Consider the Java code snippet below. Is it a legal use of Java synchronization? What happens if two threads A and B call get() on an object supporting this API, and then a thread C calls give() on that object? Summarize the outcome for all three threads. [30 points]

```java
Boolean ready; /* initialized to false */
synchronized void get() {
    synchronized(ready) {
        while(ready.isFalse()) /* ready == false? */
            ready.wait();
    }
}
synchronized void give() {
    synchronized(ready) {
        ready.setTrue(); /* ready = true */
        ready.notify();
    }
}
```

Since `ready` is a Java Object, it has a monitor. The object supporting this get/give API also has a monitor. So there are two monitors. Here they are used in a nested fashion, which results in deadlock.

When A calls `get()`, it acquires the outer monitor/mutex, then acquires the inner monitor/mutex. It then waits on the inner monitor/condition, which releases the inner mutex but leaves the outer mutex held. It blocks there awaiting a notify.

When B calls `get()` it attempts to acquire the outer monitor/mutex, which is held by A, so B blocks there.

When C calls `give()` it attempts to acquire the outer monitor/mutex, which is held by A, so C blocks there.

Birrell’s “Introduction to Programming with Threads” has a discussion of a similar example in Modula-2.
A few days ago someone on Piazza asked me to explain the difference between monitors (mutexes and CVs/condition variables) and semaphores. One way to understand the difference is to implement each abstraction in terms of the other. We did that for one direction in class. Let’s try the other direction.

The pseudocode on the right is a “first cut” implementation of monitors using semaphores. What’s wrong with it? Write a list of three distinct correctness properties that this solution violates. Outline a fix for each one. Feel free to mark the code. [30 points]

(1) Monitor wait() must block the caller. Here the first caller to wait() does not block. It is easy to fix by starting the condition semaphore at 0.

(2) Monitor wait() must release the mutex before blocking, and reacquire the mutex before returning. This is easy to fix by adding the code to mutex.v and mutex.p. Be sure that you understand how to do that, and why it works without any additional synchronization.

(3) Monitor signal() may affect only a waiter who called wait() before the signal(). In this implementation, a signal() can affect a subsequent wait(), because the condition semaphore “remembers” the signal.

Problem #3 is tricky to fix. “Outlining a fix” was 5 points. I accepted any answer that gave any answer that demonstrated an awareness of its trickiness. Some answers attempted to use another semaphore to prevent signal from ticking up condition unless a waiter was present. Good try, and I accepted it. But it still falls short. The “real” answer is to put a semaphore on every thread, maintain a queue of waiting threads explicitly, and v up the semaphore for a specific waiter selected to be awakened by the signal. The problem and potential solutions are discussed in detail in the Birrell paper “Implementing Condition Variables Using Semaphores”.

Some students gave answers relating to error checking code that they felt should be present. But what we care about is that monitors behave correctly when used correctly.
Part 3. Any last requests

This problem asks you to write the core of a program to issue and service disk requests. **As always:** “Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard data structures, e.g., queues: don’t write code for those.”

There are **N** requester threads that issue requests and one servicer thread to service the requests. Requesters issue requests by placing them on a queue. The queue is bounded to at most **MAX** requests: a requester waits if the queue is full. There are two constraints. First, each requester may have at most one request on the queue. Second, the servicer passes the queue to the disk (by calling `serviceRequests()` only when it is full (MAX queued requests) or all living requesters have a pending request.

Write procedures for the requesters and servicer to use. Use monitors/mutexes/CVs to control concurrency. [40 points]

```plaintext
const int N, MAX;
queue rq; /* request queue */

startRequest(Dbufferdbuf) {
  Left as an exercise. A few points:
  - Lock it down.
  - Requester waits until there is room in the queue.
  - After issuing a request, each requester waits until its request is serviced, so that each has at most one request pending.
  - Wake up the servicer to handle the requests.
  - Servicer waits for the right time to serve requests (N requests pending, or MAX).
  - Servicer wakes up waiting requesters after servicing.
}

servicer() {
}
```
Part 4. Shorts

Answer each of these short questions in the space available. [10 points each]

(a) In Lab #3, how does your elevator decide where to go next?

I accepted anything reasonable here.

(b) In Part 3 of this exam, how does the value of \( \text{MAX} \) affect disk performance (presuming \( \text{MAX} < N \))? Explain.

Batching requests increases latency, but the main point here is that it permits the disk to choose an efficient order that reduces seeking and therefore improves peak throughput (by reducing service demand).

Note that this is roughly the same answer as part (a).

(c) Can round robin scheduling yield better average response time than FCFS/FIFO? Why or not? (If yes, give an example.)

Yes. Just give an example with a very short job stuck behind a very long job.

(d) In an I/O cache (e.g., Lab #4), could a buffer ever be dirty but not valid? Why or not? (If yes, give an example.)

No. A dirty buffer by definition contains the value last written to the block, i.e., it has the “true and correct” copy of the block. Therefore it is valid.

(e) Can LRU replacement yield a better cache hit ratio than \( \text{MIN} \)? Why or why not? (If yes, give an example.)

No. \( \text{MIN} \) is provably optimal. A few students said “yes” on the grounds that \( \text{MIN} \) cannot be implemented in practice, because it requires the system to see into the future. I accepted that answer.
Part 5. Last time?

These questions pertain to a “classic” 32-bit virtual memory system, with linear page tables and 4KB pages. Answer each question with an ordered sequence of events. Feel free to draw on the back page: otherwise, I am looking for as much significant detail as you can fit in the space provided. [10 points each]

(a) Suppose that a running thread issues a load instruction on the following 32-bit virtual address: 0x00111414. Suppose that the address misses in the TLB. How does the hardware locate the page table entry for the page? Please show your math.

See solution from the last midterm.

(b) If the page table entry (pte) is valid, what information does the pte contain? What does the hardware do with it?

See solution from the last midterm. The pte contains (at minimum) the Page Frame Number, which is essence machine memory address for the start of the 4K page. The hardware adds the 12-bit offset extracted from the virtual address to obtain the corresponding machine address.

(c) Suppose now that the pte is “empty” (marked as not valid), resulting in a fault. How does the kernel determine if the faulting reference is legal, i.e., how does it determine whether or not the fault results from an error in the program?

See solution from the last midterm.

(d) If the reference is legal, how does the kernel handle the fault? (Assume that the requested virtual page has never been modified.)

See solution from the last midterm.

(e) Why don't operating systems use LRU replacement/eviction for virtual memory pages?

LRU would be prohibitively expensive for a VM system. To implement LRU, the OS must “see” every page reference, so it can put the referenced page on the tail of the LRU list. But the OS does not see every page reference: most page references are handled by the hardware/MMU as described in part (a) and part (b), without ever informing the OS or transferring control to the OS. Instead, the OS uses a weaker sampling approximation such as FIFO-with-Second-Chance or (equivalently) the Clock Algorithm. We discussed reference sampling and FIFO-2C in recitation and in lecture, but it was not in the slides and not required in your answer.