CORE EXAMINATION
Department of Computer Science
New York University
January 22, 1999

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. You are not required to take the algorithms section of the exam if you have passed the FOCS exam in the past.

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

consider the following function in some Pascal-like language:

```
function F(x, y : integer) return integer is
begin
 x := x + 1; y := y + 1; return (x - y);
end F;
```

1. Give and example in which this program behaves differently under call-by-value/result and call-by-reference.

2. Give an example in which this program behaves differently under call-by-value/result and call-by-name.

3. Explain briefly how call-by-name is implemented, and why this is not a parameter-passing mechanism used in modern programming languages.

Solution

1. the call **F(i, i)** distinguishes between call-by-value/result and call-by-reference.
2. the call **F(i, A**[i]) distinguishes between call-by-value/result and call-by-name.
3. Call-by-name has a high run-time cost associated with it, both in representation and in procedure invocation. Recall that the evaluation of the actuals occurs in the environment of the caller, and not the calle. This

   makes it necessary to change the environment to that of the caller every time a name parameter is accessed. This is commonly implemented wirth "thunks" and involving passing a parameterless function (not just a value or a location) as the actual parameter.

Question 2

Compilers for Modern RISC machines perform instruction scheduling, as a result of which instructions may be executed in an order different from that suggested by the source program. Of course, instruction scheduling must respect the semantics of the original program. Consider the following fragments of C code. In each of the following cases, state which of the following three conditions holds:

1. The statements must be executed sequentially in the order given
2. The statements can be executed in parallel
3. The statements can be reordered, but not executed in parallel

Assume that the following declarations apply in all cases:

```
int a,b,c,d;
int *pa,*pb,*pc;
```

(a)
```
a = b + 1;
c = a + 1;
```

```
(b)  a = b;
     c = d;

(c)  a = a + b;
     a = a + c;

(d)  *pa = b;
     c = d;

(e)  a = *pa;
     b = c;

(f)  pa = pb;
     a = *pb;
```

(g) Describe a general rule that determines which of the three situations holds for two consecutive assignment statements.

(h) It is clear that pointers cause trouble, especially in C. What kinds of information might a global optimizer be able to collect to remove some of these limitations?

Solution
First let's answer question g) and then apply the rule:

g) We gather the set of variables referenced and assigned by each assignment statement, being conservative if we do not have accurate information, for example, a reference to *p is considered to be a reference to all variables of the designated type of the pointer p.

Call these sets A1, A2, R1, R2, (A1/A2 are sets of variables assigned by statement 1 and 2 in the pair being considered, R1/R2 are the corresponding reference sets).

```
if A1 does not intersect R2 and
   A2 does not intersect R1 and
   A1 does not intersect A2
then
   statements can be executed in parallel, case 2.
   (it does not matter if R1 and R2 have a non-null intersection)

elsif A1 does not intersect A2
then
```

```
      statements must be executed in sequence, because one statement
      sets variables referenced in the other.
```

Now we are considering possible case 3's. It is hard to catch all possible cases of reordering being allowed. The following case is feasible (and suggested by the examples).

```
   else
      if A1 and A2 consist of exactly one variable, and both statements
      are induction statements of the form

         var := var op expression;

      and var appears in neither expression
      and op is associative

   then the statements can be reordered
```

otherwise they must be executed sequentially

Now apply this to the examples:

```
a) A1 = {a}, R1 = {b}
   A2 = {c}, R2 = {a}


      A1 intersects R2 => case 1 sequential

b) A1 = {a}, R1 = {b}
   A2 = {c}, R2 = {d}

      No intersections => case 2 parallel

c) A1 = {a}, R1 = {a,b}
   A2 = {a}, R2 = {a,c}
```

Intersections suggest case 1, but this pair of assignments meets the requirements for case 3. If you are being really pedantic, and reading the ANSI C standard carefully, and worrying about undefined overflow conditions, then you could argue for case 1, but in practice no compilers would be affected by this (an answer of case 1 here would only be acceptable with this explanation, which no one gave!)

```
d) A1 = {all int variables}, R1 = {b}
```

```
   A2 = {c}, R2 = {d}

      A1 intersects R2 => case 1 sequential

e) A1 = {a}, R1 = {b}
   A2 = {c}, R2 = {a}

      A1 intersects R2 => case 1 sequential

f) A1 = {pa}, R1 = {pb}
   A2 = {a}, R2 = {all int variables}

      No intersections (pa is not an int variable) => case 1 sequential
```

Finally, the answer to h).

A global optimizer can find out tighter information on what pointers might point to. Trivially if the & operator is never applied to a non-array variable, then it is impossible for a pointer to legitimately point to this variable. For example in d, if the & operator was never applied to the variable d, then we know the intersection is false, and the statements can be executed in parallel. .

