CSEP505: Programming Languages

Lecture 10: Atomicity; Memory Management

Dan Grossman

Spring 2006

Where are we

- Thread creation
- Communication via shared memory
  - Synchronization w/ join, locks
- Message passing a la Concurrent ML
  - Very elegant
  - Can wrap synchronization abstractions to make new ones
  - In my opinion, quite under-appreciated
- Back to shared memory for software transactions

Atomicity is modular

Atomicity restricts interleaving, regardless of other code from yesterday or tomorrow

```
let xfer src dst x =
 Mutex.lock src.lk;
 Mutex.lock dst.lk;
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x;
 Mutex.unlock src.lk;
 Mutex.unlock dst.lk
```

```
atomic:
  (unit->α)->α

let xfer src dst x =
atomic (fun()->
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x
)
```

Different viewpoints

Software transactions good for:

1. Software engineering (avoid races & deadlocks)
2. Performance (optimistic "no conflict" without locks)

key semantic decisions depend on emphasis

Research should be guiding:

A. New hardware with transactional support
B. Language implementation for expected platforms

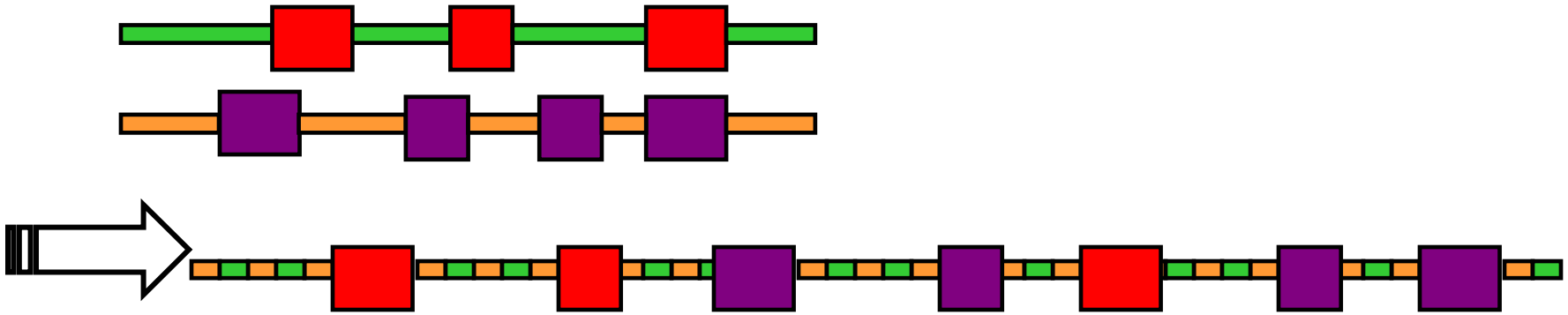
"is this a hw or sw question or both"

Full disclosure: I'm a "1B extremist"

Strong atomicity

(behave as if) no interleaved computation

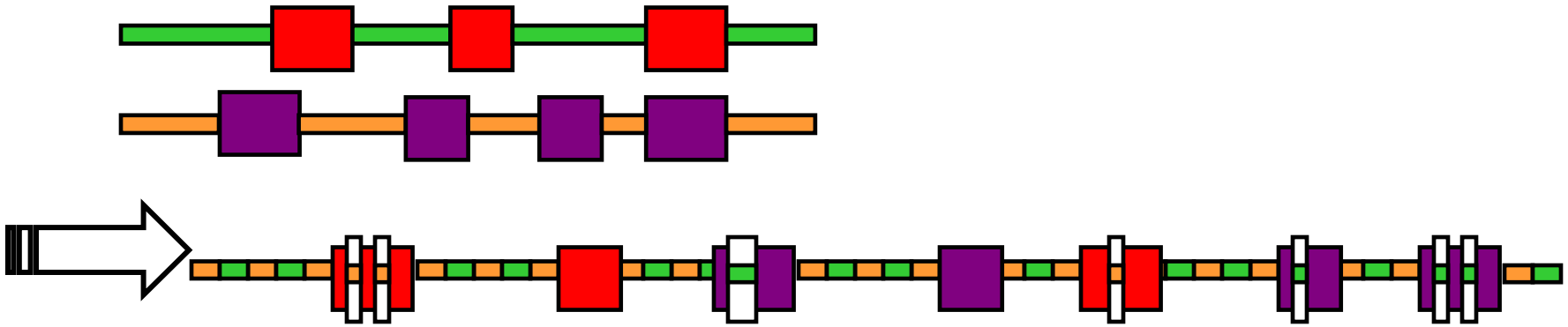
- Before a transaction "commits"
  - Other threads don't "read its writes"
  - It doesn't "read other threads' writes"
- This is just the semantics
  - Can "behave as though transaction started later"



