Algorithms for Dynamic Memory Management
(236780)
Lecture 1

Lecturer: Erez Petrank
Class on Wednesdays 10:30-12:30, Taub 5
Reception hours: Wednesdays, 14:30
Office 528, phone 829-4942
Web: http://www.cs.technion.ac.il/~erez/courses/gc
Topics today

• Administration
• Overview on memory management:
  • 3 classic algorithms.
• Course topics
• The Mark & Sweep algorithm
Grades

- Test (closed material).
- Homework (about 3 (dry) exercises)
- Grade:
  - Test - at least 75%,
    The test will occur early.
  - Homework - at least 10%
  - Participation in class - at most 15%.
The Book


(Available at the library)
The nature of this course

- This is an applied course.
- Memory management attracts attention from the programming languages community, the algorithms community, and the systems community.
  - We will tend to focus on the algorithmic aspects, system requirements, and parallelism.
  - We’ll mention standard engineering techniques.
The course of this course

- We’ll start with simple algorithms.
- Then, their evolution according to studies on “typical” programs/systems behavior.
- Next, we’ll cope with the platforms
  - The importance of cache misses
  - Parallel machines
  - Real time requirements
  - Distributed machines
- One theoretical lower bound
- The focus: what is actually used today in the industry.
FAQ

• זה קשה?
• יש הרבה עבודה?
• איך הצוイン?
• איך הצוイン?
קורס בחירה

משאבים

חומר כתוב מלא (ספר)

חוברות

תירגולים, תירגולי זדヘ, בحياة פتروוה, ...

וידיאו

השלמות
What is it?

The classical algorithms
Dynamic allocation

- All programs use local variables --- static allocation (integer i).
- But sometimes dynamic allocation is required.
  - Usually in more involved programs where length or number of allocated objects depends on the input.
- "Manual" dynamic allocation and de-allocation:

```c
Ptr = malloc (256 bytes);
/* Use ptr */
Free (Ptr);
```
Is “manual” management good?

- Practice shows that manual allocation is problematic.
  1. Memory leaks.
  2. Dangling pointers.

- In large projects in which objects are shared by various components of the software, it is sometimes difficult to tell when an object is not needed anymore.
Solution

- **Automatic memory management:**
  - User allocates space for an object.

```java
course c = new course(236780)
c.class = "TAUB 3"
Faculty.add(c)
```
Solution

- **Automatic memory management:**
  - User allocates space for an object.
  - When system “knows” the object will not be used anymore, it reclaims its space.

- “Knows”?
  - Telling whether an object will be accessed after a given line of code is undecidable.
  - A conservative approximation is used instead.
Solution

- **Automatic memory management:**
  - User allocates space for an object.
  - When system “knows” the object will not be used anymore, it reclaims its space.

- “Knows”?
  - Reachability: the program does not have a path of pointers from a program variables to the object.
    (A simple approximation that usually works well.)
  - Other ideas: the compiler can sometimes tell, the user can help with “hints”, etc.
What's good about automatic “garbage collection”?

- **Software engineering:**
  - Relieves users of the book-keeping burden.
  - Stronger reliability, faster debugging.
  - Code understandable and reliable. (Less interaction between modules.)

- **Security (Java):**
  - Program never gets a pointer to “play with”.
GC and languages

- Sometimes it’s built in:
  - LISP, Java, C#.
  - The user cannot free an object.
- Sometimes it’s an added feature:
  - C, C++.
  - User can choose to free objects or not. The collector frees all objects not freed by the user.
- Most modern languages are supported by garbage collection.
Most modern languages rely on GC