A more elaborate approach could use dataflow to determine more precisely what possible variables could be referenced by a given pointer at each point in the flow graph, again giving more precise results on the intersection tests, and allowing more statements to be executed in parallel (or reordered as the case may be).

Question 3

1. Explain the purpose of the dispatch table (the vtable in C++ parlance) in the run-time environment of object-oriented languages.

2. Given the following declaration:

```
class A {
    int value;
public:
    void Reset { value = 0;}
    virtual int retrieve { return value;};
    virtual int set (int New_Val) { value = New_Val;}
    virtual void combine (A thing);
}
```

Indicate the data layout of an instance of A, and the layout of the vtable for A.

3. Suppose class B is an extension of A. How do you declare a method *combine* in class B, that overrides the method combine inherited from A? What does the vtable of B look like?

4. Suppose class B also declares the following methods:

```
virtual void combine (int x1, x2);
int mangle (double z);
```

   What is the vtable for B now?

5. Consider the following code:

```
A* thing1 = new B;
A* thing2 = new B;
...
thing2 -> Combine (thing1);
```

   Sketch the code that the compiler must generate for the call to Combine. You can use C for your answer. or assembly code for any machine you know.

Solution

1. The dispatch table is a run-time data-structure used to support polymorphism in object-oriented languages. It is a table of pointers to

   virtual methods (primitive operation in Ada) whose identity is determined at run-time by the class of the object to which they are applied.

2. For class A, the dispatch table will hold pointer to methods *retrieve*, *set*, and *combine*. In C++, only methods that are declared virtual need to appear in the dispatch table (in Java, all methods are virtual, and they are all present in the dispatch table). An instance of A stores the data members (in this case only *value*) and a **pointer** to the dispatch table. The dispatch table is shared by all instances of the class, so there is no need to place it in each object.

3. To override an inherited method, the new definition must have the exact same signature:

```
virtual void combine (A thing);
```

   The dispatch table for B has the same layout as that for A, but the entry for *combine* points to the body of the overriding definition, not the inherited one.

4. The declaration of *combine* has a different signature, and is an overloading of the previous one, so it has its own entry in the dispatch table, which is now extended.

5. The run-time call must retrieve the operation to ve invoked from the corresponding entry in the dispatch table. The method *combine* appears in the third slot in the table, so the code generated has the form:

```
(*thing2.vtabl(2))(&thing2, thing1);
```

Note that the dispatching call must carry the actual parameter for *this*.

Question 1

- List three scheduling criteria that might impact the choice of uniprocessor CPU scheduling policy in an operating system.
- Choosing one or more of the criteria above, show examples of situations in CPU scheduling where:

  1. FIFO (non-preemptive) performs better that Round Robin
  2. Round Robin performs better than FIFO (non-preemptive).

  Assume that a process can be modeled as alternating bursts of CPU and I/O activity, a process blocks for I/O; upon completion of I/O activity it rejoins the CPU ready queue.

  To answer the above, in each case, show the CPU and I/O burst nature of the processes and draw the diagrams. Also, indicate the criterion by which you can say that a policy 'performs better'.

Solution

OS 1.1

The choice of CPU scheduling algorithms is influenced by the criteria that are being optimized by the operating system. The question was *not* referring to job or system characteristics (such as lengths of CPU and I/O bursts, or the amount of memory in the system): these influence how the algorithm may perform, but not what it does. Any three of the following list of criteria would have been an acceptable solution:

Long- and Medium-term Schedulers:

1. Relative importance given to different classes of jobs (e.g., batch, interactive, etc.).
2. The degree of multiprogramming.

Short-term Schedulers: From the system's perspective:

1. Average waiting time: how long do jobs wait to get the CPU?
2. Throughput: the number of processes completed per unit time.

3. CPU utilization: percentage of time the CPU is busy.

4. Fairness: ensuring that no process is starved for resources.

5. Enforcing priorities: higher-priority processes should not wait for lower-priority processes.

From the user's perspective:

1. Response time: time it takes to produce the first response.

2. Turnaround time: time spent from the time of submission to time of completion.

3. Deadlines: time within which the program must complete. The CPU scheduling algorithm would attempt to maximize number of deadlines met.

4. Predictability: expectation that the job runs the same regardless of system load.

OS 1.2

The question asked for examples of situations where the first-come first-served (FCFS) scheduling policy (without preemption) would outperform the round-robin (RR) policy, and where RR would outperform FCFS. One would get only partial credit for general discussions of job characteristics better suited to FCFS as compared to RR. To get complete credit, the answer needed to specify: (a) the metric by which one policy outperforms the other, (b) process arrival times and order, (c) process CPU and I/O burst lengths, (d) the quanta used by the RR policy, and (e) the overhead of context switching.

