CS111, Lecture 3

Unix V6 Filesystem

😷 masks strongly recommended

Announcements

- Sections start in person this week! Check the course website for your section assignment. Bring a laptop with you if you have one.
  - You can change to any section with space available
  - If you are e.g., sick, you can attend another section as a guest that week, but please email the section TA to confirm there is space
  - If you have exceptional circumstances that prevent you from attending any section during a given week, please email the instructor
  - Sections rely on material through each Wed. lecture
- Lecture credit starts with today's lecture – you can submit PollEV polls in person during live lecture, or complete the corresponding Canvas quiz by Wed 11:30AM, to get credit.
  - Update to misses policy; changing "2 pre-excused misses" to "drop 2 lowest scores"
- Assign0 due tonight at 11:59PM, late deadline Wed.
- assign1 released by 6PM today (assign1 and assign3 are longer than 0 and 2)

CS111 Topic 1: Filesystems

Key Question: *How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?*

Filesystems introduction and design	→	Case study: Unix V6 Filesystem	→	Filesystem System calls and file descriptors	→	Crash recovery
Lecture 2		**Today / Lecture 4**		**Lecture 5**		**Lectures 6-7**

assign1: implement portions of the Unix v6 filesystem!

Learning Goals

- Explore the design of the Unix V6 filesystem
- Understand how we can use inodes to store and access file data
- Learn about how inodes can accommodate small and large files

Plan For Today

- **Recap**: filesystems so far
- The Unix V6 Filesystem and Inodes
- **Practice**: reading file data
- Large files and Singly-Indirect Addressing
- **Practice**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing
- Assignment 1

Plan For Today

- **Recap: filesystems so far**
- The Unix V6 Filesystem and Inodes
- **Practice**: reading file data
- Large files and Singly-Indirect Addressing
- **Practice**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing
- Assignment 1

Recap: Filesystems So Far

A **filesystem** is the portion of the OS that manages the disk (persistent storage). The disk only knows how to read a sector and write a sector.

- Blocks are the storage unit used by the filesystem, can be 1 or more sectors.

Designs we've discussed so far:

- *Contiguous allocation* allocates a file in one contiguous space
- *Linked files* allocates files by splitting them into blocks and having each block store the location of the next block.
- *Windows FAT* is like linked files but stores the links in a "file allocation table" in memory for faster access.
- *Multi-level indexes* instead store all block numbers for a file together so we can quickly jump to any point in the file (but how?). Example: Unix v6 Filesystem
- Many other designs possible – many use a tree-like structure

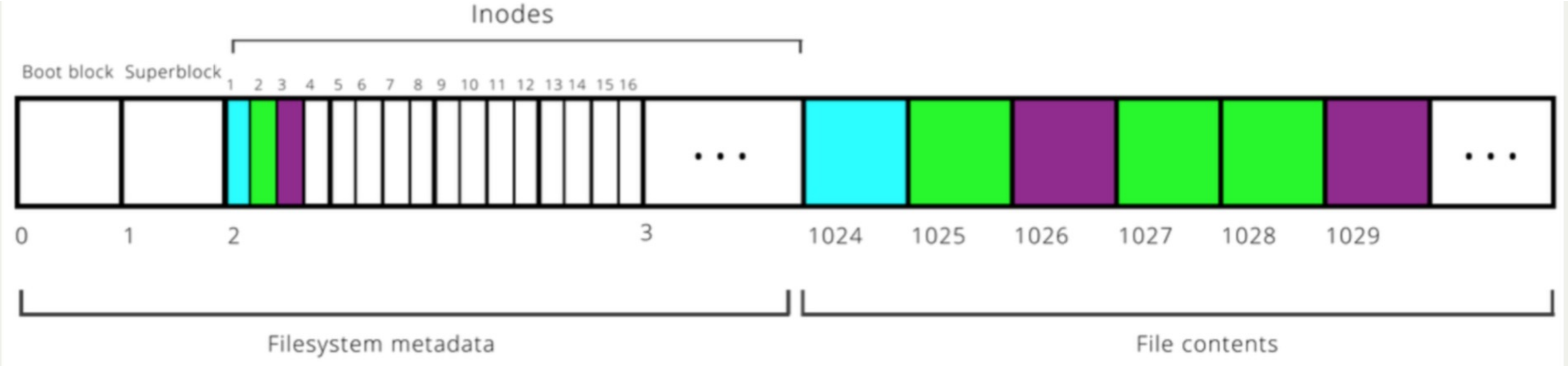
Filesystem Designs

- **Internal Fragmentation**: space allocated for a file is larger than what is needed. A file may not take up all the space in the blocks it's using. E.g. block = 512 bytes, but file is only 300 bytes. (you could share blocks between multiple files, but this gets complex)
- **External Fragmentation (issue with contiguous allocation)**: no single space is large enough to satisfy an allocation request, even though enough aggregate free disk space is available
- Wait, how do we look up / find files? (we'll talk more about this!)

