SCC.211 Assessed Exercise 2: *Filesystem*

Table of Contents

Overview

The module is coursework weighted and the final exercise is more of a stretch goal. You are tasked with creating an implementation for a constrained, but real world, Systems problem.

The exercise is to design and build a program that can read files from the type of filesystem commonly used on SD cards, etc.

By the end of this exercise, you should have an in-depth understanding of how one of the most widely used filesystems stores and indexes files, and have the confidence to take on real world problems.

Plagiarism

You should know this by now…

This is an individual exercise and the normal plagiarism rules apply. You must not use, include, or follow code found on the web, repositories, or any other external source. You must not solicit code from others. Anyone found to have taken code from, or provided code to, another individual, or to have uploaded code to a public or open code repository before or after submission, will be awarded a mark of zero.

Having said that, you are encouraged to discuss concepts and general principles, but keep discussion away from the specifics of solutions – the submission must be your own work.

Remember, one of the key aims of these exercises is to develop understanding and skills in design and coding. Taking designs or code from elsewhere will mean you struggle later on – if you are having problems, ask one of the Teaching Assistants.

Teaching Assistants are there to help, but do not expect them to just give you solutions, or a step-by-step guide on how to develop a solution. The aim is to develop your understanding and skills.

Beware using online sources as they will often discuss more general solutions that without constraints are more complex than expected.

Background

Filesystems

Filesystems provide a structured and managed way to store, index, and access files. There are many different filesystems, each optimized for the typical size and access pattern of files, the size and type of media it is typically held on, whether Hard Disk (HD), Secure Digital (SD) card, Solid State Drive (SSD), etc.; however, they are mostly variations on a small number of basic designs.

Filesystems often occupy only part of the media they reside on, sitting alongside other filesystems perhaps optimized for a different task, or they may span multiple storage devices to gain extra capacity or redundancy. Where a filesystem shares a storage device with other filesystems, we say the device or media is partitioned into multiple volumes. A filesystem thus sits within a volume, and the size of the filesystem may not exceed the size of the volume.

As a filesystem must strictly define how and where data is placed within the area it manages, there is typically a set of metadata at the start of the filesystem that describes how the filesystem is tuned to fit into the space it manages. With this metadata, and a set of 'rules' set out in the design, it should be possible to locate any part of any file held within the filesystem.

Many filesystems use terminology from when hard disks were the main form of storage media. You will often find metadata using heads and sectors. Recall that for simple hard disks, the head defines which disk surface is being accessed. Heads step between distinct tracks, each split into a number of sectors. A sector is thus a region on a disk seen by a disk head as it sweeps through a defined arc within a specific track.

Disks are block devices as we always read or write multiple bytes at a time. As disks are addressed in sectors, the block size of a filesystem must be a multiple of the sector size. For efficiency, a filesystem may read and write multiple adjacent sectors with each access, in which case a filesystem block is a *cluster* of sectors.

Note that the filesystem always reads or writes whole blocks whether an application request asks for this or not. If the filesystem retains a copy of recently accessed blocks in memory this can be much more efficient than repeatedly accessing the external device, particularly if it is an old hard disk. This can also reduce the number of writes to the device, which can shorten the life of some types of storage device.

One final thing of note is that filesystems sometimes include metadata to identify the boot code that must run to start an installed operating system. This removes the need for the initial boot code to understand the filesystem structure.

FAT16

FAT16 is one of a family of closely related filesystems developed for PCs and now widely used due to their relative simplicity. The version of FAT used, FAT12, FAT16, or FAT32, is defined by the size of the media the filesystem resides on, thus optimizing the size/ limit of index values and data structures to the storage limit of the device; thus, large data structures are not set up on devices where there will never be data fill them.

The name FAT comes from the File Allocation Table used to track the clusters that form each file. This class of filesystem can be very inefficient, as the FAT must be accessed every time a file read or write crosses a cluster boundary, but the FAT designs keep the File Allocation Table small enough to be cached in memory for fast access.

The Exercise

Your task is to create a clean-slate implementation of the FAT16 filesystem that can read disk images created on regular devices using FAT, such as Linux or Microsoft Windows.

