Garbage Collection

Concepts, Algorithms and Java Case-Study

Luís Lopes

Faculdade de Ciências
Universidade do Porto

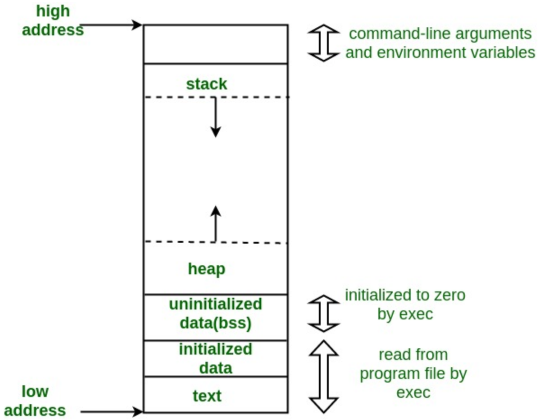
lmlopes@fc.up.pt

Overview

1. programming without garbage collection
2. basic concepts of garbage collection
3. garbage collection algorithms
4. garbage collection in Java

Programming without Garbage Collection

- C program memory layout:



Programming without Garbage Collection

- static space: `.text`, `.data` and `.bss`
- dynamic space: `stack` and `heap`
- stack keeps track of function calls
- data stored in the stack is ephemeral
- heap used to store longer-lived data
- variables in the stack keep pointers to heap data

Programming without Garbage Collection

- heap management is done by the programmer:
- `void* malloc(size_t size)`
- `void* calloc(size_t count, size_t size)`
- `void* realloc(void *ptr, size_t size)`
- `void  free(void *ptr)`
- functions available from `libc`;
- C keeps information on allocated blocks
- is the block used? what is the size of the block?
- freed blocks are reused in further `malloc()` calls
- blocks have different sizes, hence fragmentation of heap

Programming without Garbage Collection

- problem: heap size is limited
- hence, from application or libraries
- at some point `malloc()` may fail
- heap size can be extended
- usually this resorts to calls that change size of data segment:
- `void* brk(const void* addr)` (mostly deprecated)
- `void* mmap(void* addr, ...)` (current implementations)
- ultimately, the operating system limits heap growth

Some Languages *without* Garbage Collection

- C
- C++
- Objective-C
- Rust

Some Languages *with* Garbage Collection

- Java
- Python
- Haskell
- Go

Basic Concepts

- concept attributed to John McCarthy (LISP, 1959)
- main idea: the heap is automatically managed
- no interference from programmer
- automatic identification of items no longer in use
- reuse the space freed by these items
- eventually optimize layout (e.g., compaction)

Basic Concepts

- but why use garbage collection?
- manual heap management is prone to error:
- e.g., memory leaks
- e.g., memory corruption
- it is especially hard for inexperienced programmers
- therefore:
- most modern programming languages use garbage collection

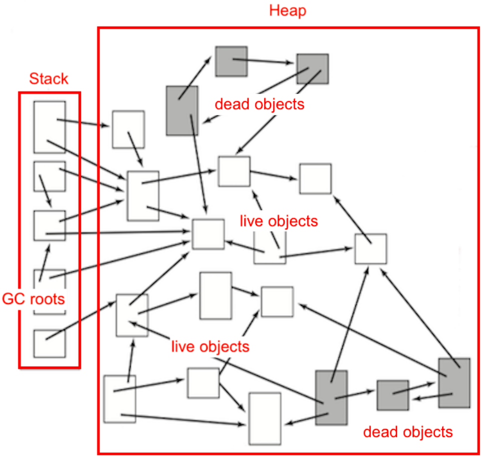
Basic Concepts > Heap

- Heap (H): the region of memory where objects reside
- $|H|$: size of Heap in bytes
- $\#H$: number of objects in Heap

