CSc 352

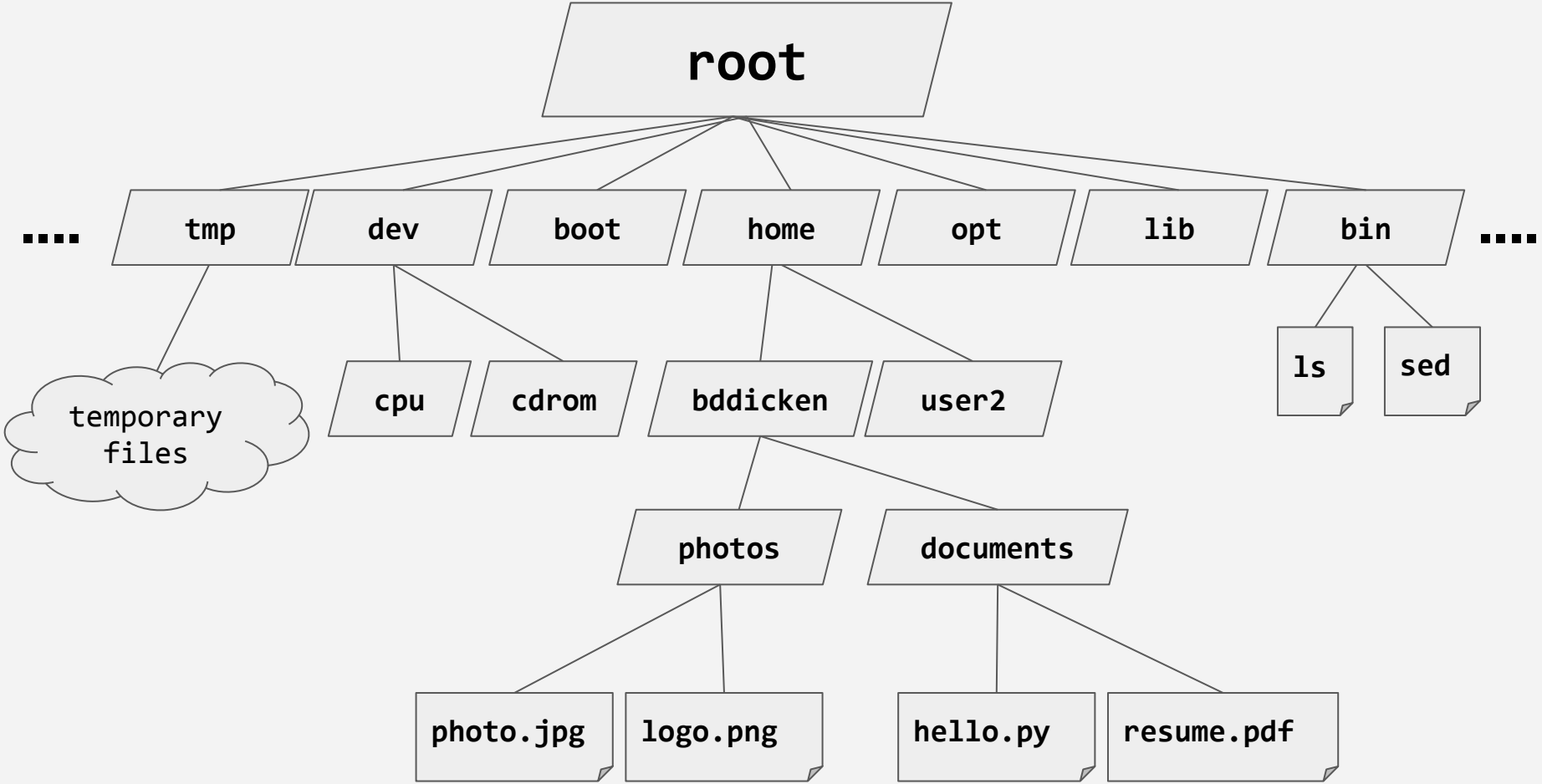
C Programming files

Benjamin Dicken

The UNIX File System

- The file system is a core component to a UNIX operating system
- There are different specific implementations, but there are shared general-principles
  - UFS, EXT2, EXT3, EXT4, ZFS, etc, etc
- We will focus on the general principles

https://en.wikipedia.org/wiki/Unix_filesystem


root
....
tmp
dev
boot
home
opt
lib
bin
....
temporary files
cpu
cdrom
bddicken
user2
ls
sed
photos
documents
photo.jpg
logo.png
hello.py
resume.pdf

Files vs Directories

- A “regular” file (.txt, .c, .out, etc) and a directory are both just files
- A directory files contains a list of inodes including itself, its parent, and its child inodes
- Try: **`$ ls -l`**

https://en.wikipedia.org/wiki/Inode

iNodes and Data Blocks

- Can abstract a hard drive (or SSD) to be an extremely large array of sequential data.
- UNIX files systems divide up this data into two types of “chunks”: iNodes and Data Blocks