To be clear, you will read a file that is a byte-by-byte copy of a volume or 'disk' holding a FAT16 filesystem. **You are not expected to:**

- Write to the file or modify it in any way
- Support any version of FAT other than FAT16

The exercise will take time to develop so start early and give yourself time to think and plan, ask for and digest support. If you become stuck, do not leave asking for help so late it becomes too late to act on any suggestions.

You will no doubt have noticed that this document is longer than that for previous exercises – you will hopefully come to appreciate that this is something to welcome, as the detail is there to help you. Proper specifications do tend to be long, but this leaves fewer things open to question, which makes things far simpler in the end.

The problem is broken down into a set of sub-tasks allowing you to more easily work toward a complete solution. While some tasks are more tricky than others, you should find most relatively straightforward. If you are struggling, you may be looking at the problem in the wrong way or reading more into the task than expected.

Having said that, you will need to build on your knowledge and experience from first-year, particularly on C structures, C pointers, and general data structures. If you remain unsure of C, and these topics, all the more reason to start early.

While it is good to look ahead and get an overview of the whole exercise, **you are not expected or required to understand all the detail before you start**. The tasks build on each other, so a task may be much clearer once you have completed those before it. Focus on the task in hand.

This is a Systems problem, so pay real attention to the detail of what you are doing, double check any assumptions, and test everything as you go. If you write too much code before proper testing, it will become increasingly difficult and time consuming to locate errors. Always ask yourself what could go wrong with any code you write, and make sure your code will handle any odd edge cases. There is nothing worse than spending hours debugging code, only to realise the problem is with another, older piece of code you failed to debug properly at the time.

Reading the Disk Image

The disk image is a file containing a byte-by-byte copy of a FAT16 volume and you will need code that can read a set number of bytes starting at a defined position in the file. For example, the filesystem metadata might state that the first cluster of a file starts at byte 40960 and the second cluster of the file starts at byte 65536; to assemble the file your code will need to be able to jump to the first cluster, read it, jump to the second cluster, read that, and so on.

The first thing is therefore to ensure you can read set parts of files, and for this, you should look at the following system calls:

- open( ) -- tells the system you want to read the FAT16 image
- lseek( ) – move to a given point in the file
- read( ) – read a number of bytes from a file into a given area of memory
- close( ) – tells the system any resources used for reading the image can be released.

To properly understand how to use these calls you should look at the manual pages that will state which include files are needed, any #define lines required, give you the order and types of the parameters, and a detailed description of what each function does.

Unix Manual Pages

You can get the manual for system calls by typing, say, `man 2 read`

The 2 specifies the section of the manual to search, as read is a common name that may appear as a command, a system call, etc. You will often see this referenced as, for example, read(2).

There are eight sections:

1. Commands -- things you can type at the command prompt
2. System calls -- functions that directly communicate with the operating system kernel
3. Libraries -- functions that build on the system calls, for example, printf( ), which provides useful functionality to applications by building on the write( ) system call
4. Special files -- typically devices under /dev, such as /dev/tty
5. File formats -- content of configuration files such as /etc/fstab
6. Games
7. Miscellaneous -- includes macro packages and standards such as ASCII, IP, TCP, UDP, etc.
8. Maintenance -- special commands such as sudo, etc.

Example Manual Entry

The following shows the key sections of a manual page, in this case for open( ).

We can see that open is one of a set of three closely related functions, open( ), openat( ), and creat( ), and that some have different forms, in this case possibly including a mode. Each of these is described within the description.

Notice from the synopsis that we need to include three header files in order to use these particular functions within our code, and that the type of the return value and all the parameters are listed for each function. Any non-standard types are defined in the header files and explained within the description.

At the end of the manual page, you will find details of what is returned by the function in the case of success or failure, and what the error cases are.

```
NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

DESCRIPTION
…


RETURN VALUE
       open(), openat(), and creat() return new file descriptor, or -1 if
       an error occurred (in which case, errno is set appropriately).

ERRORS
       open(), openat(), and creat() can fail with the following errors:

       EACCES The requested access to the file is not allowed, or…
```

TASK 1

Demonstrate you can open a file, perhaps a text file, jump to a specific point given a byte offset into the file, and read a given number of bytes or characters before closing the file.

