CMPSCI 377 Operating Systems **Spring 2010**

Lecture 19: April 1

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

19.1 How Disks Work

A typical hard disk drive is composed of one or more circular platters and a disk head that can read data from the magnetic material used to build each platter. The surface of each platter is divided into concentric rings (like the rings on a tree), known as **tracks** (or **cylinders**. Each track is split into **sectors** also known as **blocks**, which are the minimum read or write unit on the disk. The disk platter is constantly spinning, and the disk head can be moved up or down to reach a specific track on the platter. These mechanical operations are the primary causes of disk overheads—the **latency** of a disk access is the amount of time to initiate a disk transfer of 1 byte and is composed of the seek time and rotational time. The seek time is how long it takes for the head to reach the proper track, and the rotational time is how long it takes for the correct sector to rotate under the head so that it can be read. In addition to latency, the main performance characteristic of disks is the **bandwidth** that they can achieve once the transfer has been initiated, typically in megabytes per second.

19.2 Organizing Data on Disk

In the previous lecture we discussed how a file is composed of multiple data blocks. When a file is stored to disk, a mapping must be kept that translates the block of a particular file to a precise location on the disk. A disk location has three parts to its address, the platter number, the cylinder, and the sector. The disk and file system need a way to maintain these mappings using an efficient data structure.

A **file descriptor** is the structure on a disk that stores information about each file. The file descriptor must specify where on disk the different blocks are stored, as well as other attribute information described in the previous lecture. There are many approaches to organizing files on disk. In general, the on disk layout is optimized for the following common characteristics

- Most files are small
- Most disk space is taken up by a small number of large files
- I/O operations will target both small and large files.

These requirements mean that the cost for accessing each file must be low, since a typical disk has a very large number of small files. However, the system must also provide good performance for large files, as they often consume a large portion of the total disk space.

19.2.1 Contiguous Allocation

In a Contiguous Allocation disk layout the OS maintains an ordered list of all disk blocks that are currently free. When a file is created, the OS allocates it a contiguous chunk of free blocks for it to use. With this simple scheme, the file descriptor only needs to know the start location and length of a file in order to find all of its disk blocks. This system is very simple and can provide very good performance when making sequential reads to a file since all of its disk blocks are laid out in contiguous sectors, but it runs into problems when the size of files change over time. When files are growing or shrinking, this approach can exhibit the same type of fragmentation that we saw with contiguous memory allocation schemes—small groups of sectors can appear between files that are not large enough to be used by any new files. As a result, Contiguous Allocation disk layouts are not commonly used in modern systems.

19.2.2 Linked Files

A Linked File disk layout each file is stored similar to a linked list data structure. The OS again keeps a list of all free sectors, and when a new file is created a file descriptor is made with a pointer to the first sector for its data. At the end of each sector, an additional pointer is kept to the next sector used by the file. This is a much more flexible scheme since files can grow to be very long by simply extending the chain of pointers, and it eliminates fragmentation since disk sectors for a file no longer have to be contiguous. Sequential disk reads can still be performed reasonably efficiently since after reading a sector the OS knows the address to the next sector, but they may require a large number of disk seeks (one per sector) if none of the sectors are contiguous. This approach performs very poorly when a process makes **random reads** to arbitrary locations within a file, rather than just sequential ones. To handle a random read in a Linked File based system, the OS still must start at the first block and read through every single block until it eventually reaches the desired block. For example, if you had a 1GB file and only wanted to read the very last block in the file (typically the last 1KB), the OS would still need to sequentially read the full 1GB file before it could return the data!

19.2.3 Indexed Files

An Indexed File layout tries to support good performance for both sequential and random accesses by maintaining a special index within the file descriptor. This index is simply an array containing pointers to each of the disk blocks used by the file. This approach is quite efficient since files do not need to be contiguous, but random reads can still be performed efficiently by quickly looking up block addresses in the index. However, the system requires that a maximum file size be set ahead of time since each file must have a fixed number of entries in its index. It also can lead to large numbers of seeks if data is not placed in contiguous sectors.

19.2.4 Multilevel Indexed Files

A Multilevel Index can be used to eliminate the maximum file constraint of regular Indexed File layouts. In this case, the index maintained for each file contains multiple different types of entries: (1) pointers to data blocks (as in the previous scheme), (2) pointers to other index blocks, or (3) pointers to blocks containing more pointers to index blocks. For example, a file may be created with an index with 14 entries. The first 12 entries point directly to data blocks—an ordinary small file would only require the use of these pointers. The 13th entry would be a pointer to a second index (with the same structure of 14 entries), providing a level of indirection to additional data blocks and index pointers. Finally, the 14th entry would provide a second level of indirection by pointing to a block filled with pointers to other index blocks. Having these extra levels of indirection means that the index can accomodate very large files. However, it produces a performance difference for random reads to small and large files. A random read to a small file may not require any extra accesses, but reading from a large file could require jumping between several levels of indirection. This solution also does not try to place data in contiguous sectors, potentially leading to large numbers of seeks.

19.3 Free-Space Management

The OS needs a way to keep track of which blocks are used or free for all of the disk layouts described above. This is a similar problem to maintaing a free-space list for main memory described in previous lectures. The goal of free-space management is to be able to quickly find free blocks on disk, and to be able to efficiently mark blocks as free or in use. A simple approach is to use a **bitmap**. A bitmap is a string of bits where each bit represents a block on disk. If the bit is set to 1, then the block si free, if it is 0, then the block has already been allocated. Using a bitmap makes it very easy to check if a particular block is free—the address of the block can be used to lookup the appropriate region in the bitmap and bit operations can be used to find out if the block is free or to mark it as used. One problem with using a bitmap approach is that it can require a very large amount of memory. Since 1 bit is required for every block on disk, larger disks require larger bitmaps. If a system uses 512 Byte disk blocks and the disk holds 2GB, then the bitmap must have 4,000,000 bits in it, or 500,000 bytes. Disks today can be hundreds of GB or more, which could require substantial amounts of memory.

A second big problem with bitmaps is that it can be very slow at finding a free block, since the OS may need to scan a large part of the bitmap before a ”1” entry is found. To reduce the cost of finding free blocks, a linked list based approach can be used instead. In this case, the list of free blocks is maintained as a linked list in kernel memory. This can make finding free blocks must faster, but can require even larger amounts of memory to maintain the full data structure.