Final Exam
CSC 2/454
18 December 2012

Directions; PLEASE READ

This exam comprises a mixture of short-answer and problem/essay questions. Values are indicated for each; they total 68 points. Question 15 is worth up to 8 extra credit points; it won’t factor into your exam score, but may help to raise your letter grade when I calculate final grades.

This is a closed-book exam. You must put away all books and notes. Please confine your answers to the space provided. In the interest of fairness, I will generally decline to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You will have a maximum of three hours to complete the exam, though hopefully you won’t need it all. Any remaining exams will be collected promptly at 10:30 pm. Good luck!

A Freebie

1. (2 points) Put your name on every page (so if I lose a staple I won’t lose your answers).

Short Answer

(2 points each) Answer the following questions in one sentence each.

2. List the three defining characteristics of object-oriented programming.
   Answer: Encapsulation, inheritance, and dynamic method binding.

3. Why do many programming languages require the step size of a for loop (for i := first to last by step_size do...) to be a compile-time constant?
   Answer: So they can tell whether the step is positive or negative; this determines the “direction” of the loop termination test.

4. As a general rule, when should a thread use scheduler-based synchronization instead of busy-wait synchronization?
   Answer: Whenever there may be something else useful for the processor to do, and the expected wait time is more than twice the cost of a context switch.

5. Why doesn’t a C program need to create a subroutine closure when it passes a function as an argument?
   Answer: Because subroutines don’t nest: the referencing environment always consists of local and global objects only.
6. Many programming languages provide multiple parameter passing modes. Why do Lisp, Python, and Smalltalk need only one?

**Answer:** Because they use a uniform reference model of variables: every parameter is a reference.

7. Why is it not possible to build a stop-and-copy or generational garbage collector for C?

**Answer:** These algorithms require that the language be strongly typed, so the run-time system can find (and if necessary update) all the pointers.

8. Why do many languages permit operations on strings (concatenation, dynamic re-sizing, etc.) that they do not in general permit on arrays?

**Answer:** Because string operations have a big impact on how easy a language is to use, and because strings have properties (one-dimensional, byte-size elements, no nested structure, and [in many languages] immutable content) that make special operations easier to implement than they are in the general case.

**Longer Answers**

9. Commutative and associative laws for addition and multiplication in mathematics state that 

\[(x + y) = (y + x), ((x + y) + z) = (x + (y + z)), (x \times y) = (y \times x), \text{ and } ((x \times y) \times z) = (x \times (y \times z)).\]

(a) (2 points) Explain why it may be desirable for a compiler to use commutative or associative laws to re-arrange expressions in a user’s program.

**Answer:** By re-arranging an expression, a compiler may be able to reduce the number of registers needed for intermediate results, isolate common sub-expressions that only need to be computed once, improve packing of the processor pipeline, or overlap likely cache misses with useful computation.

(b) (2 points) Explain why such re-arrangements may not be safe.

**Answer:** Unlike math, computer arithmetic has limited precision; use of associativity may introduce overflow or underflow. Use of commutativity may produce side effects in a different order, leading to different (though not necessarily incorrect) results.

(c) (3 points) Describe a reasonable set of rules for evaluation order within expressions. Your rules should balance the compiler writer’s desire for implementation flexibility against the programmer’s need to control program behavior.

**Answer:** Language manuals commonly state that there is no implied order of evaluation of operands, so the compiler may apply commutative laws at its discretion. Associative laws, on the other hand, are generally applied only in the absence of parentheses, or in the presence of a specific compile-time directive.

10. (a) (5 points) List five programming language features that were introduced in large part to make it easier for the compiler to generate time or space-efficient code.

**Answer:** `case`/`switch` statements; decimal arithmetic; packed types; `with` statements; short-circuit Boolean evaluation; operator= assignments, explicit deletion of dynamic objects in Pascal, C, etc.; imperative features in Lisp; ‘register’, ‘restrict’, keywords in C; ‘inline’ keyword in C++; ‘volatile’ keyword in C and C++.
(b) (5 points) List five features omitted from many languages largely (despite their conceptual appeal) because of concern for their implementation cost.

