COMP3911 Suggested Exercises #2a - Solutions

1)

a) To calculate an average 16Kbyte read:

Since there are

480 sectors of 512bytes on a track --> 240Kbytes/track

and

7200RPM --> 120 rev/sec --> 8.33msec for 1 revolution

therefore:

average seek + 0.5 rotation + data transfer

8 + (8.33 * 0.5) + (8.33 * 16/240) = 12.6msec

1000/12.6 = 79 operations/sec

b) For 64Kbytes, all is the same except the last term is:

8.33 * 64/240, so

8 + (8.33 * 0.5) + (8.33 * 64/240) = 14.4msec

1000/14.4 = 69 operations/sec

c) The I/O bandwidth is the amount of data (Kbytes) transferred/sec, so:

For 16Kbyte reads: 16 * 79 = 1264Kbytes/sec

For 64Kbyte reads: 64 * 69 = 4416Kbytes/sec

d) Replacing a 7.200RPM drive with a 10,000RPM drive:

10,000RPM --> 166.7 rev/sec --> 6msec/rev

So, 6.0 replaces 8.33 in the above formulaes

For 16Kbyte reads: 12.6 - 11.2 = 1.4

For 64Kbyte reads: 14.4 - 11.8 = 2.6

e) No, unless there is a RAM cache on the drive, read and write operations take the same amount of time.

2) File system reading performance will not require that any changes to the file system's metadata (structural information) be made, but will require that the appropriate data (and metadata, as required) be read into the buffer cache from disk for the file read to be completed.

On the other hand, writing only requires the data be copied to the buffer cache. Any writing to the disk that is done synchronously (during the file writing operation) is done for reliability purposes and depends on the file system's design and implementation. As such, the design and implementation of the file system will determine how long a file write operation takes.

3) a) First, figure out the data block #s for each file block #, from the FAT table, noting that they are numbered from 0 and actually start at Data Block #4.

Block of File:	0	1	2	3	4
Data Block #:	7	3	4	5	2
Raw Block#:	B	7	8	9	6 (data in above line, +4)

i) 1010/1024 = 0 rem 1010 --> 1024 - 1010 = 14 'i', then 55 - 14 = 41 left
--> 41 'b' from file block #1

ii) 4000/1024 = 3 rem 928 --> (1024 - 928) = 96 'f', then 200 - 96 = 104 left
--> 104 'g' from file block #4

iii)4400/1024 = 4 rem 304 --> (1024 - 304) = 720 however this block is the last block of the file we must calculate the number of bytes in this block: 5090 / 1024 = 4 rem 994. As only 994 bytes are valid in this block we load (994 - 304) = 690 'f' then EOF.

iv) 5060/1024 = 4 rem 964 --> (1024 - 964) = 60 'x'
written to end of file block 4 so physical block 2 has 964 'f' followed by 60 'x', 100 - 60 = 40 left
--> 40 'x' written at start of file block 5.
File block 5 is a new block, so it must be allocated from the list of "free" blocks -- there is only one, so we use data block A (physical block E) and write the remaining 40 'x' characters on top of the 'o's in that block.
FAT table entry #2 is updated to A; FAT table entry A is updated to EOF; the file size is updated to the new total size of 5160 bytes; and the modify time is updated

v) This location is 990 (6080 - 5090) bytes after the current end of file, so first off, 990 '\0' bytes must be appended on to the file.
5090 / 1024 = 4 rem 994 -> (1024 - 994) = 30 bytes of ' ' written over 'f' at the end of file block #4, (990 - 30) = 960 ' ' bytes left, to place in file block #5, which is a new block (as in the part iv), so we search and find free data block A (physical block E) and write the remaining 960 ' ' characters there.
FAT table entry #2 is updated to A; FAT table entry A is updated to EOF.

Now we can begin to write the supplied 'y' characters, however after writing (1024 - 960) = 64 'y' characters up to the end of block A, we need another block with (200 - 64) left. We search the FAT table for another free block, and find none. At this point, the device is full, so the write will fail, however before returning the file size is updated to 6 * 1024 = 6144 bytes and the file modify time is updated.

b) First, use the i-node and indirect block to translate the file block #s to data block #s, noting they are numbered from 0 and actually start at 3.

File Block#s	0	1	2	3	4	5
I-node blk#s	6	8	1	3		
Indir block#s					NUL	2
Data Block#	9	B	4	6	NUL	5

i) 1010/1024 = 0 rem 1010 --> (1024 - 1010) = 14 ’g’, 55 - 14 = 41 left
--> 41 ’i’ from file block 1, which is data block B

ii) 4000/1024 = 3 rem 928 --> (1024 - 928) = 96 ’d’, 200 - 96 = 104 left
--> 104 ’\0’ (bytes of 0x0, ie. all bits 0) since file block 4 is NULL and therefore a hole

iii)4400/1024 = 4 rem 304 --> (1024 - 304) = 720 ’\0’ (bytes of 0x0, ie. all bits 0) from hole, 2000 - 720 = 12
--> 980 ’c’, since file size 6100 - 5120 = 980

