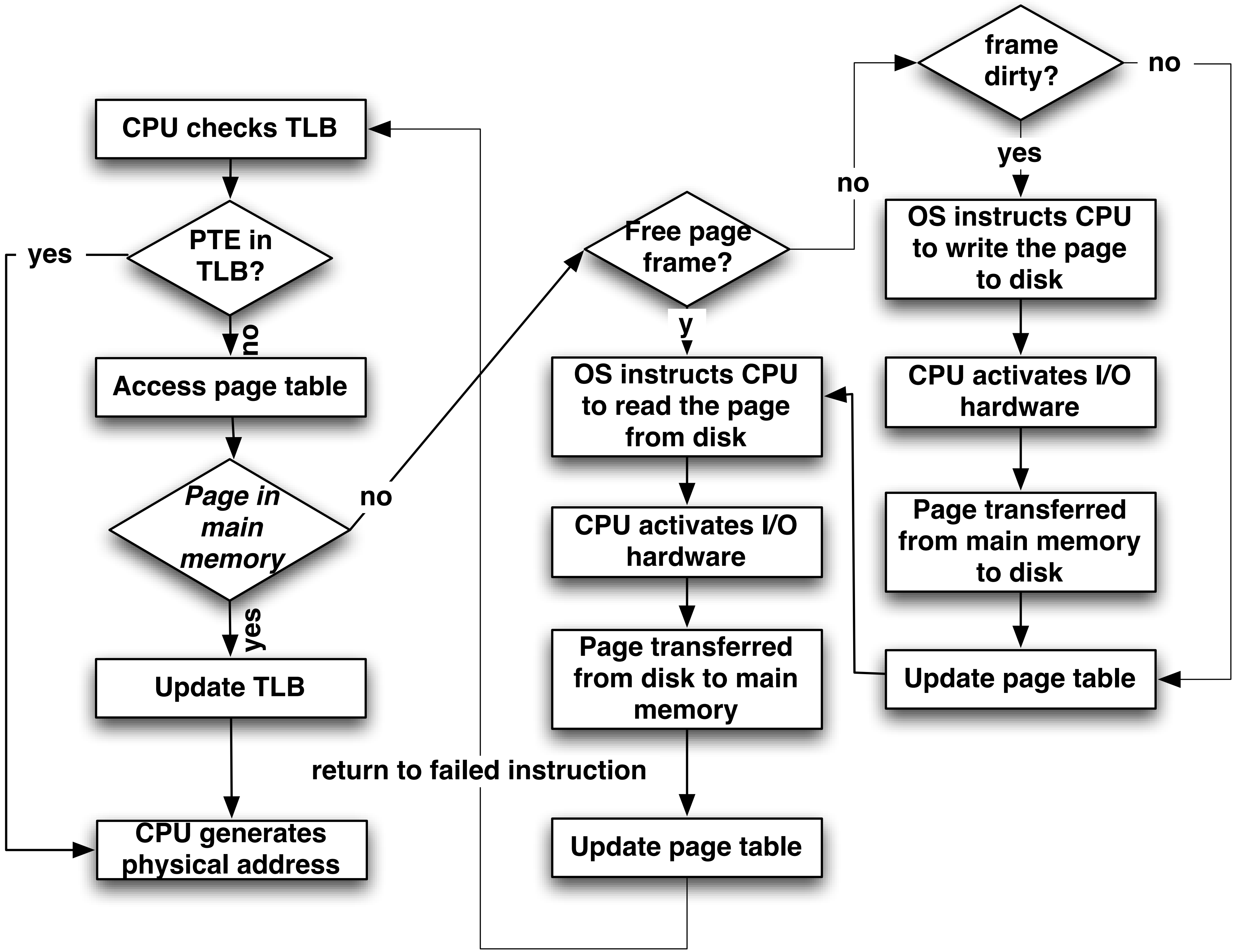
CS 134
Operating Systems

March 5, 2019

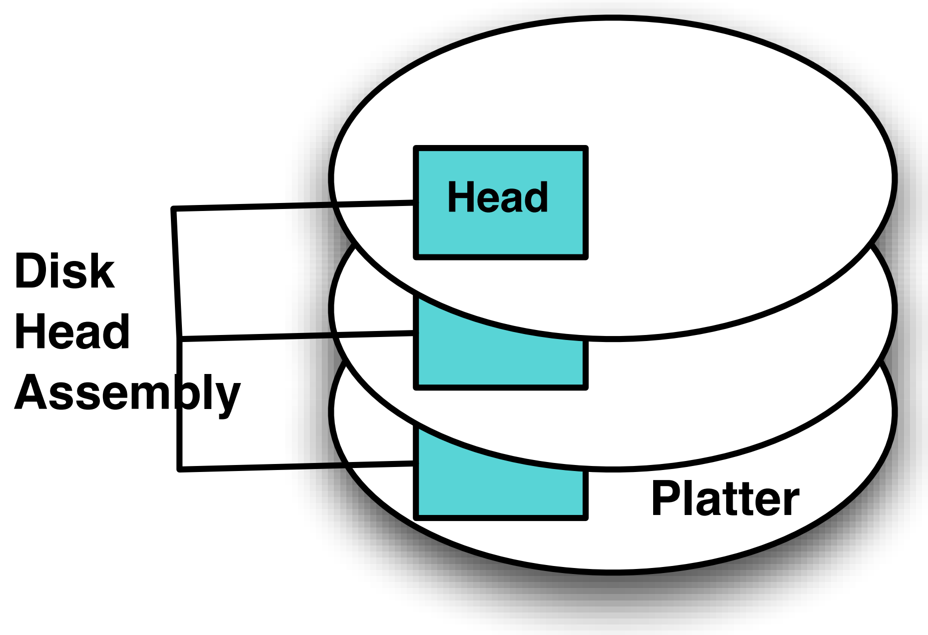
File System (2/2)

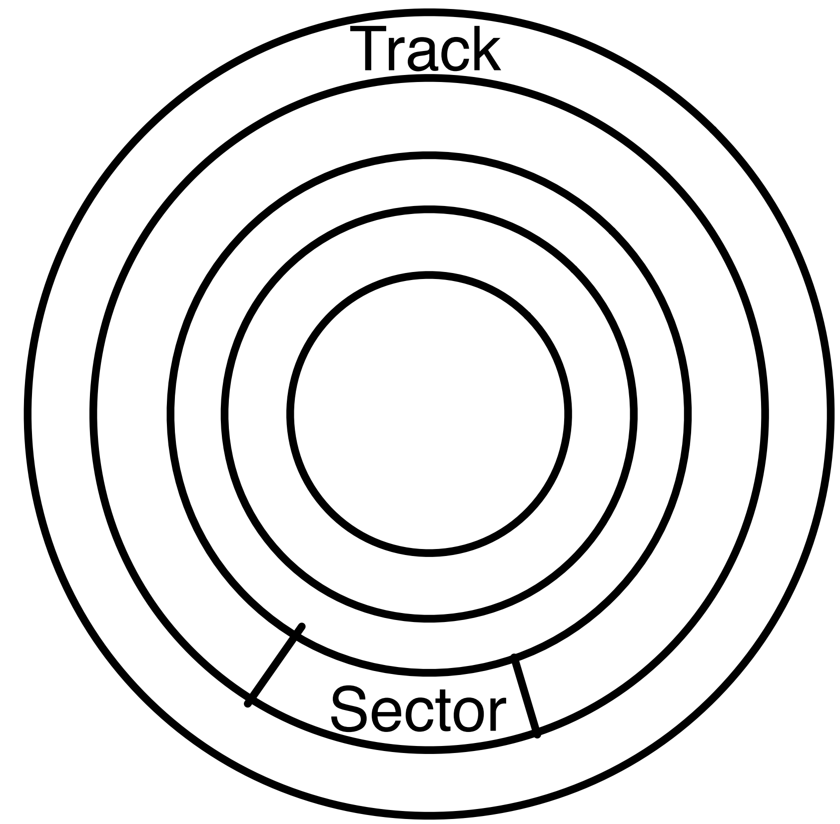
Paging and Translation Lookaside Buffer (TLB)



Disks

- Hardware





- Time to access a block (sector)
  - Seek time (time to move the head in or out to the appropriate track)
  - Rotational latency (time for the disk to spin so that the beginning of the sector is under the head)
  - Transfer time (time for the data to be read from the sector).

