Operating Systems Engineering
Assignment 2: Q1 & Q2

R00159222 - Zachary Dair
zachary.dair@mycit.ie

This document provides answers to the theory questions, **Q1** and **Q2 (parts a and b)**

Question 1:

Cat.c reads a file and displays the contents back to the user, to accomplish this the data of the file to be read must be retrieved from memory, once retrieved from the disk this is data is found in the block layer of the file system.

In the code of Cat.c the ***read*** function is a user level function that correlates to the ***sys_read*** system call this enables the arguments to be passed from the user level into the kernel level. These arguments are the file descriptor (***fd***), the buffer (***buf***) and the size of the buffer (***512 bytes***).

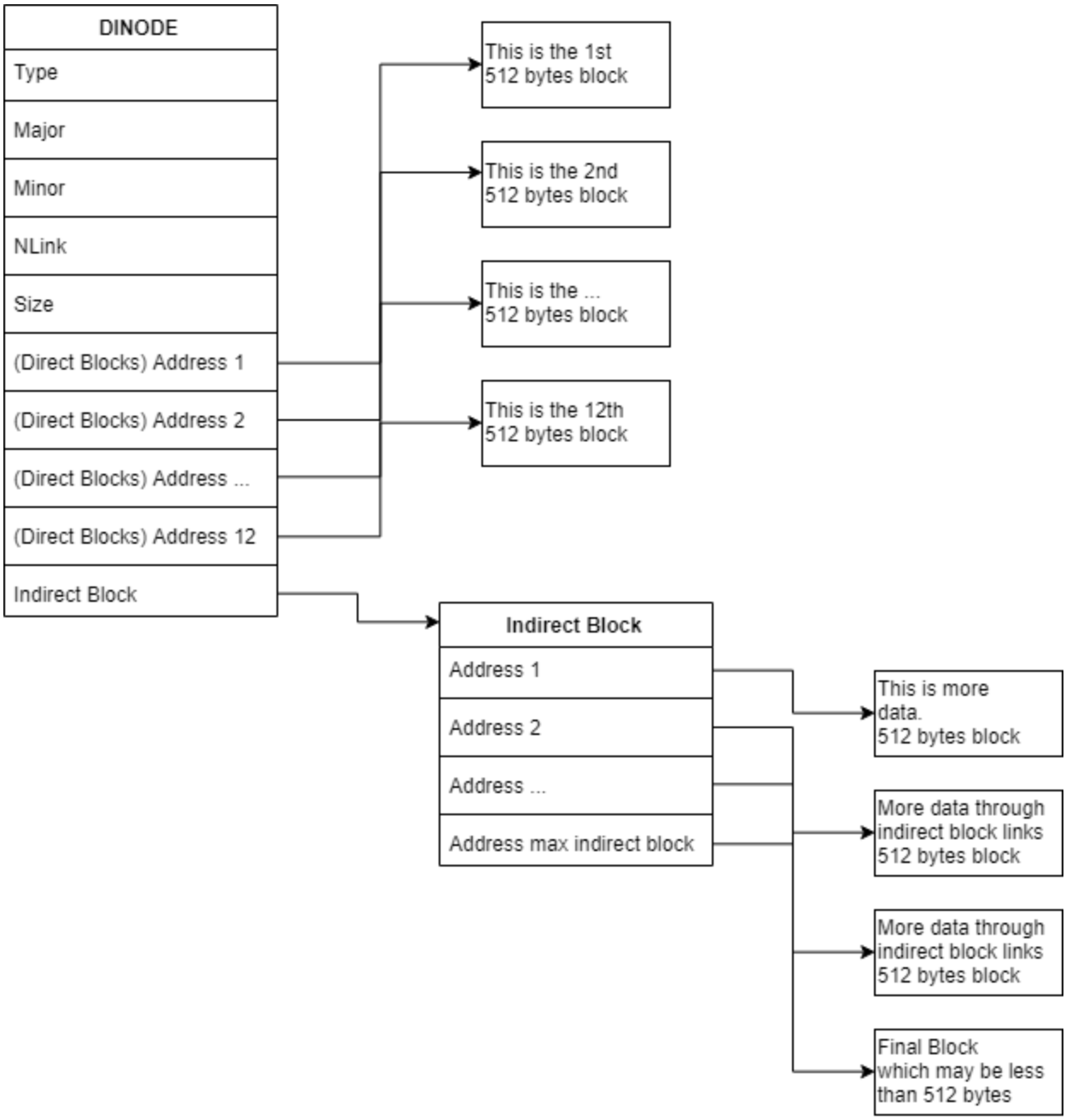
The ***sys_read*** function in ***sysfile.c*** returns the result from the ***fileread*** function which takes the aforementioned arguments, to retrieve the correct file contents. Specifically the file descriptor argument is then used with ***argfd*** function to identify the ***file*** struct associated with that file descriptor. The *file* struct contains various attributes such as ***readable*** or ***writable***, the ***offset*** and more.

