

Garbage Collection

Last time

- Compiling Object-Oriented Languages

Today

- Motivation behind garbage collection
- Garbage collection basics
- Garbage collection performance
- Specific example of using GC in C++

Acknowledgements

- These slides are based on Kathryn McKinley's slides on garbage collection as well as E Christopher Lewis's slides

Background

Static allocation: variables are bound to storage at compile-time

- pros: easy to implement
- cons: no recursion, data structure sizes are compile-time constants, data structures cannot be dynamic

Stack allocation: dyn. alloc. stack frame for each proc. invocation

- pros: recursion is possible, data structure sizes may depend on parameters
- cons: stack allocated data is not persistent, stack allocated data cannot outlive the procedure for which it is defined

Heap allocation: arbitrary alloc. and dealloc. of objects in *heap*

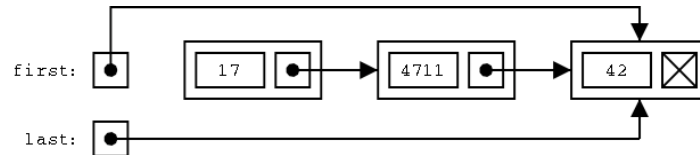
- pros: solves above problems: dynamic, persistent data structures
- cons: very difficult to explicitly manage heap

Memory Management

Ideal (not possible)

- deallocate all data that will not be used in the future

What is garbage?



Manual/Explicit

- programmer deallocates with free or delete

Automatic/Implicit

- garbage collection

Explicit versus Automatic

Explicit

- + efficiency can be very high
- + gives programmers “control”
- more code to maintain
- correctness is difficult
 - core dump if an object is freed too soon, dangling pointers
 - space is wasted if an object is freed too late
 - if never free, at best waste space, at worst fail

Automatic

- + reduces programmer burden
- + eliminates sources of errors
- + integral to modern OOP languages (ie. Java, C#)
- can not determine all objects that won't be used in the future
- may or may not hurt performance

Key Issues

For both

- Fast allocation
- Fast reclamation
- Low fragmentation (wasted space)

For Garbage Collection

- How to discriminate between live objects and garbage

Basic approaches to garbage collection

- reference counting
- reachability

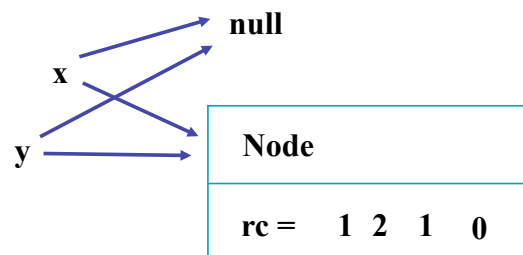
Reference Counting

Idea

- for each heap allocated object, maintain count of # of pointers to it
- when creating object x , $rc[x] = 0$
- when creating new reference to object x , $rc[x]++$
- when removing reference to object x , $rc[x]--$
- if ref count goes to zero, free object (i.e., place on free list)

