6. Advanced File Systems

Special Topics in Computer Systems:
Modern Storage Systems
(SE820-01)

Instructor:
Prof. Sungjin Lee (sungjin.lee@dgist.ac.kr)

What is a File System?

- **Provides a virtualized logical view of information stored on various storage media, such as disks, tapes, and flash-based SSDs**

- **Two key abstractions have developed over time in the virtualization of storage**
  - ***File***: A linear array of bytes, each of which you can read or write
    - Its contents are defined by a creator (e.g., text and binary)
    - It is often referred to as its *inode* number
  - ***Directory***: A special file that is a collection of files and other directories
    - Its contents are quite specific – it contains a list of (user-readable name, inode #) pairs (e.g., (“foo”, 10))
    - It has a hierarchical organization (e.g., tree, acyclic-graph, and graph)
    - It is also identified by an inode number

Operations on Files and Directories

POSIX Operations on Files

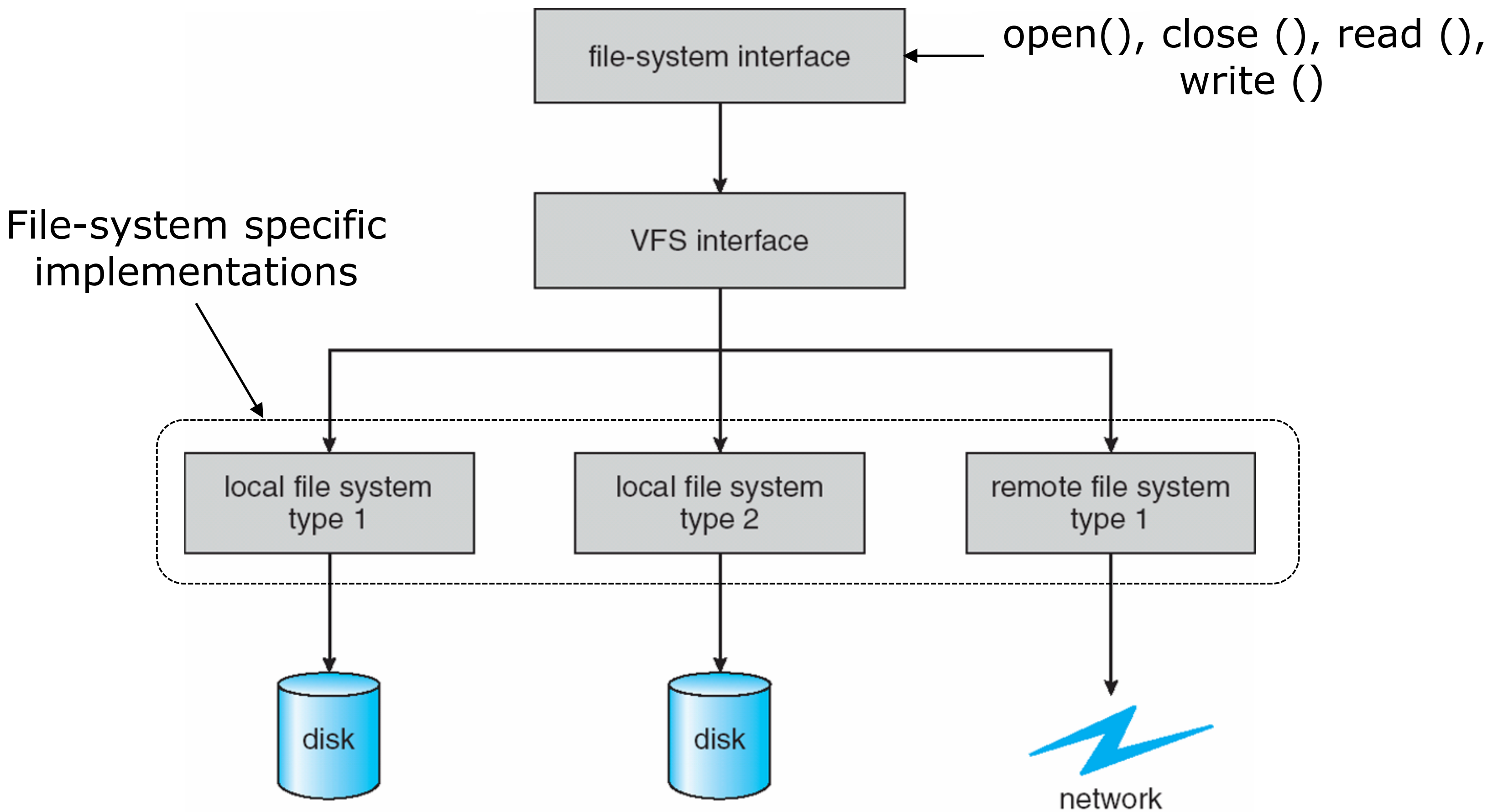
POSIX APIs	Description
creat ()	Create a file
open ()	Create/open a file
write ()	Write bytes to a file
read ()	Read bytes from a file
lseek ()	Move byte position inside a file
unlink ()	Remove a file
truncate ()	Resize a file
close ()	Close a file
	…

POSIX Operations on Directories

POSIX APIs	Description
opendir ()	Open a directory for reading
closedir ()	Close a directory
readdir ()	Read one directory entry
rewinddir ()	Rewind a directory so it can be reread
mkdir ()	Create a new directory
rmdir ()	Remove a directory
	…

Virtual File System

- The POSIX API is to the VFS interface, rather than any specific type of file system



Evolution of File System

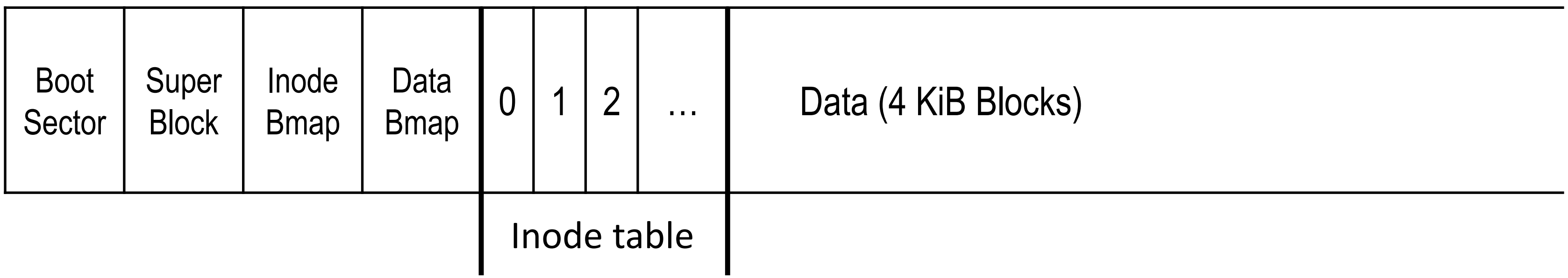
- UNIX File System
- Fast File System
- Journaling File System
- Extents File System
- Log-structured File Systems

Evolution of File System

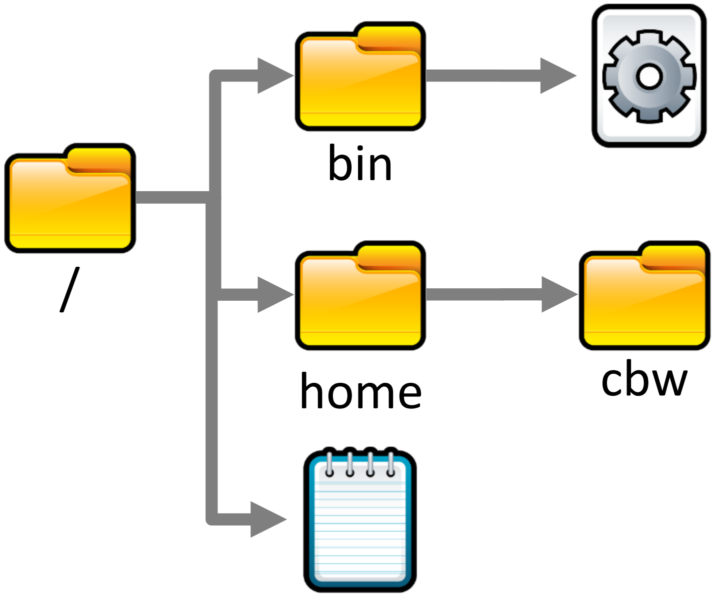
- UNIX File System ← ext1
- Fast File System ← ext2
- Journaling File System ← ext3
- Extents File System ← ext4
- Log-structured File Systems ← WAFL, F2FS, …

