香港中文大學
The Chinese University of Hong Kong

CSCI5550 Advanced File and Storage Systems

Lecture 04: File System Designs

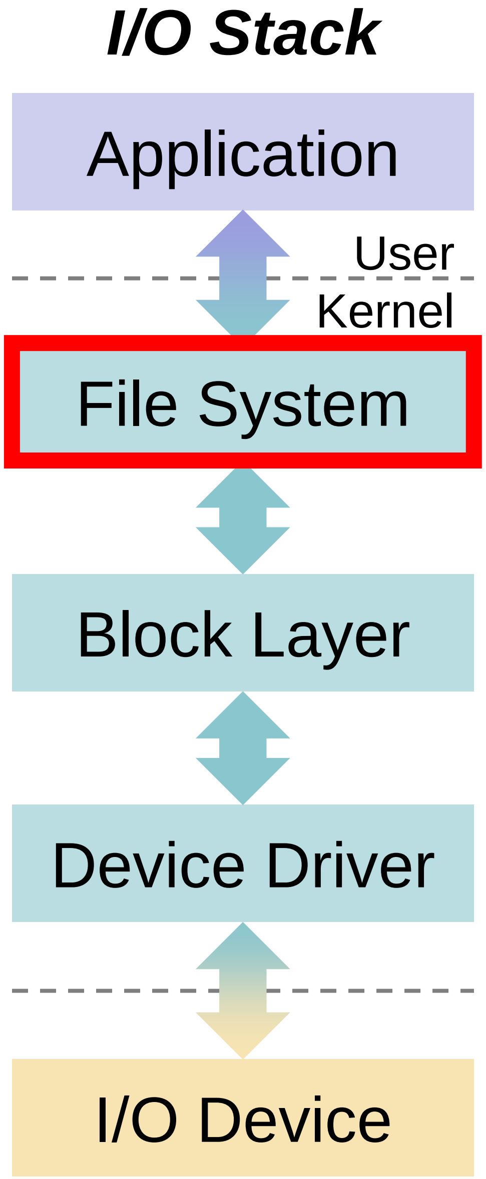
Ming-Chang YANG

mcyang@cse.cuhk.edu.hk

Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery

- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation

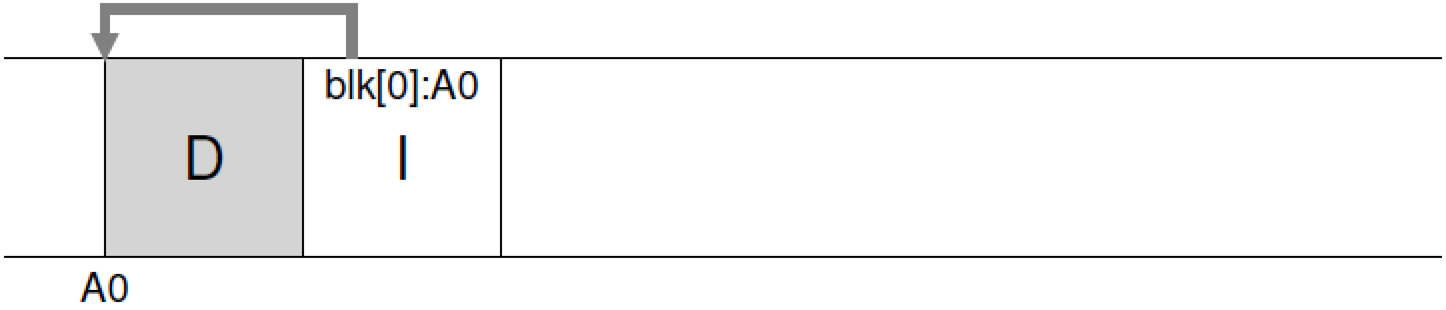


Motivation: Why to develop LFS?

- We need a file system that improves **writes**:
  - ① System memories are growing.
    - More data can be cached in memory to service reads effeciently.
    - Disk traffic increasingly consists of writes.
  - ② There is a large gap between random I/O and sequential I/O performance in disk.
    - Disk transfer bandwidth has increased a lot over the years.
      - By packing more bits into the surface of a disk.
    - Seek and rotational delay costs have decreased slowly.
  - ③ Existing file systems perform poorly.
    - FFS incurs many short seeks and rotational delays.
  - ④ File systems are not RAID-aware.
    - Both RAID-4 and RAID-5 have the small-write problem.
    - Existing file systems do not avoid this RAID writing behavior.

Log-structured File System (LFS)

- **Log-structured File System (LFS)**
  - Writes everything (*including data blocks and inodes, etc.*) to the disk sequentially.
  - Ex: Writing a data block **`D`** and updated inode **`I`** to the disk.

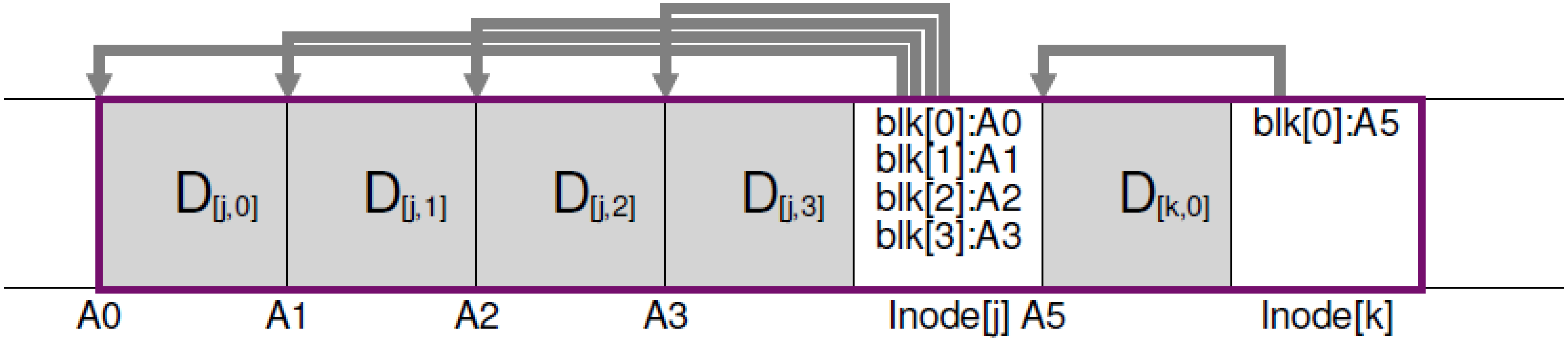


- *Note: in most systems, data blocks are 4 KB in size, whereas an inode is much smaller (e.g., 128 B).*

- The idea looks simple, but the devil is in the details!
  - Several design issues must be handled carefully.

Writing Sequentially, and Effectively!

- Writing to disk sequentially is not (alone) enough to guarantee efficient writes.
  - In-between the first and second writes, the disk has rotated.
- LFS first **buffers** all writes in an in-memory segment; when the segment is large enough, LFS **commits** the segment to disk as a single large write.
  - This technique is well known as write buffering.
  - It is possible to buffer writes to different files in a segment.



Issue #1: How Much to Buffer? (1/2)

- Assume that
  - $T_{position}$ is time to position (i.e., $T_{rotation} + T_{seek}$) the disk head
  - R_{peak} is the disk transfer rate
  - D is the amount of data to buffer

- Then we can derive
  - The time to write the data: $T_{write} = T_{position} + \frac{D}{R_{peak}}$
  - The effective rate of write: $R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}$

Issue #1: How Much to Buffer? (2/2)

- How to get the effective rate close to the peak rate?
- The effective rate is some fraction F of the peak rate:

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak}$$

- And we can solve for $\boldsymbol{D}$:

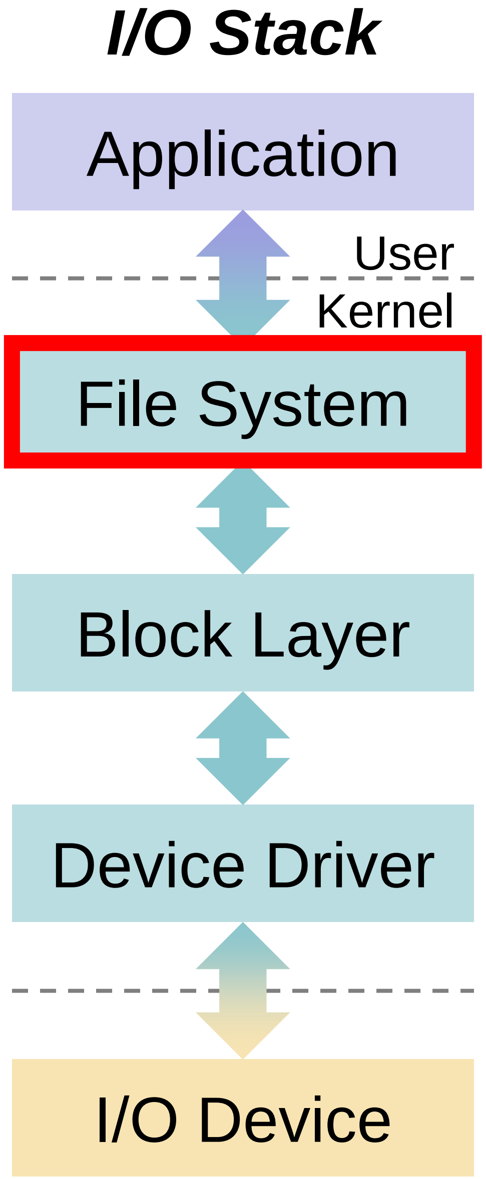
$$\boldsymbol{D} = F \times R_{peak} \times \left(T_{position} + \frac{\boldsymbol{D}}{R_{peak}} \right) = \frac{F}{1 - F} \times R_{peak} \times T_{position}$$

- For example, if $T_{position} = 10\,ms$, $R_{peak} = 100\,MB/s$, and we want $F = 0.9$ (i.e., 90% of the peak):

$$D = \frac{0.9}{1 - 0.9} \times 100 \left(\frac{MB}{s} \right) \times 10\,(ms) = 9\,(MB)$$

Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery
- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation



I/O Stack

Application

User

Kernel

File System

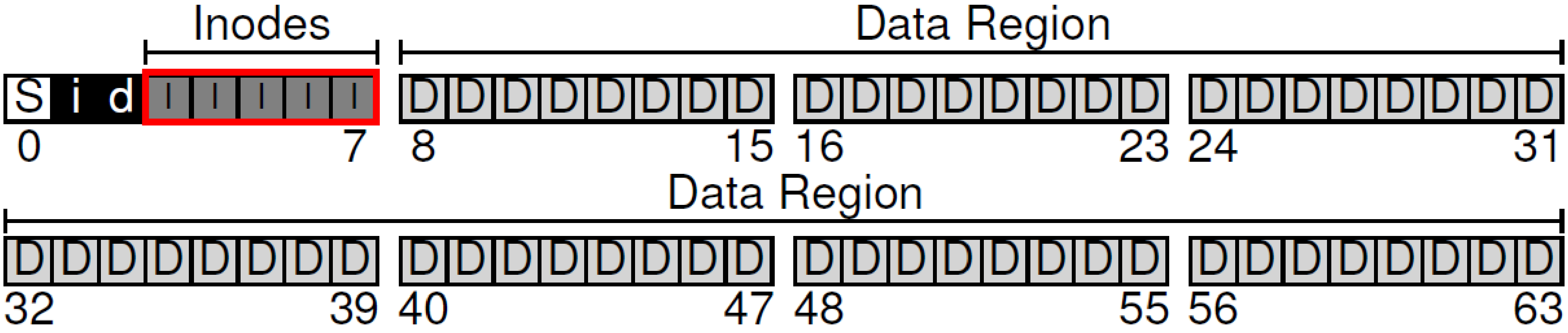
Block Layer

Device Driver

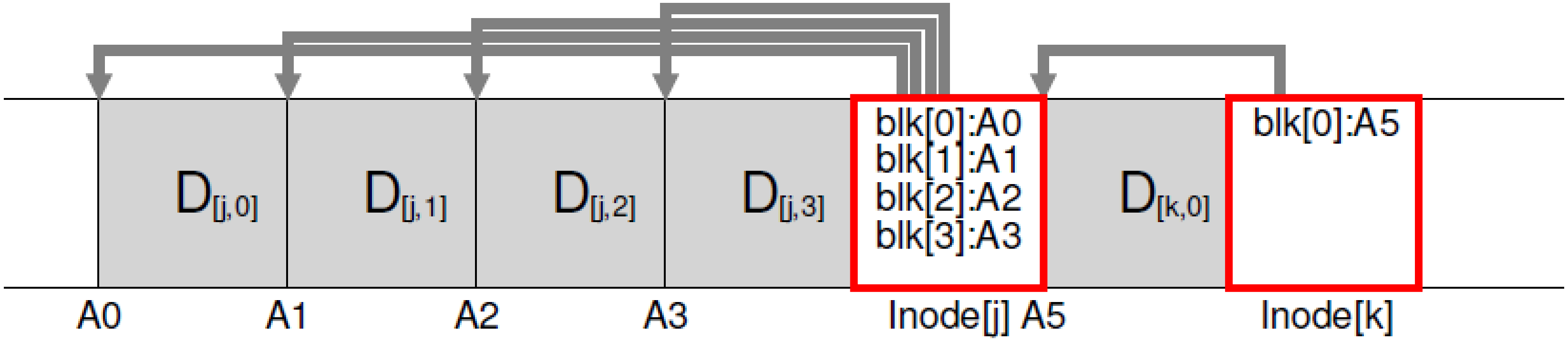
I/O Device

Issue #2: How to Find Inodes? (1/3)

- UNIX file system keeps inodes at fixed locations.

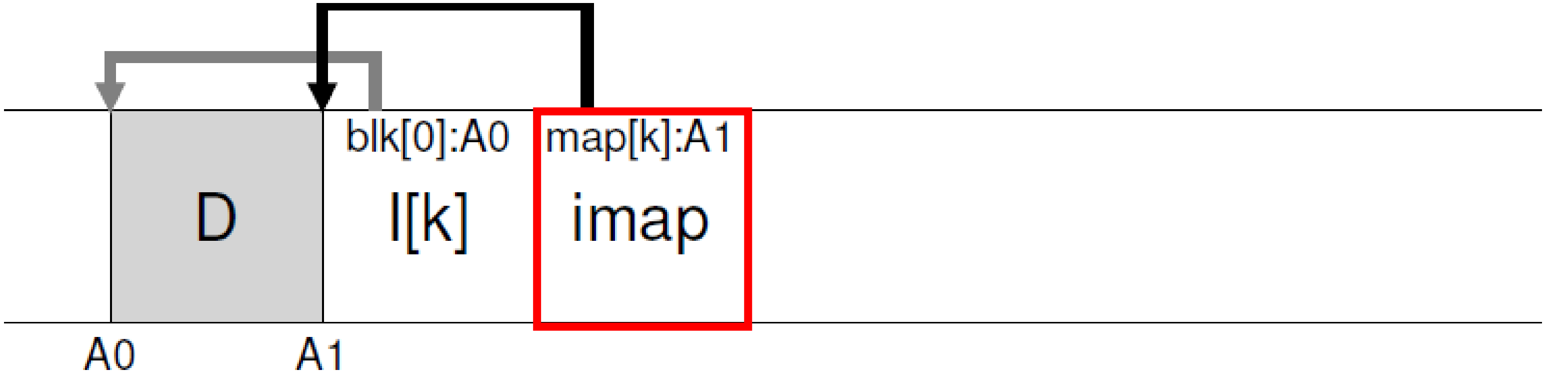


- In LFS, inodes are scattered throughout disk.



Issue #2: How to Find Inodes? (2/3)

- **Solution through Indirection**: The **Inode Map** (**`imap`**)
  - Maps from an `inode-number` to the `disk-address` of the *most recent version* of the inode (i.e., one more mapping!).
  - Implemented as an array of 4 bytes (disk pointer) per entry.
  - Updated whenever an inode is written to disk.
- LFS places the `imap` right next to where it is writing.
  - E.g., when appending a data block, the new data block (D), its node (I[k]), and `imap` are written to disk together:

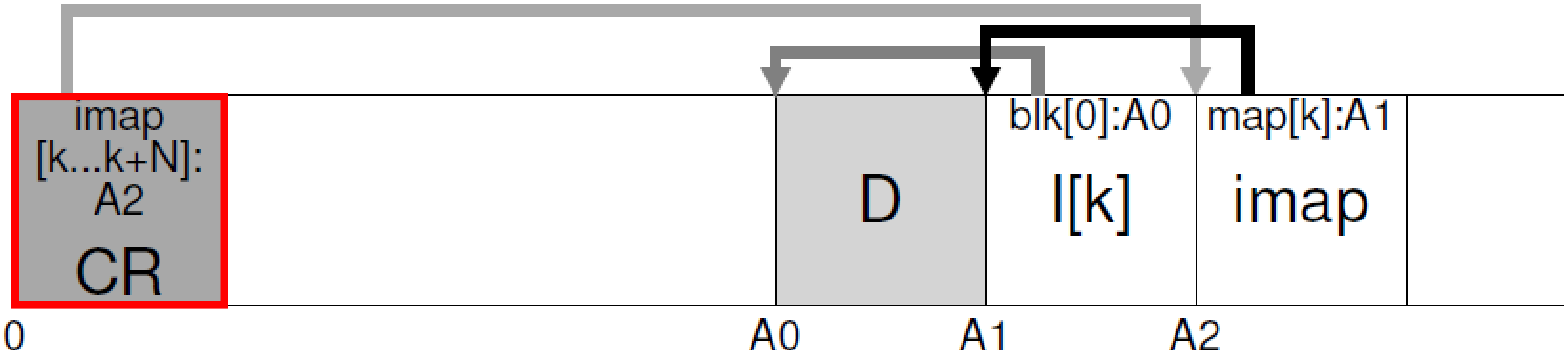


- Now we can find inodes: But how to find the `imap`?

Issue #2: How to Find Inodes? (3/3)

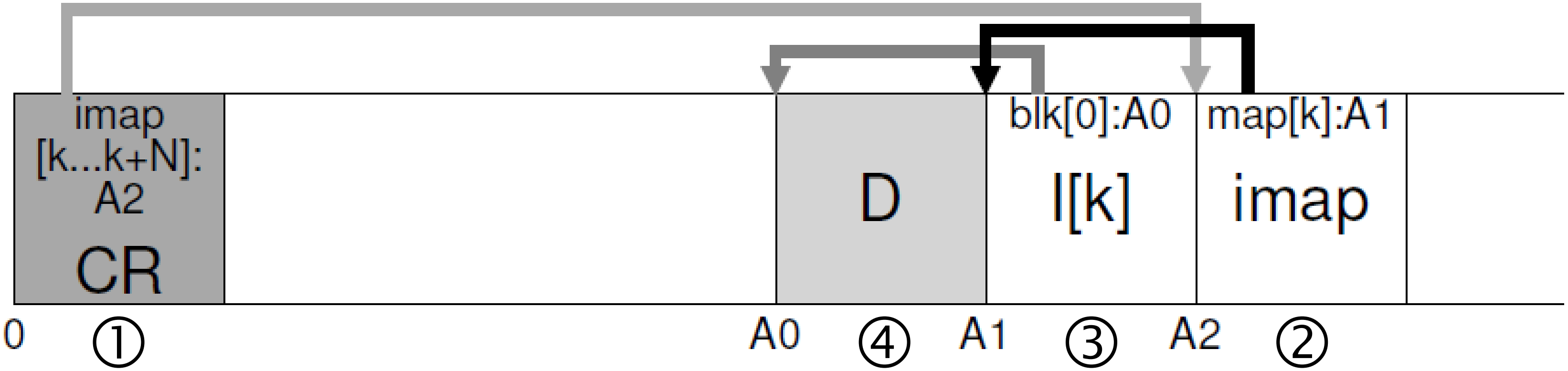
- The pieces of imap are also spread across the disk.
- Every file system must have some fixed and known location on disk to being a file lookup.

- **Complete Solution**: The Checkpoint Region (`CR`) records disk pointers to all latest pieces of `imap`.
  - Flushed to disk periodically (e.g., every 30 seconds).



