Lecture 12: Memory management
25 Feb 2010

Nate Nystrom
University of Texas at Arlington
Some content from

- Tim Teitelbaum, Cornell CS 412 slides, 2008
Memory management

Global memory
- allocated at a fixed location in the process
- static fields in Java, singleton objects in Scala, globals and statics in C, module-scoped variables in ML, M3

Stack
- local variables, formal parameters, this
- local arrays

Registers
- local variables, ...

Heap
- dynamically allocated data structures
Explicit memory management

Automatic memory management
- reference counting
- Deutsch-Bobrow deferred reference counting
- mark and sweep
- copying GC
- concurrent/incremental GC
- generational GC
- BDW collector
Explicit memory management

Unix interface:
- `void *malloc(size_t n)`
  - allocate `n` bytes of storage on the heap and return its address

- `void free(void *addr)`
  - release storage allocated by `malloc` at address `addr`

User-level library manages heap, issues `brk` calls when necessary to grow the heap

C++: `new/delete` usually just call `malloc/free`
Malloc implementation

Blocks of unused memory stored in a *freelist*

- `malloc` finds unused block on freelist
- `free` puts block onto head of freelist

Simple, but:

- *External fragmentation* = small free blocks scattered in the heap
  - Cannot alloc a large block even if sum of all free blocks is enough
- `malloc` can be $O(\mid \text{heap} \mid)$
Buddy system

Maintain freelists for different allocation sizes, all powers of 2

\texttt{malloc}(n)

\begin{itemize}
  \item round $n$ up to nearest power of 2
  \item if no block of size $n$ — i.e., freelist($n$) is empty
    \begin{itemize}
      \item get block of size $2n$
      \item split: return block of size $n$, add other block (its buddy) to freelist($n$)
    \end{itemize}
\end{itemize}

\texttt{free}

\begin{itemize}
  \item coalesce free buddies of size $n$ into a single free block of size $2n$
\end{itemize}

\textit{Internal fragmentation}: alloc larger blocks because of rounding

Trade external for internal fragmentation

\texttt{malloc, free} are $O(1)$
Aside

How to check if an integer n is a power of 2

- \( n \ & (n-1) \neq 0 \)

How to round an integer n up to the next power of 2

- \( p = 1; \ while (p < n) p <<= 1; \)

Without branching:

- \( n -= 1 \)
  - \( n |= (n >>> 1) \)
  - \( n |= (n >>> 2) \)
  - \( n |= (n >>> 4) \)
  - \( n |= (n >>> 8) \)
  - \( n |= (n >>> 16) \)
  - return n + 1
Node *x = new Node("happy")
Node *p = x;
delete x;    // But I’m not dead yet!
Node *y = new Node("sad");
cout << p->data << endl;    // sad
Problems

Might forget to call `free`

Might `free` a pointer not returned by `malloc`, corrupting the freelist

Might call `free` twice on same address, corrupting the freelist
Problems

Makes modular programming more difficult

Every interface needs to agree on a contract
- Have to know what code “owns” a given object so that objects are deleted exactly once
Garbage collection

Better: automatically collect unused memory

Gives the programmer the illusion that they have infinite memory
Garbage collection

Garbage collector is usually the most complex part of the run-time environment

Want to delete objects if they won’t be used again

- This is undecidable!
- So must be conservative
- Use *reachability* as an approximation of liveness:
  - if there is no way to reach the object from globals, stack, registers, then object cannot be used again

- might still retain objects that won’t be used again
- but will not free objects that will be used again
Stack, registers, globals are *roots* of the object graph.
Anything reachable from the roots is *live*, all else is *garbage*. 
Stack, registers, globals are *roots* of the object graph

Anything reachable from the roots is *live*, all else is *garbage*
GC implementation
Reference counting

Idea:
- associate a *reference count* with each object
- number of references (pointers) to the object

Keep track of reference counts
- For assignment $x = e$
  - decrement ref count for object referenced by $x$ (if any)
  - increment ref count for object referenced by $e$
  - do the assignment

When reference count hits 0, object is unreachable => free it
Problem
Problem

Remove a reference from the stack
Decrement rc
Problem

Deallocate object
Decrement rc for all its out-references
Problem: cycles

Reference counting does not collect cycles!
Problem: performance

Consider assignment: \(x.f = y\)