iv) 5060/1024 = 4 rem 964 --> (1024 - 964) = 60 ’x’ written to end of file block 4
since file block 4 is NULL, must allocate a free block
from Free Bitmap, bit 7, which implies block 7 (data block A) is free
set first byte of Free Block Bitmap to 0, set first entry of Indirect
Block (data block 7) to 7
--> write data block A with 964 ’\0’ (bytes of 0x0, ie. all bits 0) bytes followed by 60 ’x’ bytes
100 - 60 = 40 left
--> write 40 ’x’ at beginning of file block 5 (data block 5) so
data block 5 has 40 ’x’, 940 ’c’ and 44 bytes of anything
set modify time

v) 6080/1024 = 5 rem 960 --> (1024 - 960) = 64 ’y’ at end of data block 5 so
data block 5 has 960 ’c’ followed by 64 ’y’, 200 - 64 = 136 left
must allocate a new block, same as iv) except 7 is entered in third field
of Indirect block (data block 7)
--> write data block A with 136 ’y’ followed by anything
set file size to 6280 and update modify time

4) The first blocks will be allocated in the same cylinder group as the one the i-node resides in. Then, every K megabytes, it will switch to a different cylinder group with lots of free data blocks in it. This is done in an effort to avoid running cylinder groups out of free data blocks. (Once cylinder groups run out of free blocks, file system fragmentation occurs, since free blocks must be allocated to files from other cylinder groups all over the file system.)

5) i) Space is wasted by the internal fragment, which is the unused portion of the last block, because files aren't exact multiples of blocksize in size. The larger the size of chunk used for this fragment, the more space that will be wasted by any given file. Applied to all files, the total wasted space will be greater for a larger chunk size.

a) Uses a 16,384 byte chunk size
b) Uses a 16,384/8 = 2,048 byte chunk size
c) Uses a 1,024 byte chunk size

Therefore iii) wastes the least amount of space.

ii) a) will achieve the fastest average write performance, because it will be using a 16,384 byte write size.

iii) For b), a fragment of the last block might be allocated for the last bytes of the file, with the next fragment of the block allocated to the last bytes of a different file. When the file is grown for this case, the last data bytes in the fragment must be copied to a new block and the file system's metadata must be updated to recognize the first fragment of the old last block is now available. However, overall write performance for b) will be almost as fast as a).

6) a) The total number of blocks that can be allocated to a file is:

8 (in i-node) + 2048 (in indirect block) = 2056
Maximum file size = 2056 * 8192 = 16842752bytes

b) For a read of 1000 bytes at file offset 65000:

File block# = 65000 / 8192 = 7
File block# 7 is in the i-node
but 1000 bytes spills into File Block #8, which is in the indirect block
--> 1 - to read in the i-node
1 - to read in the indirect block
2 - to read in the two data blocks
---
4 in total

c) For a write of 500 bytes at file offset 67000 with the indirect block pointer NULL:

File block# = 67000 / 8192 = 8
and the 500 bytes are all in file block# 8
--> 1 - for the indirect block
1 - for the data block
---
2 in total

All block#s in the indirect block, except the first, would be set to NULL and all data bytes except the 500 written would be set to ' ' in the data block.

d) Since the indirect block was NULL, no data has been written past this write. So:

File size = 67000 + 500 = 67500 bytes

e) Data byte 66600 will be in a hole, so the value read would be ' '. (I meant to state in the question "after doing the write d)".)

f) Whatever block# was allocated to File block# 8 by d) from the free list.

7) No. Normally a file deletion marks all allocated data blocks FREE in the FAT table, but does not overwrite the data in the data blocks. (If, for security reasons, the data must be deleted, garbage bits must be written to the file before deletion of the file.)

8) a) FIFO

7	6	8
9	7	A
8	B	C
A	7	

b) LRU

7	6	8
B	9	C
A	A	
	8	
	7	

c) Belady's

7	6	8
B	9	7
C	A	

d) Use Bit (falling back to FIFO)

7	6	8
B	9	C
A	A	
	8	
	7	

9) Small pages will require many page faults to occur when paging in large programs; large pages will incur a greater waste as the pages at the ends of segments are half-full.

In many ways this is same dilemma seen by file systems where small blocks incur less waste but cost more time in disk seeks. The main difference is that the "fragment" scheme in UFS disk blocks cannot have any similar structure in pages as the pages must reflect the organization of RAM; the O/S is not free to arbitrarily begin grouping parts of RAM together simply to reduce the number of partially full pages -- as the pages are accessed through the MMU, the page organization must directly reflect what the MMU expects to see, and fragments are not part of this scheme.

10) If a "lot" of swapping (or indeed any swapping) is occurring on a regular basis, the machine desperately needs more RAM. Swapping is a desperation measure, so if a "lot" of swapping occurs, the available RAM is drastically over-used (the machine is said to be "thrashing").

If the machine is doing a "lot" of paging (ie; if a lot of page faults are being observed), more RAM will also help, although this is a much less dramatic situation than if several events involving whole-process swapping are occurring.