Weak atomicity

(behave as if) no interleaved transactions

- Before a transaction "commits"
  - Other threads' transactions don't "read its writes"
  - It doesn't "read other threads' transactions' writes"
- This is just the semantics
  - Can "behave as though transaction started later"



Wanting strong

Software-engineering advantages of strong atomicity

1. Sequential reasoning in transaction
   - Strong: sound
   - Weak: only if all (mutable) data is not simultaneously accessed outside transaction
2. Transactional data-access a local code decision
   - Strong: new transaction "just works"
   - Weak: what data "is transactional" is global
3. Fairness: Long transactions don't starve others
   - Strong: true; no other code sees effects
   - Weak: maybe false for non-transactional code

Where are we

- Atomicity, especially strong sounds great
  - Another Dan mantra:

  "Atomic is to concurrency as garbage collection is to memory management"
- But there are some tough issues
  - Irreversible operations ("launch missiles")
- Now: a note on implementations
  - Like GC: It does the whole-program protocols!
  - Like GC: In practice, use low-level tricks

Implementing atomicity

Easier than you think (modulo performance):

Option 1:

- In a transaction, ensure exclusive owernship of everything you touch (avoid races)
- Rollback somebody on contention (avoid deadlock)
- Shameless plug: elegant and fast on a uniprocessor

Option 2:

- In a transaction, compute with private *version* of memory
- On commit, use a fancy protocol to ensure consistency and progress

The key point

There is data-level synchronization control, but now it's in the compiler and run-time

- Programmer declaratively restricts interleavings

Language-implementation solutions scale because implementation complexity doesn't depend on program size

But they are "one-size-fits-most"

Now what…

- Course evaluations
- Memory management
- Brief course wrap-up
- (Final next Tuesday 6:30-8:30)

You have grading to do…

I am going to distribute course evaluation forms so you may rate the quality of this course. Your participation is voluntary, and you may omit specific items if you wish. To ensure confidentiality, do not write your name on the forms. There is a possibility your handwriting on the yellow written comment sheet will be recognizable; however, I will not see the results of this evaluation until after the quarter is over and you have received your grades.
Please be sure to use a No. 2 PENCIL ONLY on the scannable form.

I have chosen _____ to distribute and collect the forms. When you are finished, he/she will collect the forms, put them into an envelope and mail them to the Office of Educational Assessment. If there are no questions, I will leave the room and not return until all the questionnaires have been finished and collected. Thank you for your participation.

From the beginning

Problem:

1. Why do we need memory management?
   - Same reason for any finite reusable resource
2. What does safety mean?
3. What is drag?

Solutions:

1. How does garbage collection (GC) work?
2. What other ways for safe memory management?
   a. Unique pointers
   b. (Automatic) reference-counting
   c. Regions

Why reuse?

- Values/objects/code take up space
- Using too much space slows down programs
  - Eventually they stop (memory exhaustion)
- Optimal space: reclaim immediately after last use
  - Earlier is incorrect (dangling-pointer dereference)
  - *Drag* is time between last use and reclamation
- But:
  - Last-use trivially undecidable
  - Batched reclamation can gain time for space

The view from C/C++

- *Stack* objects reclaimed at end of block/function
- *Heap* objects reclaimed with call to free/delete
- Dangling-pointers fine; dereferencing them unsafe
  - "Double-free" also unsafe
- Unreclaimed objects that become unreachable will:
  - Never be used
  - Never be reclaimed
  - So drag until termination ("space leak")

Reachability

Reachability soundly approximates “may be used again”

Inductive definition (transitive “points to”):

- Global variables reachable
- Unreclaimed stack objects reachable
  - *Liveness analysis* can do a bit better
- Objects pointed to by reachable objects are reachable

C: Avoid leaks by freeing before unreachable

Garbage-collected language: Make things unreachable

Reachability and leaks

- GC'd languages reclaim unreachable objects
  - So by some definitions "leaks are impossible"
- But "infinite drag times" are possible
  - Example: large unused data structure in a global
- Programming for space in GC'd languages
  - Usually ignore the issue
  - Set pointers to `null` when done with them
    - Error-prone!
  - Use *weak pointers* where appropriate
    - Provided as a language feature, dereference can fail

Where are we

Problem:

1. Why do we need memory management?
   - Same reason for any finite reusable resource
2. What does safety mean?
3. What is drag?

