) || firstAligned))
     {
         ReadAt(&buf[(lastSector - firstSector) * SectorSize],
                SectorSize, lastSector * SectorSize);
                    }

// copy in the bytes we want to change
    bcopy(from, &buf[position - (firstSector * SectorSize)], numBytes);

// write modified sectors back
    for (i = firstSector; i <= lastSector; i++)
        kernel->synchDisk->WriteSector(hdr->ByteToSector(i * SectorSize),
                    &buf[(i - firstSector) * SectorSize]);

    if(!fileName.empty())
        kernel->filelock[fileName]->Release();

    delete [] buf;
    return numBytes;
}
```

Filesys.cc

The open function was modified as below. The file is not allowed to open again if it was called by some thread to delete.

```
OpenFile *
FileSystem::Open(char *name)
{
    ListIterator<string> *deleteList = new ListIterator<string>(kernel-
>DeleteFilesList);
    while(!deleteList->IsDone())
    {
        string file = deleteList->Item();
        if(file == name)
        {
            cout << "File cannot be opened as it is being deleted by other
thread" << endl;
            return NULL;
        }
        deleteList->Next();
    }
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
```

```
    int sector;
    kernel->OpenFileMap[name]++;
    DEBUG(dbgFile, "Opening file" << name);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
    openFile = new OpenFile(sector);    // name was found in directory
    delete directory;
    return openFile;                // return NULL if not found
}
```

The remove function was modified to prevent the deletion of file if it was opened by some other thread.

```
bool
FileSystem::Remove(char *name)
{
    if(!kernel->DeleteFilesList->IsInList(name))
        kernel->DeleteFilesList->Append(name);

    while(kernel->OpenFileMap[name] != 0)
    {
        cout << "File : >--< " << name << " >--< not closed by other threads" <<
endl;
        return false;
    }
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *fileHdr;
    int sector;

    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector == -1) {
       delete directory;
       return FALSE;             // file not found
    }

    fileHdr = new FileHeader;

    fileHdr->FetchFrom(sector);

    freeMap = new PersistentBitmap(freeMapFile,NumSectors);
```

```
    fileHdr->Deallocate(freeMap);       // remove data blocks
    freeMap->Clear(sector);         // remove header block

    directory->Remove(name);
    freeMap->WriteBack(freeMapFile);        // flush to disk
    directory->WriteBack(directoryFile);        // flush to disk



    cout << "Deleting File after being closed by all threads : " << name << endl;
    delete fileHdr;
    delete directory;
    delete freeMap;
    return TRUE;
}
```

Filehdr.cc

The Allocate and Deallocate functions were modified for extending the files using indirect blocks as shown below:

```
bool
FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    numBytes = fileSize;
    int requirednumSectors  = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < requirednumSectors)
    return FALSE;       // not enough space

    DoublyIndirectBlock *dblock;
    int allocated = 0;
    for (int i = 0; i < NumDirect && allocated < requirednumSectors; ++i) {
        dblock = new DoublyIndirectBlock();
        if(dataSectors[i] == EMPTY_BLOCK)
            dataSectors[i] = freeMap->FindAndSet();
        else
        {
            dblock->FetchFrom(dataSectors[i]);
        }
    // since we checked that there was enough free space,
    // we expect this to succeed
    ASSERT(dataSectors[i] != EMPTY_BLOCK);
    int output = dblock->Allocate(freeMap, requirednumSectors - allocated);
    ASSERT(output != -1);
```

```
    dblock->WriteBack(dataSectors[i]);                          // write doubly
indirect block back
    allocated += output;                                       // decrease
remaining sectors to be allocated
    delete dblock;
    }
    ASSERT(requirednumSectors - allocated <= 0);
    numBytes += fileSize;
    numSectors  += divRoundUp(fileSize, SectorSize);
    DEBUG('e', "file header allocated\n");
    return TRUE;
}

//----------------------------------------------------------------------
// FileHeader::Deallocate
//  De-allocate all the space allocated for data blocks for this file.
//
//  "freeMap" is the bit map of free disk sectors
//----------------------------------------------------------------------

void
FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    DoublyIndirectBlock *dblock;
    for (int i = 0,sector; i < NumDirect; ++i) {

        sector = dataSectors[i];
        if(sector == EMPTY_BLOCK)
            continue;

        ASSERT(freeMap->Test(sector));
        dblock = new DoublyIndirectBlock();
        dblock->FetchFrom(sector);
        dblock->Deallocate(freeMap);
        ASSERT(freeMap->Test(sector));
    freeMap->Clear(sector);
    delete dblock;
    }
}
```

Threadtest.cc

The test cases are created for the tasks:

a. Multiple threads reading the file

b. Extendable files
c. Largest file size
d. Deletion of the file during multiple thread access

```
#include "kernel.h"
#include "main.h"
#include "thread.h"
#include "filehdr.h"
#include "openfile.h"
#include "filesys.h"

