CS131 Week 5

Week 5 Lecture 1

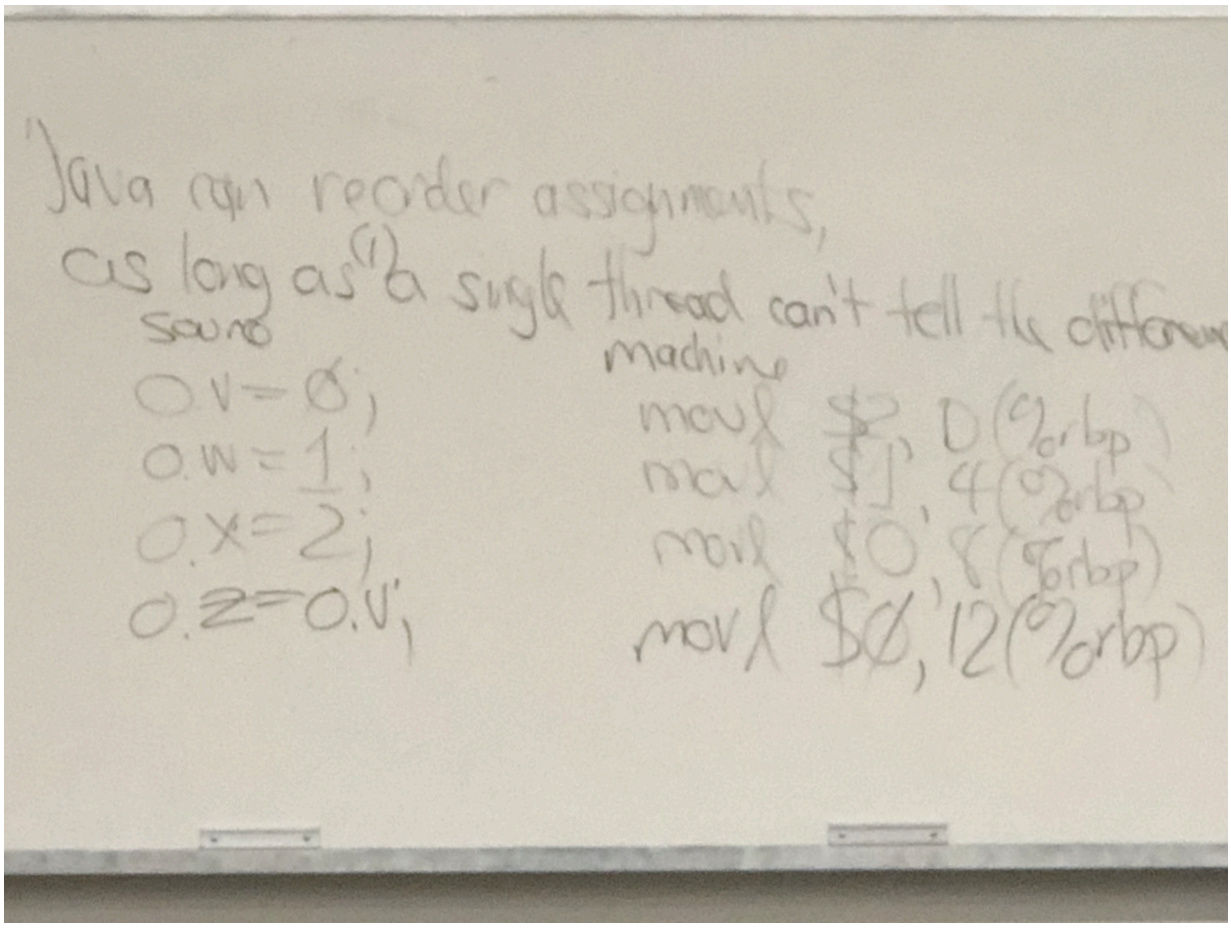
==========

Agenda
Language level synchronization
Names & Bindings
Java wants portability, so methods reflect that

---

1. o.v = 1
2. o.v = 0; o.w = 3;
3. print (o.v);

Java can reorder assignments as long as a single thread can't tell the difference e.g.



volatile

- accesses must be done in order
- specified in source
- NOT atomic

synchronized code

mutual exclusion

- achieved by any synchronized method running on an object
- LOCKS ENTIRE OBJECT
  - no other synchronized method in same object can run
- why lock entire object?
  - 2 methods may conflict with each other
  - e.g. method1 increments i, method2 reads i
- what if each function modifies different fields?
  - method1 modifies o.i, method2 modifies o.j
  - completely safe
  - don't use synchronized: if method1 running for 1 thread, method2 won't run for other thread
  - reentrant lock is better

