File Systems

Week 10 – Tutorial

Problem 4.3

- In early UNIX systems, executable files (a.out files) began with a very specific magic number, not one chosen at random. These files began with a header, followed by the text and data segments. Why do you think a very specific number was chosen for executable files, whereas other file types had a more-or-less random first word?

Problem 4.3 – Solution

- The reason for using a special magic number in executable file is that header itself is not executable part of a program
- To avoid trying to execute the header as code, the magic number was a BRANCH instruction with a target address just above the header
- Thus, the binary file could be read into the address space of the new initiating process without even knowing how big the header was

Problem 4.4

- Is the **open()** system call in UNIX absolutely essential? What would the consequences be of not having it?

Problem 4.4 – Solution (1/2)

- If open() system call wouldn't exist, a programmer should have taken care of all the file related routines:
  - Every time when we need to read from a file, it would be necessary to specify its name
  - The system would then need to fetch the i-node for the file
  - Then, occasionally, the i-node should be flushed back to disk. First, we have to choose an appropriate moment. Second, we have to deal with possible time-outs

Problem 4.4 – Solution (2/2)

- Resolution: it is possible to live without open(), however, it is much more complicated

Problem 4.6

- Some operating systems provide a system call **rename()** to give a file a new name. Is there any difference at all between using this call to rename a file and just copying the file to a new file with the new name, followed by deleting the old one?

Problem 4.6 – Solution

- There are two major differences:
  - If disk is almost full, copy might fail
  - Copying a file and deleting an old one will result in change of file attributes, such as creation or last modification time

Problem 4.8

- A simple operating system supports only a single directory but allows it to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?

Problem 4.8 – Solution

- Yes, we can use slashes in file names emulating hierarchical file paths. For example:
  - /usr/local/bin/myfile1
  - /usr/local/bin/myfile2
- In this case we could, for instance, list all the files in /usr/local/bin/ “directory” by typing:
  - ls /usr/local/bin/*

Problem 4.12

- Describe the effects of a corrupted data block for a given file for:
  A. contiguous
  B. linked
  C. indexed (or table based)

Problem 4.12 – Solution (1/3)

- Contiguous: a file is stored as a chain of contiguous blocks. The operating system stores the address of the first block and the number of blocks
- If one block is corrupted, the other ones are still usable

Problem 4.12 – Solution (2/3)

- Linked: a file is stored as a linked list of the blocks. Each block's address is stored as a pointer in previous block. The operating system only stores pointers to the first and the last blocks
- If one block is corrupted, all the blocks following this one become unusable

Problem 4.12 – Solution (3/3)

- Indexed: to the physical addresses of all the blocks of a file, operating system uses a table which is stored on a disk too
- If a block containing addresses is corrupted, it might be not possible to read any block belonging to the file
- If a block with data is corrupted, other blocks are not affected

Problem 4.17

- For a given class, the student records are stored in a file. The records are randomly accessed and updated. Assume that each student's record is of fixed size. Which of the three allocation schemes (contiguous, linked and table/indexed) will be most appropriate?

Problem 4.17 – Solution (1/2)

- For both operations we need to access a data block containing a student record
  - Since records are of fixed size, we can easily calculate a block number for a given record
  - Then, for contiguous allocation we can calculate an address of the block by using formula:
    first_block_addr + block_size * (block_num - 1)

Problem 4.17 – Solution (2/2)

- For indexed allocation we just use block number as an index and a value would be block address
- For linked-list allocation, however, it will require multiple disk reads per operation since it is not possible to get a physical address of a block without reading all the previous blocks

Problem 4.19 (1/2)

- It has been suggested that efficiency could be improved and disk space saved by storing the data of a short file within the i-node. For the i-node of Fig. 4-13, how many bytes of data could be stored inside the i-node?

Problem 4.19 (2/2)

