Part 1. Sticking points

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();
    }
}
```
Part 2. You tell me

A few days ago one 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]
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 \( \text{MAX} \) 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 \texttt{serviceRequests()} ) only when it is full (\( \text{MAX} \) 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]

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

startRequest( Dbuffer dbuf ) {

}

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?

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

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

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

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

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

(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?

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

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