Example: Reading a File

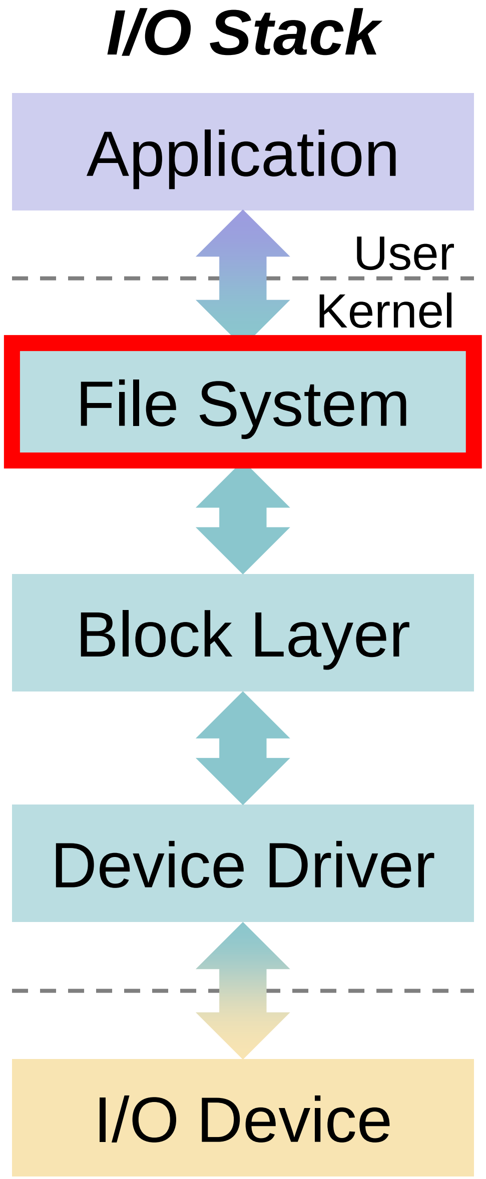
- To read a file from disk, LFS needs to
  ① Read the checkpoint region to find the latest `imap`;
  ② Read the latest `imap` to have the disk location of the inode;
  ③ Read the most recent version of the inode (`I[k]`);
  ④ Read data blocks using direct/indirect pointer as usual.



- To perform the same number of I/Os as UNIX FS, LFS must cache the checkpoint region (`CR`) and the entire `imap` in the system memory.

Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery
- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation



I/O Stack

Application

User

Kernel

File System

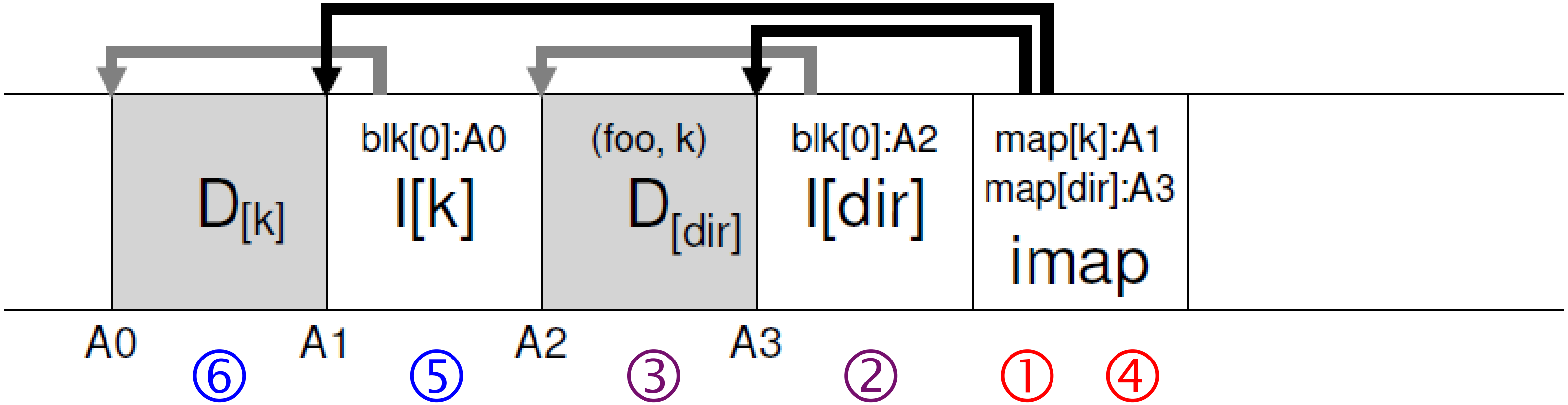
Block Layer

Device Driver

I/O Device

Issue #3: What about Directories? (1/2)

- The directory structure of LFS is identical to UNIX FS.
  - The directory is a collection of (`name`, `inode-num`) entries.
- When creating a file, LFS writes the data and the new inode, the directory and its inode, and the latest imap.
  - LFS will do so sequentially on the disk as follows:



- When reading a file in the directory, LFS looks up ① imap *(often cached in memory)*, ② directory inode, ③ directory data, ④ imap, ⑤ file inode, and ⑥ file data.

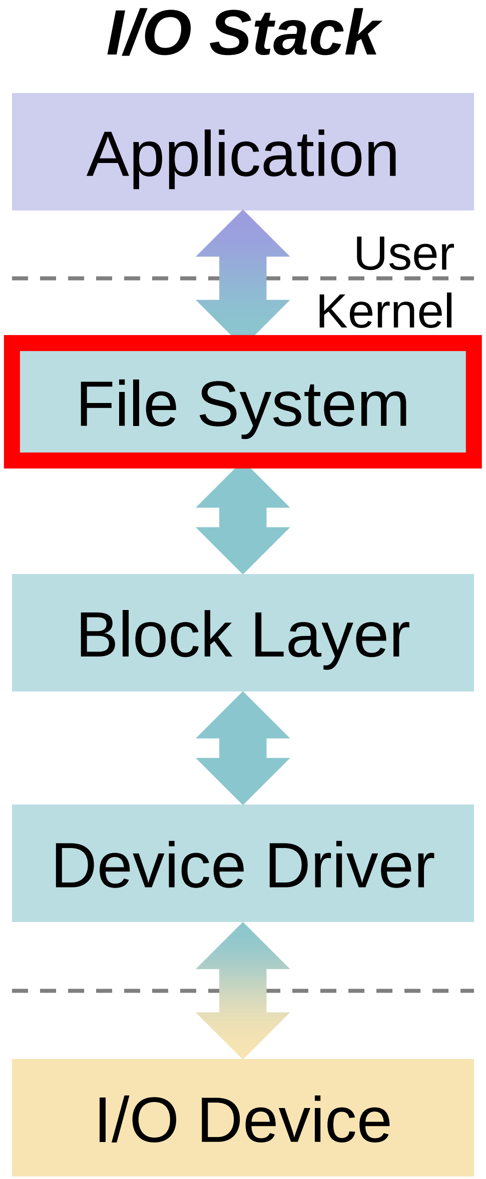
Issue #3: What about Directories? (2/2)

- **Recursive Update Problem**: A serious problem arisen in any file system that never updates in place.
  - Whenever an inode is updated, its location on disk changes.
    - To keep track of inodes, a directory may record a collection of (`name`, `inode-location`) entries.
  - This would have also entailed **recursive updates** to the directory that points to this file, the parent of that directory, …, all the way up the file system tree.

- LFS cleverly avoids this problem with `imap`.
  - The directory is a collection of (`name`, `inode-num`) entries.
  - The imap keeps `inode-num` to `inode-location` mappings.
    - Even though the location of an inode may change, the change is never reflected in the **directory itself**.

Outline

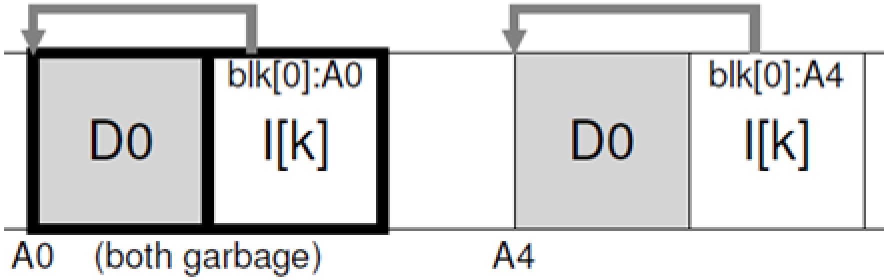
- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery
- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation



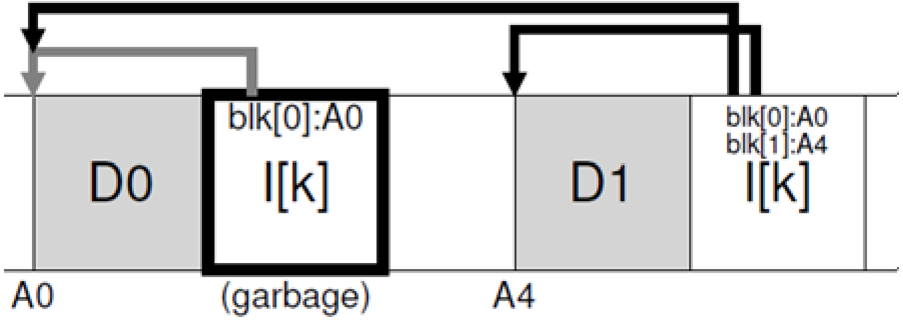
Issue #4: Garbage Collection (1/4)

- LFS *never overwrites* but writes to free locations.
  - Multiple versions of data may co-exist across the disk.
    - The old version(s) of data are usually called **garbage**.

Case 1: Updating a data block `D0`



Case 2: Appending a data block `D1`



- One could keep older versions and allow accessing.
  - Such a file system is known as a **versioning file system**.
- LFS keeps only the latest *live* versions of data, and periodically cleans old *dead* versions of data.
  - The process of cleaning is called **garbage collection (GC)**.

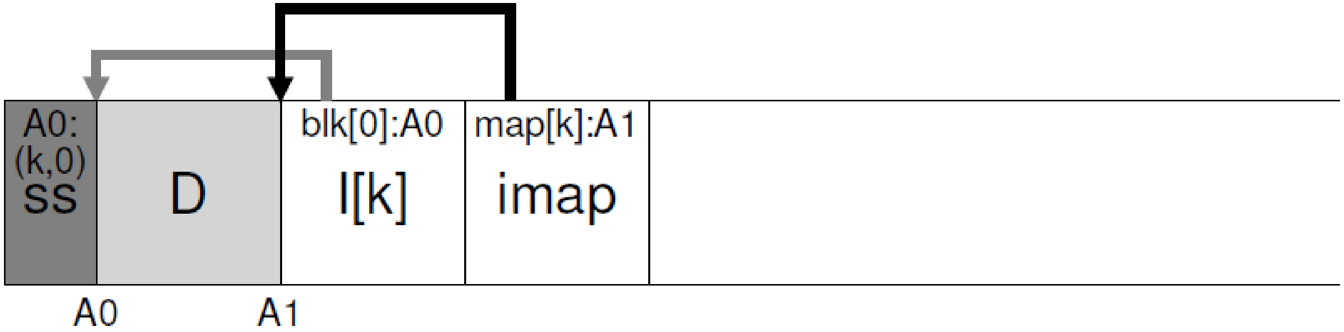
Issue #4: Garbage Collection (2/4)

- LFS adopts a **segment-based cleaning** as follows:
  - ① Reads in M partially-used segments;
  - ② Determines which blocks are live within these segments;
  - ③ Compacts only *live* contents into N new segments ($N < M$);
  - ④ Writes out N segments to disk in new locations;
  - ⑤ Frees old M segments for subsequent writing.

