

## Automatic Memory Management

Adapted from Lectures by  
Prof. Alex Aiken and George Necula (UCB)

## Lecture Outline

- Why Automatic Memory Management?
- Garbage Collection
- Three Techniques
  - Mark and Sweep
  - Stop and Copy
  - Reference Counting

## Why Automatic Memory Management?

- Storage management is still a hard problem in modern programming
- C and C++ programs have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident
  - and so on...
- Storage bugs are hard to find
  - a bug can lead to a visible effect far away in time and program text from the source

## Type Safety and Memory Management

- Some storage bugs can be prevented in a strongly typed language
  - E.g., you cannot overrun the array limits
  - E.g., some fancy type systems (linear types) were designed to prevent errors in programs with manual allocation and deallocation but they complicate programming significantly
- If you want type safety then you must use automatic memory management

## Automatic Memory Management

- This is an old problem:
  - studied since the late 1950s for LISP
- Until recently they were unpopular outside the LISP family of languages
  - Used in earlier functional and object-oriented languages
  - Major impetus only after the advent of Java

## The Basic Idea

- When an object that takes memory space is created, unused space is automatically allocated
  - In Cool, new objects are created by `new X`
- After a while there is no more unused space
- Some space is occupied by objects that will never be used again
- This space can be freed to be reused later

## The Basic Idea (Cont.)

- How can we tell whether an object will "never be used again"?
  - in general it is impossible to tell
  - So use a heuristic to find many (not all) objects that will never be used again
- **Observation:** a program can use only the objects that it can find:

```
let x : A ← new A in { x ← y; ... }
```

  - After  $x \leftarrow y$  there is no way to access the newly allocated object

CS781(Prasad)

L30GC

7

## Garbage

- An object  $x$  is **reachable** if and only if:
  - a register contains a pointer to  $x$ , or
  - another reachable object  $y$  contains a pointer to  $x$
- You can find all reachable objects by starting from registers and following all the pointers
- An unreachable object can never be referred to by the program
  - these objects are called **garbage**

CS781(Prasad)

L30GC

8

## Reachability is an Approximation

- Consider the program:

```
x ← new A;  
y ← new B;  
x ← y;  
if alwaysTrue() then x ← new A else x.foo() fi
```
- **After**  $x \leftarrow y$  (assuming  $y$  becomes dead there)
  - the object  $A$  is not reachable anymore
  - the object  $B$  is reachable (through  $x$ )
  - thus  $B$  is not garbage and is not collected
  - but object  $B$  is never going to be used

CS781(Prasad)

L30GC

9

## Tracing Reachable Values in Coolc

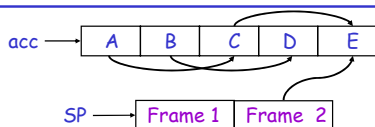
- In coolc, the only register is the accumulator
  - it points to an object
  - and this object may point to other objects, etc.
- The stack is more complex
  - each stack frame contains pointers
    - e.g., method parameters
  - each stack frame also contains non-pointers
    - e.g., return address
  - if we know the layout of the frame we can find the pointers in it

CS781(Prasad)

L30GC

10

## A Simple Example



- In Coolc, we start tracing from **acc** and **stack**
  - they are called the **roots**
- Note that **B** and **D** are not reachable from **acc** or the **stack**
- Thus we can reuse their storage

CS781(Prasad)

L30GC

11

## Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
    - a) Compute what objects might be used again (generally by tracing objects reachable from a set of "root" registers)
    - b) Free the space used by objects not found in (a)
- Some strategies perform garbage collection before the space actually runs out

CS781(Prasad)

L30GC

12

## Mark and Sweep

- When memory runs out, GC executes two phases
  - the **mark phase**: traces reachable objects
  - the **sweep phase**: collects garbage objects
- Every object has an extra bit: the **mark** bit
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the reachable objects in the mark phase

CS781(Prasad)

