CSE 333
Lecture 9 - storage

Steve Gribble

Department of Computer Science & Engineering

University of Washington

Administrivia

Colin's away this week

- Aryan will be covering his office hours (check the schedule for the location)

Reminder about coding exercises

- the way to build intuition and skill in systems programming is to **write a lot of code**
- we strongly advise you to do **all** of the exercises
  - this means writing your own solution ***before*** looking at ours! :)

Administrivia

HW2 is out today

- more complex than HW1
  - you will finish our implementation of a file system crawler, indexer, and query processor (i.e., a search engine!)
  - you will need to teach yourself about several system calls along the way (we tell you which man pages to read)
  - there is a more code for you to read and understand
- please, please, please
  - **start early** and come see us when you run into issues!

Administrivia

HW2 teams

- you can work solo if you want
- or, you can team up with somebody else (teams of 2)
  - you need to find a teammate; you can use the discussion board
- if you work in a team, you need to be together when you code
  - one of you writes code, the other watches and suggests/bughunts
  - also, one of you must code parts A & C, the other codes B & D

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

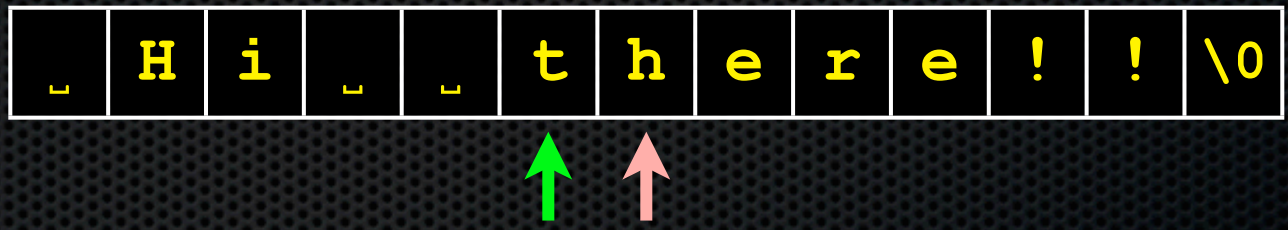
In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

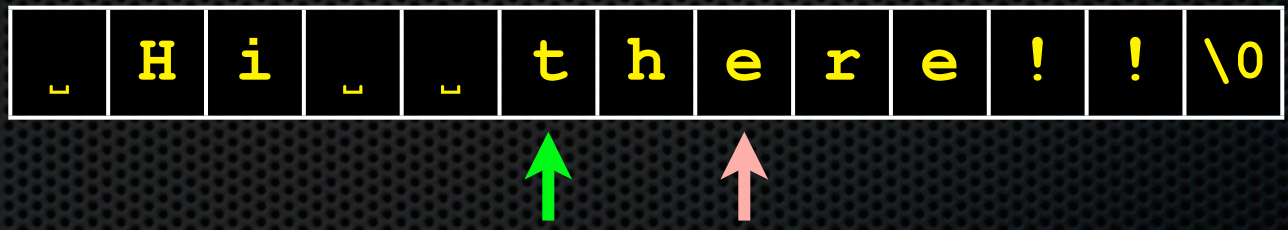


↑ = word start

↑ = word end

HW2: parsing a text file

In part B, we ask you to walk through a giant in-memory C string, extracting out individual words. To do it, you'll "walk" two pointers down the string in place.