//creating threads for testing
Thread *t2 = new Thread("forked thread");
Thread *t3 = new Thread("forked thread");
Thread *t4 = new Thread("forked thread");

void
SimpleThread(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", which, num);
        kernel->currentThread->Yield();
    }
}

//fileoperation 1 and 2 for reading multiple files
void FileOperation1(int i)
{
   OpenFile *file1;
   //open the file 1
    file1 = kernel->fileSystem->Open("file1");
    //buffer to store the content to tfile 1
    char buffer[27]("Additional write in file 1");
    if(file1 != NULL)
    {
        //seek the location of the file to be written to
    file1->Seek(25);
    //write to the file
    file1->Write(buffer, 27);
    cout << "Next Wrote in file 1 : " << buffer << endl;
//close the file after writing
```

```
kernel->fileSystem->CloseFile("file1");
    }
    //clear the buffer
    memset(buffer,0,27);
    OpenFile *file;
    //open the file for reading
    file = kernel->fileSystem->Open("file1");
    if(file != NULL)
    {
     file->Read(buffer,10);
     cout << "Thread 1 Read from file 1 : " << buffer << endl;
    //close the file
     kernel->fileSystem->CloseFile("file1");
    }
    kernel->scheduler->ReadyToRun(t2);

 }

void FileOperation2(int i)
{
    kernel->currentThread->Sleep(FALSE);
    char buffer2[50]; //buffer to read into
    memset(buffer2,0,50);
    OpenFile *file;
    file = kernel->fileSystem->Open("file1"); //open the file
    if(file != NULL)
    {
     file->Read(buffer2,15); //read from file 1 concurrently
     cout << "Thread 2 Read from file 1 :" << buffer2 << endl;
     kernel->fileSystem->CloseFile("file1");
    cout << endl;
    cout << "Thread 2 deleting file 1" << endl;
    kernel->fileSystem->Remove("file1"); //delete the file
    cout << endl;
    }
    cout << ">------------------< Test case done for Multiple Read/Write and file
extendable >-----------------<" << endl << endl;
    kernel->scheduler->ReadyToRun(t3); //run the next test case
    kernel->scheduler->ReadyToRun(t4);

}

void FileOperation3(int i)
{
    kernel->currentThread->Sleep(FALSE);
```

```
    cout << endl;
    cout << ">-------------------< Test for Deleting file even when other thread
is using the file >--------------< " << endl << endl;

    OpenFile *file1;

    file1 = kernel->fileSystem->Open("file2"); //open file 2

    char buffer[27]("Data in file 2"); //store the data from buffer to file 2

    if(file1 != NULL)
    {


    file1->Write(buffer, 27); //write data to file 2
    cout << "Wrote in file 2 : " << buffer << endl;

kernel->fileSystem->CloseFile("file2"); //close the file

    }
    memset(buffer,0,27); //reset the buffer
    OpenFile *file;
    file = kernel->fileSystem->Open("file2"); //open the file 2 for reading
    if(file != NULL)
    {
     file->Read(buffer,8); // read from the file into buffer
     cout << "Thread 3 Read from file 2 : " << buffer << endl;

     kernel->fileSystem->CloseFile("file2");
     cout << "Thread 3 deleting file 2..." << endl;
     kernel->fileSystem->Remove("file2"); //delete the file
    }

}


void FileOperation4(int i)
{
    kernel->currentThread->Sleep(FALSE);

    char buffer2[50];
    memset(buffer2,0,50);
    OpenFile *file;
    file = kernel->fileSystem->Open("file2"); //open file 2
```

```
    if(file != NULL)
    {
     file->Read(buffer2,15); //read from the buffer
     cout << "Thread 4 Read from file 2 :" << buffer2 << endl;
     kernel->fileSystem->CloseFile("file2"); //close the file
    cout << endl;
    cout << "Thread 4 deleting file 2..." << endl;
    kernel->fileSystem->Remove("file2"); //delete the file
    }
    cout << ">------------------< Test case done for Delete if the file is not
closed by other threads >-----------------<" << endl << endl;


}


void
ThreadTest()
{
    cout << ">==============================< Assignment 3
>============================<" << endl << endl;

OpenFile *file1;
    file1 = kernel->fileSystem->Open("file1");
    char buffer[25]("This is a Data of File 1");
    file1->Write(buffer, 22);
    cout << "Wrote in file 1 : " << buffer << endl;
    kernel->fileSystem->CloseFile("file1");
    Thread *t1 = new Thread("forked thread");
    t1->Fork((VoidFunctionPtr) FileOperation1, (void *) 1);

    t2->Fork((VoidFunctionPtr) FileOperation2, (void *) 1);

     t3->Fork((VoidFunctionPtr) FileOperation3, (void *) 1);

     t4->Fork((VoidFunctionPtr) FileOperation4, (void *) 1);