11) Assuming we have the following variables:

```
short num;
long value;
char name[9];
```

they can be loaded into a string to be sent as a packet using the following code fragment:

```
{
    short netNum;
    long netValue;
    char packetBuffer[15];

    netNum = htons(num);
    netValue = htonl(value);

    /** sizeof(short) == 2 */
    bcopy(netNum, &packetBuffer[0], sizeof(short));

    /** sizeof(long) == 4 */
    bcopy(netValue, &packetBuffer[2], sizeof(short));

    bcopy(name, &packetBuffer[6], 9);
}
```

at this point, the contents of the buffer `packetBuffer` can be sent as a UDP style packet using sendto(), or on a TCP packet based connection using send() or write().

12) Multiprocessing assumes that requests for disk I/O from different programs may overlap in time. As it is critically important to maintain filesystem integrety, semaphores are used to ensure that only one program has access to shared filesystem structures (such as the free-block list) at a time. This prevents the filesystem corruption which would occur because of race conditions arising through multiple programs writing to the file system at the same time.

13)

a) Using the flags `O_EXCL` and `O_CREAT` together allows a program to create a file if and only if no such file by that name already exists in the filesystem. The successful creation of such a file could indicate the acquisition of a "lock", where any programs coming late to create the lock will have their `open()` call fail as the file already exists. Assuming that the file is removed when the lock is to be cleared (using the `unlink()` function or similar), then a single program waiting to acquire the lock could then succeed in opening the file. This assumes that all waiting processes are "spinning" and trying to open the file:

```
int lockfd;

while ((lockfd = open(LOCK_FILENAME, O_CREAT|O_EXCL, 0)) < 0)
    ;       /** spin, waiting until file can be opened */
```

While this may result in significant CPU load, it will achieve lock functionality.

b) NFS file systems introduce network latency and potential packet loss into the equation. In such an environment it is *much* more difficult to ensure that a single remote program acquires the lock in a first-writer-wins manner.

This is compounded by the fact that the NFS server is designed to function in a "stateless" fashion, with no memory on the server devoted to an attempt to track various mounting machines.

Together, this means that "filesystem locks" cannot be depended on when functioning over NFS. Much effort has been made to create "NFS lock servers", but historically these have only worked when the client and server are running exactly the same version of the NFS code -- a task which is essentially impossible in the heterogeneous O/S environment that defines any real-world network.

14)

a) A DISK_BLOCKSIZE buffer (4k) is allocated in the kernel. For the first write (1k, less than the size of this buffer), the buffer contents are first read from disk, so that the remaining 3k have the correct contents.

If "write-behind" is used, this buffer remains in the kernel cache for some time, otherwise it is written to the disk.

As the second, third and fourth writes to the file from user space occur, the remainder of the kernel buffer is filled. Each will cause a write to the disk to occur, unless "write behind" is in effect.

The fifth kilobyte of the file will cause a second buffer allocation, again with its contents initially loaded from the disk surface. For each group of 4 user writes, a single buffer is used.

In total, we will have 16k / 4096 = 4 rem 0 kernel buffer used. For each kernel buffer there will be 1 initial read, and 4 writes, for a total of 4 reads and 16 writes. If write-behind is used, this will be reduced to 1 initial read and 1 write per disk block, for a total of 4 reads and 4 writes.

b) To accomodate the first user write, 4 DISK_BLOCKSIZE buffers are allocated in the kernel. As the first user write exactly fills all 4 buffers, none of these need to be initialized with any data from the disk. After being loaded from the user data, these can be written to disk using 4 separate write operations.

The second, third and fourth user data operations will be identical to the first, while using separate buffers.

In total, 0 reads need be done, and (4 * 4) = 16 write operations of 1kb each. In this example, "write behind" has no effect as each file block is only accessed in a single user write operation.

c) For each of 32 disk blocks, the initial 1 byte write will cause a read into a 512 byte kernel buffer, followed by the storage of the first byte, and then a write (again, assuming no write-behind). The subsequent 511 bytes falling into the same block will simply cause an update of the existing kernel buffer and then a write (still assuming no write-behind).

In total, this will cause (32 * 1) = 32 reads plus (32 * 512) = 16384 writes. If write-behind is used, this value drops to 32 reads and 32 writes.

d) Certainly the "read" overhead can be completely removed by planning on the user’s part. For this reason, b) is more "efficient" as no reads need be done.

Note that only if write-behind is used is a) equivalent to b). Seeing as the choice of both blocksize and use of write-behind is not something which will be available to a user program, this shows that by batching disk writes into chunks of complete blocks increases I/O efficiency.

15) Telnet’s "line at a time" mode dates from a time when network communication speeds were low enough that efficiency in the packet data was important. The use of "line at a time" mode simply means that a single packet containing an entire line of text would be transmitted from telnet client to server -- this packet would then be more "full" than a packet with one character. As the minimum packet size on networks such as Ethernet are stipulated (Ethernet minimum = 46 byte payload), "byte at a time" packets travel around largely empty, taking up network bandwidth. On a slow network, the arrival of the entire line at one time could therefore give the user a more pleasant, and faster remote programming experience.