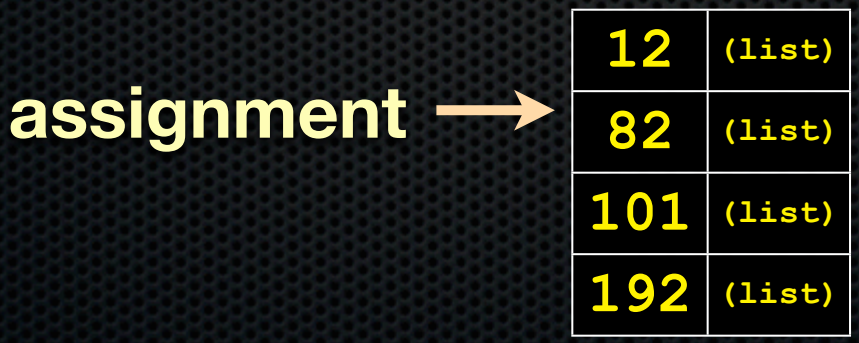


↑ = word start

↑ = word end

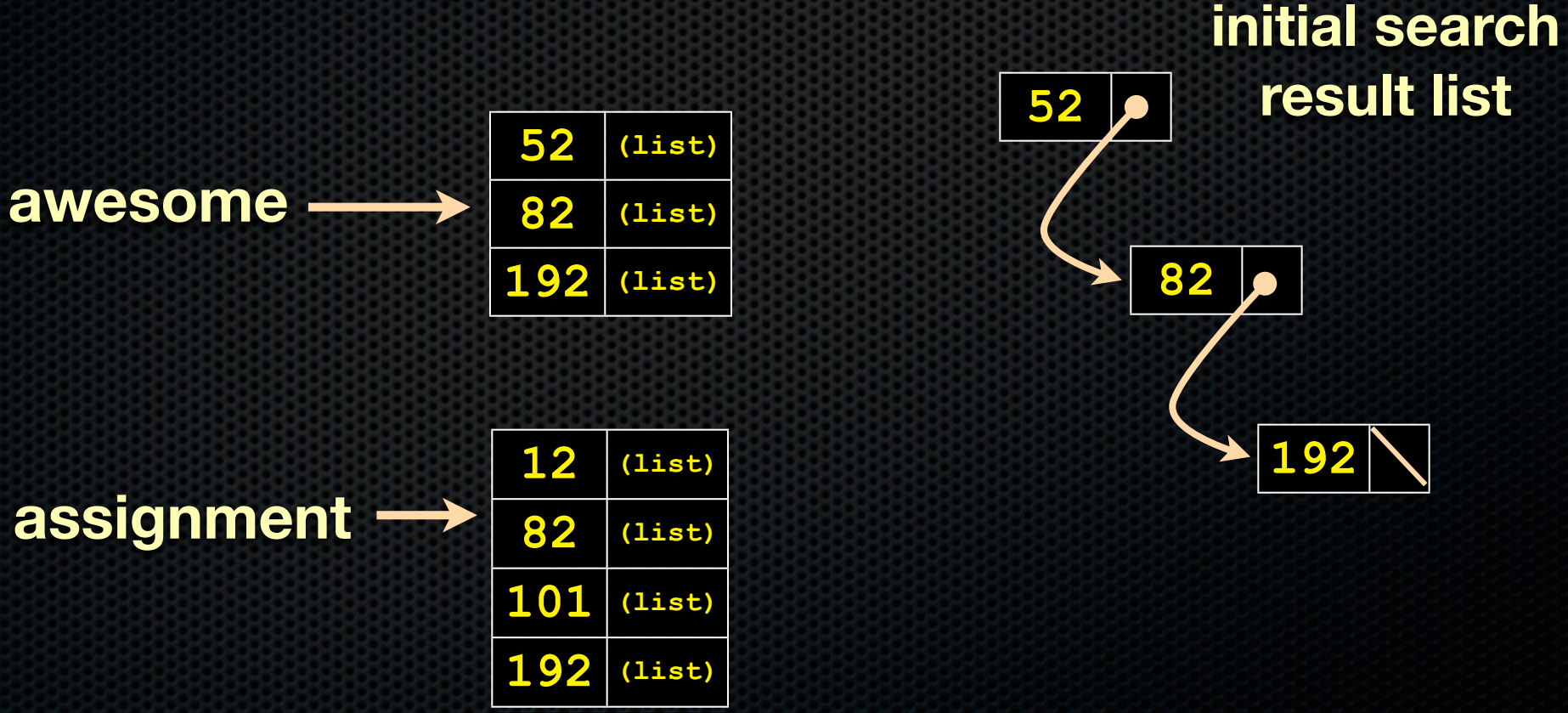
HW2: parsing a text file

In part C, we ask you to intersect two "posting lists" when processing a query



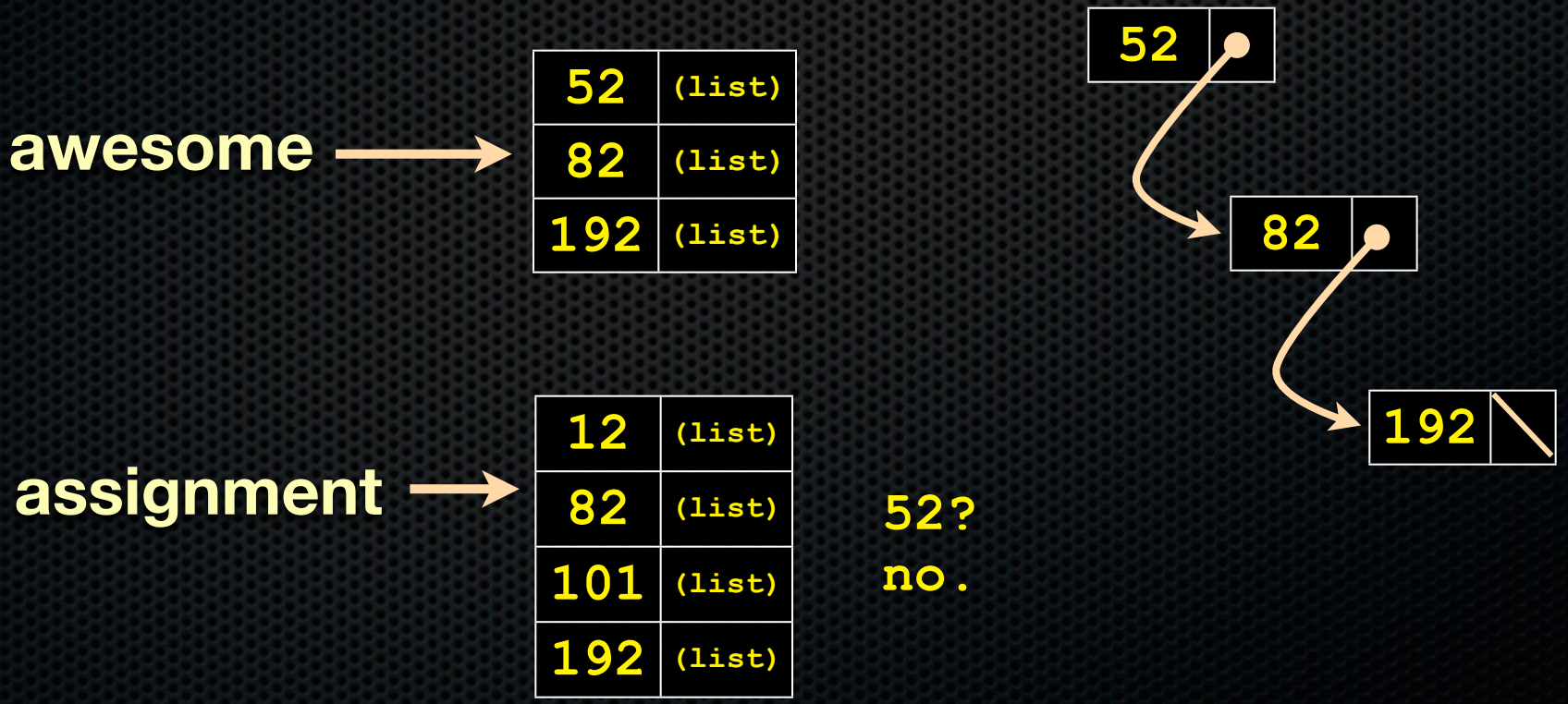
HW2: parsing a text file

In part C, we ask you to intersect two "posting lists" when processing a query



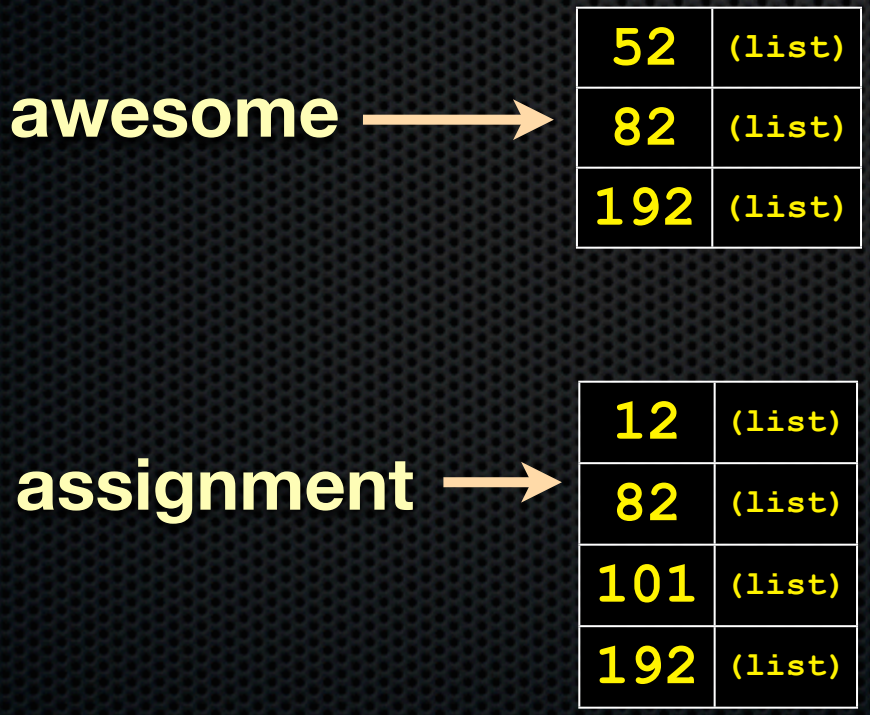
HW2: parsing a text file

In part C, we ask you to intersect two "posting lists" when processing a query



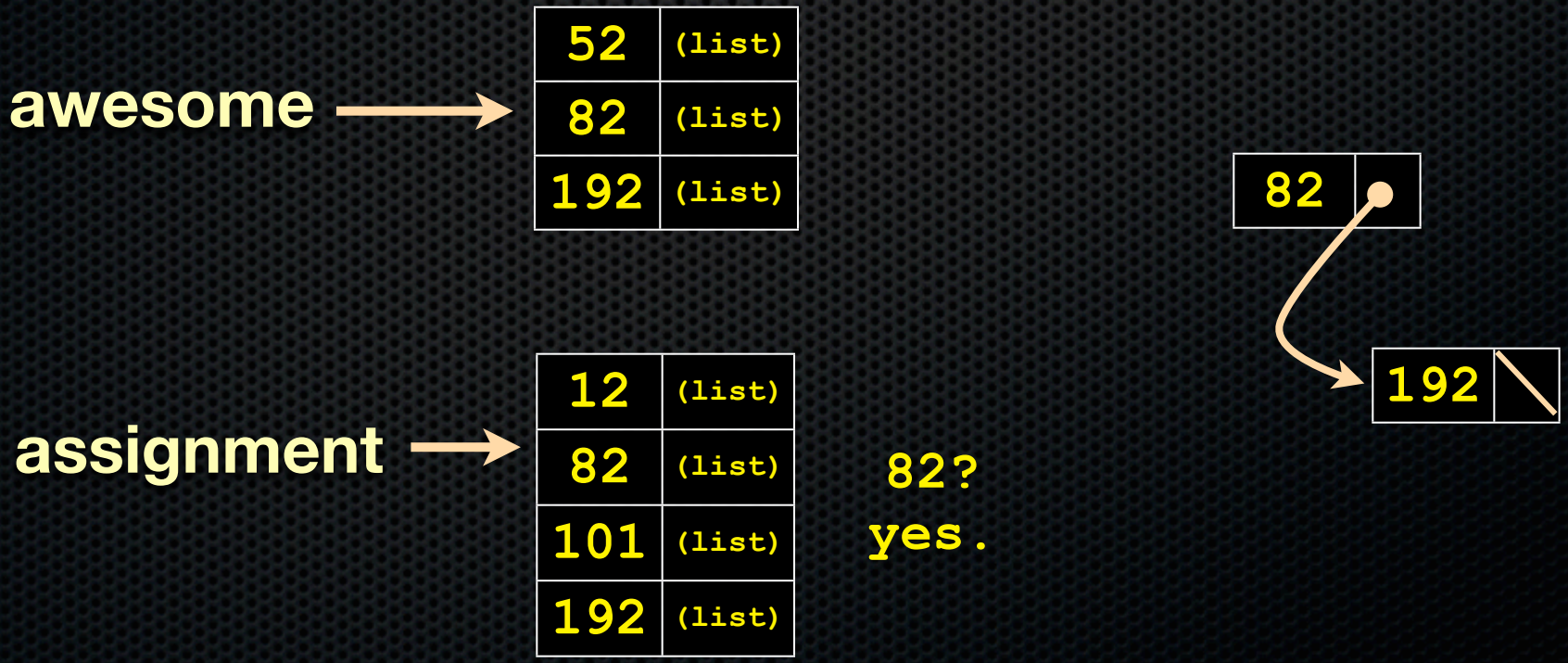
HW2: parsing a text file

In part C, we ask you to intersect two "posting lists" when processing a query



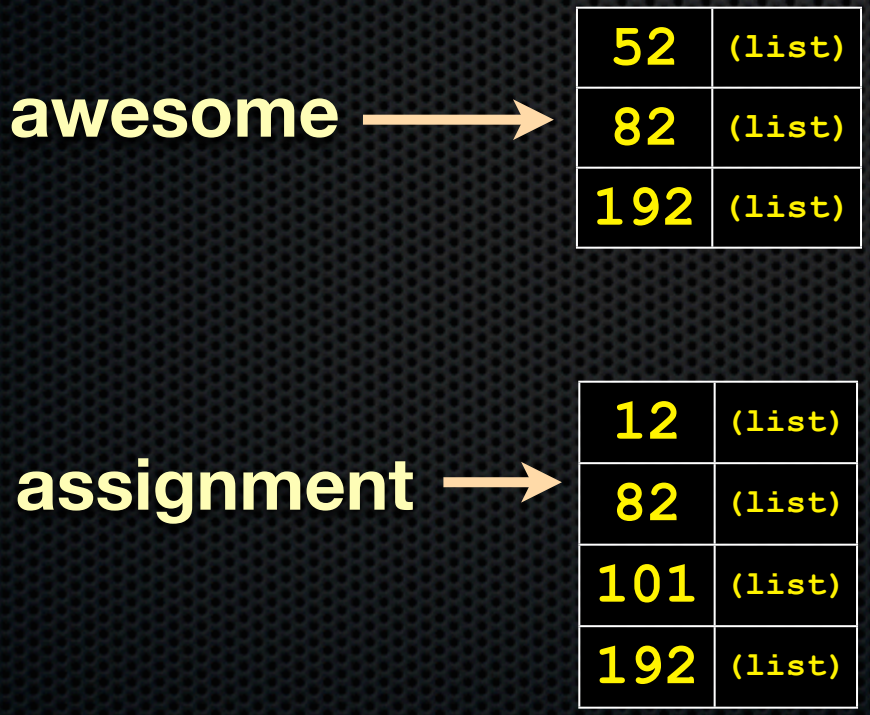
HW2: parsing a text file

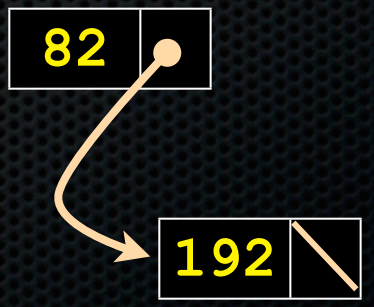
In part C, we ask you to intersect two "posting lists" when processing a query



HW2: parsing a text file

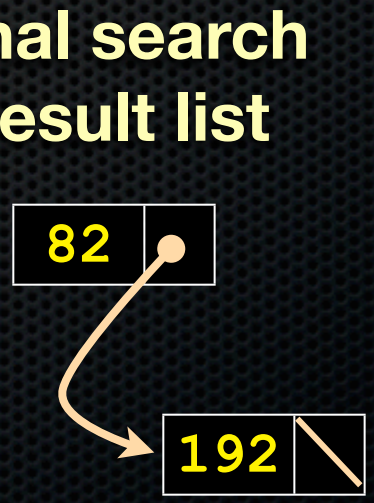
In part C, we ask you to intersect two "posting lists" when processing a query





HW2: parsing a text file

In part C, we ask you to intersect two "posting lists" when processing a query



