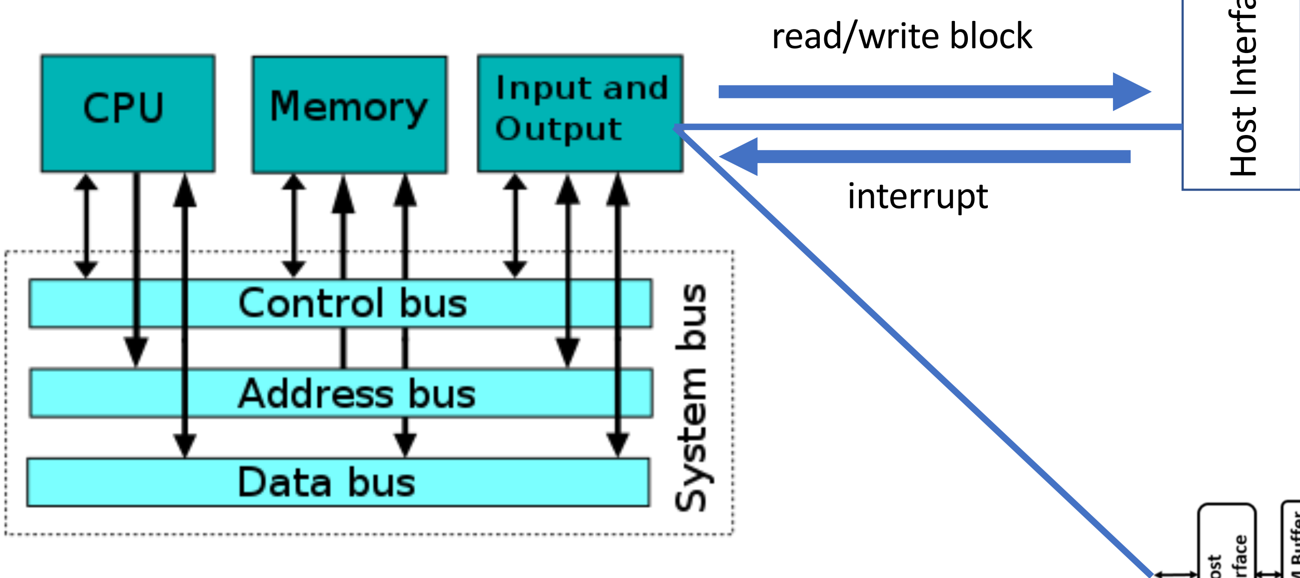
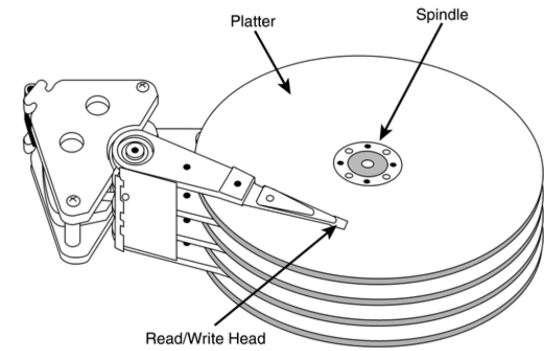
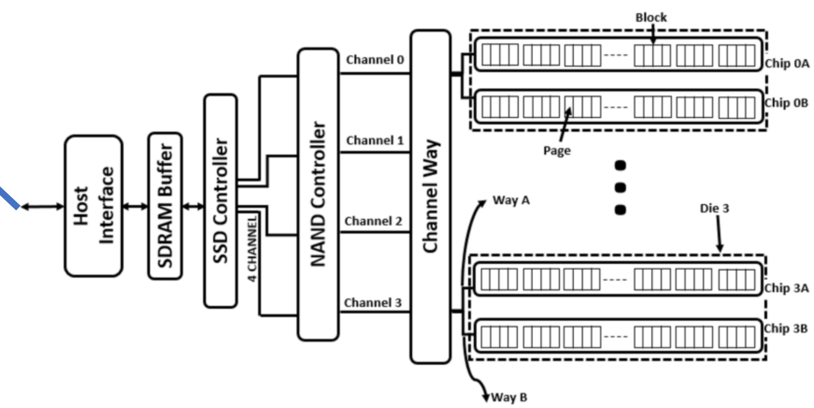
Persistent Storage Architecture

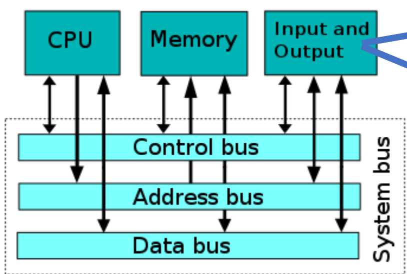


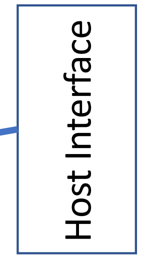


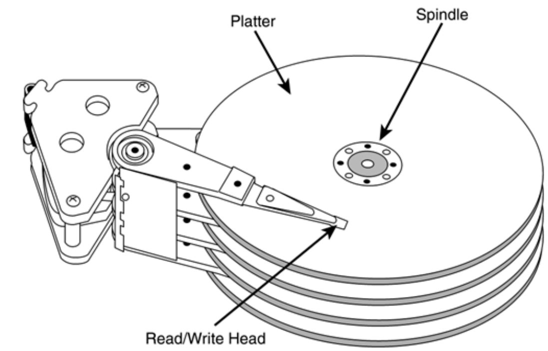


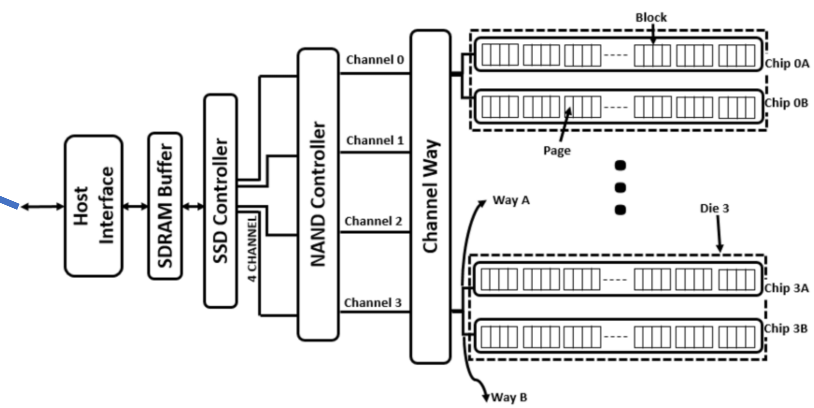
-- Memory contains code that is specific to each host interface (called a driver)
-- CPU execute driver code which directs the I/O subsystem to contact and interact with each device
-- synchronous I/O
        CPU requests
        Device Interrupts

Persistent Storage is Block Accessed









-- Devices transfer 4K data blocks
-- Each data block has a unique block number (called a block address)
-- Driver requests access (read or write) using block number
-- Host interface translates that to physical storage location
  -- platter, track, sector (disk)
  -- die, chip, location (ssd)
-- Host interface also implements read and write protocols (over write, load leveling, etc.)
-- **All persistent storage devices appear as "block transfer devices" that do synchronous I/O to the OS**

Linux Files

- Block storage devices are all accessed as linear lists of 4K blocks
- Files are not lists of 4K blocks
  - A file is a list of bytes
  - A file has no unallocated bytes within it
  - A file has a read/write pointer indicating the next byte to be written (it is not addressed by byte number)
  - A file has a name
- **How are files implemented using block storage devices?**

For example

- Consider this code:
  - What does it do?
  - What data (if any) is stored in persistent storage?

```
int fd;
double buffer[10000];
int i;


for(i=0; I < 10000; i++) {
   buffer[i] = drand48();
}

fd = open("/tmp/foo",O_RDWR | O_CREAT, 0600);
write(fd,buffer,sizeof(buffer));
close(fd);
```

What does the code do?

- The code allocates a buffer containing 80000 bytes (since each double is 8 bytes)
- It initializes each IEEE double precision value in the buffer to a random number between 0 and 1
- It opens a file called /tmp/foo and writes the 80000 bytes to persistent storage that can be accessed as a file by that name

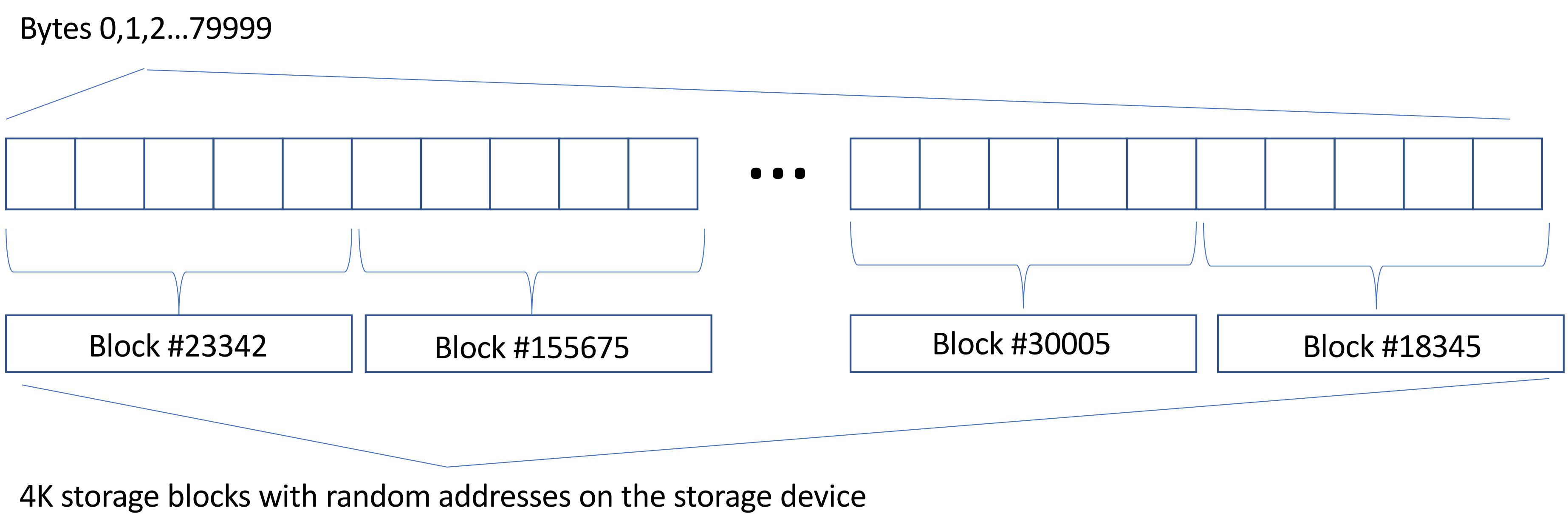
```
int fd;
double buffer[10000];
int i;


for(i=0; I < 10000; i++) {
   buffer[i] = drand48();
}

fd = open("/tmp/foo",O_RDWR | O_CREAT, 0600);
write(fd,buffer,sizeof(buffer));
close(fd);
```

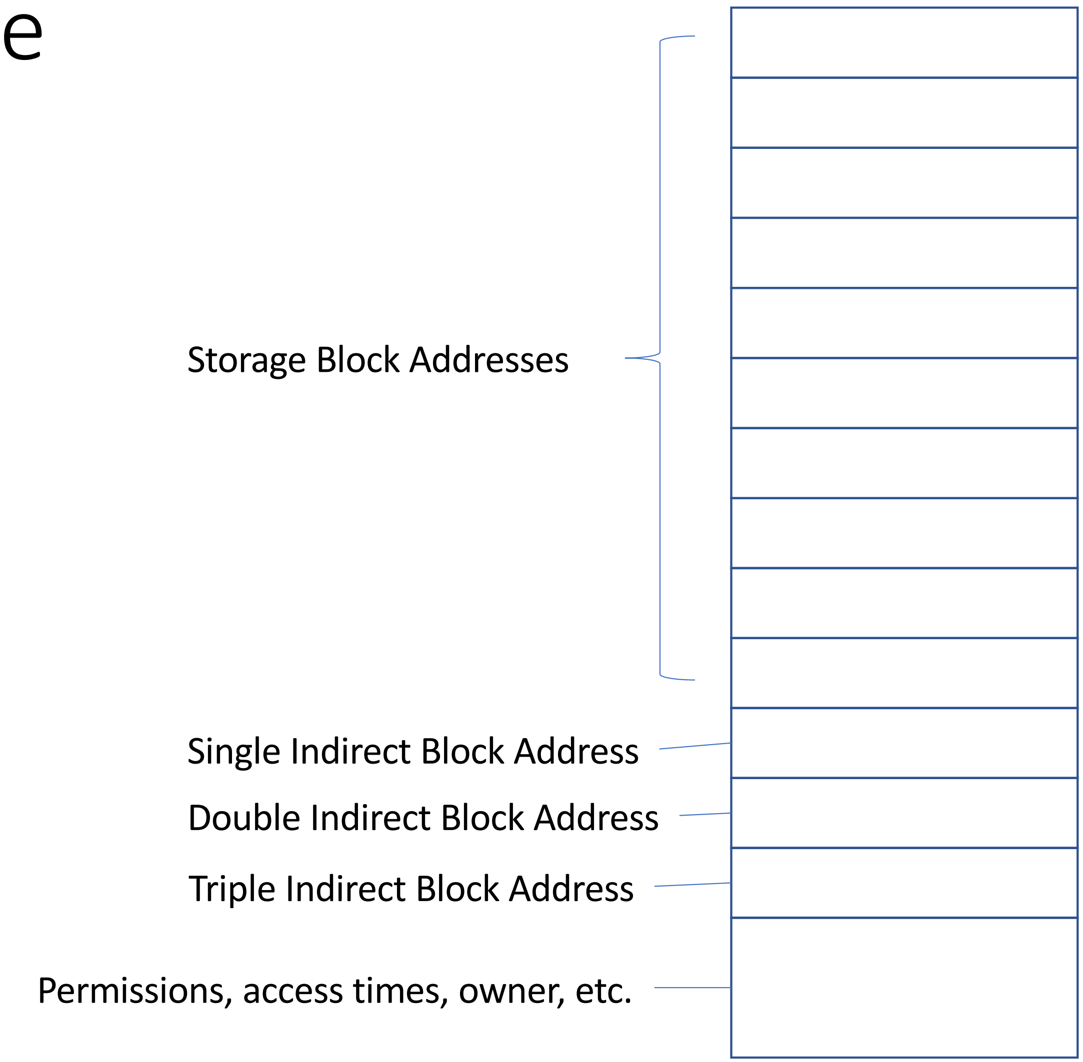
How is the data stored?

- The file data is stored in persistent storage as separate 4K blocks
- For Linux, each block is accessed separately by block address
- The file is a logically contiguous array of bytes

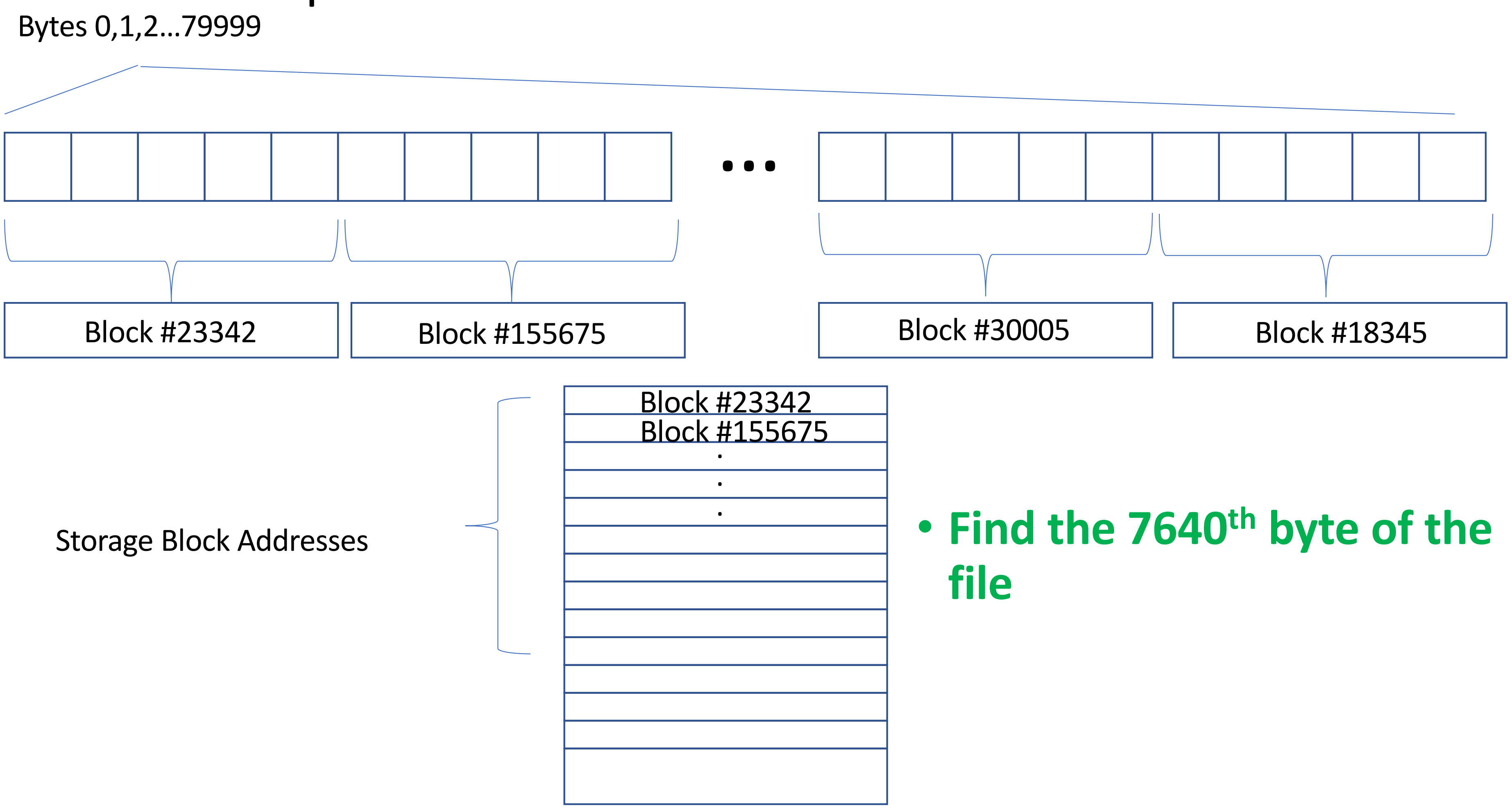


Need a Map: The inode

- Each storage block address contains a 4K segment of the file
- The index into the storage block addresses indicates which 4K segment



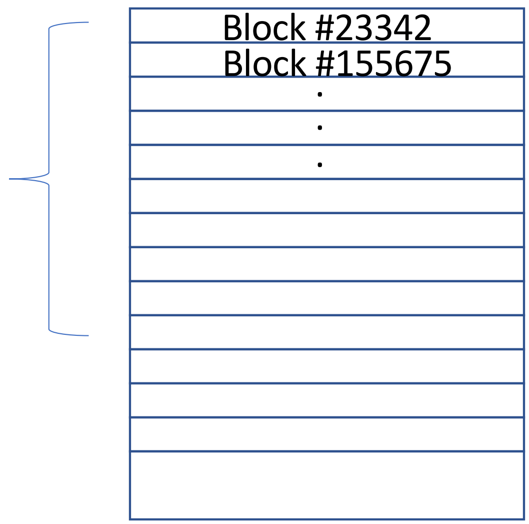
For example



- **Find the 7640th byte of the file**

Where is the 7640th byte of the file?