- Two more problems:
  - How to determine if a block is *live* (or *dead*)?
  - How often, and which segments to clean?

Issue #4: Garbage Collection (3/4)

- LFS adds extra information, at the head of each segment, called the **segment summary block (SS)**.
  - It records, for each data block `D` in the segment, its inode number `N` and its offset `T` (e.g., (k, `0`)).



  - The *liveness* for a block `D` of address `A` can be determined:

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Optimization:

- Keeping a version number in both imap and SS, extra reads of inodes can be further avoided.
- The version number should be incremented whenever the file is truncated or deleted.

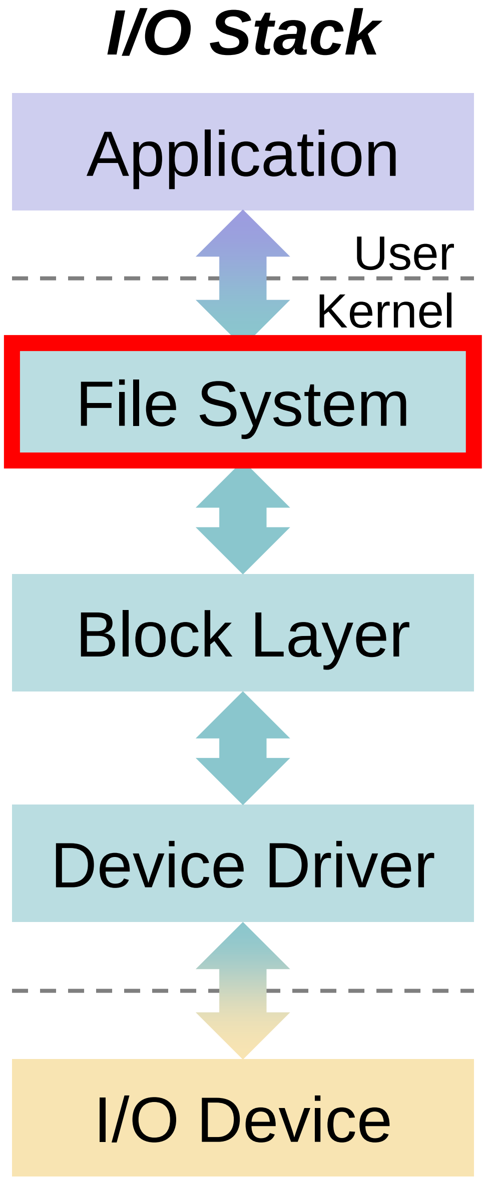
Issue #4: Garbage Collection (4/4)

- When to clean?
  - Either periodically, during idle time, or when the disk is full.

- Which segments are worth cleaning?
  - LFS tries to segregate hot and cold segments.
    - A hot segment consists of frequently-over-written blocks.
    - A cold segment may only have a few over-written (dead) blocks.
  - LFS cleans cold segments sooner and hot segments later.
    - Since as time goes by, more and more blocks in the hot segment may get over-written (in new segments).
    - This policy is heuristic but not perfect.

Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery
- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation



I/O Stack

Application

User

Kernel

File System

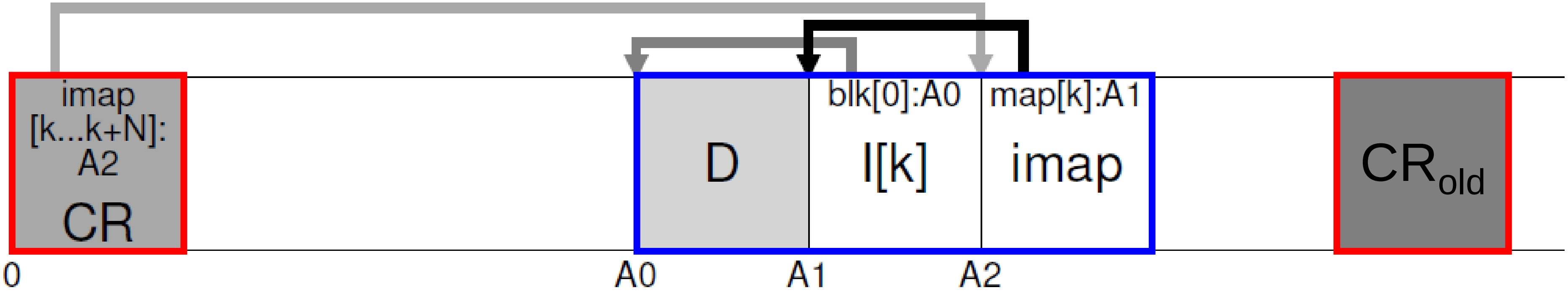
Block Layer

Device Driver

I/O Device

Issue #5: Crash Recovery

- Crashes when writing to the checkpoint region:
  - – **Solution:** Keeps **two CRs** (e.g., one at the head and one at the end) and writes to them alternately.
    - It first writes a header (with a timestamp), then the body of CR, and then an end marker (with a timestamp).
    - Inconsistent pair of timestamps implies an error.



- Crashes when writing to a segment:
  - – **Roll Forwarding**: Starts with the **last checkpoint region** and rebuilds all “non-checkpointed” but “committed” segments (please read the paper for details).

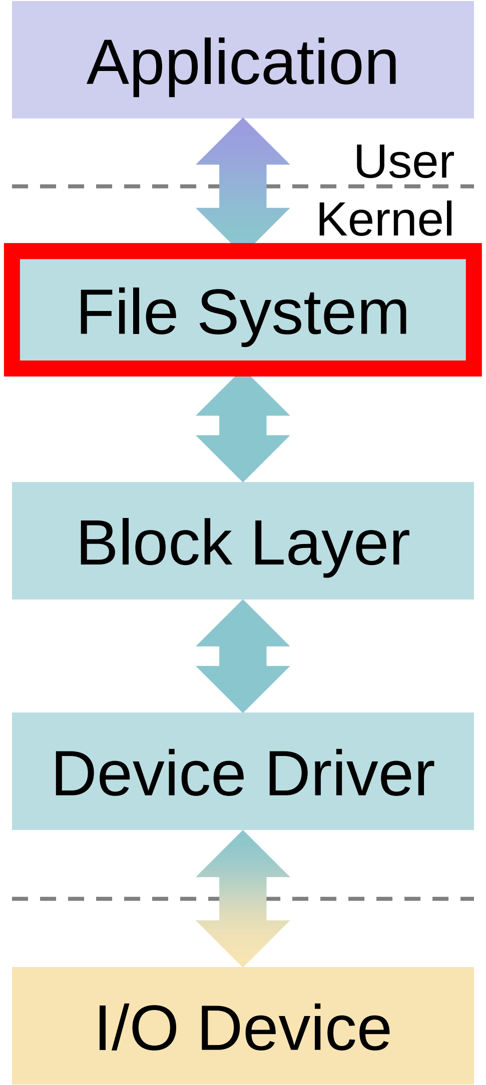
Recall: Metadata Journaling

- The sequence of metadata journaling:
  - ① **Data Write**: Write data to final location
  - ② **Journal Metadata Write**: Write the begin block (`TxB`) and metadata (`I[v2]`, `B[v2]`) to log
  - ③ **Journal Commit**: Write the transaction commit block (`TxE`)
  - ④ **Checkpoint Metadata**: Write the contents of metadata update to their final locations within the file system
  - ⑤ **Free**: Mark the transaction free in the journal superblock
- Notes:
  - Forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required.
  - The only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

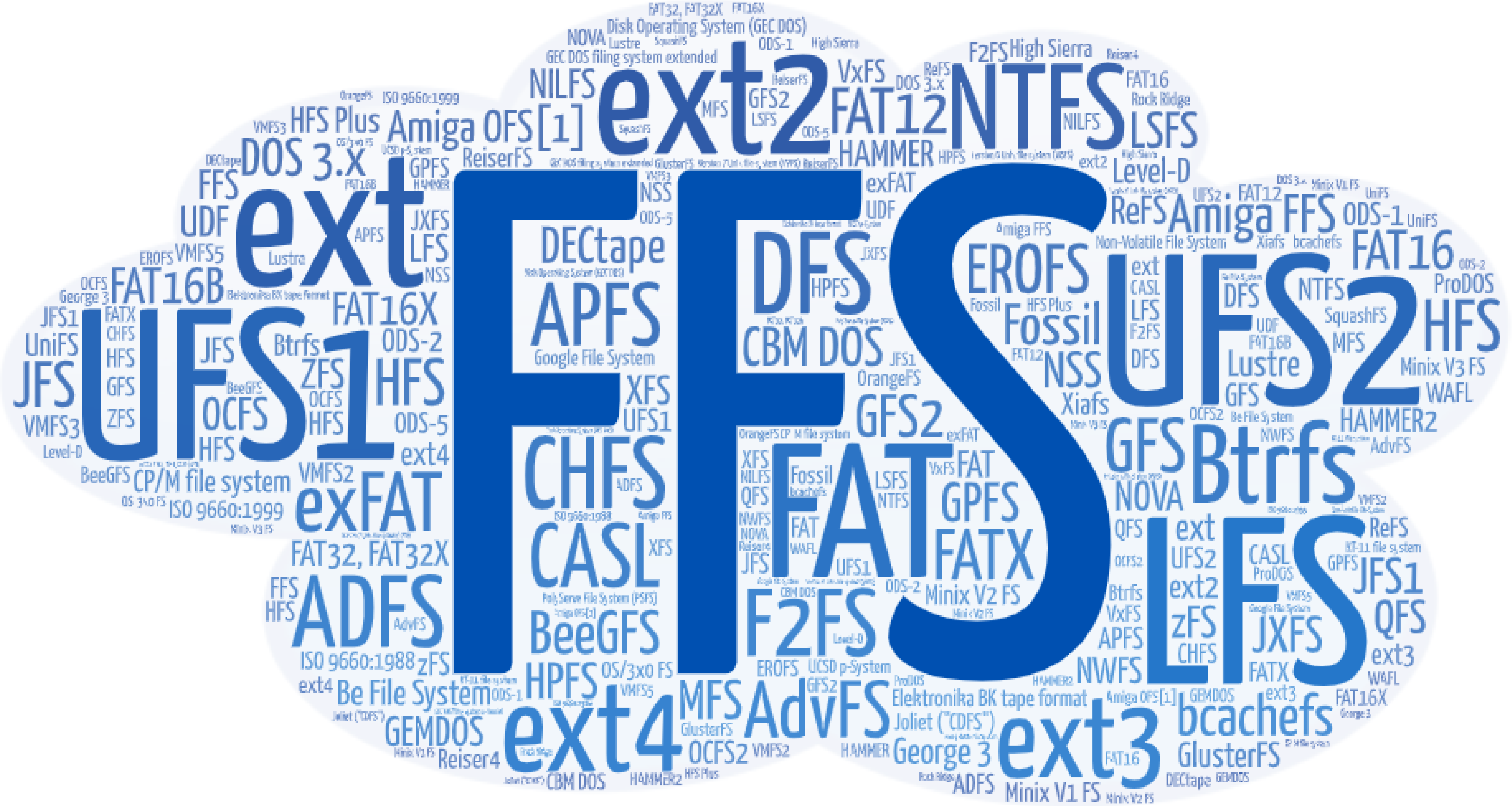
Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery
- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation

I/O Stack

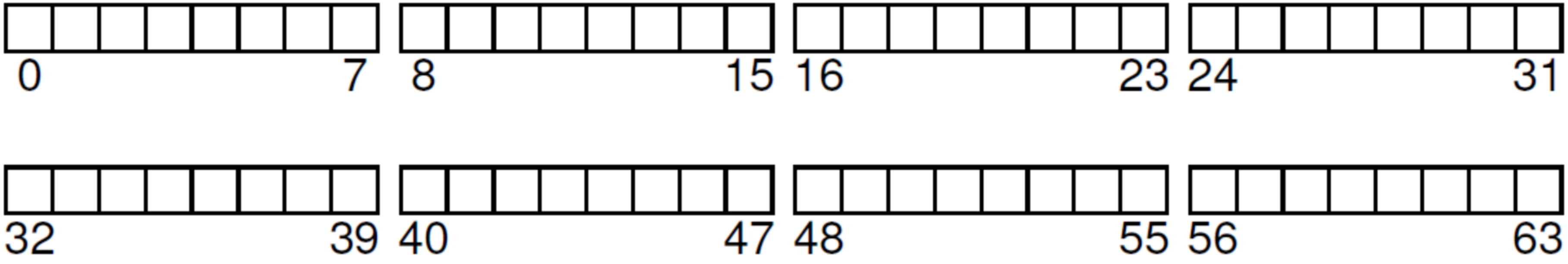


Big Family of File Systems



File Implementation: Block Allocation

- Block Allocation: How to allocate disk space to files

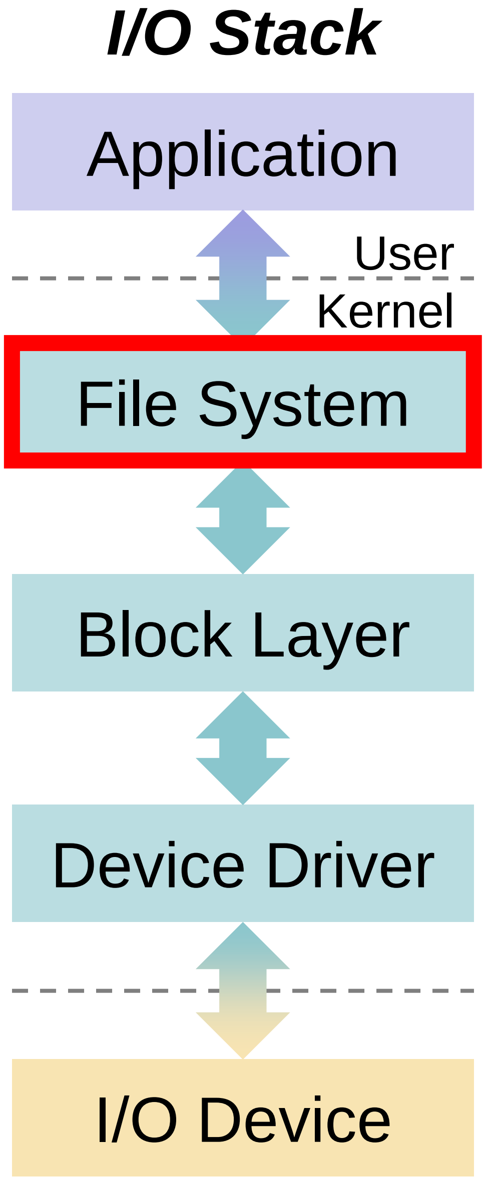


- It is a typical way to classify file system designs:
  - ① **Indexed Allocation**: an index block keeps block pointers
    - Examples: `UNIX FS`, `FFS`, `ext2`, `LFS`
  - ② **Linked Allocation:** each file is of linked blocks
    - Examples: `FAT`
  - ③ **Contiguous Allocation:** each file is of contiguous blocks
    - Examples: `ext4`

Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery

- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation



I/O Stack

Application

User

Kernel

File System

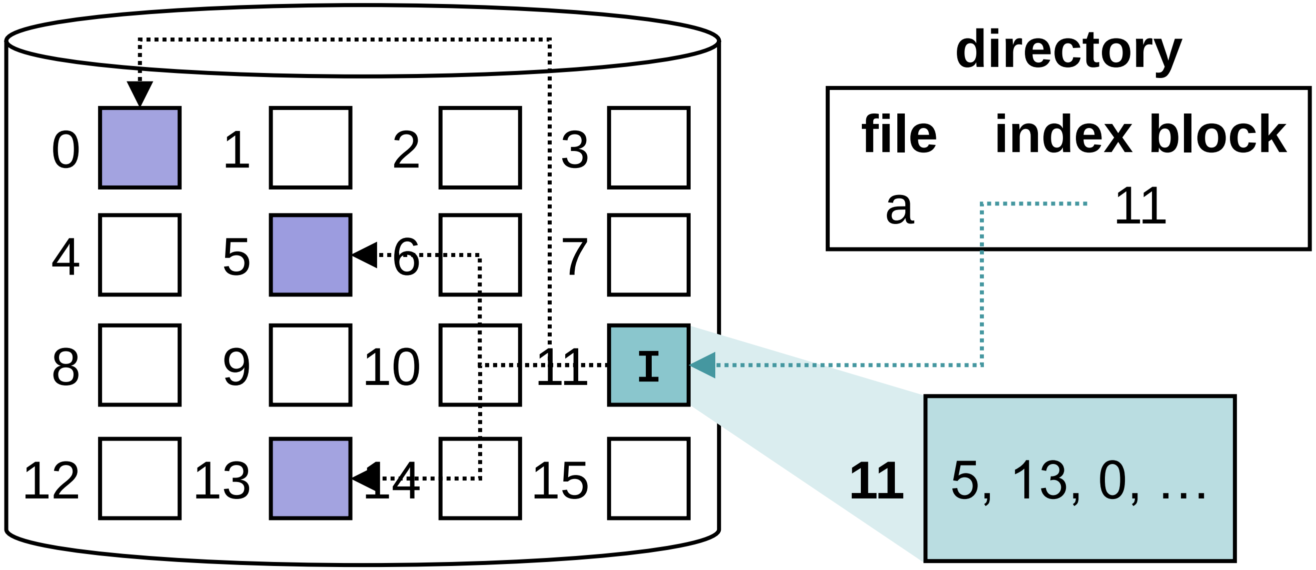
Block Layer

Device Driver

I/O Device

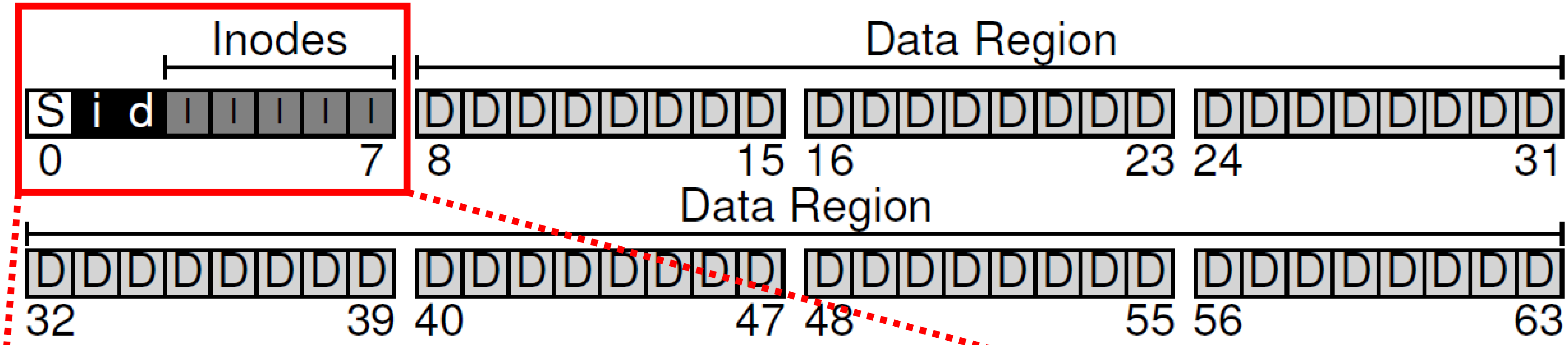
① Indexed Allocation

- Each file has its own index block, which keeps track of all block pointers/locations of a file.
  - The i^{th} entry in the index block points to the i^{th} block.
- Potential Issues:
  - The index block could be far away from data blocks.
  - Data blocks are scattered across the disk.

