Name: _____ CruzID: _____

CSE 130 Midterm 1 Key

Spring 2022

Part II. Free Response Questions

---

1. **The Q Operating System:** Dr. Quinn's lab is designing a new operating system, called "The Q". His team is considering the design, which consists of 12 modules. They need your help!

   This question includes parts (a)–(d).

   (a) (5 points) Dr. Quinn suggests a flat structure in which each module connects to every other module. In this design, each module will require 80 lines of code. How many total lines of code does his design use? How many connections does it have?

   Solution: Lines of Code: There are 12 modules, each with 80 lines of code. Thus, there are $12 * 80 = 960$ lines of code.
   Connections: There are 12 modules, each of which connects to all other modules. Thus, there are $12 * 11/2 = 6 * 11 = 66$ connections.

   Explanation: We used the following rubric:

Points	Category
1.25	Lines calculated as $Mods * Num\_Lines$.
1.25	Lines is 960.
1.25	Cons calculated as $\frac{Mods(Mods-1)}{2}$ or $Mods(Mods-1)$.
1.25	Cons is 66 or 132.
-1.25	Unidirectional links.

   (b) (5 points) Brian suggests an alternative, "what if we use layering!?". Brian's design arranges the 12 modules into a stack–each module connects to the module immediately above or below. To manage the new structure, each module will require 125 lines of code. How many lines of code does the layered design entail? How many connections will it have?

   Solution: Lines of Code: There are 12 modules, each with 125 lines of code. Thus, there are $12 * 125 = 1500$ lines of code.
   Connections: There are 12 modules arranged in layers, with a connection between each module. This makes 11 connections.

   Explanation: We used the following rubric:

Points	Category
1.25	Lines calculated as $Mods * Num\_Lines$.
1.25	Lines is 1500.
2.5	Conns is 11 or 22.
-1.25	Unidirectional links.

(c) (5 points) Nayan suggests yet another alternative, "let's use a hierarchy!". The hierarchy creates 3 "super-modules" and places 4 modules into each super-module. Each super-module connects with each of the other super-modules; each module within a super-module connects to all of the other modules within the same super-module. The original modules each require 80 lines of code; each super-module requires an additional 80 lines of code. How many lines of code does the hierarchical design entail? How many connections does it have?

Solution: Lines of Code: There are 12 modules, each with 80 lines of code, and 3 super-modules, each also with 80 lines of code. Thus, there are $15 * 80 = 1200$ lines of code.
Connections: There are three connections at the super-module layer. There are $4 * 3/2 = 6$ connections within each super-module, making a total of $3 * 6 + 3 = 21$ connections in the design.

Explanation: We used the following rubric:

Points	Category
1.25	Lines using $(Mods + SMods * Num\_Lines)$.
1.25	Lines is 1200.
1.25	Cons calculated as $\frac{(SMods*(SMods-1))}{2} + SMods * \frac{Mods*(Mods-1)}{2}$ or same $*2$
1.25	Cons is 21 or 42.
-1.25	Unidirectional links.

(d) (10 points) The lab wants to estimate how much time it will take to debug each design. Suppose that each bug is caused by a single line of code and that there is a bug every 30 lines of code distributed uniformly throughout each design. Assume that it takes 10 seconds per connection to isolate a bug to an individual module. I.e., it takes 10 seconds per connection to determine which super-module is buggy, 10 seconds per connection to determine which module within a super-module is buggy, 10 seconds per connection to determine which layer is buggy, and 10 seconds per connection to determine which module is buggy within the flat structure. Finally, once a bug has been isolated to a particular module, suppose that it takes one second to identify which line of code within the module is buggy. Which design will minimize the debugging time? Show your work for full credit.

Solution: Time for flat: There are $960/30 =$ bugs in the design. Identifying the module takes $66 * 10$ seconds, while finding the bug in a module takes 1 seconds. The total seconds is:

$$\begin{aligned}(960/30) * (66 * 10 + 1) &= 32 * 740 \\ &= 21152\end{aligned}$$

Time for layers: There are $1500/30 =$ bugs in the design. Identifying the module takes $11 * 10$ seconds, while finding the bug in a module takes 1 seconds. The total seconds is:

$$\begin{aligned}(1500/30) * (11 * 10 + 1) &= 50 * 235 \\ &= 5550\end{aligned}$$

Time for hierarchy: There are $1200/30 =$ bugs in the design. Identifying the super-module takes $3 * 10$ seconds, the module takes another $6 * 10$ seconds, and finding the bug in a module takes 1 seconds. The final number of seconds is:

$$\begin{aligned}(1200/30) * ((3 * 10) + (6 * 10) + 1) &= 40 * 170 \\ &= 1700 * 4 \\ &= 3640\end{aligned}$$

In conclusion, hierarchy has fewer bugs than layering has fewer bugs than flat.