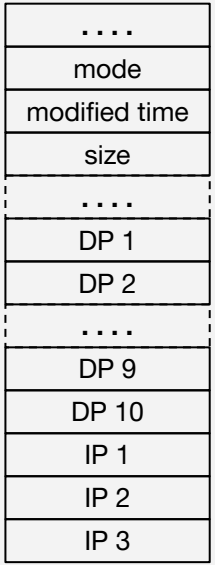
(This is a major simplification, but fine for our purposes for now)

iNodes

Behind the scenes, a **file** is really a node (segment of the HD space) containing a collection of **metadata** and pointers to data blocks

Called **iNodes**

The file systems has a structure (table, tree, etc) of **iNodes** on the hard drive



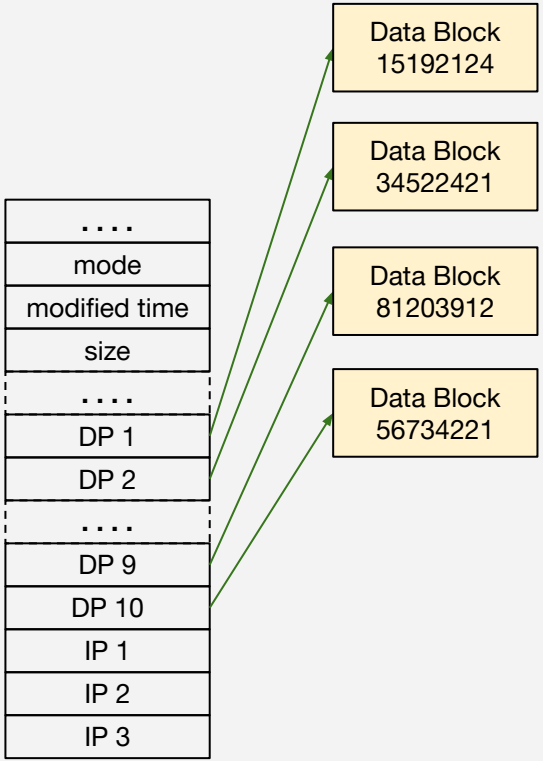
https://www.researchgate.net/figure/The-architecture-of-an-inode-in-EXT3-file-system_fig2_258396310

Data Blocks

Much of the remaining space on an HD not consumed by **iNodes** are **data blocks**

Sequences of data of some size (for example 4096 bytes) that store the actual file data

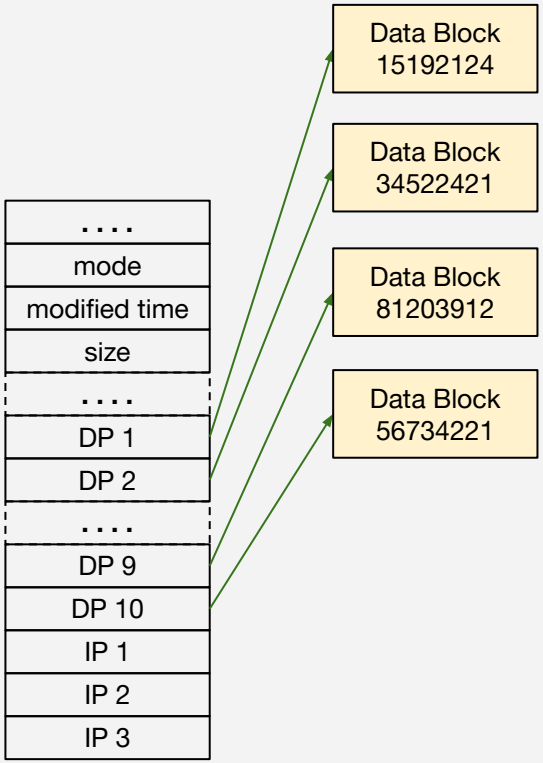
iNodes have *pointers* to one or more data blocks



https://en.wikipedia.org/wiki/Inode_pointer_structure
http://www.linuxintro.org/wiki/Blocks,_block_devices_and_block_sizes

