

MONASH UNIVERSITY

INFORMATION TECHNOLOGY

Clayton School



FIT2022 Computer Systems 2 (Moodle)

AJH

@murtoa.local

[Assessment](#) | [Contacts](#) | [Laboratories](#) | [Lectures](#) | [Resources](#) | [Timetables](#) | [Tutorials](#) | [Unit Outline](#)

Last modified: 20080812:140306/minor fixes

[FIT2022 AJH-2008-26](#)

FIT2022 Computer Systems 2

Laboratory Session 3

[Objectives and Outcomes](#) | [Introduction](#) | [The Disk Subsystem](#) | [The File Subsystem](#) | [Contiguous Allocation](#) | [Indexed vs. Contiguous Allocation](#)

Introduction to Disks and Files

SVN server address: <http://svnte.infotech.monash.edu.au/svn/fit2022-labs>

1. Objectives and Outcomes

1.1 Objectives

1. to model the implementation of a disk system;
2. to model the implementation of a file system;
3. to understand the underlying layers of a file system;
4. to understand the interface between operating system level file system calls and the underlying implementation;
5. to follow a series of models (not necessarily of a file system), each improving upon the previous one.

1.2 Outcomes

At the end of this lab session, you should:

1. understand how a file system is implemented;
2. understand the interface between the operating system and the underlying file system hardware;
3. understand how the different file allocation methods are implemented;
4. understand some of the trade-offs in OS design;
5. have more familiarity with low level computer data.

Your demonstrator may ask you questions about these points, and will only give you a "satisfactory" for the lab if you can respond correctly.

2. Introduction

You will doubtless be aware of modern file systems, as realised in typical operating systems. File systems provide resource management of storage space, using a model that is directed at the user, rather than the underlying system characteristics. A user can store information in files, which are managed by the operating system into read and write requests onto the disk subsystem.

In this laboratory, we build a **model file system**, including the underlying disk subsystem: a disk with n blocks (or sectors) of k bytes (here $n = 1024$ and $k = 256$).

In the second part, the file system is modelled after a flat single-level directory system, which includes directories and data files, using index linked allocation.

The user interface will include routines to perform file open, close, read, write, as well as file creation and deletion. You can download the two files generated by this documents: [disk_sys.py](#) and [file_sys.py](#) to use in the following exercises.

3. The Disk Subsystem

We shall build our disk subsystem as a homogeneous array of blocks, each of k bytes in length. Get a copy of [disk_sys.py](#), and follow the code as you read the following sections.

"[disk_sys.py](#)" 3.1 =

```
import string
<block class definition 3.2>
<disk class definition 3.10>
```

There are two main parts to the disk subsystem: a class that describes the disk data block structure, and a class that describes the disk operation itself.

3.1 The block class

First we define the block class. Each instance of a block (a block object) represents one disk block.

<block class definition 3.2> =

```
class block:
    BlockSize = 256
    <block: class definitions 3.3,3.4,3.5,3.6,3.7,3.8>
Chunk referenced in 3.1
```

Compile in the block size as a fixed constant, 256 bytes per block.

<block: class definitions 3.3> =

```
def __init__(self):
    self.nbytes = self.BlockSize
    self.bytes = []
    self.bytes[0:self.nbytes] = self.nbytes*[0]
Chunk referenced in 3.2
Chunk defined in 3.3,3.4,3.5,3.6,3.7,3.8
```

Initialization function for block instantiation. Set the number of bytes, and initialize the array to contain exactly this number of zeros. Note the notation $n*[0]$, which in Python creates a list of n copies of 0 (NB, you should be careful to distinguish this from n copies of the single element list $[0]$).

<block: class definitions 3.4> =

```
def getbyte(self,n):
    if n < self.nbytes:
        return self.bytes[n]
    else:
        print "*** I/O error: byte adr (%02x) out of range" % (n)
        return -1
Chunk referenced in 3.2
Chunk defined in 3.3,3.4,3.5,3.6,3.7,3.8
```

Function to return the byte at the given address n . The address is checked to ensure that it is within the range of byte addresses within a block. Return a negative value if adr is not in range. (In general, we use negative return values to indicate error. In practice, we might encode more meaning into this negative value, but here we keep it simple and just use -1.)