**Answer:** recursion in Fortran; closures, first-class subroutines, continuations, unlimited extent; type inference by unification; comparing records with holes for equality; dynamic arrays, variable-length strings; array slice operations; garbage collection; exception handling; coroutines, iterators; higher-order functions, side-effect freedom, lazy evaluation; multiple inheritance; type checked separate compilation; sequential consistency; uniform size numbers across target architectures.

11. (8 points) We have defined a *tail recursive* function as one that makes recursive call(s) (if any) only at the *end* of its code, and returns as its result the result of the recursive call. Write (in Haskell, Scheme, or high-level functional pseudo-code) a tail-recursive factorial function. Show (in imperative pseudo-code) what a good compiler would produce as the translation of this function. (For purposes of this question, you may assume that parameters are evaluated eagerly, and passed by value.)

**Answer:** The obvious answer:

```
fact n
| n == 0 = 1
| otherwise = n * fact (n - 1)
```

is not tail recursive. The following is:

```
fact n =
    helper n 1
where helper n s
      | n == 0 = s
      | otherwise = helper (n - 1) (s * n)
```

This would be translated roughly as:

```
fact (n):
    s = 1
start:
    if n = 0 return s
    s := s * n
    n := n - 1
    goto start
```

12. Consider the Java program shown in Figure 1. Assume that this is to be compiled to native code on a machine with 4-byte addresses.

(a) (4 points) Draw a picture of the layout in memory of the object created at line 17. Show all virtual function tables.
interface Pingable {
    public void ping();
}

class Counter implements Pingable {
    int count = 0;
    public void ping() {
        ++count;
    }
    public int val() {
        return count;
    }
}

public class Ping {
    public static void main(String args[]) {
        Counter c = new Counter();
        c.ping();
        c.ping();
        int v = c.val();
        System.out.println(v);
    }
}

Figure 1: A simple program in Java.

Answer:

![Counter vtable diagram]

(b) (2 points) Give assembly-level pseudocode that can be used to call c.val at line 20. You may assume that the address of c is in register r1 immediately before the call, and that this same register should be used to pass the hidden this parameter. You may ignore the need to save and restore registers, and don’t worry about where to put the return value.

Answer:

// &c in r1
r2 = *r1    // Counter vtable
call *(r2+8) // invoke val (2nd method)

(c) (3 points) Give assembly-level pseudocode that can be used to call c.ping at line 18. Again, assume that the address of c is in register r1, that this is the same register that should be used to pass this, and that you don’t need to save or restore any registers.
(d) (4 points) Give assembly-level pseudocode for the body of method Counter.ping (again ignoring register save/restore).

Answer:

```assembly
Counter.ping: // this in r1
    r2 = *r1 // Pingable vtable
    r1 -= *r2 // offset back to beginning of Counter object
    *(r1+4)++ // increment count
    return
```

13. (8 points) Discuss the circumstances under which a case (switch) statement should be implemented using

- a sequence of if tests
  
  Answer: when the number of arms is small

- a characteristic array
  
  Answer: when the number of arms is large and the set of labels is dense, with no ranges

- a sorted array
  
  Answer: when the number of arms is large and the labels include large ranges

- a hash table
  
  Answer: when none of the above is true. [Post-exam note: many students seemed to think that the order in which the labels appear in the source code was important. It’s not. The compiler will typically sort the labels before choosing how to implement the mapping from values to arms.]

14. Consider an implementation of sets as singly-linked lists in C++, along the lines shown in Figure 2. This code is correct (if overly simplistic) when executed by a single thread.

(a) (3 points) Give an example of incorrect behavior that might arise if two threads try to execute methods of the same set concurrently.

