We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

Q1

When comparing random access to sequential access for reading data from a hard disk, which of the following statements is true?

- ○ Random access is faster due to quicker access to specific data points.
- ○ Random access and sequential access have the same speed.
- ● Sequential access is faster because it minimises head movement and rotational latency.
- ○ Disk transfer rates are not affected by the access pattern.
- ○ It depends on the locality of access

THE UNIVERSITY *of* ADELAIDE

Q2

Match the description with the term

y a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention	DMA engine
a software program that manages and abstract a peripheral attached to a computer	Device driver
a generic interface that consists of a status register, a data register and a command register	device controller
when the device's register are read using the same address space that memory	memory mapped I/O

THE UNIVERSITY *of* ADELAIDE

Q3

In computer input/output operations, what does Programmed I/O refer to?

- ○ A technique that prioritises high-speed data access over other I/O operations.
- ○ A method of data transfer that relies on hardware DMA controllers.
- ○ A form of I/O where the device initiates data transfers independently of the CPU.
- ● The CPU directly manages data transfer between the I/O device and memory.

Q4

Which of the following factors does NOT contribute to the access time of a disk read

- Number of platters
- Rotational speed
- Current location of disk's arm
- Transfer rate
- Controller overhead

THE UNIVERSITY *of* ADELAIDE

Q5

A disk may have one or more platters, each of them having 2 sides or surfaces.

Data is encoded on each surface in concentric circles called tracks

THE UNIVERSITY of ADELAIDE

Q6

In older hard disk drive (HDD) systems, what method was used to address specific data locations of a block on the disk?

- ( ) Platter-Surface-Cylinder (PSC)
- ( ) Cylinder-Platter-Surface (CPS)
- ( ) Track-Platter-Block (TPB)
- (•) Cylinder-Head-Sector (CHS)

Q7

What is the purpose of track skew in hard disk drives (HDDs)?

- It enhances the data encryption process to improve security.
- It helps to prevent data loss in case of a power outage.
- It aligns the read/write heads with specific tracks to optimise data access.
- It reduces the mechanical wear and tear on the spindle motor.

Q8

In storage systems, how does Logical Block Addressing (LBA) work?

- ◯ It uses physical cylinder, head, and sector numbers to locate data.
- ◯ It utilises geographical coordinates to determine data locations.
- ◉ It relies on a virtual addressing scheme based on block numbers.
- ◯ It assigns a unique barcode to each data block for identification.
- ◯ It checks the block address for logical errors

Q9

What is the primary purpose of a cache in a hard disk drive (HDD)?

- ○ To reduce the amount of storage space required on the disk.
- ● To temporarily store frequently accessed data for faster retrieval.

Quiz 4 - Persistence

- ○ To protect the data on the disk from physical damage.
- ○ To store permanent copies of frequently accessed files.
- ○ To maintain copies of edited files for back-up

THE UNIVERSITY *of* ADELAIDE

Q10

If you had four identical drives, which of these RAID types would provide both redundancy and the most available storage space for your data?

- (selected) RAID 5
- RAID 1
- RAID 0
- RAID 8
- None of the above

Operating Systems

COMP SCI 3004 / COMP SCI 7064

Week 10 – File and Directories

make history.

THE UNIVERSITY of ADELAIDE

Persistent Storage

Keep data intact even if there is a power loss.

- Hard disk drive
- Solid-state storage device

Two key abstractions in the virtualisation of storage

- File
- Directory

What is a File?

Array of persistent bytes that can be read/written

File system consists of many files

- Refers to collection of files
- Also refers to part of OS that manages those files

Files need names to access correct one

Three types of names

- Unique id: i-node numbers
- Path
- File descriptor

THE UNIVERSITY of ADELAIDE

What does “i” stand for?

“In truth, I don't know either. It was just a term that we started to use. ‘Index’ is my best guess, because of the slightly unusual file system structure that stored the access information of files as a flat array on the disk…”

~ Dennis Ritchie

THE UNIVERSITY of ADELAIDE

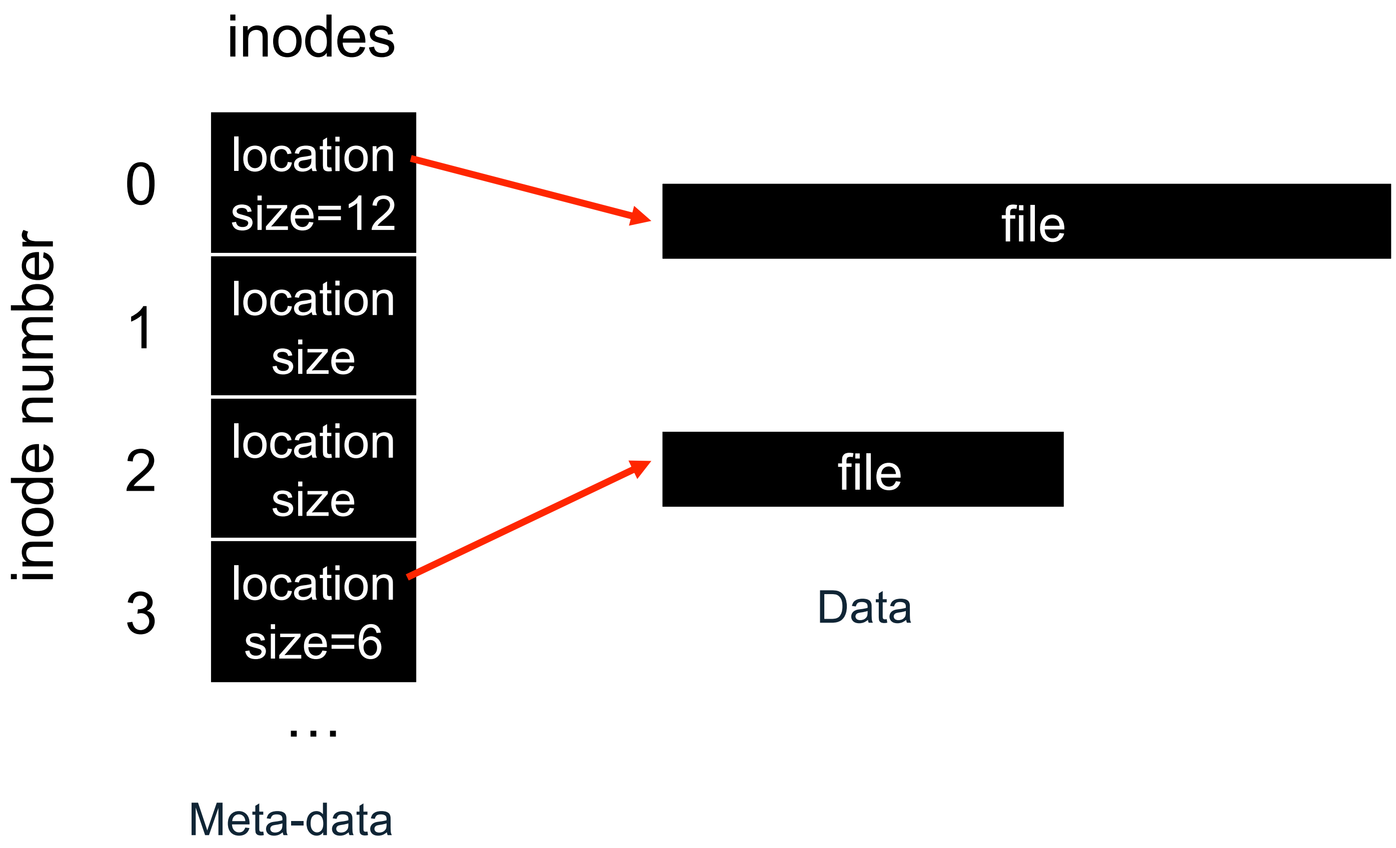
Inode Number

Each file has exactly one inode number

Inodes are unique (at a given time) within file system

Different file system may use the same number,
numbers may be recycled after deletes

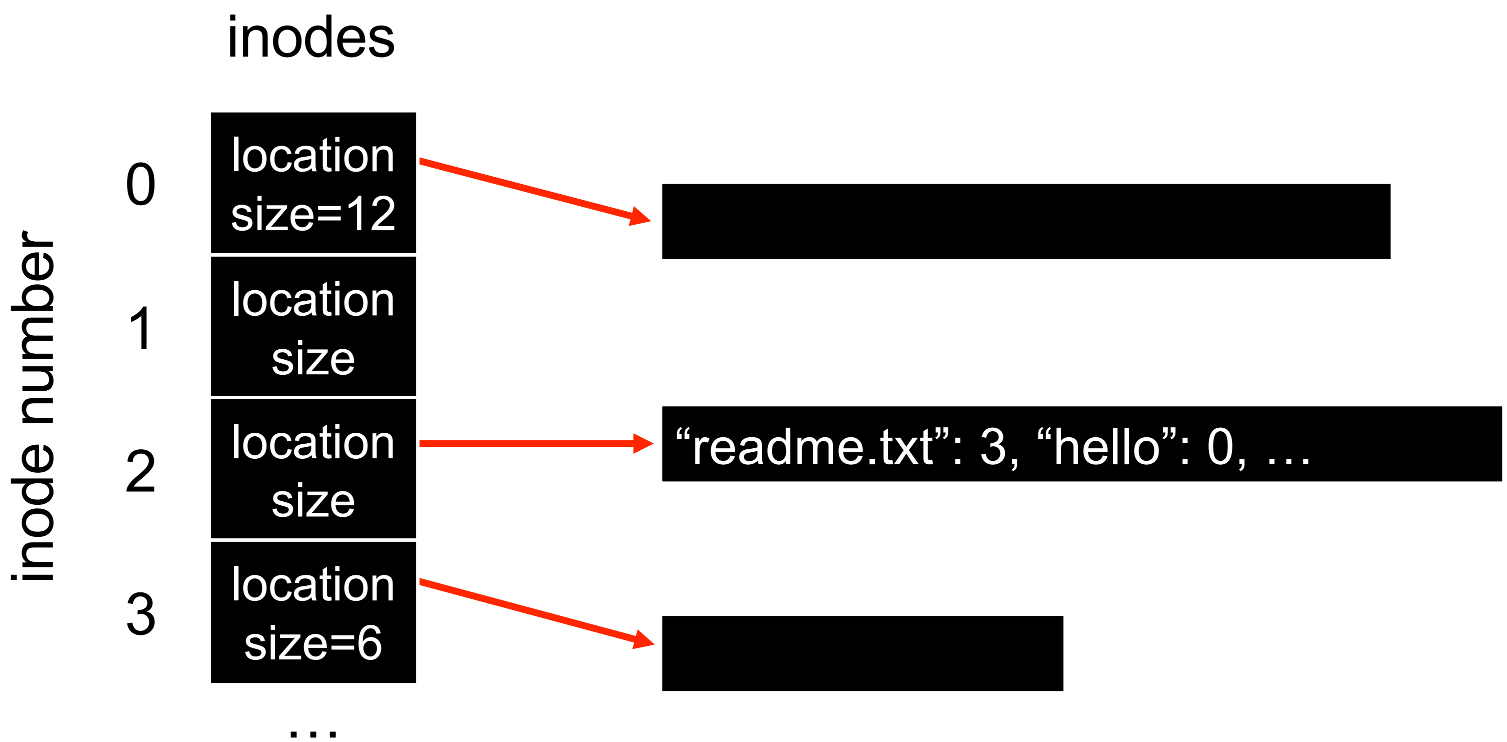
See inodes via “ls –i”; see them increment…


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
Meta-data
file
file
Data

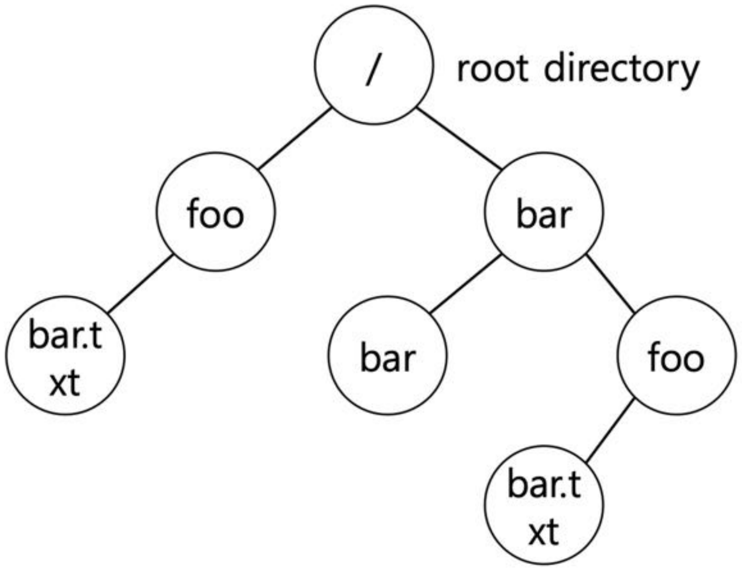
Paths

The file system interacts with i-node numbers, but for humans, strings are easier than arbitrary numbers.

Store *path-to-inode* mappings in a special file or rather a Directory! This originated in a predetermined "root" file (typically inode 2).