<block: class definitions 3.5> =

```
def setbyte(self,n,b):
    if n < self.nbytes:
        self.bytes[n]=b
    else:
        print "*** I/O error: byte adr (%02x) out of range" % (n)
        return -1
```

Chunk referenced in [3.2](#)

Chunk defined in [3.3,3.4,3.5,3.6,3.7,3.8](#)

Function to set the byte at the given address `n` to the value `b`. Again, the address is checked to ensure that it is within the range of byte addresses within a block.

<block: class definitions 3.6> =

```
def getword(self,n):
    b1 = self.getbyte(n); b2 = self.getbyte(n+1)
    return b1 << 8 ^ b2
```

Chunk referenced in [3.2](#)

Chunk defined in [3.3,3.4,3.5,3.6,3.7,3.8](#)

See below.


<block: class definitions 3.7> =

```
def setword(self,n,w):
    b1 = w >> 8; b2 = w & 0xff
    self.setbyte(n,b1); self.setbyte(n+1,b2)
    return
```

Chunk referenced in [3.2](#)

Chunk defined in [3.3,3.4,3.5,3.6,3.7,3.8](#)

Two routines to get and set 16 bit quantities from a block. `n` is assumed to be a byte address, and does not point to the last byte in a block (or an address check error is forced).

 **Warning:** This exercise should be completed only after all other basic sections have been completed, and are working satisfactorily.

Revise `getword/setword` to return an error if `n>=nbytes-1`.

Now read on: 

<block: class definitions 3.8> =

```
def __str__(self):
    printable = "....." + \
        "!\"#$%&'()*+,-./0123456789:;<=>?" + \
        "@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\]^_" + \
        "`abcdefghijklmnopqrstuvwxy{|}~." + \
        "....." + \
        "!\"#$%&'()*+,-./0123456789:;<=>?" + \
        "@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\]^_" + \
        "`abcdefghijklmnopqrstuvwxy{|}~."
    res = ""; i = 0
    while i < self.nbytes:
        if i % 16 == 0:
            res = res + ("%02x: " % (i))
            hex = ""; str = ""
        res=res + ( "%02x%02x " % (self.bytes[i],self.bytes[i+1]))
        chr1 = chr(self.bytes[i]); chr2 = chr(self.bytes[i+1])
        chr1 = string.translate(chr1,printable)
        chr2 = string.translate(chr2,printable)
        str = str + chr1 + chr2
        i = i + 2
        if i % 16 == 0:
            res = res + hex + str + "\n"
            hex = ""; str = ""
    return res + str
```

Chunk referenced in [3.2](#)

Chunk defined in [3.3,3.4,3.5,3.6,3.7,3.8](#)

Generate a printable representation of a block. There are three parts to this representation:

1. The address within the block, in hexadecimal
2. The hexadecimal data of the block, 16 bytes at a time.
3. The ascii representation of the data in the block, 16 characters at a time. Unprintable characters are represented as "." characters.

Example:

```
00: 0000 0000 0000 0000 0000 0000 0000 0000 .....
10: 0000 0000 0000 002a 0000 0000 0000 0000 .....*.....
20: 0000 0000 0000 0000 0000 0048 656c 6c6f .....Hello
30: 0000 0000 0000 0000 0000 0000 0000 0000 .....
40: 0000 0000 0000 0000 0000 0000 0000 0000 .....
50: 0000 0000 0000 0000 0000 0000 0000 0000 .....
60: 0000 0000 0000 0000 0000 0000 0000 0000 .....
70: 0000 0000 0000 0000 0000 0000 0000 0000 .....
80: 0000 0000 0000 0000 0000 0000 0000 0000 .....
90: 0000 0000 0000 0000 0000 0000 0000 0000 .....
a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

Exercise 1

Run the test program `test1.py` that imports the disk subsystem, creates a block, and writes a various sequence of bytes to the block, then prints the block. Read through the code to make sure you follow what is happening.

"test1.py" 3.9 =

