

Masters Examination

New Format

Department of Computer Science
The University of Arizona
Fall 2002

October 25, 2002

Instructions

This examination consists of ten problems. The questions are in three areas:

1. Theory: CSc 545, 573, and 520;
2. Systems: CSc 552, 553, and 576; and
3. Applications: CSc 560, 525, 522, and 533.

You are to answer any *two* questions from each area (i.e., a total of *six* questions). If more than two questions are attempted in any area, the two highest scores will be used.

You have two hours to complete the examination. Books or notes are not permitted during the exam.

At the end of the examination, submit just your six solutions. Do not include scrap paper.

Write your answers on the tablet paper provided separately. Start the answer to each question on a separate sheet of paper. On *each page* that you hand in, write the problem number in the upper left corner and the last four digits of your CSID number (*not* your social security number!) in the upper right corner. Hand in your solutions in the envelope provided. Results will be posted using the four-digit numbers.

Some problems will ask you to write an algorithm or a program. Unless directed otherwise, use an informal pseudo-code. That is, use a language with syntax and semantics similar to that of C or Pascal. You may invent informal constructs to help you present your solutions, provided that the meaning of such constructs is clear and they do not make the problem trivial. For example, when describing code that iterates over the elements of a set, you might find it helpful to use a construct such as

for $c \in S$ **do** *Stmt*

where S is a set.

SOLUTIONS

1 Theory 1 (CSc 520: Principles of Programming Languages)

A modification of the *call-by-name* parameter-passing scheme is the so-called *call-by-need* or *normal* rule (also called the “delay rule” or “lazy evaluation rule”). Under this rule, an actual argument to a call is not evaluated *unless and until* the formal parameter to which it is bound is referenced in the called procedure, i.e., no argument is evaluated until it is needed. Furthermore *if* an actual argument is evaluated, the *same value is used thereafter* for all subsequent references to the corresponding formal parameter in the called procedure. An argument is thus evaluated 0 or 1 times but no more.

Assume in this problem that the languages under discussion do *not* have pointer types, so that pointers cannot be passed by value as arguments.

- (a) Suppose you have a programming language that implements parameter-passing using call-by-reference. Describe a general method for simulating procedures with call-by-need parameters in this language. The technique should work for any procedure p —it should not be *ad hoc* for a specific example. Furthermore, it should involve only changes to the code of p (and the code of its caller) that could be done by an automated translator that knows nothing about the meaning of p .
- (b) Suppose you have a programming language (*no pointers*), that uses call-by-need. Show how to simulate the effect of a procedure call in which all parameters are treated as call-by-value. Again it should be possible to automate your translation (in principle). You may assume that the language provides a primitive operation $\text{force}(e)$ that completely evaluates the expression e .

Solution

- (a) The key point here is that we have to make sure that the actual parameter is not evaluated until the first time it is actually referenced during execution; and that subsequent references should not cause it to be reevaluated. So we have to pass a data structure that (i) has enough information in it to allow the actual parameter to be evaluated (in the appropriate environment) at the first runtime reference; and (ii) is able to keep track of whether or not the parameter has been evaluated. This data structure will be passed by reference.

To handle an actual parameter e of type T , the translator creates a new object of type `closure_T`, which is a thunk consisting of (a reference to the code for) a function f_e that evaluates e and returns a value of type T , together with an environment for that function. The parameter that it is actually passed is now (a reference to) a variant record with Boolean tag `evaluated`, as follows:

```
type lazyT = record
  case evaluated : Boolean of
    val: T;
    thnk: closure_T;
end
```

At the call site, the actual parameter e is translated to a `lazyT` object with `evaluated = FALSE` and `thnk = $\langle f_e, \text{caller's environment} \rangle$` . Each reference to the corresponding formal parameter x in the body of the callee is replaced by code that behaves as follows:

- if `x.evaluated == FALSE` then:
 - (i) $x_{tmp} = \text{evaluate } x.\text{thnk}$;
 - (ii) set `x.evaluated = TRUE`;
 - (iii) set `x.val = x_{tmp}` ;where x_{tmp} is a new temporary variable of type T .
- use `x.val`

- (b) The key point here is that we have to force the actual parameters to be evaluated in the caller's environment prior to the actual call itself, regardless of whether the argument gets referenced eventually by the callee.