UNIX File System

- A traditional file system first developed for UNIX systems

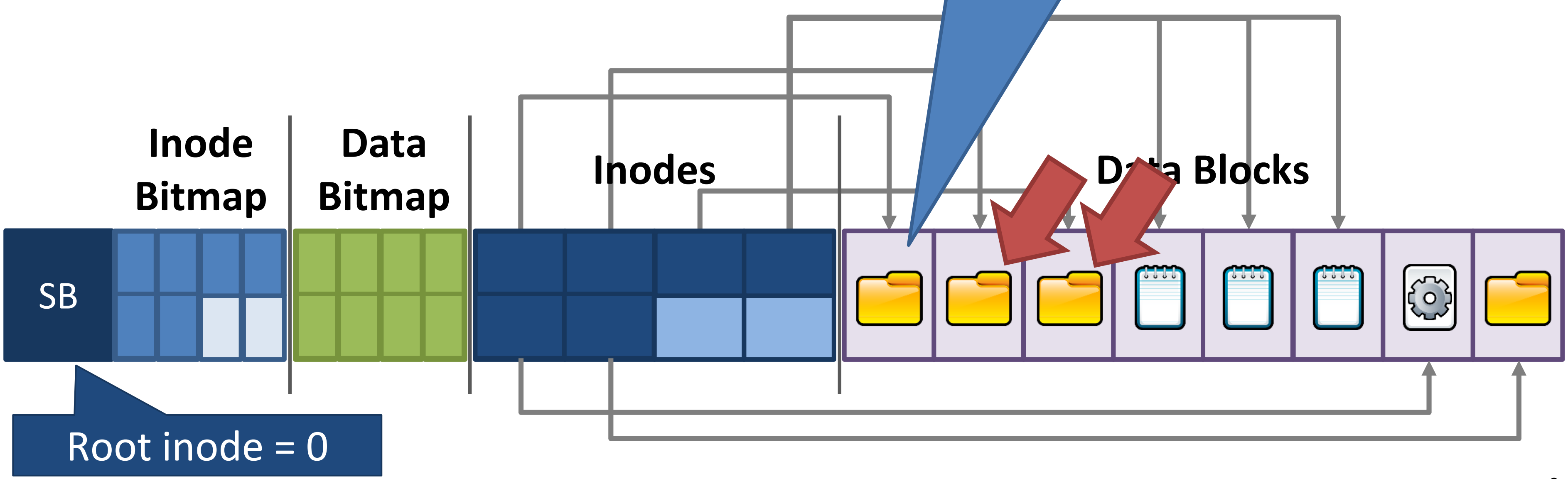


  - **Boot sector**: Information to be loaded into RAM to boot up the OS
  - **Superblock**: File system's metadata (e.g., file system type, size, …)
  - **Inode & data Bmaps**: Keep the status of blocks belonging to an inode table and data blocks
  - **Inode table**: Keep file's metadata (e.g., size, permission, …) and data block pointers
  - **Data blocks**: Keep users' file data

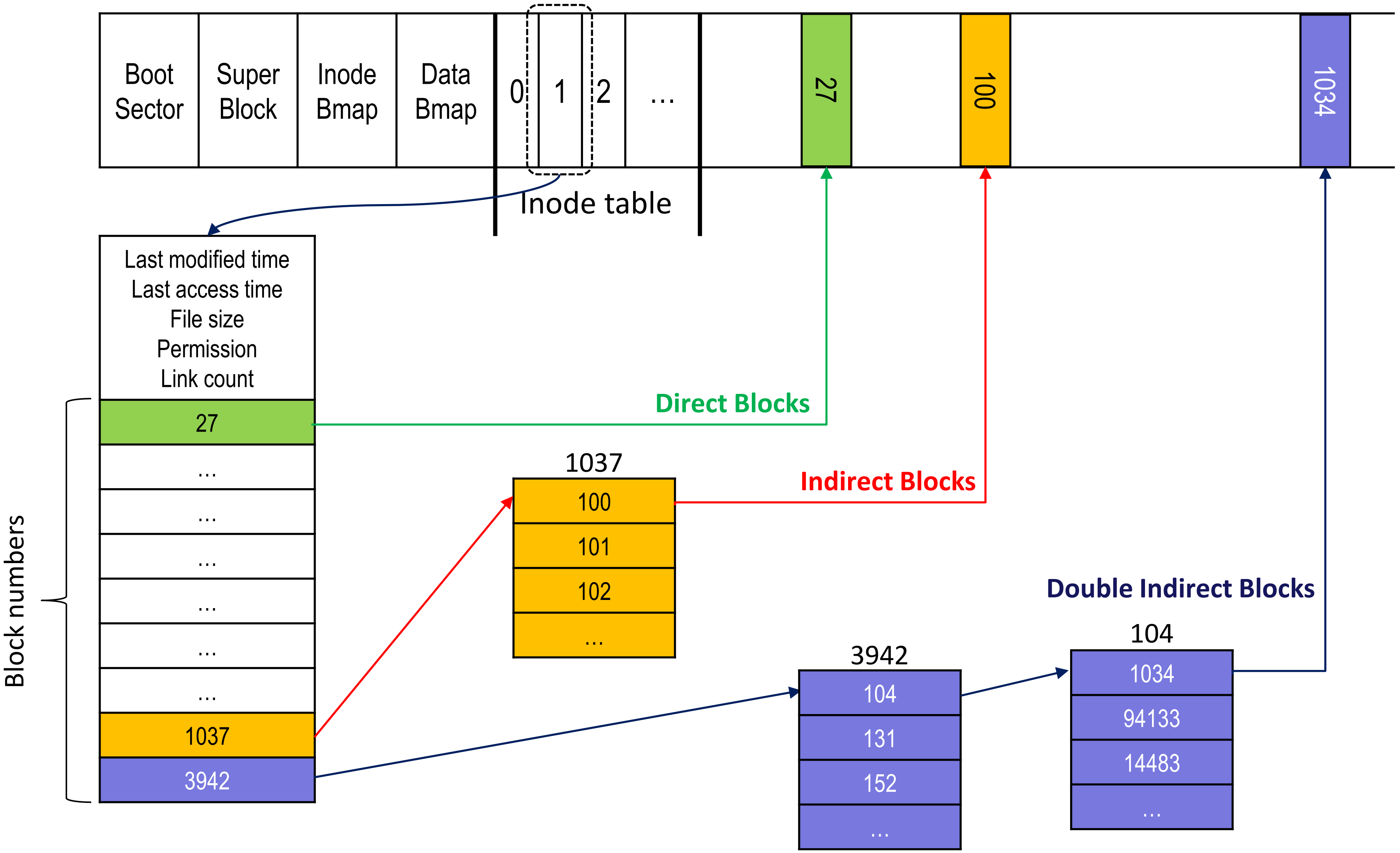


- Directories are files
- Contains the list of entries in the directory

- Each inode can directly point to 12 blocks
- Can also indirectly point to blocks at 1, 2, and 3 levels of depth

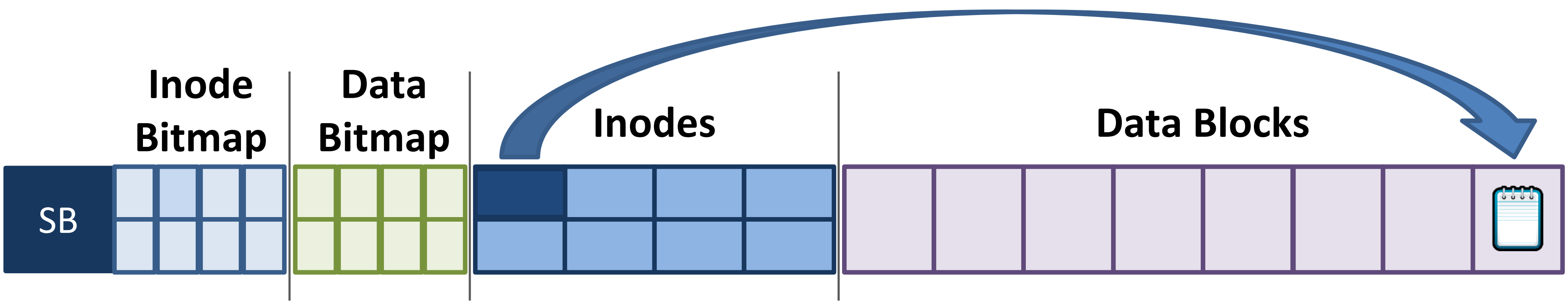


Inode & Block Pointers



Good and Bad of UNIX File System

- **The Good – ext file system (inodes) support:**
  - All the typical file/directory features
  - Hard and soft links
- **The Bad: poor locality**
  - It is not optimized for spinning disks
  - inodes and associated data are far apart on the disk!



Evolution of File System

- UNIX File System ← ext1
- Fast File System ← ext2
- Journaling File System ← ext3
- Extents File System ← ext4
- Log-structured File Systems ← WAFL, F2FS, …

Fast File System (FFS)

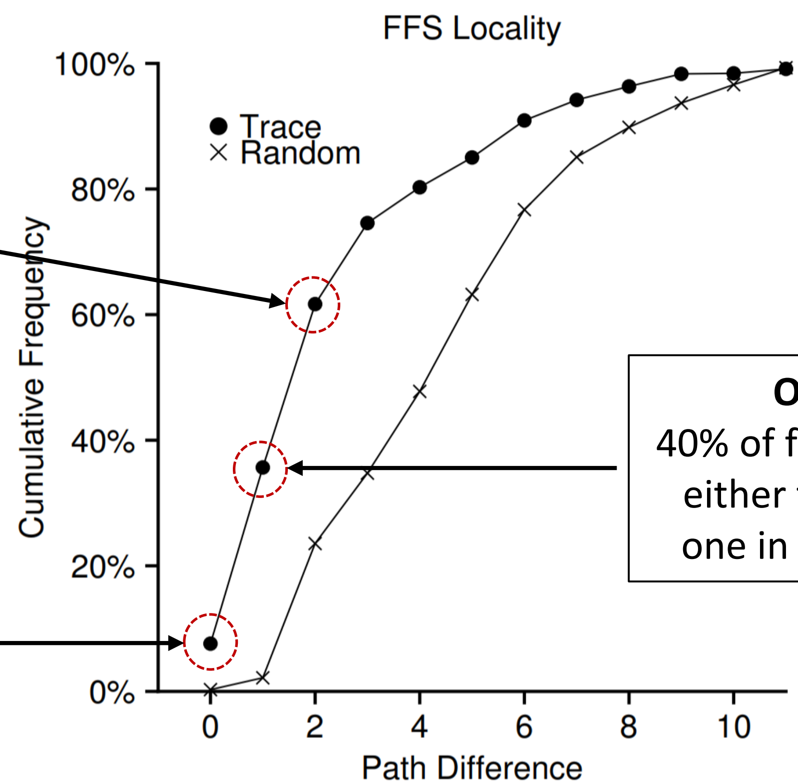
- **Fast File System (FFS) developed at Berkeley in 1984**
  HDD의 하드웨어에 맞게 설계됨
  - First attempt at a disk aware file system
  - Optimized for performance on spinning disks