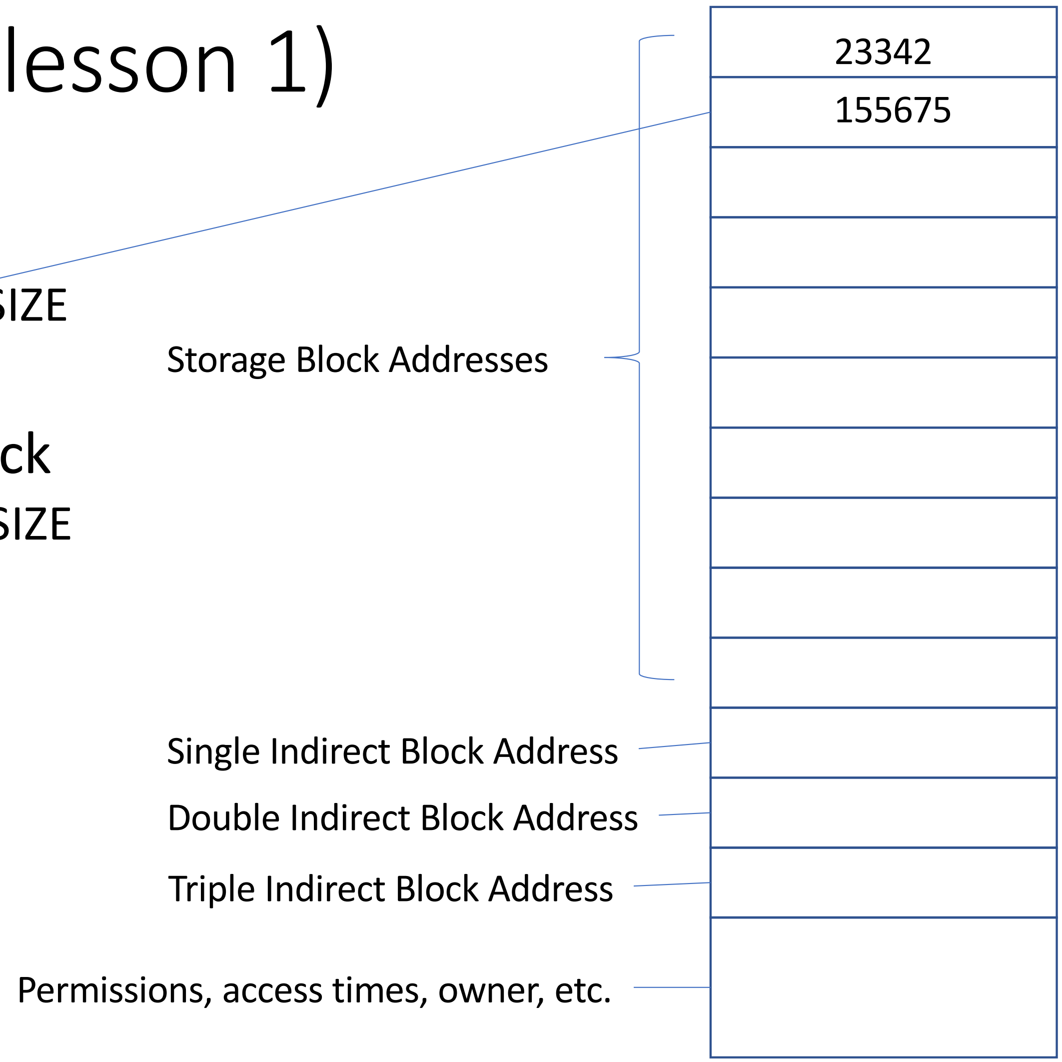
Storage Block Addresses



- Block #23342 contains the first 4K of the file
  - *Is 7640 between 0 and 4K-1*? => no
- Block #155675 contains the next 4K of the file
  - *Is 7640 between 4K and 8K-1*? => yes
  - The 7640th byte must be located in Block #155675 on the device
- Where in Block #155675 is the 7640th byte located?
  - It is the 3544th byte inside Block #155675
  - 7640 – 4096 = 3544

Byte locations in File (lesson 1)

- First, find the block
  - BLOCKSIZE = 4K
  - inode index = (byte #) ”div” BLOCKSIZE
  - 7640 / 4096 = 1
- Next, find the byte within the block
  - Byteoffset = (byte #) ”mod” BLOCKSIZE
  - Byteoffset = 7640 % 4096 = 3544



Modifying a byte in a file

- char c = 0;
- fd = lseek(fd,19230,SEEK_POS);
- write(fd,&c,1);

- Find the block on the device
  - inode index = 19230 / 4096 = 4
- Get Block address from inode at index
  - Block address = 7615
- Fetch the block from block address into memory
  - Read_disk(7615, Buffer)
- Compute byte offset
  - Byteoffset = 19230 % 4096 = 2846
- Change byte in the buffer
  - Buffer[2846] = 0
- Write the block back out
  - Write_disk(7615,Buffer)

Storage Block Addresses	23342
	155675
	3004
	22456
	7615
	33456
	100675
	2234
	34567
	5674
Single Indirect Block Address	
Double Indirect Block Address	
Triple Indirect Block Address	
Permissions, access times, owner, etc.	

i-node single block addressing summary

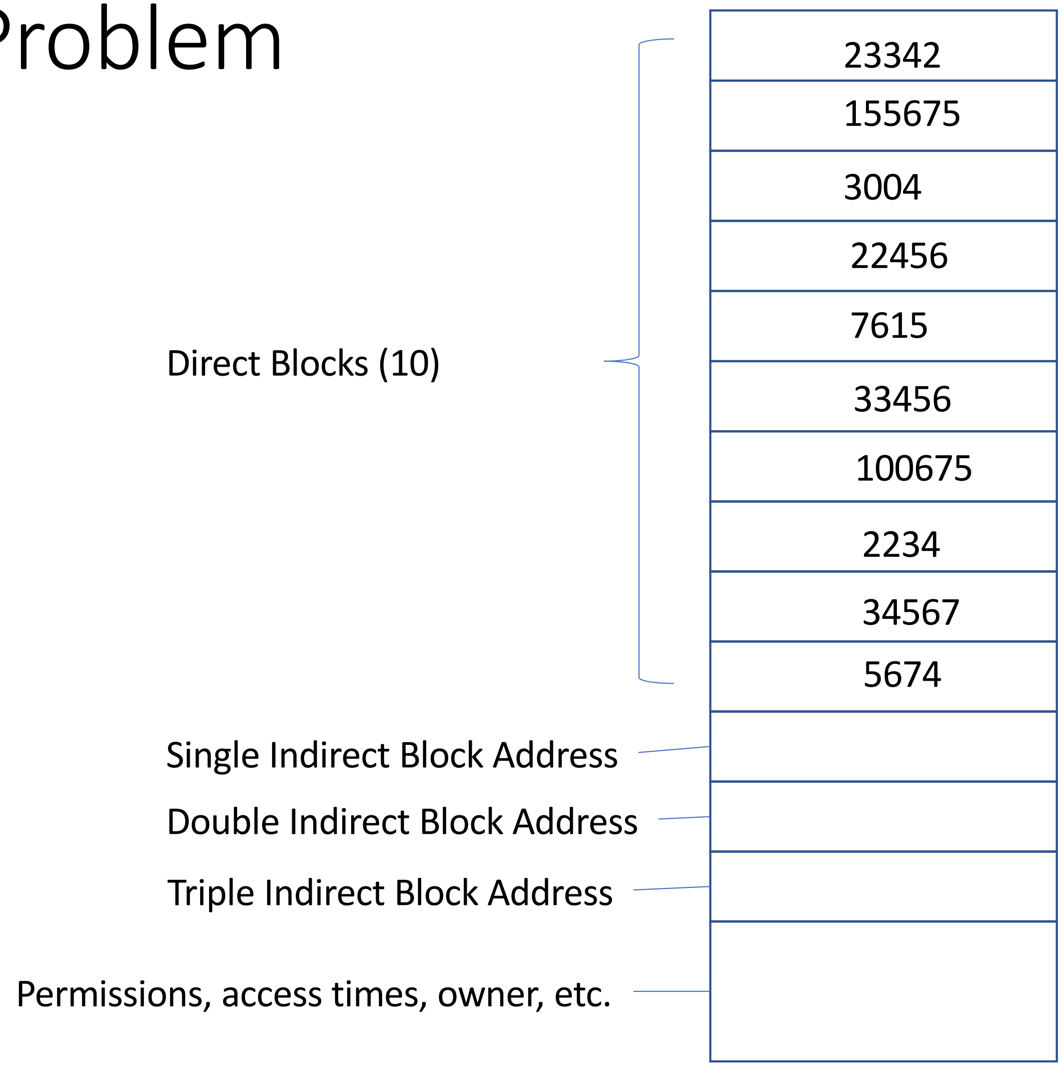
- inode contains an array of single block addresses, each of which can contain the address of a block on the device with a fixed block size (4K for modern Linux)
- Index into the array computes the number of blocks from the beginning of the file that a byte in the file must occupy
  - Index = (Byte #) / BLOCKSIZE
- Offset into the block is the "remainder" of the offset of the byte from the beginning of the file
  - Byteoffset = (Byte #) % BLOCKSIZE
- To access a byte within a file
  - Compute inode index
  - Get block number
  - Get Block from device
  - Compute offset into block
  - Access byte at offset into block
  - If the operation is a write, write the block back out

Some thoughts about i-nodes

- They are fixed size => block addresses are in an array – not a linked list
  - inode structure is defined statically when the kernel is compiled. Cannot be changed dynamically.
- i-nodes **must be stored on the device** (otherwise file would be lost when power is off) => where are they stored?
  - Well-known location (block numbers 1 through N on the device are reserved)
- The general term for this type of interface is “scatter/gather”
  - The disk is addresses randomly so data can be scattered => much better for fragmentation
  - The file is continuous so that data must be “gathered” (at least logically)
- At this point, answer the following question
  - ***How large is the largest file that can be stored using this method?***

Maximum File Size Problem

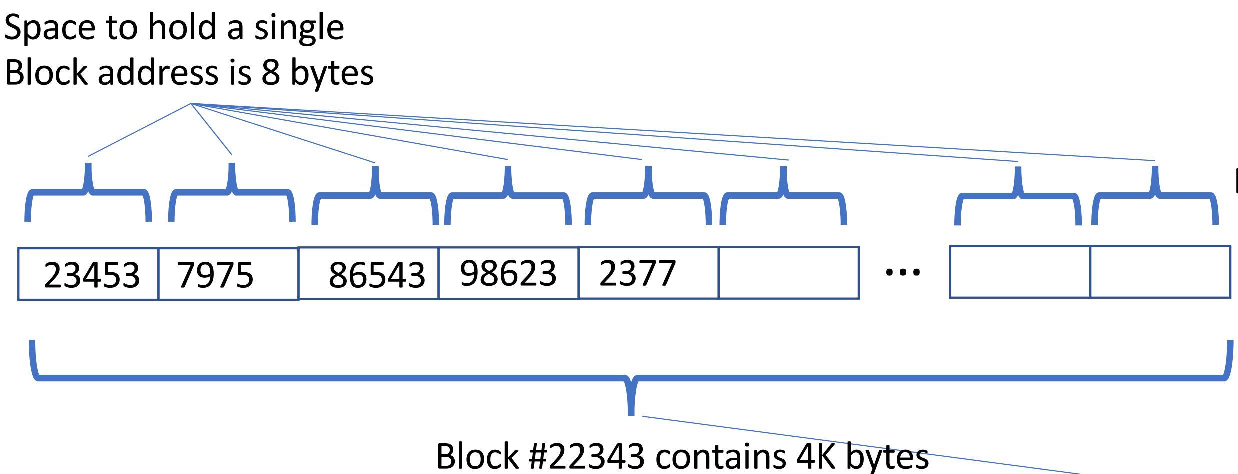
- Each entry in the i-node array contains a block address of a 4K block
  - Now call these "direct blocks"
- 10 direct block entries => **largest file is 40K**

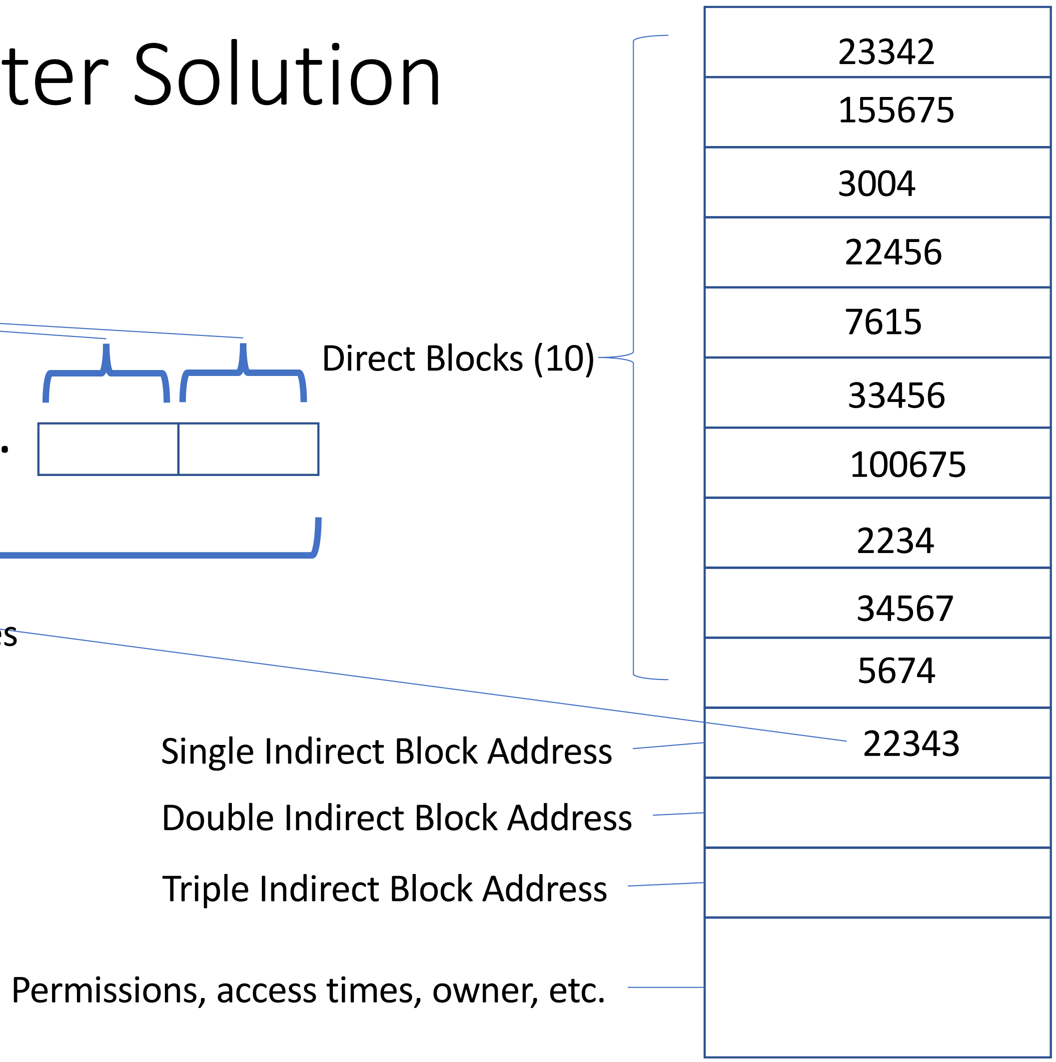


Can’t we make it bigger?

- Sure. Let’s give the i-node 1000 direct blocks
  - Largest file = 1000 * 4K = 4MB
- Bigger: 10^6 direct blocks
  - Largest file: 1,000,000 * 4K = 4GB
- How big is the i-node now?
  - Each element in a inode array is 8 bytes (block addresses are 8 bytes in length)
  - For a 4GB file, the inode would be 8MB
  - 8MB is > 4K => **we’d need another map to map 4K blocks for the i-node!**
  - **Still max file is only 4GB**

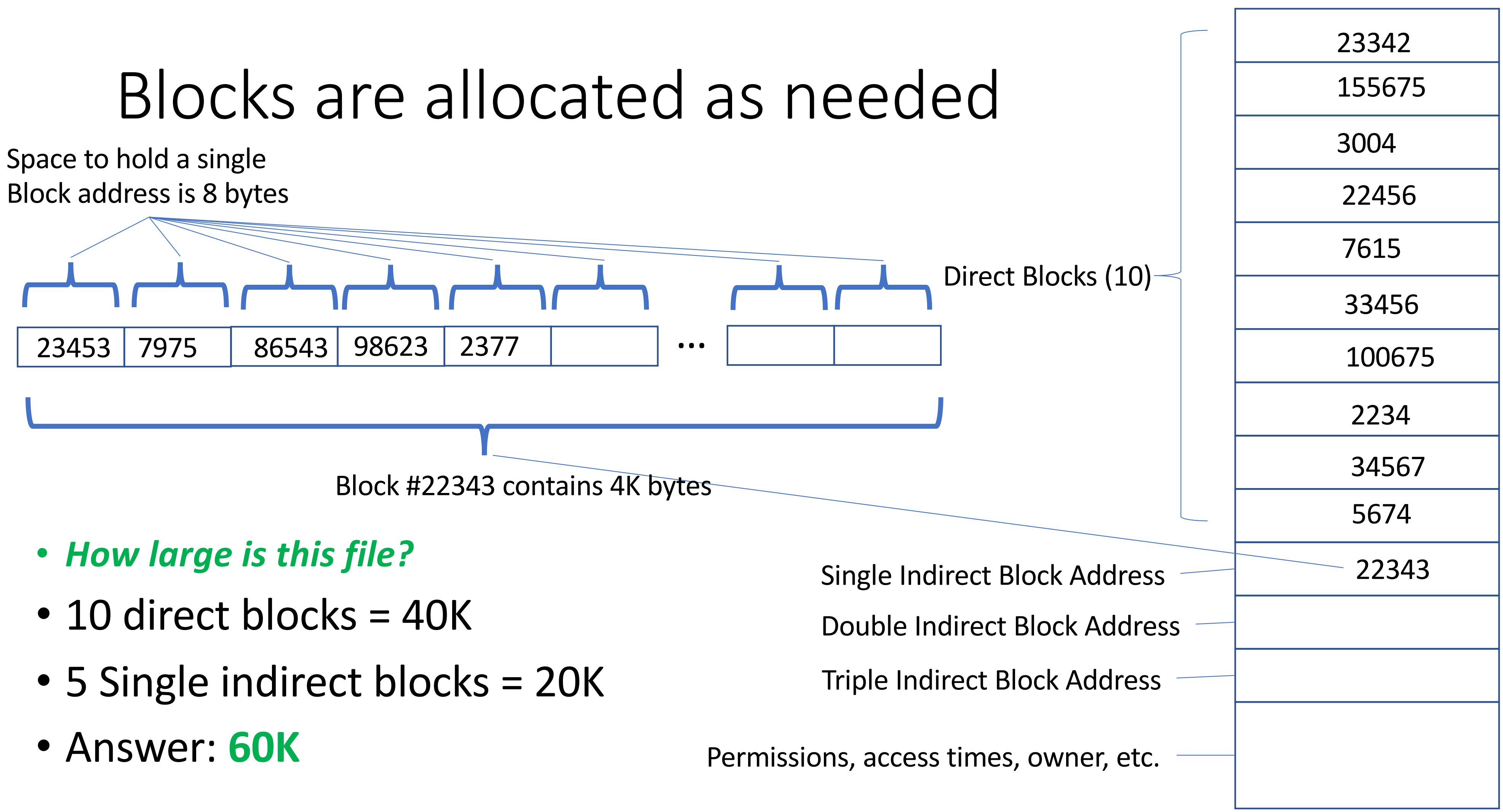
Indirect Block: A Better Solution





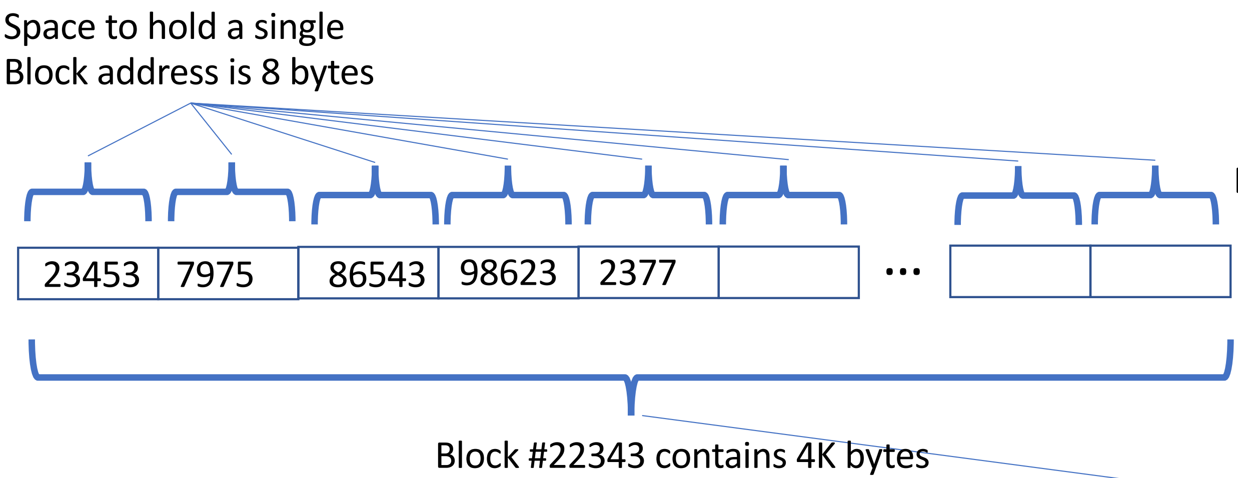
- ***How large is this file?***
- ***How large is the largest file that can be represented?***

