Algorithms for Dynamic Memory Management (236780)
Lecture 1

Lecturer: Erez Petrank

Class on Tuesdays 10:30-12:30

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

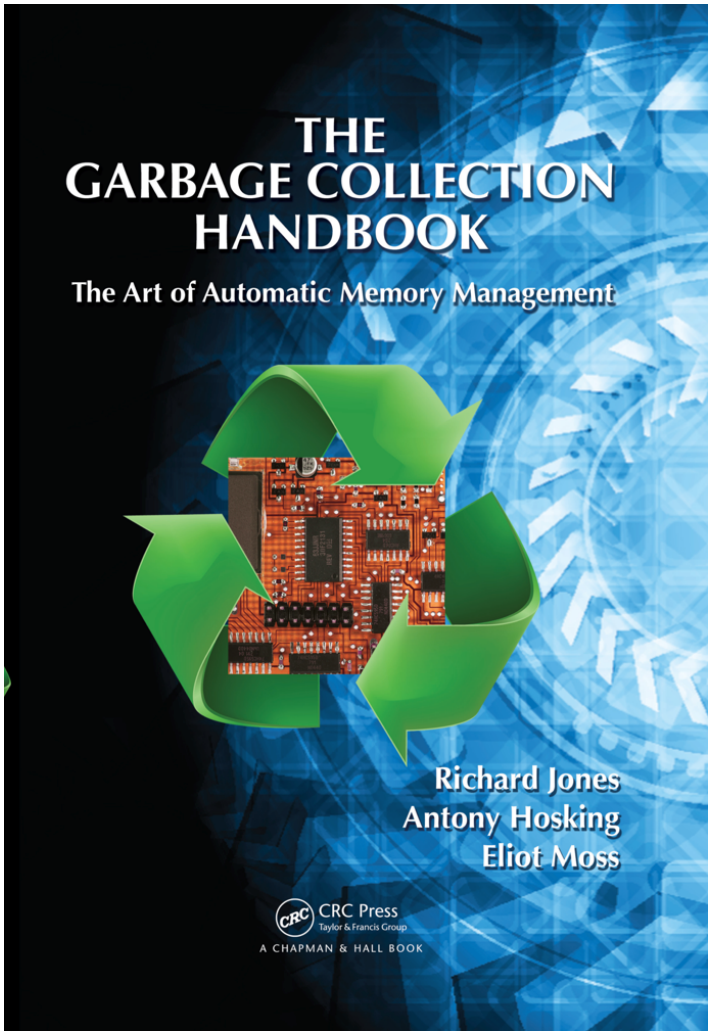
Grades

- Test (closed material).
- Homework (about 3 (dry) exercises)
  - In the cells on floor 1
- Grade:
  - Test - at least 75%
  - Homework – at least 10%
  - Participation in class – at most 15%.

The Book

"The Garbage Collection Handbook: The Art of Automatic Memory Management" by Richard Jones, Anthony Hosking, and Eliot Moss.

(Available at the library)



The nature of this 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.

- This is an applied course.

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 algorithms in practical use.

FAQ

- זה קשה?
- יש הרבה עבודה?
- איך הציונים?

קורס בחירה

אין

- משאבים
- חומר כתוב מלא (ספר)
- חוברות
- תירגולים, תירגולי חזרה, בחינות פתורות, ...
- משלוח אימיילים להתראה שיש תרגיל בית עבור אנשים שלא באים להרצאות
- וידיאו
- השלמות

קורס בחירה

- הרצאות
- שקפים
- תרגילי בית

עוד אדמיניסטרציה

- אין חובת נוכחות
  - אבל אין דרך להשלים הרצאות
  - אז אם אי אפשר לבוא להרצאות עדיף לוותר על הקורס
- יש חובת קדמים
  - אלגוריתמים 1
  - מערכות הפעלה

הודעות

- אתר הקורס:

http://www.cs.technion.ac.il/~erez/courses/gc/index.html

(אפשר לחפש: Algorithms for Dynamic Memory Management)

- אנא הירשמו ל-GR.

- קריטי הסמסטר: חדר הלימוד!