```
import disk_sys
nblock = disk_sys.block()
nblock.setbyte(0x2b,0x48)
nblock.setbyte(0x2c,0x65)
nblock.setbyte(0x2d,0x6c)
nblock.setbyte(0x2e,0x6c)
nblock.setbyte(0x2f,0x6f)
print nblock
```

Exercise 2

Try modifying the code to generate your own short string (the variable `printable` might help you, particularly if you know that there are 32 (or 0x20) characters per line).

Exercise 3

Write a test program `test.py` that imports the disk subsystem, creates a block, and writes the sequence of bytes 87,101,108,108,32,100,111,110,101,33 (in decimal) to addresses 62 and following. Call `print <block>` on your updated block to see the result.

3.2 The disk class

<disk class definition 3.10> =

```
class disk:
    DiskSize = 1024
    <disk: class definitions 3.11,3.12,3.13>
```

Chunk referenced in [3.1](#)

Define our disk system to contain 1024 disk blocks, as indicated by the constant `DiskSize`.

<disk: class definitions 3.11> =

```
def __init__(self):
    self.nblocks = self.DiskSize
```

```
self.blocks = []
for i in range(self.nblocks):
    self.blocks.append(block())
```

Chunk referenced in [3.10](#)

Chunk defined in [3.11,3.12,3.13](#)

Create a simulated disk with **DiskSize** blocks in it, initialized to zero. You need a loop to initialize all the blocks, as each block must be separately instantiated.

<disk: class definitions 3.12> =

```
def getblockref(self,n):
    if n < self.nblocks:
        return self.blocks[n]
    else:
        print "*** I/O error: block adr (%04x) out of range" % (n)
        return -1
```

Chunk referenced in [3.10](#)

Chunk defined in [3.11,3.12,3.13](#)

Address a block on the disk at specified block number and return (a reference to the) block as result. Note that because we return a reference to the block, there is no need to provide an equivalent setblockref routine.

<disk: class definitions 3.13> =

```
def block2str(self,n):
    return "Disk block %03x\n%s" % (n,self.getblockref(n))
```

Chunk referenced in [3.10](#)

Chunk defined in [3.11,3.12,3.13](#)

Create a printable representation of a specified block number.

Exercise 4

Use the test program [test2.py](#) to create and test a disk object. Make sure you understand what is happening. Hint: python uses *reference semantics*. What does this mean? Your tutor can explain if you are not sure

"[test2.py](#)" 3.14 =

```
import disk_sys
mydisk = disk_sys.disk()
print mydisk.getblockref(13)
b = mydisk.getblockref(13)
b.setbyte(0x2b,0x48)
b.setbyte(0x2c,0x65)
b.setbyte(0x2d,0x6c)
b.setbyte(0x2e,0x6c)
b.setbyte(0x2f,0x6f)
print mydisk.getblockref(13)
```

4. The File Subsystem

What sort of interface to the file system do we want? That is, what routines are we going to define that provide file system functionality? Here is a first pass at defining those routines:

- a routine to initialize a directory structure in a raw disk system (this is like an **mkfs** command in Unix).
- a routine to create a file located in a given directory
- a routine to open a file
- a routine to read from a file
- a routine to write to a file
- a routine to close a file

"[file_sys.py](#)" 4.1 =