In this case, a procedure call

$$p(e_1, \dots, e_n)$$

will be translated as follows. We introduce n new temporary variables tmp_1, \dots, tmp_n in the caller, and translate the above call to

$$tmp_1 := \text{force}(e_1); \dots; tmp_n := \text{force}(e_n);$$
$$p(tmp_1, \dots, tmp_n)$$

2 Theory 2 (CSc 545: Design and Analysis of Algorithms)

Consider the rectangle R whose vertices are in coordination $(0, 0), (0, 1), (n, 0), (n, 1)$, where $n \geq 1$ is an integer. See Figure 1. A *normal cover* of R is a set of triangles $\{\Delta_1 \dots \Delta_m\}$ such that

- the endpoints of each edge of each triangle Δ_i are at some points $(0, i), (1, j)$ for some **integers** $0 \leq i \leq n, 0 \leq j \leq n$, (that is, every edge crosses R from bottom to top, and has integer coordination).
- no two triangles overlap, except maybe at their boundaries,
- No triangle has zero area.
- Each point of R is in some triangles $\Delta_k \in \{\Delta_1 \dots \Delta_m\}$.

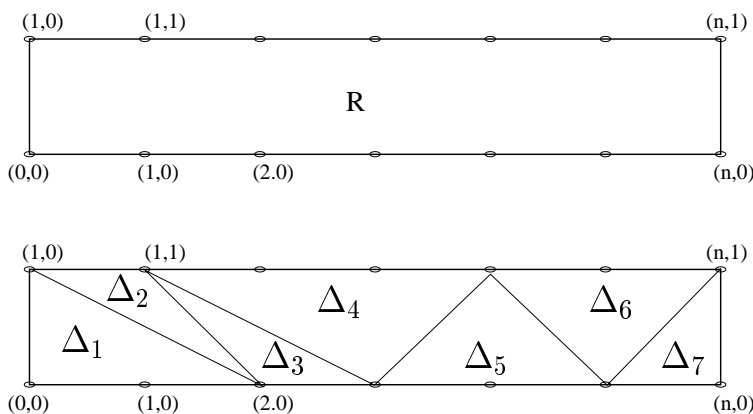


Figure 1: Above — the rectangle R . Below — a example of a normal cover

Assume you are given a function f that assign for each triangle $\Delta_k \in \{\Delta_1 \dots \Delta_m\}$ some number. That is, $f(\Delta_k)$ is defined. For example, $f(\Delta_k)$ can be the square root of the area of Δ_k . However, your answer should not depend on what f is exactly. The computation of $f(\Delta_k)$ can be done in time $O(1)$.

Let the *cost* of the normal cover $\{\Delta_1 \dots \Delta_m\}$ be defined as the sum $\sum_{i=1}^m f(\Delta_i)$. Describe an algorithm which finds a normal cover whose cost is minimal. The algorithm should be polynomial in n , and its running time should be as fast as possible.

Solution

We use dynamic programming. Let a *normal triangle* be defined as a triangle that participate in a normal cover of R . Let T be an $n \times n$ table, such that $T[i, j]$ contains the cost of of a set of triangles $\{\Delta_i \dots \Delta_k\}$, such that each point of R to the left of the edge $(i, 0)(j, 1)$ is contained in some triangle. Clearly $T[n, n]$ contains the cost of the optimal cover. Now

$T[0, 0] = 0$, and for every p, q ,

$$T[p, q] = \min(\begin{array}{l} \min_{0 \leq i < p} f(\Delta(i, 0)(p, 0)(q, 1)) + T[i, q] \\ \min_{0 \leq j < q} f(\Delta(j, 1)(p, 0)(q, 1)) + T[p, j] \end{array})$$

Since we spend linear time for each pair $0 \leq p, q \leq n$, the total time is $\Theta(n^3)$.