- **Observation:**
  - Processes tend to access files that are in the same (or close) directories

- **Key idea:**
  - place groups of directories and their files into cylinder groups
  - Introduced into ext2, called block groups

FFS Locality for SEER Traces

Observation 3:
About 65% of file accesses were shorter than two distances



Observation 2:
40% of file accesses were to either the same file or to one in the same directory

Observation 1:
7% of file accesses were to the same file

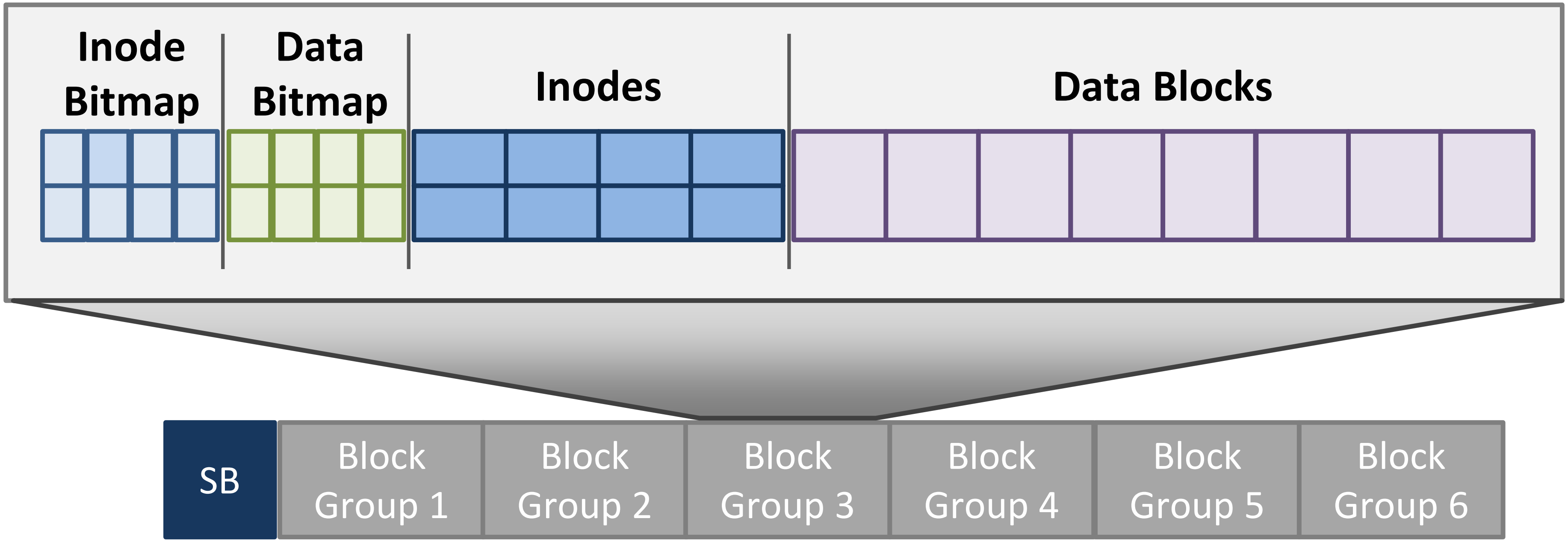
```
The distance of two file access is 1
proc/src/foo.c
proc/src/bar.c
```

```
The distance of two file access is 2
proc/src/foo.c
proc/obj/foo.o
```

Block Groups

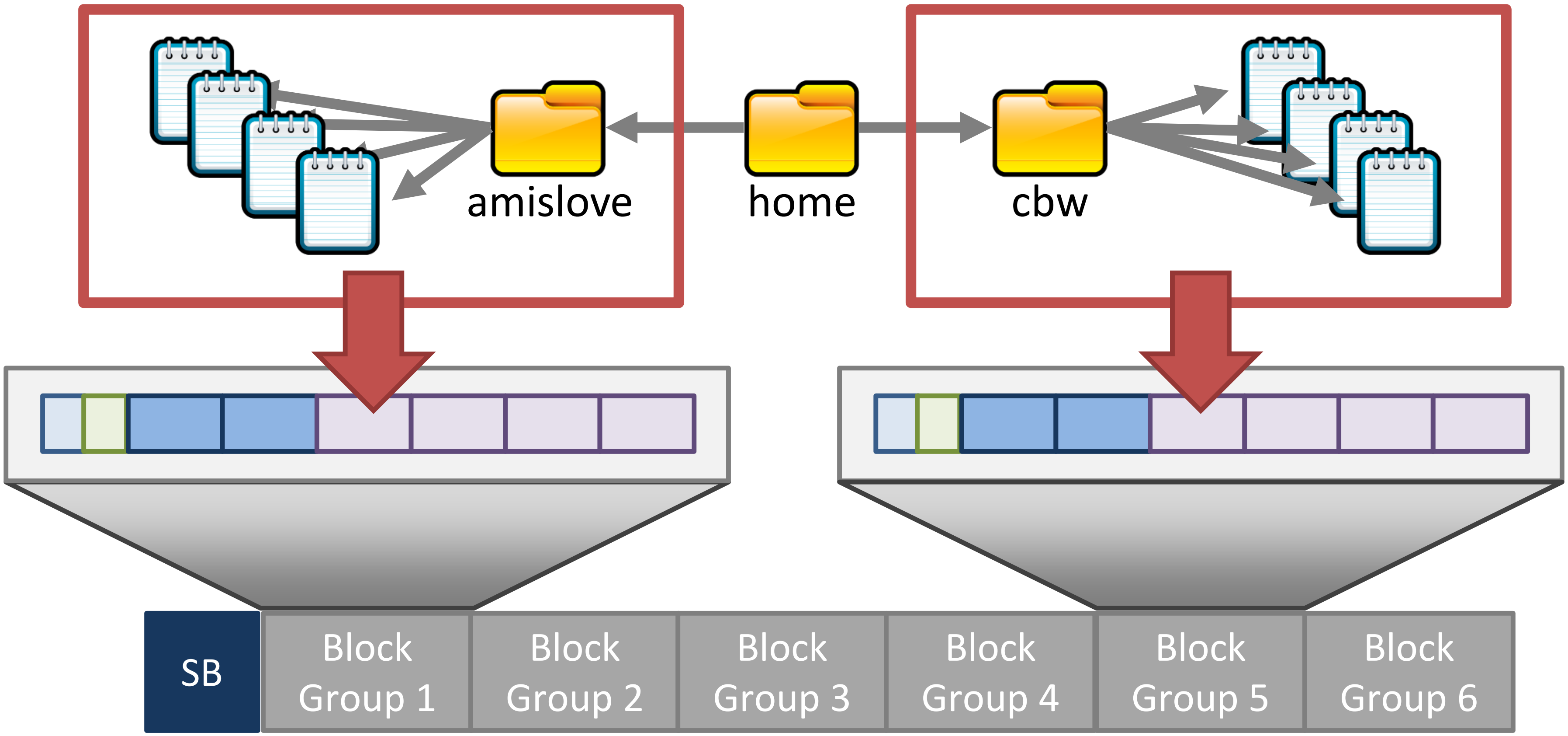
unix fs의 layer 자체를
block 하나마다 넣었다.
directory에 따라 locality가 있으므로
block 하나에 속한 file들은 빠르게 찾을 수 있다.

- **In ext, there is a single set of key data structures**
  - One data bitmap, one inode bitmap
  - One inode table, one array of data blocks
- **In ext2, each block group contains its own key data structures**



Allocation Policy

- ext2 attempts to keep related files and directories within the same block group



The Good and the Bad of FFS

- **The good – ext2 supports:**
  - All the features of ext with even better performance because of increased spatial locality

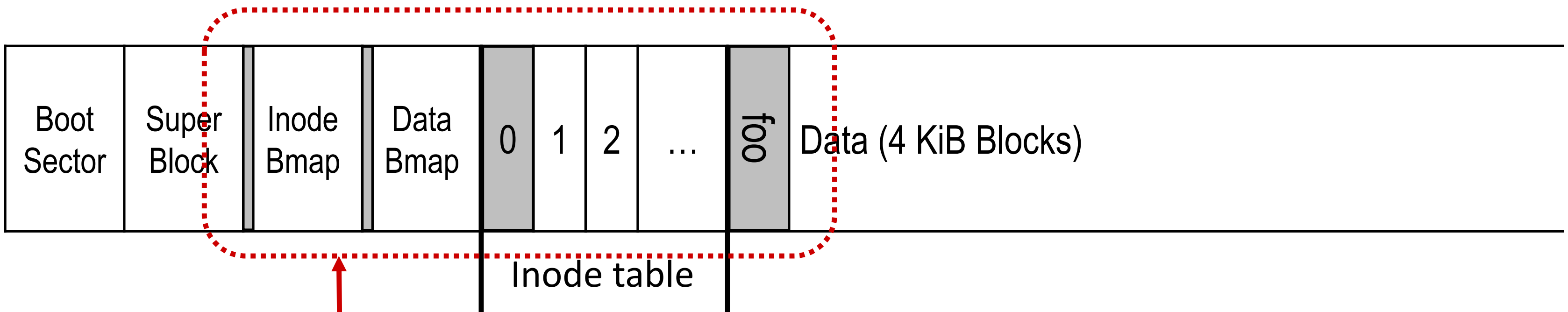
- **The bad**
  - Large files must cross block groups
  - As the file system becomes more complex, the chance of file system corruption grows
    - E.g. invalid inodes, incorrect directory entries, etc.

