Marks : 14 CSCI131
Spring Session 2015
Due date: September 18th

Assignment 3
Linux OS interface and OS responsibilities
7 marks

Complete the associated exercises before attempting the assignment

Aim

This assignment and its associated exercises introduce the services offered by an OS and, through a simple simulation, provide some familiarity with how an OS supports a "file system" on disk.

Objectives

On completion of this assignment and its associated exercise, you will be able to:

- Explain how an OS provides functionality through "supervisor calls";
- Correctly use some of the basic Linux supervisor calls for process management and byte-level I/O;
- Explain how the Linux file hierarchy can be explored and data on files retrieved;
- Explain how a simple "file system" can be implemented;
- Explain how the Linux i-node structures allow for more sophisticated approaches to the mapping of a file to disk blocks;
- Link code that you have written with supplied libraries.

Task

You are to implement a simple file management system and make it work with a supplied driver function that will create and delete files using your simulated system.

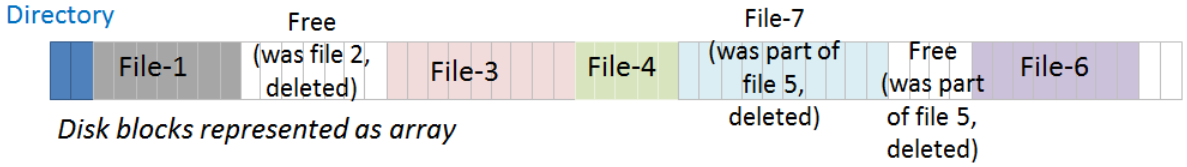
A simple file management system

A "file management system" is a major component in any operating system. Modern operating systems typically support several different file system structures; you may see references to "FAT32", "NTFS", "HFS", "ISO9660", Linux ext3, or Linux ext4. All file systems have to maintain meta-data about files and free-space, and must arrange for the mapping of files to disk blocks.

Your simulation program will be modelling a simple "*contiguous file allocation*" scheme. This was the kind of file system that was available on the micro-computer operating systems of the late 1970s and early 1980s. These used small exchangeable disks – the original IBM-PC and Apple computers had 8-inch "floppy disks" with only ~80kbytes (representing maybe

160 disk blocks); by 1984, the Macintosh had 2.5-inch 400Kbyte disks (maybe 400 disk blocks).

- Simple approach (as in early OSs and 1980s micro-computer systems) – ***contiguous allocation***



Disk blocks represented as array

  - Disadvantages
    - Need to know file size in advance (ask for number of blocks when create file)
    - Leads to fragmentation of free space on disk
      - Have to regularly run a program that moves files so as to coalesce free blocks
  - Advantages
    - Fast when reading all file sequentially
    - Minimal information required to define file-blocks (1st block, and number of blocks)

These were "flat file systems". There was no file hierarchy; instead all files were identified in a single directory. This directory occupied specific blocks at the start of the disk.

The directory structure would have had two parts.

The first part would be something like an array of structs:

```
struct direntry {
    char filename[NAMESIZE];
    int start;
    int size;
    …
};

typedef struct direntry Direntry;

static Direntry directory[MAXFILES];
```

The essential fields in a "directory entry" structure are a name (with a fixed maximum length "NAMESIZE", which would have been limited to 6 or 8 characters in those days), the start block number, and the number of blocks allocated to the file. Usually, there were additional data fields in entries such as a creation date-time field and often a "file type" (which would in some way identify the application used to create and manipulate the file). Along with this "array of structs", there would be a count for the number of entries in use.

Such a directory structure has many limitations. All files must have to have unique names. There is a maximum limit on the number of files that you can have on disk.

Another part of the directory structure at the start of the disk would have been a "block map".

This "block map" would have been a bit-map – one bit for each block on the disk. A zero value would indicate that the block was free; a one value would indicate that the block was allocated to a file.

A new disk would be formatted with an empty directory structure on the first disk blocks, and the bit-map entries set to indicate that all the remaining disk blocks were free.

The system would provide a "createFile" function that would take as arguments a file-name and requested size. This function would fail if there were already a file with that name. If the name was unique, the system would try to find a contiguous set of blocks to satisfy the request.