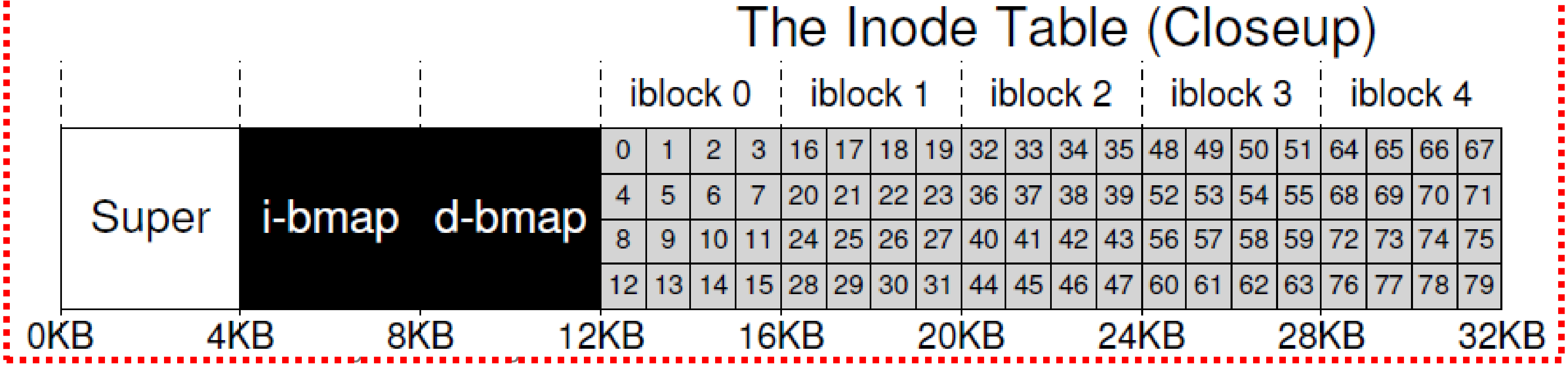


Recall: UNIX FS and its Variants

- **`UNIX file system`** (and its variants **`FFS`**, **`ext`**, **`ext2`**, etc.) are typical representatives of indexed allocation.

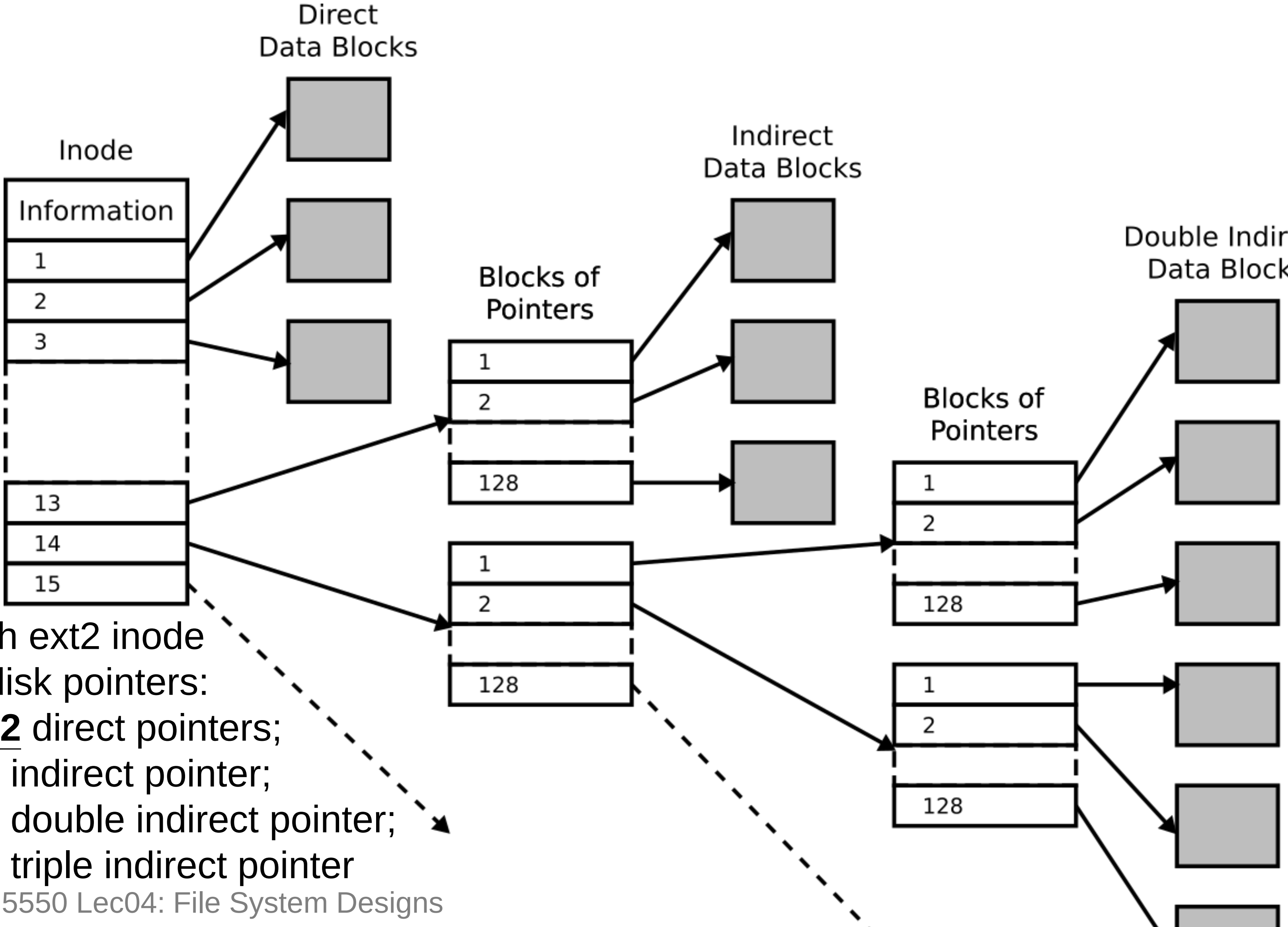


  - **Metadata Region**: tracks data and file system information.
  - **Data Region**: stores user data and occupies most space.



Recall: Multi-Level Index

- Multi-level index supports files of **big** sizes.

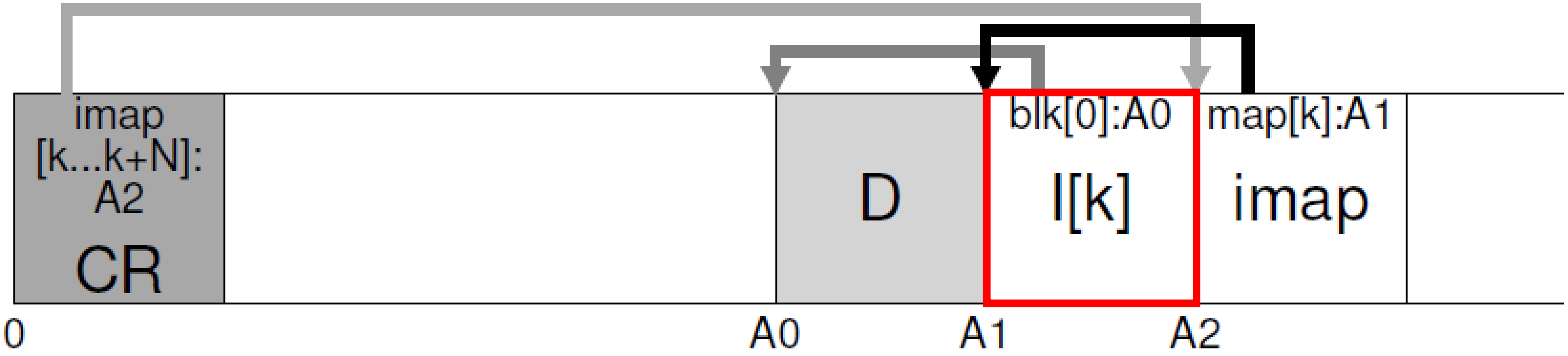


Each ext2 inode
<u>15</u> disk pointers:

- <u>12</u> direct pointers;
- <u>1</u> indirect pointer;
- <u>1</u> double indirect pointer;
- <u>1</u> triple indirect pointer

Recall: Log-structured File System

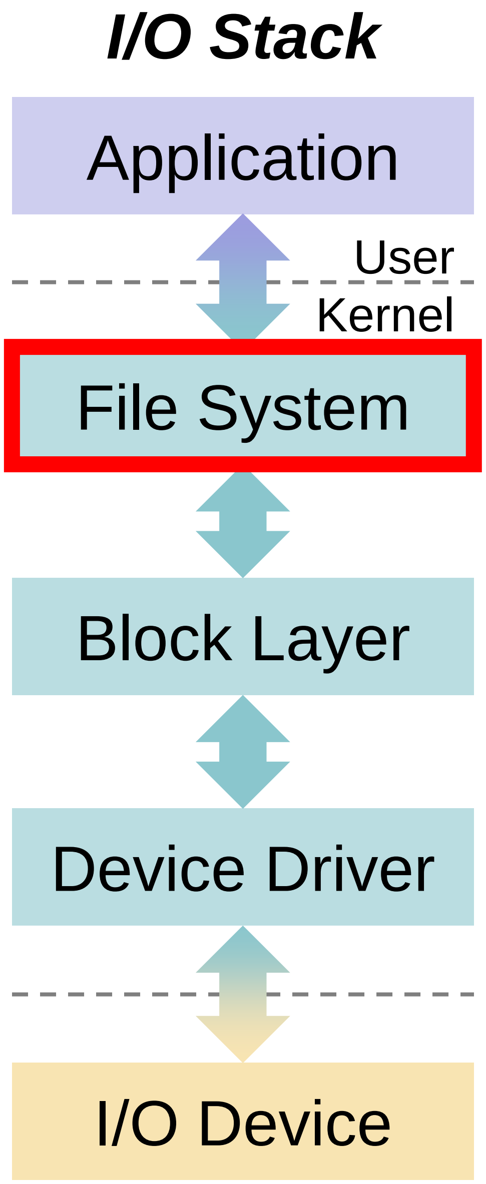
- LFS can be also considered as indexed allocation, in which the indirection is further introduced:
  - The Checkpoint Region (**`CR`**):
    - Records disk pointers to all latest pieces of **`imap`**.
    - Flushed to disk periodically (e.g., every 30 seconds).
  - The Inode Map (**`imap`**)
    - Maps from an `inode-number` to the `disk-address` of the *most recent version* of the **inode** (i.e., one more mapping!).
    - Updated whenever an inode is written to disk.
    - Placed right next to where data block (**`D`**) and **inode** (**`I[k]`**) reside.