<table>
<thead>
<tr>
<th>Well-known languages supported by garbage collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionScript (2000)</td>
</tr>
<tr>
<td>AspectJ (2001)</td>
</tr>
<tr>
<td>Managed-C++ (2002)</td>
</tr>
<tr>
<td>CLU (1974)</td>
</tr>
<tr>
<td>Dynace (1993)</td>
</tr>
<tr>
<td>Emerald (1988)</td>
</tr>
<tr>
<td>F# (2005)</td>
</tr>
<tr>
<td>Haskell (1990)</td>
</tr>
<tr>
<td>Java (1994)</td>
</tr>
<tr>
<td>Limbo (1996)</td>
</tr>
<tr>
<td>Lua (1994)</td>
</tr>
<tr>
<td>Miranda (1985)</td>
</tr>
<tr>
<td>Objective-C (2007)</td>
</tr>
<tr>
<td>PHP (1995)</td>
</tr>
<tr>
<td>Python (1991)</td>
</tr>
<tr>
<td>Sather (1990)</td>
</tr>
<tr>
<td>Self (1986)</td>
</tr>
<tr>
<td>SISAL (1990)</td>
</tr>
<tr>
<td>Squeak (1996)</td>
</tr>
<tr>
<td>VB.NET (2002)</td>
</tr>
<tr>
<td>Algol-68 (1965)</td>
</tr>
<tr>
<td>Beta (1983)</td>
</tr>
<tr>
<td>Cecil (1992)</td>
</tr>
<tr>
<td>D (2007)</td>
</tr>
<tr>
<td>Eiffel (1986)</td>
</tr>
<tr>
<td>Erlang (1990)</td>
</tr>
<tr>
<td>Green (2005?)</td>
</tr>
<tr>
<td>Hope (1978)</td>
</tr>
<tr>
<td>JavaScript (1994)</td>
</tr>
<tr>
<td>Lingo (1991)</td>
</tr>
<tr>
<td>Mercury (1993)</td>
</tr>
<tr>
<td>ML (1990)</td>
</tr>
<tr>
<td>Obliq (1993)</td>
</tr>
<tr>
<td>Pliant</td>
</tr>
<tr>
<td>Rexx (1979)</td>
</tr>
<tr>
<td>Scala (2003)</td>
</tr>
<tr>
<td>SETL (1969)</td>
</tr>
<tr>
<td>Smalltalk (1972)</td>
</tr>
<tr>
<td>tcl (1990)</td>
</tr>
<tr>
<td>VBScript (1995?)</td>
</tr>
<tr>
<td>AppleScript (1993)</td>
</tr>
<tr>
<td>C (1999)</td>
</tr>
<tr>
<td>Clean (1984)</td>
</tr>
<tr>
<td>Dylan (1992)</td>
</tr>
<tr>
<td>Elastic-C (1997)</td>
</tr>
<tr>
<td>Euphoria (1993)</td>
</tr>
<tr>
<td>Groovy (2004)</td>
</tr>
<tr>
<td>Icon (1977)</td>
</tr>
<tr>
<td>Liana (1991)</td>
</tr>
<tr>
<td>LotusScript (1995)</td>
</tr>
<tr>
<td>Modula-3 (1988)</td>
</tr>
<tr>
<td>Oberon (1985)</td>
</tr>
<tr>
<td>Perl (1986)</td>
</tr>
<tr>
<td>PostScript (1982)</td>
</tr>
<tr>
<td>Ruby (1993)</td>
</tr>
<tr>
<td>Scheme (1975)</td>
</tr>
<tr>
<td>Simula (1964)</td>
</tr>
<tr>
<td>Snobol (1962)</td>
</tr>
<tr>
<td>Theta (1994)</td>
</tr>
<tr>
<td>VHDL (1987)</td>
</tr>
<tr>
<td>YAFFL</td>
</tr>
</tbody>
</table>

Languages developed since 1990 not supported by garbage collection

- Alef (1995)                                      
- Autot (1999)                                    
- Cilk (1995)                                     
- Delphi (partly, 1995)                           
- Visual Basic (1991)                             
- Befunge (1993)                                  
- Goedel (1994)                                   

What’s bad about automatic “garbage collection”?

- It has a cost:
  - Old Lisp systems 40%.
  - Today’s Java program (if the collection is done “right”) 5-15%.

- Considered a major factor determining program efficiency.

- Techniques have evolved since the 60’s. In this course we investigate the ideas developed from then until now.
How have the techniques evolved?

- Hard to compare collection algorithms.
- Asymptotic analysis not relevant.
- Implementation is sometimes more important than the algorithm.

- But - good ideas caught.
Note:

- We will discuss memory management in the context of a programming language (mostly with Java).
- The ideas are useful for
  - other languages
  - operating systems memory management
  - disks management, etc.
Memory Systems Impact

- Memory access is the performance bottleneck in modern computing systems (both time & energy).
The Classical Algorithms
Reference counting [Collins 1960]

- Recall that we would like to know if an object is reachable from the roots.
- Associate a reference count field with each object: how many pointers reference this object.
- When nothing points to an object, it can be deleted.
- Very simple, used in many systems.
Basic Reference Counting

- Each object has an RC field, new objects get o.RC:=1.
- When p that points to o₁ is modified to point to o₂ we execute: o₁.RC--, o₂.RC++.
- if then o₁.RC==0:
  - Delete o₁.
  - Decrement o.RC for all “children” of o₁.
  - Recursively delete objects whose RC is decremented to 0.
3 years later...