3 Theory 3 (CSc 573: Theory of Computation)

Under the assumption that $P \neq NP$, can you obtain a polynomial-time algorithm for the following problems? In each case, justify your answer. In the case that a polynomial-time algorithm exists, give the most efficient one.

- (a) Given an undirected graph $G = (V, E)$, find a maximal-cardinality set $V' \subseteq V$ of vertices, so that each pair of vertices of V' are connected via an edge of E .
- (b) Given an undirected graph $G = (V, E)$, and two vertices $u, v \in V$ (where $u \neq v$), find the longest path π_l connecting them, that does not use the same vertex twice.
- (c) Given an undirected graph $G = (V, E)$, and two vertices $u, v \in V$ (where $u \neq v$), find the shortest path π_s connecting them, that uses each vertex (excluding v) on π_s exactly twice.

Solution

- (a) This problem is in NP . Shown by creating a graph $G^c = (V, E^c)$ in which an edge $e \in E$ if and only if $e \notin E^c$. Then V' is a maximal independent set of G^c . The latter one is known to be NP-hard.
- (b) This problem is in NP , since we can use such an algorithm for finding Hamiltonian path.
- (c) This problem is in P . The path π_s is obtained by finding the shortest path between u, v , and repeat this path from v “back” to u . Finding shortest path is clearly doable using Dijkstra’s algorithm in time $O((|E| + |V|) \log |V|)$

4 Systems 1 (CSc 552: Advanced Operating Systems)

A *thread* is a unit of execution within an address space. A single address space may contain multiple threads, enabling concurrent execution using shared memory. Most operating systems now provide *kernel threads*, threads implemented in the kernel much like traditional processes. An alternative is *user-level threads*, threads implemented at user-level (probably by a library).

The paper “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism” by Anderson *et al.* promotes the scheduler activation mechanism as providing the benefits of both user-level threads and kernel threads.

- (a) Discuss the advantages and disadvantages of each approach.
- (b) Briefly describe scheduler activations.
- (c) How do scheduler activations retain the benefits of kernel threads?
- (d) How do scheduler activations retain the benefits of user-level threads?

Solution

- (a) Kernel threads are slower, but better integrated with system services. They are slow because 1) thread management operations such as thread creation, destruction, etc., are implemented via system calls, 2) a context-switch between threads also implies a system call, and 3) thread facilities must be provided to support all possible applications (e.g. a variety of scheduling policies). They are better integrated because they are scheduled by the kernel, so pre-emptive scheduling is easy, and if one thread of an application blocks another can be run.

User-level threads are fast for the opposite reasons as those given above: 1) thread management operations are only procedure calls, 2) a context-switch is also a procedure call, and 3) thread facilities can be application-specific. They are poorly integrated with system services for two primary reasons: 1) changes in the state of the underlying kernel threads such as preemption and I/O blocking are handled invisibly by the kernel, and 2) the kernel threads are scheduled by the kernel which is unaware of the state of the user-level threads. This means that runnable user-level threads may not run because another user-level thread initiated I/O, or a kernel thread was preempted, for example.

- (b) Scheduler activations combine both user-level threads and kernel threads. The operating system creates one kernel thread per processor, and assigns them to processes according to some allocation policy. These kernel threads invoke the scheduler in the user-level thread package of the application (hence the name “scheduler activations”). The user-level scheduler then decides which user-level thread to run on the particular activation, and can switch between user-level threads according to its own scheduling policy. If the user-level thread does something that causes the activation to block, the operating system creates a new activation that invokes the user-level scheduler, allowing another user-level thread to run. When the blocked activation unblocks, the user-level scheduler is notified that the user-level thread is now unblocked, and that the process will lose an activation (so that it has only one running activation per processor allocated to the process).
- (c) Scheduler activations are integrated with the kernel, so that if a user-level thread blocks, another activation is given to the process and the whole process doesn’t block. They also allow the application to take advantage of multiple processors, as it gets one activation per processor.
- (d) Scheduler activations multiplex multiple user-threads on the activations, allowing for efficient thread manipulation and application-specific thread facilities