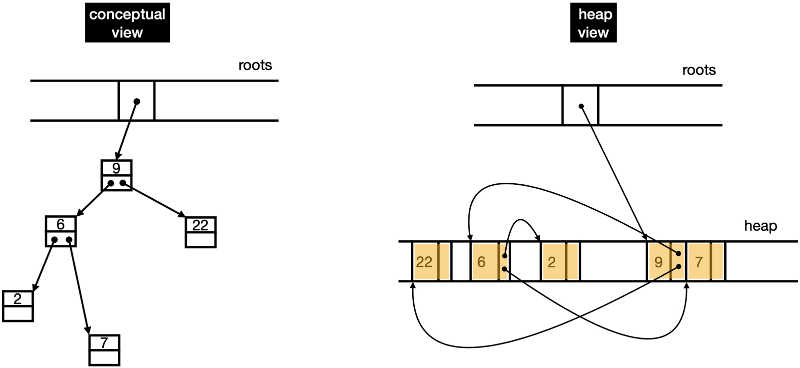
Basic Concepts > Roots & Live Set

- GC Root: any object referenced from outside the Heap
- e.g., a variable in the Stack
- Live Set (LS): all objects in Heap reachable from GC roots
- objects may be reachable via long chains of references
- $|LS|$: size of Live Set in bytes
- $\#LS$: number of objects in Live Set

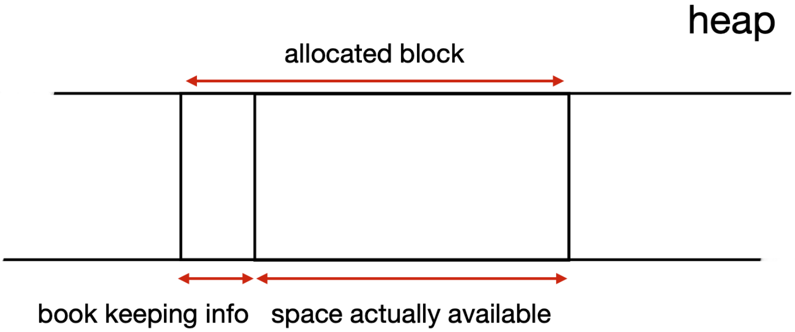
Basic Concepts > Roots & Live Set



Basic Concepts > Roots & Live Set



Basic Concepts > Roots & Live Set



Basic Concepts > Mutator

- Mutators: application threads that change the Heap
- mutation rate: frequency of Heap changes by application
- e.g., by changing references, chains, eventually deleting objects
- allocation rate: frequency of object allocation by application

Basic Concepts > Collector

- Collector: thread that collects dead objects in Heap
- there can be multiple threads
- typically part of the run-time system
- types:
- “pauseless” (runs concurrently with application)
- “stop the world” (stops the application)
- “serial” (one thread)
- “parallel” (multiple threads)

Basic Concepts > GC Safepoint

- GC Safepoint: an execution state for which it is safe to GC
- consider an application thread
- if thread at a GC Safepoint then GC *can* be performed
- thread cannot leave GC Safepoint until after GC ends
- GC Safepoints should be frequent during application execution
- (otherwise, few opportunities for GC)

Basic Concepts > GC Trigger

- usually, GC is triggered when the heap is full
- e.g., an allocation returns `null`
- or, when a given threshold is exceeded
- e.g., only 5% of space in the heap is still available

Some Algorithms

- Reference Counting
- Mark & Sweep
- Mark & Compact
- Copy Collection
- Generational Garbage Collection

Algorithms > Reference Counting

- for every object in Heap there is a counter
- new reference to the object, counter incremented
- e.g., local var assignment
- reference to the object removed, counter decremented
- e.g., local var on returning function
- when counter = 0, space allocated to object is reclaimed
- reclaimed space is reused for new objects
- $+$ lightweight, real-time GC
- $-$ cyclic references are a problem