Finding free blocks is a matter of searching through the bit-map. Several approaches were used in the different systems, such as:

- "**First Fit**"
  Start at the beginning (starting after the fixed directory blocks at the first block that can be allocated to a file) and searching in the bit-map for a sequence of free blocks greater or equal in size to the requested size.
- "**Best Fit**"
  Search the entire bit-map for the sequence of free blocks whose length equals the request or exceeds the request by the smallest amount (can stop searching if find a sequence of free blocks that exactly matches the request size).
- "**Cyclic**"
  (This needs an extra field in the fixed directory structure that stores the block number where the last file was allocated.)
  The search continues from the last allocation spot searching towards the end of disk, then if necessary resumes at the start of the disk until a sufficient sized sequence of free blocks (greater than or equal to request) is found. (The search terminates unsuccessfully if it gets back to its starting point.)

The createFile request would fail if there was no adequate set of contiguous free blocks.

If the space was available, a new entry would be made in the directory recording the file-name, start block, and size; the bit-map would be updated to indicate that all the blocks were in use.

The operating system would maintain some hidden structure associated with the program that had opened the file. Subsequent readBlock and writeBlock requests would work via this OS structure. The OS would verify that they were valid (not attempting to use a non-existent file-block) and would work out the actual disk block that was to be used for the transfer. (The integer file-descriptor used on Linux is an index into a table of such file access structures.)

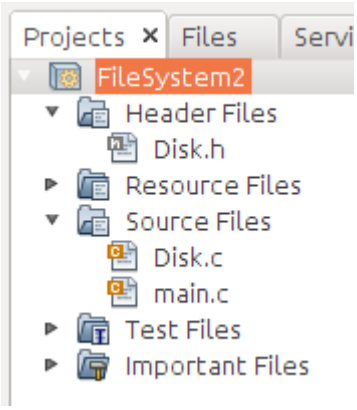
The corresponding "deleteFile" request would update the bit-map marking the blocks as again free and remove the directory entry. Typically, the directory would be re-written with all entries that had been after the file that was deleted being moved up one slot.

A problem with this simple allocation approach is that it leads to fragmentation of the free space. There may be many free blocks on disk, more than enough for the next file request, but these blocks are all separated by blocks that are allocated to other files.

Such file systems had to provide compaction operations. Compaction would be invoked at user command. The compaction process would shift files on disk, copying the data blocks and updating the directory entries as it worked. When the compaction process was completed, all the free space would be in one contiguous region.

Your simulation - overview

Your simulation is to be built as a NetBean's C application project.



The main() function will end up being quite simple:

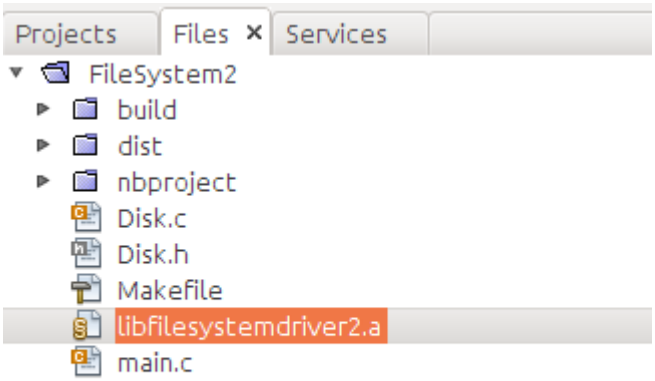
```
#include <stdio.h>
#include <stdlib.h>

extern void runSimpleSimulation();
extern void runSomeErrorTests();
// Argument k for runSimulation is a seed for random number generator
// Use 0 to get random seed, or value greater than zero for fixed seed
extern void runSimulation(int k);

int main(int argc, char** argv) {
    //runSimpleSimulation();
    //runSomeErrorTests();
    runSimulation(1);
    sleep(1);
    return (EXIT_SUCCESS);
}
```

Why "sleep(1)"?
Sometimes, when a C program terminates NetBeans will close its connection to the process before all of the output text has been flushed from buffers and displayed in NetBeans' output pane.

