

## COS214 Tutorial 8

Roberto Togneri, 2000

### **SOLUTIONS**

1. (a) Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?
- (b) Fragmentation on a storage device could be eliminated by re-compaction of the information. Typical disk devices do not have relocation or base registers (such as are used when memory is to be compacted), so how can we relocate files? Give reasons why re-compacting and relocation of files often are avoided.

#### Answer

- (a) *Contiguous* – if file is relatively small or never updated (i.e. write once, read only!).  
*Linked* – if file is large and usually accessed sequentially.  
*Indexed* – if file is large and usually accessed randomly.
- (b) Relocation of files on secondary storage involves considerable overhead —data blocks would have to be read into main memory and written back out to their new locations. Relocation registers only make sense for contiguous allocation files, and most file-systems do not use contiguous allocation. Furthermore, many new files will not require contiguous disk space; even sequential access files can be allocated non-contiguous blocks if links between logically sequential blocks are maintained by the disk system. So fragmentation is not that much of a problem, especially with proper read and write caching.

2. Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), how is the logical-to-physical address mapping accomplished in this system? Assume a pointer size of 1 byte. (For the indexed allocation, also assume that a file is always less than 512 blocks long.)

#### Answer

Let  $Z$  be the logical address of the file.

*Contiguous.* Divide  $Z$  by 512 with  $X$  and  $Y$  the resulting quotient and remainder respectively. Add  $X$  to the first block number of the file,  $S$ , to obtain the required physical disk block,  $S + X$ .  $Y$  is the displacement into that block.

*Linked.* Divide  $Z$  by 511 (byte 0 is the pointer) with  $X$  and  $Y$  the resulting quotient and remainder respectively. Chase down the linked list, reading  $X + 1$  blocks, the last block read is the required physical disk block.  $Y + 1$  is the displacement into this block (byte 0 is the pointer).

*Indexed.* Divide  $Z$  by 512 with  $X$  and  $Y$  the resulting quotient and remainder respectively. The required physical block address is contained in location  $X$  of the index block.  $Y$  is the displacement into the desired physical block.

3. Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Describe the type of operations required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.
  - (a) The block is added at the beginning.
  - (b) The block is added at the end.

#### Answer

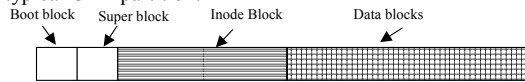
- (a) *Contiguous:* The 100 blocks have to be copied to another free partition in order to include the new block at the beginning.  
*Linked:* The head of the linked list is updated to include the new first data block.  
*Indexed:* The index entries have to be shifted down by one entry which may require copying to a new index. The new block address is added in as the first entry.
- (b) *Contiguous:* The block is simply appended to the last block.  
*Linked:* The linked list has to be read in its entirety to locate the last block and then the new block is added to the tail of the list.  
*Indexed:* The new block address is simply added to the index table.

4. Consider a system where free space is kept in a free-space list.
  - (a) Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
  - (b) Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

#### Answer

- (i) In order to reconstruct the free list, it would be necessary to perform “garbage collection.” This would entail searching the entire directory structure to determine which blocks are already allocated to files. Those remaining unallocated blocks could be relinked as the free-space list.
- (ii) The free-space list pointer could be stored on the disk, perhaps in several places.

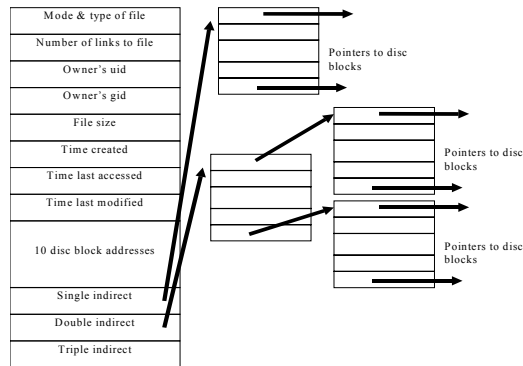
5. Consider a typical Unix partition:



The Unix directory entry looks like:



The other information is kept in the inode entry:



Assume disk blocks are 8K bytes and that disc addresses are 32 bits

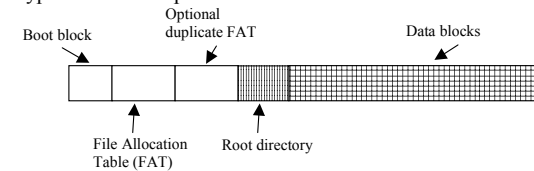
- How is a file opened? Use as an example the file `/usr/home/test.c`.
- What size of file can be directly addressed from the information in the inode?
- What size of file requires a double indirect block?
- What is the largest possible file?

**Answer**

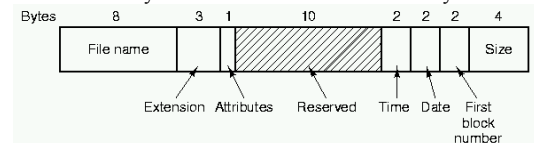
(a) First the root of the file system is located. In the filesystem the first block is reserved as the boot block, the next block is the superblock that describes the filesystem and then there are the blocks for the inodes (inode Block). Inode #1 is reserved for linking bad blocks to and inode #2 is reserved for the root of the file system. The inode #2 is used to locate the root directory data. The root directory is scanned for the "usr" entry which returns the "usr" inode number. The "usr" inode is retrieved from the Inode Block and the inode entry lists the data blocks for the "usr" directory file. These blocks are retrieved to form the "usr" directory. The "usr" directory is then scanned for the "home" entry which returns the "home" inode number. The same process is repeated to retrieve the "home" directory. The "home" directory is scanned for the "test.c" entry which returns the "test.c" inode number. The "test.c" inode is retrieved from the Inode Block and the inode entry is used to retrieve the data blocks for the "test.c" data file.

- disk blocks = 8k bytes; disk address = 32 bits  
→ 10 disk block address X 8k = 80k can be directly accessed.
- single indirect: 1x 8k block of 32 bit addresses  
→  $8 \times 1024 \text{ bytes} / 4 \text{ bytes} = 2048$  addresses of 8k blocks = 16M  
double indirect block:  $2048 \times 2048$  addresses of 8k blocks = 32G  
Can access a file of size no larger than 32G (+16M+80k)
- triple indirect block:  $2048 \times 32G = 64$  Terabytes (+ 32G+16M+80k) → largest file

6. Consider a typical MS-DOS partition:



Both regular files and directory files can be defined and located by the FAT. Each directory contains an entry for each file within that directory:



Clearly show how the FAT is used to retrieve the data from the file `c:\windows\system.ini`.

**Answer**

The Root directory is scanned for the "windows" directory entry. The "windows" directory entry contains the first block number of the "windows" directory file. The FAT is used to retrieve the "windows" directory data. The "windows" directory is then scanned for the "system.ini" directory entry. The "system.ini" directory entry contains the first block number of the "system.ini" data file. The FAT is then used to retrieve the "system.ini" data.

7. A file system checker has built up its two lists as shown below:

<b>Block #</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<b>In Use:</b>	1	0	1	0	0	1	1	1	1	0	2	0	0	1	0
<b>Free:</b>	0	0	0	1	1	1	0	0	0	1	0	1	2	0	1

Are there any errors? If so, are they serious? Why?

**Answer**

Errors:

- block #1 → not on free list, not used → just add block to free list, not serious
- block #5 → on free list, also used → remove from free list, not serious
- block #10 → used twice! that means there are two inodes referencing data block. If not a "hard linked" data block, this is serious since two files are sharing the same data and they shouldn't be → copy data block so each inode references a different data block, one or both files possibly corrupted and may have to be deleted
- block #12 → freed twice! this means the free list is corrupted or incorrect and will need to be rebuilt, this should not be serious.