K22 – Operating Systems: Design Principles and Internals

Fall 2025 @dit

Vaggelis Atlidakis

Lecture 20

References: Similar OS courses @Columbia, @Stanford, @UC San Diego, @Brown, @di (previous years); and textbooks: Operating Systems: Three Easy Pieces, Operating Systems: Principles and Practice, Operating System Concepts, Linux Kernel Development, Understanding the Linux Kernel

Overview

- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - ~~Runtime [Q: How does a program look like in memory?]~~
  - ~~Processes [Q: What is a process?]~~
  - ~~IPC [Q: How do processes communicate?]~~
  - ~~Threads [Q: What is a thread?]~~
  - ~~Synchronization [Q: What goes wrong w/o synchronization?]~~
  - ~~Time Management [Q: What is scheduling?]~~
  - ~~Memory Management [Q: What is virtual memory?]~~
  - ~~Files [Q: What is a file descriptor?]~~
  - **Storage Management [Q: How do we allocate disk space to files?]**

* Basic (H/W & S/W)
* Abstractions
* Primitives
* Mechanisms

Overview

- **Storage**
  - **Q1**: Allocating storage space to files?
  - **Q2**: File system layout?
  - **Q3**: Files and directory names?
  - **Q4**: Path name resolution?
  - **Q5**: Linux file system data structures?
  - **Q6**: Achieving crash tolerance?
  - **Q7**: Achieving fault tolerance?

Allocating storage space to files

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

> Storage devices w/o moving parts

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

> Storage devices w/o moving parts

> Faster and more reliable than HDD

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

> Storage devices w/o moving parts

> Faster and more reliable than HDD

> Still quite slow, compared to main memory

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

> Storage devices w/o moving parts

> Faster and more reliable than HDD

> Still quite slow, compared to main memory

- Be proactive => Prefetch data (leverage spatial locality)

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

> Storage devices w/o moving parts

> Faster and more reliable than HDD

> Still quite slow, compared to main memory

- Be proactive => Prefetch data (leverage spatial locality)
- Hide latency => Do slow storage operations asynchronously

Calculating major vs minor page fault latency

```c
int main(int argc, char **argv) {
    // Assuming file already exists at path
    if ( (fd = open("/tmp/foo.txt", O_RDONLY, 0664)) < 0)
        return -1;
   char *buf = mmap(NULL, page_size, PROT_READ, MAP_PRIVATE, fd, 0);
   start_time = clock_gettime_ns();
   char a = buf[0];
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu ns (1st read) \n", end_time -  start_time);
   start_time = clock_gettime_ns();
   char a = buf[0];
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu ns (2nd read) \n", end_time -  start_time);
}
```

```
# 1 -> Drop in-kernel page caches
➜  git:(master) ✗  echo 1 | sudo tee /proc/sys/vm/drop_caches
1
➜  git:(master) ✗  ./demo
Time elapsed: 448,834 ns (1-st read) <-- Major page fault / Going to strorage
Time elapsed: 41 ns (2-nd read)
➜  git:(master) ✗  ./demo
Time elapsed: 11,041 ns (1-st read) <-- Minor page fault / Staying in main mem.
Time elapsed: 42 ns (2-nd read)
```

Allocating storage space to files

How do we allocate persistent storage space to files?

> Systems people are very predictable…

Allocating storage space to files

How do we allocate persistent storage space to files?

> Systems people are very predictable…

> Split storage to fixed-size chucks, called *blocks*

Allocating storage space to files

How do we allocate persistent storage space to files?

- Systems people are very predictable…
- Split storage to fixed-size chucks, called *blocks*
- Use n blocks to serve a file, where n = filesize / blocksize
- Allocation strategies

Allocating storage space to files

How do we allocate persistent storage space to files?

> Systems people are very predictable…

> Split storage to fixed-size chucks, called *blocks*

> Use n blocks to serve a file, where n = filesize / blocksize

> Allocation strategies

- Contiguous allocation
- Linked allocation
- Indexed allocation
- Multi-level indexed allocation

Allocating storage space to files

How do we allocate persistent storage space to files?

> Systems people are very predictable…

> Split storage to fixed-size chucks, called *blocks*

> Use n blocks to serve a file, where n = filesize / blocksize

> Allocation strategies

- Contiguous allocation
- Linked allocation
- Indexed allocation
- Multi-level indexed allocation

Contiguous allocation

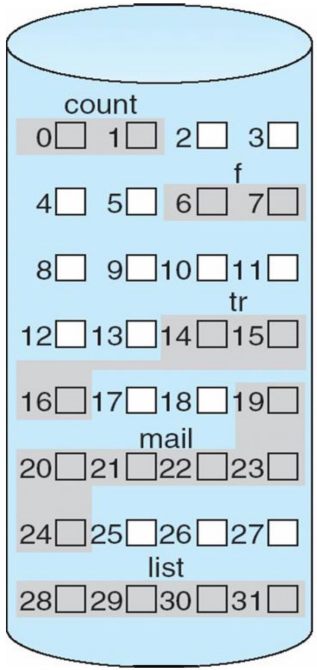
Allocate a contiguous set of blocks, of sufficient size to each file

- File metadata: Starting block and no of blocks
- System-wide bitmap of free blocks

Contiguous allocation

Allocate a contiguous set of blocks, of sufficient size to each file

- File metadata: Starting block and no of blocks
- System-wide bitmap of free blocks



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Image from: OS Concepts, by A. Silberschatz et al.

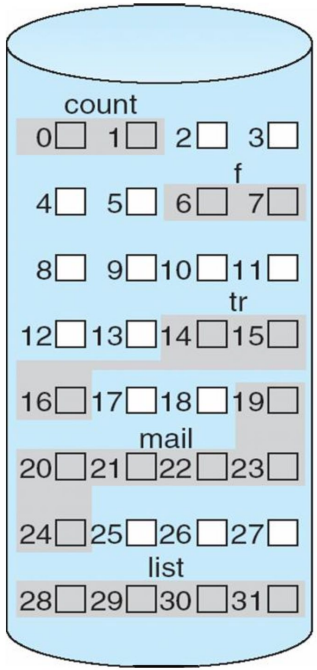
Contiguous allocation

Allocate a contiguous set of blocks, of sufficient size to each file

- File metadata: Starting block and no of blocks
- System-wide bitmap of free blocks

> Advantages

- Low storage overhead => Two vars per file
- Fast sequential access => Consecutive blocks
- Quick calculation of blocks for random accesses



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Image from: OS Concepts, by A. Silberschatz et al.

Contiguous allocation

Allocate a contiguous set of blocks, of sufficient size to each file

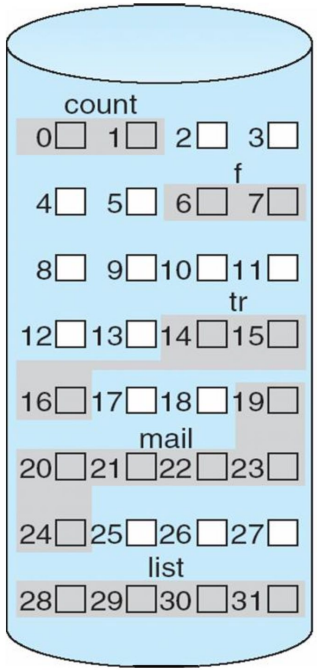
- File metadata: Starting block and no of blocks
- System-wide bitmap of free blocks

› Advantages

- Low storage overhead => Two vars per file
- Fast sequential access => Consecutive blocks
- Quick calculation of blocks for random accesses

› Disadvantages

- Difficult to grow a file
- External fragmentation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

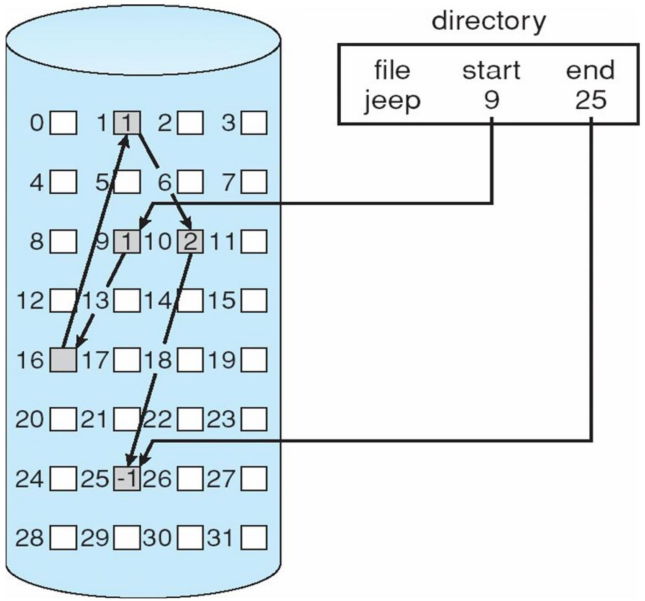


Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

> Advantages
  - No fragmentation
  - Files can easily grow dynamically

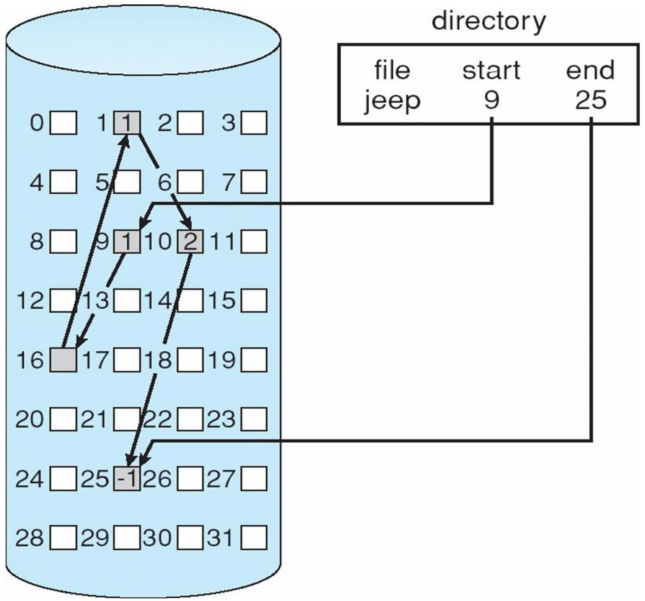


Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

› Advantages
  - No fragmentation
  - Files can easily grow dynamically

› Disadvantages
  - Slow on random accesses

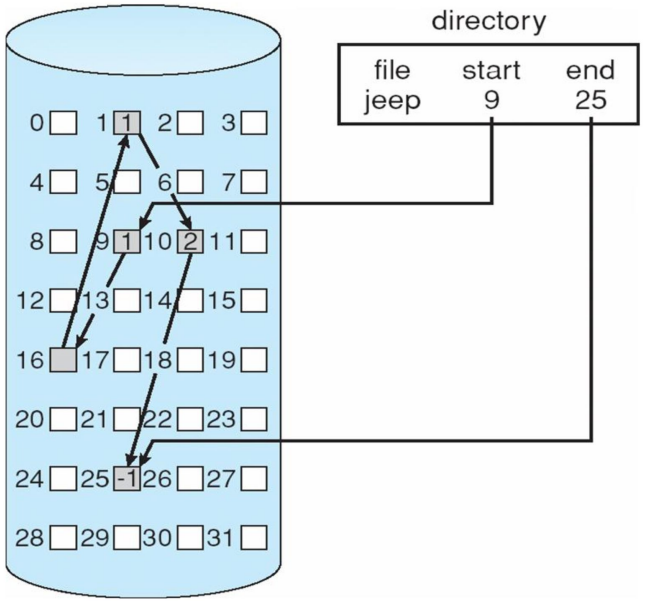


Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

› Advantages
  - No fragmentation
  - Files can easily grow dynamically

› Disadvantages
  - Slow on random accesses
  - Storage overhead => One ptr per block

