| | |
|---|---|
| **Systems Programming and Operating Systems** | **Induprakas Kodukula** |
| **CS414, Summer 1996** | **Class#14, 7/30/96** |

# 1 Class #11

Having studied the two introductory aspects of file systems, namely **directories** and **physical alloction of files to disk**, it is now time to turn one's attention to the remaining four aspects of file systems of interest. These are

Management of free space on disk

Directory implementation

Disk scheduling

Future directions

The this lecture is devoted to these four issues.

## 1.1 Free space management

As has already been seen, it is unrealistic to lay out an entire file as a contiguous set of blocks on a disk. Consequently, one has to manage the break up of a file onto various blocks of the disk. This section studies the various options possible and briefly discusses the trade-offs.

The simplest way to store information about free blocks of a disk is using a bit vector. The length of the bit vector is equal to the number of free blocks on disk. Every free block has a bit for it in the bit vector; the bit can be set to 0 or 1 depending on whether the block is filled or occupied. The main advantage of this approach is simplicity. In addition, if there are hardware instructions supporting bit operations, then this method can be quite fast. The main disadvantage of this approach is that the bit vector must be stored in memory and can be quite big. For a 1.3 gigabyte disk with 512 byte blocks, the size of the bit vector can be as large as 310K.

```
Obpuzzle: How do you count the number of 1's in a 32-bit word in
time proportional to the number of 1's?
```

A second approach is to link together all the free blocks of the system. A special pointer points to the first free block on disk, which contains a pointer to the second free block and so on. The disadvantage of this approach is the usual: a *disk block* read is required to obtain the location of each (next) free block.

A modification to the previous approach is to actually store at the first free block of the disk, the addresses of the next $n$ free blocks. The first $n-1$ blocks are actually free. The last block contains a bunch of pointers to the next $n$ free blocks.

The final optimization one can perform is to note that very often blocks may be allocated or freed in a sequence at a time. So, very often in the previous case, many of the succesively stored free block addresses can be consecutive. In this case, it is more efficient to replace a consecutive list of block addresses by two fields (a start block address, and an integer indicating the number of consecutive free blocks starting from that address).

### 1.2 Directory Implementation

Next, it is important to study how directories are implemented. Since directories are responsible for storing information about files, it is necessary that file lookup, search, insertion and deletion operations are implemented efficiently.

The first attempt at implementing a directory is by a linked list mechanism. An entry for each file would contain all the specific information in addition to a pointer to the data of the file. However, a linear linked list would require linear amount of time to perform all the operations described above.

```
ObPuzzle: How do you insert an element at the end of the list, given
a pointer only to the last element?
```

The second attempt at a directory implementation is using hash tables. A directory is created with a certain number of entries. For any given file name, a hash function is used to map the filename into an entry number in the directory. This makes operations such as insertion and deletion really straightforward and efficient. There is one complexity that needs to be handled: which is that the **hashing function** maps distinct file names onto the same entry - this is known as a **collision**. In this case, special policies can be used to still locate the desired files efficiently.

### 1.3 Disk Scheduling

The organization of a disk is usually in the form of several platters on top of each other. Each platter consists of *tracks*. Each track contains several *sectors*. A block is contained in a sector. When a request is made for a block to be read, the following sequence of events happens:

Head positions itself over the track (known as seek time delay)

Track rotates until the required sector comes under the head (latency delay)

The data is transferred from the disk to disk controller (transfer time)

The first time is typically the order of milliseconds, the second time depends on the rotation speed of the disk (typically 3000 rpm) and the last time is dependent on the amount of data being transferred (tens of MBytes/sec). The total time spent reading a block is the sum of these three times. Keeping this in mind, the following disk scheduling algorithms are studied.

The first approach to disk scheduling is *first-come-first-served* The problem with this approach is that requests arriving in order can have widely varying track and sector numbers. So, processing disk block requests in FCFS order could lead to excessively large seek times.

The second approach is called *Shortest-seek-time-first.* If the head is currently at a particular track, the next disk read request to be satisfied is the one that has the closest track number to the current disk operation.

The *SCAN* algorithm is the third approach to disk scheduling. In this approach, the head starts off at the outermost track and proceeds in satisfying any disk read requests. When it reaches the innermost track, it reverses direction and starts to satisfy requests in the reverse direction. The head continuously scans the disk from end to end.

The *C-SCAN* algorithm is similar to the *SCAN* algorithm with the following difference: When the disk head reaches the innermost track, it simply returns to the first track immediately. In other words, the disk is treated as if it was circular, with the last track adjacent to the first one.

Finally, the *LOOK* procedure realizes that one need not move all the way to the innermost or outermost position if there were no more requests to be satisfied in that direction. Essentially, the disk *looks* before proceeding in the current direction. If there are no more requests to be satisfied in the current direction, the disk reverses direction.

### 1.4 RAID

RAID stands for Redundant Array of Inexpensive Disks and is a very recent innovation to store and access data at much higher speeds than in conventional machines. The key intuition is that if the data is stored on a large number of small and fast disks, then several disk reads can be progressing at the same time, leading to higher throughput. This method is known as **data-striping**. A typical RAID system would use of the order of 100 disks.

There is a problem of availability of the disk service with RAID however. The *mean-time-between-failures* for a typical disk is about 200,000 hrs. However, in a system with a 100 disks, this time now reduces to only about 2000 hrs (since if even one of the disks fails, the system crashes), which is about 3 months. So, a RAID system must now provide some amount of fault-tolerance. RAID systems are still quite expensive and their use is quite limited at the moment.