1. (a) i. 10 marks

L1 = L2 = L3 = L4 = 0;

COBEGIN

BEGIN S1; P(L1); S3; V(L2); S4; V(L3); P(L4); S6; END

BEGIN S2; V(L1); P(L2); S5; V(L4); P(L3); S7; END

COEND

ii. 3 marks

<-p

/  |  COBEGIN
/  /  \
| S1... S2...
\  \ /
'--->| COEND

(b) i. 2 marks No, move empty.V() after remove element.

ii. 3 marks No, requires mutual exclusion on insert/remove for producers/consumers. One mutex lock or two, depends if the buffer supports simultaneous insert/remove.

(c) 2 marks A shadow queue retains the kind of task blocked on a condition variable.

A shadow queue allows readers/writers to wait on the same condition queue in temporary order, while identifying multiple adjacent reads at the head of queue.

(d) 2 marks List of separate semaphores, one for each waiting task, versus multiple waiting tasks on a single semaphore.

Private semaphores maintain temporal ordering among blocking readers or blocking writers because of preemption.
2. (a) **4 marks**
   
i. All cars simultaneously gave up waiting and moved into the intersection.
   
ii. Do not enter the intersection until it is possible to proceed through it.
   
iii. deadlock (gridlock)
   
iv. circular hold-and-wait

(b)  
   
i. **1 mark** Each task acquires the right resource first and then the left (or vice versa).
   
ii. **1 mark** Any one of:
      
      • Each even task acquires the right resource first and then the left, while each odd task does the reverse (left than right).
      
      • Each task acquires the right resource first and then the left, except for one task, which acquires the left resource first and then the right.
      
      • A task cannot acquire a resource unless it can acquire both of them.

(c) **4 marks** No deadlock

(d) **1 mark** Sally Sparrow is the protagonist in the Angels video (Dr. Who episode *Blink*).
3. (a) i. 6 marks

```c
shared list/queue<int> q; // shared queue of integers

producer task
1 region q
1 await q.size() < 10 {
1 // add item to queue
1 }
}
```

consumer task

```c
region q
1 await ! q.size() > 0 {
1 // remove item from queue
1 }
```

ii. 1 mark A false condition ⇒ the region lock is released and entry is restarted (busy waiting).

(b) 1 mark An object cannot be safely deleted, if there is a task running within it.

(c) 2 marks Multiple Vs may start multiple tasks simultaneously, while multiple signals only start one task at a time because each task must exit serially through the monitor.

(d) 2 marks A Java signal is nonblocking, i.e., the signaller continues executing in the monitor and the signalled task is deferred.

A Java monitor puts signalled tasks on the calling/entry queue.

(e) 1 mark A stream of calling tasks causes starvation (barging) of signalled/signaller tasks.

(f) 2 marks

- Nth task does notifyAll, leaves monitor and performs its ith step, and then races back (barging) into the barrier before any notified task restarts.
- It sees count still at N and incorrectly starts its i\(th + 1\) step before the current tasks have completed their ith step.

4. (a) 2 marks Indirect communication occurs through a third party, like a monitor, i.e., task ⇔ monitor ⇔ task. Direct communication occurs directly between tasks, i.e., task ⇔ task (rendezvous).

(b) 7 marks

<table>
<thead>
<tr>
<th>object properties</th>
<th>member routine properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>stack</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(^a\)5 and 6 are rejected because a thread cannot execute without a stack
\(^b\)7 is rejected because the lack of mutual exclusion requires explicit locking for direct communication, which is contrary to high-level concurrency.

(c) 2 marks

```c
_when( C1 ) _accept( m1 ) S1; or _when( C2 ) _accept( m2 ) S2;
```

(d) 2 marks An administrator uses signalBlock for synchronization to force a worker task to execute next so it can exit the mutex object with new work.

(e) 1 mark Expressions are NOT evaluated left-to-right in C/C++, so f can be done before f().

(f) 2 marks f3 f1
5. (a) **1 mark** Compilers and hardware reorder statements to initiate eager reads up the memory hierarchy.

(b) **2 marks** A cache line is the minimum amount of memory (64/128/256 bytes) loaded when moving data up the memory hierarchy.

The cache-line problem is *false sharing* if multiple disjoint shared-variables are loaded together into a single cache-line.

(c) **2 marks** Force variable loads and stores to/from registers to prevent eliding loads/stores of a variable’s state in a register.

(d) **2 marks** SC directly detects changes (using a hardware Oracle) to variables instead of indirectly checking if the value in a variable has changed.

(e) **2 marks** An *if* statement is non-linear (then/else) so SIMD execution must skip one of the two clauses when executed by multiple threads with different condition values.

(f) **2 marks** Having to explicitly start a Java thread introduces an error-prone protocol for an infrequent situation, as most threads are immediately started.

Having to explicitly join a Java thread is necessary because garbage collection precludes using implicit deallocation mechanisms to indicate join.

6. (a) **14 marks**

```cpp
_Monitor GME {
    unsigned int curr_session = UINT_MAX, scnt = 0;
    uCondition waiting;
    public:
        void start( unsigned int session ) {
            if ( scnt == 0 ) { // no one using resource ?
                curr_session = session;
            } else if ( curr_session != session || ! waiting.empty() ) {
                waiting.wait( session );
                if ( ! waiting.empty() && waiting.front() == curr_session ) waiting.signal();
            }
            scnt += 1;
        } // start

        void end() {
            scnt -= 1;
            if ( scnt == 0 && ! waiting.empty() ) {
                curr_session = waiting.front();
                waiting.signal();
            }
        } // end
}; // GME
```

(b) **3 marks** No, because

- scheduling depends on member parameter value(s), e.g., compatibility code for dating
- scheduling must block in the monitor but cannot guarantee the next call fulfils cooperation
```cpp
void main() {
    Waiter * waiters[NUM_WAITERS];

    for ( int i = 0; i < NUM_WAITERS; i += 1 ) { // create waiters
        waiters[i] = new Waiter( *this );
    } // for

    for ( ;; ) { // wait for diners and waiters
        _Accept( ~Host ) { break;
        } or _Accept( requestWork, reserveTable ) {
            _Accept

        if ( ! waiterBench.empty() ) { // waiter?
            if ( ! celebrities.empty() ) { // celebrities have priority
                celebrities.front().delivery( (Waiter *)waiterBench.front() );
                celebrities.pop_front();
                waiterBench.signalBlock();
            } else if ( diners.size() >= TABLE_SIZE ) {
                for ( int i = 0; i < TABLE_SIZE; i += 1 ) {
                    diners.front().delivery( (Waiter *)waiterBench.front() );
                    diners.pop_front();
                } // for
                waiterBench.signalBlock();
            } // if
        } // if
    } // for

    closed = true;
    for ( int i = 0; i < NUM_WAITERS; i += 1 ) {
        if ( waiterBench.empty() ) _Accept( requestWork );
        waiterBench.signalBlock();
    } // for

    osacquire( cout ) << " each waiter gets:" << (double)tips / NUM_WAITERS << endl;
    for ( int i = 0; i < NUM_WAITERS; i += 1 ) {
        delete waiters[i];
    } // for
} // Host::main
```