Accessing the Boot Sector and BIOS Parameter Block

The first part of the volume, starting with the first byte, contains the Boot Sector (BS) and BIOS Parameter Block (BPB) data, which describes how the filesystem is configured for this particular volume. Note your code must not make assumptions about these values, as your code will be tested with other FAT16 volumes that may have a different configuration / set of values.

Hint: Thinking about the structure of your solution going forward, you should aim to have the code you wrote in the first stage being the only code that does file reads. Make this a sector reader that takes sector numbers and returns sectors of the configured size. Remember, every time you hide complexity like this, you make your job easier as, for example, you can forget about the detail of actually reading things from the image file. With each step, look for things you can encapsulate, simplifying the problem as you go.

Metadata structure

The beginning of a FAT12 or FAT16 volume will always have the following content describing the structure of what follows. The FAT32 structure inserts some additional values between *BPB_TotSec32* and *BS_DrvNum*, but you can safely ignore this, as you only need to handle FAT16.

```
typedef struct __attribute__((__packed__)) {
    uint8_t    BS_jmpBoot[ 3 ];      // x86 jump instr. to boot code
    uint8_t    BS_OEMName[ 8 ];      // What created the filesystem
    uint16_t   BPB_BytsPerSec;       // Bytes per Sector
    uint8_t    BPB_SecPerClus;       // Sectors per Cluster
    uint16_t   BPB_RsvdSecCnt;       // Reserved Sector Count
    uint8_t    BPB_NumFATs;          // Number of copies of FAT
    uint16_t   BPB_RootEntCnt;       // FAT12/FAT16: size of root DIR
    uint16_t   BPB_TotSec16;         // Sectors, may be 0, see below
    uint8_t    BPB_Media;            // Media type, e.g. fixed
    uint16_t   BPB_FATSz16;          // Sectors in FAT (FAT12 or FAT16)
    uint16_t   BPB_SecPerTrk;        // Sectors per Track
    uint16_t   BPB_NumHeads;         // Number of heads in disk
    uint32_t   BPB_HiddSec;          // Hidden Sector count
    uint32_t   BPB_TotSec32;         // Sectors if BPB_TotSec16 == 0
    uint8_t    BS_DrvNum;            // 0 = floppy, 0x80 = hard disk
    uint8_t    BS_Reserved1;         //
    uint8_t    BS_BootSig;           // Should = 0x29
    uint32_t   BS_VolID;             // 'Unique' ID for volume
    uint8_t    BS_VolLab[ 11 ];      // Non zero terminated string
    uint8_t    BS_FilSysType[ 8 ];   // e.g. 'FAT16   ' (Not 0 term.)
} BootSector;
```

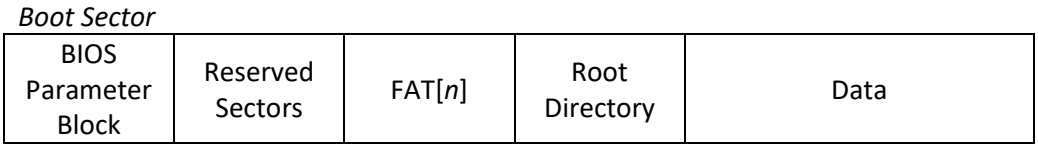
Hint: Do you understand how C lays out structures in memory? If you are unsure, take a look at: C Structs and Pointers (andrew-scott.uk) You might also find: C Pointer Arithmetic (andrew-scott.uk) useful. These links are also available via Moodle.

Task 2

Demonstrate that, given a file containing a FAT filesystem image, you can load the above structure and output the values of the highlighted configuration parameters.

Loading and Following a File Allocation Table

The layout of the filesystem is:



Notice there are some reserved sectors – while we do not need to worry about the content of these, we do need to know the number of reserved sectors so we can work out the sector where the file allocation tables start. This number is the *BPB_RsvdSecCnt* value you have already read.

FAT Format

The FAT is simply an array of entries, one for each cluster, which states the next cluster in the sequence that makes up a file. Thus, each file stored in the file system will have a chain of entries in the FAT that identifies the clusters making up the file, and the order they appear in the sequence.

We will see later that the start of each chain is given in the directory entry that describes the file the chain forms.