```
import disk_sys
```

```

class file:
  <file: initialization routine 4.5,4.6>
  <file: create routine 4.7>
  <file: open routine 4.12>
  <file: read routine 4.20,4.21> **** Chunk omitted!
  <file: write routine 4.15,4.18>
  <file: close routine 4.22>
  <file: housekeeping routines 4.2,4.3,4.4,4.13>

```

4.1 What is the directory structure?

A directory defines what files are accessible within the file system. Generally, the directory stores information - such as name, location, size and type - for all files on a particular disk partition. Some of the operations that are to be performed on a directory include searching for a file, creating a file, deleting a file, listing a directory, renaming a file and traversing the file system.

For this prac, we will assume that a directory is a sequence of entries, each (conceptually) containing two key pieces of information: the identity of a file (synonymous for the moment with file name), and the location of the file. These need to be mapped into byte sequences for storing in disk blocks.

4.2 Allocating disk space

The flexibility in the implementation of files is enabled by the direct-access nature of disks. With so many files stored on the same disk, the main issue to be concerned with is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.

We propose several models (the reader may like to think of others) for the file and disk information:

File Name Length

- fixed length
- arbitrary length

Directory Structures

- single-level directory
- two-level directory
- tree structured directory
- acyclic-graph directory
- general graph directory

Allocation Method

- contiguous
- linked
- indexed

The first part of this section will be individual work, we will implement a file system with fixed length names, flat single level directory structure, with indexed allocation. In the next part, as a group, you will be required to implement a file system also with fixed length names with the same flat single level directory structure, but with the contiguous allocation method.

4.3 Directory structure

Since there are 1024 disk blocks on a disk, we require 10 bits to represent a block pointer. We will be generous and allocate 16! This means that we can store 128 block addresses in a single block. (A block address will need two bytes anyway. We could squash block pointers up, and get $256 * 8 / 10 = 204.8$ adrs per block, but it is hardly worth the trouble.)

We must also decide on the format of a directory entry. Given that there are a fixed number of characters to represent the directory name, we can choose say 12, with 4 bytes left over to make a 16 byte entry, one line of the block dump format. Such choices are arbitrary, and demonstrate the variability possible in file system design.

But beware! Choices like this often come back to haunt the designer, and are usually the reason why file systems must subsequently be modified as technology advances. It is the reason why, for example, Windows FAT16 file systems are limited to 2Gb in total size (see for example, [Converting your hard disk to the FAT32 file systems](#) and [Scalability and Performance in Modern File Systems](#)).

So our directory structure will look like this, and we (somewhat arbitrarily) choose to store this at block 0 on the disk. In practice, block 0 would be used for bootstrap purposes, and the root of the file system would be located at block 1 (or higher).

```
00: rrrr bbbb NNNN NNNN NNNN NNNN NNNN NNNN ....File Name 00
10: rrrr bbbb NNNN NNNN NNNN NNNN NNNN NNNN ....File Name 01
20: rrrr bbbb NNNN NNNN NNNN NNNN NNNN NNNN ....File Name 02
30: rrrr bbbb NNNN NNNN NNNN NNNN NNNN NNNN ....File Name 03
```

where **rrrr** is reserved for future use, **bbbb** is a two byte block address, and **NN...NN** is a 12 character file name (with blank fill). With a block size of 256 bytes, this restricts us to just 16 directory entries!!

Since block 0 is reserved for the directory table, we use a block address of 0 in the directory entry to indicate that the corresponding file does not exist, and that the directory entry is free for use. If the block address is zero, the value of the filename field is irrelevant. We assume nothing about the ordering of allocation of files to directory entries, using a first available strategy for allocation.

4.4 Free blocks

Now we must tackle another issue. How do we allocate disk blocks to particular files? We will need to keep track of which blocks are in use, and which ones are available or free. To do this, we implement a free block bitmap, which has a single bit allocated for each block in the disk, which is 0 if the block is free, and a 1 if in use.

<file: [housekeeping routines 4.2](#)> =

```
def allocateblock(self,n):
    nbyte = n / 8; nbit = n % 8
    fb = self.freebitmap
    byte = fb.getbyte(nbyte)
    mask = 1 << (7-nbit); clearmask = ~ mask
    byte = byte ^ mask # set the relevant bit
    byte = byte & 255 # for safety
    fb.setbyte(nbyte,byte)
```

Chunk referenced in [4.1](#)

Chunk defined in [4.2,4.3,4.4,4.13](#)

allocateblock reserves block numbered n to be in use. We read the appropriate byte from the bitmap, and "or" in the mask, which has a 1 shifted into the appropriate position. Note that the shift has to shift the mask into the high position (7 bit shift) when the bit number is zero, counting from the msb of the byte.

Exercise 5

On a sheet of paper, write the values (in binary and hexadecimal) of the key variables **nbyte**, **nbit**, **mask**, and **clearmask** for the following calls to **allocateblock**:

```
allocateblock(3)
allocateblock(7)
allocateblock(25)
```

<file: [housekeeping routines 4.3](#)> =

```
def freeblock(self,n):
    nbyte = n / 8; nbit = n % 8
    fb = self.freebitmap
```

```

byte = fb.getbyte(nbyte)
mask = 1 << (7-nbit); clearmask = ~ mask
byte = byte & clearmask # reset the relevant bit
byte = byte & 255 # for safety
fb.setbyte(nbyte,byte)

```

Chunk referenced in [4.1](#)

Chunk defined in [4.2,4.3,4.4,4.13](#)

freeblock frees block numbered n to be no longer in use. Note how it performs the complementary operation to **allocateblock**, using an **and** instead of an **or** (Why?).

<file: [housekeeping routines 4.4](#)> =

```

def getfreeblock(self):
    fb = self.freebitmap
    for i in range(disk_sys.block.BlockSize):
        byte = fb.getbyte(i)
        if byte < 255:
            break
    else:
        print "*** I/O error: No free disk blocks"
    byteadr = i
    nbyte = (~ byte) & 255; mask = 128; i = i*8
    # find first 1 bit in nbyte from msb
    while nbyte & mask == 0:
        mask = mask >> 1
        i = i + 1
    byte = byte ^ mask # turn bit on
    byte = byte & 255 # for safety
    fb.setbyte(byteadr,byte)
    return i

```

Chunk referenced in [4.1](#)

Chunk defined in [4.2,4.3,4.4,4.13](#)

getfreeblock returns the block adr of a free block in the disk subsystem.

Note that as block 1 is used to store the disk free block bitmap, it must itself be reserved in that bitmap.

4.5 File system creation

Now we can write the file initialization routines. There are two routines. The first is for object initialization, which creates the underlying disk structure. The second routine models the **nkfs** routine of Unix, setting up the initial directory block and free block bitmap. It must ensure that at least every block address in the directory is zeroed, and that the first two blocks in the disk subsystem are marked as allocated.

<file: [initialization routine 4.5](#)> =

```

def __init__(self):
    self.disk = disk_sys.disk()

```

Chunk referenced in [4.1](#)

Chunk defined in [4.5,4.6](#)

Creating the file system requires that we have a disk subsystem upon which it is built.

<file: [initialization routine 4.6](#)> =

```

def initialize(self):
    self.dirblock = self.disk.getblockref(0)
    self.freebitmap = self.disk.getblockref(1)
    self.freebitmap.setbyte(0,0xc0) # allocate first 2 blocks for file system
    for i in range(0,disk_sys.block.BlockSize,16): # count in 16 byte increments
        self.dirblock.setbyte(i+2,0)
        self.dirblock.setbyte(i+3,0)

```

Chunk referenced in [4.1](#)

Chunk defined in [4.5,4.6](#)

Note that **dirblock** is a reference to the directory block (block 0), and that **freebitmap** is a reference to the free block bitmap (block 1). We just use the references as a convenient way of referring permanently to these blocks. Note that these two blocks are themselves reserved in the free bitmap.

4.6 Create a file

<file: create routine 4.7> =

```
def create(self, fname):
    <file: find a free directory entry 4.8>
    <file: record the file name 4.9>
    <file: create an index block and record it 4.10>
    return 0
```

Chunk referenced in [4.1](#)

Creating a file requires firstly finding a free directory entry, saving the file name, and allocating space for the file (index table).

<file: find a free directory entry 4.8> =

