# ECE 459 W20 Final Exam Solutions

J. Zarnett

April 16, 2020

**(1)**

(a) Deadlock can occur with improper lock *acquisition* order. Release order does not really matter. Why? Releasing a lock is not a blocking operation; it can either unblock another thread/process or have no effect. But it cannot cause a deadlock to occur. If a deadlock was going to happen, it would happen on acquisition.

(b) The transaction of transferring funds is not atomic; an intermediate state can be seen where the first part has happened (sender amount decreased) and the second has not (recipient amount increased).

(c) False sharing is more likely to occur with stack memory; stack allocated memory is allocated contiguously (i.e., allocation $N+1$ is located right next to allocation $N$); this is not the case for heap memory which can be but is rarely located in adjacent memory locations.

(d) Yes, this is a good match for SIMD instructions. We're doing the same operation on a large block of data. Thus we could reduce the amount of instructions necessary to get the same result.

(e) The example that is shown in the notes is:

```
                              f = 0
/* thread 1 */                         /* thread 2 */
while (f == 0) /* spin */;             x = 42;
printf("%d", x);                       f = 1;
```

And the operations of thread 1 could be reordered.

(f) Alias analysis is difficult for many reasons (but any single answer is acceptable, if properly justified). You could argue it's because the entire program has to be analyzed, including all linked libraries (because memory can be provided as a parameter) to see if memory is aliased somewhere. You could also argue that because pointers are just numbers and you can do bad things like cast an int to a pointer and dereference it, so it's basically impossible to tell what is a pointer and what's not.

(g) This is a Binomial distribution. User's behaviour to review is independent (we'll assume, at least). If users have an $x$% chance of giving your app a review then we are interested in the cumulative number of reviews and that means Bernoulli.

You could also make an argument by analogy to the lecture notes example that says "how many disks die in a year", because the model is the same, just replace disk with user and die with review.

(h) You can argue this both ways:

Yes: the container deployment model allows you to keep old libraries/projects with security vulnerabilities in service even though the ones in the operating system would be updated already as part of standard OS updates.

No: whereas previously you have to update all the VMs/operating systems and this can take time and be forgotten; containers mean you don't have to do this and it's easier to keep images up to date.

**(2.1)**

```
function find_plaintext( rainbow_table table, string hash ) : returns string
  int chain = -1
  string candidate = hash
  while true
    chain = get_chain( table, candidate )
    if chain is not -1
      break
    end if
    string reduced = clever_reduce( candidate )
    candidate = secure_hash( reduced )
  end while

  string plaintext = get_starting_plaintext( table, chain )
  while true
    string new_hash = secure_hash( plaintext )
    if new_hash equals hash
      return plaintext
    end if
    plaintext = clever_reduce( new_hash )
  end while
  return plaintext
```

**(2.2)** (Memory.) All that is needed is a misuse of memory (e.g., use-after-free, or going off the end of the array). Use-after-free is prevented in Rust by the fact that memory management is not manual but instead the release of that memory happens automatically when it is no longer needed.

(Concurrency.) All that is needed is some code that demonstrates a race condition in C, such as thread 1 and thread 2 modifying a global variable concurrently. This is prevented in Rust by the fact that it allows only one mutable reference at a time.

**(2.3)** Some of the strategies that might be valid:

- Generate $N$ random paths and just pick the best of those
- Use historical data from trips people have taken in the past
- Pathfinding algorithm like A* with cost function based on traffic

And a brief reason for why you chose what you chose.

**(2.4)**

(Response Times.) Apply Little's Law: $E[N] = \lambda E[T]$. The average time in the system $E[T] = 6$s. Arrivals 1 in odd-numbered seconds and 2 in even-numbered seconds so the average arrival rate $\lambda = 1.5$. Then $E[N] = 1.5 \times 6 = 9$. So there are 9 requests in progress on average.

(*Enterprise*.) This is a closed system so the formulation of Little's Law is $N = XE[T]$ . We know $N = 19$ and $E[T] = 0.9 \times 1 + 0.1 \times 10 = 1.9$. That means the throughput $X = 10$ for the system. This does not make sense though, because if 10% of the items go to the database and throughput is 10 it means 1 item goes to the database every second; if it takes 10 seconds for the database to complete an item, there should be (on average) 10 items in progress at a time. So the answer of 5 makes no sense.

**(2.5)**

```
void push( lf_stack * s, s_node * n ) {
  s_node* head;
  do {
    head = s->top;
    n->next = head;
  while( !__sync_bool_compare_and_swap (s->top,
       head, n ));
}
```

```
/* Returns NULL if the queue is empty */
s_node * pop( lf_stack * s ) {
  s_node* old_top;
  s_node* new_top;
  do {
      old_top = s->top;
      if ( old_top == NULL ) {
        return NULL;
      }
      new_top = old_top.next;
  while( !__sync_bool_compare_and_swap (s->top,
       old_top, new_top ));
  return old_top;
}
```