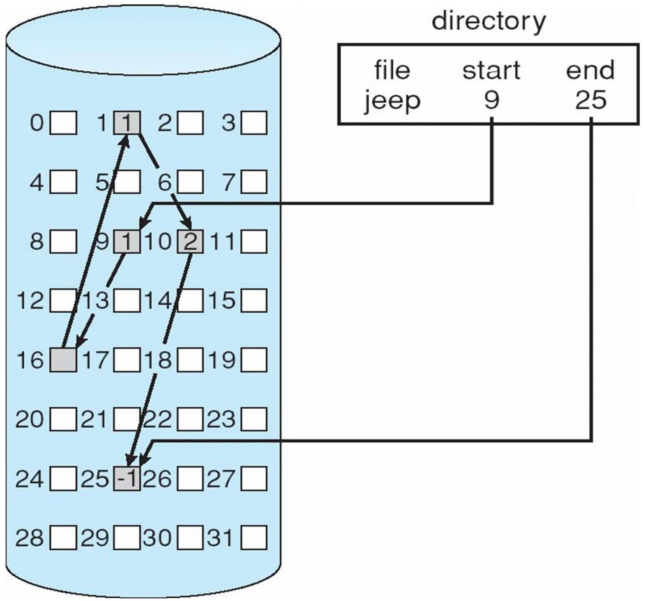


Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

› Advantages
  - No fragmentation
  - Files can easily grow dynamically

› Disadvantages
  - Slow on random accesses
  - Storage overhead => One ptr per block
  - "Unoptimizable:" Index cannot be cached

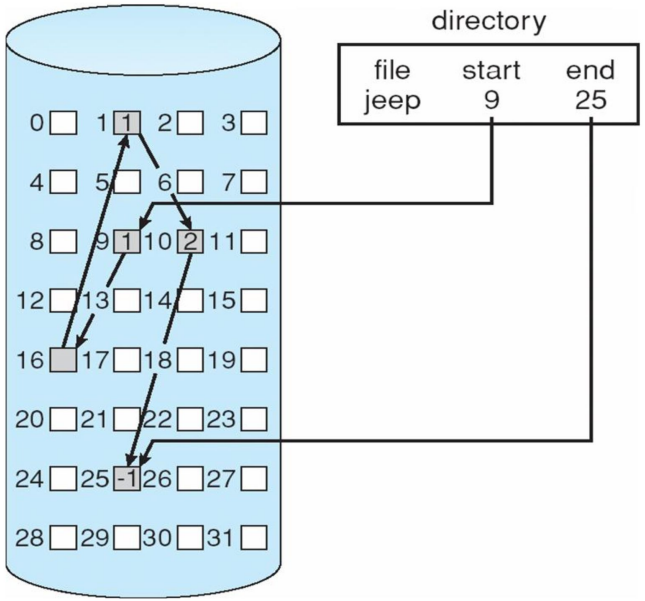


Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

> Advantages

- No fragmentation
- Files can easily grow dynamically

> Disadvantages

- Slow on random accesses
- Storage overhead => One ptr per block
- "Unoptimizable:" Index cannot be cached
- **Unreliable:** Loose one block => loose everything

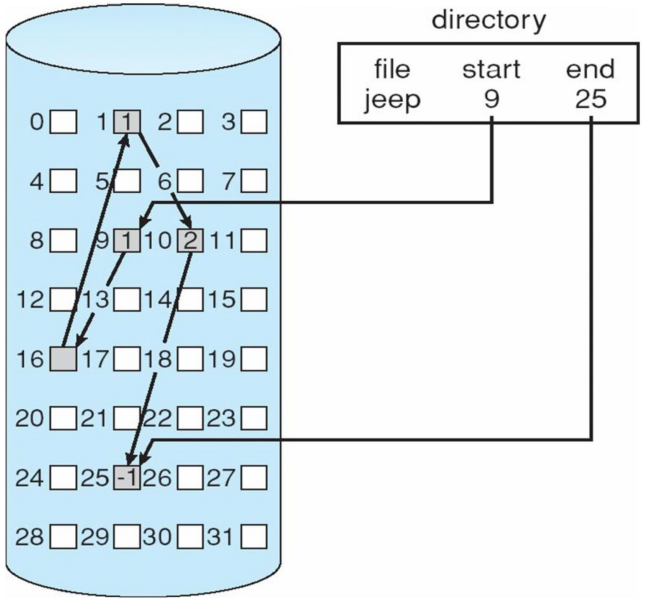


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

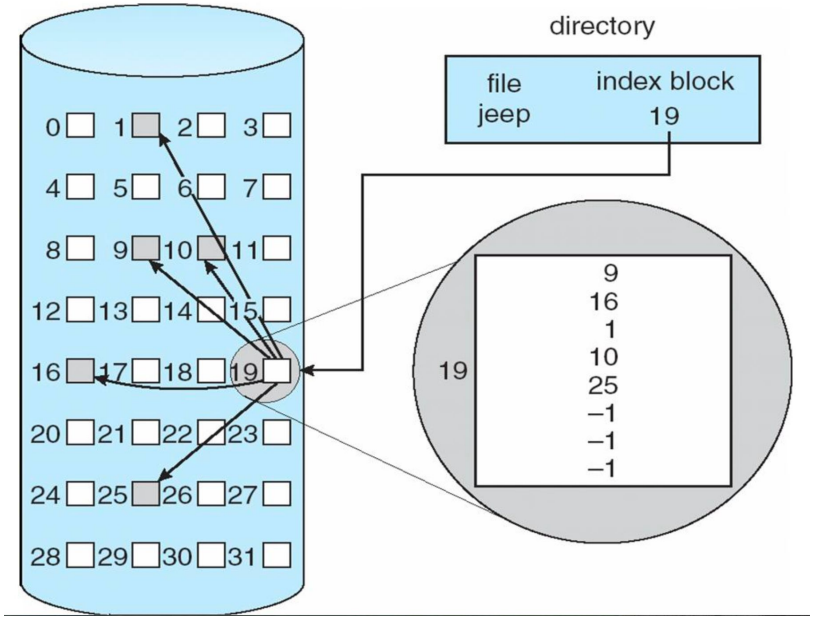


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

› Advantages

- No fragmentation
- Files can easily grow dynamically
- Fast random access

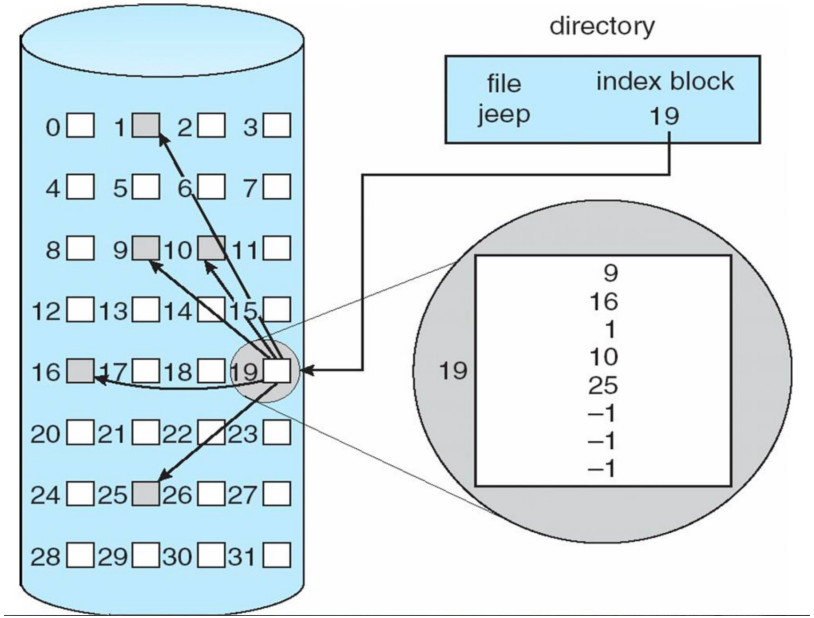


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

› Advantages

- No fragmentation
- Files can easily grow dynamically
- Fast random access (**How?**)

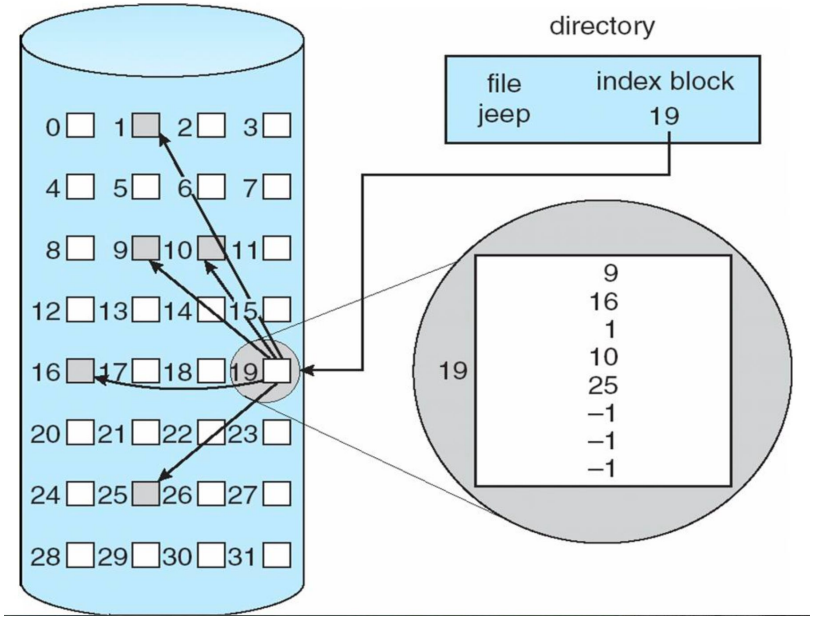


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

> Advantages
  - No fragmentation
  - Files can easily grow dynamically
  - Fast random access (**How?** Cache inodes)

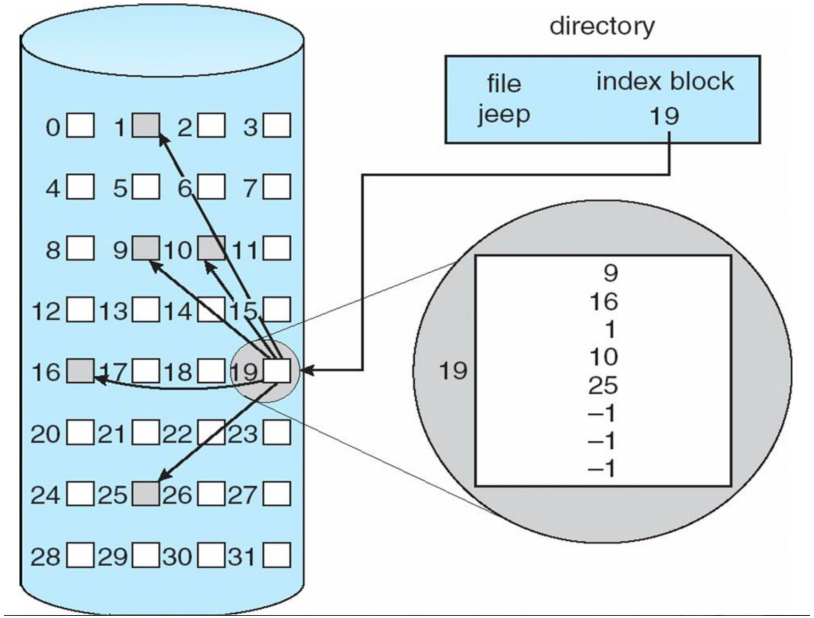


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

› Advantages

- No fragmentation
- Files can easily grow dynamically
- Fast random access (**How?** Cache inodes)

› Disadvantages

- Sequential bandwidth may not be good
- What if one index block is not big enough?