(You don't need to start with this main, you can start with code that directly invokes the functions that you are developing for your file management component "Disk.c").

It is intended that once your "Disk.c" code works that you use driver function that are in a supplied ".a" archive library. The functions in this library will make sequences of create, read, write, delete, compact requests to your file system implementation.

The file libfilesystemdriver2.a can be downloaded from the web site; it is compiled for the Ubuntu workstations in the laboratory.



There are three functions in the driver that you can invoke:

- "**Run Simple Simulation**"
  Makes a small number of create and delete requests; then invokes functions (that you supply) that show the contents of the directory and give some details of how files are mapped to disk blocks.
- "**Run Some Error Tests**"
  Makes many requests to your implementation – including invalid requests like "create a file with a duplicate name", "delete a file that was never created", "write data to a non-existent block of file". It will also do things like force compaction of files and then check that the data blocks have been correctly copied.
- "**Run Simulation**"
  A thorough test with a couple of thousand requests to your file system.

The following outputs are from my version. Your outputs should be similar but you can invent your own alternative approach to summarising the contents of the file directory and showing the mapping of files to disk blocks.

I chose to output the directory as a tabular listing with file-name, start block, and size. I chose to have an extra field in my directory entries – an identifier. My display of block allocations has a "." for a free block, and the identifier letter of a file if the block is part of a file.

(An identifier is just one of the letters [A-Za-z] allocated sequentially. My identifier was not unique – once I got to z, I started over at A again. But this is rarely going to cause any confusion.)

Following the display of file mapping to disk blocks, there are some summary statistics. You simply need to maintain counts of create and delete requests, the number of compactions, the number of files currently in the directory, and the total number of free blocks on disk.

Run Simple Simulation

```
Create 5 files
Loop 10 times, randomly pick file for deletion, then add new file
Invoke display functions
Directory listing:
       Filename                              Start      Size   (symbol)
    1                        hello5.txt          5         6       b
    2                            hlp2.c         11         5       l
    3                           prog1.o         21        18       n
    4                        report4.jpg          0         4       g
    5                          test3.pdf         39        16       o
gggg.bbbbbbllllll.....nnnnnnnnnnnnnnnnnnoooooooooooooooo.........
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
Number of file create operations        : 15
Number of file delete operations        : 10
Number of file compaction operations    : 0

Current number of directory entries     : 5
Current number of disk blocks allocated : 49
Current number of disk blocks free      : 463
```

Run Some Error Tests

```
Start by adding 25 files
A few more randomly selected add and delete requests
Start round of tests for common implementation errors
Confirmed correct handling of duplicate file names
Confirmed correct handling of invalid file sizes
Confirmed attempted writes to invalid blocks detected
Confirmed attempted reads from invalid blocks detected
Confirmed directory full condition handled
Confirmed that file compaction correctly clears up space
Error tests completed
Directory listing:
       Filename                              Start      Size   (symbol)
    1                          data4.txt       131         5       p
    2                          data5.jpg       105         1       X
    3                          data6.gif       190         1       x
    4                        hello10.exe       193         1       A
    5                         hello5.jpg        55         9       h
    6                         hello7.pdf       194        11       v
```

Run Simulation

```
   31                        test185.jpg        123         5       X
   32                          test187.o        160        11       q
   33                        test188.jpg        436        15       y
   34                         tmp173.txt         55         7       S
   35                           tmp186.o        198        12       Z
   36                         tmp187.pdf        485        18       z
   37                         var206.exe         51         4       X
   38                         var210.pdf         26         3       D
   39                         var211.txt        256        10       E
   40                           x188.jpg        148         6       v
xxxxKKKKKhhhhhooooowwwwwwwwwDDD..UUUUUDDDDDDD........XXXXSSSSSSSccc
ccc.JJJJJJJrrrrFFFFFFFFFF..OOOOOOOOOGGGG.aaaaaVVVVV...OOOOOXXXXX
.xxxxxxxxxxxxxxxxxxxvvvvvv......qqqqqqqqqq.....mmmmmmmmmmmmmmhh
hhhhh.ZZZZZZZZZZZ.......pppppppppppCCCCCCCC..bbbbbbbbb.tttttttt
EEEEEEEEEE.............................wwwwwwwwww..nnnnnnnnnnnnn
n.............................................sssssssssssssssssss
ss....QQQQQQQQQtttttttttttttttttttttuuuuuuuuuuuuuuuyyyyyyyyyyyy
yyy.................................zzzzzzzzzzzzzzzzz.........
Number of file create operations        : 2477
Number of file delete operations        : 2437
Number of file compaction operations    : 77

Current number of directory entries     : 40
Current number of disk blocks allocated : 353
Current number of disk blocks free      : 159

RUN FINISHED; exit value 0; real time: 1s; user: 0ms; system: 30ms
```

