Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 41 November 10, 1993

Problem Set #9

Due date: November 17, 1993

This problem set takes the form of a take-home quiz. It is partly intended for you to get a feel for the styles of questions you can expect on the final exam, since there are no final exams from previous years for you to look at. It is also intended to help us calibrate the difficulty of and time required for these kinds of questions. For the purpose of assigning final grades in the class, it will be treated just like any other problem set.

The quiz is designed to be about the same length as the final exam, namely three hours. Treat it like a quiz; *i.e.*, you should spend some time studying before looking at it, rather than directing your studying based solely on what is on the quiz. (Since the final is not a take-home, you won't have any choice about this on the final.) You are not required to limit your time on the quiz to three hours, but we strongly encourage you to do so. Note that the quiz only counts as much as an ordinary problem set for your final grade, so there is little point in spending a huge amount of time on it. Also, since we have not given an exam in this class before, our estimate of how long this quiz should take could be way off.

You should keep track of how much time you spend on each part of each question (including time spent reading the question, thinking about it, and writing up your answer) and include that information in the writeup you hand in. This will help us calibrate questions better on the final. Our expectation is that no part of any question should take longer than ten or fifteen minutes; if you find yourself spending significantly more time than that on any part, ask us to clarify the question.

The quiz comes in two parts. The quiz questions themselves are in the second part. The first part is some material that you should read and understand before looking at or attempting to answer the questions in the second part. The three-hour time limit applies only to the second part; reading the first part should be counted as time spent studying for the quiz. We will probably take a similar approach on the final exam of providing you with some material to read beforehand, and then asking questions about that material on the exam.

There are a total of 13 questions on the quiz, each with several parts. Your answer to each should be fairly brief, on the order of a few sentences or a few lines of code. The quiz is open book; feel free to refer to any reference material you desire while working on the questions.

You should answer each problem on a separate page, and your name should be on every page.

Part I:
Preliminary Material

Before doing the quiz, you should read and understand the material below describing another way of implementing a mutex. This material is used in Problems 9–13.

Consider the following simplified specification for `Mutex`, which simply records whether or not the mutex is held, and does not keep track of which thread holds the mutex.

```
MODULE SimpleMutex =

TYPE M = ENUM[available, held]

VAR m := available

APROC Acquire () = << m = available => m := held >>

APROC Release () = << m := available >>

END SimpleMutex
```

Now consider the following implementation of this specification. Instead of "test-and-set", it uses two other "read-modify-write" atomic operations: "fetch-and-store" and "compare-and-swap". The basic strategy is to keep a linked list of the threads waiting for the mutex; the current holder is at the head of the list, and a pointer is kept to the tail of the list so that new threads can be appended to it. When a thread releases the mutex, it sets a flag in the list cell for the next thread in the list to tell it that it can proceed. (Note: this implementation was invented by John Mellor-Crummey and Michael Scott; it is described in University of Rochester Department of Computer Science Technical Report 342, April 1990, and also in Section 2.1 of the paper "Synchronization without contention" by Mellor-Crummey and Scott on pages 269-278 of the proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 1991. The ASPLOS proceedings are available as a special issue of SIGPLAN Notices, of SIGOPS Operating Systems Review, and of SIGARCH Computer Architecture News.)

```
MODULE MutexImpl =

TYPE
   T = Thread
   TN = T + Null
   C = [b: Bool,            % Cell; has boolean flag and
        tn]                 % name of next thread in a list.
   S = T -> C               % State: a cell (or nothing) per thread.
```

```
VAR
   s := S{*->}              % Initially no cell for each thread.
   tl: TN := nil            % Tail of list; initially nil.

% A thread t waiting for the mutex will have s(t).b = false;
% the holder (at the head of the list) will have s(t).b = true.

PROC Acquire () =
   VAR tn0, t := SELF |
   << s(t) := C{b: false, tn: nil} >>;
   tn0 := FetchAndStore(t);       % Make t the tail; get the old tail.
   tn0 = nil =>                   % The list was empty before,
       << s(t).b := true >>; RET  %   so make t the holder and return.
   [*] << s(tn0).tn := t >>;      % Link t to old tail.
       DO ~s(t).b => SKIP OD      % Spin until t's b is set.

PROC Release () =
   VAR t := SELF,
       emptyList: Bool |    % Will be true iff no threads are waiting.
    s(t).b =>   % Error to release unless t is the current holder.
      emptyList := CompareAndSwap(t, nil);
      emptyList => RET
      [*] DO s(t).tn = nil =>     % Wait for next thread to link to t.
                  SKIP OD;
          << s(s(t).tn).b := true >>   % Signal next thread in line.

% Internal procedures

% Atomically fetches (and returns) old value for tl and
% stores a new value.
APROC FetchAndStore(t) -> (T + Null) =
   << VAR tn | tn := tl; tl := t; RET tn >>

% Atomically compares tl's value to t; sets tl to tn
% if tl equals t, and otherwise makes no change.  Returns
% true iff tl equals t.
APROC CompareAndSwap(t, tn) -> Bool =
   << tl = t => tl := tn; RET true
      [*] RET false >>

END MutexImpl
```

Part II:
Quiz Questions

We encourage you to finish studying before beginning work on the quiz questions, and to limit your time to three hours.

