

Problem 1: Definitions

Define each of the following terms:

- Multiprocessing:
- Multiprogramming:
- Multithreading:
- Preemption:

Problem 2: Threads

Suppose you have a variable x initialized to 0, and then start two threads running the following pieces of code:

Thread A	Thread B
<pre>for (int i = 0; i < 3; i++) { x += 1; }</pre>	<pre>for (int j = 0; j < 3; j++) { x += 2; }</pre>

Assume that load and store instructions are atomic, and that x must be loaded into a register before being increased (and stored back to memory afterwards).

a) What are the possible values of x after both threads finish? Explain at least one way in which each value can be reached.

b) Suppose that in thread B, you replace the line $x += 2$ with a special, atomic hardware instruction that increases x by 2 (this instruction cannot be preempted). What are the possible values of x after the threads finish now?

Problem 3: Readers and Writers

a) Suppose that we have implemented reader-writer locks using a lock and condition variables as in the lecture slides:

Reader	Writer
<pre>//First check self into system lock.acquire(); while ((AW + WW) > 0) { WR++; okToRead.wait(&lock); WR--; } AR++; lock.release(); // Perform actual read-only access AccessDatabase(ReadOnly); // Now, check out of system lock.acquire(); AR--; if (AR == 0 && WW > 0) okToWrite.signal(); lock.release();</pre>	<pre>// First check self into system lock.acquire(); while ((AW + AR) > 0) { WW++; okToWrite.wait(&lock); WW--; } AW++; lock.release(); // Perform actual read/write access AccessDatabase(ReadWrite); // Now, check out of system lock.acquire(); AW--; if (WW > 0){ okToWrite.signal(); } else if (WR > 0) { okToRead.broadcast(); } lock.release();</pre>

(In this code, the variables AW, WW, AR, and WR count active writers, waiting writers, active readers, and waiting readers respectively.)

Suppose that *all* of the following read and write requests arrive in very short order (while *R1* and *R2* are still executing):

Incoming stream: *R1*, *R2*, *W1*, *W2*, *R3*, *R4*, *R5*, *W3*, *R6*, *W4*, *W5*, *R7*, *R8*, *W6*, *R9*

In what order would the database process these requests? If you have a group of requests that are equivalent (unordered), indicate this by surrounding them with brackets. You can assume that the wait queues of the condition variables are FIFO (i.e. `signal()` wakes up the oldest thread on the queue).

b) How can you redesign the code in a) to run requests in an order that guarantees that a read always returns the results of writes that have *arrived* before it but not *after* it? (Another way to say this is that the reads and writes occur in the order in which they arrive, while still allowing groups of reads that arrive together to occur simultaneously.)

Problem 4: Synchronization Primitives

a) While working on project 1, one of your partners proposes to use a modified version of the Semaphore class to implement condition variables as follows:

```
class ConditionVariable {
    ConditionVariable() {
        semaphore = new Semaphore(0);
    }
    void wait(Lock lock) {
        semaphore.P(lock); // Modified version of P() that
                           // releases lock if it waits
    }
    void signal() {
        semaphore.V();
    }
}
```

Decide whether you think their proposal will work, and explain your decision.

b) Assume now that someone has implemented condition variables (and locks) correctly and efficiently for you. How can you use their implementation of condition variables to implement semaphores? Describe what your semaphore's initialize(int value), P() and V() methods would do.