Algorithms for Dynamic Memory Management (236780)

Lecture 2

Erez Petrank
Topics last week

• Overview on
  – Memory management
  – The 3 classic algorithms
  – Course topics

• The Mark & Sweep algorithm
  – Basics
  – Recursion explicit, pointer reversal, mark-bit table, lazy sweeping, bitwise sweep
The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.
Mark-Compact

• With time the heap gets fragmented.
• When space is too fragmented to allocate, a compaction algorithm is used.
A Header

- The memory manager keeps a header for each object.
- User allocates 24 bytes, the actual allocation is larger!
- Header typically has: length, control bits (for marking an object, synchronization, hashing, etc), pointer to class (for methods and fields types).
Memory Management

Compaction
Overview

• Motivation
  – Fragmentation – problem and solutions.

• Five Algorithms:
  – Two-finger Alg – for objects of equal size.
  – Lisp 2 Alg.
  – Jonkers threaded algorithm
  – SUN’s parallel algorithm
  – IBM’s parallel algorithm
  – (The Compressor, a more advanced algorithm is presented in lecture 10)

• Summary.
Motivation

• Fragmentation is the main drawback of the mark-sweep algorithms.
  – Large objects cannot be allocated (even after GC).
  – Allocation becomes difficult (and inefficient).
  – Increasing heap size means page faults and cache misses.
  – Longer sweep
  – Locality: objects allocated together tend to be accessed together. Thus, mixing allocated objects with “old” objects increases cache-misses.

• Compaction algorithms fix above problems by moving all live objects together.
The Generic Task

• Assume live objects are marked.

• Move objects to one (or a small number of) areas in the heap

• Modify pointers to reference the new locations.
Comparison Criteria

- Complexity:
  - Number of heap passes.
  - Passes over auxiliary tables.
  - Cache performance.

- Extra space required.

- Restrictions on objects (e.g., equal size).

- Compaction quality:
  - Order of objects in output.
  - Number of packed areas (best: 1 area).
Object Ordering

- **Arbitrary** – no guaranteed order.
- **Linearizing** – objects pointing to one another are moved into adjacent positions.
- **Sliding** – maintaining the original order of allocation.

- **Worst**
  - Arbitrary
    - 4 1 3 2

- **Best**
  - Sliding
    - 1 2 3 4

- **Good**
  - Linearizing
    - 1 3 4 2
The Two Finger Algorithm
[Edwards 1974]

- Simplest algorithm:
  - Designed for objects of equal size
  - Order of objects in output is arbitrary.
  - Two passes.

- First pass: compact.
- Second pass: update pointers.
Two finger, pass I - Example

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

© Erez Petrank
Pass I: Compact

- Use two pointers:
  - **free**: search from heap start for free space.
  - **Live**: search from heap end for a live object.
  - When both find, move object to free spot.

- When an object is moved, a pointer to its new location is left at its old location.
Pass II: Fix Pointers

• Go over live objects in the heap
• For each pointer
  – If points to free area: fix it using the forwarding pointer.
Partial Adaption to Variable Sized Objects

• Divide heap to regions.
• Each region has one size objects.
• Perform compaction via two fingers for each region separately.
Two finger – Properties

😊 Simple!
😊 Relatively fast: One pass + pass on live objects (and their previous location).
😊 No extra space required!
😍 Objects restricted to equal size.
😢 Order of objects in output is arbitrary.

😢 This significantly deteriorates program efficiency!
Thus – not used in practice.
The Lisp2 Algorithm

• Goals: handle variable sized objects, keep order of objects.
• Requires one additional pointer field for each object.
• The picture:

  ![Diagram](https://example.com/diagram.png)

• Note: cannot simply keep forwarding pointer in original address. It may be overwritten by a moved object.
The Lisp2 Algorithm

- **Pass 1**: Address computation. Keep new address in an additional header field.
- **Pass 2**: Pointer modification.
- **Pass 3**: Move.

