Graduate Examination

New Format

Department of Computer Science
The University of Arizona
Spring 2003

March 7, 2003

Instructions

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

1. Theory: CSc 545, 573, and 520 (and, for Spring 2003 only, 537);
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 *course number* for the problem 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)

Construct a program that prints out "`reference`" if an implementation passes `in out` parameters by reference, and "`value-result`" if it passes them by value-result.

(You may use a generic C-like syntax, as long as you indicate clearly, for each parameter of each procedure, whether it is an `in` parameter, `out` parameter, or `in out` parameter.)

Solution

To determine whether `in out` parameters are passed by reference or by value-result, we need to have an alias to the actual argument. Then, if the formal parameter is modified within the callee, the actual will be modified in the case of reference parameters but not in the case of value-result parameters. Whether or not the actual has been modified in this way can then be checked using the alias. E.g.:

```
int a;
procedure foo(int x:  in out)
{
   x := x+1;
   if (x == a)
      print "reference";
   else
      print "value-result";
}

procedure main()
{
   a := 0;
   foo(a);
}
```

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

You are given a list of people $P = \{p_1 \ldots p_n\}$ and a list of houses $H = \{h_1 \ldots h_m\}$. You have to place the largest amount of people into the houses. That is, the output of the algorithm is a list of pairs, where each pair (p_i, h_j) implies that the algorithm decided to locate person p_i in house h_j.

Each house h_j has a limit u_j of how many people can accumulate in h_j (for $i = j \ldots m$). In addition, with each person p_i we are given a list L_i specifying the houses she or he is willing to be located at. For example, $L_5 = \{h_{10}, h_{13}, h_{20}\}$ implies that person p_5 is willing to live only in the houses h_{10}, h_{13}, h_{20}, and none of the others.

Describe an algorithm that finds the maximum number of pairs (p_i, h_j) such that no house contains more people than it can hold, and no person is located in a house she or he does not like. Your algorithm should be as efficient as possible. What is its running time ?

Solution
We form the following flow problem on the graph (G, E), where $V = \{s \cup P \cup H \cup t\}$, and E is as follows:

- Every $p_i \in P$ is connected to s with an edge of capacity 1.
- There is an edge between p_i and h_j if and only p_i is willing to live in h_j. The capacity is again 1.
- Every h_j is connected to t, with capacity u_i.

3 Theory 3 (CSc 573: Theory of Computation)

Let $G(V, E)$ be an undirected graph. Is there a polynomial time algorithm, that finds a sequence $v_1, v_2, \ldots v_n$ of the vertices (every vertex of V appears exactly once in the sequence) such that no edge connects v_i to v_{i+1} ?

Solution
The problem is NP-hard, since it is equivalent to find Hamiltonian path in the graph $G(V, E')$ where (u, v) is an edge of E if and only if (u, v) is not an edge of E. The problem then is to find a Hamiltonian path in $G(V, E')$.

4 Theory 4 (CSc 537: Computational Geometry)

(a) Given a set S of n line segments in the plane, find all intersection points between pairs of segments in time $O((n+k)\log n)$ where k is the number of intersection points.

(b) Define an x-monotone path to be a path with the property that every vertical line intersects it in at most one point. Let $\{\pi_1, \pi_2, \ldots \pi_r\}$ denote a set of r *x-monotone* polygonal paths, having n segments together. Describe an algorithm that finds all intersection points between pairs of segments.

$(b.i)$ Give an $O(n \log n)$ time-algorithm for the case that $r = 2$, i.e. only 2 paths. (Hint: Start by giving a bound on the number of intersection points.)

$(b.ii)$ Given an $O(nr^2 \log n)$ time-algorithn, for the case that r is arbitrary. Bonus — can you provide an $O(nr \log n)$-time algorithm ?

Solution

The solution to all parts use standard line-sweep algorithms. The different running times results from different bounds on the number of intersections.

For part $(b.i)$ we just need the following observation: Consider the line-sweep process (executed in this case from left to right). Define an *intersection event* to be the event of intersection between a segment of π_1 and a segment of π_2. Observe that during the sweep, between two intersection events, the line sweep must reach an endpoint of a segment of either π_1 or π_2. Hence the number of intersection points is bounded by the number of endpoints of segments of π_1 and π_2, i.e. $\leq n$.

In part $(b.ii)$ the bound on the number of intersection points is obtained by summing the bound on intersection points between two paths, for all $\binom{r}{2}$ pairs of paths.

