Files vs Disks

File Abstraction	*Disk Abstraction*
Byte oriented	Block oriented
Names	Block #s
Access protection	No protection
Consistency guarantees	No guarantees beyond block write

Filesystem Requirements

Naming

- Should be flexible, e.g., allow multiple names for same files
- Support hierarchy for easy of use

Persistence

- Want to be sure data has been written to disk in case crash occurs

Sharing/Protection

- Want to restrict who has access to files
- Want to share files with other users

FS Requirements (cont'd)

Speed & Efficiency for different access patterns

- Sequential access
- Random access
- Sequential is most common & Random next
- Other pattern is Keyed access (not usually provided by OS)

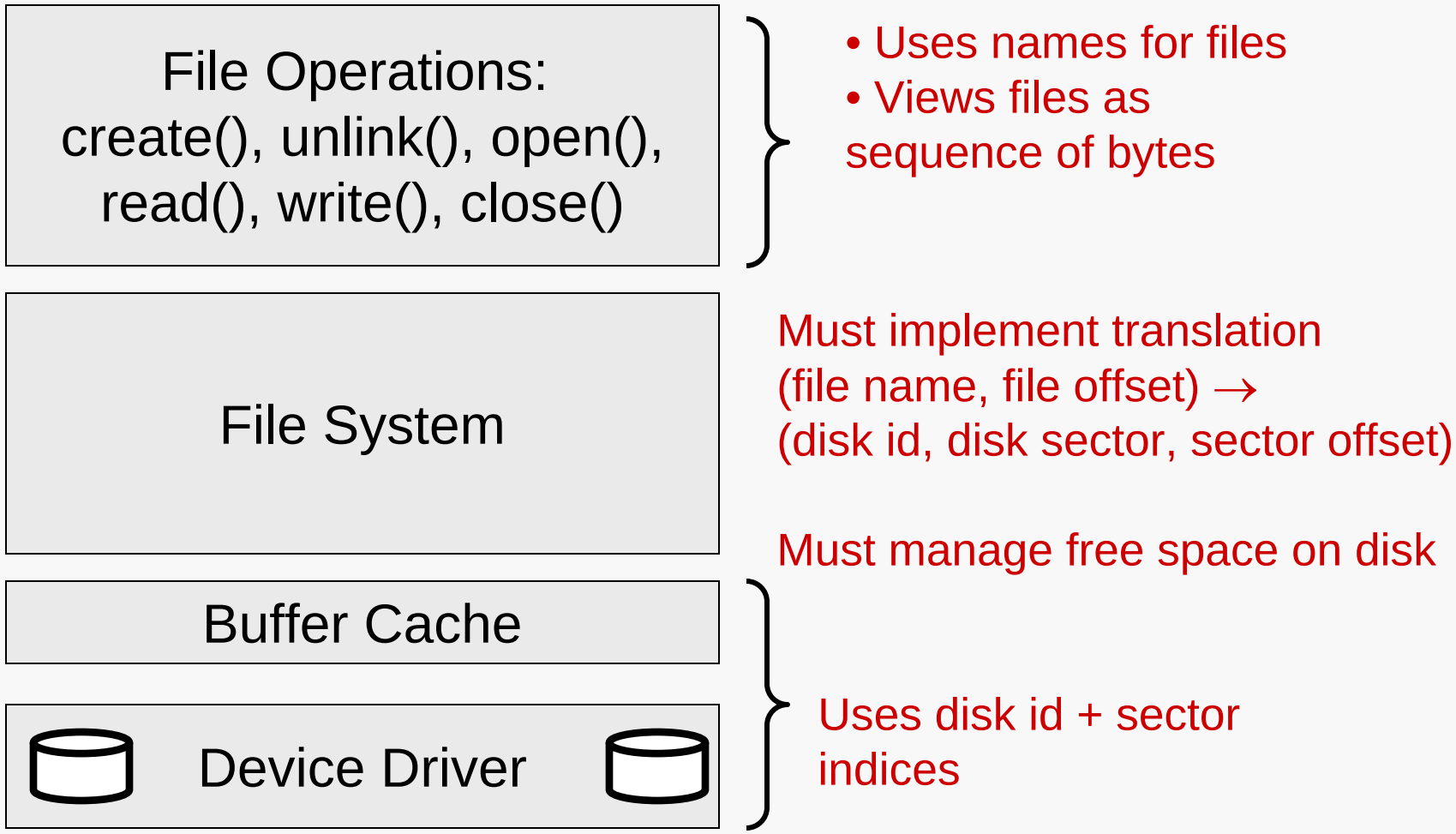
Minimum Space Overhead

- Disk space needed to store metadata is lost for user data

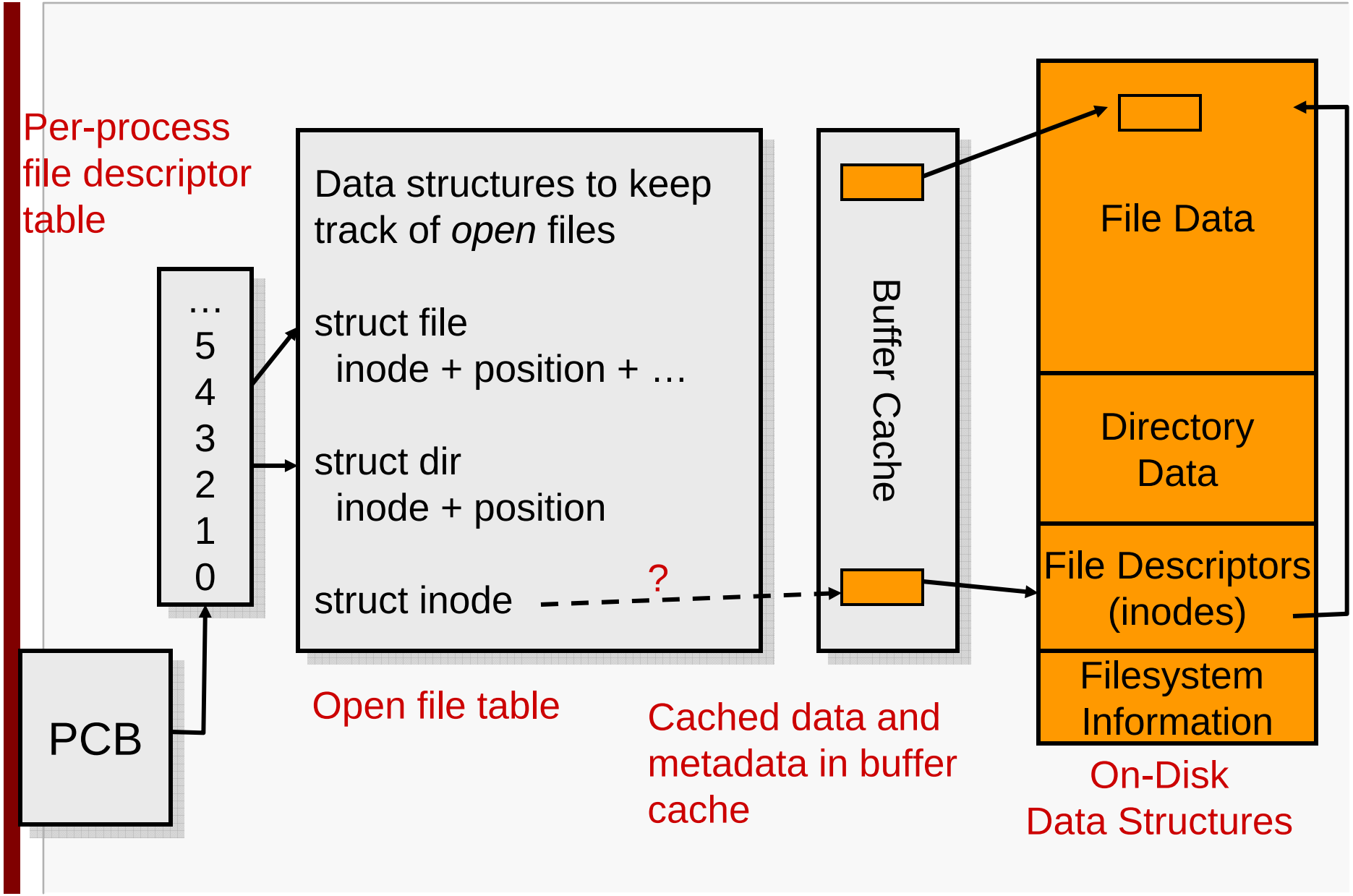
Twist: all metadata that is required to do translation must be stored on disk

