CORE COMPUTER SCIENCE EXAMINATION
Department of Computer Science
New York University
January 25, 2002

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Attempt all of the questions. Use the proper booklet for each question. Each booklet is marked with the Area and Question number, in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam. Good luck!

Programming Languages and Compilers

Question 1

We want to manipulate dynamically allocated linked lists. Each node of the list holds an integer and a pointer to the next node.

1. In the language of your choice, write the type or class declaration for such a list.
2. Write a constructor for the class, that converts an array of integers of length N into a list of integers. (If you write your answer in a language without constructors, write a function with the same characteristics).
3. We need to reclaim the storage allocated for such a list. In the language you used above, write a procedure that returns all the nodes in a list to the storage pool. Assume that the language has a primitive called 'free', and does not have a garbage collector.
4. Assume now that each node in the list holds a dynamically allocated character string. Rewrite your answer to c) to deallocate ALL the storage occupied by the list.

Solution

The following is the standard approach: define a node class, and a list class. The list class holds a pointer to the first node, which makes it easy to handle emtpty lsts.

```
    struct node {
          int value;
          node* next;
          node (int x) { value=x;};    // basic constructor
          node () {};
    };
    //  many students chose to make the data members of node private,
    //  and then make class mylist a friend of node. OK but heavy.
    //  Another approach is to make node into a nested class of mylist.
   class mylist {
      node* first;
   public:
      mylist () { first = 0;};

      mylist (int *arr, int n) {
        if (n==0) return;
        first = new node (arr[0]);

        node* n1 = first;
        for (int j=1; j < n; j++) {
           n1->next = new node (arr[j]);
           n1 = n1->next;
        }
      };
      // The cleanup can be programmed as follows: to delete a linked list
      // we must delete each node. We must be careful to delete the successor
      // before deleting the node itself, because the call to free makes it
      // illegal to access the contents of the object afterwards. One simple
      // recursive way of describing the list traversal is as follows (using
      // a while-loop is ok of course).

      void freeall (node* p) {
         if (p != 0) {
            freeall (p->next);
            free (p);
         }

      ~list () {freeall (first);};   // the real destructor
   };
```

`If the node also has a char* data member, add the corresponding call to free to the body of freeall.`

Question 2

A number of languages (including Java and Fortran90) have a **continue** statement, which is used to affect the flow of control within a loop:

Continue_Statement ::= **continue** optional_label

The semantics of the continue statement is as follows:

- If the optional label is absent, the result of executing the statement is to skip the rest of the current iteration of the immediately enclosing loop, and start the next iteration.
- If the label is present, it has to be the label of some enclosing loop. In that case the result of executing the statement is to start the next iteration of the loop that has that label. We can write for example:

```
outer: while (Cond1) {
                Compute1;
     inner:  while (Cond2) {
                      Compute2;
                      if (cond3) continue outer;
                      Compute3;
                 }
            }
```

1. Can the rule that the label must correspond to that of an enclosing loop be described in context-free syntax? Explain briefly..
2. What attributes do we need to attach to each loop construct to implement loops with continue statements?
3. Show the quadruples generated for the nested loops given above.
4. Using the answers above, what is the proper translation for a **break** statement?

1 Solution

1. The rule that a label on a continue statement must be the label of an enclosing loop statement cannot be described by a context-free

rule. Informally, the rule places a constraint on the expansion of the production for the continue statement, that depends on a production located arbitrarily far in the tree. This is a context-sensitive constraint (nothing more formal than this handwaving explanation was required).

2. The expansion of while-loop requires two labels, call them starloop and endloop, to designate the beginning of the loop and the loop exit. When the loop is expanded into quadruples, it is translated as follows:

```
startloop:

  ... quadruples to compute Cond into temporary T1
  if not T1 goto endloop;
  ... quadruples for the body of the loop;
  goto startloop;

endloop:
```

   In addition of these two attributes, which are generated internally to produce unique labels for each loop, we need to attach to the loop the label specified by the user. This is needed to verify the validity of the continue statement, and to identify the loop whose flow of control is affected by that statement. Suppose that the continue statement mentions loopi. Then the continue statement is translated into

   goto startloopi

3.
```
startloop1:
    quadruples to compute Cond1 into T1
    if not T1 goto endloop1;

startloop2
    quadruples to compute Cond into T2
    if not T2 goto endloop2;
    quadruples for Compute2

    quadruples to compute Cond3 into T3
    if T3 goto startloop1
endloop2:
endloop1:
```

4. The break statement is translated into a jump to the corresponding endloop.

Question 3

The "strength reduction" optimization involves replacing a more complex operation (e.g., multiplication), by a simpler one (e.g., addition) in the context of a loop.

1. Explain why this optimization is desirable.
2. Give a set of quadruples for the following loop

```
for (j = 0; j = N; j++)
    a [10 * j] = j;
```

3. Show how the operation of multiplication by 10 can be eliminated by strength reduction. Show the modified set of quadruples.
4. For the machine of your choice, show the optimized assembly language that might be generated for this code.