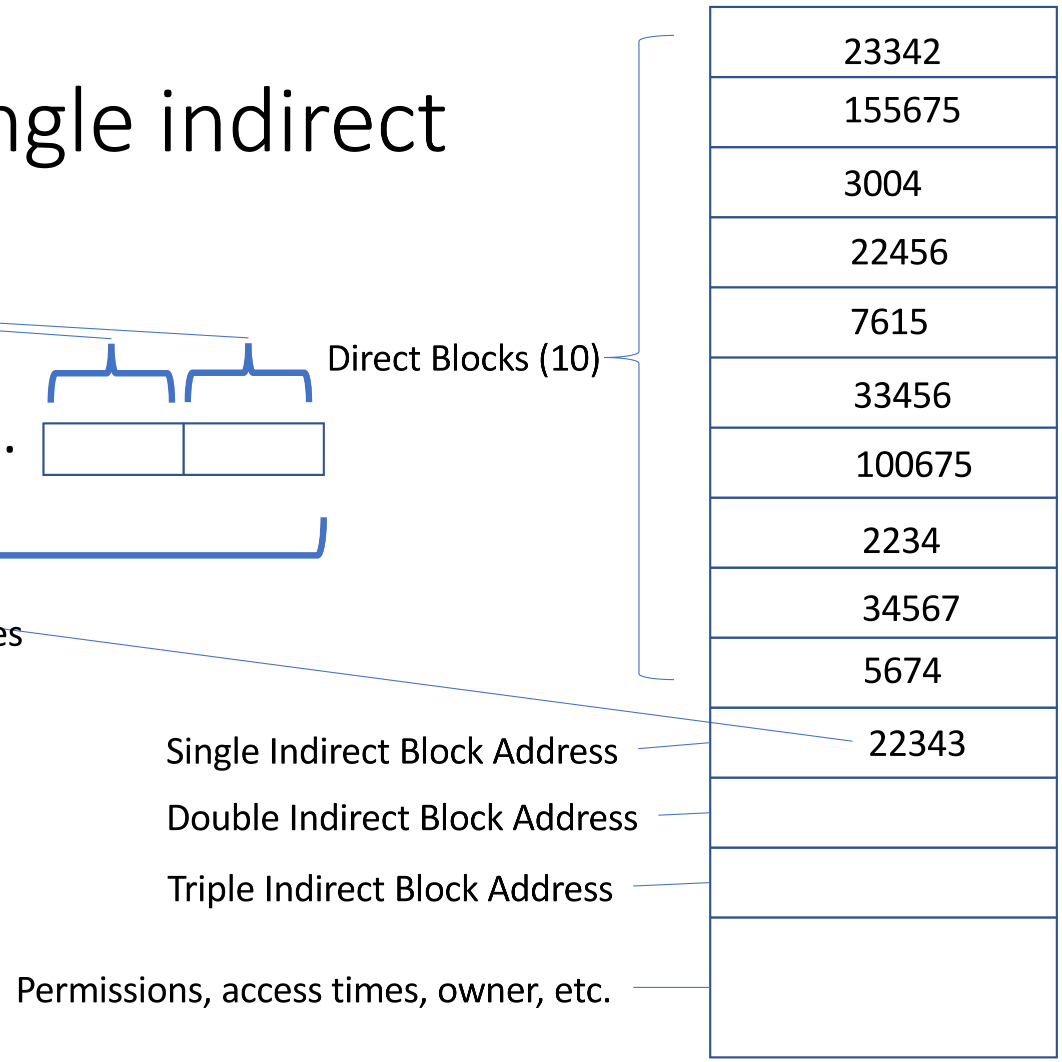
Blocks are allocated as needed



- ***How large is this file?***
- 10 direct blocks = 40K
- 5 Single indirect blocks = 20K
- Answer: **60K**

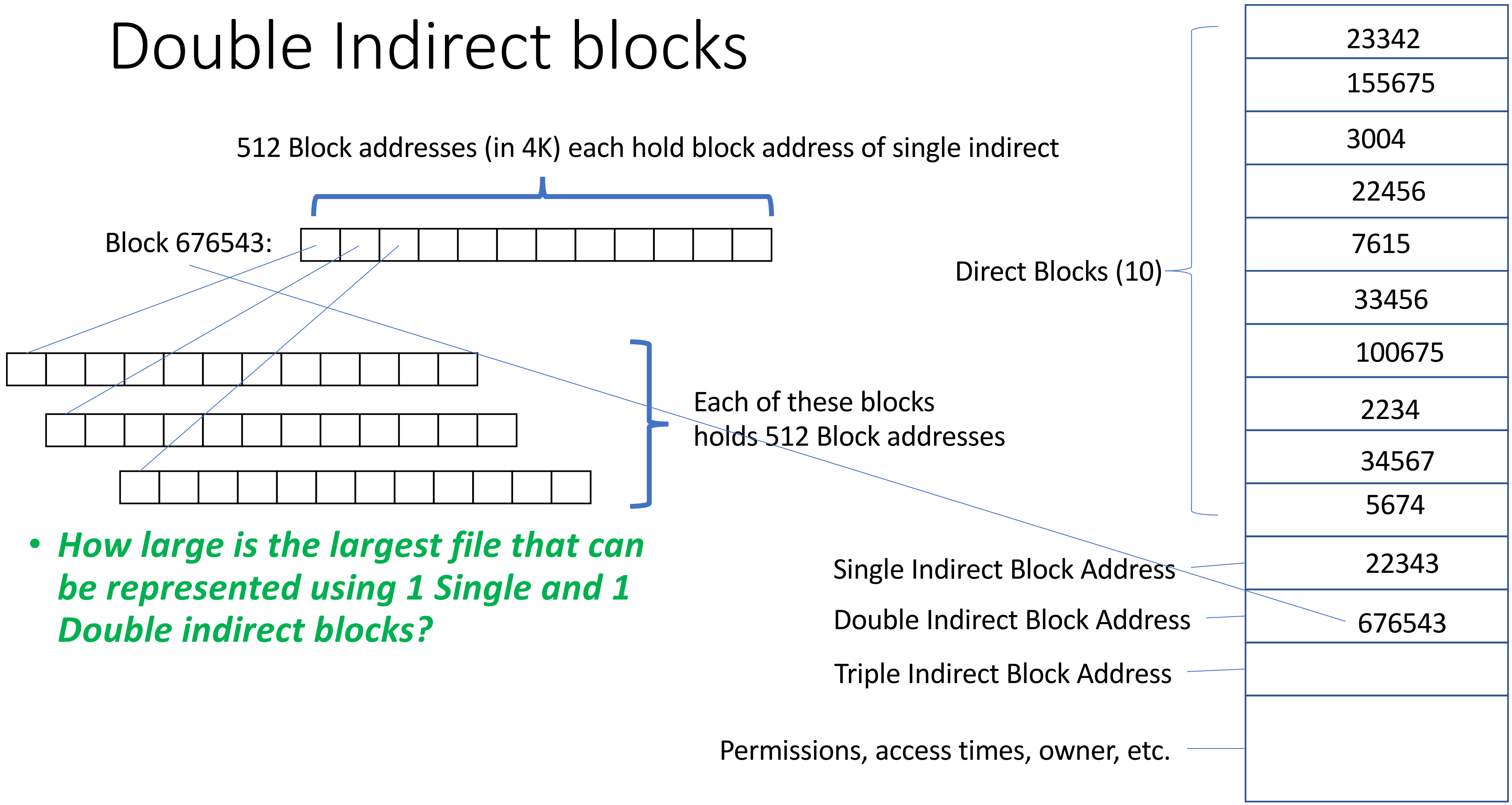
Largest File using Single indirect





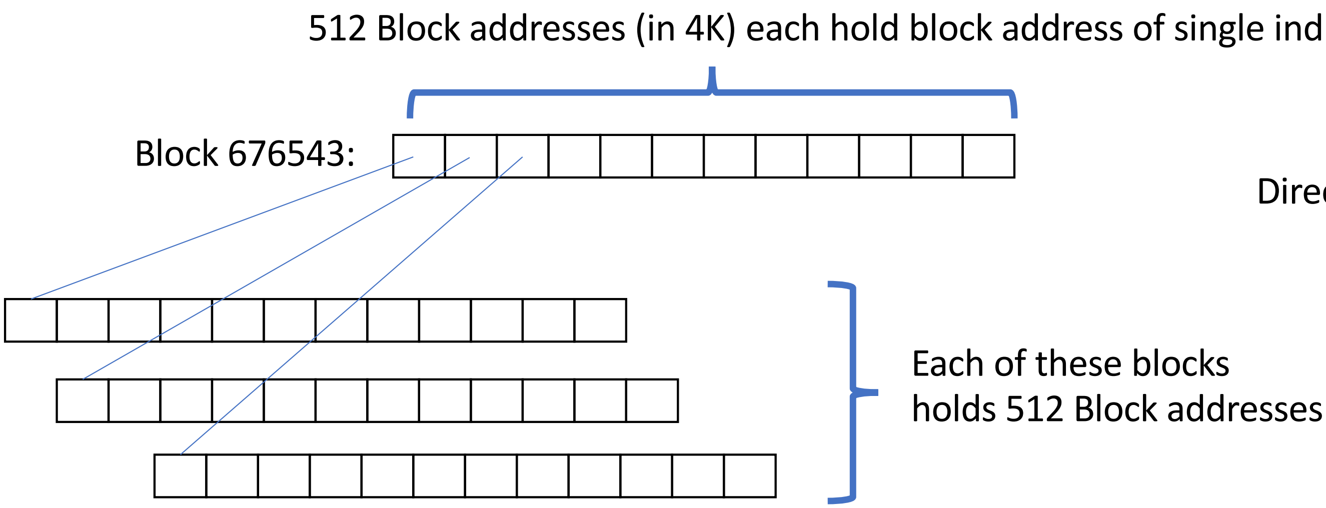
- ***How large is the largest file that can be represented?***
- One disk block can hold 4K/8 = 512 Block addresses
- 10 Direct blocks = 40K
- 512 Single indirect blocks = 2048K
- Answer: **2.088 MB**

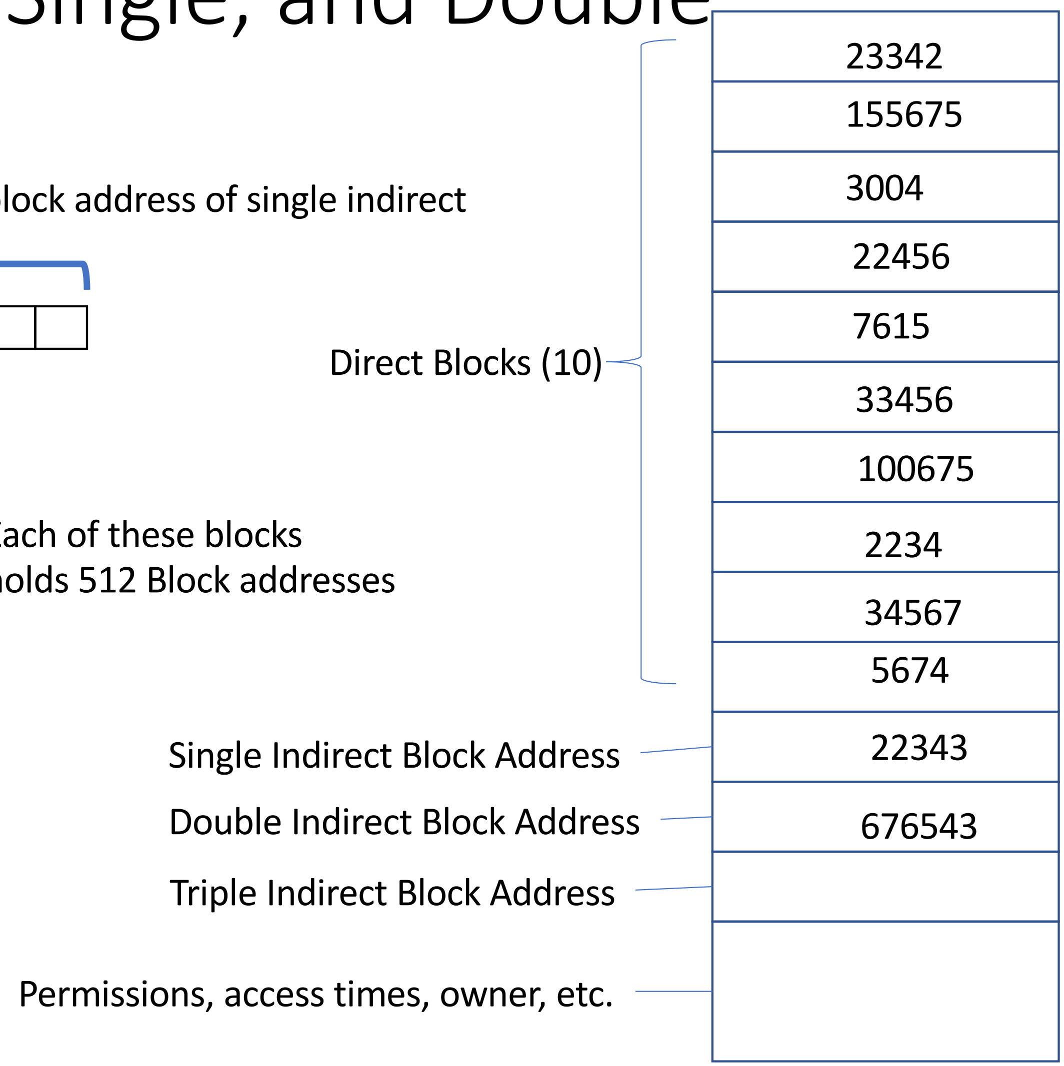
Double Indirect blocks



- ***How large is the largest file that can be represented using 1 Single and 1 Double indirect blocks?***

Largest File with Direct, Single, and Double Indirect





- 10 * 4K = 40K in Direct blocks
- 512 * 4K = 2048K in Single indirect
- 512 * 512 * 4K = 1048576K in Double
- Answer: 1050664K = **1.050664 GB**

Triple indirect block

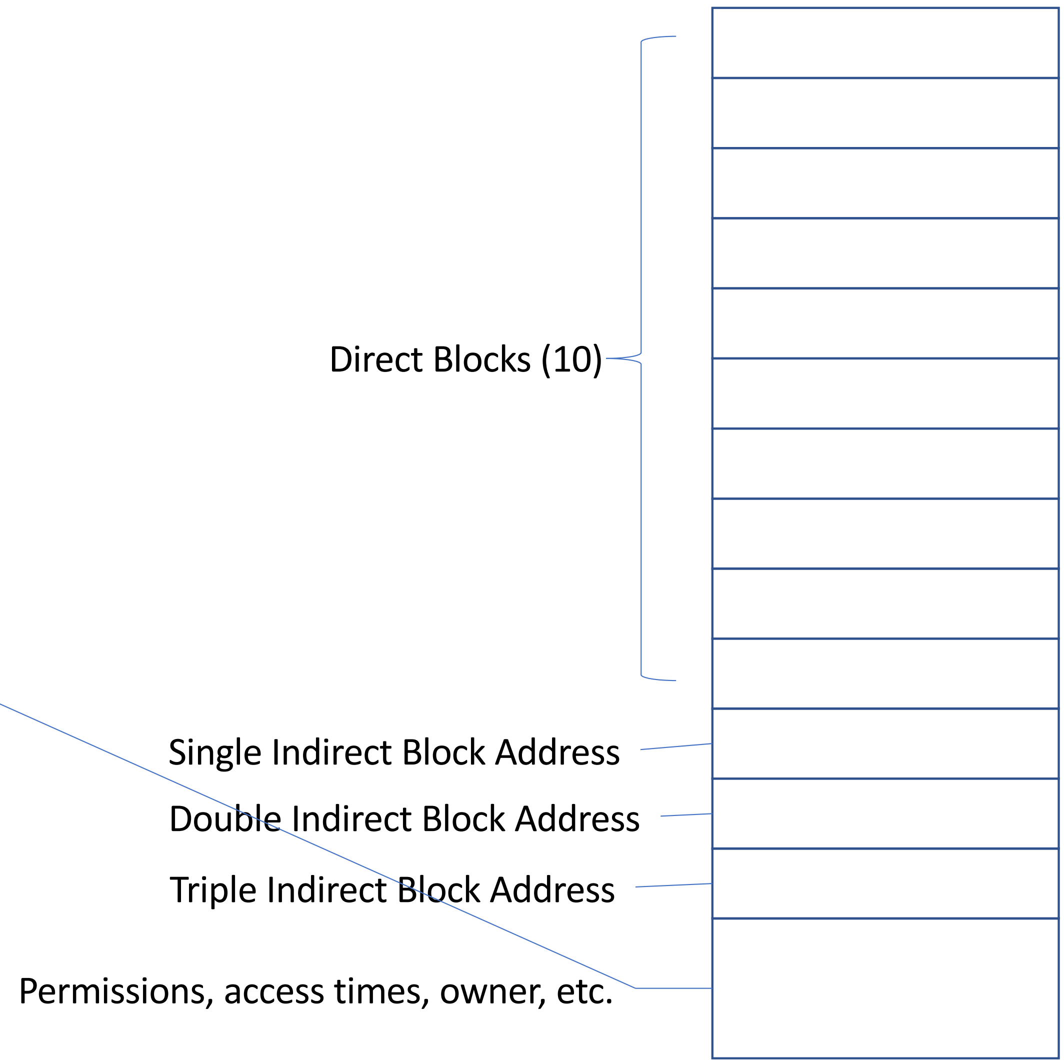
- Triple indirect block contains 512 addresses of Double indirect blocks, each containing 512 addresses of Single indirect blocks
- Max file size using 10 Direct, 1 Single, 1 Double, 1 Triple
  - 10 * 4K = 40K Direct
  - 512 * 4K = 2048K Single
  - 512 * 512 * 4K = 1048576K Double
  - 512 * 512 * 512 * 4K = 536870912 Triple
  - Answer: 537921576K = **537.921576 GB**

Current Linux EXT4

- From fs/ext4/ext4.h
  - #define **EXT4_NDIR_BLOCKS** 12
  - #define **EXT4_IND_BLOCK** **EXT4_NDIR_BLOCKS**
  - #define **EXT4_DIND_BLOCK** (**EXT4_IND_BLOCK** + 1)
  - #define **EXT4_TIND_BLOCK** (**EXT4_DIND_BLOCK** + 1)
  - #define **EXT4_N_BLOCKS** (**EXT4_TIND_BLOCK** + 1)
  - 12 Direct, 1 Single indirect, 1 Double indirect, 1 Triple indirect

File metadata

- File permission bits
- File owner and group
- File size (in bytes)
- File size (in blocks)
- i-node change time, file access time, file deletion time, file modification time
- Ref. count, allocated/free flag



More thinking about i-nodes

- The i-node is the file
  - There is no other record of the data blocks that belong to the file
  - There is one i-node / file
  - Data blocks are not shared among i-nodes
  - All accesses of the file require access to the i-node
    - Reads and writes
    - Permission changes
    - Ownership changes
- All i-nodes must be stored in persistent storage
  - If an i-node is lost, the file it describes is lost
- An i-node must be smaller than the size of a block
  - Otherwise, we'd need an i-node to describe the blocks containing an i-node
- ***How are i-nodes stored in persistent storage?***

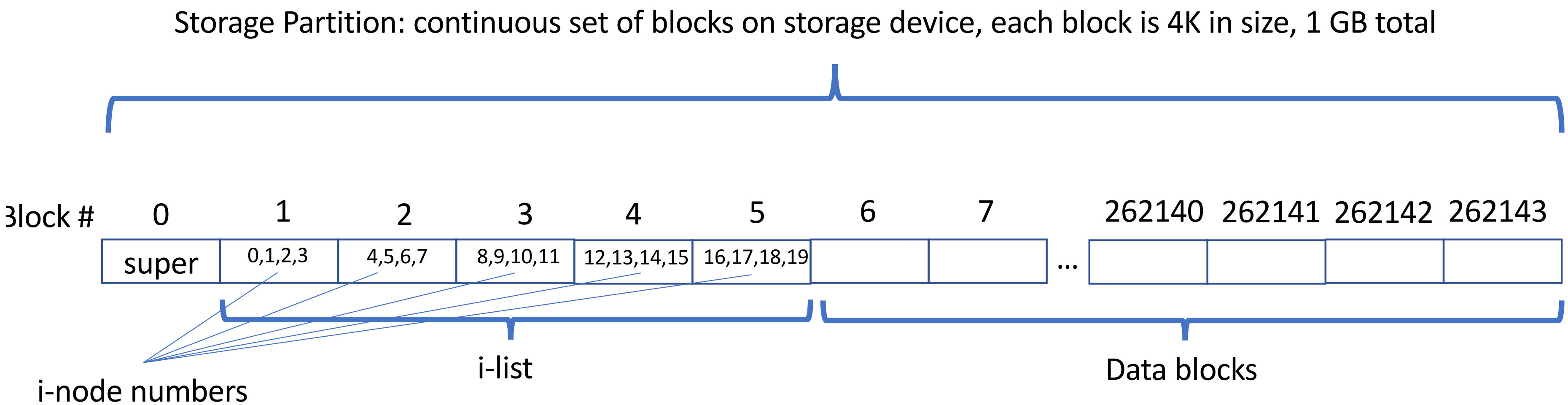
The File System

- Every file has an i-node and every file (that is not empty) has data blocks with addresses listed in the i-node or in indirect blocks linked to the i-node
- i-nodes must be stored on persistent storage and data blocks must be stored on persistent storage
- **File system**: a collection of i-nodes and data blocks that, together, can be used as files

Simple Linux File System Implementation

- **Storage partition**: a set of blocks on a storage device (possibly all of them) numbered 0 through N
  - A physical device can have multiple partitions or only one partition
- **Super block**: always block 0 in a storage partition that is being used to host a file system
- **i-list**: a contiguous list of blocks, numbered 1 to K where K << N
- **Data blocks**: all blocks that are not the super block or blocks in the i-list (N – (K+1)).