Consistent Update Problem

- What happens if sudden power loss occurs while writing data to a file

inode

write (10, “foo”, strlen (“foo”));



ext2는 이 문제를 복구하지 않아서 빠르다.

The file system will be inconsistent!!!

→ **Consistent update problem**

Evolution of File System

- UNIX File System ← ext1
- Fast File System ← ext2
- Journaling File System ← ext3
- Extents File System ← ext4
- Log-structured File Systems ← WAFL, F2FS, …

How to Ensure Consistency after a Crash?

- **Strategy 1: Don’t bother to ensure consistency**
  - Accept that the file system may be inconsistent after a crash
  - Run a program that fixes the file system during bootup
  - ***File system checker (fsck)*** 부팅할 때, fsck를 실행하여 inconsistent problem이 있는지 확인한다.

- **Strategy 2: Use a transaction log to make multi-writes atomic**
  - Log stores a history of all writes to the disk
  - After a crash, the log can be “replayed” to finish updates
  - ***Journaling file system*** DB 쪽에서 나온 아이디어. WAL

Strategy #1:
File System Checker

- **Key idea: fix inconsistent file systems during bootup**
  - Unix utility called *fsck* (*chkdsk* on Windows)
  - Scans the entire file system multiple times, identifying and correcting inconsistencies

- **Why during bootup?**
  - No other file system activity can be going on
  - After fsck runs, bootup/mounting can continue

File System Checker Tasks

- **Superblock:**
  - Validate the superblock, replace it with a backup if it is corrupted
- **Free blocks and inodes:**
  - Rebuild the bitmaps by scanning all inodes
- **Reachability:**
  - Make sure all inodes are reachable from the root of the file system
- **Inodes:**
  - Delete all corrupted inodes, and rebuild their link counts by walking the directory tree
- **Directories:**
  - Verify the integrity of all directories
- … and many other minor consistency checks

The Good and the Bad of fsck

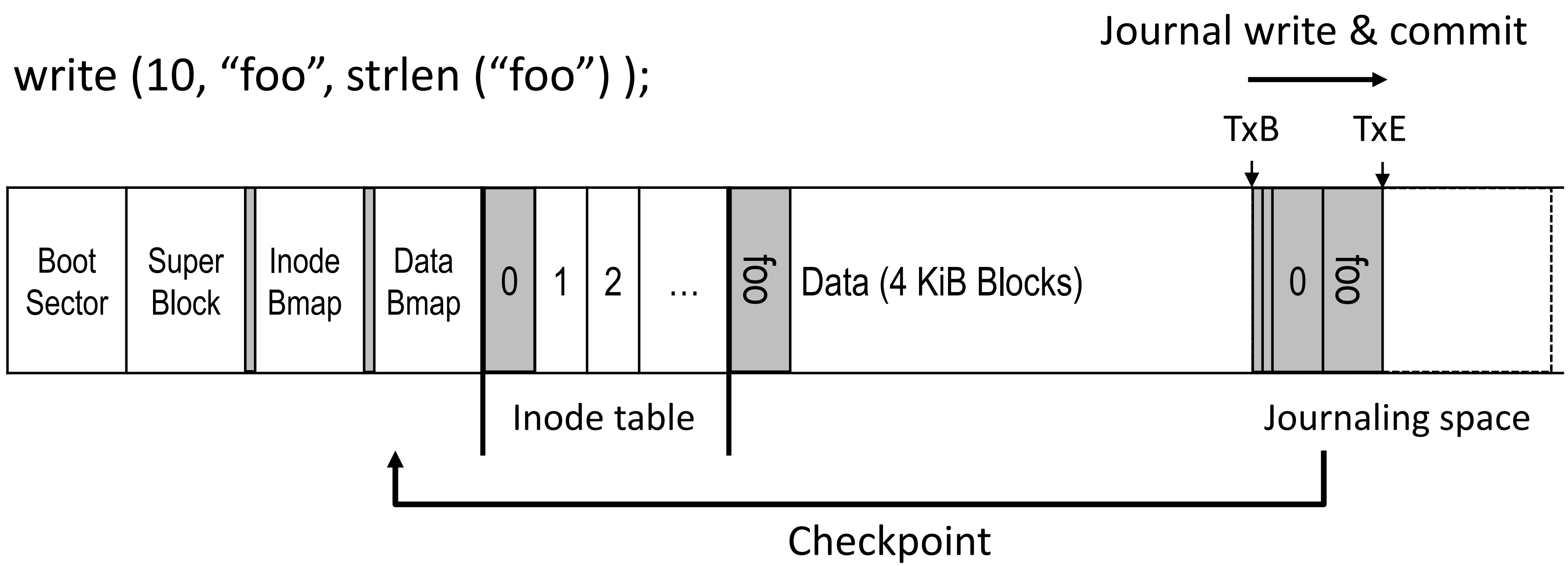
- **Advantages of *fsck***
  - Doesn’t require the file system to do any work to ensure consistency
  - Makes the file system implementation simpler

- **Disadvantages of *fsck***
  - Very complicated to implement the *fsck* program
    - Many possible inconsistencies that must be identified
    - Many difficult corner cases to consider and handle
  - *fsck* is ***super slow***
    - Scans the entire file system multiple times
    - Imagine how long it would take to fsck a 40 TB RAID array

Strategy #2:
Journaling File System

- Journaling file systems address the consistent update problem by adopting an idea of *write-ahead logging (or journaling)* from database systems
- Ext3, Ext4, ReiserFS, XFS, and NTFS are based on journaling

write (10, “foo”, strlen (“foo”));



Write-Ahead Log (WAL)

- **Key idea: writes to disk are first written into a log**
  - After the log is written, the writes execute normally
  - In essence, the log records transactions

- **What happens after a crash…**
  - If the writes to the log are interrupted?
    - The transaction is incomplete
    - The user's data is lost, but the file system is consistent
  - If the writes to the log succeed, but the normal writes are interrupted?
    - The file system may be inconsistent, but…
    - The log has exactly the right information to fix the problem

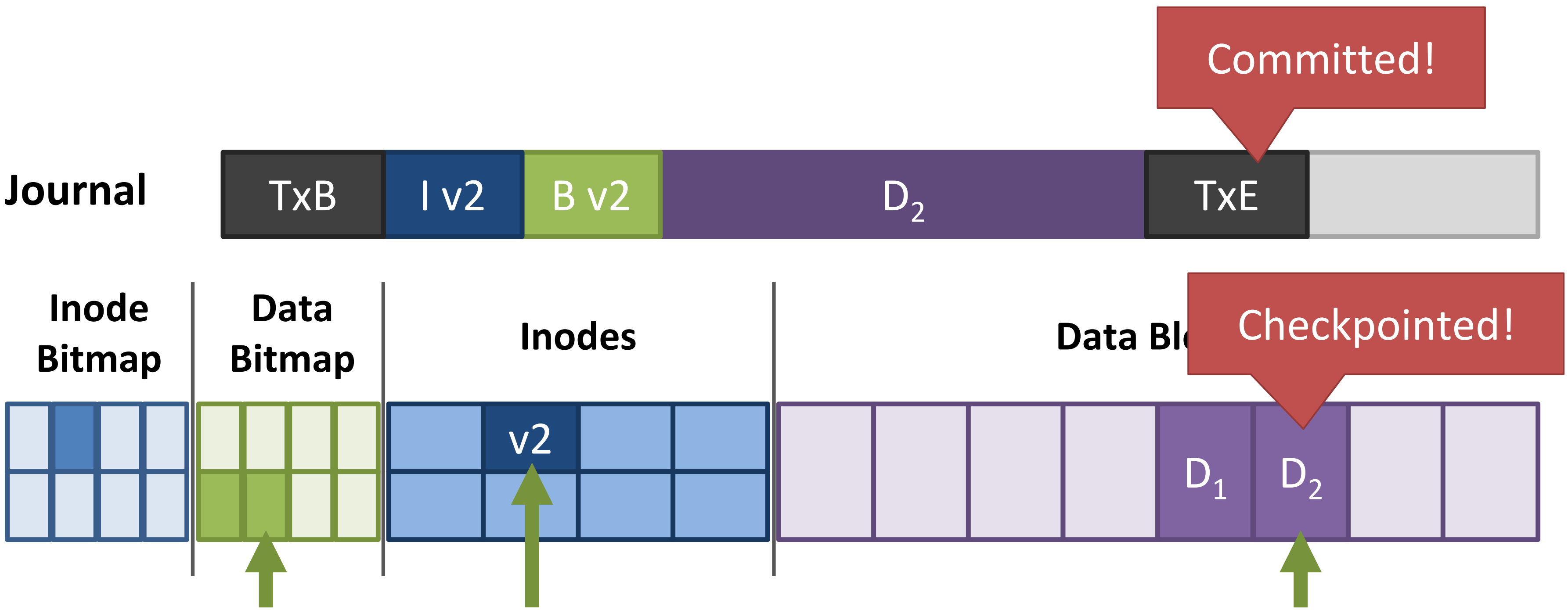
Data Journaling Example

- **Here, assume that we are appending to a file**
  - Three writes: inode v2, data bitmap v2, data D_2
- **Before executing these writes, first log them**

1. Begin a new transaction with a unique ID=k
2. Write the updated meta-data block(s)
3. Write the file data block(s)
4. Write an end-of-transaction with ID=k

Commits and Checkpoints

- A transaction is committed after all writes to the log are complete
- After a transaction is committed, the OS checkpoints the update



- Final step: **free** the checkpointed transaction

checkout 중에 failure가 나면 해당 transaction을 처음부터 다시 checkout하면 된다.