```
self.dirblock = self.disk.getblockref(0)
for free in range(0, disk_sys.block.BlockSize, 16):
    pnter = self.dirblock.getword(free+2)
    if pnter == 0:
        break
else:
    print "*** I/O error: no free directory entries"
    return -1
```

Chunk referenced in [4.7](#)

Scan through the directory, looking for zero index block pointers. If none exist, we are in trouble, and the file directory is full. so no more files can be created.

<file: record the file name 4.9> =

```
for i in range(len(fname)):
    b = ord(fname[i])
    self.dirblock.setbyte(free+i+4, b)
```

Chunk referenced in [4.7](#)

Write in the file name at the free location.

<file: create an index block and record it 4.10> =

```
index = self.getfreeblock()
indexblock = self.disk.getblockref(index)
for i in range(disk_sys.block.BlockSize):
    indexblock.setbyte(i, 0)
self.dirblock.setword(free+2, index)
```

Chunk referenced in [4.7](#)

We choose to implement fixed length names, flat single level directory, with indexed allocation. This means that the file *location* is defined by a pointer to a single disk block containing the addresses of all the constituent blocks. We must grab a free block for this index table before any file operations can occur.

Exercise 6

We now have developed enough of the system to try creating a few files. Download the file [test3.py](#) and run it. Extend it by creating additional files. Make sure you understand what happens to the directory block and free space block.

"[test3.py](#)" 4.11 =

```
import file_sys
f = file_sys.file()
f.initialize()
f.create('test')
b=f.disk
print b.block2str(0)
```

```
print b.block2str(1)
print b.block2str(2)
```

4.7 Opening a file

What does it mean to open a file? It is a signal to the operating system that you wish to interact with a particular file, and that data structures for operating on the file should be created, ready to use the file. The main data structure is called a *file descriptor*, which records information such as the address of the index table (called the *inode* in Unix terminology), and the current access position in the file.

We will follow the Unix model, and regard a file as a contiguous stream or sequence of bytes. This is of course, the *logical model*: the *physical reality* is that the bytes are scattered over whatever blocks have been allocated to the file.

We model the file descriptor as a Python list of **[indexblock,fileposition]**, returned by the open routine:

<file: open routine 4.12> =

```
def open(self, fname):
    f = self.searchdir(fname)
    if f < 0:
        print "*** I/O error: cannot open file %s" % fname
        return -1
    return [f,0]
```

Chunk referenced in [4.1](#)

Opening a file requires that we search the directory to find the corresponding file name. We call a convenient housekeeping routine to do the work, which returns a pointer to the index block or inode if the file is present in the directory, or -1 if it isn't.

<file: housekeeping routines 4.13> =

```
def searchdir(self, fname):
    self.dirblock = self.disk.getblockref(0)
    for direntry in range(0, disk_sys.block.BlockSize, 16):
        pntr = self.dirblock.getword(direntry+2)
        <file: check if dir entry valid and match: break out if so 4.14>
    else:
        print "*** I/O error: cannot find file %s" % fname
        return -1
    return pntr
```

Chunk referenced in [4.1](#)

Chunk defined in [4.2,4.3,4.4,4.13](#)

scan through the directory block, this time looking at non-zero index pointers. If we find none, or no match, then report failure.

<file: check if dir entry valid and match: break out if so 4.14> =

```
if pntr != 0:
    dirfn = ""
    for j in range(12):
        byte = self.dirblock.getbyte(direntry+4+j)
        if byte == 0:
            break
        dirfn = dirfn + chr(byte)
    if fname == dirfn:
        break
```

Chunk referenced in [4.13](#)

Collect filename from this entry, and compare to target. Break out of enclosing search loop if there is a match.

Exercise 7

Try opening one of the files you created in exercise 6, and also a file that you *haven't* created. Is this a sensible response? What might a more elaborate design do?

4.8 Writing to a file

How big is a file? One of the questions we haven't addressed yet is how much space to allocate to a file, and to do this requires some knowledge of how big the file may get.

Rather than reserve space *a priori* (Latin: meaning "before the fact or event"), we allocate no space initially to the file, other than its directory entry and index table. Then, as things are written to the file, we expand it dynamically to grow as necessary. This has the advantage (with the indexed organisation) that areas never written never use up space. A sparse file will only require a few disk blocks, even if it is conceptually megabytes in size.

Nevertheless, we still need to keep track of the logical size of a file, so that attempts to access beyond its size can be trapped. To this end, we record the maximum address ever written, known as the **end of file**. Where is this recorded? The most appropriate place is in the inode or index table. Hence we reserve the first two bytes of the inode to store the file size or end of file pointer. Two bytes limits us to files that are at most 65535 bytes in length, but since that represents a quarter of our disk system, we won't worry too much about that!

<file: write routine 4.15> =