Explanation: We used the following rubric:

Points	Category
1	Calculate Bug Count.
1	Calculate Module isolation time.
1	Add Bug isolation time.
1	Multiple bugs by isolation for flat.
1	Multiple bugs by isolation for layering.
1	Multiple bugs by isolation for hierarchy.
1	Flat time 21152 or 42272 seconds.
1	Layering time 5550 or 11050 seconds.
1	Hierarchy time 3640 or 7240 seconds.
1	Hierarchy is best.

Name: _____ CruzID: _____

2. **The Super-Large Ultra-Fast Global (SLUG) Storage System:** You were recently hired at Eugebe Data Systems, Congratulations! You'll be working on a new memory system, called the "Super Large Ultra-fast Global Storage System", or SLUG for short. SLUG provides two interfaces:

   1. `int slug_read(int addr)`: returns the current value of `addr`
   2. `void slug_write(int addr, int val)`: assigns the value of `addr` to be `val`.

   This question includes parts (a)–(d).

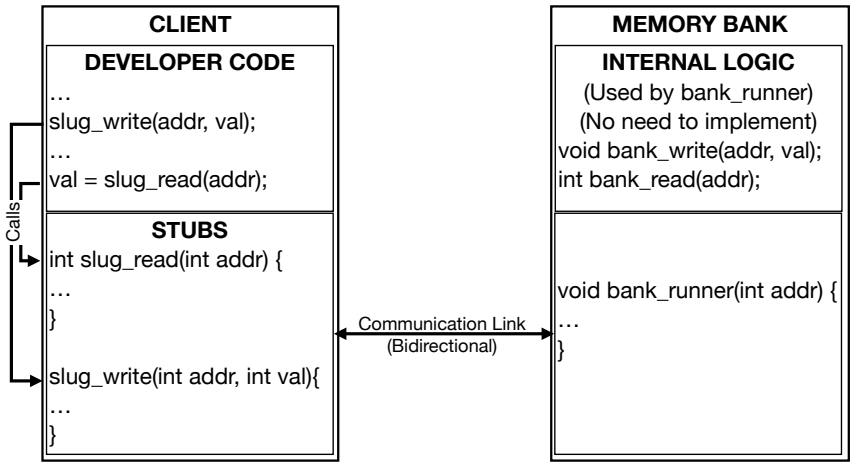
   (a) (10 points) Your boss, Ramesh, needs you to help design SLUG. SLUG uses an RPC-like design. `slug_read` and `slug_write` are stubs that a developer calls when they want to read or write data to SLUG. The stubs use an unreliable communication link to send messages to a `memory bank` that is responsible for storing actual values. The communication link enables the stubs and the `memory bank` to communicate using two operations:

   1. `void send(int one, int two, int three)`: sends three integer values, `one`, `two`, and `three`, to the other party.
   2. `int recv(int *onep, int *twop, int *threep)`: receive the values of `one`, `two`, and `three` from the other party's most recent call to `send`. `recv` will either, (1) return 0, indicating that data was successfully received, or (2) return -1, indicating that it timed out when waiting for a message.

   You need to implement the logic of the `slug_read` and `slug_write` stubs. You also need to implement the logic for `bank_runner`, the function that executes on the memory bank. `bank_runner` should execute forever (i.e., never return). Your `bank_runner` will need to receive messages from both `slug_write` and `slug_read` (think about how you can differentiate between the messages sent by each stub). Your `bank_runner` will need to call the following two functions to store and retrieve data from its internal storage:

   1. `void bank_write(int addr, int val)`: assigns the value of `addr` to be `val` in the memory bank.
   2. `int bank_read(int addr)`: returns the current value of `addr` in the bank.

   The diagram below shows the structure of the system:



   Implement the `slug_read`, `slug_write` stub functions and the `bank_runner` function on the next page. Your implementation cannot use any global variables. You may define constants (i.e., 'C' macros) to be used across the functions. Use 'C' syntax.