HW2: ugly hack

```
#include "ll.h"

void LLNullFree(void *el) { }

int main(int argc, char **argv) {
  int res = 52;
  LinkedList ll = AllocateLinkedList();
  assert(ll != NULL);

  // Store the some ints in the linked list without
  // needing to call malloc.  How?  By abusing
  // type casting and casting an (int) to a (void *).
  // UGLY HACK ALERT!   Q: when is this safe?
  PushLinkedList(ll, (void *) res);
  PushLinkedList(ll, (void *) 87);
  PopLinkedList(ll, (void **) &res);

  // Free the linked list.  Since the payload is
  // not a pointer to heap-allocated memory, our
  // free function should do nothing.
  FreeLinkedList(ll, &LLNullFree);
  return 0;
}
```

HW2

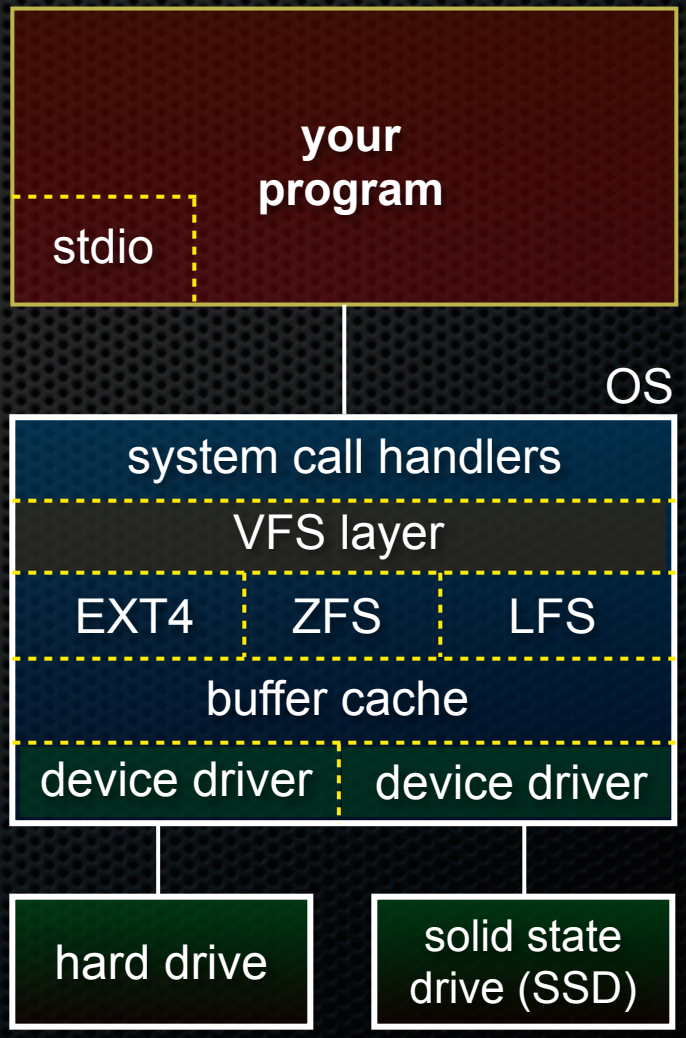
We provide you with our libhw1.a

- AFAQ: “test_suite crashes inside InsertHashTable( ). I think this means your libhw1.a has a bug in it.”
  - probably not; more likely it means that your code has a bug in it that stomps over the memory that libhw1.a relies on
  - but, if you really think we have a bug in our libhw1.a, send us the simplest piece of code that replicates the problem, and we’ll check