```
def writebyte(self,fd,byte):
    byte = byte & 0xff # for safety
    index = fd[0]; posn = fd[1]
    bn = posn / disk_sys.block.BlockSize # block number from origin
    offset = posn % disk_sys.block.BlockSize # byte number within block
    indexblock = self.disk.getblockref(index)
    <file: get data block, allocate if necessary 4.16>
    datablock.setbyte(offset,byte)
    <file: update file size and file descriptor 4.17>
    return 0
```

Chunk referenced in [4.1](#)

Chunk defined in [4.15,4.18](#)

We work out where the data is to be written by computing a block number *bn* and offset within that block. This formed from the position field (from the file descriptor) by dividing it by the block size to get the block number, and using the remainder as the offset (see lecture slide [11.9](#)). We update the block with the data (remember the *reference semantics*!).

<file: get data block, allocate if necessary 4.16> =

```
indexadr = 2+2*bn
if indexadr >= disk_sys.block.BlockSize:
    print "*** I/O error: no space in index table"
    return -1
pntr = indexblock.getword(indexadr)
if pntr == 0:
    pntr = self.getfreeblock()
    indexblock.setword(indexadr,pntr)
datablock = self.disk.getblockref(pntr)
```

Chunk referenced in [4.15](#)

As described above, the index table is extended as necessary. If the block number pointer is zero, the required block has never been allocated, so we must do that first and update the index table. Then we can grab the necessary data block.

<file: update file size and file descriptor 4.17> =

```
posn = posn + 1
eof = indexblock.getword(0)
if posn > eof:
    if posn > 65535:
        print "*** I/O error: no space in file"
        return -1
    else:
        indexblock.setword(0,posn)
        fd[1] = posn
```

Chunk referenced in [4.15](#)

We keep track of the current position in the file and update the file descriptor accordingly. In addition, if this write extends the size of the file, we must update the end of file pointer. Since this is restricted to two bytes, there is an upper bound on

the file size.

<file: write routine 4.18> =

```
def writestr(self,fd,str):
    for i in range(len(str)):
        self.writebyte(fd,ord(str[i]))
```

Chunk referenced in [4.1](#)

Chunk defined in [4.15,4.18](#)

This routine simply extends the "user friendliness" of the interface. It takes a string of data and pumps it into the file at the current position. One might write a more efficient way of doing this in a real system!

Exercise 8

Download [test4.py](#) and run it. Add some more `writestr`s and experiment to see what data gets written to disk and where.

"test4.py" 4.19 =

```
import file_sys
f = file_sys.file()
f.initialize()
b=f.disk
print b.block2str(0)
f.create('test')
f.create('newfile')
f.create('thirdfile')
fd = f.open('newfile')
f.writestr(fd,'Hello Folks! Some data in a file')
fd = f.open('thirdfile')
f.writestr(fd,'This data in the third file')
print b.block2str(0)
print b.block2str(1)
print b.block2str(2)
print b.block2str(3)
print b.block2str(4)
print b.block2str(5)
print b.block2str(6)
```

4.9 Reading from a file

<file: read routine 4.20> =

