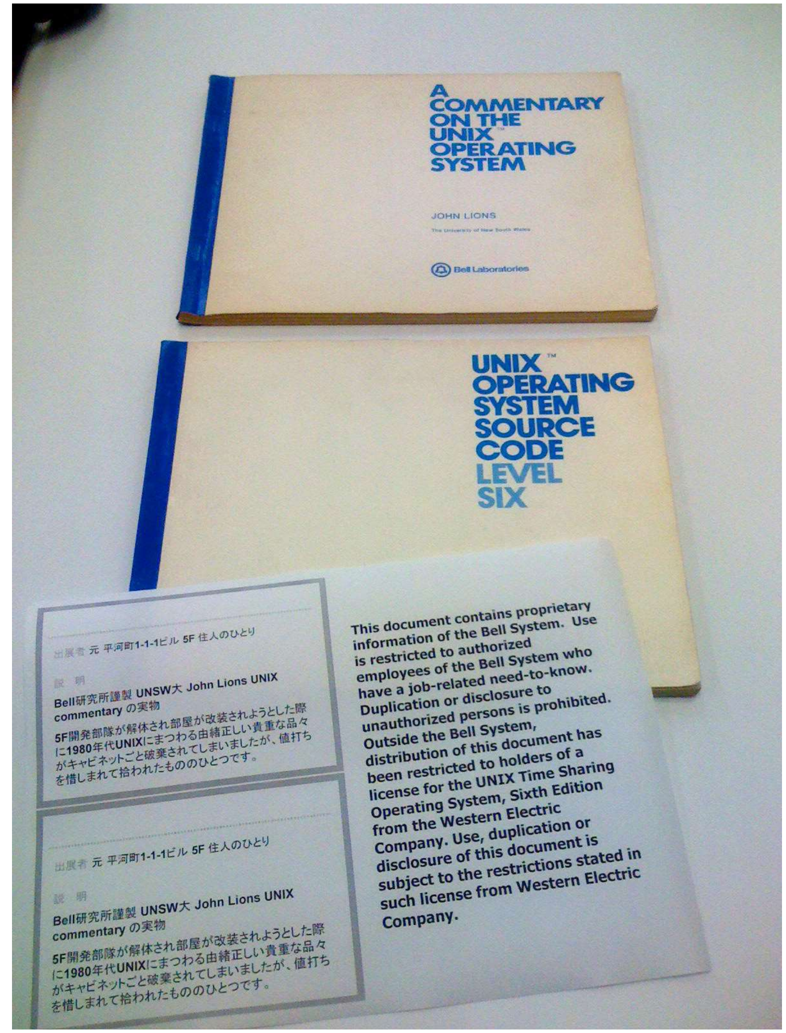
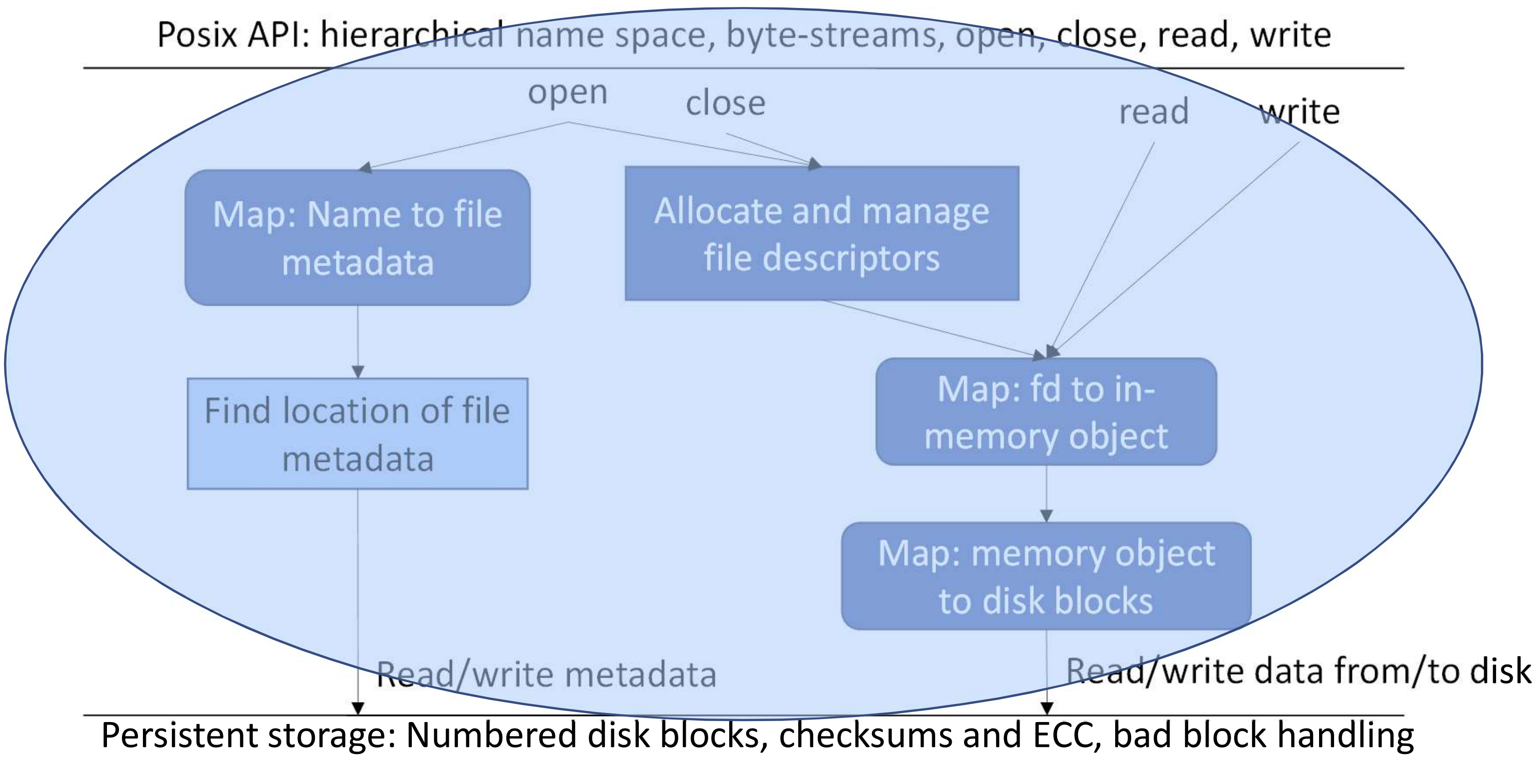
CPSC313: Computer Hardware and Operating Systems

File Systems

Case Study: V6



You are here!


Posix API: hierarchical name space, byte-streams, open, close, read, write
open
close
read
write
Map: Name to file metadata
Allocate and manage file descriptors
Find location of file metadata
Map: fd to in-memory object
Map: memory object to disk blocks
Read/write metadata
Read/write data from/to disk
Persistent storage: Numbered disk blocks, checksums and ECC, bad block handling

Outline

- How do we lay out a real file system?
  - **Disk Layout**: What goes where, and how big is it?
  - ***File System Metadata***: How do I find all the parameters of this file system?
  - ***Free Space Management***: How do I find the next block to allocate?
    How do I deallocate a block?
  - **Logical-Physical Mapping**: How do I find a particular block within a file?
  - **Directory Structures**: How do I make paths/directories work?