Details, details, ...

Your file system must support the following functions (you must use this Disk.h header file exactly as is given, the driver file has been compiled using these function definitions):

```
#ifndef DISK_H
#define DISK_H

#define MAXFILES 64
#define DISKBLOCKS 512

enum filesyserrors { NO_ERR, DIRECTORY_FULL, CREATE_FAIL,
NON_EXISTENT_FILE, DUPLICATE_NAME, INVALID_BLOCK,
ZERO_SIZE };
typedef enum filesyserrors FileSysErrors;

void initialiseFileSystem();
```

```
// createFile returns DIRECTORY_FULL if maxfiles used
// Returns DUPLICATE_NAME if there is already a file with the name
// Returns ZERO_SIZE if size <= 0
// Returns CREATE_FAIL if unable to find space for the file
FileSysErrors createFile(const char* filename, int size);

// deleteFile returns NON_EXISTENT_FILE if filename is invalid
FileSysErrors deleteFile(const char* filename);

// readBlock and writeBlock will return NON_EXISTENT_FILE if
// filename is invalid, and INVALID_BLOCK if requested block
// is invalid (<0, or >= size of file)

// blocks would have 512bytes but for simulation content of block represented by
// single integer value
FileSysErrors writeBlock(const char* filename, int block, int value);

// Read block stores value read in integer whose address is passed as
// the third argument
FileSysErrors readBlock(const char* filename, int block, int* vp);

// compactFiles returns number of free blocks - should be all in one large group
// at end of disk after compaction
int compactFiles();

// Display disk – show directory contents and provide some form of map
// identifying the mapping of files to disk blocks
void displayDisk();

// show history
// number of files created, deleted, number of entries in directory,
// number of blocks still free, number of compactions performed
void showHistory();

#endif  /* DISK_H */
```

This file system uses a small disk! It has a maximum of 64 entries in the directory (MAXSIZE). There are 512 blocks available for allocation to user files (DISKBLOCKS). (The directory doesn't occupy any space in this simulated version.)

The simulation driver does write to blocks, and then read data back to check that things work. The compaction process will move everything around but subsequent reads should still get back the same data as was written before the compaction.

In reality, blocks would have 512bytes (or more). But for the purposes of the simulation, you need use only an array of integers –

```
static int diskblockcontent[DISKBLOCKS];
```

When you write to a block of file, a particular element is set in diskblockcontent[]. When you read, you get back the value from diskblockcontent[].

The simulation is handled by the functions (functions returning FileSysErrors will return NO_ERR if things worked):

void **initialiseFileSystem**();
You initialise all the data structures that you will be using. (They may as well all be static arrays).

FileSysErrors **createFile**(const char* filename, int size);
All the file operations will use the file name. (Use #define NAMESIZE 32; names should all be shorter than 32 characters).

FileSysErrors **deleteFile**(const char* filename);
Removes the named file and tidies up.

FileSysErrors **writeBlock**(const char* filename, int block, int value);
Using data from the directory for this file, work out the actual disk block corresponding to 'block' of file – zero based array from file start. Write value into diskblockcontent for that block.

FileSysErrors **readBlock**(const char* filename, int block, int* vp);
Again, work out which entry in diskblockcontent is to be used, and copy the value from there into the integer variable whose address is passed in vp.

int **compactFiles**();
Reorganise the contents of your disk and update your directory.