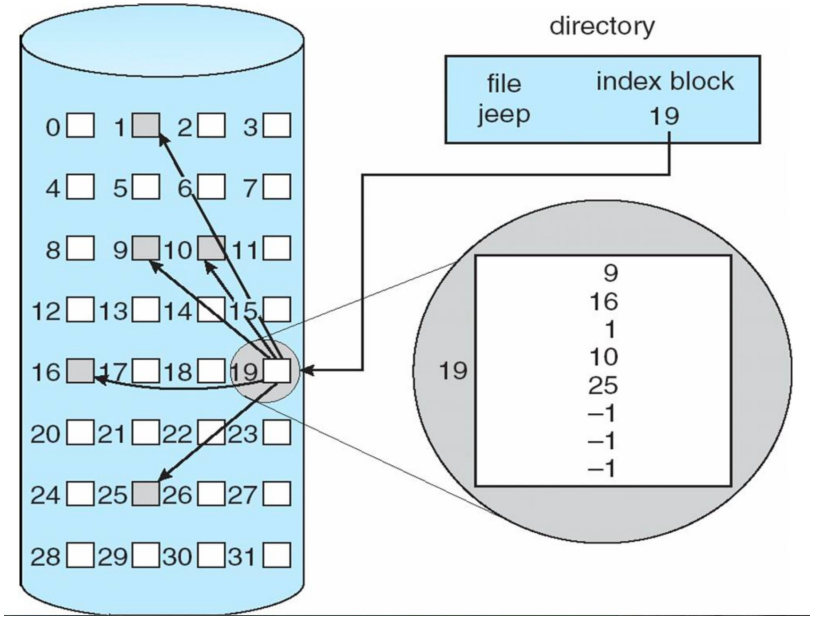


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

› Advantages
  - No fragmentation
  - Files can easily grow dynamically
  - Fast random access (**How?** Cache inodes)

› Disadvantages
  - Sequential bandwidth may not be good
  - What if one index block is not big enough?

    …We've seen this story before!

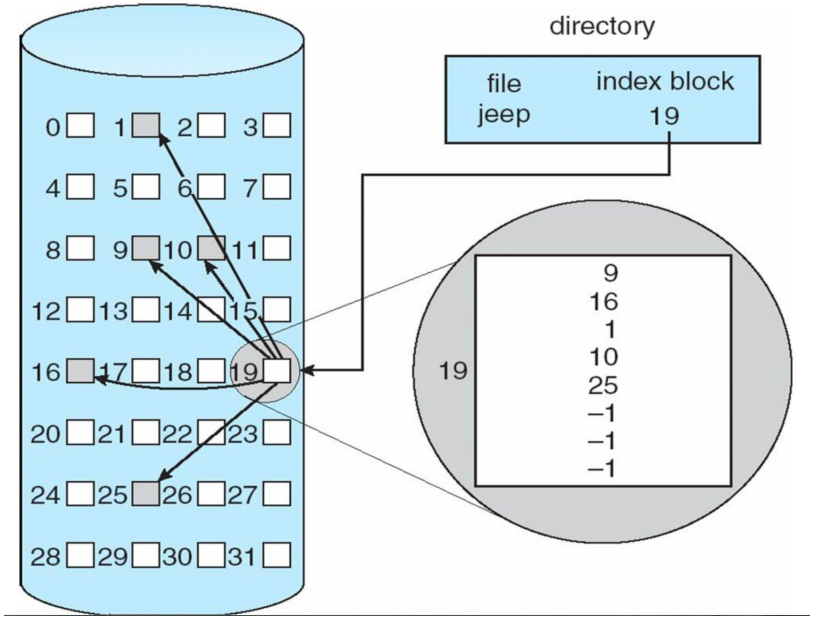


Image from: OS Concepts, by A. Silberschatz et al.

Multilevel Indexed allocation (Linux ext2/3)

Multilevel Indexed allocation (Linux ext2/3)

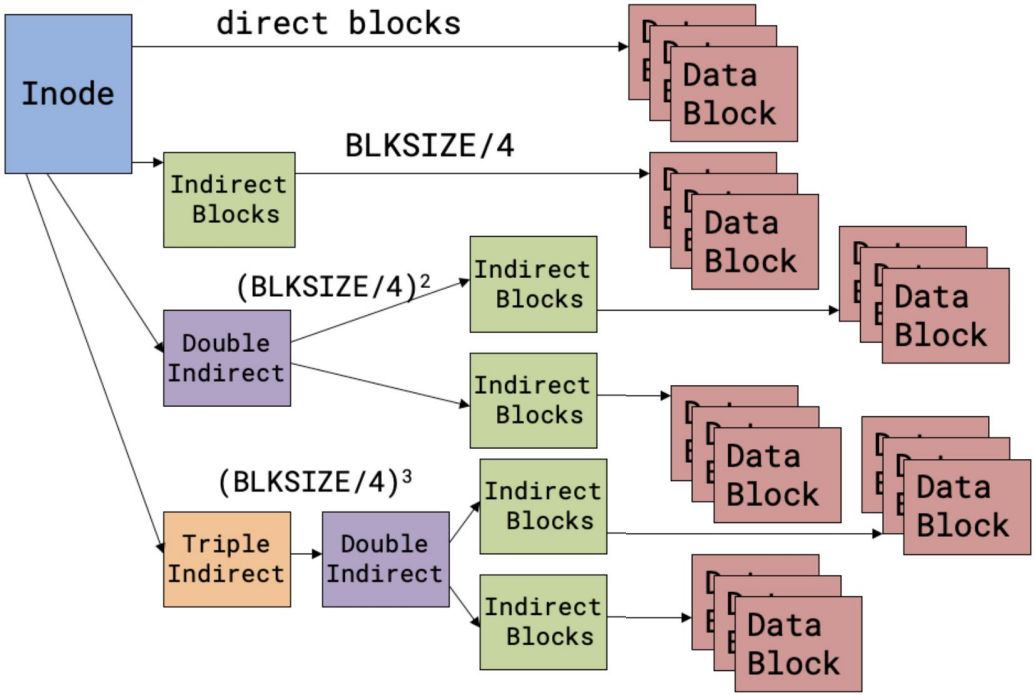
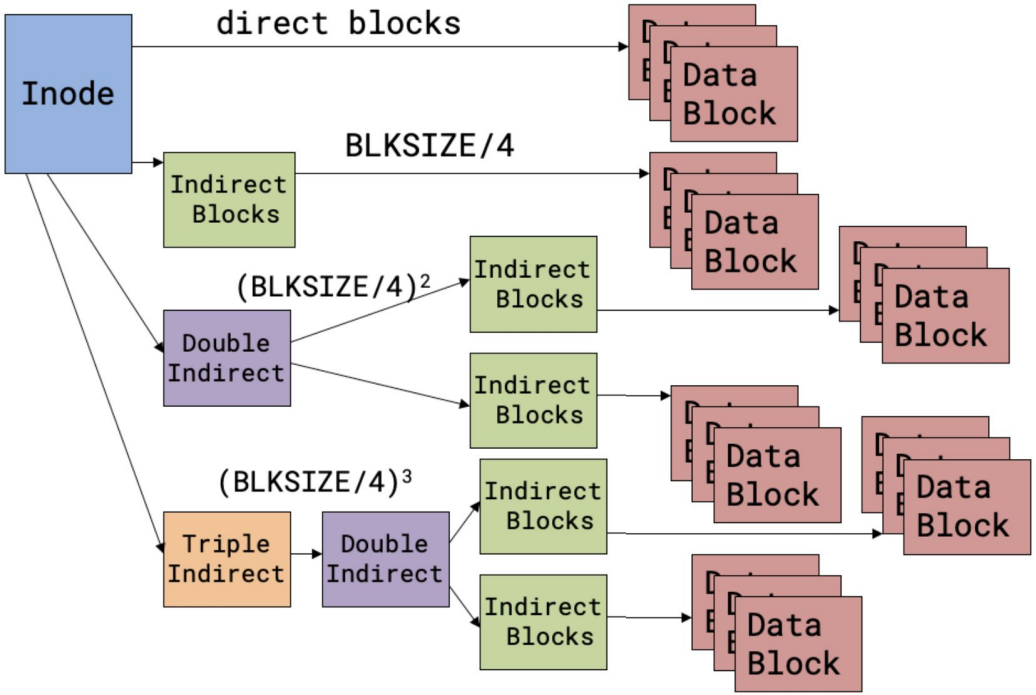


Image from: [COMS 4118, Operating Systems I]

Multilevel Indexed allocation (Linux ext2/3)



Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

Image from: COMS 4118, Operating Systems I

Multilevel Indexed allocation (Linux ext2/3)

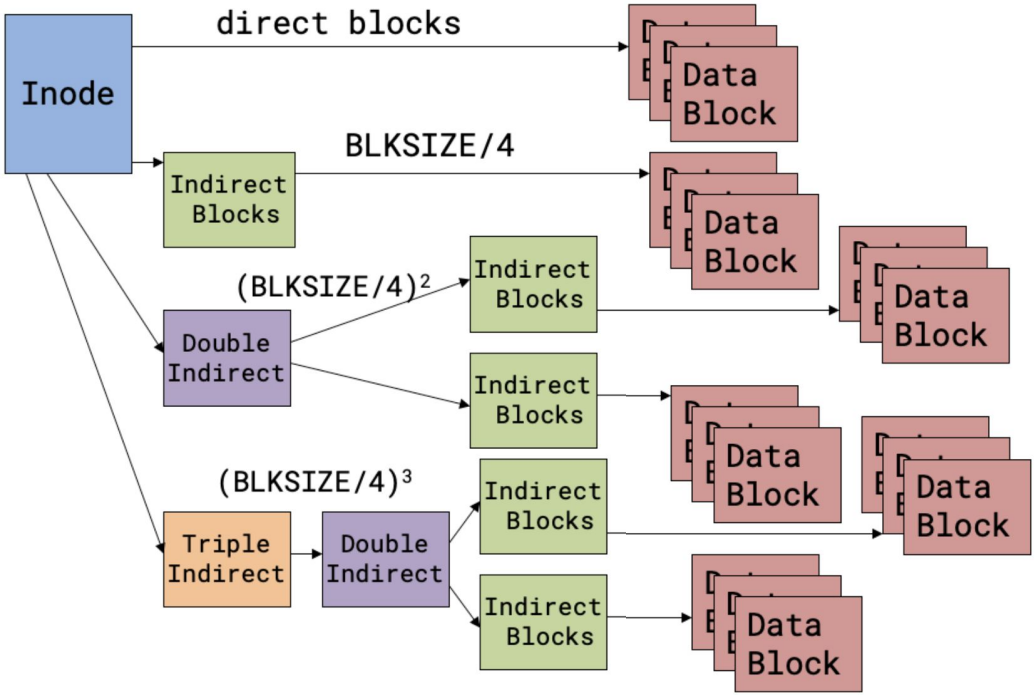


Image from: COMS 4118, Operating Systems I

Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

What is the max supported file size?

Multilevel Indexed allocation (Linux ext2/3)

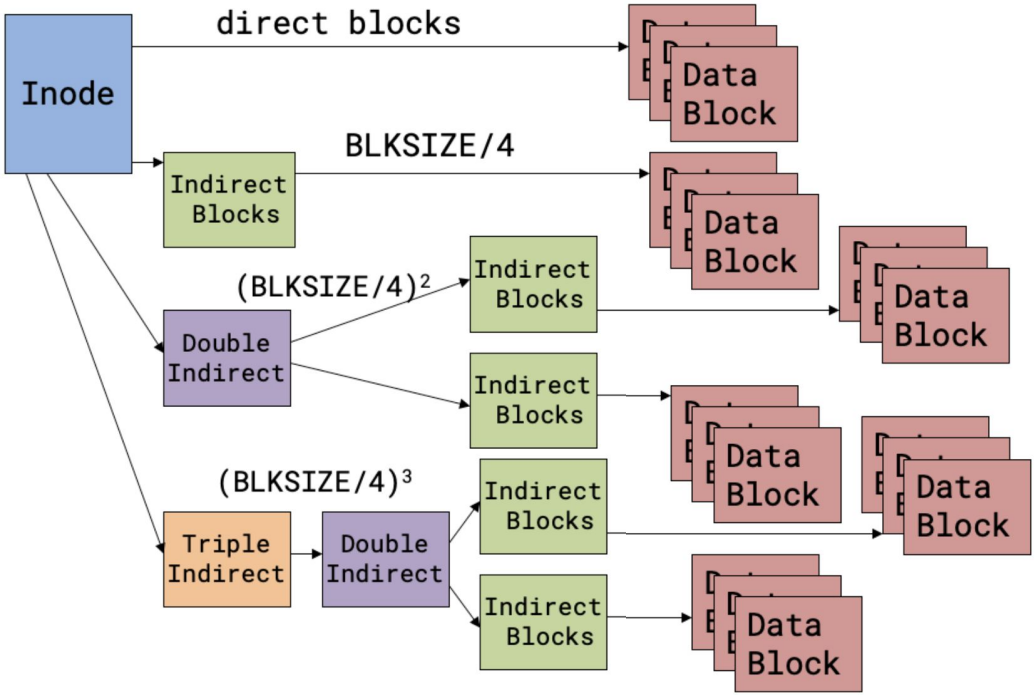


Image from: COMS 4118, Operating Systems I

Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

What is the max supported file size?

