Lecture #37

Friday: HKN surveys. Extra points awarded to those who participate!

Today: A little side excursion into nitty-gritty stuff: Storage management.
Scope and Lifetime

• **Scope** of a declaration is portion of program text to which it applies (is visible).
  - Need not be contiguous.
  - In Java, is static: independent of data.

• **Lifetime or extent** of storage is portion of program execution during which it exists.
  - Always contiguous
  - Generally dynamic: depends on data

• **Classes of extent:**
  - **Static:** entire duration of program
  - **Local or automatic:** duration of call or block execution (local variable)
  - **Dynamic:** From time of allocation statement (**new**) to deallocation, if any.
Explicit vs. Automatic Freeing

- Java has no means to free dynamic storage.
- However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

```java
IntList wasteful ()
{
    IntList c = new IntList (3, new IntList (4, null));
    return c.tail;
    // variable c now deallocated, so no way
    // to get to first cell of list
}
```

- At this point, Java runtime, like Scheme’s, recycles the object c pointed to: garbage collection.
Under the Hood: Allocation

- Java pointers (references) are represented as integer addresses.
- Corresponds to machine’s own practice.
- In Java, cannot convert integers ↔ pointers,
- But crucial parts of Java runtime implemented in C, or sometimes machine code, where you can.
- Crude allocator in C:

```c
char store[STORAGE_SIZE]; // Allocated array
size_t remainder = STORAGE_SIZE;

/** A pointer to a block of at least N bytes of storage */
void* simpleAlloc (size_t n) { // void*: pointer to anything
    if (n > remainder) ERROR ();
    remainder = (remainder - n) & ~0x7; // Make multiple of 8
    return (void*) (store + remainder);
}
```
Example of Storage Layout: Unix

- OS gives way to turn chunks of unallocated region into heap.
- Happens automatically for stack.
Explicit Deallocating

- C/C++ normally require explicit deallocation, because of
  - Lack of run-time information about what is array
  - Possibility of converting pointers to integers.
  - Lack of run-time information about unions:
    
    ```c
    union Various {
        int Int;
        char* Pntr;
        double Double;
    } X; // X is either an int, char*, or double
    ```

- Java avoids all three problems; automatic collection possible.

- Explicit freeing can be somewhat faster, but rather error-prone:
  - Memory corruption
  - Memory leaks
Free Lists

- Explicit allocator grabs chunks of storage from OS and gives to applications.
- Or gives recycled storage, when available.
- When storage is freed, added to free list data structure to be recycled.
- Used both for explicit freeing and some kinds of automatic garbage collection.
- Problem: free memory fragments.
Boundary Tag Methods

G1 = malloc(96);
X = malloc(115);
Y = malloc(156);
G2 = malloc(19);
Z = malloc(26);
G3 = malloc(155);
free(G1); free(G3); free(G2);
Simplifying Coalescence: The Buddy System

- Allocate in powers of 2.

- Coalesce only with your buddy:
  - For object of size $2^n$ at byte $\# M$, buddy at byte $\# (M \sim (1 << n))$.
  - Just need a bit to indicate if it is allocated, plus list of free blocks for each $n$. 

![Diagram of Buddy System](image-url)
Buddy System at Work

$X = \text{malloc (32)}$;

$Y = \text{malloc (32)}$; $Z = \text{malloc (64)}$; $Q = \text{malloc (32)}$; $R = \text{malloc (64)}$

$\text{free (X)}$; $\text{free (Q)}$;

$\text{free (Y)}$; $\text{free (R)}$ ;
Garbage Collection: Reference Counting

- Idea: Keep count of number of pointers to each object. Release when count goes to 0.

\[
\begin{align*}
\text{X:} & \quad 1 \quad 1 \quad 1 \\
& \quad 1A \quad 1B \quad 1C \\
\text{Y:} & \quad \_
\end{align*}
\]

\[
\begin{align*}
\text{Y = X.tail;} \\
\text{X:} & \quad 1 \quad 2 \quad 1 \\
& \quad 1A \quad 1B \quad 1C \\
\text{Y:} & \quad \_
\end{align*}
\]

\[
\begin{align*}
\text{X = Y;} \\
\text{X:} & \quad 0 \quad 3 \quad 1 \\
& \quad 1A \quad 1B \quad 1C \\
\text{Y:} & \quad \_
\end{align*}
\]

\[
\begin{align*}
\text{Y:} & \quad \_
\end{align*}
\]

\[
\begin{align*}
\text{X:} & \quad 2 \quad 1 \quad 1 \\
& \quad OA \quad 1B \quad 1C \\
\text{etc.}
\end{align*}
\]
Garbage Collection: Mark and Sweep

1. Traverse and mark graph of objects.

2. Sweep through memory, freeing unmarked objects.

Before sweep:

After sweep:
**Copying Garbage Collection**

- Mark-and-sweep algorithms don't move any existing objects—pointers stay the same.

- The total amount of work depends on the amount of memory swept—i.e., the total amount of active (non-garbage) storage + amount of garbage. Not necessarily a big hit: the garbage had to be active at one time, and hence there was always some “good” processing in the past for each byte of garbage scanned.

- Another approach: *copying garbage collection* takes time proportional to amount of active storage:
  
  - Traverse the graph of active objects breadth first, copying them into a large contiguous area (called “to-space”).
  - As you copy each object, mark it and put a *forwarding pointer* into it that points to where you copied it.
  - The next time you have to copy a marked object, just use its forwarding pointer instead.
  - When done, the space you copied from (“from-space”) becomes the next to-space; in effect, all its objects are freed in constant time.
Copying Garbage Collection Illustrated

(a)

 Roots
\[
\begin{array}{c}
B \\
B' \\
E \\
E'
\end{array}
\]

 from: \[
42 \quad D \quad G \quad F \quad A \quad \times \quad 7 \quad G \quad D \quad \times \quad C \quad \times \quad E
\]

to: 


(b)

 Roots
\[
\begin{array}{c}
B' \\
B \\
E' \\
E
\end{array}
\]

 from: \[
42 \quad B' \quad G \quad F \quad A \quad \times \quad 7 \quad G \quad E' \quad \times \quad C \quad \times \quad E
\]

to: \[
D \quad G \quad D \quad \times \quad E'
\]

 B: Old object
B': New object

(c)

 Roots
\[
\begin{array}{c}
B' \\
B \\
E' \\
E
\end{array}
\]

 from: \[
42 \quad B' \quad G \quad F \quad A \quad D' \quad 7 \quad G \quad E' \quad \times \quad C \quad G' \quad E
\]

to: \[
D' \quad G' \quad D \quad \times \quad 7 \quad G \quad E
\]

(d)

 Roots
\[
\begin{array}{c}
B' \\
B \\
E' \\
E
\end{array}
\]

 from: \[
42 \quad B' \quad G \quad F \quad A \quad D' \quad 7 \quad G \quad E' \quad \times \quad C \quad G' \quad E
\]

to: \[
D' \quad G' \quad D' \quad \times \quad 7 \quad G' \quad E'
\]

Last modified: Mon Nov 26 15:00:50 2012