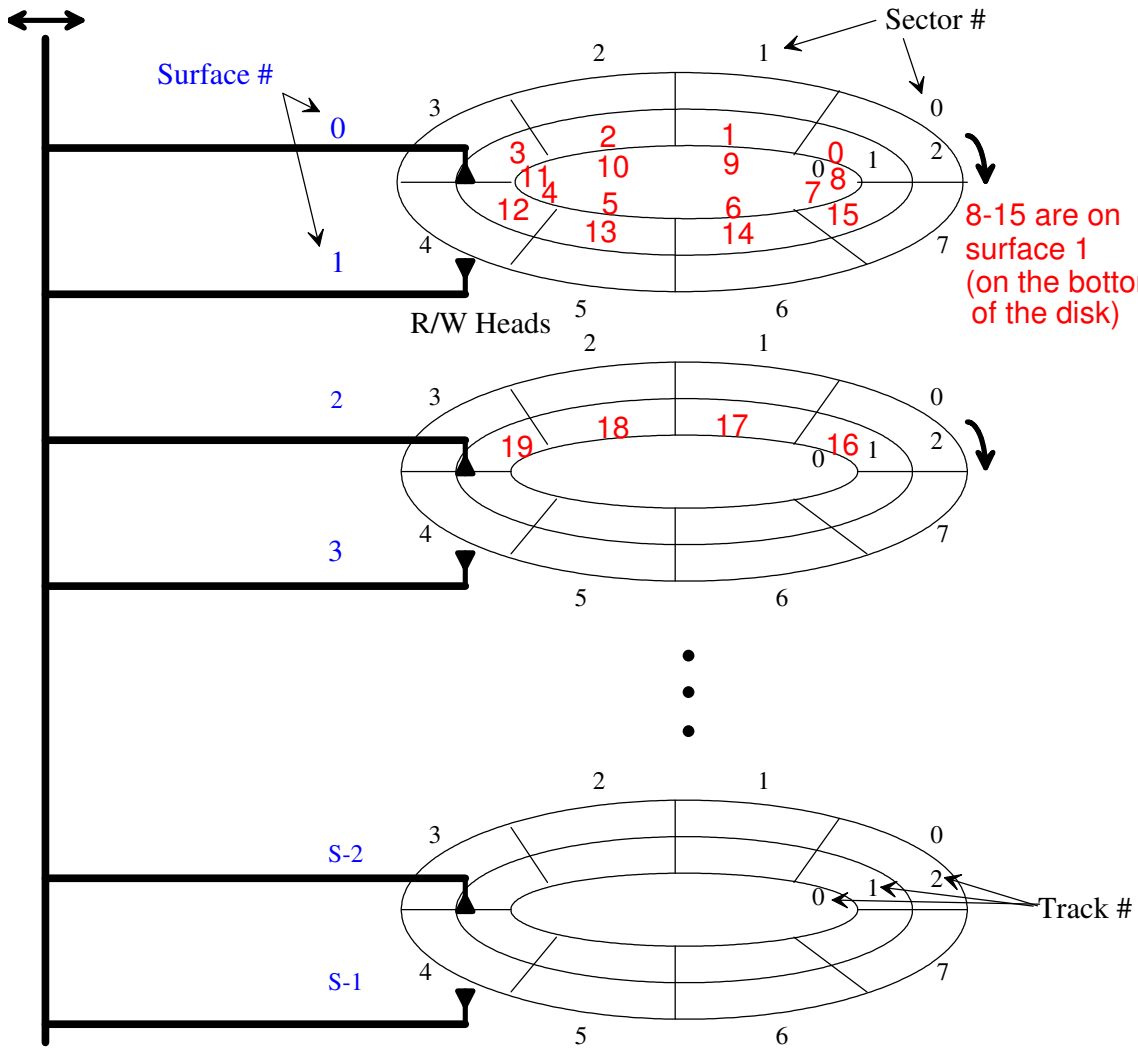
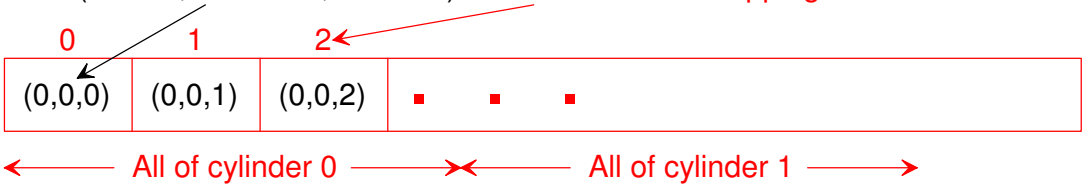
Logical View of Disk as Linear Collection of Blocks



(track #, surface #, sector #) to Linear block # mapping



Bits of linear block # : | track # | surface # | sector # |

1. Disk-access time = (seek time) + (rotational delay) + (date transfer time). How is each component of the disk-access time effected by increasing the disk's RPMs (revolutions per minute)?

b) If we want fast access to a collection of sectors, where can we place them to minimize seek time and rotational delay?

