The Scalable Commutativity Rule

```
  * First and foremost, MIDTERM on Monday.
    - covering all material that we've covered
    - actually reading the papers is EXTREMELY important
      - we obviously don't cover everything in lecture (see last lecture)
    - look at previous years for examples
    - can bring/use any printed material
      - papers, lecture notes, lab code, wikipedia, etc.
      - won't have enough time to read paper on the spot
    - excellent performance on final can make up for poor midterm
      - not totally, of course
  * Lab grades will be posted tonight.

Last lecture ending a bit abruptly.
  KEY POINTS:
    1) minimize sharing => achieve scalability
      - focus is on false sharing, where allocate can induce but shouldn't
    2) reuse unused resources => bound blow-up
      - strategy based on empty fraction
```

```
We continue to discuss scalability.

  - What we want (scalability):

        |           /
     w  |         /
     o  |       /
     r  |     /
     k  |   /
        | /
      ----------------
        worker count

  - What we actually (usually) get:

     w  |
     o  |
     r  |     _____
     k  |   /       \
        | /           \
      ----------------
        worker count

  Q) Why? (assuming there's more than enough work to do)
    A) Workers are not working independently; time is spent communicating
       instead of performing useful work.
```

```
The Scalable Commutativity Rule
  MIT CSAIL + Harvard (Eddie); SOSP 2013
  Best Paper Award SOSP 2013
  Best Thesis Award OSDI 2014

  Q) What does the paper have to say about scalability?
    A) Paper provides a _rule_ that we can use to know when there exists a
       scalable implementation.

  informally:
```

interfaces operations commute => they can be implemented in a scalable way

formally:

```
      H = X || Y      H â\210\210 S       Y SIM-commutes in H
  ------------------------------------------------------
      â\210\203 implementation of S where Y is conflict-free
```

(don’t actually need H â\210\210 S there since SIM-commuting requires that as well)

* we have histories (H with concatenation of histories X and Y)
* we have specifications (S)
* we have "SIM-commutativity"
* we have conflict freedom

Q) What does it mean for operations to commute?
  A) It means that there’s no way to distinguish the order in which they
     executed, given some interface.

  A) Regardless of the order that those operations are performed in, the output
     will always be "the same" (as specified by some specification).

     Example: creat("/a/b") creat("/a/c")
     Example: open(x) => rand_fd open(y) => rand_fd
     Not Example: close(x), open(x)

Q) Why is this interesting? What might we do different because of this rule?
  A) We’d likely take a completely (!!!) different approach to designing a
     scalable system. Prior to this rule, designing for scalability was
     iterative: plot, fix, plot, fix. Now, design for scalability _before
     writing a single line of code!_ Very, VERY different.

Q) Doesn’t "what scales" depend on the hardware/architecture?
  A) Yes. Recall that we get scalability when workers are performing work
     independently. Dependence in modern system occurs through cache coherence
     (MESI). So as long as we don’t have cache coherence occuring, we’re okay.
     If that happens, workers can perform reads/writes independently:
     conflict-free.

Quick Review of MESI and Cache Coherence

Let’s see why reading/writing to unshared cache lines leads to conflict freedom
and scalability in MESI-like architectures.

A simplified view of memory hierarchy looks as follows:

```
         -----------------------------------------
         |              main memory              |
         -----------------------------------------
                             |
                             |
         -----------------------------------------
         |                                       |
     P1 CACHE                                P2 CACHE
         |                                       |
    -----------                             -----------
    |         |                             |         | <- cache line = 64 bytes
    -----------                             -----------
    |         |                             |         |
    -----------                             -----------
    |         |                             |         |
    -----------                             -----------
    |         |                           |         |
    -----------                           -----------
    |         |                           |         |
    -----------                           -----------
    |         |                           |         |
    -----------                           -----------
```

  * All processors can communicate with eachother as well as main memory.
  * Each processor has its local caches.
  * Cache is seperated into cache lines.
  * Data is read from main memory in cache-line >= chunks into local cache.
  * Data is read and written directly from cache.

