These are not the only answers that are acceptable, but these answers come from the notes or lectures.

1. (a) **2 marks** The Angels are in a livelock because, after the humans leave, and a cardboard is used to cover one of the Angles eyes, it can move and then so can the other Angels. The Angels are not holding any resource or waiting for a resource (no hold and wait cycle).

(b) **2 marks** A shadow queue keeps track of the kind of waiting tasks. It is used when preventing staleness/freshness after blocking different kinds of tasks in temporal order on the same lock.

(c) i. **1 mark** Staleness/freshness can occur among tasks trying to atomically block and release a lock.

   ii. **2 marks** One of:

      - Need atomic block and release
        
        \[
        X, q, P(\text{entry}_q); \quad // uC++ semaphore
        \]

      - readers/writers take ticket before putting baton down, and each task checks ticket with serving value and one proceeds while others reblock

      - list of private semaphores, one for each waiting task, versus multiple waiting tasks on a semaphore.

(d) **1 mark** A split-binary semaphore is used when different kinds of tasks have to block separately.

(e) **2 marks** A race condition occurs when there is missing synchronization or mutual exclusion. It is hard to locate because the program runs but problems do not occur immediately because of non-determinism.

(f) i. **1 mark** mutual exclusion deadlock

   ii. **3 marks**

   \[
   \begin{align*}
   &L1.P() & 1 & L1.P() \\
   &R1 \quad \quad \quad \quad \quad \quad \quad \quad L2.P() & 1 & L2.P() \\
   &R1 \ & R2 \quad \quad \quad \quad \quad \quad R2 \quad // access resource \\
   &R2 \ & R1 \quad \quad \quad \quad \quad \quad // access resource
   \end{align*}
   \]

2. **12 marks**

   ```
   uSemaphore full(0), empty(10),
   ilock(1), rlock(1);
   void Producer::main() {
     for ( ;; ) {
       // produce an item
       1 empty.P();
       1 ilock.P();
       // add element to buffer
       1 ilock.V();
       1 full.V();
       // produce a stopping value
     }
   }
   void Consumer::main() {
     for ( ;; ) {
       full.P();
       rlock.P();
       // remove element from buffer
       1 rlock.V();
       1 empty.V();
       if ( stopping value ? ) break;
       // process or consume the item
     }
   }
   ```
3. (a) 5 marks

```cpp
class Mon {
    MutexLock mlock;
    int v;
    public:
        int x(...) {
            mlock.acquire();
            // compute v
            int temp = v;
            mlock.release();
            return temp;
        }
};
```

(b) 2 marks Internal scheduling controls selection of a task for execution in the monitor from the internal condition queues using `wait/signal`. 

*External scheduling* controls selection of a task for execution in the monitor from the external entry queues using `_Accept`, which form from calls to mutex members.

(c) 2 marks A monitor condition-lock uses external locking. The monitor lock protects the condition locks in all cases so no internal locking is required in the condition lock.

(d) 2 marks Barging can produce a performance gain because barging tasks may result in less blocking because they can steal a resource without blocking, reducing the total blocking in the system. For barging to be allowed, tasks accessing a critical section must be identical and it does not matter which one takes the resource.

(e) 9 marks

i. e  
ii. a and b called X, c called Y  
iii. d on A; g, h on B  
iv. tasks f accepts a mutex queue with e at the front, and e does an accept  
tasks e and f were on the condition variable and both were signalled  
v. signalling task  
vi. signalled task or a calling task  
vii. d (or signalled task)  
viii. signalling task, and tasks g h are moved to the A/S stack  
ix. a enters, and the accepting task, e and f are on A/S stack

4. (a) 3 marks

i. object (class)  
ii. monitor  
iii. coroutine monitor (monitor)  

(b) 2 marks S2 then S1

(c) 2 marks If code is moved from the server’s member routine to the server’s main, the client blocks on entry, and when it unblocks, it must determine if it is needs to raise an exception.
(d) 2 marks

```cpp
Task T {  
public:
    void start() {}
    ...
private:
    void main() {
        __Accept( start ); // delay until start called
        ...
    }
}
```

(e) 2 marks inheritance

The most derived construction is guaranteed to finish before the thread starts because the declaration precedes the call to start.

