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 *11.9*)
- *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 *8.8*)
- *restartable atomic region* (Problem set *9*)
- *self-describing storage* (Exercise *6.8*)
- *serializability* (Problem set *36*)
- *timed capability* (Exercise *11.7*)

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

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

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 1-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.

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

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

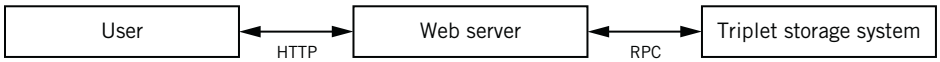
a Triplet Storage System (TSS), which stores and retrieves arbitrary triples of the form {*subject, relationship, object*} according to the following specification:

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 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.

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 e.g.,

```
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 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.

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.

*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 or ERROR
        rhandle ← RPC (host, "OPEN", filename, mode)
    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 1

Pseudocode for FSD client library, Version 1

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 reexecuting 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

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

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 $[0, \ldots, \text{NSPOTS} - 1]$ 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.

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 U.S. 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*

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

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.

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 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

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

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.

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 multithreaded: 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 service each start with one thread. There are no user programs other than the client and service 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.

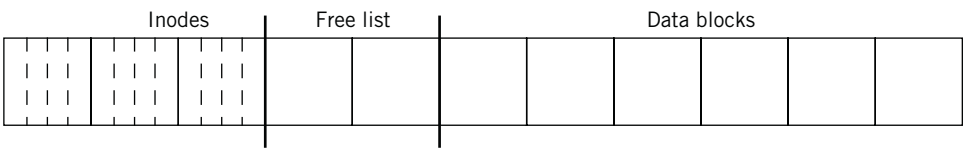
2004-1-4...10

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:



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

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 below. 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.

```
procedure CREATE (filename)
    scan all non-free inodes to ensure filename is not a duplicate (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)
```

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. 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 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

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

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 $71FFF2_{hex}$, 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:

```
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 454.
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.

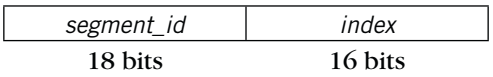
The Amazing Computer Company has asked every designer in the company to come up with a design to enforce modularity further. In a dusty book about the PDP-11/70, Ben finds a description of a hardware gadget that sits between the processor and the physical memory, translating virtual addresses to physical 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 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
```

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

```
ADVANCE (inCnt)
```

Network manager using events

```
inCnt ← inCnt + 1       //D
NOTIFYEVENT (input)     //E
```

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_infos*. 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).

```
structure port_info
    semaphore instance n initially 0
    message instance buffer [NMSG]                  // an array of 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.in)

procedure RECEIVE_MESSAGE (port_info reference port)
1   ...                                             // another line of code will go here
    DOWN (port.in)
    m ← port.buffer[port.in modulo NMSG]
    port.out ← port.out + 1
    return m
```

The kernel schedules threads preemptively.

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 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

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

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

14 Toastac-25*

(Chapters 5 and 7 [on-line])

Louis P. Hacker bought a used Therac-25 (the medical irradiation machine that was involved in several fatal accidents—see 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.

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

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.

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
```

*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.

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 partly in hardware and partly 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 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 now is 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 × 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 ()
```

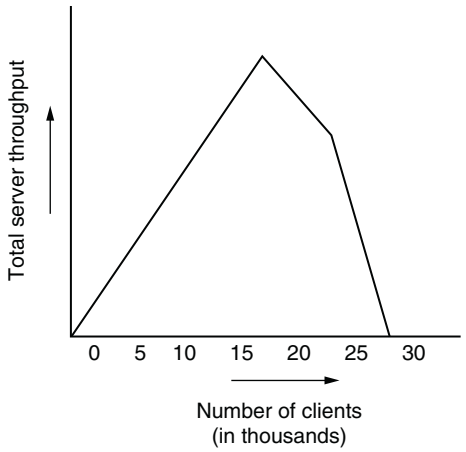
Q 16.8 Which of the following could happen under high load?

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 below. 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