- Learning Outcomes
  - Discuss the tradeoffs in file system design choices
  - Describe how the Unix v6 file system represented files

How Do We Decide?

We're designing all the moving pieces of a file system. How do we pick what it looks like?

Usable! Good for our human end users (and programs!)

Fast! (When working with big files? Small? Allocating new blocks?)

Compact! (In terms of internal fragmentation? External? Persistent FS data structures?)

Easy to restore in the face of damage to the media?

Our Real File System: Unix v6

With enormous thanks to:

Ken Thompson and Dennis Ritchie, John Lions (https://en.wikipedia.org/wiki/Lions'_Commentary_on_UNIX_6th_Edition,_with_Source_Code), Keith Bostic, Whoever is behind: http://v6.cuzuco.com/

The Unix v6 File System

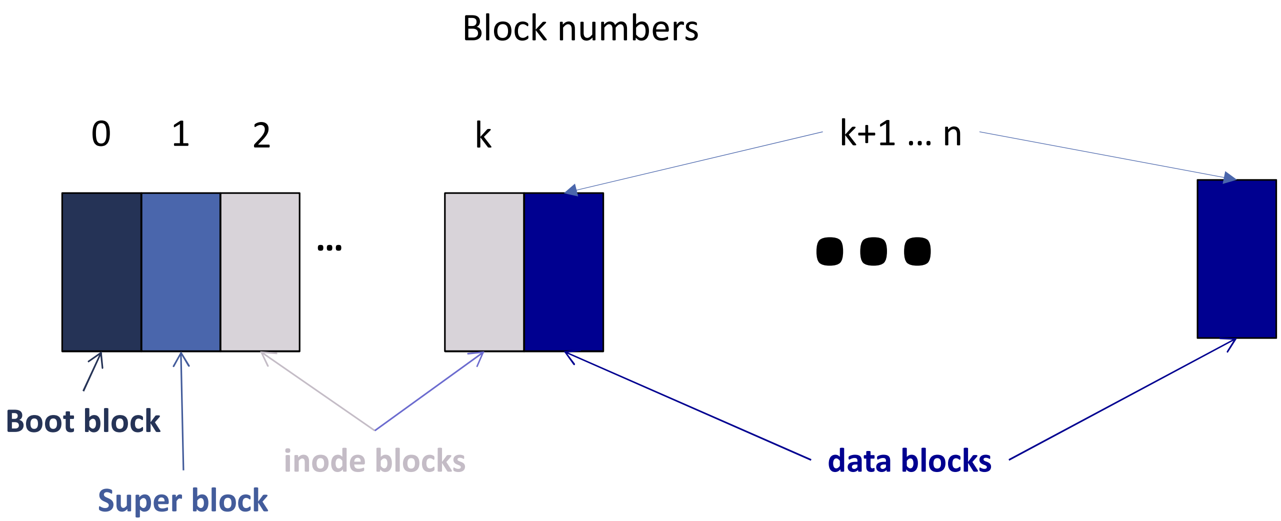
One of the first widely distributed versions of Unix (1975), ran on a “PDP-11” minicomputer from DEC. Critically for us:

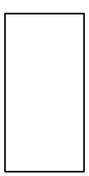
- **Released as full source code**
- **Documented** by John Lions of the University of New South Wales.

We start with this *relatively* simple ancestor of most modern FS’s.

In 1975, **disks were tiny**. You’ll notice ***compactness*** is a key goal, sometimes at the expense of others!

Disk Layout





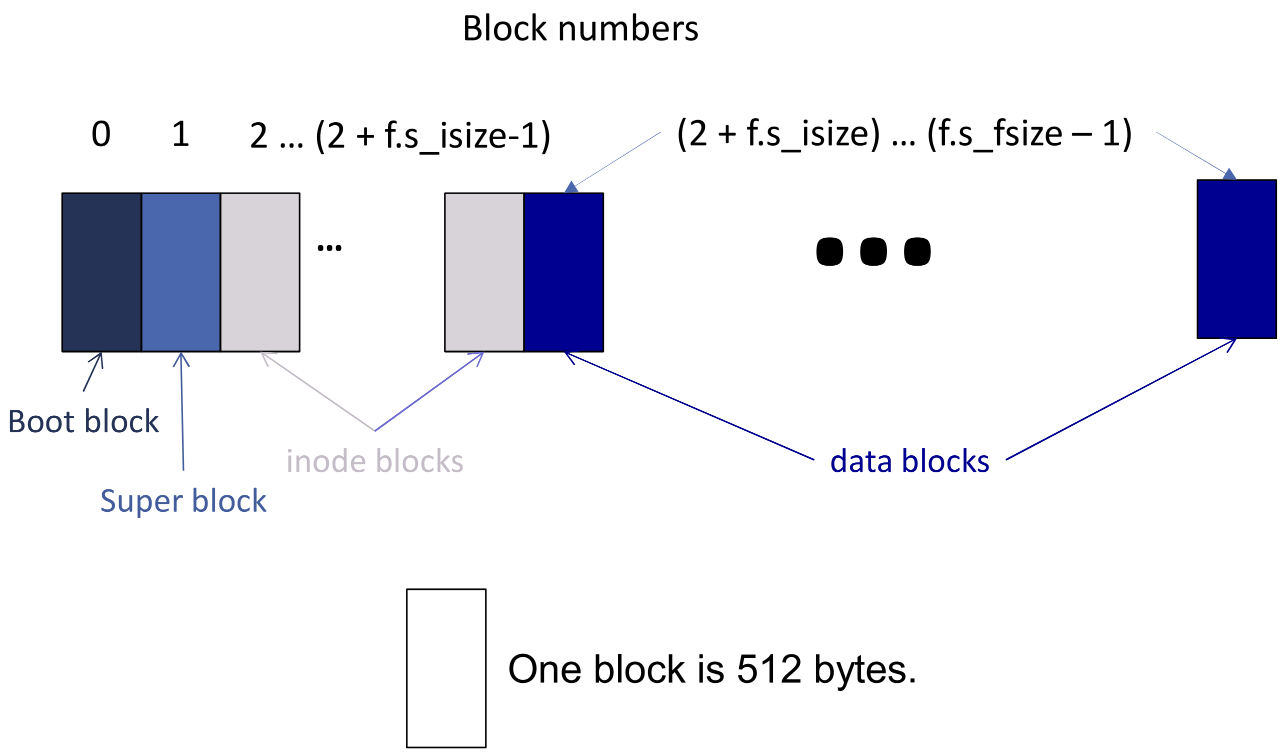
One block is 512 bytes.