Example

```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```



Complication

- what if freed object contains pointers?

Reference Counting Analysis

How it handles key issues

- **allocation** is expensive because searching a freelist
- **reclamation** is local and incremental
- **fragmentation** is high

Further analysis

- + relatively simple
- + very simple run-time system
- cannot reclaim cyclic data structures (shifts burden to programmer)
- high runtime overhead (must manipulate ref counts for every reference update)
- space cost
- complicates compilation

Trace Collecting

Observation

- rather than explicitly keep track of the number of references to each object we can traverse all reachable objects and discard unreachable objects

Details

- start with a set of **root** pointers (program vars), root set
- global pointers
- pointers in stack and registers
- traverse objects recursively from root set
- visit **reachable** objects
- unvisited objects are garbage
- we might visit an object even if it's dynamically dead (ie, we are only conservatively approximating dead object discovery)

When do we collect?

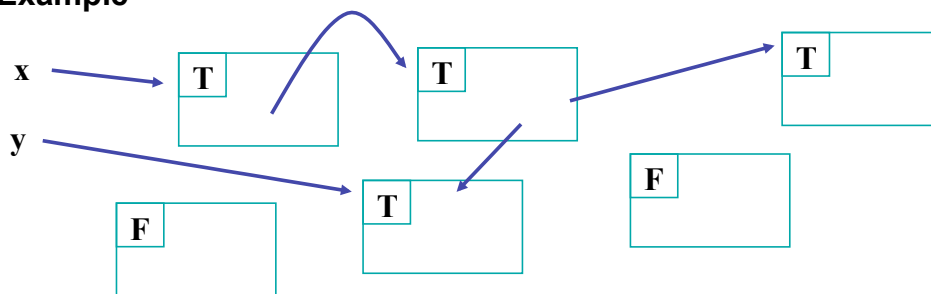
- when the heap is full

Mark-Sweep Collecting

Simple trace collector

- trace reachable objects marking reachable objects
- sweep through all of heap
- add unmarked objects to **free list**
- clear marks of marked objects

Example



Mark-Sweep Collecting Analysis

How it handles the key issues

- **allocation** is expensive because searching a freelist
- **reclamation** can result in the “embarrassing pause” problem
 - poor memory locality when tracing
- **fragmentation** is high

Further analysis

- + collects cyclic structures
- + simple
- must be able to dynamically identify pointers in vars and objects
- more complex runtime system
- space overhead is only one bit per data object

Mark-Compact Collecting or Copy Collecting

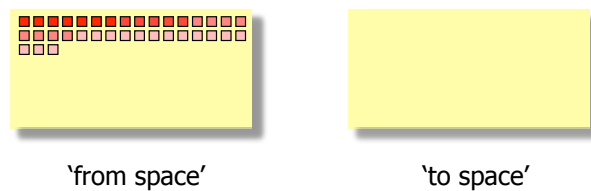
Idea

- move objects to “new” heap while tracing

Details

- divide heap in half (prog. allocs. in from-space, to-space is empty)
- when from-space is full...
 - copy non-garbage from from-space to to-space (to-space is compact) when visiting object during tracing
 - leave forwarding pointer in from-space version of object
 - if revisit this object, redirect pointer to to-space copy

Copying Garbage Collection



Mark-Compact Collecting Analysis

How it handles the key issues

- **allocation** is very fast since there is no free list to search
- **reclamation** can result in the “embarrassing pause” problem
 - poor memory locality when tracing
 - copying data from one heap to another
 - changing pointers because objects are being moved
 - + visits only reachable objects
- + no **fragmentation**

Further Analysis

- + collects cyclic structures
- requires twice the (virtual) memory
- breadth-first traversal means to-space objects could have poor locality

Hybrid Collectors

Idea

- different collection techniques may be combined

Example: Mark-Sweep/Copy collector

- big objects managed with mark-sweep (avoids copy time)
- small objects managed with copy collector

Analysis

- + may be more efficient
- more complex

Generational Collecting

Observation

- "young" objects are most likely to die soon, while "old" objects are more likely to live on

Idea

- exploit this fact by concentrating collection on "young" objects

Details

- divide heap in generations (G0, G1, ...; G0 for youngest objects)
- collect G0 most frequently, G1 less frequently, *etc.*
- object is "tenured" from one gen. to next after surviving several GCs

Result

- usually only have to collect a small sub-heap

Generational Collecting (cont)

Additional issues

- need to encode "age" in object
- root set for objects in one generation may come from another gen.
- generation G_i should be k times larger than G_{i-1}
- each generation may be collected with different algorithm

Dealing with cross-generation pointers

- older to younger (*i.e.*, G_i to G_j for $i > j$) are uncommon
- search all of G_i ?
- write barriers
- younger to older (*i.e.*, G_j to G_i for $i > j$) are very common
- collect G_j when collecting G_i

Generational Collecting Analysis

How it handles the key issues

- **allocation** in the youngest heap is fast if a copy collector is used
- **reclamation** is fast because doing collection on smaller heap
- **fragmentation** depends on collector used in each heap

Further Analysis

- + less memory is required if use mark-sweep for older generations
- + possibly better locality
- still sometimes do full, slow collections (embarrassing pause!)
- need to record age with each object

Who does what? Pointers

Issues

- in order to trace reachable objects, we must be able to dynamically determine what is a pointer
 - imagine doing this in C!
 - easier in Java
- how?
 - compiler support and/or runtime tagging
 - convention about what can be a pointer
- what if we're not certain about what is a pointer?
 - be conservative; assume anything that may be a pointer is
 - may keep extra garbage
 - can not move objects (mark-compact)
 - conservative garbage collectors can be used with C

Who does what? Scheduling Garbage Collection

Generally

- When allocation is no longer possible, garbage collection is necessary
- VM usually stops all mutator threads
- JIT generated code must properly handle out-of-memory exception

Write barriers (for reference counting and generational collection)

- Each time a write to a pointer occurs, a write barrier catches this and performs some action
- generation collection needs the write barrier to keep track of pointers from older generations to new generations
- reference counting requires write barriers to detect when any pointer changes

“...Performance Impact of Garbage Collection” [Blackburn et al 2004]

Experiment setup

- use Jikes RVM (research virtual machine), highly optimized
- MMtk (The Memory Management Toolkit) is a framework for construction of garbage collectors within Jikes RVM.
- Machines: Athlon (best performance), Pentium 4, Power PC
- Benchmarks: SPEC JVM benchmarks and pseudojbb (variant of SPEC JBB2000)
- garbage collection algorithms
 - semi-space, a copying tracing collector
 - mark-sweep
 - reference-counting

[Blackburn et al 2004] Some Conclusions

Contiguous allocation is better than free-list allocators

- mutator performance is 5-15% better due to improved data locality

Generational collectors are better than whole heap collectors

- write barrier overhead is only (2-14%) and is outweighed by improvements in collection time

Tracing collection is better than reference counting

- the overhead of reference counting is too expensive
- reference counting can be beneficial for older generations of the heap

Nursery size in generational collector should be about 4-8MB

- debunks the myth that the size should be about L2 cache size (512KB)
- have to get to the point where constant number of roots (about 64KB)

Concrete Memory Management Problem

OpenAnalysis

- goal is to do program analysis of large programs, therefore can't just leak memory
- explicit management is very error prone
 - difficult to debug segfaults
 - for analysis results it isn't clear which objects should own which other objects, therefore an explicit management policy proved very problematic

Options

- use one of the conservative garbage collectors
 - have portability issues
- smart pointers

Smart Pointers

auto_ptr (in C++ standard)

- basic idea