5 Systems 2 (CSc 553: Principles of Compilation)

For each of the following, say, in a sentence or two, what it is and what role it plays in the compilation process:

- (a) lexical analysis
- (b) register allocation
- (c) peephole optimization
- (d) instruction scheduling
- (e) symbol table
- (f) syntax tree
- (g) parsing

Solution

- (a) *Lexical analysis*: This refers to the processing of the input stream of bytes that is read in, to organize them into "tokens", e.g., "keyword while", "identifier", "assignment op", etc., that match specific patterns, and also to discard comments, whitespace, etc., that are not important for the remainder of the compilation process.
Its role is to organize the input character sequence into units that can be more easily processed by the parser.
- (b) *Register allocation*: This refers to identifying which variables in a program can and should be kept in registers at any given program point, and determining which register should be used for each such variable. Its purpose is to improve program performance by reducing the number of memory operations.
- (c) *Peephole optimization*: This refers to the process of examining short sequences of instructions with the aim of replacing commonly occurring instruction sequences by equivalent but more efficient ones. Its role is to improve program performance.
- (d) *Instruction scheduling*: This refers to rearranging the machine instructions generated by a compiler so as to hide the latencies of long-latency instructions. The goal is to improve the program's execution speed.
- (e) *Symbol table*: This is a data structure that is used to store information about variables, functions, etc. Its role is to hold information that is generated during one phase of compilation (e.g., parsing or type checking) so that it can be accessed and used during another compilation phase (e.g., code generation).
- (f) *Syntax tree*: This is an internal representation of a program where leaf nodes represent variables and constants, internal nodes represent computations, and the children of a node represent the program components that operation is applied to. It serves to abstract away from details of the actual ("concrete") syntax of the program and represent the program in a form that simplifies subsequent code generation.
- (g) *Parsing*: This is the process of examining the sequence of tokens produced by the lexical analyzer to verify that the program adheres to the syntax of the programming language, and determine the structure of the program (e.g., in the form of a syntax tree). Its purpose is to organize the input stream of tokens into a form suitable for subsequent processing.

6 Systems 3 (CSc 576: Computer Architecture)

Superscalar processors use a number of techniques including branch prediction, out-of-order execution, and speculative execution to achieve high performance. Give the reasons for the following:

- (a) Why is a two-bit branch predictor more effective than a one-bit predictor in the handling of loops?
- (b) Why are certain types of data hazards encountered in presence of out-of-order execution but not during strict in-order execution?
- (c) What additional complication results in the handling of exceptions in presence of speculative execution?

Solution

- (a) Once the branch predictor is warmed up, the one-bit prediction scheme will mispredict the loop branch during the first and last loop iteration. On the other hand, a two-bit prediction scheme will only mispredict the loop branch during the last loop iteration.
- (b) Since the instructions execute out-of-order, they complete execution out-of-order thus possibly leading to WAR and WAW hazards. Such hazards cannot occur during strict in-order execution.
- (c) An exception caused by a speculatively executed instruction must be delayed till the correctness of the speculation decision has been verified. In case of misspeculation, the exception is discarded/suppressed; otherwise it is reported.

7 Applications 1 (CSc 522: Parallel and Distributed Programming)

- (a) [2 points] Define the term "barrier synchronization point".
- (b) [4 points] Give the code each of two processes, P_i and P_j , would execute to form a two-process symmetric barrier. Also declare and initialize shared variables that you use. The processes should use flags or counters and spin waiting (busy waiting).
- (c) [4 points] Show the structure of an 8-process butterfly or dissemination barrier. You do not need to give any code, but show clearly which process interacts with which in each stage of the barrier.