User Program - HLL programming language make system calls to OS to:

1. open file - establish a link between file variable and file for either reading, writing, or both
2. access file - read or write one piece of data at a time (e.g., char., record, etc.)
3. close file - flush changes to disk

Operating System - manage and control access secondary storage through its file system which contains information about every file: location on disk, ownership and security/protection
OS views disk as linear sequence of blocks (block 0, block 1, etc.), but assumes closeness in block # means close with respect to access time. OS maintains free disk space "list"
OS buffers some blocks in memory to improve efficiency

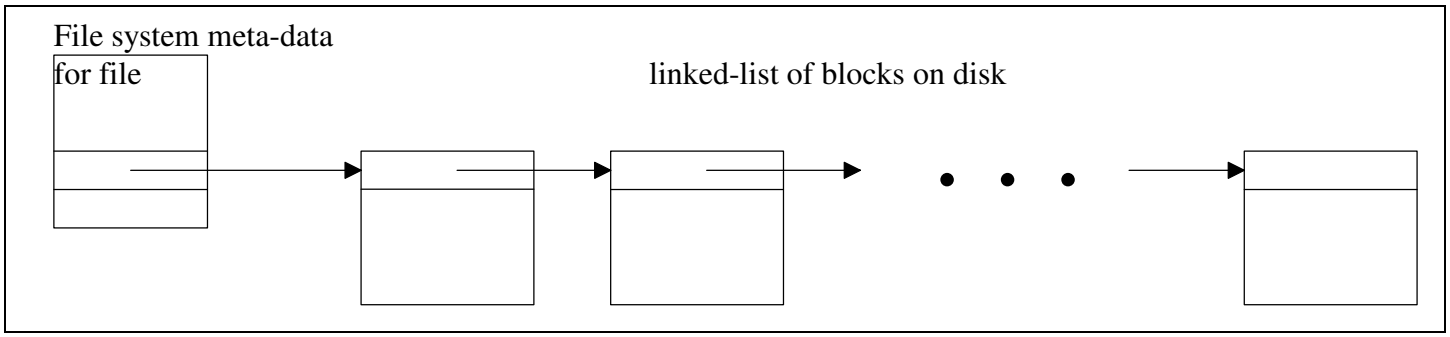
Secondary Storage Device - accepts R/W requests from OS for block# and maps block# to internals physical address (e.g., (track #, surface #, sector #) - more complex than above picture!)

Kinds of File Access:

- serial/sequential files - open at the beginning and read sequentially from beginning to end linearly
- random-access files - "*seek*" to any position by specifying a byte-offset from the beginning of the file, record #, etc.
- random-access of a record by key

Implementation of Files on Disk- how are blocks allocated?

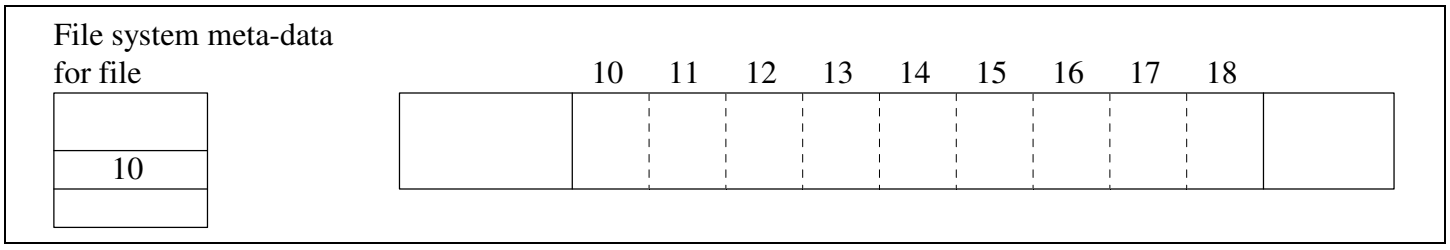
2. non-contiguous - scattered across linear address space of OS and disk