File system level metadata

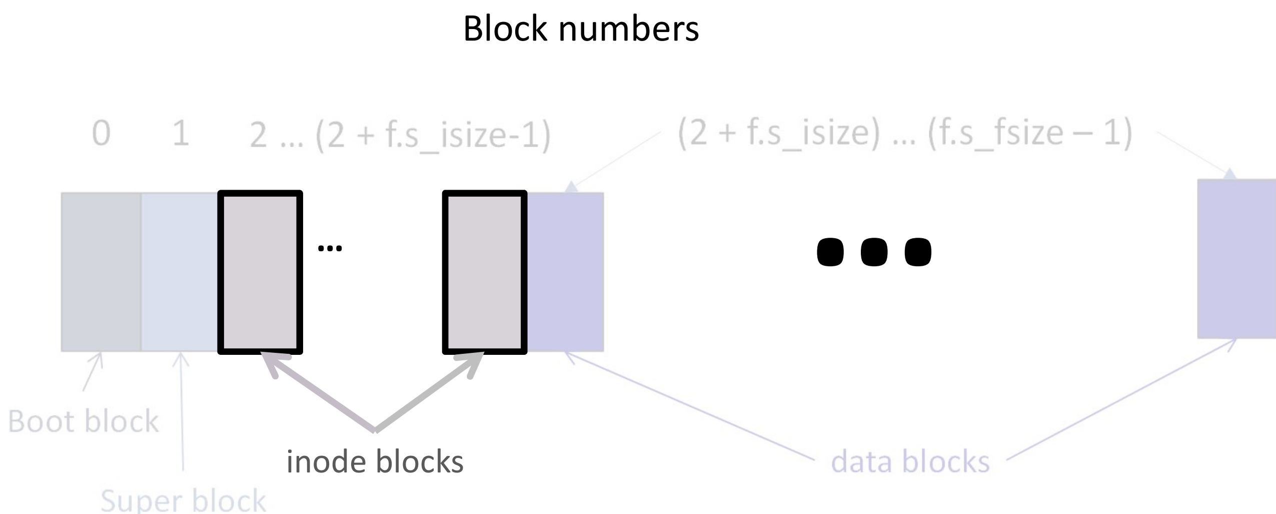
- `struct filsys`
  - Today, we call this the superblock
  - Created when you create the file system
  - Read when you mount the file system

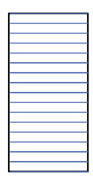
```
struct filsys {
 int s_isize;       /* size in blocks of the I list */
 int s_fsize;       /* size in blocks of the entire volume */
 int s_nfree;       /* number of in core free blocks(between 0 and 100) */
 int s_free[100];   /* in core free blocks */
 int s_ninode;      /* number of in core I nodes (0-100) */
 int s_inode[100];  /* in core free I nodes */
 char s_flock;      /* lock during free list manipulation */
 char s_ilock;      /* lock during I list manipulation */
 char s_fmod;       /* super block modified flag */
 char s_ronly;      /* super block read-only flag */
 int s_time[2];     /* current date of last update */
 int pad[50];
}
```

Disk Layout


Block numbers
0
1
2 … (2 + f.s_isize-1)
(2 + f.s_isize) … (f.s_fsize – 1)
…
Boot block
Super block
inode blocks
data blocks
One block is 512 bytes.

Disk Layout: What's in an inode block?





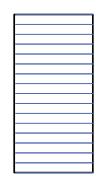
Each inode is 32 bytes in length.
16 fit in a 512 byte block.

Per-file Metadata (on-disk) [inode]

Size in bytes..

- On disk inode (note: int is 2 bytes):

```
struct ino {
  2  int    i_mode;      /* File type, size, permissions */
  1  char   i_nlink;     /* Link count */
  1  char   i_uid;       /* Owner user id */
  1  char   i_gid;       /* Group id */
  1  char   i_size0;     /* most significant bits of size */
  2  int    i_size1;     /* least sig */
  16 int    i_addr[8];   /* Disk addresses of blocks */
  4  int    i_atime[2];  /* Access time */
  4  int    i_mtime[2];  /* Modified time */
}
  32!
```



Each inode is 32 bytes in length.
16 fit in a 512 byte block.

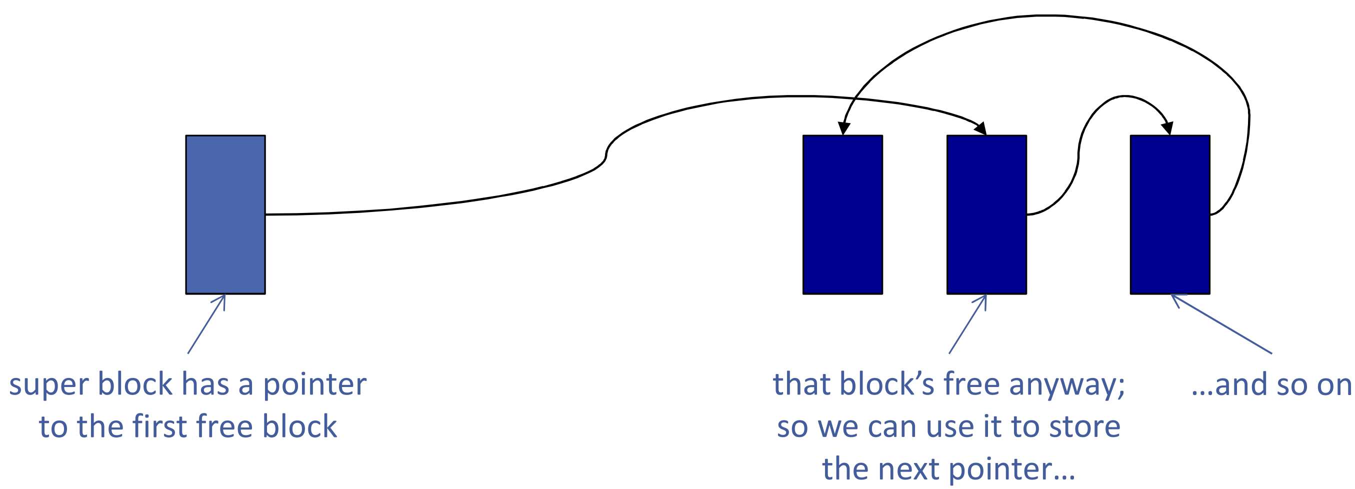
Let's come back to this later when
we map logical -> physical blocks!

Outline

- How do we lay out a real file system?
  - **Disk Layout**: What goes where, and how big is it?
  - ***File System Metadata***: How do I find all the parameters of this file system?
  - ***Free Space Management***: How do I find the next block to allocate?
    How do I deallocate a block?
  - **Logical-Physical Mapping**: How do I find a particular block within a file?
  - **Directory Structures**: How do I make paths/directories work?

- Learning Outcomes
  - Discuss the tradeoffs in file system design choices
  - Describe how the Unix v6 file system represented files

Lie #1: Just Have a Linked-List of Free Blocks!



Problems?

Every allocate or deallocate messes with a random block.