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“readme.txt”: 3, “hello”: 0, …

Paths - Directory Tree

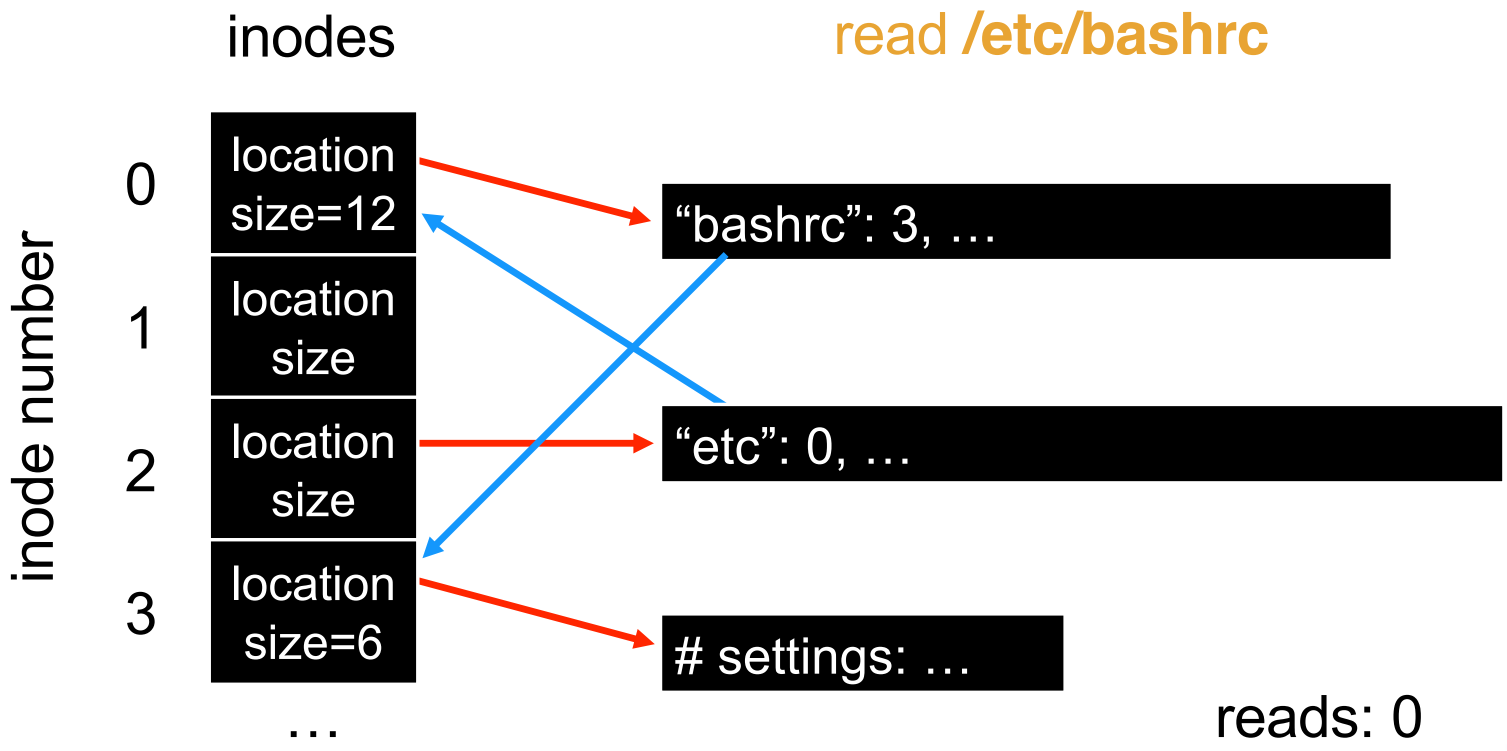


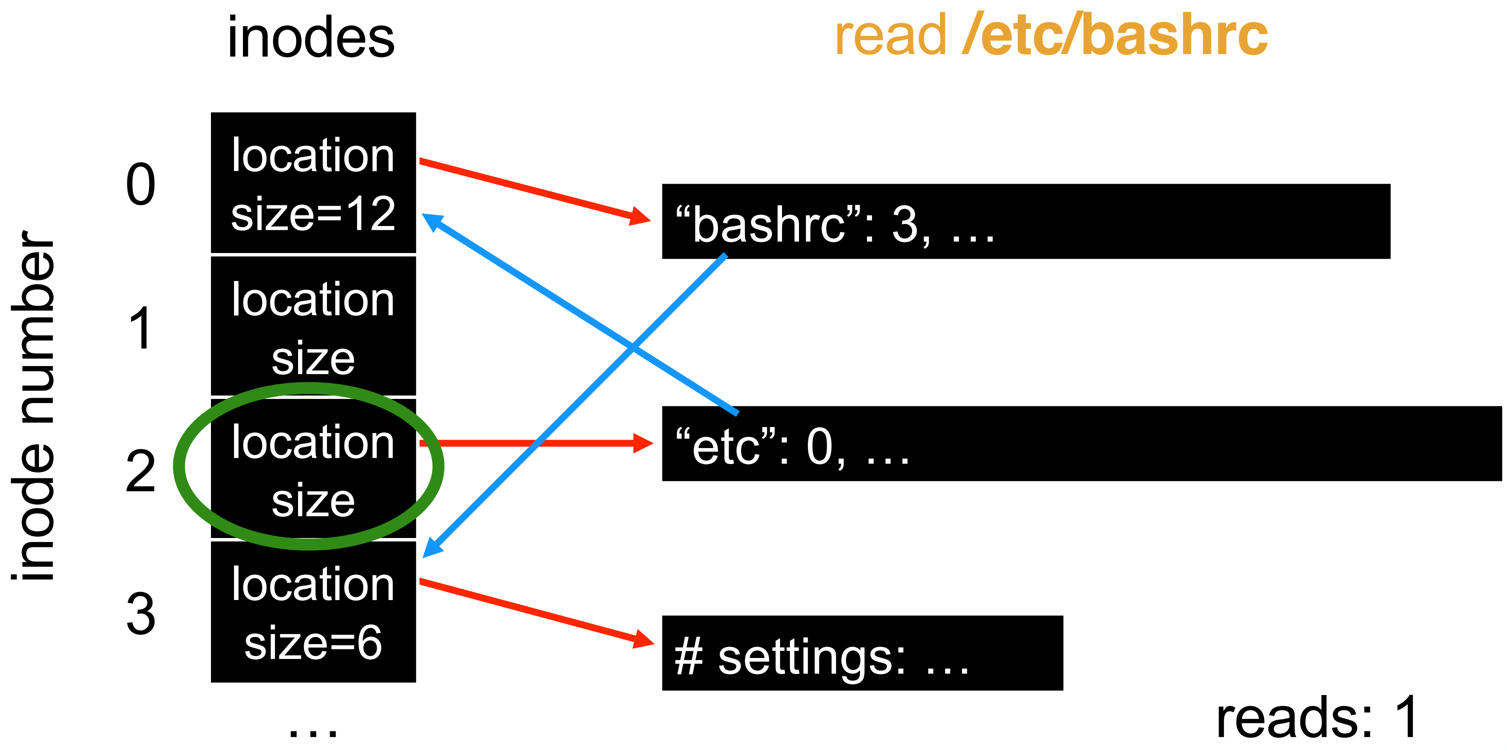
An Example Directory Tree

Valid files (absolute pathname) :
/foo/bar.txt
/bar/foo/bar.txt

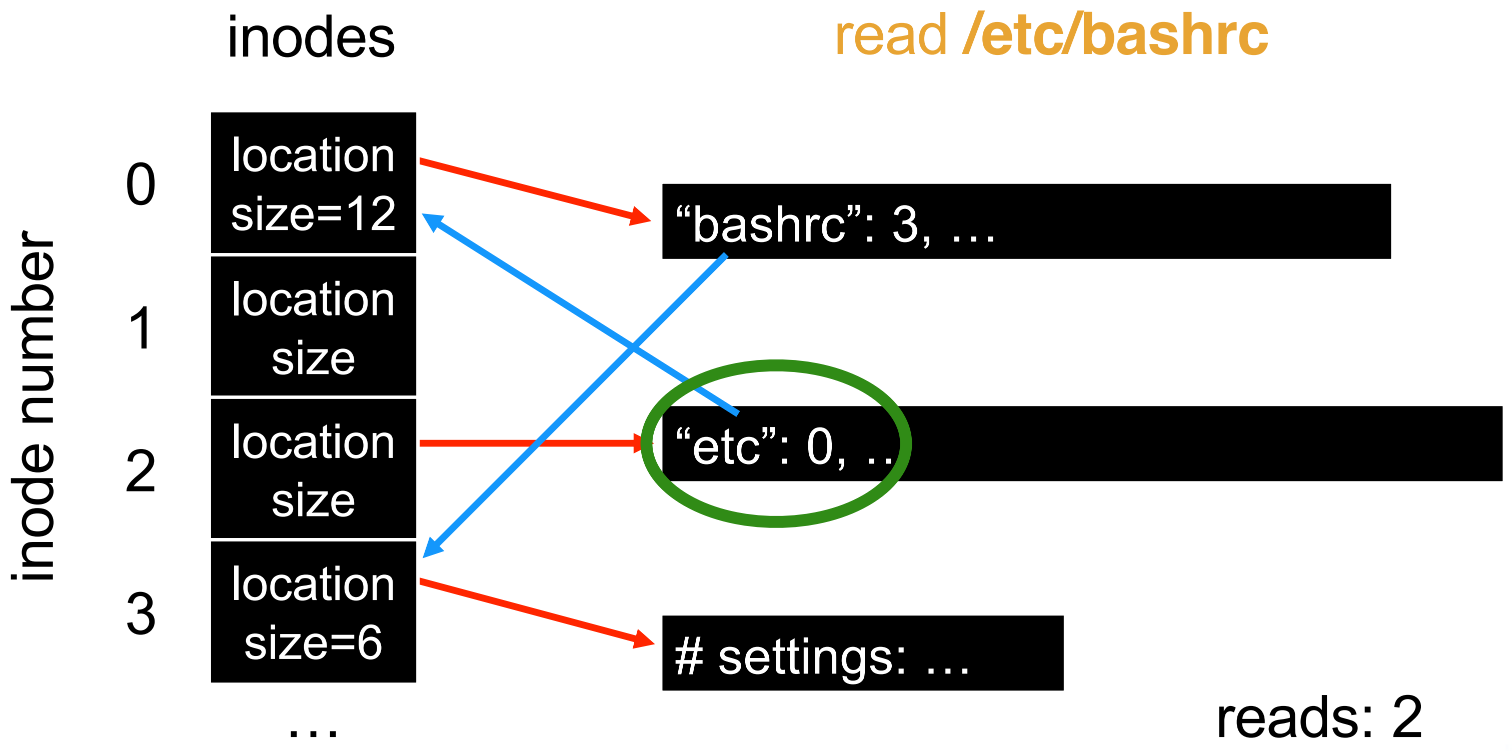
Valid directory :
/
/foo
/bar
/bar/bar
/bar/foo/

Sub-directories


inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"bashrc": 3, …
"etc": 0, …
settings: …
reads: 0

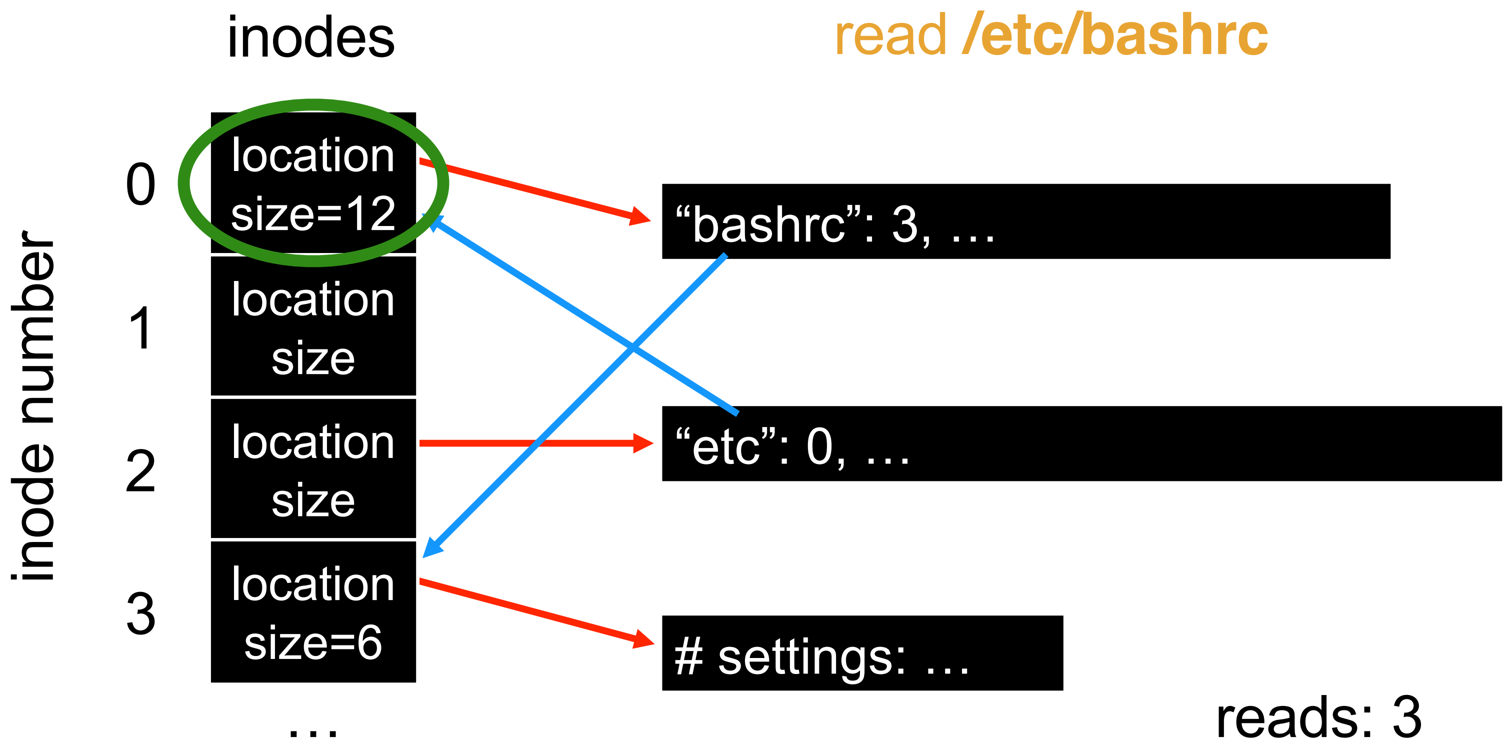

inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"bashrc": 3, …
"etc": 0, …
settings: …
reads: 1