- Translation scheme should minimize number of additional accesses for a given access pattern
- Harder than, say page tables where we assumed page tables themselves are not subject to paging!

Overview



The Big Picture



Steps in Opening & Reading a File

Lookup (via directory)

- find on-disk file descriptor's block number

Find entry in open file table (struct inode list in Pintos)

- Create one if none, else increment ref count

Find where file data is located

- By reading on-disk file descriptor

Read data & return to user

Open File Table

inode – represents file
- at most 1 in-memory instance per unique file
- #number of openers & other properties

file – represents one or more processes using an file
- With separate offsets for byte-stream

dir – represents an open directory file

Generally:
- None of data in OFT is persistent
- Reflects how processes are currently using files
- Lifetime of objects determined by open/close
  - Reference counting is used

File Descriptors ("inodes")

Term "inode" can refer to 3 things:

1. in-memory inode
   - Store information about an open file, such as how many openers, corresponds to on-disk file descriptor
2. on-disk inode
   - Region on disk, entry in file descriptor table, that stores persistent information about a file – who owns it, where to find its data blocks, etc.
3. on-disk inode, when cached in buffer cache
   - A bytewise copy of 2. in memory

Q.: Should in-memory inode store a pointer to cached on-disk inode? (Answer: No.)

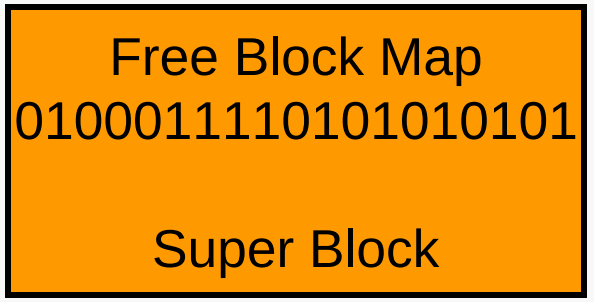
Filesystem Information

Contains “superblock”
stores information such as
size of entire filesystem, etc.

- Location of file descriptor table & free map

Free Block Map

- Bitmap used to find free blocks
- Typically cached in memory



Superblock & free map often replicated in different positions on disk

File Allocation Strategies

Contiguous allocation

Linked files

Indexed files

Multi-level indexed files

Contiguous Allocation

Idea: allocate files in contiguous blocks

File Descriptor = (first block, length)

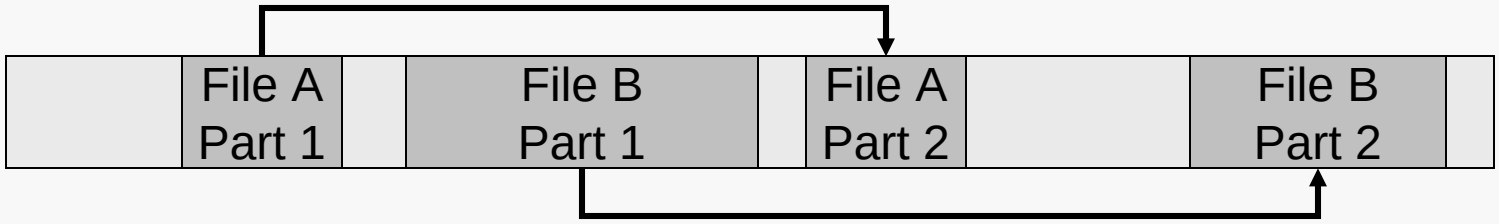
Good sequential & random access

Problems:

- hard to extend files – may require expensive compaction
- external fragmentation
- analogous to segmentation-based VM

Pintos's baseline implementation does this

Linked Files



Idea: implement linked list

- either with variable sized blocks
- or fixed sized blocks ("clusters")

Solves fragmentation problem, but now

- need lots of seeks for sequential accesses and random accesses
- unreliable: lose first block, may lose file

Solution: keep linked list in memory

- DOS: FAT File Allocation Table

DOS FAT

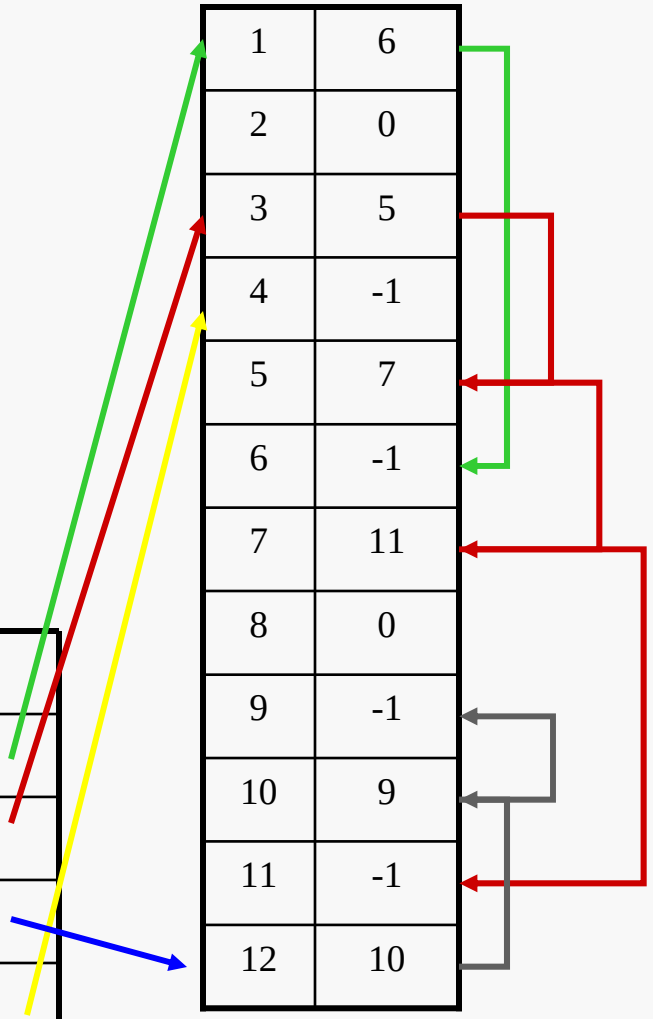
FAT stored at beginning of disk & replicated for redundancy