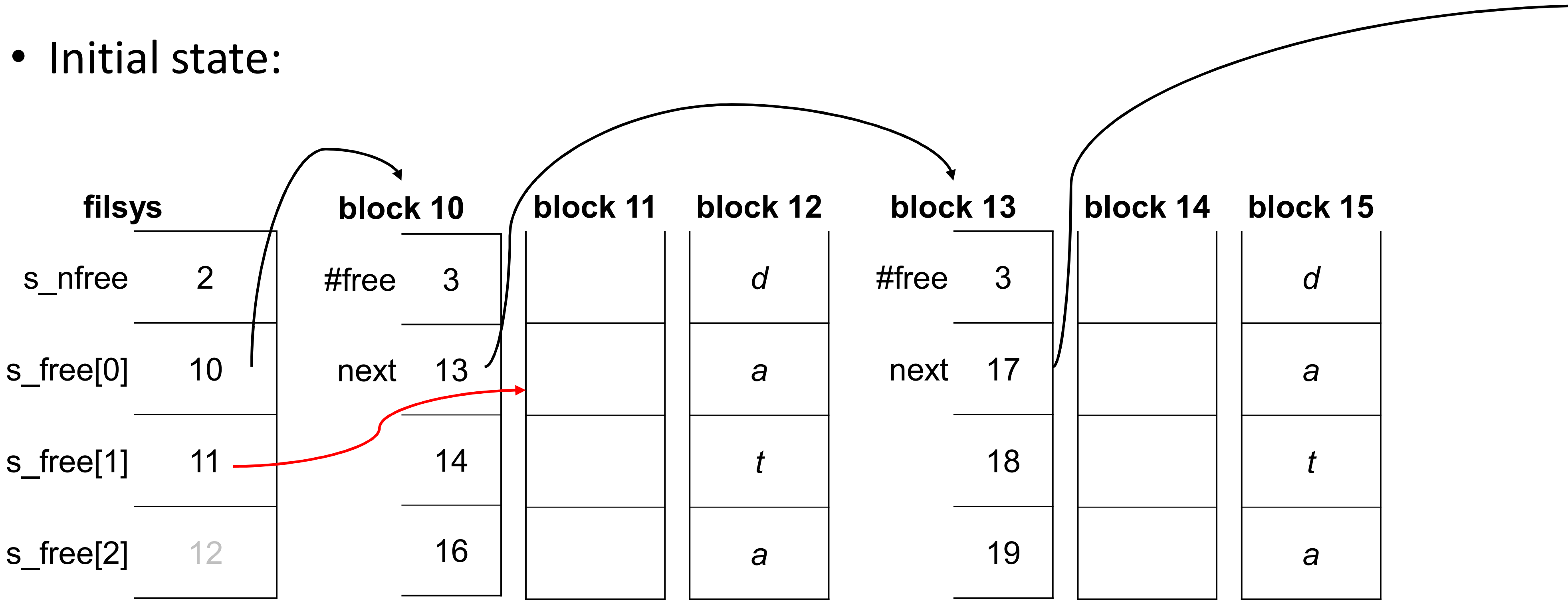
We'd rather touch only the (probably-cached!) super block whenever possible.

Wait… a block number (pointer) is small! Let's put **three** free blocks in each linked list node instead of one.

Tiny disk! Don't waste a bunch of space tracking free blocks. **Use** the free blocks to **track** the free blocks!

Lie #2: Each node has an array of up to 3 block #s. (Our **`next`** pointer is **`s_free[0]`**.)

- Initial state:



Block 11 is also free. If I use it, the update to my data structure stays in `filsys`.

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate a block: *#11*

filsys		block 10		block 11	block 12	block 13		block 14	block 15
s_nfree	2	#free	3		*d*	#free	3		*d*
s_free[0]	10	next	13		*a*	next	17		*a*
s_free[1]	11		14		*t*		18		*t*
s_free[2]	12		16		*a*		19		*a*

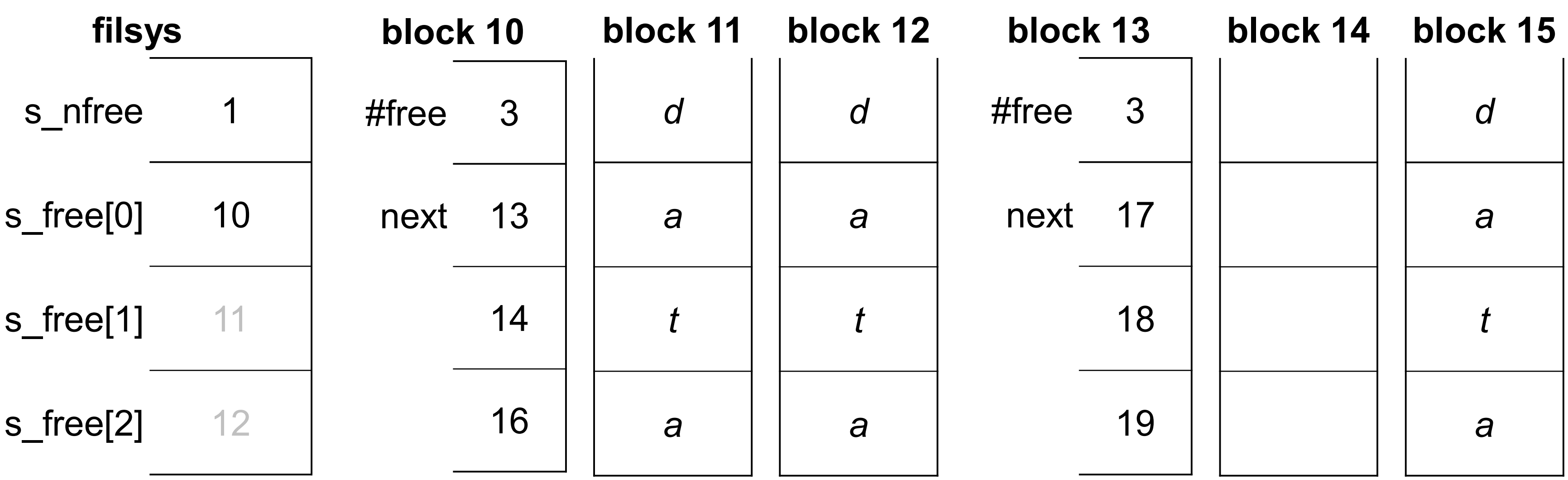
Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate a block: *#11*

	filsys		block 10	block 11	block 12		block 13	block 14	block 15
s_nfree	1	#free	3	*d*	*d*	#free	3		*d*
s_free[0]	10	next	13	*a*	*a*	next	17		*a*
s_free[1]	11		14	*t*	*t*		18		*t*
s_free[2]	12		16	*a*	*a*		19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate another block: ***#10***