Solutions:

1. How does garbage collection (GC) work?
2. What other ways for safe memory management?
   a. Unique pointers
   b. (Automatic) reference-counting
   c. Regions

Reachability, cont'd

Algorithm sketch to find all reachable objects:

- Start at *roots* (globals and stack objects)
- Follow all pointers, but do not go around cycles

Problems:

- Find all pointers in pointed-to object
  - How big is the object?
  - What fields are integers?
- Avoid cycles (solution depends on GC technique)

Finding sizes

Garbage collector must know an object's size

- free/delete need to know too!

Solutions:

- A header word (e.g., before object) with the size
  - Class pointer can "serve double-duty"
- Size segregation and a global table of "page to size"

Bottom line:

- Allocator and/or compiler must collaborate with GC

Finding pointers

Does the GC know which fields/roots are pointers?

- Yes: accurate GC
- No: conservative GC

Theory: With conservative GC, “one unlucky int” could keep huge amount of data

Practice: Conservative GC tends to work

Accurate GC techniques:

- Class-pointer can “serve triple-duty”
- Low-order bit tricks (e.g., Caml ints are 31-bits)

Conservative GC for C

Yes, you can (conservatively) GC a C program

- The Boehm-Demers-Weiser conservative collector

2 of many interesting details:

- Use collector's malloc (so GC knows the size)
- Possible b/c C bans code most people think is legal:

```
void f() {
 int * p = malloc(100*sizeof(int));
 p += 1000; /* not allowed */
 p[-950] = 17;
 p += 100;  /* allowed */
 p[-50] = 17;
}
```

Compile-time flag to "add a byte or keep 2 objects"

Semispace copying collection

- Divide memory into 2 equal-size contiguous pieces
- Allocate objects into one-space until full
  - Easy and fast ("bump an allocation-pointer")
- Now have a full from-space & an empty to-space
  - Copy reachable objects into end of to-space
  - Set allocation-pointer just past them in to-space
  - Restart the program (semispaces reversed roles)

Wait a minute

Skimmed over key details

- We moved objects; must update all pointers to them
- Must avoid cycles
- The GC can run without much extra space (good)

How:

- "*Cheney queue*" just two pointers in to-space
  - Objects to scan (update pointers and maybe add pointed-to objects to queue)
- Cycle avoidance: *forwarding-pointers* in from-space
  - Easy to tell what space is pointed-to

Mark-sweep collection

- Allocate objects until you have almost no room left
- Mark all reachable objects (bit in header word)
  - Avoid cycle by checking bit
- Sweep through memory
  - If object unmarked, reclaim it
  - If object marked, unmark it

No 2x space and no moving objects, but…

Wait another minute

- In practice, if more than 2/3 of objects or so are reachable, you spend lots of time in GC

- Allocation is complicated
  - Must find enough space for the new object
  - Fragmentation can hurt performance
    - Or exhaust memory before copying GC does

- No “Cheney” queue, so GC needs an explicit stack or low-level cleverness to run in little space

Generational

Copying and mark-sweep from about 1960

Generational GC a key mid-80s optimization because

- Most objects die young
- Most old objects never get mutated to point to young

How:

- Allocate in a nursery
- Empty nursery has no pointers into it!
- Fill nursery like in copying collection
- Also track mutations to record pointers into nursery
  - Yet another reason to avoid mutation (slower)
- To collect nursery, ignore rest of heap except recorded pointers

Some more terms

Just sketched the basics of copying and mark-sweep

And generational (non-nursery technique orthogonal)

Some other terms worth knowing:

- Incremental GC (do a little bit on each allocation)
  - Avoid large pause times
- Concurrent GC (collector thread in parallel with the program)
- Parallel GC (multiple collector threads)
- Lots of other important tricks: (lazy-sweeping, large-object spaces, …)

GC Summary

Great survey paper:

Paul R. Wilson. Uniprocessor Garbage Collection Techniques. International Workshop on Memory Management 1992

Available at:
http://www.cs.utexas.edu/users/oops/papers.html

- Programmer must know about reachability, that objects may move, that mutation may cost, etc.
- GC implementor must try to do well without knowing the application's memory behavior
  - But done by memory-system experts!

