

Programming Languages and Systems  
Comprehensive Exam, Spring 2003

Yale University

May 20, 2003

First Name: \_\_\_\_\_

Last Name: \_\_\_\_\_

Yale NetID: \_\_\_\_\_

**Instructions**

- This exam contains 7 problems, solve all of them. Do not spend all your time on a single problem! If you get stuck, move on. Some questions are vague on purpose, please state your understanding and then explain your answer based on your assumptions.
- The exam is “nearly open book.” You may bring textbooks from the reading list, plus one page of notes.
- Answer each question on a separate piece of paper so that different faculty members can grade different questions in parallel. Try to write the solution in the space provided with each problem. You can use the back side of each sheet or the blank pages provided at the end of this booklet (i.e., pages 18-20).
- Write your name on top of each page so that the individual sheets can be separated.

Problem	Topic	Points
1	Concurrency	
2	Virtual Memory	
3	Processor Design	
4	Functional Programming	
5	Code Generation	
6	Computer Networks	
7	Formal Semantics	
TOTAL		

---

Your Name:

Yale NetID:

---

## Problem 1. Concurrency (10 Points)

The following code implements the readers-writers problem with semaphores. At any given point, the program allows one of the following two scenarios:

- multiple readers are accessing the shared buffer but no writers are present; or
- exactly one writer is accessing the shared buffer.

The main thread of the program (not shown) creates many reader and writer threads, starts each of them, and then waits forever for them to terminate (via calling each thread's `join()` method).

```
int sharedBuffer[] = new int[1000];
Semaphore mutex = new Semaphore(??); // initial value missing
Semaphore OKToRead = new Semaphore(??); // initial value missing
Semaphore OKToWrite = new Semaphore(??); // initial value missing

int ActiveWriters = 0;
int WaitingWriters = 0;
int ActiveReaders = 0;
int WaitingReaders = 0;

int READER = 0;
int WRITER = 1;

int whichAmI; // initialized by main

void run() {
    while (true) {
        if (whichAmI == READER) {
            reader();
        } else {
            writer();
        }
    }
}

void reader() {
    mutex.P();
    if ((ActiveWriters + WaitingWriters) == 0){
        OKToRead.V();
        ActiveReaders++;
    } else {
        WaitingReaders++;
    }
    mutex.V();
}
```

```

    OKToRead.P();

    // Perform read of sharedBuffer

    mutex.P();
    ActiveReaders--;
    if ((ActiveReaders == 0) && (WaitingWriters > 0)) {
        OKToWrite.V();
        ActiveWriters++;
        WaitingWriters--;
    }
    mutex.V();
}

void writer() {

    mutex.P();
    if ((ActiveWriters + ActiveReaders + WaitingWriters) == 0) {
        OKToWrite.V();
        ActiveWriters++;
    } else {
        WaitingWriters++;
    }
    mutex.V();
    OKToWrite.P();

    // Perform write of sharedBuffer

    mutex.P();
    ActiveWriters--;
    if (WaitingWriters > 0) {
        OKToWrite.V();
        ActiveWriters++;
        WaitingWriters--;
    } else {
        while (WaitingReaders > 0) {
            OKToRead.V();
            ActiveReaders++;
            WaitingReaders--;
        }
    }
    mutex.V();
}

```

You should assume that semaphores have been implemented in a reasonable way. You should assume that all semaphores are implemented with a First-in-First-out (FIFO) list of processes; that is, the process that has been waiting the longest for a semaphore will be the next one woken up.

- a) What values should each of the semaphores (i.e., mutex, OKToRead, OKToWrite) be initialized to?
- b) Assume a reader is currently using the buffer. If there are both waiting writers and waiting readers,

---

will a writer or a reader get access to the buffer first?

- c) Assume a writer is currently using the buffer. If there are both waiting writers and waiting readers, will a writer or a reader get access to the buffer first?
- d) Is it possible for readers to starve with this code? Is it possible for writers?
- e) Can the semaphore OKToRead have a value greater than 1? Can OKToWrite?
- f) Assume there are many readers currently accessing the buffer. Is the first writer to execute `mutex.P()` guaranteed to be the first writer to use the buffer? Is the first writer to execute `OKToWrite.P()` guaranteed to be the first writer to use the buffer?