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.

```
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

Well-known languages supported by garbage collection

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

Languages developed since 1990 not supported by garbage collection

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

Source: "The Garbage Collection Handbook" by Richard Jones, Anthony Hosking, and Eliot Moss.

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.

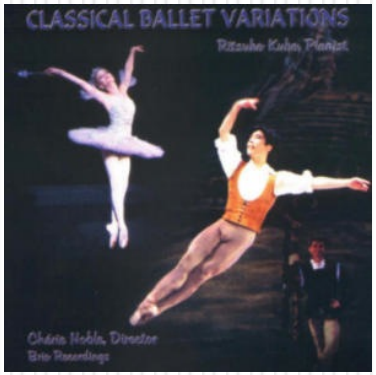
Note:

- 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]

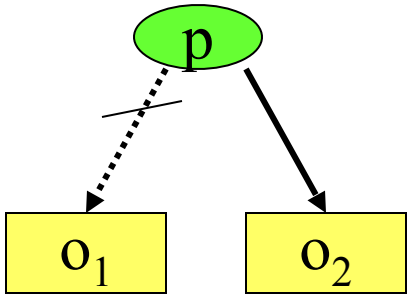
- Goal: determine when an object is unreachable from the roots.
  - Roots: local pointers, global pointers, Java Native Interface, etc.

- 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.

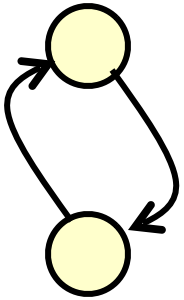
