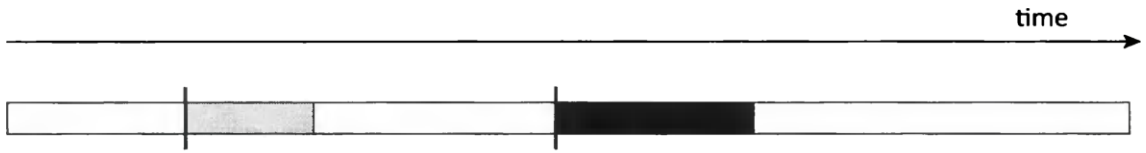
Concurrent GC

Schedule

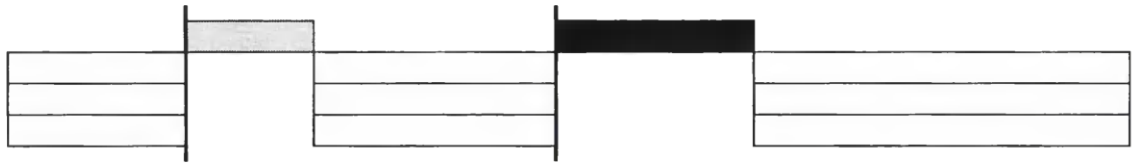
	M	W
Sept 14	Intro/Background	Basics/ideas
Sept 21	Allocation/layout	GGGGC
Sept 28	Mark/Sweep	Copying GC
Octo 5	Details	Ref C
Octo 12	Thanksgiving	Mark/Compact
Octo 19	Partitioning/Gen	Generational
Octo 26	Other part	Runtime
Nove 2	Final/weak	Conservative
Nove 9	Ownership	Adv topics
Nove 16	Adv topics	Adv topics
Nove 23	Presentations	Presentations
Nove 30	Presentations	Presentations

Final stuff

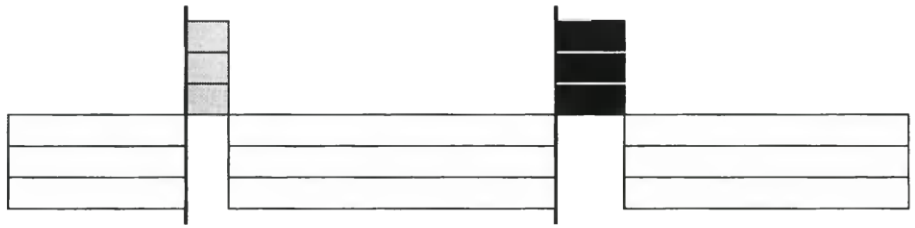
- Presentations start next Monday
- If you don’t schedule with me by Friday, 0% on the final presentation
- Final project due the last week



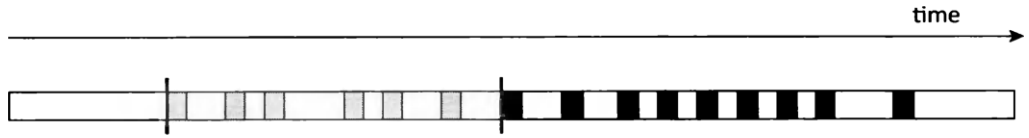
(a) Stop-the-world collection, single thread



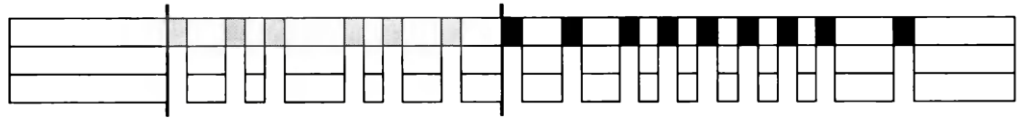
(b) Stop-the-world collection on multiprocessor, single collector thread