Solution:

```
#define WRITE 0
#define READ 1

void slug_write(int addr, int val) {
    int acked = 0, rop, raddr, rval, rc;

    // try to write
    send(WRITE, addr, val);

    // retry if no ack of current WRITE
    if (recv(&rop, &raddr, &rval) < 0 ||
        !(rop == WRITE || raddr == addr || rval == val))
        slug_write(addr, val);
}

int slug_read(int addr) {
    int acked = 0, rop, raddr, rval, rc;

    // try to read
    send(READ, addr, 0);

    // retry if no ack of current READ.
    if (recv(&rop, &raddr, &rval) < 0 ||
        !(rop == READ || raddr == addr || rval == val))
        rval = slug_read(addr);
    return rval;
}

void bank_runner() {
    int oper, addr, val;
    while (1) {

        // recv till request---brackets are for the faint of heart :)
        int rc = -1;
        while (rc < 0) rc = recv(&oper, &addr, &val);

        //do oper, send ack
        switch (oper) {
            case WRITE:
                bank_write(addr, val);
                send(WRITE, addr, val);
                break;
            case READ:
                val = bank_read(addr);
                send(READ, addr, val);
                break;
        }
    }
}
```

Explanation on Grading: This was a question about implemented read/write using send/recv. But, we changed the words and the semantics of send/recv to make you *apply* your knowledge. It was tough! We used the following grading scale
The high level approach is to have the read and write continuously try to send their request to the bank until they receive an ack. We used the following grading scale:

Points	Category
1	*send*() and *recv*() in a loop or recursive in *slug_read*().
1	*send*() and *recv*() in a loop or recursive in *slug_write*() .
1	*recv*() == 0/< 0 means ack/no ack in *slug_read*().
1	*recv*() == 0/< 0 means ack/no ack in *slug_write*().
2	*bank_runner*() uses an infinite loop.
2	Message types differentiate *read*() from *write*().
1	*bank_runner*() calls *bank_read*() for *read*().
1	*bank_runner*() calls *bank_write*() for *write*().
-1	Small bug.

We did not take off for submissions that did not handle stale acknowledgements (e.g., you don't have to have an ack message send the current operation, addr and val), since we didn't discuss this in class. There's actually an assumption that this implementation makes about send—I'll give you an extra credit point for each of the following tasks: (1) Draw a timing diagram of an execution of the algorithm that does not work correctly and (2) articulate the property of send that prevents the erroneous timing diagram from (1) from happening. Stated another way—what does this implementation require of send in order to work correctly? Submit your answers via the canvas extra credit quiz by April 29th.

(b) (5 points) You've been promoted—your hard work is paying off! Your new boss, Chris, wants to implement a "hot" idea coming out of cutting-edge research: "in-memory computing". In this design, the SLUG memory bank needs to be extended with an operation that increments the value of an address, `addr`, by a value, `val`, called `int bank_increment(int addr, int val)`. Additionally, SLUG would need a new stub, `void slug_update(int addr, int val)`, for developers to call when using SLUG and would modify `bank_runner` to handle the new operation. Could you reuse the implementation from (2a) to implement `slug_update` and the new `bank_runner`? If not, what would you have to change, and why would you have to change it?

Solution: You cannot reuse the implementation from (2a) because `slug_update` is not idempotent. Consequently, any `send` calls made by the `bank_runner` that are lost will result in extra updates to `addr`.

To fix this problem, the `bank_runner` would need to keep track of the operations that it has already processed. The best solution is to add an id to uniquely identify each message. `slug_update` would send this message on each request. `bank_runner` would track all of the message ids that it has already processed and only process a request if it has not received the message id in the past. A good approach for creating and maintaining such ids is to use the count of how many calls were made to `slug_update`.

(c) (10 points) Wow! You're crushing it. Eugene is even taking an interest in your work! He wants you to implement a new stub, `void slug_write_async(int addr, int val)`, that asynchronously assigns the value of `addr` to be `val`. SLUG's structure is unchanged from (2a): the `slug_write_async` stub is connected to a memory bank using an unreliable communication link.

Luckily, the team has already implemented the memory bank component and provided you with two convenient wrappers around `send` and `recv` to simplify your task. In particular, your `slug_write_async` stub can call the following functions:

1. `void rpc_write(int addr, int val)`: sends a message to the bank to assign the value of `addr` to be `val`. Note, the message may not be delivered.
2. `int rpc_recv(int *addr, int *val)`: either (1) receives the next acknowledgement message from the bank that the value of `addr` was assigned to `val` and returns 0, or (2) immediately returns -1 if there are no outstanding acknowledgement messages from the bank.

The bank implementation ensures that whenever it receives a message from `rpc_write`, it performs the write and sends an acknowledgement message that indicates the `addr` and `val`. Acknowledgement messages can be lost. Acknowledgement messages that are successfully delivered to the client will be returned to your `slug_write_async` function the next time it calls `rpc_recv`.

`slug_write_async` must ensure that there are at most `N` outstanding writes to the memory bank at all points in time, where an outstanding write is a write for which the client has not yet received an acknowledgement. You may assume that `N` is a small integer. Your function may store up to `2N ints` in global memory. Implement the `slug_write_async` stub below using the 'C' language.