Where are we

Problem:

1. Why do we need memory management?
   - Same reason for any finite reusable resource
2. What does safety mean?
3. What is drag?

Solutions:

1. How does garbage collection (GC) work?
2. What other ways for safe memory management?
   a. Unique pointers
   b. (Automatic) reference-counting
   c. Regions

Now forget GC

Idioms that avoid dangling-pointer dereferences

- And languages and/or types to enforce them!
- A language can have more than one
- More work than GC, but safer than unchecked malloc/free

Worth knowing just for the idioms

Unique pointers

- If p is the only pointer to o, then free(p) can't lead to dangling-pointer dereferences provided *p is not used afterwards

- Unique-pointers allow only trees (no dags or cycles)

- Maintaining uniqueness invariant
  - Dynamic: destructive-reads
    - (p=q and free(q) set q to null)
  - Static: linear type systems and/or flow analysis

Reference-counting

(Dynamic) ref-counting basics:

- Store number of pointers to object with object
- If count goes to zero, free it

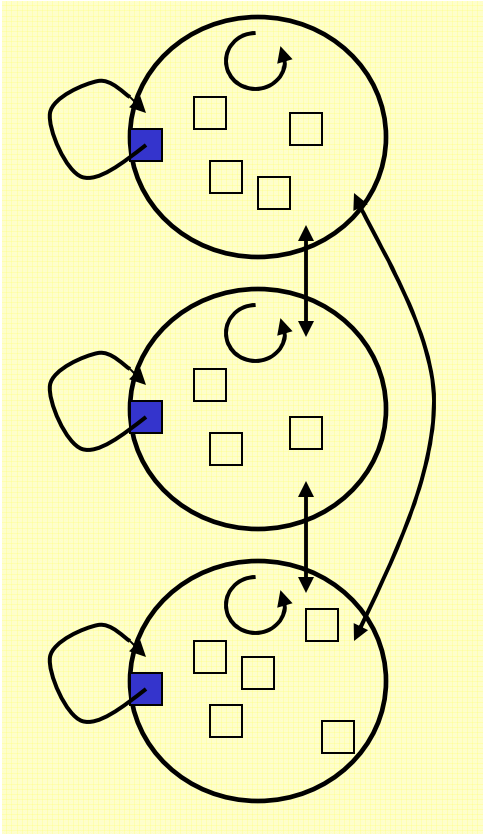
Can automate this easily enough

But:

- Cycles never get reclaimed unless programmer breaks the cycle
- Expensive without tricks (“deferred ref-counting”)

Regions

- A decades-old idiom also known as zones, arenas, …
- Partition memory into region; every object in one region
- API basics
  - new_region returns a handle
  - new_object takes a handle
  - free_region takes a handle
- No free_object



What did we do

- Accomplished nothing if we put every object in a different region
- But now intra-region pointers “can’t go wrong”
  - Programmer puts objects with similar lifetimes in same region
  - To avoid leaks, just don’t lose the handle
- For inter-region pointers, options:
  - Dynamic ref-count (see RC or RTSJ)
  - Type-system to restrict “what points where” and when pointers can be dereferenced (see Cyclone)

A common idiom

- Far too painful in C: caller knows lifetime of result, callee knows size and structure of result
  - Leads to evil stack-allocated buffers
- Region solution: a region-handle argument
  - Easy even if result is some complicated graph

```
result_t g(handle_t, …);
void f() {
 handle_t h = new_region();
 result_t r = g(h,…);
 /* compute with r */
 free_region(h);
}
```

Course summary

- Defining languages is hard but worth it
  - Interpretation vs. translation
  - Inference rules vs. a PL for the metalanguage
- Essential features we investigated
  - Mutable variables (and loops)
  - Higher-order functions
  - Objects
  - Threads (and locks and channels)
- Types restrict programs (that's a good thing!)
  - But want polymorphism for reuse

Penultimate slide

- We avoided:
  - Subjective non-science (“I like curly braces”)
  - Real-world issues (“well-known libraries for X”)
- Focused on:
  - Concepts that almost every language has, including the next fad that doesn’t exist yet
  - Connections (objects and closures are different, but not totally different)
  - Reference implementations, not fast or industrial-strength ones

Questions?

Questions?