The storage "stack"

Like most systems, has many, many layers of abstraction

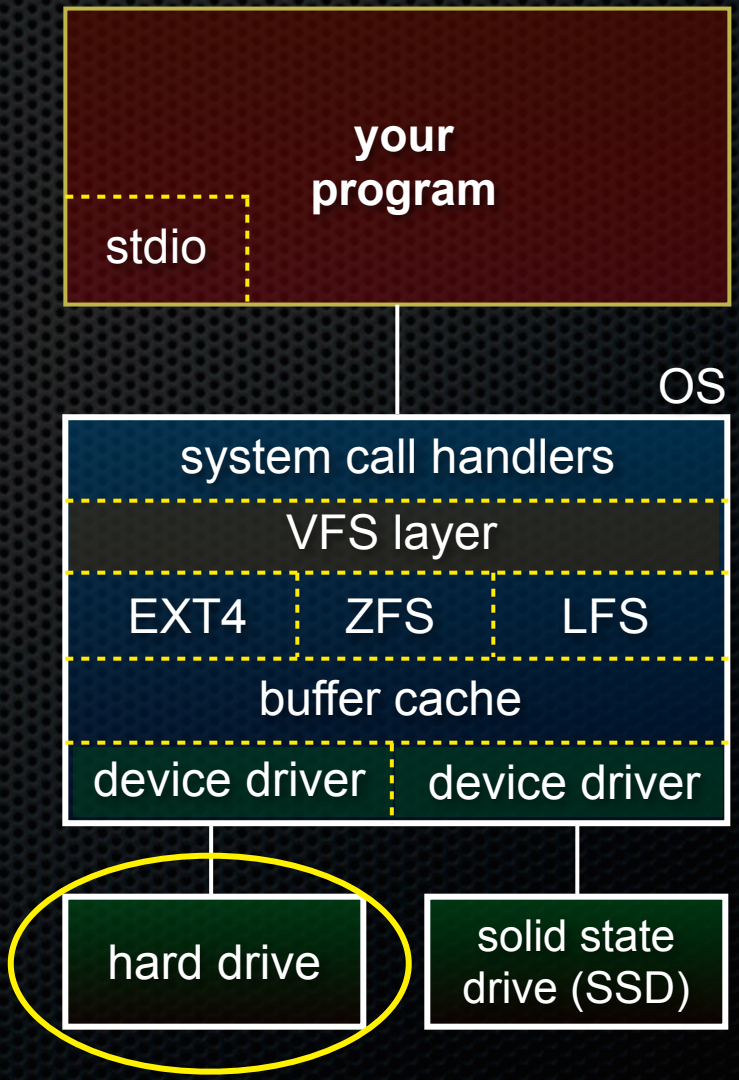
- lots of complexity, but each layer is understandable on its own
- layer X
  - relies on the features of layer X-1
  - provides more features to layer X+1



Storage hardware

Hard drive

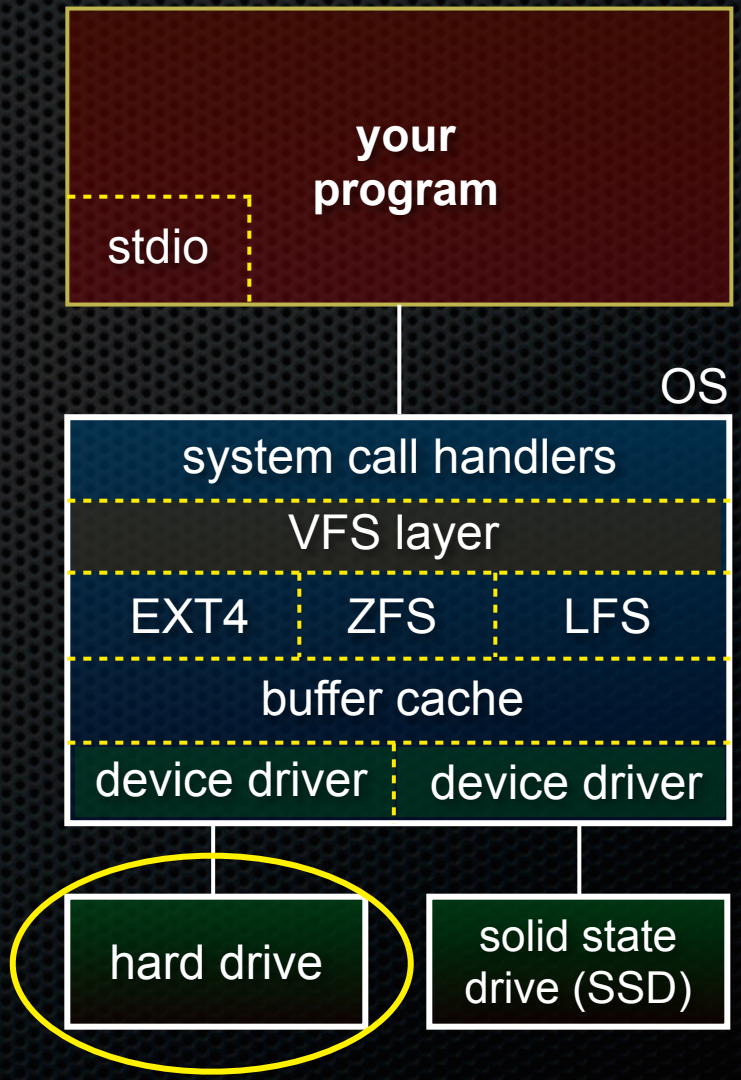
- spinning magnetic platters
  - spins at ~7200 RPM
- read/write head on an arm
  - moves back and forth; ~5ms to move to a new location



Storage hardware

Hard drive characteristics

- exponentially cheaper capacity
  - 1TB = $60; ~2x every 18 months
- great "sequential" bandwidth
  - ~200MB/s, improving exponentially along with capacity
- terrible "random" bandwidth
  - ~1MB/s, not improving, since it's mechanically limited
- this difference dominates the design of higher layers



