Principles of Computer System Design

An Introduction

Problem Sets

Jerome H. Saltzer

M. Frans Kaashoek

Massachusetts Institute of Technology

Version 5.0

Suggestions, Comments, Corrections, and Requests to waive license restrictions: Please send correspondence by electronic mail to:

Saltzer@mit.edu

and

kaashoek@mit.edu

Problem Sets

TABLE OF CONTENTS

Introduction

These problem sets seek to make the student think carefully about how to apply the concepts of the text to new problems. These problems are derived from examinations given over the years while teaching the material in this textbook. Many of the problems are multiple choice with several right answers. The reader should try to identify all right options.

Some significant and interesting system concepts that are not mentioned in the main text, and therefore at first read seem to be missing from the book, are actually to be found within the exercises and problem sets. Definitions and discussion of these concepts can be found in the text of the exercise or problem set in which they appear. Here is a list of concepts that the exercises and problem sets introduce:

- *action graph* (problem set *36*)
- *ad hoc wireless network* (problem sets *19* and *21*)
- *bang-bang protocol* (exercise 7.13)
- *blast protocol* (exercise 7.25)
- *commutative cryptographic transformation* (exercise 11.4)
- *condition variable* (problem set *13*)
- *consistent hashing* (problem set *23*)
- *convergent encryption* (problem set *48*)
- *cookie* (problem set *45*)
- *delayed authentication* (exercise 11.10)
- *delegation forwarding* (exercise 2.1)
- *event variable* (problem set *11*)
- *fast start* (exercise 7.12)
- *flooding* (problem set *20*)
- *follow-me forwarding* (exercise 2.1)
- *Information Management System atomicity* (exercise 9.5)
- *mobile host* (exercise 7.24)
- *lightweight remote procedure call* (problem set *7*)
- *multiple register set processor* (problem set *9*)

- *object-oriented virtual memory* (problem set *15*)
- *overlay network* (problem set *20*)
- *pacing* (exercise 7.16)
- *peer-to-peer network* (problem set *20*)
- *RAID 5,* with rotating parity (exercise 8.10)
- *restartable atomic region* (problem set *9*)
- *self-describing storage* (exercise 6.8)
- *serializability* (problem set *36*)
- *timed capability* (exercise 11.8)

Exercises for Chapter 7 and above are in on-line chapters, and problem sets numbered 17 and higher are in the on-line book of problem sets.

Some of these problem sets span the topics of several different chapters. A parenthetical note at the beginning of each set indicates the primary chapters that it involves. Following each exercise or problem set question is an identifier of the form "1978–3–14". This identifier reports the year, examination number, and problem number of the examination in which some version of that problem first appeared. For those problem sets not developed by one of the authors, a credit line appears in a footnote on the first page of the problem set.

1 Bigger Files*

(Chapter 2)

For his many past sins on previous exams, Ben Bitdiddle is assigned to spend eternity maintaining a PDP-11 running version 7 of the UNIX operating system. Recently, one of his user's database applications failed after reaching the file size limit of 1,082,201,088 bytes (approximately 1 gigabyte). In an effort to solve the problem, he upgraded the computer with an old 4-gigabyte (2^{32} byte) drive; the disk controller hardware supports 32-bit sector addresses, and can address disks up to 2 terabytes in size. Unfortunately, Ben is disappointed to find the file size limit unchanged after installing the new disk.

In this question, the term *block number* refers to the block pointers stored in inodes. There are 512 bytes in a block. In addition, Ben's version 7 UNIX system has a file system that has been expanded from the one described in Section 2.5: its inodes are designed to support larger disks. Each inode contains 13 block numbers of 4 bytes each; the first 10 block numbers point to the first 10 blocks of the file, and the remaining 3 are used for the rest of the file. The 11th block number points to an indirect block, containing 128 block numbers, the 12th block number points to a double-indirect block, containing 128 indirect block numbers, and the 13th block number points to a triple-indirect block, containing 128 double-indirect block numbers. Finally, the inode contains a four-byte file size field.

Q 1.1 Which of the following adjustments will allow files larger than the current one gigabyte limit to be stored?

- **A.** Increase just the file size field in the inode from a 32-bit to a 64-bit value.
- **B.** Increase just the number of bytes per block from 512 to 2048 bytes.
- **C.** Reformat the disk to increase the number of inodes allocated in the inode table.
- **D.** Replace one of the direct block numbers in each inode with an additional triple-indirect block number.

2008–1–5

Ben observes that there are 52 bytes allocated to block numbers in each inode (13 block numbers at 4 bytes each), and 512 bytes allocated to block numbers in each indirect block (128 block numbers at 4 bytes each). He figures that he can keep the total space allocated to block numbers the same, but change the size of each block number, to increase the maximum supported file size. While the number of block numbers in inodes and indirect blocks will change, Ben keeps exactly one indirect, one double-indirect and one triple-indirect block number in each inode.

* Credit for developing this problem set goes to Lewis D. Girod.

Q 1.2 Which of the following adjustments (without any of the modifications in the previous question), will allow files larger than the current approximately 1 gigabyte limit to be stored?

A. Increasing the size of a block number from 4 bytes to 5 bytes.
B. Decreasing the size of a block number from 4 bytes to 3 bytes.
C. Decreasing the size of a block number from 4 bytes to 2 bytes.

2008–1–6

2 Ben's Stickr*

(Chapter 4)

Ben is in charge of system design for Stickr, a new Web site for posting pictures of bumper stickers and tagging them. Luckily for him, Alyssa had recently implemented a Triplet Storage System (TSS), which stores and retrieves arbitrary triples of the form {*subject, relationship, object*} according to the following specification:

```
procedure FIND (subject, relationship, object, start, count)
    // returns OK + array of matching triples

procedure INSERT (subject, relationship, object)
    // adds the triple to the TSS if it is not already there and returns OK

procedure DELETE (subject, relationship, object)
    // removes the triple if it exists, returning TRUE, FALSE otherwise
```

Ben comes up with the following design:

As shown in the figure, Ben uses an RPC interface to allow the Web server to interact with the triplet storage system. Ben chooses *at-least-once* RPC semantics. Assume that the triplet storage system never crashes, but the network between the Web server and triplet storage system is unreliable and may drop messages.

Q 2.1 Suppose that only a single thread on Ben's Web server is using the triplet storage system and that this thread issues just one RPC at a time. What types of incorrect behavior can the Web server observe?

A. The FIND RPC stub on the Web server sometimes returns no results, even though matching triples exist in the triplet storage system.
B. The INSERT RPC stub on the Web server sometimes returns OK without inserting the triple into the storage system.
C. The DELETE RPC stub on the Web server sometimes returns FALSE when it actually deleted a triple.
D. The FIND RPC stub on the Web server sometimes returns triples that have been deleted.

Q 2.2 Suppose Ben switches to *at-most-once* RPC; if no reply is received after some time, the RPC stub on the Web server gives up and returns a "timer expired" error code. Assume again that only a single thread on Ben's Web server is using the triple storage

* Credit for developing this problem set goes to Samuel R. Madden.

system and that this thread issues just one RPC at a time. What types of incorrect behavior can the Web server observe?

A. Assuming it does not time out, the FIND RPC stub on the Web server can sometimes return no results when matching triples exist in the storage system.

B. Assuming it does not time out, the INSERT RPC stub on the Web server can sometimes return OK without inserting the triple into the storage system.

C. Assuming it does not time out, the DELETE RPC stub on the Web server can sometimes return FALSE when it actually deleted a triple.

D. Assuming it does not time out, the FIND RPC stub on the Web server can sometimes return triples that have been deleted.

2007–1–5/6

3 Jill's File System for Dummies*

(Chapter 4)

Mystified by the complexity of NFS, Moon Microsystems guru Jill Boy decides to implement a simple alternative she calls File System for Dummies, or FSD. She implements FSD in two pieces:

1. An FSD server, implemented as a simple user application, which responds to FSD requests. Each request corresponds exactly to a UNIX file system call (e.g. READ, WRITE, OPEN, CLOSE, or CREATE) and returns just the information returned by that call (status, integer file descriptor, data, etc.).
2. An FSD client library, which can be linked together with various applications to substitute Jill's FSD implementations of file system calls like OPEN, READ, and WRITE for their UNIX counterparts. To avoid confusion, let's refer to Jill's FSD versions of these procedures as FSD_OPEN, and so on.

Jill's client library uses the standard UNIX calls to access local files but uses names of the form

```
/fsd/hostname/apath
```

to refer to the file whose absolute path name is /apath on the host named hostname. Her library procedures recognize operations involving remote files (e.g.

```
FSD_OPEN("/fsd/cse.pedantic.edu/foobar", READ_ONLY)
```

and translates them to RPC requests to the appropriate host, using the file name on that host (e.g.

```
RPC("/fsd/cse.pedantic.edu/foobar", "OPEN", "/foobar", READ_ONLY).
```

The RPC call causes the corresponding UNIX call, for example,

```
OPEN("/foobar", READ_ONLY)
```

to be executed on the remote host and the results (e.g., a file descriptor) to be returned as the result of the RPC call. Jill's server code catches errors in the processing of each request and returns ERROR from the RPC call on remote errors.

Figure PS.1 describes pseudocode for Version 1 of Jill's FSD client library. The RPC calls in the code relay simple RPC commands to the server, using *exactly-once* semantics. Note that no data caching is done by either the server or the client library.

* Credit for developing this problem set goes to Stephen A. Ward.

```
// Map FSD handles to host names, remote handles:
string handle_to_host_table[1000]                    // initialized to unused
integer handle_to_rhandle_table[1000]                // handle translation table

procedure FSD_OPEN (string name, integer mode)
    integer handle ← FIND_UNUSED_HANDLE ()
    if name begins with "/fsd/" then
        host ← EXTRACT_HOST_NAME (name)
        filename ← EXTRACT_REMOTE_FILENAME (name)   // returns remote file handle
        rhandle ← RPC (host, "OPEN", filename, mode)          // or ERROR
    else
        host ← ""
        rhandle ← OPEN (name, mode)
    if rhandle ← ERROR then return ERROR
    handle_to_rhandle_table[handle] ← rhandle
    handle_to_host_table[handle] ← host
    return handle

procedure FSD_READ (integer handle, string buffer, integer nbytes)
    host ← handle_to_host_table[handle]
    rhandle ← handle_to_rhandle_table[handle]
    if host = "" then return READ (rhandle, buffer, nbytes)
    // The following call sets "result" to the return value from
    // the read(...) on the remote host, and copies data read into buffer:
    result, buffer ← RPC (host, "READ", rhandle, nbytes)
    return result

procedure FSD_CLOSE (integer handle)
    host ← handle_to_host_table[handle]
    rhandle ← handle_to_rhandle_table[handle]
    handle_to_rhandle_table[handle] ← UNUSED
    if host = "" then return CLOSE (rhandle)
    else return RPC (host, "CLOSE", rhandle)
```

FIGURE PS.1

Pseudocode for FSD client library, Version 1.

Q 3.1 What does the above code indicate via an empty string ("") in an entry of handle to host table?

A. An unused entry of the table.
B. An open file on the client host machine.
C. An end-of-file condition on an open file.
D. An error condition.

Mini Malcode, an intern assigned to Jill, proposes that the above code be simplified by eliminating the *handle_to_rhandle_table* and simply returning the untranslated handles returned by OPEN on the remote or local machines. Mini implements her simplified client

library, making appropriate changes to each FSD call, and tries it on several test programs.

Q 3.2 Which of the following test programs will continue to work after Mini's simplification?

A. A program that reads a single, local file.
B. A program that reads a single remote file.
C. A program that reads and writes many local files.
D. A program that reads and writes several files from a single remote FSD server.
E. A program that reads many files from different remote FSD servers.
F. A program that reads several local files as well as several files from a single remote FSD server.

Jill rejects Mini's suggestions, insisting on the Version 1 code shown above. Marketing asks her for a comparison between FSD and NFS (see Section 4.5).

Q 3.3 Complete the following table comparing NFS to FSD by circling yes or no under each of NFS and FSD for each statement:

Statement	NFS	FSD
remote handles include inode numbers	Yes/No	Yes/No
read and write calls are idempotent	Yes/No	Yes/No
can continue reading an open file after deletion (e.g., by program on remote host)	Yes/No	Yes/No
requires mounting remote file systems prior to use	Yes/No	Yes/No

Convinced by Moon's networking experts that a much simpler RPC package promising *at-least-once* rather than *exactly-once* semantics will save money, Jill substitutes the simpler RPC framework and tries it out. Although the new (Version 2) FSD works most of the time, Jill finds that an FSD_READ sometimes returns the wrong data; she asks you to help. You trace the problem to multiple executions of a single RPC request by the server and are considering

- A response cache on the client, sufficient to detect identical requests and returning a cached result for duplicates without resending the request to the server.
- A response cache on the server, sufficient to detect identical requests and returning a cached result for duplicates without re-executing them.
- A monotonically increasing *sequence number* (nonce) added to each RPC request, making otherwise identical requests distinct.

Q 3.4 Which of the following changes would you suggest to address the problem introduced by the *at-least-once* RPC semantics?

A. Response cache on each client.
B. Response cache on server.
C. Sequence numbers in RPC requests.
D. Response cache on client AND sequence numbers.
E. Response cache on server AND sequence numbers.
F. Response caches on both client and server.

2007–2–7…10

4 EZ-Park*

(Chapter 5 in Chapter 4 setting)

Finding a parking spot at Pedantic University is as hard as it gets. Ben Bitdiddle, deciding that a little technology can help, sets about to design the EZ-Park client/server system. He gets a machine to run an EZ-Park server in his dorm room. He manages to convince Pedantic University parking to equip each car with a tiny computer running EZ-Park client software. EZ-Park clients communicate with the server using remote procedure calls (RPCs). A client makes requests to Ben's server both to find an available spot (when the car's driver is looking for one) and to relinquish a spot (when the car's driver is leaving a spot). A car driver uses a parking spot if, and only if, EZ-Park allocates it to him or her.

In Ben's initial design, the server software runs in one address space and spawns a new thread for each client request. The server has two procedures: FIND_SPOT () and RELINQUISH_SPOT (). Each of these threads is spawned in response to the corresponding RPC request sent by a client. The server threads use a shared array, *available*[], of size NSPOTS (the total number of parking spots). *available*[*j*] is set to TRUE if spot *j* is free, and FALSE otherwise; it is initialized to TRUE, and there are no cars parked to begin with. The NSPOTS parking spots are numbered from 0 through NSPOTS - 1. *numcars* is a global variable that counts the total number of cars parked; it is initialized to 0.

Ben implements the following pseudocode to run on the server. Each FIND_SPOT() thread enters a **while** loop that terminates only when the car is allocated a spot:

```
1  procedure FIND_SPOT ()                  // Called when a client car arrives
2      while TRUE do
3          for i ← 0 to NSPOTS do
4              if available[i] = TRUE then
5                  available[i] ← FALSE
6                  numcars ← numcars + 1
7                  return i                // Client gets spot i

8  procedure RELINQUISH_SPOT (spot)        // Called when a client car leaves
9      available[spot] ← TRUE
10     numcars ← numcars - 1
```

Ben's intended correct behavior for his server (the "correctness specification") is as follows:

- **A.** FIND_SPOT() allocates any given spot in [0, …, NSPOTS - 1] to at most one car at a time, even when cars are concurrently sending requests to the server requesting spots.
- **B.** *numcars* must correctly maintain the number of parked cars.
- **C.** If at any time (1) spots are available and no parked car ever leaves in the future, (2) there are no outstanding FIND_SPOT() requests, and (3) exactly one client makes a FIND_SPOT request, then the client should get a spot.

---

* Credit for developing this problem set goes to Hari Balakrishnan.

Ben runs the server and finds that when there are no concurrent requests, EZ-Park works correctly. However, when he deploys the system, he finds that sometimes multiple cars are assigned the same spot, leading to collisions! His system does not meet the correctness specification when there are concurrent requests.

Make the following assumptions:

1. The statements to update *numcars* are *not* atomic; each involves multiple instructions.
2. The server runs on a single processor with a preemptive thread scheduler.
3. The network delivers RPC messages reliably, and there are no network, server, or client failures.
4. Cars arrive and leave at random.
5. ACQUIRE and RELEASE are as defined in Chapter *5*.

Q 4.1 Which of these statements is true about the problems with Ben's design?

A. There is a race condition in accesses to *available*[], which may violate one of the correctness specifications when two FIND_SPOT() threads run.
B. There is a race condition in accesses to *available*[], which may violate correctness specification A when one FIND_SPOT() thread and one RELINQUISH_SPOT() thread runs.
C. There is a race condition in accesses to *numcars*, which may violate one of the correctness specifications when more than one thread updates *numcars*.
D. There is no race condition as long as the average time between client requests to find a spot is larger than the average processing delay for a request.

Ben enlists Alyssa's help to fix the problem with his server, and she tells him that he needs to set some locks. She suggests adding calls to ACQUIRE and RELEASE as follows:

```
1   procedure FIND_SPOT ()                  // Called when a client car wants a spot
2       while TRUE do
!→          ACQUIRE (avail_lock)
3           for i ← 0 to NSPOTS do
4               if available[i] = TRUE then
5                   available[i] ← FALSE
6                   numcars ← numcars + 1
!→                  RELEASE (avail_lock)
7                   return i                // Allocate spot i to this client
!→          RELEASE (avail_lock)

8   procedure RELINQUISH_SPOT (spot)        // Called when a client car is leaving spot
!→      ACQUIRE (avail_lock)
9       available[spot] ← TRUE
10      numcars ← numcars - 1
!→      RELEASE (avail_lock)
```

Q 4.2 Does Alyssa's code solve the problem? Why or why not?

Q 4.3 Ben can't see any good reason for the RELEASE (*avail_lock*) that Alyssa placed after line 7, so he removes it. Does the program still meet its specifications? Why or why not?

Hoping to reduce competition for *avail_lock*, Ben rewrites the program as follows:

```
1   procedure FIND_SPOT ()              // Called when a client car wants a spot
2       while TRUE do
3           for i ← 0 to NSPOTS do
!→          ACQUIRE (avail_lock)
4               if available[i] = TRUE then
5                   available[i] ← FALSE
6                   numcars ← numcars + 1
!→                  RELEASE (avail_lock)
7                   return i                    // Allocate spot i to this client
!→              else RELEASE (avail_lock)

8   procedure RELINQUISH_SPOT (spot)    // Called when a client car is leaving spot
!→      ACQUIRE (avail_lock)
9       available[spot] ← TRUE
10      numcars ← numcars - 1
!→      RELEASE (avail_lock)
```

Q 4.4 Does that program meet the specifications?

Now that Ben feels he understands locks better, he tries one more time, hoping that by shortening the code he can really speed things up:

```
1   procedure FIND_SPOT ()              // Called when a client car wants a spot
2       while TRUE do
!→          ACQUIRE (avail_lock)
3           for i ← 0 to NSPOTS do
4               if available[i] = TRUE then
5                   available[i] ← FALSE
6                   numcars ← numcars + 1
7                   return i                    // Allocate spot i to this client

8   procedure RELINQUISH_SPOT (spot)    // Called when a client car is leaving spot
9       available[spot] ← TRUE
10      numcars ← numcars - 1
!→      RELEASE (avail_lock)
```

Q 4.5 Does Ben's slimmed-down program meet the specifications?

Ben now decides to combat parking at a truly crowded location: Pedantic's stadium, where there are always cars looking for spots! He updates NSPOTS and deploys the system during the first home game of the football season. Many clients complain that his server is slow or unresponsive.

Q 4.6 If a client invokes the FIND_SPOT() RPC when the parking lot is full, how quickly will it get a response, assuming that multiple cars may be making requests?

A. The client will not get a response until at least one car relinquishes a spot.
B. The client may never get a response even when other cars relinquish their spots.

Alyssa tells Ben to add a client-side timer to his RPC system that expires if the server does not respond within 4 seconds. Upon a timer expiration, the car's driver may retry the request, or instead choose to leave the stadium to watch the game on TV. Alyssa warns Ben that this change may cause the system to violate the correctness specification.

Q 4.7 When Ben adds the timer to his client, he finds some surprises. Which of the following statements is true of Ben's implementation?

A. The server may be running multiple active threads on behalf of the same client car at any given time.
B. The server may assign the same spot to two cars making requests.
C. *numcars* may be smaller than the actual number of cars parked in the parking lot.
D. *numcars* may be larger than the actual number of cars parked in the parking lot.

Q 4.8 Alyssa thinks that the operating system running Ben's server may be spending a fair amount of time switching between threads when many RPC requests are being processed concurrently. Which of these statements about the work required to perform the switch is correct? Notation: PC = program counter; SP = stack pointer; PMAR = page-map address register. Assume that the operating system behaves according to the description in Chapter *5*.

A. On any thread switch, the operating system saves the values of the PMAR, PC, SP, and several registers.
B. On any thread switch, the operating system saves the values of the PC, SP, and several registers.
C. On any thread switch between two RELINQUISH_SPOT() threads, the operating system saves *only* the value of the PC, since RELINQUISH_SPOT() has no return value.
D. The number of instructions required to switch from one thread to another is proportional to the number of bytes currently on the thread's stack.

5 Goomble*

(Chapter 5)

Observing that US legal restrictions have curtailed the booming on-line gambling industry, a group of laid-off programmers has launched a new venture called Goomble. Goomble's Web server allows customers to establish an account, deposit funds using a credit card, and then play the Goomble game by clicking a button labeled **I FEEL LUCKY**. Every such button click debits their account by $1, until it reaches zero.

Goomble lawyers have successfully defended their game against legal challenges by arguing that there's no gambling involved: the Goomble "service" is entirely deterministic.

The initial implementation of the Goomble server uses a single thread, which causes all customer requests to be executed in some serial order. Each click on the **I FEEL LUCKY** button results in a procedure call to LUCKY (*account*), where *account* refers to a data structure representing the user's Goomble account. Among other data, the account structure includes an unsigned 32-bit integer *balance*, representing the customer's current balance in dollars.

The LUCKY procedure is coded as follows:

```
1  procedure LUCKY (account)
2      if account.balance > 0 then
3          account.balance ← account.balance - 1
```

The Goomble software quality control expert, Nellie Nervous, inspects the single-threaded Goomble server code to check for race conditions.

Q 5.1 Should Nellie find any potential race conditions? Why or why not?

2007-1-8

The success of the Goomble site quickly swamps their single-threaded server, limiting Goomble's profits. Goomble hires a server performance expert, Threads Galore, to improve server throughput.

Threads modifies the server as follows: Each **I FEEL LUCKY** click request spawns a new thread, which calls LUCKY (*account*) and then exits. All other requests (e.g., setting up an account, depositing, etc.) are served by a single thread. Threads argues that the bulk of the server traffic consists of player's clicking **I FEEL LUCKY**, so that his solution addresses the main performance problem.

Unfortunately, Nellie doesn't have time to inspect the multithreaded version of the server. She is busy with development of a follow-on product: the Goomba, which simultaneously cleans out your bank account and washes your kitchen floor.

---

* Credit for developing this problem set goes to Stephen A. Ward.

Q 5.2 Suppose Nellie had inspected Goomble's multithreaded server. Should she have found any potential race conditions? Why or why not?

2007-1-9

Willie Windfall, a compulsive Goomble player, has two computers and plays Goomble simultaneously on both (using the same Goomble account). He has mortgaged his house, depleted his retirement fund and the money saved for his kid's education, and his Goomble account is nearly at zero. One morning, clicking furiously on **I FEEL LUCKY** buttons on both screens, he notices that his Goomble balance has jumped to something over four billion dollars.

Q 5.3 Explain a possible source of Willie's good fortune. Give a simple scenario involving two threads, T1 and T2, with interleaved execution of lines 2 and 3 in calls to LUCKY (*account*), detailing the timing that could result in a huge *account.balance*. The first step of the scenario is already filled in; fill as many subsequent steps as needed.

1. T1 evaluates "**if** *account.balance* > 0", finds statement is true
2.
3.
4.

2007-1-10

Word of Willie's big win spreads rapidly, and Goomble billionaires proliferate. In a state of panic, the Goomble board calls you in as a consultant to review three possible fixes to the server code to prevent further "gifts" to Goomble customers. Each of the following three proposals involves adding a lock (either global or specific to an account) to rule out the unfortunate race:

Proposal 1

```
procedure LUCKY (account)
    ACQUIRE (global_lock)
    if account.balance > 0 then
        account.balance ← account.balance - 1
    RELEASE (global_lock)
```

Proposal 2

```
procedure LUCKY (account)
    ACQUIRE (account.lock)
    temp ← account.balance
    RELEASE (account.lock)
    if temp > 0 then
        ACQUIRE (account.lock)
        account.balance ← account.balance - 1
        RELEASE (account.lock)
```

Proposal 3

```
procedure LUCKY (account)
    ACQUIRE (account.lock)
    if account.balance > 0 then
        account.balance ← account.balance - 1
    RELEASE (account.lock)
```

Q 5.4 Which of the three proposals have race conditions?

2007-1-11

Q 5.5 Which proposal would you recommend deploying, considering both correctness and performance goals?

2007-1-12

6 Course Swap*

(Chapter 5 in Chapter 4 setting)

The Subliminal Sciences Department, in order to reduce the department head's workload, has installed a Web server to help assign lecturers to classes for the Fall teaching term. There happen to be exactly as many courses as lecturers, and department policy is that every lecturer teach exactly one course and every course have exactly one lecturer. For each lecturer in the department, the server stores the name of the course currently assigned to that lecturer. The server's Web interface supports one request: to swap the courses assigned to a pair of lecturers.

Version One of the server's code looks like this:

```
// CODE VERSION ONE

    assignments[] // an associative array of course names indexed by lecturer

procedure SERVER ()
    do forever
        m ← wait for a request message
        value ← m.FUNCTION (m.arguments, …) // execute function in request message
        send value to m.sender

procedure EXCHANGE (lecturer1, lecturer2)
    temp ← assignments[lecturer1]
    assignments[lecturer1] ← assignments[lecturer2]
    assignments[lecturer2] ← temp
    return "OK"
```

Because there is only one application thread on the server, the server can handle only one request at a time. Requests comprise a function and its arguments (in this case EXCHANGE (*lecturer1*, *lecturer2*)), which is executed by the *m*.FUNCTION (*m.arguments*, …) call in the SERVER () procedure.

For all following questions, assume that there are no lost messages and no crashes. The operating system buffers incoming messages. When the server program asks for a message of a particular type (e.g., a request), the operating system gives it the oldest buffered message of that type.

Assume that network transmission times never exceed a fraction of a second and that computation also takes a fraction of a second. There are no concurrent operations other than those explicitly mentioned or implied by the pseudocode, and no other activity on the server computers.

* Credit for developing this problem set goes to Robert T. Morris.

Suppose the server starts out with the following assignments:

assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"

Q 6.1 Lecturers Herodotus and Augustine decide they wish to swap lectures, so that Herodotus teaches Numerology and Augustine teaches Steganography. They each send an EXCHANGE ("Herodotus", "Augustine") request to the server at the same time. If you look a moment later at the server, which, if any, of the following states are possible?

A.
assignments["Herodotus"] = "Numerology"
assignments["Augustine"] = "Steganography"

B.
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"

C.
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Steganography"

D.
assignments["Herodotus"] = "Numerology"
assignments["Augustine"] = "Numerology"

The Department of Dialectic decides it wants its own lecturer assignment server. Initially, it installs a completely independent server from that of the Subliminal Sciences Department, with the same rules (an equal number of lecturers and courses, with a one-to-one matching). Later, the two departments decide that they wish to allow their lecturers to teach courses in either department, so they extend the server software in the following way. Lecturers can send either server a CROSSEXCHANGE request, asking to swap courses between a lecturer in that server's department and a lecturer in the other server's department. In order to implement CROSSEXCHANGE, the servers can send each other SET-AND-GET requests, which set a lecturer's course and return the lecturer's previous course. Here's Version Two of the server code, for both departments:

```
// CODE VERSION TWO

procedure SERVER ()                    // same as in Version One
procedure EXCHANGE ()                  // same as in Version One

procedure CROSSEXCHANGE (local-lecturer, remote-lecturer)
    temp1 ← assignments[local-lecturer]
    send {SET-AND-GET, remote-lecturer, temp1} to the other server
    temp2 ← wait for response to SET-AND-GET
    assignments[local-lecturer] ← temp2
    return "OK"

procedure SET-AND-GET (lecturer, course) {
    old ← assignments[lecturer]
    assignments[lecturer] ← course
    return old
```

Suppose the starting state on the Subliminal Sciences server is:
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"
And on the Department of Dialectic server:
assignments["Socrates"] = "Epistemology"
assignments["Descartes"] = "Reductionism"

Q 6.2 At the same time, lecturer Herodotus sends a CROSSEXCHANGE ("Herodotus", "Socrates") request to the Subliminal Sciences server, and lecturer Descartes sends a CROSSEXCHANGE ("Descartes", "Augustine") request to the Department of Dialectic server. If you look a minute later at the Subliminal Sciences server, which, if any, of the following states are possible?

A.
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"

B.
assignments["Herodotus"] = "Epistemology"
assignments["Augustine"] = "Reductionism"

C.
assignments["Herodotus"] = "Epistemology"
assignments["Augustine"] = "Numerology"

In a quest to increase performance, the two departments make their servers multi-threaded: each server serves each request in a separate thread. Thus, if multiple requests arrive at roughly the same time, the server may process them in parallel. Each server has multiple processors. Here's the threaded server code, Version Three:

```
// CODE VERSION THREE

procedure EXCHANGE ()                  // same as in Version Two
procedure CROSSEXCHANGE ()             // same as in Version Two
procedure SET-AND-GET ()               // same as in Version Two

procedure SERVER ()
    do forever
        m ← wait for a request message
        ALLOCATE_THREAD (DOIT, m) // create a new thread that runs DOIT (m)

procedure DOIT (m)
    value ← m.FUNCTION(m.arguments, …)
    send value to m.sender
    EXIT ()     // terminate this thread
```

Q 6.3 With the same starting state as the previous question, but with the new version of the code, lecturer Herodotus sends a CROSSEXCHANGE ("Herodotus", "Socrates") request to the Subliminal Sciences server, and lecturer Descartes sends a CROSSEXCHANGE ("Descartes", "Augustine") request to the Department of Dialectic server, at the same time. If you look a minute later at the Subliminal Sciences server, which, if any, of the following states are possible?

A.
assignments["Herodotus"] = "Steganography"
assignments["Augustine"] = "Numerology"

B.
assignments["Herodotus"] = "Epistemology"
assignments["Augustine"] = "Reductionism"

C.
assignments["Herodotus"] = "Epistemology"
assignments["Augustine"] = "Numerology"

An alert student notes that Version Three may be subject to race conditions. He changes the code to have one lock per lecturer, stored in an array called *locks*[]. He changes EXCHANGE CROSSEXCHANGE, and SET-AND-GET to ACQUIRE locks on the lecturer(s) they affect. Here is the result, Version Four:

```
// CODE VERSION FOUR

procedure SERVER ()                    // same as in Version Three
procedure DOIT ()                      // same as in Version Three

procedure EXCHANGE (lecturer1, lecturer2)
    ACQUIRE (locks[lecturer1])
    ACQUIRE (locks[lecturer2])
    temp ← assignments[lecturer1]
    assignments[lecturer1] ← assignments[lecturer2]
    assignments[lecturer2] ← temp
    RELEASE (locks[lecturer1])
    RELEASE (locks[lecturer2])
    return "OK"

procedure CROSSEXCHANGE (local-lecturer, remote-lecturer)
    ACQUIRE (locks[local-lecturer])
    temp1 ← assignments[local-lecturer]
    send SET-AND-GET, remote-lecturer, temp1 to other server
    temp2 ← wait for response to SET-AND-GET
    assignments[local-lecturer] ← temp2
    RELEASE (locks[local-lecturer])
    return "OK"

procedure SET-AND-GET (lecturer, course)
    ACQUIRE (locks[lecturer])
    old ← assignments[lecturer]
    assignments[lecturer] ← course
    RELEASE (locks[lecturer])
    return old
```

Q 6.4 This code is subject to deadlock. Why?

Q 6.5 For each of the following situations, indicate whether deadlock can occur. In each situation, there is no activity other than that mentioned.

A. Client A sends EXCHANGE ("Herodotus", "Augustine") at the same time that client B sends EXCHANGE ("Herodotus", "Augustine"), both to the Subliminal Sciences server.

B. Client A sends EXCHANGE ("Herodotus", "Augustine") at the same time that client B sends EXCHANGE ("Augustine", "Herodotus"), both to the Subliminal Sciences server.

C. Client A sends CROSSEXCHANGE ("Augustine", "Socrates") to the Subliminal Sciences server at the same time that client B sends CROSSEXCHANGE ("Descartes", "Herodotus") to the Department of Dialectic server.

D. Client A sends CROSSEXCHANGE ("Augustine", "Socrates") to the Subliminal Sciences server at the same time that client B sends CROSSEXCHANGE ("Socrates", "Augustine") to the Department of Dialectic server.

E. Client A sends CROSSEXCHANGE ("Augustine", "Socrates") to the Subliminal Sciences server at the same time that client B sends CROSSEXCHANGE ("Descartes", "Augustine") to the Department of Dialectic server.

7 Banking on Local Remote Procedure Call

(Chapter 5)

The bank president has asked Ben Bitdiddle to add enforced modularity to a large banking application. Ben splits the program into two pieces: a client and a service. He wants to use remote procedure calls to communicate between the client and service, which both run on the same physical machine with one processor. Ben explores an implementation, which the literature calls **lightweight remote procedure call** (LRPC). Ben's version of LRPC uses user-level gates. User gates can be bootstrapped using two kernel gates—one gate that registers the name of a user gate and a second gate that performs the actual transfer:

- REGISTER_GATE (*stack*, *address*). It registers address *address* as an entry point, to be executed on the stack *stack*. The kernel stores these addresses in an internal table.
- TRANSFER_TO_GATE (*address*). It transfers control to address *address*. A client uses this call to transfer control to a service. The kernel must first check if *address* is an address that is registered as a gate. If so, the kernel transfers control; otherwise it returns an error to the caller.

We assume that a client and service each run in their own virtual address space. On initialization, the service registers an entry point with REGISTER_GATE and allocates a block, at address *transfer*. Both the client and service map the transfer block in each address space with READ and WRITE permissions. The client and service use this shared transfer page to communicate the arguments to and results of a remote procedure call. The client and server each start with one thread. There are no user programs other than the client and server running on the machine.

The following pseudocode summarizes the initialization:

Service

```
procedure INIT_SERVICE ()
    REGISTER_GATE (STACK, receive)
    ALLOCATE_BLOCK (transfer)
    MAP (my_id, transfer, shared_server)
    while TRUE do YIELD ()
```

Client

```
procedure INIT_CLIENT ()
    MAP (my_id, transfer, shared_client)
```

When a client performs an LRPC, the client copies the arguments of the LRPC into the transfer page. Then, it calls TRANSFER_TO_GATE to transfer control to the service address space at the registered address *receive*. The client thread, which is now in the service's address space, performs the requested operation (the code for the procedure at the address *receive* is not shown because it is not important for the questions). On returning from the requested operation, the procedure at the address *receive* writes the result parameters in the transfer block and transfers control back to the client's address space to the procedure RETURN_LRPC. Once back in the client address space in RETURN_LRPC, the

client copies the results back to the caller. The following pseudocode summarizes the implementation of LRPC:

```
1  procedure LRPC (id, request)
2      COPY (request, shared_client)
3      TRANSFER_TO_GATE (receive)
4      return
5
6  procedure RETURN_LRPC()
7      COPY (shared_client, reply)
8      return (reply)
```

Now that we know how to use the procedures REGISTER_GATE and TRANSFER_TO_GATE, let's turn our attention to the implementation of TRANSFER_TO_GATE (*entrypoint* is the internal kernel table recording gate information):

```
1  procedure TRANSFER_TO_GATE (address)
2      if id exists such that entrypoint[id].entry = address then
3          R1 ← USER_TO_KERNEL (entrypoint[id].stack)
4          R2 ← address
5          STORE R2, R1                    // put address on service's stack
6          SP ← entrypoint[id].stack       // set SP to service stack
7          SUB 4, SP                       // adjust stack
8          PMAR ← entrypoint[id].pmar      // set page map address
9          USER ← ON                       // switch to user mode
10         return                          // returns to address
11     else
12         return (ERROR)
```

The procedure checks whether or not the service has registered *address* as an entry point (line 2). Lines 4–7 push the entry address on the service's stack and set the register SP to point to the service's stack. To be able to do so, the kernel must translate the address for the stack in the service address space into an address in the kernel address space so that the kernel can write the stack (line 3). Finally, the procedure stores the page-map address register for the service into PMAR (line 8), sets the user-mode bit to ON (line 9), and invokes the gate's procedure by returning from TRANSFER_TO_GATE (line 10), which loads *address* from the service's stack into PC.

The implementation of this procedure is tricky because its switches address spaces, and thus the implementation must be careful to ensure that it is referring to the appropriate variable in the appropriate address space. For example, after line 8 TRANSFER_TO_GATE runs the next instruction (line 9) in the service's address space. This works only if the kernel is mapped in both the client and service's address space at the same address.

Q 7.1 The procedure INIT_SERVICE calls YIELD. In which address space or address spaces is the code that implements the supervisor call YIELD located?

Q 7.2 For LRPC to work correctly, must the two virtual addresses *transfer* have the same value in the client and service address space?

Q 7.3 During the execution of the procedure located at address *receive* how many threads are running or are in a call to YIELD in the service address space?

Q 7.4 How many supervisor calls could the client perform in the procedure LRPC?

Q 7.5 Ben's goal is to enforce modularity. Which of the following statements are true statements about Ben's LRPC implementation?

A. The client thread cannot transfer control to any address in the server address space.
B. The client thread cannot overwrite any physical memory that is mapped in the server's address space.
C. After the client has invoked TRANSFER_TO_GATE in LRPC, the server is guaranteed to invoke RETURN_LRPC.
D. The procedure LRPC ought to be modified to check the response message and process only valid responses.

Q 7.6 Assume that REGISTER_GATE and TRANSFER_TO_GATE are also used by other programs. Which of the following statements is true about the implementations of REGISTER_GATE and TRANSFER_TO_GATE?

A. The kernel might use an invalid address when writing the value *address* on the stack passed in by a user program.
B. A user program might use an invalid address when entering the service address space.
C. The kernel transfers control to the server address space with the user-mode bit switched OFF.
D. The kernel enters the server address space only at the registered address entry *address*.

Ben modifies the client to have multiple threads of execution. If one client thread calls the server and the procedure at address *receive* calls YIELD, another client thread can run on the processor.

Q 7.7 Which of the following statements is true about the implementation of LRPC with multiple threads?

A. On a single-processor machine, there can be race conditions when multiple client threads call LRPC, even if the kernel schedules the threads non-preemptively.
B. On a single-processor machine, there can be race conditions when multiple clients threads call LRPC and the kernel schedules the threads preemptively.
C. On multiprocessor computer, there can be race conditions when multiple client threads call LRPC.
D. It is impossible to have multiple threads if the computer doesn't have multiple physical processors.

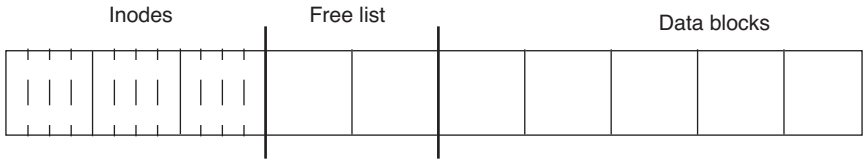
2004–1–4…10

8 The Bitdiddler*

(Chapter 5)

Ben Bitdiddle is designing a file system for a new handheld computer, the Bitdiddler, which is designed to be especially simple for, as he likes to say, "people who are just average, like me."

In keeping with his theme of simplicity and ease of use for average people, Ben decides to design a file system without directories. The disk is physically partitioned into three regions: an inode list, a free list, and a collection of 4K data blocks, much like the UNIX file system. Unlike in the UNIX file system, each inode contains the name of the file it corresponds to, as well as a bit indicating whether or not the inode is in use. Like the UNIX file system, the inode also contains a list of blocks that compose the file, as well as metadata about the file, including permission bits, its length in bytes, and modification and creation timestamps. The free list is a bitmap, with one bit per data block indicating whether that block is free or in use. There are no indirect blocks in Ben's file system. The following figure illustrates the basic layout of the Bitdiddler file system:



The file system provides six primary calls: CREATE, OPEN, READ, WRITE, CLOSE, and UNLINK. Ben implements all six correctly and in a straightforward way, as shown in Figure PS.2. All updates to the disk are synchronous; that is, when a call to write a block of data to the disk returns, that block is definitely installed on the disk. Individual block writes are atomic.

Q 8.1 Ben notices that if he pulls the batteries out of the Bitdiddler while running his application and then replaces the batteries and reboots the machine, the file his application created exists but contains unexpected data that he didn't write into the file.

* Credit for developing this problem set goes to Samuel R. Madden.

```
procedure CREATE (filename)
    scan all non-free inodes to avoid duplicate filenames (return error if duplicate)
    find a free inode in the inode list
    update the inode with 0 data blocks, mark it as in use, write it to disk
    update the free list to indicate the inode is in use, write free list to disk

procedure OPEN (filename) // returns a file handle
    scan non-free inodes looking for filename
    if found, allocate and return a file handle fh that refers to that inode

procedure WRITE (fh, buf, len)
    look in file handle fh to determine inode of the file, read inode from disk
    if there is free space in last block of file, write to it
    determine number of new blocks needed, n
    for i ← 1 to n
        use free list to find a free block b
        update free list to show b is in use, write free list to disk
        add b to inode, write inode to disk
        write appropriate data for block b to disk

procedure READ (fh, buf, len)
    look in file handle fh to determine inode of the file, read inode from disk
    read len bytes of data from the current location in file into buf

procedure CLOSE (fh)
    remove fh from the file handle table

procedure UNLINK (filename)
    scan non-free inodes looking for filename, mark that inode as free
    write inode to disk
    mark data blocks used by file as free in free list
    write modified free list blocks to disk
    Ben writes the following simple application for the Bitdiddler:
    CREATE (filename)
    fh ← OPEN (filename)
    WRITE (fh, app_data, LENGTH (app_data)) // app_data is some data to be written
    CLOSE (fh)
```

FIGURE PS.2

The Bitdidder file system.