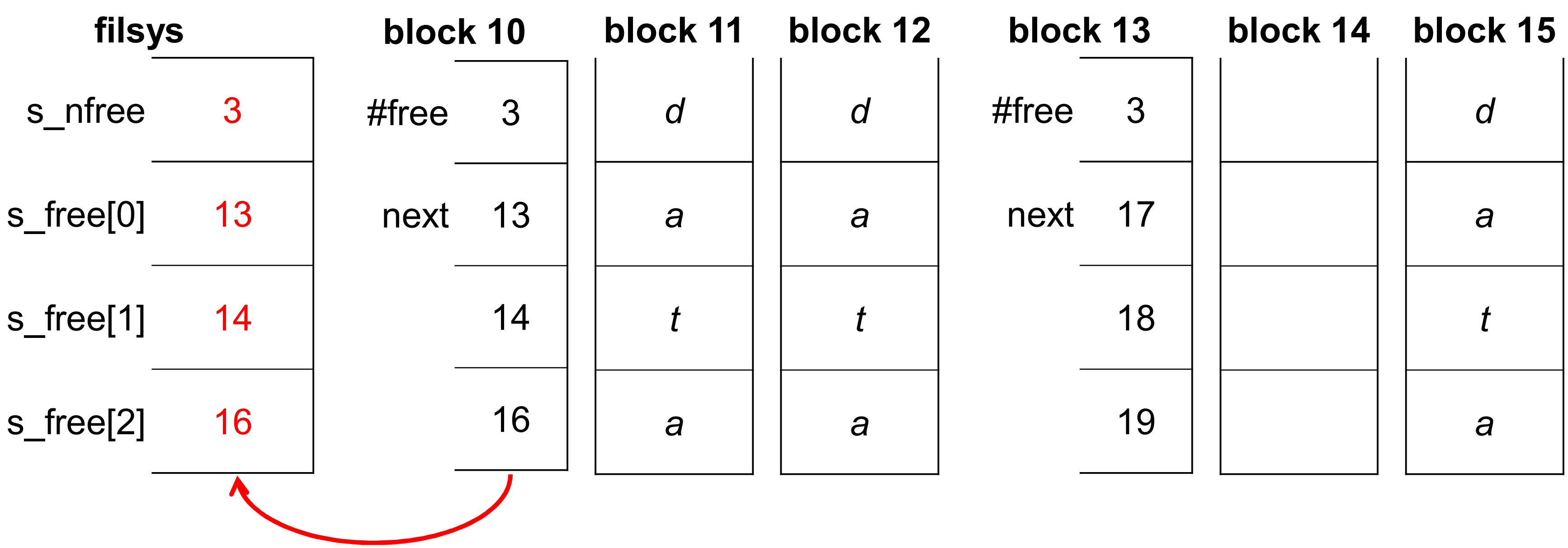
Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate another block: *#10 [step 1]*

filsys		block 10		block 11	block 12	block 13		block 14	block 15
s_nfree	1	#free	3	*d*	*d*	#free	3		*d*
s_free[0]	10	next	13	*a*	*a*	next	17		*a*
s_free[1]	11		14	*t*	*t*		18		*t*
s_free[2]	12		16	*a*	*a*		19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate another block: *#10 [step 1]*



Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate another block: *#10 [step 2]*

	filsys	block 10	block 11	block 12	block 13		block 14	block 15
s_nfree	3	*d*	*d*	*d*	#free	3		*d*
s_free[0]	13	*a*	*a*	*a*	next	17		*a*
s_free[1]	14	*t*	*t*	*t*		18		*t*
s_free[2]	16	*a*	*a*	*a*		19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate another block: *#16*

filsys		block 10	block 11	block 12	block 13		block 14	block 15
s_nfree	3	*d*	*d*	*d*	#free	3		*d*
s_free[0]	13	*a*	*a*	*a*	next	17		*a*
s_free[1]	14	*t*	*t*	*t*		18		*t*
s_free[2]	16	*a*	*a*	*a*		19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We allocate another block: ***#16***

filsys		block 10	block 11	block 12	block 13		block 14	block 15
s_nfree	2	*d*	*d*	*d*	#free	3		*d*
s_free[0]	13	*a*	*a*	*a*	next	17		*a*
s_free[1]	14	*t*	*t*	*t*		18		*t*
s_free[2]	16	*a*	*a*	*a*		19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We deallocate block *#12*

	filsys	block 10	block 11	block 12	block 13		block 14	block 15
s_nfree	2	*d*	*d*	*d*	#free	3		*d*
s_free[0]	13	*a*	*a*	*a*	next	17		*a*
s_free[1]	14	*t*	*t*	*t*		18		*t*
s_free[2]	16	*a*	*a*	*a*		19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We deallocate block *#12*

	filsys	block 10	block 11	block 12		block 13	block 14	block 15
s_nfree	3	*d*	*d*		#free	3		*d*
s_free[0]	13	*a*	*a*		next	17		*a*
s_free[1]	14	*t*	*t*			18		*t*
s_free[2]	12	*a*	*a*			19		*a*

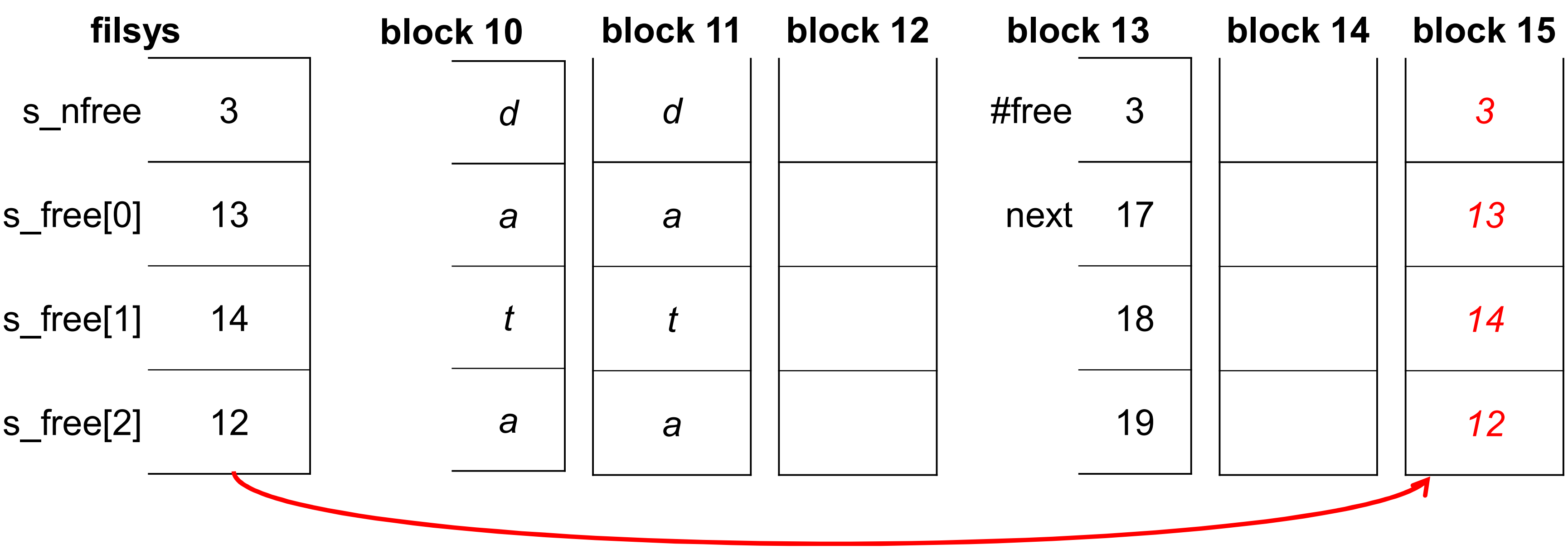
Free Space Management Example

- Assume we store block numbers in groups of 3.
- We deallocate block ***#15***

	filsys	block 10	block 11	block 12		block 13	block 14	block 15
s_nfree	3	*d*	*d*		#free	3		*d*
s_free[0]	13	*a*	*a*		next	17		*a*
s_free[1]	14	*t*	*t*			18		*t*
s_free[2]	12	*a*	*a*			19		*a*

Free Space Management Example

- Assume we store block numbers in groups of 3.
- We deallocate block *#15 [step 1]*



Free Space Management Example

- Assume we store block numbers in groups of 3.
- We deallocate block *#15 [step 2]*

	filsys	block 10	block 11	block 12		block 13	block 14	block 15
s_nfree	1	*d*	*d*		#free	3		*3*
s_free[0]	15	*a*	*a*		next	17		*13*
s_free[1]	14	*t*	*t*			18		*14*
s_free[2]	12	*a*	*a*			19		*12*

Strength? Weaknesses?

