Mark and compact

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	Regions etc
Nove 16	Adv topics	Adv topics
Nove 23	Presentations	Presentations
Nove 30	Presentations	Presentations

Final presentation

- Short presentation on a memory management paper
- Suggested source: ISMM
  [International Symposium on Memory Management]
- 20 minute limit (conference style)

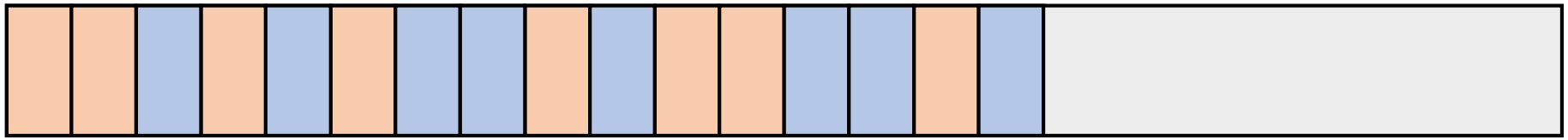
Final project

- Due last day of class
- Topic and style broad: Code, paper, survey, …
- Please discuss with me as soon as you're comfortable
  - (The sooner we agree, the more time you have!)

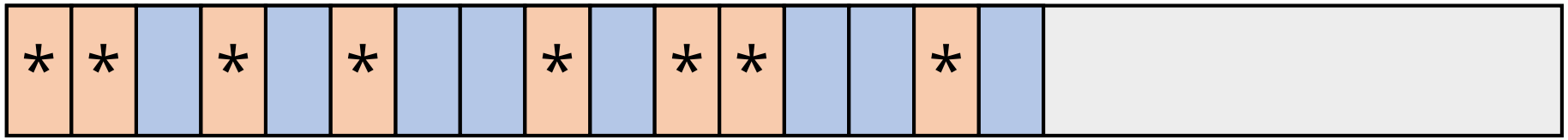
Review

- Roots → objects → reachable objects → discard unreachable
- Mark and sweep: Mark reachable objects, sweep unreachable
- Semispace copying: Copy reachable objects to second heap, ignore unreachable
- Reference counting: Continuously monitor reachability