L30GC

13

## The Mark Phase

```
let todo = { all roots }
while todo ≠ ∅ do
  pick v ∈ todo
  todo ← todo - { v }
  if mark(v) = 0 then (* v is unmarked yet *)
    mark(v) ← 1
    let v1, ..., vn be the pointers contained in v
    todo ← todo ∪ {v1, ..., vn}
  fi
od
```

CS781(Prasad)

L30GC

14

## The Sweep Phase

- The sweep phase scans the heap looking for objects with mark bit 0
  - these objects have not been visited in the mark phase
  - they are garbage
- Any such object is added to the free list
- The objects with a mark bit 1 have their mark bit reset to 0

CS781(Prasad)

L30GC

15

## The Sweep Phase (Cont.)

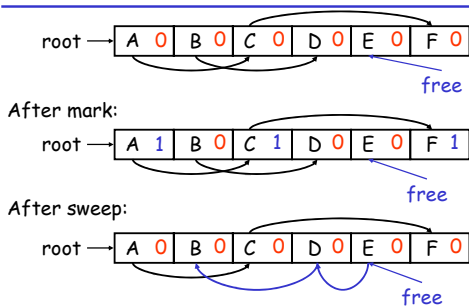
```
(* sizeof(p) is the size of block starting at p *)
p ← bottom of heap
while p < top of heap do
  if mark(p) = 1 then
    mark(p) ← 0
  else
    add block p...(p+sizeof(p)-1) to freelist
  fi
  p ← p + sizeof(p)
od
```

CS781(Prasad)

L30GC

16

## Mark and Sweep Example



CS781(Prasad)

L30GC

17

## Details

- While conceptually simple, this algorithm has a number of tricky details
  - this is typical of GC algorithms
- A serious problem with the mark phase
  - it is invoked when we are out of space
  - yet it needs space to construct the todo list
  - the size of the todo list is unbounded so we cannot reserve space for it a priori

CS781(Prasad)

L30GC

18

## Mark and Sweep: Details

- The todo list is used as an auxiliary data structure to perform the reachability analysis
- There is a trick that allows the auxiliary data to be stored in the objects themselves
  - *pointer reversal*: when a pointer is followed it is reversed to point to its parent
- Similarly, the free list is stored in the free objects themselves

CS781(Prasad)

L30GC

19

## Mark and Sweep. Evaluation

- Space for a new object is allocated from the new list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
  - the left-over is put back in the free list
- Mark and sweep can fragment the memory
- **Advantage**: objects are not moved during GC
  - no need to update the pointers to objects
  - works for languages like C and C++

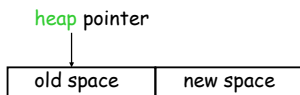
CS781(Prasad)

L30GC

20

## Another Technique: Stop and Copy

- Memory is organized into two areas
  - **old space**: used for allocation
  - **new space**: used as a reserve for GC



- The heap pointer points to the next free word in the old space
- allocation just advances the **heap pointer**

CS781(Prasad)

L30GC

21

## Stop and Copy Garbage Collection

- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - garbage is left behind
  - after the copy phase, the new space uses less space than the old one before the collection
- After the copy, the roles of the old and new spaces are reversed and the program resumes

CS781(Prasad)

L30GC

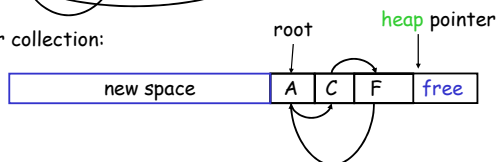
22

## Stop and Copy Garbage Collection. Example

Before collection:



After collection:



CS781(Prasad)

L30GC

23

## Implementation of Stop and Copy

- We find and copy all the reachable objects into the new space
  - And we have to fix ALL pointers pointing to moved objects!
- As we copy an object, we store in the old copy a **forwarding pointer** to the new copy
  - when we later reach an object with a forwarding pointer, we know it was already copied

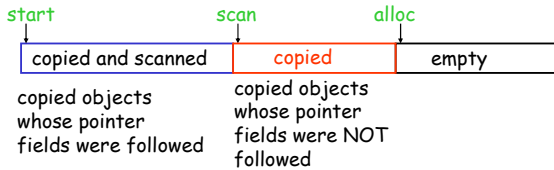
CS781(Prasad)