We need very little extra space to store our free list!

But the free list works a lot like a linked-list-style malloc/memory manager. Will we end up with a lot of contiguous or nearby blocks in a given file?

vs.

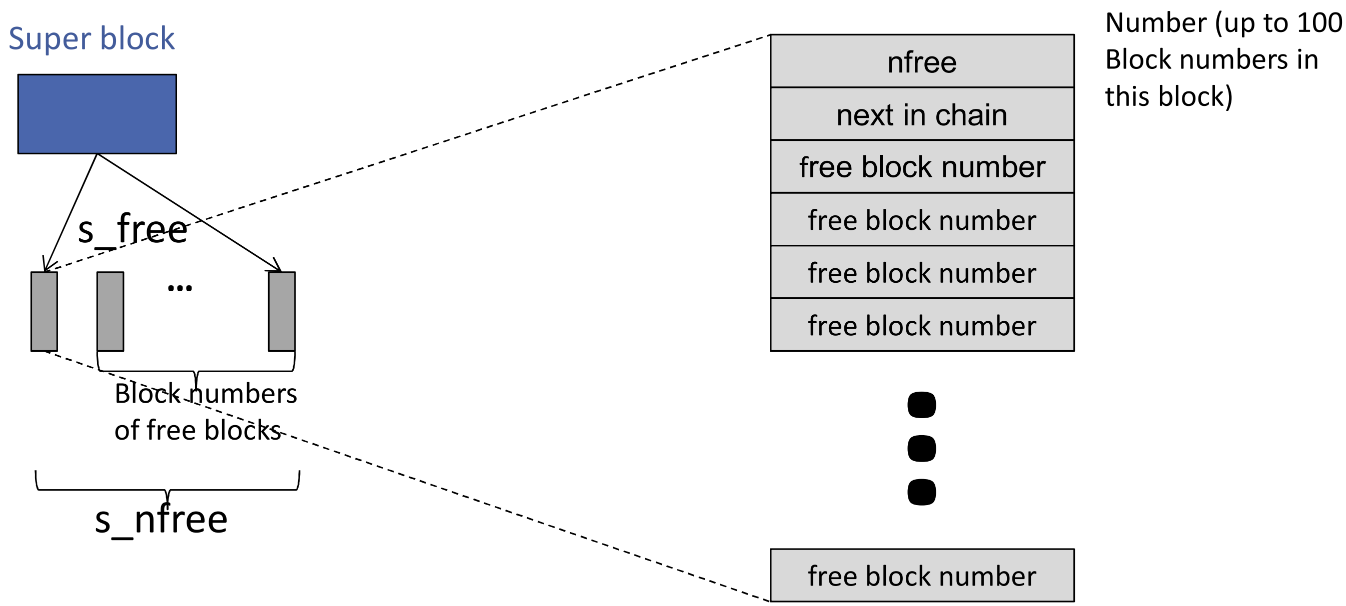
?

Actual V6 Free Space Management: 100 not 3

- `struct filsys`
  - Today, we call this the superblock
  - Created when you create the file system
  - Read when you mount the file system

```
struct filsys {
 int s_isize;        /* size in blocks of the I list */
 int s_fsize;        /* size in blocks of the entire volume */
 int s_nfree;        /* number of in-core free blocks (between 0 and 100) */
 int s_free[100];    /* in core free blocks */
 int s_ninode;       /* number of in-core I nodes (0-100) */
 int s_inode[100];   /* in core free I nodes */
 char s_flock;       /* lock during free list manipulation */
 char s_ilock;       /* lock during I list manipulation */
 char s_fmod;        /* super block modified flag */
 char s_ronly;       /* super block read-only flag */
 int s_time[2];      /* current date of last update */
 int pad[50];
}
```

Free Space Management



Outline

- How do we lay out a real file system?
  - **Disk Layout**: What goes where, and how big is it?
  - ***File System Metadata***: How do I find all the parameters of this file system?
  - ***Free Space Management***: How do I find the next block to allocate?
How do I deallocate a block?
  - **Logical-Physical Mapping**: How do I find a particular block within a file?
  - **Directory Structures**: How do I make paths/directories work?

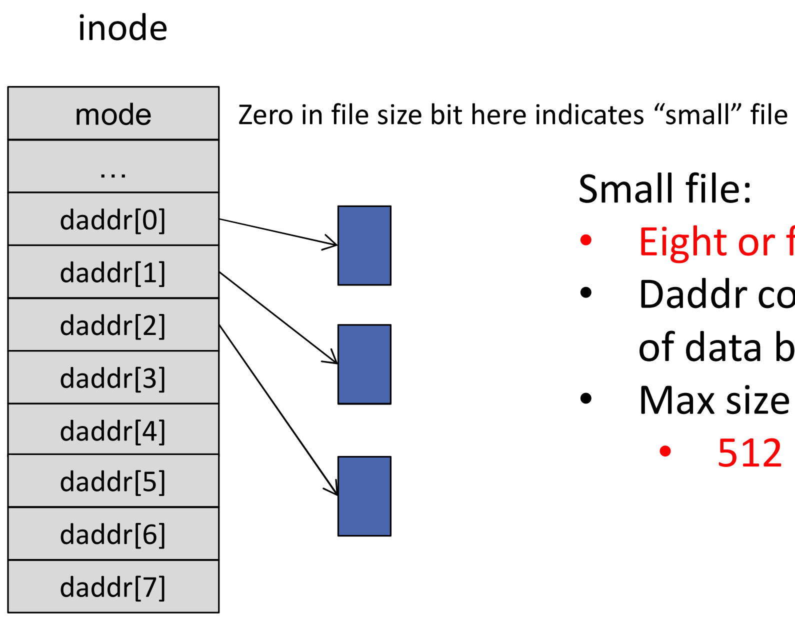
- Learning Outcomes
  - Discuss the tradeoffs in file system design choices
  - Describe how the Unix v6 file system represented files

Per-file Metadata (on-disk) [inode]

- On disk inode (note: int is 2 bytes):

```
struct ino {
    int     i_mode;         /* File type, size, permissions */
    char    i_nlink;        /* Link count */
    char    i_uid;          /* Owner user id */
    char    i_gid;          /* Group id */
    char    i_size0;        /* most significant bits of size */
    int     i_size1;        /* least sig */
    int     i_addr[8];      /* Disk addresses of blocks */
    int     i_atime[2];     /* Access time */
    int     i_mtime[2];     /* Modified time */
}
```