Which of the following are possible explanations for this behavior? (Assume that the disk controller never writes partial blocks.)

A. The free list entry for a data page allocated by the call to WRITE was written to disk, but neither the inode nor the data page itself was written.
B. The inode allocated to Ben's application previously contained a (since deleted) file with the same name. If the system crashed during the call to CREATE, it may cause the old file to reappear with its previous contents.
C. The free list entry for a data page allocated by the call to WRITE as well as a new copy of the inode were written to disk, but the data page itself was not.
D. The free list entry for a data page allocated by the call to WRITE as well as the data page itself were written to disk, but the new inode was not.

Q 8.2 Ben decides to fix inconsistencies in the Bitdiddler's file system by scanning its data structures on disk every time the Bitdiddler starts up. Which of the following inconsistencies can be identified using this approach (without modifying the Bitdiddler implementation)?

A. In-use blocks that are also on the free list.
B. Unused blocks that are not on the free list.
C. In-use blocks that contain data from previously unlinked files.
D. Blocks used in multiple files.

2007-3-6&7

9 Ben's Kernel

(Chapter 5)

Ben develops an operating system for a simple computer. The operating system has a kernel that provides virtual address spaces, threads, and output to a console.

Each application has its own user-level address space and uses one thread. The kernel program runs in the kernel address space but doesn't have its own thread. (The kernel program is described in more detail below.)

The computer has one processor, a memory, a timer chip (which will be introduced later), a console device, and a bus connecting the devices. The processor has a user-mode bit and is a **multiple register set** design, which means that it has two sets of program counter (PC), stack pointer (SP), and page-map address registers (PMAR). One set is for user space (the user-mode bit is set to ON): *upc*, *usp*, and *upmar*. The other set is for kernel space (the user-mode bit is set to OFF): *kpc*, *ksp*, and *kpmar*. Only programs in kernel mode are allowed to store to *upmar*, *kpc*, *ksp*, and *kpmar*—storing a value in these registers is an illegal instruction in user mode.

The processor switches from user to kernel mode when one of three events occurs: an application issues an illegal instruction, an application issues a supervisor call instruction (with the SVC instruction), or the processor receives an interrupt in user mode. The processor switches from user to kernel mode by setting the user-mode bit OFF. When that happens, the processor continues operation but using the current values in the *kpc*, *ksp*, and *kpmar*. The user program counter, stack pointer, and page-map address values remain in *upc*, *usp*, and *upmar*, respectively.

To return from kernel to user space, a kernel program executes the RTI instruction, which sets the user-mode bit to ON, causing the processor to use *upc*, *usp*, and *upmar*. The *kpc*, *ksp*, and *kpmar* values remain unchanged, awaiting the next SVC. In addition to these registers, the processor has four general-purpose registers: *ur0*, *ur1*, *kr0*, and *kr1*. The *ur0* and *ur1* pair are active in user mode. The *kr0* and *kr1* pair are active in kernel mode.

Ben runs two user applications. Each executes the following set of programs:

```
integer t initially 1              // initial value for shared variable t
procedure MAIN ()
    do forever
        t ← t + t
        PRINT (t)
        YIELD ()

procedure YIELD
    SVC 0
```

PRINT prints the value of *t* on the output console. The output console is an output-only device and generates no interrupts.

The kernel runs each program in its own user-level address space. Each user address space has one thread (with its own stack), which is managed by the kernel:

```
integer currentthread      // index for the current user thread

structure thread[2]        // Storage place for thread state when not running
    integer sp             // user stack pointer
    integer pc             // user program counter
    integer pmar           // user page-map address register
    integer r0             // user register 0
    integer r1             // user register 1

procedure DOYIELD ()
    thread[currentthread].sp ← usp                          // save registers
    thread[currentthread].pc ← upc
    thread[currentthread].pmar ← upmar
    thread[currentthread].r0 ← ur0
    thread[currentthread].r1 ← ur1
    currentthread ← (currentthread + 1) modulo 2            // select new thread
    usp ← thread[currentthread].sp                          // restore registers
    upc ← thread[currentthread].pc
    upmar ← thread[currentthread].pmar
    ur0 ← thread[currentthread].r0
    ur1 ← thread[currentthread].r1
```

For simplicity, this non-preemptive thread manager is tailored for just the two user threads that are running on Ben's kernel. The system starts by executing the procedure KERNEL. Here is its code:

```
procedure KERNEL ()
    CREATE_THREAD (MAIN)                    // Set up Ben's two threads
    CREATE_THREAD (MAIN)                    //
    usp ← thread[1].sp                      // initialize user registers for thread 1
    upc ← thread[1].pc
    upmar ← thread[1].pmar
    ur0 ← thread[1].r0
    ur1 ← thread[1].r1
    do forever
        RTI                                 // Run a user thread until it issues an SVC
        n ← ???                             // See question Q 9.1
        if n = 0 then DOYIELD()
```

Since the kernel passes control to the user with the RTI instruction, when the user executes an SVC, the processor continues execution in the kernel at the instruction following the RTI.

Ben's operating system sets up three page maps, one for each user program, and one for the kernel program. Ben has carefully set up the page maps so that the three address spaces don't share any physical memory.

Q 9.1 Describe how the supervisor obtains the value of *n*, which is the identifier for the SVC that the calling program has invoked.

Q 9.2 How can the current address space be switched?

A. By the kernel writing the *kpmar* register.
B. By the kernel writing the *upmar* register.
C. By the processor changing the user-mode bit.
D. By the application writing the *kpmar* or *upmar* registers.
E. By DOYIELD saving and restoring *upmar.*

Q 9.3 Ben runs the system for a while, watching it print several results, and then halts the processor to examine its state. He finds that it is in the kernel, where it is just about to execute the RTI instruction. In which procedure(s) could the user-level thread resume when the kernel executes that RTI instruction?

A. in the procedure KERNEL.
B. in the procedure MAIN.
C. in the procedure YIELD.
D. in the procedure DOYIELD.

Q 9.4 In Ben's design, what mechanisms play a role in enforcing modularity?

A. Separate address spaces because wild writes from one application cannot modify the data of the other application.
B. User-mode bit because it disallows user programs to write to upmar and kpmar.
C. The kernel because it forces threads to give up the processor.
D. The application because it has few lines of code.

Ben reads about the timer chip in his hardware manual and decides to modify the kernel to take advantage of it. At initialization time, the kernel starts the timer chip, which will generate an interrupt every 100 milliseconds. (Ben's computer has no other sources of interrupts.) Note that the interrupt-enable bit is OFF when executing in the kernel address space; the processor checks for interrupts only before executing a user-mode instruction. Thus, whenever the timer chip generates an interrupt while the processor is in kernel mode, the interrupt will be delayed until the processor returns to user mode. An interrupt in user mode causes an SVC -1 instruction to be inserted in the instruction stream. Finally, Ben modifies the kernel by replacing the **do forever** loop and adding an interrupt handler, as follows:

```
do forever
   RTI                                  // Run a user thread until it issues an SVC
   n ← ???                              // Assume answer to question Q 9.1
   if n = 1 then DOINTERRUPT ()
   if n = 0 then DOYIELD ()

procedure DOINTERRUPT ()
   DOYIELD ()
```

Do not make any assumption about the speed of the processor.

Q 9.5 Ben again runs the system for a while, watching it print several results, and then he halts the processor to examine its state. Once again, he finds that it is in the kernel,

where it is just about to execute the RTI instruction. In which procedure(s) could the user-level thread resume after the kernel executes the RTI instruction?

A. in the procedure DOINTERRUPT.
B. in the procedure KERNEL.
C. in the procedure MAIN.
D. in the procedure YIELD.
E. in the procedure DOYIELD.

Q 9.6 In Ben's second design, what mechanisms play a role in enforcing modularity?

A. Separate address spaces because wild writes from one application cannot modify the data of the other application.
B. User-mode bit because it disallows user programs to write to UPMAR and KPMAR.
C. The timer chip because it, in conjunction with the kernel, forces threads to give up the processor.
D. The application because it has few lines of code.

Ben modifies the two user programs to share the variable *t*, by mapping *t* in the virtual address space of both user programs at the same place in physical memory. Now both threads read and write the same *t*.

Note that registers are not shared between threads: the scheduler saves and restores the registers on a thread switch. Ben's simple compiler translates the critical region of code:

$t \leftarrow t + t$

into the processor instructions:

```
100 LOAD t, r0      // read t into register 0
104 LOAD t, r1      // read t into register 1
108 ADD r1, r0      // add registers 0 and 1, leave result in register 0
112 STORE r0, t     // store register 0 into t
```

The numbers in the leftmost column in this code are the virtual addresses where the instructions are stored in both virtual address spaces. Ben's processor executes the individual instructions atomically.

Q 9.7 What values can the applications print (don't worry about overflows)?

A. Some odd number.
B. Some even number other than a power of two.
C. Some power of two.
D. 1

In a conference proceedings, Ben reads about an idea called **restartable atomic regions**[*] and implements them. If a thread is interrupted in a critical region, the thread

[*] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992), pages 223–233.

manager restarts the thread at the beginning of the critical region when it resumes the thread. Ben recodes the interrupt handler as follows:

```
procedure DOINTERRUPT ()
    if upc ≥ 100 and upc ≤ 112 then// Were we in the critical region?
        upc ← 100              // yes, restart critical region when resumed!
    DOYIELD ()
```

The processor increments the program counter after interpreting an instruction and before processing interrupts.

Q 9.8 Now, what values can the applications print (don't worry about overflows)?

A. Some odd number.
B. Some even number other than a power of two.
C. Some power of two.
D. 1

Q 9.9 Can a second thread enter the region from virtual addresses 100 through 112 while the first thread is in it (i.e., the first thread's *upc* contains a value in the range 100 through 112)?

A. Yes, because while the first thread is in the region, an interrupt may cause the processor to switch to the second thread and the second thread might enter the region.
B. Yes, because the processor doesn't execute the first three lines of code in DOINTERRUPT atomically.
C. Yes, because the processor doesn't execute DOYIELD atomically.
D. Yes, because MAIN calls YIELD.

Ben is exploring if he can put just any code in a restartable atomic region. He creates a restartable atomic region that contains three instructions, which swap the content of two variables *a* and *b* using a temporary *x*:

```
100 x ← a
104 a ← b
108 b ← x
```

Ben also modifies DOINTERRUPT, replacing 112 with 108:

```
procedure DOINTERRUPT ()
    if upc ≥ 100 and upc ≤ 108 then// Were we in the critical region?
        upc ← 100;       // yes, restart critical region when resumed!
    DOYIELD ()
```

Variables *a* and *b* start out with the values *a* = 1 and *b* = 2, and the timer chip is running.

Q 9.10 What are some possible outcomes if a thread executes this restartable atomic region and variables *a*, *b*, and *x* are not shared?

A. *a* = 2 and *b* = 1
B. *a* = 1 and *b* = 2
C. *a* = 2 and *b* = 2
D. *a* = 1 and *b* = 1

2003–1–5…13

10 A Picokernel-Based Stock Ticker System

(Chapter 5)

Ben Bitdiddle decides to design a computer system based on a new kernel architecture he calls *picokernels* and on a new hardware platform called *simplePC*. Ben has paid attention to Section 1.1 and is going for extreme simplicity. The simplePC platform contains one simple processor, a page-based virtual memory manager (which translates the virtual addresses issued by the processor), a memory module, and an input and output device. The processor has two special registers, a program counter (PC) and a stack pointer (SP). The SP points to the value on the top of the stack.

The calling convention for the simplePC processor uses a simple stack model:

- A call to a procedure pushes the address of the instruction after the call onto the stack and then jumps to the procedure.
- Return from a procedure pops the address from the top of the stack and jumps.

Programs on the simplePC don't use local variables. Arguments to procedures are passed in registers, which are *not* saved and restored automatically. Therefore, the only values on the stack are return addresses.

Ben develops a simple stock ticker system to track the stocks of the start-up he joined. The program reads a message containing a single integer from the input device and prints it on the output device:

```
101. boolean input_available

1.  procedure READ_INPUT ()
2.      do forever
3.          while input_available = FALSE do nothing // idle loop
4.          PRINT_MSG(quote)
5.          input_available ← FALSE

200. boolean output_done
201. structure output_buffer at 71fff2hex          // hardware address of output buffer
202.        integer quote

12. procedure PRINT_MSG (m)
13.     output_buffer.quote ← m
14.     while output_done = FALSE do nothing   // idle loop
15.     output_done ← FALSE

17. procedure MAIN ()
18.     READ_INPUT ()
19.     halt                                   // shutdown computer
```

In addition to the MAIN program, the program contains two procedures: READ_INPUT and PRINT_MSG. The procedure READ_INPUT spin-waits until *input_available* is set to TRUE by the input device (the stock reader). When the input device receives a stock quote, it places the quote value into *msg* and sets *input_available* to TRUE.

The procedure PRINT_MSG prints the message on an output device (a terminal in this case); it writes the value stored in the message to the device and waits until it is printed; the output device sets *output_done* to TRUE when it finishes printing.

The numbers on each line correspond to addresses as issued by the processor to read and write instructions and data. Assume that each line of pseudocode compiles into one machine instruction and that there is an implicit **return** at the end of each procedure.

Q 10.1 What do these numbers mentioned on each line of the program represent?

A. Virtual addresses.
B. Physical addresses.
C. Page numbers.
D. Offsets in a virtual page.

Ben runs the program directly on simplePC, starting in MAIN, and at some point he observes the following values on the stack (remember, only the stock ticker program is running):

```
stack
 19
 5  ← stack pointer
```

Q 10.2 What is the meaning of the value 5 on the stack?

A. The return address for the next return instruction.
B. The return address for the previous return instruction.
C. The current value of PC.
D. The current value of SP.

Q 10.3 Which procedure is being executed by the processor?

A. READ_INPUT
B. PRINT_MSG
C. MAIN

Q 10.4 PRINT_MSG writes a value to *quote*, which is stored at the address $71ff2_{hex}$, with the expectation that the value will end up on the terminal. What technique is used to make this work?

A. Memory-mapped I/O.
B. Sequential I/O.
C. Streams.
D. Remote procedure call.

Ben wants to run multiple instances of his stock ticker program on the simplePC platform so that he can obtain more frequent updates to track more accurately his current

net worth. Ben buys another input and output device for the system, hooks them up, and he implements a trivial thread manager:

```
300. integer threadtable[2];                    // stores stack pointers of threads.
                                                // first slot is threadtable[0]
302. integer current_thread initially 0;

21. procedure YIELD ()
22.     threadtable[current_thread] ← SP        // move value of SP into table
23.     current_thread ← (current_thread + 1) modulo 2
24.     SP ← threadtable[current_thread]        // load value from table into SP
25.     return
```

Each thread reads from and writes to its own device and has its own stack. Ben also modifies READ_INPUT from page ps–37:

```
100. integer msg[2]                             // CHANGED to use array
102. boolean input_available[2]                 // CHANGED to use array

30. procedure READ_INPUT ()
31.     do forever
32.         while input_available[current_thread] = FALSE do        // CHANGED
33.         YIELD ()                            // CHANGED
34.         continue                            // CHANGED
35.       PRINT_MSG (msg[current_thread])       // CHANGED to use array
36.       input_available[current_thread] ← FALSE   // CHANGED to use array
```

Ben powers up the simplePC platform and starts each thread running in MAIN. The two threads switch back and forth correctly. Ben stops the program temporarily and observes the following stacks:

```
stack of thread 0          stack of thread 1
    19                         19
    36 ← stack pointer         34 ← stack pointer
```

Q 10.5 Thread 0 was running (i.e., *current_thread* = 0). Which instruction will the processor be running after thread 0 executes the **return** instruction in YIELD the next time?

A. 34. **continue**
B. 19. **halt**
C. 35. PRINT_MSG (*msg*[*current_thread*]);
D. 36. *input_available*[*current_thread*] ← FALSE;

and which thread will be running?

Q 10.6 What address values can be on the stack of each thread?

A. Addresses of any instruction.
B. Addresses to which called procedures return.
C. Addresses of any data location.
D. Addresses of instructions and data locations.

Ben observes that each thread in the stock ticker program spends most of its time polling its input variable. He introduces an explicit procedure that the devices can use to notify the threads. He also rearranges the code for modularity:

```
400.integer state[2];

40. procedure SCHEDULE_AND_DISPATCH ()
41.     threadtable[current_thread] ← SP
42.     while (what should go here?) do                    // See question Q 10.7.
43.         current_thread ← (current_thread + 1) modulo 2
45.     SP ← threadtable[current_thread];
46.     return

50. procedure YIELD()
51.     state[current_thread] ← WAITING
52.     SCHEDULE_AND_DISPATCH ()
53.     return

60. procedure NOTIFY (n)
61.     state[n] ← RUNNABLE
62.     return
```

When the input device receives a new stock quote, the device interrupts the processor and saves the PC of the currently running thread on the currently running thread's stack. Then the processor runs the interrupt procedure. When the interrupt handler returns, it pops the return address from the current stack, returning control to a thread. The pseudocode for the interrupt handler is:

```
procedure DEVICE (n)                                  // interrupt for input device n
    push current thread's PC on stack pointed to by SP
    while input_available[n] = TRUE do nothing;       // wait until read_input is done
                                                      // with the last input
    msg[n] ← stock quote
    input_available[n] ← TRUE
    NOTIFY (n)                                        // notify thread n
    return                                            // i.e., pop PC
```

During the execution of the interrupt handler, interrupts are disabled. Thus, an interrupt handler and the procedures that it calls (e.g., NOTIFY) cannot be interrupted. Interrupts are reenabled when DEVICE returns.

Using the new thread manager, answer the following questions:

Q 10.7 What expression should be evaluated in the **while** at address 42 to ensure correct operation of the thread package?

A. *state*[*current_thread*] = WAITING
B. *state*[*current_thread*] = RUNNABLE
C. *threadtable*[*current_thread*] = SP
D. FALSE

Q 10.8 Assume thread 0 is running and thread 1 is not running (i.e., it has called YIELD). What event or events need to happen before thread 1 will run?

A. Thread 0 calls YIELD.
B. The interrupt procedure for input device 1 calls NOTIFY.
C. The interrupt procedure for input device 0 calls NOTIFY.
D. No events are necessary.

Q 10.9 What values can be on the stack of each thread?

A. Addresses of any instruction except those in the device driver interrupt procedure.
B. Addresses of all instructions, including those in the device driver interrupt procedure.
C. Addresses to which procedures return.
D. Addresses of instructions and data locations.

Q 10.10 Under which scenario can thread 0 deadlock?

A. When device 0 interrupts thread 0 just before the first instruction of YIELD.
B. When device 0 interrupts just after thread 0 completed the first instruction of YIELD.
C. When device 0 interrupts thread 0 between instructions 35 and 36 in the READ_INPUT procedure on page ps–37.
D. When device 0 interrupts when the processor is executing SCHEDULE_AND_DISPATCH and thread 0 is in the WAITING state.

2000–1–7…16

11 Ben's Web Service

(Chapter 5)

Ben Bitdiddle is so excited about Amazing Computer Company's plans for a new segment-based computer architecture that he takes the job the company offered him.

Amazing Computer Company has observed that using one address space per program puts the text, data, stack, and system libraries in the same address space. For example, a Web server has the program text (i.e., the binary instructions) for the Web server, its internal data structures such as its cache of recently-accessed Web pages, the stack, and a system library for sending and receiving messages all in a single address space. Amazing Computer Company wants to explore how to enforce modularity even further by separating the text, data, stack, and system library using a new memory system.

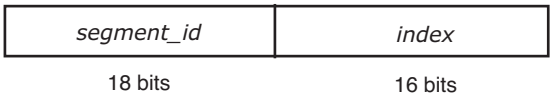
The Amazing Computer Company has asked every designer in the company to come up with a design to enforce modularity further. In a dusty book about the PDP 11/70, Ben finds a description of a hardware gadget that sits between the processor and the physical memory, translating virtual addresses to physical addresses. The PDP 11/70 used that gadget to allow each program to have its own address space, starting at address 0.

The PDP 11/70 did this through having one segment per program. Conceptually, each segment is a variable-sized, linear array of bytes starting at virtual address 0. Ben bases his memory system on the PDP 11/70's scheme with the intention of implementing hard modularity. Ben defines a segment through a segment descriptor:

```
structure segmentDescriptor
    physicalAddress physAddr
    integer length
```

The *physAddr* field records the address in physical memory where the segment is located. The *length* field records the length of the segment in bytes.

Ben's processor has addresses consisting of 34 bits: 18 bits to identify a segment and 16 bits to identify the byte within the segment:



A virtual address that addresses a byte outside a segment (i.e., an *index* greater than the *length* of the segment) is illegal.

Ben's memory system stores the segment descriptors in a table, *segmentTable*, which has one entry for each segment:

```
structure segmentDescriptor
    segmentTable[NSEGMENT]
```

The segment table is indexed by *segment_id*. It is shared among all programs and stored at physical address 0.

The processor used by Ben's computer is a simple RISC processor, which reads and writes memory using LOAD and STORE instructions. The LOAD and STORE instructions take

a virtual address as their argument. Ben's computer has enough memory that all programs fit in physical memory.

Ben ports a compiler that translates a source program to generate machine instructions for his processor. The compiler translates into a position-independent machine code: JUMP instructions specify an offset relative to the current value of the program counter. To make a call into another segment, it supports the LONGJUMP instruction, which takes a virtual address and jumps to it.

Ben's memory system translates a virtual address to a physical address with TRANSLATE:

```
1  procedure TRANSLATE (addr)
2      segment_id ← addr[0:17]
3      segment ← segmentTable[segment_id]
4      index ← addr[18:33]
5      if index < segment.length then return segment.physAddr + index
6      …       // What should the program do here? (see question Q 11.4, below)
```

After successfully computing the physical address, Ben's memory management unit retrieves the addressed data from physical memory and delivers it to the processor (on a LOAD instruction) or stores the data in physical memory (on a STORE instruction).

Q 11.1 What is the maximum sensible value of NSEGMENT?

Q 11.2 Given the structure of a virtual address, what is the maximum size of a segment in bytes?

Q 11.3 How many bits wide must a physical address be?

Q 11.4 The missing code on line 6 should

A. signal the processor that the instruction that issued the memory reference has caused an illegal address fault
B. signal the processor that it should change to user mode
C. **return** *index*
D. signal the processor that the instruction that issues the memory reference is an interrupt handler

Ben modifies his Web server to enforce modularity between the different parts of the server. He allocates the text of the program in segment 1, a cache for recently used Web pages in segment 2, the stack in segment 3, and the system library in segment 4. Segment 4 contains the text of the library program but no variables (i.e., the library program doesn't store variables in its own segment).

Q 11.5 To translate the Web server the compiler has to do which of the following?

A. Compute the physical address for each virtual address.
B. Include the appropriate segment ID in the virtual address used by a LOAD instruction.
C. Generate LONGJUMP instructions for calls to procedures located in different segments.
D. Include the appropriate segment ID in the virtual address used by a STORE instruction.

Ben runs the segment-based implementation of his Web server and to his surprise observes that errors in the Web server program can cause the text of the system library to be overwritten. He studies his design and realizes that the design is bad.

Q 11.6 What aspect of Ben's design is bad and can cause the observed behavior?

A. A STORE instruction can overwrite the segment ID of an address.
B. A LONGJMP instruction in the Web server program may jump to an address in the library segment that is not the start of a procedure.
C. It doesn't allow for paging of infrequently used memory to a secondary storage device.
D. The Web server program may get into an endless loop.

Q 11.7 Which of the following extensions of Ben's design would address each of the preceding problems?

A. The processor should have a protected user-mode bit, and there should be a separate segment table for kernel and user programs
B. Each segment descriptor should have a protection bit, which specifies whether the processor can write or only read from this segment
C. The LONGJMP instruction should be changed so that it can transfer control only to designated entry points of a segment
D. Segments should all be the same size, just like pages in page-based virtual memory systems
E. Change the operating system to use a preemptive scheduler

The system library for Ben's Web server contains code to send and receive messages. A separate program, the network manager, manages the network card that sends and receives messages. The Web server and the network manager each have one thread of execution. Ben wants to understand why he needs eventcounts for sequence coordination of the network manager and the Web server, so he decides to implement the coordination twice, once using eventcounts and the second time using event variables.

Here are Ben's two versions of the Web server:

Web server using eventcounts

```
eventcount inCnt
integer doneCnt


procedure SERVE ()
do forever
    AWAIT (inCnt, doneCnt);
    DO_REQUEST ();
    doneCnt ← doneCnt + 1;
```

Web server using events

```
event input
integer inCnt
integer doneCnt

procedure SERVE ()
do forever
    while inCnt ≤ doneCnt do  // A
        WAITEVENT (input);    // B
    DO_REQUEST ();            // C
    doneCnt ← doneCnt + 1;    // D
```

Both versions use a thread manager as described in Chapter *5*, except for the changes to support eventcounts or events. The eventcount version is exactly the one described in

Chapter *5*. The AWAIT procedure has semantics for eventcounts: when the Web server thread calls AWAIT, the thread manager puts the calling thread into the WAITING state unless *inCnt* exceeds *doneCnt*.

The event-based version is almost identical to the eventcount one but has a few changes. An **event variable** is a list of threads waiting for the event. The procedure WAITEVENT puts the current executing thread on the list for the event, records that the current thread is in the WAITING state, and releases the processor by calling YIELD.

In both versions, when the Web server has completed processing a packet, it increases *doneCnt*.

The two corresponding versions of the code for handling each packet arrival in the network manager are:

Network manager using eventcounts	*Network manager using events*	
ADVANCE (*inCnt*)	*inCnt* ← *inCnt* + 1	// E
	NOTIFYEVENT (*input*)	// F

The ADVANCE procedure wakes up the Web server thread if it is already asleep. The NOTIFYEVENT procedure removes all threads from the list of the event and puts them into the READY state. The shared variables are stored in a segment shared between the network manager and the Web server.

Ben is a bit worried about writing code that involves coordinating multiple activities, so he decides to test the code carefully. He buys a computer with one processor to run both the Web server and the network manager using a preemptive thread scheduler. Ben ensures that the two threads (the Web server and the network manager) never run inside the thread manager at the same time by turning off interrupts when the processor is running the thread manager's code (which includes ADVANCE, AWAIT, NOTIFYEVENT, and WAITEVENT).

To test the code, Ben changes the thread manager to preempt threads frequently (i.e., each thread runs with a short time slice). Ben runs the old code with eventcounts and the program behaves as expected, but the new code using events has the problem that the Web server sometimes delays processing a packet until the next packet arrives.

Q 11.8 The program steps that might be causing the problem are marked with letters in the code of the event-based solution above. Using those letters, give a sequence of steps that creates the problem. (Some steps might have to appear more than once, and some might not be necessary to create the problem.)

2002–1–4…11

12 A Bounded Buffer with Semaphores

(Chapter 5)

Using semaphores, DOWN and UP (see Sidebar 5.7), Ben implements an in-kernel bounded buffer as shown in the pseudocode below. The kernel maintains an array of *port_info*s. Each *port_info* contains a bounded buffer. The content of the message structure is not important for this problem, other than that it has a field *dest_port*, which specifies the destination port. When a message arrives from the network, it generates an interrupt, and the network interrupt handler (INTERRUPT) puts the message in the bounded buffer of the port specified in the message. If there is no space in that bounded buffer, the interrupt handler throws the message away. A thread consumes a message by calling RECEIVE_MESSAGE, which removes a message from the bounded buffer of the port it is receiving from.

To coordinate the interrupt handler and a thread calling RECEIVE_MESSAGE, the implementation uses a semaphore. For each port, the kernel keeps a semaphore *n* that counts the number of messages in the port's bounded buffer. If *n* reaches 0, the thread calling DOWN in RECEIVE_MESSAGE will enter the WAITING state. When INTERRUPT adds a message to the buffer, it calls UP on *n*, which will wake up the thread (i.e., set the thread's state to RUNNABLE).

The kernel schedules threads preemptively.

```
structure port_info
    semaphore instance count initially 0
    message instance buffer[NMSG]                  // an array of NMSG messages
    long integer in initially 0
    long integer out initially 0

procedure INTERRUPT (message instance m, port_info reference port)
    // an interrupt announcing the arrival of message m
    if port.in  – port.out  ≥ NMSG then            // is there space?
        return                                     // No, ignore message
    port.buffer[port.in modulo NMSG] ← m
    port.in ← port.in + 1
    UP(port.count)

procedure RECEIVE_MESSAGE (dest_port, port_info reference port)
1   ...
    DOWN(port.count)
    m ← port.buffer[port.out modulo NMSG]
    port.out ← port.out + 1
    return m
```

Q 12.1 Assume that there are no concurrent invocations of INTERRUPT and that there are no concurrent invocations of RECEIVE_MESSAGE on the same port. Which of the following statements is true about the implementation of INTERRUPT and RECEIVE_MESSAGE?

A. There are no race conditions between two threads that invoke RECEIVE_MESSAGE concurrently on different ports.
B. The complete execution of UP in INTERRUPT will not be interleaved between the statements labeled 15 and 16 in DOWN in Sidebar 5.7.
C. Because DOWN and UP are atomic, the processor instructions necessary for the subtracting of *sem* in DOWN and adding to *sem* in UP will not be interleaved incorrectly.
D. Because *in* and *out* may be shared between the interrupt handler running INTERRUPT and a thread calling RECEIVE_MESSAGE on the same port, it is possible for INTERRUPT to throw away a message, even though there is space in the bounded buffer.

Alyssa claims that semaphores can also be used to make operations atomic. She proposes the following addition to a *port_info* structure:

```
semaphore instance mutex initially ????          // see question below
```

and adds the following line to RECEIVE_MESSAGE on line 1 in the pseudocode above:

```
DOWN(port.mutex)                                 // enter atomic section
```

Alyssa argues that these changes allow threads to concurrently invoke RECEIVE_MESSAGE on the same port without race conditions, even if the kernel schedules threads preemptively.

Q 12.2 To what value can *mutex* be initialized (by replacing ???? with a number in the *semaphore* declaration) to avoid race conditions and deadlocks when multiple threads call RECEIVE_MESSAGE on the same port?

A. 0
B. 1
C. 2
D. -1

2006–1–11&12

13 The Single-Chip NC*

(Chapter 5)

Ben Bitdiddle plans to create a revolution in computing with his just-developed $15 single chip Network Computer, NC. In the NC network system the network interface thread calls the procedure MESSAGE_ARRIVED when a message arrives. The procedure WAIT_FOR_MESSAGE can be called by a thread to wait for a message. To coordinate the sequences in which threads execute, Ben deploys another commonly used coordination primitive: **condition variables**.

Part of the code in the NC is as follows:

```
1  lock instance m
2  boolean message_here
3  condition instance message_present
4
5  procedure MESSAGE_ARRIVED ()
6      message_here ← TRUE
7      NOTIFY_CONDITION (message_present) // notify threads waiting on this condition
8
9  procedure WAIT_FOR_MESSAGE ()
10     ACQUIRE (m)
11     while not message_here do
12         WAIT_CONDITION (message_present, m);// release m and wait
13     RELEASE (m)
```

The procedures ACQUIRE and RELEASE are the ones described in Chapter *5*. NOTIFY_CONDITION (*condition*) atomically wakes up all threads waiting for *condition* to become TRUE. WAIT_CONDITION (*condition*, *lock*) does several things atomically: it tests *condition*; if TRUE it returns; otherwise it puts the calling thread on the waiting queue for *condition* and releases *lock*. When NOTIFY_CONDITION wakens a thread, that thread becomes runnable, and when the scheduler runs that thread, WAIT_CONDITION reacquires *lock* (waiting, if necessary, until it is available) before returning to its caller.

Assume there are no errors in the implementation of condition variables.

Q 13.1 It is possible that WAIT_FOR_MESSAGE will wait forever even if a message arrives while it is spinning in the **while** loop. Give an execution ordering of the above statements that would cause this problem. Your answer should be a simple list such as 1, 2, 3, 4.

Q 13.2 Write new version(s) of MESSAGE_ARRIVED and/or WAIT_FOR_MESSAGE to fix this problem.

1998–1–3a/b

* Credit for developing this problem set goes to David K. Gifford.

14 Toastac-25*

(Chapters 5 and 7[on-line])

Louis P. Hacker bought a used Therac-25 (the medical irradation machine that was involved in several accidents [Suggestions for Further Reading 1.9.5]) for $14.99 at a yard sale. After some slight modifications, he has hooked it up to his home network as a computer-controllable turbo-toaster, which can toast one slice in under 2 milliseconds. He decides to use RPC to control the Toastac-25. Each toasting request starts a new thread on the server, which cooks the toast, returns an acknowledgment (or perhaps a helpful error code, such as "Malfunction 54"), and exits. Each server thread runs the following procedure:

```
procedure SERVER ()
    ACQUIRE (message_buffer_lock)
    DECODE (message)
    ACQUIRE (accelerator_buffer_lock)
    RELEASE (message_buffer_lock)
    COOK_TOAST ()
    ACQUIRE (message_buffer_lock)
    message ← "ack"
    SEND (message)
    RELEASE (accelerator_buffer_lock)
    RELEASE (message_buffer_lock)
```

Q 14.1 To his surprise, the toaster stops cooking toast the first time it is heavily used! What has gone wrong?

A. Two server threads might deadlock because one has *message_buffer_lock* and wants *accelerator_buffer_lock*, while the other has *accelerator_buffer_lock* and wants *message_buffer_lock*.
B. Two server threads might deadlock because one has *accelerator_buffer_lock* and *message_buffer_lock*.
C. Toastac-25 deadlocks because COOK_TOAST is not an atomic operation.
D. Insufficient locking allows inappropriate interleaving of server threads.

Once Louis fixes the multithreaded server, the Toastac gets more use than ever. However, when the Toastac has many simultaneous requests (i.e., there are many threads), he notices that the system performance degrades badly—much more than he expected. Performance analysis shows that competition for locks is not the problem.

Q 14.2 What is probably going wrong?

A. The Toastac system spends all its time context switching between threads.
B. The Toastac system spends all its time waiting for requests to arrive.
C. The Toastac gets hot, and therefore cooking toast takes longer.
D. The Toastac system spends all its time releasing locks.

---

* Credit for developing this problem set goes to Eddie Kohler.

Q 14.3 An upgrade to a supercomputer fixes that problem, but it's too late—Louis is obsessed with performance. He switches from RPC to an asynchronous protocol, which groups several requests into a single message if they are made within 2 milliseconds of one another. On his network, which has a very high transit time, he notices that this speeds up some workloads far more than others. Describe a workload that is sped up and a workload that is not sped up. (An example of a possible workload would be one request every 10 milliseconds.)

Q 14.4 As a design engineering consultant, you are called in to critique Louis's decision to move from RPC to asynchronous client/service. How do you feel about his decision? Remember that the Toastac software sometimes fails with a "Malfunction 54" instead of toasting properly.

1996–1–5c/d & 1999–1–12/13

15 BOOZE: Ben's Object-Oriented Zoned Environment

(Chapters 5 and 6)

Ben Bitdiddle writes a large number of object-oriented programs. Objects come in different sizes, but pages come in a fixed size. Ben is inspired to redesign his page-based virtual memory system (PAGE) into an object memory system. PAGE is a page-based virtual memory system like the one described in Chapter *5* with the extensions for multilevel memory systems from Chapter *6*. BOOZE is Ben's **object-based virtual memory** system.* Of course, he can run his programs on either system.

Each BOOZE object has a unique ID called a UID. A UID has three fields: a disk address for the disk block that contains the object; an offset within that disk block where the object starts; and the size of the object.

```
structure uid
    integer blocknr     // disk address for disk block
    integer offset      // offset within block blocknr
    integer size        // size of object
```

Applications running on BOOZE and PAGE have similar structure. The only difference is that on PAGE, program refer to objects by their virtual address, while on BOOZE programs refer to objects by UIDs.

The two levels of memory in BOOZE and PAGE are main memory and disk. The disk is a linear array of fixed-size blocks of 4 kilobytes. A disk block is addressed by its block number. In *both* systems, the transfer unit between the disk and main memory is a 4-kilobyte block. Objects don't cross disk block boundaries, are smaller than 4 kilobytes, and cannot change size. The page size in PAGE is equal to the disk block size; therefore, when an application refers to an object, PAGE will bring in all objects on the same page.

BOOZE keeps an object map in main memory. The object map contains entries that map a UID to the memory address of the corresponding object.

```
structure mapentry
    uid instance UID
    integer addr
```

On all references to an object, BOOZE translates a UID to an address in main memory. BOOZE uses the following procedure (implemented partially in hardware and partially

* Ben chose this name after reading a paper by Ted Kaehler, "Virtual memory for an object-oriented language" [Suggestions for Further Reading 6.1.4]. In that paper, Kaehler describes a memory management system called the Object-Oriented Zoned Environment, with the acronym OOZE.

in software) for translation:

```
procedure ObjectToAddress(UID) returns address
    addr ← isPresent(UID)              // is UID present in object map?
    if addr ≥ 0 then return addr       // UID is present, return addr
    addr ← findFreeSpace(UID.size)     // allocate space to hold object
    readObject(addr, UID)              // read object from disk & store at addr
    enterIntoMap(UID, addr)            // enter UID in object map
    return addr                        // return memory address of object
```

isPresent looks up *UID* in the object map; if present, it returns the address of the corresponding object; otherwise, it returns 1. findFreeSpace allocates free space for the object; it might evict another object to make space available for this one. readObject reads the *page* that contains the object, and then copies the *object* to the allocated address.

Q 15.1 What does *addr* in the *mapentry* data structure denote?

A. The memory address at which the object map is located.
B. The disk address at which to find a given object.
C. The memory address at which to find a given object that is *currently* resident in memory.
D. The memory address at which a given non-resident object *would have to be loaded*, when an access is made to it.

Q 15.2 In what way is BOOZE better than PAGE?

A. Applications running on BOOZE generally use less main memory because BOOZE stores only objects that are in use.
B. Applications running on BOOZE generally run faster because UIDs are smaller than virtual addresses.
C. Applications running on BOOZE generally run faster because BOOZE transfers objects from disk to main memory instead of complete pages.
D. Applications running on BOOZE generally run faster because typical applications will exhibit better locality of reference.

When findFreeSpace cannot find enough space to hold the object, it needs to write one or more objects back to the disk to create free space. findFreeSpace uses writeObject to write an object to the disk.

Ben is figuring out how to implement writeObject. He is considering the following options:

1.
```
procedure writeObject (addr, UID)
    write(addr, UID.blocknr, 4096)
```

2.
```
procedure writeObject(addr, UID)
    read(buffer, UID.blocknr, 4096)
    copy(addr, buffer + UID.offset, UID.size)
    write(buffer, UID.blocknr, 4096)
```

read (*mem_addr*, *disk_addr*, 4096) and write (*mem_addr*, *disk_addr*, 4096) read and write a 4-kilobyte page from/to the disk. copy (*source*, *destination*, *size*) copies *size* bytes from a source address to a destination address in main memory.

Q 15.3 Which implementation should Ben use?

A. Implementation 2, since implementation 1 is incorrect.
B. Implementation 1, since it is more efficient than implementation 2.
C. Implementation 1, since it is easier to understand.
D. Implementation 2, since it will result in better locality of reference.

Ben now turns his attention to optimizing the performance of BOOZE. In particular, he wants to reduce the number of writes to the disk.

Q 15.4 Which of the following techniques will reduce the number of writes without losing correctness?

A. Prefetching objects on a read.
B. Delaying writes to disk until the application finishes its computation.
C. Writing to disk only objects that have been modified.
D. Delaying a write of an object to disk until it is accessed again.

Ben decides that he wants even better performance, so he decides to modify FINDFREESPACE. When FINDFREESPACE has to evict an object, it now tries not to write an object modified in the last 30 seconds (in the belief that it may be used again soon). Ben does this by setting the *dirty* flag when the object is modified. Every 30 seconds, BOOZE calls a procedure WRITE_BEHIND that walks through the object map and writes out all objects that are dirty. After an object has been written, WRITE_BEHIND clears its *dirty* flag. When FINDFREESPACE needs to evict an object to make space for another, clean objects are the *only* candidates for replacement.

When running his applications on the latest version of BOOZE, Ben observes once in a while that BOOZE runs out of physical memory when calling OBJECTTOADDRESS for a new object.

Q 15.5 Which of these strategies avoids the above problem?

A. When FINDFREESPACE cannot find any clean objects, it calls WRITE_BEHIND and then tries to find clean objects again.
B. BOOZE could call WRITE_BEHIND every 1 second instead of every 30 seconds.
C. When FINDFREESPACE cannot find any clean objects, it picks *one* dirty object, writes the block containing the object to the disk, clears the *dirty* flag, and then uses that address for the new object.
D. All of the above strategies.

1999–1–7…11

16 OutOfMoney.com

(Chapter 6, with a bit of Chapter 4)

OutOfMoney.com has decided it needs a real product, so it is laying off most of its Marketing Department. To replace the marketing folks, and on the advice of a senior computer expert, OutOfMoney.com hires a crew of 16-year-olds. The 16-year-olds get together and decide to design and implement a video service that serves MPEG-1 video, so that they can watch Britney Spears on their computers in living color.

Since time to market is crucial. Mark Bitdiddle—Ben's 16-year-old kid brother, who is working for OutOfMoney—surfs the Web to find some code from which they can start. Mark finds some code that looks relevant, and he modifies it for OutOfMoney's video service:

```
procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← GET_FILE_FROM_DISK (request)
        REPLY (file)
```

The SERVICE procedure waits for a message from a client to arrive on the network. The message contains a *request* for a particular file. The procedure GET_FILE_FROM_DISK reads the file from disk into the memory location *file*. The procedure REPLY sends the file from memory in a message back to the client.