L30GC

24

## Implementation of Stop and Copy (Cont.)

- We still have the issue of how to implement the traversal without using extra space
- The following trick solves the problem:
  - partition the new space in three contiguous regions



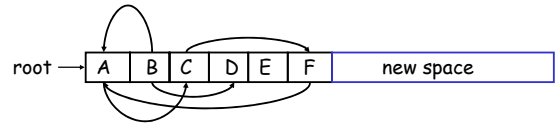
CS781(Prasad)

L30GC

25

## Stop and Copy. Example (1)

- Before garbage collection



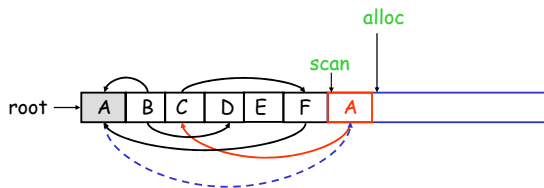
CS781(Prasad)

L30GC

26

## Stop and Copy. Example (3)

- Step 1:** Copy the objects pointed by roots and set forwarding pointers



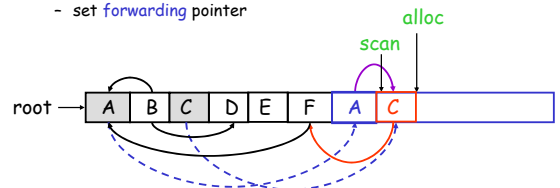
CS781(Prasad)

L30GC

27

## Stop and Copy. Example (3)

- Step 2:** Follow the pointer in the next unscanned object (A)
  - copy the pointed objects (just C in this case)
  - fix the pointer in A
  - set forwarding pointer



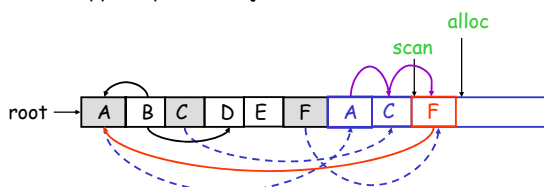
CS781(Prasad)

L30GC

28

## Stop and Copy. Example (4)

- Follow the pointer in the next unscanned object (C)
  - copy the pointed objects (F in this case)



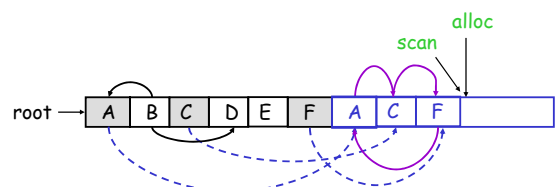
CS781(Prasad)

L30GC

29

## Stop and Copy. Example (5)

- Follow the pointer in the next unscanned object (F)
  - the pointed object (A) was already copied. Set the pointer same as the forwarding pointer



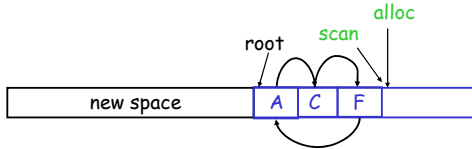
CS781(Prasad)

L30GC

30

## Stop and Copy. Example (6)

- Since scan caught up with **alloc** we are done
- Swap the role of the spaces and resume the program



## The Stop and Copy Algorithm

```
while scan <> alloc do
  let O be the object at scan pointer
  for each pointer p contained in O do
    find O' that p points to
    if O' is without a forwarding pointer
      copy O' to new space (update alloc pointer)
      set 1st word of old O' to point to the new copy
      change p to point to the new copy of O'
    else
      set p in O equal to the forwarding pointer
  fi
end for
increment scan pointer to the next object
od
```

## Stop and Copy. Details.

- As with mark and sweep, we must be able to tell how large is an object when we scan it
  - and we must also know where are the pointers inside the object
- We must also copy any objects pointed to by the stack and update pointers in the stack
  - this can be an expensive operation