FAT cached in memory

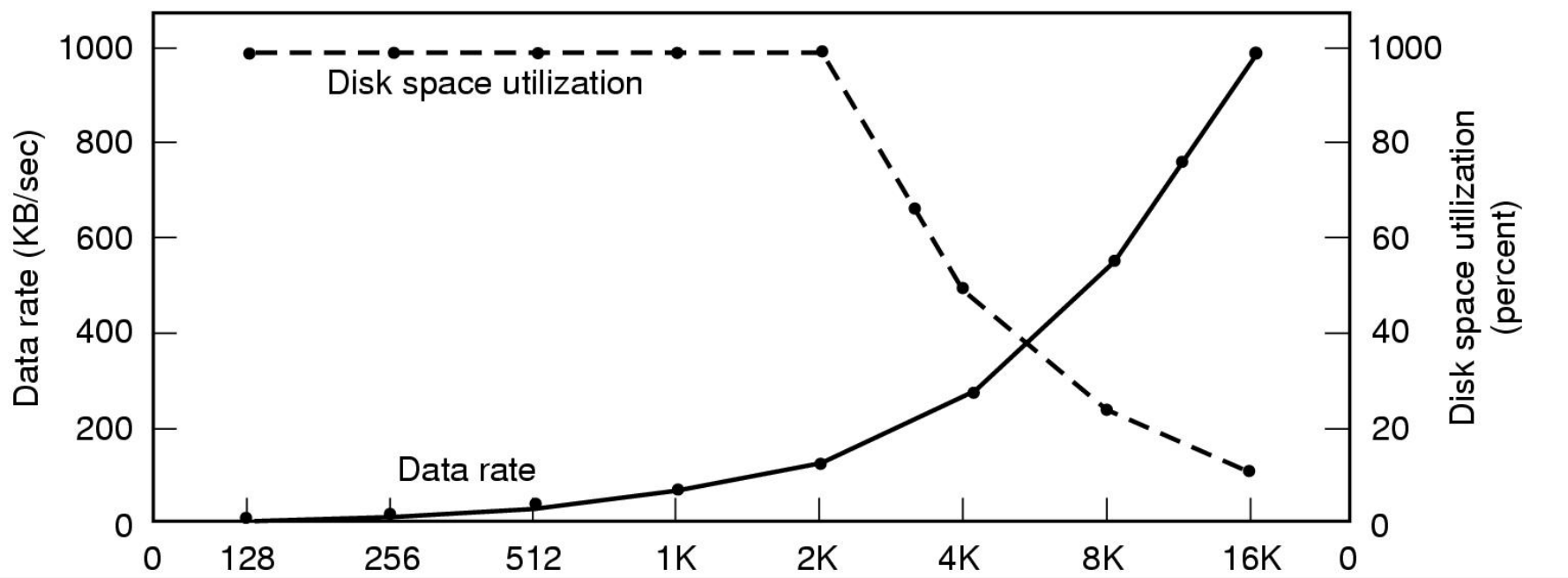
Size: n-bit entries, m-bit blocks → 2^(m+n) limit
- n=12, 16, 28
- m=9 … 15 (0.5KB-32KB)

As disk size grows, m & n must grow
- Growth of n means larger in-memory table

Filename	Length	First Block
“a”	2	1
“b”	4	3
“c”	3	12
“d”	1	4



Blocksize Trade-Offs

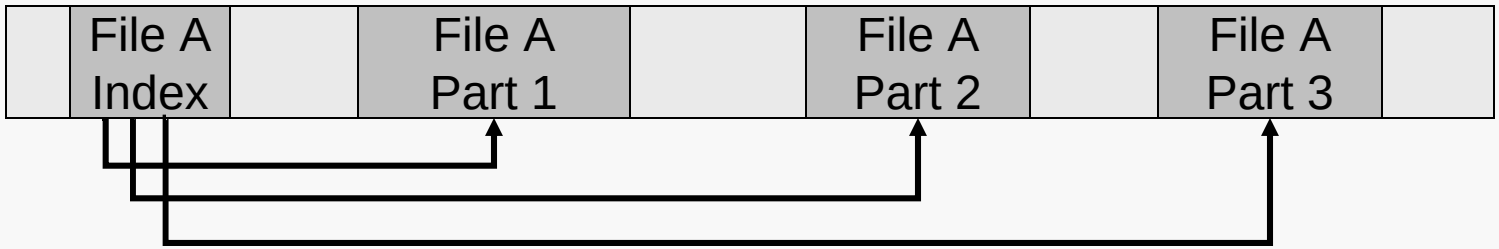


Assume all files are 2KB in size (observed median filesz is about 2KB)

- Larger blocks: faster reads (because seeks are amortized & more bytes per transfer)
- More wastage (2KB file in 32KB block means 15/16th are unused)