read **/etc/bashrc**



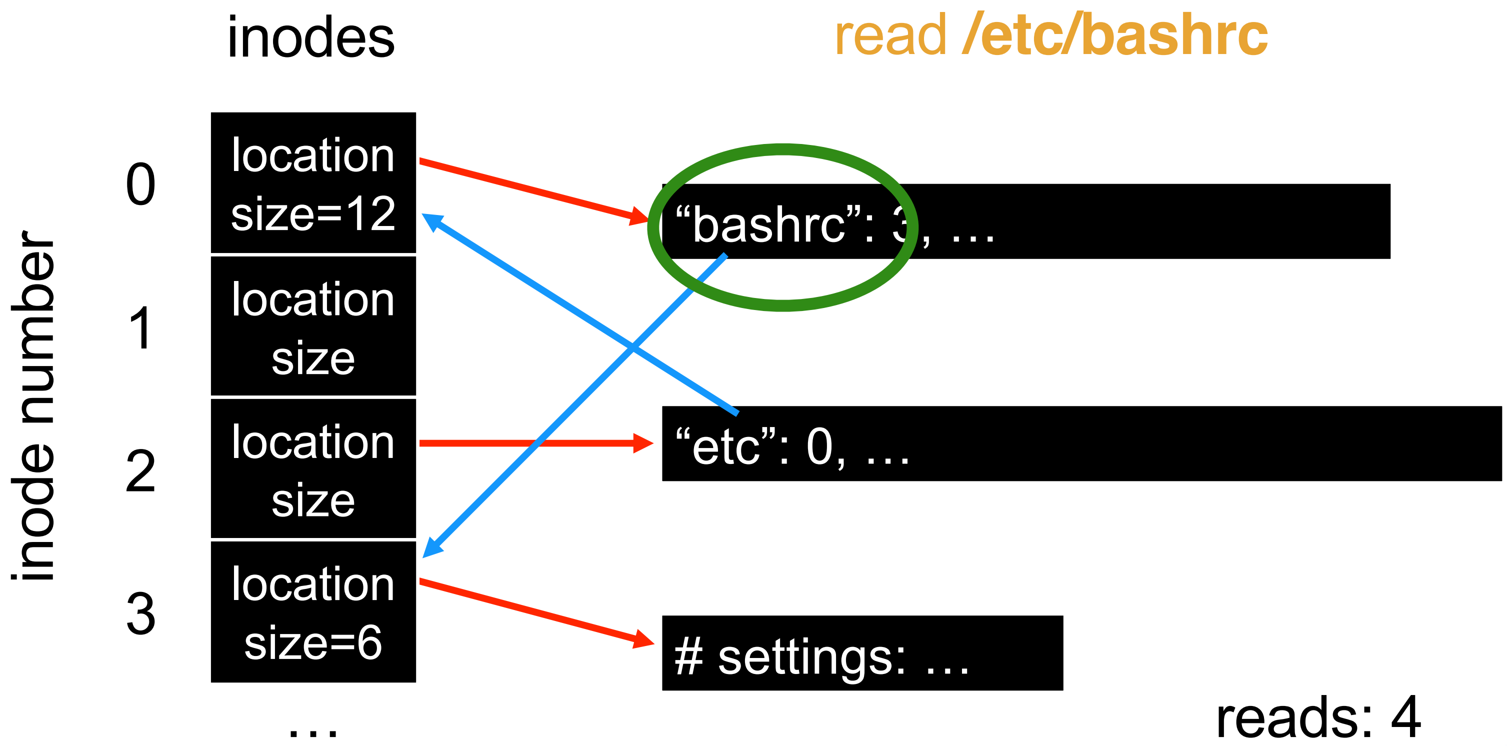
reads: 2

read /etc/bashrc



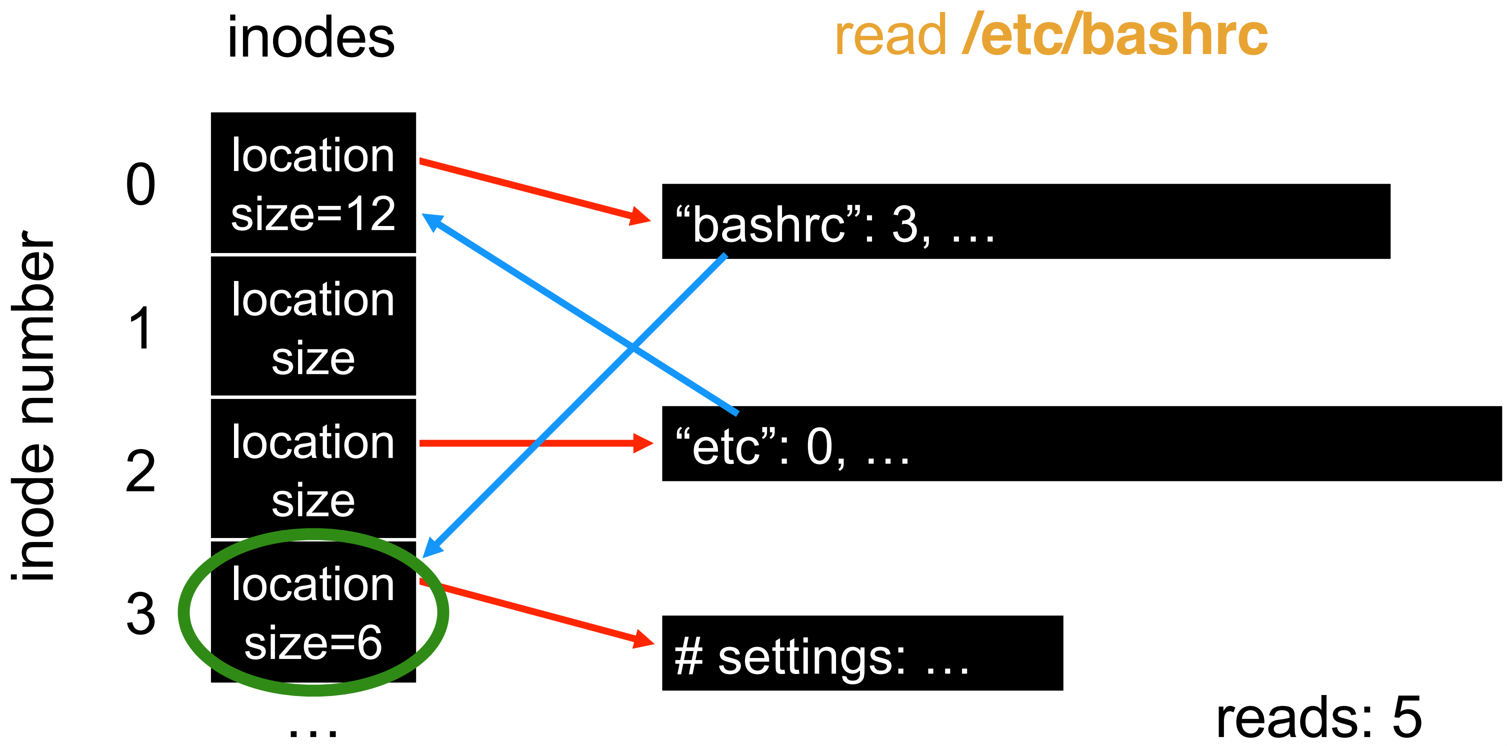
reads: 3

read **/etc/bashrc**

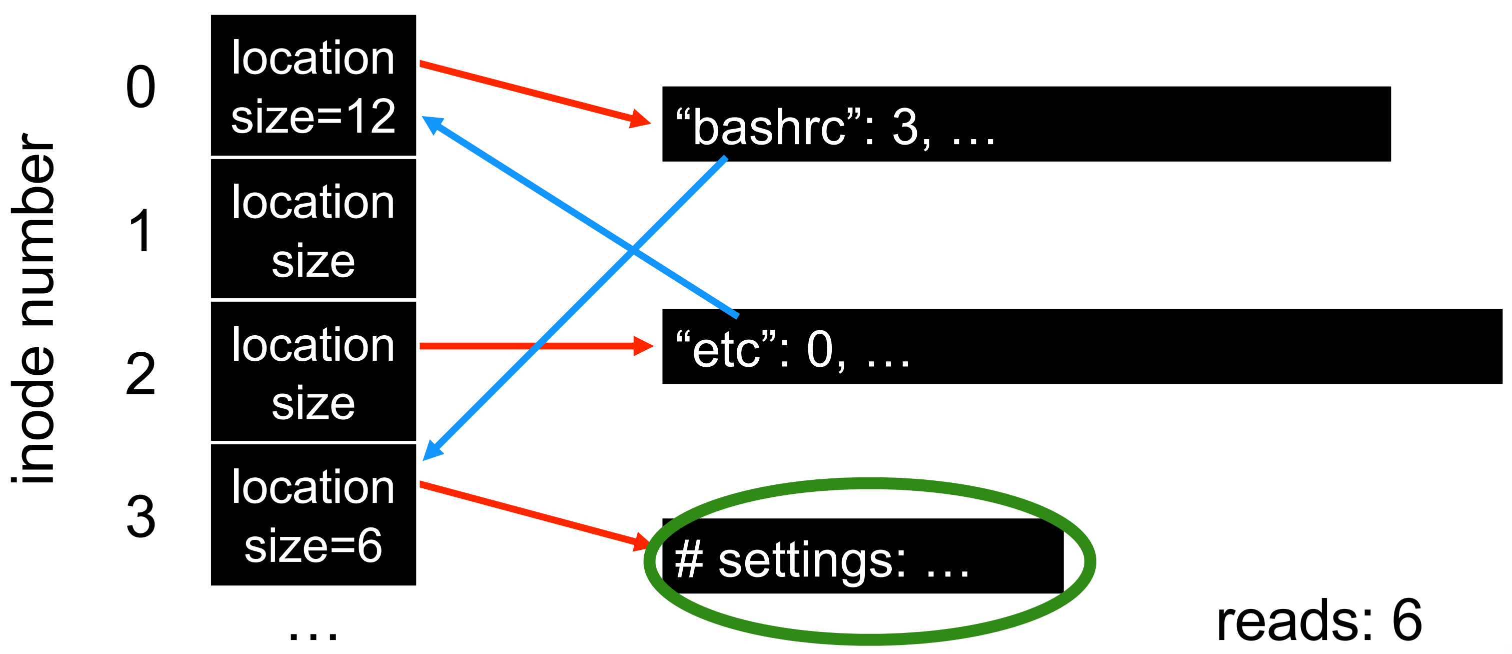


reads: 4

read **/etc/bashrc**



read /etc/bashrc



Reads for getting final inode called "traversal"

Read root dir (inode and data);
read etc dir (inode and data);
read bashrc file (indode and data)

Special Directory Entries

```
$ ls -la
total 728
drwxr-xr-x  34 olafm  staff    1156 Oct 19 11:41 .
drwxr-xr-x+ 59 olafm  staff    2006 Oct  8 15:49 ..
-rw-r--r--@  1 olafm  staff    6148 Oct 19 11:42 .DS_Store
-rw-r--r--   1 olafm  staff     553 Oct  2 14:29 asdf.txt
-rw-r--r--   1 olafm  staff     553 Oct  2 14:05 asdf.txt~
drwxr-xr-x   4 olafm  staff     136 Jun 18 15:37 backup
…
```

File Descriptor (fd)

Idea:

- Do expensive traversal once (open file)
- Store inode in descriptor object (kept in memory).
- Do reads/writes via descriptor, which tracks offset

Each process:

- File-descriptor table contains pointers to open file descriptors

Integers used for file I/O are indexes into this table

stdin: 0, stdout: 1, stderr: 2

File API

```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```

advantages:
- string names
- hierarchical
- traverse once
- different offsets precisely defined

File Names

Three types of names:

- inode

- path

- file descriptor

Code Snippet