Example File System Layout

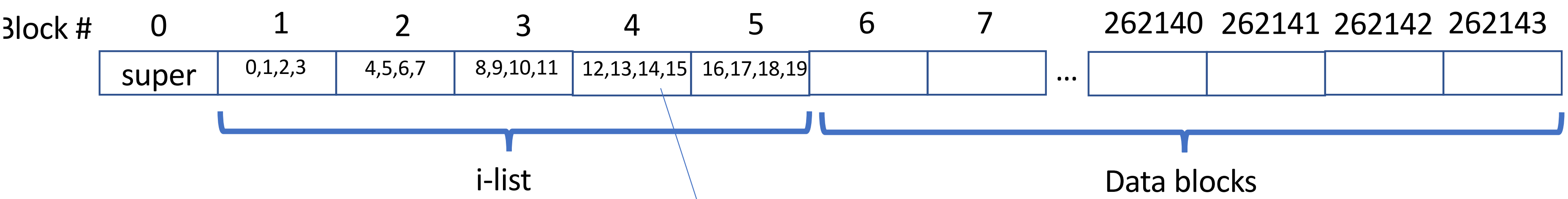


- Super block is block 0
- i-node is 1K (imagine)
- 5 blocks in the i-list

Observations About the Simple Linux File System

- The i-nodes are in a well-known location (just after the superblock)
- The size of the i-list (in blocks) is fixed when the file system is configured
- The size of an i-node must be smaller than the size of a block
  - Multiple i-nodes fit into a single block
- Each i-node has a unique number
- ***Given an i-node number, what disk block is the i-node in?***

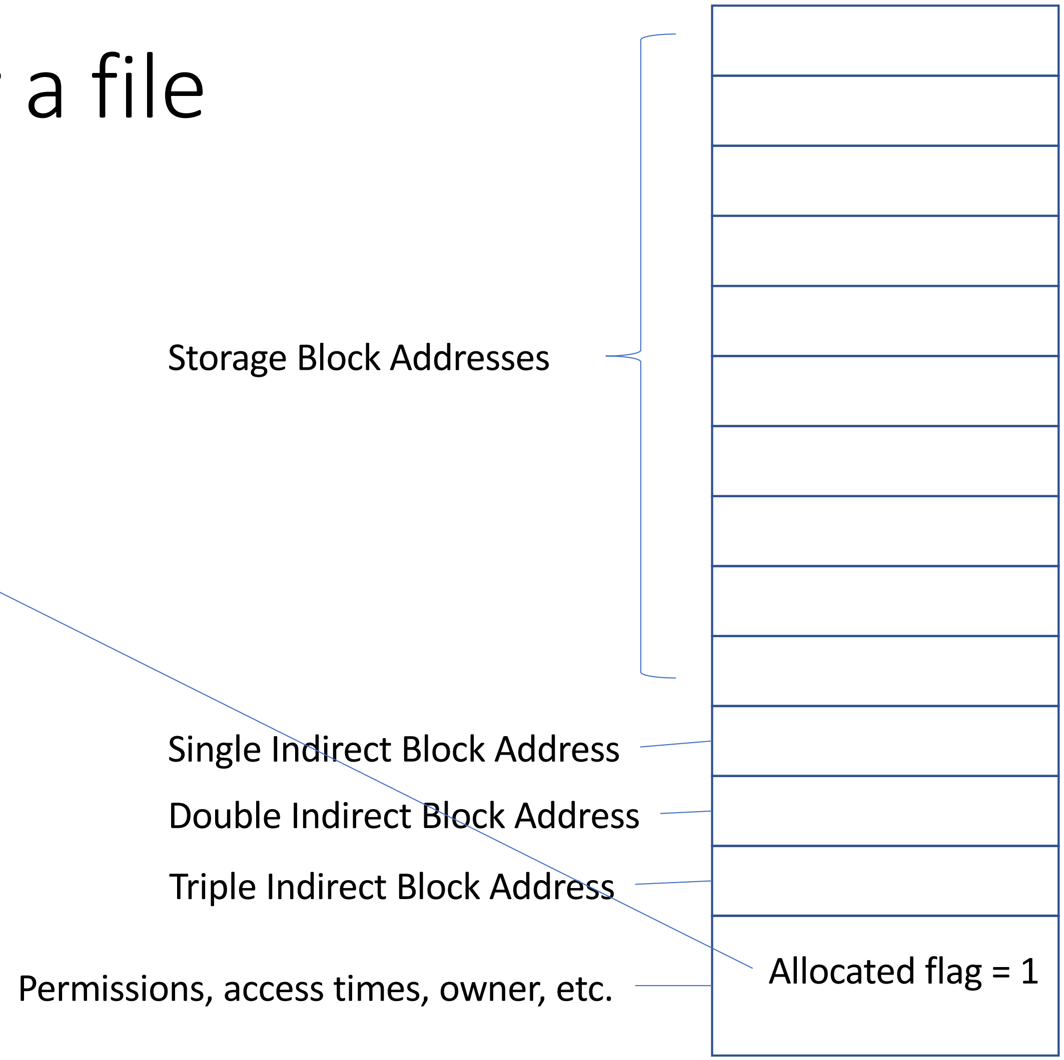
For example: Find i-node 14



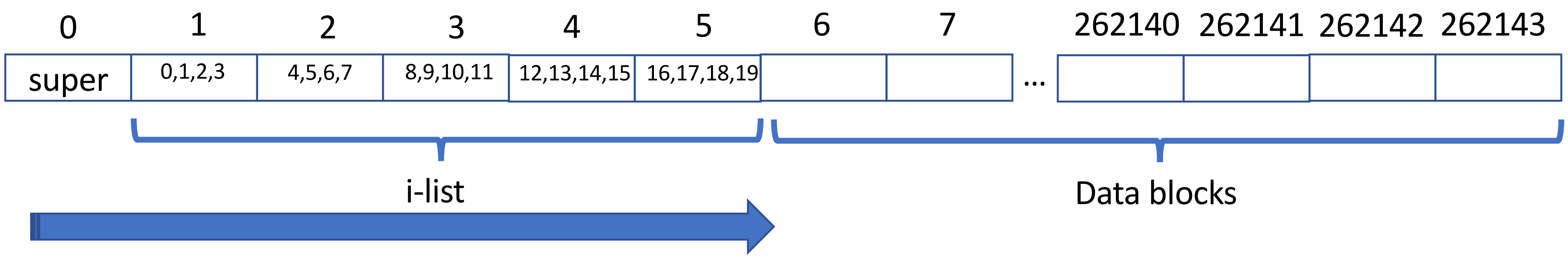
- i-nodes/block = (block size) / (i-node size) = 4
- Block # (i-node 14) in i-list = (i-node #) / (i-nodes/block) = 14/4 = 3
- Add block number in i-list to start of list = 3 + 1 = 4
- i-node 14 is in block 4 in the disk partition
- Which i-node is i-node 14 in block 4?
  - (i-node #) % (i-nodes/block) = 14 % 4 = 2
- **i-node 14 in is block 4 in position 2**

Allocating an i-node for a file

- When a file is created, an i-node is chosen from the i-list and allocated to the file
- i-node contains an allocated flag indicating it is the i-node for a file
    - fd = open("foo",O_CREAT,0600);
- The create code in the kernel scans the i-list looking for the first i-node with the allocated flag not set
- No data blocks are allocated when the file is created



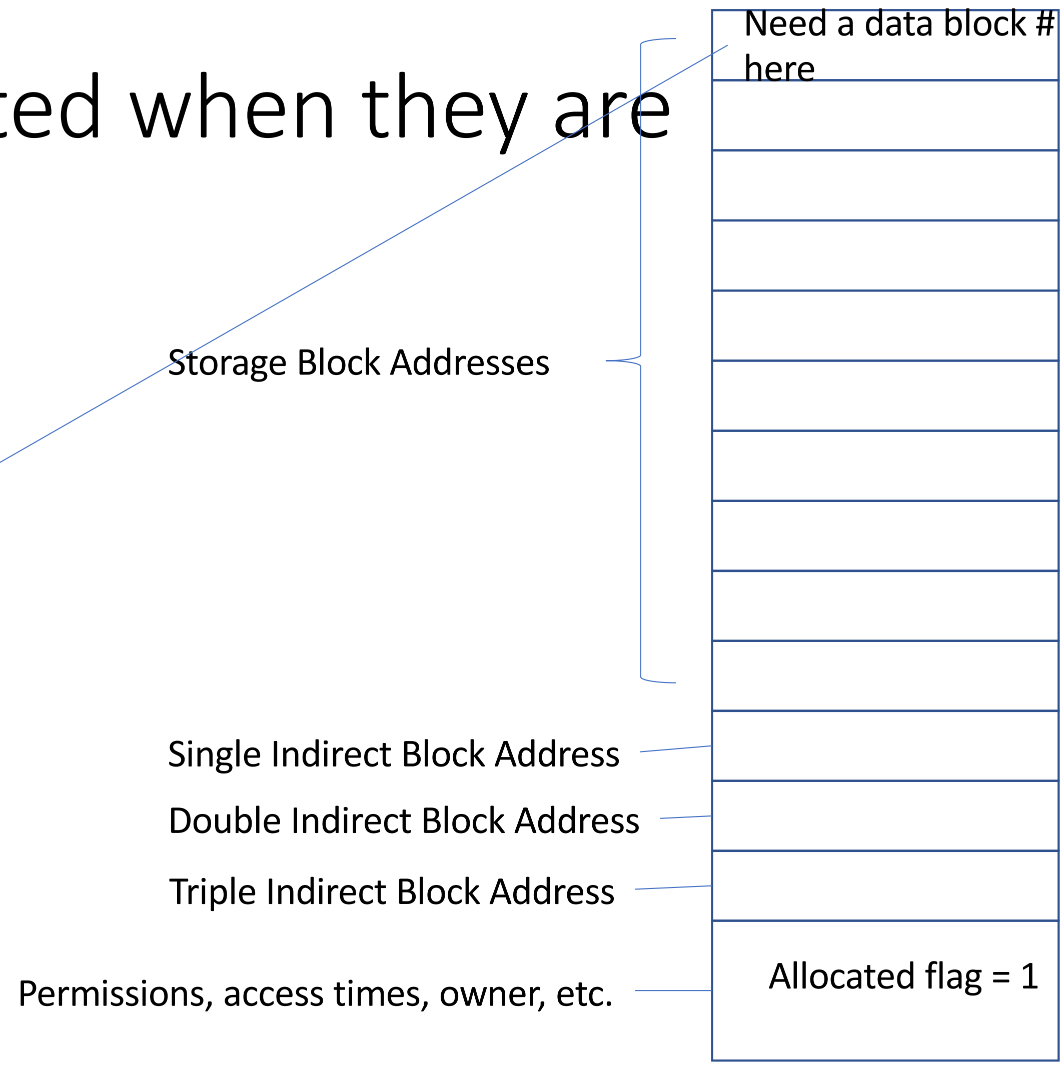
Scan the i-list for first free i-node on create



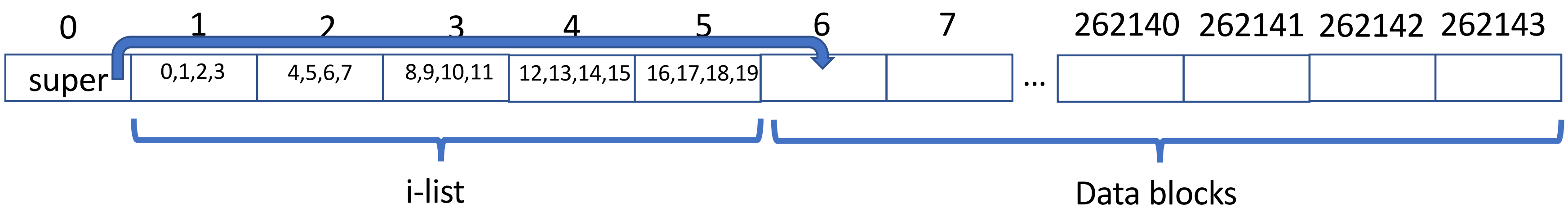
- On file create
  - Read each block in the i-list
  - Check allocated flag in each i-node in each block in the i-list
  - Return the first free i-node

Data blocks are allocated when they are needed

- char *buffer = “hello world\n”;
- write(fd,buffer,strlen(buffer));
- ***How does the kernel find a free block to allocate?***



Data block free list



- Unlike i-nodes, data blocks cannot be marked allocated/free
- When the file system is created, all data blocks are put on a free list
- The head of the free list is kept in the super block
  - When a free block is needed, it is taken from the head of the free list
  - When a block is freed (the file is deleted) it is put at the head of the free list
  - Only the head of the free list need be stored in the super block

Simple File System Free List

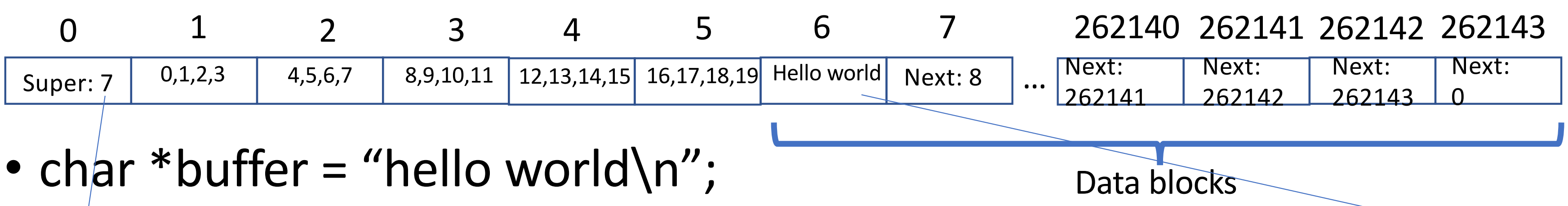
0	1	2	3	4	5	6	7		262140	262141	262142	262143
Super: 6	0,1,2,3	4,5,6,7	8,9,10,11	12,13,14,15	16,17,18,19	Next: 7	Next: 8	…	Next: 262141	Next: 262142	Next: 262143	Next: 0

i-list

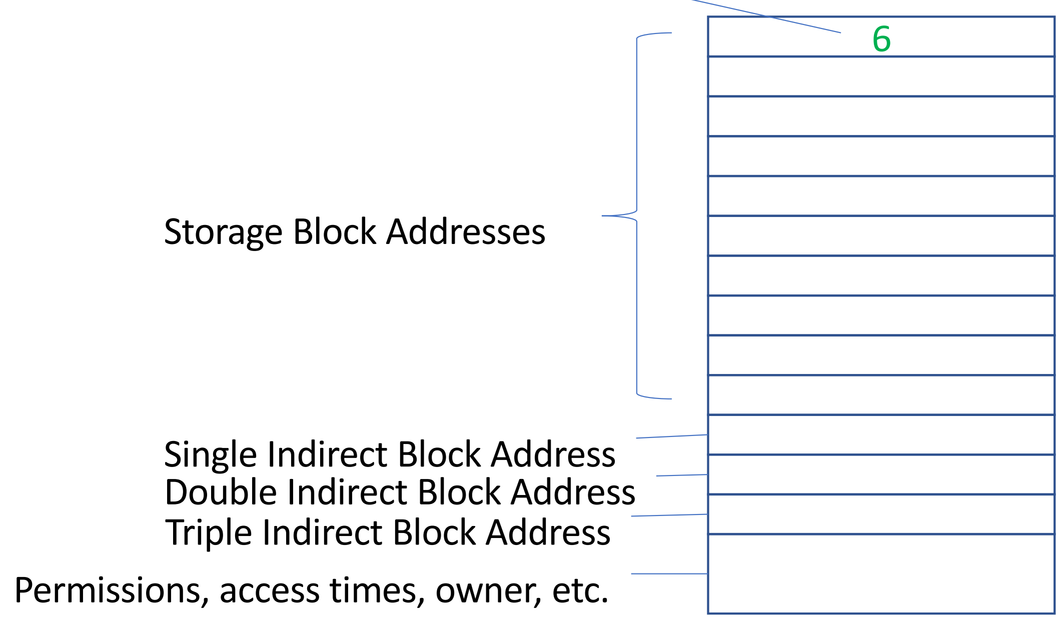
Data blocks

- Write first free block number in super block
- Write next free block number in that block
- Write next next free block number in next free block…an so on
- Next value of 0 indicates end of the list
- In this example, when the first block is allocated to a file
  - Block 6 is chosen on put in the i-inode in direct block index 0
  - 7 replaces 6 as the head of the free list in the super block

First write of example file



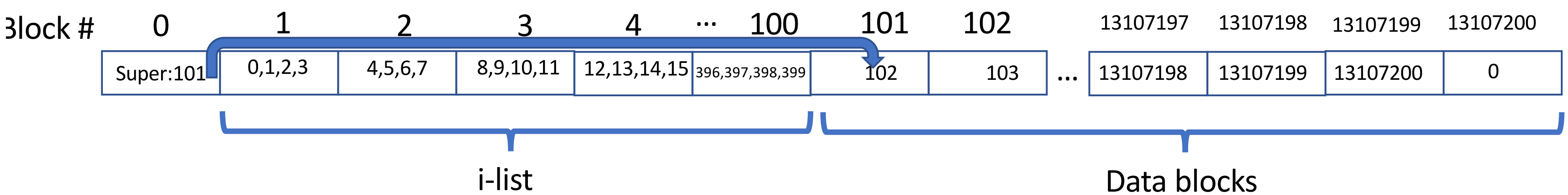
- char *buffer = “hello world\n”;
- write(fd,buffer,strlen(buffer));
- Block 6 comes off the free list and is allocated to the file by putting 6 in the file’s i-node
- 7 becomes the new head of the free list



Creating the Simple Free List

- This approach works, but there is a problem
- ***How many free data blocks in a 500 GB file system?***
  - Need to know how many blocks in the i-list
- ***How many free data blocks in a 500 GB file system with 100 blocks allocated to the i-list?***
- (500 GB / 4K) - 100 = 131071900 free data blocks
- Imagine each write is 20 ms
- ***How much time is required to create the free list?***

500 GB File System, 100 blocks on i-list, Simple Free List



- 20ms / write * 131071900 writes = 2621438 seconds
- 2621438 seconds is **30.3 days**
- **Simple free list takes too long to create when the file system is first configured**

Linux SysV Free List

- Recall
  - Each block address is 8 bytes
  - Each 4K data block can hold 4K / 8 = 512 block addresses
- Idea:
  - create blocks on the free list with addresses of free blocks (512 in each block)