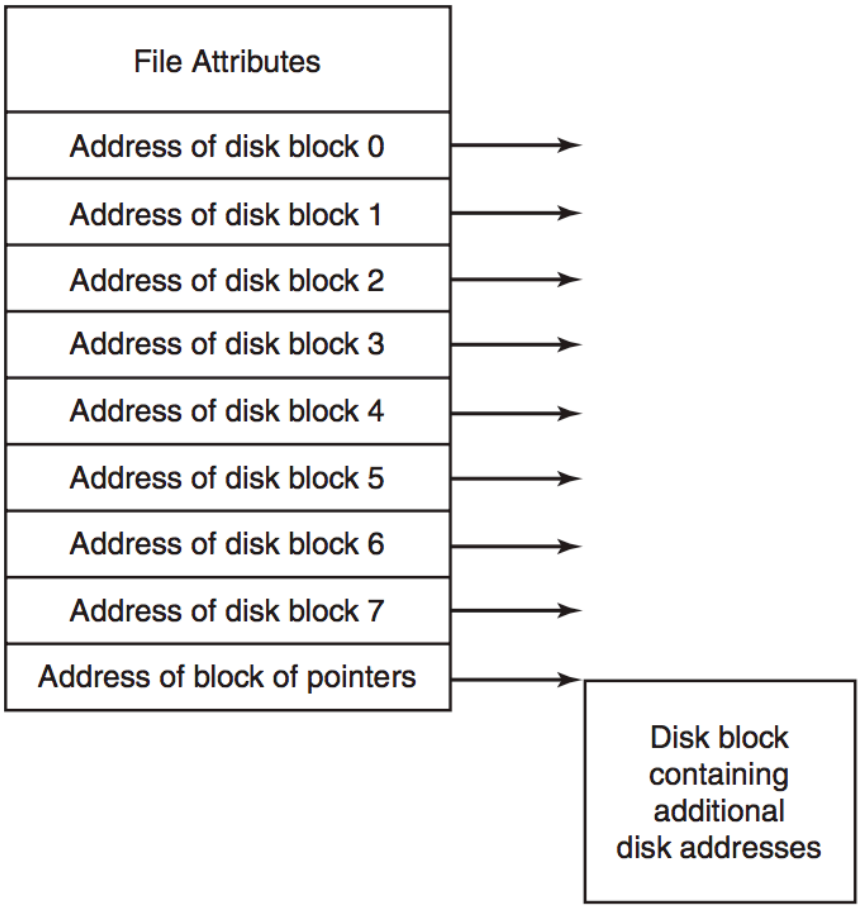

Figure 4-13. An example i-node

Problem 4.19 – Solution

- The exact amount of bytes really depends on length of pointers. However, we need to take into consideration how we are going to indicate whether an i-node contains addresses or data
- If it has a free bit among the attributes, we can use this bit and store data instead of nine pointers
- Otherwise, we can make the first address to be invalid, thus, marking following bytes as data
- If the length of the pointer is n, then we can store $9n$ or $8n$ bytes respectively

Problem 4.20

- Two computer science students, Carolyn and Elinor, are having a discussion about i-nodes
- Carolyn maintains that memories have gotten so large and so cheap that when a file is opened, it is simpler and faster just to fetch a new copy of the i-node into the i-node table, rather than search the entire table to see if it is already there. Elinor disagrees
- Who is right?

Problem 4.20 – Solution

- Having two copies of the i-node in the table at the same time introduces all kinds of synchronization related troubles.
- In particular, if multiple i-nodes are used for modifying a file and each i-node updates some part of it, then the i-node which would be the last one to be written to the disk, would win
- The information in others would be lost

Problem 4.25

- The beginning of a free-space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest-numbered block, so after writing file A, which uses six blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:
    A. File B is written, using five blocks
    B. File A is deleted
    C. File C is written, using eight blocks
    D. File B is deleted

Problem 4.25 – Solution

A. File B is written, using five blocks:
   1111 1111 1111 0000
B. File A is deleted:
   1000 0001 1111 0000
C. File C is written, using eight blocks:
   1111 1111 1111 1100
D. File B is deleted:
   1111 1110 0000 1100

Problem 4.27

- Oliver Owl's night job at the university computing center is to change the tapes used for overnight data backups. While waiting for each tape to complete, he works on writing his thesis that proves Shakespeare's plays were written by extraterrestrial visitors. His text processor runs on the system being backed up since that is the only one they have
- Is there a problem with this arrangement?

Problem 4.27 – Solution

- Most probably, everything but Oliver's file will backup
- This happens because a backup program may detect that a file is open for writing and pass over it since the state of file content may be indeterminate.

Problem 4.38

- Given a disk-block size of 4 KB and block-pointer address value of 4 bytes, what is the largest file size (in bytes) that can be accessed using 10 direct addresses and one indirect block?

Problem 4.38 – Solution

- In total, there are 1034 addresses:
  - 10 direct addresses
  - 1024 (4 KB / 4 bytes) addresses that one indirect block contains
- Each address points to a 4 KB block
- Therefore, the largest file size is:
  1043 * 4 KB = 4 272 128 bytes

Problem 4.41

- How many disk operations are needed to fetch the i-node for a file with the path name /usr/ast/courses/os/handout.t?
- Assume that the i-node for the root directory is in memory, but nothing else along the path is in memory
- Also assume that all directories fit in one disk block

Problem 4.41 – Solution

- The following disk reads are needed:
  - directory for /
  - i-node for /usr
  - directory for /usr
  - i-node for /usr/ast
  - directory for /usr/ast
  - i-node for /usr/ast/courses
  - directory for /usr/ast/courses
  - i-node for /usr/ast/courses/os
  - directory for /usr/ast/courses/os
  - i-node for /usr/ast/courses/os/handout.t

- In total, 10 disk reads are required

End

Week 10 – Tutorial