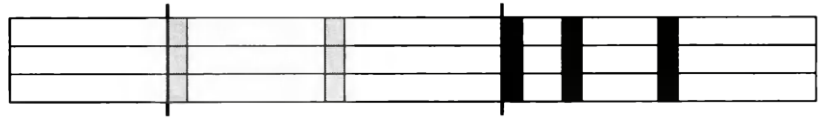
(c) Stop-the-world parallel collection



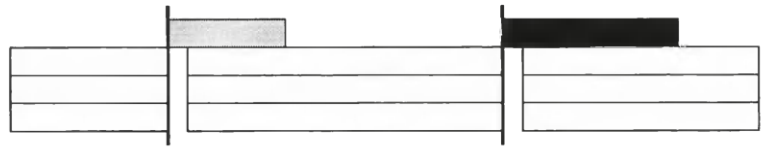
(a) Incremental uniprocessor collection



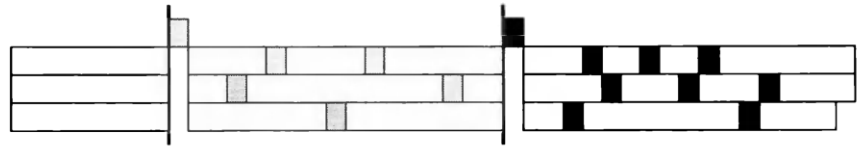
(b) Incremental multiprocessor collection



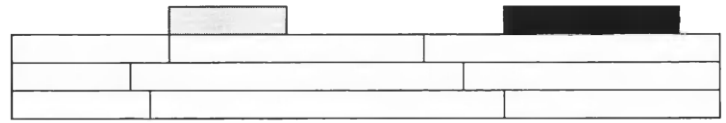
(c) Parallel incremental collection



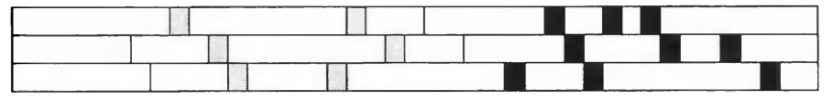
(d) Mostly-concurrent collection



(e) Mostly-concurrent incremental collection



(f) On-the-fly collection



(g) On-the-fly incremental collection

Requirements

- *Safety*: At least all reachable objects are retained
- *Liveness*: Garbage collection eventually terminates
- *Precision*: Unreachable objects are collected

Goals

- Parallel GC can improve throughput
- Concurrent GC can improve latency
- Improving latency this way invariably hampers throughput
- Must choose which matters when

Phases

- Moving is tricky, but not impossible (we focus on mark-and-sweep)
- Mark phase is when mutator and collector collide
- If mark is correct, mutator cannot touch any unmarked objects, so sweep is easy

Atomicity

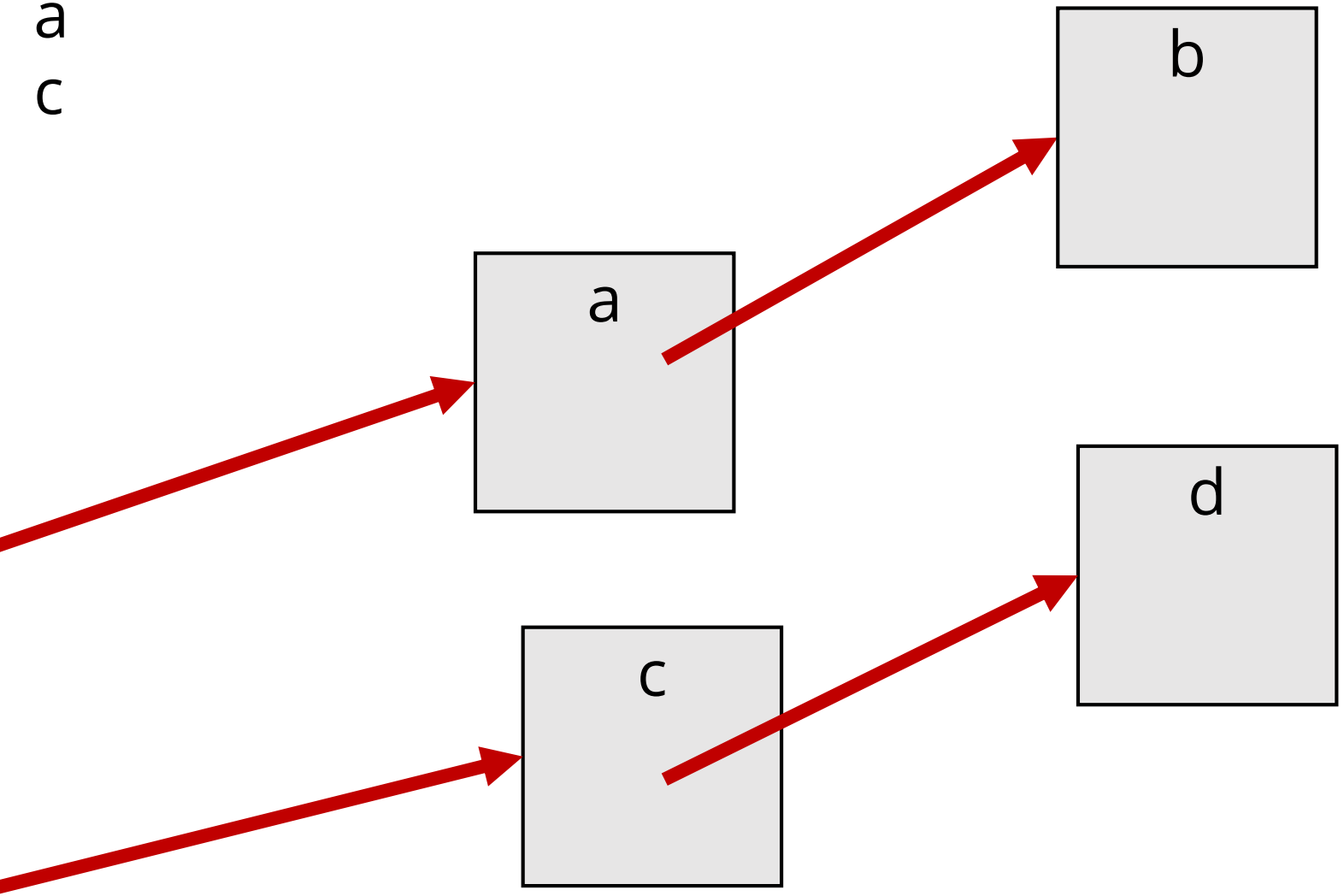
- Without concurrent GC, collection is *atomic* to mutators
- Now, mutator and collector must communicate
- Need to worry about individual actions' atomicity: Read/write fields, read/write roots, scan objects, etc.

