# COMP8051 – Operating System Engineering - Assignment 2

**Completion Date: 16th April 2021**

**Value: 30 marks**

**On completion please zip up your files and upload to Canvas.**

## Question 1 Inode and Block layer in xv6

xv6 divides the disk into several sections, as shown in the Figure below.

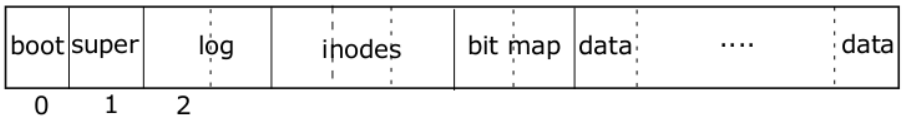


**Figure 6-2.** Structure of the xv6 file system. The header `fs.h` (4050) contains constants and data structures describing the exact layout of the file system.

The file system does not use block 0 (it holds the boot sector). Block 1 is called the superblock, it contains metadata about the file system (the file system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold inodes. After those come bitmap blocks tracking which data blocks are in use. Most of the remaining blocks are data blocks. The blocks at the end of the disk hold the logging layer's log.

The term inode can have one of two related meanings. It might refer to the on-disk data structure ( struct dinode, fs.h ) containing a file's size and list of data block numbers. Or "inode" might refer to an in-memory inode ( struct inode, file.h ), which contains a copy of the on-disk inode as well as extra information needed within the kernel.

The on-disk inode is defined by a struct dinode. The type field distinguishes between files, directories, and special files (devices). A type of zero indicates that an on-disk inode is free. The nlink field counts the number of directory entries that refer to this inode, in order to recognize when the on-disk inode and its data blocks should be freed. The size field records the number of bytes of content in the file. The addrs array records the block numbers of the disk blocks holding the file's content.

The function *readi* is essentially called by the user level read function to read an amount of data from a file. The readi parameters are:

```
readi(struct inode *ip, char *dst, uint off, uint n)
```

The inode pointed to by *ip* abstracts the file layout/structure in the filesystem. Readi uses the addrs array to find the block numbers that are associated with the file. It then reads in all the blocks from the disk to satisfy the read request, in xv6 each block is the same size as a disk sector, which is 512 bytes.

The function *Bmap* makes it easy for readi and writei to get at an inode's data. Readi (5503) starts by making sure that the offset and count are not beyond the end of the file. Reads that start beyond the end of the file return an error (5514-5515) while reads that start at or cross the end of the file return fewer bytes than requested (5516-5517) . The main loop processes each block of the file, copying data from the buffer into dst (5519-5524) . writei (5553) is identical to readi, with three exceptions: writes that start at or cross the end of the file grow the file, up to the maximum file size (5566-5567) ; the loop copies data into the buffers instead of out (5572) ; and if the write has extended the file, writei must update its size (5577-5580) .

**Read chapter 6 of the xv6 book and briefly explain how the read calls in the following code taken from cat.c**

```
void cat(int fd)
{
  int n;

  while((n = read(fd, buf, sizeof(buf))) > 0) {
    if (write(1, buf, n) != n) {
      printf(1, "cat: write error\n");
      exit();
    }
  }
  if(n < 0){
    printf(1, "cat: read error\n");
    exit();
  }
}
```

**are associated with sectors on the disk by the xv6 operating system. You may use a diagram such as the one here to illustrate your answer.**

***(10 marks)***

## Question 2 IDE Disk Driver

Read the sections on Drivers, Code: Drivers in chapter 3 and section 36.8 of file-devices.pdf, which gives a summary of the IDE disk controller protocol. See in particular the code in **ide.c**. See image below

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6    5      4    3    2    1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6    5      4    3    2    1    0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

The xv6 source code includes a working IDE driver in ide.c. For example the piece of code in *idestart*

*outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b→sector>>24)&0x0f));*

is associated with:

I/O Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive

The 0xe is to set 1B1 in bits 5,6,7 B=1 indicates that we are going to have the low 4 bits at address 0x1f6 set the top 4 bits of the Logical Block Address (LBA).

((b→dev&1)<<4) sets the D bit.