The Good and the Bad of Journaling

- **Advantages of journaling**
  - Robust, fast file system recovery
    - No need to scan the entire journal or file system
  - Relatively straight forward to implement

- **Disadvantages of journaling**
  - Write traffic to the disk is ***doubled***
    - Especially the file data, which is probably large
  - Deletes are very hard to correctly log

Making Journaling Faster

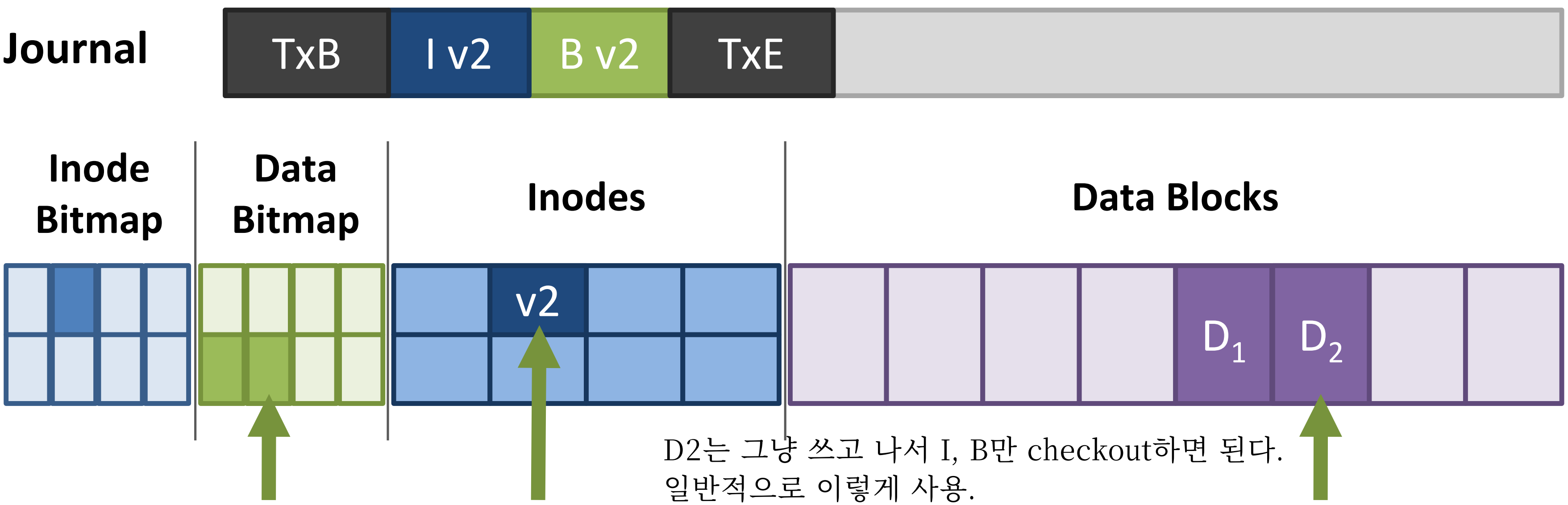
- **Journaling adds a lot of write overhead**
- **OSes typically batch updates to the journal**
  - Buffer sequential writes in memory, then issue one large write to the log
  - Example: ext3 batches updates for 5 seconds

memory에 buffering해두었다가
모아서 한번에 batch updates
그러나 손실 위험

- **Tradeoff between performance and persistence**
  - Long batch interval = fewer, larger writes to the log
    - Improved performance due to large sequential writes
  - But, if there is a crash, everything in the buffer will be lost

Meta-Data Journaling

- **The most expensive part of data journaling is writing the file data twice**
  - Meta-data is small (~1 sector), file data is large
- **ext3 implements meta-data journaling**



D2는 그냥 쓰고 나서 I, B만 checkout하면 된다.
일반적으로 이렇게 사용.

Journaling Wrap-Up

- **Today, most OSes use journaling file systems**
  - ext3/ext4 on Linux
  - NTFS on Windows
- **Provides excellent crash recovery with relatively low space and performance overhead**
- **Next-gen OSes will likely move to file systems with copy-on-write semantics**
  - btrfs and zfs on Linux

Evolution of File System

- UNIX File System ← ext1
- Fast File System ← ext2
- Journaling File System ← ext3
- **Extents File System ← ext4**

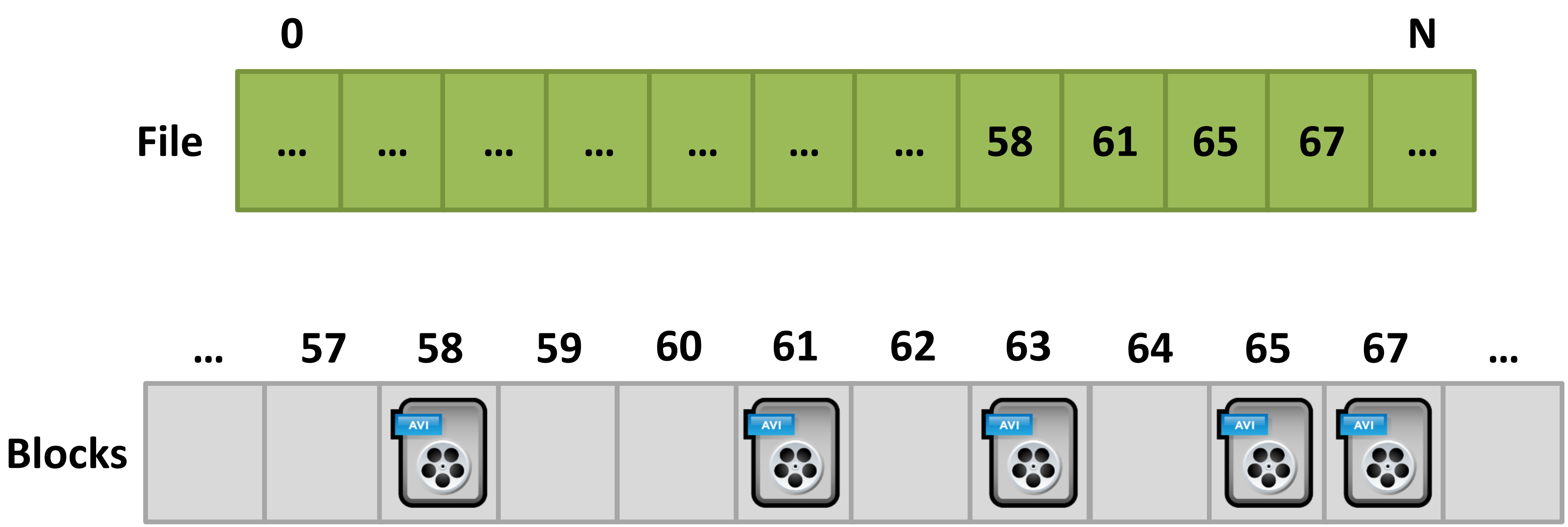
ext3에서 directory data structure와 기타 등등을 약간 수정함.

- Log-structured File Systems ← WAFL, F2FS, …

Revisiting Inodes

파일은 logical하게는 contiguous하지만, physical하게는 아니다.
따라서 ext3에서는 Fragmentation 문제가 발생할 수 있고,
inode가 해당 블록들을 다 가리켜야 한다.

