File-System Implementation

CHAPTER 14

Practice Exercises

14.1 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. Calculate how many disk I/O operations are 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 at the beginning but there is room to grow at the end. Also 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 in the middle.

c. The block is added at the end.

d. The block is removed from the beginning.

e. The block is removed from the middle.

f. The block is removed from the end.

Answer:
The results are:

	Contiguous	Linked	Indexed
a.	201	1	1
b.	101	52	1
c.	1	3	1
d.	198	1	0
e.	98	52	0
f.	0	100	0

14.2 Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

Answer:
In case of a system crash (memory failure), the free-space list would not be lost, as it would be if the bit map had been stored in main memory.

14.3 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?

Answer:

- **Contiguous**—if file is usually accessed sequentially, if file is relatively small.
- **Linked**—if file is large and usually accessed sequentially.
- **Indexed**—if file is large and usually accessed randomly.

14.4 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

Answer:
This method requires more overhead then the standard contiguous allocation. It requires less overhead than the standard linked allocation.

14.5 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

Answer:
Caches allow components of differing speeds to communicate more efficiently by storing data from the slower device, temporarily, in a faster device (the cache). Caches are, almost by definition, more expensive than the devices they are caching for, so increasing the number or size of caches would increase system cost.

14.6 Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

Answer:
Dynamic tables allow more flexibility as a system grows—tables are never exceeded, avoiding artificial use limits. Unfortunately, kernel structures and code are more complicated, so there is more potential for bugs. Dynamic tables use more system resources than static tables, thus potentially taking system resources away from other parts of the system as the system grows

Exercises

14.7 Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

a. All extents are of the same size, and the size is predetermined.

b. Extents can be of any size and are allocated dynamically.

c. Extents can be of a few fixed sizes, and these sizes are predetermined.

Answer:
If all extents are of the same size, and the size is predetermined, then it simplifies the block-allocation scheme. A simple bit map or free list for extents will suffice. If the extents can be of any size and are allocated dynamically, then more complex allocation schemes are required. It might be difficult to find an extent of the appropriate size, and there might be external fragmentation. We could use the Buddy system allocator discussed in the previous chapters to design an appropriate allocator. When the extents can be of a few fixed sizes, and these sizes are predetermined, we must maintain a separate bit map or free list for each possible size. This scheme is of intermediate complexity and of intermediate flexibility compared with the other two schemes.

14.8 Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

Answer:

- **Contiguous, Sequential**. Works very well; sequential access simply involves traversing the contiguous disk blocks.
- **Contiguous, Random**. Works very well, as you can easily determine the adjacent disk block containing the position you wish to seek to.
- **Linked, Sequential**. Satisfactory, as you are simply following the links from one block to the next.
- **Linked, Random**. Poor, as it may require following the links to several disk blocks until you arrive at the intended seek point of the file.
- **Indexed, Sequential**. Works well, as sequential access simply involves sequentially accessing each index.
- **Indexed, Random**. Works well, as it is easy to determine the index associated with the disk block containing the position you wish to seek to.

14.9 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. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at `/a/b/c`? Assume that none of the disk blocks is currently being cached.

c. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

Answer:

a. 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 pages are already allocated to jobs. The remaining unallocated pages could be relinked as the free-space list.

b. Reading the contents of the small local file */a/b/c* involves four separate disk operations: (1) reading in the disk block containing the root directory /, (2) and (3) reading in the disk block containing the directories *b* and *c*, and reading in the disk block containing the file *c*.

c. The free-space list pointer could be stored on the disk, perhaps in several places.

14.10 Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

Answer:
The primary difficulty that might arise is due to delayed updates of data and metadata. Updates could be delayed in the hope that the same data might be updated in the future or that the updated data might be temporary and might be deleted in the near future. However, if the system were to crash without having committed the delayed updates, then the consistency of the file system would be destroyed.

14.11 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), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

Answer:
Let Z be the starting file address (block number).

- **Contiguous**. Divide the logical address by 512, with X and Y the resulting quotient and remainder, respectively.
  a. Add X to Z to obtain the physical block number. Y is the displacement into that block.
  b. 1
- **Linked**. Divide the logical physical address by 511, with X and Y the resulting quotient and remainder, respectively.
  a. Chase down the linked list (getting $X + 1$ blocks). $Y + 1$ is the displacement into the last physical block.
  b. 4
- **Indexed**. Divide the logical address by 512, with X and Y the resulting quotient and remainder, respectively.
  a. Get the index block into memory. Physical block address is contained in the index block at location X. Y is the displacement into the desired physical block.
  b. 2

14.12 Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Answer:
(12 * 8 /KB/) + (2048 * 8 /KB) + (2048 * 2048 * 8 /KB/) + (2048 * 2048 * 2048 * 8 /KB) = 64 terabytes

14.13 Explain why logging metadata updates ensures recovery of a file system after a file-system crash.

Answer:
For a file system to be recoverable after a crash, it must be consistent or must be able to be made consistent. Therefore, we have to prove that logging metadata updates keeps the file system in a consistent or able-to-be-consistent state. For a file system to become inconsistent, the metadata must be written incompletely or in the wrong order to the file-system data structures. With metadata logging, the writes are made to a sequential log. The complete transaction is written there before it is moved to the file-system structures. If the system crashes during file-system data updates, the updates can be completed based on the information in the log. Thus, logging ensures that file-system changes are made completely (either before or after a crash). The order of the changes is guaranteed to be correct because of the sequential writes to the log. If a change was made incompletely to the log, it is discarded, with no changes made to the file-system structures. Therefore, the structures are either consistent or can easily be made consistent via metadata logging replay.