Writing

- When we write a reference, we inherently *delete* a reference
- This means we can move references around
- None of this changed without concurrency, and it wreaks Hell on reachability

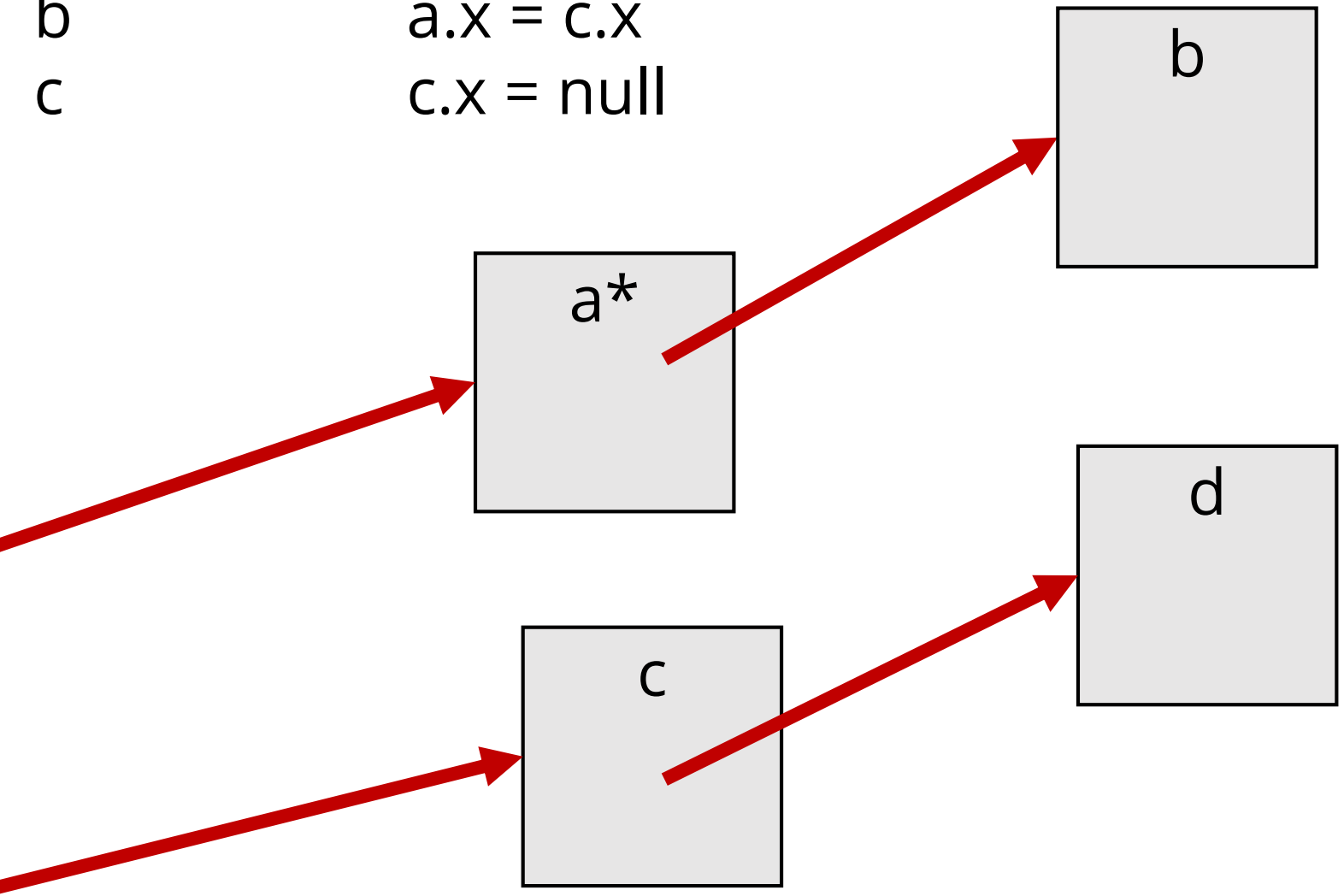
Worklist:
a
c

Mutator:



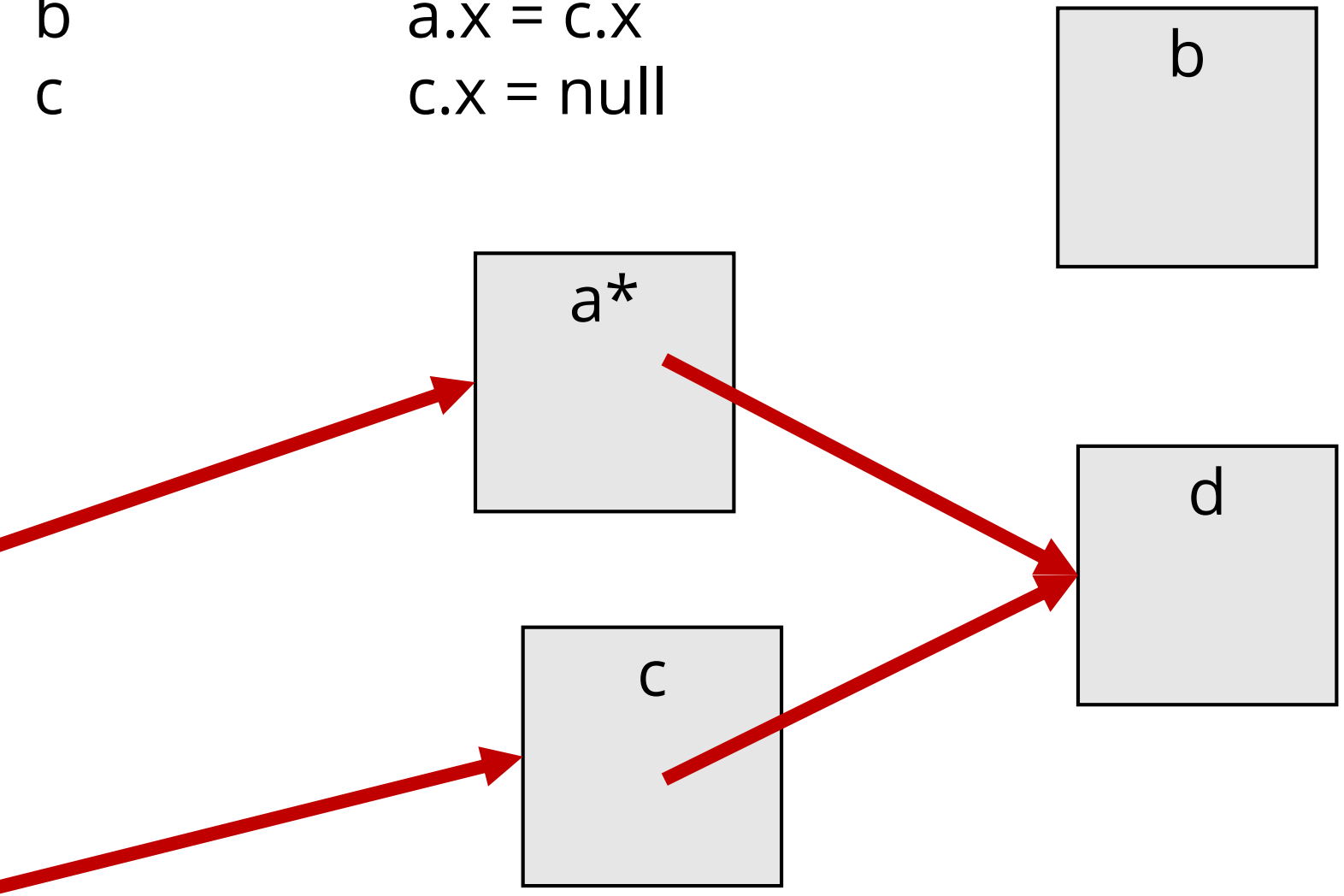
Worklist:
b
c

Mutator:
a.x = c.x
c.x = null



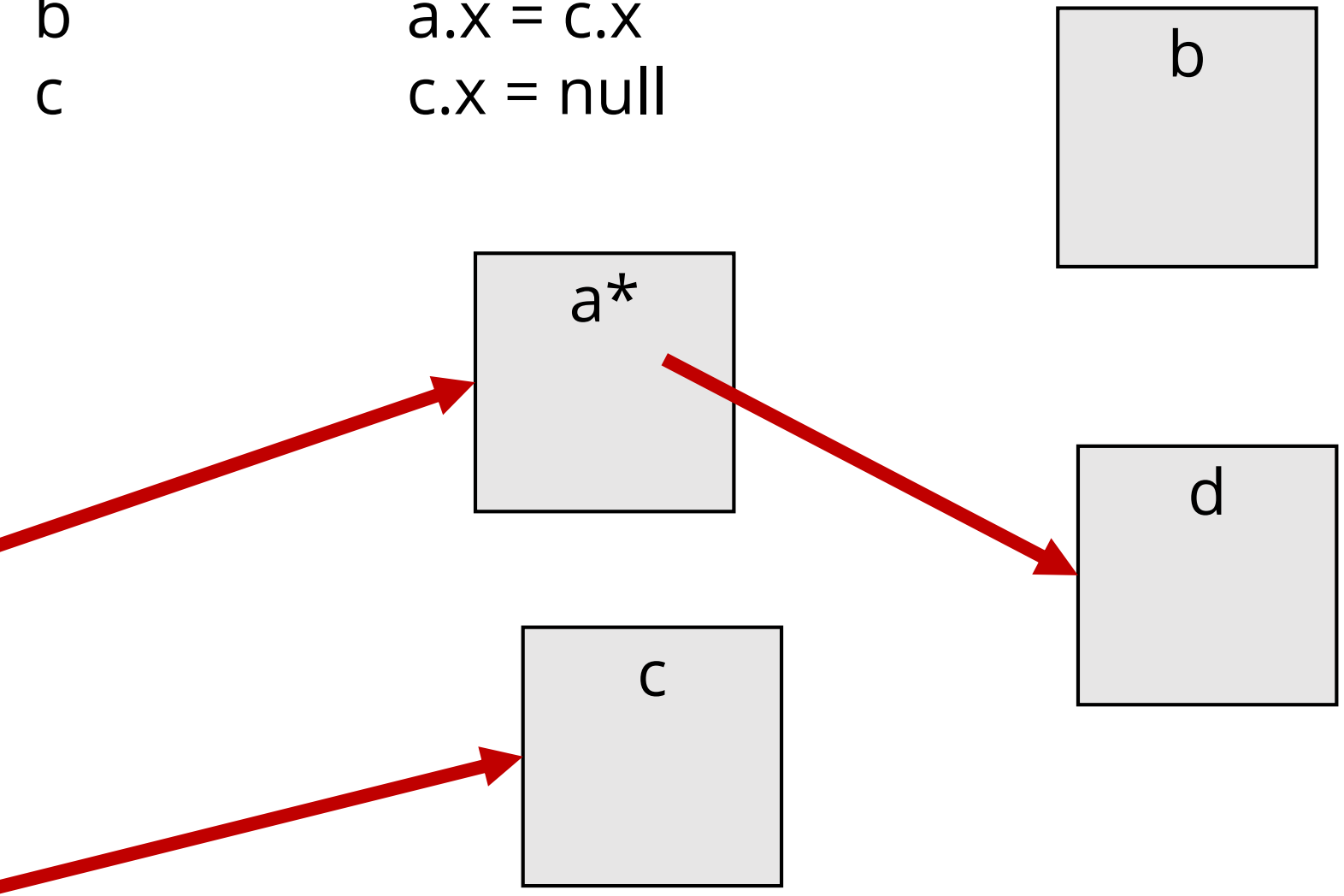
Worklist:
b
c

Mutator:
a.x = c.x
c.x = null

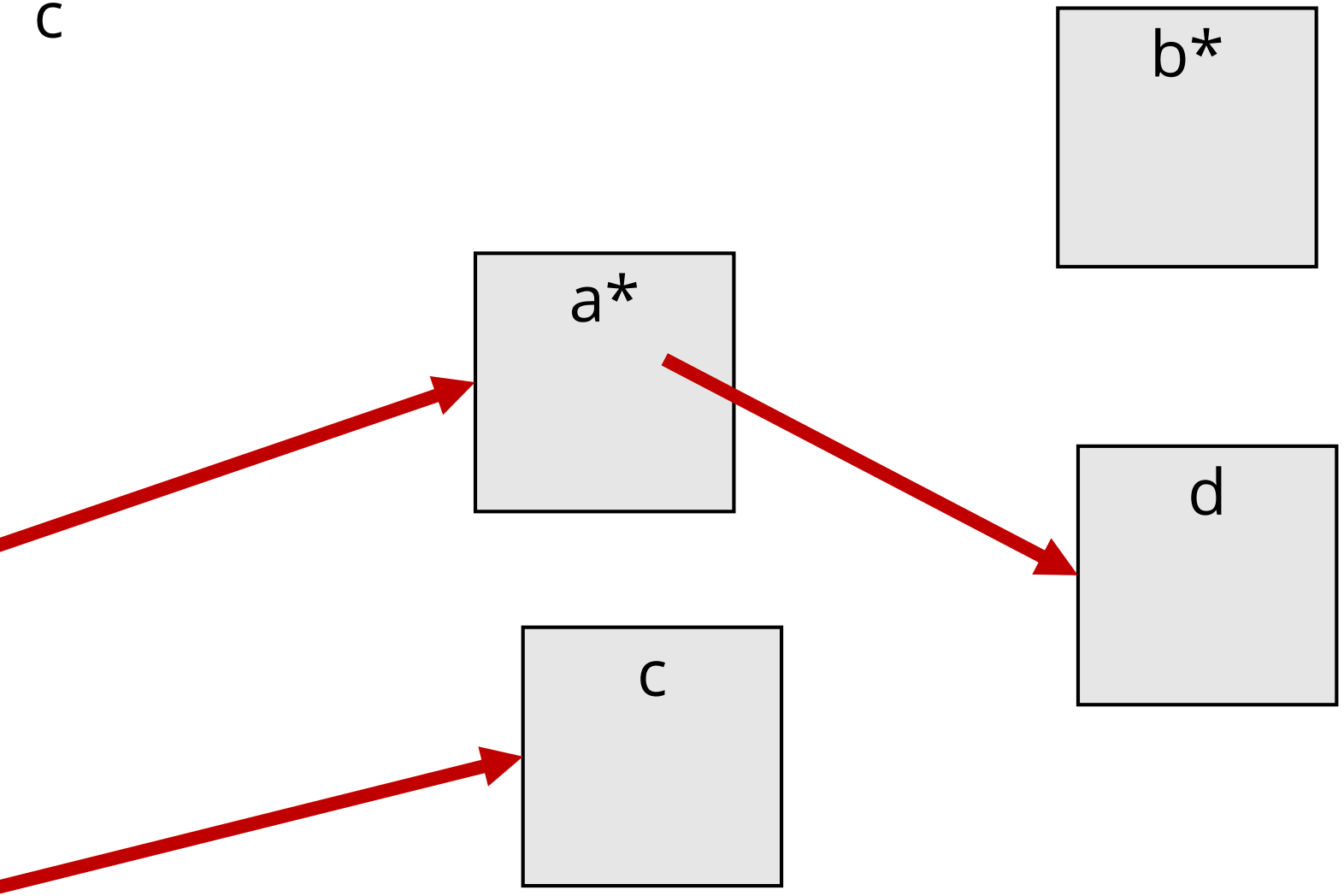


Worklist:
b
c

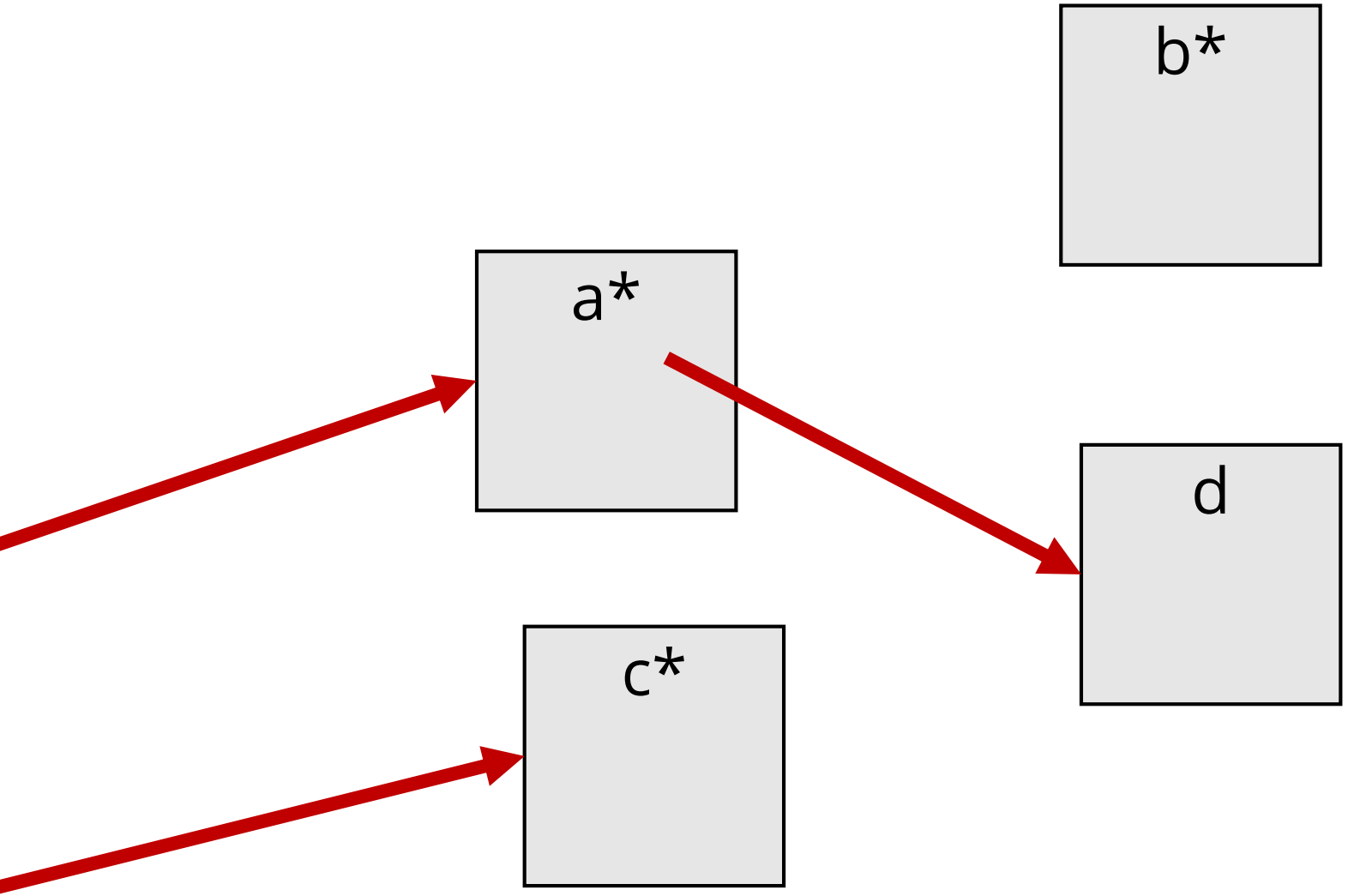
Mutator:
a.x = c.x
c.x = null



Worklist: Mutator:
c



Worklist: Mutator:



d is reachable
but unmarked!

Mutator/collector agreement

- Problem arises when:
  - A reference to an unmarked object…
  - is written into a marked object…
  - and no reference remains from an unmarked object.
- If marking is atomic, mutator can check all of this

Return of the revenge of the write barrier

- Write barrier must somehow handle such “bad references”
- Different techniques have different benefits and flaws
- Remember: Safety, liveness, precision

Steele barrier:

```
write(obj, loc, ref):
  *(obj+loc) := ref
  if obj->mark:
    if !ref->mark:
      unmarkAndAddToWorklist(obj)
```

Dijkstra et al barrier:

```
write(obj, loc, ref):
  *(obj+loc) := ref
  if !ref->mark:
    markAndScan(ref)
```

Boehm et al barrier:

```
write(obj, loc, ref):
  *(obj+loc) := ref
  if obj->mark:
    unmarkAndAddToWorklist(obj)
```

