;login:

THE MAGAZINE OF USENIX & SAGE

February 2001 • volume 26 • number 1

inside:

GLEN MCCLUSKEY
JAVA PERFORMANCE
RECYCLING OBJECTS

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

java performance

by Glen McCluskey

Glen McCluskey is a consultant with 15 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

<glenm@glenmccl.com>

Recycling Objects

If you've ever looked at trying to optimize dynamic storage allocation (using malloc/free functions in C), one of the issues that comes up is object recycling. The idea is to somehow keep a list of freed objects around as part of your application, so that you don't incur the overhead of malloc/free, and can instead reuse objects.

The Java language uses a different model of storage allocation than C, with a "new" operator for allocating objects, and garbage collection for reclaiming them. But the same idea of object recycling can potentially be applied, and it's worth considering an example and some of the trade-offs with such an approach.

An Example

The example we'll use for this discussion is one that inserts nodes in a binary tree:

```
import java.util.Random;

public class Tree {
    // true if should recycle tree nodes
    static boolean recycle_flag;
    // class for tree nodes
    static class Node {
        int key;
        Node left;
        Node right;
        static Node freelist;
    }
    // root of binary tree
    private Node root;
    // insert into tree
    private Node insert2(Node p, int k) {
        if (p == null) {
            if (Node.freelist != null) {
                p = Node.freelist;
                Node.freelist = Node.freelist.left;
                p.left = null;
                p.right = null;
            }
            else {
                p = new Node();
            }
            p.key = k;
        }
        else if (k < p.key) {
            p.left = insert2(p.left, k);
        }
        else if (k > p.key) {
            p.right = insert2(p.right, k);
        }
        return p;
    }
    public void insertKey(int k) {
        root = insert2(root, k);
    }
    // look up a key

    public boolean findKey(int k) {
        Node curr = root;
        while (curr != null) {
            if (k == curr.key)
                return true;
            if (k < curr.key)
                curr = curr.left;
            else
                curr = curr.right;
        }

        return false;
    }

    // delete the tree and recycle nodes

    private void delete2(Node p) {
        if (p == null)
            return;

        delete2(p.left);
        delete2(p.right);

        p.left = Node.freelist;
        Node.freelist = p;
    }

    public void deleteTree() {
        if (recycle_flag)
            delete2(root);

        root = null;
    }

    // driver

    public static void main(String args[]) {
        Random rn = new Random(0);

        recycle_flag = true;

        Tree t = new Tree();

        long start = System.currentTimeMillis();

        for (int i = 1; i <= 500; i++) {
            int n = (int)(rn.nextFloat() * 25000);
            for (int j = 1; j <= n; j++) {
                int r = rn.nextInt();
                t.insertKey(r);
                if (!t.findKey(r))
                    System.err.println("error: " + r);
            }
            t.deleteTree();
        }

            long elapsed = System.currentTimeMillis() - start;

            System.out.println(elapsed);

    }
}
```

The program does 500 iterations, with 0–25000 random nodes inserted at each iteration.

At the end of an iteration, tree nodes are freed. If recycle_flag is false, then freeing consists simply of saying:

```
root = null;
```

which makes all the tree nodes unreachable, and subject to garbage collection. If instead we want to recycle nodes, the tree must be traversed and all the nodes added to a free list, a list whose head is represented as a static field (a single copy across all Node objects) in Node. The free list is consulted in the insert2() method, and a reclaimed Node object is used if possible.

Performance

When recycle_flag is set to true, the demo program runs about 20% faster than the standard approach, using a couple of different Java compilers for timing.

In this particular application, it's expensive to walk the binary tree and reclaim all the nodes, and some other simpler use of nodes might result in a greater speedup. For example, you might have a linked list, and you can reclaim the whole list of nodes at fixed cost, by manipulating a few links.

Discussion

Suppose that you use recycling in your program, and it does give you enough of a speed increase to be worth the more complex logic. What are the issues with this approach, other than performance?

One issue is constructors. In Java programming, an object is typically initialized by a constructor, used to set the initial object state. In the example above, however, this issue is sidestepped. For example, in insert2(), when an object is reclaimed from the free list, the "left" and "right" fields must be set to null, which normally would be done as part of default object initialization or by a constructor.

Another point concerns thread-safe programming. If multiple threads are executing the code above, with multiple tree structures active, then there's a big issue with locking while updating the free list. Our demo program doesn't do this, and would need to use synchronized statements to implement such locks. Locking comes at a price, which works against the 20% advantage we claimed earlier.

A third issue is that the program may hold a lot of memory on its free list, which means that the memory is unavailable to the rest of the application.

Is recycling useful? Yes, but perhaps only if you have simple data structures like linked lists, where you can free up all the nodes quickly, and the nodes themselves have obvious initialization semantics. If these requirements are not met, then the costs of recycling seem to outweigh the advantages.