(In the pseudocode, undeclared variables are local variables of the procedure in which they are used, and the variables are thus stored on the stack or in registers.)

Mark and his 16-year-old buddies also write code for a network driver to SEND and RECEIVE network packets, a simple file system to PUT and GET files on a disk, and a loader for booting a machine. They run their code on the bare hardware of an off-the-shelf personal computer with one disk, one processor (a Pentium III), and one network interface card (1 gigabit per second Ethernet). After the machine has booted, it starts one thread running SERVICE.

The disk has an average seek time of 5 milliseconds, a complete rotation takes 6 milliseconds, and its throughput is 10 megabytes per second when no seeks are required.

All files are 1 gigabyte (roughly a half hour of MPEG-1 video). The file system in which the files are stored has no cache, and it allocates data for a file in 8-kilobyte chunks. It pays no attention to file layout when allocating a chunk; as a result, disk blocks of the same file can be all over the disk. A 1-gigabyte file contains 131,072 8-kilobyte blocks.

Q 16.1 Assuming that the disk is the main bottleneck, how long does the service take to serve a file?

Mark is shocked about the performance. Ben suggests that they should add a cache. Mark, impressed by Ben's knowledge, follows his advice and adds a 1-gigabyte cache, which can hold one file completely:

```
cache [1073741824]        // 1-gigabyte cache

procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← LOOK_IN_CACHE (request)
        if file = NULL then
            file ← GET_FILE_FROM_DISK (request)
            ADD_TO_CACHE (request, file)
        REPLY (file)
```

The procedure LOOK_IN_CACHE checks whether the file specified in the request is present in the cache and returns it if present. The procedure ADD_TO_CACHE copies a file to the cache.

Q 16.2 Mark tests the code by asking once for every video stored. Assuming that the disk is the main bottleneck (serving a file from the cache takes 0 milliseconds), what is now the average time for the service to serve a file?

Mark is happy that the test actually returns every video. He reports back to the only person left in the Marketing Department that the prototype is ready to be evaluated. To keep the investors happy, the marketing person decides to use the prototype to run OutOfMoney's Web site. The one-person Marketing Department loads the machine up with videos and launches the new Web site with a big PR campaign, blowing their remaining funding.

Seconds after they launch the Web site, OutOfMoney's support organization (also staffed by 16-year-olds) receives e-mail from unhappy users saying that the service is not responding to their requests. The support department measures the load on the service CPU and also the service disk. They observe that the CPU load is low and the disk load is high.

Q 16.3 What is the most likely reason for this observation?

A. The cache is too large.
B. The hit ratio for the cache is low.
C. The hit ratio for the cache is high.
D. The CPU is not fast enough.

The support department beeps Mark, who runs to his brother Ben for help. Ben suggests using the example thread package of Chapter 5. Mark augments the code to use the thread package and after the system boots, it starts 100 threads, each running SERVICE:

```
for i from 1 to 100 do CREATE_THREAD (SERVICE)
```

In addition, Mark modifies RECEIVE_MESSAGE and GET_FILE_FROM_DISK to release the processor by calling YIELD when waiting for a new message to arrive or waiting for the disk to complete a disk read. In no other place does his code release the processor. The implementation of the thread package is non-preemptive.

To take advantage of the threaded implementation, Mark modifies the code to read blocks of a file instead of complete files. He also runs to the store and buys some more memory so he can increase the cache size to 4 gigabytes. Here is his latest effort:

```
cache [4 x 1073741824]              // The 4-gigabyte cache, shared by all threads.

procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← NULL
        for k from 1 to 131072 do
            block ← LOOK_IN_CACHE (request, k)
            if block = NULL then
                block ← GET_BLOCK_FROM_DISK (request, k)
                ADD_TO_CACHE (request, block, k)
            file ← file + block          // + concatenates strings
        REPLY (file)
```

The procedure LOOK_IN_CACHE (*request*, *k*) checks whether block *k* of the file specified in *request* is present; if the block is present, it returns it. The procedure GET_BLOCK_FROM_DISK reads block *k* of the file specified in *request* from the disk into memory. The procedure ADD_TO_CACHE adds block *k* from the file specified in *request* to the cache.

Mark loads up the service with one video. He retrieves the video successfully. Happy with this result, Mark sends many requests for the single video in parallel to the service. He observes no disk activity.

Q 16.4 Based on the information so far, what is the most likely explanation why Mark observes no disk activity?

Happy with the progress, Mark makes the service ready for running in production mode. He is worried that he may have to modify the code to deal with concurrency—

his past experience has suggested to him that he needs an education, so he is reading Chapter 5. He considers protecting ADD_TO_CACHE with a lock:

```
lock instance cachelock// A lock for the cache

procedure SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE ()
        file ← NULL
        for k from 1 to 131072 do
            block ← LOOK_IN_CACHE (request, k)
            if block = NULL then
                block ← GET_BLOCK_FROM_DISK (request, k)
                ACQUIRE (cachelock)          // use the lock
                ADD_TO_CACHE (request, block, k)
                RELEASE (cachelock)          // here, too
            file ← file + block
        REPLY (file)
```

Q 16.5 Ben argues that these modifications are not useful. Is Ben right?

Mark doesn't like thinking, so he upgrades OutOfMoney's Web site to use the multithreaded code with locks. When the upgraded Web site goes live, Mark observes that most users watch the same three videos, while a few are watching other videos.

Q 16.6 Mark observes a hit-ratio of 90% for blocks in the cache. Assuming that the disk is the main bottleneck (serving blocks from the cache takes 0 milliseconds), what is the average time for SERVICE to serve a single movie?

Q 16.7 Mark loads a new Britney Spears video onto the service and observes operation as the first users start to view it. It is so popular that no users are viewing any other video. Mark sees that the first batch of viewers all start watching the video at about the same time. He observes that the service threads all read block 0 at about the same time, then all read block 1 at about the same time, and so on. For this workload what is a good cache replacement policy?

A. Least-recently used.
B. Most-recently used.
C. First-in, first-out.
D. Last-in, first-out.
E. The replacement policy doesn't matter for this workload.

The Marketing Department is extremely happy with the progress. Ben raises another round of money by selling his BMW and launches another PR campaign. The number of users dramatically increases. Unfortunately, under high load the machine stops serving requests and has to be restarted. As a result, some users have to restart their videos from the beginning, and they call up the support department to complain. The problem appears to be some interaction between the network driver and the service threads. The driver and service threads share a fixed-sized input buffer that can hold 1,000 request

messages. If the buffer is full and a message arrives, the driver drops the message. When the card receives data from the network, it issues an interrupt to the processor. This interrupt causes the network driver to run immediately on the stack of the currently running thread. The code for the driver and RECEIVE_MESSAGE is as follows:

```
buffer[1000]
lock instance bufferlock

procedure DRIVER ()
    message ← READ_FROM_INTERFACE ()
    ACQUIRE (bufferlock)
    if SPACE_IN_BUFFER () then ADD_TO_BUFFER (message)
    else DISCARD_MESSAGE (message)
    RELEASE (bufferlock)

procedure RECEIVE_MESSAGE ()
    while BUFFER_IS_EMPTY () do YIELD ()
    ACQUIRE (bufferlock)
    message ← REMOVE_FROM_BUFFER ()
    RELEASE (bufferlock)
    return message

procedure INTERRUPT ()
    DRIVER ()
```

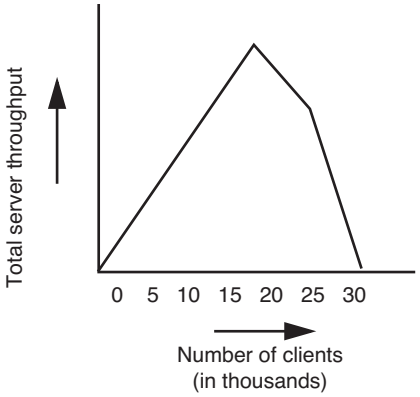
Q 16.8 Which of the following could happen under high load?

A. Deadlock when an arriving message interrupts DRIVER.
B. Deadlock when an arriving message interrupts a thread that is in RECEIVE_MESSAGE.
C. Deadlock when an arriving message interrupts a thread that is in REMOVE_FROM_BUFFER.
D. RECEIVE_MESSAGE misses a call to YIELD when the buffer is not empty because it can be interrupted between the BUFFER_IS_EMPTY test and the call to YIELD.

Q 16.9 What fixes should Mark implement?

A. Delete all the code dealing with locks.
B. DRIVER should run as a separate thread, to be awakened by the interrupt.
C. INTERRUPT and DRIVER should use an eventcount for sequence coordination.
D. DRIVER shouldn't drop packets when the buffer is full.

Mark eliminates the deadlock problems and, to attract more users, announces the availability of a new Britney Spears video. The news spreads rapidly and an enormous number of requests for this one video start hitting the service. Mark measures the throughput of the service as more and more clients ask for the video. The resulting graph is plotted at the right. The throughput first increases while the number of clients increases, then reaches a maximum value, and finally drops off.



Q 16.10 Why does the throughput decrease with a large number of clients?

A. The processor spends most of its time taking interrupts.
B. The processor spends most of its time updating the cache.
C. The processor spends most of its time waiting for the disk accesses to complete.
D. The processor spends most of its time removing messages from the buffer.

2001–1–6…15

17 Quarria*

(Chapters 4, 6, and 7[on-line])

Quarria is a new country formed on a 1 kilometer rock island in the middle of the Pacific Ocean. The founders have organized the Quarria Stock Market in order to get the economy rolling. The stock market is very simple, since there is only one stock to trade (that of the Quarria Rock Company). Moreover, due to local religious convictions, the price of the stock is always precisely the wind velocity at the highest point on the island. Rocky, Quarria's president, proposes that the stock market be entirely network based. He suggests running the stock market from a server machine, and requiring each investor to have a separate client machine which makes occasional requests to the server using a simple RPC protocol. The two remote procedures Rocky proposes supporting are

- BALANCE(): requests that the server return the cash balance of a client's account. This service is very fast, requiring a simple read of a memory location.
- TRADE*(nshares)*: requests that *nshares* be bought (assuming *nshares* is positive) or *nshares* be sold (if *nshares* is negative) at the current market price. This service is potentially slow, since it potentially involves network traffic in order to locate a willing trade partner.

Quarria implements a simple RPC protocol in which a client sends the server a request message with the following format:

```
structure Request
    integer Client      // Unique code for the client
    integer Opcode      // Code for operation requested
    integer Argument    // integer argument, if any
    integer Result      // integer return value, if any
```

The server replies by sending back the same message, with the *Result* field changed. We assume that all messages fit in one packet, that link- and network-layer error checking detect and discard garbled packets, and that Quarria investors are scrupulously honest; thus any received message was actually sent by some client (although sent messages might get lost).

Q 17.1 Is this RPC design appropriate for a connectionless network model, or is a connection-based model assumed?

The client RPC stub blocks the client thread until a reply is received, but includes a timer expiration allowing any client RPC operation to return with the error code TIME_EXPIRED if no response is heard from the server after *Q* seconds.

Q 17.2 Give a reason for preferring returning a TIME_EXPIRED error code over simply having the RPC operation block forever.

---

* Credit for developing this problem set goes to Stephen A. Ward.

Q 17.3 Give a reason for preferring returning a TIME_EXPIRED error code over having the RPC stub transparently retransmit the message.

Q 17.4 Suppose you can bound the time taken for a request, including network and server time, at 3 seconds. What advantage is there to setting the expiration time, *Q*, to 4 seconds instead of 2 seconds?

Unfortunately, no such bound exists for Quarria's network.

Q 17.5 What complication does client message retransmission introduce into the RPC semantics, in the absence of a time bound?

Rocky's initial implementation of the server is as follows:

```
integer Cash[1000]                          // Cash balance of each client
integer Shares[1000]                        // Stock owned by each client

procedure SERVER ()
    Request instance req, rep               // Pointer to request message
    do forever              // loop forever...
        req ← GETNEXTREQUEST ()             // take next incoming request,
        if req.Opcode = 1 then              // …and dispatch on opcode.
            rep ← BALANCE (req)             // Request 1: return balance
            SEND (rep)
        if req.Opcode = 2 then              // Request 2: buy/sell stock
            rep ← TRADE (req);
            SEND (rep)

// Process a BALANCE request...
procedure BALANCE (Request instance req)
    client ← req.Client                     // Get client number from request
    req.Result ← Cash[client]               // Return his cash balance
    return req                              // and return reply.

// Perform a trade: buy/sell Argument/-Argument shares of stock and
// return the total number of shares owned after trade.
procedure TRADE (Request instance req)
    client ← req.Client                     // The client who is requesting
    p ←STOCKPRICE ()                        // Price, using network requests
    nshares ← req.Argument// Number of shares to buy/sell
    actual ← CONFIRMTRADE (req, p, nshares) // See how many shares we can trade
    Cash[client]← Cash[client] + p x actual // Update our records
    Shares[client] ← Shares[client] + actual
    req.Result ← actual
    return req
```

Note that CONFIRMTRADE uses network communication to check on available shares, executes the trade, and returns the number of shares that have actually been bought or sold.

Rocky tests this implementation on a single server machine by having clients scattered around the island sending BALANCE requests as fast as they can. He discovers that after

some point adding more clients doesn't increase the throughput—the server throughput tops out at 1000 requests per second.

Q 17.6 Rocky is concerned about performance, and hires you to recommend steps for improvement. Which, if any, of the following steps might significantly improve Rocky's measured 1000 BALANCE requests per second?

A. Use faster client machines.
B. Use multiple client threads (each making Balance requests) on each client.
C. Use a faster server machine.
D. Use faster network technology.

Stone Galore, a local systems guru, has another suggestion to improve the performance generally. He proposes multithreading the server, replacing calls to service procedures like

```
BALANCE (req)                    // Run BALANCE, to service request
```

with

```
CREATE_THREAD (BALANCE, req)     // create thread to run BALANCE (req)
```

The CREATE_THREAD primitive creates a new thread, runs the supplied procedure (in this case BALANCE) in that thread, and deactivates the thread on completion. Stone's thread implementation is preemptive.

Stone changes the appropriate three lines of the original code according to the above model, and retries the experiment with BALANCE requests. He now measures a maximum server throughput of 500 requests per second.

Q 17.7 Explain the performance degradation.

Q 17.8 Is there an advantage to the use of threads in other requests? Explain.

Q 17.9 Select the best advice for Rocky regarding server threads:

A. Don't use threads; stick with your original design.
B. Don't use threads for Balance requests, but use them for other requests.
C. Continue using them for all requests; the benefits outweigh the costs.

Independently of your advice, Stone is determined to stick with the multithreaded implementation.

Q 17.10 Should the code for TRADE be changed to reflect the fact that it now operates in a multithreaded server environment? Explain, suggesting explicit changes as necessary.

Q 17.11 What if the client is multithreaded and can have multiple request outstanding? Should the code for TRADE be changed? Explain, suggesting explicit changes as necessary.

Rocky decides that multithreaded clients are complicated and abandons that idea. He hasn't read about RPC in Chapter 4, and isn't sure whether his server requires at-most-once RPC semantics.

Q 17.12 Which of the requests require at-most-once RPC semantics? Explain.

Q 17.13 Suggest how one might modify Rocky's implementation to guarantee at-most-once semantics. Ignore the possibility of crashes, but consider lost messages and retransmissions.

1997–1–2a…m

18 PigeonExpress!.com I

(Chapter 7[on-line])

Ben Bitdiddle cannot believe the high valuations of some Internet companies, so he is doing a startup, PigeonExpress!.com, which provides high-performance networking using pigeons. Ben's reasoning is that it is cheaper to build a network using pigeons than it is to dig up streets to lay new cables. Although there is a standard for transmitting Internet datagrams with avian carriers (see network RFC 1149) it is out of date, and Ben has modernized it.

When sending a pigeon, Ben's software prints out a little header on a sticky label and also writes a compact disk (CD) containing the data. Someone sticks the label on the disk and gives it to the pigeon. The header on the label contains the Global Positioning System (GPS) coordinates of the destination and the source (the point where the pigeon is taking off), a type field indicating the kind of message (REQUEST or ACKNOWLEDGMENT), and a sequence number:

```
structure header
   GPS source
   GPS destination
   integer type
   integer sequence_no
```

The CD holds a maximum of 640 megabytes of data, so some messages will require multiple CD's. The pigeon reads the header and delivers the labeled CD to the destination. The header and data are never corrupted and never separated. Even better, for purposes of this problem, computers don't fail. However, pigeons occasionally get lost, in which case they never reach their destination.

To make life tolerable on the pigeon network, Ben designs a simple end-to-end protocol (Ben's End-to-End Protocol, BEEP) to ensure reliable delivery. Suppose that there is a single sender and a single receiver. The sender's computer executes the following code:

```
shared next_sequence initially 0          // a global sequence number, starting at 0.

procedure BEEP (destination, n, CD[])     // send n CDs to destination
    header h                              // h is an instance of header.
    nextCD ← 0
    h.source ← MY_GPS                     // set source to my GPS coordinates
    h.destination ← destination           // set destination
    h.type ← REQUEST                      // this is a request message
    while nextCD < n do                   // send the CDs
        h.sequence_no ← next_sequence     // set seq number for this CD
        send pigeon with h, CD[nextCD]    // transmit
        wait 2,000 seconds
```

Pending and incoming acknowledgments are processed *only* when the sender is waiting:

```
procedure PROCESS_ACK (h)                 // process acknowledgment
    if h.sequence_no = sequence then      // ack for current outstanding CD?
        next_sequence ← next_sequence + 1
        nextCD ← nextCD + 1               // allow next CD to be sent
```

The receiver's computer executes the following code. The arrival of a request triggers invocation of PROCESS_REQUEST:

```
procedure PROCESS_REQUEST (h, CD)         // process request
    PROCESS (CD)                          // process the data on the CD
    h.destination ← h.source              // send to where the pigeon came from
    h.source ← MY_GPS
    h.sequence_no ←h.sequence_no          // unchanged
    h.type ← ACKNOWLEDGMENT
    send pigeon with h                    // send an acknowledgment back
```

Q 18.1 If a pigeon travels at 100 meters per second (these are express pigeons!) and pigeons do not get lost, then what is the maximum data rate observed by the caller of BEEP on a 50,000 meter (50 kilometer) long pigeon link? Assume that the processing delay at the sender and receiver are negligible.

Q 18.2 Does at least one copy of each CD make it to the destination, even though some pigeons are lost?

A. Yes, because *nextCD* and *next_sequence* are incremented only on the arrival of a matching acknowledgment.
B. No, since there is no explicit loss-recovery procedure (such as a timer expiration procedure).
C. No, since both request and acknowledgments can get lost.
D. Yes, since the next acknowledgment will trigger a retransmission.

Q 18.3 To guarantee that each CD arrives at most once, what is required?

A. We must assume that a pigeon for each CD has to arrive eventually.
B. We must assume that acknowledgment pigeons do not get lost and must arrive within 2,000 seconds after the corresponding request pigeon is dispatched.
C. We must assume request pigeons must never get lost.
D. Nothing. The protocol guarantees at-most-once delivery.

Q 18.4 Ignoring possible duplicates, what is needed to guarantee that CDs arrive in order?

A. We must assume that pigeons arrive in the order in which they were sent.
B. Nothing. The protocol guarantees that CDs arrive in order.
C. We must assume that request pigeons never get lost.
D. We must assume that acknowledgment pigeons never get lost.

To attract more users to PigeonExpress!, Ben improves throughput of the 50 kilometer long link by using a window-based flow-control scheme. He picks *window* (number of CDs) as the window size and rewrites the code. The code to be executed on the sender's computer now is:

```
procedure BEEP (destination, n, CD[])          // send n CDs to destination
    nextCD ← 0
    window ← 10                                // initial window size is 10 CDs
    h.source ← MY_GPS                          // set source to my GPS coordinates
    h.destination ← destination                // set destination to the destination
    h.type ← REQUEST                           // this is a request message
    while nextCD < n do                        // send the CDs
        CDsleft ← n - nextCD
        temp ← FOO (CDsleft, window) // FOO computes how many pigeons to send
        for k from 0 to (temp - 1) do
            h.sequence_no ← next_sequence;  // set seq number for this CD
            send pigeon with h, CD[nextCD + k]     // transmit
        wait 2,000 seconds
```

The procedures PROCESS_ACK and PROCESS_REQUEST are unchanged.

Q 18.5 What should the procedure FOO compute in this code fragment?

A. minimum.
B. maximum.
C. sum.
D. absolute difference.

Q 18.6 Alyssa looks at the code and tells Ben it may lose a CD. Ben is shocked and disappointed. What should Ben change to fix the problem?

A. Nothing. The protocol is fine; Alyssa is wrong.
B. Ben should modify PROCESS_REQUEST to accept and process CDs in the order of their sequence numbers.
C. Ben should set the value of *window* to the delay × bandwidth product.
D. Ben should ensure that the sender sends at least one CD after waiting for 2,000 seconds.

Q 18.7 Assume pigeons do not get lost. Under what assumptions is the observed data rate for the window-based BEEP larger than the observed data rate for the previous BEEP implementation?

A. The time to process and launch a request pigeon is less than 2,000 seconds;
B. The sender and receiver can process more than one request every 2,000 seconds;
C. The receiver can process less than one pigeon every 2,000 seconds;

After the initial success of PigeonExpress!, the pigeons have to travel farther and farther, and Ben notices that more and more pigeons don't make it to their destinations because they are running out of food. To solve this problem, Ben calls up a number of his friends in strategic locations and asks each of them to be a hub, where pigeons can reload on food.

To keep the hub design simple, each hub can feed one pigeon per second and each hub has space for 100 pigeons. Pigeons feed in first-in, first-out order at a hub. If a pigeon arrives at a full hub, the pigeon gets lucky and retires from PigeonExpress!. The hubs run a patented protocol to determine the best path that pigeons should travel from the source to the destination.

Q 18.8 Which layer in the reference model of Chapter *7[on-line]* provides functions most similar to the system of hubs?

A. the end-to-end layer
B. the network layer
C. the link layer
D. network layer and end-to-end layer
E. the feeding layer

Q 18.9 Assume Ben is using the window-based BEEP implementation. What change can Ben make to this BEEP implementation in order to make it respond gracefully to congested hubs?

A. Start with a window size of 1 and increase it by 1 upon the arrival of each acknowledgment.
B. Have PROCESS_REQUEST delay acknowledgments and have a single pigeon deliver multiple acknowledgments.
C. Use a window size smaller than 100 CDs, since the hub can hold 100 pigeons.
D. Use multiplicative decrease and additive increase for the window size.

1999–2–7…15

19 Monitoring Ants

(Chapter 7[on-line] with a bit of Chapter 11[on-line])

Alice has learned that ants are a serious problem in the dorms. To monitor the ant population she acquires a large shipment of motes. Motes are tiny self-powered computers the size of a grain of sand, and they have wireless communication capability. She spreads hundreds of motes in her dorm, planning to create an **ad hoc wireless network**. Each mote can transmit a packet to another mote that is within its radio range. Motes forward packets containing messages on behalf of other motes to form a network that covers the whole dorm. The exact details of how this network of motes works are our topic.

Each mote runs a small program that every 1 millisecond senses if there are ants nearby. Each time the program senses ants, it increments a counter, called `SensorCount`. Every 16 milliseconds the program sends a message containing the value of `SensorCount` and the mote's identifier to the mote connected to Alice's desktop computer, which has identifier `A`. After sending the message, the mote resets `SensorCount`. All messages are small enough to fit into a single packet.

Only the radio consumes energy. The radio operates at a speed of 19.2 kilobits per second (using a 916.5 megahertz transceiver). When transmitting the radio draws 12 milliamperes at 3 volts DC. Although receiving draws 4.5 millamperes, for the moment assume that receiving is uses no power. The motes have a battery rated at 575 milliamperehours (mAh).

Q 19.1 If a mote transmits continuously, about how long will it be until its battery runs down?

Q 19.2 How much energy (voltage x current x time) does it take to transmit a single bit (1 watt-second = 1 joule)?

Because the radio range of the motes is only ten meters, the motes must cooperate to form a network that covers Alice's dorm. Motes forward packets on behalf of other motes to provide connectivity to Alice's computer, `A`. To allow the motes to find paths and to adapt to changes in the network (e.g., motes failing because their batteries run down), the motes run a routing protocol. Alice has adopted the path-vector routing protocol from Chapter 7. Each mote runs the following routing algorithm, which finds paths to mote A:

```
n ← MYID
if n = A then path ← []
else path ← NULL

procedure ADVERTISE ()
    if path ≠ NULL then TRANSMIT ({n, path})          // send marshaled message

procedure RECEIVE (p)
    if n in p then return
    else if (path = NULL) or (FIRST (p) = FIRST(path)) or (LENGTH (p) < LENGTH(path)
        then path ← p

procedure TIMER ()
    if HAVENOTHEARDFROMRECENTLY (FIRST (path)) then path ← NULL
```

When a mote starts it initializes its variables *n* and *path*. Each mode has a unique ID, which the mote stores in the local variable *n*. Each mote stores its path to *A* into the *path* variable. A path contains a list of mote IDs; the last element of this list is *A*. The first element of the list (FIRST **(***path*)) is the first mote on the path to *A*. When a mote starts it sets path to NULL, except for Alice's mote, which sets path to the empty path ([]).

Every *t* seconds a mote creates a path that contains its own ID concatenated with the value of path, and transmits that path (see ADVERTISE) using its radio. Motes in radio range may receive this packet. Motes outside radio range will not receive this packet. When a mote receives a routing packet, it invokes the procedure RECEIVE with the argument set to the path *p* stored in the routing packet. If *p* contains *n*, then this routing packet circled back to *n* and the procedure RECEIVE just returns, rejecting the path. Otherwise, RECEIVE updates path in three cases:

- if *path* is NULL, because *n* doesn't have any path to *A* yet.
- if the first mote on *path* is the mote from which we are receiving *p*, because that mote might have a new path to *A*.
- if *p* is a shorter path to *A* than *path*. (Assume that shorter paths are better.)

A mote also has a timer. If the timer goes off, it invokes the procedure TIMER. If since the last invocation of TIMER a mote hasn't heard from the mote at the head of the path to *A*, it resets *path* to NULL because apparently the first node on *path* is no longer reachable.

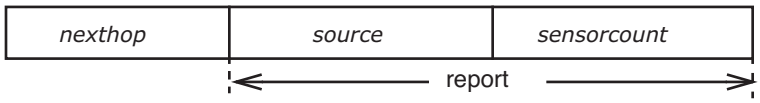
The forwarding protocol uses the paths found by the routing protocol to forward reports from a mote to *A*. Since *A* may not be in radio range, a report packet may have to be forwarded through several motes to reach *A*. This forwarding process works as follows:

```
structure report
    id
    data

procedure SEND (counter)
    report.id ← n
    report.data ← counter
    if path ≠ NULL then TRANSMIT (FIRST (path), report);

procedure FORWARD (nexthop, report)
    if n ≠ nexthop then return
    if n = A then DELIVER (report)
    else TRANSMIT (FIRST (path), report)
```

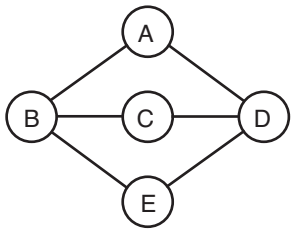
The procedure SEND creates a report (the mote's ID and its current counter value) and transmits a report packet. The report packet contains the first hop on *path*, that is FIRST (*path*), and the report:



The *nexthop* field contains the ID of the mote to which this packet is directed. The *source* field contains the ID of the mote that originated the packet. The *sensorcount* field contains the sensor count. (If *path* is NULL, SEND has no mote to forward the report to so the mote drops the packet.)

When a mote receives a report packet, it calls FORWARD. If a mote receives a report packet for which it is not the next hop, it ignores the packet. If a report packet has reached its final destination *A*, FORWARD delivers it to Alice's computer. Otherwise, the mote forwards the report by setting *nexthop* to the first mote on its path variable. This process repeats until the report reaches *A*, and the packet is delivered.

Suppose we have the following arrangement of motes:



The circles represent motes with their node IDs. The edges connect motes that are in radio range of one another. For example, when mote A transmits a packet, it may be received by B and D, but not by C and E.

Packets may be lost and motes may also fail (e.g., if their batteries run down). If a mote fails, it just stops sensing, transmitting, and receiving packets.

Q 19.3 If motes may fail and if packets may be lost, which of the following values could the variable *path* at node E have?

A. NULL
B. [B, C]
C. [D, A]
D. [B, A]
E. [B, C, D, A]
F. [C, D, A]
G. [D, A, B, C, D, A]

Q 19.4 If no motes fail and packets are not lost, what properties hold for Alice's routing and forwarding protocols with the given arrangement of motes?

A. Every mote's *path* variable will contain a path to A.
B. After the routing algorithm reaches a stable state, every mote's path variable will contain a shortest path (in hops) to A.
C. After the routing algorithm reaches a stable state, the routing algorithm may have constructed a forwarding cycle.
D. After the routing algorithm reaches a stable state, the longest path is two hops.

Q 19.5 If packets may be lost but motes don't fail, what properties hold for Alice's routing and forwarding protocols with the given arrangement of motes?

A. Every mote's path variable will contain a path to A.
B. The routing algorithm may construct forwarding cycles.
C. The routing algorithm may never reach a stable state.
D. When a mote sends a report packet using send, the report may or may not reach A.

A report is 13 bits: an 8-bit node ID and a 5-bit counter. A report packet is 21 bits: an 8-bit next hop and a report. Assume that your answer to question *Q 19.2* (the energy for transmitting one bit) is j joules. Further assume that to start the radio for transmission takes s joules. Thus, transmitting a packet with r bits takes $s + r \times j$ joules.

Q 19.6 Assuming the routing algorithm reaches a stable state, no node and packet failures, how much total energy does it take for every node to send one report to *A* (ignoring routing packets)?

Q 19.7 Which of the following changes to the Alice's system will reduce the amount of energy needed for each node to send one report to A?

A. Add a nonce to a report so that Alice's computer can detect duplicates.
B. Delay forwarding packets and piggyback them on a report packet from this mote.
C. Use 4-bit node IDs.
D. Use a stop-and-wait protocol to transmit each report reliably from one mote to the next.

The following question is based on Chapter 11[on-line].

To be able to verify the integrity of the reports, Alice creates for each mote a public key pair. She manually loads the private key for mote *i* into mote *i*, and keeps the corresponding public keys on her computer. A mote signs the contents of the report (the counter and the source) with its private key and then transmits it.

Thus, a signed report consists of a 5-bit counter, a node ID, and a signature. When Alice's computer receives a signed report, it verifies the signed report and rejects the ones that don't check out.

Q 19.8 Assuming that the private keys on the motes are not compromised, and the SIGN and VERIFY procedures are correctly implemented, which of the following properties hold for Alice's plan?

A. A mote that forwards a report is not able to read the report's content.
B. A mote that forwards a report may be able to duplicate a report without Alice's computer rejecting the duplicated report.
C. A mote that forwards a report can use its private key to verify the report.
D. When Alice receives a report for which VERIFY returns ACCEPT, she knows that the report was signed by the mote that is listed in the report and that the report is fresh.

2003–2–5…12

20 Gnutella: Peer-to-Peer Networking

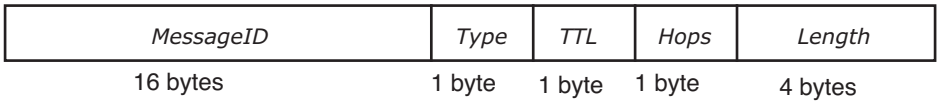
(Chapters 7[on-line] and 11[on-line])

Ben Bitdiddle is disappointed that the music industry is not publishing his CD, a rap production based on this textbook. Ben is convinced there is a large audience for his material. Having no alternative, he turns his CD into a set of MP3 files, the digital music standard understood by music playing programs, and publishes the songs through Gnutella.

Gnutella is a distributed file sharing application for the Internet. A user of Gnutella starts a Gnutella node, which presents a user interface to query for songs, talks to other nodes, and makes files from its local disk available to other remote users. The Gnutella nodes form what is called an **overlay network** on top of the existing Internet. The nodes in the overlay network are Gnutella nodes and the links between them are TCP connections. When a node starts it makes a TCP connection to various other nodes, which are connected through TCP connections to other nodes. When a node sends a message to another node, the message travels over the connections established between the nodes. Thus, a message from one node to another node travels through a number of intermediate Gnutella nodes.

To find a file, the user interface on the node sends a query (e.g., "System Design Rap") through the overlay network to other nodes. While the search propagates through the Gnutella network, nodes that have the desired file send a reply, and the user sees a list filling with file names that match the query. Both the queries and their replies travel through the overlay network. The user then selects one of the files to download and play. The user's node downloads the file directly from the node that has the file, instead of through the Gnutella network.

The format of the header of a Gnutella message is:



This header is followed by the payload, which is Length bytes long. The main message Types in the Gnutella protocol are:

- PING: A node finds additional Gnutella nodes in the network using PING messages. A node wants to be connected to more than one other Gnutella node to provide a high degree of connectivity in the case of node failures. Gnutella nodes are not very reliable because a user might turn off his machine running a Gnutella node at any time. PING messages have no payload.
- PONG: A node responds by sending a PONG message via the Gnutella network whenever it receives a PING message. The PONG message has the same *MessageID* as the corresponding PING message. The payload of the PONG message is the Internet address of the node that is responding to the PING Message.

- QUERY: Used to search the Gnutella network for files; its payload contains the query string that the user typed.
- QUERYHIT: A node responds by sending a QUERYHIT message via the Gnutella network if it has a file that matches the query in a QUERY message it receives. The payload contains the Internet address of the node that has the file, so that the user's node can connect directly to the node that has the song and download it. The QUERYHIT message has the same *MessageID* as the corresponding QUERY message.

(The Gnutella protocol also has a PUSH message to deal with firewalls and network address translators, but we will ignore it.)

In order to join the Gnutella network, the user must discover and configure the local node with the addresses of one or more existing nodes. The local node connects to those nodes using TCP. Once connected, the node uses PING messages to find more nodes (more detail below), and then directly connects to some subset of the nodes that the PING message found.

For QUERY and PING messages, Gnutella uses a kind of broadcast protocol known as **flooding**. Any node that receives a PING or a QUERY message forwards that message to all the nodes it is connected to, except the one from which it received the message. A node decrements the *TTL* field and increments the *Hops* field before forwarding the message. If after decrementing the *TTL* field, the *TTL* field is zero, the node does not forward the message at all. The *Hops* field is set to zero by the originating user's node.

To limit flooding and to route PONG and QUERYHIT messages, a node maintains a message table, indexed by *MessageID* and *Type*, with an entry for each message seen recently. The entry also contains the Internet address of the Gnutella node that forwarded the message to it. The message table is used as follows:

- If a PING or QUERY message arrives and there is an entry in the message table with the same *messageID* and *Type*, then the node discards that message.
- For a QUERYHIT or PONG message for which there is a corresponding QUERY or PONG entry with the same *messageID* in the message table, then the node forwards the QUERYHIT or PONG to the node from which the QUERY or PING was received.
- If the corresponding QUERY or PING message doesn't appear in the table, then the node discards the QUERYHIT or PONG message.
- Otherwise, the node makes a new entry in the table, and forwards the message to all the nodes it is connected to, except the one from which it received the message.

Q 20.1 Assume one doesn't know the topology of the Gnutella network or the propagation delays of messages. According to the protocol, a node should forward all QUERYHIT messages for which it saw the corresponding QUERY message back to the node

from which it received the QUERY message. If a node wants to guarantee that rule, when can the node remove the QUERY entry from the message table?

A. Never, in principle, because a node doesn't know if another QUERYHIT for the same Query will arrive.
B. Whenever it feels like, since the table is not necessary for correctness. It is only a performance optimization.
C. As soon as it has forwarded the corresponding QUERYHIT message.
D. As soon as the entry becomes the least recently used entry.

Both the Internet and the Gnutella network form graphs. For the Internet, the nodes are routers and the edges are links between the routers. For the Gnutella network, the nodes are Gnutella nodes and the edges are TCP connections between the nodes. The shortest path in a graph between two nodes A and B is the path that connects A with B through the fewest number of nodes.

Q 20.2 Assuming a stable Internet and Gnutella network, is the shortest path between two nodes in the Gnutella overlay network always the shortest path between those two nodes in the Internet?

A. Yes, because the Gnutella network uses the Internet to set up TCP connections between its nodes.
B. No, because TCP is slower than UDP.
C. Yes, because the topology of the Gnutella network is identical to the topology of the Internet.
D. No, because for node A to reach node B in the Gnutella network, it might have to go through node C, even though there is a direct, Internet link between A and B.

Q 20.3 Which of the following relationships always hold? (TTL(i) and HOP(i) are the values of TTL and Hop fields respectively after the message has traversed i hops)?

A. TTL(0) = HOPS(i) - TTL(i)
B. TTL(i) = TTL(i - 1) - 1, for $i > 0$
C. TTL(0) = TTL(i) + HOPS(i)
D. TTL(0) = TTL(i) x HOPS(i)

Q 20.4 Ben observes that both PING and QUERY messages have the same forwarding rules, so he proposes to delete PING and PONG messages from the protocol and to use a QUERY message with a null query (which requires a node to respond with a QUERYHIT message) to replace PING messages. Is Ben's modified protocol a good replacement for the Gnutella protocol?

A. Yes, good question. Beats me why the Gnutella designers included both PING and QUERY messages.
B. No, a PING message will typically have a lower value in the *TTL* field than a QUERY message when it enters the network
C. No, because PONG and QUERYHIT messages have different forwarding rules.
D. No, because there is no way to find nodes using QUERY messages.

Q 20.5 Assume that only one node S stores the song "System Design Rap," and that the query enters the network at a node C. Further assume *TTL* is set to a value large enough to explore the whole network. Gnutella can still find the song "System Design Rap" despite the failures of some sets of nodes (either Gnutella nodes or Internet routers). On the other hand, there are sets of nodes whose failure would prevent Gnutella from finding the song. Which of the following are among the latter sets?

A. any set containing S
B. any set containing a single node on the shortest path from C to S
C. any set of nodes that collectively disconnects C from S in the Gnutella network
D. any set of nodes that collectively disconnects C from S in the Internet

The following questions are based on Chapter 11[on-line].

Q 20.6 To which of the following attacks is Gnutella vulnerable (i.e., an attacker can implement the described attack)?

A. A single malicious node can always prevent a client from finding a file by dropping QUERYHITs.
B. A malicious node can respond with a file that doesn't match the query.
C. A malicious node can always change the contact information in a QUERYHIT message that goes through the node, for example, misleading the client to connect to it.
D. A single malicious node can always split the network into two disconnected networks by never forwarding PING and QUERY messages.
E. A single malicious node can always cause a QUERY message to circle forever in the network by incrementing the *TTL* field (instead of decrementing it).

Q 20.7 Ben wants to protect the content of a song against eavesdroppers during downloads. Ben thinks a node should send ENCRYPT (*k*, *song*), using a shared-secret algorithm, as the download, but Alyssa thinks the node should send CSHA (*song*), where CSHA is a cryptographically secure hash algorithm. Who is right?

A. Ben is right because no one can compute *song* from the output of CSHA (*song*), unless they already have *song*.
B. Alyssa is right because even if one doesn't know the shared-secret key *k* anyone can compute the inverse of the output of ENCRYPT (*k*, *song*).
C. Alyssa is right because CSHA doesn't require a key and therefore Ben doesn't have to design a protocol for key distribution.
D. Both are wrong because a public-key algorithm is the right choice, since encrypting with a public key algorithm is computationally more expensive than either CSHA or a shared-secret algorithm.

Ben is worried that an attacker might modify the "System Design Rap" song. He proposes that every node that originates a message signs the payload of a message with its private key. To discover the public keys of nodes, he modifies the PONG message to contain the public key of the responding node along with its Internet address. When a node is asked to serve a file it signs the response (including the file) with its private key.

Q 20.8 Which attacks does this scheme prevent?

A. It prevents malicious nodes from claiming they have a copy of the "System Design Rap" song and then serving music written by Bach.
B. It prevents malicious nodes from modifying QUERY messages that they forward.
C. It prevents malicious nodes from discarding QUERY messages.
D. It prevents nodes from impersonating other nodes and thus prevents them from forging songs.
E. None. It doesn't help.

2002–2–5…12

21 The OttoNet*

(Chapter 7[on-line], with a bit of Chapter 11[on-line])

Inspired by the recent political success of his Austrian compatriot, "Arnie," in Caleeforneea, Otto Pilot decides to emigrate to Boston. After several months, he finds the local accent impenetrable, and the local politics extremely murky, but what really irks him are the traffic nightmares and long driving delays in the area.

After some research, he concludes that the traffic problems can be alleviated if cars were able to discover up-to-date information about traffic conditions at any specified location, and use this information as input to software that can dynamically suggest good paths to use to go from one place to another. He jettisons his fledgling political career to start a company whose modest goal is to solve Boston's traffic problems.

After talking to car manufacturers, Otto determines the following:

1. All cars have an on-board computer on which he can install his software. All cars have a variety of sensors that can be processed in the car to provide traffic status, including current traffic speed, traffic density, evidence of accidents, construction delays, etc.

2. It is easy to equip a car with a Global Positioning System (GPS) receiver (in fact, an increasing number of cars already have one built-in). With GPS, software in the car can determine the car's location in a well-known coordinate system. (Assume that the location information is sufficiently precise for our purposes.)

3. Each car's computer can be networked using an inexpensive 10 megabits per second radio. Each radio has a spherical range, R, of 250 meters; i.e., a radio transmission from a car has a non-zero probability of directly reaching any other car within 250 meters, and no chance of directly reaching any car outside that range.

Otto sets out to design the *OttoNet*, a network system to provide traffic status information to applications. OttoNet is an *ad hoc wireless network* formed by cars communicating with each other using cheap radios, cooperatively forwarding packets for one another.