```
prompt> strace cat file.txt
…
open("file.txt", O_RDONLY|O_LARGEFILE)      => 3
read(3, "hello\n", 4096)                    => 6 // file descriptor 1: stdout
write(1, "hello\n", 6)                      => 6
hello
read(3, "", 4096)                           => 0 // 0: no bytes left to read
close(3)
…
prompt>
```

open(file descriptor, flags)

- Return file descriptor (3 in example)
- File descriptor 0, 1, 2, is for standard input/output/error.

read(file descriptor, buffer pointer, the size of the buffer)

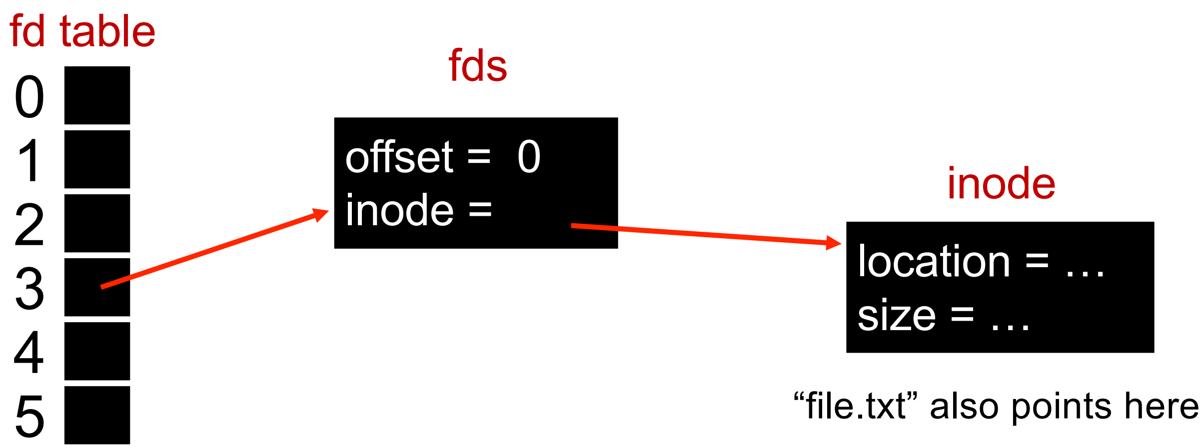
- Return the number of bytes it read

write(file descriptor, buffer pointer, the size of the buffer)

- Return the number of bytes it writes

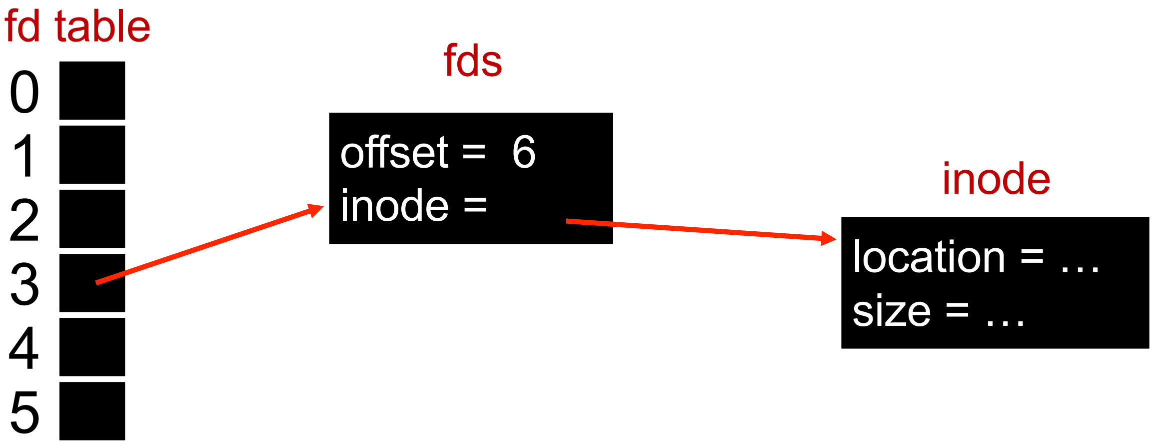
THE UNIVERSITY of ADELAIDE

Code Snippet



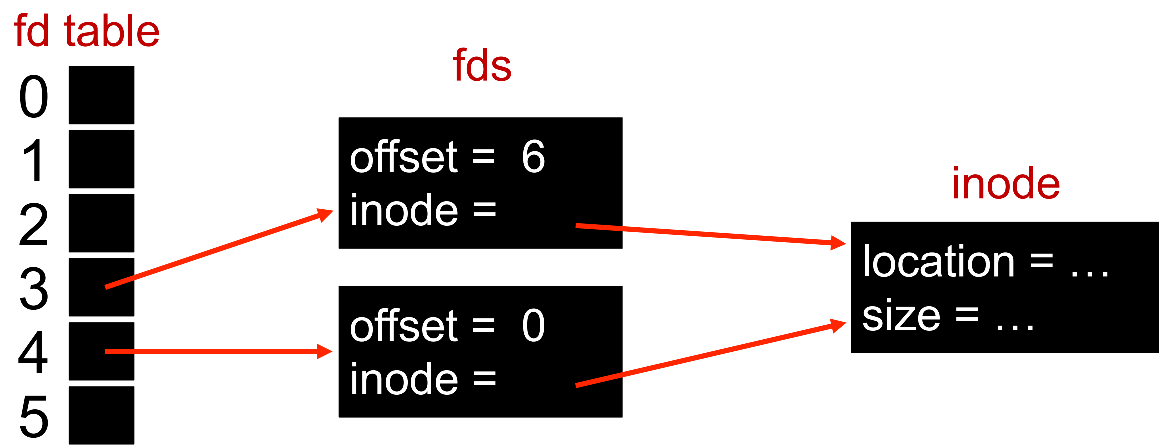

```
int fd1 = open(“file.txt”); // returns 3
```

Code Snippet




```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 6);
```

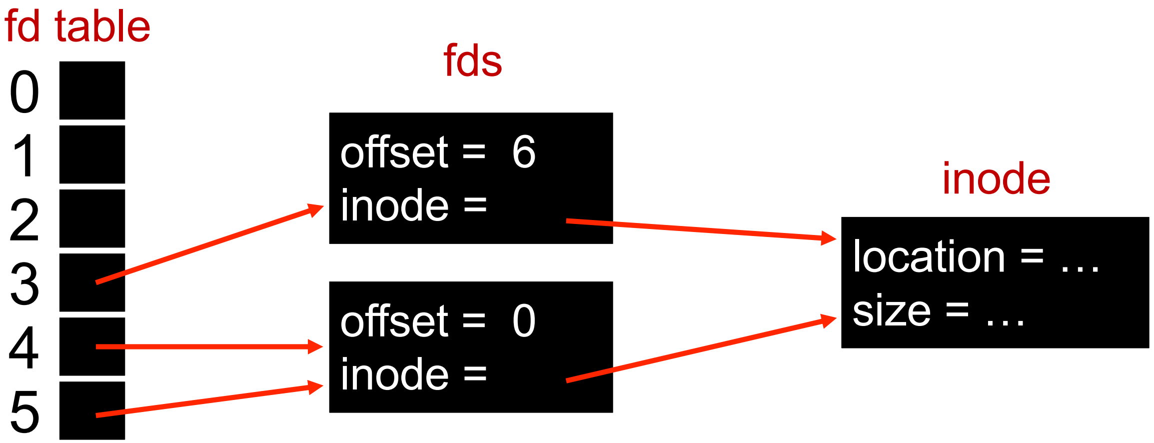
Code Snippet




```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
```

THE UNIVERSITY of ADELAIDE

Code Snippet




```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
int fd2 = open(“file.txt”); // returns 4
int fd3 = dup(fd2);         // returns 5
```

Deleting Files

Inode (and associated file) is garbage collected when there are no references (from paths or fds)

Paths are deleted when: `unlink()` is called

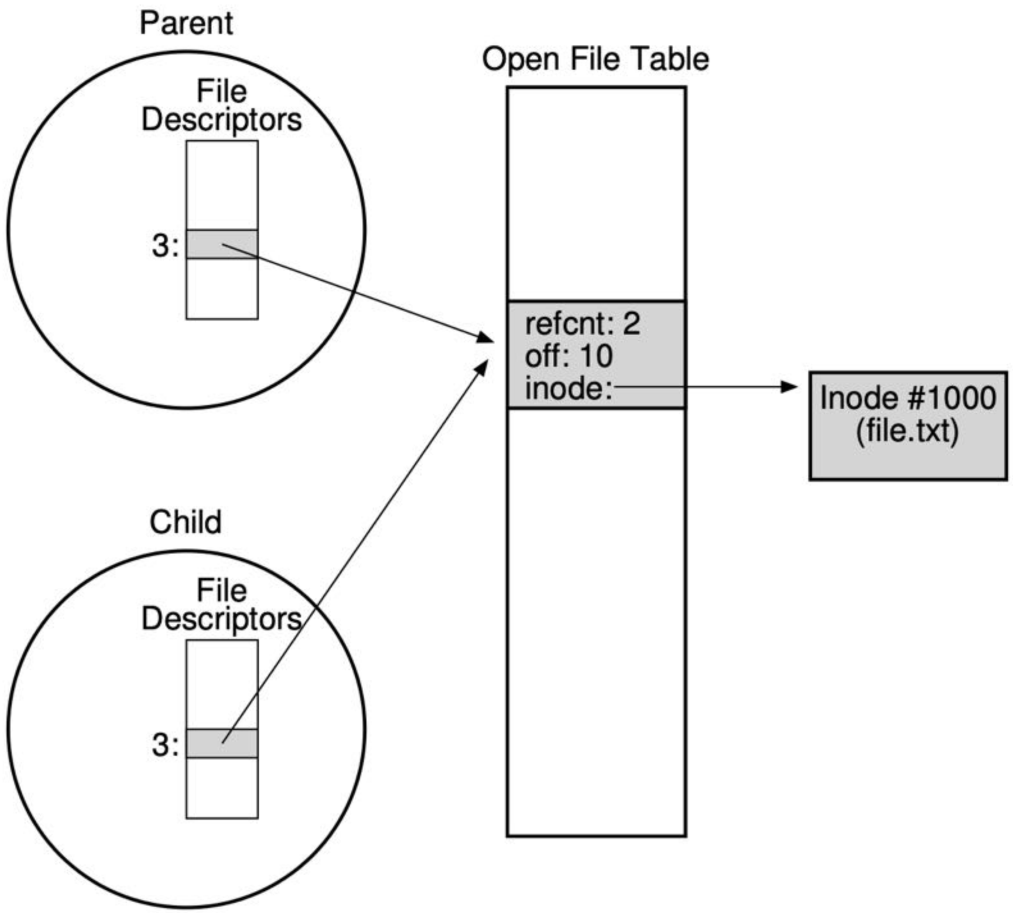
FDs are deleted when: `close()` or the process quits

Reading And Writing, But Not Sequentially

- An open file has a current offset.
  - Determine where the next read (or write) will begin reading from (or writing) to within the file.

- Update the current offset
  - Implicitly: A read or write of N bytes takes place; N is added to the current offset.
  - Explicitly: lseek()

Fork()


Parent
File
Descriptors
3:
Open File Table
refcnt: 2
off: 10
inode:
Inode #1000
(file.txt)
Child
File
Descriptors
3:

Hard links

`link(old pathname, new one)`

Link a new file name to an old one

Create another way to refer to the same file

The command-line link program : **`ln`**

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2 // create a hard link, link file to file2
prompt> cat file2
hello
```

Hard links

`link(old pathname, new one)`

Link a new file name to an old one

Create another way to refer to the same file

The command-line link program : **`ln`**

```
prompt> rm file    // removed 'file'
prompt> cat file2 // Still access the file
hello
```

THE UNIVERSITY of ADELAIDE

Symbolic links

Symbolic link is more useful than *Hard link*.

- Hard Link cannot be created to a directory (because of cycles).
- Hard Link cannot be created to a file to another partition (because inode numbers are only unique within a file system).

Create a symbolic link: `ln -s`

```
prompt> echo hello > file
prompt> ln -s file file2 /* option -s : create a symbolic link, */
prompt> cat file2
hello
prompt> rm file           // remove the original file
prompt> cat file2
cat: file2: No such file or directory
```

Unix UID & GID

Principals: User and group IDs

Subjects: Processes

Each process is associated with a uid and several gids

```
uid=503(a1234567)
gid=20(staff)
groups=20(staff),101(access_bpf),12(everyone),…
```

Unix File Permissions

File permissions:

```
-rwxr-xr-x  1 root  wheel 187120 14 Aug 13:30 /bin/ls
```

Owner Group

Unix File Permissions

File permissions:

```
-rwxr-xr-x  1 root  wheel 187120 14 Aug 13:30 /bin/ls
```

Owner   Group   Others

+s Commonly noted as **SUID**, the special permission for the user access level has a single function: A file with **SUID** always executes as the user who owns the file, regardless of the user passing the command. If the file owner doesn't have execute permissions, then use an uppercase **S** here.

Role-based access control

Extension of grouping – a *role* is another type of principals

Subjects assigned to roles

At each time a subject has one active role

Access rights depend on the active role

Can be implemented using any of the mechanisms mentioned earlier.

Example:

selinux uses special ACLs for role-based file access

Many File Systems

Users often want to use many file systems

For example:

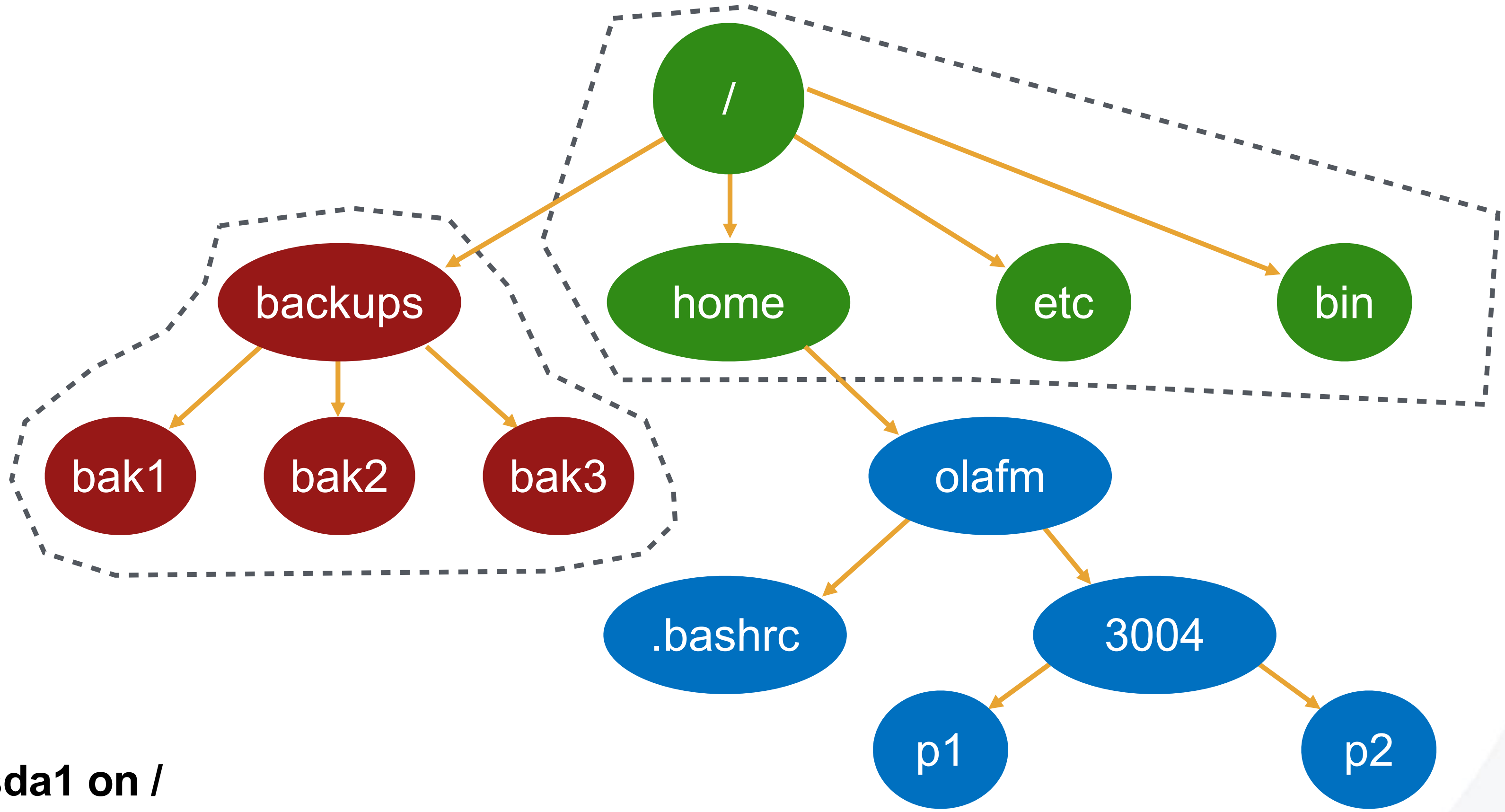
- main disk
- backup disk
- AFS
- thumb drives

What is the most elegant way to support this?