Plan For Today

- **Recap**: filesystems so far
- **The Unix V6 Filesystem and Inodes**
- **Practice**: reading file data
- Large files and Singly-Indirect Addressing
- **Practice**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing
- Assignment 1

Unix V6 Inodes



- An **inode** ("index node") is a grouping of data about a single file.
- The Unix v6 filesystem stores inodes on disk together in a fixed-size **inode table**. Each inode lives on disk, but we can read one into memory when a file is open.
- The inode table starts at block 2 (block 0 is "boot block" containing hard drive info, block 1 is "superblock" containing filesystem info). The inode table can span many blocks. Typically, at most 10% of the drive stores metadata.
- Inodes are 32 bytes big, and 1 block = 1 sector = 512 bytes, so 16 inodes/block.
- Filesystem goes from **filename** to **inode number** ("inumber") to **file data**.

Unix V6 Inodes

Each inode stores file information and has space for 8 block numbers.

```
struct inode {
  uint16_t  i_mode;     // bit vector of file
                        //   type and permissions
  uint8_t   i_nlink;    // number of references
                        //   to file
  uint8_t   i_uid;      // owner
  uint8_t   i_gid;      // group of owner
  uint8_t   i_size0;    // most significant byte
                        //  of size
  uint16_t  i_size1;    // lower two bytes of size
                        //  (size is encoded in a
                        //  three-byte number)
  uint16_t  i_addr[8];  // device addresses
                        //   constituting file
  uint16_t  i_atime[2]; // access time
  uint16_t  i_mtime[2]; // modify time
};
```

For now, we just need **i_addr**; an array of 8 block numbers. **i_addr** entries are in <u>file data order</u>, not necessarily numerical order - the blocks could be scattered all over disk. E.g. a file could have i_addr = [12, 200, 56, …].

Unix V6 Inodes

Let's imagine that the hard disk creators provide software to let us interface with the disk.

```
void readSector(size_t sectorNumber, void *data);
void writeSector(size_t sectorNumber, const void *data);
```

*(Refresher: size_t is an unsigned number, void * is a generic pointer)*

How can we get the first 16 inodes from disk?

Reading First 16 Inodes From Disk

```
char buf[DISKIMG_SECTOR_SIZE];
readSector(2, buf); // always reads in 512 bytes

// now buf is filled with 512 bytes from block 2
// but it’s an array of chars…
```

A Better Way

```
struct inode {
  uint16_t  i_addr[8];  // block numbers
  ...
};

...

int inodesPerBlock = DISKIMG_SECTOR_SIZE / sizeof(struct inode);
struct inode inodes[inodesPerBlock];
readSector(2, inodes);

// Loop over each inode in sector 2
for (size_t i = 0; i < sizeof(inodes) / sizeof(inodes[0]); i++) {
    printf("%d\n", inodes[i].i_addr[0]); // print first block num
}
```

Plan For Today

- **Recap**: filesystems so far
- The Unix V6 Filesystem and Inodes
- **Practice: reading file data**
- Large files and Singly-Indirect Addressing
- **Practice**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing

Practice #1: Inodes

Let's say we have an inode that looks like the following (remember 1 block = 1 sector = 512 bytes):

<u>Inode:</u>

size = 600 bytes

i_addr = [122, 56, X, X, X, X, X, X]   *(“X” means garbage/unused)*

- How many bytes of block 122 store file payload data?
- How many bytes of block 56 store file payload data?

Bytes 0-511 (512 bytes) reside within block 122, bytes 512-599 (88 bytes) within block 56.

Practice #2: Inodes

Let's say we have an inode that looks like the following (remember 1 block = 1 sector = 512 bytes):

Inode:

size = 2000 bytes

i_addr = [56, 122, 45, 22, X, X, X, X] *("X" means garbage/unused)*

Which block number stores the index-1500th byte (0-indexed) of the file?

Respond on PollEv: pollev.com/cs111fall23 or text CS111FALL23 to 22333 once to join.

With file size = 2000 bytes and block numbers = [56, 122, 45, 22], which block number stores the index-1500th (0-indexed) byte of the file?