Two pointers (free & live) run from the bottom. Live objects are moved to free space keeping their original order.
Lisp 2 – Properties

😊 Simple enough.
😊 No constraints on object sizes.
😊 Order of objects preserved.
🙁 Slower: 3 passes.
😔 Extra space required – a pointer per object.
Notes on Previous Algorithms

• LISP2: extra space for forwarding pointers & three passes..
• Two-fingers: creates arbitrary order.
• Pointer fix up: using forwarding pointers.
  – Either before moving the objects (LISP2)
  – or after (two fingers).
• The next algorithm is more complicated.
  – Fixing pointers while moving objects.
  – No extra space required.
  – Order of objects preserved.
Jonker’s Algorithm [1979]: Eliminate Extra Space

- **No extra space**: can't keep new location for each object.
- **Where do we move an object?**
- **An important point**: we know where to move each object when we get to it. If we don't keep this information, we lose it.
Jonker’s Algorithm [1979]: Eliminate Extra Space

- No extra space: can't keep new location for each object.
- Where do we move an object?
- An important point: we know where to move each object when we get to it. If we don't keep this information, we lose it.
- Basic idea (threading method): for each object O, keep list of all pointers that reference it. (The pointers are “threaded”.)

Issues to solve:
- list with no extra space = in objects,
- objects that move foil the list structure.
Threading: a List with no Space Overhead

• Observations for a Java-Like Environments.
• Pointers only point to object head.
• JVM keeps a header for each object.
  – Size of header larger than a pointer.
  – Info in header distinguishable from a pointer (e.g., pointer to class info).
• Use this structure to “thread” pointers referencing an object.
  – Let’s thread 3 pointers referencing object D…
Threading Example

Before Threading D

1) header info moves to pointer
2) pointer location put in header.
After threading D with A

1) header info moves to pointer
2) pointer location put in header.
Threading Example

After threading D with A and B

1) header info moves to pointer
2) pointer location put in header.
After threading D with A, B and C

From now on, when we say “thread p” for pointer p, we mean:
1) header of referenced object replaces pointer
2) put pointer location in header.

Threading Example
After threading D with A, B and C

Note that if we move one of the objects now, we destroy the list!
Modify pointers on a threaded list to reference a new location

// Update thread, starting from node P to point to new location of P
update( P, new-location ) {
  next = Heap[ P ];
  while pointer( next )
  
    // Update thread to point to the location of
    // P, free, till data different from pointer
    // reached (‘info’ in our example)
    temp = Heap[next];
    Heap[next] = new-location; // Point to new location
    next = temp; // Get next object to update

  Heap[ P ] = next; // Put ‘info’ back in P
}

© Erez Petrank
A Simplified Version: 3 Passes

• Go over the heap once and thread all pointers.
• Go over the heap again and fix pointers:
  – When reaching an object O, its new address is known.
  – Use the threaded list to fix all pointers to O.
  – Un-thread O's list to restore the heap.
• Go over the heap again and move objects.

Can we do this with only 2 heap passes?
Forwards and Backwards Pointers

• While going over the heap and threading.
• **Observation 1**: when reaching an object in the first pass all forwards pointers to it are threaded.
• Action 1: at that time --- **update** these pointers.
• **Observation 2**: when completing the first pass, all objects have all backwards pointers threaded to them.
• During second pass: **update** the threaded backwards pointers and **move** the object.

Note different terms:
**Forwarding** pointer: a pointer that shows where object has moved
**Forwards** pointer: a property of a pointer (points to higher addresses)
Threaded Methods – P’s Point of View

Initial configuration – forward and backward pointers to P.
Threaded Methods – P’s Point of View

When P is first reached in first pass-all forward pointers to P are threaded.
Threaded Methods – P’s Point of View

After update(P, free) was called by First-pass - forward pointers refer to P’s new location.
Threaded Methods – P’s Point of View

By the end of first-pass backward pointers to P are threaded.
Threaded Methods – P’s Point of View

At the end of update_backward_pointers - backward pointers are updated and P is moved.
Jonker’s Algorithm

• First heap pass: for each object O
  – **Determine** where O should move
  – **Update** all (incoming) forwards pointers to O (already threaded)
  – **Thread** O’s (outgoing) forwards & backwards pointers

• Second heap pass: for each object
  – **Determine** where it should move
  – **Update** all (incoming) backward pointers (already threaded)
  – **Move** object
Jonker’s Algorithm – Pass I

Current object

Current free location
Jonker’s Algorithm – Pass I

Step 1: Update threaded pointers with new location. And return the header.
Jonker’s Algorithm – Pass I

Step 1: Update threaded pointers with new location. And return the header.
Jonker’s Algorithm – Pass I

Step 2: Move free forwards according to the length of orange.