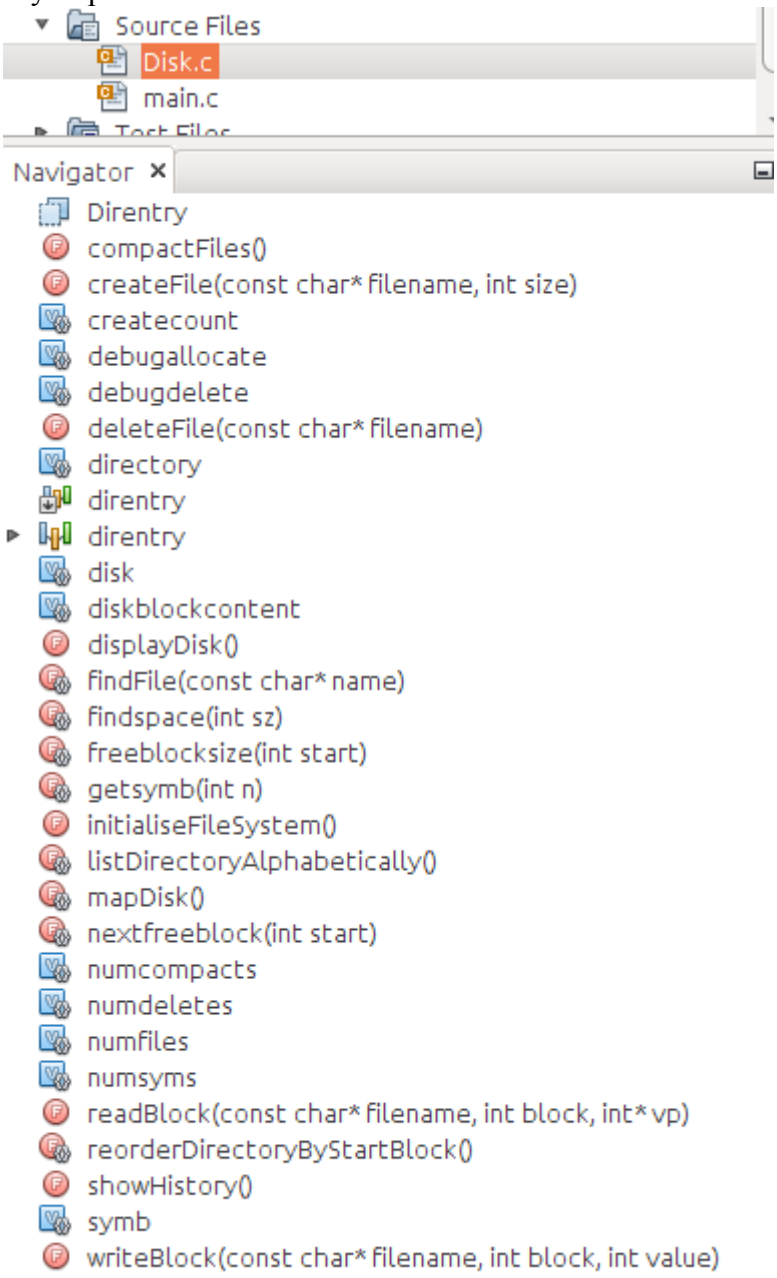
The display functions that you must also implement are:
void **displayDisk**();
Display the contents of your directory and provide some visual display of how files are mapped to blocks on disk.

void **showHistory**();
Provide summary of calls to your simulated system.

Your implementation

You must define appropriate data structures and implement the functions as shown above. You will need several private auxiliary functions to handle details.

My implementation was:



The entries are the functions listed above – compactFiles, createFile, deleteFile, displayDisk, initialiseFileSystem, readBlock, showHistory, writeBlock.

The entries correspond to auxiliary functions that I added when implementing the main functions. For example, my displayDisk invokes listDirectoryAlphabetically to list the directory entries in alphabetic order.

You do not need to implement the bit-map as a bit map! For this simulation, it is acceptable to use an integer array with 0/1 values to indicate whether a block is in use. (Using an actual

bit-map just entails some fussy code to convert a block number into a byte identifier and a bit identifier for an array of unsigned bytes.)

The hardest functions are likely to be createFile and compactFiles.