Reminder: please keep track of how much time you spend on each part of each question (including reading the question, thinking about it, and writing up your answer), and include that information in the writeup you hand in. You will lose points if you fail to do this.

Problem 1 (20 points):

Consider the following trivial module that describes a counter based on the Int type, restricted to the operations `Inc` (increment), `Dec` (decrement) and `IsZero` (test for zero).

```
MODULE SimpleInt

   VAR i: Int := 0

   APROC Inc() = << i := i+1 >>

   APROC Dec() RAISES {underflow} =
      << IsZero() => RAISE underflow [*] i := i-1 >>

   FUNC IsZero()->Bool = RET i=0

END SimpleInt
```

The `SimpleInt` counter can count arbitrarily high and still perform correctly; on real computers, one generally has a choice between putting an arbitrary bound on the size of integers and using some sort of BigNum representation; *i.e.*, an encoding of integers as sequences of bytes or words. Consider, then, the following implementation of `SimpleInt`:

```
MODULE BigInt

   TYPE B = Int
        I = SEQ B

   VAR MaxB: Int := 255  % Constant

   VAR i := {0}

   APROC Inc() =
      << i(0) := i(0) + 1;                                %add 1
         BEGIN VAR j := 0 |                               %carry
                 DO i(j) > MaxB => i(j) := 0;
                                   j := j+1;
                                   i(j) := i(j) + 1 OD END;
         i.last > 0 => i := i ++ 0 >>                  %add leading 0

   APROC Dec() =
      << BEGIN IsZero() => RAISE underflow
               [*] i(0) := i(0) - 1 END;                  %subtract 1
         VAR j := 0 |                                     %carry
```

```
            DO i(j) < 0 => i(j) := MaxB;
                           j := j+1;
                           i(j) := i(j) - 1 OD >>

    FUNC IsZero()->Bool = (ALL j | i!j ==> i(j) = 0)

  END BigInt
```

(A) Give the abstraction function and invariant required to prove that `BigInt` implements `SimpleInt`.

(B) Is `SimpleInt` an implementation of `BigInt`? Explain briefly why or why not.

Now consider the following stupid specification:

```
MODULE UselessInt

   VAR null := nil

   APROC Int() = RET
   APROC Dec() RAISES {underflow} = HAVOC
   APROC IsZero()->Bool = HAVOC

END
```

(C) Is `UselessInt` an implementation of `SimpleInt`? Explain briefly why or why not.

(D) Is `BigInt` an implementation of `UselessInt`? Explain briefly why or why not.

Problem 2 (12 points):

You have a 1.2 GByte disk with 3 MB/sec transfer rate and 10 ms average seek time, spinning at 5400 rpm. You are going to choose a file at random from a set of 1000 1-MByte files on the disk, read its entire contents, and repeat this process indefinitely. The disk has the "spiral" property that it can transfer any number of consecutive blocks at full rate. Assume that you don't need any disk accesses to obtain the sequence of extents for a file.

(A) What is the average time to read one file, if each file is allocated contiguously on the disk?

(B) What is the average time to read one file, if the file contents are allocated in 10 KByte blocks arranged randomly on the disk? By what factor is this slower than in part (A)?

(C) What is the smallest block size that yields only 10% slowdown from the result of part (A) under the assumptions of part (B)?

Problem 3 (6 points):

Give an encoding procedure for the variable `dir` declared by:

```
TYPE N  = String
     IN = Int

VAR  dir: N -> IN
```

given `enc` methods for `N` and `IN`.

Problem 4 (12 points):

One way to build a file system without any index information is to store each block of a file at a disk address obtained by hashing the block number and the file's inode number, or at an address obtained by following a linked list from that point. Here are some relevant declarations; note that the type `DB` has been extended with additional information to allow blocks in a hash bucket to be recognized and chained together.

```
TYPE DB  = [next: DA, inode: Int, blockno: Int, data: SEQ Byte]
     N   = String

VAR  dbSize := ...
     disk: SEQ DB
     dir : N -> [inode: Int, size: Int]
```

(A) Give the abstraction function from this state to the specification for files in Handout 9.

(B) Why doesn't anyone implement a file system this way?