Current object

Current free location
Jonker’s Algorithm – Pass I

Step 2: Move free forwards according to the length of orange.
Step 3: Thread all orange’s pointers to their targets.
Step 3: Thread all orange’s pointers to their targets.

Current object

Current free location
Jonker’s Algorithm – Pass I

Current object

Current free location

Step 4: Move to next object.
Step 4: Move to next object.
Step 1: Update threaded pointers with new location. And return the header.
Jonker’s Algorithm – Pass I

Step 1: Update threaded pointers with new location. And return the header.
Jonker’s Algorithm – Pass I

Step 2: Advance free pointer.

Current object
Current free location
Jonker’s Algorithm – Pass I

Step 2: Advance free pointer.

Current object

Current free location

© Erez Petrank
Jonker’s Algorithm – Pass I

Step 3: Thread red’s pointer

Current object

Current free location
Jonker’s Algorithm – Pass I

Step 3: Thread red’s pointer

Current object

Current free location
Step 4: When trying to move to next object - no more objects.
Jonker’s Algorithm – Pass I

Step 4: When trying to move to next object - no more objects.
Step 1: find first (green) object and update pointers to object.
Step 1: find first (green) object and update pointers to object.
Jonker’s Algorithm – Pass II

Step 2: move (green) object.

Current object

Current free location
Jonker’s Algorithm – Pass II

Step 2: move (green) object.

Current object

Current free location
Moving during Second Pass

- Can’t move an object if its fields are involved in a list.
- Claim: when moving an object (second phase) none of its fields are part of a threaded list.
- Threaded lists: due to its header or pointers.
- It’s header has been handled before move.
- Forwards pointers: have already been handled in first pass.
- Backwards pointers (in this object) point to objects that we are done handling.
Threaded Methods - Forward pointers

First-pass( ){
    for R in Roots // Thread the roots first
        thread ( R );
    free = Heap_bottom; // ‘free’ is a next free space variable,
P = Heap_bottom; // P will be the “live” pointer
    while P <= Heap_top
        if marked( P ) // Check that P is a live object
            update( P, free ); // When P is reached, forward pointers are
            // threaded and can be updated with ‘free’
            for Q a pointer in P // Thread all pointers of a live object
                thread( Q );
            free = free + size( P ); // Location for the next live object
        P = P + size( P ); // Go to next object
}
Threaded Methods - Backward pointers

Second-pass( ) {
    free = Heap_bottom;
    P = Heap_bottom;
    while P <= Heap_top
        if marked( P )  // Check that P is a live object
            update( P, free );  // When P is reached again, backward pointers
                              // are threaded and can be updated with ‘free’.
                              // Self reference is treated as back pointer
            move( P, free );  // Move P to its new location - ‘free’
        free = free + size( P );  // Calculate the location for the next live cell
        P = P + size( P );  // Go to next object
Threaded Methods - Analysis

- No extra space required
- Variable size objects
- Preserves order
- Two passes
- But:
  - each iteration may touch several other objects.
  - requires a header distinguishable from pointer.
Threaded Methods - Analysis

- How many times is each object touched?
  - Once by first pass
  - Once by second pass
  - For each pointer referencing it, it is touched once when threading the pointer.
  - For each pointer in the object, it is touched during update.

- Asymptotic complexity $O(M)$ (who cares?)
## Summary --- Single Threaded Compaction

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space</th>
<th>Passes</th>
<th>Obj size</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-finger</td>
<td>None</td>
<td>2</td>
<td>Fixed</td>
<td>Arbitrary</td>
</tr>
<tr>
<td>LISP2</td>
<td>1 pointer-sized per object</td>
<td>3</td>
<td>Variable</td>
<td>Sliding</td>
</tr>
<tr>
<td>Threaded</td>
<td>(Pointer-sized headers)</td>
<td>2</td>
<td>Variable</td>
<td>Sliding</td>
</tr>
</tbody>
</table>
Parallel Compaction: SUN’s Version

• [Flood Detlefs Shavit Zhang 2001]
• First parallel compaction
• 3 phases (similar to the LISP 2 algorithm):
  – Forwarding pointers installation
  – Fix up pointers phase
  – Move phase
• Each phase done in parallel
Splitting the work