Identifying a block

- Logical Block Number (LBN): 1-N
- Maps to: Cylinder/head/sector
- Who does the mapping?

Blocks go bad

- Blocks written with ECC
  - Soft error
  - Hard error
- Fixes:
  - Sector sparing
  - Sector slipping

Reading/Writing a block

- Seek first
- Wait for sector to rotate under head
- Read (or write)

Disk Specs

	Western Digital VelociRaptor WD1500AHFD	Seagate Desktop HDD 1.5
Capacity	1 TB	4 TB
Rotational Speed	10,000 RPM	5,900 RPM
Average rotational latency	3 ms	4 ms
Average access time (seek + rotational)	6.8 ms	17 ms
Average sustained transfer rate	164 MB/s	132 MB/s
Buffer size	64 MiB	64 MiB

Disk Scheduling

- FCFS
- Shortest-seek time first (SSTF)
- SCAN (elevator)
- C-SCAN

Solid State Drives (SSD)

- Non-volatile “flash” memory
- Random access: 100 microseconds
- Sequential: 500 MB/sec
- Internally complex
  - Flash must be erased before it is written
  - Limit to the number of times a flash block can be written
  - SSD remaps blocks as necessary

Disk blocks

- Most OSes use blocks of multiple sectors
  - e.g., 4 KB block = 8 sectors
  - to reduce bookkeeping and seek overheads
  - xv6 uses single-sector blocks for simplicity

High-level choices visible in the Unix FS API

- Object: files (vs. virtual disk/DB)
- Content: byte array (vs. 80-byte records, BTree)
- Naming: human-readable (vs. object IDs)
- Organization: name hierarchy
- Synchronization: none (vs. locking, versions)
- There are other (sometimes *quite* different) file system APIs

A few implications of the Unix API

- FD refers to something
  - that is preserved even when the name changes
  - or if file is deleted while open
- A file can have multiple (hard) links
  - i.e., occur in multiple directories
  - no one of those occurrences is special
  - so file must be stored somewhere other than directory
- Thus:
  - FS records file info in an *inode* on disk
  - FS refers to inode with i-number (internal version of FD)
  - inode must have link count (tells us when to free)
  - inode must have count of open FDs.
  - inode deallocation deferred until last link and FD are gone)

xv6

- FS software layers
  - system calls
  - name ops/FD ops
  - inodes
  - inode cache
  - log
  - buffer cache
  - IDE driver

On-disk layout

- xv6 file system on 2nd IDE disk drive
  - First just has the kernel
- xv6 treats drive as an array of sectors (ignores tracks)

Block num	Usage
0	unused (usually boot block)
1	super block
2	log for transactions
32	array of inodes, packed into blocks
58	Block in-use bitmap (0=free, 1=used)
59	file/dir content blocks
…	…

Mkfs

- xv6's mkfs program generates this layout for an empty file system
- This layout is static for the lifetime of the file system

On-disk inode

```
#define NDIRECT 12
// On-disk inode structure
struct dinode {
  short type;              // File type
  short major;             // Major device number (T_DEV only)
  short minor;             // Minor device number (T_DEV only)
  short nlink;             // Number of links to inode in file system
  uint size;               // Size of file (bytes)
  uint addrs[NDIRECT+1];// Data block addresses
};
```

How to find block number containing byte 8000 of a file:

logical block number: 8000/512 = __________

Find actual block number: 3rd entry in the indirect block (@ addrs[12])

Each inode has an inumber

- Easy to turn inumber into inode
  - inode is 64 bytes long
  - can store 8 per block (`IPB=      /                   `)
  - block num on disk: `32 + inumber/8`
  - Offset in block = `(inumber % 8) * 64`

Directory contents

- Contents is an array of dirent

```
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

- dirent is free if inum is 0

xv6 in action

- Focus on disk writes
- Illustrate on-disk data structures via how updated

How does xv6 create a file?

