We'll post an outline of each recitation after the fact so you know what was covered in case you had to miss class that day. The recitation outlines are not a full replacement for recitation!

This week's recitation was largely a problem-solving session (good for exam prep). These are the problems we went over; some sections may not have gotten to every problem.

The Tail at Scale

Consider some distributed systems — called A, B, C, D, E — where a client can send a request to any one of *n* servers for an answer. There are many clients and many servers in each system. The table below indicates some differing characteristics of these systems

System	Unpredictable?	Crash?	Client Cancellation?	Server Cancellation?
A	No	No	No	No
B	Yes	No	No	No
C	Yes	Yes	Yes	Yes
D	No	Yes	Yes	Yes
E	Yes	No	Yes	Yes

- **Unpredictable** means there is a small chance (~3%) for each server on each request that it will take 2-10x longer than usual to give an answer.
- **Crash** means there is a small chance (<1%) for each server on each request that it will crash and never give an answer
- **Client Cancellation** means that clients in the system can *optionally* cancel a previous request to a server, ending any computation happening there
- **Server Cancellation** means that servers in the system cooperate: when a server starts on a request, it cancels all duplicate requests queued at other servers

Additionally:

- Every server gives the same answer for the same request
- All *n* servers are roughly the same distance from the client

Question 1: Describe these systems! What are they each like?

Now consider the following request strategies:

Strategy	Description
SINGLE	Client sends a single request to a single server
INDEPENDENT	Client sends two requests (roughly simultaneously) to two different servers
CANCEL	Client sends two requests (roughly simultaneously) to two different servers; on receipt of the first response, the client cancels the other request. This strategy is only applicable to systems that support client cancellation.
HEDGED	Client sends a single request to a single server; then, if response is not received within typical timing, the client sends an additional request to a different server.

Question 2: What are the pros and cons of each of these strategies for each of the systems above? Is there a clear "best" strategy for each system?

Threads #1

Ben Bitdiddle wants to implement a concurrent application. Instead of doing the sensible thing and holding a lock while manipulating shared data, Ben decides to be clever. Ben's code, shown below, runs in two separate threads on a computer with two CPUs; no other threads are running on the system. There are three shared global variables, initialized before the threads start: i is initialized to zero, x is initialized to a three-element array of zeros, and x_lock is in the unlocked state.

Thread 1:

```
while i < 3 do:
     while i % 2 == 0 do:
          # (while i is even)
          yield()
     acquire(x_lock)
     x[i] = 1
     i = i + 1
     release(x_lock)
```

Thread 2:

```
while i < 3 do:
     while i % 2 == 1 do:
          # (while i is odd)
          yield()
     acquire(x_lock)
     x[i] = 2
     i = i + 1
     release(x_lock)
```

After both threads reach the end of their code segments, which of the following are possible values for the contents of the x[] array?

A. x=2,1,1
B. x=2,2,2
C. x=2,1,2
D. x=2,2,1

Now suppose that Thread 1 instead runs the following variation below. Thread 2 runs the same code as before (shown below for convenience), and global variables are initialized in the same way as before.

Thread 1:

```
a = 0
while i < 3 do:
      while i % 2 == 0 do:
            # (while i is even)
            yield()
            a = i
      acquire(x_lock)
      x[a] = 1
      i = a + 1
      release(x_lock)
```

Thread 2:

```
while i < 3 do:
      while i % 2 == 1 do:
            # (while i is odd)
            yield()

      acquire(x_lock)
      x[i] = 2
      i = i + 1
      release(x_lock)
```

After both threads reach the end of their code segments, which of the following are possible values for the contents of the x[] array?

A. x=2,1,1
B. x=2,2,2
C. x=2,1,2
D. x=2,2,1
E. One or both threads might not reach the end of the code segments

Having come to his senses, Ben wants to repair his code so that it doesn't take him nearly this long to understand its behavior. Which of the following options would make it easier to reason about the concurrency in Ben's code, while not introducing new errors?

A. Remove acquire() and release() to avoid reasoning about locks.
B. Move acquire() and release() around the entire outer while loop.
C. Acquire and release the lock around every single access to i.
D. Acquire the lock before the outer while loop and release it only around the yield call.

Threads #2

Jordan is implementing a bounded buffer. Currently he has written the following code for send() and receive():

```
send(bb, message):
      acquire(bb.lock)
      while True:
            if bb.in - bb.out < N:
                  bb.buf[bb.in mod N] <- message
                  bb.in <- bb.in + 1
                  release(bb.lock)
                  return

receive(bb):
      acquire(bb.lock)
      while True:
            if bb.out < bb.in:
                  message <- bb.buf[bb.out mod N]
                  bb.out <- bb.out + 1
                  release(bb.lock)
                  return message
```

Multiple senders and receivers share a bounded buffer bb that uses the above code. Assume that N is so large that every time a sender calls send, there will be room in the buffer for the message (i.e., bb will never be full).

Given that the buffer can never be full, will this code never deadlock, sometimes deadlock, or always deadlock?

Jordan updates his code to use condition variables. Assume Jordan's code is now correct. What is the benefit of using condition variables as opposed to calling acquire()/release() directly as in his initial code? Circle all that apply.

A. Condition variables eliminate the need for locks.
B. If used properly, condition variables help prevent threads from being woken up when they don't have work to do.
C. Condition variables allow threads to explicitly tell the processor when to suspend the current thread as opposed to waiting to be preempted.
D. Condition variables prevent threads from being preempted by the operating system.