Each car in OttoNet has a client application and a server application running on its computer. OttoNet provides two procedures that run on every car, which the client and server applications can use:

1. QUERY (*location*): When the client application running on a car calls QUERY (*location*), OttoNet delivers a packet containing a *query* message to at least one car within distance R (the radio range) of the specified location, according to a best-effort contract. A packet containing a *query* is 1,000 bits in size.

2. RESPOND (*status_info*, *query_packet*): When the server application running on a car receives a query message, it processes the query and calls RESPOND (*status_info*, *query_packet*). RESPOND causes a packet containing a *response* message to be delivered to the client that performed the query, again according to a best-effort contract. A response

* Credit for developing this problem set goes to Hari Balakrishnan.

message summarizes local traffic information (*status_info*) collected from the car's sensors and is 10,000 bits in size.

For packets containing either query or response messages, the cars will forward the packet cooperatively in best-effort fashion toward the desired destination location or car. Cars may move arbitrarily, alternating between motion and rest. The maximum speed of a car is 30 meters per second (108 kilometers per hour or 67.5 miles per hour).

Q 21.1 Which of the following properties is true of the OttoNet, as described thus far?

A. Because the OttoNet is "best-effort," it will attempt to deliver query and response messages between client and server cars, but messages may be lost and may arrive out of order.

B. Because the OttoNet is "best-effort," it will ensure that as long as there is some uncongested path between the client and server cars, query and response messages will be successfully delivered between them.

C. Because the OttoNet is "best-effort," it makes no guarantees on the delay encountered by a query or response message before it reaches the intended destination.

D. An OttoNet client may receive multiple responses to a query, even if no packet retransmissions occur in the system.

Otto develops the following packet format for OttoNet (all fields except *payload* are part of the packet header):

```
structure packet
    GPS dst_loc              // intended destination location
    integer_128 dst_id       // car's 128-bit unique ID picked at random
    GPS src_loc              // location of car where packet originated
    integer_128 src_id       // unique ID of car where packet originated
    integer hop_limit        // number of hops remaining (initialized to 100)
    integer type             // query or response
    integer size             // size of packet
    string payload           // query request string or response status info
packet instance pkt;         // pkt is an instance of the structure packet
```

Each car has a 128-bit unique ID, picked entirely at random. Each car's current location is given by its GPS coordinates. If the sender application does not know the intended receiver's unique ID, it sets the *dst_id* field to 0 (no valid car has an ID of 0).

The procedure FORWARD (*pkt*) runs in each car, and is called whenever a packet arrives from the network or when a packet needs to be sent by the application. FORWARD maintains a table of the cars within radio range and their locations, using broadcasts every second to determine the locations of neighboring cars, and implements the following steps:

F1. If the car's ID is *pkt.dst_id* then deliver to application (using *pkt.type* to identify whether the packet should be delivered to the client or server application), and stop forwarding the packet.

F2. If the car is within R of *pkt.dst_id* and *pkt.type* is QUERY, then deliver to server application, and forward to any one neighbor that is even closer to *dst_loc*.

F3. *Geographic forwarding step:* If neither F1 nor F2 is applicable, then among the cars that are closer to *pkt.dst_loc*, forward the packet to some car that is closer in distance to *pkt.dst_loc*. If no such car exists, drop the packet.

The OttoNet's QUERY (*location*) and RESPOND (*status_info*, *query_packet*) procedures have the following pseudocode:

```
1  procedure QUERY (location)
2      pkt.dst_lock ← location
3      pkt.dst_id ← X                  // see question 21.2.
4      pkt.src_loc ← my_gps
5      pkt.src_id ← my_id
6      pkt.payload ← "What's the traffic status near you?"
7      SEND (pkt)

8  procedure RESPOND (status_info, query_packet)
9      pkt.dst_loc ←query_packet.src_loc
10     pkt.dst_id ← Y                  // see question 21.2.
11     pkt.src_loc ← my_gps
12     pkt.src_id ← my_id
13     pkt.payload ← "My traffic status is: " + status_info // "+" concatenates strings
14     SEND (pkt)
```

Q 21.2 What are suitable values for **X** and **Y** in lines 3 and 10, such that the pseudocode conforms to the specification of QUERY and RESPOND?

Q 21.3 What kinds of names are the ID and the GPS location used in the OttoNet packets? Are they addresses? Are they pure names? Are they unique identifiers?

Q 21.4 Otto outsources the implementation of the OttoNet according to these ideas and finds that there are times when a QUERY gets no response, and times when a receiver receives packets that are corrupted. Which of the following mechanisms is an example of an application of an end-to-end technique to cope with these problems?

- **A.** Upon not receiving a response for a QUERY, when a timer expires retry the QUERY from the client.
- **B.** If FORWARD fails to deliver a packet because no neighboring car is closer to the destination, store the packet at that car and deliver it to a closer neighboring car a little while later.
- **C.** Implement a checksum in the client and server applications to verify if a message has been corrupted.
- **D.** Run distinct TCP connections between each pair of cars along the path between a client and server to ensure reliable end-to-end packet delivery.

Otto decides to retry queries that don't receive a response. The speed of the radio in each car is 10 megabits per second, and the response and request sizes are 10,000 bits and 1,000 bits respectively. The car's computer is involved in both processing the packet, which takes 0.1 microsecond per bit, and in transmitting it out on the radio (i.e., there's

no pipelining of packet processing and transmission). Each car's radio can transmit and receive packets at the same time.

The maximum queue size is 4 packets in each car, the maximum radio range for a single hop is 250 meters, and that the maximum possible number of hops in OttoNet is 100. Ignore media access protocol delays. The server application takes negligible time to process a request and generate a response to be sent.

Q 21.5 What is the smallest "safe" timer expiration setting that ensures that the retry of a query will happen only when the original query or response packet is guaranteed not to still be in transit in the network?

Otto now proceeds to investigate why FORWARD sometimes has to drop a packet between a client and server, even though it appears that there is a sequence of nodes forming a path between them. The problem is that geographic forwarding does not always work, in that a car may have to drop a packet (rule F3) even though there is some path to the destination present in the network.

Q 21.6 In the figure below, suppose the car at F is successfully able to forward a packet destined to location D using rule F3 via some neighbor, N. Assuming that neither F or N has moved, clearly mark the region in the figure where N must be located.

Q 21.7 Otto decides to modify the client software to make pipelined QUERY calls in quick succession, sending a query before it gets a response to an earlier one. The client now needs to match each response it receives with the corresponding query. Which of these statements is correct?

A. As long as no two pipelined queries are addressed to the same destination location (the *dst_loc* field in the OttoNet header), the client can correctly identify the specific query that caused any given response it receives.

B. Suppose the OttoNet packet header includes a nonce set by the client, and the server includes a copy of the nonce in its response, and the client maintains state to match

nonces to queries. This approach can always correctly match a response to a query, including when two pipelined queries are sent to the same destination location.

C. Both the client and the server need to set nonces that the other side acknowledges (i.e., both sides need to implement the mechanism in choice B above), to ensure that a response can always be correctly matched to the corresponding query.

D. None of the above.

Q 21.8 After running the OttoNet for a few days, Otto notices that network congestion occasionally causes a congestion collapse because too many packets are sent into the network, only to be dropped before reaching the eventual destination. These packets consume valuable resources. Which of the following techniques is likely to reduce the likelihood of a congestion collapse?

A. Increase the size of the queue in each car from 4 packets to 8 packets.

B. Use exponential backoff for the timer expiration when retrying queries.

C. If a query is not answered within the timer expiration interval, multiplicatively reduce the maximum rate at which the client application sends OttoNet queries.

D. Use a flow control window at each receiver to prevent buffer overruns.

The following question is based on Chapter 11[on-line].

Q 21.9 The OttoNet is not a secure system. Otto has an idea—he observes that the 128-bit unique ID of a car can be set to be the public key of the car! He proposes the following protocol. On a packet containing a query message, sign the packet with the client car's private key. On a packet containing a response, encrypt the packet with the client car's public key (that public key is in the packet that contained the query). To allow packets containing responses to be forwarded through the network, the server does not encrypt the destination location and ID fields of those packets. Assume that each car's private key is not compromised. Which of the following statements are true?

A. A car that just forwards a packet containing queries can read that packet's payload and verify it.

B. The only car in the network that can decrypt a response from a server is the car specified in the destination field.

C. The client cannot always verify the message integrity of a response, even though it is encrypted.

D. If every server at some queried location is honest and not compromised, the client can be sure that an encrypted response it receives for a query actually contains the correct traffic status information.

2004–2–5…13

22 The Wireless EnergyNet*

(Chapter 7[on-line] and a little bit of 8)

2005–2–7

Sara Brum, an undergraduate research assistant, is concerned about energy consumption in the Computer Science building and decides to design the EnergyNet, a wireless network of nodes with sensors to monitor the building. Each node has three sensors: a power consumption sensor to monitor the power drawn at the power outlet to which it is attached, a light sensor, and a temperature sensor. Sara plans to have these nodes communicate with each other via radio, forwarding data via each other, to report information to a central monitoring station. That station has a radio-equipped node attached to it, called the **sink**.

There are two kinds of communication in EnergyNet:

A. **Node-to-sink reports**: A node sends a **report** to the sink via zero or more other nodes.
B. **EnergyNet routing protocol**: The nodes run a distributed routing protocol to determine the next hop for each node to use to forward data to the sink. Each node's next hop en route to the sink is called its **parent**.

EnergyNet is a best-effort network. Sara remembers from reading Chapter that layering is a good design principle for network protocols, and decides to adopt a three-layer design similar to the Chapter *7[on-line]* reference model. Our job is to help Sara design the EnergyNet and its network protocols. We will first design the protocols needed for the node-to-sink reports without worrying about how the routing protocol determines the parent for each node.

To start, let's assume that each node has an unchanging parent, every node has a path to the sink, and nodes do not crash. Nodes may have hardware or software faults, and packets could get corrupted or lost, though.

Sara develops the following simple design for the three-layer EnergyNet stack:

Layer	Header fields	Trailer fields
E2E report protocol	*location* *time*	*e2e_cksum* (32-bit checksum)
Network	*dstaddr* (16-bit network address of destination)	
Link	*recvid* (32-bit unique ID of link-layer destination) *sendid* (32-bit unique ID of link-layer sournce)	*ll_cksum* (32-bit checksum)

* Credit for developing this problem set goes to Hari Balakrishnan.

In addition to these fields, each report packet has a **payload** that contains a report of data observed by a node's sensors. When sending a report packet, the end-to-end layer at the reporting node sets the destination network-layer address to be a well-known 16-bit value, SINK_ADDR. The end-to-end layer at the sink node processes each report. Any node in the network can send a report to the sink.

If a layer has a checksum, it covers that layer's header and the data presented to that layer by the higher layer. Each EnergyNet node has a first-in, first-out (FIFO) queue at the network layer for packets waiting to be transmitted.

Q 22.1 What does an EnergyNet report frame look like when sent over the radio from one node to another? Fill in the rectangle below to show the different header and trailer fields in the correct order, starting with the first field on the left. Be sure to show the payload as well. You do not need to show field sizes.

Start of frame	

Q 22.2 Sara's goal is to ensure that the end-to-end layer at the sink passes on (to the application) only messages whose end-to-end header and payload are correct. Assume that the implementation of the functions to set and verify the checksum are correct, and that there are no faults when the end-to-end layer runs.

A. Will using just *ll_cksum* and not *e2e_cksum* achieve Sara's goal?
B. Will using just *e2e_cksum* and not *ll_cksum* achieve Sara's goal?
C. Must each node on the path from the reporting node to the sink recalculate *e2e_cksum* in order to achieve Sara's goal?

To recover lost frames, Sara decides to implement a link-layer retransmission scheme. When a node receives a frame whose *ll_cksum* is correct, it sends an acknowledgment (ACK) frame to the *sendid* of the frame. If a sender does not receive an ACK before a timer expires, it retransmits the frame. A sender attempts at most three retransmissions for a frame.

Q 22.3 Which of these statements is true of Sara's link-layer retransmission scheme if no node changes its parent?

A. Duplicate error-free frames may be received by a receiver.
B. Duplicate error-free frames may be received by a receiver even if the sending node's timeout is longer than the maximum possible round trip time between sender and receiver.
C. If each new frame is sent on a link only after all link-layer retransmissions of previous frames, then the *sink* may receive packets from a given node in a different order from the way in which they were sent.
D. If Sara were to implement an end-to-end retransmission scheme in addition to this link-layer scheme, the resulting design would violate an end-to-end argument.

Q 22.4 EnergyNet's radios use phase encoding with the Manchester code. Sara finds that if the frequency of level transitions of voltage is set to 500 kilohertz, the link has an acceptably low bit error rate when there is no radio channel interference, noise, or any other concurrent radio transmissions. What is the data rate corresponding to this level transition frequency (specify the correct units)?

Q 22.5 Consider the transmission of an error-free frame (that is, one that never needed to be retransmitted) over one radio hop from node *A* to node *B*. Which of the delays in the right column of the table below contribute to the time duration specified in the left column? (There may be multiple contributors.)

1. Time lag between first bit leaving *A* and that bit reaching *B*	A. Processing delay
2. Time lag between first bit reaching *B* and last bit reaching *B*.	B. Propagation delay
3. Time lag between when the last bit of the packet was received at *A* and the first bit of the same packet begins to be sent by *A*'s link layer to *B*.	C. Queuing delay
	D. Transmission delay

Q 22.6 Sara finds that EnergyNet often suffers from congestion. Which of the following methods is likely to help reduce EnergyNet's congestion?

A. If no link-layer ACK is received, the sender should use exponential backoff before sending the next frame over the radio.
B. Provision the network-layer queue at each node to ensure that no packets ever get dropped for lack of queue space.
C. On each link-layer ACK, piggyback information about how much queue space is available at a parent, and slow down a node's rate of transmission when its parent's queue occupancy is above some threshold.

Now, let's assume that nodes may crash and each node's parent may change with time.

Let us now turn to designing the routing protocol that EnergyNet nodes use to form a routing tree rooted at the sink. Once each second, each node picks a parent by optimizing a ''quality'' metric and broadcasts a routing advertisement over its radio, as shown in the BROADCAST_ADVERTISEMENT procedure. Each node that receives an advertisement processes it and incorporates some information in its routing table, as shown in the HANDLE_ADVERTISEMENT procedure. These routing advertisements are not acknowledged by their recipients.

An advertisement contains one field in its payload: *quality*, calculated as shown in the pseudocode below. The **quality** of a path is a function of the success probability of frame

delivery across each link on the path. The success probability of a link is the probability that a frame is received at the receiver and its ACK received by the sender.

In the pseudocode below, *quality_table* is a table indexed by *sendid* and stores an object with two fields: *quality*, the current estimate of the path quality to the parent via the corresponding *sendid*, and *lasttime*, the last time at which an advertisement was heard from the corresponding *sendid*.

```
procedure BROADCAST_ADVERTISEMENT ()        // runs once per second at each node
    if quality_table = EMPTY and node != sink then return
    REMOVE_OLD_ENTRIES (quality_table)      // remove entries older than 5 seconds
    if node = sink then
        adv.quality ← 1.0
    else
        parent ← PICK_BEST(quality_table) // returns node with highest quality value
        adv.quality ← quality_table[parent].quality
    NETWORK_SEND (RTG_BCAST_ADDR, adv)      // broadcasts adv over radio

procedure HANDLE_ADVERTISEMENT (sendid, adv)
    quality_table[sendid].lasttime ← CURRENT_TIME ()
    quality_table[sendid].quality ← adv.quality × SUCCESS_PROB (sendid)
```

When BROADCAST_ADVERTISEMENT runs (once per second), it first removes all entries older than 5 seconds in *quality_table*. Then, it finds the best parent by picking the *sendid* with maximum *quality*, and broadcasts an advertisement message out to the network-layer address (RTG_BCAST_ADDR) that corresponds to all nodes within one network hop.

Whenever a node receives an advertisement from another node, *sendid*, it runs HANDLE_ADVERTISEMENT (). This procedure updates *quality_table*[*sendid*]. It calculates the path quality to reach the sink via *sendid* by multiplying the advertised quality with the success probability to this *sendid*, SUCCESS_PROB (*sendid*). The implementation details of SUCCESS_PROB () are not important here; just assume that all the link success probabilities are estimated correctly.

Assume that no "link" is perfect; i.e., for all i, j, $p_{ij} < 1$ (strictly less) and that every received advertisement is processed within 100 ms after it was broadcast.

Q 22.7 Ben Bitdiddle steps on and destroys the parent of node *N* at time $t = 10$ seconds. Assuming that node *N* has a current entry for its parent in its *quality_table*, to the nearest second, what are the earliest and latest times at which node *N* would remove the entry for its parent from its *quality_table*?

See Figure PS.2. The picture shows the success probability for each pair of transmissions (only non-zero probabilities are shown). The number next to each radio link is the link's success probability, the probability of a frame being received by a receiver and its ACK being received successfully by the sender.

Q 22.8 In Figure PS.2, suppose B is A's parent and B fails. Louis Reasoner asserts that as long as no routing advertisements are lost and there are no software or hardware bugs or failures, a routing loop can never form in the network. As usual, Louis is wrong. Explain why, giving a scenario or sequence of events that can create a routing loop.

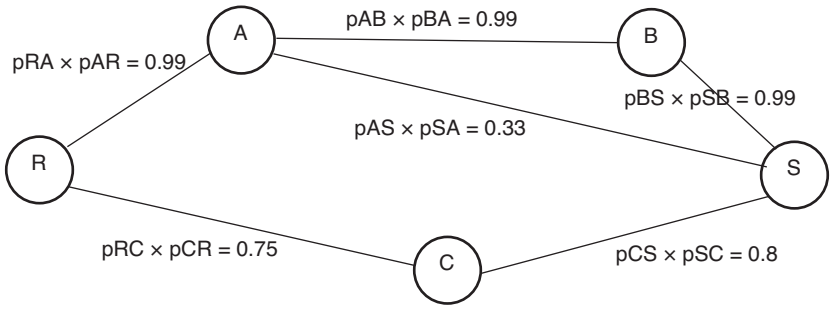


FIGURE ps.2

Network topology for some EnergyNet questions.

Q 22.9 Describe a modification to EnergyNet's routing advertisement that can prevent routing loops from forming in any EnergyNet deployment.

Q 22.10 Suppose node B has been restored to service and the success probabilities are as shown. Which path between R and S would be chosen by Sara's routing protocol and why? Name the path as a sequence of nodes starting with R and ending with S.

Q 22.11 Returning once again to Figure PS.2, recall that the nodes use link-layer retransmissions for report packets. If you want to minimize the *total expected number of non-ACK radio transmissions* needed to successfully deliver the packet from R to S, which path should you choose? You may assume that frames are lost independently over each link and that the link success probabilities are independent of each other. (Hint: If a coin has a probability p of landing "heads", then the expected number of tosses before you see "heads" is $1/p$.)

The remaining questions are on topics from Chapter 8.

Sara finds that each sensor's reported data is noisy, and that to obtain the correct data from a room, she needs to deploy $k > 1$ sensors in the room and take the average of the k reported values. However, she also finds that sensor nodes may fail in fail-fast fashion. Whenever there are fewer than k working sensors in a room, the room is considered to have "failed", and its data is "unavailable". When that occurs, an administrator has to go and replace the faulty sensors for the room to be "available" again, which takes time Tr. Tr is smaller than the MTTF of each sensor, but non-zero.

Assume that the sensor nodes fail independently and that Sara is able to detect the failure of a sensor node within a time much smaller than the node's MTTF.

Sara deploys $m > k$ sensors in each room. Sara comes up with three strategies to deploy and replace sensors in a room:

A. Fix each faulty sensor as soon as it fails.
B. Fix the faulty sensors as soon as all but one fail.
C. Fix each faulty sensor as soon as data from the room becomes unavailable.

Q 22.12 Rank these strategies in the order of highest to lowest availability for the room's sensor data.

Q 22.13 Suppose that each sensor node's failure process is memoryless and that sensors fail independently. Sara picks strategy C from the choices in the previous question. What is the resulting MTTF of the room?

23 SureThing*

(Chapter 7[on-line])

2006–2–7

Alyssa P. Hacker decides to offer her own content delivery system, named SURETHING. A SURETHING system contains 1000 computers that communicate via the Internet. Each computer has a unique numerical identifier ID#, and the computers are thought of as (logically) being organized in a ring as in Figure PS.3. Each computer has **successors** as shown in the figure. The ring wraps around: the immediate successor of the computer with the highest ID# (computer N251 in the figure) is the computer with the lowest ID# (computer N8).

Each content item also has a unique ID, *c*, and the content is stored at *c*'s **immediate successor**: the first computer in the ring whose ID# exceeds the ID# of *c*. This scheme is called **consistent hashing**.

Alyssa designs the system using two layers: a forwarding and routing layer (to find the IP address of the computer that stores the content) and a content layer (to store or retrieve the content).

Building a Forwarding and Routing Layer. Inspired by reading a paper on a system named Chord† that uses consistent hashing, Alyssa decides that the routing step will work as follows: Each computer has a local table, *successors[i]*, that contains the ID and IP address of its 4 successors (the 4 computers whose IDs follow this computer's ID in the ring); the entries are ordered as they appear in the ring. These tables are set up when the system is initialized.

The forwarding and routing layer of each node provides a procedure GET_LOCATION that can be called by the content layer to find the IP address of the immediate successor of some content item *c*. This procedure checks its local *successors* table to see if it contains the immediate successor of the requested content; if not, it makes a remote procedure call to the GET_LOCATION procedure on the *most distant* successor in its own *successors* table. That computer returns the immediate successor of *c* if it is known locally in its *successors* table; otherwise that node returns its *most distant* successor, and the originating computer continues the search there, iterating in this way until it locates *c*'s immediate successor.

For example, if computer N232 is looking for the immediate successor of *c* = C165 in the system shown in Figure PS.3, it will first look in its local table; since this table doesn't contain the immediate successor of *c*, it will request information from computer N36. Computer N36 also doesn't have the immediate successor of C165 in its local *suc-*

---

* Credit for developing this problem set goes to Barbara Liskov.

† Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proceedings of the ACM SIGCOMM '01 Conference*, 2001, August.

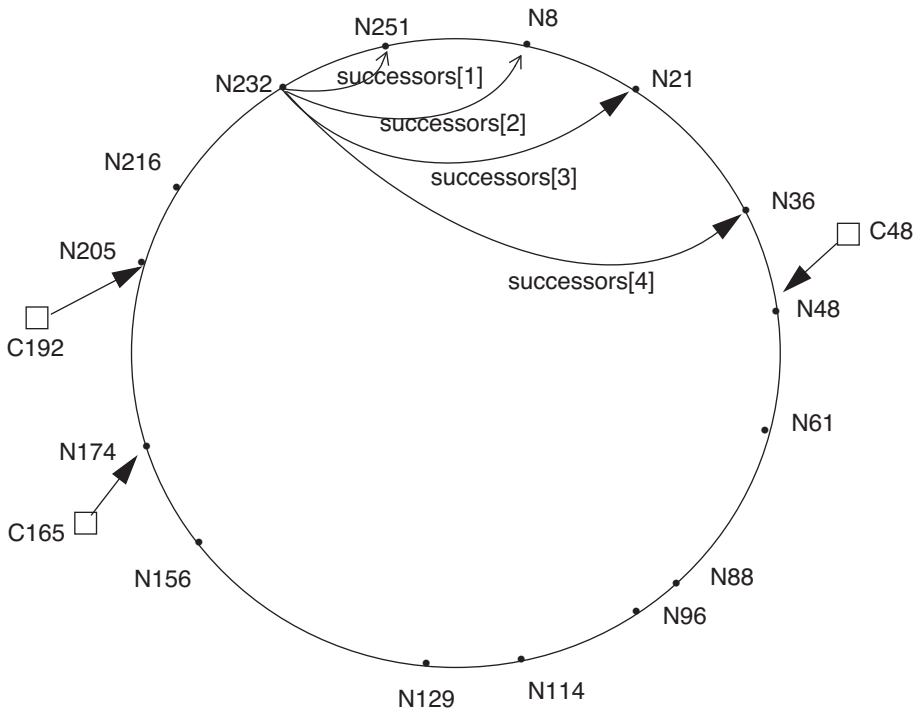


FIGURE PS.3

Arrangement of computers in a ring. Computer N232's pointers to its 4 successors lead to computers N251, N8, N21, and N36. The content item C192 is stored at computer N205 because #205 is the next larger computer ID# after C192's ID#. Similarly, content item C48 is stored at its immediate successor, computer N48; and item number C165 is stored at its immediate successor, computer N174.

cessors table, and therefore it returns the IP address of computer N96. Computer N96 does have the immediate successor (computer N174) in its local *successors* table and it returns this information. This sequence of RPC requests and responses is shown in Figure PS.4.

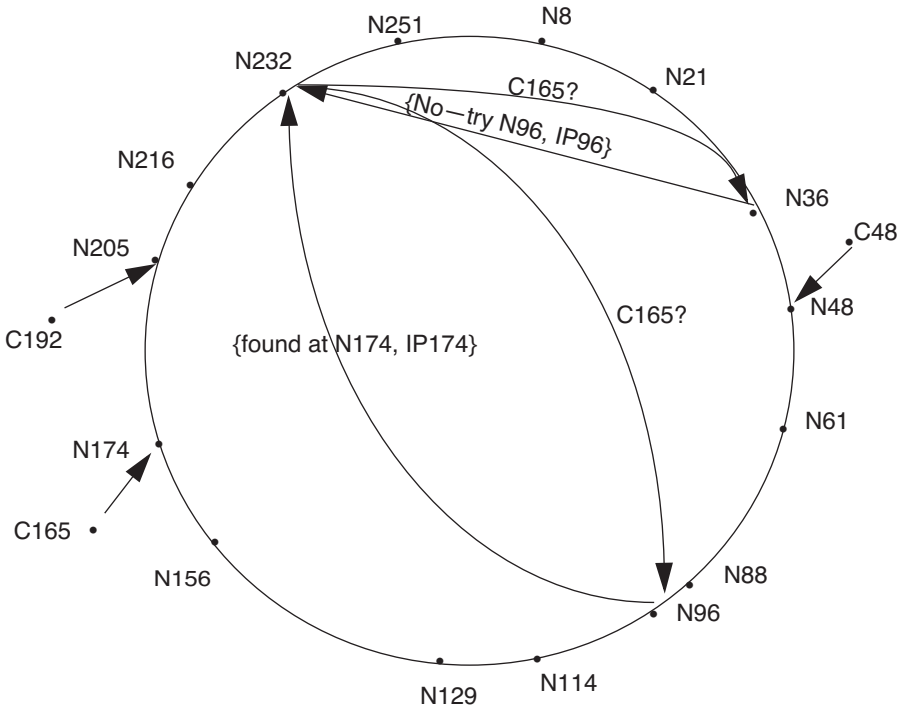


FIGURE PS.4

Sequence of RPCs and replies required for computer N232 to find the immediate successor of the content item with ID C165.

Q 23.1 While testing SURETHING, Alyssa notices that when the Internet attempts to deliver the RPC packets, they don't always arrive at their destination. Which of the following reasons might prevent a packet from arriving at its destination?

A. A router discards the packet.
B. The packet is corrupted in transit.
C. The payload of an RPC reply contains the wrong IP address.
D. The packet gets into a forwarding loop in the network.

For the next two questions, remember that computers don't fail and that all tables are initialized correctly.

Q 23.2 Assume that *c* is an id whose immediate successor is not present in *successors*, and *n* is the number of computers in the system. In the *best case*, how many remote lookups are needed before GET_LOCATION (*c*) returns?

A. 0
B. 1
C. 2
D. O(log *n*)
E. O(*n*)
F. O(n^2)

Q 23.3 Assume that *c* is an id whose immediate successor is not present in *successors*, and *n* is the number of computers in the system. In the *worst case,* how many remote lookups are needed before GET_LOCATION (*c*) returns?

A. 0
B. 1
C. 2
D. O(log *n*)
E. O(*n*)
F. O(n^2)

Building the Content Layer. Having built the forwarding and routing layer, Alyssa turns to building a content layer. At a high level, the system supports storing data that has an ID associated with it. Specifically, it supports two operations:

A. PUT (*c*, *content*) stores *content* in the system with ID *c*.
B. GET (*c*) returns the content that was stored with ID *c*.

Content IDs are integers that can be used as arguments to GET_LOCATION. (In practice, one can ensure that IDs are integers by using a hash function that maps human-readable names to integers.)

Alyssa implements the content layer by using the forwarding and routing layer to choose which computers to use to store the content. For reliability, she decides to store every piece of content on two computers: the two immediate successors of the content's ID. She modifies GET_LOCATION to return both successors, calling the new version GET_2LOCATIONS. For example, in Figure PS.3, if GET_2LOCATIONS is asked to find the content item with ID C165, it returns the IP addresses of computers N174 and N205.

Once the correct computers are located using the forwarding and routing layer, Alyssa's implementation sends a PUT RPC to each of these computers to store the content in a file in both places. (If one of the computers is the local computer, it does that store with a local call to PUT rather than an RPC.)

To retrieve the content associated with a given ID, if either ID returned by GET_2LOCATIONS is local it reads the file with a local GET. If not, it sends a GET RPC to the computer with the first ID, requesting that the computer load the appropriate file from

disk, if it exists, and return its contents. If that RPC fails for some reason, it tries the second ID.

Q 23.4 Which of the following are the end-to-end properties of the content layer? Assume that there are no failures of computers or disks while the system is running, that all tables are initialized correctly, and that the network delivers every message correctly.

A. GET (*c*) always returns the same content that was stored with ID *c*.
B. PUT (*c*, *content*) stores the content at the two immediate successors of *c*.
C. GET returns the content from the immediate successor of *c*.
D. If the content has been stored on some computer, GET will find it.

Q 23.5 Now, suppose that individual computers may crash but the network continues to deliver every message correctly. Which of the following properties of the content layer are true?

A. One of the computers returned by GET_2LOCATIONS might not answer the GET or PUT call.
B. PUT will sometimes be unable to store the content at the content's two immediate successors.
C. GET will successfully return the requested content, assuming it was stored previously.
D. If one of the two computers on which PUT stored the content has not crashed when GET runs, GET will succeed in retrieving the content.

Improving Forwarding Performance. We now return to the forwarding and routing layer and ignore the content layer.

Alyssa isn't happy with the performance of the system, in particular GET_LOCATION. Her friend Lem E. Tweakit suggests the following change: each computer maintains a *node_cache*, which contains information about the IDs and IP addresses of computers in the system. The *node_cache* table initially contains information about the computers in *successors*.

For example, initially the *node_cache* at computer N232 contains entries for computers N251, N8, N21, N36. But after computer N232 communicates with computer N36 and learns the ID and IP address of computer N96, N232's *node_cache* would contain entries for computers N251, N8, N21, N36, and N96.

Q 23.6 Assume that *c* is a content ID whose immediate successor is not one of the computers listed in *successors*, and *n* is the number of computers in the system. In the *best case*, how many remote lookups are needed before GET_LOCATION (*c*) returns?

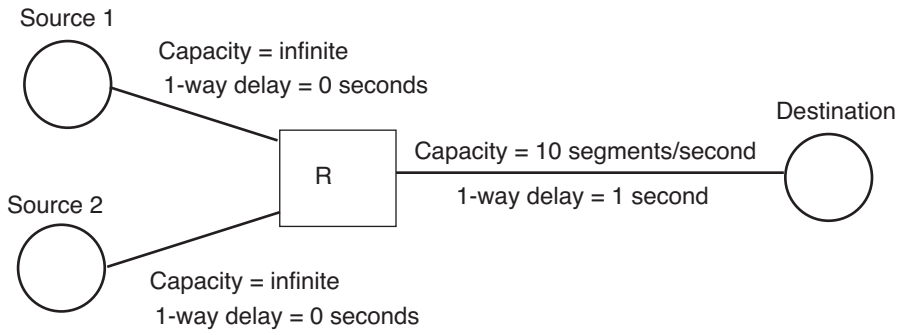
A. 0
B. 1
C. 2
D. $O(\log n)$
E. $O(n)$
F. $O(n^2)$

24 Sliding Window*

(Chapter 7[on-line])

2008–2–7

Consider the sliding window algorithm described in Chapter 7[on-line]. Assume the topology in the figure below, where all links are duplex and have the same capacity and delay in both directions. The capacities of the two links on the left are very large and can be assumed infinite, while their propagation delays are negligible and can be assumed zero. Both sources send to the same destination node.



Q 24.1 Assume the window size is fixed and only Source 1 is active. (Source 2 does not send any traffic.) What is the smallest sliding window that allows Source 1 to achieve the maximum throughput?

Source 1 does not know the bottleneck capacity and hence cannot compute the smallest window size that allows it to achieve the maximum throughput. Ben has an idea to allow Source 1 to compute the bottleneck capacity. Source 1 transmits two data segments back-to-back, i.e., as fast as possible. The destination sends an acknowledgment for each data segment immediately.

Q 24.2 Assume that acks are significantly smaller than data segments, all data segments are the same size, all acks are the same size, and only Source 1 has any traffic to transmit. In this case, which option is the best way for Source 1 to compute the bottleneck capacity?

A. Divide the size of a data segment by the interarrival time of two consecutive acks.
B. Divide the size of an ack by the interarrival time of two acks.
C. Sum the size of a data segment with an ack segment and divide the sum by the ack interarrival time.

---

* Credit for developing this problem set goes to Dina Katabi.

Now assume both Source 1 and Source 2 are active. Router R uses a large queue with space for about 10 times the size of the sliding window of question 24.1. If a data segment arrives at the router when the buffer is full, R discards that segment.

Source 2 uses standard TCP congestion control to control its window size. Source 1 also uses standard TCP, but hacks its congestion control algorithm to always use a fixed-size window, set to the size calculated in question 24.1.

Q 24.3 Which of the following is true?

A. Source 1 will have a higher average throughput than Source 2.
B. Source 2 will have a higher average throughput than Source 1.
C. Both sources get the same average throughput.

25 Geographic Routing*

(Chapter 7[on-line])

2008–2–3

Ben Bitdiddle is excited about a novel routing protocol that he came up with. Ben argues that since Global Positioning System (GPS) receivers are getting very cheap, one can equip every router with a GPS receiver so that the router can know its location and route packets based on location information.

Assume that all nodes in a network are in the same plane and nodes never move. Each node is identified by a tuple (x, y), where x and y are its GPS-derived coordinates, and no two nodes have the same coordinates. Each node is joined by links to its neighbors, forming a connected network graph. A node informs its neighbors of its coordinates when it joins the network and whenever it recovers after a failure.

When a source sends a packet, in place of a destination IP address, it puts the destination coordinates in the packet header. (A sender can learn the coordinates of its destination by asking Ben's modified DNS, which he calls the Domain Name Location Service.) When a router wants to forward a packet, it checks whether any of its neighbors are closer to the destination in Euclidean distance than itself. If none of its neighbors is closer, the router drops the packet. Otherwise the router forwards the packet to the neighbor closest to the destination. Forwarding of a packet stops when that packet either reaches a node that has the destination coordinates or is dropped.

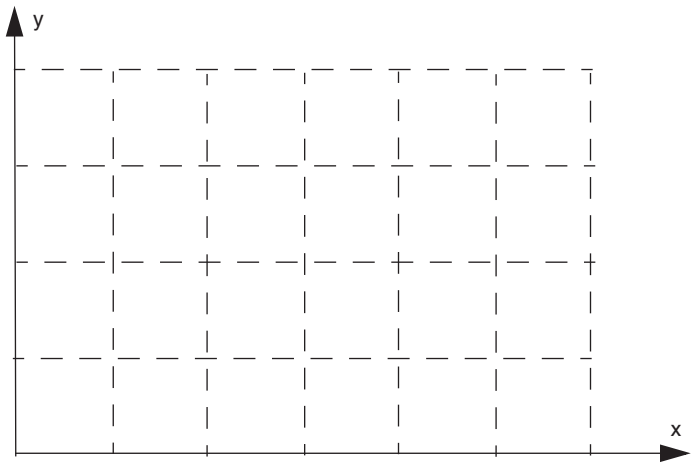
Q 25.1 Which of these statements are true about the Ben's geographic routing algorithm?

- **A.** If there are no failures, and no nodes join the network while packets are en route, no packet will experience a routing loop.
- **B.** If nodes fail while packets are en route, a packet may experience a routing loop.
- **C.** If nodes join the network while packets are en route, a packet may experience a routing loop.

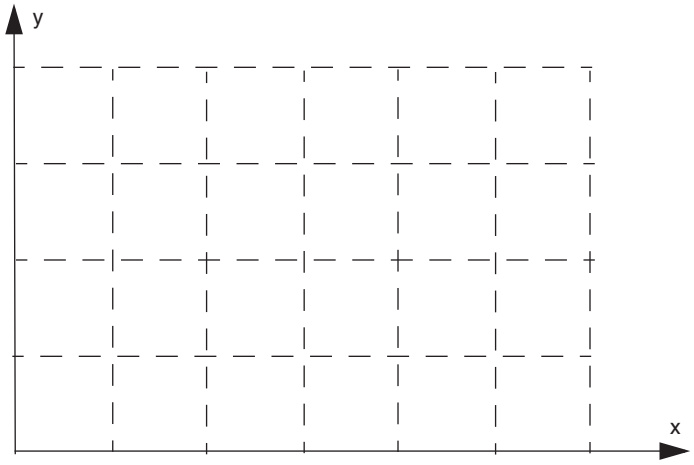
Suppose that that there are no failures of either links or nodes, and also that no node joins the network.

* Credit for developing this problem set goes to Dina Katabi.

Q 25.2 Can Ben's algorithm deliver packets between any source-destination pair in a network? If yes, explain. If no, draw a counter example in the grid below, placing nodes on grid intersections and making sure that links connect all nodes.



Q 25.3 For all packets that Ben's algorithm delivers to their corresponding destinations, does Ben's algorithm use the same route as the path vector algorithm described in Section *7.4.2*? If your answer is yes, then explain it. If your answer is no, then draw a counter example.



26 Carl's Satellite*

(Chapter 8[on-line])

Carl Coder decides to quit his job at an e-commerce start-up and go to graduate school. He's curious about the possibility of broadcasting data files through satellites, and decides to build a prototype that does so.

Carl decides to start simple. He launches a satellite into a geosynchronous orbit, so that the satellite is visible from all points in the United States. The satellite listens for incoming bits on a radio up-channel, and instantly retransmits each bit on a separate down-channel. Carl builds two ground stations, a sender and a receiver. The sending station sends on a radio to the satellite's up-channel; the receiving station listens to the satellite's down-channel.

Carl's test application is to send Associated Press (AP) news stories from a sending station, through the satellite, to a receiving station; the receiving station prints each story on a printer. AP stories always happen to consist of 1024 characters. Carl writes the code at the left to run on computers at the sending and receiving stations (Scheme 1).

```
procedure SENDER ()
    byte buffer[1024]
    do forever
        read next AP story into buffer // may wait for next story
        SEND_BUFFER (buffer)

procedure SEND_BUFFER (byte buffer[1024])
    for i from 0 to 1024 do
        SEND_8_BITS (buffer[i])

procedure RECEIVER ()
    byte buffer[1024]
    do forever
        ok ← RECV_BUFFER (buffer)
        if ok = TRUE then
            print buffer on a printer

procedure RECV_BUFFER (byte buffer[1024])
    for i from 0 to 1024 do
        buffer[i] ← RECV_8_BITS ()
    return (TRUE)
```

The receiving radio hardware receives a bit if and only if the sending radio sends a bit. This means the receiver receives the same number of bits that the sender sent. However, the receiving radio may receive a bit incorrectly, due to interference from sources near the receiver. The radio doesn't detect such errors; it just hands the incorrect bit to the computer at the receiving ground station with no warning. These incorrect bits are the only potential faults in the system, other than (perhaps) flaws in Carl's design. If the computer tells the printer to print an unprintable character, the printer prints a question mark instead.

* Credit for developing this problem set goes to Robert T. Morris.

After running the system for a while, Carl observes that it doesn't always work correctly. He compares the stories that are sent by the sender with the stories printed at the receiver.

Q 26.1 What kind of errors might Carl see at the receiver's printer?

A. Sometimes one or more characters in a printed story are incorrect.
B. Sometimes a story is repeated.
C. Sometimes stories are printed out of order.
D. Sometimes a story is entirely missing.

Q 26.2 The receiver radio manufacturer claims that the probability of receiving a bit incorrectly is one in 10^5, and that such errors are independent. If these claims are true, what fraction of stories is likely to be printed correctly?

Carl wants to make his system more reliable. He modifies his sender code to calculate the sum of the bytes in each story, and append the low 8 bits of that sum to the story. He modifies the receiver to check whether the low 8 bits of the sum of the received bytes match the received sum. His new code (Scheme 2) is at the right.

```
procedure SEND_BUFFER (byte buffer[1024])
    byte sum ← 0   // byte is an eight-bit unsigned integer
    for i from 0 to 1024 do
        SEND_8_BITS (buffer[i])
        sum ← sum + buffer[i]
    SEND_8_BITS (sum)

procedure RECV_BUFFER (byte buffer[1024])
    byte sum1, sum2
    sum1 ← 0
    for i from 0 to 1024 do
        buffer[i] ← RECV_8_BITS()
        sum1 ← sum1 + buffer[i]
    sum2 ← RECV_8_BITS()
    if sum1 = sum2 then return TRUE
    else return FALSE
```

Q 26.3 What kind of errors might Carl see at the receiver's printer with this new system?

A. Sometimes one or more characters in a printed story are incorrect.
B. Sometimes a story is repeated.
C. Sometimes stories are printed out of order.
D. Sometimes a story is entirely missing.

Q 26.4 Suppose the sender sends 10,000 stories. Which scheme is likely to print a larger number of these 10,000 stories correctly?