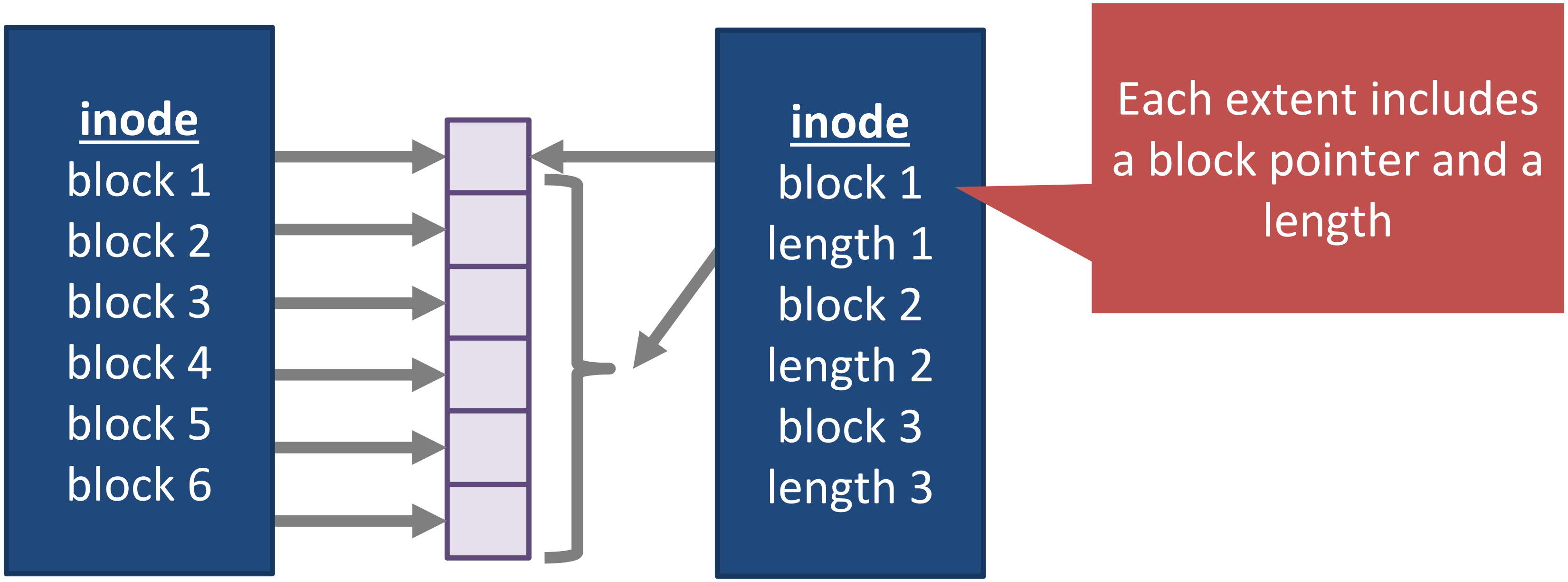
- **Inodes use indirection to acquire additional blocks of pointers**
- **Problem: inodes are not efficient for large files**
  - e.g., For a 100MB file, you need 25600 block pointers (assuming 4KB blocks)
- **This is unavoidable if the file is 100% fragmented**
  - However, what if large groups of blocks are contiguous?



From Pointers to Extents

- **Modern file systems try hard to minimize fragmentation**
  - Since it results in many seeks, thus low performance
- **Extents are better suited for contiguous files**

contiguous file에 대해서는 length만으로 다음 블록을 가리킬 수 있다.



Revisiting Directories

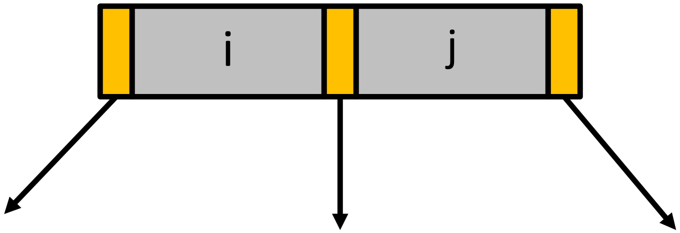
- **Each directory is a file with a list of entries**
  - Entries are not stored in sorted order
  - Some entries may be blank, if they have been deleted
- **Problem: searching for files in large directories takes O(n) time**
  - Practically, you can't store >10K files in a directory
  - It takes way too long to locate and open files

디렉토리 안에서 file entry를 찾으려면 linear search해야 했다.
그래서 너무 느리다.

From Lists to B-Trees

ext4부터는 디렉토리의 entry를 B - tree로 관리했다. 그래서 더 빨라졌다.

- **ext4 and NTFS encode directories as B-Trees to improve lookup time to O(log N)**
- **A B-Tree is a type of balanced tree optimized for disk**
  - Items are stored in sorted order in blocks
- **Suppose items *i* and *j* are in the root of the tree**
  - The root must have 3 children, since it has 2 items
  - The three child groups contain items $a < i$, $i < a < j$, and $a > j$



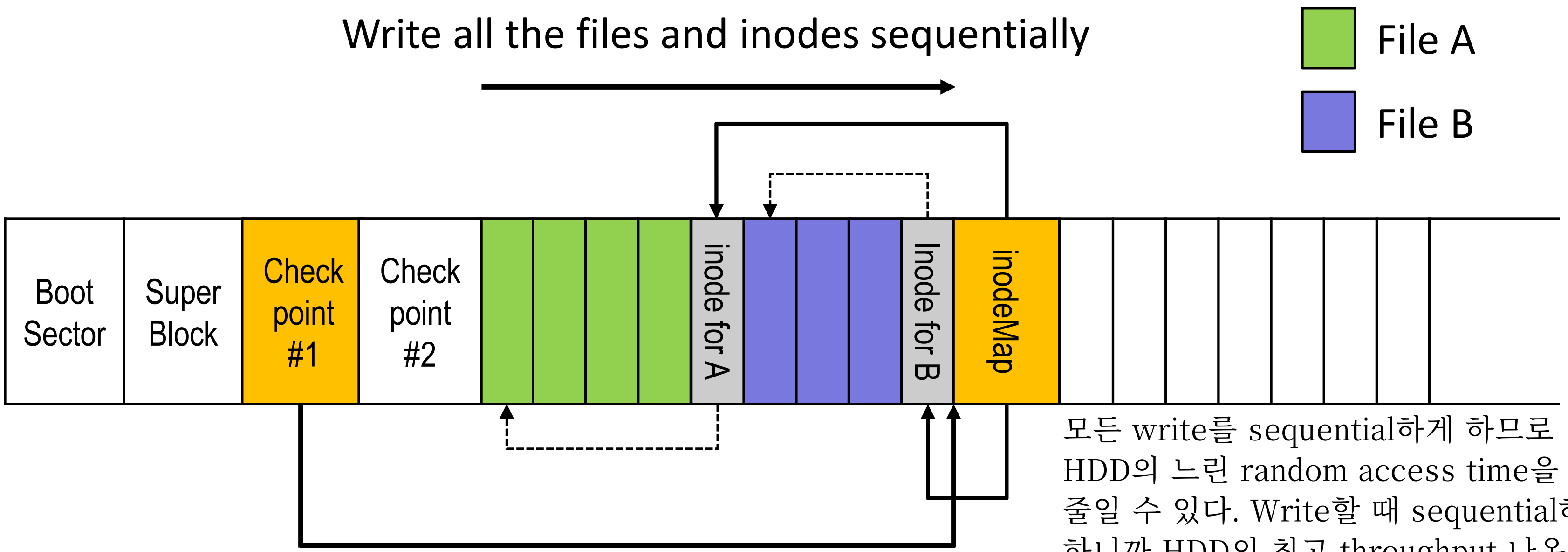
Evolution of File System

- UNIX File System ← ext1
- Fast File System ← ext2
- Journaling File System ← ext3
- Extents File System ← ext4
- **Log-structured File Systems ← WAFL, F2FS, …**

기존의 시스템과 다른 Log - structured FS
버클리에서 90년대 초에 제안됨.
상업적으로는 NetApp에서 자기거 FS에 CoW approach로 함

Log-structured File System

- Log-structured file systems (LFS) treat a storage space as a huge log, appending all files and directories sequentially
- The state-of-the-art file systems are based on LFS or CoW
  - e.g., Sprite LFS, F2FS, NetApp's WAFL, Btrfs, ZFS, …



모든 write를 sequential하게 하므로 HDD의 느린 random access time을 줄일 수 있다. Write할 때 sequential하게 하니까 HDD의 최고 throughput 나옴.

Log-structured File System (Cont.)

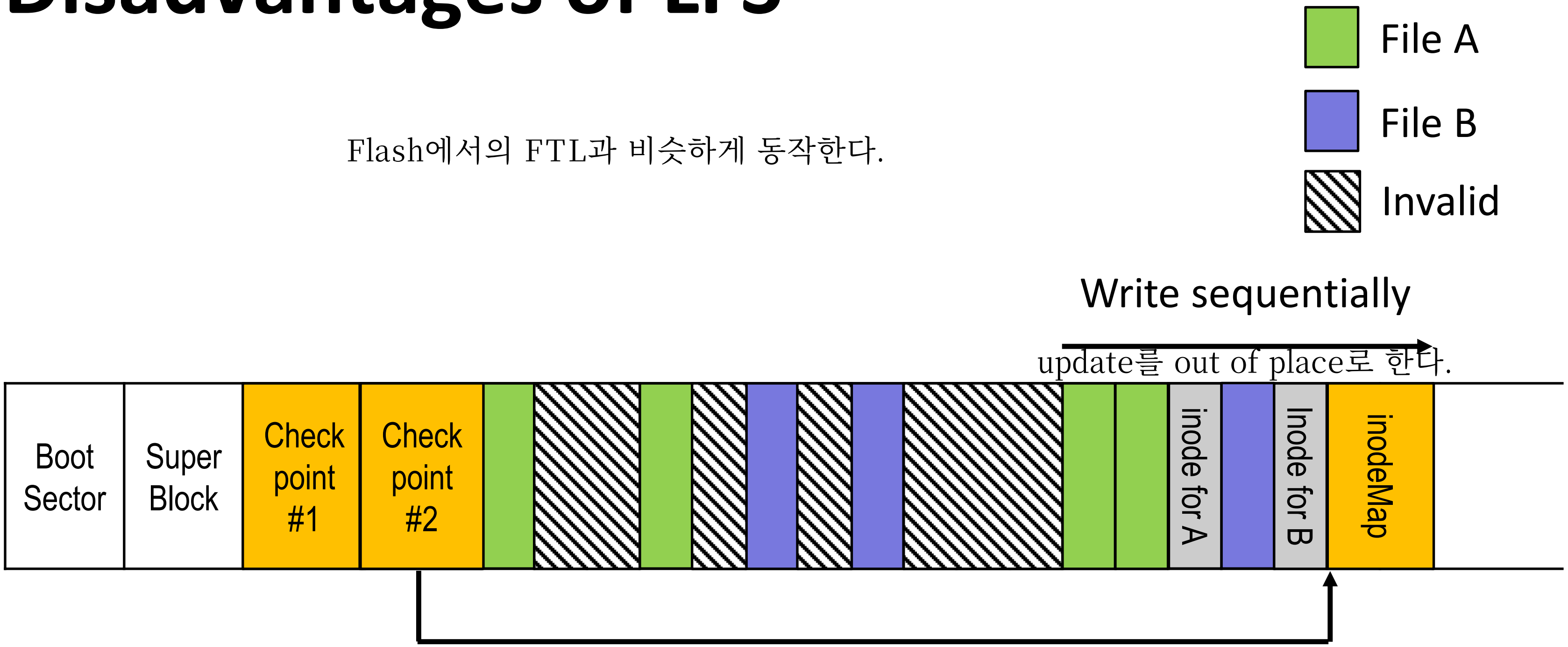
Journaling과 비슷하지만, Journaling space만 logging하는데
LFS는 data/inode도 log 방식으로.