My createFile function utilized some helper functions:

```
static int nextfreeblock(int start) {

static int freeblocksize(int start) {

static int findspace(int sz) {
```

My findspace function returns –ve if it cannot find a set of contiguous free blocks of sufficient size; or returns the start block number. It uses nextfreeblock to find the next individual block that is free, and freeblocksize to determine how many blocks were available at a given point.

The compactFiles function – just careful house-keeping in some while-loops. An example:

```
Directory listing:
       Filename                                Start       Size    (symbol)
    1                              data4.c         44          4       G
    2                             data5.gif        80          7       I
    3                            hello6.txt        63          6       u
    4                            hello7.jpg        24          4       N
    5                             log11.gif       108         22       M
    6                              log9.exe        28          8       p
    7                              note1.o         56          4       B
    8                             prog1.exe        49          7       h
    9                             prog2.pdf        91          4       k
   10                           report6.jpg         0          4       K
   11                             test2.jpg        11          7       c
   12                             test3.txt        39          5       f
   13                              tmp1.txt         5          5       x
   14                              tmp3.gif        95         13       H
   15                              var4.pdf        18          6       J
KKKK.xxxxx.cccccccJJJJJJNNNNpppppppp...fffffGGGG.hhhhhhhBBBB...u
uuuuu..........IIIIIII....kkkkHHHHHHHHHHHHHHMMMMMMMMMMMMMMMMMMMM
MM............................................................
............................................................
```

1) Move tmp1.txt (x) one block

```
KKKKxxxxx..cccccccJJJJJJNNNNpppppppp...fffffGGGG.hhhhhhhBBBB...u
uuuuu..........IIIIIII....kkkkHHHHHHHHHHHHHHMMMMMMMMMMMMMMMMMMMM
MM............................................................
```

2) Move test2.jpg (c) two blocks

```
KKKKxxxxxccccccc..JJJJJJNNNNpppppppp...fffffGGGG.hhhhhhhBBBB...u
uuuuu..........IIIIIII....kkkkHHHHHHHHHHHHHHMMMMMMMMMMMMMMMMMMMM
MM............................................................
```

3) Move var4.pdf (J) two blocks
4) Move hello7.jpg (N) two blocks

5) Move log9.exe (p) two blocks
6) Move test3.txt (f) 5 blocks

```
KKKKxxxxxccccccccJJJJJJNNNNpppppppppfffff.....GGGG.hhhhhhhBBBB...u
uuuuu..........IIIIIII....kkkkHHHHHHHHHHHHHHMMMMMMMMMMMMMMMMMMMM
MM.....................................................................
```

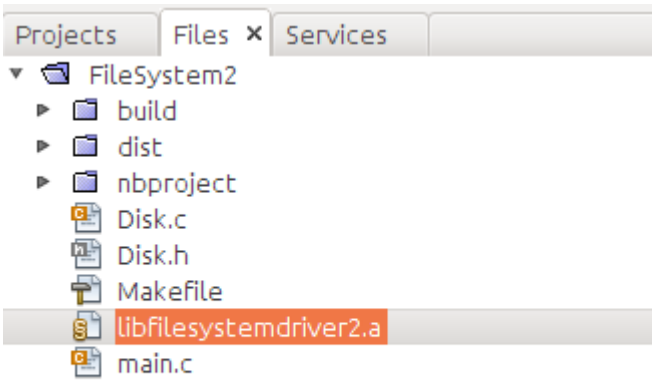
The start block field in the directory entry for each file is updated as the process continues; so for this partially completed compaction process one would get:

```
Directory listing:
       Filename                              Start       Size    (symbol)
    1                                data4.c      44          4       G
    2                               data5.gif     80          7       I
    3                              hello6.txt     63          6       u
    4                              hello7.jpg     22          4       N
    5                               log11.gif    108         22       M
    6                                log9.exe     26          8       p
    7                                note1.o      56          4       B
    8                               prog1.exe     49          7       h
    9                               prog2.pdf     91          4       k
   10                             report6.jpg      0          4       K
   11                               test2.jpg      9          7       c
   12                               test3.txt     34          5       f
   13                                tmp1.txt      4          5       x
   14                                tmp3.gif     95         13       H
   15                                var4.pdf     16          6       J
```

You will probably find it useful to have your compactFiles function start by creating a "list" of current directory entries ordered by start block.