Carl decides his new system is good enough to test on a larger scale, and sets up 3 new receive stations scattered around the country, for a total of 4. All of the stations can hear the AP stories from his satellite. Users at each of the receivers call him up periodically with a list of articles that don't appear in their printer output so Carl can have the system re-send them. Users can recognize which stories don't appear because the Associated Press includes a number in each story, and assigns numbers sequentially to successive stories.

Q 26.5 Carl visits the sites after the system has been in operation for a week, and looks at the accumulated printouts (in the order they were printed) at each site. Carl notes that the first and last stories were received by all sites and all sites have received all retransmissions they have requested. What kind of errors might he see in these printouts?

A. Sometimes one or more characters in a printed story are incorrect.
B. Sometimes a story is repeated.
C. Sometimes stories are printed out of order.
D. Sometimes a story is entirely missing.

Q 26.6 Suppose Carl sends out four AP stories. Site 1 detects an error in just the first story; site 2 detects an error in just the second story; site 3 detects an error in just the third story; and site 4 receives all 4 stories correctly. How many stories will Carl have to re-send? Assume any resent stories are received and printed correctly.

After hearing about RAID, Carl realizes he could improve his system even more. He modifies his sender to send an extra "parity story" after every four AP stories; the parity story consists of the exclusive or of the previous four real stories. If one of the four stories is damaged, Carl's new receiver reconstructs it as the exclusive or of the parity and the other three stories.

His new pseudocode uses the checksumming versions of SEND_BUFFER () and RECV_BUFFER () to detect damaged stories.

```
procedure SENDER ()
    byte buffer[1024]
    byte parity[1024]
    do forever
        clear parity[] to all zeroes
        for i from 0 to 4 do
            read next AP story into buffer
            SEND_BUFFER (buffer)
            parity ← parity ⊕ buffer            // XOR's the whole buffer
        SEND_BUFFER (parity)

procedure RECEIVER ()
    byte buffers[5][1024]                       // holds the 4 stories and the parity
    boolean ok[5]                               // records which ones have been
                                                //         received correctly
    integer n                                   // count buffers received correctly
    do forever
        n ← 0
        for i from 0 to 5 do
            ok[i] ← RECV_BUFFER (buffers[i])
            if ok[i] then n ← n + 1
        for i from 0 to 4 do
            if ok[i] then print buffers[i]      // buffers[i] is correct
            else if n = 4 then                  // reconstruct buffers[i]
                clear buffers[i] to all zeroes
                for j from 0 to 5 do
                    if i ≠ j then
                        buffers[i] ← buffers[i] ⊕ buffers[j] // XOR two buffers
                print buffers[i]
            // don't print if you cannot reconstruct
```

Q 26.7 Suppose Carl sends out four AP stories with his new system, followed by a parity story. Site 1 is just missing the first story; site 2 is just missing the second story; site 3 is just missing the third story; and site 4 receives all stories correctly. How many stories will Carl have to re-send? Assume any re-sent stories are received and printed correctly.

Q 26.8 Carrie, Carl's younger sister, points out that Carl is using two forms of redundancy: the parity story and the checksum for each story. Carrie claims that Carl could do just as well with the parity alone, and that the checksum serves no useful function. Is Carrie right? Why or why not?

2001–3–6…13

27 RaidCo*

(Chapter 8[on-line])

2007–2–11

RaidCo is a company that makes pin-compatible hard disk replacements using tiny, chip-sized hard disks ("microdrives") that have become available cheaply. Each RaidCo product behaves like a hard disk, supporting the operations:

- *error* ← GET (*nblocks*, *starting_block_number*, *buffer_address*)
- *error* ← PUT (*nblocks*, *starting_block_number*, *buffer_address*)

to get or put an integral number of consecutive blocks from or to the disk array. Each operation returns a status value indicating whether an error has occurred.

RaidCo builds each of its disk products using twelve tiny, identical microdrives configured as a RAID system, as described in Section 2.1.1.4. A team of ace students designed RaidCo's system, and they did a flawless job of implementing six different RaidCo disk models. Each model uses identical hardware (including a processor and the twelve microdrives), but the models use different forms of RAID in their implementations and offer varying block sizes and performance characteristics to the customer. Note that the RAID systems' block sizes are not necessarily the same as the sector size of the component microdrives.

The models are as follows (they are described in the text at the places indicated in parentheses):

- R0: sector-level striping across all twelve microdrives, no redundancy/error correction (see Section 6.1.5)
- R1: six pairs of two mirrored microdrives, no striping (see Section 8.5.4.6)
- R2:12-microdrive RAID 2, using bit-level striping, error detection, and error correction); microdrive's internal sector-level error detection is disabled.
- R3: 12-microdrive RAID 3, using sector-level striping and error correction.
- R4: 12-microdrive RAID 4, no striping, dedicated parity disk (see Figure 8.6)
- R5: 12-microdrive RAID 5, no striping, distributed parity (see exercise 8.10)

The microdrives each conform to the same read/write API sketched above, each microdrive providing 100,000 sectors of 1,000 bytes each, and offering a uniform 10 millisecond seek time and a read/write bandwidth of 100 megabytes per second; thus the entire 100 megabytes of data on a microdrive can be fetched using a single GET operation in one second. The RaidCo products do no caching or buffering: each GET or PUT involves actual data transfer to or from the involved microdrives. Since the microdrives have uniform seek time, the RaidCo products do not need, and do not use, any seek optimizations.

---

* Credit for developing this problem set goes to Stephen A. Ward.

Q 27.1 As good as the students were at programming, they unfortunately left the documentation unfinished. Your job is to complete the following table, showing certain specifications for each model drive (i.e., the size and performance parameters of the API supported by each RAID system). Entries assume error-free operation, and ignore transfer times that are small compared to seek times encountered.

	R0	R1	R2	R3	R4	R5
Block size (kilobytes) exposed to GET/PUT	1 kB	1 kB		11 kB		1 kB
Capacity, in blocks						1,100,000
Max time for a single 100 megabyte GET (seconds)	1/12 s				1 s	1 s
Time for a 1-block PUT (milliseconds)	10 ms	10 ms	10 ms			20 ms
Typical number of microdrives involved in a 1-block GET	1					1
Typical number of microdrives involved in 2-block GET		2			1	1
Typical number of microdrives involved in 2-block PUT		2				

28 ColdFusion*

(Chapter 8[on-line], with a bit of Chapter 9[on-line])

Alyssa P. Hacker and Ben Bitdiddle are designing a hot new system, called *ColdFusion*, whose goal is to allow users to back up their storage systems with copies stored in a distributed network of ColdFusion servers. Users interact with ColdFusion using PUT and GET operations.

- PUT (*data*, *fid*) takes a data buffer and reliably stores it under a unique identifier *fid*, a positive integer, on some subset of the servers. It returns SUCCESS if it was successful in storing it on all the machines in the chosen subset, and FAILURE otherwise.
- GET (*fid*) returns the contents of the most recent successful PUT to the system for the file identified by *fid*.

Because high availability is a key competitive advantage, Alyssa decides to replicate user data on more than one server machine. But rather than replicate each file on *every* server, she decides to be clever and use only a subset of the servers for each file. Thus, the PUT of a file stores it on some number *(A)* of the servers, invoking a SERVER_PUT operation on each server. *server_put* is *atomic* and is implemented by each server.

If PUT is unable to successfully store the file on *A* servers, it returns FAILURE.

When a client does a GET of the file, the GET software attempts to retrieve the file from some subset of the servers and picks the version using an election algorithm. It chooses *B* servers to read data from, using an *atomic* SERVER_GET operation implemented by each server, following which it calls PICK_MAJORITY (). PICK_MAJORITY returns valid data corresponding to a version that is shared by *more than 50%* of the *B* copies retrieved, and NULL otherwise. Even though the client may not know which specific servers hold the current copy, the number of servers *(A)* in PUT and the number *(B)* in GET are chosen so that if a client GET succeeds, it is certain to have received the most recent copy.

They write the following code for PUT and GET. There are *S* servers in all, and $S > 2$. The particular ordering of the servers in the code below may be different at different clients, but all clients have the same list. They hire you as a consultant to help them figure out the missing parameters (*A* and *B*) and analyze the system for correctness.

* Credit for developing this problem set goes to Hari Balakrishnan.

```
// Internet addresses of the S servers, S > 2
ip_address server[S]                    // each client may have a different ordering

procedure PUT (byte data[], integer fid)
    integer ntried, nputs ← 0           // # of servers tried and # successfully put
    for ntried from 0 to S do
        // Put "data" into a file identified by fid at server[ntried]
        status ← SERVER_PUT (data, fid, server[ntried])
        if status = success then nputs ← nputs + 1
        if nputs ≥ A then               // yes! have SERVER_PUT () to A servers!
            return SUCCESS;
    return FAILURE                      // found < A servers to SERVER_PUT() to

procedure GET (integer fid, byte data[])
    integer ntried, ngets ← 0           // # times tried and
                                        // # times server_get  returned success
    byte files[S][MAX_FILE_LENGTH]      // array of files; an entry is a copy from a server
    integer index
    byte data[]
    for ntried from 0 to S do
        // Get file fid into buffer files[ntried] from server[ntried]
        status ← SERVER_GET (fid, files[ntried], server[ntried])
        if status = success then ngets ← ngets + 1
        if ngets ≥ B then               // yes! have gotten data from B servers
            // PICK_MAJORITY () takes the array of files and magically
            // knows which ones are valid. It scans the ngets valid
            // ones and returns an index in the files[] array for one
            // of the good copies, which corresponds to a version returned
            // by more than 50% of the servers. Otherwise, it returns –1.
            // If ngets = 1, PICK_MAJORITY () simply returns an index to
            // that version.
            index ← PICK_MAJORITY (files, ngets);
            if index ≠ –1 then
                COPY (data, files[index]) // copy into data buffer
                return SUCCESS
            else return FAILURE
    return failure                      // didn't find B servers to SERVER_GET from
```

For questions *Q 28.1* through *Q 28.4*, assume that operations execute serially (i.e., there is no concurrency). Assume also that the end-to-end protocol correctly handles all packet losses and delivers messages in to a recipient in the same order that the sender dispatched them. In other words, no operations are prevented from completing because of lost or reordered packets. However, servers may crash and subsequently recover.

Q 28.1 Which reliability technique is the best description of the one being attempted by Alyssa and Ben?

A. Fail-safe design.
B. N-modular redundancy.
C. Pair-and-compare.
D. Temporal redundancy.

Q 28.2 Which of the following combinations of A and B in the code above ensures that GET returns the results of the last successful PUT, as long as no servers fail? (Here, $\lfloor x \rfloor$ is the largest integer $\leq x$, and $\lceil y \rceil$ is the smallest integer $\geq y$. Thus $\lfloor 2.3 \rfloor = 2$ and $\lceil 2.3 \rceil = 3$. Remember also that $S > 2$.)

A.	$A = 1$	$B = S$
B.	$A = \lceil S/3 \rceil$	$B = S$
C.	$A = \lceil S/2 \rceil$	$B = S$
D.	$A = \lceil (3S)/4 \rceil$	$B = \lfloor S/2 \rfloor + 1$
E.	$A = S$	$B = 1$

Q 28.3 Suppose that the number of servers S is an odd number larger than 2, and that the number of servers used for PUT is $A = \lceil S/2 \rceil$. If only PUT and no **get** operations are done, how does the mean time to failure (MTTF) of the PUT operation change as S increases? The PUT operation fails if the return value from PUT is FAILURE. Assume that the process that causes servers to fail is *memoryless*, and that no repairs are done.

A. As S increases, there is more redundancy. So, the MTTF increases.
B. As S increases, one still needs about one-half of the servers to be accessible for a successful PUT. So, the MTTF does not change with S.
C. As S increases the MTTF decreases even though we have more servers in the system.
D. The MTTF is not a monotonic function of S; it first decreases and then increases.

Q 28.4 Which of the following is true of ColdFusion's PUT and GET operations, for choices of A and B that guarantee that GET successfully returns the data from the last successful PUT when no servers fail.

A. A PUT that fails because some server was unavailable to it, done after a successful PUT, may cause subsequent GET attempts to fail, even if B servers are available.
B. A failed PUT attempt done after a successful PUT *cannot* cause subsequent GET attempts to fail if B servers are available.
C. A failed PUT attempt done after a successful PUT *always* causes subsequent GET attempts to fail, even if B servers are available.
D. None of the above.

ColdFusion unveils their system for use on the Internet with $S = 15$ servers, using $A = 2S/3$ and $B = 1 + 2S/3$. However, they find that the specifications are not always met—several times, GET does not return the data from the last PUT that returned SUCCESS.

In questions *Q 28.5* through *Q 28.8*, assume that there may be concurrent operations.

Q 28.5 Under which of these scenarios does ColdFusion *always* meet its specification (i.e., GET returns SUCCESS *and* the data corresponding to the last successful PUT)?

- **A.** There is no scenario under which ColdFusion meets its specification for this choice of *A* and *B*.
- **B.** When a user PUT's data to a file with some *fid*, and at about the same time someone else PUT's different data to the same *fid*.
- **C.** When a user PUT's a file successfully from her computer at home, drives to work and attempts to GET the file an hour later. In the meantime, no one performs any PUT operations to the same file, but three of the servers crash and are unavailable when she does her GET.
- **D.** When the PUT of a file succeeds at some point in time, but some subsequent PUT's fail because some servers are unavailable, and then a GET is done to that file, which returns SUCCESS.

2000–3–12

You tell Ben to pay attention to multisite coordination, and he implements his version of the two-phase commit protocol. Here, each server maintains a log containing READY (a new record he has invented), ABORT, and COMMIT records. The server always returns the last COMMITed version of a file to a client.

In Ben's protocol, when the client PUTs a file, the server returns SUCCESS or FAILURE as before. If it returns SUCCESS, the server appends a READY entry for this *fid* in its log. If the client sees that all the servers it asked returns SUCCESS, it sends a message asking them all to COMMIT. When a server receives this message, it writes a COMMIT entry in its log together with the file's *fid*. On the other hand, if even one of the servers returns FAILURE, the client sends a message to all the servers asking them to abort the operation, and each server writes an ABORT entry in its log. Finally, if a server gets a server put request for some *fid* that is in the READY state, it returns FAILURE to the requesting client.

Q 28.6 Under which of these scenarios does ColdFusion *always* meet its specification (i.e., GET returns SUCCESS *and* the data corresponding to the last successful PUT)?

- **A.** There is no scenario under which ColdFusion meets its specification for this choice of *A* and *B*.
- **B.** When a user PUT's data to a file with some *fid*, and at about the same time someone else PUT's different data to the same *fid*.
- **C.** When a user PUT's a file successfully from her computer at home, drives to work and attempts to GET the file an hour later. In the meantime, no one performs any PUT operations to the same file, but three of the servers crash and are unavailable when she does her GET.
- **D.** When the PUT of a file succeeds at some point in time and the corresponding COMMIT messages have reached the servers, but some subsequent PUTs fail because some servers are unavailable, and then a GET is done to that file, which returns SUCCESS.

Q 28.7 When a server crashes and recovers, the original clients that initiated PUT's may be unreachable. This makes it hard for a server to know the status of its READY actions,

since it cannot ask the clients that originated them. Assuming that no more than one server is unavailable at any time in the system, which of the following strategies allows a server to correctly learn the status of a past READY action when it recovers from a crash?

A. Contact any server that is up and running and call SERVER_GET with the file's *fid*; if the server responds, change READY to COMMIT in the log.
B. Ask all the other servers that are up and running using server GET() with the file's *fid*; if more than 50% of the other servers respond with identical data, just change READY to COMMIT.
C. Pretend to be a client and invoke GET with the file's *fid*; if GET is successful and the data returned is the same as what is at this server, just change READY to COMMIT.
D. None of the above.

To accommodate the possibility of users operating on entire directories at once, ColdFusion adds a two-phase locking protocol on individual files within a directory. Alyssa and Ben find that although this sometimes works, deadlocks do occur when a GET owns some locks that a PUT needs, and vice versa.

Q 28.8 Ben analyzes the problem and comes up with several "solutions" (as usual). Which of his proposals will actually work, *always* preventing deadlocks from happening?

A. Ensure that the actions grab locks for individual files in increasing order of the fid of the file.
B. Ensure that no two actions grab locks for individual files in the same order.
C. Assign an incrementing timestamp to each action when it starts. If action A_i needs a lock owned by action A_j with a *larger* timestamp, abort action A_i and continue.
D. Assign an incrementing timestamp to each action when it starts. If action A_i needs a lock owned by action A_j with a *smaller* timestamp, abort action A_i and continue.

2000–3–8…15

29 AtomicPigeon!.com

(Chapter 9[on-line] but based on Chapter 7[on-line])

After selling PigeonExpress!.com and taking a trip around the world, Ben Bitdiddle is planning his next start-up, AtomicPigeon!.com. AtomicPigeon improves over PigeonExpress by offering an atomic data delivery system.

Recall from problem set *18* that when sending a pigeon, Ben's software prints out a little header and writes a CD, both of which are given to the pigeon. The header contains the GPS coordinates of the sender and receiver, a type (REQUEST or ACKNOWLEDGMENT), and a sequence number:

```
structure header
    GPS source
    GPS destination
    integer type
    integer sequence_no
```

Ben starts with the code for the simple end-to-end protocol (BEEP) for PigeonExpress!.com. He makes a number of modifications to the sending and receiving code.

At the sender, Ben simplifies the code. The BEEP protocol transfers only a single CD:

```
shared next_sequence initially 0    // a globally shared sequence number.

procedure BEEP (target, CD[])       // send 1 CD to target
    header h                        // h is an instance of header.
    h.source ← MY_GPS               // set source to my GPS coordinates
    h.destination ← target          // set destination
    h.type ← REQUEST                // this is a request message
    h.sequence_no ← next_sequence // set seq number
    // loop until we receive the corresponding ack, retransmitting if needed
    while h.sequence_no = next_sequence do
        send pigeon with h, CD      // transmit
        wait 2,000 seconds
```

As before, pending and incoming acknowledgments are processed *only* when the sender is waiting:

```
procedure PROCESS_ACK (h)               // process acknowledgment
    if h.sequence_no = sequence then    // ack for current outstanding CD?
        next_sequence ← next_sequence + 1
```

Ben makes a small change to the code running on the receiving computer. He adds a variable *expected_sequence* at the receiver, which is used by PROCESS_REQUEST to filter duplicates:

```
integer expected_sequence initially 0              // duplicate filter.

procedure PROCESS_REQUEST (h, CD)                  // process request
    if h.sequence_no = expected_sequence then      // the expected seq #?
        PROCESS (CD)                               // yes, process data
        expected_sequence ← expected_sequence + 1      // increase expectation
    h.destination ← h.source                   // send to where the pigeon came from
    h.source ← MY_GPS
    h.sequence_no ← h.sequence_no              // unchanged
    h.type ← ACKNOWLEDGMENT;
    send pigeon with h                         // send an acknowledgment back
```

The assumptions for the pigeon network are the same as in problem set *18*:

- Some pigeons might get lost, but, if they arrive, they deliver data correctly (uncorrupted)
- The network has one sender and one receiver
- The sender and the receiver are single-threaded

Q 29.1 Assume the sender and receiver do not fail (i.e., the only failures are that some pigeons may get lost). Does PROCESS in PROCESS_REQUEST process the value of CD exactly once?

A. Yes, since *next_sequence* is a nonce and the receiver processes data only when it sees a new nonce.

B. No, since *next_sequence* and *expected_sequence* may get out of sync because the receiver acknowledges requests even when it skips processing.

C. No, since if the acknowledgment isn't received within 2,000 seconds, the sender will send the same data again.

D. Yes, since pigeons with the same data are never retransmitted.

Ben's new goal is to provide atomicity, even in the presence of sender or receiver failures. The reason Ben is interested in providing atomicity is that he wants to use the pigeon network to provide P-commerce (something similar to E-commerce). He would like to write applications of the form:

```
procedure TRANSFER (amount, destination)
    WRITE (amount, CD)              // write amount on a CD
    BEEP (destination, CD)          // send amount
```

The amount always fits on a single CD.

If the sender or receiver fails, the failure is fail-fast. For now, let's assume that if the sender or receiver fails, it just stops and does ***not*** reboot; later, we will relax this constraint.

Q 29.2 Given the current implementation of the BEEP protocol and assuming that only the sender may fail, what could happen during, say, the 100th call to TRANSFER?

A. That TRANSFER might never succeed.
B. That TRANSFER might succeed.
C. PROCESS in PROCESS_REQUEST might process *amount* more than once.
D. PROCESS in PROCESS_REQUEST might process *amount* exactly once.

Ben's goal is to make transfer always succeed by allowing the sender to reboot and finish failed transfers. That is, after the sender fails, it clears volatile memory (including the nonce counter) and restarts the application. The application starts by running a recovery procedure, named RECOVER_SENDER, which retries a failed transfer, if any.

To allow for restartable transfers, Ben supplies the sender and the receiver with durable storage that never fails. On the durable storage, Ben stores a log, in which each entry has the following form:

```
structure log_entry
    integer type            // STARTED or COMMITTED
    integer sequence_no     // a sequence number
```

The main objective of the sender's log is to allow RECOVER_SENDER to restore the value of *next_sequence* and to allow the application to restart an unfinished transfer, if any.

Ben edits TRANSFER to use the log:

```
1  procedure TRANSFER (amount, destination)
2      WRITE (amount, CD)                      // write amount on a CD
3      ADD_LOG (STARTED, next_sequence)        // append STARTED record
4      BEEP (destination, CD)                  // send amount (BEEP increases next_sequence)
5      ADD_LOG (COMMITTED, next_sequence – 1) // append COMMITTED record
```

ADD_LOG atomically appends a record to the log on durable storage. If ADD_LOG returns, the entry has been appended. Logs contain sufficient space for new records and they don't have to be garbage collected.

Q 29.3 Identify the line in this new version of TRANSFER that is the commit point.

Q 29.4 How can the sender discover that a failure caused the transfer not to complete?

A. The log contains a STARTED record with no corresponding COMMITTED record.
B. The log contains a STARTED record with a corresponding ABORTED record.
C. The log contains a STARTED record with a corresponding COMMITTED record.
D. The log contains a COMMITTED record with no corresponding STARTED record.

Ben tries to write RECOVER_SENDER recover *next_sequence*, but his editor crashes before committing the final editing and the expression in the **if**-statement is missing, as indi-

cated by a "?" in the code below. Your job is to edit the code such that the correct expression is evaluated.

```
procedure RECOVER_SENDER ()
    next_sequence ← 0
    starting at end of log…
    for each entry in log do
        if ( ? ) then                  // What goes here? (See question 29.6)
            next_sequence ← (sequence_no of entry) + 1
            break                      // terminate scan of log
```

Q 29.5 After you edit RECOVER_SENDER for Ben, which of the following sequences could appear in the log? (The log records are represented as <*type sequence_no*>.

A. …, <STARTED 1>, <COMMITTED 1>, <STARTED 2>, <COMMITTED 2>
B. …, <STARTED 1>, <STARTED 1>, <COMMITTED 1>
C. …, <STARTED 1>, <STARTED 1>, <STARTED 2>, <COMMITTED 1>, <STARTED 2>
D. …, <STARTED 1>, <COMMITTED 1>, <COMMITTED 1>

Q 29.6 What expression should replace the **?** in the RECOVER_SENDER code above?

A. *entry.type* = COMMITTED
B. *entry.type* = STARTED
C. *entry.type* = ABORTED
D. FALSE

Q 29.7 Given the current implementation of the BEEP protocol what could happen, say, during the 100th call to TRANSFER? (Remember only the sending computer may fail.)

A. If the sending computer keeps failing during recovery, that TRANSFER might never succeed.
B. That TRANSFER might succeed.
C. PROCESS in PROCESS_REQUEST might process *amount* more than once.
D. PROCESS in PROCESS_REQUEST might process *amount* exactly once.

Ben's next goal is to make PROCESS_REQUEST all-or-nothing. In the following questions, assume that whenever the receiving computer fails, it reboots, calls RECOVER_RECEIVER, and after RECOVER_RECEIVER is finished, it waits for messages and calls PROCESS_REQUEST on each message.

To make *expected_sequence* all-or-nothing, Ben tries to change the receiver in a way similar to the change he made to the sender. Again, his editor didn't commit all the changes in time. The missing code is marked by "?" and "#". The missing expression

marked by "?" evaluates the same expression as did the "?" in RECOVER_SENDER. The new missing expressions are marked by "#" in PROCESS_REQUEST:

```
procedure RECOVER_RECEIVER ()
    expected_sequence ← 0
    starting at end of log…
    for each entry in log do
        if ( ? ) then                  // The expression of question 29.6
            expected_sequence ← sequence_no of entry + 1
            break                      // terminate scan of log
```

```
1   procedure PROCESS_REQUEST (h, CD)
2       if h.sequence_no = expected_sequence then // the expected seq #?
3           ADD_LOG (#, #)             // ? See question 29.8.
4           PROCESS (CD)               // yes, process data
5           expected_sequence ← expected_sequence + 1// increase expectation
6           ADD_LOG (#, #)             // ? See question 29.8.
7       h.destination ← source of h // send to where the pigeon came from
8       source of h.source ← MY_GPS
9       h.sequence_no ← h.sequence_no      // unchanged
10      h.type ← ACKNOWLEDGMENT
11      send pigeon with h             // send an acknowledgment back
```

As you can see from the code, Ben chose not to implement a write-ahead protocol because PROCESS is implemented by a third party, for example, a bank: PROCESS might be a call into the bank's transaction database system.

Q 29.8 Complete the ADD_LOG calls on the lines 3 and 6 in PROCESS_REQUEST such that *expected_sequence* will be all-or-nothing.

```
(   3    ADD_LOG          ,                )
    6    ADD_LOG (        ,                )
```

Q 29.9 Can PROCESS in PROCESS_REQUEST be called multiple times for a particular call to TRANSFER?

A. No, because *expected_sequence* is recovered and *h.sequence_no* is checked against it.
B. Yes, because failed transfers will be restarted and result in the acknowledgment being retransmitted.
C. Yes, because after 2,000 seconds a request will be retransmitted.
D. Yes, because the receiver may fail after PROCESS, but before it commits.

Q 29.10 How should PROCESS, called by PROCESS_REQUEST, be implemented to guarantee exactly-once semantics for transfers? (Remember that the sender is persistent.)

A. As a normal procedure call;
B. As a remote procedure call;
C. As a nested transaction;
D. As a top-level transaction.

1999–3–5…14

30 Sick Transit*

(Chapter 9[on-line])

Gloria Mundi, who stopped reading the text before getting to Chapter 9[on-line], is undertaking to resurrect the failed London Ambulance Service as a new streamlined company called Sick Transit. She has built a new computer she intends to use for processing ST's activities.

A key component in Gloria's machine is a highly reliable sequential-access infinite tape, which she plans to use as an **append-only** log. Records can be appended to the tape, but once written are immutable and durable. Records on the tape can be read any number of times, from front-to-back or from back-to-front. There is no disk in the ST system; the tape is the only non-volatile storage.

Because of the high cost of the infinite tape, Gloria compromised on the quality of more conventional components like RAM and CPU, which fail frequently but fortunately are fail-fast: every error causes an immediate system crash. Gloria plans to ensure that, after a crash, a consistent state can be reconstructed from the log on the infinite tape.

Gloria's code uses transactions, each identified by a unique transaction ID. The visible effect of a completed transaction is confined to changes in global variables whose WRITE operations are logged. The log will contain entries recording the following operations:

```
BEGIN (tid)          //  start a new transaction, whose unique ID is tid
COMMIT (tid)         // commit a transaction
ABORT (tid)          // abort a transaction
WRITE (tid, variable, old_value, new_value)
                     // write a global variable, specifying previous & new values.
```

To keep the system simple, Gloria plans to use the above forms as the application-code interface, in addition to a READ (*tid, variable*) call which returns the current value of *variable*. Each of the calls will perform the indicated operation and write a log entry as appropriate. Reading an unwritten variable is to return ZERO.

Gloria begins by considering the single-threaded case (only one transaction is active at any time). She stores values of global variables in a table in RAM. Gloria is now trying to figure out how to reset variables to committed values following a crash, using the log tape.

* Credit for developing this problem set goes to Stephen A. Ward.

Q 30.1 In the single-threaded case, what value should variable *v* be restored to following a crash?

A. *37*
B. *new_value* from the last logged WRITE (*tid*, *v*, *old_value*, *new_value*) or ZERO if unwritten.
C. *new_value* from the last logged WRITE (*tid*, *v*, *old_value*, *new_value*) that is not followed by an ABORT (*tid*), or ZERO if unwritten.
D. *new_value* from the last logged WRITE (*tid*, *v*, *old_value*, *new_value*) that is followed by a COMMIT (*tid*), or ZERO otherwise.
E. Either *old_value* or *new_value* from the last logged WRITE (*tid*, *v*, *old_value*, *new_value*), depending on whether that WRITE is followed by a COMMIT on the same *tid*, or ZERO if unwritten.

Gloria now tries running concurrent transactions on her system. Accesses to the log are serialized by the sequential-access tape drive.

Her first trial involves concurrent execution of these two transactions:

```
BEGIN (t1)                          BEGIN (t2)
    t1x ← READ (x)                      t2x ← READ (x)
    t1y ← READ (y)                      t2y ← READ (y)
    WRITE (t1, x, t1x, t1y + 1)         WRITE (t2, y, t2y, t2x + 2)
COMMIT (t1)                         COMMIT (t2)
```

The initial values of *x* and *y* are ZERO, as are all uninitialized variables in her system. Here the READ primitive simply returns the most recently written value of a variable from the RAM table, ignoring COMMITs.

Q 30.2 In the absence of locks or other synchronization mechanism, will the result necessarily correspond to some serial execution of the two transactions?

A. Yes.
B. No, since the execution might result in $x = 3$, $y = 3$.
C. No, since the execution might result in $x = 1$, $y = 3$.
D. No, since the execution might result in $x = 1$, $y = 2$.

Gloria is considering using locks, and automatically adding code to each transaction to guarantee before-or-after atomicity. She would like to maximize concurrency; she is, however anxious to avoid deadlocks. For each of the following proposals, decide whether the approach (1) yields semantics consistent with before-or-after atomicity and (2) introduces potential deadlocks.

Q 30.3 A single, global lock which is ACQUIREd at the start of each transaction and RELEASEd at COMMIT.

Q 30.4 A lock for each variable. Every READ or WRITE operation is immediately surrounded by an ACQUIRE and RELEASE of that variable's lock.

Q 30.5 A lock for each variable that a transaction READS or WRITEs, acquired immediately prior to the first reference to that variable in the transaction; all locks are released at COMMIT.

Q 30.6 A lock for each variable that a transaction READS or WRITEs, acquired in alphabetical order, immediately following the BEGIN. All locks are released at COMMIT.

Q 30.7 A lock for each variable a transaction WRITEs, acquired, in alphabetical order, immediately following the BEGIN. All locks are released at COMMIT.

In the general case (concurrent transactions) Gloria would like to avoid having to read the entire log during crash recovery. She proposes periodically adding a CHECKPOINT entry to the log, and reading the log backwards from the end restoring committed values to RAM. The backwards scan should end as soon as committed values have been restored to all variables. Each CHECKPOINT entry in the log contains current values of all variables and a list of uncommitted transactions at the time of the CHECKPOINT.

Q 30.8 What portion of the tape must be read to properly restore values committed at the time of the crash?

A. All of the tape; checkpoints don't help.
B. Enough to include the STARTED record from each transaction that was uncommitted at the time of the crash.
C. Enough to include the last CHECKPOINT, as well as the STARTED record from each transaction that was uncommitted at the time of the crash.
D. Enough to include the last CHECKPOINT, as well as the STARTED record from each transaction that was uncommitted at the time of the last checkpoint.

Simplicity Winns, Gloria's one-time classmate, observes that since global variable values can be reconstructed from the log their storage in RAM is redundant. She proposes eliminating the RAM as well as all of Gloria's proposed locks, and implementing a READ (*tid*, *var*) primitive which returns an appropriate value of *var* by examining the log.

Simplicity's plan is to implement READ so that each transaction *a* "sees" the values of global values at the time of BEGIN (*a*), as well as changes made within *a*. She quickly sketches an implementation of READ which she claims gives appropriate atomicity semantics:

```
procedure READ(tid, var)
    winners ← EMPTY                    // winners is a list.
    prior ← FALSE
    for each entry in log do
        if entry is STARTED (tid) then prior ← TRUE
        if entry is COMMITTED (Etid) and prior = TRUE then add Etid to winners
        if entry is WRITE (Etid, var, old_value, new_value) then
            if Etid = tid then return new_value
            if Etid is in winners then return new_value
    return 0
```

Gloria is a little dazed by Simplicity's quick synopsis, but thinks that Simplicity is likely correct. Gloria asks your help in figuring out what Simplicity's algorithm actually does.

Q 30.9 Suppose transaction *t* READs variable *x* but does not write it. Will each READ of *x* in *t* see the same value? If so, concisely describe the value returned by each READ; if not, explain.

Q 30.10 Does Simplicity's scheme REALLY offer transaction semantics yet avoid deadlocks?

A. Yup. Read it and weep, Gloria.
B. It doesn't introduce deadlocks, but doesn't guarantee before-or-after transaction semantics either.
C. It gives before-or-after transactions, but introduces possible deadlocks.
D. Simplicity's approach doesn't work even when there's no concurrency—it gives wrong answers.

Q 30.11 The real motivation of the *Sick Transit* problem is a stupid pun. What does *sic transit gloria mundi* actually mean?

A. Thus passes a glorious Monday.
B. Thus passes the glory of the world.
C. Gloria threw up on the T Monday.
D. This bus for First Class and Coach.
E. This is the last straw! If I wanted to take @#%!*% *Latin*, I'd have gone to Oxford.

1997–3–6…15

31 The Bank of Central Peoria, Limited

(Chapter 9[on-line])

Ben Bitdiddle decides to go into business. He bids $1 at a Resolution Trust Corporation auction and becomes the owner of the Bank of Central Peoria, Limited (BCPL).

When he arrives at BCPL, Ben is shocked to learn that the only programmer who understood BCPL's database has left the company to work on new animation techniques for South Park. Hiring programmers is difficult in Peoria, so Ben decides to take over the database code himself.

Ben learns that an account is represented as a structure with the following information:

```
structure account
    integer account_id      // account identification number
    integer balance         // account balance
```

The BCPL system implements a standard transaction interface for accessing accounts:

```
tid ← BEGIN ()        // Starts a new transaction that will be identified as number tid
balance ← READ (tid, account.account_id)      // Returns the balance of an account
WRITE (tid, account.account_id, newbalance)  // Updates the balance of an account
COMMIT (tid)      // Makes the updates of transaction tid visible to other transactions
```

The BCPL system uses two disks, both accessed synchronously (i.e., GET and PUT operations on the disks won't return until the data is read from the disk or has safely been written to the disk, respectively). One disk contains nothing but the account balances, indexed by number. This disk is called the *database disk*. The other disk is called the *log disk* and exclusively stores, in chronological order, a sequence of records of the form:

```
structure logrecord
    integer op              // WRITE, COMMIT, or END
    integer tid             // transaction number
    integer account_id      // account number
    integer new_balance     // new balance for account "account"
```

where the meaning of each record is given by the *op* field:

```
op = WRITE       // Update of an account to a new balance by transaction tid
op = COMMITTED  // Transaction tid's updates are now visible to other transactions
                        and durable across crashes
op = END      // Transaction tid's writes have all been installed on the database disk
```

For each active transaction, the BCPL system keeps a list in volatile memory called *intentions* containing pairs (*account_id*, *new_balance*). The implementation of READ is as follows:

```
procedure READ (tid, account_id)
    if account_id is in intentions of tid then
        pair ← last pair containing account_id from intentions of tid
        return pair.new_balance
    else
        GET account containing account_id from database
        return account.balance
```

A. Recovery

For this section, assume that there are no concurrent transactions.

Ben asks whether the database computer has ever crashed and learns that it crashed frequently due to intense sound vibrations from the jail next door. Ben decides he had better understand how recovery works in the BCPL system. He examines the implementation of the recovery procedure. He finds the following code:

```
1  procedure RECOVERY ()
2      winners ← NULL
3      reading the log from oldest to newest,
4      for each record in log do
5          if record.op = COMMITTED then add record.tid to winners
6          if record.op = END remove then record.tid from winners
7      again reading the log from oldest to newest,
8      for each record in log do
9          if record.op = WRITE and record.tid is in winners then
10             INSTALL (record.new_balance in database for record.account_id
11     for each tid in winners do
12         LOG {END, tid}
```

Q 31.1 What would happen if lines 11 and 12 were omitted?

A. The system might fail to recover correctly from the first crash that occurs.

B. The system would recover correctly from the first crash but the log would be corrupt so the system might fail to recover correctly from the second crash.

C. The system would recover correctly from multiple crashes but would have to do more work when recovering from the second and subsequent crashes.

Q 31.2 For the RECOVERY and READ procedures to be correct, which of the following could be correct implementations of the COMMIT procedure?

A.
```
procedure COMMIT (tid) {}
```
B.
```
procedure COMMIT (tid)
    for each pair in tid.intentions do
        INSTALL (pair.new_balance in database for pair.account_id)
    tid.intentions ← NULL
    LOG {COMMITTED, tid}
```
C.
```
procedure COMMIT (tid)
    LOG {END, tid}
    for each pair in tid.intentions do
        INSTALL (pair.new_balance in database for pair.account_id)
    tid.intentions ← NULL
    LOG {COMMITTED, tid}
```
D.
```
procedure COMMIT (tid) {
    LOG {COMMITTED, tid}
    for each pair in tid.intentions do
        INSTALL (pair.new_balance in database for pair.account_id)
    tid.intentions ← NULL
    LOG {END, tid}
```

Q 31.3 For the RECOVERY and READ procedures to be correct, which of the following could be correct implementations of the WRITE procedure?

A.
```
procedure WRITE (tid, account_id, new_balance)
    LOG {WRITE, tid, account_id, new_balance}
```
B.
```
procedure WRITE (tid, account_id, new_balance)
    add the pair {account_id, new_balance} to tid.intentions
    LOG {WRITE, tid, account_id, new_balance}
```
C.
```
procedure WRITE (tid, account_id, new_balance)
    LOG {WRITE, tid, account_id, new_balance}
    add the pair {account_id, new_balance} to tid.intentions
```
D.
```
procedure WRITE (tid, account_id, new_balance)
    LOG {WRITE, tid, account_id, new_balance}
    add the pair {account_id, new_balance} to tid.intentions
    INSTALL new_balance in database for account_id
```

Ben is rather surprised to see there is no ABORT (*tid*) procedure that terminates a transaction and erases its database updates. He calls up the database developer who says it should be easy to add. Ben figures he might as well add the feature now, and adds a new log record type ABORTED.

Q 31.4 Which of the following could be correct implementations of the ABORT procedure? Assume that the RECOVERY procedure is changed correspondingly.

A.
```
procedure ABORT (tid)
    tid.intentions ← NULL
```
B.
```
procedure ABORT (tid)
    LOG {ABORTED, tid}
```
C.
```
procedure ABORT (tid)
    LOG {ABORTED, tid}
    tid.intentions ← NULL
```
D.
```
procedure ABORT (tid)
    tid.intentions ← NULL
    LOG {ABORTED, tid}
```

B. Buffer cache

BCPL is in intense competition with the nearby branch of Peoria Authorized Savings, Credit and Loan. BCPL's competitive edge is lower account fees. Ben decides to save the cost of upgrading the computer system hardware by adding a volatile memory buffer cache, which will make the database much more efficient on the current hardware. The buffer cache is used for GETs and PUTs to the database disk only; GETs and PUTs to the log disk remain write-through and synchronous.

The buffer cache uses an LRU replacement policy. Each account record on the database disk is cached or replaced separately. In other words, the cache block size, disk block size, and account record sizes are all identical.

In section B, again assume that there are no concurrent transactions.

Q 31.5 Why will adding a buffer cache for the database disk make the system more efficient?

A. It is faster to copy from the buffer cache than to GET from the disk.
B. If common access patterns can be identified, performance can be improved by prefetching multiple account balances into the cache.
C. It reduces the total number of disk GETs when one transaction reads the same account balance multiple times without updating it.
D. It reduces the total number of disk GETs when multiple consecutive transactions read the same account balance.

Ben then makes a mistake. He reasons that the intentions list described in section A is now unnecessary, since the list just keeps in-memory copies of database data, which is the same thing done by the buffer cache. He deletes the intentions list code and modifies PUT so it updates the copy of the account balance in the buffer cache. He also modifies the system to delay writing the END record until all buffered accounts modified by that transaction have been written back to the database disk. Much to his horror, the next

time the inmates next door try an escape and the resulting commotion causes the BCPL system to crash, the database does not recover to a consistent state.

Q 31.6 What might have caused recovery to fail?

A. The system crashed when only some of the modifications made by a committed transaction had reached the database disk.
B. The LRU replacement policy updated the database disk with data modified by an uncommitted transaction, which later committed before the crash.
C. The LRU replacement policy updated the database disk with data modified by an uncommitted transaction, which failed to commit before the crash.
D. The LRU replacement policy updated the database disk with data modified by a committed transaction, which later completed before the crash.
E. The LRU replacement policy updated the database disk with data modified by a committed transaction, which did not complete before the crash.

C. Concurrency

Ben restores the intention-list code, deletes the buffer cache code and goes back to the simpler system described in section A.

He is finally ready to investigate how the BCPL system manages concurrent transactions. He calls up the developer and she tells him that there is a lock stored in main memory for each account in the database, used by the CONCURRENT_BEGIN and CONCURRENT_COMMIT procedures. Since BCPL runs concurrent transactions, all its applications actually use these two procedures rather than the lower-level BEGIN and COMMIT procedures described earlier.