Pointers

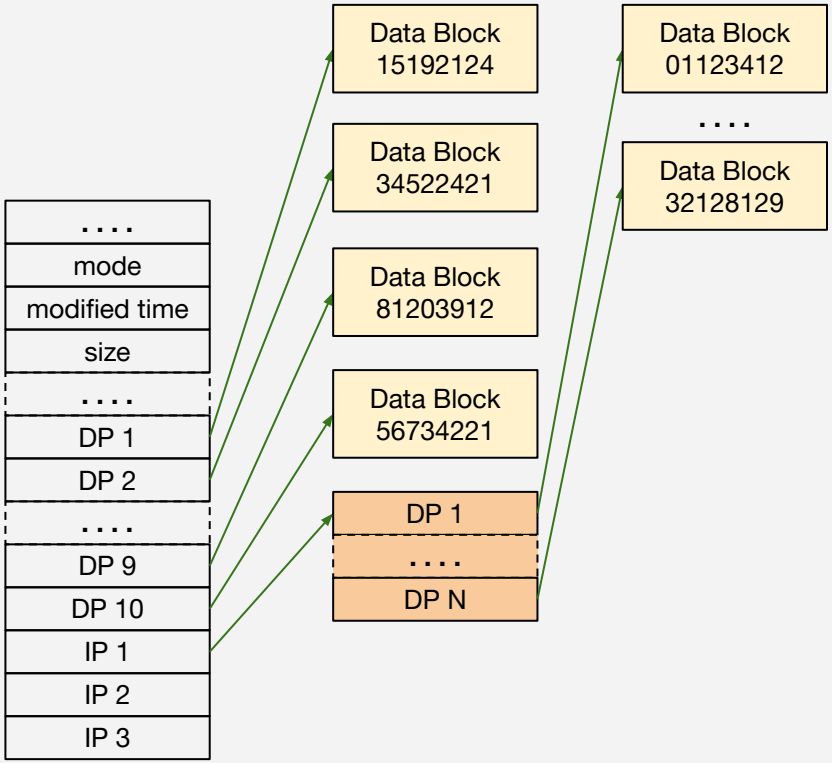
- For small files, can store data within the blocks from the direct pointers
- For larger files, use some of the indirect pointers



https://en.wikipedia.org/wiki/Inode_pointer_structure
http://www.linuxintro.org/wiki/Blocks,_block_devices_and_block_sizes

Pointers

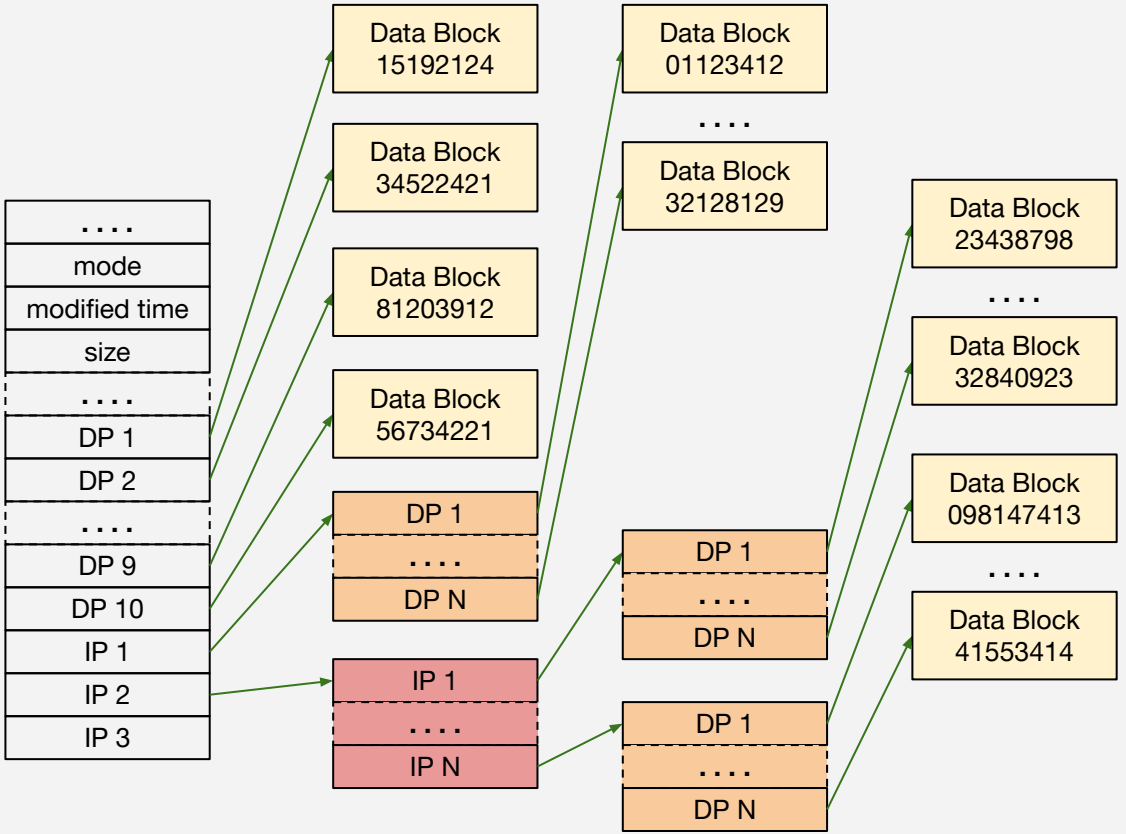
- For small files, can store data within the blocks from the direct pointers
- For larger files, use some of the indirect pointers



https://en.wikipedia.org/wiki/Inode_pointer_structure
http://www.linuxintro.org/wiki/Blocks,_block_devices_and_block_sizes