Many File Systems:

Idea: stitch all the file systems together into a super file system!

```
prompt> mount
/dev/sda1 on / type ext4 (rw)
/dev/sdb1 on /backups type ext4 (rw)
AFS on /home type afs (rw)
```



/dev/sda1 on /

/dev/sdb1 on /backups

AFS on /home

Communicating Requirements: fsync

File system keeps newly written data in memory for a while

Write buffering improves performance

But what if the system crashes before the buffers are flushed?

If the application cares:

`fsync(int fd)` forces buffers to flush to disk and (usually) tells disk to flush its write cache too

rename

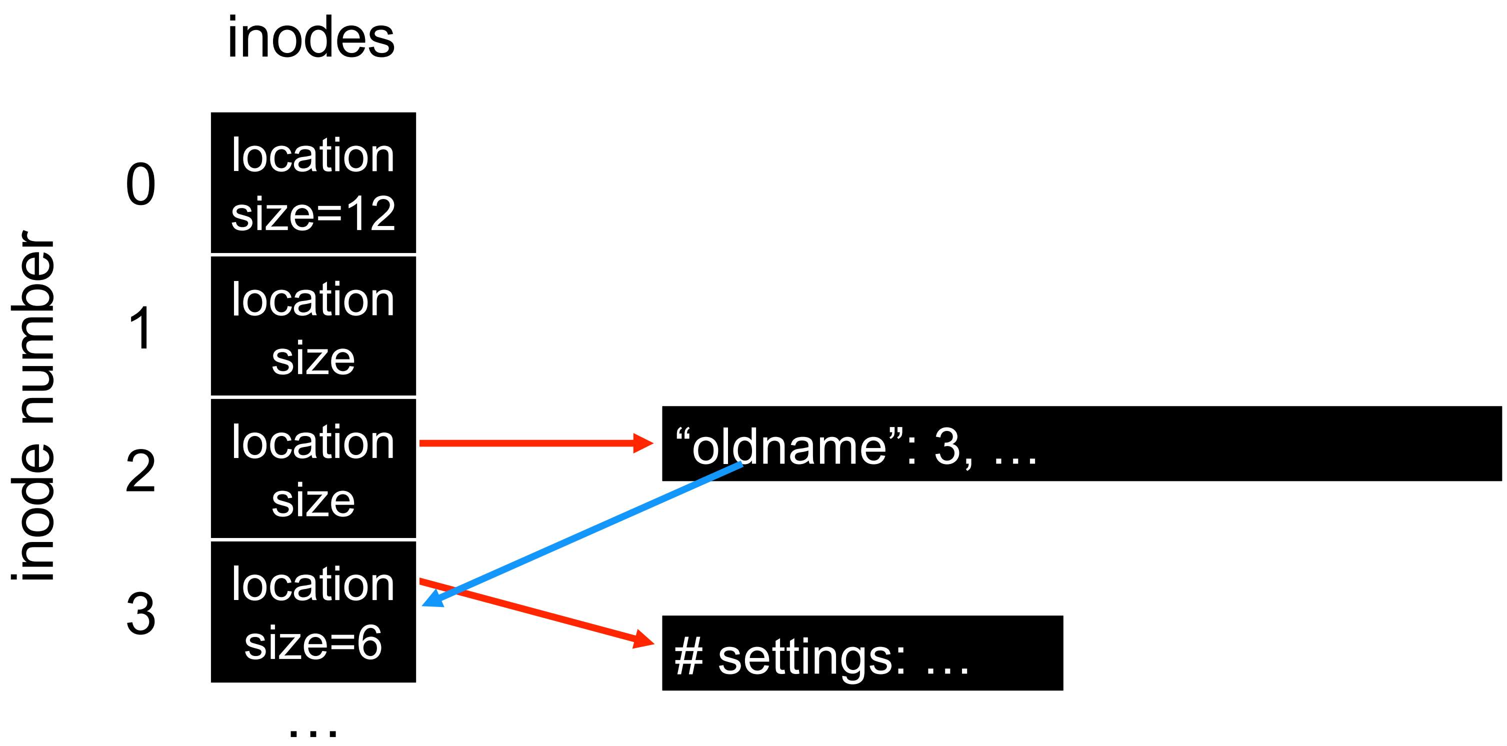
rename(char *old, char *new):

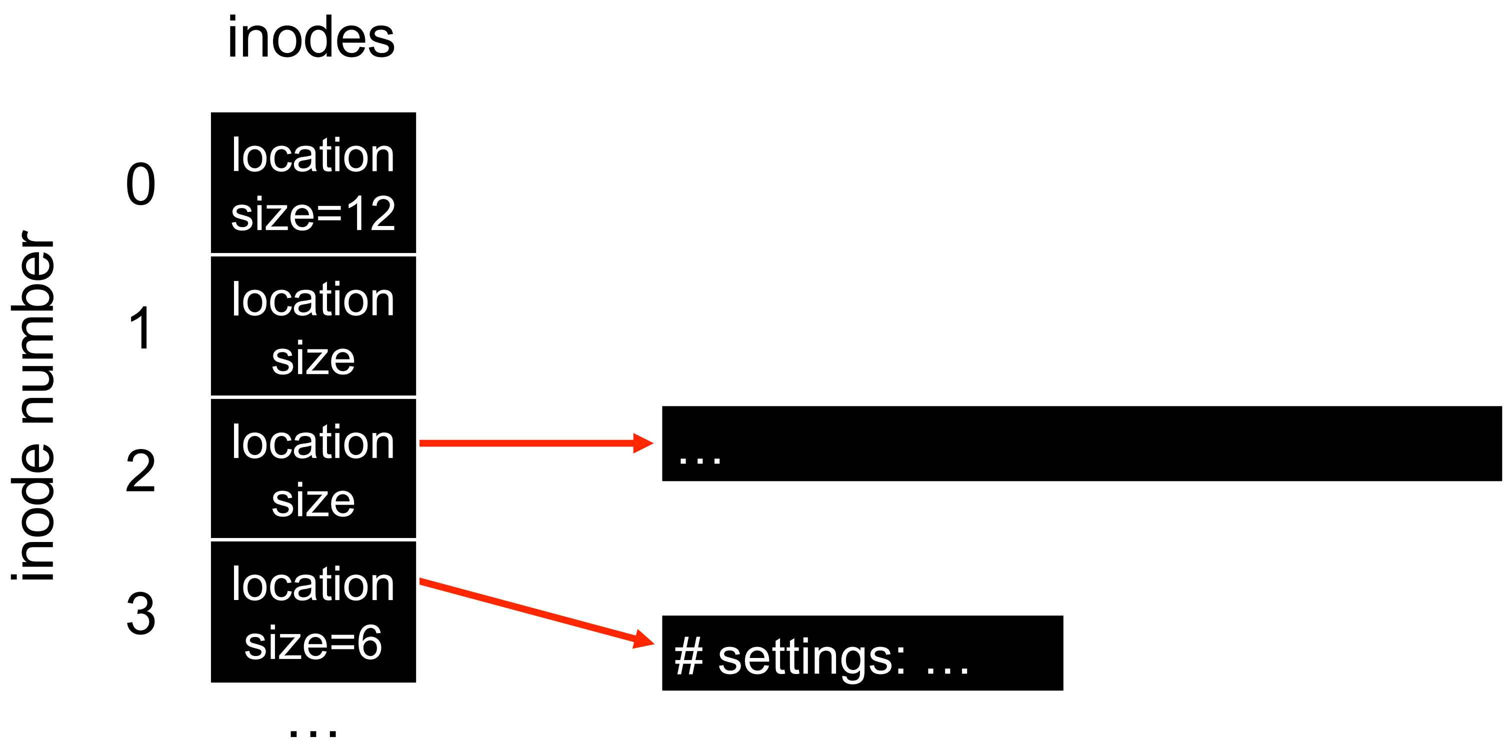
- deletes an old link to a file

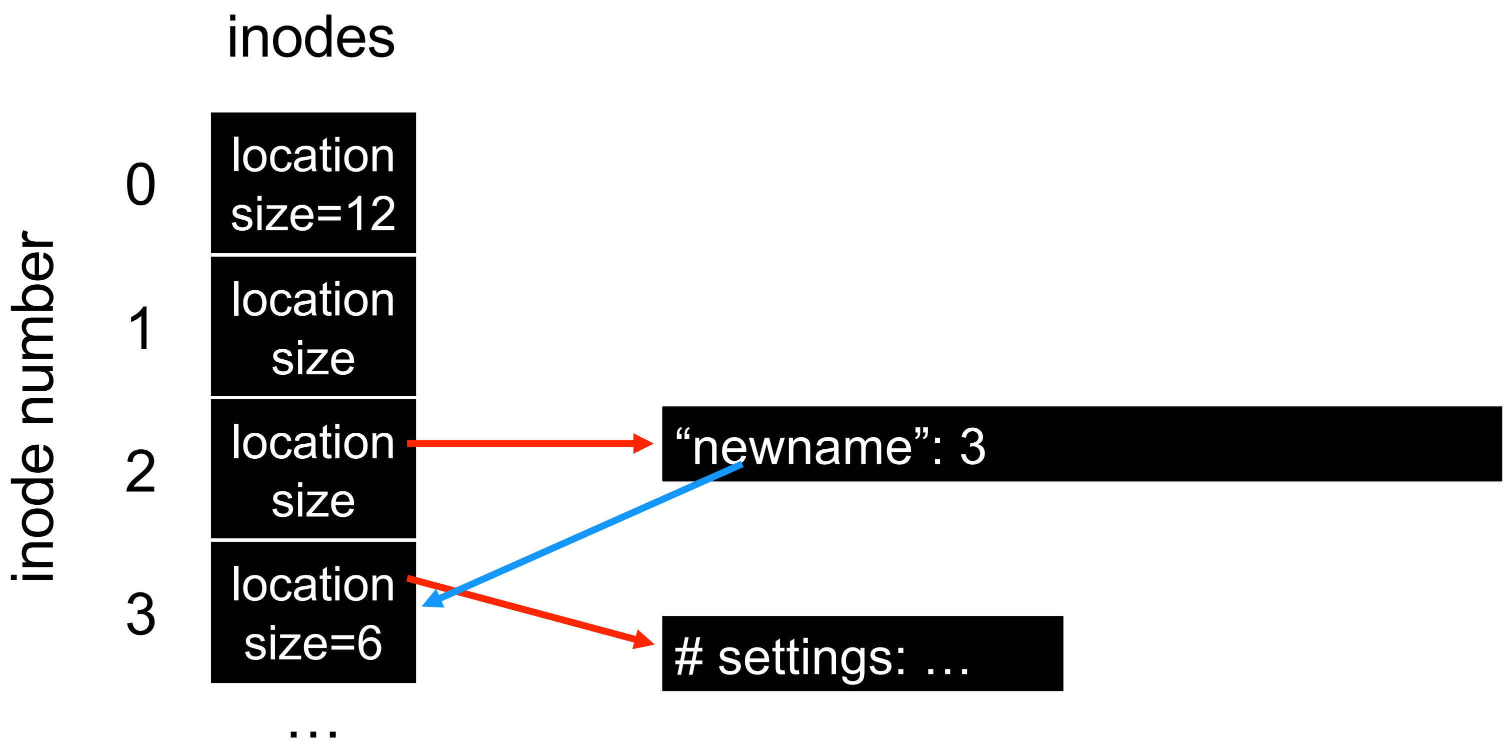
- creates a new link to a file

Just changes name of file, does not move data

Even when renaming to new directory


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“oldname”: 3, …
settings: …


inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
…
settings: …


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"newname": 3
settings: …

rename

rename(char *old, char *new):

- deletes an old link to a file
- creates a new link to a file

What if we crash?

FS does extra work to guarantee atomicity; return to this issue later…

Summary

Using multiple types of name provides

- convenience
- efficiency

Mount and link features provide flexibility.

Special calls (fsync,…) let developers communicate special requirements to file system

COMP SCI 3004 Operating Systems

FS Implementation

make history.

THE UNIVERSITY of ADELAIDE

Implementation

1. On-disk structures

   - how does file system represent files, directories?

2. Access methods

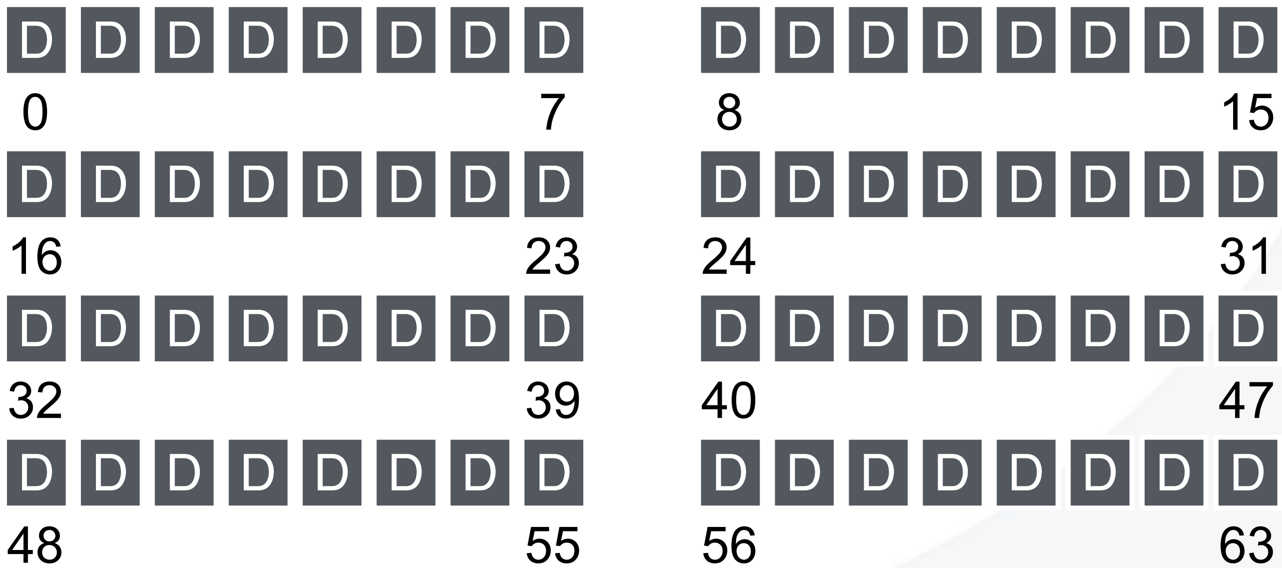
   - what steps must reads/writes take?