Source: Tanenbaum, Modern Operating Systems

Indexed Allocation

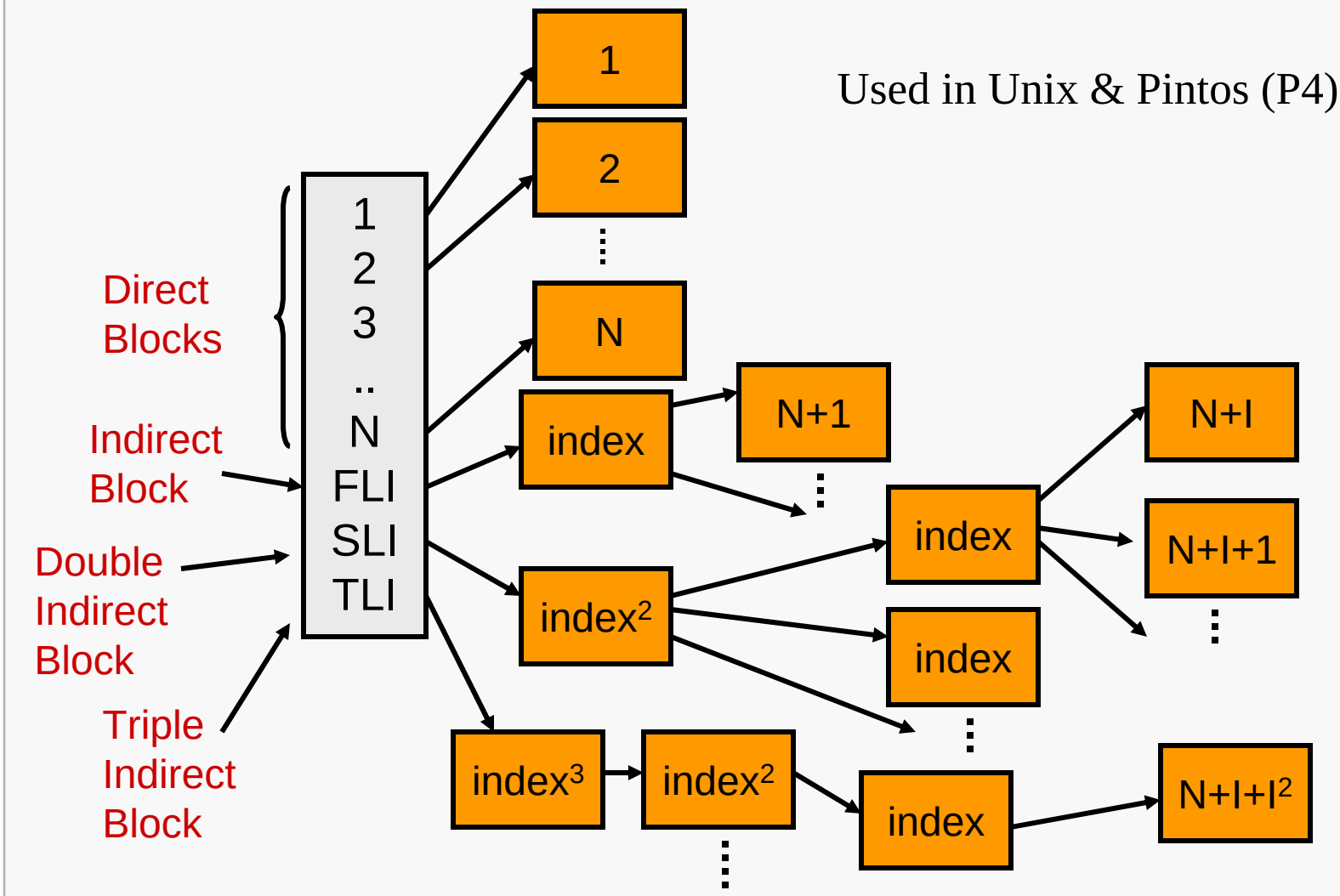


Single-index: specify maximum filesize, create index array, then note blocks in index

- Random access ok – one translation step
- Sequential access requires more seeks – depending on contiguous allocation

Drawback: hard to grow beyond maximum

Multi-Level Indices



Multi-Level Indices

If filesz < N * BLKSIZE, can store all information in direct block array

- Biased in favor of small files (ok because most files are small…)