Building your program

You must copy the supplied library archive file into the directory for your NetBeans project.

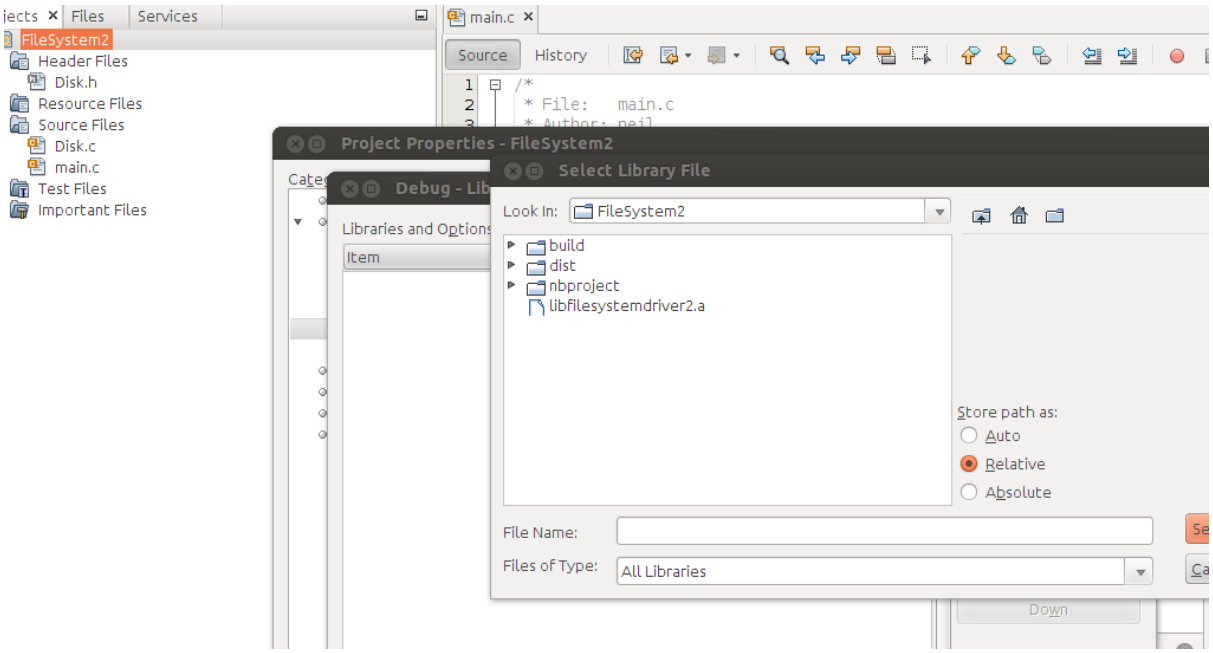


You must edit the project's "properties". You need to specify additional libraries for the linking step.

