Garbage Collection

Vitaly Shmatikov
Major Areas of Memory

- **Static area**
  - Fixed size, fixed content, allocated at compile time

- **Run-time stack**
  - Variable size, variable content (activation records)
  - Used for managing function calls and returns

- **Heap**
  - Fixed size, variable content
  - Dynamically allocated objects and data structures
    - Examples: ML reference cells, `malloc` in C, `new` in Java
Cells and Liveness

- **Cell** = data item in the heap
  - Cells are “pointed to” by pointers held in registers, stack, global/static memory, or in other heap cells
- **Roots**: registers, stack locations, global/static variables
- A cell is **live** if its address is held in a root or held by another live cell in the heap
Garbage

- Garbage is a block of heap memory that cannot be accessed by the program
  - An allocated block of heap memory does not have a reference to it (cell is no longer “live”)
  - Another kind of memory error: a reference exists to a block of memory that is no longer allocated
- Garbage collection (GC) - automatic management of dynamically allocated storage
  - Reclaim unused heap blocks for later use by program
Example of Garbage

class node {
    int value;
    node next;
}
node p, q;

p = new node();
q = new node();
q = p;
delete p;
Why Garbage Collection?

◆ Today’s programs consume storage freely
  • 1GB laptops, 1-4GB desktops, 8-512GB servers
  • 64-bit address spaces (SPARC, Itanium, Opteron)
◆ … and mismanage it
  • Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
  • Poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging
◆ Explicit memory management breaks high-level programming abstraction
GC and Programming Languages

◆ GC is not a language feature
◆ GC is a pragmatic concern for automatic and efficient heap management

- Cooperative langs: Lisp, Scheme, Prolog, Smalltalk …
- Uncooperative languages: C and C++
  - But garbage collection libraries have been built for C/C++

◆ Recent GC revival

- Object-oriented languages: Modula-3, Java
  - In Java, runs as a low-priority thread; System.gc may be called by the program

- Functional languages: ML and Haskell
The Perfect Garbage Collector

- No visible impact on program execution
- Works with any program and its data structures
  - For example, handles cyclic data structures
- Collects garbage (and only garbage) cells quickly
  - Incremental; can meet real-time constraints
- Has excellent spatial locality of reference
  - No excessive paging, no negative cache effects
- Manages the heap efficiently
  - Always satisfies an allocation request and does not fragment
Summary of GC Techniques

- **Reference counting**
  - Directly keeps track of live cells
  - GC takes place whenever heap block is allocated
  - Doesn’t detect all garbage

- **Tracing**
  - GC takes place and identifies live cells when a request for memory fails
  - Mark-sweep
  - Copy collection

- **Modern techniques: generational GC**
Reference Counting

- Simply count the number of references to a cell
- Requires space and time overhead to store the count and increment (decrement) each time a reference is added (removed)
  - Reference counts are maintained in real-time, so no “stop-and-gag” effect
  - Incremental garbage collection
- Unix file system uses a reference count for files
- C++ “smart pointer” (e.g., auto_ptr) use reference counts
Reference Counting: Example
Reference Counting: Strengths

- Incremental overhead
  - Cell management interleaved with program execution
  - Good for interactive or real-time computation

- Relatively easy to implement

- Can coexist with manual memory management

- Spatial locality of reference is good
  - Access pattern to virtual memory pages no worse than the program, so no excessive paging

- Can re-use freed cells immediately
  - If RC == 0, put back onto the free list
Reference Counting: Weaknesses

- **Space overhead**
  - 1 word for the count, 1 for an indirect pointer

- **Time overhead**
  - Updating a pointer to point to a new cell requires:
    - Check to ensure that it is not a self-reference
    - Decrement the count on the old cell, possibly deleting it
    - Update the pointer with the address of the new cell
    - Increment the count on the new cell

- One missed increment/decrement results in a dangling pointer / memory leak

- **Cyclic data structures** may cause leaks
Reference Counting: Cycles

Heap space

Memory leak
“Smart Pointer” in C++

Similar to std::auto_ptr<T> in ANSI C++

- Fits in a register
- Easily passed by value as an argument or result of a function
- Takes no more space than regular pointer, but much “safer” (why?)