> (12 + 1024 + 1024^2 + 1024^3) *4KB > **4TB**

Multilevel Indexed allocation (Linux ext2/3)

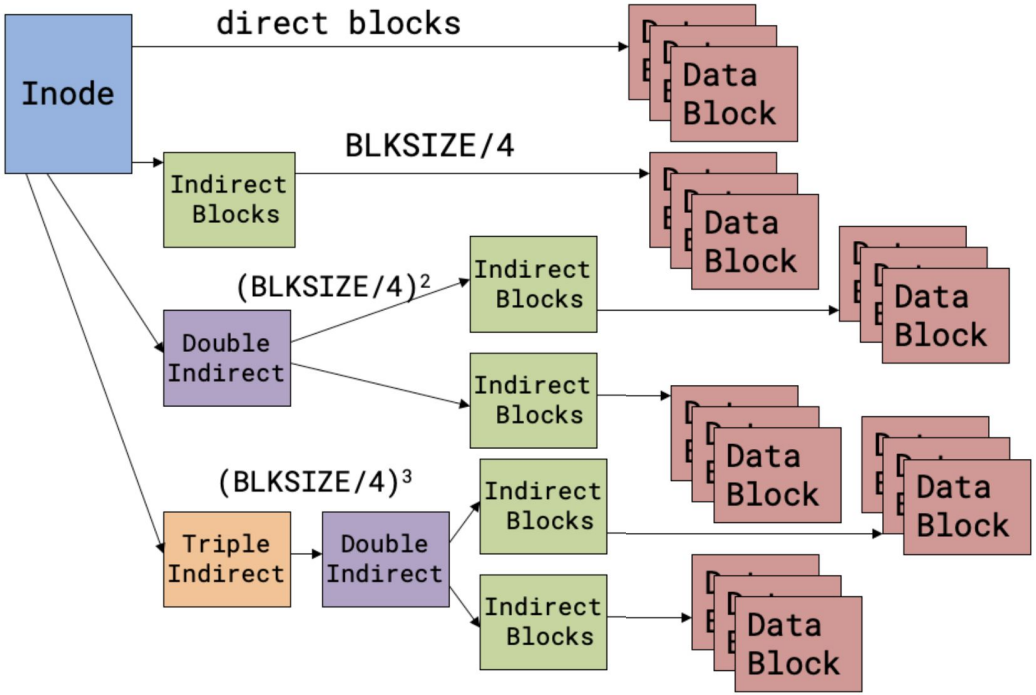


Image from: COMS 4118, Operating Systems I

Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

What is the max supported file size?

> (12 + 1024 + 1024^2 + 1024^3) *4KB > **4TB**

And for what index size?

Multilevel Indexed allocation (Linux ext2/3)

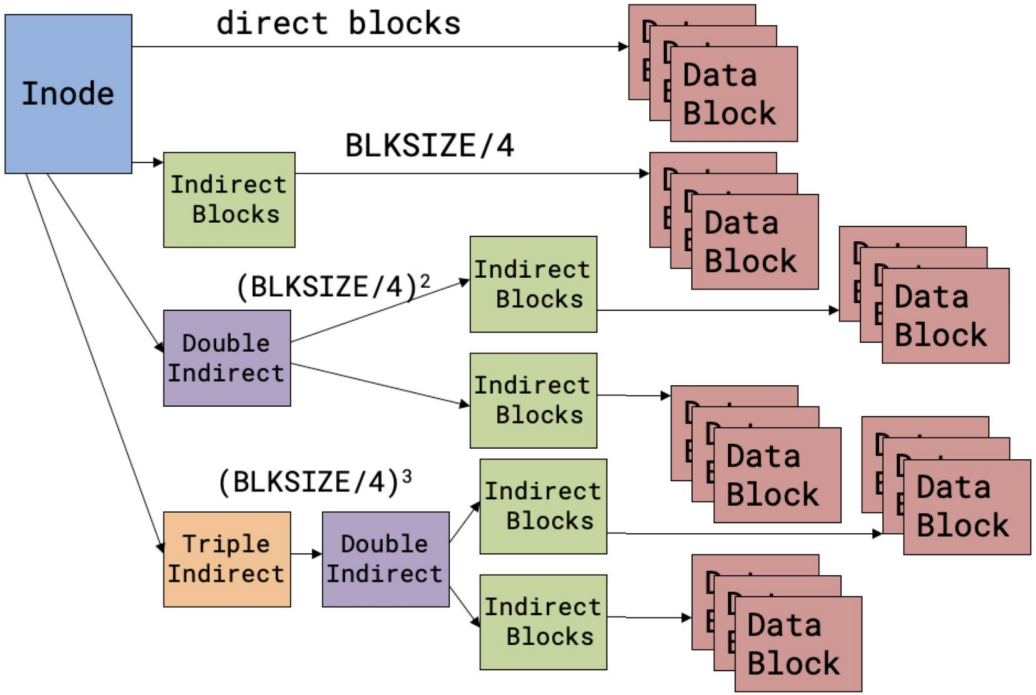


Image from: COMS 4118, Operating Systems I

Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

What is the max supported file size?

> (12 + 1024 + 1024^2 + 1024^3) *4KB > **4TB**

And for what index size?

> 12 + (1 + 1024 + 1024^2) * 4KB ~ **4GB**

Multilevel Indexed allocation (Linux ext2/3)

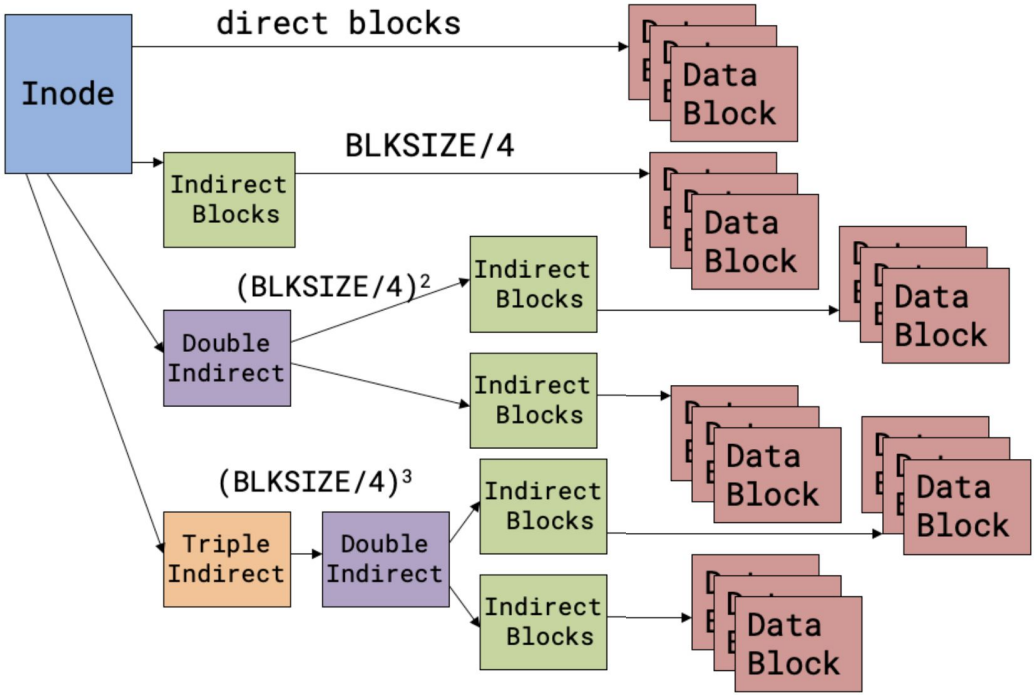


Image from: COMS 4118, Operating Systems I

Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

What is the max supported file size?

> (12 + 1024 + 1024^2 + 1024^3) *4KB > **4TB**

And for what index size?

> 12 + (1 + 1024 + 1024^2) * 4KB ~ **4GB**

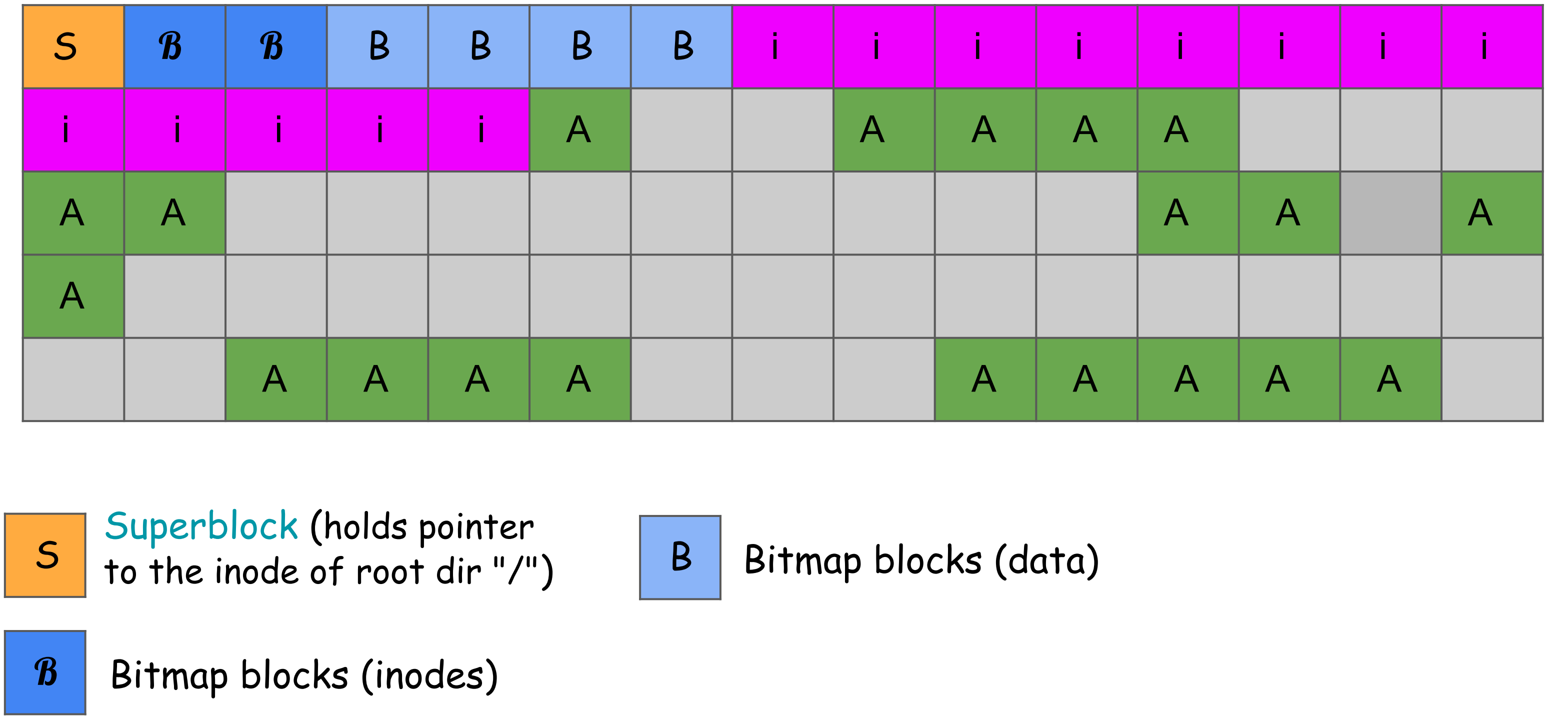
Index grows dynamically, on demand...