0

Practice #2: Inodes

Let's say we have an inode that looks like the following (remember 1 block = 1 sector = 512 bytes):

<u>Inode:</u>

size = 2000 bytes

i_addr = [56, 122, 45, 22, X, X, X, X] *("X" means garbage/unused)*

Which block number stores the index-1500th byte (0-indexed) of the file?

Bytes 0-511 reside within block 56, bytes 512-1023 within block 122, bytes 1024-1535 within <u>block 45</u>, and bytes 1536-1999 at the front of block 22.

Plan For Today

- **Recap**: filesystems so far
- The Unix V6 Filesystem and Inodes
- **Practice**: reading file data
- **Large files and Singly-Indirect Addressing**
- **Practice**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing
- Assignment 1

File Size

Problem: with 8 block numbers per inode, the largest a file can be is 512 * 8 = 4096 bytes (~4KB). That definitely isn't realistic!

Assuming that the size of an inode is fixed, what can we do?

Solution: Unix V6 has two inode “modes”: *small* and *large*, that dictate how it uses i_addr.

```
if ((inode.i_mode & ILARG) != 0) { // inode is “large mode”
```

A “small mode” inode stores 8 block numbers, just as we have seen already. But what is a ”large mode” inode?

“Large Mode” Inodes

Let’s say a file’s data is stored across 300 blocks:

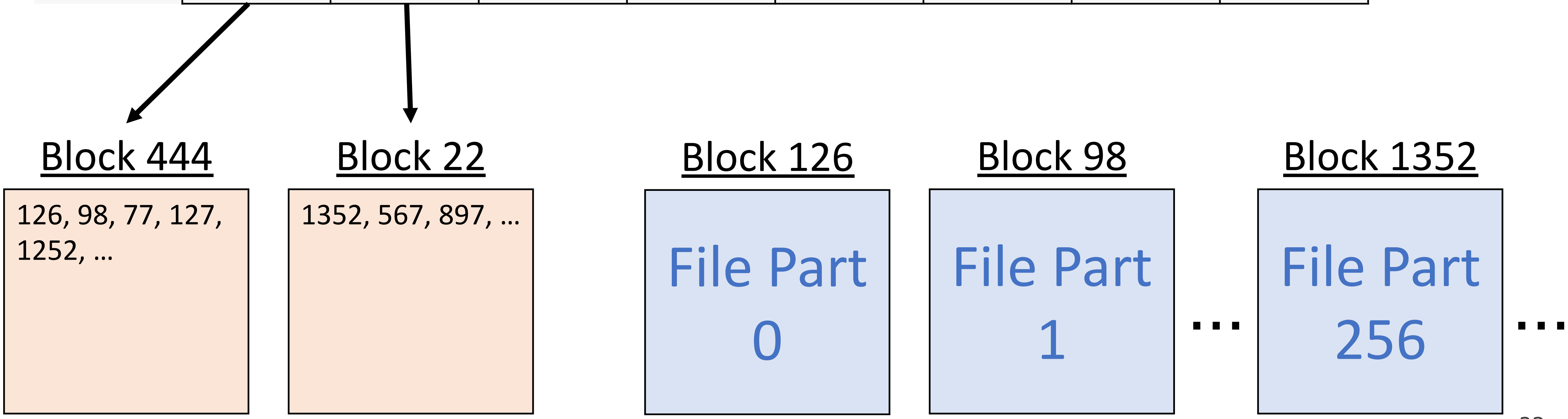
126, 98, 77, 127, 1252*, 377, 81, 48, 198, 409, 150, 105, 110, 143, 338, 382, 173, 149, 178, 423…*

Key idea: the inode only fits 8. So let's store each group of 256 (512 / 2-byte-numbers) block numbers *in a block*, and then store *that* block's number in the inode! This approach is called *singly-indirect addressing*.

Singly-Indirect Addressing

If the inode is "large mode", **i_addr** stores 7 numbers of blocks that contain block numbers, and those block numbers are of blocks that contain payload data. 8th block number? we'll get to that :)

index	0	1	2	3	4	5	6	7
i_addr	444	22	…	…	…	…	…	???



“Large Mode” Inodes

The Unix V6 filesystem uses *singly-indirect addressing* (blocks that store payload block numbers) just for “large mode” files.

- check flag in inode to know whether it is a “small mode” file (direct addressing) or “large mode” one (indirect addressing)
  - If **small**: all 8 block numbers are direct block numbers (block numbers of blocks that store file data)
  - If **large: first 7 block numbers** are singly-indirect block numbers (block numbers of blocks that store direct block numbers), and 8th is TBD ☺

Indirect Addressing

Let's assume for now that an inode for a large file uses all 8 block numbers for singly-indirect addressing. What is the largest file size this supports? Each block number is 2 bytes big.