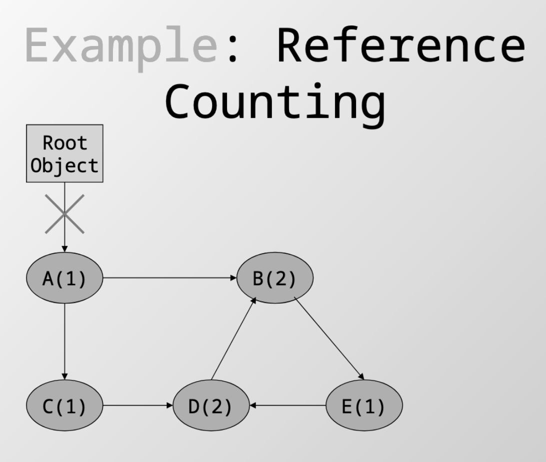
Algorithms > Reference Counting

- reference counting preserves the following invariant:
- an object A is alive iff rc(A) > 0

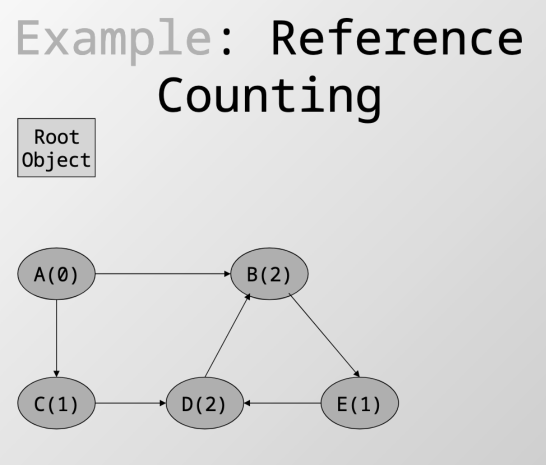
Algorithms > Reference Counting

- object A has rc(A)
- if rc(A) = 0, A is collected
- reference counting also applied to object B referenced from A
- if rc(B) = 0, B is collected
- and so on...
- collecting an object may induce a cascade of collections
- cyclic references are a problem however
- objects that should be collected will never be so

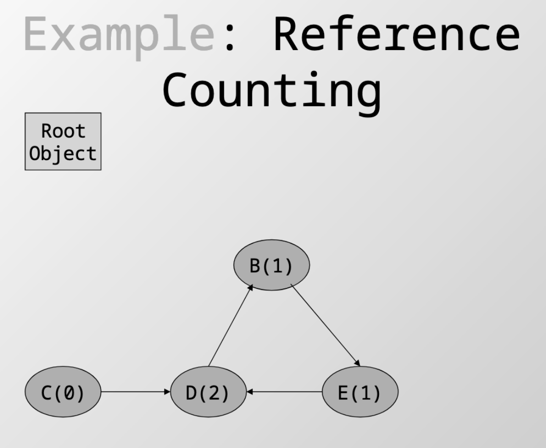
Algorithms > Reference Counting



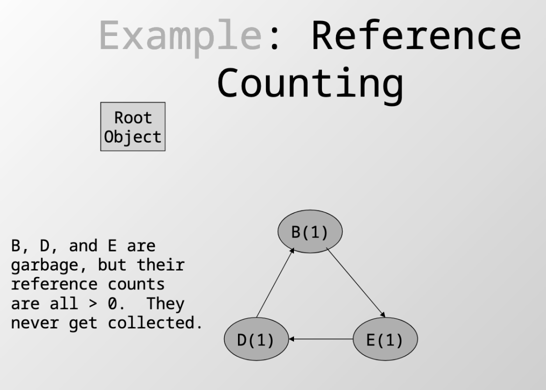
Algorithms > Reference Counting



Algorithms > Reference Counting



Algorithms > Reference Counting



Algorithms > Reference Counting

```
new():
  ref <- allocate()
  if ref = null
    error "Out␣of␣Memory"
  rc(ref) <- 0
  return ref
```

Algorithms > Reference Counting

```
atomic write(src, ref):
  addReference(ref)
  deleteReference(src)
  src <- ref
```

Algorithms > Reference Counting

```
addReference(ref):
  if ref <> null
    rc(ref) <- rc(ref) + 1
```

