Storage Management, part 2

Michael Brockway

November 10, 2014

Contents

References

- Operating System Concepts (8th Ed), Silberschatz et al, chapter 11. This lecture is based on the module text: you are recommended to read chapter 11.
- Wikipedia articles on FAT, EXT2, EXT3, NFS and Journalling File systems

File System Structure

A file system resides on *secondary storage* - disks, flash memory (USB sticks) etc

Each file is managed by a data structure called a *File Control Block* (FCB), containing

- permissions for the file
- dates/times (creation, last access, last modification)
- owner, group, who has access control
- size
- pointers to data block(s)

Directory Implementation

- linear list:
  - just list of file names with pointers to FCB
  - simple to implement
  - inefficient in practice
- hash table: a linear list with a *hash* data structure
  - A *collision resistant hash function* maps each file name to a FCB/data location
  - needs a strategy for resolving collisions
  - efficient when collision frequency is low
  - will meet hash tables in another module: meanwhile look at the wikipedia article!

Allocation Methods

How is data storage space allocated to a file?

- Contiguous allocation: allocate required number of bytes starting immediately after allocation to previous file
- Linked allocation: allocate in smallish chunks, linked together in a *linked list*
- Indexed allocation: see below.

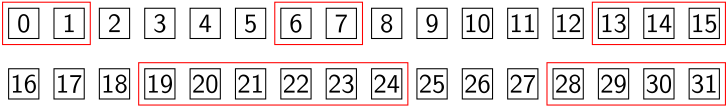
Space is divided into *blocks* for purposes of allocation to files. Blocks should be big enough that there are no too many pere file, but not so big that small files (fitting into 1 block) leave a lot of unused space in the block.

Contiguous Allocation

- Contiguous allocation: simple idea
- simple to implement: only start block number and number of block allocated need to be recorded
- A file cannot grow unless completed rewritten to a new allocation of blocks. Can the old blocks be reallocated? Only to a file no bigger than this one.
- Tricky to manage reallocation.
- Potentially wasteful of space.

To illustrate, imagine a storage medium with just 32 blocks (usually many more than this) ...

Contiguous Allocation - Example



The red enclosing rectangles show a possible allocation of blocks to the files in the follwing directory listing.

file	start	length
work.doc	0	2
assgn.doc	13	3
diary.txt	19	6
log.txt	28	4
conts.txt	6	2

Notice that a file of more than 4 blocks in size will not fit antwhere!

Linked Allocation

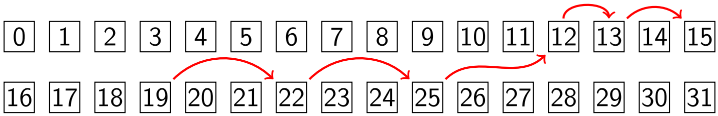
- Each file is a linked list of blocks.
- Directory entry gives (pointer to) first block;
- Each block contains pointer to next block,
  - *null* $\Rightarrow$ last block
  - ... followed by file data.
- Blocks for a file may be scattered all over the medium, in no particular order.
- Simple: Directory entry needs only starting block number.
- No waste space:
  - free blocks managed in a linked list;
  - allocated to files as needed by file;
  - returned to free list as no longer needed.

Linked Allocation - Addressing

- Assuming block size = 512 bytes: 4 for 'next block' pointer, 508 for file data. Where is byte n of a file?
  - Let L denote linked list block-numbers allocated to file, and $L[i]$ the i^{th} item in the list ...
  - Byte n resides in block $L[n/508]$
  - ... at *offset* $(4 + n\%508)$
- More generally if block size = b wth 4 bts for 'next block' pointer,
  - Byte n resides in block $L[n/(b-4)]$
  - ... at *offset* $(4 + n\%(b-4))$
- No logical limit on size linked allocation supports
- but random access may become slow if a long linked list has to be traversed.

Linked Allocation - Example

Imagine a storage medium with just 32 blocks as before ...



file	start	length
diary.txt	19	6

- The directory entry says the file data starts at block 19
- Blocks 19, 22, 25, 12, 13, 15 each contain a pointer to the next, plus file data bytes;
- Block 15 contains a null pointer plus the last lot of file data bytes.

Linked Allocation and FAT File Systems

FAT file systems use linked allocation just like this, except the pointers reside not in the blocks but in a separate *File Allocation Table* for the volume/partition.

- The FAT is an array of block numbers, indexed by block numbers
- In the example above, we would have FAT[19] = 22, FAT[22] = 25, FAT[25] = 12, FAT[12] = 13, FAT[13] = 15, FAT[15] = null.
- FAT-16 (16-bit block numbering) was devised originally for MsDOS.
- FAT-32 is still used for memory sticks and portable hard drives as it is supported by a range of operating systems - *nix, Windows, MacOS.

Exercise - How does the addressing scheme change with this variation? What are the block number and offset within the block of byte n?

See Wikipedia article on FAT.

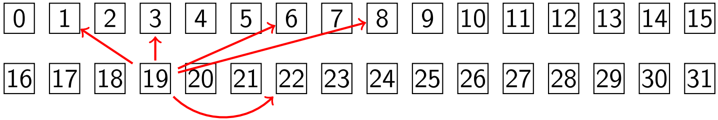
Indexed Allocation

In this method, a block stores in an array the numbers of all the blocks containing file data.

- The directory entry for the file gives the block number of the *index block*
- From this is read all the data block numbers, in order.

Indexed Allocation - Example

Medium with just 32 blocks as before ...



file	start	length
diary.txt	19	6

- The directory entry says the file data starts at block 19
- block 19 = [6, 22, 8, 3, 1, null, null, ...]