An application doing a concurrent transaction must declare the list of accounts it will use as an argument to the CONCURRENT_BEGIN procedure.

```
procedure CONCURRENT_BEGIN (account_list)
    do atomically
        for each account in account_list do
            ACQUIRE (account.lock)
    tid ← BEGIN ()
    tid.account_list ← account_list
    return tid

procedure CONCURRENT_COMMIT (tid)
    COMMIT (tid)
    for each account in tid.account_list do
        RELEASE (account.lock)
```

Ben runs two transactions concurrently. Both transactions update account number 2:

```
tida ← CONCURRENT_BEGIN (MAKELIST (2))
tmpa ← READ (tida, 2)
WRITE (tida, 2, tmpa + 1)
CONCURRENT_COMMIT (tida)
```

```
tidb ← CONCURRENT_BEGIN (MAKELIST (2))
tmpb ← READ (tidb, 2)
WRITE (tidb, 2, tmpb + 2)
CONCURRENT_COMMIT (tidb)
```

MAKELIST creates a list from its arguments; in this case the list has just one element. The

initial balance of account 2 before these transactions start is 0.

Q 31.7 What possible values can account 2 have after completing these two transactions (assuming no crashes)?

A. 0
B. 1
C. 2
D. 3
E. 4

Ben is surprised by the order of the operations in CONCURRENT_COMMIT, since COMMIT is expensive (requiring synchronous writes to the log disk). It would be faster to release the locks first.

Q 31.8 If the initial balance of account 2 is zero, what possible values can account 2 have after completing these two transactions (assuming no crashes) if the locks are released before the call to COMMIT?

A. 0
B. 1
C. 2
D. 3
E. 4

1998–3–7…14

32 Whisks*

(Chapter 9[on-line])

The Odd Disk Company (ODC) has just invented a new kind of non-volatile storage, the Whisk. A Whisk is unlike a disk in the following ways:

- Compared with disks, Whisks have very low read and write latencies.
- On the other hand, the data rate when reading and writing a Whisk is much less than that of a disk.
- Whisks are *associative*. Where disks use sector addresses, a Whisk block is named with a pair of items: an address and a *tag*. We write these pairs as A/*t*, where A is the address and *t* is the tag. Thus, for example, there might be three blocks on the Whisk with address 49, each with different tags: 49/1, 49/2, and 49/97.

The Whisk provides four important operations:

- *data* ← GET (*A/t*): This is the normal read operation.
- PUT (*A/t*, *data*): Just like a normal disk. If the system crashes during a WRITE, a partially written block may result.
- *boolean* ← EXISTS (*A/t*): Returns TRUE if block *A/t* exists on the Whisk.
- CHANGE_TAG (*A/m*, *n*): Atomically changes the tag *m* of block *A/m* to *n* (deleting any previous block *A/n* in the process). The atomicity includes both all-or-nothing atomicity and before-or-after atomicity.

Ben Bitdiddle is excited about the properties of Whisks. Help him develop different storage systems using Whisks as the medium.

Q 32.1 At first, Ben emulates a normal disk by writing all blocks with tag 0. But now he wants to add an ATOMIC_PUT operation. Design an ATOMIC_PUT for Ben's Whisk, and identify the step that is the commit point.

Ben has started work on a Whisk transaction system; he'd like you to help him finish it. Looking through his notes, you see that Ben's system will use no caches or logs: all writes go straight to the Whisk. One sentence particularly catches your eye: a joyfully scrawled "Transaction IDs Are Tags!!" Ben's basic idea is this. The current state of the database will be stored in blocks with tag 0. When a transaction t writes a block, the data is stored in the separate block A/t until the transaction commits.

Ben has set aside a special disk address, ComRec, to hold commit records for all running transactions. For a transaction t, the contents of ComRec/t is either committed, aborted, or pending, depending on the state of transaction t.

* Credit for developing this problem set goes to Eddie Kohler.

So far, three procedures have been implemented. In these programs, *t* is a transaction ID, *A* is a Whisk block address, and *data* is a data block.

```
procedure AA_BEGIN (t)
    PUT (ComRec/t, PENDING)

procedure AA_READ (t, A)
    if EXISTS (A/t) then
        return GET (A/t)
    else // uninitialized!
        return GET (A/0)

procedure AA_WRITE (t, A, data)
    PUT (A/t, data)
```

The following questions are concerned only with all-or-nothing atomicity; there are no concurrent transactions.

Q 32.2 Write pseudocode for AA_COMMIT (*t*) and AA_ABORT (*t*), and identify the commit point in AA_COMMIT. Assume that the variable *dirty*, an array with *num_dirty* elements, holds all the addresses to which *t* has written. (Don't worry about any garbage an aborted transaction might leave on disk, and assume transaction IDs are never reused.)

Q 32.3 Write the pseudocode for AA_RECOVER, the program that handles recovery after a crash. Ben has already done some of the work: his code examines the *ComRec* blocks and determines which transactions are COMMITTED, ABORTED, or PENDING. When your pseudocode is called, he has already set 6 variables for you (you might not need them all):

```
num_committed       // the number of committed transactions
committed[i]        // an array holding the committed transactions' IDs
num_aborted         // the number of aborted transactions
aborted[i]          // an array holding the aborted transactions' IDs
num_pending         // the number of transactions in progress
in_progress[i]      // an array holding the in-progress transactions' IDs
```

Whisk addresses run from 0 to *N*.

1996–3–4a…d

33 ANTS: Advanced "Nonce-ensical" Transaction System*

(Chapter 9[on-line])

Sara Bellum, forever searching for elegance, sets out to design a new transaction system called ANTS, based on the idea of nonces. She observes that the locking schemes she learned in Chapter 9[on-line] cause transactions to wait for locks held by other transactions. She observes that it is possible for a transaction to simply abort and retry, instead of waiting for a lock. A little bit more work convinces her that this idea may allow her to design a system in which transactions don't need to use locks for before-or-after atomicity.

Sara sets out to write pseudocode for the following operations: BEGIN (), READ (), WRITE (), COMMIT (), ABORT (), and RECOVER (). She intends that, together, these operations will provide transaction semantics: before-or-after atomicity, all-or-nothing atomicity, and durability. You may assume that once any of these operations starts, it runs to completion without preemption or failure, and that no other thread is running any of the procedures at the same time. The system may interleave the execution of multiple transactions, however.

Sara's implementation assigns a transaction identifier (TID) to a transaction when it calls BEGIN (). The TIDs are integers, and ANTS assigns them in numerically increasing order. Sara's plan for the transaction system's storage is to maintain cell storage for variables, and a write-ahead log for recovery. Sara implements both the cell storage and the log using non-volatile storage. The log contains the following types of records:

- STARTED *TID*
- COMMITTED *TID*
- ABORTED *TID*
- UPDATED *TID*, *Variable Name*, *Old Value*

Sara implements BEGIN (), COMMIT (), ABORT (), and RECOVER () as follows:

- BEGIN () allocates the next TID, appends a STARTED record to the log, and returns the TID.
- COMMIT () appends a COMMITTED record to the log and returns.
- ABORT (*TID*) undoes all of transaction *TID*'s WRITE () operations by scanning the log backwards and writing the old values from the transaction's UPDATED records back to the cell storage. After completing the undo, ABORT (*TID*) appends an ABORTED entry to the log, and returns.
- RECOVER () is called after a crash and restart, before starting any more transactions. It scans the log backwards, undoing each WRITE record of each transaction that had neither committed nor aborted at the time of the crash. RECOVER () appends one ABORTED record to the log for each such transaction.

* Credit for developing this problem set goes to Hari Balakrishnan.

Sara's *before-or-after intention* is that the result of executing multiple transactions concurrently is the same as executing those same transactions one at a time, in increasing *TID* order. Sara wants her READ () and WRITE () implementations to provide before-or-after atomicity by adhering to the following rule:

Suppose a transaction with TID *t* executes READ (*X*). Let *u* be the highest TID < *t* that calls WRITE (*X*) and commits. The READ (*X*) executed by *t* should return the value that *u* writes.

Sara observes that this rule does not require her system to execute transactions in strict TID order. For example, the fact that two transactions call READ () on the same variable does not (by itself) constrain the order in which the transactions must execute.

To see how Sara intends ANTS to work, consider the following two transactions:

	Transaction T_A	Transaction T_B
1	tid_a ← BEGIN () // returns 15	
2		tid_b ← BEGIN () //returns 16
3	*va* ← READ (tid_a, *X*)	
4	*va* ← *va* + 1	
5		*vb* ← READ (tid_a, *X*)
6		*vb* ← *vb* + 1
α	WRITE (tid_a, *X*, *va*)	WRITE (tid_b, *X*, *vb*)
β	COMMIT (tid_a)	COMMIT (tid_b)

Each transaction marks its start with a call to BEGIN, then reads the variable *X* from the cell store and stores it in a local variable, then adds one to that local variable, then writes the local variable to *X* in the cell store, and then commits. Each transaction passes its TID (tid_a and tid_b respectively) to the READ, WRITE, and COMMIT procedures.

These transactions both read and write the same piece of data, *X*. Suppose that T_A starts just before T_B, and Sara's BEGIN allocates TIDs 15 and 16 to T_A and T_B, respectively. Suppose that ANTS interleaves the execution of the transactions as shown through line 6, but that ANTS has not yet executed lines α and β. No other transactions are executing, and no failures occur.

Q 33.1 In this situation, which of the following actions can ANTS take in order to ensure before-or-after atomicity?

A. Force just T_A to abort, and let T_B proceed.
B. Force just T_B to abort, and let T_A proceed.
C. Force neither T_A nor T_B to abort, and let both proceed.
D. Force both T_A and T_B to abort.

To help enforce the before-or-after intention, Sara's implementation of ANTS maintains the following two pieces of information for each variable:

- *ReadID* — the TID of the highest-numbered transaction that has successfully read this variable using READ.

- *WriteID* — the TID of the highest-numbered transaction that has successfully written this variable using WRITE.

Sara defines the following utility procedures in her implementation of ANTS:

- INPROGRESS (*TID*) returns FALSE if *TID* has committed or aborted, and otherwise TRUE. (All transactions interrupted by a crash are aborted by the RECOVER procedure.)
- EXIT () terminates the current thread immediately.
- LOG () appends a record to the log and waits for the write to the log to complete.
- READ_DATA (*x*) reads cell storage and returns the corresponding value.
- WRITE_DATA (*x*, *v*) writes value v into cell storage x.

Sara now sets out to write pseudocode for READ and WRITE:

```
1  procedure READ (tid, x)     // Return the value stored in cell x
2      if tid < x.WriteID then
3          ABORT (tid)
4          EXIT ()
5      if tid > x.WriteID and INPROGRESS (x.WriteID) then
6          ABORT (tid)             // Last transaction to have written x is still in progress
7          EXIT ()
8      v ← READ_DATA (x)           // In all other cases execute the read
9      x.ReadID ← MAX (tid, x.ReadID)  // Update ReadID of x
10     return v

11 procedure WRITE (tid, x, v)// Store value v in cell storage x
12     if tid < x.ReadID then
13         ABORT (tid)
14         EXIT ()
15     else if tid < x.WriteID then
16         [Mystery Statement I]    // See question 33.3
17     else if tid > x.WriteID and INPROGRESS(x.WriteID) then
18         ABORT (tid)
19         EXIT ()
20     LOG (WRITE, tid, x, READ_DATA (x))
21     WRITE_DATA (x, v)
22     [Mystery Statement II]       // Now update ReadID of x (see question 33.5)
```

Help Sara complete the design above by answering the following questions.

Q 33.2 Consider lines 5–7 of READ. Sara is not sure if these lines are necessary. If lines 5–7 are removed, will the implementation preserve Sara's before-or-after intention?

A. Yes, the lines can be removed. Because the previous WRITE to *x*, by the transaction identified by *x.WriteID*, cannot be affected by transaction *tid*, READ_DATA (*x*) can safely execute.

B. Yes, the lines can be removed. Suppose transaction T_1 successfully executes WRITE (*x*), and then transaction T_2 executes READ (*x*) before T_1 commits. After this, T_1 cannot

execute WRITE (*x*) successfully, so T_2 would have correctly read the last written value of *x* from T_1.

C. No, the lines cannot be removed. One reason is: The only transaction that can correctly execute READ_DATA (*x*) is the transaction with TID equal to *x.WriteID*. Therefore, the condition on line 5 of READ should simply read: "**if** *tid* > *x.WriteID*".

D. No, the lines cannot be removed. One reason is: before-or-after atomicity might not be preserved when transactions abort.

Q 33.3 Consider Mystery Statement I on line 16 of WRITE. Which of the following operations for this statement preserve Sara's before-or-after intention?

A. ABORT (*tid*); EXIT ();
B. **return** (without aborting *tid*)
C. Find the higher-numbered transaction T_h corresponding to *x.WriteID*; ABORT (T_h) and terminate the thread that was running T_h; perform WRITE_DATA (*x*, *v*) in transaction *tid*; and return.
D. All of the above choices.

Q 33.4 Consider lines 17–19 of WRITE. Sara is not sure if these lines are necessary. If lines 17–19 are removed, will Sara's implementation preserve her before-or-after intention? Why or why not?

A. Yes, the lines can be removed. We can always recover the correct values from the log.
B. Yes, the lines can be removed since this is the WRITE call; it's only on a READ call that we need to be worried about the partial results from a previous transaction being visible to another running transaction.
C. No, the lines cannot be removed. One reason is: If transaction T_1 writes to cell *x* and then transaction T_2 writes to cell *x*, then an abort of T_2 followed by an abort of T_1 may leave *x* in an incorrect state.
D. No, the lines cannot be removed. One reason is: If transaction T_1 writes to cell *x* and then transaction T_2 writes to cell *x*, then an abort of T_1 followed by an abort of T_2 may leave *x* in an incorrect state.

Q 33.5 Which of these operations for Mystery Statement II on line 22 of WRITE preserves Sara's before-or-after intention?

A. (*x.WriteID*) ← *tid*
B. (*x.WriteID*) ← MIN(*x.WriteID*, *tid*)
C. (*x.WriteID*) ← MAX(*x.WriteID*, *tid*)
D. (*x.WriteID*) ← MAX(*x.WriteID*, *x.ReadID*)

Ben Bitdiddle looks at the READ and WRITE pseudocode shown before for Sara's system and concludes that her system is in fact nonsensical! To make his case, he constructs the following concurrent transactions:

	Transaction T_1	Transaction T_2
1	$tid_1 \leftarrow$ BEGIN ()	
2		$tid_2 \leftarrow$ BEGIN ()
3	WRITE (tid_1, A, v_1)	
4		$v_2 \leftarrow$ READ (tid_2, A)
5		WRITE (tid_2, B, v_2)
6		COMMIT (tid_2)
7	$v1 \leftarrow$ READ (tid_1, B)	
8	COMMIT (tid_1)	

The two transactions are interleaved in the order shown above. Note that T_1 begins before T_2. Ben argues that this leads to a deadlock.

Q 33.6 Why is Ben's argument incorrect?

A. Both transactions will abort, but they can both retry if they like.
B. Only T_2 will abort on line 4. So T_1 can proceed.
C. Only T_1 will abort on line 7. So T_2 can proceed.
D. Sara's system does not suffer from deadlocks, though concurrent transactions may repeatedly abort and never commit.

Recall that Sara uses a write-ahead log for crash recovery.

Q 33.7 Which of these statements is true about log entries in Sara's ANTS implementation?

A. The order of STARTED entries in the log is in increasing TID order.
B. The order of COMMITTED entries in the log is in increasing TID order.
C. The order of ABORTED entries in the log is in increasing TID order.
D. The order of UPDATED entries in the log for any given variable is in increasing TID order.

Q 33.8 The WRITE procedure appends the UPDATED record to the log before it installs in cell storage. Sara wants to improve performance by caching the non-volatile cell storage in the volatile main memory. She changes READ_DATA to read the value from the cache if it is there; if it isn't, READ_DATA reads from non-volatile cell storage. She changes WRITE_DATA to update just the cache; ANTS will install to non-volatile cell storage later.

Can ANTS delay the install to non-volatile cell storage until after the COMMITTED record has been written to the log, and still ensure transaction semantics?

A. No, because if the system crashed between the COMMIT and the write to non-volatile storage, RECOVER would not recover cell storage correctly.
B. Yes, because the log contains enough information to undo uncommitted transactions after a crash.
C. Yes, because line 3 of READ won't let another transaction read the data until after the write to non-volatile storage completes.
D. None of the above.

2002–3–6…13

34 KeyDB*

(Chapter 9[on-line])

2005–3–13

Keys–R–Us has contracted with you to implement an in-memory key-value transactional store named KeyDB. KeyDB provides a hash table interface to store key-value bindings and to retrieve the value previously associated with a key.

You decide to use locks to provide before-or-after atomicity. Lock L_k is a lock for key k, which corresponds to the entry $KeyDB[k]$. A single transaction may read or write multiple KeyDB entries. Your goal is to achieve correct before-or-after atomicity for all transactions that use KeyDB. Transactions may abort. ACQUIRE (L_k) is called before the first READ or WRITE to $KeyDB[k]$ and RELEASE (L_k) is called after the last access to $KeyDB[k]$.

Q 34.1 For each of the following locking rules, is the rule is **necessary**, **sufficient**, or **neither necessary nor sufficient** to *always* guarantee correct before-or-after atomicity between any set of concurrent transactions?

A. An ACQUIRE (L_k) must be performed after the start of a transaction and before the first READ or WRITE of $KeyDB[k]$, and a RELEASE (L_k) must be performed some time after the last READ or WRITE of $KeyDB[k]$ and before the end of the transaction.

B. ACQUIREs of every needed lock must occur after the start of a transaction and before any other operation, and there can be no RELEASE of a lock before COMMIT or ABORT if the corresponding data item was modified by the thread.

C. ACQUIREs of every needed lock must occur after the start of a transaction and before the first RELEASE, and there can be no RELEASE of a lock before COMMIT or ABORT if the corresponding data item was modified by the thread.

D. All threads that ACQUIRE more than one lock must ACQUIRE the locks in the same order, and there may be no RELEASEs of locks before COMMIT or ABORT.

E. ACQUIREs of every needed lock must occur after the start of a transaction and before the first RELEASE, and a lock may be RELEASEd at at any time after the last READ or WRITE of the corresponding data before COMMIT or ABORT.

* Credit for developing this problem set goes to Hari Balakrishnan.

Q 34.2 Determine whether each of the following locking rules either avoids or is likely (with probability approaching 1 as time goes to infinity) to eliminate permanent deadlock between any set of concurrent transactions.

A. ACQUIREs of every needed lock must occur after the start of a transaction and before any other operation, and there can be no RELEASE of a lock before COMMIT or ABORT.

B. ACQUIREs of every needed lock must occur after the start of a transaction and before the first RELEASE, and there can be no RELEASE of a lock before COMMIT or ABORT.

C. All threads that ACQUIRE more than one lock must ACQUIRE the locks in the same order.

D. When a transaction begins, set a timer to a value longer than the transaction is expected to take. If the timer expires, ABORT the transaction and try it again with a timer set to a value chosen with random exponential backoff.

35 Alice's Reliable Block Store*

(Chapter 9)

2006–3–9

Alice has implemented a version of ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET using only two copies, based an idea she got by reading Section 9.7.1. Her implementation appears below. In her implementation each virtual all-or-nothing sector *x* is stored at two disk locations, *x.D0* and *x.D1*, which are updated and read as follows:

```
// Write the bits in data at item x
1  procedure ALL_OR_NOTHING_PUT (data, x)
2      flag ← CAREFUL_GET (buffer, x.D0);          // read into a temporary buffer
3      if flag = OK then
4          CAREFUL_PUT (data, x.D1);
5          CAREFUL_PUT (data, x.D0);
6      else
7          CAREFUL_PUT (data, x.D0);
8          CAREFUL_PUT (data, x.D1);

// Read the bits of item x and return them in data
1  procedure ALL_OR_NOTHING_GET (reference data, x)
2      flag ← CAREFUL_GET (data, x.D0);
3      if flag = ok then
4          return;
5      CAREFUL_GET (data, x.D1);
```

The CAREFUL_GET and CAREFUL_PUT procedures are as specified in Section 8.5.4.5 and Figure 8.12. The property of these two procedures that is relevant is that CAREFUL_GET can detect cases when the original data is damaged by a system crash during CAREFUL_PUT.

Assume that the only failure to be considered is a fail-stop failure of the system during the execution of ALL_OR_NOTHING_GET or ALL_OR_NOTHING_PUT. After a fail-stop failure the system restarts.

Q 35.1 Which of the following statements are true and which are false for Alice's implementation of ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET?

A. Her code obeys the rule "never overwrite the only copy".

B. ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET ensure that if just one of the two copies is good (i.e., CAREFUL_GET will succeed for one of the two copies), the caller of ALL_OR_NOTHING_GET will see it.

C. ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET ensure that the caller will always see the result of the last ALL_OR_NOTHING_PUT that wrote at least one copy to disk.

Q 35.2 Suppose that when ALL_OR_NOTHING_PUT starts running, the copy at *x.D0* is good. Which statement's completion is the commit point of ALL_OR_NOTHING_PUT?

---

* Credit for developing this problem set goes to Barbara Liskov.

Q 35.3 Suppose that when ALL_OR_NOTHING_PUT starts running, the copy at *x.D0* is bad. For this case, which statement's completion is the commit point of ALL_OR_NOTHING_PUT?

Consider the following chart showing possible states that the data could be in prior running ALL_OR_NOTHING_PUT:

	State 1	State 2	State 3
x.D0	old	old	bad
x.D1	old	bad	old

For example, when the system is in state 2, *x.D0* contains an old value and *x.D1* contains a bad value, meaning that CAREFUL_GET will return an error if someone tries to read *x.D1.*

Q 35.4 Assume that ALL_OR_NOTHING_PUT is attempting to store a new value into item x and the system fails. Which of the following statements are true?

A. (*x.D0* = new, *x.D1* = new) is a potential outcome of ALL_OR_NOTHING_PUT, starting in any of the three states.
B. Starting in state S1, a possible outcome is (*x.D0* = bad, *x.D1* = old).
C. Starting in state S2, a possible outcome is (*x.D0* = bad, *x.D1* = new).
D. Starting in state S3, a possible outcome is (*x.D0* = old, *x.D1* = new).
E. Starting in state S1, a possible outcome is (*x.D0* = old, *x.D1* = new).

Ben Bitdiddle proposes a simpler version of ALL_OR_NOTHING_PUT. His simpler version, named SIMPLE_PUT, would be used with the existing ALL_OR_NOTHING_GET.

```
procedure SIMPLE_PUT (data, x)
    CAREFUL_PUT (data, x.D0)
    CAREFUL_PUT (data, x.D1)
```

Q 35.5 Will the system work correctly if Ben replaces ALL_OR_NOTHING_PUT with SIMPLE_PUT? Explain.

Q 35.6 Now consider failures other than system failures while running the original ALL_OR_NOTHING_PUT. Which of the following statements is true and which false?

A. Suppose *x.D0* and *x.D1* are stored on different disks. Then ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET also mask a single head crash (i.e., the disk head hits the surface of a spinning platter), assuming no other failures.
B. Suppose *x.D0* and *x.D1* are stored as different sectors on the same track. Then ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET also mask a single head crash, assuming no other failures.
C. Suppose that the failure is that the operating system overwrites the in-memory copy of the data being written to disk by ALL_OR_NOTHING_PUT. Nevertheless,

ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET mask this failure, assuming no other failures.

Now consider how to handle decay failures. The approach is to periodically correct them by running a SALVAGE routine. This routine checks each replicated item periodically and if one of the two copies is bad, it overwrites that copy with the good copy. The code for SALVAGE is in Figure 9.38.

Assume that there is a decay interval D such that at least one copy of a duplicated sector will probably still be good D seconds after the last execution of ALL_OR_NOTHING_PUT or SALVAGE on that duplicated sector. Further assume that the system recovers from a failure in less than F seconds, where $F << D$, and that system failures happen so infrequently that it is unlikely that more than one will happen in a period of D seconds.

Q 35.7 Which of the following methods ensures that the approach handles decay failures with very high probability?

A. SALVAGE runs only in a background thread that cycles through the disk with the guarantee that each replicated sector is salvaged every P seconds, where P is less than $(D - F)$.

B. SALVAGE runs as the first step of ALL_OR_NOTHING_PUT, and only then.

C. SALVAGE runs as the first step of both ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET, and only in then.

D. SALVAGE runs on all duplicated sectors as part of recovering from a fail-stop failure and only then.

36 Establishing Serializability*

(Chapter 9[on-line])

2006–3–17

Chapter 9[on-line] explained that one technique of ensuring correctness is to serialize concurrent transactions that act on shared variables, and it offered methods such as version histories or two-phase locking to ensure serialization. Louis Reasoner has come up with his own locking scheme that does not have an easy proof of correctness, and he wants to know whether or not it actually leads to correct results. Louis implements his locking scheme, runs a particular set of three transactions two different times, and observes the order in which individual actions of the transactions occur. The observed order is known as a *schedule*.

Here are Louis's three transactions:

- T1: BEGIN (); WRITE (*x*); READ (*y*); WRITE (*z*); COMMIT ();
- T2: BEGIN (); READ (*x*); WRITE (*z*); COMMIT ();
- T3: BEGIN (); READ (*z*); WRITE (*y*); COMMIT ();

The records *x*, *y* and *z* are stored on disk. Louis's first run produces schedule 1 and his second run produces schedule 2:

	Schedule 1	**Schedule 2**
1	T1: WRITE (*x*)	T3: READ (*z*)
2	T2: READ (*x*)	T2: READ (*x*)
3	T1: READ (*y*)	T1: WRITE (*x*)
4	T3: READ (*z*)	T3: WRITE (*y*)
5	T3: WRITE (*y*)	T1: READ (*y*)
6	T2: WRITE (*z*)	T2: WRITE (*z*)
7	T1: WRITE (*z*)	T1: WRITE (*z*)

The question Louis needs to answer is whether or not these two schedules can be serialized. One way to establish serializability is to create what is called an *action graph*. An action graph contains one node for each transaction and an arrow (directed edge) from T_i to T_j if T_i and T_j both use the same record *r* in conflicting modes (that is, both transactions write *r* or one writes *r* before the other reads *r*) *and* T_i uses *r* first. If for a particular schedule there is a cycle in its action graph, that schedule is *not* serializable. If there is no cycle, then the arrows reveal a serialization of those transactions.

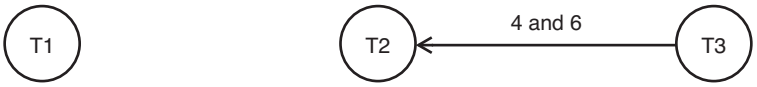
Q 36.1 The table below lists all of the possible arrows that might lead from one transaction to another. For schedule 1, fill in the table, showing whether or not that

* Credit for developing this problem set goes to Barbara Liskov.

arrow exists, and if so list the two steps that create that arrow. To get you started, one row

Arrow	Exists?	Steps
T1 → T2		
T1 → T3		
T2 → T1		
T2 → T3		
T3 → T1		
T3 → T2	Yes	4 and 6

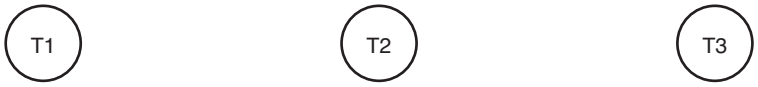
is filled in. And draw the arrows:



Q 36.2 Is schedule 1 serializable? If not, explain briefly why not. If so, give a serial schedule for it.

Q 36.3 Now fill in the table for schedule 2. This time you get to fill in the whole table yourself.

Arrow	Exists?	Steps
T1 → T2		
T1 → T3		
T2 → T1		
T2 → T3		
T3 → T1		
T3 → T2		



Q 36.4 Is schedule 2 serializable? If not, explain why not. If so, give a serial schedule for it.

Q 36.5 Could schedule 2 have been produced by two-phase locking, in which a transaction acquires a lock on an object as the first part of the step in which it first uses that object? For example, step 3 of schedule 2 is the first time that transaction T1 uses record *x*, so it would start that step by acquiring a lock for *x*. Explain.

Louis is also concerned about recovery. When he ran the three transactions and obtained schedule 2, he found that the system generated the following log:

```
1  BEGIN (transaction: T1)
2  BEGIN (transaction: T2)
3  BEGIN (transaction: T3)
4  CHANGE (transaction: T1, record: x, undo: 1, redo: 2)
5  CHANGE (transaction: T3, record: y, undo: 1, redo: 2)
6  CHANGE (transaction: T2, record: z, undo: 1, redo: 2)
7  COMMIT (transaction: T3)
8  COMMIT (transaction: T2)
9  CHANGE (transaction: T1, record: z, undo: 2, redo: 3)
10 COMMIT (transaction: T1)
```

In a CHANGE record the undo field gives the old value before this change, and the redo field gives the new value afterwards. For example, entry 4 indicates that the old value of *x* was 1 and the new value is 2. The system uses the redo/undo recovery procedure of Figure 9.23.

Q 36.6 Suppose the system crashed after record 7 of the log has made it to disk but before record 8 is written. What states do *x*, *y*, and *z* have after recovery is complete?

Q 36.7 Suppose instead that the system crashed after record 9 of the log has made it to disk but before record 10 is written. What states do *x*, *y*, and *z* have after recovery is complete?

Louis's database consists of a collection of integer objects stored on disk. Each WRITE operation increments by 1 the object being modified. The system is using a write-ahead logging protocol and there is an in-memory cache that the system periodically flushes to disk, without checking to see if the cached objects belong to committed transactions.

To save space in the log, Louis's friend Ben Bitdiddle suggests that CHANGE records could just indicate the operation that was performed. For example, log entry 4 would be:

```
4  CHANGE (transaction: T1, record: x, operation: increment)
```

When the recovery manager sees this entry, it performs the specified operation: increment *x* by 1. Ben makes no other changes to the recovery protocol.

Q 36.8 All objects are initialized to 0. Louis tries Ben's plan, but after the first system crash and recovery he discovers that it doesn't work. Explain why.

37 Improved Bitdiddler*

(Chapter 9[on-line])

2007–3–8

Alyssa points out Ben's Bitdiddler with synchronous block writes (see problem set 5) doesn't guarantee that file system calls (e.g., WRITE, CLOSE, etc.) provide all-or-nothing atomicity. She suggests that Ben use a logging approach to help provide all-or-nothing atomicity for each file system call.

She proposes that the file system synchronously write a log record before every CREATE, WRITE, or UNLINK call. Each log record contains the type of operation performed, the name of the file, and for writes the old and new values of the data as well as the offset where the new data will be written. The system ensures that log record writes are atomic and it places the log records in a separate log file on a separate disk.

Ben modifies the Bitdiddler code to perform these logging operations before doing the create, write, or unlink operations themselves. He also implements a crash recovery protocol that scans the log after a crash as part of a crash recovery protocol intended to ensure all-or-nothing atomicity.

Q 37.1 Which of the following crash recovery protocols ensures that file system calls are all-or-nothing (assuming there was at most one file system call running when the system crashed)?

- **A.** Scan the log from the beginning to the end; re-apply each logged operation to the specified file in forward-scan order.
- **B.** Scan the log from the end to the beginning; re-apply each logged operation to the specified file in reverse-scan order.
- **C.** Read the last log record and re-apply it.
- **D.** Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from beginning to end, re-doing CREATEs and WRITEs for those files in forward-scan order.
- **E.** Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from the end to the beginning, re-doing CREATEs and WRITEs for those files in reverse-scan order.

High-Performance Logging Bitdiddler. Ben observes that synchronous writes slow down the performance of his file system. To improve performance with this logging approach, Ben modifies the Bitdiddler to include a large file system cache. He arranges that WRITE, CREATE, and UNLINK update blocks in the cache. To maximize performance, the file system propagates these modified blocks to disk asynchronously, in an arbitrary order, and at a time of its own choosing. Ben's file system still writes log records synchro-

* Credit for developing this problem set goes to Sam Madden.

nously to ensure that these are on disk *before* executing the corresponding file system operation.

Q 37.2 Which of the following crash recovery protocols ensures that file system calls are all-or-nothing in this high performance version of the Bitdiddler (assuming there was at most one file system call running when the system crashed)?

A. Scan the log from the beginning to the end; re-apply each logged operation to the specified file in forward-scan order.
B. Scan the log from the end to the beginning; re-apply each logged operation to the specified file in reverse-scan order.
C. Read the last log record and re-execute it.
D. Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from beginning to end, re-doing CREATEs and WRITEs for those files in forward-scan order.
E. Scan the log from the beginning to end and identify all the files that should have been created but don't exist (e.g., don't have an inode and were not deleted). Then, scan the log from the end to the beginning, re-doing CREATEs and WRITEs for those files in reverse-scan order.

Q 37.3 Alyssa suggests that Ben might want to modify his system to periodically write checkpoints to make recovery efficient. Which of the following checkpoint protocols will allow Ben's recovery code to start recovering from the latest checkpoint while still ensuring all-or-nothing atomicity of each file system call in the high performance, asynchronous Bitdiddler?

A. Complete any currently running file system operation (e.g., OPEN, WRITE, UNLINK, etc.), stop processing new file system operations, write all modified blocks in the file system cache to disk, and then write a checkpoint record to the log containing a list of open files.
B. Complete any currently running file system operation, stop processing new file system operations, write all modified blocks in the file system cache to disk, and then write a checkpoint record to the log containing no additional information.
C. Write all modified blocks in the file system cache to disk without first completing current file system operations, and then write a checkpoint record to the log containing a list of open files.
D. Write a checkpoint record to the log (containing a list of open files), but do not write all modified blocks to disk.

Transactional Bitdiddler. By now, Ben is really excited about his file system so he decides to add some advanced features. From studying Chapter 9, he knows that transactions are a way to make multiple operations appear as though they are a single before-or-after, all-or-nothing atomic action, and he decides he would like to make his file system transactional, so that programs can commit changes to several files as a part of one

transaction, and so that concurrent users of the file system don't ever see the effects of others' partially complete transactions. He adds three new procedures:

- *tid* ← BEGIN_TRANSACTION ()
- COMMIT (*tid*)
- ABORT (*tid*)

Ben renames his existing OPEN procedure to DO_OPEN so that he can insert a layer named OPEN () that takes a *tid* parameter that specifies the transaction that this file access will be a part of. Ben's plan is that one transaction can OPEN, READ, and WRITE multiple files, but those changes be visible to other transactions only after the originating transaction calls COMMIT. If the system crashes before a transaction COMMITs, its actions are undone during recovery.

Ben decides to use locking to ensure before-or-after atomicity. He places a single exclusive lock on each file, and programs OPEN to attempt to ACQUIRE that lock before returning. If another transaction currently holds the lock, OPEN waits until the lock is free.

Here is the implementation of Ben's new OPEN procedure:

```
procedure OPEN ( tid, file_name)
    integer locking_tid
    do
        locking_tid ← TEST_AND_ACQUIRE_LOCK (file_name, tid)
    while locking_tid != tid
    return DO_OPEN (file_name)          // returns a file handle.
```

TEST_AND_ACQUIRE_LOCK () tests to see if the lock is currently acquired by some transaction, and if it is, returns the id of the locking transaction. If the lock is not currently acquired, it acquires the lock on behalf of *tid*, and returns *tid*.

Ben modifies his logging code so that each log record includes the *tid* of the transaction it belongs to and adds COMMIT and ABORT records to indicate the outcome of transactions.

Ben is writing the code for the CLOSE and COMMIT functions, and is trying to figure out when he should release the locks acquired by his transaction. His code is as follows:

```
procedure CLOSE (file_handle)
    remove file_handle from file handle list
    A:

procedure COMMIT (tid)
    file_handles[] ← GET_FILES_LOCKED_BY (tid)
    for each f in file_handles do
        if IS_OPEN (f) then CLOSE (f)
    B:
    log a COMMIT record for tid              // commit point
    C:
```

Note that COMMIT first closes any open files, though files may also be closed before COMMIT is called.

Q 37.4 When can Ben's code release a lock on a file (or all files) while still ensuring that the locking protocol implements before-or-after atomicity?

A. At the line labeled A:
B. At the line labeled B:
C. At the line labeled C:

Ben begins running his new transactional file system on the Bitdiddler. The Bitdiddler allows multiple programs to run concurrently, and Ben is concerned that he may have a bug in his implementation because he finds that sometimes some of his applications block forever waiting for a lock. Alyssa points out that he may have deadlocks.

Ben hires you to help him figure out whether there is a bug in his code or if applications are just deadlocking. He shows you several traces of file system calls from several programs; your job is to figure out out for each trace whether the operations indicate a deadlock, and if not, to report what apparent before-or-after order the transactions shown in the trace appeared to have run.

At the end of each trace, assume that any uncommitted transactions issue no more READ or OPEN calls but that each uncommitted transaction will go on to COMMIT if it has not deadlocked.

Alyssa helps out by analyzing and commenting on the first trace for you. In these traces, time goes down the page; so the first one shows that the first action is BEGIN (*T1*) and the second action is BEGIN (*T2*):

Alyssa's sample annoted program trace:

Transaction 1:	**Transaction** 2:
BEGIN (*T1*)	
	BEGIN (*T2*)
fh ← OPEN (*T1*,'foo')	
	fh2 ← OPEN (*T2*,'foo') // blocks waiting for *T1*
WRITE (*fh*,'hi')	
CLOSE (*fh*)	
COMMIT (*T1*)	
	WRITE (*fh2*,'hello') // *T2* can commit without deadlocking

The result is as if these transactions ran in the order T1, then T2.

Q 37.5 **Trace 1**: Does the following set of three transactions deadlock? If not, what serial ordering of these transactions would produce the same result?

```
Transaction 1:      Transaction 2:      Transaction 3:

BEGIN (T1)
                    BEGIN (T2)
                                        BEGIN (T3)
fh ← OPEN (T1,'foo')
                    fh2 ← OPEN (T2,'bar')
                                        fh3 ← OPEN (T3,'baz')
WRITE (T1,'hi')
                    fh4 ← OPEN (T2,'baz')
                                        fh5 ← OPEN (T3,'foo')
CLOSE (fh)
COMMIT (T1)
```

Q 37.6 **Trace 2**: Does the following set of four transactions deadlock? If not, what serial ordering of these transactions would produce the same result?

```
Transaction 1:      Transaction 2:      Transaction 3:      Transaction 4:

BEGIN (T1)          BEGIN (T2)
fh ← OPEN (T1,'foo')
WRITE (fh,'boo')
                    fh2 ← OPEN (T2,'bar')
                    WRITE (fh2,'car')
CLOSE (fh)
COMMIT (T1)
                                        BEGIN (T3)
                                                            BEGIN (T4)
                                                            fh3 ← OPEN (T4,'foo')
                                        fh4 ← OPEN (T3,'bar')
                    fh5 ← OPEN (T2,'foo')
                                                            fh6 ← OPEN (T4,'bar')
                      ...                   ...                 ...
```

Q 37.7 **Trace 3**: Does the following set of four transactions deadlock? If not, what serial ordering of these transactions would produce the same result?

```
Transaction 1:          Transaction 2:          Transaction 3:          Transaction 4:

BEGIN (T1)
                        BEGIN (T2)
fh ← OPEN (T1,'foo')
WRITE (fh,'boo')
                        fh2 ← OPEN (T2,'bar')
                        WRITE (fh2,'car')
CLOSE(fh)
COMMIT (T1)
                                                BEGIN (T3)
                                                                        BEGIN (T4)
                                                                        fh3 ← OPEN (T4,' foo')
                                                fh4 ← OPEN (T3,'foo')
                        fh5 ← OPEN (T2,'foo')
                                                                        fh6 ← OPEN (T4, 'baz')
                            …                   …                       …
```

Transactional, Distributed Bitdiddler. Ben begins to get really carried away. He decides that he wants the Bitdiddler to be able to access files of remote Bitdiddlers via a networked file system protocol, but he wants to preserve the transactional behavior of his system, such that one transaction can update files on several different computers. He remembers that one way to provide atomicity when there are multiple participating sites is to use the two-phase commit protocol.

The protocol works as follows: one site is appointed the coordinator. The program that is reading and writing files runs on this machine, and issues requests to BEGIN and COMMIT transactions and READ and WRITE files on both the local and remote file systems (the "workers").

When the coordinator is ready to commit, it uses the logging-based two-phase commit protocol, which works as follows: First, the coordinator sends a PREPARE message to each of the workers. For each worker, if it is able to commit, it writes a log record indicating it is entering the PREPARED state and send a YES vote to the coordinator; otherwise it votes NO. If all workers vote YES, the coordinator logs a COMMIT record and sends a COMMIT outcome message to all workers, which in turn log a COMMIT record. If any worker votes NO, the coordinator logs an ABORT record and sends an ABORT message to the workers, which also log ABORT records. After they receive the transaction outcome, workers send an ACKNOWLEDGMENT message to the coordinator. Once the coordinator has received an acknowledgment from all of the workers, it logs an END record. Workers that have not learned the outcome of a transaction periodically contact the coordinator asking for the outcome. If the coordinator does not receive an ACKNOWLEDGMENT from some worker, after a timer expiration it resends the outcome to that worker, persistently if necessary.

Figure PS.5 shows a coordinator node issuing requests to BEGIN a transaction and to READ and WRITE files on two worker nodes.

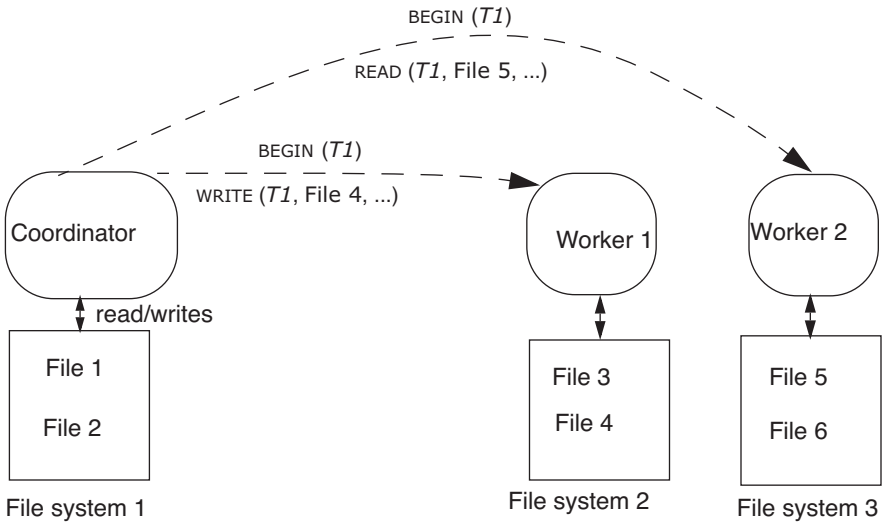


FIGURE PS.5

Coordinator issuing transactional READs and WRITEs to two workers in the two-phase commit based distributed file system for the Bitdiddler.