Solution:

```
// Solution 1: from class

// store N outstanding writes
typedef struct{ int addr; int val;} update;
static update buffer[N];

// finds an entry in buffer matching addr, val.
update* find_match(int addr, int val) {
    for (update *curr = buffer; curr < buffer + N; ++curr)
        if (curr->addr == addr && curr->val == val) return curr;
    return NULL;
}

// finds an open entry in buffer
update *get_open() {
    // find unused slot
    update *up = find_match(0, 0);

    int rc = -1, addr, val;
    while (up == NULL) {

        // check for ack
        rc = rpc_recv(&addr, &val);

        // if ack, see if matches an element in buffer.
        // else, resend all writes to ensure eventual ack.
        // (brackets are still for wimps :))
        if (rc == 0)
          up = find_match(addr, val);
        else
            for (update *curr = buffer;
                 curr < buffer + N && curr->addr != 0;
                 ++curr)
                rpc_write(curr->addr, curr->val);
    }
    return up;
}

void slug_write_async(int addr, int val) {
    // get open slot, perform write, place write into slot

    update *open = get_open();
    rpc_write(addr, val);
    open->addr = addr;
    open->val = val;
}
```

Solution:

```
// Solution 2: updated since class
// store N outstanding writes
typedef struct{ int addr; int val;} update;
static update buffer[N];

// finds an entry in buffer matching addr, val.
update* find_and_clear(int addr, int val) {

    // iterate until end. Return if match on addr, val
    for (update *curr = buffer; curr < buffer + N; ++ curr)
        if (curr->addr == addr && curr->val == val)) {

            // found match; shift and clear buffer
            while ((curr + 1) < (buffer + N) && curr->addr) {
                *curr = curr[1];
                ++curr;
            }
            *curr = {0, 0};  //unnecessary, but it *feels* safer
            return curr;
        }
    return NULL;
}

// finds an open entry in buffer
update *get_open() {
    update *up = find_and_clear(0, 0);

    while (up == NULL) {
        int raddr, val;

        // if ack, see if matches an element in buffer.
        // else, resend all writes to ensure eventual ack.
        if (rpc_recv(&raddr, &val) == 0)
            up = find_and_clear(raddr, val);
        else
            for (update *curr = buffer;
                 curr < buffer + N && curr->addr != 0;
                 ++curr)
                rpc_write(curr->addr, curr->val);
    }
    return up;
}

void slug_write_async(int addr, int val) {
    // get open slot, perform write, place write into slot

    update *open = get_open();
    rpc_write(addr, val);
    open->addr = addr;
    open->val = val;
}
```

Explanation: This was a s t r e t c h question. We were hoping to be... dazzled. The point of the question is—how do you ensure at least once rpc, but allow there to be multiple outstanding operations?

The answer is complex, and we even updated our answer after class to cover a surprising issue! The high level approach for both solutions is to store at most N outstanding write operations in a buffer. The solutions eagerly use the full buffer at first, and then block if the buffer is full. When the buffer fills, the functions block until they successful receive an ack about an outstanding write. One crucial corner case is that the function *must* resend unacked writes—otherwise it might continuously fail to make progress because all of the outstanding writes/acks are lost! We call this type of problem a *liveness* problem; more on these later in the quarter.

We would have given full credit for both solutions listed here. We gave as many points as possible for solutions that were close, or showed that they understood something about rpc, send/recv semantics, etc. We used the following grading scale:

Points	Category
2	Stores non-acked writes across calls to *slug_async_write*.
2	Tracks outstanding writes in global structure.
1	Stores up to $2N$ elements in global structure.
1	Calls *rpc_write*() on *addr*, *val*.
2	Calls *rpc_recv*() in loop for *rpc_recv*() timeout.
1	Calls *rpc_write*() in loop for *rpc_recv*() timeout.
1	Calls *rpc_write*() on outstanding writes when *rpc_recv*() timeout.
-1	Small bug

I'll give you extra credit if you can answer: why did I update the solution? In particular, you'll get an extra credit point on the exam for each of the following tasks: (1) Draw a timing diagram that can happen (and is a bug) in Solution 1 but cannot in Solution 2 and (2) articulate the bug in terms of what property it violates. Submit your answer via the canvas extra credit exam assignment by April 29th.

(d) (5 points) Suppose that the design of `slug_read` is unchanged from (2a). If a client decides to use the `slugs_write_async` interface instead of the `slugs_write` interface, what memory properties are they trading off?

Solution: The client is trading off memory coherency to get lower latency.