Pointers

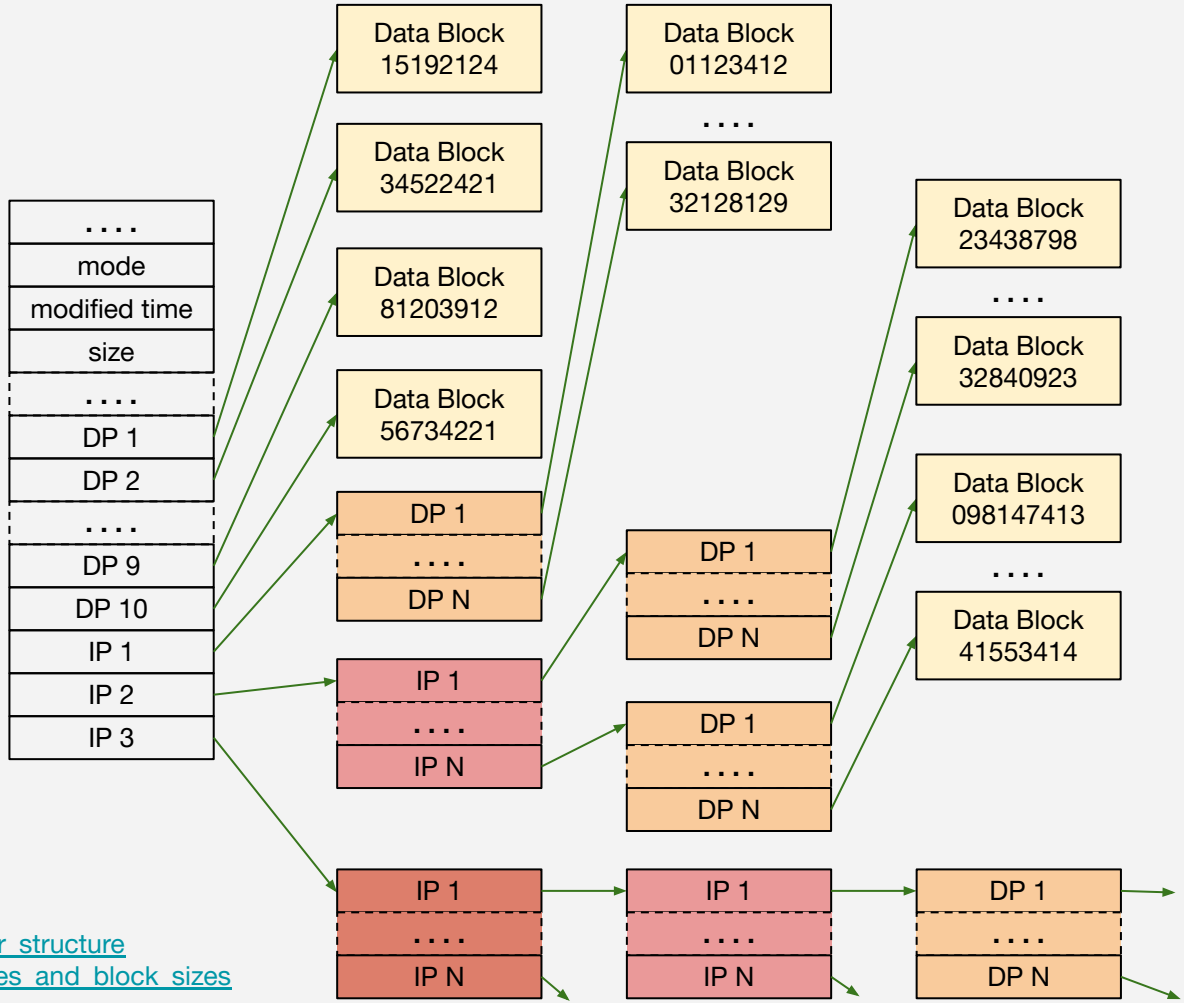
- For small files, can store data within the blocks from the direct pointers
- For larger files, use some of the indirect pointers



https://en.wikipedia.org/wiki/Inode_pointer_structure
http://www.linuxintro.org/wiki/Blocks,_block_devices_and_block_sizes