Problem 5 (12 points):

In a token ring it's sometimes necessary to regenerate the token (for example, during initialization, or if it is damaged by a transmission error). This is implemented by electing the node with the smallest address to regenerate the token, as follows. A node determines (by timeout) that the token is lost, and sends out a special "elect address A" chunk, where A is the node's address (of course, more than one node may do this). When a node with address B receives "elect A", it proceeds as follows:

If A < B, it forwards the chunk.
If A > B, it forwards "elect B".
If A = B, it declares itself elected.

(A) What is the invariant maintained by this algorithm that justifies a node declaring itself elected when it sees A=B? State the invariant in Spec using the following declarations (adapted from PS 8):

```
TYPE
    A     = Int                                 % Address
    Elect = [a]
    C     = (... + Elect)                       % Chunk
    B     = SEQ C                               % Buffer
    NState= [b, a, ...]                         % Node state
    N     = Int                                 % Node: index in z

VAR z     : SEQ NState                          % the state of the nodes
```

You may use the function `Between` defined by

```
FUNC Between(n, n0, n1) =
    n0 < n1 => RET   (n0 < n /\ n < n1)
    [*]        RET ~ (n1 <= n /\ n <= n0)
```

(B) How should the timeout that initiates the election be chosen? You will need to impose some limits on the configuration and operation of the ring to answer this question.

Problem 6 (12 points):

This problem deals with crash recovery for a transaction system. Recall that the `LogRecovery` implementation of `SequentialTransactions` in Handout 31 requires the *log idempotence* property for updates recorded in the stable and volatile logs: if `s` is a state and `l` is a log (a sequence of updates), then `s + l + l = s + l`. Ben Bitdiddle argues that log idempotence is too complicated for mere mortals to understand, and proposes to replace it with the following simpler "update idempotence" property: if `s` is a state and `u` is an update, then `s + {u} + {u} = s + {u}`—*i.e.*, doing `u` twice has the same effect as doing it once.

(A) Suppose we require only update idempotence for the updates in the stable and volatile logs of `LogRecovery`. Explain briefly why the implementation fails to work.

(B) Describe briefly how you could change the implementation of the `Redo` operation in `LogRecovery` so that it works correctly when the updates in the log have only the simpler update idempotence property. (You may either describe your changes in English or give the revised code.)

Problem 7 (12 points):

In the spirit of KISS, Ben Bitdiddle proposes to simplify `LogAndCache` by eliminating the stable redo log (`sl`).

(A) Explain briefly how the implementation must change to work correctly. (Hint: you may assume for this part that each action can be performed as a single atomic update to stable storage; otherwise it's difficult to make things work.)

(B) Briefly discuss the performance implications of the changes you described in part (A).

Problem 8 (12 points):

This problem consists of two short answer questions.

(A) Explain briefly how the Ethernet network size is affected by a fixed minimum packet size.

(B) Describe two important differences between virtual circuit addressing and Internet addressing. Illustrate the differences with simple examples.

Note: problems 9–13 refer to the material in Part I.

Problem 9 (12 points):

Suppose we define the abstraction function for the variable `m` of `SimpleMutex` to be

```
m = (tl=nil=>available [*] held)
```

For each of the following transitions in `MutexImpl`, describe the sequence of specification transitions that explains it.

(A) A transition from before an invocation of `Acquire` to the beginning of the body of `Acquire`.

(B) The execution of the statement << `s(t) := C{...}` >> in `Acquire`.

(C) The execution of the body of `FetchAndStore` (called from `Acquire`).

Problem 10 (10 points):

Assume that each thread that acquires the mutex eventually calls `Release` to release it, and that the underlying scheduler is fair (*i.e.*, if a thread has an enabled transition, it will eventually be given a chance to run). Is it possible for `Acquire` or `Release` to fail to terminate? Explain briefly.

Problem 11 (10 points):

Show (with Spec code) how to implement `FetchAndStore` in terms of `CompareAndSwap`. Do not use atomicity brackets in your implementation; you can use assignment and expression evaluation, and you can call `CompareAndSwap`, but you cannot use any other atomic operations. (Recall that an entire assignment statement is not atomic; evaluation of the right-hand-side expression is one atomic action, and assigning that value to the left-hand-side is another.)

Problem 12 (12 points):

Suppose we changed `FetchAndStore` from an atomic procedure to a non-atomic procedure, removing the atomicity brackets around its body. Which of the following problems could occur that can not occur in the implementation shown above? Explain briefly why each can or cannot occur.

(A) Several threads can call `Acquire` concurrently and return from `Acquire`, each thinking it has the mutex.

(B) A thread calling `Release` can cause several threads spinning in `Acquire` to return.

(C) A thread can call `Acquire` and wait forever, even if after some point no other thread holds the mutex or calls `Acquire` or `Release`.

Problem 13 (8 points):

Give a scenario involving only two threads in which a thread in `Release` executes the body of the loop

```
DO s(t).tn = nil => SKIP OD
```