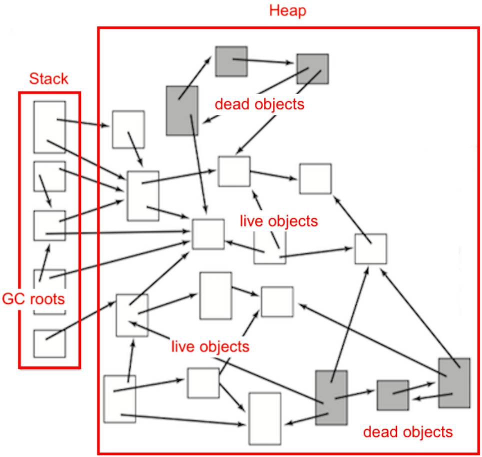
Algorithms > Reference Counting

```
deleteReference(ref):
  if ref <> null
    rc(ref) <- rc(ref) - 1
    if rc(ref) = 0
      for field in Pointers(ref)
        deleteReference(*field)
      free(ref)
```

Algorithms > Mark & Sweep

- Mark phase:
- every object has live bit set to 0
- start from GC roots
- traverse graph, set bit to 1 for every object visited
- Sweep phase:
- linear traversal of heap, bit 0 objects added to free list
- Mark: $\mathcal{O}(\#LS)$
- Sweep: $\mathcal{O}(|H|)$
- $+$ simple, references do not change
- $-$ external fragmentation

Algorithms > Mark & Sweep



Algorithms > Mark & Sweep

Algorithms > Mark & Sweep

```
new():
  ref <- allocate()
  if ref = null
    collect()
    ref <- allocate()
    if ref = null
      error "Out of Memory"
  return ref
```

Algorithms > Mark & Sweep

```
atomic collect():
  markFromRoots();
  sweep(heapStart, heapEnd);
```

Algorithms > Mark & Sweep

```
initialise():
  workList <- empty
```

Algorithms > Mark & Sweep

```
markFromRoots():
  initialise(workList)
  for field in Roots
    ref <- *field
    if ref <> null && not isMarked(ref)
      setMarked(ref)
      add(workList, ref)
      mark()
```

Algorithms > Mark & Sweep

```
mark():
  while not isEmpty(workList)
    ref <- remove(workList)
    for field in Pointers(ref)
      child <- *field
      if child <> null && not isMarked(child)
        setMarked(child)
        add(workList, child)
```

Algorithms > Mark & Sweep

```
sweep(start, end):
  scan <- start
  while scan < end
    if isMarked(scan)
      unsetMarked(scan)
    else
      free(scan)
    scan <- nextObject(scan)
```

Algorithms > Mark & Compact

- Mark phase:
- same as above
- $\mathcal{O}(\#LS)$
- Compact phase:
- move objects (in same Heap space)
- fix references
- $\mathcal{O}(|H| + |LS|)$
- $+$ no free list
- $+$ no external fragmentation
- $-$ delicate compact operation, references change

Algorithms > Mark & Compact

Algorithms > Mark & Compact

```
new(): /* same as M&S */
  ref <- allocate()
  if ref = null
    collect()
    ref <- allocate()
    if ref = null
      error "Out␣of␣Memory"
  return ref
```

Algorithms > Mark & Compact

```
atomic collect():
  markFromRoots() /* same as M&S */
  compact(heapStart, heapEnd)
```

Algorithms > Mark & Compact

```
/* e.g., Lisp 2 algorithm */
compact(heapStart, heapEnd):
  computeLocations(heapStart, heapEnd, heapStart)
  updateReferences(heapStart, heapEnd)
  relocate(heapStart, heapEnd)
```

Algorithms > Mark & Compact

```
computeLocations(start, end, toRegion)
  scan <- start
  free <- toRegion
  while scan < end
    if isMarked(scan)
      forwardingAddress(scan) <- free
      free <- free + size(scan)
    scan <- scan + size(scan)
```