The SysV Free List Organization

- Super: 101

Block 101 = 4K = 512 block address entries

614	102	103	104	···	611	612	613

Block 614 = 4K = 512 block address entries

1127	615	616	617	···	1124	1125	1126

Block 1127 = 4K = 512 block address entries

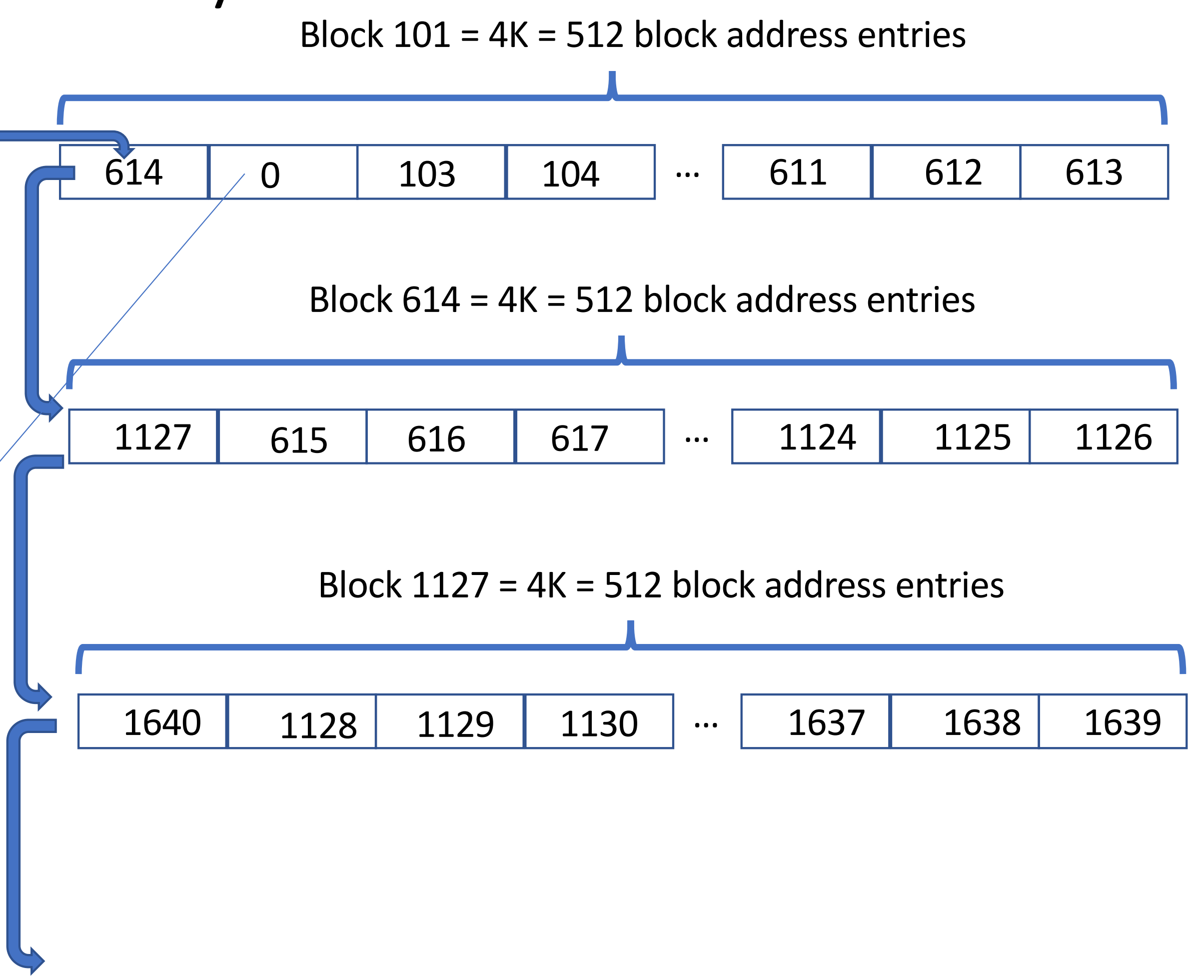
1640	1128	1129	1130	···	1637	1638	1639

How long does SysV take?

- Each write creates 512 free blocks
- 20ms / write * (131071900/512) writes = 5120 seconds
- 5120 seconds is **<u>1.4 hours</u>**
- **SysV can take a while to create a file system of this size**

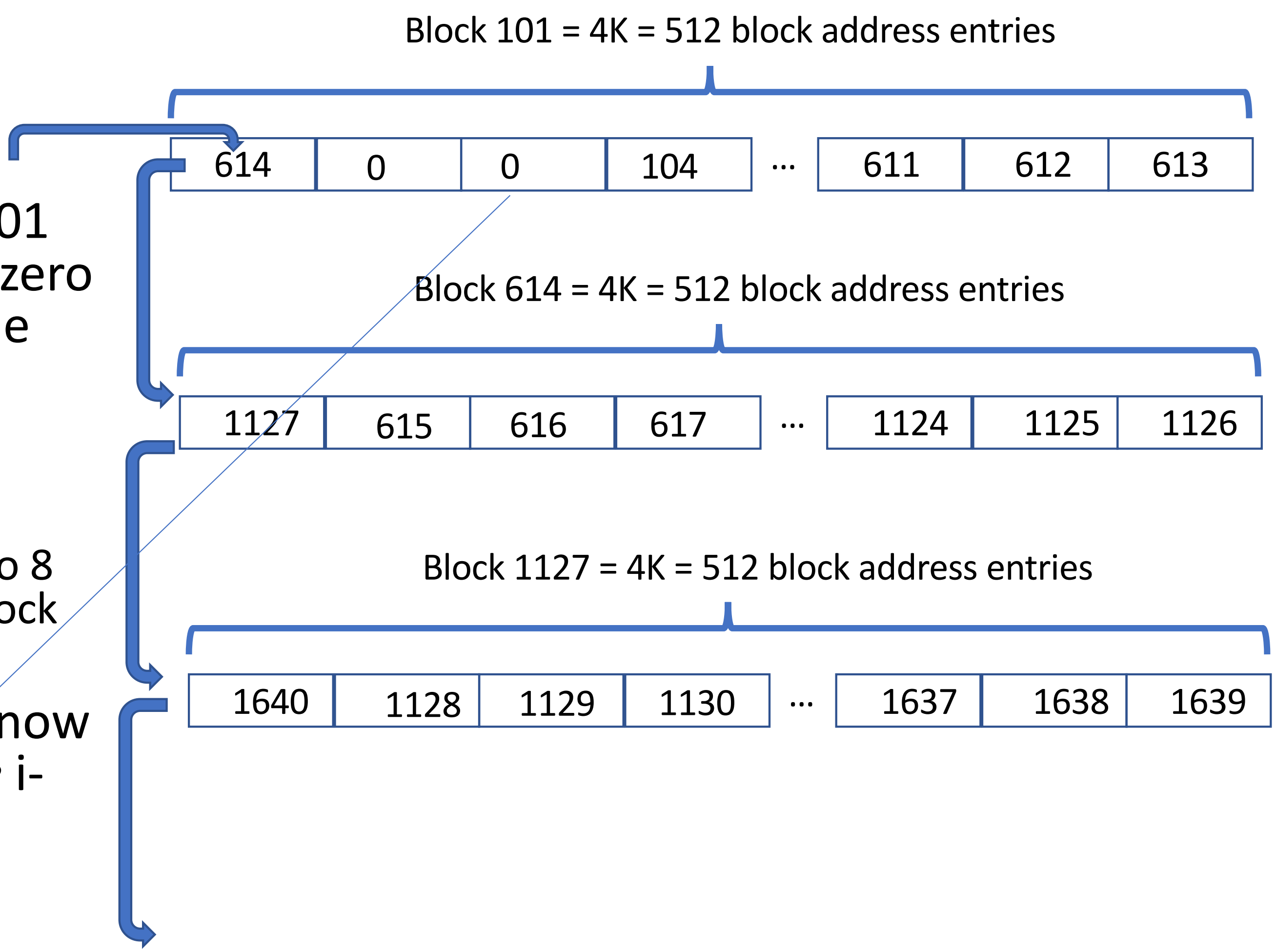
Allocating from SysV Free list

- Super: 101
- Scan block 101 for first non-zero entry (not the first entry)
- After 1st allocation
  - Write 0 into 8 bytes of block address
- Block 102 is now part of *some* i-node



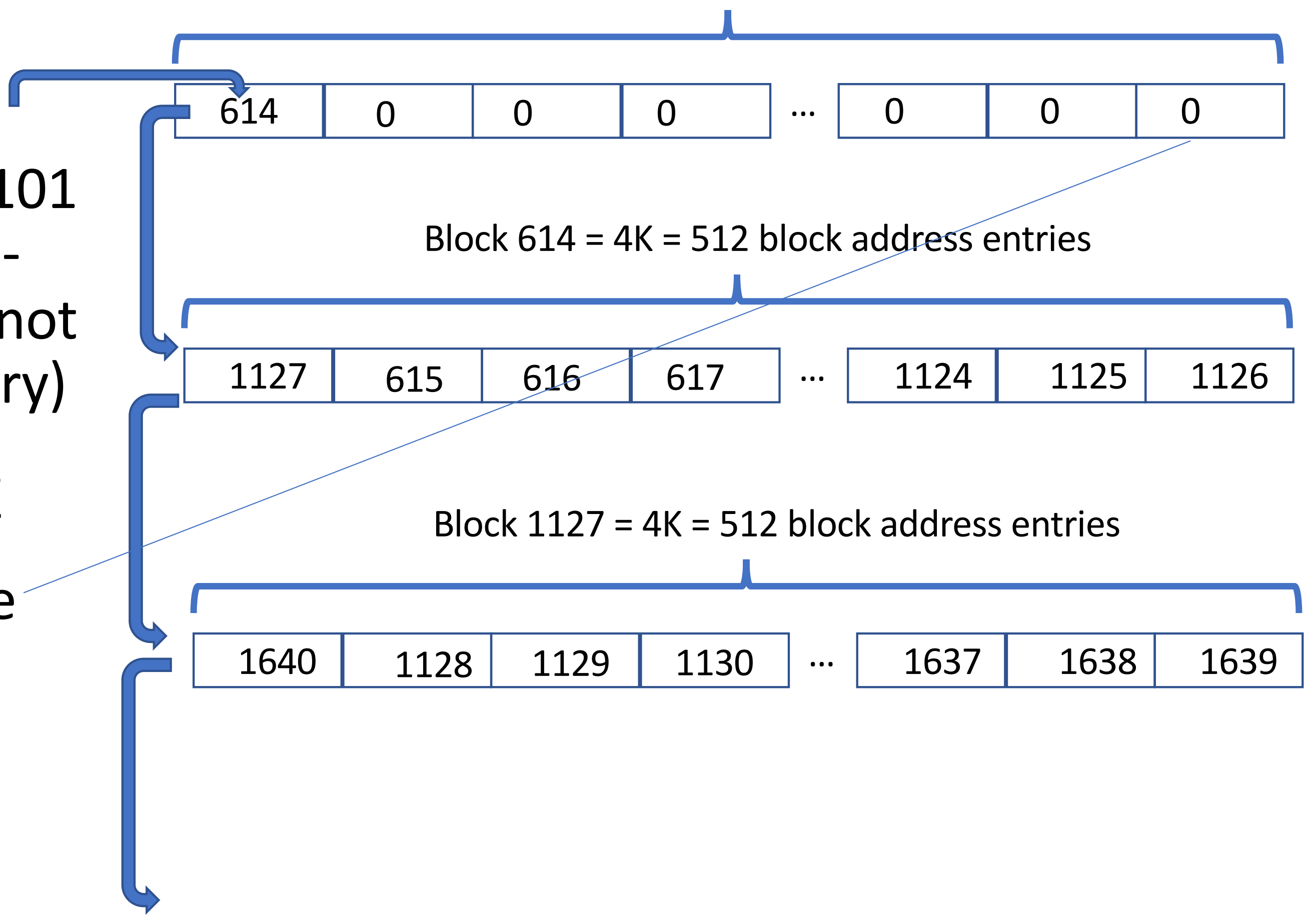
Another Allocation

- Super: 101
- Scan block 101 for first non-zero entry (not the first entry)
- After 2nd allocation
  - Write 0 into 8 bytes of block address
- Block 103 is now part of *some* i-node



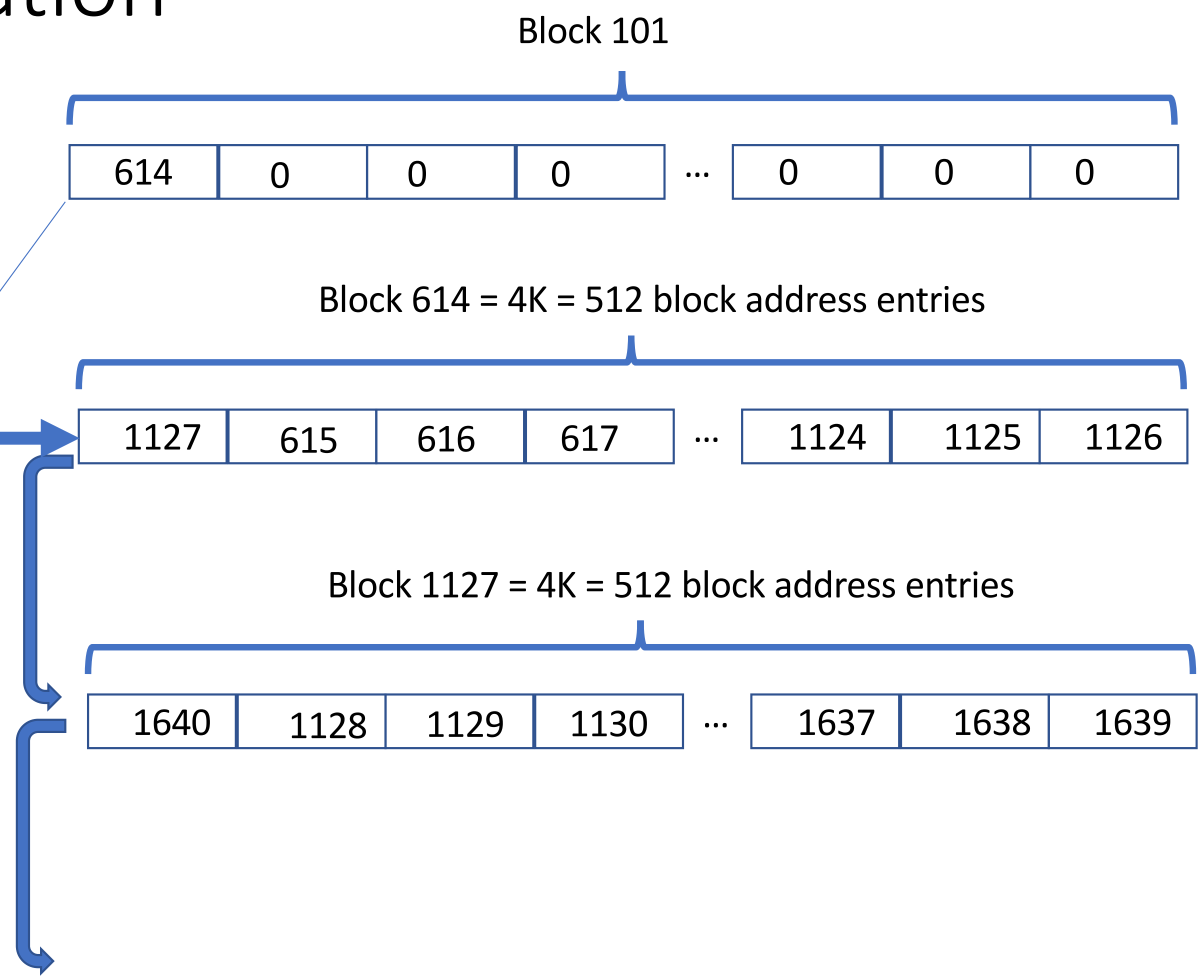
511th Allocation

- Super: 101
- Scan block 101 for first non-zero entry (not the first entry)
- Block 613 is now part of *some* i-node



512^{th} Allocation

- Block 101 is all zeros except for first entry
- Set Head pointer to block address in first entry
- Super: **614**
- Block 101 is now part of *some* i-node



Some thoughts about SysV File System

- File delete returns blocks from i-nodes to the free list at the head
  - Find zero entries in head of free list using a scan
  - When head of free list has 512 non-zero entries, returned block becomes new head of free list
    - Zero out the new block
    - Put old free list head in first entry
    - Change free list head in super block to new block address
  - Details left as an exercise
- In the example SysV File System (500 GB with 100 blocks of i-list)
  - ***What is the largest file that can be stored?***
  - ***How many files can this file system store?***

Largest file in the file system?

- Largest file that can be stored in file system
  - ((1024 * 1024 * 1024 * 500)/(4*1024) – 101) * (4 * 1024) = 536,870,498,304 bytes = **499 GB**
  - Recall if the i-node has 10 direct, 1 single, 1 double, and 1 triple
    - 10 * 4K = 40K Direct
    - 512 * 4K = 2048K Single
    - 512 * 512 * 4K = 1048576K Double
    - 512 * 512 * 512 * 4K = 536870912 Triple
    - Answer: 537921576K = **537.921576 GB**
  - **No file will ever use all of the i-node block addresses**

How many files?

- 100 blocks in the i-list and an i-node that is 1K => 4 i-nodes per block in the i-list
- File system can hold 400 files
- **Two ways that SysV file system becomes full**
  - Runs out of free data blocks on the free list
  - Runs out of free i-nodes in the i-list

mkfs: configuring the file system

- mkfs utility creates a file system
  - All raw disk partitions are represented in the /dev directory
  - mkfs has default a default size for the i-list (10% for i-list is common)
- mkfs for SysV
  - Zeros out i-list
  - Creates the SysV free list
  - Writes head of free list into super block
- *You will need to write a version of mkfs for your file system in this class*

Reading and writing files

- The kernel can only move blocks between persistent storage and memory
  - Read => get a block from storage and put it in memory
  - Write => put a block from memory into persistent storage
- Files are not accessed as blocks via the Linux file interface
  - char *buffer = “hello world\n”;
  - char *read_buffer[50];
  - int fd;
  - fd = open(“foo”,O_CREAT|O_RDWR,0600);
  - write(fd,buffer,strlen(buffer));
  - lseek(fd,0,SEEK_SET);
  - read(fd,read_buffer,strlen(“hello”));
  - printf(“%s\n”,read_buffer);

Allocate i-node and read into memory

```
char *buffer = “hello world\n”;
char *read_buffer[50];
int fd;
fd = open(“foo”,O_CREAT|O_RDWR,0600);
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen(“hello”));
printf(“%s\n”,read_buffer);
```

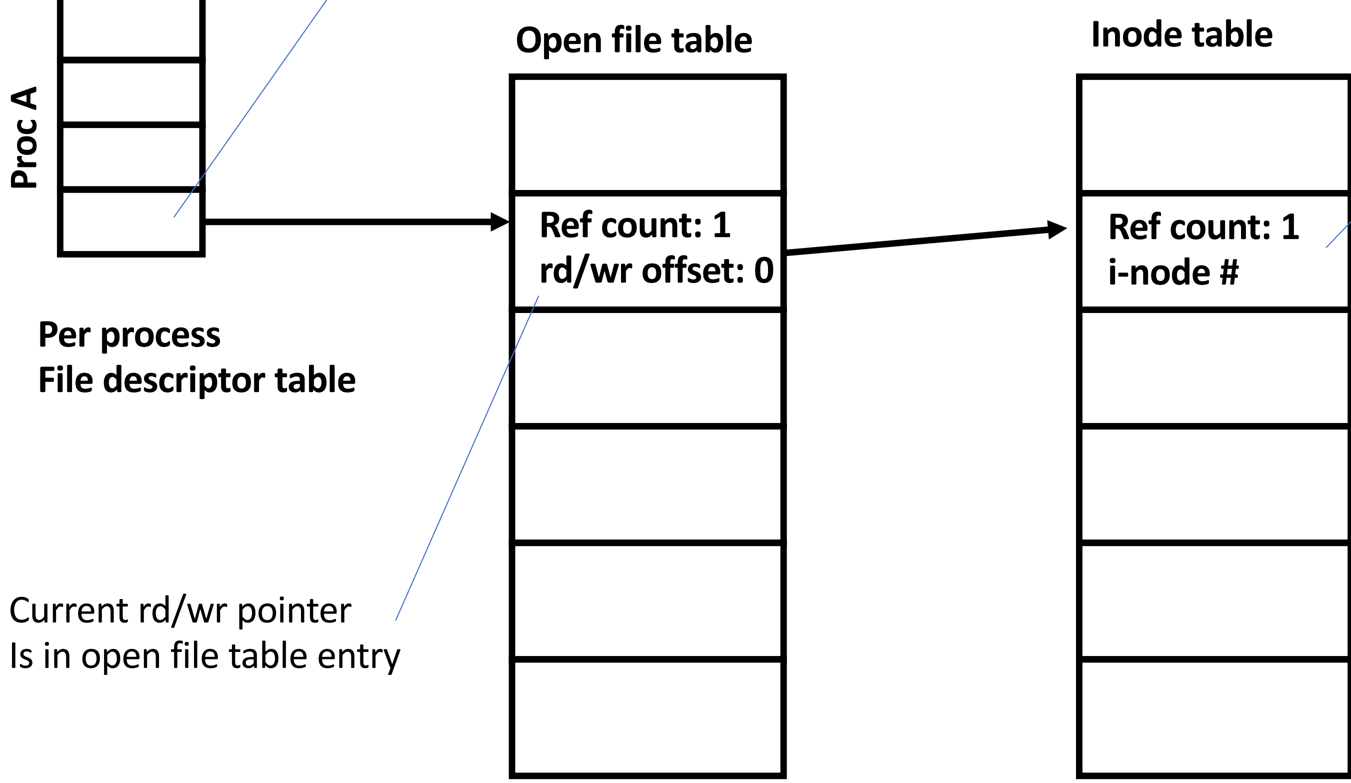
Open() system call in the kernel

- **Step 1**: Allocate a free i-node
  - **Step 1A**: allocate a buffer large enough to hold a disk block in kernel memory
  - **Step 1B**: read the first block of the i-list into that memory buffer
  - **Step 1C**: examine the allocate flag of each i-node in the disk block
    - If allocate flag clear, copy i-node into memory buffer for i-node and note the i-node number
  - **Step 1D**: if no free i-node in this block, repeat **1B** with next block in i-list
- **Step 2**: record permissions, creation time, and file owner in i-node in memory buffer
- **Step 3**: write the block containing the i-node back to disk