Simplified Filesystem Layout (Linux ext2/3)

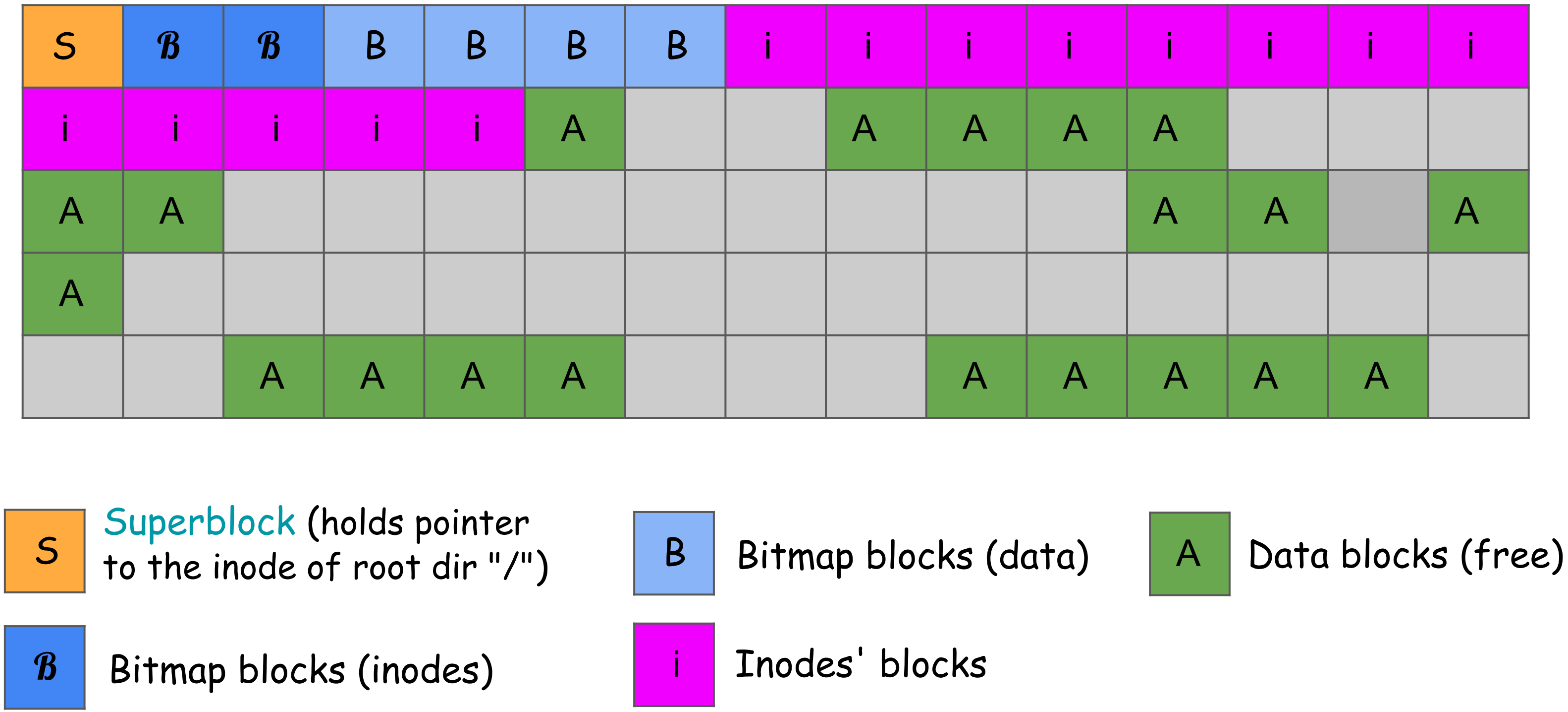
S	B	B	B	B	B	B	i	i	i	i	i	i	i	i
i	i	i	i	i	A			A	A	A	A			
A	A										A	A		A
A														
		A	A	A	A				A	A	A	A	A	

S — Superblock (holds pointer to the inode of root dir "/")

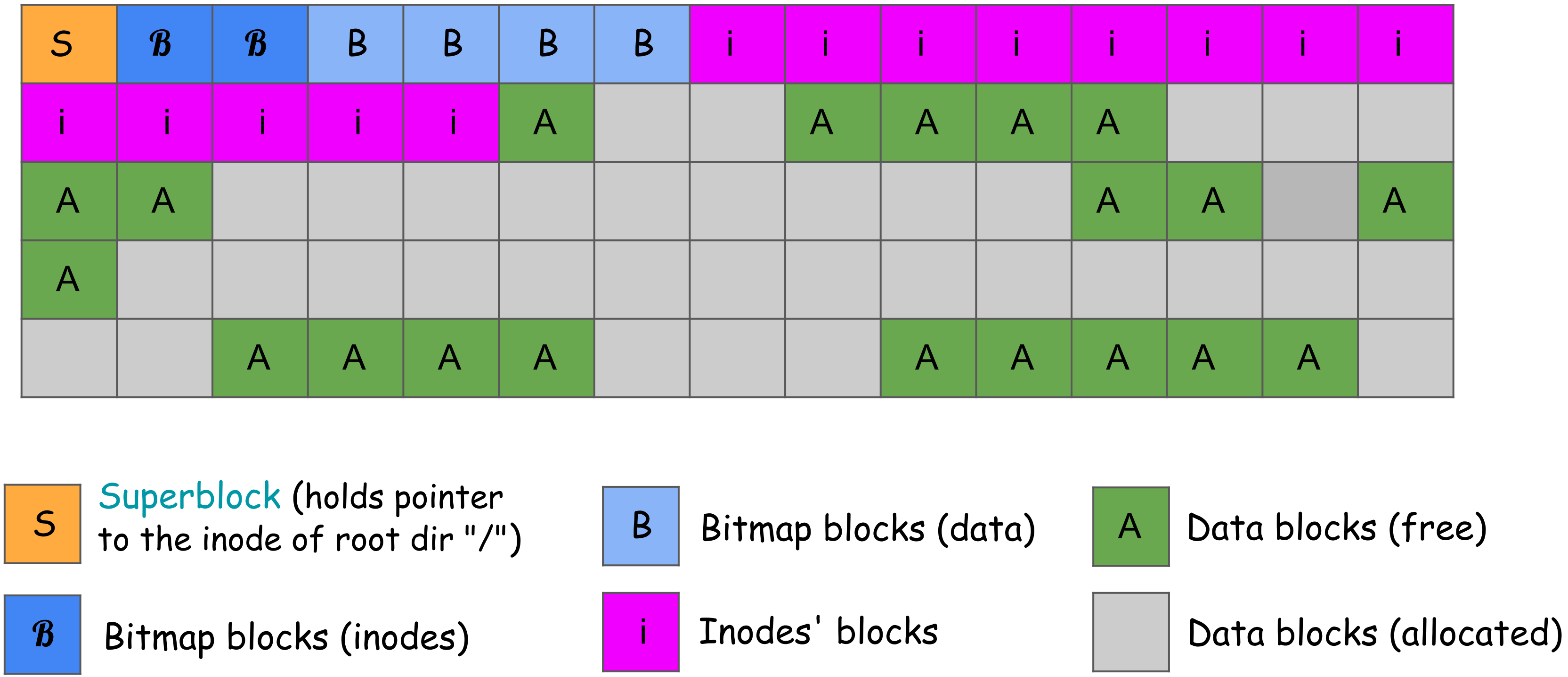
Simplified Filesystem Layout (Linux ext2/3)



Simplified Filesystem Layout (Linux ext2/3)



Simplified Filesystem Layout (Linux ext2/3)



File and Directory Names

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it
  - "/foo/test.txt" => "/" -> "foo" -> "test.txt"

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it
  - "/foo/test.txt" => "/" -> "foo" -> "test.txt"
  - "/foo/bar/test.txt" => "/" -> "foo" -> "bar" -> "test.txt"

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it
  - "/foo/test.txt" => "/" -> "foo" -> "test.txt"
  - "/foo/bar/test.txt" => "/" -> "foo" -> "bar" -> "test.txt"

File and directory aliases

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it
  - "/foo/test.txt" => "/" -> "foo" -> "test.txt"
  - "/foo/bar/test.txt" => "/" -> "foo" -> "bar" -> "test.txt"

File and directory aliases

> Hard link: Associates a name with an inode (>=1 files, =1 dirs)

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it
  - "/foo/test.txt" => "/" -> "foo" -> "test.txt"
  - "/foo/bar/test.txt" => "/" -> "foo" -> "bar" -> "test.txt"

File and directory aliases

> Hard link: Associates a name with an inode (>=1 files, =1 dirs)

> Soft link: Associates a name with an inode of a file containing paths to files

Path Name Resolution

Path Name Resolution

We need a fast translation from path names to inodes

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"
4. Read inode block of "/foo" to look up data blocks of "/foo"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"
4. Read inode block of "/foo" to look up data blocks of "/foo"
5. Read data block of "/foo" to look up inode number of "/foo/test.txt"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"
4. Read inode block of "/foo" to look up data blocks of "/foo"
5. Read data block of "/foo" to look up inode number of "/foo/test.txt"
6. Read inode block of "/foo/test.txt" to look up data blocks of "foo/test.txt"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"
4. Read inode block of "/foo" to look up data blocks of "/foo"
5. Read data block of "/foo" to look up inode number of "/foo/test.txt"
6. Read inode block of "/foo/test.txt" to look up data blocks of "foo/test.txt"
7. Read data blocks of "foo/test.txt"

Path Name Resolution

We need a fast translation from path names to inodes

> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

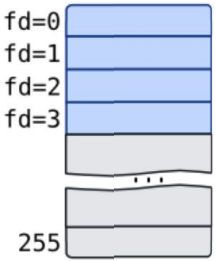
1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"
4. Read inode block of "/foo" to look up data blocks of "/foo"
5. Read data block of "/foo" to look up inode number of "/foo/test.txt"
6. Read inode block of "/foo/test.txt" to look up data blocks of "foo/test.txt"
7. Read data blocks of "foo/test.txt"