Answer: Here’s one possible scenario: Suppose two threads execute `insert` concurrently, and reach line 25 more or less simultaneously. Both will then update `n->next`. Whichever one does so first will have its value immediately overwritten. The list will still be intact, but the value inserted by the first thread will be lost. Other possible scenarios may leave the list structurally invalid, or cause a thread to dereference a NULL pointer and crash the program.
class node {
    int val;
    node* next;
    friend class list;
};
class list {
    node head;
    node* find_prev(int v) {
        node* n = &head;
        while (true) {
            if (n->next == NULL || n->next->val >= v) return n;
            n = n->next;
        }
    }
};

public:
    list() {
        head.next = NULL;
    }
    void insert(int v) {
        node* n = find_prev(v);
        if (n->next != NULL && n->next->val == v) return;
        node* m = new node;
        m->val = v;
        m->next = n->next;
        n->next = m;
    }
    void remove(int v) {
        node* n = find_prev(v);
        if (n->next == NULL || n->next->val != v) return;
        node* m = n->next;
        n->next = m->next;
        delete m;
    }
};

Figure 2: Implementation of sets as singly-linked lists in C++. Destructor and additional methods
are not shown.

(b) (3 points) Explain how you might use locks to make concurrent execution safe.

Answer: The most straightforward solution is to add a lock field to each set, acquire the
lock at the beginnings of insert and remove, and release it at the end and immediately
before each of the explicit return statements. This solution serializes all operations on
a given list. A small optimization is to release the lock in remove before deleting m.
Higher concurrency can be obtained by putting a lock in each node and then acquiring
those locks in “hand over hand” fashion as one moves down the list. This approach is
substantially more complex, and tricky to get right.

15. (Extra Credit only) Ace programmer Jane Doe has observed that an insert operation in
Figure 2 really “happens” at line 25, where the new node becomes visible to other threads. A
delete operation, likewise, “happens” at line 31, where a node becomes no longer visible. Jane
remembers learning about the atomic compare_and_swap (CAS) instruction, which does the
following, atomically, in hardware:
CAS(word* loc, word old, word new):
    if (*loc == old) {
        *loc = new;
        return true;
    } else return false;

Using CAS, Jane rewrites insert and remove as follows:

    void insert(int v) {
        while (1) {
            node* n = find_prev(v);
            if (n->next != NULL && n->next->val == v) return;
            node* m = new node;
            m->val = v;
            m->next = n->next;
            if (CAS(&n->next, m->next, m)) return;
            else delete m;
        }
    }

    void remove(int v) {
        while (1) {
            node* n = find_prev(v);
            if (n->next == NULL || n->next->val != v) return;
            node* m = n->next;
            if (CAS(&n->next, m, m->next)) {
                delete m;
                return;
            }
        }
    }

(a) (4 points max) Explain Jane’s thinking: why might this (almost) work?

Answer: All of the work in insert prior to line 25, and in remove prior to line 31, is really just preparation, and invisible to other threads, even without synchronization. The delete at line 32 is likewise simply cleanup. The purpose of the CAS—and the return to the top of the loop when it fails—is to perform the update, making changes visible to other threads, only if no one has changed the list out from under us since we decided to make the change.

(b) (4 points max) Give an example of what can still go wrong.

Answer: The heart of the problem is that CAS checks for an expected value only in the location about to be modified. To know that the update is still appropriate, however, we need to know not only that the next pointer we are changing hasn’t been changed by anybody else, but also that the node in which it lies hasn’t been deleted by changing its predecessor’s next pointer.
Consider, for example, a list containing nodes A, B, C, and D in order, and suppose that threads T_1 and T_2 decide, at approximately the same time, to delete nodes B and C, respectively. If T_1 updates A.next to point to C and T_2 updates B.next to point to D, both CASes may succeed. The net effect will be that B is indeed removed, but C remains accessible.

Another problem has to do with reclamation of nodes that have been removed: if their space is reused for another purpose while another thread is scanning down the list, the scanning thread can end up following a garbage pointer.