Pointers

- For small files, can store data within the blocks from the direct pointers
- For larger files, use some of the indirect pointers



https://en.wikipedia.org/wiki/Inode_pointer_structure
http://www.linuxintro.org/wiki/Blocks,_block_devices_and_block_sizes

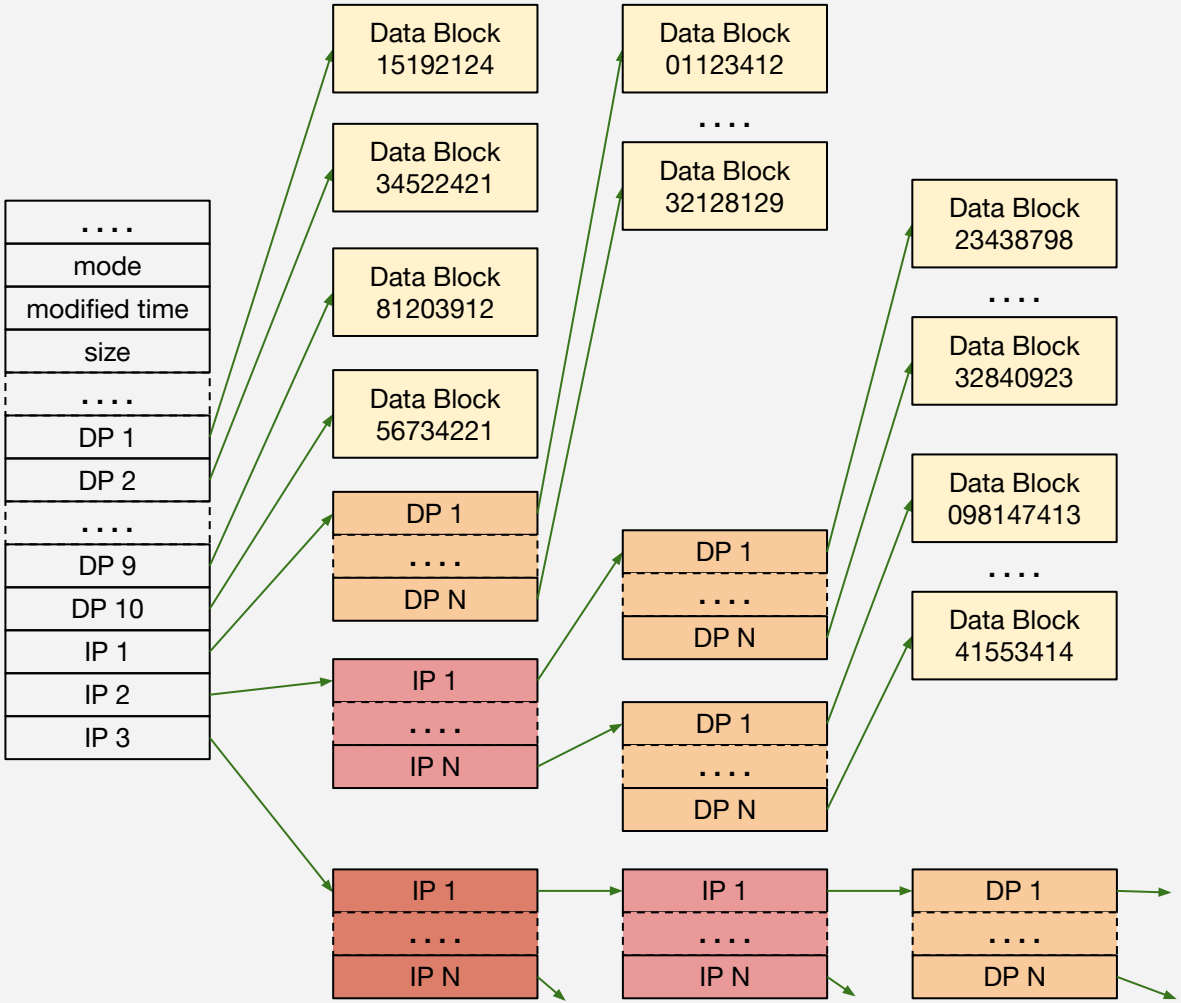
Pointers

Find block size:

```
$ stat -f /
```

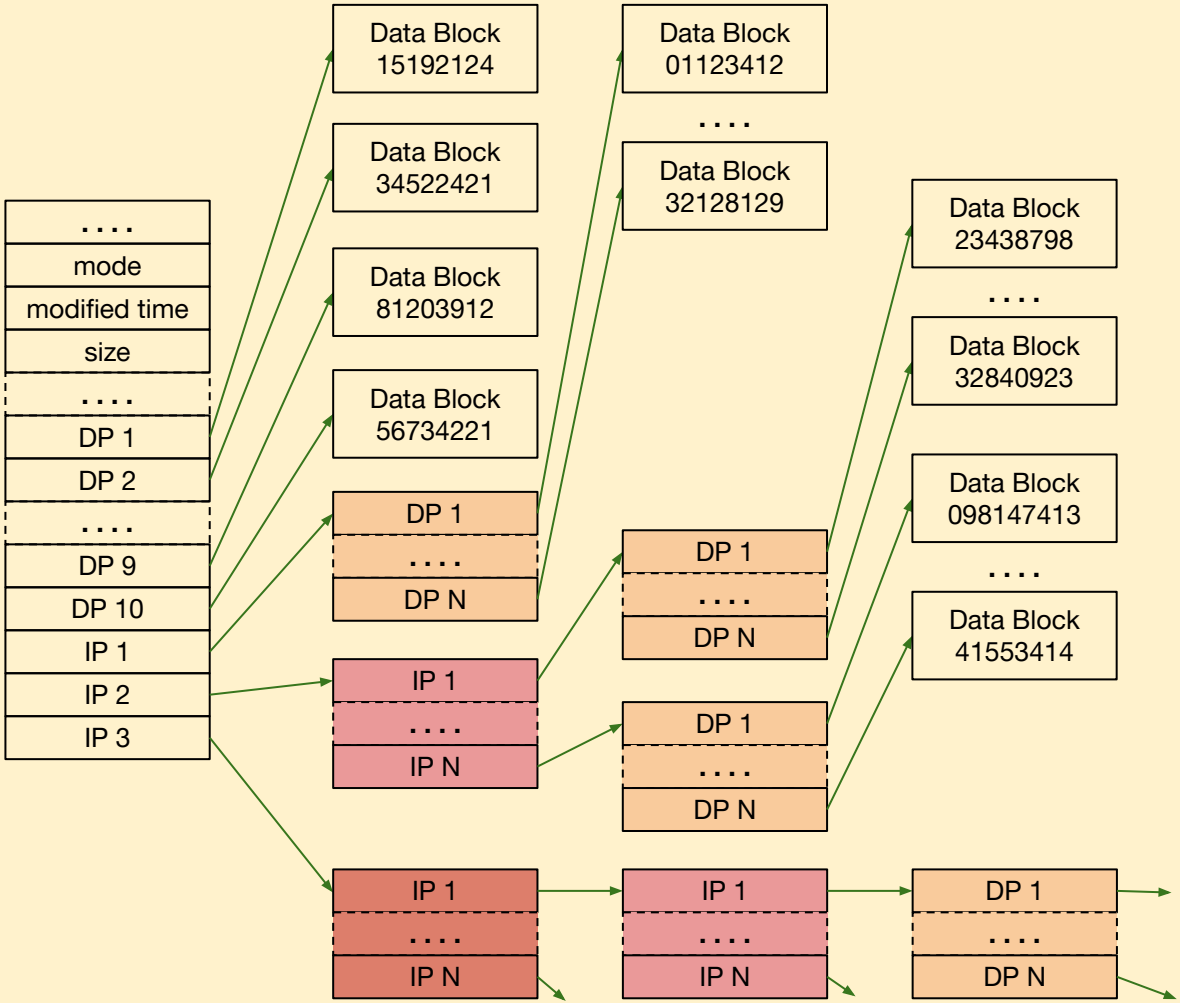
Inode number:

```
$ ls -i
```



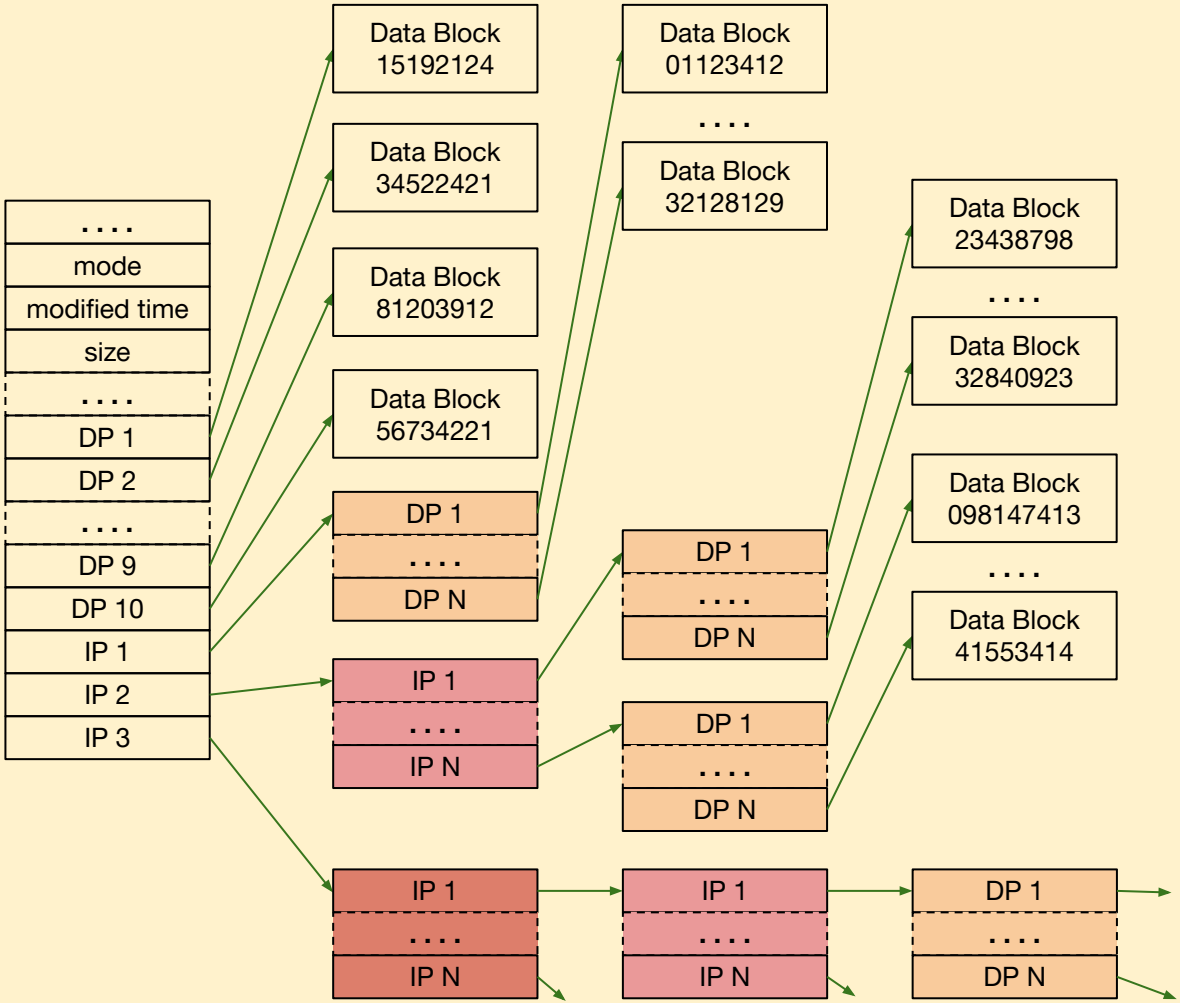
Pointers

How many block pointers will be required for a text file with 5,000 ascii characters with a block size of 1024 bytes and a block pointer size of 64 bits?



Pointers

How many block pointers will be required for a text file with 25,000 ascii characters with a block size of 2048 bytes and a block pointer size of 64 bits?



iNodes

- iNodes themselves are stored in a known location on a hard drive