}
```

Log File System:

The log file system can be implemented by follows:

- The files are written sequentially to the sectors as FFS.
- When the file is to be written second time, All the contents of the file previously stored in the sectors are shifted to the next available sector.
- The previous sectors are marked invalid.
- The Inode Map for the file header is updated.
- The additional data to be written is then appended to the file.
- A cleaning process is done at some random time to shift the invalid sectors towards the end of the disk.

Testing

How to run the Test

1. Copy the nachos folder from your local machine to the server.
2. Copy the modified folder filesys from the local machine to the server
3. Navigate to the directory nachos/code/build.linux
4. Execute the below commands
    a. make clean
    b. make depend
    c. make nachos
5. To run the testcase. Execute command ./nachos -K -f

Added Files:

blockfile.h and blockfilecc to the location /nachos/code/filesys/

Modified Files:

filesys.cc in the location /nachos/code/filesys/

openfie.cc in the location /nachos/code/filesys/

filehdr.cc in the location /nachos/code/filesys/

filesys.h in the location /nachos/code/filesys/

openfie.h in the location /nachos/code/filesys/

filehdr.h in the location /nachos/code/filesys/

MakeFile

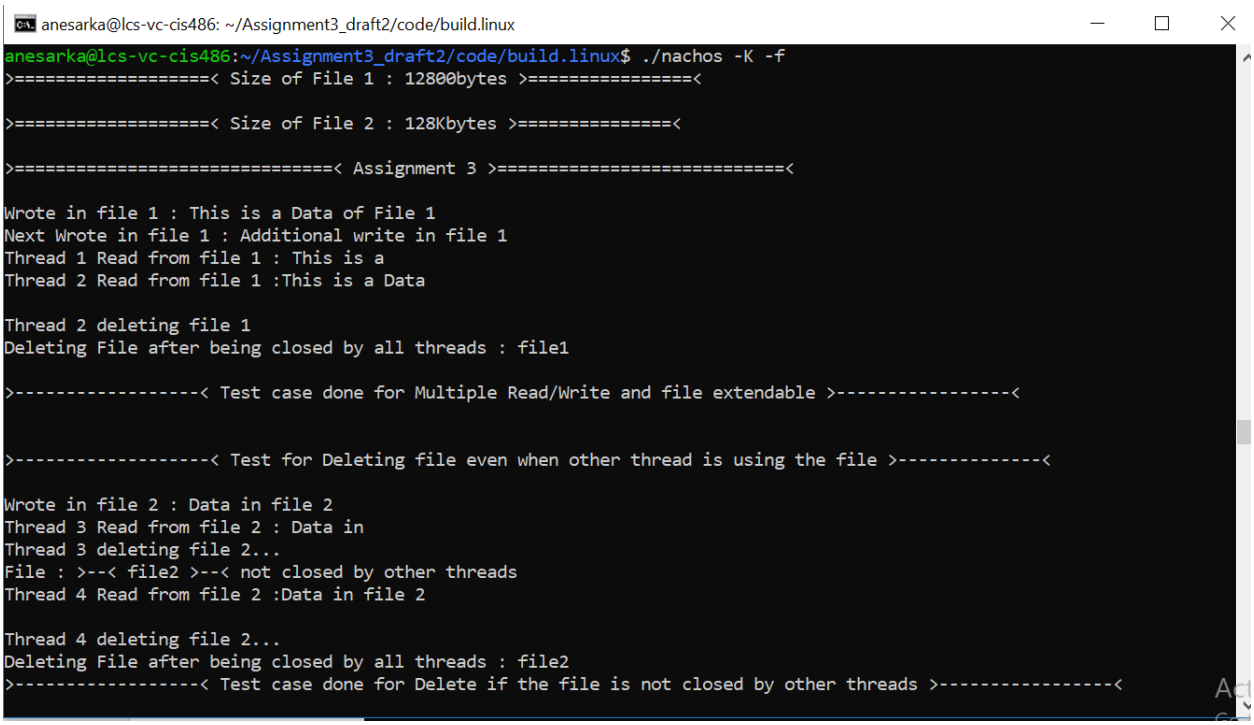
```
DEFINES = -DRDATA -DSIM_FIX
```

```
FILESYS_H =../filesys/directory.h \
        ../filesys/filehdr.h\
        ../filesys/filesys.h \
        ../filesys/openfile.h\
        ../filesys/pbitmap.h\
        ../filesys/synchdisk.h\
        ../filesys/blockfile.h

FILESYS_C =../filesys/directory.cc\
        ../filesys/filehdr.cc\
        ../filesys/filesys.cc\
        ../filesys/pbitmap.cc\
        ../filesys/openfile.cc\
        ../filesys/synchdisk.cc\
        ../filesys/blockfile.cc

FILESYS_O =directory.o filehdr.o filesys.o pbitmap.o openfile.o synchdisk.o
blockfile.o
```

OUTPUT



- **Multiple Threads reading the file**
- **Maximum Size of the file**
- **Not deleting a file if it is still opened by other thread**
- **Extending the file size beyond current size.**