For FAT16, each entry in the FAT is an unsigned 16-bit integer. In case you were wondering, FAT32 uses 32-bit entries, but again, you only need to handle FAT16.

Finding FAT Entries

With each FAT entry being a simple integer identifying the next entry, or cluster, in the chain, it is easy to scan through files. Note there is no direct way of identifying cluster *n* in a file, you have to start at the first cluster, move to the next, then the next, until you find the *n*th cluster. This is the big problem with FAT based systems; however, they do work well for small filesystems where the entire FAT can be cached in memory, thus reducing each current cluster to next cluster mapping to a fast in-memory read.

In order to cache a FAT, we need to know the size of each FAT in the filesystem. This is the *BPB_FATSz16* value you read earlier.

There is **one last important point**, and that is the way clusters are numbered. The first usable cluster for files is always considered to be cluster two, even if that does not match the natural numbering of sectors, and so the very first entry in the FAT is a 16-bit unsigned integer that identifies the second cluster of the file that starts in cluster two.

End of File

An entry of 0xfff8 or greater indicates that there are no further clusters in the corresponding file, in other words, the end of the file has been reached.

Task 3

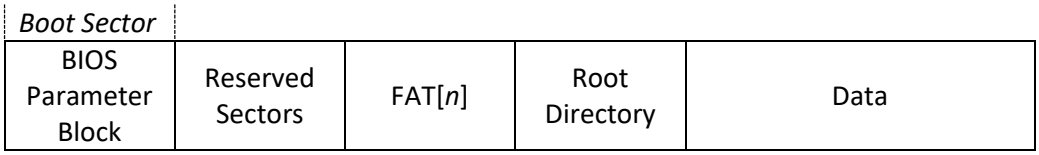
Demonstrate that, given a FAT16 filesystem, you can load a copy of the first FAT into memory and that, given a starting cluster number, you can produce an ordered list of all the clusters that make up a file.

Reading the Root Directory

The root directory of a FA12 or FAT16 filesystem is special in that it has a set place in the structure. While this makes it easy to find, it comes with some limitations that we can safely ignore for now.

We read the root directory in the same way we do for any other directory, but we must special case its position, so bear in mind that you will need to extend any code you write when you come to look at other directories.

You can identify the location of the root directory by calculating where the last FAT ends. The following sector will be the start of the root directory. Remember the layout:



The first sector of the root directory is therefore: *BPB_RsvdSecCnt* + *BPB_NumFATs* * *BPB_FATSz16*.

As the FAT is so important for understanding which clusters belong to which files, multiple copies of the FAT are stored for resilience – if one is corrupt, there should normally be another to allow the filesystem to be reconstructed. The value *BPB_NumFATs* gives the number of FATs in the filesystem, which should normally be two for FAT16, but you should always check, as other values are valid.

Basic Directory Structure

Each entry in a directory has the following content:

```
uint8_t    DIR_Name[ 11 ];       // Non zero terminated string
uint8_t    DIR_Attr;             // File attributes
uint8_t    DIR_NTRes;            // Used by Windows NT, ignore
uint8_t    DIR_CrtTimeTenth;     // Tenths of sec. 0...199
uint16_t   DIR_CrtTime;          // Creation Time in 2s intervals
uint16_t   DIR_CrtDate;          // Date file created
uint16_t   DIR_LstAccDate;       // Date of last read or write
uint16_t   DIR_FstClusHI;        // Top 16 bits file's 1st cluster
uint16_t   DIR_WrtTime;          // Time of last write
uint16_t   DIR_WrtDate;          // Date of last write
uint16_t   DIR_FstClusLO;        // Lower 16 bits file's 1st cluster
uint32_t   DIR_FileSize;         // File size in bytes
```

A directory is simply an array of these entries, each giving details of a file, including the first cluster number, { *DIR_FstClusHI*, *DIR_FstClusLO* }, for the corresponding file, directory, etc. Subsequent clusters are found in the FAT.

File Names

Names are held in the old eight dot three format, such as, *ABSTRACT.TXT*, thus eleven ASCII characters given the dot is not stored. Note that this is not a conventional zero-terminated C string.

The eight-character name and three-character extension are padded with spaces (0x20) that are not considered part of the string and are not output. If the first byte of the directory name is zero, there are no further valid entries in the directory, and if the byte is 0xE5, the specific entry is currently unused and should be ignored, perhaps due to a deleted file.