- Can be an array / table or B / B+ tree

https://xerxes.cs.manchester.ac.uk/comp251/kb/Inodes

File-related Commands

```
stat

df

ls -i
```

Text File I/O in C

- Can read and write text to and from files
- Similar to reading/writing to stdin/stdout
- stdio/stderr are basically just “files” that have already been opened for you

```c
#include <stdio.h>
#include <errno.h>

int main() {
  FILE* test_file;
  test_file = fopen("file.txt", "w");
  if (test_file == NULL) {
    fprintf(stderr, "Opening file failed with code %d.\n", errno);
    return 1;
  }
  fprintf(test_file, "Number: %d\n", 25);
  fflush(test_file);
  fclose(test_file);
  return 0;
}
```

```c
#include <stdio.h>
#include <errno.h>

int main() {
  FILE* test_file;
  test_file = fopen("file.txt", "w");
  if (test_file == NULL) {
    fprintf(stderr, "Opening file failed with code %d.\n", errno);
    return 1;
  }
  fprintf(test_file, "Number: %d\n", 25);
  fflush(test_file);
  fclose(test_file);
  return 0;
}
```

What is a FILE* ?

Many different possible modes (see man pages)

Same function, different locations to send the output to

See man pages for fopen, fprintf, fflush, fclose, fscanf

Implement Sum

Write a C program that

1. Prompts the user for a file name
2. Opens this file
3. Reads through each line of file, assuming each line will have exactly 1 integer number
4. Sum the numbers, save the result to sum.txt

What is a FILE?

A structure containing the necessary information to manage that particular file

See the standard!

http://port70.net/~nsz/c/c11/n1570.html

What is a FILE?

Investigate on lectura. You can use:

```
$ locate stdio.h

$ echo '#include <stdio.h>' | cpp -H -o /dev/null 2>&1 | head -n1
```

Can you figure out what a FILE actually is?

What is a FILE?