Without ref counts:
\[tx + \text{off}] = ty\]

With ref counts:
\[t1 = [tx + \text{off}]\]
\[c = [t1 + \text{rc}]\]
\[c = c - 1\]
\[t1 + \text{rc}] = c\]
if \(c == 0\) reclaim_object(t1)
\[c = [ty + \text{rc}]\]
\[c = c + 1\]
\[ty + \text{rc}] = c\]
\[tx + \text{off}] = ty\]

Large run-time overhead!
- Worse: reclaim_object might trigger traversal of large graph to decrement ref counts in the heap
Don’t count reference from the stack

When rc goes to 0, insert into the *zero count set* (ZCS)

When ZCS is full

- scan stack, incrementing counts of all objects referred to
- scan ZCS, reclaim any objects with zero count
- set ZCS to empty
- scan stack, decrementing counts of all objects referred to
  - if rc goes to 0, insert into new ZCS

Still can’t collection cycles
Mark-sweep GC

The classic algorithm

Two phases:

- Mark phase
  - start from roots, trace object graph, marking every object reached
- Sweep phase
  - iterate through all objects in the heap
  - reclaim unmarked objects
  - clear marks

- optional: compact live objects in heap (called mark-compact)
Object layout

Can use bit vector to record marks

Or store a mark bit in the object header.

Add another word to the header:
- [mark bit]
- [dispatch table]
- [fields...]

Or: just use a bit of the dispatch table pointer
- pointers aligned 4 have 2 free bits
- need to mask off lsb on method dispatch
Mark phase

Implemented as depth-first search of object graph

Natural recursive implementation

```python
for each ref p in rootSet:
    mark(p)

mark(p) {
    if (*p marked) return
    mark *p
    for each reference-type field x in *p:
        mark(p->x)
}
```
Question: what happens when we try to mark a long linked list recursively?
Mark phase

stack = new Stack()
for each ref p in rootSet:
    stack.push(p)

while (! stack.empty) {
    p = stack.pop
    if (*p marked) continue
    mark *p
    for each reference-type field x in *p:
        stack.push(p->x)
}
Mark phase

Question: what happens when we try to mark a long linked list while maintaining a stack?
Deutsch-Waite-Schorr algorithm

Idea:
- during DFS, each pointer is followed only once
- reverse the pointers after following them – no stack needed!

Implication:
- objects are broken while being traversed
- all computation over objects must be halted during mark phase
- no concurrency allowed
Cost of mark-sweep

Accesses all memory in use by the program
Mark phase reads only live (reachable) data
Sweep phase reads all of the data (live + garbage)

=> run time proportional to total amount of data!
=> can cause long program pauses!
Conservative mark-sweep

Allocated storage contains both pointers and non-pointers
- Integers may look like pointers

Treating a pointer as a non-pointer
- objects may be garbage collected even though still reachable (unsafe)

Treating a non-pointer as a pointer
- objects are not garbage collected, even if not reachable (safe, but not precise)

Conservative collection:
- assumes values are pointers unless they can’t be (e.g., not in the range of the heap)
- requires no language support ==> works for C!
Boehm-Demers-Weiser collector

AKA Boehm-Weiser, BDW


Conservative mark-sweep GC for C, C++

Drop-in replacement for malloc
- `malloc = GC_malloc`
- `free = no-op`
- On Linux: `LD_PRELOAD=/usr/local/lib/libgc.so ./a.out`

Can also be used as a leak detector
Safepoints

Stack + registers + globals are roots for object graph traversal

In precise GC, collector needs to know which stack locations contain pointers

Idea:
- for each program point, compute a static map indicating which stack locations in the current frame and which registers contain pointers

Better idea:
- do this for only some program points (safepoints)
- only GC when program is at these points
- Placement: at the head of each loop, entry to each function
Copying collection

Idea: use two heaps
- one in use by the program
- one sits idle until GC needs it

GC:
- copy all live objects from active heap ("from space") to the inactive heap ("to space")
- dead objects are left in the from space
- heaps then switch roles

Issue: must rewrite references between objects
Initialize to-space as empty queue
Copy all root objects into queue
**Copying collection (Cheney)**

**Copy** all root objects into queue

**Copy** operation leaves forwarding pointer in from-space to copy in to-space
**Copy operation leaves forwarding pointer in from-space to copy in to-space**
**Copying collection (Cheney)**

**Scan**: Dequeue each object \( x \) in to-space and **copy** objects pointed to by \( x \) that are still in the from-space.