Algorithms > Mark & Compact

```
updateReferences(start, end):
  /* first update only roots */
  for field in Roots
    ref <- *field
    if ref <> null
      *field <- forwardingAddress(ref)
  /* now update internal references */
  scan <- start
  while scan < end
    if isMarked(scan)
    for field in Pointers(scan)
      if *field = null
        *field <- forwardingAddress(*field)
      scan <- scan + size(scan)
```

Algorithms > Mark & Compact

```
relocate(start, end):
  scan <- start
  while scan < end
    if isMarked(scan)
      dest <- forwardingAddress(scan)
      move(scan, dest)
      unsetMasked(dest)
    scan <- scan + size(scan)
```

Algorithms > Copy Collector

- Heap divided into “fromSpace” and “toSpace”
- Mark phase:
- same as above, with objects in ”fromSpace”
- Copy phase:
- all live objects (bit = 1) copied to “toSpace”
- “toSpace” becomes “fromSpace” and vice versa
- “stop the world”, copy operation expensive
- naturally compact, no fragmentation
- twice the amount of heap required
- Mark: $\mathcal{O}(\#LS)$, Copy: $\mathcal{O}(|LS|) = \mathcal{O}(\#LS)$

Algorithms > Copy Collection

```
createSemiSpace()
  fromSpace <- heapStart
  extent <- (heapEnd - heapStart) / 2
  toSpace <- heapStart + extent
  top <- toSpace
  free <- fromSpace
```

Algorithms > Copy Collection

```
flip():
  fromSpace, toSpace <- toSpace, fromSpace
  top <- fromSpace + extent
  free <- fromSpace
```

Algorithms > Copy & Collection

```
new(): /* same as M&S */
  ref <- allocate()
  if ref = null
    collect()
    ref <- allocate()
    if ref = null
      error "Out␣of␣Memory"
  return ref
```

Algorithms > Copy Collection

```
atomic allocate(size):
  result <- free
  newfree <- result + size
  if newfree > top
    return null
  free <- newfree
  return result
```

Algorithms > Copy Collection

```
atomic collect():
  flip()
  initialise(workList)
  /* first copy the roots */
  for field in Roots
    process(field)
  /* then the internal references */
  while not isEmpy(workList)
    ref <- remove(workList)
    scan(ref)
```

Algorithms > Copy Collection

```
process(field):
  fromRef <- *field
  if fromRef <> null
    *field <- forward(fromRef)
```

```
scan(ref):
  for field in Pointers(ref)
    process(field)
```

Algorithms > Copy Collection

```
forward(fromRef):
  toRef <- forwardingAddress(fromRef)
  if toRef = null
    toRef = copy(fromRef)
  return toRef
```

Algorithms > Copy Collection

```
copy(fromRef):
  toRef <- free
  free <- free + size(fromRef)
  move(fromRef,toRef)
  forwardingAddress(fromRef) <- toRef
  add(worklist, toRef)
  return toRef
```

Algorithms > Generational Garbage Collection

- “Weak Generational Hypothesis”:
- most objects survive for short periods
- few references from older to new objects

Algorithms > Generational Garbage Collection

- idea:
- organize heap into regions of increasingly longer-lived objects
- the regions are called “generations”
- youngest generation is called “Eden”
- contains objects just created
- changes faster than all others

Algorithms > Generational Garbage Collection

- long-lived objects are moved to another “generation”
- a GGC is efficient because it spends most of its time at “Eden”
- memory recovered here is usually enough to keep app going
- ignore the other generations most of the time
- can use different GC algorithms for each generation

Performance

- which garbage collector to use?
- criteria:
- impact on application throughput
- memory footprint
- latency

Performance

- throughput:
- assume GC running α ms out of runtime β ms
- throughput = $\frac{(\beta-\alpha)}{\beta}$
- e.g., GC active 50ms every second ($\alpha = 50$ms, 1s$=1000$ms)
- e.g., throughput = $(1000 - 50)/1000 = 95\%$