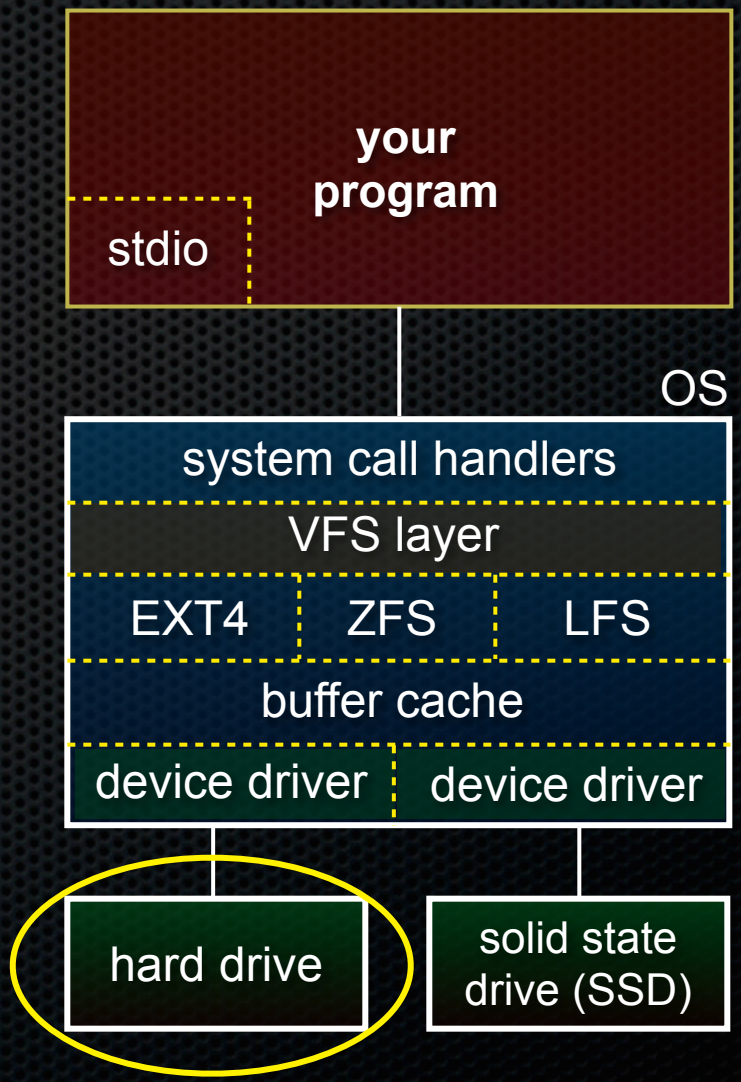
Storage hardware

Hard drive interface

- an array of 512 byte sectors
- read / write entire sector at a time

Hard drive internals

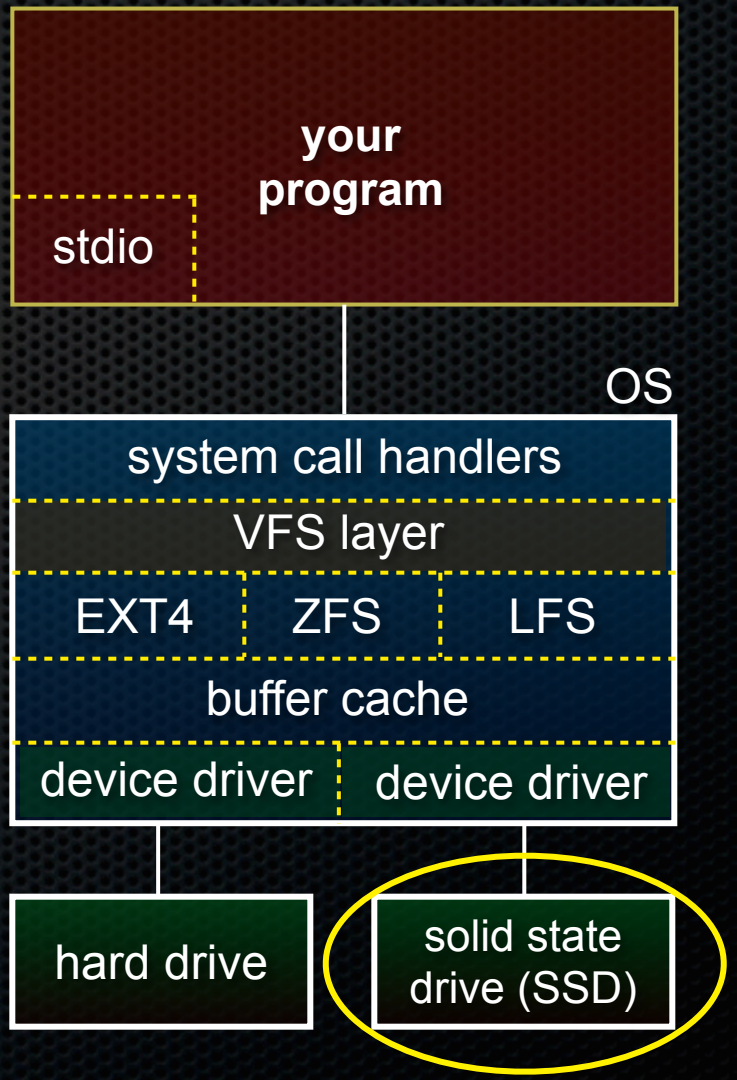
- remaps bad sectors
  - sequentiality can be tricky
- has an on-controller RAM buffer
  - writes may indicate completion before they hit the platter!



Storage hardware - SSDs

banks of NAND flash chips

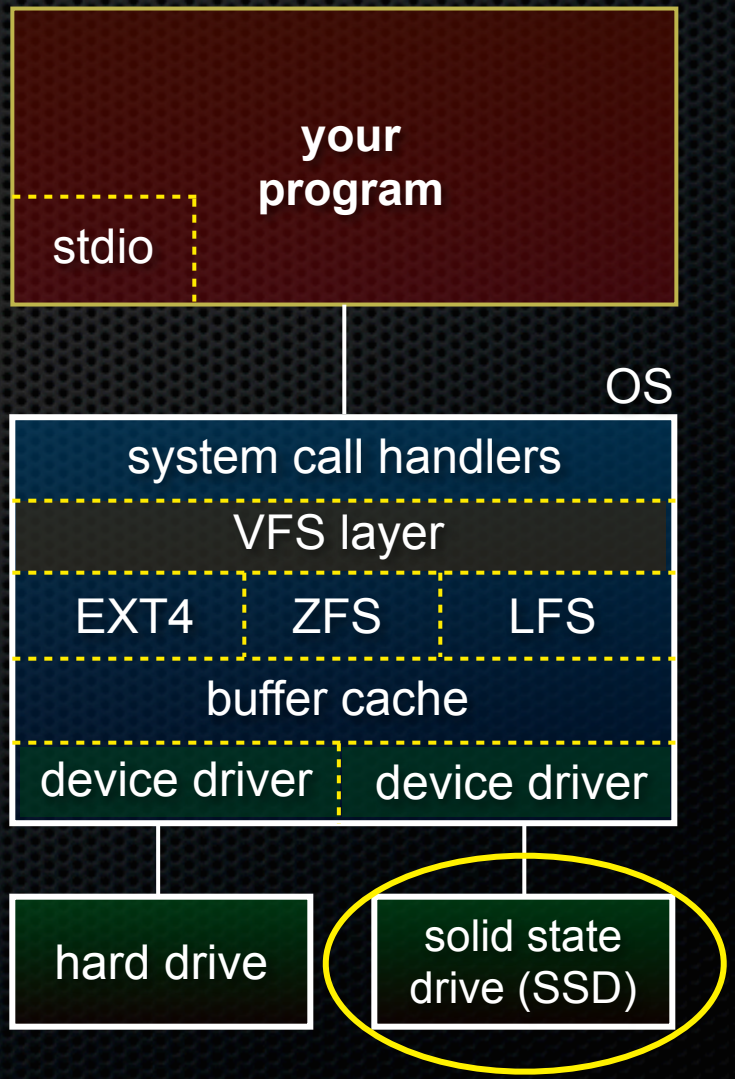
- unit of read/write is ~4KB *page*
  - before write, must *erase* entire ~512KB block to all 1s, then can set individual bits to 0
  - limited # of writes per block
- no mechanical parts!



Storage hardware - SSDs

SSD characteristics

- 20x more expensive than HD
  - 1TB = $2K; ~2x better per year
- fantastic read bandwidth
  - ~40K IOPS, ~250MB/s
  - same for random & sequential!
- good sequential write bandwidth
  - ~30K IOPS, ~175 MB/s
- but, random writes are slower
  - ~3K IOPS, ~10 MB/s