Part 1: Disk Structures

Persistent Store

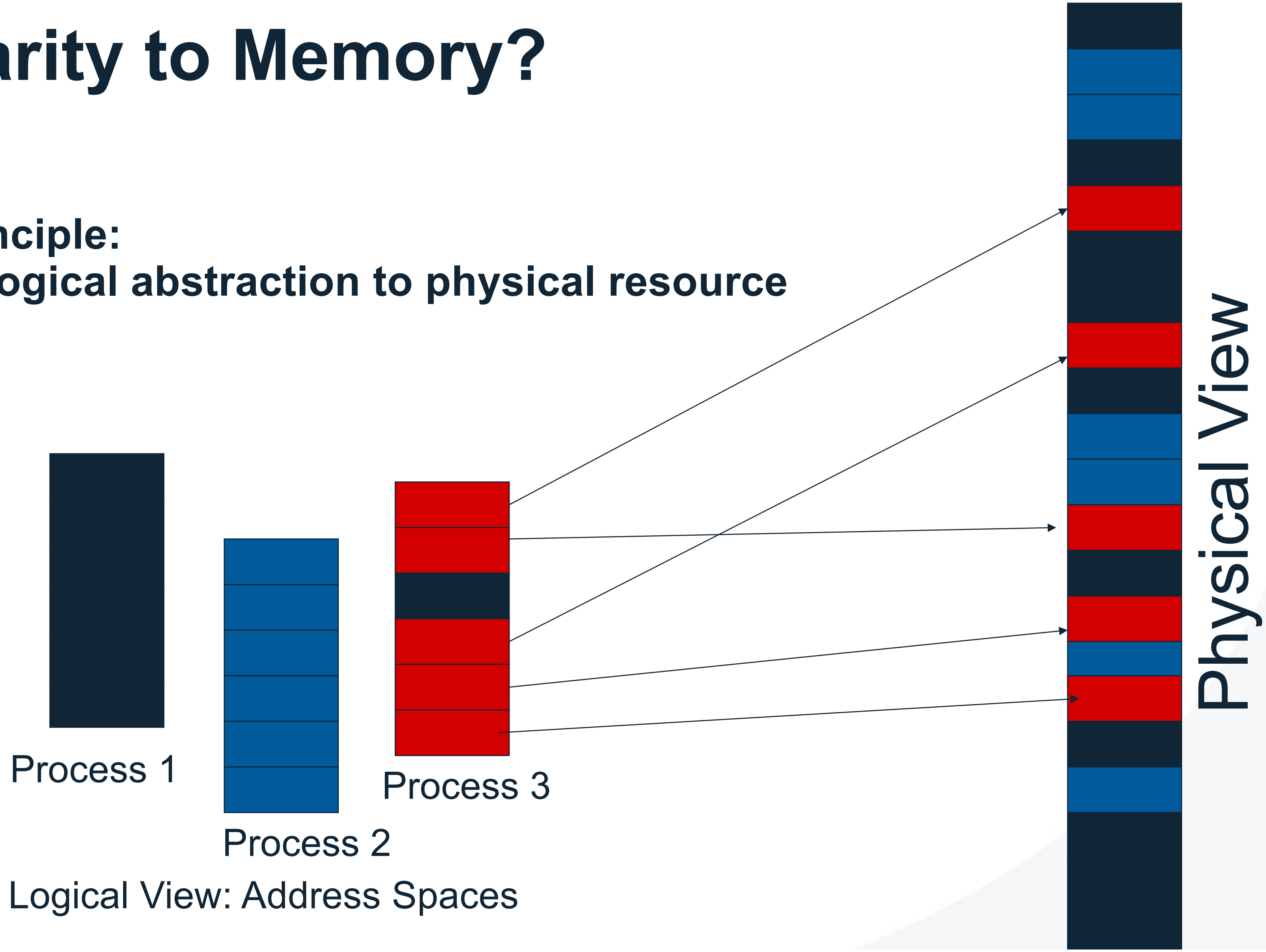
Given: large array of blocks on disk

Want: some structure to map files to disk blocks



Similarity to Memory?

Same principle:
map logical abstraction to physical resource



Outlook: Allocation Strategies

Many different approaches

Contiguous

Extent-based

Linked

File-allocation Tables

Indexed

Multi-level Indexed

Questions

Amount of fragmentation (internal and external)

– freespace that can't be used

Ability to grow file over time?

Performance of sequential accesses (contiguous layout)?

Speed to find data blocks for random accesses?

Wasted space for meta-data overhead (everything that isn't data)?

- Meta-data must be stored persistently too!

Contiguous Allocation

Allocate each file to contiguous sectors on disk

Meta-data: Starting block and size of file

OS allocates by finding sufficient free space

- Must predict future size of file; Should space be reserved?

Example: IBM OS/360

			A	A	A		B	B	B	B	C	C	C					

Fragmentation (internal and external)?	- Horrible external frag (needs periodic compaction)
Ability to grow file over time?	- May not be able to without moving
Seek cost for sequential accesses?	+ Excellent performance
Speed to calculate random accesses?	+ Simple calculation
Wasted space for meta-data?	+ Little overhead for meta-data

Small # of Extent

Allocate multiple contiguous regions (extents) per file

Meta-data:

Small array (2-6) designating each extent
Each entry: starting block and size

		A	A	A		B	B	B	B	C	C	C		
D	D	A	A	A	D	B	B	B	B	C	C	C	B	B

Fragmentation (internal and external)? - Helps external fragmentation

Ability to grow file over time? - Can grow (until run out of extents)

Seek cost for sequential accesses? + Still good performance

Speed to calculate random accesses? + Still simple calculation

Wasted space for meta-data? + Still small overhead for meta-data

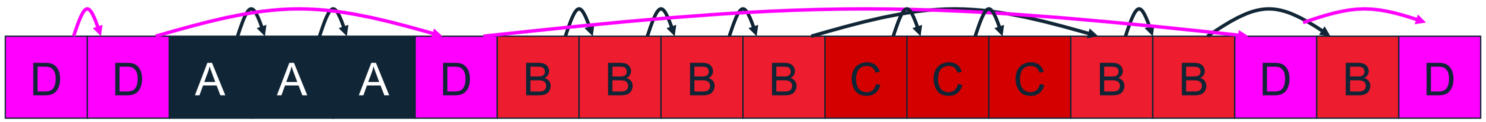
Linked Allocation

Allocate linked-list of fixed-sized blocks (multiple sectors)

Meta-data: Location of first block of file

Each block also contains pointer to next block

Examples: TOPS-10, Alto



Fragmentation (internal and external)? + No external frag (use any block); internal?

Ability to grow file over time? + Can grow easily

Seek cost for sequential accesses? +/- Depends on data layout

Speed to calculate random accesses? - Ridiculously poor

Wasted space for meta-data? - Waste pointer per block

Trade-off: Block size (does not need to equal sector size)

THE UNIVERSITY of ADELAIDE

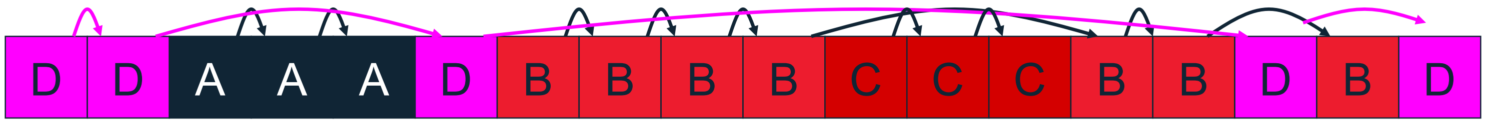
File-Allocation Table (FAT)

Variation of Linked allocation

Keep linked-list information for all files in on-disk FAT table

Meta-data: Location of first block of file

- And, FAT table itself



Draw corresponding FAT Table?

Comparison to Linked Allocation

- Same basic advantages and disadvantages
- Disadvantage: Read from two disk locations for every data read
- Optimization: Cache FAT in main memory
  – Advantage: Greatly improves random accesses
  – What portions should be cached? Scale with larger file systems?

Indexed Allocation

Allocate fixed-sized blocks for each file

Meta-data: Fixed-sized array of block pointers

Allocate space for ptrs at file creation time

D	D	A	A	A	D	B	B	B	B	C	C	C	B	B	D	B	D

Advantages

- No external fragmentation
- Files can be easily grown up to max file size
- Supports random access

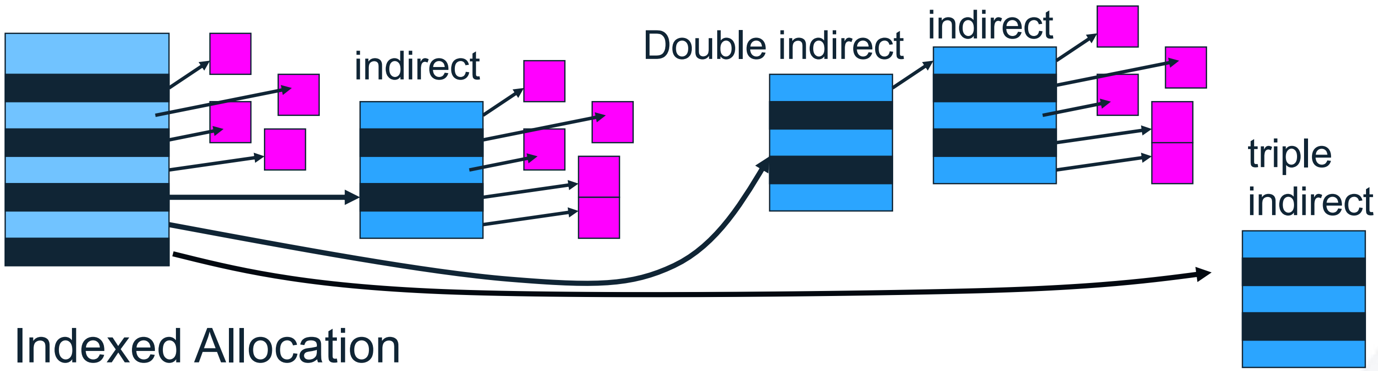
Disadvantages

- Large overhead for meta-data:
  – Wastes space for unneeded pointers (most files are small!)

Multi-Level Indexing

Variation of Indexed Allocation
Dynamically allocate hierarchy of pointers to blocks as needed
Meta-data: Small number of pointers allocated statically
- Additional pointers to blocks of pointers

Examples: UNIX FFS-based file systems, ext2, ext3



Comparison to Indexed Allocation
- Advantage: Does not waste space for unneeded pointers
  - Still fast access for small files
  - Can grow to what size??
- Disadvantage: Need to read indirect blocks of pointers to calculate addresses (extra disk read)
  - Keep indirect blocks cached in main memory

THE UNIVERSITY of ADELAIDE

Flexible # of Extents

Modern file systems:
Dynamic multiple contiguous regions (extents) per file

Organize extents into multi-level tree structure

- Each leaf node: starting block and contiguous size
- Minimizes meta-data overhead when have few extents
- Allows growth beyond fixed number of extents

Fragmentation (internal and external)?	+ Both reasonable
Ability to grow file over time?	+ Can grow
Seek cost for sequential accesses?	+ Still good performance
Speed to calculate random accesses?	+/- Some calculations depending on size
Wasted space for meta-data?	+ Relatively small overhead