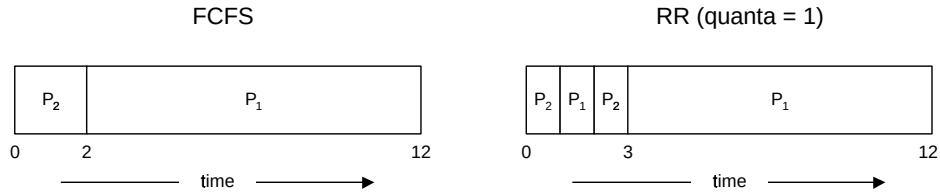
One of the simplest examples showing the relative merits of FCFS and RR involves two jobs, P_1 and P_2, with the following characteristics:

- P_1 arrives at time t=0 and comprises a single long CPU burst of 10 units.

- P_2 arrives at time t=0 and comprises a single short CPU burst of 2 units.

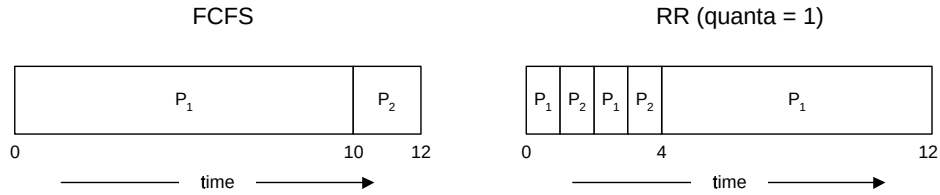
- The quanta used by the RR policy is 1 unit, and the overhead of context switching is assumed to be negligible (as compared to the chosen time scale).

Using **average completion time** as the metric, FCFS outperforms RR when P_2 arrives before P_1. The average completion time using FCFS is



(2+12)/2 = 7. The average completion time using RR is (3+12)/2 = 7.5. The Gantt charts for the two policies are shown below:

Using **average completion time** as the metric, RR outperforms FCFS when P_1 arrives before P_2. The average completion time using FCFS is (10+12)/2 = 11. The average completion time using RR is (4+12)/2 = 8. The Gantt charts for the two policies are shown below:



Question 2

There are four processes, P0, P1, P2, P3. There are four semaphores in an array called *s*. The structure of each process is:

```
while (1) {
   P(s[id]);
   print id;
   V (s[(id+1) mod 4];
}
```

id is a variable local to each process, that contains the process-id of the process, that is to say id=0 for P0. etc.

Semaphore s[0] is initialized to 1, and all the others are initialized to zero.

- Describe the execution behavior of the processes, and indicate what the output of the system is.
- Write the code for the processes using a Monitor.

Solution

OS 2.a

Initially, only process P_0 will be able to proceed past the `P(s[id])` statement since only `s[0]` is initialized to 1. After P_0 finishes executing the `print id` statement, it will set `s[1]` to 1 and go back to the head of the while loop where it will wait for someone to set `s[0]` to 1 again. Meanwhile P_1 can proceed past the `P(s[id])` and wakes up P_2, which in turn wakes up P_3, which in turn wakes up P_0 and so on. Note that the semaphores ensure that only one process executes the print statement at any one time, resulting in the following output: `0, 1, 2, 3, 0, 1, 2, 3, ....`

OS 2.b

The following shows how the process synchronization structure can be expressed using Conditional Critical Regions:

Define a critical region `v` consisting of an integer field, `turn`, which signifies the process that has to go next. Initially, `turn = 0`.

```
var v : shared record
                    turn: integer;
                end;
```

Process P_i executes the following code. The condition ensures that the process waits until it is its turn, prints its `id`, and then updates `turn` to allow P_{i+1} to proceed.

```
while (1) {
  region v  when (turn == id)
    do begin
         print id;
         turn := (turn + 1) mod 4;
    end;
}
```

OS 2.c

The following shows how the process synchronization structure can be expressed using Monitors:

Define a monitor type, `turnMonitor`, which contains an integer `turn` variable, and an array of condition variables, `queue`. The `turn` variable is used to designate the process whose turn it is next, and the `queue` condition variable is used to queue processes that enter the monitor out of turn. Note that the use of per-process condition variables ensures that a process is woken up only when it is its turn. The `doPrint` procedure just queues a process if it is not yet its turn, signalling the next process upon completion.

```
type turnMonitor =  monitor
     var turn: integer;
     var queue:  array [0..3] of condition;

     procedure entry doPrint( id: integer );
        begin
           if (turn != id) then queue[id].wait;
          print id;
          queue[(id+1) mod 4].signal;
        end;

     begin
       turn := 0;
     end;
```

Process P_i executes the following code:

```
 var tm : turnMonitor;

 while (1) {
  tm.doPrint( id );
}
```