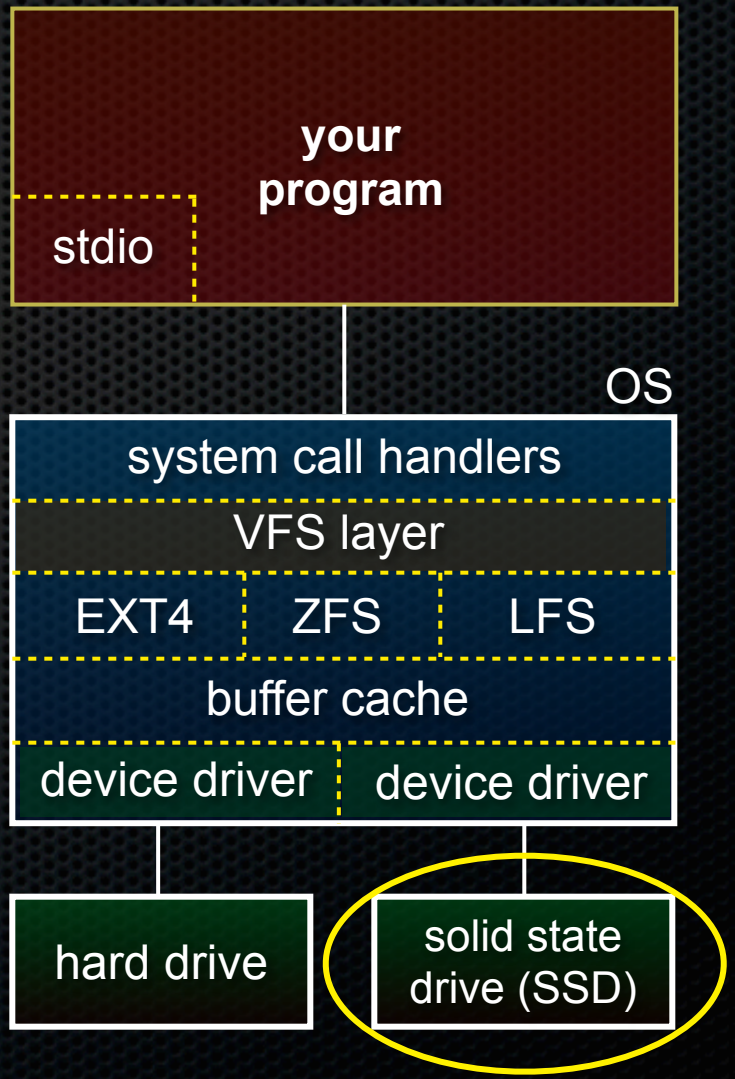
Storage hardware - SSDs

SSD interface

- an array of 4096 byte page
- read / write entire page at a time

SSD internals

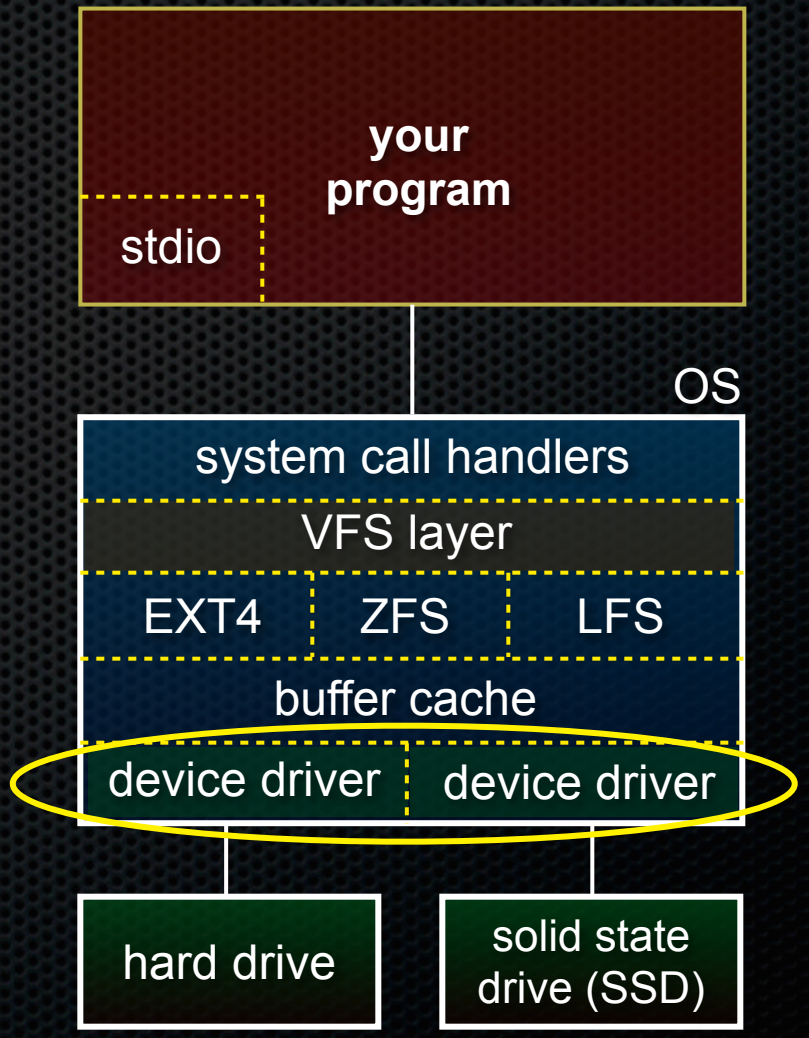
- flash translation layer (FTL)
  - wear leveling, background erasing & remapping to maintain a pool of writeable blocks



Device drivers

Software layer at bottom of OS

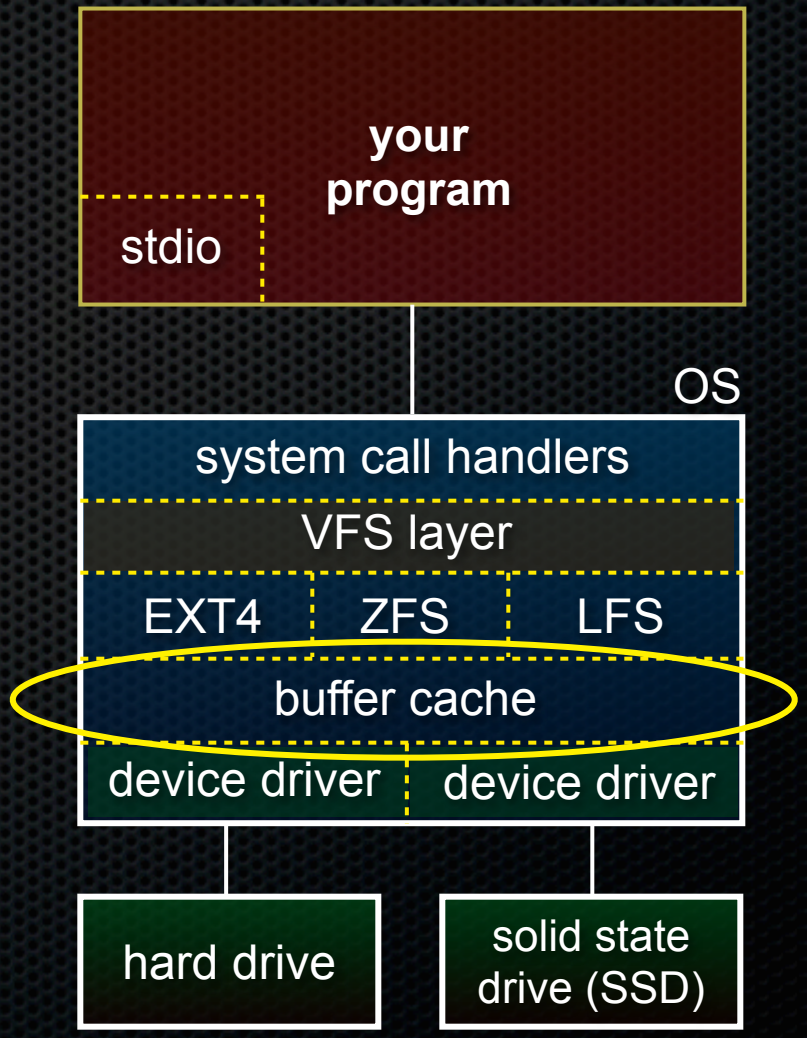
- abstracts away the details of communicating with different storage interfaces
  - IDE, SCSI, etc.
- probes the device to learn its characteristics
- permits higher-level software to issue commands to read and write *blocks*