Solution

- (a) A barrier synchronization point is a point in the code of every process (thread) that all processes must reach before any proceed.
- (b) shared variables: `arrive[i] = 0, arrive[j] = 0`

```
code for process i:  await (arrive[i] == 0); # safety check
                    arrive[i] = 1;         # announce arrival
                    await (arrive[j] == 1); # wait for other process
                    arrive[j] = 0;         # clear other's flag
```

The first line is needed so the code can be reused at the next barrier. If it is not there, process i could set `arrive[i]` to 1 before process j clears the flag.

An alternative way to code this is to increasing integer counters.

```
shared variables: arrive[i] = 0, arrive[j] = 0

code for process i:  arrive[i]++;
                    await (arrive[j] >= arrive[i]);
```

The first and fourth lines of the flag-based solution are not needed, because the counters continually increase. Note, however, that `>=` is needed in the `await` statement. Checking only for equality could lead to livelock, because the equality point might be missed.

- (c) You need three stages for an 8-process barrier. The connections in a dissemination barrier are as follows:

```
stage 1: connect 1 to 2, 2 to 3, 3 to 4, ..., 7 to 8, 8 to 1
stage 2: connect 1 to 3, 2 to 4, ..., 6 to 8, 7 to 1, 8 to 2
stage 4: connect 1 to 5, 2 to 6, ..., 4 to 8, 5 to 1, 6 to 2, ...
```

In a butterfly barrier, connections are symmetric, as follows:

```
stage 1: connect 1 to 2, 2 to 1, 3 to 4, 4 to 3, ..., 7 to 8, 8 to 7
stage 2: connect 1 to 3, 3 to 1, 2 to 4, 4 to 1, 5 to 7, ..., 8 to 6
stage 4: connect 1 to 5, 5 to 1, 2 to 6, 6 to 2, ..., 5 to 8, 8 to 5
```

A dissemination barrier is better in general because it works for any number of processes. A butterfly barrier requires a power of 2.

8 Applications 2 (CSc 525: Principles of Computer Networking)

In the last few years, there have been proposals for an Integrated Services Internet to support, among other things, real time multimedia applications. A number of different protocols and mechanisms have been invented to provide Integrated Services, including the Resource Reservation Protocol (RSVP) and Weighted Fair Queuing (WFQ). Please answer the following questions about these mechanisms:

- (a) How does RSVP reserve resources? What are the two basic kinds of messages used in RSVP and how are they used to reserve resources? Is it the receiver or the sender that reserves resources in RSVP and how? Contrast RSVP's method of reserving resources with ATM's (Asynchronous Transfer Mode)
- (b) What is Weighted Fair Queuing (WFQ)? What problems can WFQ solve? How is WFQ different from first-in-first-out (FIFO) queuing? How is bandwidth in a router divided using WFQ? How can WFQ achieve both fairness and efficiency?

Solution

- (a) RSVP reserves resources by the sender sending messages indicating the amount of resources needed by the sending/receiving application through the Internet. When a receiver receives one of these messages from the sender and wishes to make a reservation for the resources, the receiver sends messages upstream toward the source. Where the sending messages and receiving messages cross in the routers, reservations are made in the direction of the receiver. RSVP uses soft-state, so the sender continues to send messages periodically as well as the receiver. When no messages appear at the router for a given reservation for a period of time, the state for the reservation disappears. The messages the sender sends are called PATH messages, and the messages the receiver sends are called Reservation or RESV messages. In RSVP, it is the receiver that actually reserves the resources. This is in contrast to ATM, for example, where the sender reserves the resources on behalf of itself and the receivers. Also, ATM is different than RSVP in that ATM establishes hard-state in the ATM switches to reserve resources on behalf of its virtual circuit connections.
- (b) Weighted Fair Queuing (WFQ) is based on Fair Queuing (FQ), which is a queuing discipline in a router or switch meant to replace the commonly used FIFO queuing. In FQ, each flow or traffic class is given a separate queue. The router then services these queues in a round-robin manner. When a flow sends packets too quickly, then its queue fills up. When a queue reaches a particular length, additional packets belonging to that queue are discarded. WFQ adds a policy or quality of service component to FQ by assigning a weight to each queue indicating the priority of the traffic in that queue. The traffic is then forwarded according to these weights so that, e.g., for every bit that is transmitted in a queue with a weight of 1, two bits are transmitted in a queue with a weight of 2, 3 in 3, and so forth. Fairness is achieved in WFQ in that each flow or traffic class gets service in proportion to the weight determined by the policy installed in the router. WFQ is efficient because if, e.g., the highest weighted queue has no packets in it, then the other queues are serviced splitting up the remaining amount of router resources relative to the weights given to the queues that do hold packets. Thus, no bandwidth need be wasted because a queue for a flow or class of traffic happens to be empty.