Ben is having a hard time figuring out what to do when one of the nodes crashes in middle of the two phase commit protocol. When a worker node restarts and finds log records for a transaction, it has several options:

- W1. Abort the transaction by writing an "abort" record
- W2. Commit the transaction by writing a "commit" record
- W3. Resend its vote to the server and ask it for transaction outcome

Similarly, the coordinator has several options when it recovers from a crash. It can:

- C1. Abort the transaction by writing an ''abort'' record
- C2. Do nothing
- C3. Send commit messages to the workers

Q 37.8 For each of the following situations, of the above actions choose the best action that a worker or coordinator should take.

A. The coordinator crashes, finds log records for a transaction but no COMMIT record
B. The coordinator crashes, finds a COMMIT record for a transaction but no END record indicating the transaction is complete
C. A worker crashes, finds a PREPARE record for a transaction
D. A worker crashes, and finds log records for a transaction, but no PREPARE or COMMIT records

38 Speedy Taxi Company*

(Chapter 9[on-line])

2008–2–9

The Speedy Taxi company uses a computer to help its dispatcher, Arnie. Customers call Arnie, each asking for a taxi to be sent to a particular address, which Arnie enters into the computer. Arnie can also ask the computer to assign the next waiting address to an idle taxi; the computer indicates the address and taxi number to Arnie, who informs that taxi over his two-way radio.

Arnie's computer stores the set of requested addresses and the current destination address of each taxi (if not idle) in an in-memory database. To ensure that this information is not lost in a power failure, the database logs all updates to an on-disk log. Since the database is kept in volatile memory only, the state must be completely reconstructed after a power failure and restart, as in Figure 9.22. The database uses write-ahead logging as in Chapter 9: it always appends each update to the log on disk, and waits for the disk write to the log to complete before modifying the cell storage in main memory. The database processes only one transaction at a time (since Arnie is the only user, there is no concurrency).

The database stores the list of addresses waiting to be assigned to taxis as a single variable; thus any change results in the system logging the entire new list. The database stores each taxi's current destination as a separate variable. A taxi is idle if it has no address assigned to it.

Consider one action that uses the database: DISPATCH_ONE_TAXI. Arnie's computer presents a UI to him consisting of a button marked DISPATCH_ONE_TAXI. When Arnie presses the button, and there are no failures, the computer takes one address from the list of addresses waiting to be assigned, assigns it to an idle taxi, and displays the address and taxi to Arnie.

Here is the code for DISPATCH_ONE_TAXI:

* Credit for developing this problem set goes to Robert T. Morris.

```
1  procedure DISPATCH_ONE_TAXI ()
2      BEGIN_TRANSACTION
3                  // read and delete the first address in list
4          list ← READ ()
5          if LENGTH (list) < 1 then
6              ABORT_TRANSACTION
7          address ← list[0]
8          DELETE (list[0])
9          WRITE (list)
10                 // find first free taxi
11         taxi_index ← -1
12         for i from 0 until NUMBER_OF_TAXIS - 1
13             taxis[i] ← READ ()
14             if taxis[i] = NULL and taxi_index = -1 then
15                 taxi_index ← i
16             if taxi_index = -1 then
17                 ABORT_TRANSACTION
18                 // record address as the taxi's destination
19         taxis[taxi_index] ← address
20         WRITE (taxis[taxi_index])
21     COMMIT_TRANSACTION
22     display "DISPATCH TAXI " + taxi_index + " TO " + address
```

When Arnie starts work, *list* contains exactly two addresses a_1 and a_2. There are two taxis (*taxis*[0] and *taxis*[1]) and both are idle (NULL). Arnie pushes the DISPATCH_ONE_TAXI button, but he sees no DISPATCH TAXI display, and the computer crashes, restarts, and runs database recovery. Arnie pushes the button a second time, again sees no DISPATCH TAXI display, and again the computer crashes, restarts, and runs recovery. There is no activity beyond that described or necessarily implied.

Q 38.1 If you were to look at last few entries of the database log at this point, which of the following might you see, and which are not possible? Bx stands for a BEGIN record for transaction ID *x*, Mx is a MODIFY (i.e. change) record for the indicated variable and new value, and Cx is a COMMIT record.

A. No log records corresponding to Arnie's actions.
B. B101; M101 *list*=a_2; M101 *taxis*[0]=a_1; C101; B102; M102 *list*=(empty); M102 *taxis*[1]=a_2; C102
C. B101; M101 *list*=a_2; M101 *taxis*[0]=a_1; B102; M102 *list*=(empty); M102 *taxis*[1]=a_2
D. B101; M101 *list*=a_2; M101 *taxis*[0]=a_1; C101; B102; M102 *list*=a_2; M102 *taxis*[0]=a_1
E. B101; M101 *list*=a_2; M101 *taxis*[0]=a_1; B102; M102 *list*=a_2; M102 *taxis*[0]=a_1

Suppose again the same starting state (the address list contains a_1 and a_2, both taxis are idle). Arnie pushes the button, the system crashes without displaying a DISPATCH TAXI message, the system reboots and runs recovery, and Arnie pushes button again.

This time the system does display a DISPATCH TAXI message. Again, there is no activity beyond that described or necessarily implied.

Q 38.2 Which of the following are possible messages?

A. DISPATCH TAXI 0 TO a_1
B. DISPATCH TAXI 0 TO a_2
C. DISPATCH TAXI 1 TO a_1
D. DISPATCH TAXI 1 TO a_2

Arnie questions whether it's necessary to make the whole of DISPATCH_ONE_TAXI a single transaction. He suggests that it would work equally well to split the program into two transactions, the first comprising lines 2 through 9, and the other comprising lines 12 through 21. Arnie makes this change to the code.

Suppose again the same starting state and no other activity. Arnie pushes the button, the system crashes without displaying a DISPATCH TAXI message, the system reboots and runs recovery, and Arnie pushes button again. This time the system displays a DISPATCH TAXI message.

Q 38.3 Which of the following are possible messages?

A. DISPATCH TAXI 0 TO a1
B. DISPATCH TAXI 0 TO a2
C. DISPATCH TAXI 1 TO a1
D. DISPATCH TAXI 1 TO a2

39 Locking for Transactions*

(Chapter 9[on-line])

2008–3–14

Alyssa has devised a database that uses logs as described in Section 9.3. The logging and recovery works as shown in Figure 9.22 (the in-memory database with write-ahead logging). Alyssa claims that if programmers insert ACQUIRE and RELEASE calls properly they can have transactions with both before-or-after and all-or-nothing atomicity.

Alyssa has programmed the following transaction as a demonstration. As Alyssa claims, it has both before-or-after and all-or-nothing atomicity.

```
T1:
    BEGIN_TRANSACTION ()
    ACQUIRE (X.lock)
    ACQUIRE (Y.lock)
    X ←X + 1
    if X = 1 then
        Y ← Y + 1
    COMMIT_TRANSACTION()
    RELEASE (X.lock)
    RELEASE (Y.lock)
```

X and Y are the names of particular database fields, not parameters of the transaction.

Q 39.1 The database starts with contents X=0 and Y=0. Two instances of T_1 are started at about the same time. There are no crashes, and no other activity. After both transactions have finished, which of the following are possible database contents?

A. X=1 Y=1
B. X=2 Y=0
C. X=2 Y=1
D. X=2 Y=2

Ben changes the code for T_1 to RELEASE the locks earlier:

```
T1b:
    BEGIN_TRANSACTION ()
    ACQUIRE (X.lock)
    ACQUIRE (Y.lock)
    X ←X + 1
    if X = 1 then
        Y ← Y + 1
    RELEASE (X.lock)
    RELEASE (Y.lock)
    COMMIT_TRANSACTION ()
```

With this change, Louis suspects that there may be a flaw in the program.

* Credit for developing this problem set goes to Robert T. Morris.

Q 39.2 The database starts with contents X=0 and Y=0. Two instances of T_{1b} are started at about the same time. There is a crash, a restart, and recovery. After recovery completes, which of the following are possible database contents?

A. X=1 Y=1
B. X=2 Y=0
C. X=2 Y=1
D. X=2 Y=2

Ben and Louis devise the following three transactions. Beware: the locking in T_2 is flawed.

T_2:
```
BEGIN_TRANSACTION ()
ACQUIRE (M.lock)
temp ← M
RELEASE (M.lock)
ACQUIRE (N.lock)
N ← N + temp
COMMIT_TRANSACTION ()
RELEASE (N.lock)
```

T_3:
```
BEGIN_TRANSACTION ()
ACQUIRE (M.lock)
M ← 1
COMMIT_TRANSACTION ()
RELEASE (M.lock)
```

T_4:
```
BEGIN_TRANSACTION ()
ACQUIRE (M.lock)
ACQUIRE (N.lock)
M ← 1
N ← 1
COMMIT_TRANSACTION ()
RELEASE (M.lock)
RELEASE (N.lock)
```

Q 39.3 The initial values of M and N in the database are M=2 and N=3. Two of the above transactions are executed at about the same time. There are no crashes, and there is no other activity. For each of the following pairs of transactions, decide whether concurrent execution of that pair could result in an incorrect result. If the result is always correct, give an argument why. If an incorrect result could occur, give an example of such a result and describe a scenario that leads to that result.

A. T_2 and T_2:
B. T_2 and T_3:
C. T_2 and T_4:

40 "Log"-ical Calendaring*

(Chapters 9[on-line] and 10[on-line])

Ally Fant is designing a calendar server to store her appointments. A calendar client contacts the server using the following remote procedure calls (RPCs):

- ADD (*timeslot*, *descr*): Adds the appointment description (*descr*) to the calendar at time slot *timeslot*.
- SHOW (*timeslot*): Reads the appointment at time slot *timeslot* from the calendar and displays it to the user. (If there is no appointment, SHOW displays an empty slot.)

The RPC between client and server runs over a transport protocol that provides "at-most-once" semantics.

The server runs on a separate computer and it stores appointments in an append-only log on disk. The server implements ADD in response to the corresponding client request by appending an appointment entry to the log. Each appointment entry has the following format:

```
structure appt_entry
    integer id          // unique id of action that created this entry
    string timeslot     // the timeslot for this appointment
    string descr        // description of this appointment
```

Ally would like to make the ADD action atomic. She realizes that she can use ALL_OR_NOTHING_PUT (*data*, *sector*) and ALL_OR_NOTHING_GET (*data*, *sector*) as described in Section 9.2.1. These procedures guarantee that a single all-or-nothing sector is written either completely or not at all.

Each appointment entry is for one *timeslot*, which specifies the time interval of the appointment (e.g., 1:30 pm–3:00 pm on May 20, 2005). Each appointment entry is exactly as large as a single all-or-nothing sector (512 bytes). The first all-or-nothing sector on disk, numbered 0, is the *master_sector*, which stores the all-or-nothing sector number where the next log record will be written. The number stored in *master_sector* is called the end of the log, *end_of_log*, and is initialized to 1.

Ally designs the following procedure:

```
1    procedure ADD (timeslot; descr)
2        id ← NEW_ACTION_ID ()                          // returns a unique identifier
3        appt ← MAKE_NEW_APPT (id; timeslot; descr) // make and fill in an appt entry
4        if ALL_OR_NOTHING_GET (end_of_log; master_sector) ≠ OK then return
5        if ALL_OR_NOTHING_PUT (appt; end_of_log) ≠ OK then return
6        end_of_log ← end_of_log + 1
7        if ALL_OR_NOTHING_PUT (end_of_log; master_sector) ≠ OK then return
```

The procedure NEW_ACTION_ID returns a unique action identifier. The procedure

* Credit for developing this problem set goes to Hari Balakrishnan.

MAKE_NEW_APPT allocates an *appt_entry* structure and fills it in, padding it to 512 bytes.

Ally implements SHOW as follows:

1. Use ALL_OR_NOTHING_GET to read the master sector to determine the end of the log.
2. Scan the log backwards starting from the last written all-or-nothing sector (*end_of_log* – 1), using ALL_OR_NOTHING_GET on each sector, and stopping as soon as an entry for the timeslot is found.

To help understand if her implementation of the calendar system is correct or not, Ally defines the following properties that her calendar server should ensure:

P1: SHOW (*timeslot*) should display the appointment corresponding to the last committed ADD to that timeslot, even if system crashes occur during calls to ADD.

P2: The calendar server must store the appointments corresponding to all committed ADD actions for at least three years.

P3: If multiple ADD and SHOW actions run concurrently, their execution should be serializable and property P1 should hold.

P4: No ADD should be committed if it has a time slot that overlaps with an existing appointment.

Ally has learned a number of apparently relevant concepts: before-or-after atomicity, all-or-nothing atomicity, constraint, durability, and transaction.

Q 40.1 Which of the apparently relevant concepts does ADD correctly implement?

Q 40.2 For each of the properties P2, P3, and P4, identify the apparently relevant concept that best describes it.

Q 40.3 What is the earliest point in the execution of the ADD procedure that ensures that a subsequent SHOW is guaranteed to observe the changes made by the ADD. (Assume that the SHOW does not fail.)

A. The successful completion of ALL_OR_NOTHING_PUT in line 5 of ADD.
B. The successful completion of line 6.
C. The successful completion of ALL_OR_NOTHING_PUT in line 7.
D. The instant that ADD returns to its caller.

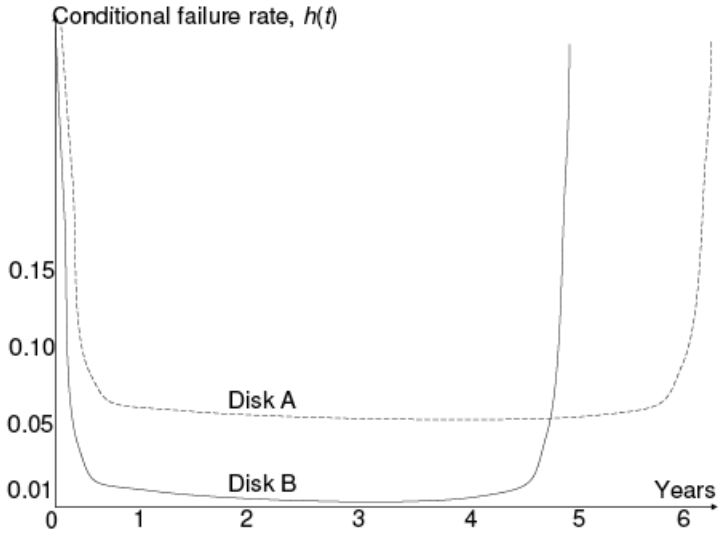
Q 40.4 Ally sometimes uses the calendar server concurrently from different client machines. Which of these statements is true of properties P3 and P4? (Assume that no failures occur, but that the server may be processing multiple RPCs concurrently.)

A. If exactly one ADD and several SHOW actions run concurrently on the server, then property P3 is satisfied even if those actions are for the same timeslot.
B. If more than one ADD and exactly one SHOW run concurrently on the server, then property P3 is satisfied as long as the actions are for different timeslots.
C. Suppose ADD (*timeslot*, *descr*) calls SHOW (*timeslot*) before line 7 and immediately returns to its caller if the timeslot already has an appointment. If multiple ADD and SHOW

actions run concurrently on the server, then property P4 is satisfied whether or not property P3 holds.

D. Suppose ADD (*timeslot*, *descr*) calls SHOW (*timeslot*) before line 7 and immediately returns to its caller if the timeslot already has an appointment. If multiple ADD and SHOW actions run concurrently on the server, then property P4 is satisfied as long as property P3 holds.

Q 40.5 Ally finds two disks A and B whose conditional failure probabilities follow the "bathtub curve", shown below. She also learns that the disk manufacturers sell units that have been "burned in," but otherwise are unused. Which disk should she buy new to have a higher likelihood of meeting property P2 for at least one year?



Multi-user calendar. Ally becomes president of Scholarly University and opens her server calendar to the entire University community to add and show entries. People start complaining that it takes a long time for them to SHOW Ally's appointments. Ally's new provost, Lem E. Fixit, tells her that a single log makes reading slow.

Lem convinces Ally to use the log as a recovery log, and use a volatile in-memory table, named table, to store the appointments to improve the performance of SHOW. The table is indexed by the timeslot. SHOW is now a simple table lookup, keyed by the timeslot.

If the system crashes, the table is lost; when the system recovers, the recovery procedure reinstalls the table. Lem shows Ally how to modify the recovery log to include an "undo" entry in it, as well as a "redo" entry. All the log writes are done using ALL_OR_NOTHING_PUT.

Ally writes the following lines in her NEW_ADD pseudocode. (For now, the writes to the log are only shown in COMMIT.)

```
1  procedure NEW_ADD (timeslot, descr)
2      id ← NEW_ACTION_ID ()
3      appt ← MAKE_NEW_APPT (id, timeslot, descr)
4      table[timeslot] ← appt
5      if OVERLAPPING (table, appt) then ABORT (id)
6      COMMIT (id)

7  procedure COMMIT (id)
8      if ALL_OR_NOTHING_GET (end_of_log, master_sector) ≠ OK then ABORT (id)
9      if ALL_OR_NOTHING_PUT ("COMMITTED", id, end_of_log) ≠ OK then ABORT (id)
10     end_of_log ← end_of_log + 1
11     if ALL_OR_NOTHING_PUT (end_of_log, master_sector) ≠ OK then ABORT (id)
```

The procedure named OVERLAPPING checks *table* to see if *appt* overlaps with a previously committed appointment (property P4). ABORT uses the log to undo any changes to table made by NEW_ADD, releases any locks that NEW_ADD set, and then terminates the action.

Ally modifies SHOW to look up an appointment in *table*, instead of scanning the log.

Q 40.6 Which of the following statements is true for NEW_ADD with respect to property P1? (Assume that there are no concurrent actions.)

A. If NEW_ADD writes the log entry corresponding to the *table* write just before line 4, then P1 holds.

B. If NEW_ADD writes the log entry corresponding to the table write just before line 6, then P1 holds.

C. Because table is in volatile memory, there is no need for ABORT to undo any changes made by NEW_ADD in order for P1 to hold.

D. If Ally had designed table to be in non-volatile storage, and NEW_ADD inserts the log entry just before line 4, then P1 holds.

Lem convinces Ally that using locks can be a good way to ensure property P3. Ally uses two locks, λ_t and λ_g. λ_t protects *table*[*timeslot*] and λ_g protects accesses to the log. She needs help to figure out where to place the lock ACQUIRE and RELEASE statements to ensure that property P3 holds when multiple concurrent NEW_ADD and SHOW actions run.

Q 40.7 Which of the following placements of ACQUIRE and RELEASE statements in NEW_ADD correctly ensures property P3? Assume that SHOW implements correct locking.

A.
ACQUIRE (λ_t) just before line 3,
RELEASE (λ_t) just after line 6,
ACQUIRE (λ_g) just before line 3,
RELEASE (λ_g) just after line 6.

B.
ACQUIRE (λ_t) just before line 4,
RELEASE (λ_t) just after line 5,
ACQUIRE (λ_g) just before line 6 but after RELEASE(λ_t),
RELEASE (λ_g) just after line 6.

C. None of the above.

Disconnected calendar. Ally Fant wants to use her calendar in disconnected operation, for example, from her PDA, cell phone, and laptop. Ally modifies the client software as follows. Just before a client disconnects, the client copies the log from the calendar server atomically, and then reinstalls *table* locally. When the user (i.e., Ally) adds an item, the client runs NEW_ADD on the client, updating the local copy of the log and *table*.

When the client can connect to the calendar server or any other client, it reconciles. When reconciling, one of the two machines is the primary. If a client connects to the calendar server, the server is the primary; if a client connects to another client, then one of them is the primary. The client that is not the primary calls RECONCILE, which runs locally on the client:

```
1  procedure RECONCILE (primary, client_log)
2      for each entry ∈ client_log do
3          if entry.state = COMMITTED then
4              invoke NEW_ADD(entry.timeslot, entry.descr) at primary
5      COPY (primary.log, client_log)          // overwrite client_log
6      DELETE (table)
7      rebuild table from client_log           // create new table
```

Assume that RECONCILE is atomic and that no crashes occur during reconciliation. Assume also that between any pair of nodes there is at most one active RECONCILE at any time.

Q 40.8 Which of the following statements is true about the implementation that supports disconnected operation?

- **A.** RECONCILE will resolve overlapping appointments in favor of appointments already present on the primary.
- **B.** Some appointments added on a disconnected client may not appear in the output of SHOW after the reconciliation is completed.
- **C.** The result of client C1 reconciling with client C2 (with C2 as the primary), and then reconciling C2 with the calendar server, is the same as reconciling C2 with client C1 (with C1 as the primary), and then reconciling C1 with the calendar server.
- **D.** Suppose Ally stops making changes, and then reconciles all clients with the server once. After doing that, the logs on all machines will be the same.

Lem E. Fixit notices that the procedure RECONCILE is slow. To speed it up, Lem invents a new kind of record, called the "RECONCILED" record. Each time RECONCILE runs, it appends a RECONCILED record listing the client's unique identifier to the primary's log just before line 5.

Q 40.9 Which of the following uses of the RECONCILED record speeds up RECONCILE correctly? (Assume that clients reconcile only with the calendar server.)

A. Modify line 2 to scan the client log backwards (from the end of the log), terminating the scan if a RECONCILED record with the client's identifier is found, and then scan forward until the end of the log calling NEW_ADD on the appointment entries in the log.

B. Modify line 2 to scan the client log forwards (from the beginning of the log) calling NEW_ADD on the appointment entries in the log, but terminating the scan if a RECONCILED record with the client's identifier is found.

C. Don't reinstall table from scratch at the end of reconciliation, but instead update it by adding the entries in the primary log (which the client just copied) that are between the previous RECONCILED record and the RECONCILED record from the current reconciliation. If an entry in the log overlaps with an entry in the table, then replace the table entry with the one in the log.

D. Assign Lem E. Fixit a different job. None of these optimizations maintains correctness.

2004–3–7…15

41 Ben's Calendar*

(Chapter 10[on-line])

Ben Bitdiddle has just been promoted to Engineering Manager. He quickly notices two facts about his new job. First, keeping an accurate appointment calendar is crucial. Second, he no longer has any programming responsibilities. He decides to address both problems at once by building his own highly available replicated calendar system.

Ben runs a client user interface on his workstation. The client talks over the network to one of three replicated servers. Ben places the three servers, called S1, S2, and S3, in three different cities to try to ensure independent failure. Ben only runs one client at a time.

Each server keeps a simple database of Ben's appointments. The database holds a string for every hour of every day, describing the appointment for that hour. The string for each hour starts out empty. A server can perform just two operations on its own database:

- DBREAD (*day*, *hour*) returns the appointments for a particular day and hour. The argument *day* indicates the desired day, where 0 means January 1st, 2000. The argument *hour* is an integer between 0 and 23, inclusive.
- DBWRITE (*day*, *hour*, *string*) changes the string for the hour *hour*. Writing an empty string to an hour effectively deletes any existing appointment for that hour.

Each server allows Ben's client to invoke these operations by RPC. The RPC system uses a powerful checksum that detects all errors and discards any corrupted message. If the RPC client implementation doesn't receive a response from the server within a few seconds, it times out, sets the variable *rpc_OK* to false, and returns NIL. If the client does receive a reply from the server, the RPC implementation sets *rpc_OK* to true and returns the result from the server, if any. The RPC system does not resend requests. Thus, for example, if the network discards or corrupts the request or response message, the RPC call returns with *rpc_OK* set to false.

Ben's client user interface can display the appointments for a day and also change an appointment. To support these operations, Ben writes client software based on this pseudocode (the notation S[*i*].F indicates an RPC call to procedure F on server *i*):

* Credit for developing this problem set goes to Robert T. Morris.

```
procedure CLIENTREAD (day, hour)
    string s
    for i from 1 to 4 do                // try each server one by one
        s ← S[i].DBREAD (day, hour)
        if rpc_OK then return s         // return with the first reply
    return "Error"

procedure CLIENTWRITE (day, hour, what)
    for i from 1 to 4 do                // write to all three servers
        boolean done ← FALSE
        while done = FALSE do
            S[i].DBWRITE (day, hour, what)
            if rpc_OK then done ← TRUE
```

Q 41.1 Suppose the network connecting Ben's client to servers S1 and S2 is fast and reliable, but the network between the client and S3 often stops working for a few minutes at a time. How will Ben's system behave in this situation?

A. CLIENTWRITE will often take a few minutes or more to complete.
B. CLIENTREAD will often take a few minutes or more to complete.
C. CLIENTWRITE will often fail to update all of the servers.
D. CLIENTREAD will often fail, returning "Error".

Ben tests his system by reading and writing the entry for January 1st, 2000, 10 a.m.: he calls:

```
CLIENTWRITE (0, 10, "Staff Meeting")
CLIENTWRITE (0, 10, "Breakfast")
CLIENTREAD (0, 10)
```

Q 41.2 Suppose there are no faults. What string will the CLIENTREAD call return?

Just to be sure, Ben tries a different test, involving moving a meeting from 10 a.m. to 11 a.m., and scheduling breakfast:

```
CLIENTWRITE (0, 10, "Free at 10")
CLIENTWRITE (0, 11, "Free at 11")
CLIENTWRITE (0, 10, "Talk to Frans at 10")
CLIENTWRITE (0, 11, "Talk to Frans at 11")
CLIENTWRITE (0, 10, "Breakfast at 10")
```

Ben starts the test, but trips over the power cord of his client computer while the test is running, causing the client to reboot. The client forgets that it was executing the test after the reboot; it doesn't re-start or continue the test. After the reboot Ben calls CLIENTREAD (0, 10) and CLIENTREAD(0, 11). Other than the mentioned client reboot, the only faults that might occur during the test are lost messages (and thus RPC failures).

Q 41.3 Which of the following results might Ben see?

A. Breakfast at 10, Talk to Frans at 11
B. Talk to Frans at 10, Talk to Frans at 11
C. Breakfast at 10, Free at 11
D. Free at 10, Talk to Frans at 11

Q 41.4 Ben is getting a little paranoid, so he calls ClientRead(0, 10) twice, to see how consistent his database is. Which of the following results might Ben see?

A. Breakfast at 10, Breakfast at 10
B. Talk to Frans at 10, Talk to Frans at 10
C. Free at 10, Breakfast at 10
D. Talk to Frans at 10, Free at 10

Ben feels this behavior is acceptable. Before he starts to use the system, however, his younger brother Mark points out that Ben's system won't be able to complete updates if one of the servers is down. Mark says that if a server is down, a DBWRITE RPC to that server will time out, so CLIENTWRITE will have higher availability if it ignores RPC timer expirations. Mark suggests the following changed CLIENTWRITE:

```
procedure CLIENTWRITE (day, hour, what)
    for i from 1 to 4 do
        S[i].DBWRITE (day, hour, what)
        // Ignore RPC failure
```

Ben adopts this change, and starts using the system to keep his appointments. However, his co-workers start to complain that he is missing meetings. Suspicious of Mark's change, Ben tests the system by manually clearing all database entries on all servers to empty strings, then executing the following code on his client:

```
CLIENTWRITE (0, 10, "X")
v1 ← CLIENTREAD (0, 10)
CLIENTWRITE (0, 10, "Y")
v2 ← CLIENTREAD (0, 10)
CLIENTWRITE (0, 10, "Z")
v3 ← CLIENTREAD (0, 10)
```

Assume that the only possible faults are losses of RPC messages, and that RPC messages are delivered instantly.

Q 41.5 With Mark's change, which of the following sequences of *v1*, *v2*, and *v3* are possible?

A. X, Y, Z
B. X, Z, Y
C. X, X, Z
D. X, Y, X

Q 41.6 In Ben's original system, what sequences of *v1*, *v2*, and *v3* would have been possible?

A. X, Y, Z
B. X, Z, Y
C. X, X, Z
D. X, Y, X

2001–3–14…19

42 Alice's Replicas

(Chapter 10[on-line])

After reading Chapter 10 and the end-to-end argument, Alice explores designing an application for reconciling UNIX file systems. Her program, RECONCILE, takes as input the names of two directory trees and attempts to reconcile the differences between the two trees. The typical scenario for RECONCILE is that one of the directory trees is on a file service. The other one is a replica of that same directory tree, located on Alice's laptop.

When Alice is disconnected from the service, she modifies files on her laptop while her friends may modify files on the service When Alice reconnects to the service, she runs RECONCILE to reconcile the differences between the directory tree on her laptop and the service so that they are identical again. For example, if a file has changed on her laptop, but not on the service, RECONCILE copies the file from the laptop to the service. If the file has changed on both the laptop and server, then RECONCILE requires guidance to resolve conflicting changes.

The RECONCILE program maintains on each host a database named *fsinfo*, which is stored outside the directory tree being reconciled. This database is indexed by path name, and it stores:

```
character pathname[1024] // path name of this file
integer160 hash   // cryptographic hash of the content of the file
```

On disk a UNIX file consists of metadata (the inode) and content (the data blocks). The cryptographic hash is computed using only the file's content. Path names are less than 1024 bytes. For this problem, ignore the details of reconciling directories, and assume that Alice has permission to read and write everything in both directory trees.

The RECONCILE program operates in 4 phases:

- Phase 1: Compute the set of changes on the laptop since the last reconciliation and the set of changes on the server since the last reconciliation.
- Phase 2: The laptop retrieves the set of changes from the service. Using the two change sets, the laptop computes a set of actions that must be taken to reconcile the two directory trees. In this phase, reconcile might decide that some files cannot be reconciled because of conflicting changes.
- Phase 3: The laptop carries out the actions determined in phase 2. The laptop updates the files and directories in its local directory tree, and retrieves files, sends files, and sends instructions to the server to update its directory tree.
- Phase 4: Both the laptop and the service update the *fsinfo* they used to reflect the new content of files that were successfully reconciled on this run.

Assume that RECONCILE runs to completion before starting again. Furthermore, assume that when reconcile runs no concurrent threads are allowed to modify the file system. Also assume that initially the *fsinfo* databases are identical in content and computed correctly, and that after reconciliation they also end up in an identical state.

The first phase of reconcile runs the procedure COMPUTEMODIFICATIONS on both the laptop and the service. On each machine COMPUTEMODIFICATIONS produces two sets: a set of files that changed on that machine and a set of files that were deleted on that machine.

```
set changeset, deleteset;

procedure COMPUTEMODIFICATIONS (path, fsinfo)
    changeset ← NULL
    deleteset ← NULL
    COMPUTECHANGESET (path, fsinfo)
    COMPUTEDELETESET (fsinfo)

procedure COMPUTECHANGESET (path, fsinfo)
    info ← LOOKUP (path, fsinfo)
    if info = NULL then ADD (path, changeset)
    else if ISDIRECTORY (path) then
        for each file in path do COMPUTECHANGESET (path/file, fsinfo)
    else if CSHA (CONTENT (path) ≠ info.hash then ADD (path, changeset)

procedure COMPUTEDELETESET (fsinfo)
    for each entry in fsinfo do
        if not (EXIST (entry.pathname)) then ADD (pathname, deleteset)
```

The COMPUTEMODIFICATIONS procedure takes as arguments the path name of the directory tree to be reconciled and the *fsinfo* database. The procedure COMPUTECHANGESET walks the directory tree and checks every file to see if it was created or changed since the last time RECONCILE ran. CSHA is a cryptographic hash function, which has the property that it is computationally infeasible to find two different inputs *i* and *j* such that

CSHA (*i*) = CSHA (*j*)

The COMPUTEDELETESET procedure checks for each entry in the database whether the corresponding file still exists; if not, it adds it to the set of files that have been deleted since the last run of RECONCILE.

Q 42.1 What files will RECONCILE add to *changeset* or *deleteset*?

A. Files whose content has decayed.
B. Files whose content has been modified.
C. Files that have been created.
D. Files whose inode have been modified.
E. Files that have been deleted.
F. Files that have been deleted but recreated with identical content.
G. Files that have been renamed.

The second phase of reconcile compares the two changesets and the two deletesets to compute a set of actions for reconciling the two directory trees. To avoid confusion we call *changeset* on the laptop *changeLeft*, and *changeset* on the service *changeRight*. Similarly, *deleteset* on the laptop is *deleteLeft* and *deleteset* on the service is *deleteRight*. The second phase consists of running the procedure COMPUTEACTIONS with the 4 sets as input. COMPUTEACTIONS produces 5 sets:

- *additionsLeft*: files that must be copied from server to the laptop
- *additionsRight*: files that must be copied from laptop to the service
- *removeLeft*: file that must be removed from laptop
- *removeRight*: file that must be removed from service
- *conflicts*: files that have conflicting changes

In the following code fragment, the notation *A* - *B* indicates all elements of the set *A* after removing the elements that also occur in the set *B*. With this notation, the 5 sets are computed as follows:

```
conflicts ← NULL;

procedure COMPUTEACTIONS (changeLeft, changeRight, deleteLeft, deleteRight)
    for each file ∈ changeLeft do
        if file ∈ (changeRight ∪ deleteRight) then ADD (file, conflicts)
    for each file ∈ (deleteLeft) do
        if file ∈ (changeRight) then ADD (file, conflicts)
    additionsRight ← changeLeft - conflicts
    additionsLeft ← changeRight - conflicts
    removeRight ← deleteLeft - conflicts
    removeLeft ← deleteRight - conflicts
```

Q 42.2 What files end up in the set *additionsRight*?

A. Files created on the laptop that don't exist on the service.
B. Files that have been removed on the server but not changed on the laptop.
C. Files that have been removed on the laptop but not on the service.
D. Files that have been modified on the laptop but not on the service.
E. Files that have been modified on the laptop and on the service.

Q 42.3 What files end up in the set *conflicts*?

A. Files that have been modified on the laptop and on the service.
B. Files that have been removed on the laptop but that exist unmodified on the service.
C. Files that have been removed on the laptop and on the service.
D. Files that have been modified on the service but not on the laptop.
E. Files that have been created on the laptop and on the service but with different content.
F. Files that have been created on the laptop and on the service with the same content.

Phase 3 of the reconcile executes the actions: deleting files, transferring files, and resolving conflicts. All conflicts are resolved by asking the user.

We focus on transferring files from laptop to service. Alice wants to ensure that transfers of files are atomic. Assume that all file system calls execute atomically. The RECONCILE program transfers files from *additionsRight* by invoking the remote procedure RECEIVE on the service:

```
procedure RECEIVE (data, size, path)
    tname ← UNIQUENAME ()
    fd ← CREATE_FILE (tname)
    if fd ≥ 0 then
        n ← WRITE (fd, data, size)
        CLOSE (fd)
        if n = size then RENAME (tname, path)
        else DELETE (tname)
        return (n = size)              // boolean result tells success or failure
    else return (FALSE)
```

The RECEIVE procedure takes as arguments the new content of the file (*data* and *size*) and the name (*path*) of the file to be updated or created. As its first step, RECEIVE creates a temporary file with a unique name (*tname*) and writes the data into in it. After the write is successful, receive renames the temporary file to its real name (*path*), which incidentally removes any existing old version of *path*; otherwise, it cleans up and deletes the temporary file. Assume that RENAME always executes successfully.

Q 42.4 Where is the commit point in the procedure RECEIVE?

A. right after RENAME completes
B. right after CLOSE completes
C. right after CREATE_FILE completes
D. right after DELETE completes
E. right after WRITE completes
F. none of the above

After the server or laptop fails, it calls a recovery procedure to back out or roll forward a RECEIVE operation that was in progress when the host failed.

Q 42.5 What must this recovery procedure do?

A. Remove any temporary files left by receive.
B. Nothing.
C. Send a message to the sender to restart the file transfer.
D. Rename any temporary files left by receive to their corresponding path name.

Q 42.6 Which advantages does this version of RECONCILE have over the reconciliation procedure described in Chapter 10?

A. This RECONCILE repairs files that decay.
B. This RECONCILE doesn't require changes to the underlying file system implementation.
C. This version of RECONCILE doesn't require a log on the laptop.
D. This RECONCILE propagates changes from the laptop to the service, and vice versa.
E. This RECONCILE will run much faster on big files.

Alice wonders if her code extends to reconciling more than two file systems. Consider 3 hosts (A, B, and C) that all have an identical copy of a file *f*, and the following sequence of events:

- at noon B modifies file *f*
- at 1 pm B reconciles with A
- at 2 pm C modifies *f*
- at 3 pm B reconciles with C
- at 4 pm A modifies *f*
- at 5 pm B reconciles with A

Assume that B has two distinct *fsinfo* databases, one used for reconciling with A and one for reconciling with C.

Q 42.7 Which of the following statements are correct, given this sequence of events and Alice's implementation of RECONCILE?

A. If the conflict at 3 pm is reconciled in favor of B's copy, then RECONCILE will not report a conflict at 5 pm.

B. If the conflict at 3 pm is reconciled in favor of C's copy, then RECONCILE will report a conflict at 5 pm.

C. If the conflict at 3 pm is resolved by a modification to *f* that merges B's and C's versions, then reconcile will report a conflict at 5 pm.

D. If the conflict at 3 pm is resolved by removing *f* from B and C, then RECONCILE will not report a conflict at 5 pm.

2003–3–6…12

43 JailNet*

(Some Chapter 7[on-line], but mostly Chapter 11[on-line])

The Computer Crimes Correction Facility, a federal prison for perpetrators of information-related crimes, has observed curious behavior among their inmates. Prisoners have discovered that they can broadcast arbitrary binary strings to each other by banging cell bars with either the tops or bottoms of their tin cups, making distinct sounds for "0" and "1". Since such sounds made in any cell can typically be heard in every other cell, they have devised an Ethernet-like scheme for communicating varying-length packets among themselves.

The basic communication scheme was devised by Annette Laire, a CCCF lifer convicted of illegal exportation of restricted information when the GIF she e-mailed to her cousin in El Salvador was found to have some bits in common with a competent cryptographic algorithm.

Annette defined the following basic communication primitive:

```
procedure SEND (message, from, to)
    BANG (ALLONES)                          // Start with a byte of eight 1's
    BANG (to)                               // destination inmate number
    BANG (from)                             // source inmate number
    BANG (message)                          // the message data
    BANG (CHECKSUM ({to, from, message}))   // Checksum of whole message
```

where the operation BANG (*data*) is executed by banging one's tin cup to signal the sequence of bits corresponding to the specified null-terminated character string, including the zero byte at its end. The special string ALLONES sent initially has a single byte of (eight) 1 bits (followed by the terminating null byte). The high-order bit of each 8-bit character (in *to*, *from*, *message*, and the result of CHECKSUM) is zero.

Annette specified that the *to* and *from* strings be the unique numbers printed on every inmate's uniform, since all of the nerd-inmates quickly learn the numbers of each of their colleagues. Each inmate listens more or less continuously for packets addressed to him, ignoring those whose *to* field don't match his number or whose checksums are invalid.

Q 43.1 What function(s) are served by sending the initial byte of all 1s?

A. Bit framing.
B. Byte (character) framing.
C. Packet framing.
D. Packet Reassembly.
E. None of the above.

Typical higher-level protocols involve sequences of packets exchanged between inmates, for example:

* Credit for developing this problem set goes to Stephen A. Ward.

Annette ⇒ Ty: SEND ("I thought the lobster bisque was good tonight", ANNETTE, TY);

Ty ⇒ Annette: SEND ("Yes, but the filet was a bit underdone", TY, ANNETTE);

where the symbols ANNETTE and TY are bound to character strings containing the uniform numbers of Annette and Ty, respectively.

Of course, prison guards quickly catch on to the communication scheme, listen in on the conversations, and sometimes even inject messages of their own, typically with false source addresses:

Guard: SEND ("Yeah? Then it's dog food for you tomorrow!", JIMMIETHEGEEK, ANNETTE);

Such experiences motivate Ty Debole, the inmate in charge of cleaning, to add security measures to the JailNet protocols. Ty reads up on public-key cryptography and decides to use it as the basis for JailNet security. He chooses a public-key algorithm and asks each inmate to generate a public/private key pair and tell him the public key.

- KEY represents the inmate's public key. Since Ty runs the CCCF laundry, he prints the numbers on inmate's uniforms. He replaces each inmate's assigned number with a representation of KEY;
- $KEY is the inmate's private key. This key is known only to the inmate whose uniform bears KEY.

Ty assures each inmate that so long as they don't reveal their private $KEY, nobody else—inmates or guards—will be able to determine it. Inmates continue to address each other by the numbers on their uniforms, which now specify their public *Key*s.

Q 43.2 What is an assumption on which Ty bases the security of the secret $KEY?

A. $KEY is theoretically impossible to compute from KEY.
B. $KEY takes an intractably long time to compute from KEY.
C. $KEY takes at least as long to compute from KEY as the generation of the KEY, $KEY pair.
D. There is a reasonably efficient way to compute $KEY, but it's not generally known by guards and inmates.

Ty then teaches inmates the 4 security primitives for messages of up to 1,500 bytes:

- ENCRYPT (*plaintext*, KEY) // returns a message string
- DECRYPT (*ciphertext*, $KEY) // returns a message string
- SIGN (*message*, $KEY) // returns an authentication tag
- VERIFY (*message*, KEY, *signature*) // returns ACCEPT or REJECT

These primitives have the properties described in Chapter 11[on-line].

Ty proposes improving the security of communications by replacing calls to SEND with calls like:

SEND (TYCODE (*message*, *from*, *to*), *from*, *to*);

where TYCODE is defined as

procedure TYCODE (*message*, *from*, *to*)
  return ENCRYPT (*message*, *to*)

Ty and Annette are smugly confident that although the guards might hear their conversation, they won't be able to understand it since the encrypted message appears as gibberish until properly decoded.