File Attributes

To understand what a directory entry represents, we need to look at the *DIR_Attr* value, which has the following structure:

Bit:	7	6	5	4	3	2	1	0
Entry Type:			Archive	Direct-ory	Volume Name	System	Hidden	Read Only

If both the directory bit and the volume ID/ Name bit are both zero, the entry corresponds to a regular file, such as a text file, a PDF, etc.

Looking at both bits three and four: if just the volume bit is set, the entry represents the name of the 'disk', which is normally shown alongside the drive letter in Windows; however, if just the directory bit is set, the entry represents the name of a directory, or folder in Windows.

If all lower four bits, 0...3, are set, and bits .4 and 5 are both zero, the directory entry should be ignored for now. We will see how to handle these entries later.

Date Format

The creation date, *DIR_CrtDate*, and the last write/ modification date, *DIR_WrtDate*, have the following representation:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Year, 0 → 1980							Month, 1..12				Day, 1..31				

To be clear, if the year bits are all zero, the date would be in 1980, and if just the first bit in the year field is set, the year would be 1981, etc.

Time Format

Times are held as the number of two-second intervals since midnight. When the FAT specification was originally written, two-second accuracy was sufficient; however, it was seen as necessary to have a more accurate indication of the time files are created. An additional field, *DIR_CrtTimeTenth*, holds values 0...199, giving the tenths of seconds, giving finer granularity for the creation time.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Hours (0..23)					Minutes (0..59)						Seconds / 2 (0..29)				

Note that there is no indication of time zone; times are typically mapped to the local time zone.

Task 4

Demonstrate that you can decode and output the list of files in the root directory. For each file, you should output the first/ starting cluster, the last modified time and date, the file attributes using a single letter for each, i.e., ADVSHR, with a – (dash/ hyphen) used to indicate an unset flag, the length of the file, and finally the filename. Output should be formatted neatly in columns.

Accessing a File

In the simplest sense, reading a file is simple as we get the first cluster number from the corresponding directory entry, and follow the FAT chain from that entry. This will give an ordered set of clusters holding the file content.

You should always check that the directory entry refers to a regular file by checking both the volume and directory attributes are both set to zero.

The first step is to build on your FAT handing code to develop a function that can return a given set of bytes from the *n*th cluster from a file.

The next step is to build a proper, user-centric, file interface able to read an arbitrary set of bytes from a file, subject to the stated file length. Importantly, you should not be able to read beyond the end of a file, and you must handle cases where the bytes requested do not align with cluster boundaries, and span multiple clusters.

Depending on how you implemented your FAT handling, this could be a very simple mapping to what you already provide.

At this point, it would be good to mirror the normal Unix file functions, giving something like:

```
extern File *   openFile  ( Volume *, ShortDirEntry * );
extern off_t    seekFile  ( File *, off_t offset, int whence );
extern size_t   readFile  ( File *, void * buffer, size_t length );
extern void     closeFile ( File * );
```

Look at Unix manual pages for read(2), lseek(2), and close(2) to see how this type of interface would normally work.

Task 5

Demonstrate the provision of the above functions that, given a short directory entry as read in the previous task, can output the contents of the corresponding file.

Handling Long File Names

When discussing file attributes, we said: If all lower four bits, 0...3, are set, and bits 4 and 5 are both zero, the directory entry should be ignored for now. This was because this combination of attributes signals the entry is part of a long file name.

When FAT was first introduced, it was common to have simple file name formats such as the familiar 8.3 you have been working with so far. There was soon a need to improve the naming scheme, but this had to be done within the limitations imposed by the existing directory entry structure.

The solution was to add a set of new (long) directory entries, immediately before the main 8.3 directory entry, that together form an alternative long name for the file, so the traditional short 8.3 name is only used if no long name immediately precedes it.