9 Applications 3 (CSc 533: Computer Graphics)

Describe a linear transformation, presented as the multiplications of matrices, that transform the square whose vertices are in

$(0, 0), (0, 1), (1, 1), (1, 0)$, into the parallelogram whose vertices are in

$(5, 5), (5, 8), (7, 12), (7, 15)$. Write the matrix of each elementary transformation you use separately, and mention the effect of each of them.

Solution

We use homogeneous coordination. Let T be the corresponding matrix. Then $T = T_{translate} \cdot T_{sheer} \cdot T_{scale}$ where T_{scale} is a scaling transform, where T_{sheer} is a scaling transform, and $T_{translate}$ is a translation transform. Here

$$T_{scale} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad T_{sheer} = \begin{pmatrix} 1 & 0 & 0 \\ 7/2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad T_{translate} = \begin{pmatrix} 1 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{pmatrix}$$

Here T_{scale} brings the unit square into a rectangle R whose lower left corner is still on the origin, its bottom and left edge are on the x and y axes respectively, its width is 2 and its height is 3. The matrix T_{sheer} leaves the lower left corner of R unchanged, and brings the lower right corner to $(2, 7)$. Finally $T_{translate}$ shifts the new shape so its lower left corner is in $(5, 5)$.

10 Applications 4 (CSc 560: Database Systems)

Consider a database of a large fact table F (with tens of millions of tuples) and three relatively small dimension tables D_1, D_2 and D_3 (each with no more than ten thousand tuples). Assume that the database has been designed following the *star schema*. Suggest at least three different ways of processing the following 4-way join query, and discuss the processing costs of the suggested approaches. Your answer must include one by the *star join* algorithm.

```
SELECT *
FROM F D1 D2 D3
WHERE F.d1 = D1.d1 AND F.d2 = D2.d2 AND F.d3 = D3.d3
      AND p1(D1) AND p2(D2) AND p3(D3)
```

In the above query, $F.d_i$ is a foreign key attribute and $D_i.d_i$ is the corresponding primary key attribute of D_i ; $p_i(D_i)$ is a selection predicate of D_i .

Solution

Answers may vary. Suggested answers are:

1. By a series of two-way join computations. That is,

$$((F \bowtie_{F.d_1=D_1.d_1} (\sigma_{p_1} D_1)) \bowtie_{F.d_2=D_2.d_2} (\sigma_{p_2} D_2)) \bowtie_{F.d_3=D_3.d_3} (\sigma_{p_3} D_3)$$

This approach may produce a large intermediate result (as large as the fact table or even larger than that), which can in turn incur a large amount disk accesses.

2. By a Cartesian product of dimension tables followed by a join. That is,

$$((\sigma_{p_1} D_1) \times (\sigma_{p_2} D_2) \times (\sigma_{p_3} D_3)) \bowtie_{F.d_1=D_1.d_1 \wedge F.d_2=D_2.d_2 \wedge F.d_3=D_3.d_3} F$$

No intermediate result is produced, but the result of the Cartesian product may be very large.

3. By the star join algorithm. That is,

```
Rowset ← F
for i = to 3 do
  Rowset ← Rowset ∩ (σpi Di) ⋈F.di=Di.di F
end
```

The set operation on the rowset can be efficiently implemented by a bit vector whose size is equal to the cardinality of F .