The bonus part is obtained by arguing that if the path π_i has n_i segments, and

$$\sum_{1}^{r} n_i = n$$

then the number of intersection points between all paths is

$$\sum_{1 \leq i < j \leq r} (n_i + n_j) \leq \sum_{1 \leq i,j \leq r} (n_i + n_j) \leq 2r \sum_{i=1}^{r} n_i = O(nr)$$

5 Systems 1 (CSc 552: Advanced Operating Systems)

The UNIX `fork()` system call creates a new process that is an identical copy of the calling process. The new process can share the original process's text segment (because code is read-only), but it must have its own copies of the heap and stack segments. Copying the heap and stack segments can consume most of the time in `fork()`. A common technique for improving `fork()` performance is to use copy-on-write to share the pages of the heap and stack segments.

(a) What is copy-on-write, and how can it improve copy performance?

(b) Outline the actions taken by the operating system to implement copy-on-write for a page, assuming that both processes eventually write the page. What changes, if any, need to be made to the operating system's virtual memory data structures to support copy-on-write?

(c) Under what circumstances can copy-on-write have worse performance than simply copying a page during `fork()`?

(d) Why does copy-on-write work well for UNIX `fork()`?

Solution

(a) Copy-on-write copies pages in a lazy fashion. Initially, processes share the same physical page, but when one of them tries to write the page they are given their own copies. Copy-on-write avoids copying pages that are never modified.

(b) During `fork()` the operating system creates a new page table for the new process, and copies the entries for the copy-on-write pages from the original process's page table. The entries for each copy-on-write page are marked as read-only and copy-on-write. When a process tries to write the page, it traps into the kernel because the page is marked read-only. The page is marked copy-on-write, so the operating system copies the page into a free page, points one of the page tables to the new page, and removes the read-only and copy-on-write indications from both page tables. The process is then allowed to continue.

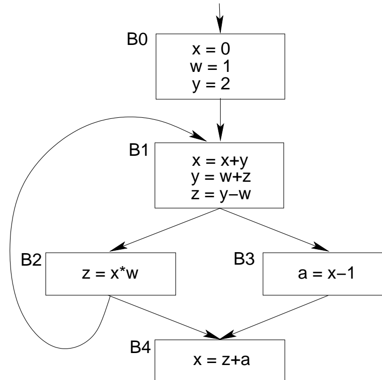
The page tables must be modified to include a copy-on-write indication. Also, a single page can now be shared by multiple heap and stack segments, so it may be necessary to modify data structures so that a shared page in these segments is not freed until all processes sharing the page exit.

(c) Copy-on-write only improves performance if some of the pages are never written. Copying a page via copy-on-write is more expensive than simply copying it during `fork()` because of the overhead of marking the page in both page tables, handling the trap, and unmarking the pages. If a page will eventually be written, it is better to make the copy in `fork()`.

(d) A new UNIX process created via `fork()` typically calls `exec()` within a short time. `exec()` replaces the process's segments with those from an executable file, thus few of the copy-on-write pages are modified by the new process.

6 Systems 2 (CSc 553: Principles of Compilation)

Consider the following control flow graph. You may assume that the only variable live at the end of the exit node B4 is x.



(*a*) [5 points] Complete the following table by filling in information about what variables are live at the entry to, and exit from, each basic block:

Basic Block B	IN(B)	OUT(B)
B0		
B1		
B2		
B3		
B4		

(*b*) [5 points] Briefly explain the concept of a register interference graph and its role in register allocation.

Solution

(*a*)

Basic Block B	IN(B)	OUT(B)
B0	z, a	x, y, z, w, a
B1	x, y, z, w, a	w, x, z, a
B2	x, y, w, a	x, y, z, w, a
B3	x	z, a
B4	z, a	x

(*b*) A register interference graph is a graph where each vertex represents a live range in a program (a set of basic blocks or instructions over which a particular variable is live), and where there is an edge between two vertices a and b if the corresponding live ranges overlap.

The register interference is used in register allocation by graph coloring. The basic idea is that given that we have n registers available for allocation, we try to find an n-coloring of the interference graph. If this is deemed to not be possible, we simplify the graph by deleting some nodes (with the corresponding variables being spilled to memory) and attempt to n-color the resulting graph, and so on.

7 Systems 3 (CSc 576: Computer Architecture)

Provide brief answers to the following questions:

(a) What problems arise when virtually addressed caches are used? Give possible solutions for handling these problems.

(b) Describe a cache design that reduces the cost (amount) of tag storage without increasing cache miss penalty.

(c) How are spurious exceptions caused by speculatively executed instructions suppressed in a VLIW machine?

(d) How is speculative execution of instructions achieved in a dynamic issue processor vs. a VLIW processor?

Solution

(a) Every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. The need to flush the cache can be avoided by associating process-identifier tag with each cache block. Programs may use two different virtual addresses for the same physical address. If both addresses (aliases) are cached, and one of them is modified, the other will have the wrong value. One solution to this problem is page coloring that forces aliases to share some address bits so that they cannot be simultaneously present in the cache.