- **Advantages**
  - (+) No consistent update problem
  - (+) No double writes – an LFS itself is a log!
  - (+) Provide excellent write performance – disks are optimized for sequential I/O operations
  - (+) Reduce the movements of disk headers further (e.g., inode update and file updates)

- **Disadvantages**
  - (–) Expensive garbage collection cost
  - (–) Slow read performance

Disadvantages of LFS

Flash에서의 FTL과 비슷하게 동작한다.



update를 out of place로 한다.

이전의 obsolete data는 GC되어야 한다.
다 읽으면서, live data만 뒤로 다시 append해야 한다.

- **Expensive garbage collection cost**: invalid blocks must be reclaimed for future writes; otherwise, free disk space will be exhausted
- **Slow read performance**: involve more head movements for future reads (e.g., when reading the file A)

Write Cost

- **Write cost with GC is modeled as follows**
  - Note: a segment (seg) is a unit of space allocation and GC

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

다 차게되면 GC할 것.
비었을 때는 그냥 cost=1일 것.
이 공식은 asymptotic. 평균적.

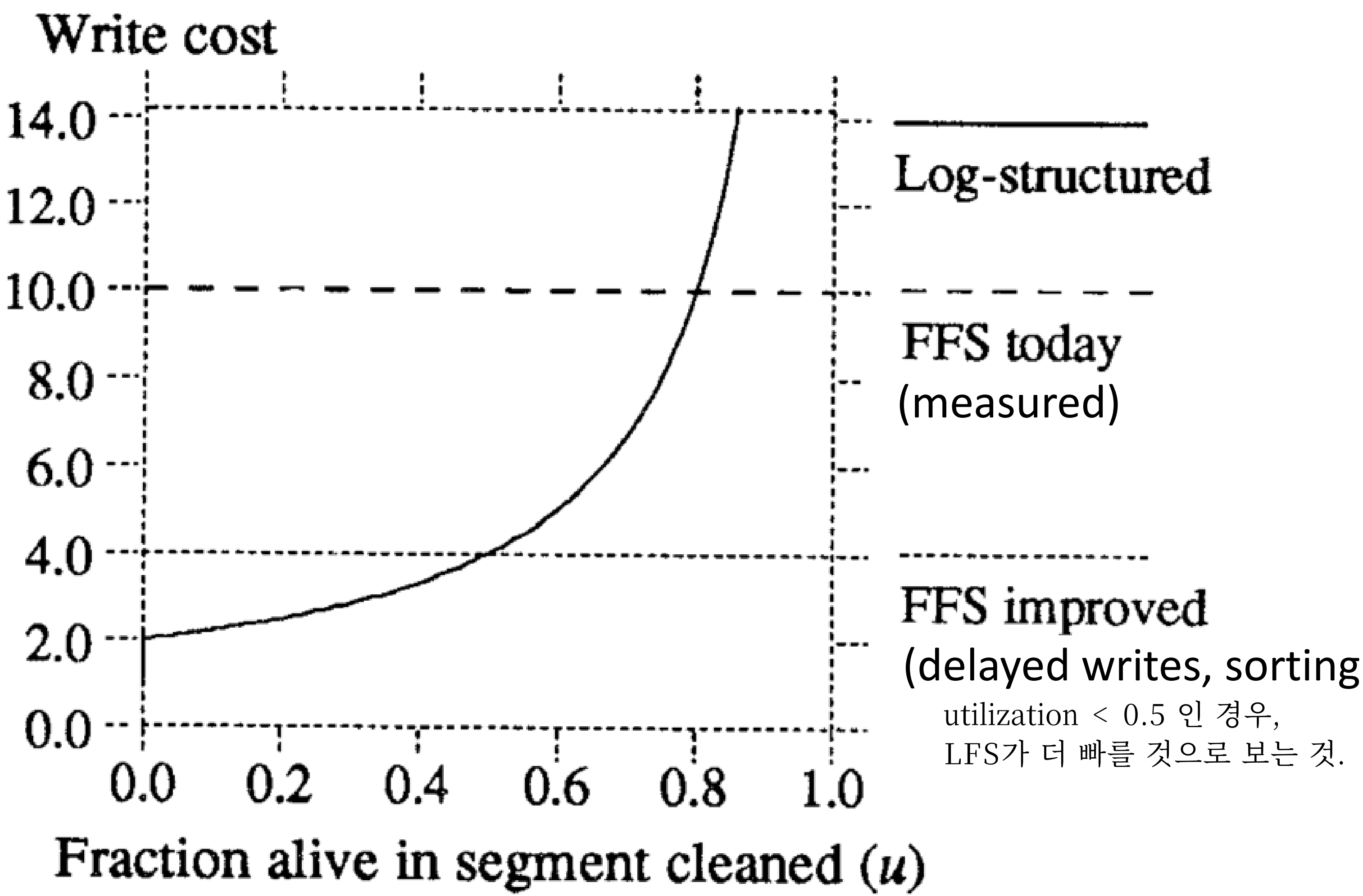
$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{\mathrm{N} + \mathrm{N}{*}u + \mathrm{N}{*}(1-u)}{\mathrm{N}{*}(1-u)} = \frac{2}{1-u}$$

u=0일 땐 1.
이 수식은 u!=0일 때 기준.

  - N is the number of segments
  - μ is the utilization of the segments ($0 \leq \mu < 1$)
  - If segments have no live data ($\mu = 0$), write cost becomes 1.0

Write Cost Comparison

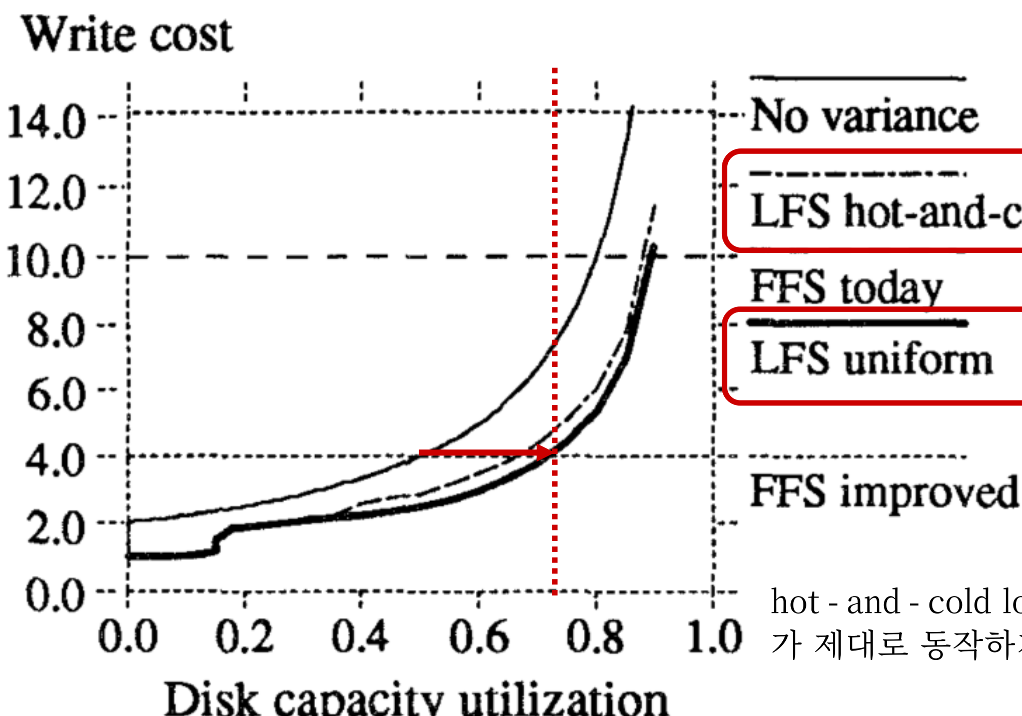


utilization < 0.5 인 경우,
LFS가 더 빠를 것으로 보는 것.