- [Harold-McBeth 1963] The Reference counting algorithm does not reclaim cycles!
- But:
  - “Normal” programs do not use too many cycles.
  - So, other methods are used “infrequently” to collect the cycles.
During the years

- Many improvements over the basic reference counting method.
- Cycle collection algorithms.
- To be discussed in a separate lecture.
The Mark-and-Sweep Algorithm [McCarthy 1960]

- **Mark phase:**
  - Start from **roots** and traverse all objects **reachable** by a path of pointers.
  - Mark all traversed objects.

- **Sweep phase:**
  - Go over all objects in the heap.
  - Reclaim objects that are not marked.
The Mark-Sweep algorithm

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

Note! This is not the heap data structure!
Mark-Compact

- Gradually, the heap gets fragmented.
- When space is too fragmented to allocate, a compaction algorithm is used.
- We will see several compactors. Issues:
  - Keep order of objects?
  - Use extra space?
  - How many heap passes?
  - Parallelism?
Copying garbage collection

- **Heap** partitioned into two.
- Part 1 takes all allocations.
- Part 2 is reserved.
- During GC, the collector traces all reachable objects and copies them to the reserved part.
- After copying, activity goes to part 2. Part 1 is reserved till next collection.
Copying garbage collection

Part I

Part II

Roots

A  B

C

D  E
The collection copies...
Roots are updated; Part I reclaimed.
Properties

- Compaction for free
- **Major disadvantage:** half of the heap is not used.
- “Touch” only the live objects
  - **Good when most objects are dead.**
A very simplistic comparison

<table>
<thead>
<tr>
<th>Copying</th>
<th>Mark &amp; sweep</th>
<th>Reference Counting</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live objects</td>
<td>Size of heap (live objects)</td>
<td>Pointer updates + dead objects</td>
<td></td>
</tr>
<tr>
<td>Half heap wasted</td>
<td>Bit/object + stack for DFS</td>
<td>Count/object + stack for DFS</td>
<td>Space overhead</td>
</tr>
<tr>
<td>For free</td>
<td>Additional work</td>
<td>Additional work</td>
<td>Compaction</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>Mostly short</td>
<td>Pause time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cycle collection</td>
<td>More issues</td>
</tr>
</tbody>
</table>
Some terms to be remembered

- Heap, objects
- Allocate, free (deallocate, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, tracing algorithms
- Fragmentation
Course Topics
Course Topics

- Introduction + Mark and Sweep algorithms
- Compaction algorithms
- Copying algorithms
  - Baker’s incremental copying [1978].
- Generational collectors
  - The Train algorithm.
- Concurrent & On-the-fly algorithms
  - Mostly concurrent collection (SUN, IBM, BEA, others?).
  - On the fly collection.
- Parallel collection.
Course Topics

- Snapshot & Sliding Views collections.
- Reference Counting collectors (also sliding views).
- Cycle Collection.
- Allocation techniques.
- Cache-conscious garbage collection:
  - Including an impossibility result.
- Real-time support.
- Distributed garbage collection.
Additional Possible Topics

- Conservative garbage collection
- Worst-case fragmentation.
- Verifying memory managers.
- Related compiler and runtime issues
- Other collection ideas: age-oriented, age-based, regions, etc.
Mark & Sweep
The basic idea
[McCarthy 1960]:

1. Mark all the reachable objects from the roots.
2. Scan all the heap and reclaim all unmarked objects.

John McCarthy (1927--2011) in PLDI 2002
Triggering

New(A)=
   if no available allocation space
      mark_sweep()
      if no available allocation space
         return ("out-of-memory")
   pointer = allocate(A)
   return (pointer)
# Basic Algorithm (Cont.)

mark_sweep() =
   for Ptr in Roots
      mark(Ptr)
   sweep()

mark(Obj) =
   if mark_bit(Obj) == unmarked
      mark_bit(Obj) = marked
   for C in Children(Obj)
      mark(C)

Sweep() =
   p = Heap_bottom
   while (p < Heap_top)
      if (mark_bit(p) == unmarked) then free(p)
      else mark_bit(p) = unmarked;
      p = p + size(p)
Properties of Mark & Sweep

- **Does not move objects:**
  - ![Conservative collection possible (when pointers cannot be accurately identified).](image)
  - ![Fragmentation](image)

- **Complexity:**
  - ![Mark phase: live objects (dominant phase)](image)
  - ![Sweep phase: heap size.](image)

- **Termination:** each pointer traversed once.