Update pointer in \( x \) to to-space copies.

**Copy** operation leaves forwarding pointer in from-space to copy in to-space.
Copying collection (Cheney)

Continue to dequeue and copy until end of queue.
Once head = tail, all uncopied objects in from-space are garbage

Root pointers (registers, stack) are swung to point to to-space, making it active
Benefits

Simple, no stack space needed
Run time proportional to number of live objects
Automatically compacts, eliminating fragmentation
Bump pointer allocation
  - malloc(n) implemented as tail = tail + n
Implementation issues

Precise pointer information required
  - can’t use on languages like C

Uses twice as much memory
  - but: its *virtual* memory
  - still: might be inappropriate to use copying GC in embedded systems
Bump pointer allocation

With copying GC, allocation is just a pointer increment

Java: if using fresh pages from the OS, pages will be zeroed, so no need to explicitly zero fields of new objects

With multiple threads: maintain one allocation pointer per thread

Contrast with malloc
Incremental GC

GC might have to “stop the world”
- GC pauses unacceptable for interactive applications or real time applications

Idea: run collector and program at the same time
- Need to trace object graph while program is mutating the graph

Program only holds pointers to to-space.

On field fetch
- if pointer to uncopied object in from-space, copy object and update pointer to to-space
- if pointer to copied object, follow the forwarding pointer to get at copied object in to-space
Generational GC

Observation:
- if an object has been reachable for a long time, it is likely to remain so
  - globals, objects referenced from main function
- most objects die young

```java
String toString() {
    String s = "";
    for (x : this)
        s += x;
    return s;
}
```

In a long-running system, mark-sweep, copying collection wastes time by scanning/copying older objects
Generational GC

Assign heap objects to different generations G0, G1, ...

G0 contains newest objects, most likely to become garbage
Generations

Consider a two-generation system

- G0 = new objects, “the nursery”
- G1 = tenured objects

G0 is scanned for garbage more often than G1
G0 often much smaller than G1
New objects eventually get tenured
Generational copying GC

Allocate objects in nursery
Divide tenured generation into from and to-spaces
When collecting nursery, copy into tenured generation
Roots of G0 include all objects in G1 + stack + registers

- Tenured objects might point to new objects
- How to avoid scanning them all?

In practice, few tenured objects point to new objects

- unusual for an object to point to a newer object
- can only happen if older object is modified long after creation to point to a new object

Keeping track of pointers from old generation to new

- remembered sets
- card marking
Remembered sets

Want to identify:
- \( x.f = y \)
- \( x \) is tenured, \( y \) is not

**Write barrier**
- Compiler inserts code when storing into a **pointer** field of an old object, record the pointer value

Store \( G_1 \rightarrow G_0 \) pointers in a **remembered set**

Roots of \( G_0 \) are stack + registers + remembered set of \( G_1 \)
Card marking

divide memory into cards of $2^k$ words (say $k = 5...7$)
maintain a bit vector with a bit per card
when storing into a location, mark the card for that location
  ▪ (so marking cards in the old generation)
At GC time, use marked cards in the old gen as roots

Advantage over remembered sets:
  ▪ faster write barrier (an extra shift and store)
Disadvantage:
  ▪ less precise
    ▪ all pointers on a marked card treated as roots for G0
Conventional wisdom

GC is worse than malloc because...
- extra processing
- poor cache performance
- bad page locality
- increased footprint (delayed reclamation)
Conventional wisdom

GC improves performance by...

- faster allocation  
  (fast path inlining & bump pointer allocation)
- better cache performance  
  (object reordering)
- improved page locality  
  (heap compaction)
Best collector performs as well as or better than malloc
  - up to 10% faster on some benchmarks

...but uses more memory
  - at least twice
  - sometimes 5x

GC good if:
  - system has a lot of RAM

GC bad if:
  - limited RAM
  - competition for physical memory
  - RAM relied upon for performance
    - in-memory databases, search engines, ...
Object pooling
- usually a bad idea

Marking values null to free early
- good idea (if careful)
Summary

GC simplifies interfaces, reduces memory errors

Performance often as good as malloc free
Questions?