Greedy Policy

그렇다면, victim segments는 어떻게 선정?

- The cleaner chooses the *least-utilized segments* and *sorts the live data by age* before writing it out again
- Workloads: 4 KB files with two overwrite patterns
  - (1) **Uniform**: No locality – equal likelihood of being overwritten
  - (2) **Hot-and-cold**: Locality – 10:90



Worse than a system with no locality

No variance: 모든 block의 u가 평균인 경우. 그러나 Greedy로 하니까, 이렇게 나오지는 않을 것이므로 실제로는 더 낮은 cost일 것.

The variance in segment utilization

hot - and - cold locality를 가진 workload에 대해서는 greedy가 제대로 동작하지 않는다.

Cost-Benefit Policy

greedy는 locality를 고려하지 않으므로, hot인 애들은 어차피 지금 GC해도 금방 다시 invalidated 될 거니 GC 효율이 떨어진다. 따라서, 오래된 cold data를 GC하는 게 더 좋다. 따라서 age도 인자로 넣어야 한다.

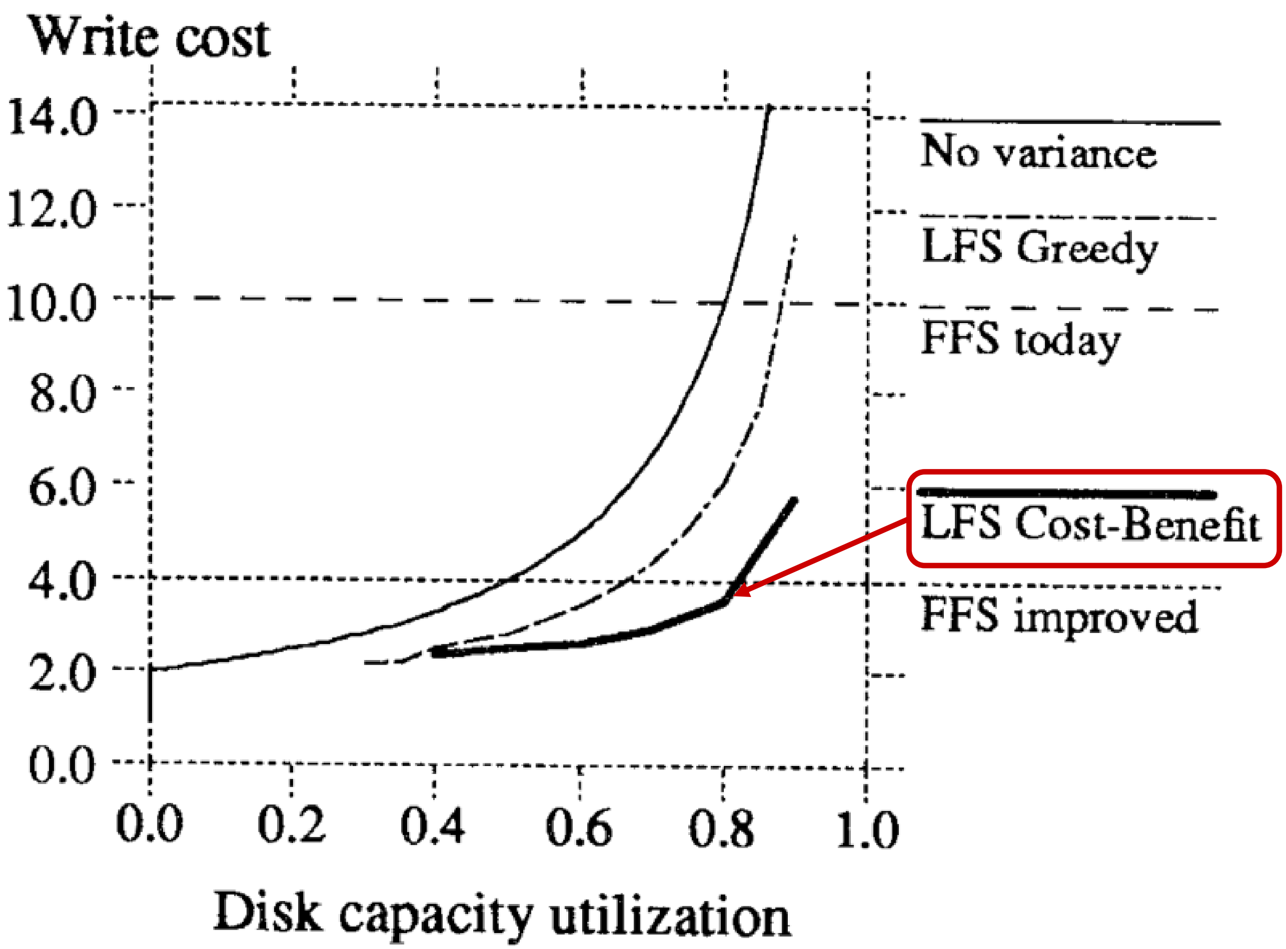
- Hot segments are frequently selected as victims even though their utilizations would drop further
  - It is necessary to delay cleaning and let more of the blocks die
  - On the other hand, free space in cold segments are valuable

- Cost-benefit policy:

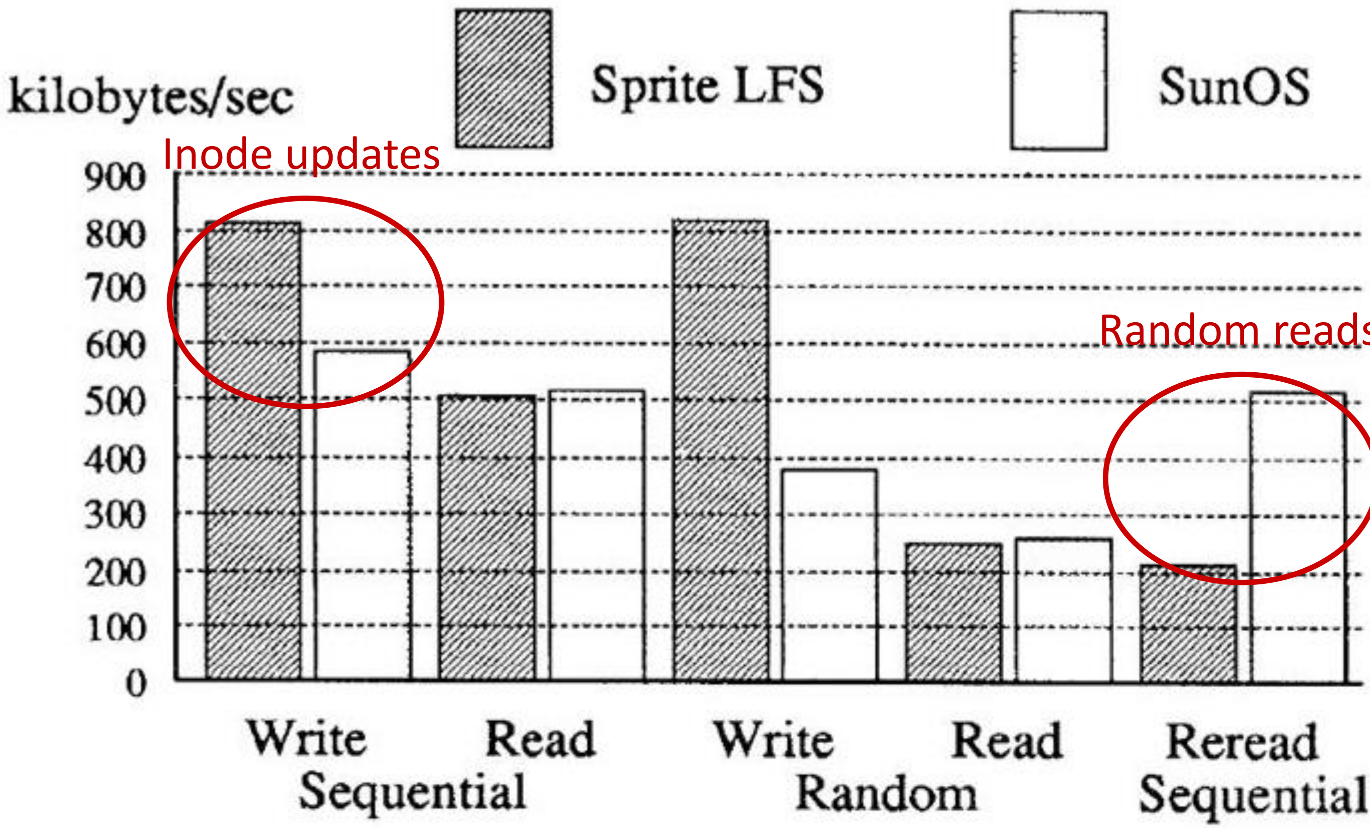
$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated*age of data}}{\text{cost}} = \frac{(1-u)\text{*age}}{1+u}.$$

이게 90년대 초에 나온 논문인데,
LFS에서 사용한 GC victim selection 기준을
기본적인 FTL 메커니즘에서도 그대로 사용했다는 면에서
의미가 크다.

Cost-Benefit Policy (Cont.)



LFS Performance



update가 여러번 일어난 뒤 GC되기 전에는 read하기 위해서 넓은 범위를 다 읽어야 한다.

End of Chapter 6