Algorithms for Dynamic Memory Management
(236780)
Lecture 1

Lecturer: Erez Petrank
Class on Tuesdays 10:30-12:30, Taub 9
Reception hours: Tuesdays, 13:30
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)
  - Type and email to Nachshon
- Grade:
  - Test - at least 75%
  - 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
  - aiming at “typical” programs/systems behavior.
  - handling “typical” platforms
- Handling platforms:
  - The importance of cache misses
  - Parallel machines
  - Real time requirements
  - Distributed machines
- One theoretical lower bound
- The focus: basic ideas and what is in use today.
FAQ

• הז קשה?
• יש חרב עבודה?
• איךנצח?
קורס בחירה

- משאבים
- חומר כתוב מלא (ספר)
- חוברות
- תרגולים, תרגולים מחזור, בחינות פתרונות...
- וייאו
- השלמות
קורס בחירה

ש

הרצאות
שקפים
בית
תרגילי בית

הרצאות
שקפים

9
עד אדמיניסטרציה

• איני חובה נוכחות
• איני דרכ להשלים הרצאות
•rí או אם אני אפשר לבוא להרצאות עדפי לוותר על הקורס

• יש חובה קדימה
• אלגוריתמים 1
• מערכות הפועלות
Memory Management Overview

- 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:

  ```
  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.addCourse(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++.

- 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>YAFFL</td>
</tr>
</tbody>
</table>

Languages developed since 1990 not supported by garbage collection

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

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 discuss memory management in the context of Java.
- The ideas are useful for
  - other programming languages
  - operating systems memory management
  - disks management, etc.
Memory Systems Impact

Memory access is the bottleneck (time & energy).
The Classical Algorithms
Reference counting [Collins 1960]

- Goal: determine when an object is unreachable from the roots.
  - Roots: local pointers, global pointers, Java Native Interface, etc.

- Associate a reference count 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 $o_1$ is modified to point to $o_2$, execute: $o_1.RC --$, $o_2.RC ++$.
- If then $o_1.RC == 0$:
  - Delete $o_1$.
  - Decrement $o.RC$ for all “children” of $o_1$.
  - 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.
Gradually, the heap gets fragmented.

Compaction algorithms compact the heap.

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

Roots

A
B
C
D
E

Part II

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

- Heap, objects
- Allocate, free (deallocating, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, tracing algorithms
- Fragmentation
Mark & Sweep
The basic idea [McCarthy 1960]:

1. Mark all objects reachable from the roots.
2. Scan heap and reclaim 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).
  ☹ Fragmentation

• Complexity:
  ☺ Mark phase: live objects (dominant phase)
  😞 Sweep phase: heap size.

• 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

```
mark_sweep()=
    for ptr in Roots
        mark(ptr)
    sweep()

mark_sweep()=
    mark_stack = empty
    for obj referenced by Roots
        mark_bit (obj) = marked
        push(obj, mark_stack)
    Mark_heap()
    sweep()
```
Modifying heap scan

\[
\text{mark}(\text{Obj}) = \\
\text{if mark\_bit(Obj) == unmarked} \\
\text{mark\_bit(Obj) = marked} \\
\text{for C in Children(Obj)} \\
\text{mark(C)}
\]

\[
\text{mark\_heap()} = \\
\text{while mark\_stack != empty} \\
\text{obj = pop (mark\_stack)} \\
\text{for C in children (obj)} \\
\quad \text{if mark\_bit (C) == unmarked} \\
\quad \text{mark\_bit (C) = marked} \\
\quad \text{push (*C, mark\_stack)}
\]
Implementing the stack

- “Standard” implementation: linked list of small mark_stacks.
- Dealing with mark stack overflow:
  - Using cyclic stack (Knuth) or dropping overflowed objects (Bohem et al)
  - After dropping objects, resurrect by searching the heap for mark -> unmarked references.
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.