basic locks

- grab lock or yield if you cant get it right away

spin lock

- grab lock or spin until preempted

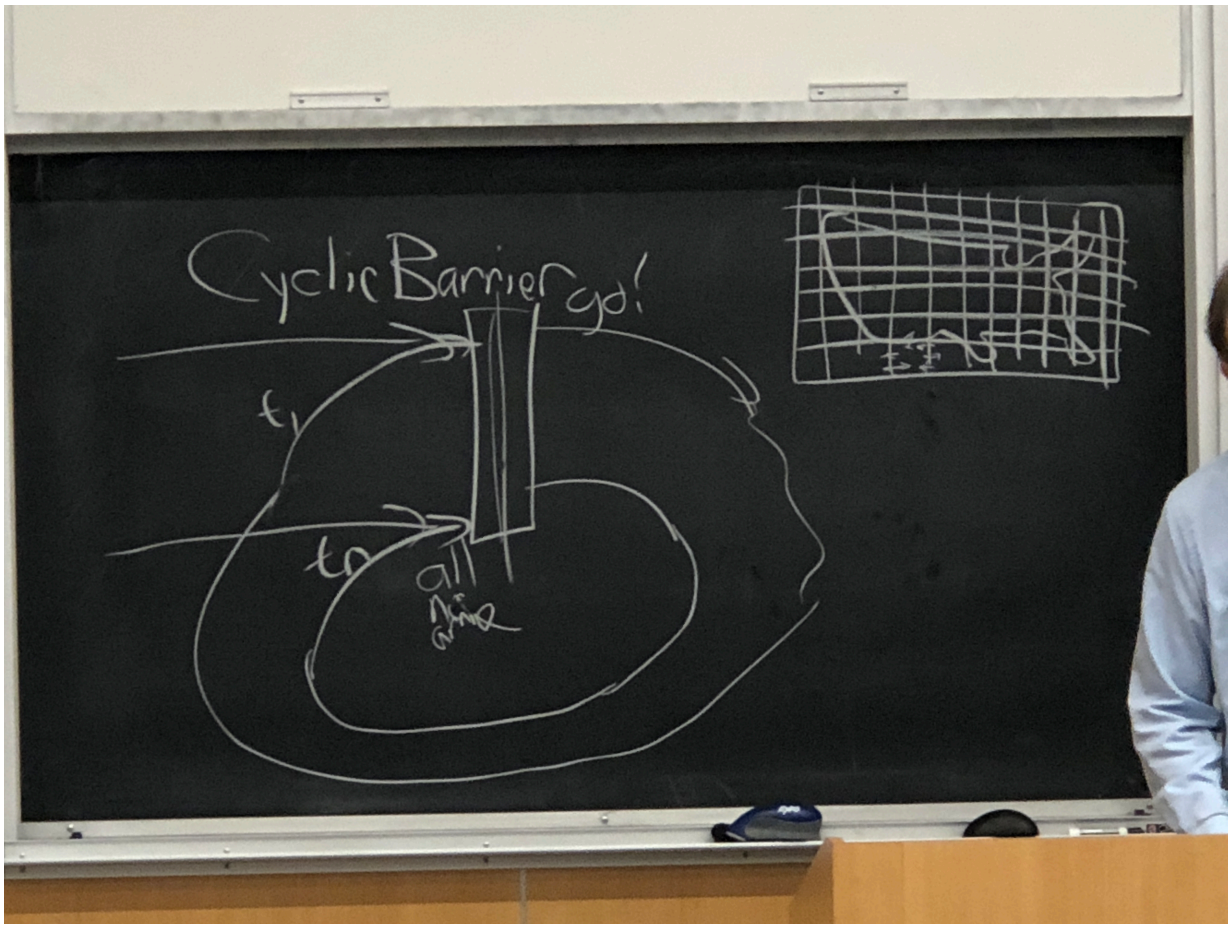
Object class methods

- o.wait() // timeout default infinity
  - removes all locks held by this thread
  - wait until o becomes available
- o.notify()
  - object is now available, wake up a waiter
- o.notifyAll()
  - object wakes up all threads when object is available