Format of Long Directory Entry

A long directory entry has the following content:

```
uint8_t    LDIR_Ord;            // Order/ position in sequence/ set
uint8_t    LDIR_Name1[ 10 ];    // First 5 UNICODE characters
uint8_t    LDIR_Attr;           // = ATTR_LONG_NAME (xx001111)
uint8_t    LDIR_Type;           // Should = 0
uint8_t    LDIR_Chksum;         // Checksum of short name
uint8_t    LDIR_Name2[ 12 ];    // Middle 6 UNICODE characters
uint16_t   LDIR_FstClusLO;      // MUST be zero
uint8_t    LDIR_Name3[ 4 ];     // Last 2 UNICODE characters
```

Note that each character is in a 16-bit UNICODE format allowing international characters to be stored. As ASCII is character set zero, ASCII characters are represented as a zero byte followed by the normal 8-bit ASCII value.

Assuming the directory is not corrupt, you only really need to worry about *LDIR_Attr*, and the name fields. If there are multiple entries forming a single long name, the first entry will be the end of the name, i.e. the last part, and the entry immediately preceding the normal 8.3 entry will be the start or first part of the long name.

Task 6

Demonstrate you can neatly output the content of the root directory with long file names properly decoded.

Paths and the Directory Tree

There is one last step and this is to be able to return the short directory entry corresponding to a file identified by its full path, for example, "/directory1/directory2/file.txt"

FAT directories are essentially just files… aside from the root directory of a FAT12 or FAT16 volume being a special case held separately, directories have at least a short directory entry, and have a chain of clusters listed in the FAT that hold the content. All directories are formed from the same directory entries you have been working with. You might consider having a generic implementation for a 'stream' that can be used as a clean way of handling the different interpretations of a chain of clusters, i.e., files and directories.

To identify a directory, check both the directory and volume attributes; only the directory attribute should be set for a directory.

As with Unix filesystems, you will typically find two special directory entries, . (dot) – a link to the current directory, and .. (dot, dot) – a link to the parent directory. So /dir1/.. would lead to the root directory, as the path goes down to directory dir1, and then back up to the root directory.

Depending on how you have structured what you have done so far, this might be a bit more tricky than previous tasks – this is where good data structures really help. You might also ponder recursive and iterative solutions.

Task 7

Demonstrate that given a path with an arbitrary mix of long and short file and directory names you can locate and output the corresponding file.

Next Steps

Not part of the assessment… If you want to go further, you might consider the following suggestions…

- Ensure your solution can handle multiple open files at once. In this case, you should be able to open multiple files, correctly handle interleaved reads from these files – i.e. keeping track of the current file position for each, and close them.
- Provide a simple shell or terminal that accepts simple commands such as ls, cat, more, hd, od, etc. Have these accept parameters and produce output as shown in the Unix/ Linux manual pages.

Or maybe you can come up with some ideas yourself…

Developing Your Code

You should provide a solution built using just the standard C library calls, and must not use any additional libraries to create your solution. Your code must read the FAT16 image file directly. The following should be sufficient for a working solution…

Useful Manual Pages

You may find the following functions useful:

`bcopy(3)`
`bzero(3)`
`close(2)`
`exit(3)`
`fprintf(3)`
`fputc(3)`
`free(3)`
`fwrite(3)`
`index(3)`
`lseek(2)`
`malloc(3)`
`memcpy(3)`
`memset(3)`
`mktime(3)`
`open(2)`
`perror(3)`
`printf(3)`
`putchar(3)`
`read(2)`
`sprint(3)`
`strlen(3)`
`strncpy(3)`
`strptime(3)`

One common point of confusion is the distinction between the two sets of file operations:

`open( ), lseek( ), read( ), write( ), close( )`, and
`fopen( ), fseek( ), fread( ), fwrite( )`, and `fclose( )`.

The first set are lower level functions that directly map to the kernel interface. The second set, with the *f* prefix, are built on the first set and offer a richer interface, including buffering, etc. Notice one set are in volume two of the manual, and the other in volume three.

As one set identifies files with a native Unix, integer based, file descriptor, and the other a FILE *, they are not directly interchangeable. A FILE * can also be used with functions such as fprintf( ) and fscanf( ).

Multiple Source Files

When splitting a C program into multiple source files you need to create an include file that contains declarations for any functions used outside of the file they are defined in.

The declaration is just the keyword *extern* followed by the function name and parameter list; the types are mandatory, but variable names are optional though often helpful.