(f) 2 marks A courier’s purpose is to make potentially blocking calls on behalf of the administrator. A courier calls an administrator to get the target administrator and its message and then calls the target administrator to deliver the message.

(g) 2 marks Returning a value is difficult because the client must make a second call to retrieve the value and the second call must be matched with the first call so the server delivers the matching value.

5. (a) 3 marks False sharing is when threads are accessing disjoint (non-shared) variables but the variables are actually shared on the same cache line.

The sharing causes cache bouncing if each thread is writing to the variables.

Aligning or separating (padding) the variables so they are on separate cache lines.

(b) i. 5 marks

```cpp
int *ip;
...
1 if ( ip == NULL ) {  // no storage ?
1    lock.acquire();  // attempt to get storage (race)
1    if ( ip == NULL ) {  // still no storage ? (double check)
1        ip = new int( 0 );  // obtain and initialize storage
1    }
1    lock.release();
1 }
```

ii. 2 marks The assignment to `p` can occur before the initialization of the storage to zero.

(c) 3 marks In Dekker entry protocol

```cpp
    me = WantIn; // W
    while ( you == WantIn ) {
        both threads read DontWantIn, both set WantIn, both see DontWantIn, and proceed.
```

(d) 2 marks SC detects the change to top rather than indirectly checking if the value in top changes (CAA).

(e) 3 marks No, an Ada monitor cannot implement the dating-service problem. Because the wait-until cannot appear at the start of the entry (mutex) routine and can only contain monitor (class) variables.

(f) 2 marks A tuple space atomically reads/writes tuples by content, where content is based on the number of tuple elements, the data types of the elements, and possibly their values.

3
6. (a) 1 mark
   1 L3: 

(b) 3 marks
   1 L1: uCondition bench;
   1 L3: bench.wait();
   1 L4: while ( ! bench.empty() ) bench.signal();
   or
   1 L5: bench.signal();

(c) 11 marks
   1 L1: unsigned int tickets = 0, serving = group;
   1 L2: unsigned int myticket = tickets;
   1 tickets += 1;
   1 while ( myticket >= serving ) {
     1 wait();
   }
   1 L3: wait();
   1 if ( numVotes == 1 ) {
     1 serving += group;
     1 signalAll();
   } // if
   1 L4: if ( group == 1 ) serving += 1;
   1 else signalAll();

(d) 6 marks
   1 L1: AUTOMATIC-SIGNAL;
   1 bool ElectionOver = false;
   1 L3: WAITUNTIL( ElectionOver, , );
   1 if ( numVotes == 0 ) ElectionOver = false;
   1 L4: ElectionOver = true;
   1 L5: RETURN( talliedResult );
7. 25 marks

```cpp
    void DollyMaid::main() {
        unsigned int i, total = 0, maidCnt = 0, hireCnt = 0;

        for (; hireCnt < MaxHires; ) {  // limit of N hires per day
            if (hireCnt >= 2 && !clients.empty()) {
                Maids ret;
                ret.maids[0] = (Maid *)unhired.front();    // get maid addresses and unblock
                unhired.signalBlock();
                ret.maids[1] = (Maid *)unhired.front();
                maidCnt -= 2;
                clients.front().delivery(ret);  // inform client of maids
                clients.pop_front();
            }

            for (i = 0; i < NoOfClients; i += 1) {  // wait for remaining clients to return
                if (clients.empty()) __Accept( hire );
                clients.front().exception(new Closed);
                clients.pop_front();
            }

            closed = true;
            for (i = 0; i < NoOfMaid; i += 1) {  // wait for all maids to return
                if (unhired.empty()) __Accept( checkIn );
                unhired.signalBlock();
            }

            osacquire( cout ) << "amount earned:" << total << endl;
        }
    }
```