Indexed Allocation - Addressing

- With block size b (eg 512 bytes), physical address of byte n of file is ...
  - Let B denote the array of block numbers for the file: the *index table*;
  - Let $q = n/b$ and $r = n\%b$.
  - Byte n resides in block $B[q]$ at offset r
- Thus, efficient random access.
- Maximum size supported is number of array elements that will fit into a block.

Mixed Allocation systems

Hybrids of linked-list and index-block approaches are possible: for instance,

- Use a linked-list of index tables (arrays).
- As above, b denotes block size, n logical address of byte in file; assume 4 bytes of a block reserved for pointer to next block ...
  - Let $q_1 = n/b(b-4)$ and $r_1 = n\%b(b-4)$;
  - q_1^{th} item in linked list is block number of index table for n
  - Let $q_2 = r_1/b$ and $r_2 = r_1\%b$;
  - q_2^{th} entry in q_1^{th} index table is block number of block containing n;
  - r_2 is offset into this block of byte n.
- No theoretical limit on size; 'semi'-efficient random access (traverse of linked-list!).

Mixed Allocation systems - 2

Similarly we can have an index table of index tables.

- As above, b denotes block size, n logical address of byte in file;
  - Let $q_1 = n/b^2$ and $r_1 = n\%b^2$;
  - q_1^{th} item in outer index is block number of index table for n
  - Let $q_2 = r_1/b$ and $r_2 = r_1\%b$;
  - q_2^{th} entry in q_1^{th} index table is block number of block containing n;
  - r_2 is offset into this block of byte n.
- Theoretical size limit is b^3.
- Efficient random access because of direct array look-up.

Unix Example

Unix *inodes* are data structures holding informaton about files. Sun Microsystems developed the following variation (1991) -

- Block size = 4 kb
- Directory entry has
  - a *mode*
  - *owners* (2 - user, group)
  - time stamps (3 - creation, last access, last modification)
  - size (block count)
  - some *direct blocks* - pointers to blocks
  - a *single indirect* pointer to an index block
    - an array of blocks
  - a *double indirect* pointer to an array of index blocks
    - an array of arrays of blocks
  - *triple indirect* is also possible.

The idea was that a small file would the array of contiguous blocks, for fast data access, but could switch seemlessly to indirect or doubly indirect allocation as the file grew.

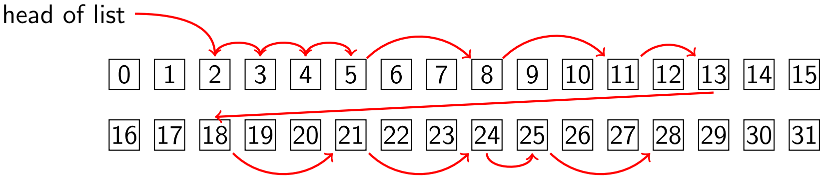
Managing Free Space

Simple idea: a *bit vector* with a bit recording the status of every block:

- bit[n] $==0 \Leftrightarrow$ block n occupied.
- Fast to find, for any n, a run of n contiguous free blocks - especially with Intel machine instruction to report first 1-bit in a word.
- The whole vector needs to be in RAM and can take a lot of space: A 1.3 Gb disk with 512 b blocks would need 332 kb for the bit vector.
- It needs to be on disk also.
- Must never allow block n to have bit$[n] = 1$ in memory but bit$[n] =$ 0 on disk
  - Always set bit$[n] = 1$ on disk, then de-allocate block n, then set bit$[n] = 1$ in RAM.

Managing Free Space ctd

Another approach is to use a linked list of free blocks -



Drawback: large linked lists are slow to access except sequentially;
$\Rightarrow$ slow to find contiguous space

Counting approaches

- For instance keep start address of first free block of each *run* of contiguous blocks, and the run *length*.

Efficiency and Performance

Efficiency depends on

- disk allocation and directory algorithms, as above
- types of data kept in files directory entry
  Tradeoffs -
- Unix tends to pre-allocate inodes throughout a disk volume
  - taking up space even when no data, files
  - ... but this leads to better performance through lower seek times
- Large pointers (64 rather than 32 or 16 bit) use space but can 'count higher'
- Large blocks 'waste disk space' but allow efficient management of large disks.

Performance

- Disk cache separate section of main memory for frequently used blocks, process pages
- *free-behind* and *read-ahead* techniques to optimise processing of sequential access
- Improve PC performance by dedicating section of memory as virtual disk, or RAM disk

Recovery

- *Consistency checking*: compares data in directory structure with data blocks on disk, and tries to fix inconsistencies;
- Use system programs to back up data from disk to another storage device
- Recover lost file or disk by restoring data from backup

Journalling File Systems

- These file systems record each update of the file system as a *transaction*
- All transactions are written to a *log*
  - A transaction is *committed* once it is written to the log
  - but the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
- When the file system is updated completely according to the transaction, the transaction is removed from the log
- If the file system crashes at any, on recovery the log shows the remaining transactions that must still be performed.

See Wikipedia articles on EXT and on Journalling file systems

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an 'unreliable' datagram protocol (UDP/IP protocol and Ethernet)

NFS (ctd)

Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner

- A remote directory is mounted over a local file system directory
  - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
- Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided.
  - Files in the remote directory can then be accessed in a transparent manner.
- Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

NFS (ctd)

Designed to operate in a heterogeneous environment

- different machines, operating systems, network architectures ...

See also the Wikipedia article on NFS