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

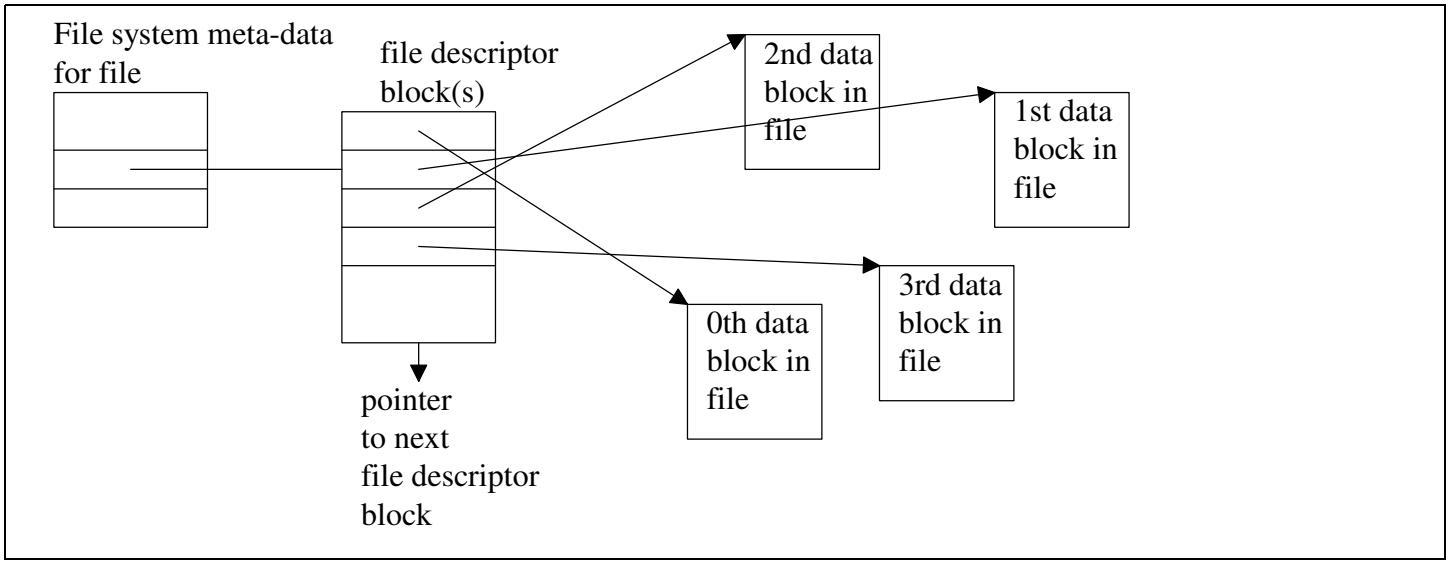
3. contiguous - sequential collection of blocks from OS linear view of disk



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

4. file descriptor blocks - list of blocks hold the address of the physical location of data blocks



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

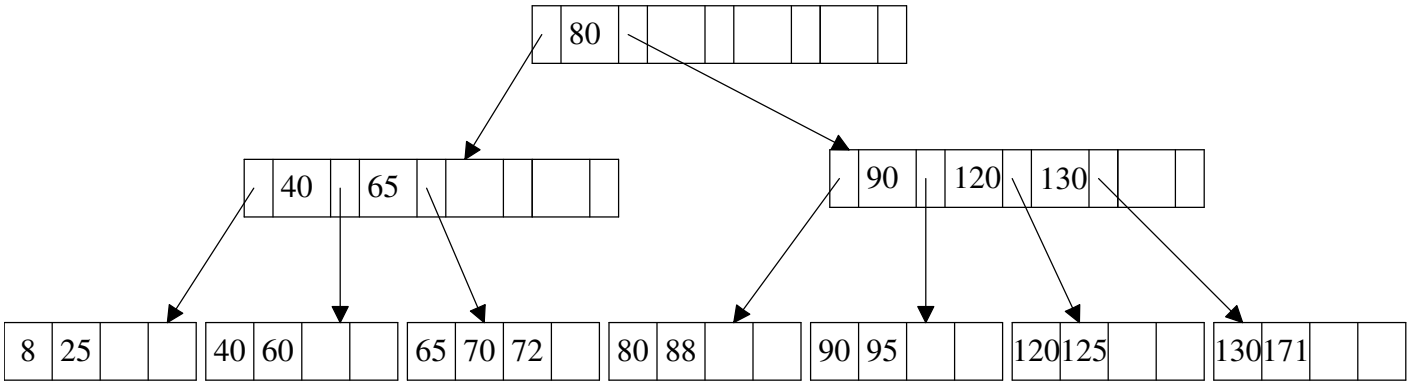
5. To implement "random-access of a record by key" in a file how might we use hashing?

6. To implement "random-access of a record by key" in a file why would an AVL tree not work well?

7. A B+ Tree is a multi-way tree (typically in the order of 100s children per node) used primarily as a file-index structure to allow fast search (as well as insertions and deletions) for a target key on disk. Two types of *pages* (B+ tree "nodes") exist:

- Data pages - which always appear as leaves on the same level of a B+ tree (usually a doubly-linked list too)
- Index pages - the root and other interior nodes above the data page leaves. Index nodes contain some minimum and maximum number of keys and pointers bases on the B+ tree's *branching factor (b)* and *fill factor*. A 50% fill factor would be the minimum for any B+ tree. All index pages must have $\lceil b/2 \rceil \leq$ # child $\leq b$, except the root which must have at least two children.

Consider an B+ tree example with b = 5.



a) How would you find 88?

b) Where would you insert 50, 100, 105, 110, 180, 200, 210?

8. For a B+ tree with a branch factor 201, what would be the worst case height of the tree if the number of keys was 1,000,000,000,000?