An important attribute is the ***offset*** as it is used to enable the operating system to know where to continue reading from on the next iteration of the file reading, to prevent reading and displaying the same block of data multiple times.

Further into the code shows ***Readi*** function which utilises an inode number and a block to be read. This block is identified using the ***bmap*** function which returns the block number of the n-th block of an inode by traversing the direct and indirect blocks. And in the case the n-th block doesn't exist, it allocates a new block and stores it's address in the inode and returns it. The specified block is retrieved and returned from the block layer through the

IDE controller using ***bread*** function. However some complications can arise when the file to be read is larger than the block structure, this then requires the usage of indirect or doubly indirect blocks that increase the amount of accessible/usable blocks for data storage.

As seen in the diagram below outlining the block structure in the case of a large file and the usage of indirect blocks.



Question 2:

Part A:

In the XV6 operating system the IDE device driver enables access to the disk. A driver is a set of instructions for a specific device found in an operating system that controls the configurations, enabling the generation and subsequent processing of interrupts.

When conducting operations on the disk such as reading data or writing the operating system needs to know when the disk is ready, this can be done naively with continuous polling however this introduces a degree of redundant processing.

To combat this redundancy and to increase efficiency interrupts were implemented into operating systems. In the case of the I/O operations in XV6, the OS waits until the drive is ready, this is communicated through the IDE driver. The driver checks for the read status register to be "***Ready***" before continuing; the implementation of this waiting function can be found in the ***idewait***.

The IDE driver then can send a request to the disk, which may be queued if other requests are currently pending processing, the driver subsequently waits for completion of the operation requests in the meantime the process that called the I/O operation is put to sleep. This is accomplished using the ***iderw*** function, which appends a block to the ***idequeue***, the actual I/O function will then be called when that block reaches the head of the queue.

This procedure utilises a loop and locks to wait until the I/O operation has completed, the interrupt handler will then wake the sleeping processes once the request has finished.

The interrupts are generated by the IDE driver, which fill specific sectors using the ***outb*** function. Such interrupts must also be handled, this is distinguished by the operation itself, a write operation requires the ***idestart*** function to notify the interrupt handler upon completion, and in the case of a read operation the interrupt signals that the data is ready to be read.

Part B:

The bootloader is a portion of instructions that are used to load the kernel into memory, found in the boot sector of memory. These instructions are loaded using a Basic Input/Output System or BIOS which firstly prepares hardware and secondly transfers control code from the boot sector of the disk. In the case of the XV6 operating system the bootloader is found in bootmain.c or in bootasm.

By analysing the code found in **IDE.c** and **Bootmain.c** we can identify how the bootloader interacts with the IDE controller.

More specifically we can see that the bootloader looks for the kernel executable, which is expected to be in **ELF** format, in the C code we can see the **readseg** function reading from the first **4096 bytes** of the disk, which is the expected location for the **ELF**. Once that segment has been read it must be validated to ensure it's a true **ELF** executable.

Subsequently we can begin to see where the interaction between the bootloader and the IDE takes place, each program segment must be read from the disk and loaded, this is accomplished using the **readseg** function, which in turn calls the **readsect** function.

Inside **readsect** on **line 60** we can see commands that are familiar from our previous analysis of the **IDE.c** code, functions such as the internal workings of **waitdisk** and **outb.**

Waitdisk blocks and checks the read status register to ensure that the disk is ready for I/O operations. This is accomplished by checking the values at the address (**0x1F7**) for the values "**READY**" or "**BUSY**". Once the disk is ready the operations of bootmain.c can continue.

Provided that the IDE controller has communicated to the bootloader that the disk is ready, the bootloader will run the **outb** function with a number of addresses and offsets that correspond to registers, which will subsequently be filled with the sector number on the disk. Finally the **waitdisk** function is called once again to ensure the disk is ready to be read, and this time we use the **insl** function, which is actually an inline assembling function similar to **outb** (both found in **x86.h**) to read the contents of the registers and therefore the sectors in memory using a specified destination and sector size. This concludes the interactions between the IDE controller and the bootloader in **bootmain.c**.