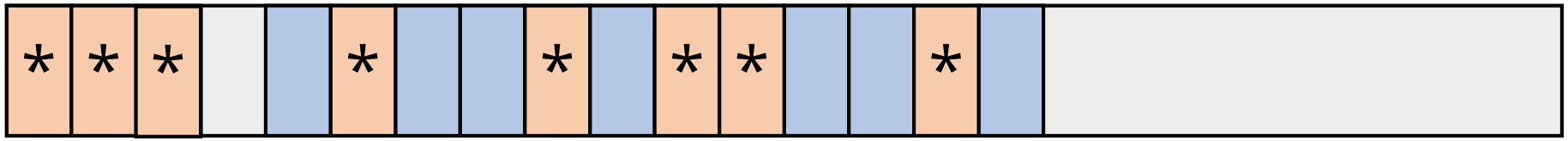
Mark and compact



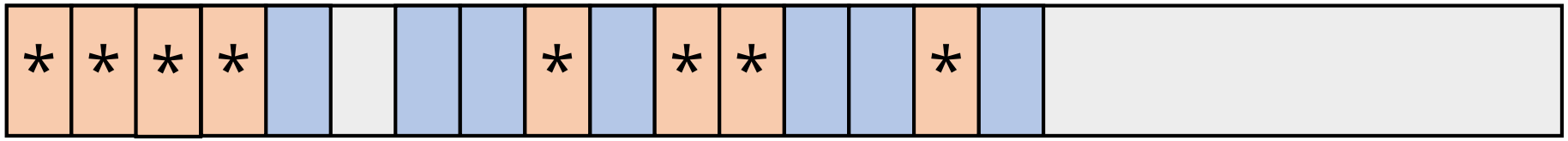
Mark and compact



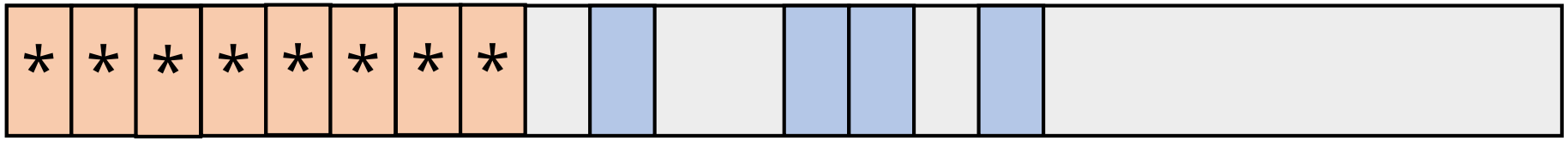
Mark and compact



Mark and compact



Mark and compact



Mark and compact

Thoughts

- Like copying:
  - Simple allocation
  - No fragmentation
  - Objects move
- Like mark-and-sweep:
  - No wasted space
  - Must “sweep” full heap (to move)

Recall copying collection:

- During “mark”, knew forwarding addresses immediately
- Always possible to redirect pointers while marking

```
process(loc):
  fromRef := *loc
  if fromRef != NULL:
    *loc := forward(fromRef)

forward(fromRef):
  if alreadyMoved(fromRef):
    return forwardingAddress(fromRef)
  toRef := (allocate in tospace)
  memcpy(toRef, fromRef, fromRef->header.size)
  setForwardingAddress(fromRef, toRef)
  worklist.push(toRef)
  return toRef
```

Quandary

- In copying collection, update pointers immediately
- In compacting, we must mark before compacting
- Don’t know where to forward pointers until after mark phase: cannot mark-and-forward

Solution

- Three-pass compacting sweep (yuck!)
  - 1: Imaginary compaction to determine forwarding addresses
  - 2: Update references to forwarding addresses
  - 3: Compact

Fewer passes?

- We'll see how later, but…
- Compute locations + update references?
  - Pointer to later object in heap won't be updated
- Update references + compact?
  - Forwarding pointer typically in object header
  - Object header will be overwritten by a compacting object

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Forwarding pointer must be in object header

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Forwarding pointer must be in object header

Allocation is imaginary bump-pointer

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Forwarding pointer must be in object header

Allocation is imaginary bump-pointer

typeInfo could have moved! Stuck putting size in header

You will pay dearly

- Running time O(H)
  - Technically not worse than mark-and-sweep!
- Constant factor very high
  - One mark phase + three sweep phases
- Ever worth it? Maybe! For infrequent GC

Doing better

- Fundamental problem: Forwarding pointers stored with objects, thus overwritten
- So store them separately!
- … but where?

Side tables

- Remember mark bitmaps?
- Store side-table for forwarding addresses
- Expensive to store forwarding per object, so divide pools into “blocks”
- Every object in a block moved by same amount
  - This means some objects unnecessarily stay alive

```
compact():
  computeLocations()
  relocate()

computeLocations():
  lastBlock := -1
  blockMarked := false
  free := start
  foreach obj in pool:
    if blockOf(obj) != lastBlock and blockMarked:
      forwards[lastBlock] = free
      free += WORDS_IN_BLOCK
      blockMarked := false
    if marked(obj):
      blockMarked := true

newAddress(old):
  block := blockOf(old)
  return forwards[block] + old%WORDS_IN_BLOCK

relocate():
  (update root references using newAddress)
  (use forwards table to move blocks)
  (update pointers in living objects using newAddress)
```

With mark bitmaps, this doesn't need to explicitly scan objects, just the mark bitmap

Problem?

- Heap parsability is painful:
  - Dead objects kept alive, but their descriptors may still be dead
  - Objects needn’t start at a block, so half-objects copied
- Still two passes (one of bitmaps only)

More advanced techniques

- Threading:
  - Remove forwarding pointers from object headers by reversing pointers during collection
- Break tables:
  - Keep forwarding info in free space between chunks of marked objects

Object lifetime

- Live objects at beginning of heap not copied
- Most objects die young
- Mark-compact will move long-lived objects to beginning of heap

Pseudo-generational

- Low heap is mostly immortal
- Start compaction part way through the heap: Only compact young objects
- Only compact full heap occasionally
- Saves time moving ancients

When to GC

- Mark-compact is *slow*
- Bigger heap slower, but not proportional
- GC as infrequently as possible
- H >>> L crucial
  - But unlike semispace, we can *use* H

Worth it?

- Next week we will talk about generational garbage collection
- “Young” generation frequently collected
- “Old” generation infrequently collected
- Mark-and-compact is most seen there
- (Copying + mark-and-compact = Java’s GC!)