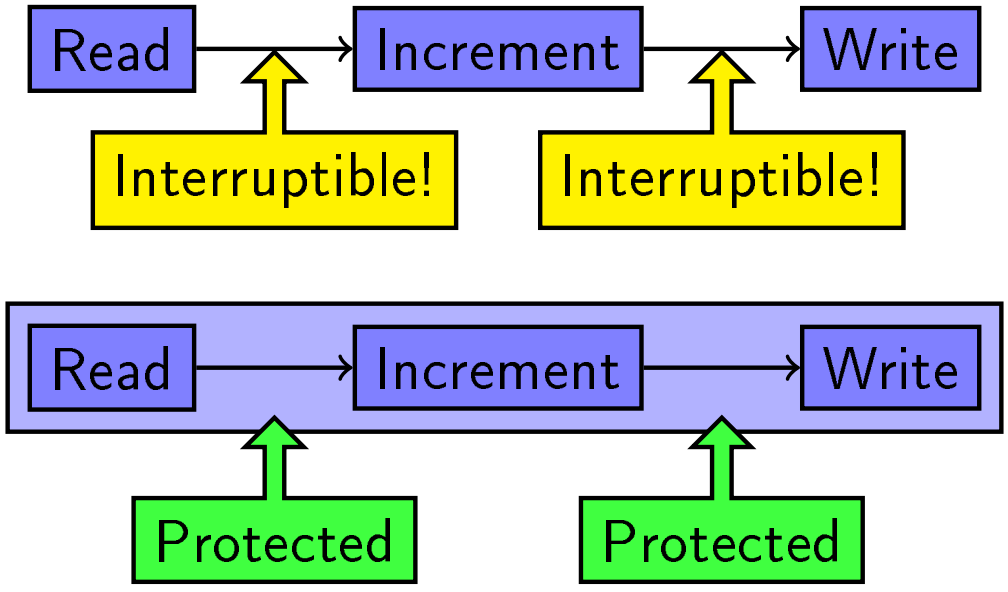
Lecture 7

- HW 2 out-ish
- Grading hw1
- First talks Fri

'Conventional' vs Atomic Memory Update



Outline

Outline

Atomic Operations: Compare-and-Swap

```
#include <stdatomic.h>
_Bool atomic_compare_exchange_strong(
        volatile A* obj,
        C* expected, C desired );
```

C11

What does `volatile` mean?

- any access is considered a side effect

What does this do?

atomically check that *obj == *expected
if so, overwrite w/ desired

How might you use this to implement atomic FP multiplication?

do
  read prev.
  update
  CAS
until successful

Memory Ordering

Why is Memory Ordering a Problem?

- OOO CPUs
- Compilers

} reorder ops.

What are the different memory orders and what do they mean?

- atomicity is unaffected
- "and then"

Types:

- seq. consistent: reorder nothing
- acquire: later loads may not reord. across
- release: earlier writes may not - - - -
- relaxed: reorder anything

HW: memory fences
Compiler: don't reorder

Example: A Semaphore With Atomics

```
#include <stdatomic.h> // mo_->memory_order, a_->atomic
typedef struct { atomic_int v;} naive_sem_t;
void sem_down(naive_sem_t *s)
{
  while (1) {
    while (a_load_explicit(&(s->v), mo_acquire) < 1)
        spinloop_body();
    int tmp=a_fetch_add_explicit(&(s->v), -1, mo_acq_rel
    if (tmp >= 1)
        break; // we got the lock
    else // undo our attempt
      a_fetch_add_explicit(&(s->v), 1, mo_relaxed);
  }
}
void sem_up(naive_s_t *s) {
  a_fetch_add_explicit(&(s->v), 1, mo_releas.);
}
```

[Cordes '16] — Hardware implementation: how?

C: What is 'order'?

C11 Committee Draft, December '10, Sec. 5.1.2.3, "Program execution":

- (3) *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is sequenced after A.) If A is not sequenced before or after B, then A and B are unsequenced. Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which. The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the *sequence points* is given in annex C.)

Q: Where is this definition used (in the standard document)?

used to order atomic operations

C: What is 'order'? (Encore)

C11 Draft, 5.1.2.4 "Multi-threaded executions and data races":

- All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M.
- An evaluation A *carries a dependency* to an evaluation B if ...
- An evaluation A is *dependency-ordered* before an evaluation B if...
- An evaluation A *inter-thread happens before* an evaluation B if...
- An evaluation A *happens before* an evaluation B if...

Why is this so subtle?

Keep juicy reordering opt
While establishing just enough off-core order

C: How Much Lying is OK?

C11 Committee Draft, December '10, Sec. 5.1.2.3, "Program execution":

- (1) The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.
- (2) Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. [. . .]

C: How Much Lying is OK?

- (4) In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).
- (6) The least requirements on a conforming implementation are:
  - Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
  - At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
  - The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

  This is the observable behavior of the program.

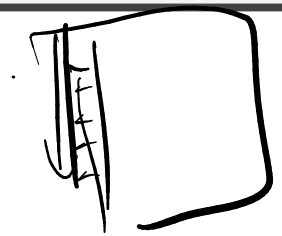
Arrays

Why are arrays the dominant data structure in high-performance code?