MESI Cache Coherence
==

```
  * Goal of cache coherence is to keep caches in-sync (coherent).
    - Want a coherent view of memory in an SMP system.
  * MESI is a message-based (really, snooping) protocol for cache coherence.
  * Each cache line is annotated with one of 4 states: M, E, S, or I.
    (I) Invalid (can be reused)
        The cache line is present only in the current cache, and is dirty.
    (S) Shared (read only)
        Indicates that this cache line may be stored in other caches of the
        machine and is clean.
    (E) Exclusive (can read/write)
        The cache line is present only in the current cache, and is clean.
    (M) Modified (can read/write)
        The cache line is present only in the current cache, and is dirty.
  * Caches start in the I state.
```

```
         -----------------------------------------
         |              main memory              |
         -----------------------------------------
                              |
                              |
         -----------------------------------------
         |                                       |
     P1 CACHE                                P2 CACHE
         |                                       |
    -----------                           -----------
    | I       |                           | I       | <- cache line = 64 bytes
    -----------                           -----------
    | I       |                           | I       |
    -----------                           -----------
    | I       |                           | I       |
    -----------                           -----------
    | I       |                           | I       |
    -----------                           -----------
    | I       |                           | I       |
    -----------                           -----------
    | I       |                           | I       |
    -----------                           -----------
```

```
  Let's walk through one scenario:

  * P1 reads 0x00 (CL0) -> change CL to E in P1.
  * P2 reads 0x10 (CL0) -> change CL to S in P1.
    * P1 notices, changes CL to S, probably forward line
  * P1 wants to write to 0x04 (CL0):
    - P1 communicates that it wants to upgrade CL to E
```

- P1 waits for everyone to report they've invalidated
- P2 marks CL as I
- P2 reports invalidation
- P2 writes to CL, marking it as M

Q) What would happen if P2 wants to write to 0x14?
A: We need to go through another round of coherence.

KEY POINT:

A processor cannot perform work independently if it sharing a cache line with another processor. Scalability is not possible without performaing work independently!

Let's walk through a different scenario:

* P1 reads 0x00 (CL0) -> change CL0 to E in P1.
* P2 reads 0x80 (CL2) -> change CL2 to E in P2.
* P1 wants to write to 0x04 (CL0):
  - Q) What happens?
    - A) P1 just writes to CL0 -> changes CL0 to M.
* P2 wants to write to 0x80 (CL1):
  - Q) What happens?
    - A) P2 just writes to CL1 -> changes CL1 to M.

KEY POINTS:

A processor CAN perform work independently if it reading and writing for exclusive (non-shared) cache lines. Scalability IS possible.

Adding more cores to a system running software that has conflict-free memory accesses should result in linear scaling since work can be performed independently on those new cores.

SO FAR, we've built up:

conflict-free implementation => implementation scales

But the rule is actually:

interfaces operations commute => they can be implemented in a scalable way

KEY INSIGHT:

interfaces operations commute => conflict-free implementation exists

We'll spend most of the rest of lecture building this up.

To build this up, we need a more rigirous understand of commutativity and conflict freedom. Let's start with the former.

ACTIONS

An action is either an invocation or a response.
- in an OS, an action is: a system call with arguments
- in an OS, a response is: the return value of the syscall

An action is comprised of exactly the following:
1) an operation class (e.g., which syscall)

2) arguments for actions, return value for responses
3) the relevant thread
4) a tag for uniqueness

Actions come in pairs: invoked/responded by same thread.

We’ll represent the invocation as a lower case letter and the response to that invocation as the same lower case letter with a ’.

```
a a’ is the pair for action a
```

We’ll offset actions vertically when they occur in different threads.

```
t1: a     a’
t2:   b b’
```

This means that a was invoked in t1 but before it could response, b was invoked and responsed t2. Finally a responses in t1.

HISTORIES

A history is defined as a sequence of actions ’H’. Common letters used to represent a history are ’H’, ’X’, ’Y’, ’Z’.

```
H = X || Y (|| means concatenation: history X then history Y)
```

So for actions a and b, we might have:

```
H = t1: a   a’
    t2:   b    b’
```

Only consider well-formed histories: one outstandard invocation at a time.