THE UNIVERSITY of ADELAIDE

Assume Multi-Level Indexing

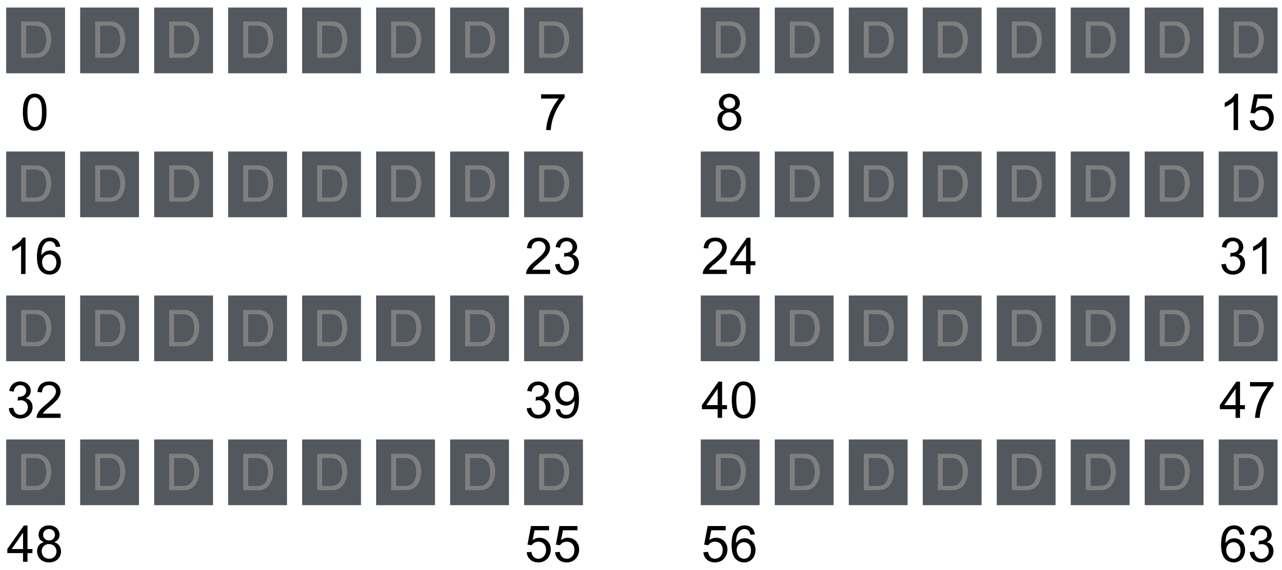
Simple approach

More complex file systems build from these basic data structures

On-Disk Structures

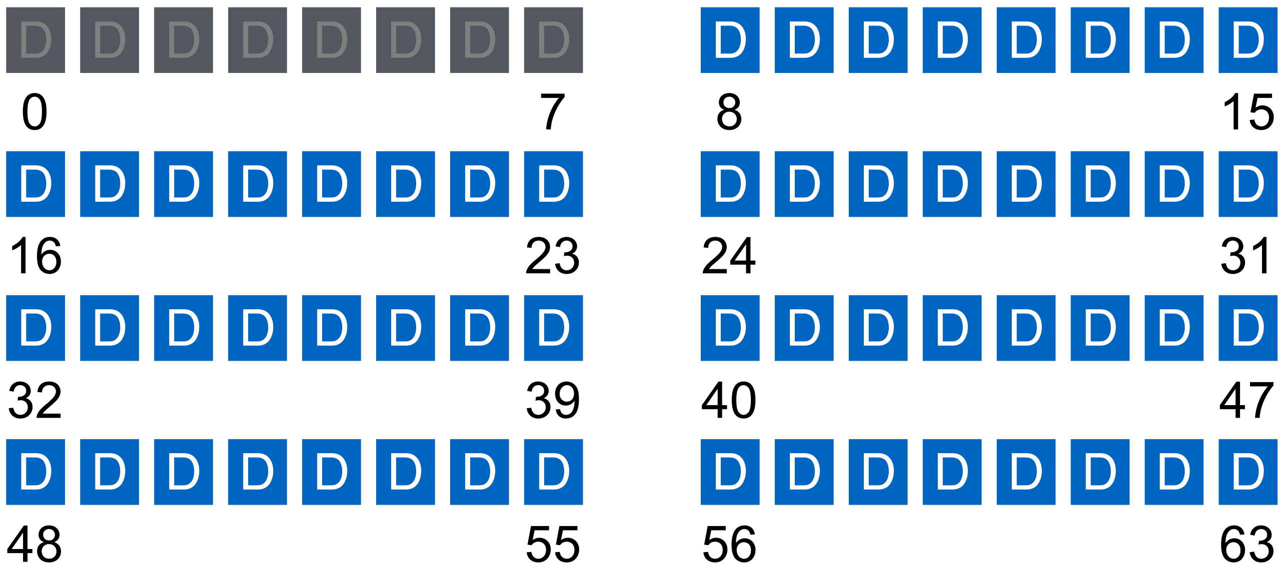
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS Structs: Empty Disk



Assume each block is 4KB

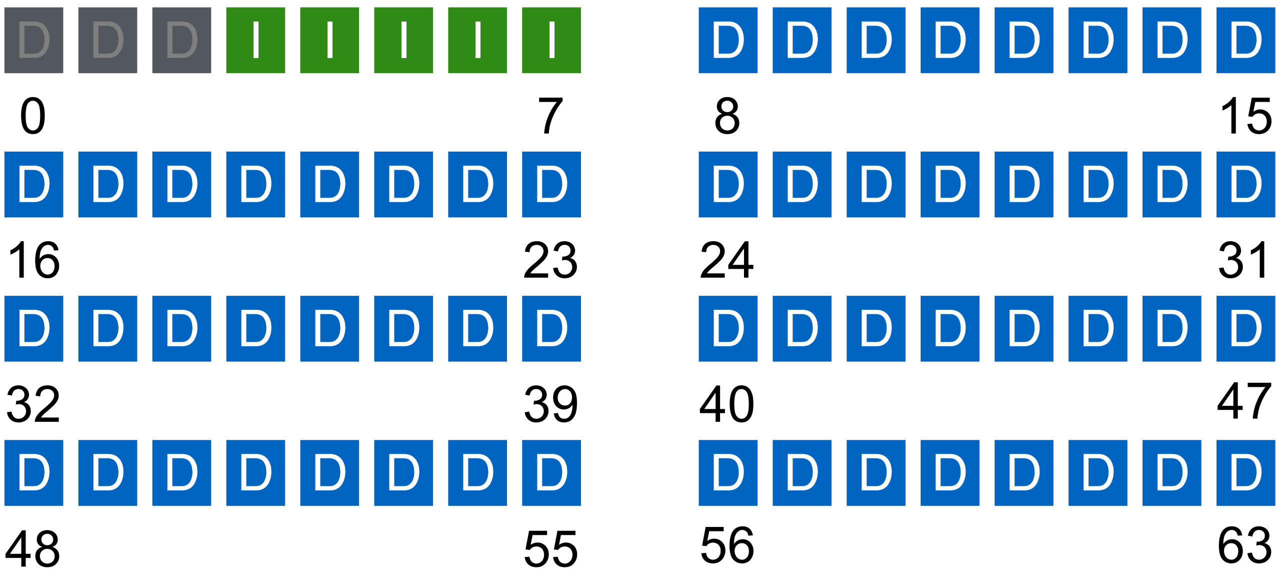
Data Blocks



Not actual layout: Simplified for illustration purposes.
Purpose: Relative number of each time of block

THE UNIVERSITY of ADELAIDE

Inodes


D D D I I I I I
D D D D D D D D
0
7
8
15
D D D D D D D D
D D D D D D D D
16
23
24
31
D D D D D D D D
D D D D D D D D
32
39
40
47
D D D D D D D D
D D D D D D D D
48
55
56
63

One Inode Block

Each inode is 256 bytes (depending on the FS, maybe 128 bytes)

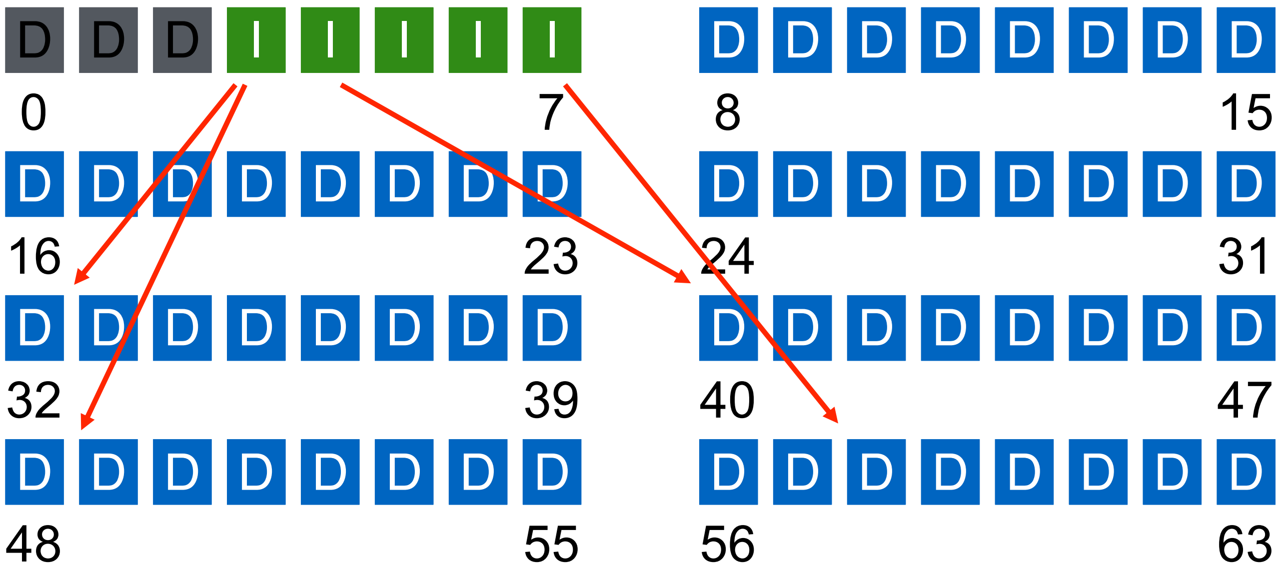
4KB disk block

16 inodes per inode block.

Inode

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

Inodes


D D D I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

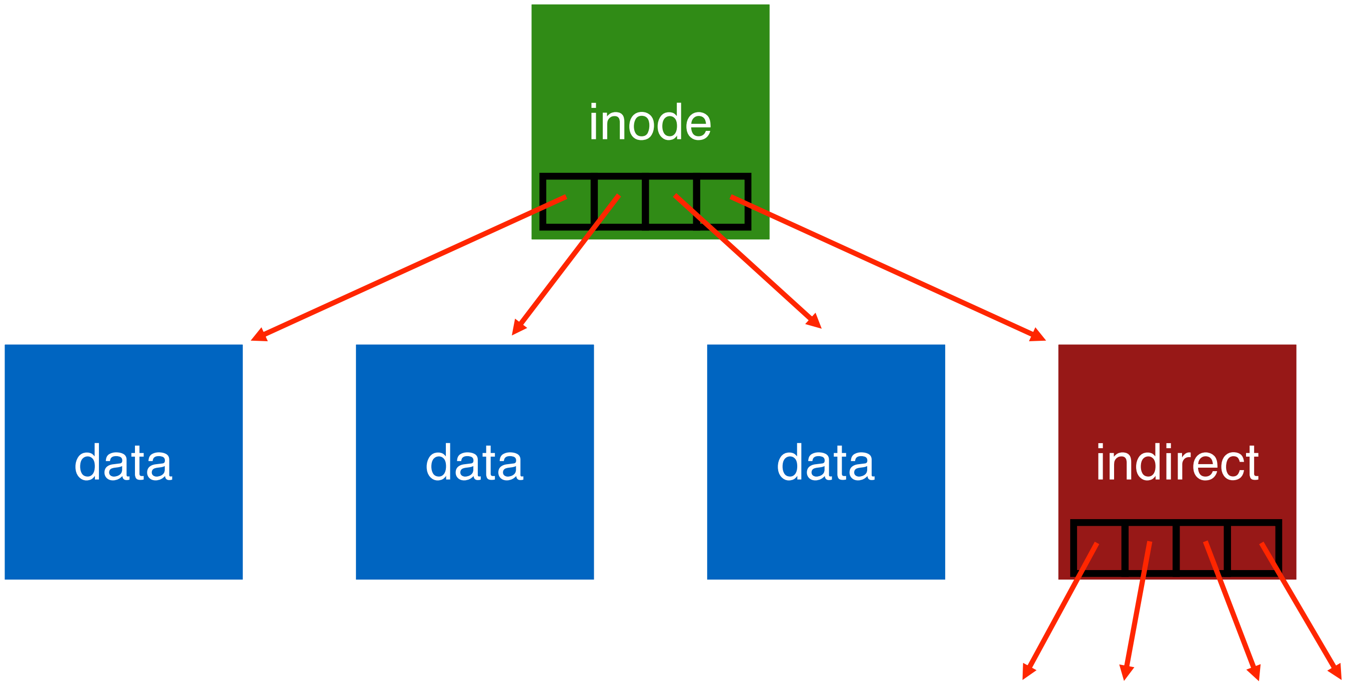
Assume single level (just pointers to data blocks)

What is max file size?
Assume 256-byte inodes (all can be used for pointers)
Assume 4-byte address

How to get larger files?

256 / 4 = 64
64 * 4K = 256 KB!

Recall



Directories

File systems vary

Common design:

Store directory entries in data blocks

    Large directories just use multiple data blocks

    Use bit in inode to distinguish directories from files

Various formats could be used

- lists
- b-trees

Simple Directory List Example

valid	name	inode
1	.	134
1	..	35
1	foo	80
1	bar	23

unlink("foo")

THE UNIVERSITY of ADELAIDE

Allocation

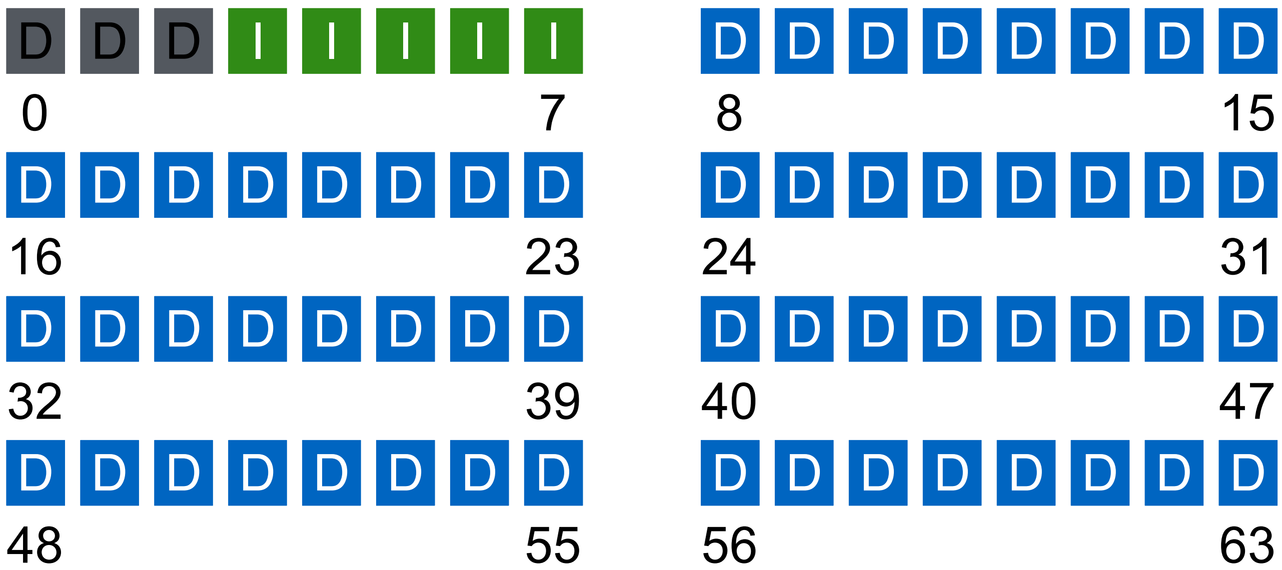
How do we find free data blocks or free inodes?