Need to speed this translation disaster up

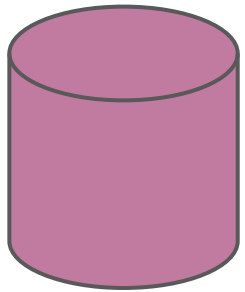
Linux File System Data Structures

Linux File System Data Structures

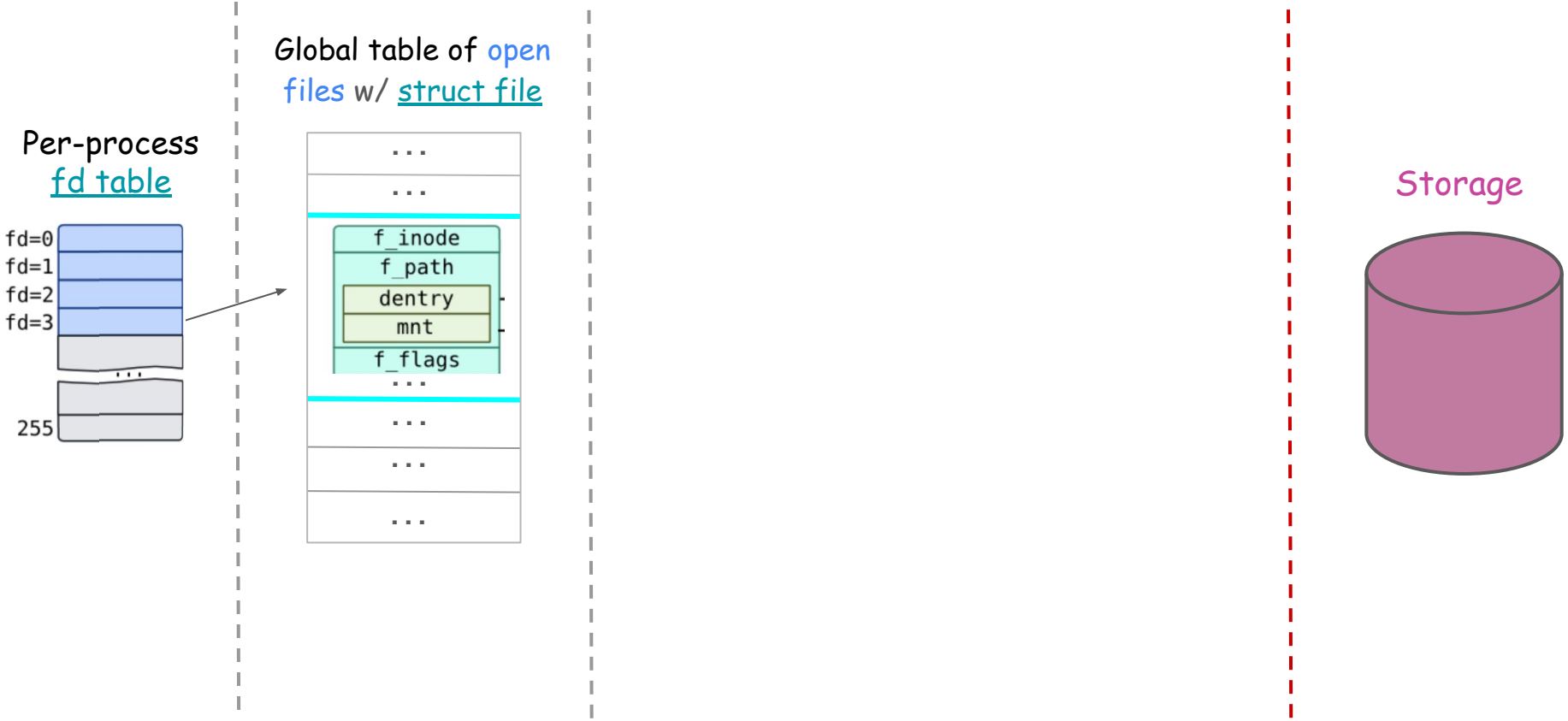
Per-process
fd table



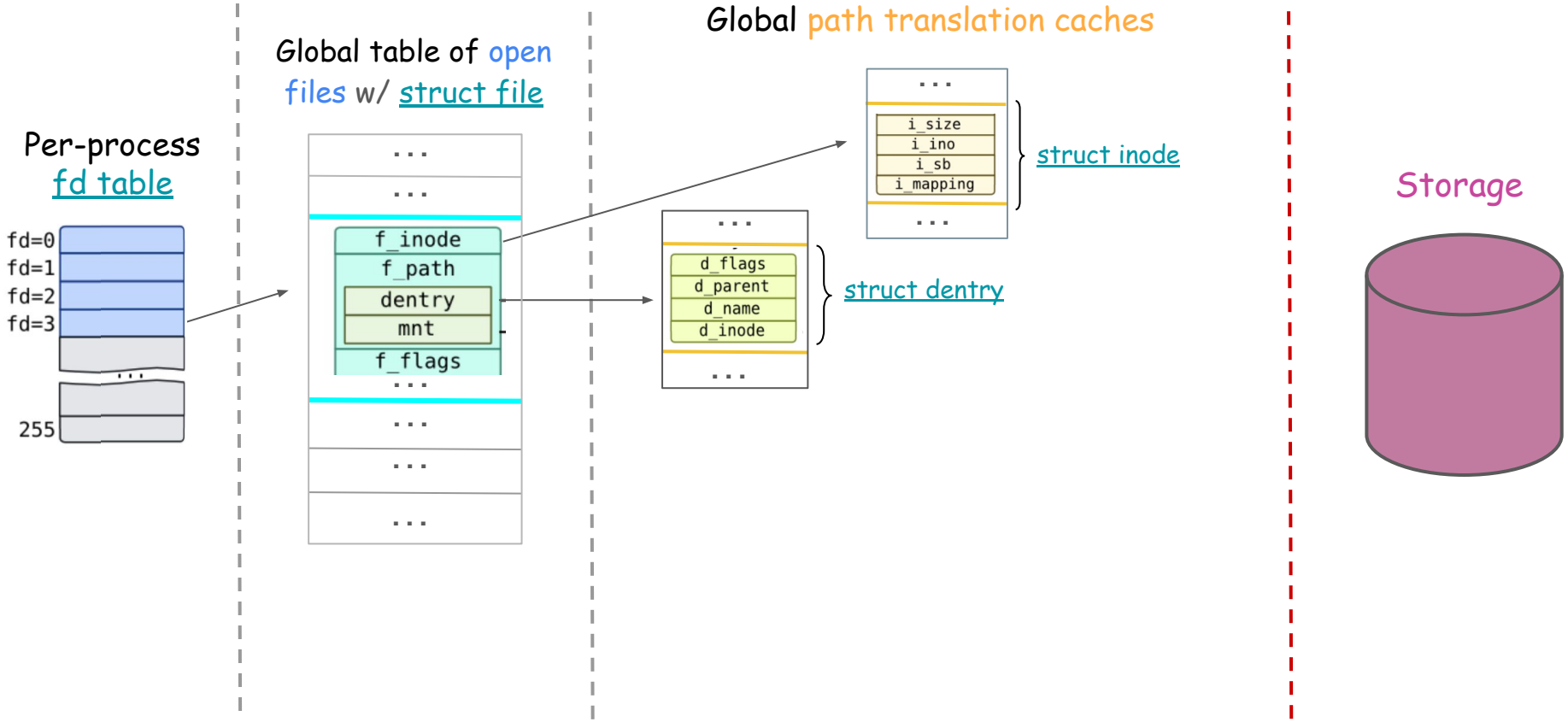
Storage



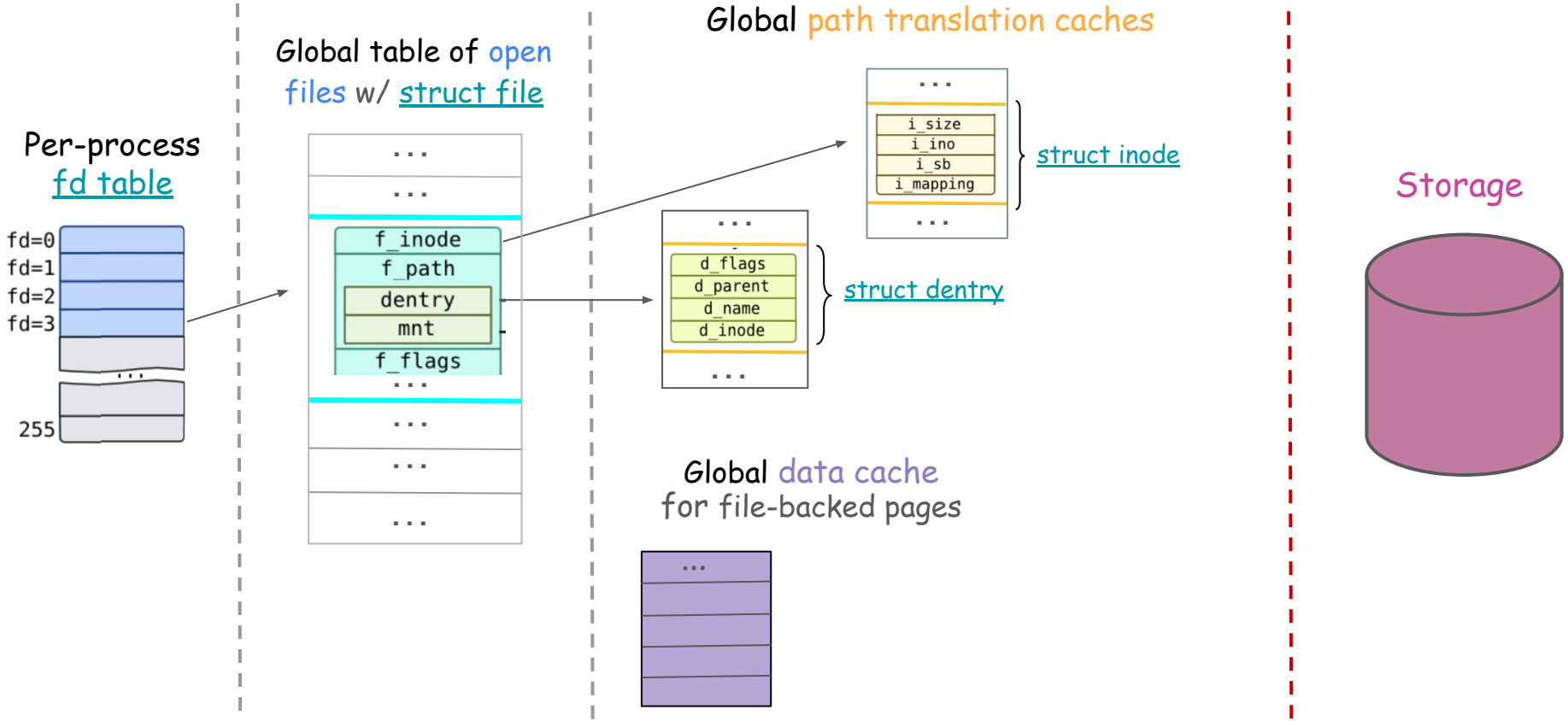
Linux File System Data Structures



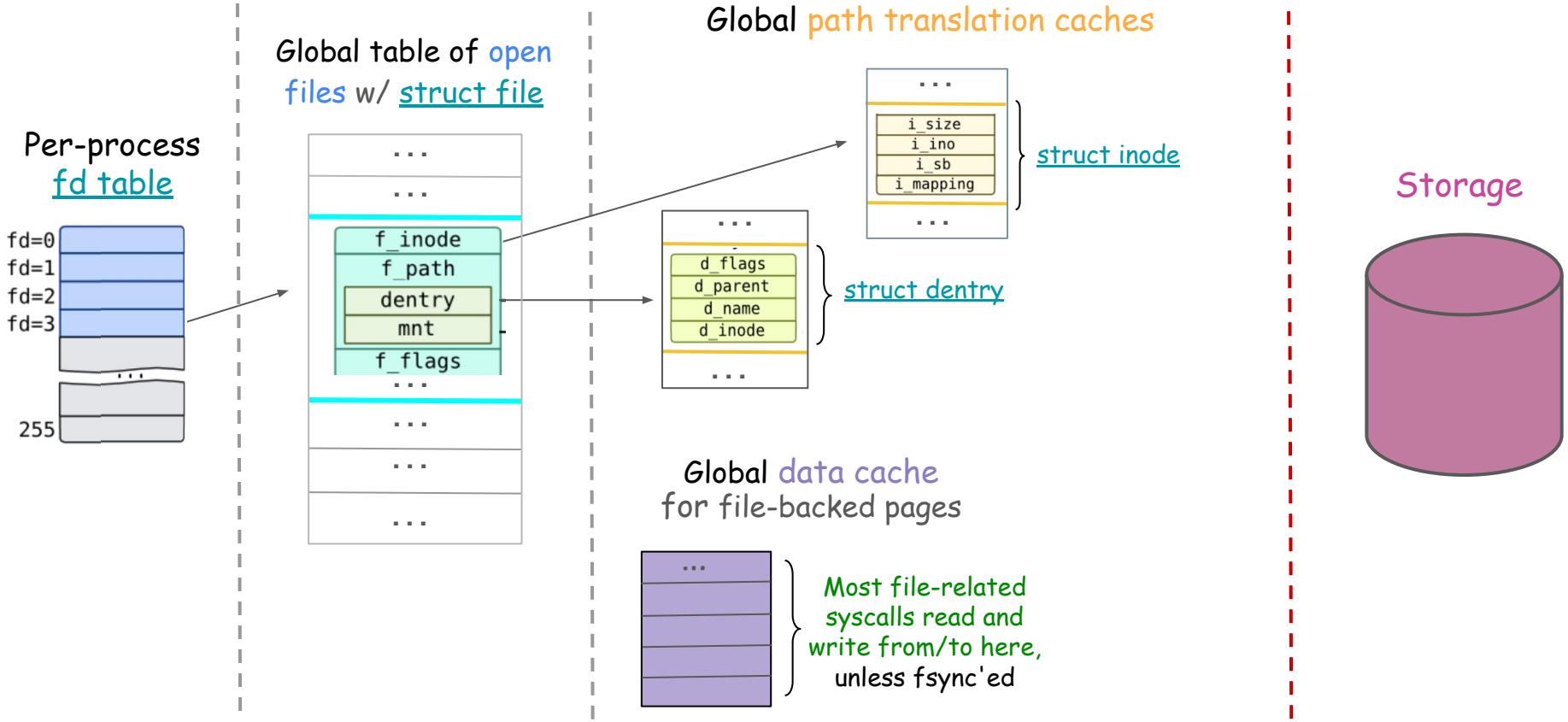
Linux File System Data Structures



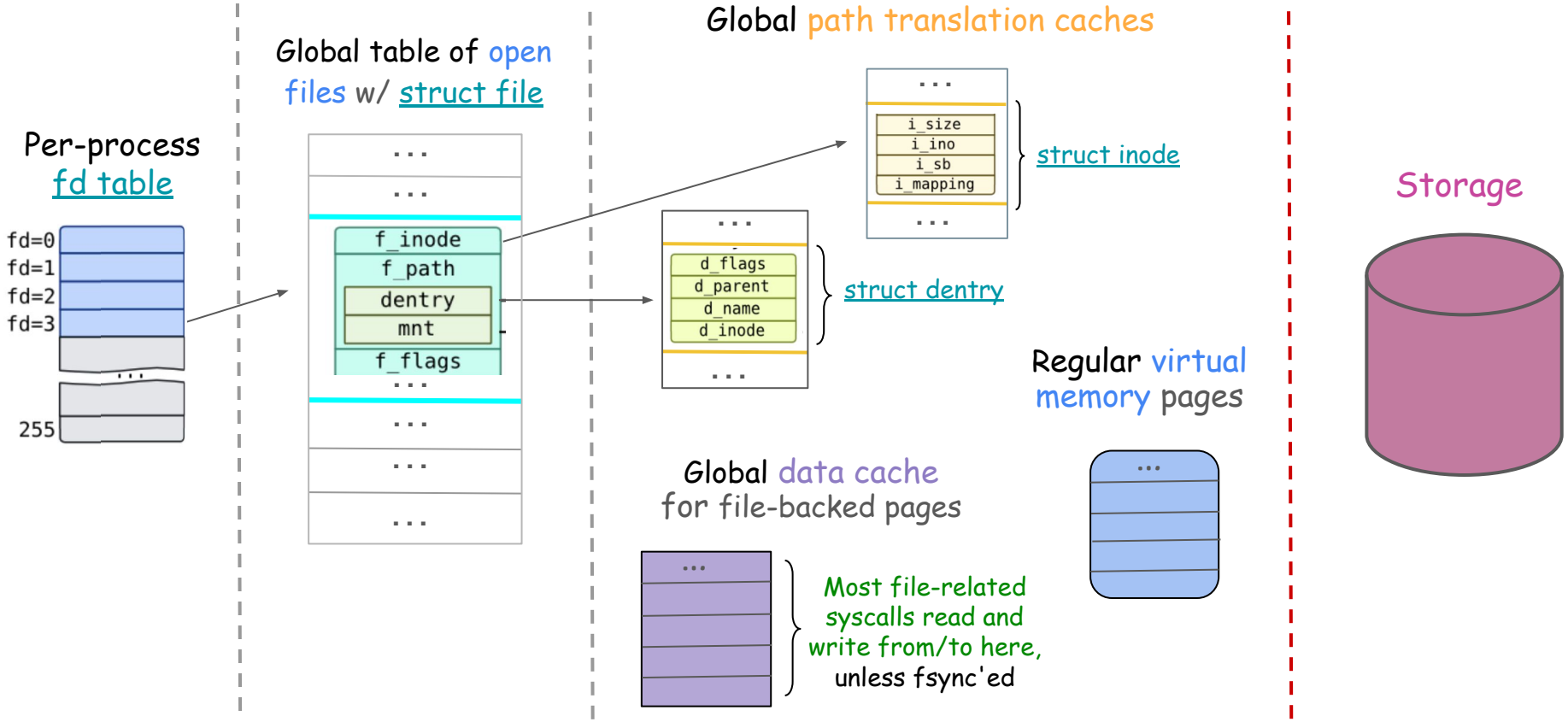
Linux File System Data Structures



Linux File System Data Structures



Linux File System Data Structures



The consistent update problem

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

› Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

› Storage devices allow atomic writes of one block at a time

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

> Storage devices allow atomic writes of one block at a time

> Crashes may happen at any time leaving the fs inconsistent

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

> Storage devices allow atomic writes of one block at a time

> Crashes may happen at any time leaving the fs inconsistent

Update data	Update data bitmap	Update inode	Outcome
yes	no	no	
no	yes	no	
no	no	yes	

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

> Storage devices allow atomic writes of one block at a time

> Crashes may happen at any time leaving the fs inconsistent

Update data	Update data bitmap	Update inode	Outcome
yes	no	no	Missed update
no	yes	no	Space leak
no	no	yes	fs inconsistent

The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

› Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

› Storage devices allow atomic writes of one block at a time

› Crashes may happen at any time leaving the fs inconsistent

Update data	Update data bitmap	Update inode	Outcome
yes	no	no	Missed update
no	yes	no	Space leak
no	no	yes	fs inconsistent

} Could read garbage data—Example?

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order: Write data blk -> Write data bitmap blk -> Write inode blk