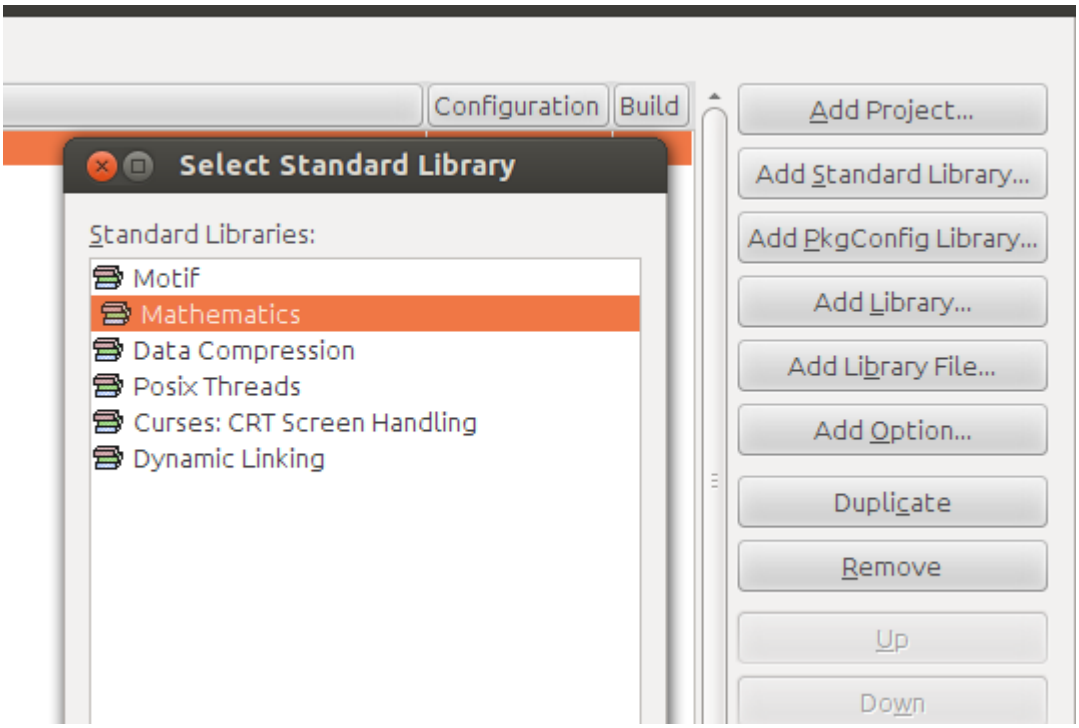
Right-click on the project and select properties.

In the dialog select linker.

Select “Add Library File”. Find the library.



You will also need the maths library. The dialog used to add libraries has a special option for standard libraries:



The program should build. The linker report should show linkage first with the .a library and then with “-lm” – the maths library.

```
gcc     -o dist/Debug/GNU-Linux-x86/filesystem2 build/Debug/GNU-Linux-x86/_ext/1330780743/Disk.o build/Debug/GNU-Linux-x86/main.o libfilesystemdriver2.a -lm
```

Your implementation strategy

1. Define the structs needed to represent your directory
2. Write some code that to place entries in the directory (test code, will get changed later) and to display directory contents. You will need to see what is happening in your directory.
3. Define your bit-map representation and create some form of display function that can indicate whether blocks are free or allocated. Test your display.
4. Define a simple createFile function. New files will simply be allocated the next set of blocks. Make this simplified version work with your directory and bit map displays.
5. Define a simple deleteFile function. Write a driver to create several files, delete a subset, and check that the results all display correctly.
6. Implement a more realistic createFile function – this one must now search for the appropriate place to create a new file (as deleteFile will have opened up gaps where a file might fit). Test your functions and your display of the results.
7. You could now probably use the "runSimpleSimulation()" function from the supplied library file.
8. Implement compactFiles. Test your code and display the results.
9. Implement the read and write operations. (Your compactFiles function had better remember to update diskFileContents[] when it moves those files.)
10. Try running with the "runSomeErrorTests()" function. If it complains, check your code.
11. Run with the "runSimulation()" function.

Assignment report

You do not submit your code files!

You are marked on a report that you write. This report presents details of your "Disk.c" implementation, and demonstrates that it works with the supplied driver.

Submission

Prepare your report and convert to PDF format as the file A3.pdf.

Submission is done electronically via a program called `turnin` that runs on "banshee" – the main CS undergraduate machine. You must first transfer your A3.pdf file to your home directory on banshee (this is different from the home directory that you access on the Linux machines). You can transfer the file using a SSH file-transfer program. The Ubuntu OS allows you to open a file-browser connected to your banshee home directory – and you can simply drag your A3.pdf file across using the visual file browser.

For CSCI131, assignments are submitted electronically via the `turnin` system. For this assignment you submit your assignment via the command:

```
turnin -c csci131 -a 3 A3.pdf
```

Late submissions would be submitted as:

```
turnin -c csci131 -a 3late A3.pdf
```

The program `turnin` only works when you are logged in to the main banshee undergraduate server machine. From an Ubuntu workstation in the lab, you must open a terminal session on the local machine, and then login to banshee via ssh and run the `turnin` program.

Marking

The assignment is worth 7 marks total.

- Appearance and structure of report: 1 mark
- Evidence for correct operation: 1 mark
- Code and explanations of your implementation: 5 marks total