The first use of TyCode involves the following message, received by Annette:

SEND (TyCode ("Meet by the wall at ten for the escape", Ty, Annette), Ty, Annette)

Q 43.3 What computation did Annette perform to decode Ty's message? Assume *rmessage* is the message as received, *message* is to be the decoded plaintext, and that *$Annette* and *$Ty* contain the secret keys of Annette and Ty, respectively.

A. *message* ← VERIFY (*rmessage*, *Ty*, *$Annette)*;
B. *message* ← ENCRYPT (*rmessage*, *$Ty*);
C. *message* ← ENCRYPT (*rmessage*, *Ty*);
D. *message* ← DECRYPT (*rmessage*, *$Annette*);
E. *message* ← SIGN (*rmessage*, *$Ty*);
F. *message* ← DECRYPT (*rmessage*, *Annette*);

After receiving the message, Annette sneaks out at ten to meet Ty who she expects will help her climb over the prison wall. Unfortunately Ty never shows up, and Annette gets caught by a giggling guard and is punished severely (early bed, no dessert). When she talks to Ty the next day, she learns that he never sent the message. She concludes that it must have been sent by a guard, but is puzzled since the cryptography is secure.

Q 43.4 What is the most likely explanation?

A. Annette's secret key was compromised during a search of her cell.
B. Some other message Ty sent was garbled in transmission, and accidentally came out "Meet me by the wall at ten for the escape".
C. Annette's secret key was broken by a dictionary attack.
D. Ty's secret key was broken by a dictionary attack.
E. Annette was victimized by a replay attack.

Annette's friend Cert Defy, on hearing this story, comes up with a new cryptographic procedure:

```
procedure CERT (message, A)
    signature ← SIGN (message, A)
    return {message, signature}
```

Unfortunately, Cert is placed in solitary confinement before fully explaining how to use this procedure, though he did state that sending a message with

SEND (CERT (*message*, *A*), *from*, *to*)

can assure the receiver of the integrity of the message body and the authenticity of the sender's identity. So the inmates need some help from you.

Q 43.5 When Ty sends a message to Annette what value should he supply for *A*?

A. ENCRYPT (*Annette*, *$Ty*)
B. *Ty*
C. *$Ty*
D. *Annette*
E. *$Annette*

After Ty determinines the answer to question *43.5*, Annette receives a packet purportedly from Ty. She splits the received packet into *message* and *signature*, and VERIFY (*message*, TY, *signature*) returns ACCEPT.

Q 43.6 Which of the following can Annette conclude about *message*?

A. *message* was initially sent by Ty.
B. The packet was sent by Ty.
C. *message* was initially sent to Annette.
D. Only Annette and Ty know the contents of *message*.
E. If Ty sent *message* to Annette and Annette only, then only they know its contents.
F. *message* was not corrupted in transmission.

Annette, intrigued by Cert's contribution, decides to combine SEND, TYCODE, and CERT to achieve both authentication and confidentiality. She proposes to use NEWSEND, combining both features:

```
procedure NEWSEND (message, A, from, to)
    SEND (TYCODE (CERT (message, A), from, to), from, to)
```

Annette engages in the following conversation:

```
Ty ⇒ Annette: NEWSEND ("Let's escape tonight at ten", TY, ANNETTE);
Ty ⇒ Annette:NEWSEND ("Not tonight, Survivor is on", ANNETTE, TY);
```

The following night, Annette gets the message

```
Ty ⇒ Annette: NEWSEND ("Let's escape tonight at ten", TY, ANNETTE);
```

Once again Annette goes to meet Ty at ten, but Ty never shows up. Eventually Annette gets bored and returns. Ty subsequently disclaims having sent the message. Again, Annette is puzzled by the failure of her allegedly secure system. She suspects that a guard has figured out how to break the system.

Q 43.7 Explain why this happened, yet no guard showed up at the wall to punish Annette for plotting to escape. Suggest a change that Ty could have made that would have eliminated the problem.

Pete O'Fender, who has been in and out of CCCF at regular intervals, wants to extend the security protocols to deal with JailNet key distribution. Whenever he's jailed, Pete is placed directly into solitary confinement where he has no contact with inmates (except via bar banging), and where the TV gets only 3 channels. The problem is complicated by the facts that (a) Everyone (including Pete) forgets Pete's uniform number as soon as he leaves, so when he returns he can't just re-use the old key; (b) Pete may not

even remember the key for Ty or other trusted long-term inmates; (c) Pete is issued an unnumbered uniform while in solitary, and (d) guards often pose as newly-jailed solitary occupants to learn inmate secrets. Pete asks you to devise JailNet key distribution protocols to address these problems.

Q 43.8 Which of the following are true of the *best* protocol you can devise, given the assumptions stated about ENCRYPT, DECRYPT, SIGN, and VERIFY?:

A. Assuming Pete is thrust into Solitary remembering no keys, he can devise a new *Key*/*$Key* pair and broadcast *Key*. Using this *Key*, Ty can be assured that messages he sends to Pete are confidential.

B. Assuming Pete is thrust into Solitary remembering no keys, he can't convince inmates that they aren't communicating with a guard.

C. If Pete remembers Ty's uniform number and trusts Ty, an authenticated broadcast message from Ty could be used to remind Pete of other inmates' uniform numbers without danger of deluding Pete.

D. Even if Pete remembers a trusted inmate's uniform number, any communication *from* Pete can be understood by guards.

E. Even if Pete remembers a trusted inmate's uniform number, any communication *to* Pete might have been forged by guards.

1998–2–7…14

44 PigeonExpress!.com II

(More pigeons, Chapter 11[on-line])

To drive up the stock value of *PigeonExpress!.com* at the planned Initial Public Offering (IPO), Ben needs to make the pigeon net secure. To focus on just security issues, assume for this problem that pigeons never get lost.

First, Ben goes for achieving confidentiality. Ben generates 20 CDs (*KCD*[0] through *KCD*[19]) filled with random numbers, makes two copies of each CD and mails the copies through a secure channel to the sender and receiver. He plans to use the CDs as a one-time pad.

Ben slightly modifies the original BEEP code (which appeared just before question *Q 18.1*) to use the key CDs. The sender's computer runs these two procedures:

```
shared next_sequence initially 0   // a global sequence number, starting at 0.
shared nextKCD initially 0         // index in the array of key CDs.

procedure SECURE_BEEP (destination, n, CD[])// send n CDs to destination
    header h                       // h is an instance of header.
    nextCD ← 0
    h.source ← MY_GPS              // set source to my GPS coordinates
    h.destination ← destination    // set destination
    h.type ← REQUEST               // this is a request message
    while nextCD < n do            // send the CDs
        h.sequence_no ← next_sequence      // set seq number for this CD
        send pigeon with {h,  (CD[nextCD] ⊕ KCD[nextKCD])} // send encrypted
        wait 2,000 seconds

procedure SECURE_PROCESS_ACK (h)       // process acknowledgment
    if h.sequence_no = sequence then   // ack for current oustanding CD?
        next_sequence ← next_sequence + 1
        nextCD ← nextCD + 1            // allow next CD to be sent
        nextKCD ← (nextKCD + 1) modulo 20      // increment with wrap-around
```

Ben also modifies the procedures running on the receiver's computer to match:

```
integer nextkcd initially 0// index in array of KCDs.

procedure SECURE_PROCESS_REQUEST (h, CD)
    PROCESS (CD ⊕ KCD[nextKCD])        // decrypt and process the data on the CD
    nextKCD ← (nextKCD + 1) modulo 20  // increment with wrap-around
    h.destination ← h.source           // send to where the pigeon came from
    h.source ← MY_GPS
    h.sequence_no ← h.sequence_no      // unchanged
    h.type ← ACKNOWLEDGMENT
    send pigeon with h                 // send an acknowledgment back
```

Q 44.1 Do SECURE_BEEP, SECURE_PROCESS_ACK, and SECURE_PROCESS_REQUEST provide confidentiality of the data on the CDs?

A. No, since acknowledgments are not signed;
B. No, since the *KCD*s are reused, SECURE_BEEP is vulnerable to a known plaintext attack;
C. Yes, since one-time pads are unbreakable;
D. No, since one can invert XOR.

To make the system more practical, Ben decides to switch to a short key and to exchange the key over the pigeon net itself instead of using an outside secure channel. Every principal has a key pair for a public-key system. He designs the following key-distribution protocol:

Alice ⇒ Bob: "I propose we use key *k*" (signed with Alice's private key)
Bob ⇒ Alice: "OK, key *k* is fine" (signed with Bob's private key)

The two key-distribution messages are written on a CD and sent with BEEP (not SECURE_BEEP). From key *k* the sender and receiver generate a bit string using a well-known pseudorandom number generator, and employ the bit string in SECURE_BEEP and SECURE_PROCESS_REQUEST to encrypt and decrypt CDs.

Q 44.2 Which statements are true of the above protocol?

A. It is insecure because key *k* travels in the clear and therefore an intruder can find out key *k* and listen in on future SECURE_BEEPs.
B. It is secure because only Bob can verify the message from Alice.
C. It is insecure because Alice's public key is widely known.
D. It is secure, since the messages are signed and key *k* is only used as a seed to a pseudorandom number generator.

1999–2–16/17

45 WebTrust.com (OutOfMoney.com, Part II)

(Chapter 11[on-line])

After their disastrous experience with OutOfMoney.com, the 16-year-old kids regroup. They rethink their business plan and switch from being a service provider to a technology provider. Reading many war stories about security has convinced the kid wizards that there should be a market for a secure client authentication product for Web services. The kids re-incorporate as WebTrust.com. The kids study up on how the Web works. They discover that HTTP 1.0 is a simple protocol whose essence consists of two remote procedure calls:

```
GET (document)              // returns a document
POST (document, form)       // sends a form and returns a document
```

The GET procedure gets the document identified by the Uniform Resource Locator (URL) *document* from a Web service. The POST procedure sends back to the service the entries that the user filled out on a form that was in a previously retrieved document. The POST procedure also gets a document. The browser invokes the POST procedure when the user hits the submit button on the form.

These remote procedure calls are sent over a reliable transport protocol, TCP. A Web browser opens a TCP connection, calls a procedure (GET or POST), and waits for a result from the service. The Web service waits for a TCP connection request, accepts the connection, and waits for a GET or POST call. Once a call arrives, the service executes it, sends the result over the connection, and closes the connection. The browser displays the response and closes the connection on its side. Thus, a new connection must be set up for each request.

Simple URLs are of the form:

```
http://www.WebTrust.com/index.html
```

Q 45.1 "`www.WebTrust.com`" in the above URL is

A. a DNS name
B. a protocol name
C. a path name for a file
D. an Internet address

The objective of WebTrust.com's product is to authenticate users of on-line services. The intended use is for a user to login once per session and to allow only logged-in users access to the rest of the site. The product consists of a collection of Web pages and some server software. The company employs its own product to authenticate customers of the company's Web site.

To allow Internet users to create an account, WebTrust.com has a Web form in which a user types in a user name and two copies of a proposed password. When the user types the password, the browser doesn't echo it, but instead displays a "•" for each typed

character. When the user hits the submit button, the user's browser calls the POST procedure to send the form to the server.

When the server receives a CREATE_ACCOUNT request, it makes sure that the two copies of the password match and makes sure that the proposed user name hasn't already been taken by someone else. If these conditions are true, then it creates an entry in its local password file. If either of the conditions is false, the server returns an error.

The form to create an account is stored in the document "`create.html`" on WebTrust's Web site. Another document on the server contains:

```
<a href="create.html">Create an account</a>
```

Q 45.2 What is the source of the context reference that identifies the context in which the name "`create.html`" will be resolved?

A. The browser derives a default context reference from the URL of the document that contains the relative URL.
B. It is configured in the Web browser when the browser is installed.
C. The server derives it from information it remembers about previous documents it sent to this client.
D. The user types it into the browser.

Q 45.3 Why does the form for creating an account ask a user to type in the password twice?

A. To allow a password not to be echoed on the screen while enabling users to catch typos.
B. To detect transmission errors between the keyboard and the browser.
C. To reduce the probability that a packet with a password has to be retransmitted if the network deletes the packet.
D. To make it harder for users to create fake accounts.

Q 45.4 In this system, to what attacks is creating an account vulnerable? (Assume an active attacker.)

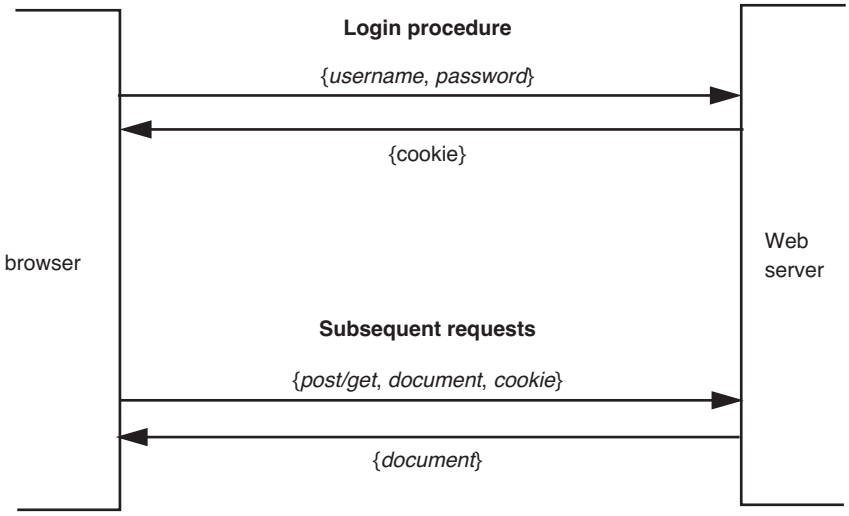
A. An attacker can learn the password for a user by eavesdropping
B. An attacker can modify the password
C. An attacker can overwhelm the service by creating many accounts
D. An attacker can run a service that pretends to be "`www.WebTrust.com`"

To login, the user visits the Web page "`login.html`", which asks the user for a user name and password. When the user hits the submit button, the browser invokes the POST procedure, which sends the user name and password to the service. The service checks the stored password against the password in the login request. If they match, the user is allowed to access the service; otherwise, the service returns an error.

Q 45.5 To what attacks is the login procedure vulnerable? (Assume an active attacker.)

A. An attacker can login by replaying a recorded POST from a legitimate login
B. An attacker can login as any user by replaying a single recorded POST for login
C. An attacker can impersonate `WebTrust.com` to any registered user
D. An attacker can impersonate `WebTrust.com` to an unregistered user

To authenticate subsequent Web requests from a user after logging in, WebTrust.com exploits a Web mechanism called *cookies*. A service can install some state (called a *cookie*) in the Web browser. The service installs this state by including in a response a SET_COOKIE directive containing data to be stored in the cookie. WebTrust.com's use of



cookies is summarized in the figure. The document containing the response to a login request comes with the directive:

POST (*webtrustcookie*)

The browser stores the cookie in memory. (In practice, there may be many cookies, so they are named, but for this problem, assume that there is only one and no name is needed.) On subsequent calls (i.e., GET or POST) to the service that installed the cookie, the browser sends the installed cookie along with the other arguments to GET or POST Thus, once WebTrust.com has set a cookie in a browser, it will see that cookie on every subsequent request from that browser.

The service requires that the browser send the cookie along with all GETs, and also all POSTs except those posting a CREATE or LOGIN form. If the cookie is missing (for example, the browser has lost the cookie because the client computer crashed, or an attacker is leav-

ing the cookie out on purpose), the service will return an error to the browser and ask the user to login again.

An important issue is to determine suitable contents for *webtrustcookie*. WebTrust.com offers a number of alternatives.

The first option is to compute the cookie as follows:

$$cookie \leftarrow \{expiration\_time\}_{key}$$

using a MAC with a shared-secret *key*. The *key* is known only to the service, which remembers it for just 24 hours. All cookies in that period use the same *key*. All cookies expire at 5 a.m., at which time the service changes to a new *key*.

When the server receives the cookie, it checks it for authenticity and expiration using:

```
procedure CHECK (cookie)
    if VERIFY (cookie, key) = ACCEPT then
        if cookie.expiration_time ≤ CURRENT_TIME () then
            return ACCEPT
    return REJECT
```

The procedure VERIFY recomputes and checks the MAC. If the MAC is valid, then the service checks whether *cookie* is still fresh (i.e., if the expiration time is later than the current time). If it is, then CHECK returns ACCEPT; the server can now execute the request. In all other cases, CHECK returns REJECT and the server returns an error to the browser.

Q 45.6 What is the role of the MAC in this protocol?

A. To help detect transmission errors
B. To privately communicate key from the server to the browser
C. To privately communicate expiration-time from the server to the browser.
D. To help detect a forged cookie.

Q 45.7 Which of these attacks does this protocol prevent?

A. Replayed cookies
B. Forged expiration times
C. Forged cookies
D. Dictionary attacks on passwords

Another option supported by webtrust.com is to compute cookie as follows:

$$cookie \leftarrow \{expiration\_time, username\}_{key}$$

The server uses for *username* the name of the user in the login request. The usage of this cookie is similar to before and the checking procedure is unchanged.

Q 45.8 If the service receives a cookie with "Alice" as *username* and CHECK returns ACCEPT, what does the service know? (Assume active attacks.)

A. No one modified the cookie
B. The server accepted a login from "Alice" recently
C. The cookie was generated recently
D. The browser of the user "Alice" sent this cookie recently

Q 45.9 Assume temporarily that all of Alice's Web requests are sent over a single TCP connection that is encrypted and authenticated, and that the setup all has been done appropriately (i.e., only the browser and server know the encryption and authentication keys). After Alice has logged in over this connection, the server has received a cookie with "Alice" as the username over this connection, and has verified it successfully (i.e., VERIFY returns ACCEPT), what does the server know? (Assume active attacks.)

A. No one but the server and the browser of the user "Alice" knows the cookie
B. The server accepted a login from "Alice" recently
C. The cookie was generated recently
D. The browser of the user "Alice" sent this cookie recently

Q 45.10 Is there any additional security risk with storing cookies durably (i.e., the browser stores them in a file on the local operating system) instead of in the run-time memory of the browser? (Assume the operating system is a multi-user operating system such as Linux or Windows, including a virtual memory system.)

A. Yes, because the file with cookies might be accessible to other users.
B. Yes, because the next user to login to the client machine might have access to the file with cookies.
C. Yes, because it expands the trusted computing base to include the local operating system
D. Yes, because it expands the trusted computing base to include the hard disk

Q 45.11 For what applications is WebTrust's product (without the encrypting and authenticating TCP connection) appropriate (i.e., usable without grave risk)?

A. For protecting access to bank accounts of an electronic bank
B. For restricting access to electronic news articles to clients that have subscription service
C. For protecting access to student data on a university's on-line student services
D. For electronic shopping, say, at amazon.com
E. None of the above

Mark Bitdiddle—Ben's 16-year kid brother—proposes to change the protocol slightly. Instead of computing cookie as:

$$cookie \leftarrow \{expiration\_time, username\}_{key}$$

Mark suggests that the code be simplified to:

$$cookie \leftarrow \{\{expiration\_time\}_{key}, username\}$$

He also suggests the corresponding change for the procedure VERIFY. The protocol, as originally, runs over an ordinary unencrypted and unauthenticated TCP connection.

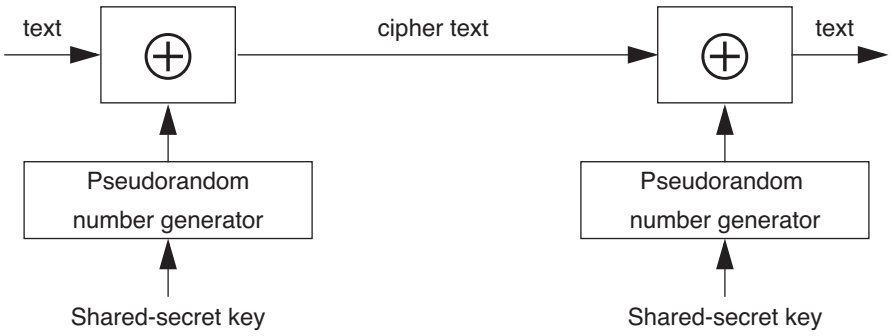
Q 45.12 Describe one attack that this change opens up and illustrate the attack by describing a scenario (e.g., "Lucifer can now … by …").

2001–2–6…17

46 More ByteStream Products

(Chapter 11[on-line])

Observing recent interest in security in the popular press, ByteStream Inc. decides to extend the function of its products to obtain confidentiality by encryption. ByteStream decides to use the simple shared-secret system shown below:



ByteStream uses the exclusive-OR (XOR, shown as ⊕) function. The pseudorandom number generator (PRNG) generates a stream of hard-to-predict bits, using the shared-secret key as a seed. Whenever it is seeded with the same key, it will generate the same bit stream. Messages are encrypted by computing the XOR of the message and the bit stream produced by the generator. The resulting ciphertext is decrypted by computing the XOR of the ciphertext and the bit stream produced by the PRNG, seeded with some key. The code for the PRNG is publicly available.

To check the implementation, ByteStream Inc. hires a tiger team that include Eve S. Dropper and Lucy Fer. The tiger team verifies that the code for computing the XOR is bug-free and the PRNG does not contain cryptographic weaknesses. The tiger team subsequently studies the following scenario. Alice shares a 200-bit key *K* with Bob. Alice encrypts a message with *K and* sends the resulting ciphertext to Bob. Bob decrypts this message with *K*. The result after decryption is Alice's message. Assume that every message is equally likely (i.e., Alice's message contains no redundancy whatever).

Q 46.1 Given that Eve sees only the cipher text, can she cryptanalyze Alice's message?

- **A.** No, since only Alice and Bob know the key, and the PRNG generates a 0 or 1 with equal probability, Eve has no way of telling what the content of Alice's message is.
- **B.** Yes, since with a supercomputer Eve could try out all possible combinations of 0s and 1s for *K* and check whether they match the cipher text.
- **C.** No, since it is hard to compute the XOR of two bit streams.
- **D.** Yes, since XOR is a simple function, Eve can just compute the inverse of XOR.

Q 46.2 Alice and Bob switch to a new shared key. Lucy mounts an active attack by tricking Alice into sending a message that begins with 500 one's, followed by Alice's original message. Given the ciphertext can Lucy cryptanalyze Alice's message?

A. Yes, since the key is smaller than 500 bits.
B. Maybe, but with probability so low that it is negligible.
C. No, since only Alice and Bob know the key and the PRNG generates a 0 or 1 with equal probability, Lucy cannot extract Alice's message.
D. No, since it is hard to compute the XOR of two bits.

ByteStream is interested in a product that supports two-way communication. ByteStream implements two-way communication by having one stream for requests and another stream for replies. ByteStream seeds both streams with the same key. Since ByteStream worries that using the same key in both directions might be a weakness, it asks the tiger team to check the implementation.

The tiger team studies the following scenario. Alice seeds the PRNG for the request stream with K and sends Bob a message. Upon receiving Alice's message, Bob seeds the PRNG for the reply stream with K, and sends a response to Alice. Again, assume that every request and response is equally likely.

Q 46.3 What can Eve deduce about the content of the messages?

A. Nothing.
B. The content of the request, but not the reply.
C. The XOR of the request and the reply.
D. The content of both the request and the reply.

1997–2–3a…c

47 Stamp Out Spam*

(Chapter 11)

2005–3–6

Spam, defined as unsolicited messages sent in large quantities, now forms the majority of all e-mail and short message service (SMS) traffic worldwide. Studies in 2005 estimated that about 100 billion (100×10^9) e-mails and SMS messages were sent per day, two-thirds of which were spam. Alyssa P. Hacker realizes that spam is a problem because it costs virtually nothing to send e-mail, which makes it attractive for a spammer to send a large number of messages every day.

Alyssa starts designing a spam control system called **SOS**, which uses the following approach:

A. **Allocation**. Every sender is given some number of **stamps** in exchange for payment. A newly issued stamp is *fresh*, while one that has been used can be *cancelled* to ensure that it is used only once.
B. **Sending**. The sender (an outgoing mail server) attaches a fresh stamp to each e-mail message.
C. **Receiving**. The receiver (an incoming mail server) tests the incoming stamp for freshness by contacting a **quota enforcer** that runs on a trusted server using a TEST_AND_CANCEL remote procedure call (RPC), which is described below. If the stamp is fresh, then the receiver delivers the message to the human user. If the stamp is found to be cancelled, then the receiver discards the message as spam.
D. **Quota enforcement**. The quota enforcer implements the TEST_AND_CANCEL RPC interface for receivers to use. If the stamp was not already cancelled, the quota enforcer cancels it in this procedure by storing cancellations in a database.

Alyssa's hope is that allocating reasonable quotas to everyone and then enforcing those quotas would cripple spammers (because it would cost them a lot), while leaving legitimate users largely unaffected (because it would cost them little).

Like postage stamps, SOS's stamps need to be unforgeable, for which cryptography can help. SOS relies on a central trusted stamp authority, SA, with a well-known public key, SA_{pub}, and a corresponding private key, SA_{priv}. Each sender S generates a public/private key pair, (S_{pub}, S_{priv}), and presents S_{pub} to SA along with some payment. In return, the stamp authority SA gives sender S a certificate (C_S) and allocates it a stamp quota.

$$C_S = \{S_{pub}, \mathit{expiration\_time}, \mathit{daily\_quota}\}_{SA_{priv}}$$

The notation $\{msg\}_k$ stands for the marshaling of *msg* and the signature (signed with key *k*) of *msg* into a buffer. We assume that signing the same message with the same key always generates the same bit string. In the certificate, *expiration_time* is set to a time one

* Credit for developing this problem set goes to Hari Balakrishnan.

year from the time that SA issued the certificate, and *daily_quota* is a positive integer that specifies the maximum number of messages per day that S can send.

S is allowed to make up to *daily_quota* stamps, each with a unique integer *id* between 1 and *daily_quota*, and the current *date*. To send a message, S constructs and attaches a stamp with the following format:

$$stamp = \{C_S, \{id, date\}_{S_{priv}}\}$$

When a receiver gets a stamp, it first checks that the stamp is **valid** by running CHECK_STAMP_VALIDITY (*stamp*). This procedure verifies that C_S is a properly signed, unexpired certificate, and that the contents of the stamp have not been altered. It also checks that the *id* is in the range specified in C_S, and that the *date* is either yesterday's date or today's date (thus a stamp has a two-day validity period).

If any check fails the receiver assumes that the message is spam and discards it. If all the checks pass, then the stamp is considered *valid*. The receiver calls TEST_AND_CANCEL on valid stamps.

Unless otherwise mentioned, assume that:

A. No entity's private key is compromised.
B. All of the cryptographic algorithms are computationally secure.
C. SA is trusted by all participants and no aspect of its operation is compromised.
D. Senders may be malicious. A malicious sender will attempt to exceed his quota; for example, he may attempt to send many messages with the same stamp, or steal another sender's unused stamps.
E. Receivers may be malicious; for example, a malicious receiver may attempt to cancel stamps belonging to other senders that it has not seen.
F. Most receivers cancel stamps that they have seen, especially those attached to spam messages.
G. Each message has exactly one recipient (don't worry about messages sent to mailing lists).
H. Spammers and other unsavory parties may mount denial-of-service and other resource exhaustion attacks on the quota enforcer, which SOS should protect against.

Alyssa implements TEST_AND_CANCEL as shown in Figure PS.6. Because spammers have an incentive to reuse stamps, she wants to keep track of the total number of TEST_AND_CANCEL requests done on each stamp. *num_uses* is a hash table keyed by *stamp* that keeps track of this number. The hash table supports two interfaces:

A. PUT (*table*, *key*, *value*) inserts the (*key*, *value*) pair into table.
B. GET (*table*, *key*) returns the value associated with key in table, if one was previously PUT, and 0 otherwise. A value of 0 is never PUT.

Q 47.1 Louis Reasoner looks at the TEST_AND_CANCEL procedure and declares, "Alyssa, the client would already have checked that the stamp is valid, so you don't need to call CHECK_STAMP_VALIDITY again." Alyssa thinks about it, and decides to keep the check. Why?

```
1     procedure TEST_AND_CANCEL (stamp, client)
2           // assume that client is not a spoofed network address
3     if CHECK_STAMP_VALIDITY (stamp) ≠ VALID then return
4     u ← GET (num_uses, stamp)
5     if u > 0 then status ← CANCELLED
6     else status ← FRESH
7     u ← u + 1
8     PUT(num_uses, stamp, u)
9     SEND(client, status);              // assume reliable data delivery
```

FIGURE PS.6

Alyssa's TEST_AND_CANCEL procedure.

Q 47.2 Suppose that a recipient R gets an e-mail message that includes a valid stamp belonging to S. Then, which of the following assertions is true?

- **A.** R can be certain that the e-mail message came from S.
- **B.** R can be certain of both the data integrity and the origin integrity of the certificate in the stamp.
- **C.** R may be able to use the information in this stamp to cancel another stamp belonging to S with a different *id*.
- **D.** If an attacker breaks into a computer that has fresh stamps on it, he may be able to use those stamps for his own messages, even though the stamps were signed by another entity.
- **E.** S can tell whether or not R received an e-mail message by calling TEST_AND_CANCEL to see if the stamp attached to that message has been cancelled at the quota enforcer.
- **F.** If S has encrypted the e-mail message with R_{pub}, then no entity other than S or R could have read the contents of the message without S or R knowing.

The United Nations Privacy Organization looks at Alyssa's proposal and throws a fit, arguing that SOS compromises the privacy of sender-receiver e-mail communication because the stamp authority, which also runs the quota enforcer, may be able to guess that a given sender communicated with a given receiver. Alyssa decides that the SOS protocol should be amended to meet two goals:

- G1. It should be computationally infeasible for the stamp authority (quota enforcer) to associate cancelled stamps with a sender-receiver pair.
- G2. It should still be possible for a receiver to call TEST_AND_CANCEL and correctly determine a stamp's freshness.

Alyssa considers several alternatives to achieve this task. Louis proposes using an encryption method he calls DETERMINISTIC_ENCRYPT (*msg*, *k*), which always produces the same output string for the same (*msg*, *k*) input. A second scheme involves an off-the-shelf ENCRYPT (*msg*, *k*) that, because it adds a timestamp to the plaintext message, always produces different output for the same (*msg*, *k*) input. A third alternative uses HASH (*msg*),

a cryptographically secure one-way hash function of *msg*. Alyssa removes line 3 of TEST_AND_CANCEL so that it no longer calls CHECK_STAMP_VALIDITY and she checks to make sure that TEST_AND_CANCEL will accept any bit-string as its first argument. S_{pub} is S's public key (from the certificate in the stamp) and R_{pub} is R's public key.

Q 47.3 Which of these methods achieves goals G1? Which achieves G2?

A. The receiving client R extracts $u = \{C_S, \{id, date\}_{S_{priv}}\}$ from the stamp, and computes e_1 = DETERMINISTIC_ENCRYPT (u, S_{pub}). It then calls TEST_AND_CANCEL (e_1, R).

B. The receiving client R extracts $u = \{C_S, \{id, date\}_{S_{priv}}\}$ from the stamp, and computes e_2 = ENCRYPT (u, R_{pub}). It then calls TEST_AND_CANCEL (e_2, R).

C. The receiving client R extracts $u = \{C_S, \{id, date\}_{S_{priv}}\}$ from the stamp, and computes h = HASH (u). It then calls TEST_AND_CANCEL (h, R).

Alyssa realizes that if SOS is to be widely used she will need several computers to run the quota enforcer to handle the daily TEST_AND_CANCEL load. Alyssa finds that storing the *num_uses* hash table used by TEST_AND_CANCEL on disk gives poor performance because the accesses to the hash table are random. When Alyssa stores this hash table in RAM, she finds that one computer can handle 50,000 TEST_AND_CANCEL RPCs per second on a realistic input workload, including the work required to find the machine storing the key (compared to ≈100 RPCs per second for a disk-based hash table implementation). The network connecting clients to the quota enforcer servers has extra capacity and is thus not the bottleneck.

The space required to store stamps in Alyssa's current design is rather large. She decides to save space by storing HASH (*stamp*) rather than the much larger *stamp*. With this optimization, storing each cancellation in the *num_uses* hash table consumes 20 bytes of space. Assume that *num_uses* stores only stamps that are from today or yesterday. Alyssa purchases computers that each have one gigabyte of RAM available for stamp storage.

Q 47.4 Alyssa finds that the peak TEST_AND_CANCEL request rate is 10 times the average. Estimate the number of servers that Alyssa needs for SOS in order to handle 100 billion TEST_AND_CANCEL operations per day. (Use the approximation that there are 10^5 seconds in one day.) Be sure to consider all of the potential bottlenecks.

Alyssa builds a prototype SOS system with multiple servers. She runs multiple TEST_AND_CANCEL threads on each server. Alyssa wants each thread to be recoverable and for all cancelled stamps to be durable for at least two days. She also wants the different concurrent threads to be isolated from one another.

Alyssa decides that a good way to implement the quota enforcer is to use transactions. She inserts a call to BEGIN_TRANSACTION at the beginning of TEST_AND_CANCEL and a call to COMMIT just before the call to SEND. She implements a disk-based undo/redo log of updates to the *num_uses* hash table using the write-ahead log protocol (each disk sector write is recoverable). She uses locks for isolation.

Because all stamp cancellations are stored in RAM, Alyssa finds that a server crash loses the entire in-RAM hash table of previously cancelled stamps. A thread could also

ABORT at any time before it COMMITs (for example, the operating system could decide to ABORT a thread that is running too long).

Q 47.5 Which of these statements about SOS's recoverability and durability is true?

- **A.** When a thread ABORTs, under some circumstances, the ABORT procedure must undo some operations from the log.
- **B.** When a thread ABORTs, under some circumstances, the ABORT procedure must redo some operations from the log.
- **C.** The failure recovery process, under some circumstances, must undo some operations from the log.
- **D.** The failure recovery process, under some circumstances, must redo some operations from the log.
- **E.** When the failure recovery process is recovering from the log after a failure, there is no need for it to ACQUIRE any locks as long as no new threads run until recovery completes.

Q 47.6 Recall that an important goal in SOS is to detect if any stamp is used more than once. Louis Reasoner asserts, "Alyssa, any reuse of stamps will be caught even if you *don't* worry about before-or-after atomicity between TEST_AND_CANCEL threads." Give an example to show why before-or-after atomicity is necessary.

Satisfied that her prototype works and that it can handle global message volumes, Alyssa turns to the problem of pricing stamps. Her goal is ''modest'': to reduce spam by a factor of 10. She realizes that her answer depends on a number of assumptions and is only a first-cut approximation.

Q 47.7 Alyssa reads various surveys and concludes that spammers would be willing to spend at most US $11 million per day on sending spam. She also concludes that 66% (two-thirds) of the 100 billion daily messages sent today are spam. Under these assumptions, what should the price of each stamp be in order to reduce the number of spam messages by at least a factor of 10?

48 Confidential Bitdiddler*

(Chapter 11)

2007–3–16

Ben uses the original Bitdiddler with synchronous writes from Problem set 5. Ben stores many files in the file system on his handheld computer, and runs out of disk space quickly. He looks at the blocks on the disk and discovers that many blocks have the same content. To reduce space consumption he augments the file system implementation as follows:

A. The file system keeps a table in memory that records for each allocated block a 32-bit non-cryptographic hash of that block. (When the file system starts, it computes this table from the on-disk state.) Ben talks to a hashing expert, who tells Ben to use the *b*-bit (here *b*=32) non-cryptographic hash function

$$H(block) = block \textbf{ modulo } P$$

where *P* is a large *b*-bit prime number that yields a uniform distribution of hash values throughout the interval [0...2*b*-1].

B. When the file system writes a block *b* for a file, it checks if the table contains a block number *d* whose block content on disk has the same hash value as the hash value for block *b*. If so, the file system frees *b* and inserts *d* into the file's inode. If there is no block *d*, the file system writes *b* to the disk, and puts *b*'s block number and its hash in the table.

To keep things simple, let's ignore what happens when a user unlinks a file.

Q 48.1 Occasionally, Ben finds that his system has files with incorrect contents. He suspects hash collisions are to blame. These might be caused by:

A. Accidental collisions: different data blocks hash to the same 32-bit value.
B. Engineered collisions: adversaries can fabricate blocks that hash to the same 32 bit value.
C. A block whose hash is the same as its block number.

Q 48.2 For each of the following proposed fixes, list which of the problem causes listed in Question 48.1 (A, B, or C) it is likely to fix:

A. Use a *b*=160-bit non-cryptographic hash in step A of the algorithm.
B. Use a 160-bit cryptographic hash such as SHA-1 in step A of the algorithm.
C. Modify step B of the algorithm so that when a matching hash is found, it compares the contents of the stored block to the data block and treats the blocks as different unless their contents match.

---

* Credit for developing this problem set goes to Sam Madden.

Ben decides he wants to encrypt the contents of the files on disk so that if someone steals his handheld computer, they cannot read the files stored on it. Ben considers two encryption plans:

- **User-key encryption**: One plan is to give each user a different key and use a secure block encryption scheme with no cryptographic vulnerabilities to encrypt the user's files. Ben implements this by storing a table of (user name, key) pairs, which the system stores securely on disk.
- **Convergent encryption**: One problem with user-key encryption is that it doesn't provide the space saving if blocks in different files of different users have the same content. To address this problem, Ben proposes to use **convergent encryption** (also called "content hash keying"), which encrypts a block using a cryptographic hash of the content of that block as a shared-secret key (that is, ENCRYPT (*block*, HASH (*block*)). Ben reasons that since the output of the cryptographic hash is pseudorandom, this is just as good as choosing a fresh random key. Ben implements this scheme by modifying the file system to use the table of hash values as before, but now the file system writes encrypted blocks to the disk instead of plaintext ones. This way blocks are encrypted but, because duplicate blocks have the same hash and thus encrypt to the same ciphertext, Ben still gets the space savings for blocks with the same content. The file system maintains a secure table of block hash values so that it can decrypt blocks when an authorized user requests a read operation.

Q 48.3 Which of the following statements are true of convergent encryption?

A. If Alyssa can guess the contents of a block (by enumerating all possibilities, or by guessing based on the file metadata, etc), it is easy for her to verify whether her guess of a block's data is correct.
B. If Alyssa can discover the 32-bit block numbers referenced by inodes in the file system, she can learn something about the contents of Ben's files.
C. The file system can detect when an adversary changes the content of a block on disk.

Q 48.4 Which of the following statements are true of user-key encryption?

A. If Alyssa can guess the contents of a block but doesn't know Ben's key, it is easy for her to verify whether her guess of a block's data is correct.
B. If Alyssa can discover the 32-bit block numbers referenced by inodes in the file system, she can learn something about the contents of Ben's files.
C. The file system can detect when an adversary changes the content of a block on disk.

49 Beyond Stack Smashing*

(Chapter 11)

2008–3-8

You are hired by a well-known OS vendor to help them defend their products against buffer overrun attacks of the kind described in Sidebar 11.4. Their team presents several proposed strategies to foil buffer exploits:

- **Random stack**: Place the stack in an area of memory randomly chosen for each new process, rather than at the same address for every process.
- **Non-executable stack**: Set the permissions on the virtual memory containing the stack to allow reading and writing but not execution as a program. Set the permissions on the memory containing the program instructions to read and execute but not write.
- **Bounds checking**: Use a language such as Java or Scheme that checks that all array/buffer indices are valid.

You are aware of several buffer overrun attacks, including the following:

- **Simple buffer overrun:** The victim program has an array on the stack as follows:

```
procedure VICTIM (data, length)
   integer buffer[100]
   COPY (buffer, data, length)// overruns array buffer if length > 100
   // ....
```

  The attacker supplies a *length* > 100 together with an array *data* that includes some new instructions and places the address of the first instruction in the position where the procedure return is stored. When VICTIM reaches its **return**, it returns to the attacker's code in the stack rather than the program that originally called VICTIM.

- **Trampoline**: The victim program has an array on the stack as in the code fragment above, but the attacker cannot predict its address, so replacing the procedure return address with the address of the attacker's code won't work. However, the attacker knows that subroutine VICTIM () leaves an address in some register (say R5) that points to a known, fixed offset within the array. The other thing that is needed is an instruction anywhere in memory at a known address *x* that jumps to wherever R5 is pointing. The attacker overruns the array with his new instructions and overwrites the procedure return address with the address *x*. When VICTIM reaches its **return**, it returns to address x, which jumps to the address in R5, which transfers control to the attacker's code.

---

* Credit for developing this problem set goes to Lewis D. Girod. This problem set was inspired by a paper by Jonathan Pincus and Brandon Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security & Privacy 2*, 4 (July/August 2004) pages 20–27. Further details and explanations can be found in that paper.

- **Arc injection** (return-to-library): Taking advantage again of knowledge that VICTIM leaves the address of a fixed offset within the array in register R5, the attacker provides some carefully selected data at that offset and also overruns the buffer with a new procedure return address. The new procedure return address is chosen to be in some system or library program known to reside elsewhere in the current address space, preferably to a place within that library program after it has checked the validity of its parameters, and is about to do something using the contents of register R5 as the address of one of its parameters. A particularly good library program to jump into is one that calls a procedure whose string name is supplied as an argument. The attacker's carefully selected data is chosen to be the string name of an existing program that the attacker would like to execute.

In the following questions, an attack is considered *prevented* if the attacker can no longer execute the intended malicious code, even if an overflow can still overwrite data or crash or disrupt the program.

Q 49.1 Which of the following attack methods are prevented by the use of the *random* stack technique?

A. Simple buffer overrun
B. Trampoline
C. Arc injection (return-to-library)

Q 49.2 Which of the following attack methods are prevented by the use of the *non-executable stack* technique?

A. Simple buffer overrun
B. Trampoline
C. Arc injection (return-to-library)

Q 49.3 Which of the following attack methods are prevented by the use of the *bounds* checking technique?

A. Simple buffer overrun
B. Trampoline
C. Arc injection (return-to-library)