Question 1

a) Consider a system using the banker's algorithm to manage 20 units of one resource type. There are three processes in the system: A, B, and C. At proces creation A claims 11 units, B claims 5, and C claims 19. After some processing, A holds 1 unit, B has 3, C has 11, and there are no outstanding requests. What is the largest request A can make at this point, that the banker wil grant? If B (instead of A) makes a request, what is the largest request the banker will grant? If C (instead of A or B) makes the request, what is the largest the banker will grant? Justify your answers, that is to say, for each of the three cases, explain why the banker will grant that particular amount and not one more.

b) Consider two binary semaphores S and T. We use Dijkstra's terminology and call the operators on semaphores P and V (Tanembaum uses Down and Up). A project will consist of two processes, one of which is written as:

```
loop forever
    P (S)
    part1 ()
    P (T)
    part2 ()
    V (T)
    V (S)
```

The second process is currently under discussion. The two possibilities being examined are:

```
loop forever            loop forever
  P (T)                      P (S)
  part3 ()                   part5 ()
  P (S)                      P (T)
  part4 ()                   part6 ()
  V (S)                      V (T)
  V (T)                      V (S)
```

part1(), part2().. are straigthforward computations that always terminate and do not use any semaphores.

One of the two possibilities for the second process has a clear (provable) advantage. What is the advantage, which version has it, and why?

Question 2

Consider two demand paging systems D and S. that use 4Kb pages on identical processors. System D uses the normal implementation, where pages are stored on disk, and demand-paged into page frames where needed. System S differs in that the pages are stored on a fast network "page server".

A page fault requires C μsec (microseconds) of CPU time for both systems. In addition, System D requires 10 milliseconds of disk time; whereas system S requires 100 μsec of network access time (including page server costs).

The following behavior has been observed for an application that does no I/O other that paging, and is the only process in the system:

- if run with sufficient memory so that no page fault occurs, the application completes in 40 seconds.
- If run with a some smaller number of page frames (call it N), the application completes in 140.5 seconds on system D, and in 41.5 seconds on system S.

Given this information, answer the following questions:

1. How many page faults occur with N frames?
2. With the same number of frames, how many page faults per second occur on each system?
3. How long would the application take to complete on each system if the CPU speed was doubled, and the same number of frames were used?

Solution

1. The first observation that one should make for this question is that the number of page faults in the two systems is the SAME. The number of page faults is dependent on application page access patterns and is independent of how demand paging is implemented (assuming identical policies for page replacement).

   Assuming that the number of page faults is X, one can then set up the following equations, which relate the total run-time of the application to (i) the execution time without any page faults, and (ii) the time spent on handling page faults.

   The first time was provided as part of the question (40 seconds), The second time can be obtained by observing that each page fault incurs both a CPU overhead component (= C microseconds) and the corresponding access cost (10 ms in system D, and 100 microseconds in system S).

For system D (demand paging against the disk):

```
[converting all times to microseconds
    1 second = 10^6 microseconds,
    1 ms = 10^3 microseconds]

40.0*10^6 + X*(C + 10*10^3) = 140.5*10^6

    For system S (demand paging against a network page server):

40.0*10^6 + X*(C + 100) = 41.5*10^6

    Given two equations and two unknowns, one can solve them to obtain
    the values of C and X:

X*(10*10^3 - 100) = (140.5 - 41.5)*10^6
X*9900 = 99*10^6
X = 10,000
  i.e., the application incurs 10,000 page faults
  with N frames.

    Substituting the value of X back into either of the equations, one
    finds that

        C = 50 microseconds
```

2. Page fault rate is defined as the number of page faults incurred per unit time. Note that although the number of page faults is the same in both cases, because the application takes different amounts of time for the two systems, the rates in the two cases are different.

```
    For system D (demand paging against disk):

page fault rate = X / 140.5 = 71 faults/second

    For system S (demand paging against a network page server):

page fault rate = X / 41.5 = 241 faults/second
```

A common mistake that students make here is computing the page fault rate with respect to only the time spent handling page faults. It is important to observe that (in the absence of information to the contrary) page faults are spread out over the entire program run.

3. This question asks one to compute execution times in the two systems for the application when the CPU speed is doubled. The first observation one should make here is that since the resulting systems are more powerful (everything else stays the same and the CPU speed is doubled), one should expect to get execution times that are SMALLER than the ones in earlier parts.

   The second step is to identify which parts of the execution time are likely to be affected by a doubling of the CPU speed. The answer here is that BOTH the base application execution time of 40 seconds (without any page faults) AND the CPU overhead portion of the page fault handling costs will be affected.

   Note that the question (intentionally) did not specify how the application execution time is affected by a doubling of the CPU speed. It is definitely INCORRECT to assume that there is no change, i.e., the time remains at 40 seconds. It is also NOT COMPLETELY CORRECT to assume (without explicitly stating this) that the time is halved to 20 seconds because of the CPU speed doubling, since other factors such as cache usage and memory access costs can also influence execution time.

   What was expected here was an explicit statement about whatever one ends up assuming about the execution time. Note that such a statement is not required for assuming that the CPU overhead portion of page fault costs are halved (to 25 microseconds) in response to the doubling of CPU speed.

   The third step is to observe that the number of page faults in both the systems stay the SAME as in part (a) since as stated earlier they are only dependent on the application page access pattern, which does not change.

   Given these new costs (assuming that the execution time without any page faults does get halved to 20 seconds), one can compute the new overall execution times as below:

```
    For system D (demand paging against the disk):

20*10^6 + 10000*(25 + 10*10^3) = 120.25*10^6 microseconds
       = 120.25 seconds

    For system S (demand paging against the page server):

20*10^6 + 10000*(25 + 100) = 21.25*10^6 microseconds
   = 21.25 seconds
```