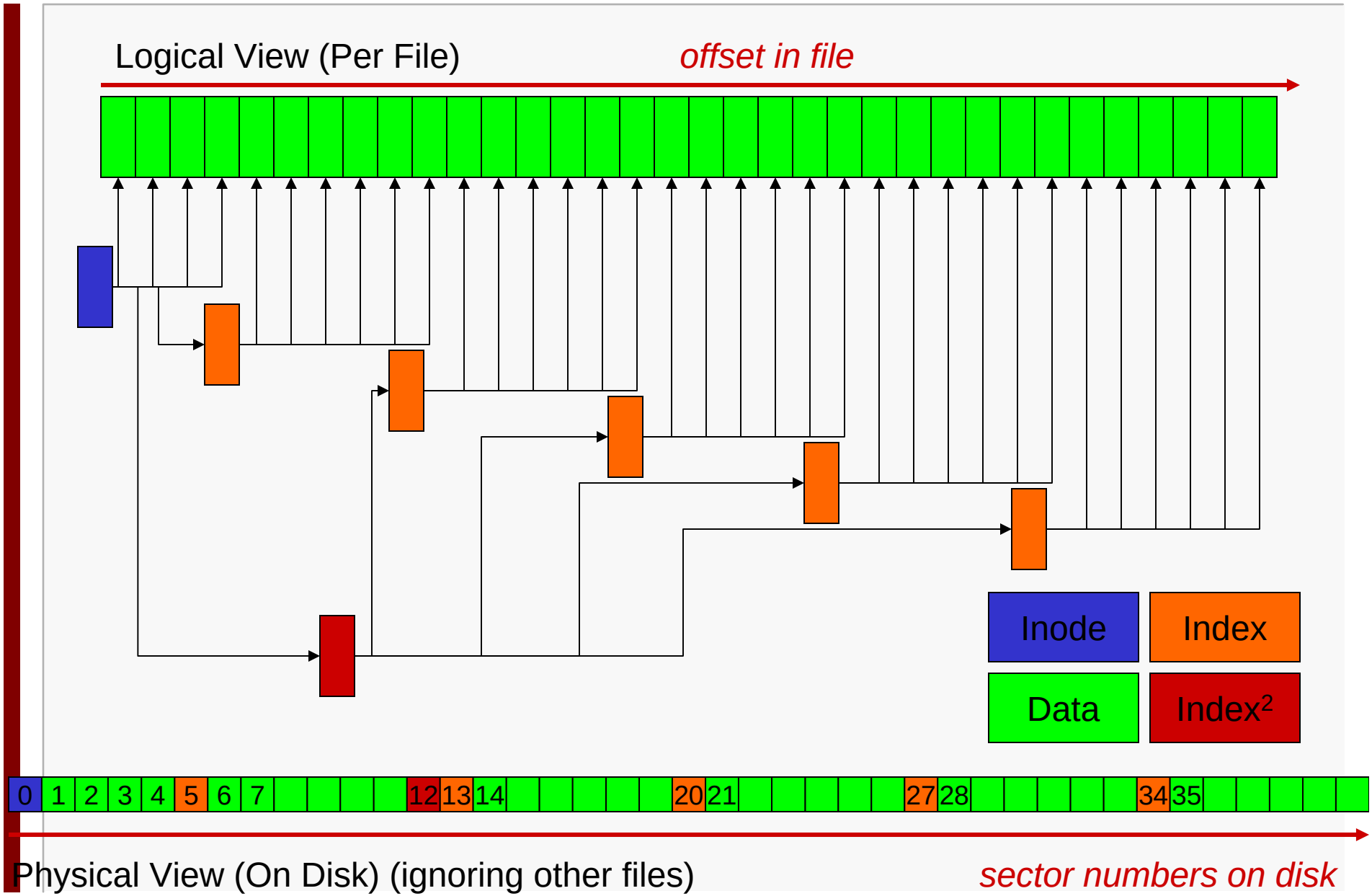
Assume index block stores I entries

- If filesz < (I + N) * BLKSIZE, 1 indirect block suffices

Q.: What's the maximum size before we need triple-indirect block?

Q.: What's the per-file overhead (best case, worst case?)

View



Details