```
void f() {  
    auto_ptr<int> p = new int;  
    *p = 5;  
    ...  
    // the dynamically created object is deleted when  
    // p goes out of scope  
}
```

- problem is that only one auto_ptr can point to a particular object at any one time

OA_ptr in OpenAnalysis

Goals

- allow multiple smart ptrs to refer to the same object, similar to shared_ptr in Boost library
- catch as many common errors statically as possible
- allow the smart pointer to be used just like a normal pointer as much as possible
- use simple template mechanisms so as not to confuse many C++ compilers

Details

- implemented using reference counting
 - the OA_ptr class has a pointer to the object and a pointer to a reference counter

Some tricky stuff

Allowing polymorphism

```
OA_ptr<foo> p;
p = new bar; // bar is a subclass of foo
p->hello(); // hello is a virtual method
(*p).hello();
```

Override the access operators

```
T* operator->() const
{
    assert(mPtr != NULL);
    return mPtr;
}
T& operator*() const
{
    assert(mPtr != NULL);
    return *mPtr;
}
```

Allow type conversions

Want to pass in a subclass to a base class

```
int foo( OA_ptr<base> ) {...}
int main() {
    OA_ptr<sub> p; p = new sub;
    foo(sub);
}
```

Implementing subclass to base class assignment in smart pointer

```
template <class T>
class OA_ptr {
    ...
    // so that can pass a subclass into base class
    // Stroustrup 349-350
    template <class T2> operator OA_ptr<T2> () const
    { return OA_ptr<T2>(mPtr,mRefCountPtr); }
```

Not a perfect solution

Decided against raw pointer comparison to catch more static errors

```
OA_ptr<Location::Location> loc = r->foo(mre);  
if (!loc.ptrEqual(NULL)) { ... }
```

Returning an OA_ptr

```
OA_ptr<LocIterator> AliasMap::getMayLocs(MemRefHandle ref) {  
    int setId = mMemRefToIdMap[ref];  
    OA_ptr<AliasMapLocIter> retval;  
    retval = new AliasMapLocIter(mIdToLocSetMap[setId]);  
    return retval; }
```

Usage that causes dynamic errors

```
OA_ptr<Location> loc; NamedLoc myLoc;  
// DO NOT assign the "this" pointer to an OA_ptr!!!  
loc = this;  
// DO NOT assign the address of a local to an OA_ptr!!!  
loc = &myLoc;
```

Summary

Categorizing garbage collection algorithms

- how is garbage identified? reference counting or tracing
- when is it collected? incrementally, generationally, or all at once
- what allocator is used? contiguous/bump allocator or freelist
- what mechanism is used for reclamation? copying or put on freelist
- how is the heap space managed? split for copying or generations?

Open problems in garbage collection

- still no conclusive evidence that it is always faster or slower than explicit memory management
- how can we measure whether it is or not?

Next Time

Tuesday

- Implementing GC for MiniJava
- Starting register allocation

Assignments

- Read the Project 2 writeup before Tuesday, will send email to list when it has been posted