Exchanger

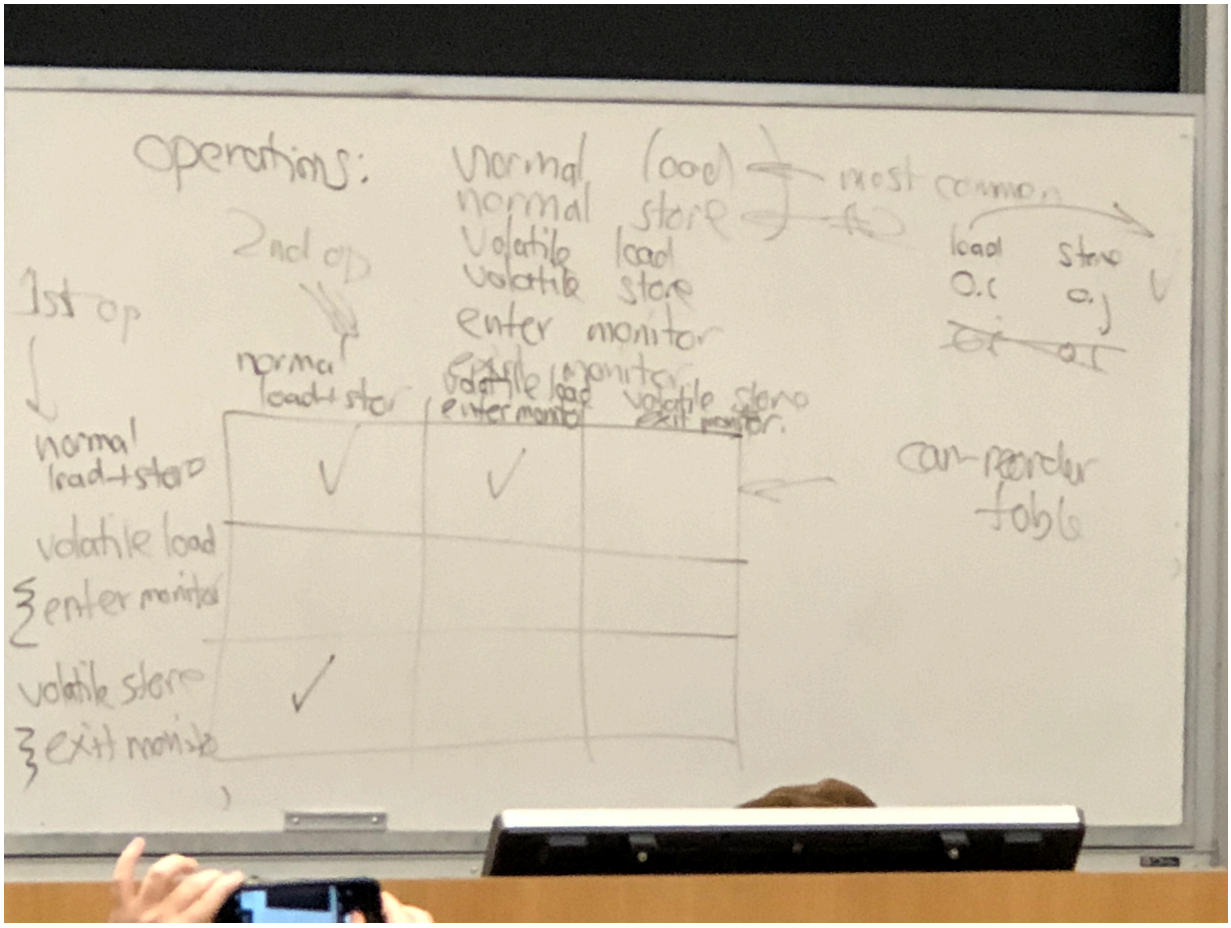
- Library classes built atop o.wait etc.
- rendezvous point for threads to "exchange" locks
  - Exchanger x;
  - t2 = x.exchange(t1) // t1 calls first, waits for another call to exchange to happen
  - t1 = x.exchange(t2) // now that t2 calls, t1 and t2 both execute

Cyclic Barrier
Basically the threads run all together and there's the middle checkpoint to make sure everything works and then reruns all threads at the same time again



Cyclic barrier is limited, so you can write your own libraries using the operations below

operations
- normal load
  - most common
- normal store
  - most common
- volatile load
- volatile store
- enter monitor (involve synchronized keyword)
- exit monitor



Agenda

Names and **Bindings**

- identifiers (int **identifier** = value)
  - case sensitive?
    - UCLA.edu vs.
    - ucla.edu
  - reserved words?
  - character set
    - int *i* = 0; // Bylorussian *i*, not normal i
    - int i = 10;
    - print(*i*); // doesnt print 10, prints 0 cuz prints Bylorussian *i*
- bindings
  - relation between identifier and what (name) it stands for

Examples

name = Sir Walter Scott

value = the author of Waverley

“The King didn’t know that **Sir Walter Scott** was the author of Waverley”

“The King didn’t know that **the author of Waverley** was the author of Waverley”

int n = 10;

name	**value**	**use**	**means**	**comment**
n	10	print(n)	print(10)	n bound to integer
n	address of n	&n	address of n	n bound to address
n	# of bytes in int representation	sizeof(n)	4	n bound to size of integer
n	byte representation of n	char buf [sizeof(n)]; memcpy(buf, &n, sizeof(n)); print(buf);	n is byte representation of n	n bound to byte representation of n

binding time

- when binding is created
- Possibilities:
  - runtime binding (at every assignment)
  - compile time, when program was statically analyzed
  - ABI definition time
  - link time (probably dynamic linker)
  - runtime, at block entry

Namespaces

partial function from names to values
primitive namespaces (builtin)
e.g. {**struct x** **{int x; }** **x;**
^ struct namespace ^ struct field namespace ^ local variable namespace
enum x {zero};
^ local variable namespace
struct y {int x; } y; print(x);
 # include <x>
 # define x 27
 # undef x
x: goto x;
}

They made different namespaces so that when people have chance collisions they will be saved

— 2/6/18