Kernel Data Structures for Files

File descriptor is an index
Into process file descriptor table

In-memory copy of i-node
stored in its own buffer



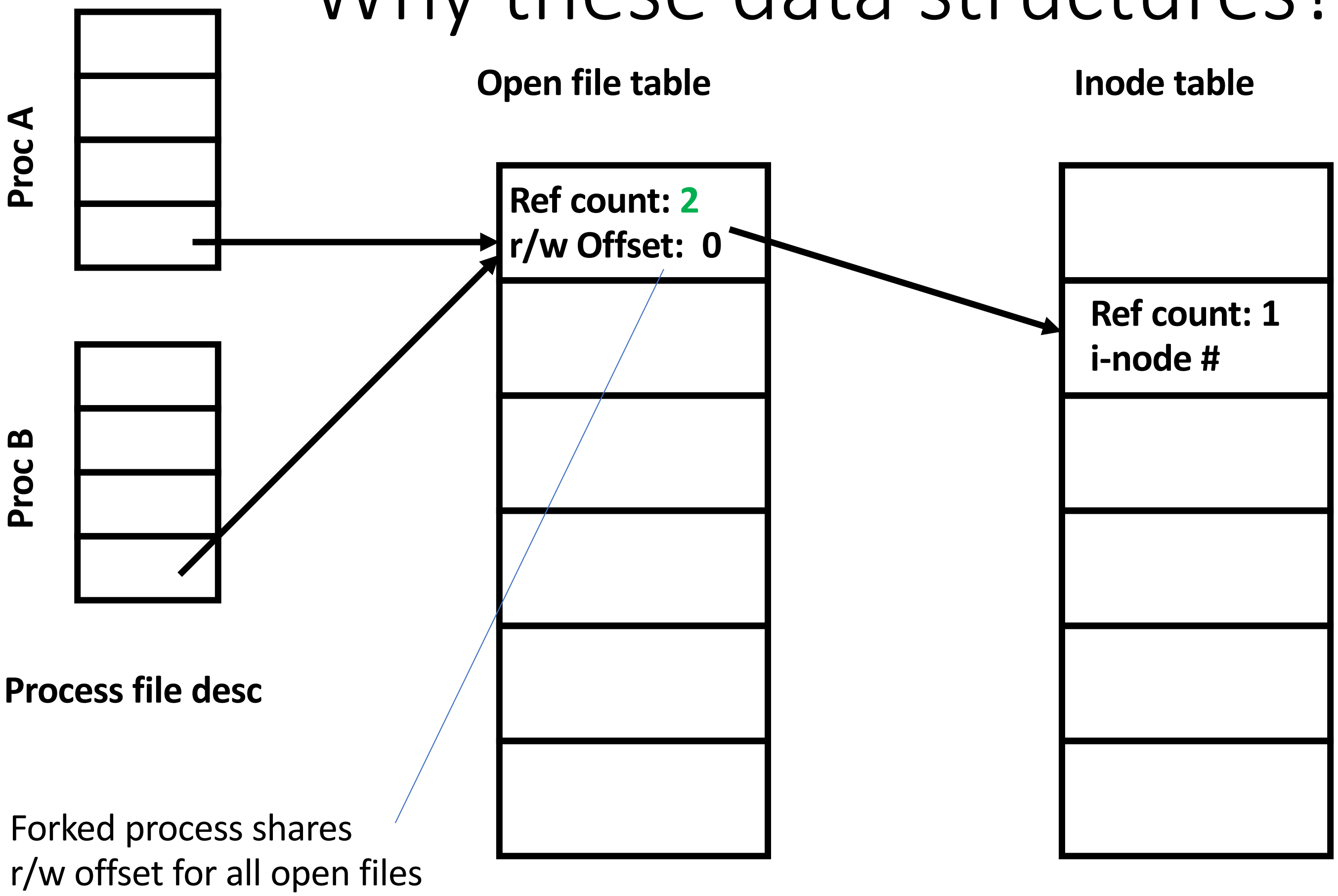
After the open() which creates a file, these are the kernel data structures that are created in kernel memory

Current rd/wr pointer
Is in open file table entry

NOTE that FUSE creates these for you

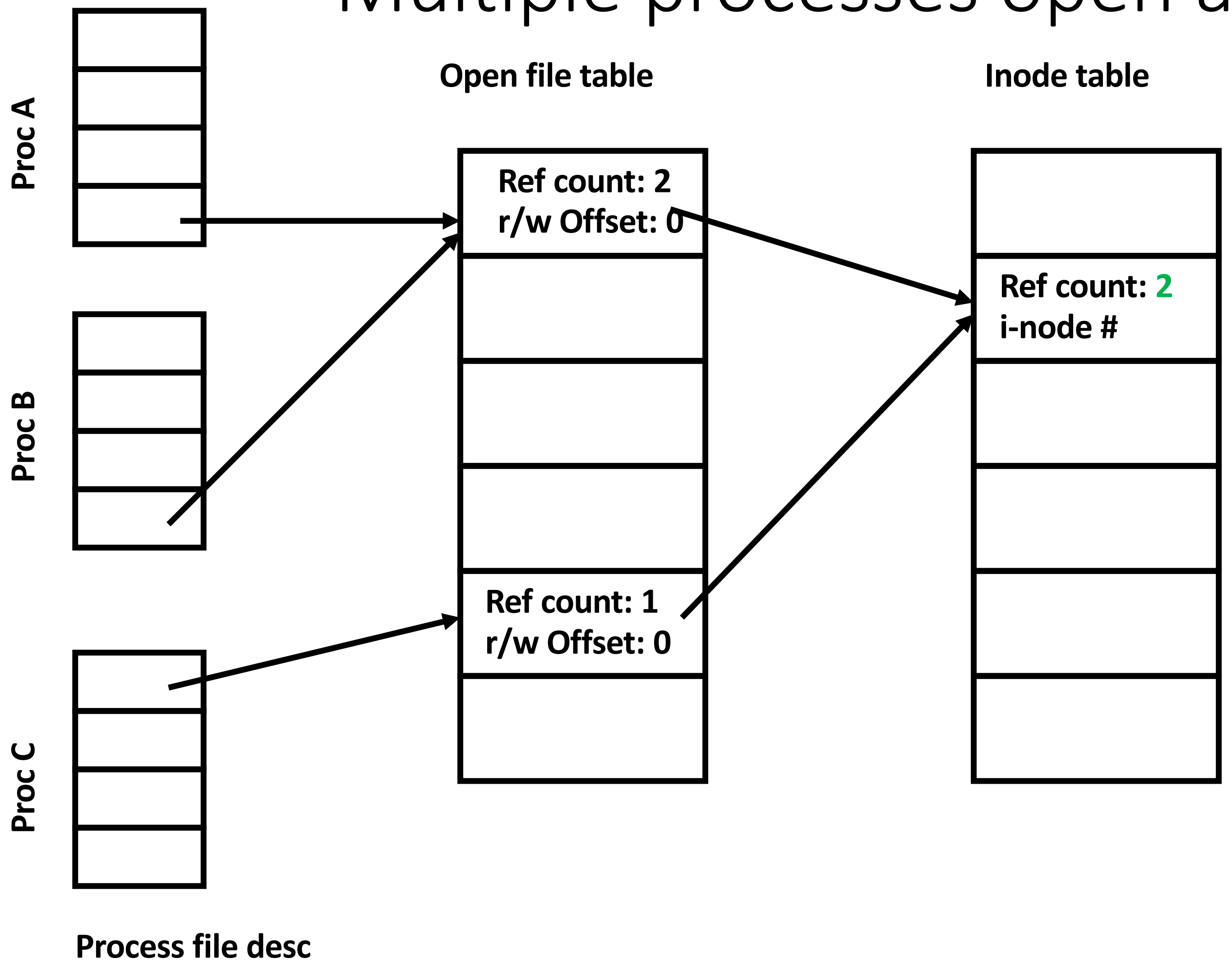
You do not need to implement these data structures in your file system

Why these data structures?




```
char *buffer = "hello world\n";
char *read_buffer[50];
int fd;
fd = open("foo",O_CREAT|O_RDWR,0600);
fork();
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen("hello"));
printf("%s\n",read_buffer);
```

Multiple processes open a file




```
char *buffer = "hello world\n";
char *read_buffer[50];
int fd;
fd = open("foo",O_CREAT|O_RDWR,0600);
fork();
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen("hello"));
printf("%s\n",read_buffer);
```

```
char *buffer = "hello world again\n";
char *read_buffer[50];
int fd;
fd = open("foo",O_CREAT|O_RDWR,0600);
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen("hello"));
printf("%s\n",read_buffer);
```

Allocate a data block for new data

```
char *buffer = “hello world\n”;
char *read_buffer[50];
int fd;
fd = open(“foo”,O_CREAT|O_RDWR,0600);
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen(“hello”));
printf(“%s\n”,read_buffer);
```

write() system call in the kernel

- **Step 1**: compute the block index and offset from the r/w pointer
  - Block index = 0 / 4K
  - Offset = 0 % 4K
- **Step 2**: If block address in i-node is 0 at the Block index
  - **Step 2A**: allocate a data block from the free list and record its block address at Block index entry of i-node
  - **Step 2B**: allocate memory buffer large enough to hold a data block
- **Step 3**: copy data from user buffer to Offset inside memory buffer holding data block
- **Step 4**: write i-node back to its location in the i-list
- **Step 5**: write memory buffer to data block address contained in Block index entry of i-node

Reset the r/w offset

```
char *buffer = "hello world\n";
char *read_buffer[50];
int fd;
fd = open("foo",O_CREAT|O_RDWR,0600);
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen("hello"));
printf("%s\n",read_buffer);
```

lseek() system call in the kernel

Step 1: find the open file table entry for the file from the file descriptor table entry indexed by the file descriptor

Step 2: set the r/w offset value to what is specified in the system call

NOT that FUSE implements the open file table for you so you do not need to implement lseek() for your file system.

Read *some* of the data back in

```
char *buffer = “hello world\n”;
char *read_buffer[50];
int fd;
fd = open(“foo”,O_CREAT|O_RDWR,0600);
write(fd,buffer,strlen(buffer));
lseek(fd,0,SEEK_POS);
read(fd,read_buffer,strlen(“hello”));
printf(“%s\n”,read_buffer);
```

read() system call in the kernel

- **Step 1**: compute the block index and offset from the r/w pointer
  - Block index = 0 / 4K
  - Offset = 0 % 4K
- **Step 2**: allocate a memory buffer large enough to hold a disk block in the kernel
- **Step 3**: read the disk block from the block address contained in the i-node at the Block index into memory buffer
- **Step 4**: copy data from Offset into memory buffer into user’s memory buffer specified in system call

Quick summary

- File system contains three components
  - Super block, i-list, data blocks
  - i-list can be fixed sized and scanned
  - Free data blocks are on free list stored in storage with head of free list in super block
  - All data blocks are either on free list or listed in a in-node
    - No data block is both
- open(), write(), and read() system calls access i-nodes and data blocks
  - Data for i-nodes and data blocks must be held in memory when being accessed
  - Data blocks allocated in write() when needed
- lseek() simply manipulates r/w pointer in open file table
- close() (not described) decrements reference counts and releases buffers as needed (e.g. when ref counts go to zero)

File names and directories

- In Linux a file name is a path in a tree starting from the tree's root
  - /home/rich/cs270/foo
- each element in the path (except maybe the last) names a directory
- Each *valid* name is contained in the directory before it in the path (except for the root)
  - "/" (called "root") contains "home"
  - "home" contains "rich"
  - "rich" contains "cs270"
  - "cs270" contains "foo"
- The last element in a path is one of three things
  - Directory
  - File
  - "Special" file (e.g. a device specifier in the directory /dev)

Directories are files with a specific structure

- Directories are files
  - All Linux files are represented by i-nodes
  - Every directory has an i-node
- Directories are files
  - They contain a map between human readable names and i-node numbers as data
  - All data in files is stored in data blocks
  - Every directory has one or more data blocks in its i-node

“text string”	i-node #
“text string”	i-node #
“text string”	i-node #
“text string”	i-node #
“text string”	i-node #
“text string”	i-node #

.
.
.

For example: ls -i

rich — rich@csilvm-01:~ — ssh csil.cs.ucsb.edu — 127×29

~ — smartfarm@euca-10-1-0-228:~ — ssh smartfarm@128.111.45.51 | ~ — rich@csilvm-01:~ — ssh csil.cs.ucsb.edu

```
Last login: Tue Oct  6 11:47:16 on ttys002
You have mail.
caprine:~ rich$ ssh csil.cs.ucsb.edu
rich@csil.cs.ucsb.edu's password:
Last failed login: Mon Oct  5 12:43:34 PDT 2020 from 54.37.150.194 on ssh:notty
There was 1 failed login attempt since the last successful login.
Last login: Mon Oct  5 10:39:29 2020 from 128.111.27.247


*********************************
*                               *
*            WELCOME            *
*                               *
*   This server reboots weekly  *
*     @ 5:00am Pacfic Time      *
*                               *
*********************************

rich@csilvm-01:~$ ls -i /
     16 bin        3 dev    655361 home        11 lost+found  2359297 opt      1008 run   3014657 srv   2883585 usr
      2 boot  393217 etc        12 lib    3145729 media             1 proc       15 sbin        1 sys   1835009 var
  21689 cs     19201 fs         14 lib64  2621441 mnt         1441793 root     3662 snap    57349 tmp
rich@csilvm-01:~$
```

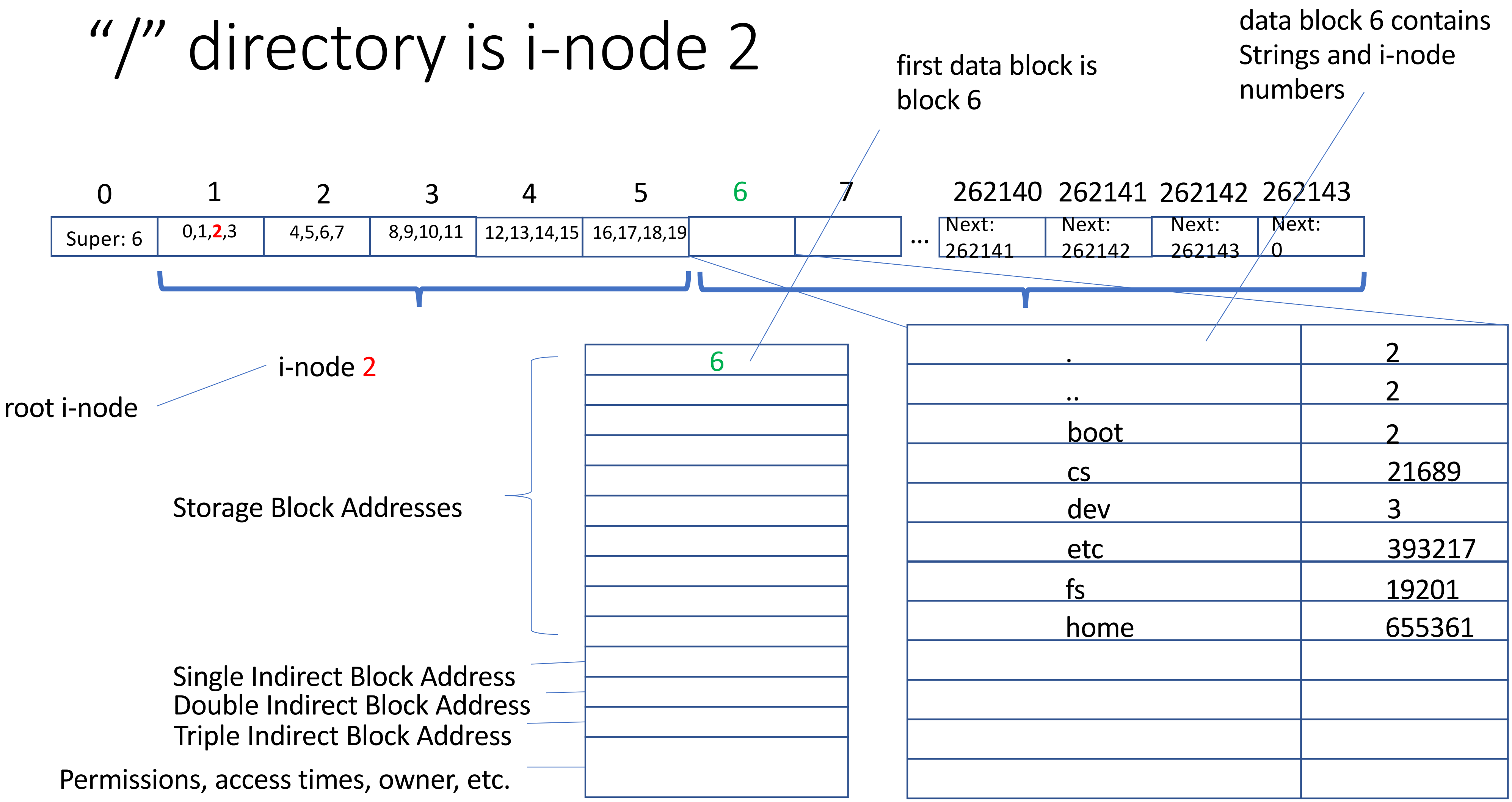
/ (the root directory) contains strings and i-node numbers

- rich@csilvm-01:~$ ls -ali /
- total 72
-       2 dr-xr-xr-x.   20 root root 4096 Sep 18 16:31 .
-       2 dr-xr-xr-x.   20 root root 4096 Sep 18 16:31 ..
-       2 dr-xr-xr-x.   6 root root 4096 Oct 5 10:22 boot
-   21689 drwxr-xr-x   5 root root   0 Oct 6 10:19 cs
-       3 drwxr-xr-x   21 root root 4000 Sep 18 16:31 dev
- 393217 drwxr-xr-x. 162 root root 12288 Oct 5 14:11 etc
-   19201 drwxr-xr-x   4 root root   0 Oct 5 16:06 fs
- 655361 drwxr-xr-x.   3 root root 4096 Sep 28 12:10 home
-

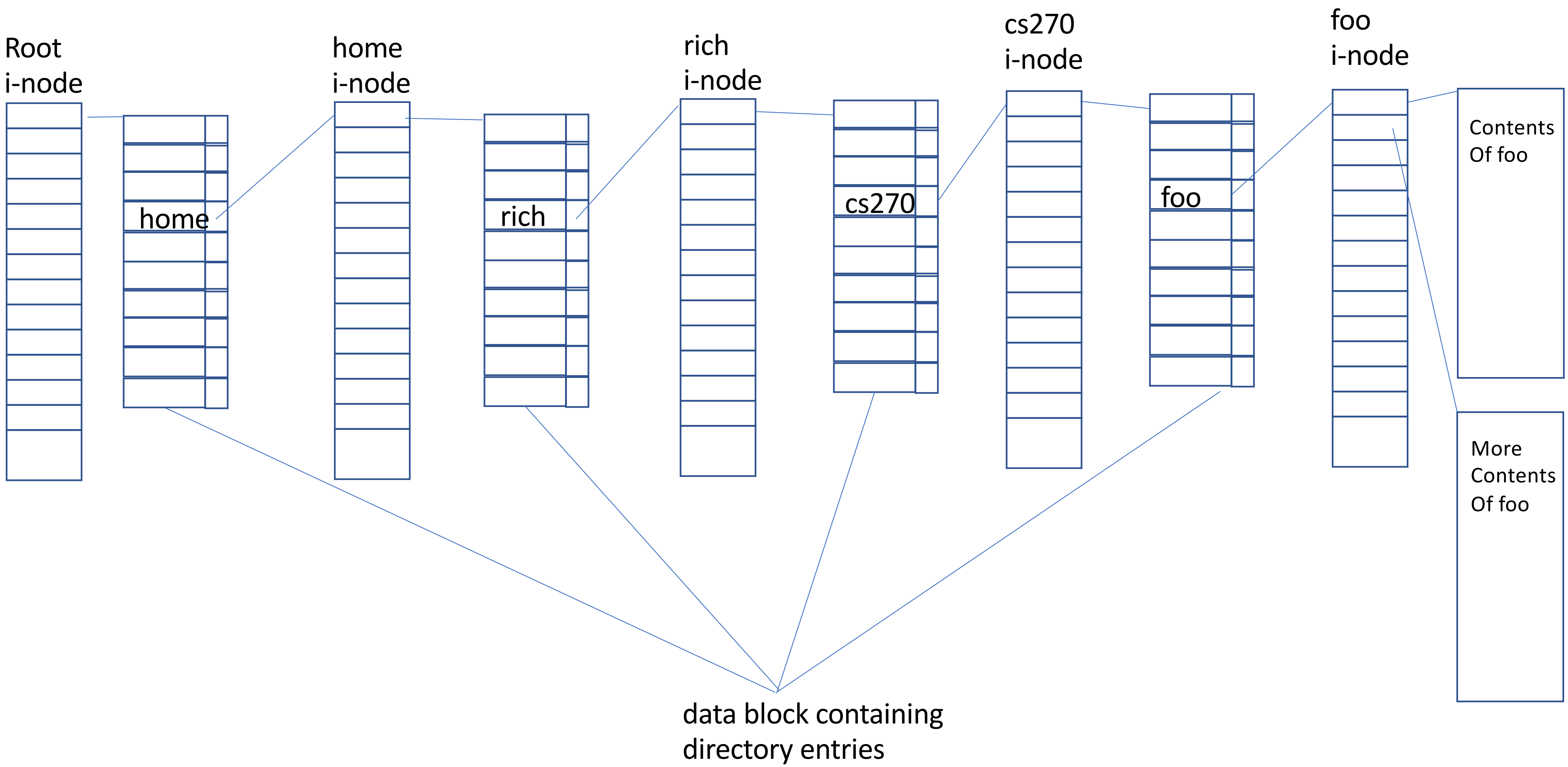
.	2
..	2
boot	2
cs	21689
dev	3
etc	393217
fs	19201
home	655361

.
.
.