PER-THREAD SUBHISTORIES

Histories can be filtered for a single thread ’t’:

```
H | t
```

This gives you only the histories for that thread.

```
H = t1: a    a’
    t2:   b    b’

H | t1 == a b
```

REORDERINGS

A history can be "reordered". A history H’ is a "reordering" of history H when

```
FOR ALL t, H | t = H’ | t
```

In other words, same actions in both histories in same order on each thread.
    but threads can interleave differently

SPECIFICATION

A "specification" declares which histories are "correct".
- a prefix-closed set of well-formed histories
  a subset S â\212\206 P is prefix closed if for all sâ\210\210P tâ\210\210S, sâ\211¤t => sâ\210\210S

A1 = getpid(), A1’ = 0 is not in the Unix spec since no process has id 0

SI COMMUTATIVITY

Given H = X || Y, Y SI-commutes in H when:

given any reordering Y’ of Y, and any sequence Z

```
(X || Y || Z) in spec IFF (X || Y’ || Z) in spec
```

- presence of any Z requires reordering are indistinguishable by future
- in particular, these means the state created by Y (or any Y’) has to be the same (as far as the specification goes) since Z could look at any component of it (commutativity)
- ensuring all reorderings of Y when concatenated with X meet spec ensures return value is valid regardless of order of action (interface-based)
- doesn’t depend on implementation, just specification! (interface-based)
- X sets up the state (state-dependent commutativity)

- Note that state-dependence means Y can commute with some Xs but not others
  - good because we don’t get an all or nothing
  - can expand situations that commute
  - we say that these "conditionally commute"
  - example: open("a", O_CREAT | O_EXCL)

SIM-COMMUTATIVITY

SI-commutativity doesn’t suffice!

- Y is a concatenation of things, Y = Y1 || Y1
- possible for Y1 not to SI-commute in H by itself, even though Y does
- example:
  ```
  Y1 = [(A1 = set(1), A1’), (B2 = set(2), B2’)]
                                          Y2 = [(C1 = set(2))]
  ```
  - all reordering of Y leave value at 2, so Y SI-commutes in any H
  - but Y1 does not SI-commute: could be 1 or 2 depending on order
  - we say Y is does not monotonically commute, increasingly commute
- need to ensure that every prefix P of Y SI-commutes in H = X || P
  - if they do, then Y SIM-commutes in H = X || Y
- so, the full SIM-commutativity statement is:
  Y SIM-commutes in a history H = X || Y when
    for any prefix P of any* reordering of Y, P SI-commutes in X || P

IMPLEMENTATION

An implementation is a step function in S x I -> S x R

```
m(state, invocation) -> (state, response)
S = set of implementation states
I = set of valid invocations
R = set of valid responses
```

Each state is some tuple s IN S = (s.0, s.1, ..., s.m)
- s(i<-x) is component replacement. IE, s.i = x, like a write

Consider a step m(s, a) producing (s’, r)
- a step ‘writes‘ state component i when s.i != s’.i
- a step ’reads’ a state component when for some y:
  m(s(i<-y), a) != (s’(i<-u), r)
  - that is, s.i being different changes the response, so it observes it

Access conflict defined as:
- two steps on different threads: one writes or reads, the other writes

So a set of steps is conflict-free when...
- no pair of steps in the set has an access conflict

MESI-based cache-coherent machines are usually scalable for conflict-free

```
PROOF

  Proof is by construction. Show that because Y SIM-commutes in H = X || Y, we
  can construct an implementation that uses per-thread histories in the region
  of 'Y'. It then handles requests for responses from different cores entirely
  independently. This is only possible because the region is commutative; so
  even if it gives responses in a different order than the "initial history",
  it'll be correct since the order is irrelevant.

--------------------------------------------------------------------------------

  1) ANALYZER takes symbolic model of interface
    - computes under which conditions the interface's operations commute

  2) TESTGEN converts ANALYZER's commutativity conditions into concrete tests
    - assigns concrete values to the symbolic state + arguments

  3) MTRACE checks whether a particular implementation is conflict-free
    - runs the test cases generated by TESTGEN
    - runs the entire OS in QEMU
    - at start of each test, MTRACE hypercalls to enable memory access recording
    - MTRACE logs all reads and writes by each core along with
      - info about currently executing thread (to filter irrelevant background)
      - also a stack trace
    - after execution, MTRACE analyzes log and reports conflicts
      - using DWARF debug to resolve data type and name, etc.
```