```
rm fs.img
make qemu-nox-gdb
…
$ echo > a
```

What is written

blocknum	func	called from	what
34	ialloc	create	mark inode allocated
34	iupdate	create	initialize nlink, major,
59	writei	dirlink	write inumber and name

- call graph:
  - sys_open sysfile.c
    - create sysfile.c
      - ialloc fs.c
      - iupdate fs.c
      - dirlink fs.c
        - writei fs.c

What's in block 34?

```
create(…) {
  …
  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);
  …
}
```

Why two writes to block 34?

Why 34 if inodes start at block 32?

What's in block 59?

```
$ ls
.                    1 1 512
..                   1 1 512
README               2 2 2327
cat                  2 3 15544
echo                 2 4 14440
forktest             2 5 8864
grep                 2 6 17552
init                 2 7 15068
kill                 2 8 14484
ln                   2 9 14364
ls                   2 10 16884
mkdir                2 11 14592
rm                   2 12 14568
sh                   2 13 26740
stressfs             2 14 15344
usertests            2 15 63548
wc                   2 16 16152
zombie               2 17 14176
console              3 18 0
a                    2 19 0
```

1	.
1	..
⋮	⋮
18	console
19	a

What if there are concurrent calls to `ialloc`?

```
void ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

How does xv6 write data to a file?

```
$ echo foo > a
```

What is written

blocknum	func	called from	what
58	balloc	bmap	mark block allocated
640	bzero	balloc	empty data block
640	writei	filewrite	write “a”
34	iupdate	writei	update size to 1 and addrs
640	writei	filewrite	write “a\n”
34	iupdate	write	update size to 2

- call graph:
  - sys_write sysfile.c
    - filewrite file.c
      - writei fs.c
        - bmap fs.c
          - balloc fs.c
            - bzero fs.c
      - iupdate fs.c

What's in block 58?

```
bmap(struct inode *ip, uint bn)
{
  …

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
```

```
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

What's in block 640?

```
a\n\0\0\0...\0
```

Why two calls to `writei`?

Why two calls to `updatei`?

How does xv6 delete a file?

```
$ rm a
```

blocknum	func	called from	what
59	writei	sys_unlink	clear dirent
34	iupdate	sys_unlink	nlink—
58	bfree	itrunc	mark block free
34	iupdate	itrunc	Size→ 0, addrs→0
34	iupdate	iput	mark not valid

What is written

- call graph:
  - sys_unlink sysfile.c
    - writei sysfile.c
      - iupdate fs.c
      - iunlockput fs.c
        - iput fs.c
          - itrunc fs.c
            - bfree fs.c
            - iupdate fs.c
          - iupdate fs.c

Block cache (bio.c)

- Block cache holds a few recently-used blocks

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

Block cache

- FS calls `bread`, which calls `bget`
  - `bget` looks to see if block is already cached
  - If present, acquire lock and then return it
  - `b->refcnt++` prevents buf from being recycled while we're waiting

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  …
}
```

Block cache

- FS calls `bread`, which calls `bget`
  - If block not already cached, reuse an existing buffer
  - `b->refcnt=1` prevents buf from being recycled while we're waiting

```
static struct buf* bget(uint dev, uint blockno)
{
  …
  // Not cached; recycle an unused buffer.
  // Even if refcnt==0, B_DIRTY indicates a buffer is in use
  // because log.c has modified it but not yet committed it.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

Two levels of locking

- `bcache.lock` protects the description of what's in the cache

- `buf->lock` protects just the one buffer

What is the block cache replacement policy?

- LRU (Least Recently Used)
- `bget` reuses the tail (`bcache.head.prev`)
- `brelse` moves block to `bcache.head.next`

```
// Release a locked buffer.
// Move to the head of the MRU list.
void
brelse(struct buf *b)
{
  …

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock);
}
```

What if lots of processes need to read the disk?

- Who goes first?
  - `iderw` appends to idequeue list
  - `ideintr` calls `idestart` on head of ideqeuue list
  - So, FIFO