((b→sector>>24)&0x0f)) sets the top 4 bits of LBA to low four bits at address 0x1f6.

An IDE disk presents a simple interface to the Disk system, consisting of four types of register: control, command block, status, and error. These registers are available by reading or writing to specific “I/O addresses” (such as 0x3F6 ) using (on x86) the in and out I/O instructions.

On page 48 of the xv6 book we read “The xv6 bootloader issues disk read commands and reads the disk controller status bits repeatedly until the data is ready (see Appendix B). This polling or busy waiting is fine in a boot loader, which has nothing better to do. In an operating system, however, it is more efficient to let another process run on the CPU and arrange to receive an interrupt when the disk operation has completed.”

**a) Explain how the xv6 operating system uses interrupts to schedule I/O requests to the disk?**

**b) Explain how the bootloader interfaces with the IDE controller to load the xv6 operating system – see Appendix B of the xv6 book (section "Code: C bootstrap" in particular) and bootmain.c in the xv6 source code?**

***(12 marks)***

## Question 3 Big Files in xv6

Currently xv6 files are limited to 140 sectors, or 71,680 bytes. This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 128 more block numbers, for a total of 12+128=140. You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 128 addresses of singly-indirect blocks, each of which can contain up to 128 addresses of data blocks. The result will be that a file will be able to consist of up to 16523 sectors (or about 8.5 megabytes).

### Preliminaries

Modify your Makefile's `CPUS` definition so that it reads:

CPUS := 1

Add

QEMUEXTRA = -snapshot

right before `QEMUOPTS`

The above two steps speed up qemu tremendously when xv6 creates large files.

`mkfs` initializes the file system to have fewer than 1000 free data blocks, too few to show off the changes you'll make. Modify `param.h` to set `FSSIZE` to:

```
#define FSSIZE       20000  // size of file system in blocks
```

Download big.c into your xv6 directory, add it to the UPROGS list, start up xv6, and run `big`. It creates as big a file as xv6 will let it, and reports the resulting size. It should say 140 sectors.

### What to Look At

The format of an on-disk inode is defined by `struct dinode` in `fs.h`. You're particularly interested in `NDIRECT`, `NINDIRECT`, `MAXFILE`, and the `addrs[]` element of `struct dinode`. Look here for a diagram of the standard xv6 inode.

The code that finds a file's data on disk is in `bmap()` in `fs.c`. Have a look at it and make sure you understand what it's doing. `bmap()` is called both when reading and writing a file. When writing, `bmap()` allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

`bmap()` deals with two kinds of block numbers. The `bn` argument is a "logical block" -- a block number relative to the start of the file. The block numbers in `ip->addrs[]`, and the argument to `bread()`, are disk block numbers. You can view `bmap()` as mapping a file's logical block numbers into disk block numbers.

## Your Job

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block.

You don't have to modify xv6 to handle deletion of files with doubly-indirect blocks.

If all goes well, `big` will now report that it can write 16523 sectors. It will take `big` a few dozen seconds to finish.

## Hints

Make sure you understand `bmap()`. Write out a diagram of the relationships between `ip->addrs[]`, the indirect block, the doubly-indirect block and the singly-indirect blocks it points to, and data blocks. Make sure you understand why adding a doubly-indirect block increases the maximum file size by 16,384 blocks (really 16383, since you have to decrease the number of direct blocks by one).

Think about how you'll index the doubly-indirect block, and the indirect blocks it points to, with the logical block number.

If you change the definition of `NDIRECT`, you'll probably have to change the size of `addrs[]` in `struct inode` in `file.h`. Make sure that `struct inode` and `struct dinode` have the same number of elements in their `addrs[]` arrays.

If you change the definition of `NDIRECT`, make sure to create a new `fs.img`, since `mkfs` uses `NDIRECT` too to build the initial file systems. If you delete `fs.img`, `make` on Unix (not xv6) will build a new one for you.

If your file system gets into a bad state, perhaps by crashing, delete `fs.img` (do this from Unix, not xv6). `make` will build a new clean file system image for you.

Don't forget to `brelse()` each block that you `bread()`.

You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.

***(8 marks)***