## Stop and Copy. Evaluation

- Stop and copy is generally believed to be the fastest GC technique
- Allocation is very cheap
  - just increment the heap pointer
- Collection is relatively cheap
  - especially if there is a lot of garbage
  - only touch reachable objects
- But some languages do not allow copying (C, C++)

## Why Doesn't C Allow Copying?

- Garbage collection relies on being able to find all reachable objects
  - and it needs to find all pointers in an object
- In C or C++, it is impossible to identify the contents of objects in memory
  - E.g., how can you tell that a sequence of two memory words is a list cell (with data and next fields) or a binary tree node (with a left and right fields)?
  - Thus we cannot tell where all the pointers are

## Soundness issue : C++ Problem

```
void main(void) {
  Point *aptr = new Point();
  // casting an object reference to an int
  int i = (int) aptr;
  // normal access to an object and its fields
  aptr->print(); aptr->printxy();
  // freeing an object and nulling a reference to it
  delete (aptr);
  aptr = NULL; aptr->print();
  // segmentation fault only when the object fields are accessed
  // aptr->printxy();
  // casting the int back to object reference
  aptr = (Point*) i;
  // object resurrected !!
  aptr->print(); aptr->printxy();
}
```

## Conservative Garbage Collection

---

- But it is OK to be **conservative**:
  - if a memory word looks like a pointer it is considered a pointer
    - it must be aligned
    - it must point to a valid address in the data segment
  - all such pointers are followed and we overestimate the reachable objects
- But we still cannot move objects because we cannot update pointers to them
  - what if what we thought to be a pointer is actually an account number?

## Reference Counting

---

- Rather than wait for memory to be exhausted, try to collect an object when there are no more pointers to it
- Store in each object the number of pointers to that object
  - this is the reference count
- Each assignment operation has to manipulate the reference count

## Implementation of Reference Counting

---

- **new** returns an object with a reference count of 1
- If  $x$  points to an object then let  $rc(x)$  point to its reference count
- Every assignment  $x \leftarrow y$  must be changed:
  - $rc(y) \leftarrow rc(y) + 1$
  - $rc(x) \leftarrow rc(x) - 1$
  - if  $(rc(x) == 0)$  then mark  $x$  as free
  - $x \leftarrow y$

## Reference Counting. Evaluation

---

- **Advantages**:
  - easy to implement
  - collects garbage incrementally without large pauses in the execution
- **Disadvantages**:
  - cannot collect circular structures
  - manipulating reference counts at each assignment is very slow

## Garbage Collection. Evaluation

---

- Automatic memory management avoids some serious storage bugs
- But it takes away control from the programmer
  - e.g., layout of data in memory
  - e.g., when is memory deallocated
- Most garbage collection implementation stop the execution during collection
  - not acceptable in real-time applications

## Garbage Collection. Evaluation

---

- Garbage collection is going to be around for a while
- Researchers are working on advanced garbage collection algorithms:
  - **concurrent**: allow the program to run while the collection is happening
  - **generational**: do not scan long-lived objects at every collection
  - **parallel**: several collectors working in parallel

## Extra

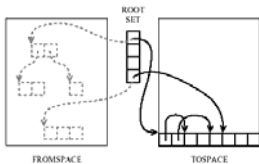
### Generational Incremental

Ref: Paul R. Wilson's  
Uniprocessor Garbage Collection  
Techniques

## Generational GC

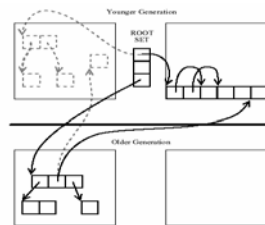
- Advancement Policy and Heap Organization
- Traversal Algorithms and Collection Scheduling
- Intergenerational References
- Pitfalls

## Observations : Copying Garbage Collector



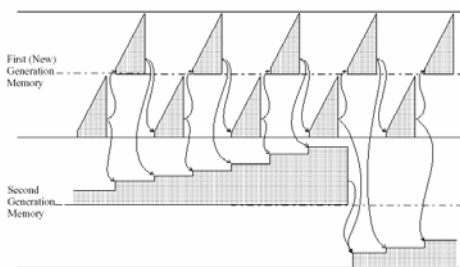
- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It is inefficient that long-lived objects be copied over and over.

