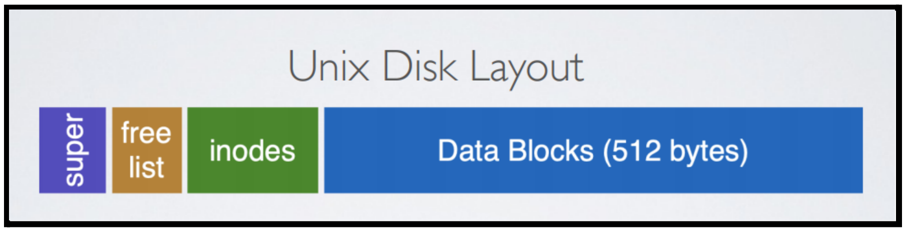
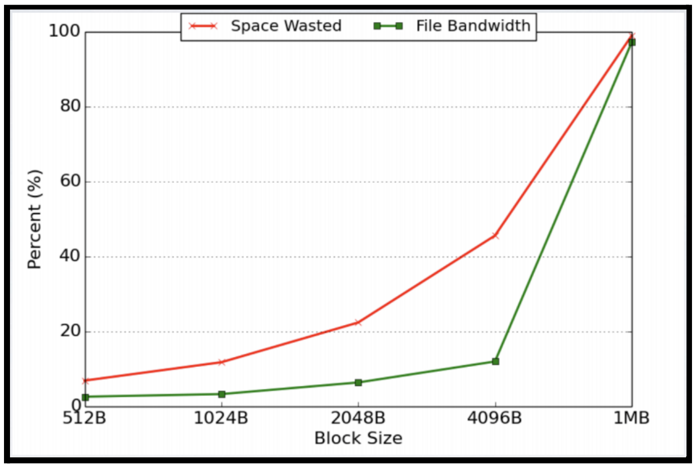
Lecture Notes:

- **Improving Performances with BSD Fast File System:**
- This is the original Unix FS layout:



- It is slow on hard disk drive - only gets 2% of disk maximum (20Kb/sec) even for sequential disk transfers.
- There were 3 problems to why it was so slow:
    1. In the original Unix File System, the blocks were too small (512 bytes).
    - Because the file index was too large, it required more indirect blocks but the transfer rate was low (get one block at time).
    2. Unorganized freelist:
    - Consecutive file blocks are not close together. This meant the OS had to pay a seek cost for even sequential access. Another issue was **aging**, which is when the freelist becomes fragmented over time.
    3. Poor locality:
    - The inodes were far from data blocks.
    - Furthermore, inodes for directories were not close together. This meant poor enumeration performance, for commands like ls.
- **Problem 1 - blocks are too small:**
- Bigger block increases bandwidth, but increases internal fragmentation as well.
- E.g.



- The solution is to use fragments.
- Allow large blocks to be chopped into small ones called "fragments".
- Ensure fragments are only used for little files or ends of files.
- Ensure that the fragment size is specified at the time that the file system is created.
- Limit the number of fragments per block to 2, 4, or 8.
- This solution has a high transfer speed for larger files and low wasted space for small files or ends of files.

- **Problem 2 - Unorganized Freelist:**
- Unorganized freelist leads to random allocation of sequential file blocks overtime.
- The solution is to use bitmaps.
- Periodical compact/defragment disk but locks up disk bandwidth during operation.
- Keep adjacent free blocks together on the freelist but costly to maintain,
- Each bit indicates whether the block is free.
- Easier to find contiguous blocks.
- Are small, so usually keep the entire thing in memory.
- The time to find free blocks increases if there are fewer free blocks.
- Here's the algorithm:
    - Allocate a block close to block x.
    - Check for blocks near bmap[x/32].
    - If the disk is almost empty, it will be likely to find one near.
    - As the disk becomes full, search becomes more expensive and less effective.
- **Problem 3 - Poor Locality:**
- The solution is to use a **cylinder group**.
- Group sets of consecutive cylinders into cylinder groups.
- Using this way, the OS can access any block in a cylinder without performing a seek as the next fastest place is the adjacent cylinder.
- Tries to put everything related in the same cylinder group.
- Tries to put everything not related in different groups.
- If you access one block, you'll probably access the next block, too, so let's try to put sequential blocks in adjacent sectors.
- If you look at an inode, you'll most likely look at the file data too, so let's try to keep the inode in the same cylinder as the file data.
- If you access one file in a directory, you'll probably frequently access the other files in that directory, so let's try to keep all inodes in a directory in the same cylinder group.
- How to keep inode close to the data block? → Use groups across disks and allocate inodes and data blocks in the same group. This way, each cylinder group is basically a mini-Unix file system.
- **Improving Reliability with Log-Structured File system (LFS) and Journaling File System (ext3):**
- What happens when there's power loss or a system crash:
    - Sectors (but not a block) are written atomically by the hard drive device.
    - But an FS operation might modify several sectors, such as metadata blocks (free bitmaps and inodes) and data blocks. Hence, a crash has a high chance of corrupting the file system.
- Solution 1 - Unix fsck (File System Checker):
    - When the system boots, check the system looking for inconsistencies and try to fix errors automatically.
    - However, this cannot fix all crash scenarios.
    - Furthermore, it has poor performance. Sometimes it takes hours to run on large disk volumes and does fsck have to run upon every reboot (Not well-defined consistency)?
- Solution 2 - Log Structure File System (LFS) or Copy-On-Write Logging:
    - The idea is to treat the disk like a tape-drive.
    - Buffer all data (including inode) in memory segment.
    - Write buffered data to a new segment on disk in a sequential log.
    - Existing data is not overwritten as the segment is always written in a free location.
    - Best performance from disk for sequential access.

- In the original Unix File System, the inode table is placed at a fixed location. However, in a Log-structured File System, the inode table is split and spread-out on the disk. Hence, the LFS needs to use an inode map (imap) to map the inode number with its location on disk.
- The OS must have some fixed and known location on disk to begin a file lookup. The check-point region (CR) contains a pointer to the latest pieces of the inode map. The CR is updated periodically (every 30 sec or so) to avoid degrading the performances.
- LFS - Crash recovery:
    - The check-point region (CR) must be updated atomically.
    - The LFS keeps two CRs and writing a CR is done in 3 steps:
        1. Writes out the header with a timestamp #1.
        2. Writes the body of the CR.
        3. Writes one last block with another timestamp #2.
    - A crash can be detected if timestamp #1 is after #2.
    - The LFS will always choose the most recent and valid CR.
    - All logs written after a successful CR update will be lost in case of a crash.
- LFS - Disk Cleaning (a.k.a Garbage Collection):
    - The LFS leaves an old version of file structures on disk.
    - The LFS keeps information of the version of each segment and runs a disk cleaning process.
    - A cleaning process removes old versions by compacting contiguous blocks in memory.
    - That cleaning process runs when the disk is idle or when running out of disk space.

- Solution 3 - Journaling or Write-Ahead Logging:
    - Write the "intent" down to disk before updating the file system.
    - When a crash occurs, look through the log to see what was going on and use the contents of the log to fix file system structures.
    - The process is called **recovery**.