(b) For a given cache size, larger block sizes can be used to reduce the number of blocks and hence the number of tags. Large block size leads to large cache miss penalty. This can be reduced by using sub-block placement. Only one sub-block is read on a miss.

(c) Two versions of each instruction are supported: speculative and nonspeculative. When speculative version of an instruction causes an exception, an exception flag associated with the result register is set. This flag is propagated to result registers of other speculative instructions which use the result of the exception causing instruction. If a nonspeculative instruction reads a register whose exception flag is set, exception is raised. If no such read ever takes place, the exception is effectively suppressed.

(d) Dynamic issue machine: through a combination of branch prediction and out-of-order execution, an instruction following a predicted branch may be speculatively executed before the branch prediction is verified. VLIW: during instruction scheduling, the compiler can perform code reordering that can cause conditionally executed instructions to be executed unconditionally (i.e., speculatively).

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

The UNIX kernel contains two primitives that are similar to the `sleep()` and `wakeup()` primitives defined by the following monitor:

```
monitor SleepWakeup {
   condition timeToGo;

   void sleep() {
      wait(timeToGo);          # go to sleep
   }

   void wakeup() {
      signal_all(timeToGo);    # awaken all sleeping processes
   }
}
```

This code assumes that signals use the Signal-and-Continue discipline (i.e., that the signaler executes next in the monitor).

Develop an implementation of `sleep()` and `wakeup()` that uses semaphores for synchronization instead of monitors. Declare and initialize any shared variables that you need, such as counters and semaphores. Include comments in your code to explain how it works.

Solution

```
int waiting = 0;
sem mutex = 1, timeToGo = 0;

void sleep() {
   P(mutex);                 # enter critical section
   waiting++;                # indicate that are about to go to sleep
   V(mutex);                 # exit critical section
   P(timeToGo);              # go to sleep
   if (waiting > 0) {        # awaken next sleeping process if there is one
      waiting--;
      V(timeToGo);
   }
   else                      # otherwise exit critical section
      V(mutex);
}

void wakeup() {
   P(mutex);                 # enter critical section
   if (waiting > 0) {        # if some process is sleeping, awaken first one
      waiting--;
      V(timeToGo);
   }
   else                      # otherwise exit critical section
      V(mutex);
}
```

Grading note : It is *incorrect* to omit the **if**/**else** statement at the end of `sleep()` and to have the wakeup code do something like

```
while (waiting > 0) {
   waiting--;
   V(timeToGo);
   }
   V(mutex);
```

This is a *very* common mistake. It looks lots simpler, but it is wrong. The problem is that there is no guarantee that a process awakened by `V(timeToGo)` will in fact execute before some process that might call `sleep()` — after the wakeup finishes and hence `mutex` == 1. At this point, the new call of `sleep()` can get to the `P(timeToGo)` and succeed, when it should have to delay. In short, a signal meant for one process can be seen by another.

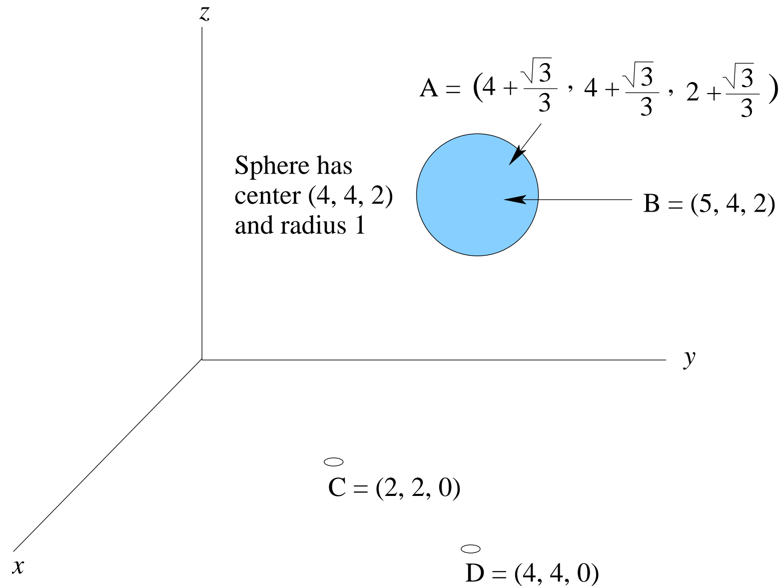
9 Applications 2 (CSc 525: Principles of Computer Networking)

Describe the slow-start algorithm in TCP and say what it is used for. What is the relationship between the congestion window and the advertized window in TCP? What role does the TCP round-trip timer play in slow-start? How do the fast-retransmit and fast recovery mechanisms work? What optimizations do these latter two provide?