• Heap divided to n regions
  – n is the number of compaction threads
  – Division not uniform; it balances work
• Each region compacted independently so compaction does not use synch’ed operations.
• Number of regions determines “quality” of compaction.
• Trade-off between quality of compaction and load balancing.
Improving quality

• In even regions – push left
• In odd regions – push right

Result: only \( \frac{n}{2} \) piles of objects (rather than \( n \))
Working in parallel

• Phase 1: each thread grabs a region and installs forwarding references.
• Phase 2: each thread grabs a region and updates its pointers
• Phase 3: each thread grabs a region and compacts the objects therein.

• Between phases threads wait for each other.
• Grabbing must be synchronized, the rest of the work is independent.
Properties

😊 Runs in parallel – good scalability!
😊 Keeps order of objects
😊 Objects are not fully packed
😊 Requires extra word per object (or a smart use of the reclaimable space)
😊 Coarse-grained load balancing
😊 3 passes
IBM’s Parallel Compaction

• [Abuaiadh-Ossia-Petrank-Silbershtein 2004]
• A more involved parallelization of the LISP-II compaction algorithm.
• Unlike SUN: Objects are packed to the bottom.
• Space overhead: replace forwarding pointer in each object with a smaller table.
• Two heap passes (each executed in parallel):
  – Move and keep some info
  – Use info to fix up pointers
Parallelism versus Compaction

- First goal: compact all objects together instead of creating several piles of objects.
- Heap is divided to \( n \) areas
- For example: \( n = 64 \) was used for a 640MB heap and 8 processors.
Squeezing the Objects in Spite of Parallelism

• **The goal**: move all objects to the lower addresses.
• Each thread compacts one area at a time.
• **Beginning**: each area is compacted into itself.
• **After a while**:
  • vacant spaces appear in compacted areas.
  • compact objects of one area into the free space of a lower area
First Phase: Moving the Objects

• A thread picks the next area to be compacted;
• it finds a lowest area with empty space to compact into;
• if no such area exists, it compact to the bottom of the same area.

• While moving the objects, record information in a small additional table that will enable updating the pointers.
  – This replaces the forwarding pointers.
  – It implements a map from old to new addresses.
Moving the objects: an Example

- Two threads, 4 area
  - (Thread#1, red area), (Thread#2, blue area)
  - (Thread#1, brown area), (Thread#2, green area)

At the end
More areas

- 4 threads, 64 areas,
- In the end we may have some holes at the last areas
- For a reasonable number of areas, these holes are insignificant.
## Area Size Tradeoff

<table>
<thead>
<tr>
<th></th>
<th>“Holes” in the Heap</th>
<th>Preserve allocation order</th>
<th>Load balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oversized areas</td>
<td>-</td>
<td>😊</td>
<td>-</td>
</tr>
<tr>
<td>“Normal” size</td>
<td>😊</td>
<td>😊</td>
<td>😊</td>
</tr>
<tr>
<td>Areas too small</td>
<td>😊</td>
<td>-</td>
<td>😊</td>
</tr>
</tbody>
</table>
Phase 2: Fix up

- Divide the heap to \( n \) areas.
- Each thread fixes up pointers in one area at a time.

Remember: Information is recorded during the move phase to allow redirecting the pointers in the second phase.
Implementing the Fix-Up Map

- We consider the heap as a sequence of blocks (say, block = 256 bytes)
- Blocks (256 bytes) << areas (10 Mbytes).
- Information is recorded per block rather than per object.
  - Objects in a block are moved together; objects of different blocks are never interleaved.

- The idea: record less information per block, but perform more computation during fix up of each reference.
The Block Table

Old Heap:

- Block 175
- Block 176
- Block 177

New Heap:

- .
- .
- .

Ptr 175
Ptr 176
Ptr 177
Recorded Information

• **Block table**: For each block keep the new location of the first object in the block.
  – One pointer per block.