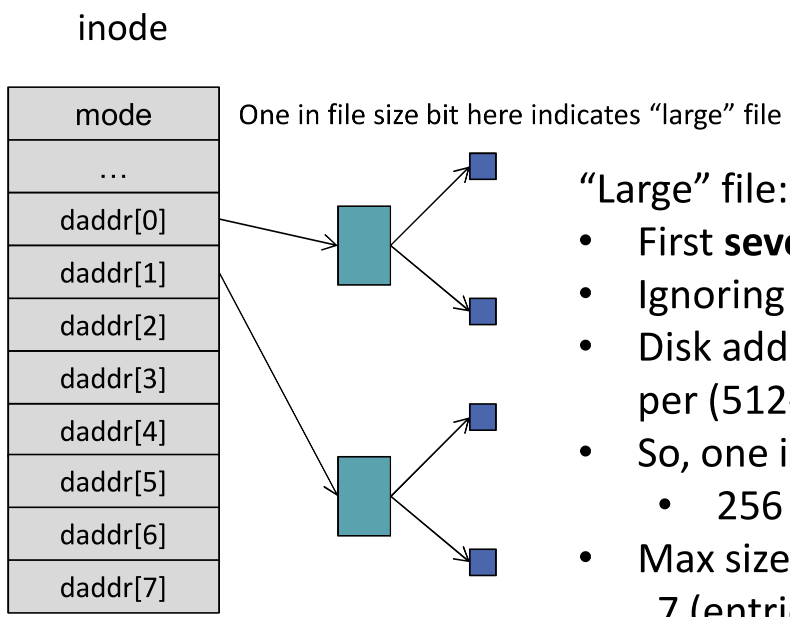
Different sized files (1)



Small file:

- Eight or fewer blocks.
- Daddr contains addresses of data blocks.
- Max size:
  - 512 * 8 = 4KB

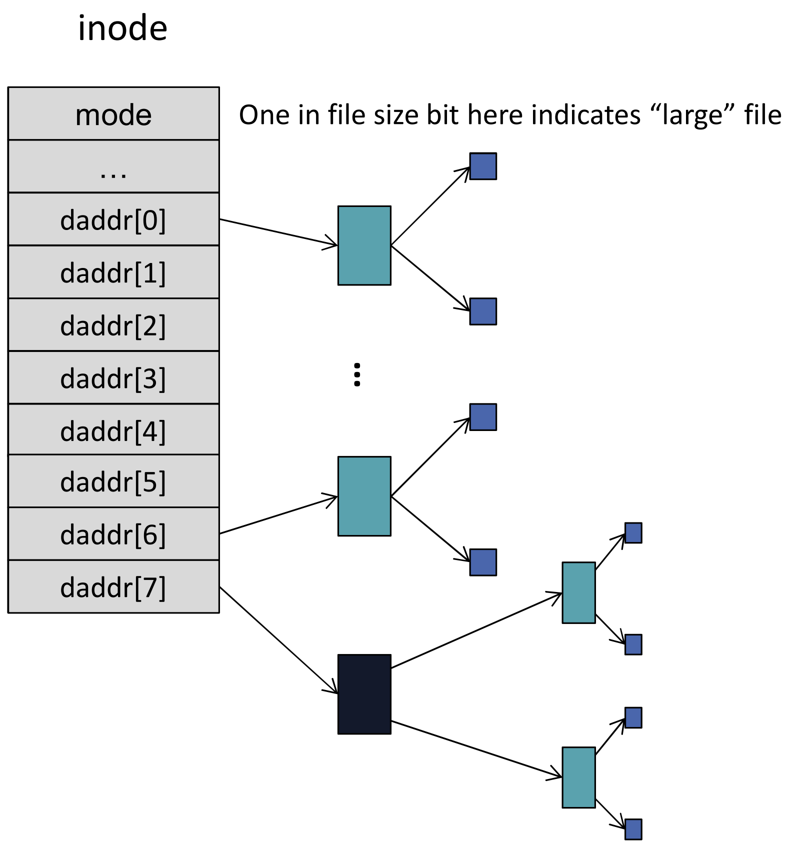
Different sized files (2)



“Large” file:

- First **seven** entries contain addresses of indirect blocks.
- Ignoring daddr[7] until the next slide ...
- Disk addresses are 2 bytes, so 512/2 or 256 disk addresses per (512-byte) block
- So, one indirect block can access:
  - 256 (daddrs per block) * 512 bytes per block = 128 KB
- Max size:
  7 (entries in daddr) * 128 KB/indirect block = 896 KB

Different sized files (3)

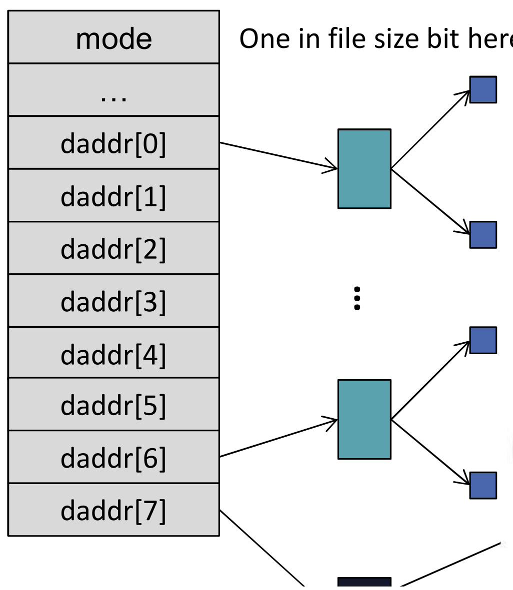


“Huge” file:

- 7 indirect blocks addresses in daddr, AND
- **daddr[7]** contains a double indirect block.
- From the previous slide:
  - 1 indirect block reaches 128 KB
- A double indirect contains 256 daddrs
- A double indirect reaches:
  - 256 * 128 KB = 32 MB
- Max size:
  - 896 KB + 32 MB

Different sized files (3)

inode



One in file size bit here indicates “large” file

“Huge” file:

- 7 indirect blocks addresses in daddr, AND
- **daddr[7]** contains a double indirect block.
- From the previous slide:
  - 1 indirect block reaches 128 KB

Fast… for small files no “wasted” indirect blocks to pass through.

Usable? 32MB **was** huge back then!

Compact? Small files don’t need “wasted” indirect blocks.
Small files were (and are) common.

Per-file Metadata (in-memory) [vnode]

- In-memory inode (note: int is 2 bytes):

```
struct inode {
     char    i_flag;
     char    i_count;        /* reference count */
     int     i_dev;          /* device where inode resides */
     int     i_number;       /* i number 1:1 w/device addr */
     int     i_mode;
     char    i_nlink;        /* directory entries*/
     char    i_uid;          /* owner */
     char    i_gid;          /* group of owner */
     char    i_size0;        /* most significant of size */
     int     i_size1;        /* least sig */
     int     i_addr[8];      /* device addresses constituting file */
     int     i_lastr;        /* last logical block read */
}
```

Outline

- How do we lay out a real file system?
  - **Disk Layout**: What goes where, and how big is it?
  - ***File System Metadata***: How do I find all the parameters of this file system?
  - ***Free Space Management***: How do I find the next block to allocate?
    How do I deallocate a block?
  - **Logical-Physical Mapping**: How do I find a particular block within a file?
  - **Directory Structures**: How do I make paths/directories work?