Solution
The slow-start algorithm in TCP attempts to control congestion in the network by creating a new variable for TCP implementations called the ”congestion window.” When an acknowledgement for a TCP segment is not received by the time TCP's round-trip timer expires, TCP assumes that there is congestion in the network and reduces the congestion window to zero. TCP sends at the minimum of the congestion and advertized windows and thus completely shuts down its sending rate. The sending rate starts slowly by sending exponentially increasing number of packets (starting at one) for every acknowledgement received until the rate equals half of the rate where congestion last occured, and thereafter increases the congestion window linearly, probing for the next congestion point. With fast retransmit, the sending TCP, upon receiving three duplicate acks, retransmits the lost packet in the sequence. Fast recovery adds to this scheme by allowing the TCP sender, upon receiving the three duplicate acks, to reduce the sending rate by only half, instead of closing down completely as in the original slow-start algorithm.

10 Applications 3 (CSc 533: Computer Graphics)

Consider the scene shown below with an infinite ground plane surface with color $(200, 200, 200)$, and the sphere drawn with color $(200, 0, 0)$. Both surfaces are diffuse (Lambertian) reflectors. (Recall that this means that their brightness is independent of viewing direction). There is exactly one point source light at infinity, with strength $(1, 1, 1)$ in the direction $(1, 1, 1)$. (The light is in the positive octant). The projection plane is $(x = 10)$ and the point of projection (eye) is $(20, 4, 2)$.



For parts (a) and (b) you can ignore light reflected from one diffuse surface onto another if you think it is an issue. Note that parts (a), (b), (c) are relatively independent of rendering strategy.

[Recall that for the "naïve" color model, we compute pixel RGB values by multiplying light vectors and surface color vectors element-wise (and similarly for specular reflection magnitudes). For convenience, we assume an 8-bit display (so reasonable values for display tend to be in the range [0, 255]), and we set the color of a perfect diffuse white at (255, 255, 255). A direct (i.e., perpendicular) light which makes that surface have RGB (255, 255, 255) is therefore (1, 1, 1). The magnitude of a specularity is handled analogous to surfaces.]

(a) Estimate the (R,G,B) for each of the four points A, B, C, D shown, assuming the "naïve" color model.

(b) Suppose that you model the sphere as a collection of polygons, and you want any specularities present to be accurately rendered. Explain why it is a good idea to use Phong shading as opposed to Gouraud shading or standard naïve shading.

(c) Suppose that you wanted to capture the effect of any reflections between diffuse surfaces that existed. What is a possible rendering method?

Solution

(a) A and B are $(200, 0, 0)\times$ dot_product(surface_normal, light_vector_normal). The sphere is a unit sphere, so we get the surface normal by simply subtracting the center from the point in question. The light vector direction needs to be normalized by dividing it by $\sqrt{3}$. C is in shadow, so ignoring inter-reflection, it gets no light $(0, 0, 0)$. The surface normal at D is $(0, 0, 1)$. D is $(200, 200, 200)\times$ dot_product(surface_normal, light_vector_normal).

A : $(200, 0, 0)$
B : $(200, 0, 0)/\sqrt{3}$
C : $(0, 0, 0)$
D : $(200, 200, 200)/\sqrt{3}$

(b) Because specularities are sensitive to the normal, and very localized, the plane approximating the piece of the sphere containing the specularity is not likely to have the correct normal for the specularity to be correct or even nearly correct, unless a very large number of polygons is used. Thus it is better to interpolate the normal across the plane (Phong shading), so that at the relevant point, the approximation of the normal would be close enough to the real normal to reveal the specularity. Gouraud shading, which just interpolates the *results* of applying shading, does not address this issue, and thus does not help much.

(c) To get the inter-reflections from diffuse surfaces one can use Radiosity. Neither standard backwards ray-tracing (or even forward ray-tracing), nor standard rendering through polygon shading, will get these effects.

11 Applications 4 (CSc 560: Database Systems)

Answer the following questions about crash recovery.

1. How do buffer management policies affect REDO and UNDO actions during normal operations and crash recovery?
2. After a system crash, the ARIES recovery manager restarts the system in three passes: Analysis, REDO and UNDO. Briefly describe what is done during the Analysis pass, and explain how the scopes of the REDO and UNDO recoveries are defined.

Solution

1. The FORCE policy enforces propagation of all modified pages at EOT. No logging for REDO nor REDO recovery is required under this policy. The STEAL policy allows modified pages to be propagated at any time. Logging for UNDO and UNDO recovery are required under this policy.
2. Scanning forward the log records from the most recent checkpoint, it reconstructs the Transaction Table and Dirty Page Table at the time of the system crash. The scope of REDO recovery is bounded by the earliest one of all log records (of any transactions) that have made a page dirty. Without checkpointing, the scope of REDO recovery could be unbounded at the presence of a hot page updated by many transactions for a long duration. The scope of UNDO recovery is bounded by the oldest log record of transactions that were active at the time of crash.