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order: Write data blk -> Write data bitmap blk -> Write inode blk

- Ensures inodes never point to uninitialized data!

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order: Write data blk -> Write data bitmap blk -> Write inode blk

- Ensures inodes never point to uninitialized data!
- Can leak resources

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order: Write data blk -> Write data bitmap blk -> Write inode blk

- Ensures inodes never point to uninitialized data!
- Can leak resources (fixable: run *fsck* periodically)

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order: Write data blk -> Write data bitmap blk -> Write inode blk

- Ensures inodes never point to uninitialized data!
- Can leak resources (fixable: run *fsck* periodically)
- Cannot reorder writes and execute asynchronously <= Dealbreaker

Journaling (also called "write-ahead" logging)

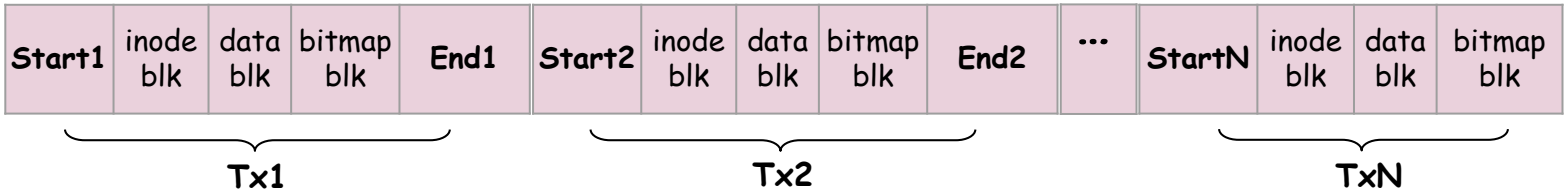
Journaling (also called "write-ahead" logging)

> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

Journaling (also called "write-ahead" logging)

> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

I) Journal Write: Write all blocks of TxN to the journal w/o *End* block

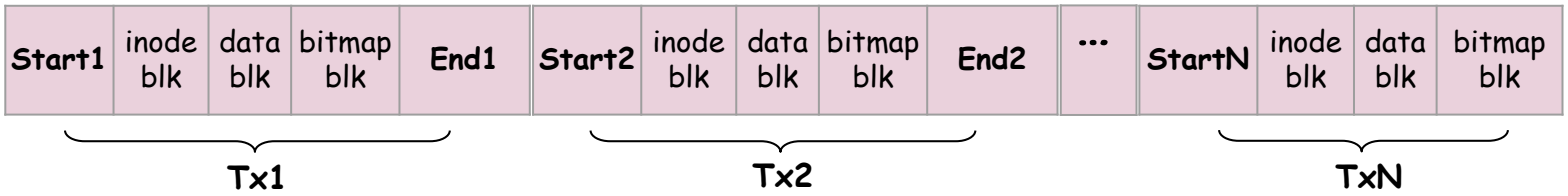


Journaling (also called "write-ahead" logging)

> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

I) Journal Write: Write all blocks of TxN to the journal w/o *End* block

II) Journal Commit: Write TxN's *End* block to the journal



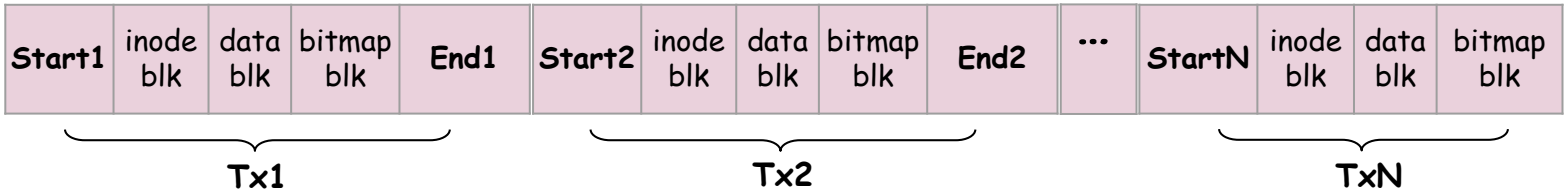
Journaling (also called "write-ahead" logging)

> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

I) Journal Write: Write all blocks of TxN to the journal w/o *End* block

II) Journal Commit: Write TxN's *End* block to the journal

III) Journal Checkpoint: After data and metadata blks have been updated at their final storage destination, update ckpt postion



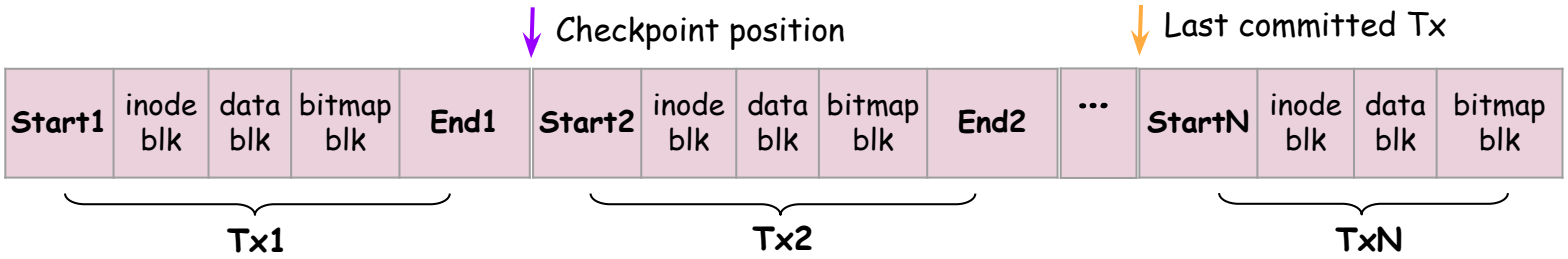
Journaling (also called "write-ahead" logging)

> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

I) Journal Write: Write all blocks of TxN to the journal w/o *End* block

II) Journal Commit: Write TxN's *End* block to the journal

III) Journal Checkpoint: After data and metadata blks have been updated at their final storage destination, update ckpt postion



Journaling (also called "write-ahead" logging)

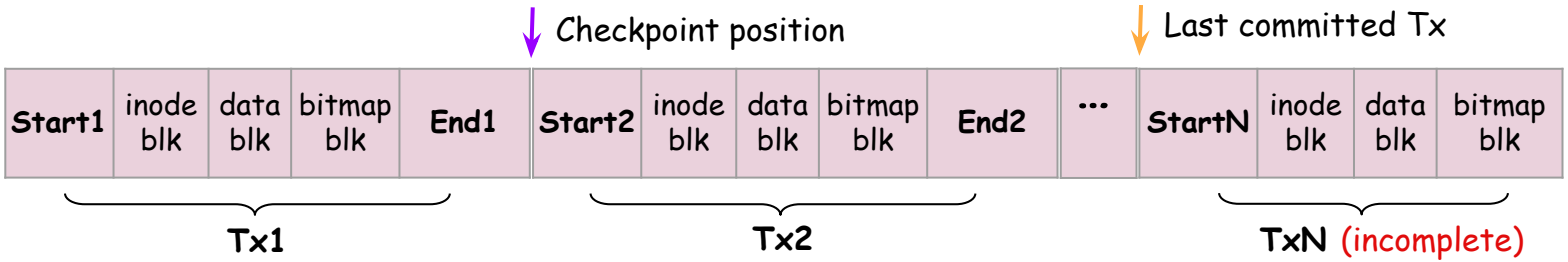
> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

I) Journal Write: Write all blocks of TxN to the journal w/o *End* block

II) Journal Commit: Write TxN's *End* block to the journal

III) Journal Checkpoint: After data and metadata blks have been updated at their final storage destination, update ckpt postion

Crash occured? Replaying committed transactions after the last checkpoint will bring the fs to a consistent state (crash tolerance)



Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

Fault Tolerance

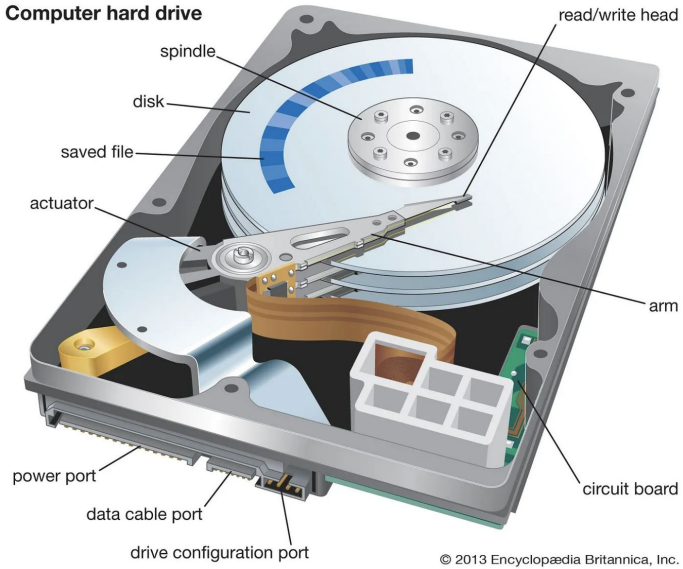
A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

- Storage failures are irreversible => No restart button

Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

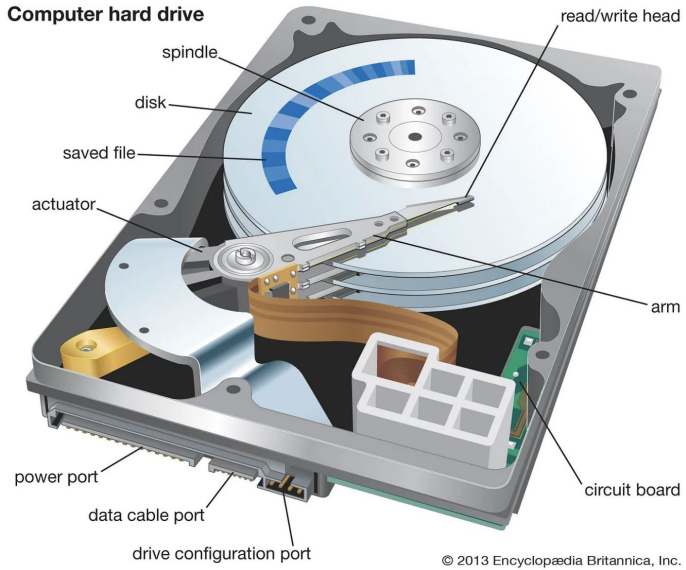
- Storage failures are irreversible => No restart button



Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

- Storage failures are irreversible => No restart button



It is impressive that this piece of machinery carried us for so many decades...

Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

- How to build fault tolerant systems using unreliable hardware?

Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

- How to build fault tolerant systems using unreliable hardware? Replication

Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

- How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

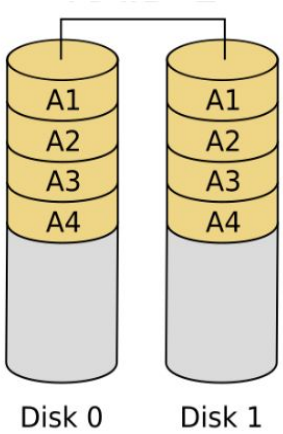
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

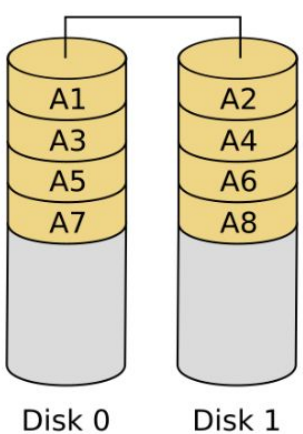
> How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

RAID-1 (disks >= 2)



RAID-0 (disks >= 2)



2x Throughput

No Fault tolerance

> Worse than single disk

Images from: Wikipedia

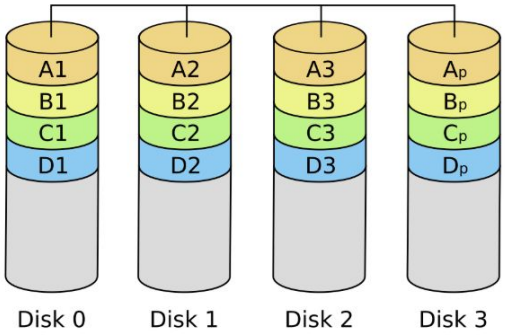
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

> How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

RAID-4 (disks >= 3)



Images from: Wikipedia

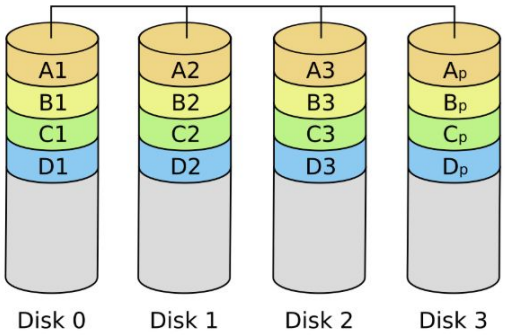
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

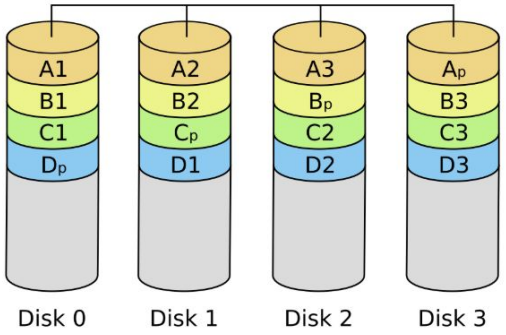
> How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

RAID-4 (disks >= 3)



RAID-5 (disks >= 3)



Images from: Wikipedia

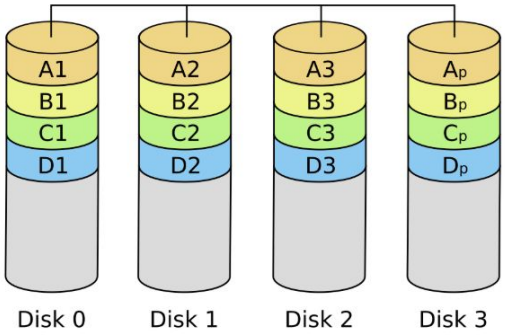
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

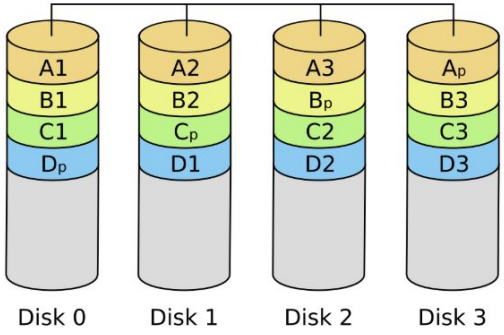
› How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

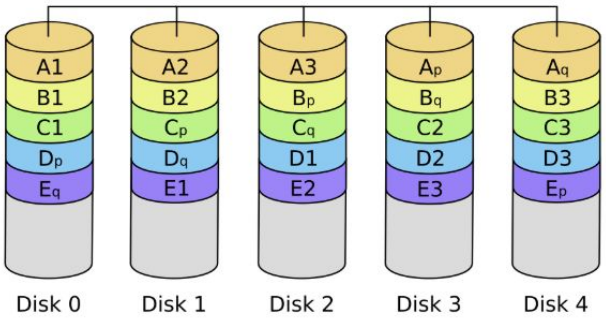
RAID-4 (disks >= 3)



RAID-5 (disks >= 3)



RAID-6 (disks >= 4)



Images from: Wikipedia

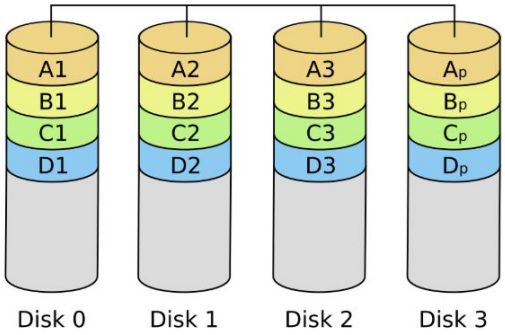
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

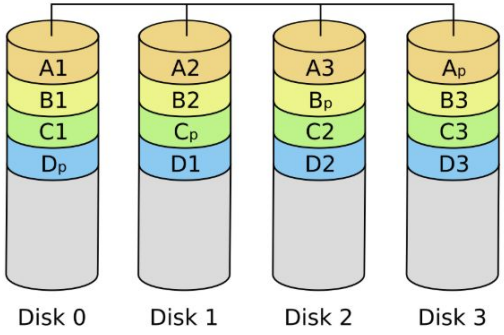
› How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

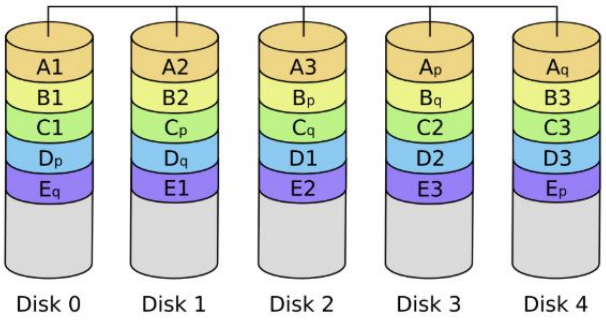
RAID-4 (disks >= 3)



RAID-5 (disks >= 3)



RAID-6 (disks >= 4)



>> **Fault Tolerance:** RAID-0 < RAID-4 ≈ RAID-5 < RAID-6 < RAID-1

Images from: Wikipedia