```
def readbyte(fd): # argument list can be extended if required
    <get the index block via the file descriptor >
    <compute index into index block >
    <get block pointer from index block >
    if <block is allocated >:
        <retrieve block >
        byte=<get byte from retrieved block >
    else: # block never allocated, return zero
        byte=0
    return byte
```

Chunk referenced in [4.1](#)

Chunk defined in [4.20,4.21](#)

<file: read routine 4.21> =

```
def readstr(fd): # argument list can be extended if required
    return ""
```

Chunk referenced in [4.1](#)

Chunk defined in [4.20,4.21](#)

Exercise 9

Code up the routine `readbyte`, and test it using the `block2str` routine provided in `disk.py`.

You can also modify the routine or extend it.

Write the **readstr** routine yourself and test it. In modifying and writing the **readstr** routine yourself, you will need to address the design questions of what happens on End-Of-File, and how long a string should be read. (How long is a piece of string?)

4.10 Closing a file

In practice, closing a file means releasing any resources used by the requesting program. These usually the file descriptors used for accessing the file: here the file descriptor is represented by a single Python list, and we do not need to do anything specific to release that. Nevertheless, we define a close routine for future compatibility!

<file: close routine 4.22> =

```
def close(self):
    pass
```

Chunk referenced in [4.1](#)

4.11 Deleting a file

Exercise 10

Write a delete file interface. Using a test file similar to `test4.py`, delete one or two files, and look particularly at what happens to the directory block and free space block.

5. Contiguous Allocation

Group Task 1

The above code fragments are based on the index allocation method. Your group will now be required to re-implement the file subsystem `file_sys.py` based on an fixed file length name with the same directory structure but with the contiguous allocation method for the same underlying disk subsystem.

Since you have already gone through the first part of the development for a simulated file system with fixed length file names and index allocation method, now, you need to apply what you have learned to implement a file system with the contiguous allocation method.

Remember, as part of your group work, you will need to:

1. Setup up your SVN repository
2. Determine the work distributions - each group member's work allocation should be logged in the SVN repository. Please note that the group leader is in charge of maintaining the repository.
3. Also, take note of the group requirements as stated in your lab guidelines.

Once you have set up the SVN repository and determined the work distribution, you can get started on your work.

Exercise 11

Discuss within your group and run these experiments.

Explore what happens when you try and fill the simulated disk: do you hit the file size limit first, or the limit on the number of index blocks? Can you work this out from the code first? Then run an experiment to check your answer. You may need to run the experiment for your tutor as well if asked.

6. Indexed vs. Contiguous Allocation

Group Task 2

In the preceding sections, we have examined the directory management options and block-allocation, specifically the single-level indexed directory structure with both the indexed and contiguous allocation methods. Now, considering the two allocation methods implemented, we evaluate the disk performance and efficiency using the two differing allocation methods used.

For this evaluation, we consider two issues:

1. The number of disk accesses required to read a file
2. The total time required to read a file

For the purpose of this evaluation and simplicity (and since we have not specified the type of disk used), we will opt to ignore the seek time (the time required to position the read-write head to a location)

Given that there are 1024 blocks to a cylinder and the **rotational latency** (the time for the proper block to rotate under the head) is 0.0000167 seconds per block with a **transfer time** (the time required to transfer data to or from the disk) for one byte of 0.00000083 seconds, we can evaluate the performances based on the number of disk accesses and the time it takes to read a file.

Now, your group task is to implement a simple routine that will calculate the total time taken to read a file, using both the indexed and contiguous allocation methods. You will need to use the **readbyte()** routine to obtain the number of disk accesses and the number of bytes read. Based on this number, you can then calculate the total time taken to read a file. Remember, you will need to consider the distances between the blocks as well as the number of bytes read within a block.

Document History

20080812:140306 1.0.1 ajh minor fixes

20080806:133037 1.0.0 ajh initial version for 2008

This page maintained by John Hurst.

Copyright [Monash University Copyright Policy](#)

Generated at 20080812:1403 from an XML file modified on 20080812:1403

Maintainer use only; not generally accessible: [Local Server](#) [Work Server](#) [CSSE Server](#)

11 accesses since
06 Aug 2008