- Learning Outcomes
  - Discuss the tradeoffs in file system design choices
  - Describe how the Unix v6 file system represented files

Directory Entries

- Hierarchical directory structure that you know and love, including "." and "..".
- Directory entries are 16 bytes (they are fixed size!):
  - 2 bytes of inode number
  - 14 bytes (right padded) of name
- A directory entry with inode = 0 is unused

Directory Exercise - revisited

```
0000000 00 01 2e 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010 01 01 2e 2e 00 00 00 00 00 00 00 00 00 00 00 00
0000020 64 00 57 6f 6d 62 61 74 00 00 00 00 00 00 00 00
0000030 65 00 4d 61 63 6b 65 72 65 6c 00 00 00 00 00 00
0000040 66 00 44 6f 76 65 00 00 00 00 00 00 00 00 00 00
0000050 67 00 43 61 74 74 6c 65 00 00 00 00 00 00 00 00
0000060 00 00 44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
0000070 02 01 4b 6f 69 00 00 00 00 00 00 00 00 00 00 00
0000080 00 00 52 61 62 62 69 74 00 00 00 00 00 00 00 00
0000090 68 00 43 6f 6e 64 6f 72 00 00 00 00 00 00 00 00
00000a0 69 00 44 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
00000b0 03 01 41 6c 70 61 63 61 00 00 00 00 00 00 00 00
00000c0 6a 00 44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
00000d0
```

```
struct dirent {
    int d_ino;    // 16 bits
    char d_name[14];
};
```

In-class Exercise Revisited

0000000	00	01	2e	00	00	00	00	00	00	00	00	00	00	00	00	00
0000010	01	01	2e	2e	00	00	00	00	00	00	00	00	00	00	00	00
0000020	64	00	57	6f	6d	62	61	74	00	00	00	00	00	00	00	00
0000030	65	00	4d	61	63	6b	65	72	65	6c	00	00	00	00	00	00
0000040	66	00	44	6f	76	65	00	00	00	00	00	00	00	00	00	00
0000050	67	00	43	61	74	74	6c	65	00	00	00	00	00	00	00	00
0000060	00	00	44	6f	6d	65	73	74	69	63	20	72	61	62	62	69
0000070	02	01	4b	6f	69	00	00	00	00	00	00	00	00	00	00	00
0000080	00	00	52	61	62	62	69	74	00	00	00	00	00	00	00	00
0000090	68	00	43	6f	6e	64	6f	72	00	00	00	00	00	00	00	00
00000a0	69	00	44	6f	6e	6b	65	79	00	00	00	00	00	00	00	00
00000b0	03	01	41	6c	70	61	63	61	00	00	00	00	00	00	00	00
00000c0	6a	00	44	6f	6d	65	73	74	69	63	20	72	61	62	62	69
00000d0																

```
struct dirent {
    int d_ino;     // 16 bits
    char d_name[14];
};
```

In-class Exercise Revisited

0000000	00	01	2e	00	00	00	00	00	00	00	00	00	00	00	00	00
0000010	01	01	2e	2e	00	00	00	00	00	00	00	00	00	00	00	00
0000020	64	00	57	6f	6d	62	61	74	00	00	00	00	00	00	00	00
0000030	65	00	4d	61	63	6b	65	72	65	6c	00	00	00	00	00	00
0000040	66	00	44	6f	76	65	00	00	00	00	00	00	00	00	00	00
0000050	67	00	43	61	74	74	6c	65	00	00	00	00	00	00	00	00
0000060	00	00	44	6f	6d	65	73	74	69	63	20	72	61	62	62	69
0000070	02	01	4b	6f	69	00	00	00	00	00	00	00	00	00	00	00
0000080	00	00	52	61	62	62	69	74	00	00	00	00	00	00	00	00
0000090	68	00	43	6f	6e	64	6f	72	00	00	00	00	00	00	00	00
00000a0	69	00	44	6f	6e	6b	65	79	00	00	00	00	00	00	00	00
00000b0	03	01	41	6c	70	61	63	61	00	00	00	00	00	00	00	00
00000c0	6a	00	44	6f	6d	65	73	74	69	63	20	72	61	62	62	69
00000d0																

```
struct dirent {
    int d_ino;     // 16 bits
    char d_name[14];
};
```

Inode numbers

In-class Exercise Revisited

	Inode numbers	Names
0000000	00 01	2e 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010	01 01	2e 2e 00 00 00 00 00 00 00 00 00 00 00 00
0000020	64 00	57 6f 6d 62 61 74 00 00 00 00 00 00 00 00
0000030	65 00	4d 61 63 6b 65 72 65 6c 00 00 00 00 00 00
0000040	66 00	44 6f 76 65 00 00 00 00 00 00 00 00 00 00
0000050	67 00	43 61 74 74 6c 65 00 00 00 00 00 00 00 00
0000060	00 00	44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
0000070	02 01	4b 6f 69 00 00 00 00 00 00 00 00 00 00 00
0000080	00 00	52 61 62 62 69 74 00 00 00 00 00 00 00 00
0000090	68 00	43 6f 6e 64 6f 72 00 00 00 00 00 00 00 00
00000a0	69 00	44 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
00000b0	03 01	41 6c 70 61 63 61 00 00 00 00 00 00 00 00
00000c0	6a 00	44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
00000d0		

```
struct dirent {
    int d_ino;     // 16 bits
    char d_name[14];
};
```

Inode numbers   Names

In-class Exercise Revisited

```
0000000  00 01  2e 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010  01 01  2e 2e 00 00 00 00 00 00 00 00 00 00 00 00
0000020  64 00  57 6f 6d 62 61 74 00 00 00 00 00 00 00 00
0000030  65 00  4d 61 63 6b 65 72 65 6c 00 00 00 00 00 00
0000040  66 00  44 6f 76 65 00 00 00 00 00 00 00 00 00 00
0000050  67 00  43 61 74 74 6c 65 00 00 00 00 00 00 00 00
0000060  00 00  44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
0000070  02 01  4b 6f 69 00 00 00 00 00 00 00 00 00 00 00
0000080  00 00  52 61 62 62 69 74 00 00 00 00 00 00 00 00
0000090  68 00  43 6f 6e 64 6f 72 00 00 00 00 00 00 00 00
00000a0  69 00  44 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
00000b0  03 01  41 6c 70 61 63 61 00 00 00 00 00 00 00 00
00000c0  6a 00  44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
00000d0
```

```
struct dirent {
    int d_ino;     // 16 bits
    char d_name[14];
};
```

Let's figure out the name ~~and type~~ for inode 0x6a

Inode numbers

Names

Domestic rabbi

In-class Exercise Revisited

```
0000000 00 01 2e 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010 01 01 2e 2e 00 00 00 00 00 00 00 00 00 00 00 00
0000020 64 00 57 6f 6d 62 61 74 00 00 00 00 00 00 00 00
0000030 65 00 4d 61 63 6b 65 72 65 6c 00 00 00 00 00 00
0000040 66 00 44 6f 76 65 00 00 00 00 00 00 00 00 00 00
0000050 67 00 43 61 74 74 6c 65 00 00 00 00 00 00 00 00
0000060 00 00 44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
0000070 02 01 4b 6f 69 00 00 00 00 00 00 00 00 00 00 00
0000080 00 00 52 61 62 62 69 74 00 00 00 00 00 00 00 00
0000090 68 00 43 6f 6e 64 6f 72 00 00 00 00 00 00 00 00
00000a0 69 00 44 6f 6e 6b 65 79 00 00 00 00 00 00 00 00
00000b0 03 01 41 6c 70 61 63 61 00 00 00 00 00 00 00 00
00000c0 6a 00 44 6f 6d 65 73 74 69 63 20 72 61 62 62 69
0
```

```
struct dirent {
    int d_ino;      // 16 bits
    char d_name[14];
};
```

Usable? No. Even back then it chafed to have tiny file names.

Compact? Yes.

BTW: Also simple to implement!