“/” directory is i-node 2



/home/rich/cs270/foo



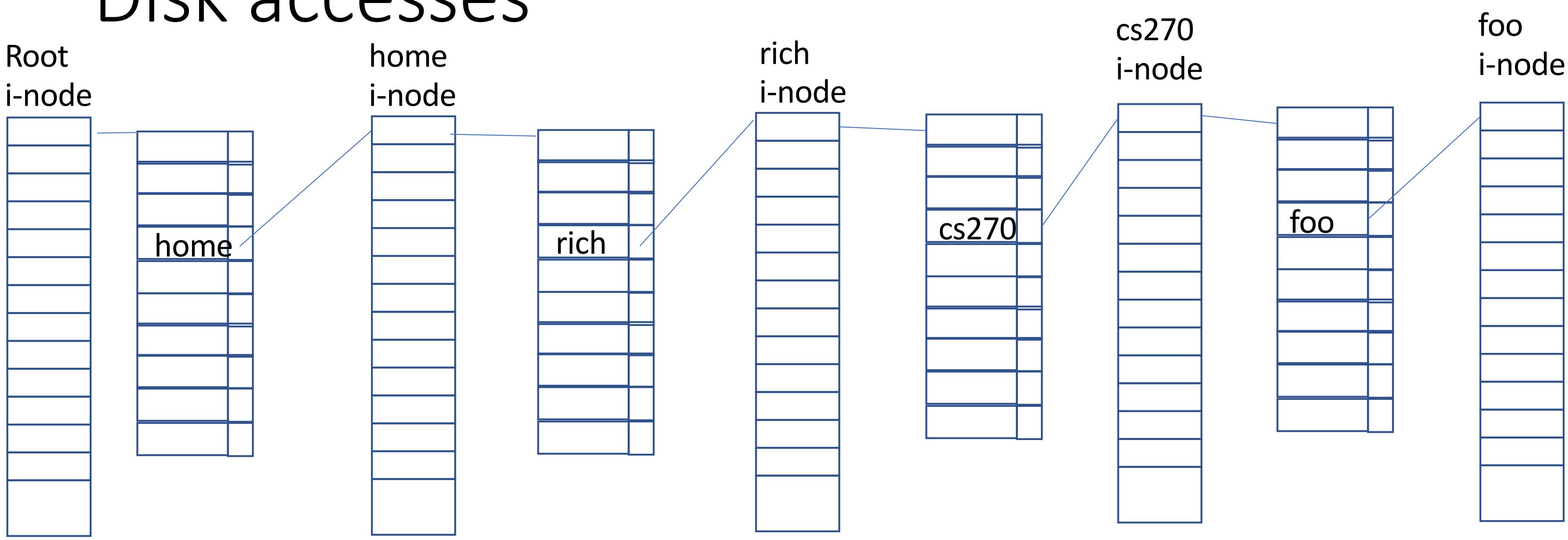
Find the i-node: namei

- All file system system calls (with the exception of lseek()) require the kernel to access the i-node associated with a file
  - open()
    - Checks permissions on the file which are in the i-node
    - Checks permissions (execute permission) in the directory if the file name is not explicit
  - read()/write()
    - Requires the i-node to get access to the data blocks
    - Changes the access times in the inode
  - chown/chmod/chgrp
    - Changes permissions and ownership in i-node
  - unlink()
    - Decrements a reference count and deletes the file if ref count is 0 (to be discussed)
- namei(): a kernel level routine
  - Find the i-node number associated associated with the file at the end of a path

Open /home/rich/cs270/foo

- fd = open("/home/rich/cs270/foo",O_RDONLY,0);
  - Read i-node for "/"
  - Read data block from first direct block in i-node for "/"
  - Scan data block for string "home"
  - Get i-node number for "home"
  - Read i-node for "home" with that number
  - Read data block from first direct block in i-node for "home"
  - Scan data block for string "rich"
  - Get i-node number for "rich"
  - Read i-node for "cs270"
  - Read data block from first direct block in i-node for "cs270"
  - Scan data block for string "foo"
  - Get i-node number for "foo"
  - Read i-node for "foo"
  - Check permissions on "foo"
  - Put i-node in in-core i-node table

Disk accesses



Thoughts on directories

- ***What happens if two different directory entries contain the same i-node #?***

- Hard link: two different names for the same file (the same i-node)

.	2
..	2
boot	2
cs	21689
dev	3
etc	393217
fs	19201
home	655361

Hard link /home/rich/cs270/foo to /home/shereen/shereenfoo

Data block for /home/rich/cs270

.	232
..	11987
c-code	87364
github	21689
foo	6654
goo	998765

Data block for /home/shereen

.	98776
..	763453
mail	874563
downloads	8846
documents	645243
shereenfoo	6654

Same i-node means same file with two different names
/home/rich/cs270/foo
/home/shereen/shereenfoo

Thoughts about hard links

- There is no file delete in Linux
  - unlink() system call removes a directory entry from a directory data block that contains a i-node #
  - When the last directory entry is unlinked
    - The data blocks are returned to the free list in the file system
    - The i-node is marked as “available” in the i–list
- i-nodes must carry a reference count on disk
  - The ref count inside the on-disk i-node counts how many directory entries refer to this i-node
  - **NOTE** that this is different than the reference count for an in-core i-node that the kernel keeps when a file is open
- Hard links can only be made between directories and files within the same file system
  - ***Why?***

Multiple File Systems

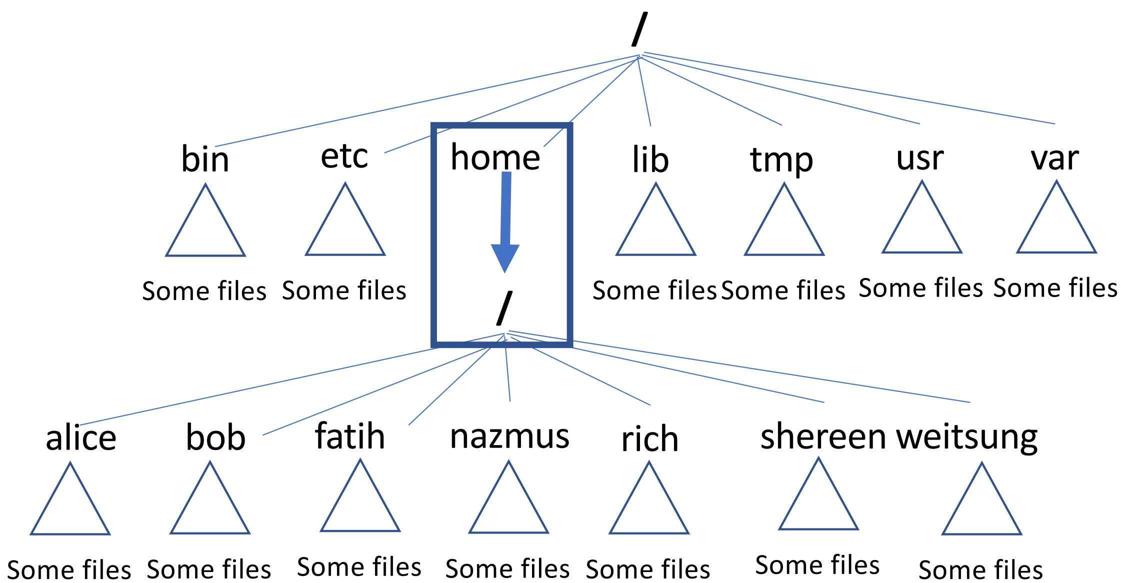
- Recall that a file system has
  - A super block
  - An i-list
  - A set of data blocks
- All of these must reside in the same partition (i.e. on the same device)
- Computers can have more than one storage device
  - Additional capacity
  - Performance
  - Removable media
- ***How does Linux configure multiple storage devices?***

Stitching together file systems

- Each device has one or more partitions
- Each partition contains its own file system
  - Every file exists in exactly one file system in one partition
- Every file is named by a unique path from the root (from "/")
- ***How does Linux create one name "tree" from multiple file systems?***

mount

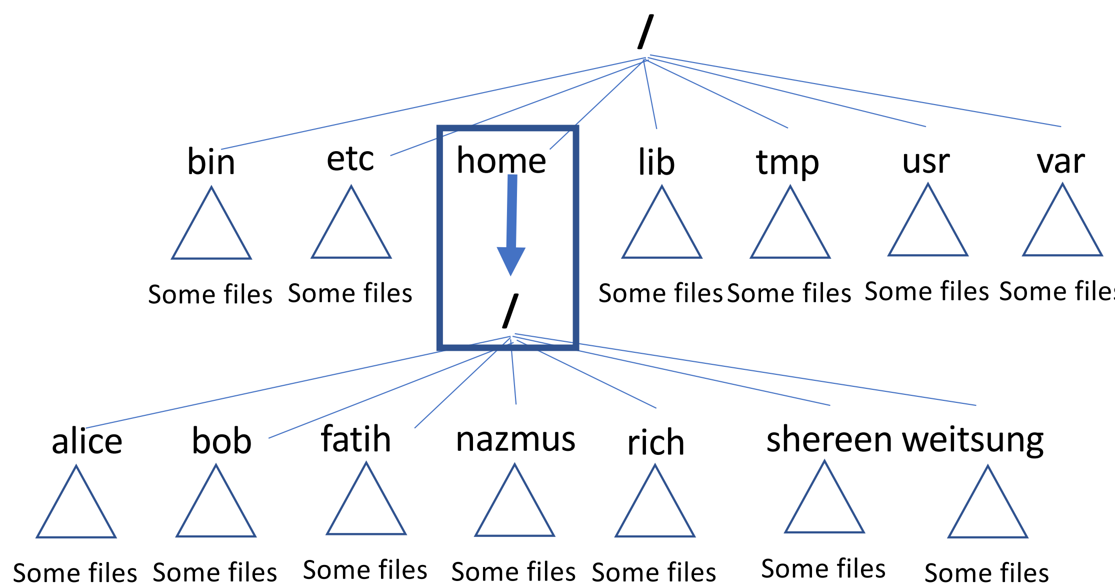
- The mount utility creates an equivalence between a *leaf* in one file system and a *root* of another in the name tree.



- The root of the file system containing home directories “lays over” the directory “home” in the “root” file system for the system

Mounting and a mounted file system

- "home" is an empty directory in the tope level "root" file system
  - Called a "mount point"
- After the second file system (lower) is mounted "on" the first (upper), *namei() will change file systems* and start at the new root at the mount point



Which file system?

- Recall that directories contain strings and i-node numbers
  - But not file system identifiers!
- For example, every file system has an i-node #7
  - ***When "7" appears in a directory entry, which file system is it in?***
  - **NOTE** that hard links create multiple names for the same i-node # => needs to be in the same file system
- The kernel contains a table that identifies mounted file systems for namei()

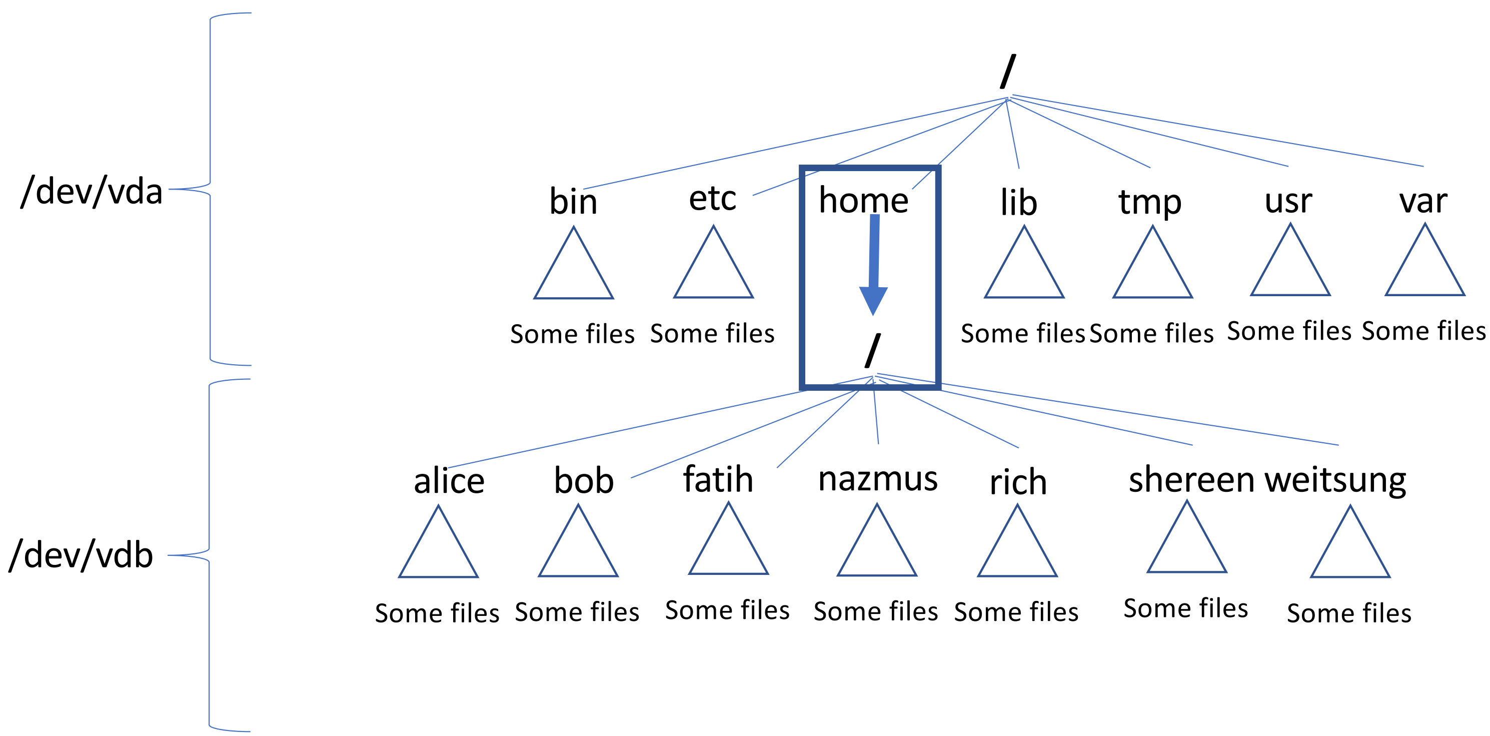
The mount table

- Kernel table that shows the mapping of file systems to mount points

device	Mount point	File system type	parameters
/dev/vda	/	xfs	rw,relatime,attr2,inode64,noquota
/dev/vdb	/home	ext4	rw,relatime

- Each partition is represented by a “special” file in the /dev directory
  - Often termed a device
- In the mount table, a device is assumed to contain a file system
  - Can be read and written as full 4K blocks addressed starting with block 0
- When namei() scans a data block and finds a string in a path, it checks the mount table to see if it is a mount point
  - Subsequent i-node numbers come from the file system specified in the mount table until name-I encounters another mount point.