Basic Reference Counting

- 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

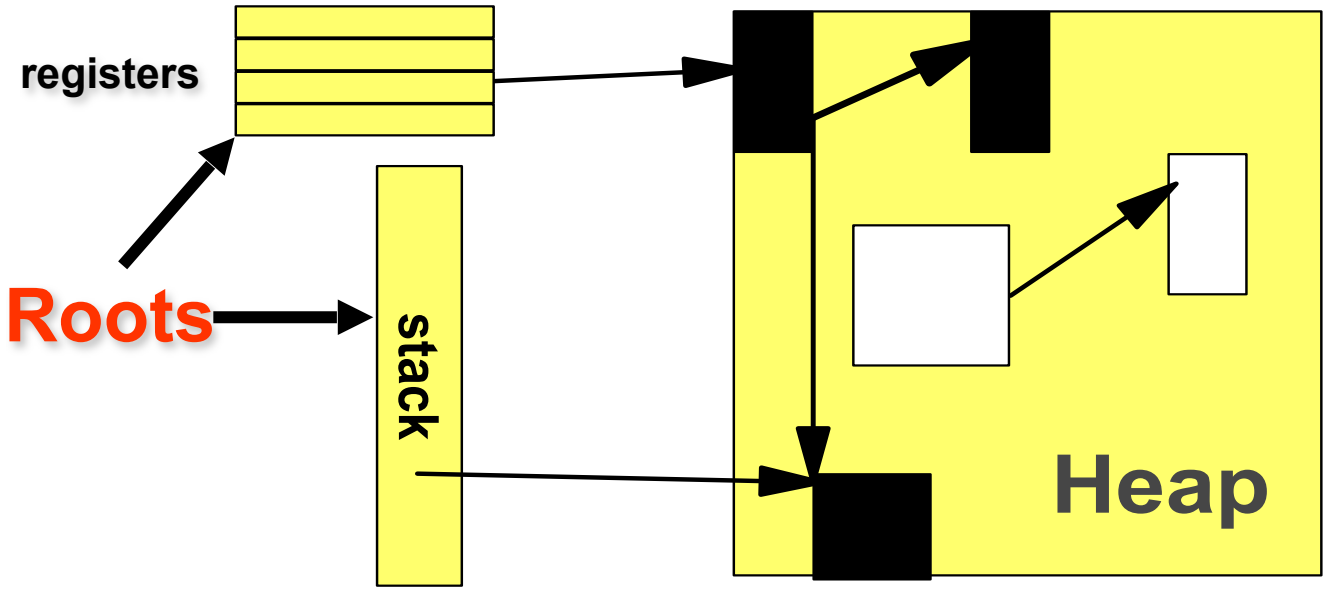
- Many improvements over the basic reference counting method.
- Cycle collection algorithms.
- To be discussed in a separate lecture.

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.

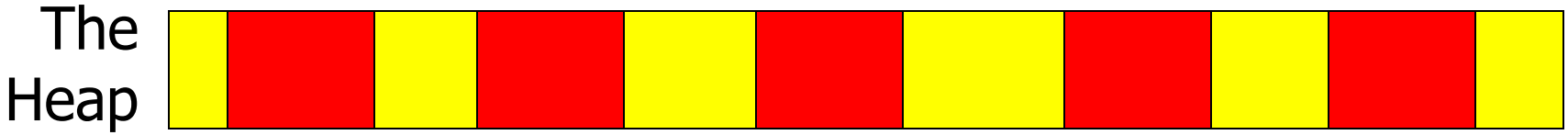
The Mark-Sweep algorithm

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

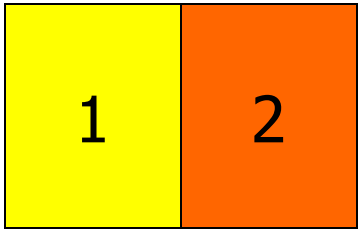


Mark-Compact

- 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?

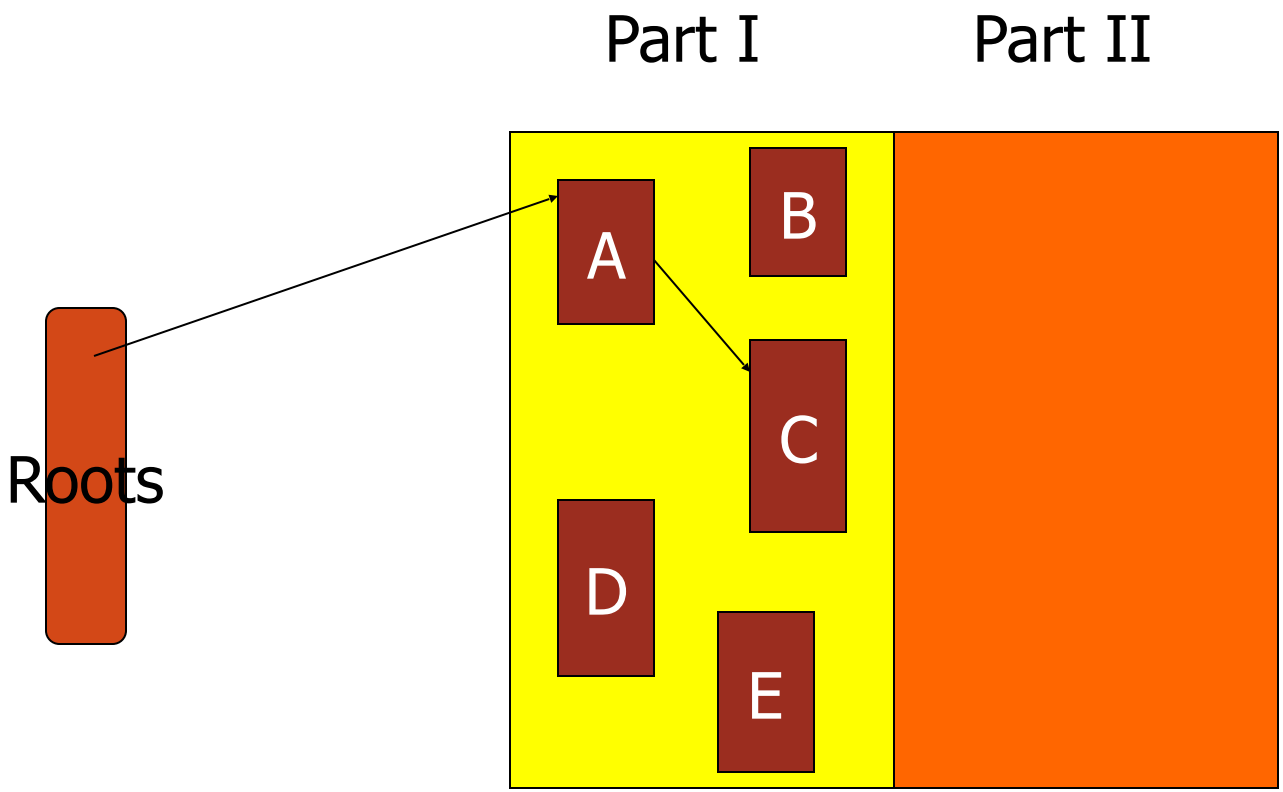


Copying garbage collection

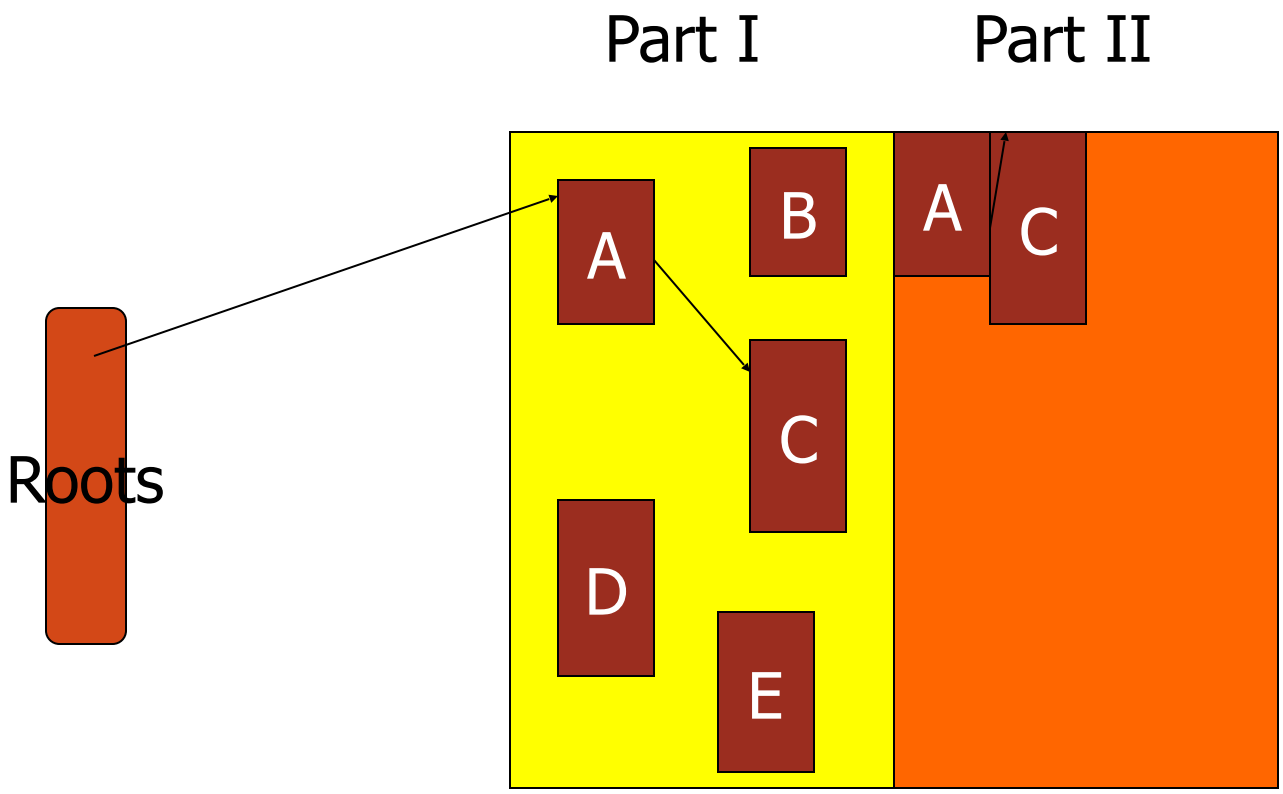


- Heap partitioned into two.
- Part 1 takes all allocations.
- Part 2 is reserved.
- During GC, the collector traces all reachable objects and copies them to the reserved part.