8 block numbers in an inode x

256 block numbers per singly-indirect block x

512 bytes per block

= ~1MB

Indirect Addressing: Design Decisions

We could use singly-indirect addressing in other ways, too – all design decisions!

- We could make *just some of* the block numbers in an inode use singly-indirect addressing, and others still be direct addressing
- We could have just one “mode” for files, instead of 2

One argument against indirect addressing: it takes more steps to get to the data and uses more blocks.

Block 450	*Block 451*
451, 42, 15, 67, 125, 665, 467, 231, 162,136	The quick brown fox jumped over the...

Practice: Indirect Addressing

Let's say we have a "large mode" inode with the following information (remember 1 block = 1 sector = 512 bytes, and block numbers are 2 bytes big):

<u>Inode:</u>

size = 200,000 bytes

i_addr = [56, 122, X, X, X, X, X, X] *("X" means garbage/unused)*

Which singly-indirect block stores the block number holding the index-150,000th byte of the file?

*Bytes 0-131,071 reside within blocks whose block numbers are in block 56. Bytes 131,072 (256*512) - 199,999 reside within blocks whose block numbers are in block 122.*

Even Larger Files

Problem: even with singly-indirect addressing, the largest a file can be is 8 * 256 * 512 = 1,048,576 bytes (~1MB). That still isn't realistic!

Solution: let's have the 8th entry in **i_addr** use *doubly-indirect addressing*; store a block number for a block that contains *singly-indirect block numbers*.

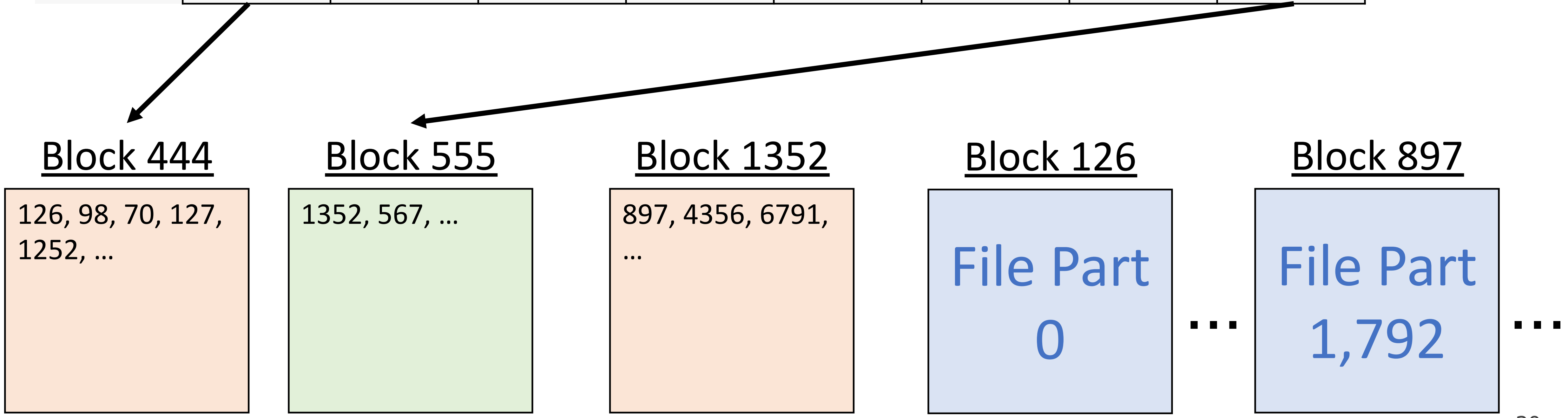
Plan For Today

- **Recap**: filesystems so far
- The Unix V6 Filesystem and Inodes
- **Practice**: reading file data
- Large files and Singly-Indirect Addressing
- **Practice**: singly-indirect addressing
- **Large files and Doubly-Indirect Addressing**
- Assignment 1

Large File Scheme

If the file is large, the first 7 entries in **i_addr** are singly-indirect block numbers. The 8th entry (if needed) is a doubly-indirect block number; the number of a block that contains singly-indirect block numbers.