Free list

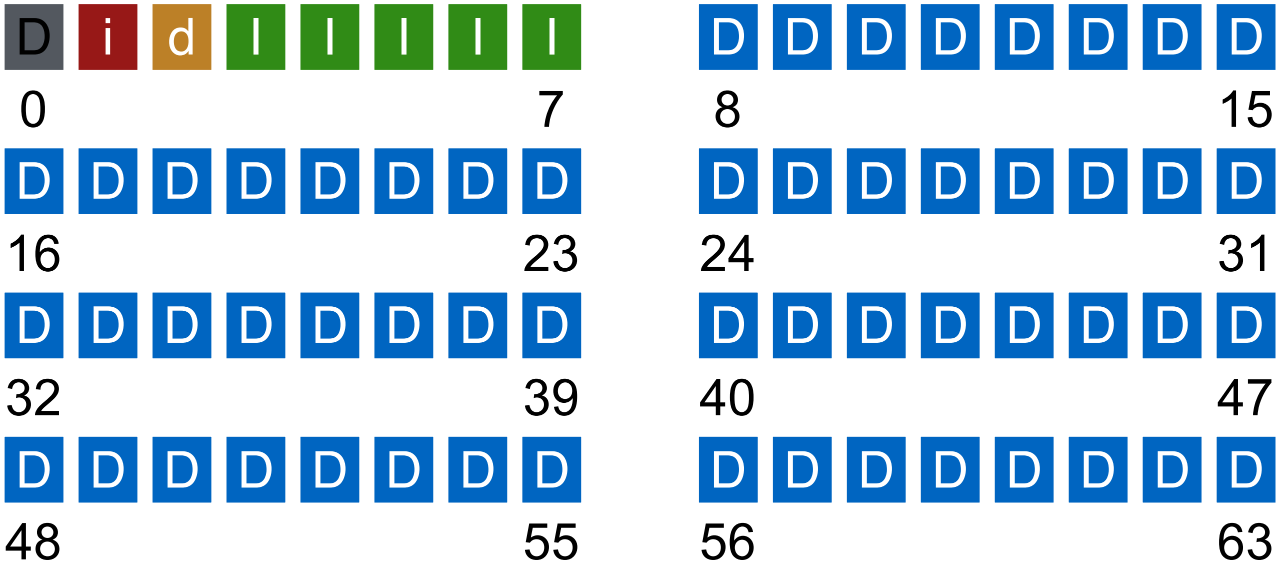
Bitmaps

Tradeoffs in next lecture…

Bitmaps


D D D I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

Bitmaps


D i d I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

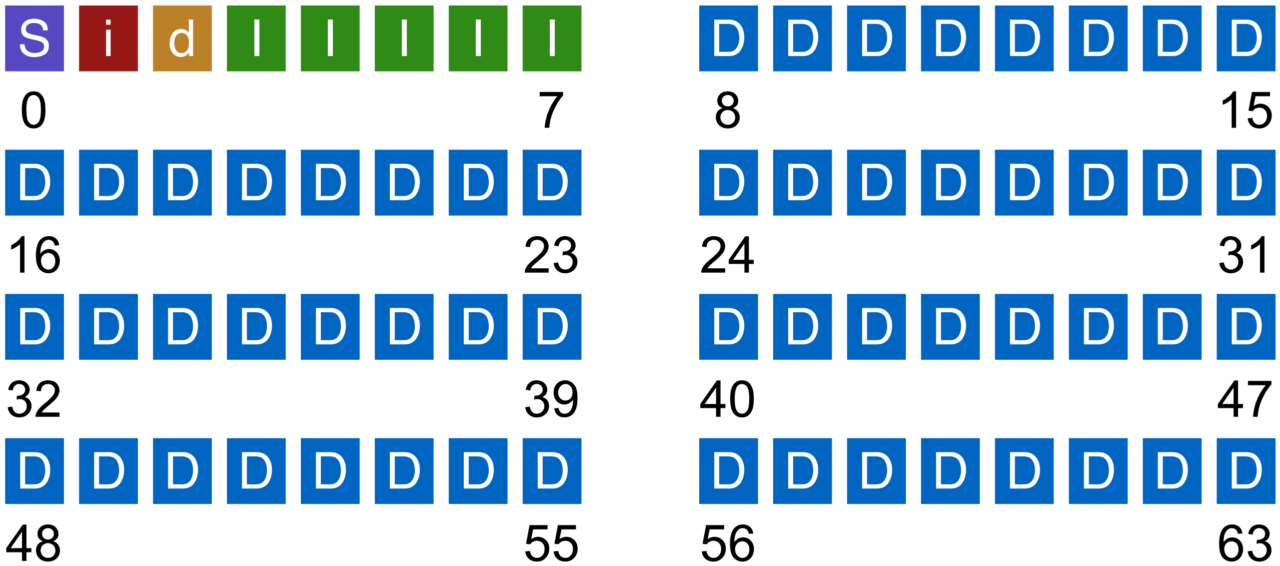
Superblock

Need to know basic FS configuration metadata, like:

- block size
- # of inodes

Store this in superblock

Super Block

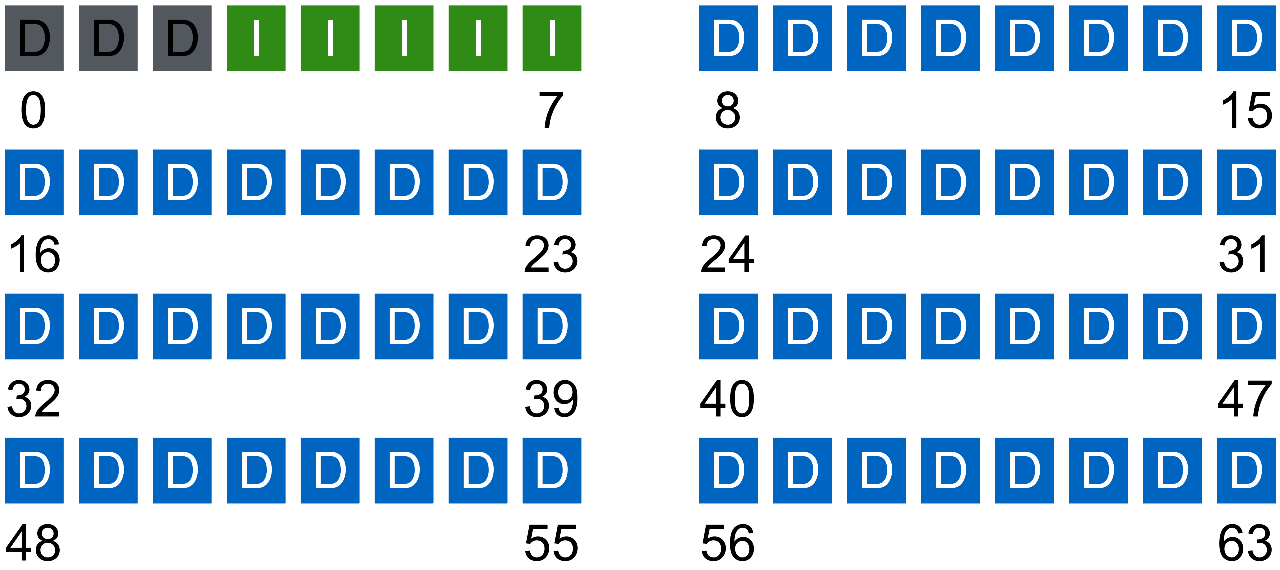

S i d I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

On-Disk Structures

Super Block | Inode Bitmap | Data Bitmap

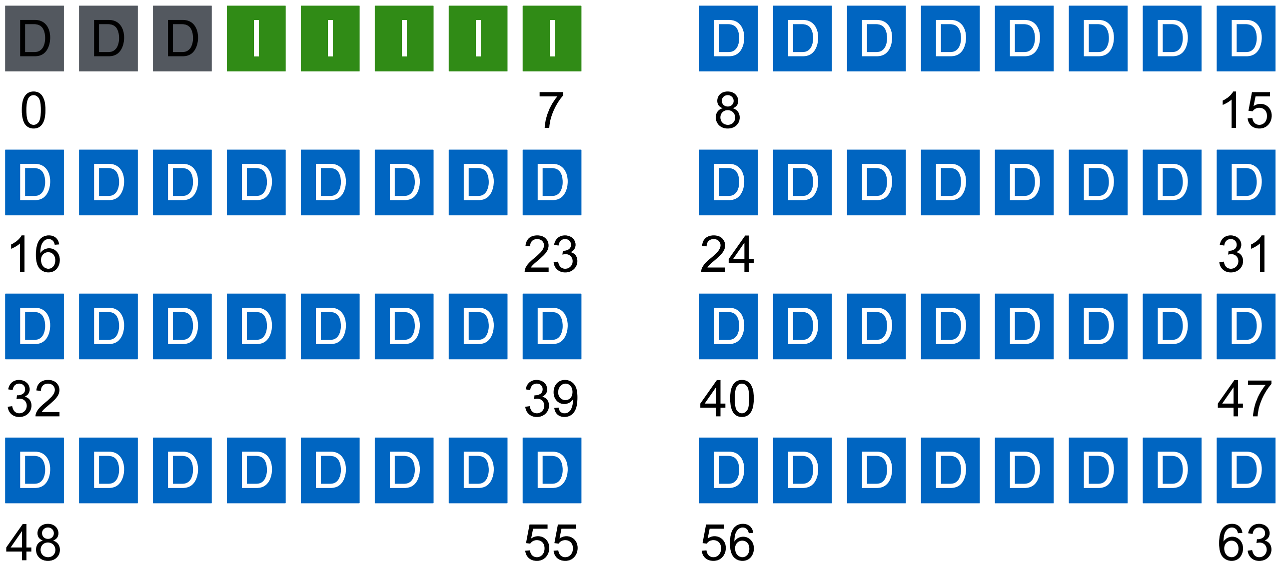
Inode Table | Data Block (directories, indirects)

Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 0?

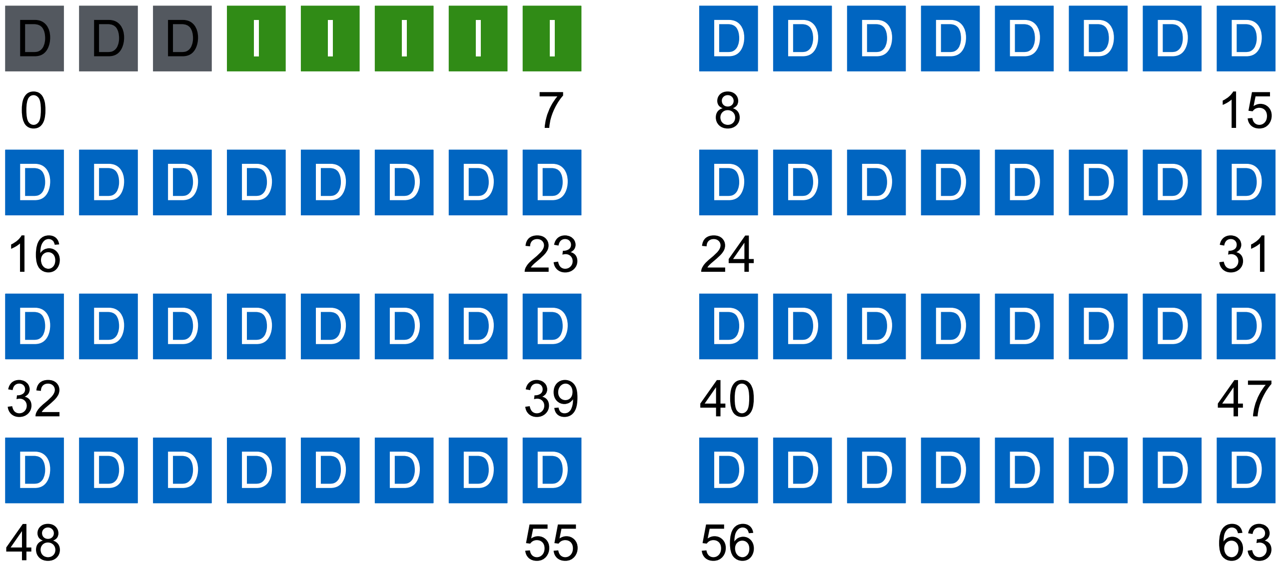


THE UNIVERSITY of ADELAIDE

Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 4?



Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 40?



Part 2 : Operations

- create file
- write
- open
- read
- close

[traverse]

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read

Verify that bar does not already exist

THE UNIVERSITY of ADELAIDE

create /foo/bar

[allocate inode]

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read					
	write					

create /foo/bar

[populate inode]

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read write					
				read write		

Why must **read** bar inode?
How to initialize inode?

[add bar to /foo]

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read					
	write					
				read		
				write		
			write			
						write

Update inode (e.g., size) and data for directory

open /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
		read					
					read		
			read				
						read	
				read			

write to /foo/bar (assume file exists and has been opened)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
							write
				write			

append to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

THE UNIVERSITY of ADELAIDE

append to /foo/bar (opened already)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			

[allocate block]

append to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							

[point to block]

append to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
				write			

[write to block]

append to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
				write			
							write

read /foo/bar – assume opened

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
							read
				write			

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

nothing to do on disk!

Efficiency

How can we avoid this excessive I/O for basic ops?

Cache for:

- reads

- write buffering

THE UNIVERSITY of ADELAIDE

Write Buffering

Why does procrastination help?

Overwrites, deletes, scheduling

Shared structs (e.g., bitmaps+dirs) often overwritten.

We decide: how much to buffer, how long to buffer…

- tradeoffs?

Summary/Future

We've described a very simple FS.

- basic on-disk structures
- the basic ops

Future questions:

- how to allocate efficiently to obtain good performance from disk?
- how to handle crashes?

THE UNIVERSITY of ADELAIDE