For example



/dev/vda
/dev/vdb
/
bin
etc
home
lib
tmp
usr
var
Some files
Some files
Some files
Some files
Some files
Some files
/
alice
bob
fatih
nazmus
rich
shereen
weitsung
Some files
Some files
Some files
Some files
Some files
Some files
Some files

Summarizing

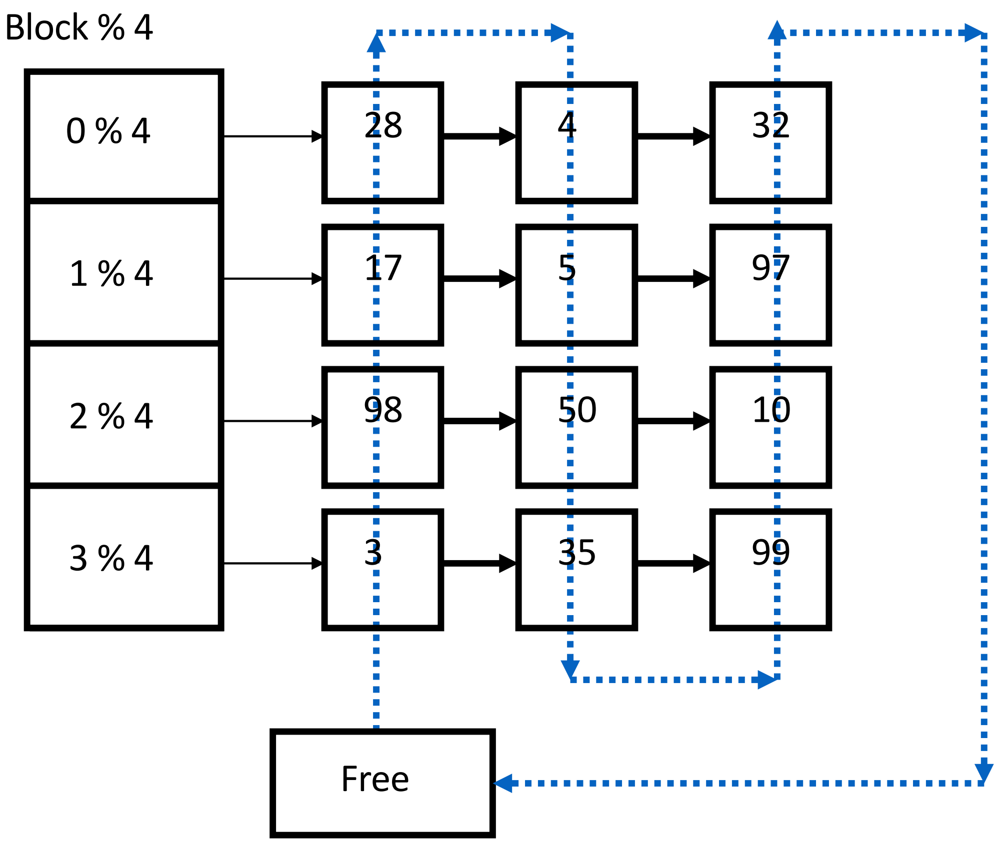
- i-nodes map data blocks to files and carry meta-data for each file
- File systems contain a supe block, an i-list of i-nodes, and data blocks
- Data blocks are either on the free list or referenced in an i-node
- File descriptors reference r/w pointer in open file table which references i-node in memory
    - Reference counts for sharing
- Directories map strings to i-node # in a path from root
- namei() resolves paths to i-nodes
- Mount allows multiple file systems to form a name “tree” where every file or directory has a unique path from the root

Performance

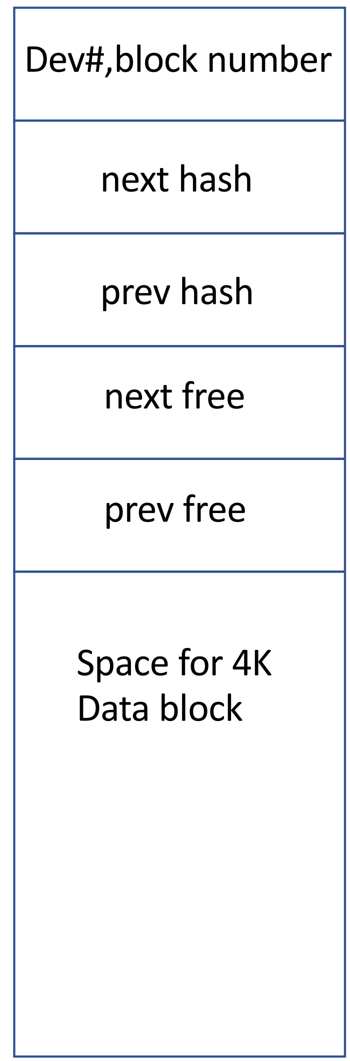
- Consider the following:
  - 3.3 GHz x86 can do 1 instruction every 10^-9 seconds (ballpark)
  - SSD/spinning disk can read/write a block every 10^-3 seconds
- CPU is 10^6 (1,000,000) x faster than persistent storage
- When CPU does a read/write to disk, it must stop and wait for the interrupt before it “knows” the i/o has completed
- Imagine the clock speed was 1Hz (1 instruction / second)
  - ***How long would the CPU wait for a disk access?***
  - ***Answer: 11.5 days***

The Buffer Cache

- All disk I/O is in blocks
- Cache of blocks
  - Hash list
  - Hash (dev#||block #)
- LFU free list
  - After a block is used, it stays in the cache
  - Moves to the tail of the free list

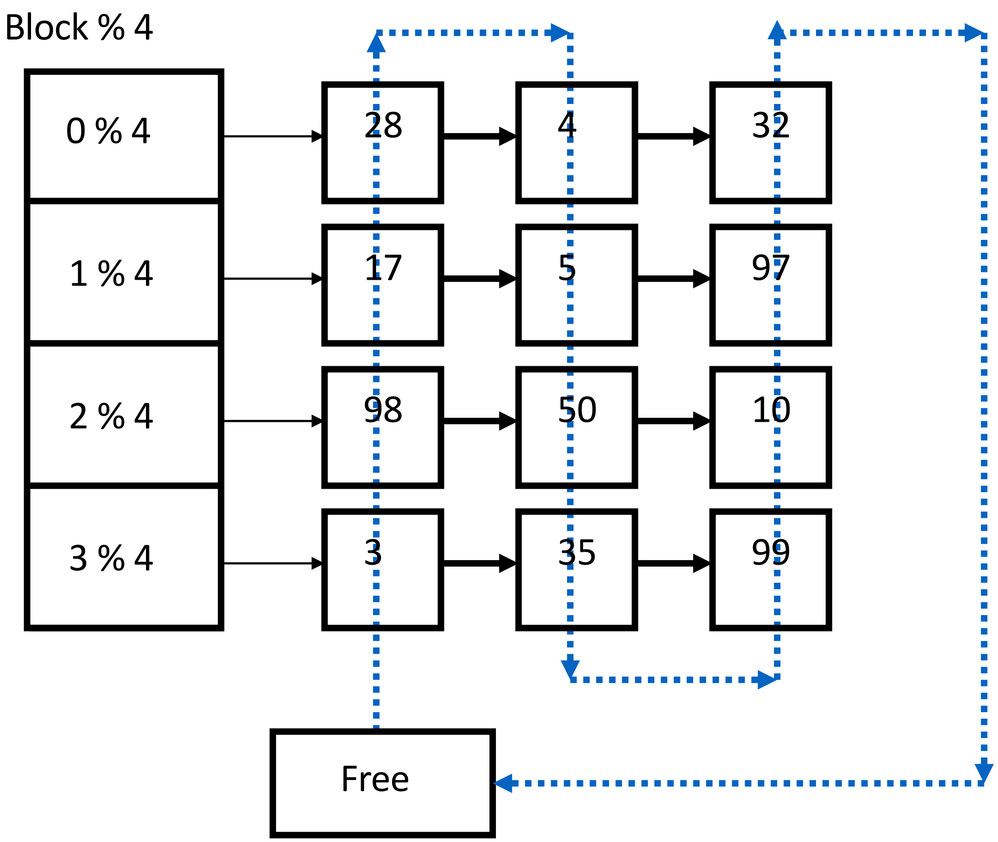


Buffer cache entry



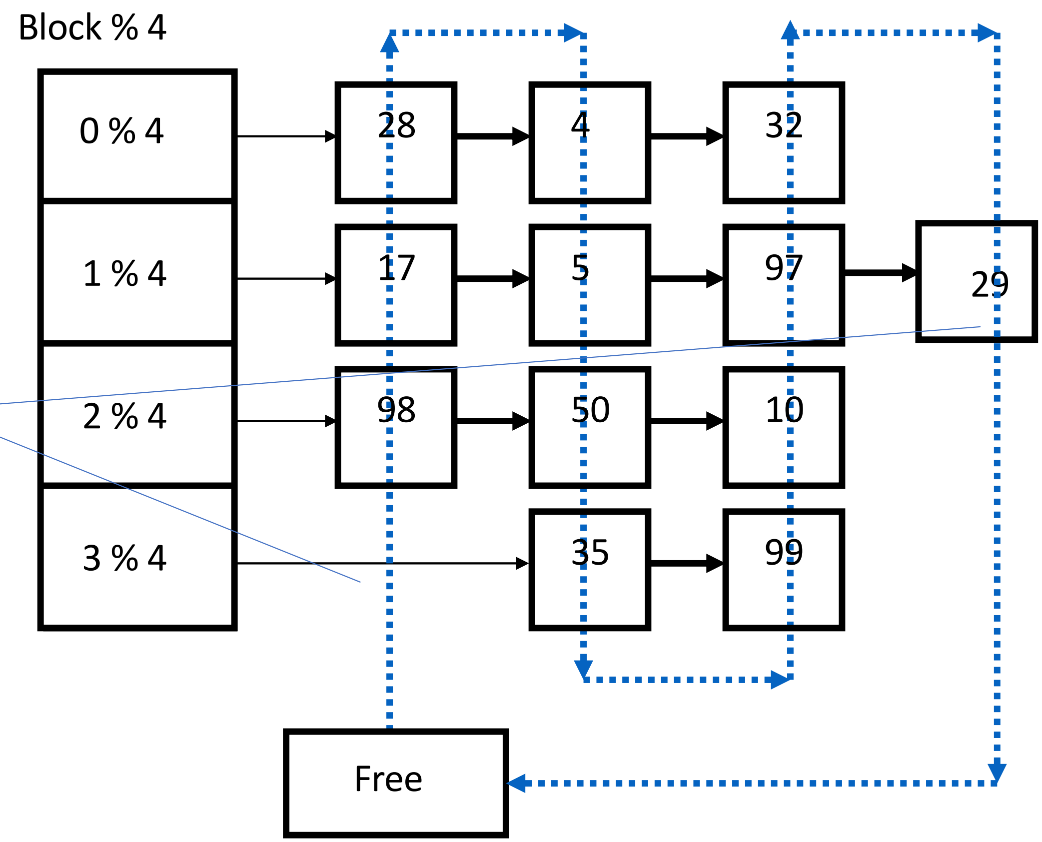
Cache miss and cache hit

- Read/Write Block 29, dev 0
- Miss: block 29, dev 0 not in hash
- Head of free list is block 3
- Steps
  - Remove block 3 from buffer cache
  - Use buffer cache entry for block 29
  - Add to hash list
  - Use the buffer
  - Put new entry at end of free list



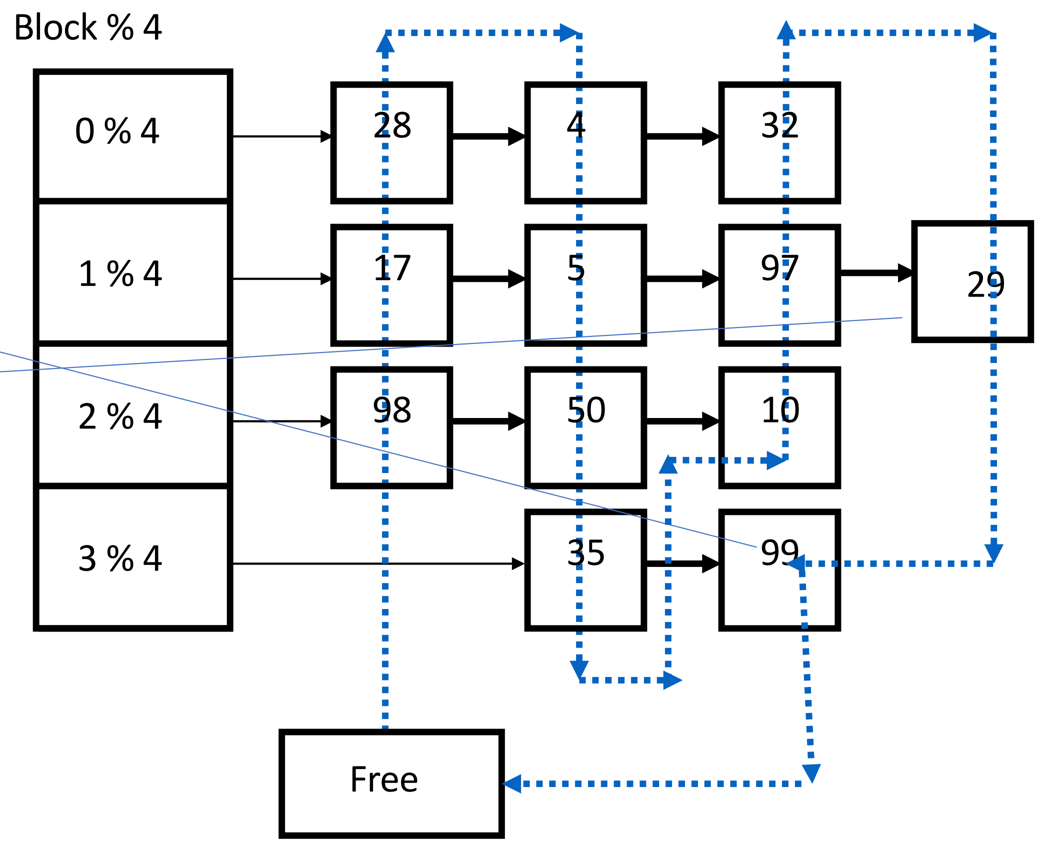
Cache miss

- Steps
  - Remove block 3 from buffer cache
  - Use buffer cache entry for block 29 => (0||29 %4) = 1
  - Add to hash list
  - Use the buffer to transfer data to/from user space
  - Put new entry at end of free list



Cache hit

- Read block 99, dev 0
- (0||99)%4
- Steps
  - Remove block 99 from free list
  - Use the buffer to transfer data to/from user space
  - Put 99 at end of free list



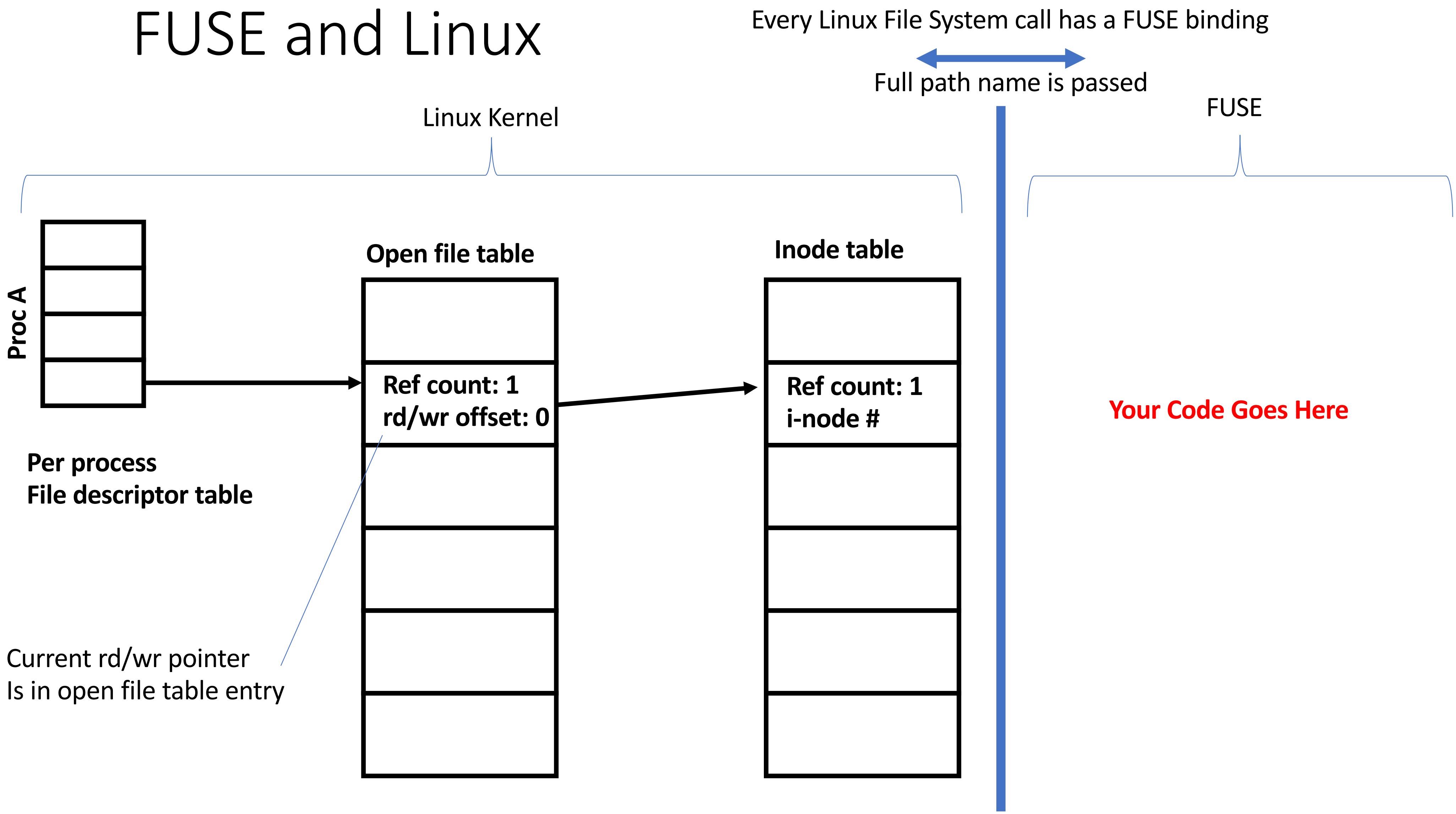
Thoughts on the buffer cache

- Three Phase process on a read or write
  - Phase 1
    - Find the block in the hash list
    - Miss: remove the head of the free list and remove the block in that entry from hash list
    - Hit: Remove it from the free list where ever it is (since it is in use)
  - Phase 2
    - Use the buffer
      - Read data from disk and/or transfer data from user space as part of read/write call
      - Read an i-node block from the i-list and copy the i-node into in-core i-node table
  - Phase 3
    - Put the block at the tail of the free list
- LFU: blocks in the hash list move toward the tail as they are used
  - Head is the least frequently used block
- Sizing the buffer cache => tricky since it is pinned down memory in the kernel

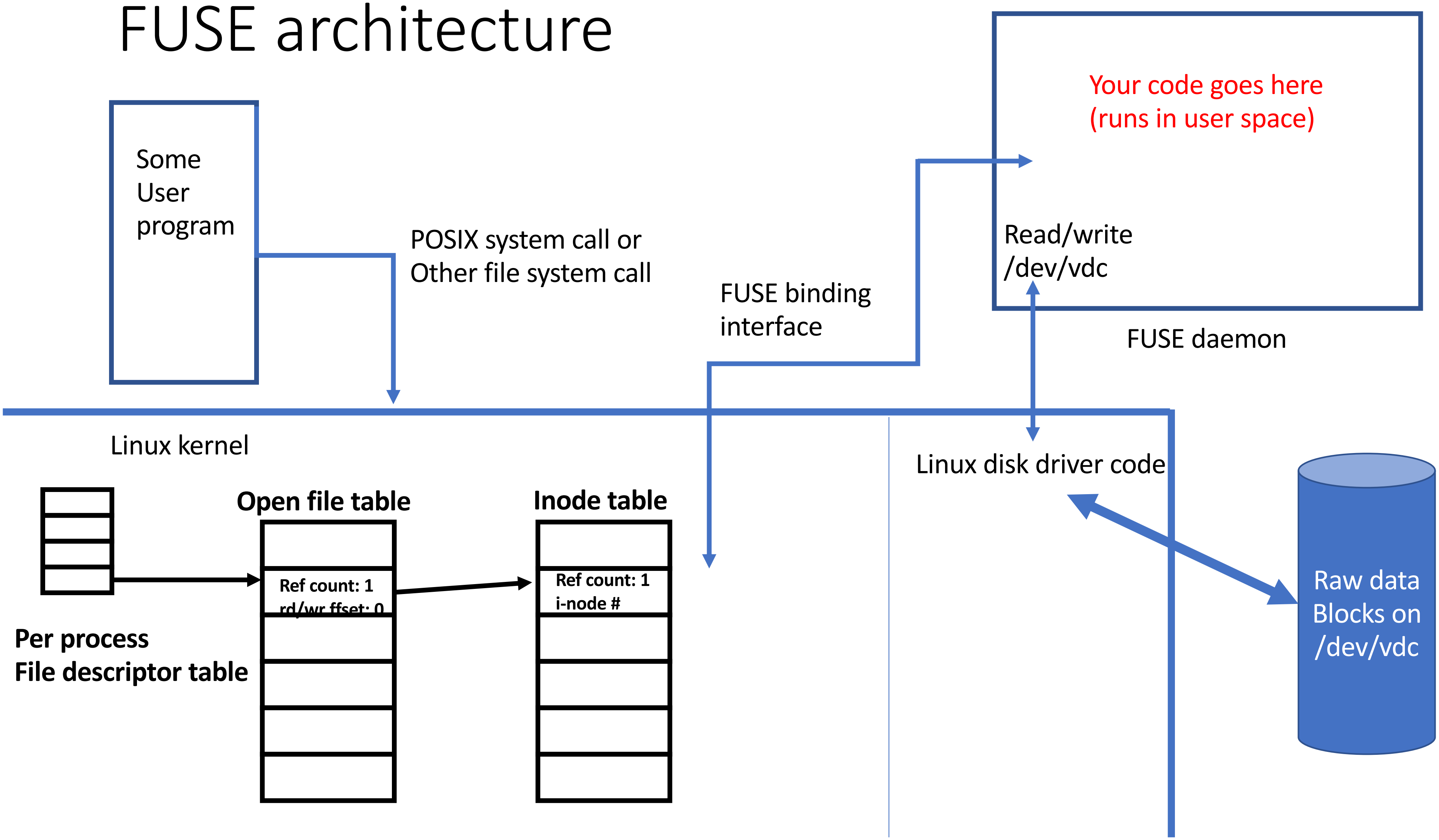
CS270 File Systems Project with FUSE

- FUSE is a recognized file system type
  - Linux will mount a FUSE file system and create a mount table entry for it
  - When Linux namei() traverses the mount point for FUSE, your FUSE code will be invoked
- FUSE “cuts in” to Linux after the i-node table

FUSE and Linux



FUSE architecture



What you need to develop

- You write the code for each file system call and bind it with the FUSE bindings
- You DO NOT need to implement the r/w pointer
  - FUSE will pass you a full path name and a file offset on each FUSE binding call
- You DO NOT need to implement a the mount table
  - Linux will take care of mount points
- You SHOULD implement namei()
  - FUSE has a way for you to access the Linux i-nodes but it is tricky
  - Simpler to create your own i-nodes and to manage them separately
  - Your namei() does not need to check a mount table
- You MUST read and write the raw disk device in 4K blocks