Buffer cache

OS-managed pool of memory

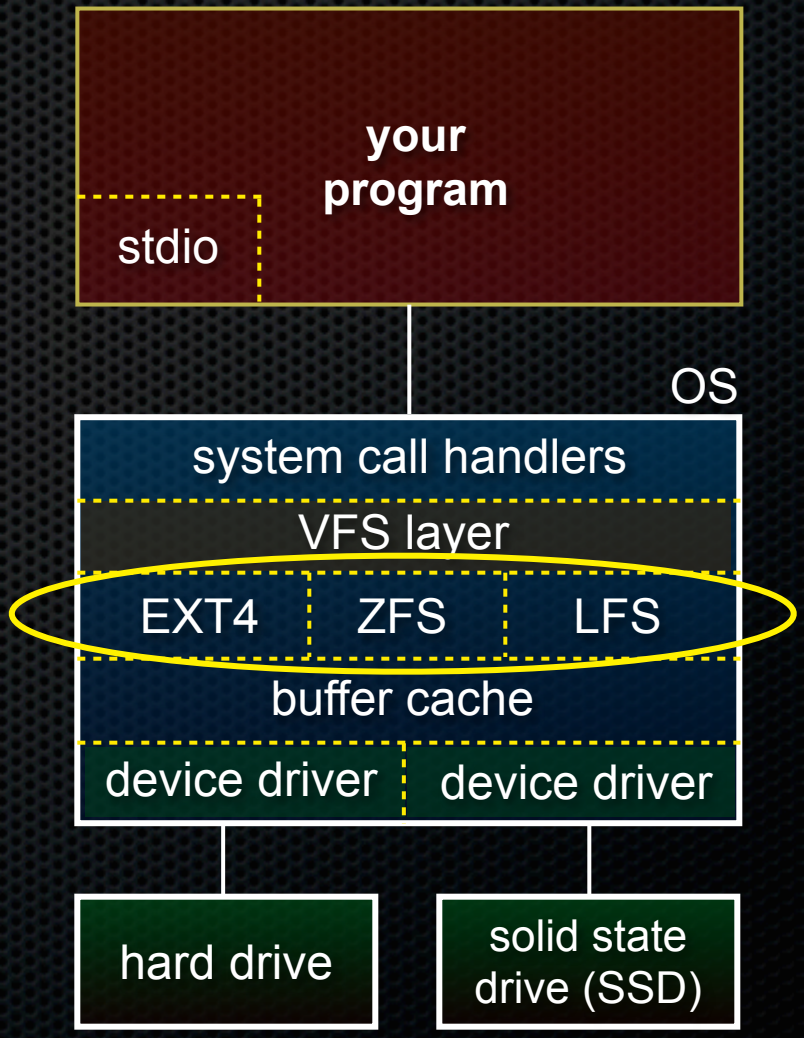
- stores recently read disk blocks
  - speed up re-reads by fetching recently read data from cache
- accumulates writes in buffer cache, eventually write back
  - reduces traffic via coalescing
  - batches, reorders writes to attempt to induce more sequential I/O
- can introduce reliability problems on OS crash, HW power loss



File system

Abstracts away disk blocks into files and directories

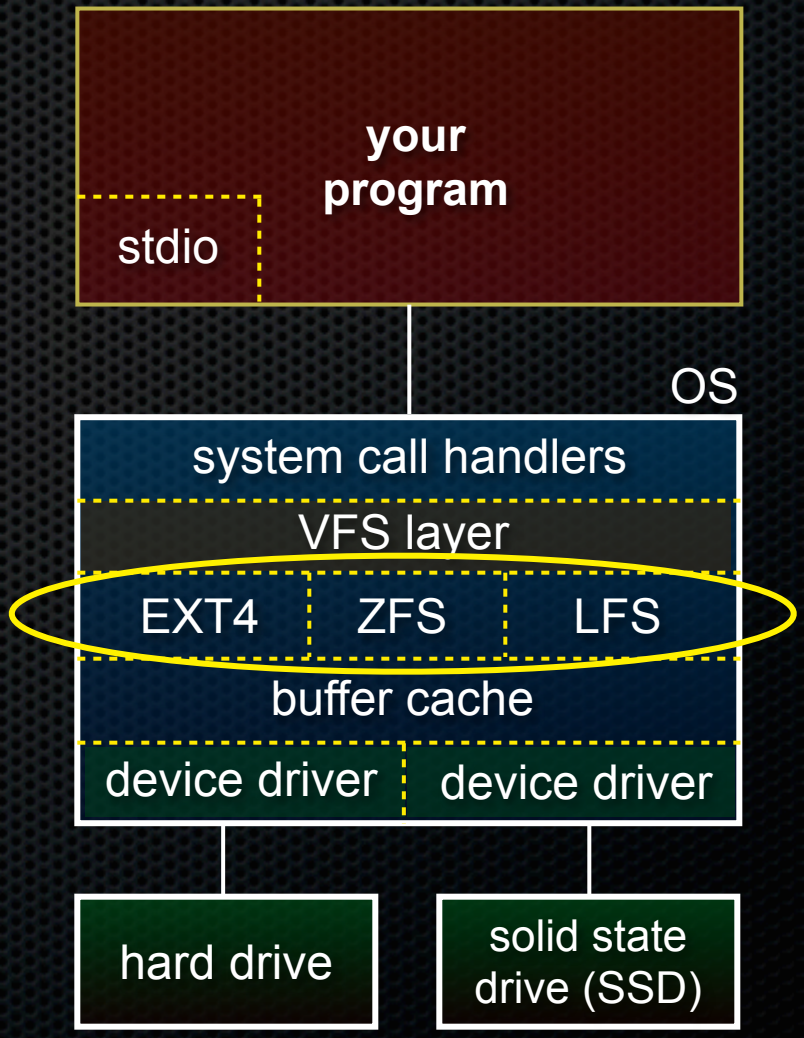
- at its core, is just maintains a data structure that lives on disk
- FS is tree of files & directories
  - a file is a tree of disk blocks
    - the root of tree is the **inode**; inode contains file metadata rather than data
  - a directory is a file
    - contains a table mapping names to inodes



File system

There are many file systems

- they differ in how they lay out the data structure on disk
  - has big performance implications
  - a good FS attempts to preserve locality, sequentiality in the layout
- they differ in how they order operations, flush the buffer cache
  - tradeoffs between consistency of the file system, performance, and the delay before writes are durable
- some permit snapshots, versions, and other features



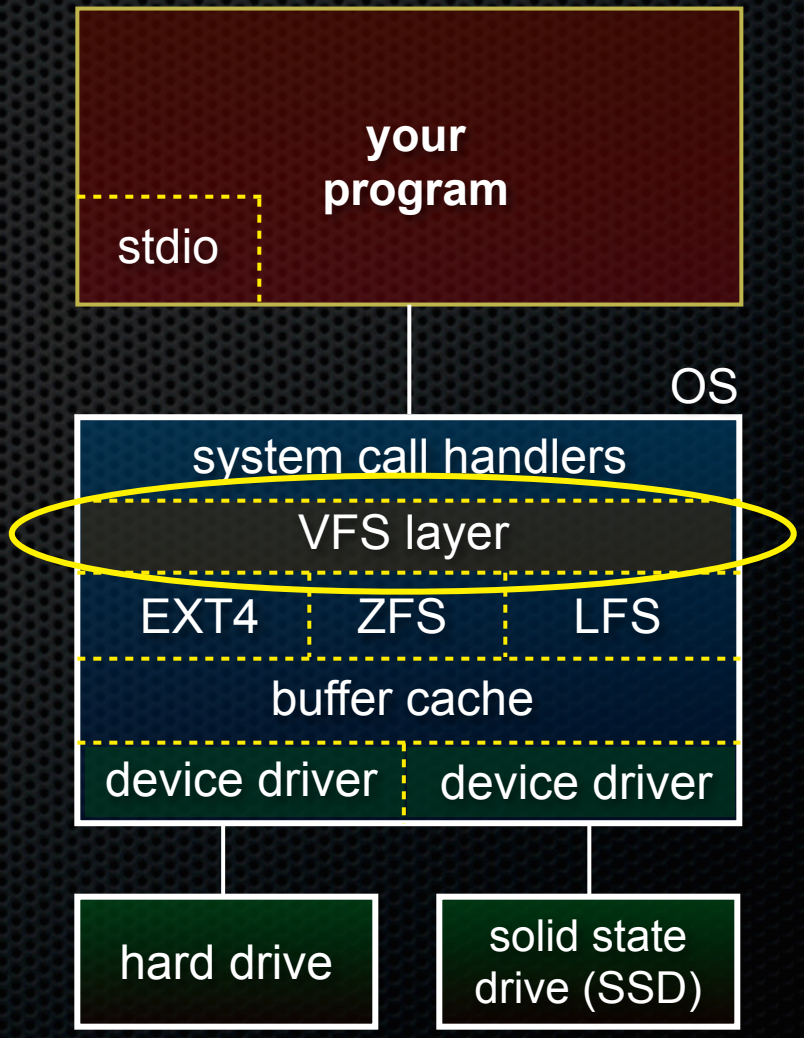
VFS layer

Level of indirection between OS API and specific file systems

- permits multiple file systems to co-exist within your computer
  - provides an API that lets concrete file system plugs into VFS
  - provides a single, uniform API to the higher layers of the OS

Why multiple file systems?