-----  
Your Name: \_\_\_\_\_

Yale NetID: \_\_\_\_\_

## Problem 2. Virtual Memory (10 Points)

Given a logical address with the following format:

2 bits	16 bits	8 bits
Seg	Page #	Page offset

- a) Calculate the following properties of the memory system.
- (1) Number of segments
  - (2) Maximum size of each segment
  - (3) Size of each page
  - (4) Maximum number of pages per segment
  - (5) Maximum size of each page table (per segment). Assume each page table entry (PTE) requires 4 bytes.
- b) Imagine that you are building a memory management system in which you want each page table to fit within a single page in physical memory.
- (1) Show how you would divide the logical address such that each page table fits on a page. Please show any necessary calculations.
  - (2) With this new structure, how many memory references are needed to translate a logical address to a physical address? (Assume that there is no Translation Lookaside Buffer.) Briefly explain each memory reference.

---

Your Name:

Yale NetID:

---

### Problem 3. Processor Design (10 Points)

Consider a standard 5-stage pipelined single-issue RISC implementation, in which the cache access stage follows the ALU stage. The stages (of the integer pipeline) are:

- IF — Instruction fetch from Icache
  - ID — Instruction decode and register read
  - ALU — ALU operation or address generation
  - MEM — cache access
  - WB — write ALU result or cache data to register
- a) This pipe structure requires a "load delay" slot after a load instruction. Please explain what a load delay slot is and what the implications are for the compiler and for the hardware.
- b) Suppose we restructure the pipe by moving the ALU operation hardware to the MEM stage. In the old ALU stage we put new hardware to do the address addition (register plus displacement). The new pipe looks like this:
- IF — Instruction fetch from Icache
  - ID — Instruction decode and register read
  - ADDR — address generation
  - MEM/ALU — cache access or ALU operation
  - WB — write ALU result or cache data to register

Please evaluate this idea. What instruction sequences get faster? What sequences get slower? Is this a good idea, in your opinion?

---

Your Name:

Yale NetID:

---

## Problem 4. Functional Programming (10 Points)

Recall the following definition of `foldr` (in Haskell):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []      = e
foldr f e (x:xs) = f x (foldr f e xs)
```

The “fusion law” for `foldr` states: If `f` is strict, `f a = b`, and `f (g x y) = h x (f y)` for all `x` and `y`, then:

$$f \cdot \text{foldr } g \ a = \text{foldr } h \ b$$

Your job is to:

- a) prove the validity of the `foldr` fusion law; and
- b) use it to provide an INDUCTIONLESS proof that: for any finite list `xs`,

$$\text{reverse (reverse xs)} = \text{xs}$$

For (b), recall that `reverse` can be defined as:

```
reverse = foldr snoc []
  where snoc x xs = xs ++ [x]
```

You may also use the lemma:

$$\text{reverse (x 'snoc' xs)} = x : \text{reverse xs}$$

and the lemma:

$$\text{foldr } (:) \ [] = \text{id}$$

---

Your Name:

Yale NetID:

---

### Problem 5. Code Generation (10 Points)

Compilers typically perform resource allocation locally within functions while relying on a single global strategy to handle resource management across function boundaries. Typically, this global strategy places some resources in fixed locations, for example a heap pointer assigned to a certain register or function arguments placed in a certain set of registers. This strategy also governs what will happen to registers that are not assigned a fixed role. These may be preserved by the function (callee saves) or may be altered during the function call (caller saves).

- a) What are the tradeoffs between the caller saves and callee saves handling of registers?
- b) One way to avoid function call overhead is inlining. When is inlining possible and when is not? Are there any drawbacks to inlining that would cause it to decrease performance?
- c) Suppose each function in a program is allowed to choose a different register protocol. Assuming the absence of higher-order functions, informally sketch a practical algorithm that find a good protocol for each function in a program. What information is needed to obtain an optimal protocol assignment? How will this algorithm interact with the register allocation? *Warning: this is an open-ended question so don't spend too much time on this!*

---

Your Name:

Yale NetID:

---

### Problem 6. Computer Networks (10 Points)

A basic function of the link layer is media access control. Below we will consider two types of media access control protocols: channel partitioning, and randomized access.

- a) One channel partitioning scheme is code division multiple access (CDMA), which is used by cellular phones such as Sprint PCS. In CDMA, a user is assigned a code and the data of the user is modulated by the code. What is the relationship among the codes assigned to different users so that they can send simultaneously?
- b) One of the random access schemes is the slotted Aloha protocol. Using slotted Aloha, a user will access the media with probability  $p$  at the beginning of a slot. Suppose there are  $N$  users. What is the optimal value of  $p$  to maximize efficiency?
- c) One criticism of the slotted Aloha protocol is that its efficiency is low, and Ethernet was proposed to improve efficiency. How does Ethernet improve efficiency and what is the efficiency of the Ethernet protocol?
- d) An Ethernet network interface card will double its contention window size after each collision. This algorithm is called exponential backoff. Why does Ethernet use exponential backoff?

Your Name:

Yale NetID:

### Problem 7. Formal Semantics (10 Points)

Let  $M$  be a countably infinite set of *message names*, which we will print in **boldface**. Consider a language of expressions with the following syntax:

$$e ::= \{\} \mid e \leftarrow \{\mathbf{m} = e'\} \mid e \leftarrow m \mid \mathbf{m}e$$

where  $m \in M$ . The subset of values is defined by

$$v ::= \{\} \mid v \leftarrow \{\mathbf{m} = e\}$$

The relation of single-step reduction of expressions  $\rightsquigarrow$  is specified inductively by the rules

$$\frac{e \rightsquigarrow e'}{e \leftarrow \{\mathbf{m} = e_1\} \rightsquigarrow e' \leftarrow \{\mathbf{m} = e_1\}} \quad \frac{e \rightsquigarrow e'}{e \leftarrow m \rightsquigarrow e' \leftarrow m} \quad \frac{v \triangleright v \xrightarrow{\mathbf{m}} e}{v \leftarrow m \rightsquigarrow e}$$

where  $\triangleright$  is a helper relation (on quadruples) defined as follows:

$$v \triangleright v' \leftarrow \{\mathbf{m} = e\} \xrightarrow{\mathbf{m}} [v/\mathbf{m}e]e \quad \frac{v \triangleright v' \xrightarrow{\mathbf{m}} e}{v \triangleright v' \leftarrow \{\mathbf{m}' = e'\} \xrightarrow{\mathbf{m}} e} \text{ if } \mathbf{m} \neq \mathbf{m}'$$

and  $[v/\mathbf{m}e]e$  denotes the expression obtained by substituting  $v$  for all occurrences of  $\mathbf{m}e$  in  $e$ . For example,

$$(*) \quad \begin{aligned} \{\} \leftarrow \{\mathbf{one} = \mathbf{m}e \leftarrow \mathbf{two}\} \leftarrow \{\mathbf{two} = \{\}\} \leftarrow \mathbf{one} &\rightsquigarrow \{\} \leftarrow \{\mathbf{one} = \mathbf{m}e \leftarrow \mathbf{two}\} \leftarrow \{\mathbf{two} = \{\}\} \leftarrow \mathbf{two} \\ \{\} \leftarrow \{\mathbf{one} = \mathbf{m}e \leftarrow \mathbf{two}\} \leftarrow \{\mathbf{two} = \{\}\} \leftarrow \mathbf{two} &\rightsquigarrow \{\} \end{aligned}$$

The reduction relation  $\rightsquigarrow^*$  is the reflexive and transitive closure of  $\rightsquigarrow$ . An expression  $e$  is *stuck* if it is not a value and there is no  $e'$  such that  $e \rightsquigarrow e'$ . An expression  $e$  is *divergent* if for all  $e'$ , if  $e \rightsquigarrow^* e'$ , then there is some  $e''$  such that  $e' \rightsquigarrow e''$ .

Questions:

- Show the derivation for the first of the single-step reductions marked by (\*) in the example above.
- Give an example of a stuck expression.
- Are there divergent expressions in this language? If yes, give an example; if no, sketch a proof.
- Define values *true*, *false*, and *if* such that

$$\begin{aligned} \mathbf{if} \leftarrow \{\mathbf{argM} = e\} \leftarrow \{\mathbf{thenM} = \{\}\} \leftarrow \{\mathbf{elseM} = \{\} \leftarrow \{\mathbf{valM} = \{\}\}\} \leftarrow \mathbf{testM} \\ \rightsquigarrow^* \begin{cases} \{\}, & \text{if } e = \mathbf{true} \\ \{\} \leftarrow \{\mathbf{valM} = \{\}\}, & \text{if } e = \mathbf{false} \end{cases} \end{aligned}$$