For example, a file that makes a call to a function myRead( ) might have the line:

```
#include "mydefs.h"
```

…referencing a file containing the line:

```
extern int myRead ( File *, char * buffer, int bytesToRead );
```

From this, we can tell the function returns an integer, and requires a reference to some File object, a pointer to a buffer presumably used to hold the bytes read, and the number of bytes to be read.

Notice we specified names for the second and third variable as they help us understand what these parameters are, but the name for the first parameter is left unspecified as the type gives sufficient context by itself.

Makefiles

It can be tempting to rely on IDEs to help with coding and building projects, but you often learn far quicker by just using a basic editor and traditional build tools such as make.

Having a makefile for your project really simplifies the build process and avoids the need to remember which files have been edited and thus need recompiling. The following is a simple makefile that assumes all your files are in the same directory as the makefile, normally called Makefile.

```
BIN = $(shell basename $(PWD))

SRCS = $(shell ls *.c)

HDRS = $(shell ls *.h)

OBJS = $(SRCS:.c=.o)

CFLAGS = -I.

%.o : %.c $(HDRS)
	$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

$(BIN): $(OBJS)
	$(CC) $(CFLAGS) $^ -o $@

.PHONY: clean
clean:
	$(RM) $(OBJS) $(BIN)
```

Note the indentation is a tab and not multiple spaces.

The example above defines a target binary on the first line that will have the same name as the directory it sits in, but you could change this to whatever you want your executable program called. As it stands, if the Makefile and code is in a directory called FATfs, the executable will be named FATfs.

The Makefile also defines a 'phony' target called clean, which simply removes compiled files to reduce space. There are therefore two commands:

`make`	to build your program
`make clean`	to remove any compiled files

This setup can be improved for larger projects by having separate subdirectories for include files, source files, and object files.

The manual page for make just covers the basic operation, but you can find complete details at: https://www.gnu.org/software/make/

Getting Support

There will be Teaching Assistants (TAs) available during your timetabled laboratory sessions to help with any problems.

Remember TAs are there to help you learn and gain experience by nudging you in the right direction, not to provide the solution.

Before you ask, always have a play, add some print functions, and see what looks right and what does not. Have a think what could go wrong… have you got an off by one error, or something similar? Check any functions you call are returning the values you expect… have you misinterpreted or missed testing something?

Make an early start in case you need to review any first-year material. The TAs are also likely to be getting fewer questions and have more time early on.

Outside of your timetabled session, use the Moodle forum for general questions, but do not post blocks of code or specific solutions. Good answers are often of the form: have you checked…, or have a look at the fprintf( ) function.

Submission and Marking

To get a mark, **you must** submit all your source files to Moodle by the deadline, **and you must** attend your timetabled demonstration session to show your work to one of the TAs. Failing to do either of these will result in a mark of zero.

You must demonstrate the code as submitted to Moodle; demonstrating code different to that submitted may result in a mark of zero. If you have fixed a bug after submission, or spent time adding a new feature, the TA might look at this after they have seen what you submitted, but you must ask first. To be fair to everyone, your grade will be based on what you submitted, with anything else used as supplementary evidence.

During your demonstration, be guided by the TA and do not feel the need to go line by line through your code… The time a TA spends looking at your code is not an indicator of the mark you can expect; there are many reasons why a TA might spend more or less time looking at a submission.

Code should be well structured and neatly presented. You should make good use of language features, and include comments throughout your code, which should make clear the role of functions, variables, and code. With few exceptions, you should also be routinely checking the return values from functions and correctly handling any error conditions.

As a rough guide, a robust and reliable solution for each task that is well structured and well commented , and conforms to the specified requirements, should gain around ten marks provided you are able to demonstrate good understanding during the demonstration. Depending on complexity, some tasks will be a little less, some a little more.

Feedback

In addition to anything that might have been said during the general laboratory sessions, a TA will give you feedback while looking at your code during your demonstration. If you want more feedback, this is the time to ask, but please note that time is necessarily limited.

Extensions

Aside from **exceptional** circumstances, **extensions need to be approved in advance** and can only be arranged through the Teaching Office: scc-teaching-office@lancaster.ac.uk

If you do feel you are subject to exceptional circumstances, do contact the Teaching Office; there is no harm in asking, and we can only help if a formal request has been made.