- mount multiple storage devices, drives with multiple partitions, USB thumbdrives, NFS, etc.



System calls

basic read / write operations

- open( ), read( ), write( ), close( ), ..

seek within a file

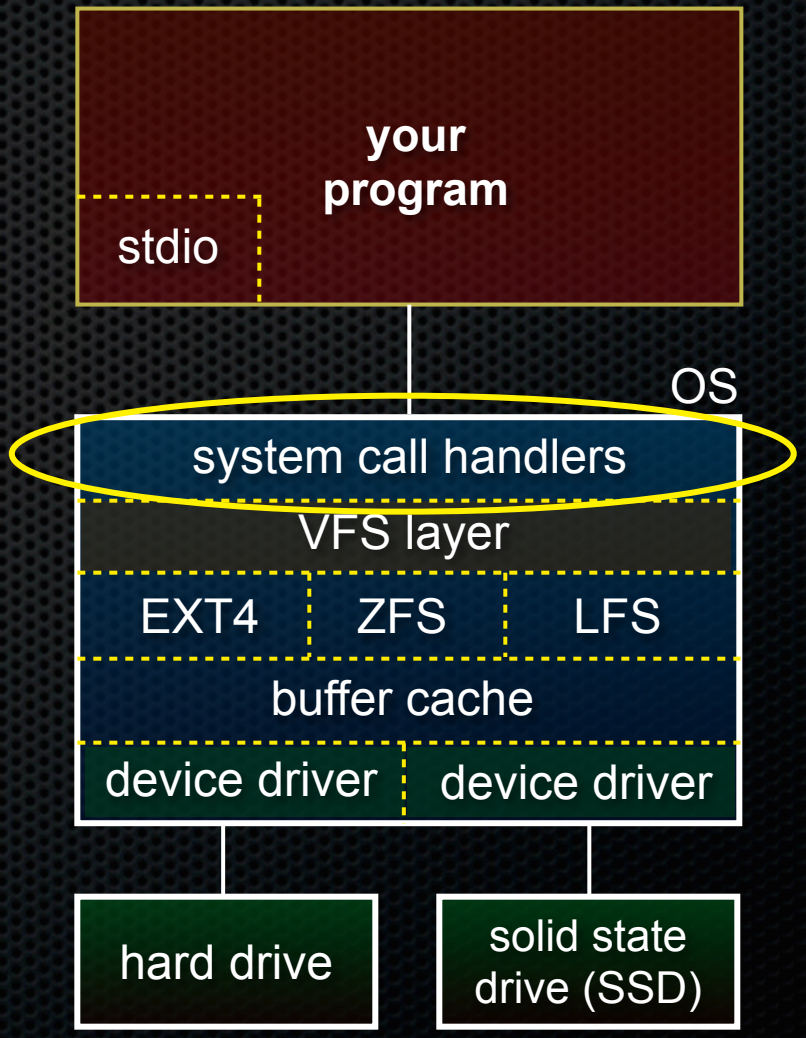
- lseek( ), ...

ability to flush dirty data from buffer cache to disk

- fflush( ), sync( )

manage access permissions

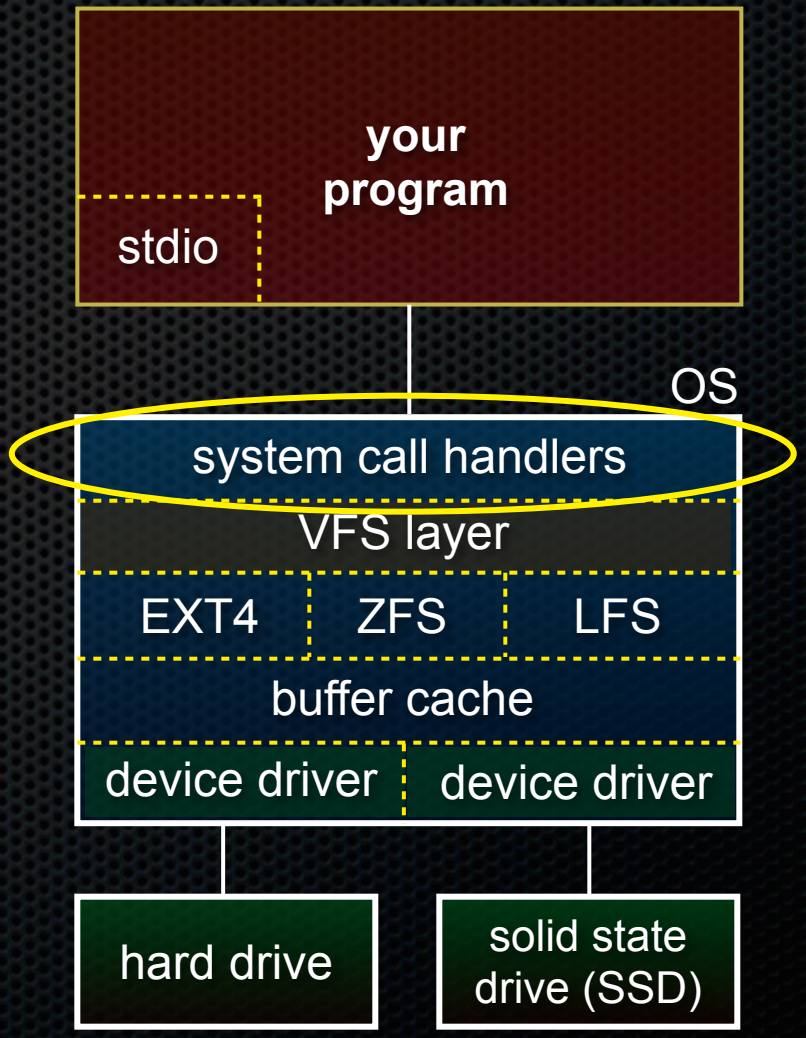
- chmod( ), chown( ), ...



System calls

Two basic styles of doing file I/O

- **blocking I/O**
  - the system call waits until the I/O completes before returning
  - the thread of execution that invoked the system call stalls until the call completes
- **non-blocking I/O**
  - system call returns immediately
    - a completion event fires later
  - thread of execution can juggle multiple, concurrent tasks



Exercise 1

Write a program that, similar to last lecture, copies the contents of a file

- use argc, argv to get the source and destination file names
- unlike last lecture, use open( ), read( ), write( ), close( )
- read the man pages for open, read, write, close
- read CSAPP chapter 10

Exercise 2

Write a program that measures the sequential bandwidth of writing data to disk

- "man gettimeofday" to measure time
- note that just because write( ) returns, it doesn't mean data is on disk
  - man "fsync" to learn how to flush a file's contents to disk
- you can assume that sequential writes to a file result in sequential writes to disk (mostly true)

Bonus: measure the random seek write bandwidth

Exercise 3

Modify your linked list implementation from HW1 to:

- add a “WriteToFile( )” function
  - pass the name of the file to create / truncate and write to as an argument
  - pass a “convert payload to bytearray” function pointer
- writes each element of the linked list to the file
  - since elements are arbitrary byte sequences, you’ll need to record the length of an element before you write the element itself
- add a “LoadLLFromFile( )” function that takes a filename and returns a linked list
  - reads the output of WriteToFile( ), obviously!

See you on Wednesday!