- **Most popular method today (at a more advanced form).**
Standard Engineering Techniques

• Mark:
  • Explicit mark-stack (avoid recursion)
  • Pointer reversal
  • Using bitmaps

• Sweep:
  • Using bitmaps
  • Lazy sweep
Making recursion explicit

- **Problem**: if object graph is “unsuitable”, then recursion becomes too deep.
  - Large space overhead (compiler stack frames)
  - Inefficient execution (function calls)
  - Potential crash (user does not understand why)

- **Solution**: use iterative loops and auxiliary data structure instead of functions calls.
Modifying mark-roots

\[
\text{mark\_sweep}() = \\
\text{for } ptr \text{ in Roots} \\
\text{mark}(ptr) \\
\text{sweep()}
\]

\[
\text{mark\_sweep}() = \\
\text{mark\_stack} = \text{empty} \\
\text{for } obj \text{ referenced by Roots} \\
\text{mark\_bit}(obj) = \text{marked} \\
\text{push}(obj, \text{mark\_stack}) \\
\text{Mark\_heap()} \\
\text{sweep()}
\]
Modifying heap scan

```markdown
mark(Obj)=
if mark_bit(Obj) == unmarked
    mark_bit(Obj) = marked
for C in Children(Obj)
    mark(C)
```

```python
mark_heap()=
while mark_stack != empty
    obj = pop (mark_stack)
    for C in children (obj)
        if mark_bit (C) == unmarked
            mark_bit(C) = marked
        push (*C, mark_stack)
```
Implementing the stack

- “Standard” implementation: linked list of small mark_stacks.
- **Dealing with markstack overflow:**
  - Using cyclic stack (Knuth) or dropping overflowed objects (Bohem et al)
  - After dropping objects from the stack, it is enough to go over the heap to find mark -> unmarked references.
Where is the mark-bit?

- **Option 1:** within the object’s header.
  - Problems: locality during sweep, concurrency of collector-program.

- **Option 2:** Keep a mark-bit table.
  - Associate the object and the bit by address translation, meaning fixed space per object (most popular).
  - Almost a must for sweep:
    
    *Go over a short table and find sequences of zero’s.*
Lazy Sweeping

- Sweep takes time while the program is halted (pause time), and it is bad for the cache.
- Can we break sweep into small segments of work?
- Lazy sweep [Hughes]: when allocating, sweep until finding an appropriate free space.
  - Better locality
  - Shorter pauses.

Cache behavior is a serious issue!
Bitwise Sweep

- A major overhead of sweep is finding & freeing each chunk of 0’s in the mark-bit table.
- **Solution**: free only large chunks of objects.
  - Sweep uses only full zero words (fast search).
  - When found - free the entire chunk of zeros (not only the space presented by the zero word).
- **Advantage**: looking for a zero **word** is fast.
- **Disadvantage**: some space wasted (but only small chunks).
Pointer Reversal
(The following is an excerpt from the Data Structures course)

- Marking with “no extra auxiliary space”.
- Mark info kept inside objects.
- Stack replaced by reversing pointers during graph traversal.
Pointer reversal: when visiting a node, keep a pointer to its parent in the left or right pointer field. A node will be labeled \{u,r,l\} signifying which pointer points to the parent. Three extra variable: root (of the tree), parent (of visited), and w.
,mapa

ללא סימון

parent

w

L

U

U

parent

w

L

U

U

U

U

U

U

U
סימון ללא ורקסה
סרפורדיה

לא ללא קורסייה

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

w

parent

W

parent

w
simdוי lakh kortsih

parent

w

R

U

L

R

U

L

parent

w
In the end of the run all visited objects are marked with R.
Pointer Reversal -
variable sized nodes

- Two additional variables for each node:
  - i-field - current recursion index.
  - n-field - node’s number of children.
Conclusion (Pointer Reversal)

😊 No additional data structure required.
😊 Every heap’s node contains extra information – $2\log_2 n$ bit/node.
😢 Every node’s visit includes several fields’ update.
Conclusion (Mark & Sweep)

- Mark and sweep is a simple method. Advanced versions are common in real systems.
- Lots of engineering tricks are available. (We've seen a few.)

- One of the main obstacles is fragmentation.

Our next topic is compaction.