/usr/include/stdio.h

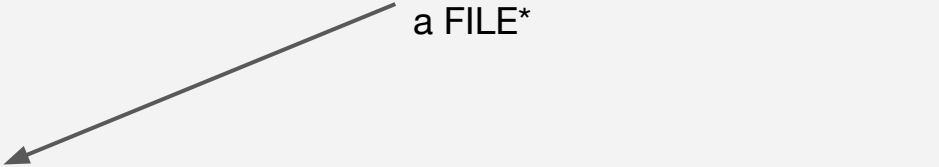
/usr/include/x86_64-linux-gnu/bits/types/struct_FILE.h

```c
#include <stdio.h>

int main() {
  FILE* test_file;
  char line[128];
  test_file = fopen("data.txt", "r");
  if (test_file == NULL) {
    fprintf(stderr, "error opening the file.\n");
    return 1;
  }

  while (fgets(line, 127, test_file) != NULL) {
      printf(">%s<\n", line);
  }

  fclose(test_file);
  return 0;
}
```



Function summary

- **`fopen`** - For opening files, getting FILE pointers. Can open in various modes
- **`fscanf / fgets`** - For reading from files
- **`fprintf`** - For writing to a file
- **`fflush`** - Ensure that any buffered content gets written to the file stream
- **`fclose`** - Close the file

Implement toupper.c

Write a C program that

1. Prompts the user for two input files names
2. The program should read in the lines from the first, convert alphabetical character to CAPS, and write to the second file
3. Close files when done

File Permissions

Each file can have designated permissions for owner, group, and everyone

For each of those, can specify if allowed to **read** and/or **write** and/or **execute**

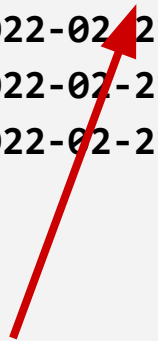
`ls -l test.c` or `stat test.c`

File Permissions

```
lectura:> stat test.c
  File: test.c
  Size: 175           Blocks: 14         IO Block: 131072 regular file
Device: 43h/67d   Inode: 4570337     Links: 1
Access: (0751/-rwxr-x--x)  Uid: (14358/bddicken)   Gid: (    0/    root)
Access: 2022-02-21 12:37:09.281929146 -0700
Modify: 2022-02-21 12:37:09.283156247 -0700
Change: 2022-02-21 12:55:26.879605124 -0700
```

File Permissions

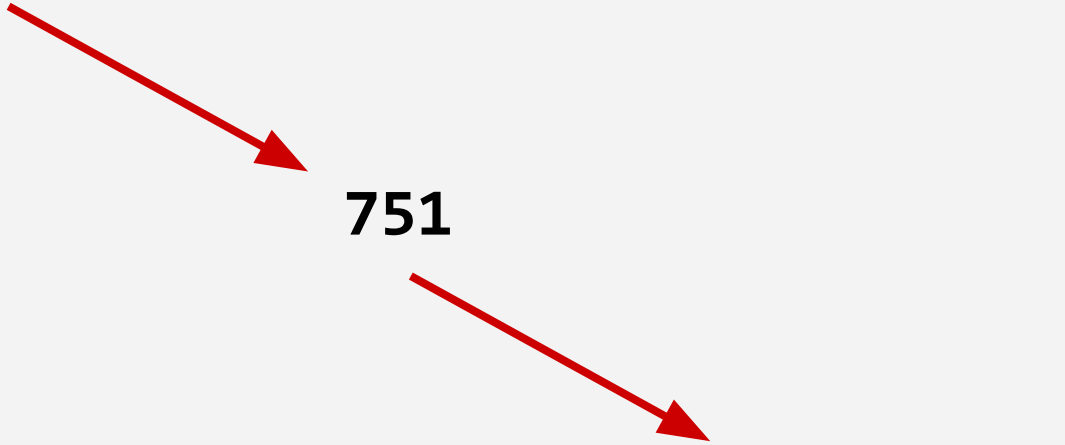
```
lectura:> stat test.c
  File: test.c
  Size: 175           Blocks: 14         IO Block: 131072 regular file
Device: 43h/67d   Inode: 4570337     Links: 1
Access: (0751/-rwxr-x--x)  Uid: (14358/bddicken)   Gid: (    0/    root)
Access: 2022-02-21 12:37:09.281929146 -0700
Modify: 2022-02-21 12:37:09.283156247 -0700
Change: 2022-02-21 12:55:26.879605124 -0700
```



Owner can read, write, exec
Group can read and exec
Everyone can exec

File Permissions

rwxr-x--x



751

111101001

Chmod

Use chmod to specify permissions

```
$ chmod 751 test.c
```

Sets permissions for test.c to **111101001** or **rwxr-x--x**

Chmod

Write the chmod command to set the permissions of the file **test.txt** to be:

`r-x--xrwx`