• **Two bit maps** (1 bit for any 8 bytes).
  – **Old bitmap** represents location of objects before the move (created while marking live objects)
  – **New bitmap** represents location of objects after the move (created while moving the objects).
  – One bit stands for 8 bytes in the heap (8-byte alignment)
Calculating a New Location

- Given an old address of an object A:
- Find A’s block (its most significant bits)
- Using the block table, obtain the new address (B) of the first object in the block.
- Using the old bitmap: find the ordinal number (i) of the object in the block.
- Using the new bitmap: find the relative new location (r) of the i-th object in the block.
- Add B+r to obtain the new location.
Example

• Calculating the new location of object C.
• Old bitmap ➔ C is third in block (i=3)
• New bitmap ➔ relative address of C (to A) (r = 0x18)
• Block table ➔ new address of A = 0x58296200
• A + r = new location = 0x58296218
Space overhead

• For each block (say, 256 bytes),
  – A pointer: 4 (or 8 for 64-bits platforms) bytes
  – 2 Bitmaps: 4+4 bytes
  – Overall: 12 (or 16) bytes for each 256 bytes (4.7-6.2%)

• Existing data structures may be reused, e.g., the GC markbits table.

• Increasing the size of the block: reduces the extra space but increases the computation cost.
Properties

- Almost all objects are condensed to the bottom of the heap.
- Order of objects is essentially preserved.
- Good parallelism with almost no contention.
- Space overhead low compared to forwarding pointers.
Measurements

• Algorithms compared:
  – Jonker’s threaded algorithm
  – Restricted parallel algorithm (to a single thread)
  – Fully parallel algorithm
• Platform: AIX (on 8-way PPC, 64 bits) and NT (on 4-way Pentium, 32 bits)
• Benchmarks: Specjbb2000 and Trade 3 on Websphere.
• Heap size: determined so that live objects take 60% of the heap: 600MB for SPECjbb and 180MB for Trade3.
Specjbb2000

- Compaction runs when a warehouse is added, those (substantial) parts of the run are not considered for the measurements.
- Thus, throughput is not affected by the compaction times.
  - May be affected by bad compaction quality.
- We measure compaction times.
Results: Compaction Times for (Specjbb2000) on a Uniprocessor

<table>
<thead>
<tr>
<th>Warehouses</th>
<th>Threaded</th>
<th>parallel-restricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>750</td>
<td>750</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>2250</td>
<td>2250</td>
</tr>
<tr>
<td>5</td>
<td>3000</td>
<td>3000</td>
</tr>
</tbody>
</table>

![Graph showing compaction times for warehouses.](image)
Results: Speedup (Specjbb2000)
Results: Throughput (Specjbb2000)
Results: Trade3 (Websphere)

- 4-way NT machine
- Heap size: 180MB
- Additional test: we forced compaction each 20gc

<table>
<thead>
<tr>
<th>Compaction type</th>
<th>Compaction time</th>
<th>#Requests per second</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Triggering</td>
<td></td>
</tr>
<tr>
<td></td>
<td>≈ 90 gc default</td>
<td>20gc</td>
</tr>
<tr>
<td></td>
<td>≈ 90 gc default</td>
<td>20gc</td>
</tr>
<tr>
<td>Threaded</td>
<td>1698</td>
<td>1671</td>
</tr>
<tr>
<td>Parallel-restricted</td>
<td>1387</td>
<td>1251</td>
</tr>
<tr>
<td>Parallel</td>
<td>499</td>
<td>440</td>
</tr>
<tr>
<td></td>
<td>219.8</td>
<td>221.7</td>
</tr>
<tr>
<td></td>
<td>222.4</td>
<td>226.1</td>
</tr>
<tr>
<td></td>
<td>224.5</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion --- IBM’s Parallel Compaction Algorithm

- More efficient than the previously used threaded algorithm even on a uniprocessor.
- Good speedup
- Good compaction quality.
The Compressor

- [Kermany-Petrunk 2006]
- The goal: concurrent and parallel compaction with low overhead.
- Overhead reduction via a single heap pass.
- Extending with parallelism and concurrency:
  - Objects are packed to the bottom, maintaining address order.
- We will study the Compressor around the 10th lecture.
Conclusion --- Compaction

• Uniprocessor compaction:
  – Two fingers, Lisp2, Threaded (Yonkers)

• Parallel compaction:
  – Sun’s compaction, IBM’s compaction.
  – (Compressor: parallel and concurrent, delayed…)

• Issues considered:
  – Efficiency, space overhead, parallelism, compaction quality, locality.