Performance

- memory overhead:
- do you need extra memory to perform GC?
- e.g., Copy Collection
- latency:
- "real-time" (e.g., reference count)
- "stop world" (e.g., Mark&Sweep, Copy Collector)
- latency: Web app $\sim$ 1s, GUI $\sim$ 100ms, real-time $\sim$ 1μs

Garbage Collection in Java

- Java uses Generational Garbage Collection
- the memory used by JVM is divided as follows:
- non-Heap: loaded classes, “permanent generation” objects
- Heap: two regions for “young” and “tenured” objects
- “young” region divided into: “Eden”, S0 and S1
- objects start in “Eden”

Garbage Collection in Java

- when “Eden” is full, Java performs “minor GC”
- during “minor GC” objects copied between “Eden”/S0/S1
- (see detailed operations on “Eden”/S0/S1 during “minor GC”)
- objects that survive several “minor GC” promoted to “tenured”
- also promoted when “S0” and “S1” are filled

Garbage Collection in Java

- we can use different algorithms to GC these regions
- “minor GC” young objects (more frequent, needs to be faster)
- e.g., minor GC - Mark&Sweep
- “major GC” tenured objects (less frequent, ”stop the world”)
- e.g., major GC - Copy Collection

Garbage Collection in Java > Serial Collector

- serial collector:
- single-threaded
- “stop the world” / monolithic
- efficient (no need for synchronization with other threads)
- Java option `-XX:+UseSerialGC`

Garbage Collection in Java > Serial Collector

- serial collector (cont.):
- “young” collected when “Eden” full
- or when `System.gc()` is called
- “tenured” uses Mark & Sweep Compact
- triggered when no space in “tenured”
- or when `System.gc()` is called
- well suited for single-core processors
- good overall performance for 128MB $\leq |H| \leq$ 256MB

Garbage Collection in Java > Parallel Collector

- parallel collector:
- multithreaded
- "stop the world" / monolithic
- Java options: `-XX:+UseParallelGC`
- multiple parameters, e.g.
- number of threads: `-XX:ParallelGCThreads=<N>`

Garbage Collection in Java > Parallel Collector

- parallel collector (cont.):
- triggers are same as serial
- “Eden” is full triggers “minor GC”
- “tenured” is full triggers “major GC”
- or both if `System.gc()` is called
- best option for multicore processors
- best overall throughput
- works well for $|H| \leq 1$GB

Garbage Collection in Java $>$ CMS

- parallel collector variant:
- Concurrent Mark & Sweep (CMS)
- runs concurrently with application / not monolithic
- does not compact heap
- Java option: `-XX:+UseConcMarkSweepGC`
- works well for $|H| > 1\text{GB}$

Garbage Collection in Java $>$ CMS

- better when application responsiveness is important
- known issues:
- lower application throughput
- requires $\sim$ 20-30% extra memory

Garbage Collection in Java $>$ G1

- Garbage First Algorithm (G1):
- targeted for server-caliber multiprocessor machines
- scales for $|H| \sim$ tens of GBs or larger, more than 50% live
- some parts run concurrently with application
- tradeoff between more, but shorter, collection pauses
- application throughput also tends to be slightly lower
- G1 is the default collector for current JVM version (Java 18)

Garbage Collection in Java > G1

- partitions Heap into set of equally sized contiguous regions
- a region is the unit of memory (de)allocation
- regions are empty or assigned to “young” or “tenured”
- in general, “young” and “tenured” are not contiguous
- just before a collection, G1 knows which regions are mostly empty (low live %)
- it reclaims this memory first
- hence the name “G1” - “garbage first”
- memory reclaimed may be enough to resume app

References

Richard Jones, Antony Hosking, Eliot Moss
The Garbage Collection Handbook
CRC Press 2012

Arjun Sreedharan
Memory Allocators 101 - Write a Simple Memory Allocator
Available here

Aashish Patil
Stages and Levels of Java Garbage Collection
Available here

Haim Yadid
Let's talk about Garbage Collection
Available here

Oracle Documentation
HotSpot Virtual Machine Garbage Collection Tuning Guide
Available here