Worklist problems

- With less liveness, we can end up in a loop, if the mutator keeps changing the same object
- The only general solution is detection and mutator pausing

More barriers more problems

- With write barriers alone, we miss the roots
- Roots are tricky for barriers: Even the variables in the barriers themselves are roots
- Common fix: Pause mutators to scan roots, collection is done only when roots point to no unmarked objects

Read barriers

- Another solution to the root problem is read barriers
- If we can scan the roots once, the only other place a mutator can get references from is objects: Put the barrier there
- Lets us ignore root changes

Baker barrier:

```
read(obj, loc):
  if !obj->mark or !obj->finished:
    ref := handle(*(obj+loc))
  else:
    ref := *(obj+loc)
  return ref
```

Appel barrier:

```
read(obj, loc):
  if !obj->mark or !obj->finished:
    handle(obj)
  ref := *(obj+loc)
  return ref
```

An object is "finished" if it's been fully scanned and (if applicable) all its references have been updated

(Note: Moving is possible here!)

Breather

- We make sure we don’t miss anything by giving the mutator some of our work
- Mutator needs write barriers, maybe even read barriers, to accomplish this
- Collector might need to rescan objects

Atomically handling worklists

- Mutator must add to worklists while collector is consuming
- Simple locking works but is infeasibly slow
- Even lock-free algorithms are infeasibly slow

Return of the revenge of cards

- Original purpose for card table: Remember objects with inter-partition references
- New purpose for card table: Remember objects which the mutator changed during collection

Steele barrier:

```
write(obj, loc, ref):
  *(obj+loc) := ref
  if obj->mark:
    if !ref->mark:
      unmarkAndAddToWorklist(obj)
```

Mark card

Boehm et al barrier:

```
write(obj, loc, ref):
  *(obj+loc) := ref
  if obj->mark:
    unmarkAndAddToWorklist(obj)
```

Mark card

Re-scan of dirty cards

```
collect():
  clearCardTable()
  clearMarks()
  scanRoots()
  while true:
    doWorklist()
    cardsClean := true
    for each card:
      if card is dirty:
        cardsClean := false
        (mark card as clean)
        addObjectsToWorklist(card)
    if cardsClean: break
```


In concurrent, this likely means "swap meaning of mark bit"

Thinking about progress

Steele barrier:

```
write(obj, loc, ref):
   *(obj+loc) := ref
   if obj->mark:
      if !ref->mark:
         unmarkAndMarkCard(obj)
```

Boehm et al barrier:

```
write(obj, loc, ref):
   *(obj+loc) := ref
   if obj->mark:
      unmarkAndMarkCard(obj)
```

Concurrent vs incremental

- Major difference: Nothing changes out from under you in incremental
- Incremental requires no special atomic operations (typically)

When to GC

- In non-concurrent GC, “when it’s full” was often good enough
- Now, that’s *never* good enough
- If mutator fills all pools during collection, it must stall
- When to allocate new pools same as any other GC

Concurrent copying

- Worst possibility: Mutator, at various times, sees and modifies both copies
- Cleanest possibility: Mutator sees only tospace objects

Mostly-concurrent copying

- Pause all mutators to scan roots and copy their objects to tospace
- Resume mutators
- If mutator finds not-yet-scanned reference, make it do the work

```
read(obj, loc):
  if !obj->finished:
    ref := handle(*(obj+loc))
  else:
    ref := *(obj+loc)
  return ref
```

Fully-concurrent copying

```
read(obj, loc):
  if obj in fromspace:
    obj := handle(obj)
  if !obj->finished:
    ref := handle(*(obj+loc))
  else:
    ref := *(obj+loc)
  return ref
```

So expensive!

- These barriers are very expensive
- Some are at least not atomic (still better than reference counting)
- Expense hugely reduces throughput
- Barriers are totally unnecessary if no GC is happening

Cheaper barrier

```
write(obj, loc, ref):
   if !gcCycleActive:
      *(obj+loc) := ref
   else:
      (full write barrier)
```

Even cheaper barrier

- Modern VMs generate machine code at runtime (JIT)
- Modern VMs allow swapping a function's machine code even while it's running
- Solution: Swap all machine code for barrier version during collection, swap back afterwards
- This magic is outside the scope of this course

Is it worth it?

- With code-swapping, still lower throughput, but not by a lot
- GC latency is either nonexistent (full concurrent) or very small (mostly concurrent)
- So: Worth it if (1) you're in a smart VM and (2) latency is important to you