- After copying, activity goes to part 2. Part 1 is reserved till next collection.

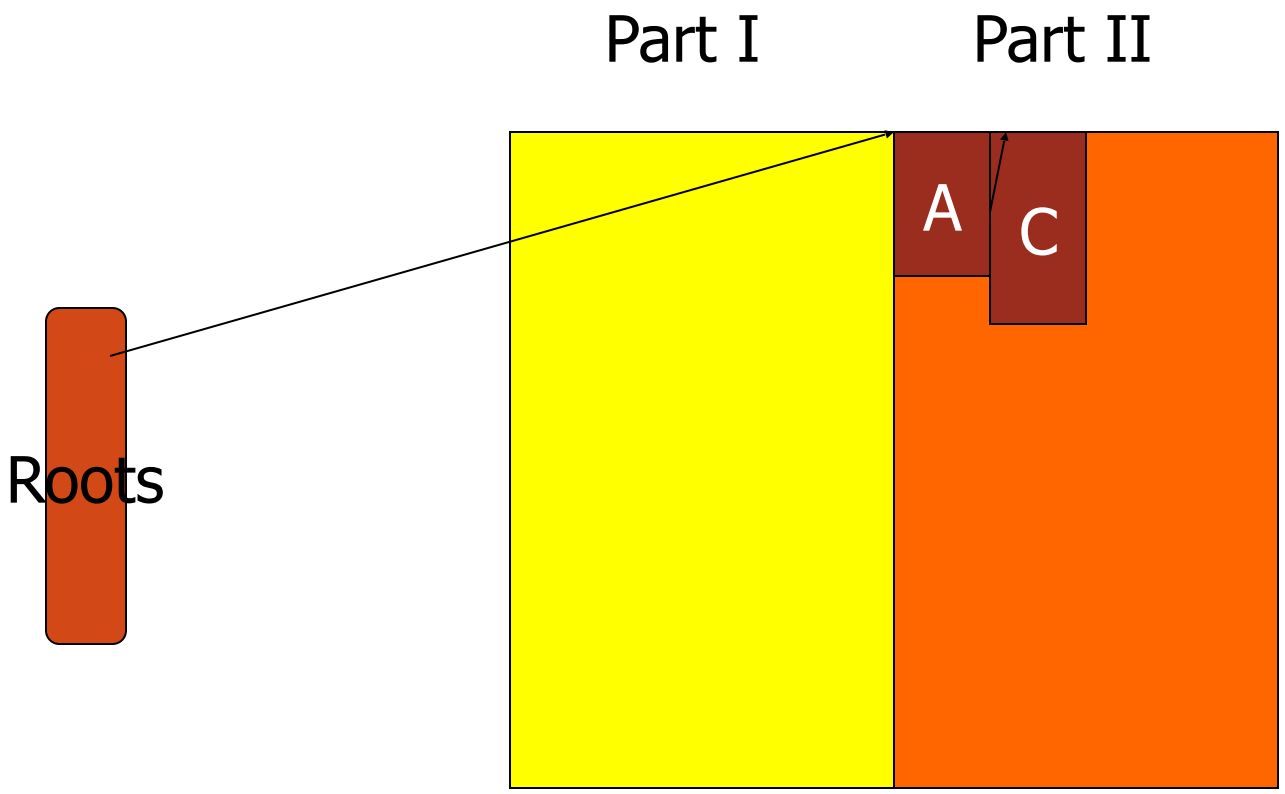
Copying garbage collection



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

	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Size of heap (live objects)	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

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

Standard Terms We Use

- Benchmarks: a set of programs for testing an implementation.
  - They should represent the behavior of “typical” programs

- Cache Miss: an access to the memory at an address that is not available in the cache.
- Cache hit: access to data that is in the cache.

Course Topics

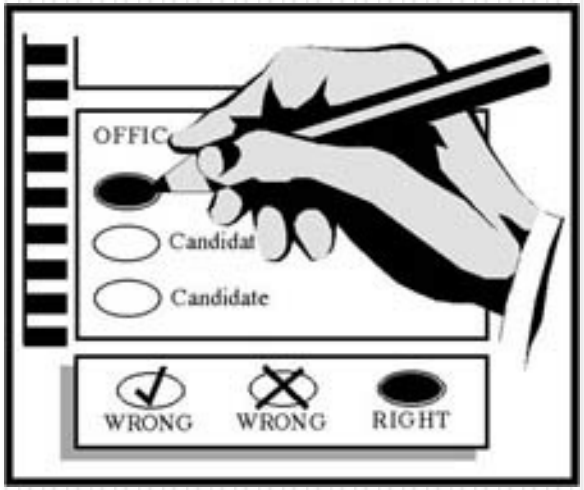
Course Topics

- 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.

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

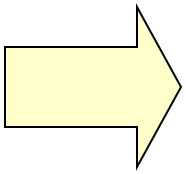
- Problem: if object graph is “unsuitable”, then recursion becomes too deep.
  - Large space overhead (compiler stack frames)
  - Inefficient execution (function calls)
  - Potential crash (user does not understand why)
- Solution: use iterative loops and auxiliary data structure instead of functions calls.

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

Modifying mark-roots

Mark_stack has all discovered objects, whose descendants have not yet been checked.

```
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

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

```
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 mark stack overflow:
  - Using cyclic stack (Knuth) or dropping overflowed objects (Bohem et al)
  - After dropping objects, recover 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.