## Generational Garbage Collection



Segregate objects into multiple areas by age, and collect older areas *less often* than the younger ones.

## Memory Use in Generational Copy Collector



## Benefits and Practice

- Improves efficiency.
  - Improves locality.
  - Reduces pause times
    - often used as a cheaper substitute for incremental garbage collector.
  - It works with all kinds of collectors.
- The number of generations: 2~8.
  - The space for a younger generation is usually much smaller than for older one.
  - Different collection frequencies for different generations.

## Advancement Policies and Heap Organization

- Advance all live data into next generation.
  - Easy to implement; do not need second semi-space.
  - Advance too fast.
- In general, if there are few generations, retain objects longer; if there are many generations, advance objects faster.
- Generational garbage collectors must be able to tell which generation an object belongs to.
  - For copying collectors:
    - Keep objects of different ages in different areas.
  - For non-copying collectors:
    - Different generations interspersed in memory, objects use a header field to record age.

CS781(Prasad)

L30GC

49

## Older Generations Treated Specially

- Number of sub-areas not same.
  - May become a static space : tenuring.
  - Copying collector may manage large objects differently, to save copying cost.
  - Objects known not to contain pointers may also be segregated from other objects, to optimize tracing and scanning costs.
- Use different algorithms like mark-compact that prefer space to speed.
  - Easy allocation, improved locality, space efficient, but requires multiple passes

CS781(Prasad)

L30GC

50

## Tracking Intergenerational References

- In order to collect younger generation without collecting older ones, intergenerational references must be taken account.
  - Use write barrier
    - It may be a major cost of GC.
    - Alternative: scan at collecting time.
  - Old-to-young pointers are important, young-to-old are not.
    - So write barrier skips object-creating code, only non-initialization pointer stores must be checked.

CS781(Prasad)

L30GC

51

## (cont'd)

*Remembered sets*: Record objects containing pointers to young by setting a header bit. Do scan at collection time.

*Page marking*: Record which virtual memory pages contain old-to-young pointer, use a page-wise table.

*Card marking*: Divide memory into intermediate-sized units called *cards*, record cards which have intergenerational pointer.

CS781(Prasad)

L30GC

52

## Pitfalls of Generational Collection

- The "pig in the python" problem.
  - A cluster of relatively long-lived objects which are created at about the same time and persist for a significant period may be very expensive.
- Small heap-allocated objects.
  - Few pointers from old to young is not always true.
- Large root sets.
  - Generational garbage collection doesn't reduce the size of root size, if the youngest generation is small and frequently collected, scanning a large root set may be a big cost.

CS781(Prasad)

L30GC

53

## Incremental GC Terminology

- Mutator / Collector
- Tricolor marking
  - Black=live, White=*unexplored yet* => garbage
  - Grey=under consideration
- *Invariant*: No black node holds a pointer directly to a white node
  - Read Barrier : mutator does not see white nodes
  - Write Barrier : save pointers to white node written into black nodes

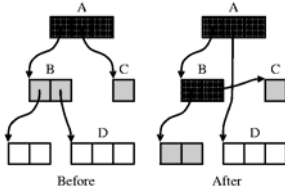
CS781(Prasad)

L30GC

54

## Soundness

- No White nodes become linked (only) to Black nodes



CS781(Prasad)

L30GC

55

## Real-time Generational Collection

- Generational collection can be combined with incremental techniques, but the marriage is not a happy one.
  - Real-time collections focuses on providing absolute worst-case guarantees; Generational collection improves the average performance at the expense of worst-case performance.
  - If generation collection heuristic fails, must do a full garbage collection.
  - May need programmers provide guarantee or "weaker assurance" for object life-times.

CS781(Prasad)

L30GC

56