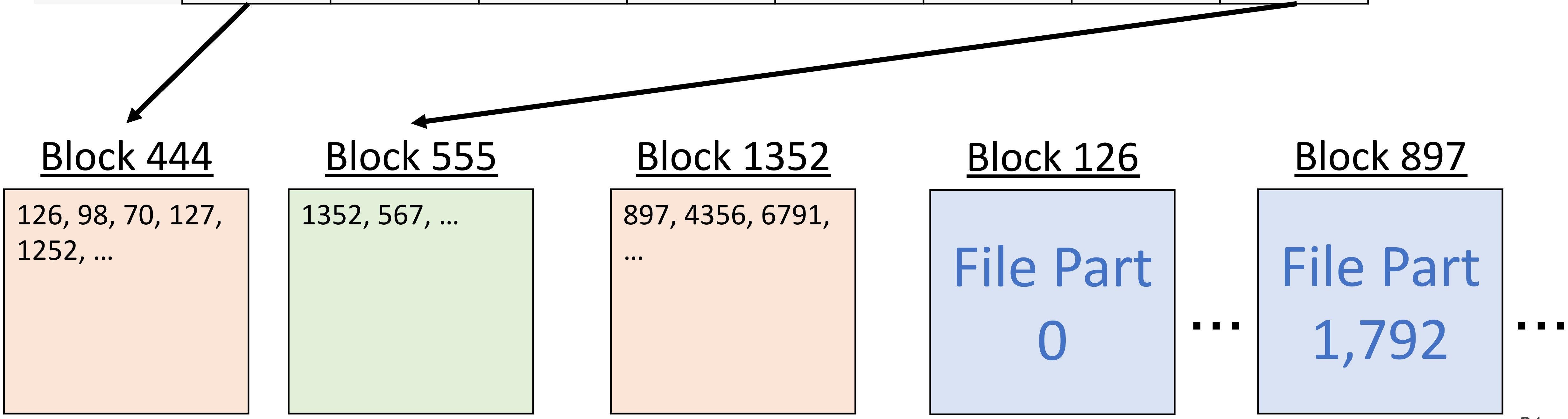
index	0	1	2	3	4	5	6	7
i_addr	444	22	34	792	168	976	2467	555



Large File Scheme

Another way to think about it: a file can be represented using at most 7 + 256 = 263 singly-indirect blocks. The first seven are stored in the inode. The remaining 256 are stored in a block whose block number is stored in the inode.

index	0	1	2	3	4	5	6	7
`i_addr`	444	22	34	792	168	976	2467	555



Indirect Addressing

An inode for a large file stores 7 singly-indirect block numbers and 1 doubly-indirect block number. What is the largest file size this supports? Each block number is 2 bytes big.

(7+256) singly-indirect block numbers total x

256 block numbers per singly-indirect block x

512 bytes per block

= ~34MB

Indirect Addressing

An inode for a large file stores 7 singly-indirect block numbers and 1 doubly-indirect block number. What is the largest file size this supports? Each block number is 2 bytes big.

OR:

*(7 * 256 * 512) + (256 * 256 * 512) ~ 34MB*

(singly indirect) + (doubly indirect)

Better! still not sufficient for today's standards, but perhaps in 1975. Moreover, since block numbers are 2 bytes, we can number at most 2^{16} - 1 = 65,535 blocks, meaning the entire filesystem can be at most 65,535 * 512 ~ 32MB.

Plan For Today

- **Recap**: filesystems so far
- The Unix V6 Filesystem and Inodes
- **Practice**: reading file data
- Large files and Singly-Indirect Addressing
- **Practice**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing
- **Assignment 1**

Assignment 1

- Assignment 1 released by 6PM today
- Implement core functions to read from a Unix v6 filesystem disk!
  - **inode_iget** -> fetch a specific inode
  - **inode_indexlookup** -> fetch a specific payload block number
  - **file_getblock** -> fetch a specified payload block
  - **directory_findname** -> fetch directory entry with the given name
  - **pathname_lookup** -> fetch inumber for the file with the given path

Assignment 1

- Assignment 1 released by 6PM today
- Implement core functions to read from a Unix v6 filesystem disk!
  - **inode_iget** -> fetch a specific inode
  - **inode_indexlookup** -> fetch a specific payload block number
  - **file_getblock** -> fetch a specified payload block
  - **directory_findname** -> fetch directory entry with the given name
  - **pathname_lookup** -> fetch inumber for the file with the given path

can start immediately (inode_iget, inode_indexlookup, file_getblock)

will discuss next time! (directory_findname, pathname_lookup)

Plan For Today

- **<u>Recap</u>**: filesystems so far
- The Unix V6 Filesystem and Inodes
- **<u>Practice</u>**: reading file data
- Large files and Singly-Indirect Addressing
- **<u>Practice</u>**: singly-indirect addressing
- Large files and Doubly-Indirect Addressing
- Assignment 1

Lecture 3 takeaway: The Unix v6 filesystem represents small files by storing direct block numbers, and larger files by using indirect addressing - storing 7 singly-indirect and 1 doubly-indirect block number.

Next time: directories, file lookup and links