Garbage Collection

Matthew Dwyer
324E Nichols Hall
dwyer@cis.ksu.edu
http://www.cis.ksu.edu/~dwyer
Garbage Collector

- is part of the run-time system: it reclaims heap-allocated records that are no longer used.

- A garbage collector should:
  - reclaim *all* unused records;
  - spend very little time per record;
  - not cause significant delays; and
  - allow all of memory to be used.

- These are difficult and often conflicting requirements.
Without Garbage Collection

• unused records must be explicitly deallocated;
• superior if done correctly;
• but it is easy to miss some records; and
• it is dangerous to handle pointers.
Memory leaks in real life (ical v.2.1)
Which records are in use?

• Ideally, records that will never be accessed in the future execution of the program.
  – but that is of course undecidable...

• Basic conservative assumption:

  A record is live if it is reachable from a stack-based program variable.

• Dead records may still be pointed to by other dead records.
Heap with Live and Dead Records
Mark and Sweep

• Algorithm
  – explore pointers starting from the program variables, and mark all
  – records encountered;
  – sweep through all records in the heap and reclaim the unmarked ones; also
  – unmark all marked records.

• Assumptions:
  – we know the size of each record;
  – we know which fields are pointers; and
  – reclaimed records are kept in a freelist.
function DFS(x)
    if x is a pointer into the heap then
        if record x is not marked then
            mark record x
            for i in 1 .. |x| do
                DFS(x.f_i)
function Mark()  
for each program variable $v$ do  
  DFS($v$)
Mark-and-sweep Code

function Sweep()
    \( p := \) first address in heap
    while \( p < \) last address in heap do
        if record \( p \) is marked then
            unmark record \( p \)
        else
            \( p.f_1 := \) freelist
            freelist := \( p \)
            \( p := p + \) sizeof(record \( p \))
    \end{while}
\end{function}
Example
Example
Example
Example
Example
Example
Example

freelist
Example

Diagram showing a linked list structure with nodes labeled 12, 15, 37, 59, and 20. The diagram includes arrows connecting the nodes and a mention of "freelist."
Analysis of Mark-and-Sweep

Assume the heap has $R$ of $H$ words that reachable.
The cost of garbage collection is:

$$c_1R + c_2H$$

Realistic values are:

$$10R + 3H$$

The cost per reclaimed word is:

$$(c_1R + c_2H)(H-R)$$

- if $R$ is close to $H$, then this is expensive;
- the lower bound is $c_2$;
- increase the heap when $R > 0.5H$; then
- the cost per word is $c_1 + 2c_2 = 16$. 
Other Issues

- The DFS recursion stack could have size $H$ (and has at least size $\log H$), which may be too much
  - however, the recursion stack can cleverly be embedded in the fields of marked records (pointer reversal).
- Records can be kept sorted by sizes in the freelist. Records may be split into smaller pieces if necessary.
- The heap may become fragmented: containing many small free records but none that are large enough.
Reference Counting

• Algorithm
  – maintain a counter of the references to each record;
  – for each assignment, update the counters appropriately; and
  – a record is dead when its counter is zero.

• Advantages:
  – is simple and attractive;
  – catches dead records immediately; and
  – does not cause long pauses.

• Disadvantages:
  – cannot detect cycles of dead records; and
  – is much too expensive.
function Increment(x)
    x.count := x.count + 1

function Decrement(x)
    x.count := x.count - 1
    if x.count == 0 then
        PutOnFreeList(x)
function PutOnFreelist(x)
    Decrement(x.f₁)
    x.f₁ := freelist
    freelist := x

function RemoveFromFreelist(x)
    for i in 2 ... |x| do
        Decrement(x.fᵢ)
Stop-and-Copy Counting

• Algorithm
  – divide the heap into two parts;
  – only use one part at a time;
  – when it runs full, copy live records to the other part; and
  – switch the roles of the two parts.

• Advantages:
  – allows fast allocation (no freelist);
  – avoids fragmentation;
  – collects in time proportional to $R$; and
  – avoids stack and pointer reversal.

• Disadvantage:
  – wastes half your memory.
Example Stop-and-copy

next identifies address of a block of free memory

limit is maximum address
Example Stop-and-copy
Example Stop-and-copy
Example Stop-and-copy

Copied records are contiguous in memory
function Copy()
    scan := next := start of to-space
    for each program variable $v$ do
        $v := \text{Forward}(v)$
    while scan < next do
        for $i$ in $1 .. |\text{scan}|$ do
            scan.$f_i := \text{Forward}(\text{scan}.f_i)$
        scan := scan + sizeof(record scan)
function Forward(p)
    if p in from-space then
        if p.f₁ in to-space then
            return p.f₁
        else
            for i in 1 .. |p| do
                next.fᵢ := p.fᵢ
                p.fᵢ := next
            next := next + sizeof(record p)
            return p.f₁
        else
            return p
    fi
fi
Analysis of Stop-and-Copy

Assume the heap has $R$ of $H$ words that reachable.
The cost of garbage collection is:

$$c_3R$$

Realistic values are:

$$10R$$

The cost per reclaimed word is:

$$(c_3R)(H/2 - R)$$

- no lower bound as $H$ grows;
- If $H = 4R$ then the cost is constant at approximately 10
Earlier Assumptions

We assumed

– we know the size of each record; and
– we know which fields are pointers.

For object-oriented languages, each record already contains a pointer to a class descriptor.

For general languages, we must sacrifice a few bytes per record.
For You To Do

- What algorithm should we use?
- Under what conditions?
Common Algorithms

• Mark-and-sweep or stop-and-copy
• Garbage collection is expensive
  – ~100 instructions for a small object
• Extensions
  – Generational and Region-based Collection
    • Identify objects that die around same time and collect them at once -- faster
  – Incremental and partial collection
    • Reclaim only a part of the heap at a time -- smoother
Generational Collection

• observation: the young die quickly;
• hence the collector should focus on young records;
• divide the heap into generations: $G_0$, $G_1$, $G_2$, …;
• all records in $G_i$ are younger than records in $G_{i+1}$;
• collect $G_0$ often, $G_1$ less often, and so on; and
• promote a record from $G_i$ to $G_{i+1}$ when it survives several collections.
Generational Collection

• How to collect the $G_0$ generation:
  – roots are no longer just program variables but also pointers from $G_1$, $G_2$, ...;
  – it might be very expensive to find those pointers;
  – fortunately, they are rare; so we can try to remember them.

• Ways to remember:
  – maintain a list of all updated records (use marks to make this a set);
  – mark pages of memory that contain updated records (in hardware or software);
  – Syntactic extensions (e.g., RTJava scoped memory)
Incremental Collection

Garbage collection may cause long pauses
  – this is undesirable for interactive or real-time programs; so
  – try to interleave the garbage collection with the program execution.

Two players access the heap:
  – the *mutator*: creates records and moves pointers around; and
  – the *collector*: tries to collect garbage.

Some invariants are clearly required to make this work.
  – The mutator will suffer some slowdown to maintain these invariants.