- Performance demands regularity (data layout/flow control)
- linear algebra

Any comments on C's arrays?

- 1D arrays
- nD arrays



a[i][j]

Arrays vs Abstraction

struct point {
float x,y,z };

YES

AOS

Arrays-of-Structures or Structures-of-Arrays? What's the difference? Give an example.

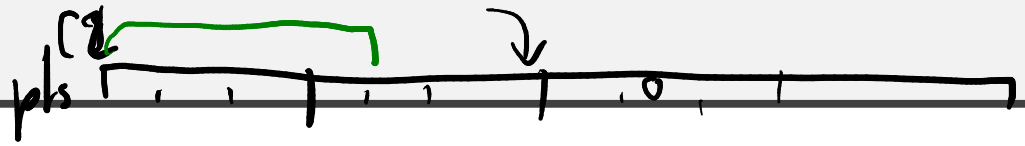
pts[i].x

xxxx

yyyy

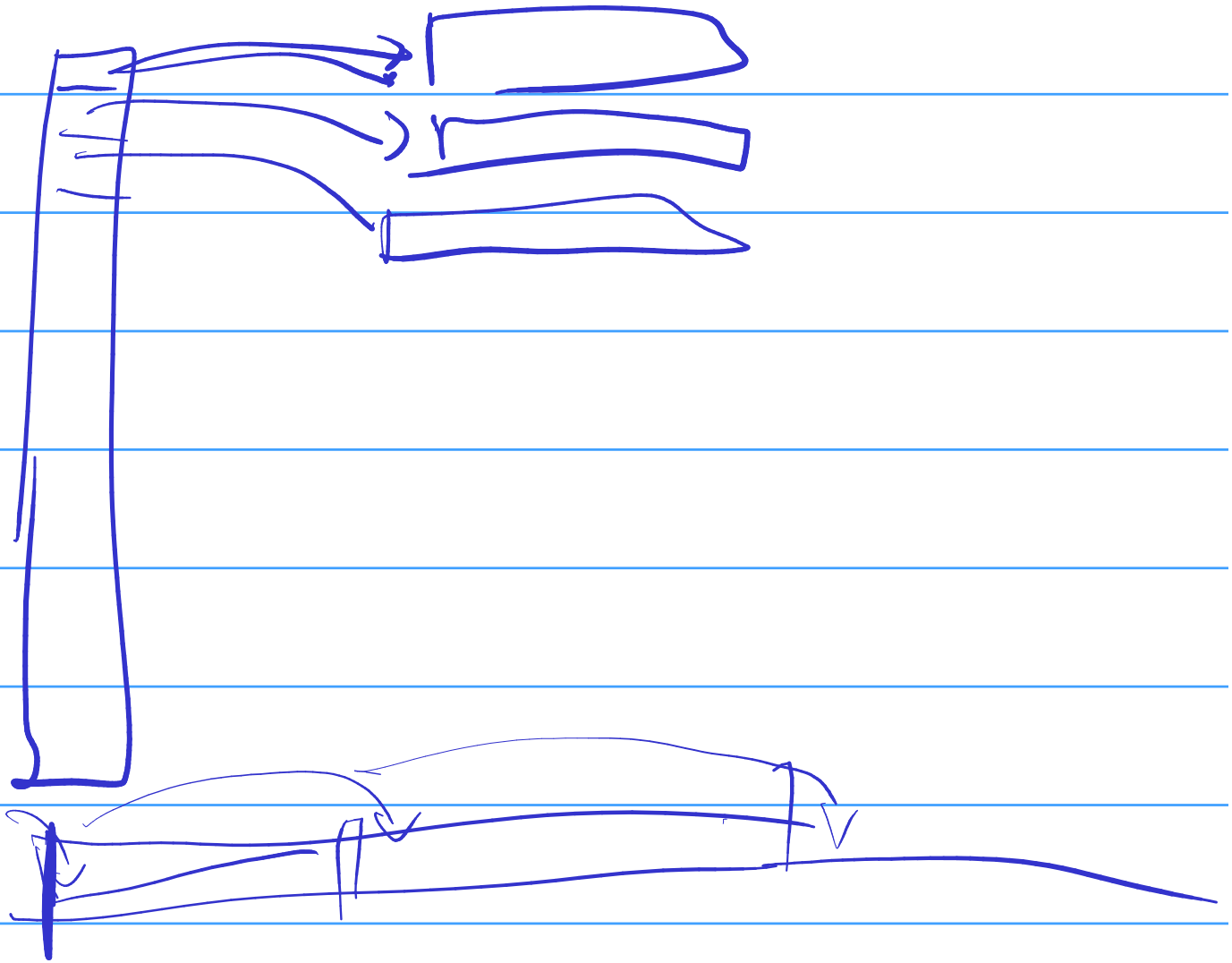
zzzz

SIMD



Language aspects of the distinction? Salient example?

Complex



SIMD

Name language mechanisms for SIMD:

- Inline assembly
- Intrinsics
- Vector types
- #pragmas in
- Merging of a scalar _mm_shuff prog. list.

Contrast *outer-loop* vs *inner-loop vectorization*.

Side q: Would you consider GPUs outer- or inner-loop-vectorizing?
[Franchetti/Püschel '08]