\[ \text{sizeof(RefObj<T>)} = 8 \text{ bytes of overhead per reference-counted object} \]

\[ \text{sizeof(Ref<T>)} = 4 \text{ bytes} \]
template<class T> class RefObj {
    T* obj;
    int cnt;
public:
    RefObj(T* t) : obj(t), cnt(0) {} ~RefObj() { delete obj; }
    int inc() { return ++cnt; }
    int dec() { return --cnt; }
    operator T*() { return obj; }
    operator T&() { return *obj; }
    T& operator *() { return *obj; }
};

template<class T> class Ref {
    RefObj<T>* ref;
    Ref<T>* operator&() {}
public:
    Ref() : ref(0) {}
    Ref(T* p) : ref(new RefObj<T>(p)) { ref->inc(); }
    Ref(const Ref<T>& r) : ref(r.ref) { ref->inc(); }
    ~Ref() { if (ref->dec() == 0) delete ref; }
    Ref<T>& operator=(const Ref<T>& that) {
        if (this != &that) {
            if (ref->dec() == 0) delete ref;
            ref = that.ref;
            ref->inc(); }
        return *this; }
    T* operator->() { return *ref; }
    T& operator*() { return *ref; }
};
Using Smart Pointers

Ref<string> proc() {
    Ref<string> s = new string(“Hello, world”); // ref count set to 1
    …
    int x = s->length(); // s.operator->() returns string object ptr
    …
    return s;
} // ref count goes to 2 on copy out, then 1 when s is auto-destructed

int main()
{
    …
    Ref<string> a = proc(); // ref count is 1 again
    …
} // ref count goes to zero and string is destructed, along with Ref and RefObj objects
Mark-Sweep Garbage Collection

- Each cell has a mark bit
- Garbage remains unreachable and undetected until heap is used up; then GC goes to work, while program execution is suspended

Marking phase
  - Starting from the roots, set the mark bit on all live cells

Sweep phase
  - Return all unmarked cells to the free list
  - Reset the mark bit on all marked cells
Mark-Sweep Example (1)
Mark-Sweep Example (2)
Mark-Sweep Example (3)
Mark-Sweep Example (4)

Heap space

Free unmarked cells

Reset mark bit of marked cells

root
set
Mark-Sweep Costs and Benefits

- **Good:** handles cycles correctly
- **Good:** no space overhead
  - 1 bit used for marking cells may limit max values that can be stored in a cell (e.g., for integer cells)
- **Bad:** normal execution must be suspended
- **Bad:** may touch all virtual memory pages
  - May lead to excessive paging if the working-set size is small and the heap is not all in physical memory
- **Bad:** heap may fragment
  - Cache misses, page thrashing; more complex allocation
Copying Collector

- Divide the heap into “from-space” and “to-space”
- Cells in from-space are traced and live cells are copied (“scavenged”) into to-space
  - To keep data structures linked, must update pointers for roots and cells that point into from-space
    - This is why references in Java and other languages are not pointers, but indirect abstractions for pointers
  - Only garbage is left in from-space
- When to-space fills up, the roles flip
  - Old to-space becomes from-space, and vice versa
Copying a Linked List

[Cheney’s algorithm]

Cells in to-space are packed
Flipping Spaces

to-space

pointer
forwarding address

from-space
root

A' B' C' D'
Copying Collector Tradeoffs

◆ Good: very low cell allocation overhead
  - Out-of-space check requires just an addr comparison
  - Can efficiently allocate variable-sized cells

◆ Good: compacting
  - Eliminates fragmentation, good locality of reference

◆ Bad: twice the memory footprint
  - Probably Ok for 64-bit architectures (except for paging)
    - When copying, pages of both spaces need to be swapped in. For programs with large memory footprints, this could lead to lots of page faults for very little garbage collected
    - Large physical memory helps
Generational Garbage Collection

Observation: most cells that die, die young
- Nested scopes are entered and exited more frequently, so temporary objects in a nested scope are born and die close together in time
- Inner expressions in Scheme are younger than outer expressions, so they become garbage sooner

Divide the heap into generations, and GC the younger cells more frequently
- Don’t have to trace all cells during a GC cycle
- Periodically reap the “older generations”
- Amortize the cost across generations
Generational Observations

- Can measure “youth” by time or by growth rate
- Common Lisp: 50-90% of objects die before they are 10KB old
- Glasgow Haskell: 75-95% die within 10KB
  - No more than 5% survive beyond 1MB
- Standard ML of NJ reclaims over 98% of objects of any given generation during a collection
- C: one study showed that over 1/2 of the heap was garbage within 10KB and less than 10% lived for longer than 32KB
Example with Immediate “Aging” (1)
Example with Immediate “Aging” (2)
Generations with Semi-Spaces

- Root set
- Middle generation(s)
- Youngest
- Oldest