Outline

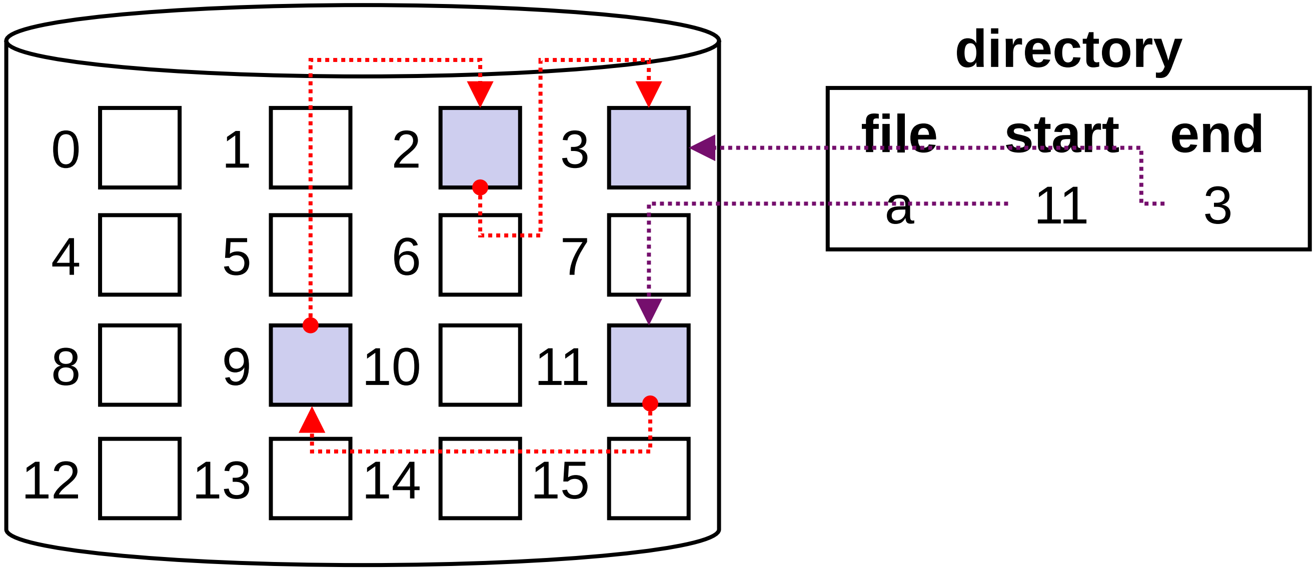
- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery

- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation


I/O Stack

② Linked Allocation (1/2)

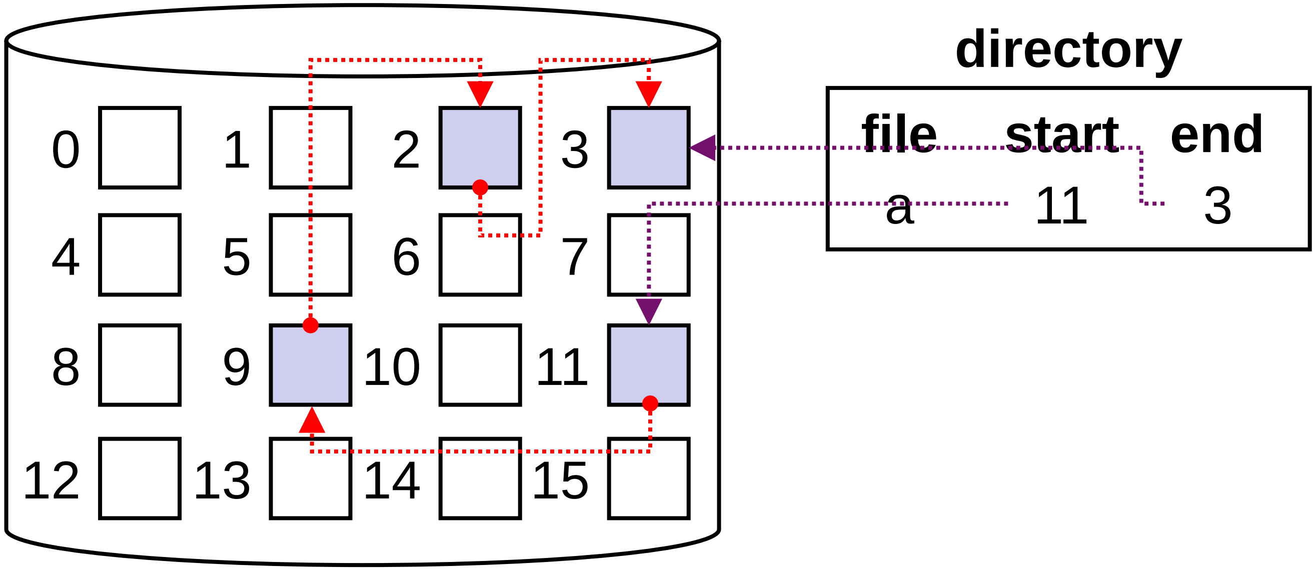
- Each file is a linked list of disk blocks, which may be scattered anywhere on the disk.
  - The directory maintains the **first** and **last** blocks of the file; every block contains a **pointer** to the next block.
    - Each 512-byte block is of 508-byte user data and 4-byte pointer.
  - A file can easily continue to grow if there are free blocks.



file	**start**	**end**
a	11	3

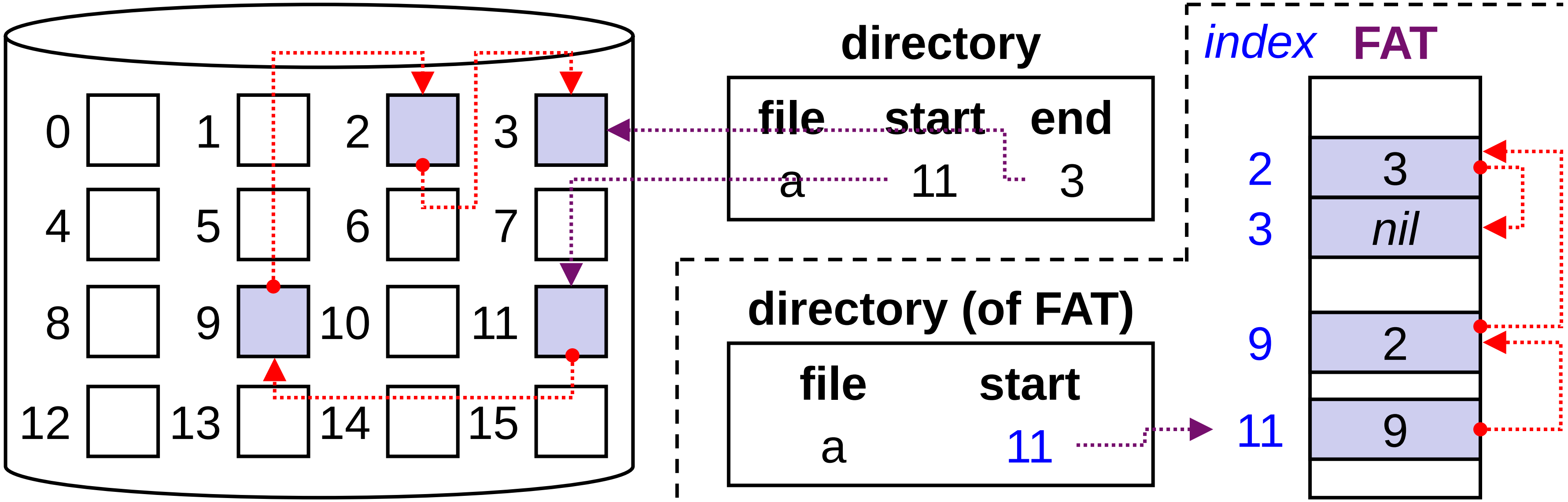
② Linked Allocation (2/2)

- Potential Issues:
  - It can be used effectively **only for** sequential-access files.
    - It is **inefficient** to arbitrarily access the i^{th} block of a file.
  - It costs 0.78% (4 B / 512 B) of the disk space for pointers.
    - One solution is to collect multiple blocks into a cluster.
  - Any lost or damaged pointer makes a big mess.
  - Data blocks may be scattered across the disk.



File Allocation Table (FAT)

- File Allocation Table (FAT):
  - A variation on linked allocation (used by MS-DOS and OS/2).
  - A table indexed by block number (i.e., one entry per block).
    - The directory entry contains the block number of the **first block**.
    - **Each FAT entry** indicates the block number of the **next block**.
    - There is **no need** to maintain the 4B block pointer in each data block.
  - Problem: The in-disk FAT could be far away from blocks.



directory

file	start	end
a	11	3

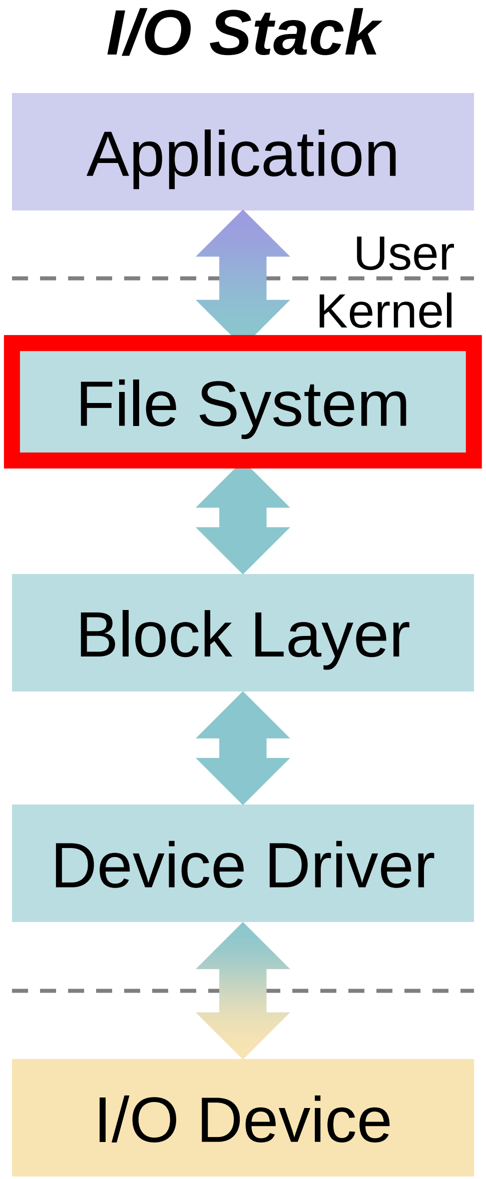
directory (of FAT)

file	start
a	11

index	**FAT**
2	3
3	*nil*
9	2
11	9

Outline

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery
- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation



I/O Stack

Application

User

Kernel

File System

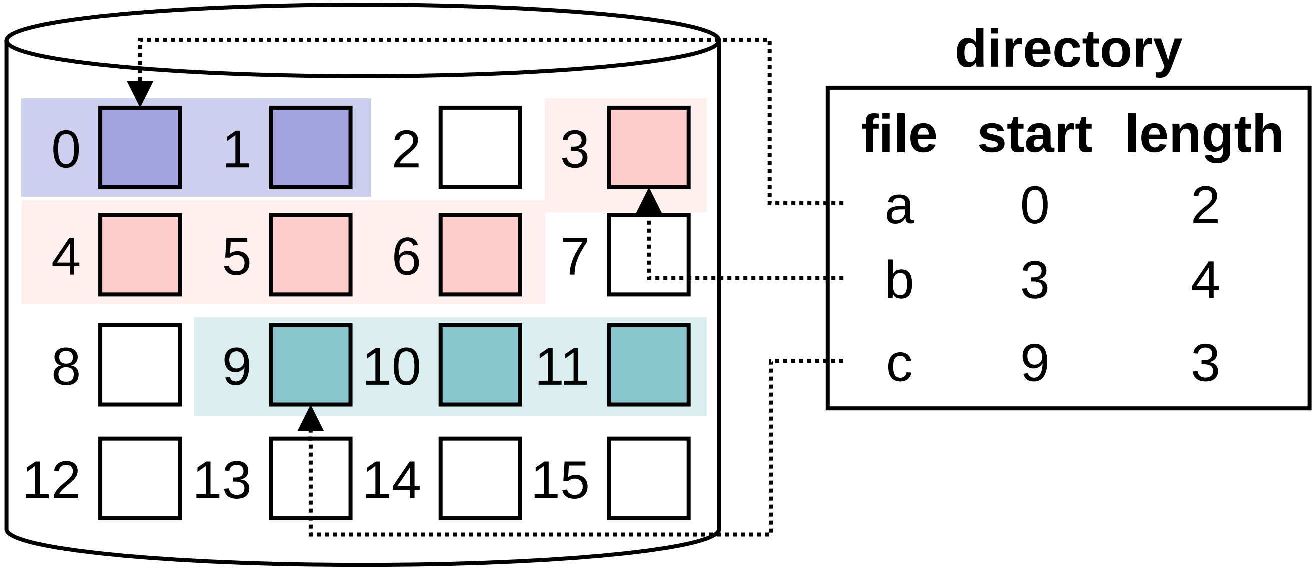
Block Layer

Device Driver

I/O Device

③ Contiguous Allocation

- Each file occupy a set of contiguous blocks.
  - Block addresses define a **linear ordering** on the disk.
  - Every allocation is defined by the **start address** and **length**.
- It is efficient for **both** sequential and direct access.
- The difficulties are to 1) *determine how much space is need*, and 2) *find contiguous space* for a file.

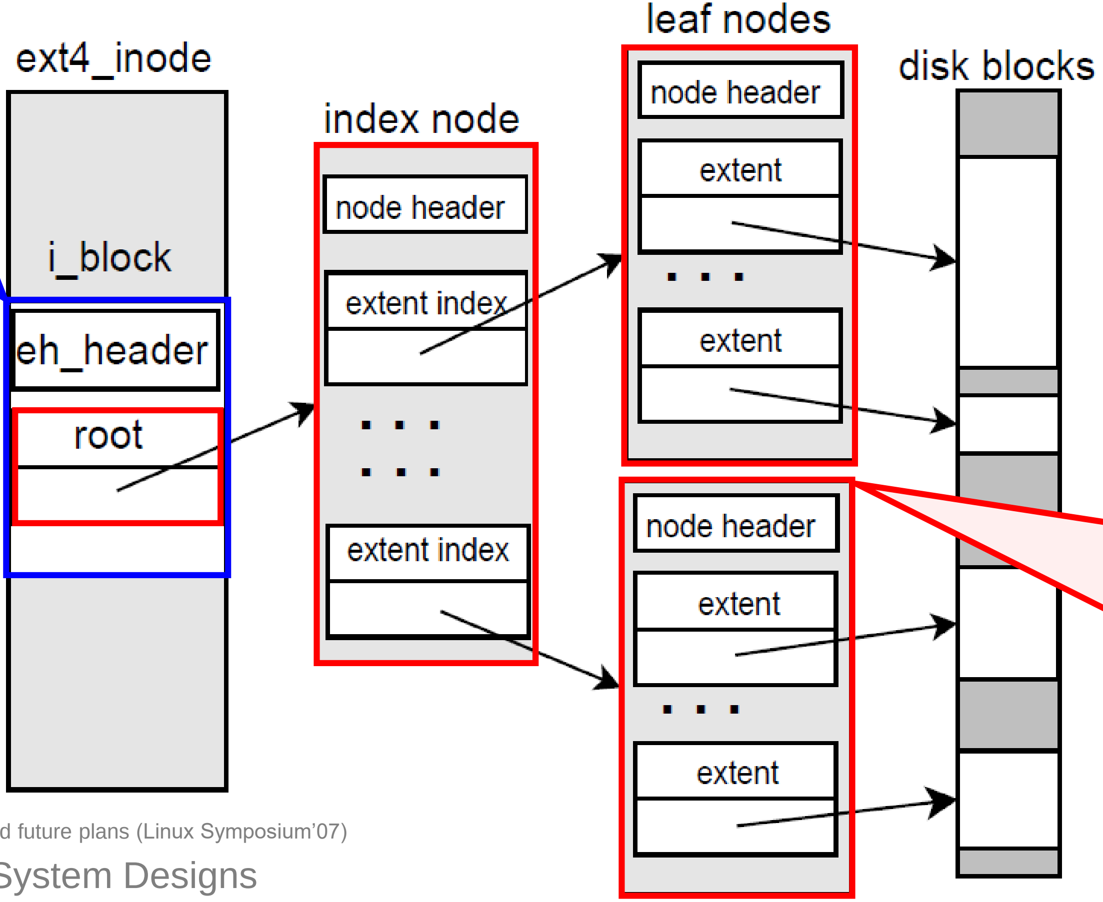


directory

file	**start**	**length**
a	0	2
b	3	4
c	9	3

Extent

- To avoid over-or-under allocation, some file systems (e.g., **`ext4`**) adopt a modified contiguous allocation.
  - A chunk of contiguous and variable-sized space, **extent**, is allocated whenever the allocated space is insufficient.



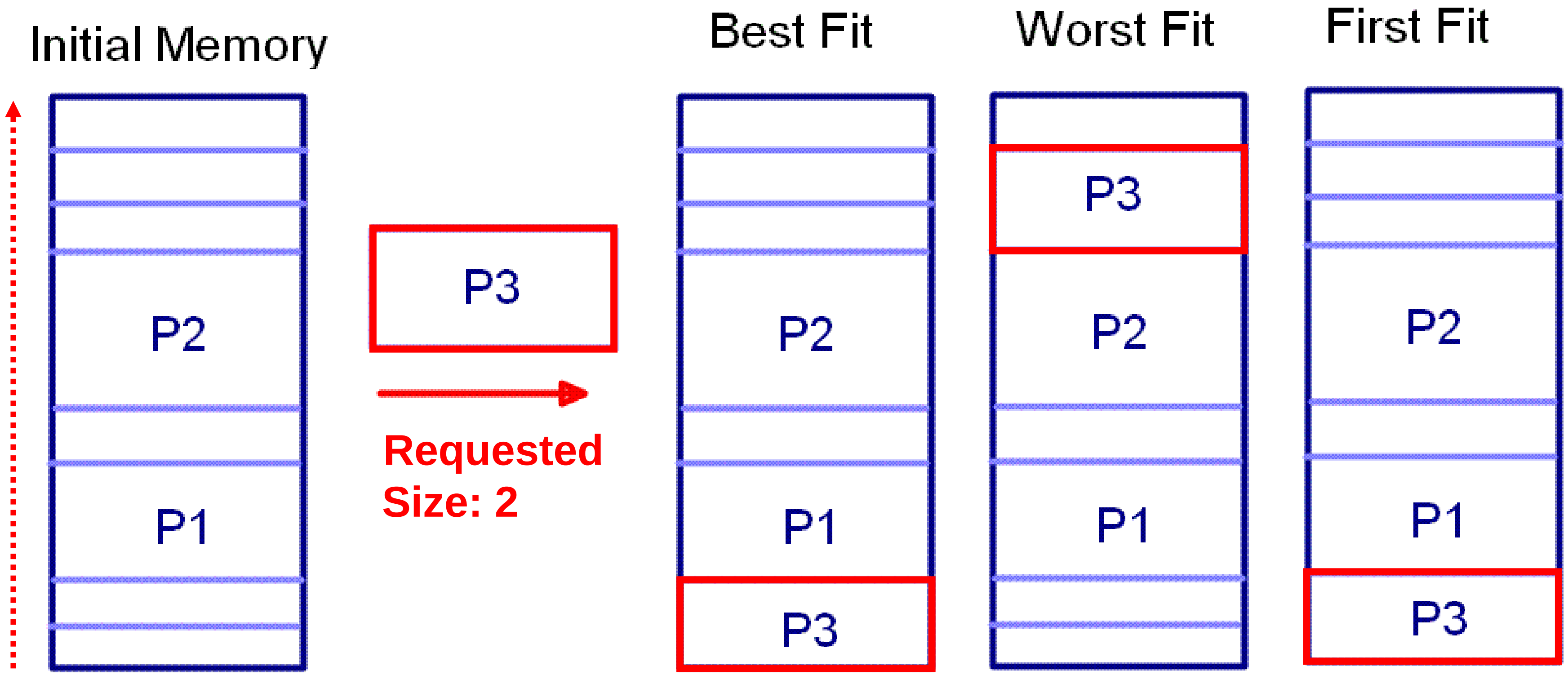
Four extents can be kept in **`ext4`** inode (not shown).

An **extent tree** is used to store the extents map for a big file.

The new ext4 filesystem: current status and future plans (Linux Symposium'07)

Dynamic Allocation Problem

- How to satisfy a request of size n from a list of *holes*?
- Common Solutions: **best-fit**, **worst-fit**, and **first-fit**.
  - It is also a common problem of memory management.



smallest but big enough

largest

first

http://www.r9paul.org/blog/2008/managing-your-memory/

Summary

- Log-structured File System (LFS)
  - Key Idea: Writing Sequentially
  - Indirect Mapping and Checkpoint Region
  - Directories
  - Garbage Collection
  - Crash Recovery

- File Implementation: Block Allocation
  - Indexed Allocation
  - Linked Allocation
  - Contiguous Allocation