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

1. (a) **6 marks**

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Mutual Exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BinSem lk(0);</td>
<td>1 BinSem lk(1);</td>
</tr>
<tr>
<td>1 S1 lk.P()</td>
<td>1 lk.P()</td>
</tr>
<tr>
<td>1 lk.V() S2</td>
<td>critical section</td>
</tr>
</tbody>
</table>

(b) **1 mark** *Simple* allows only one thread in the critical section, whereas a *complex* allows multiple threads in the critical section.

(c) **7 marks**

\[
\begin{align*}
L1 &= L2 = L3 = L4 = L5 = L6 = 0; \\
\text{COBEGIN} &\quad \text{BEGIN} \quad S1; V(L1); \quad \text{END} \\
&\quad \text{BEGIN} \quad S2; V(L2); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L1); P(L2); \quad S3; V(L3); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L3); V(L3); \quad S4; V(L5); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L3); V(L3); \quad S5; V(L6); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L5); V(L5); P(L6); V(L6); \quad S6; \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L5); V(L5); P(L6); V(L6); \quad S7; \quad \text{END} \\
\text{COEND}
\end{align*}
\]

or

\[
\begin{align*}
L1 &= L2 = L3 = L4 = L5 = L6 = L7 = L8 = 0; \\
\text{COBEGIN} &\quad \text{BEGIN} \quad S1; V(L1); \quad \text{END} \\
&\quad \text{BEGIN} \quad S2; V(L2); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L1); P(L2); \quad S3; V(L3); V(L4); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L3); \quad S4; V(L5); V(L6); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L4); \quad S5; V(L7); V(L8); \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L5); P(L7); \quad S6; \quad \text{END} \\
&\quad \text{BEGIN} \quad P(L6); P(L8); \quad S7; \quad \text{END} \\
\text{COEND}
\end{align*}
\]
2. (a) **2 marks** A *split-binary semaphore* is a collection of semaphores where at most one of the collection has the value 1.

*Baton passing* is a conceptual idea, where the baton is acquired in the entry/exit code, and once the baton is released, cannot read/write variables in entry/exit.

(b) **1 mark** µC++ condition variables provide user storage for each waiting task.

(c) **2 marks** The chair is the head of the queue of waiting tasks in case an incorrect task is unblocked and has the highest priority.

(d) **2 marks** Race condition, missing synchronization/mutual-exclusion is difficult to locate as it does not cause an immediate error.

(e) **4 marks**
   i. cars arrived simultaneously
   ii. on simultaneous arrival, the car on the right has the right-of-way
   iii. livelock
   iv. car with highest licence-number proceeds, and any other reasonable suggestions

(f) **2 marks** An algorithm that *prevents* deadlock removes one of the conditions necessary for deadlock, thus ensuring that deadlock cannot occur. An algorithm that *avoids* deadlock might move into a potentially unsafe state, but the system prevents deadlock from occurring by refusing requests that would (conservatively) lead to deadlock.

(g) **4 marks** 1 No

\[ \text{Diagram} \]

R1  \[ T2 \]  \[ R2 \]  \[ T1 \]  \[ R3 \]

R1  \[ T2 \]  \[ R2 \]  \[ T1 \]  \[ R3 \]

R1  \[ T2 \]  \[ R2 \]  \[ T1 \]  \[ R3 \]

R1  \[ T2 \]  \[ R2 \]  \[ T1 \]  \[ R3 \]
3. (a) 2 marks
   • *daisy-chain signalling*: blocked tasks are woken by having each task wake the next blocked task after it unblocks.
   • *multiple signalling*: blocked tasks are woken by a (usually) single task.

(b) 2 marks
   • knowing the value of parameters to the incoming call
   • having to block for an arbitrary time after entering

(c) 2 marks A μC++ monitor prevents *barging* by moving acceptor/signalled tasks to the internal acceptor/signalled stack, and servicing it before calling tasks.

(d) i. 2 marks No busy waiting because there are bounded number of tasks that must be restarted and their predicates checked.
    ii. 1 mark Knowing when the end of the cyclic is reached so the checking task does not loop.

(e) 8 marks

```cpp
_Monitor semaphore {
  int cnt;
public:
  semaphore( int cnt = 1 ) : cnt( cnt ) {}  
      void V() {
        cnt += 1;
        RETURN();
      }
      void P() {
        WAITUNTIL( cnt > 0, );
        cnt -= 1;
        RETURN();
      }
};
```
4. (a) **4 marks**
   i. calls to member A are accepted before calls to member B
   ii. have another accept statement that accepts B before A
   iii. first statement accepts a call to A, and the second optionally accepts a call to B if one is available.
   iv. _Else_ clause is present when the task has other work that could be done in between receiving requests

(b) i. **2 marks** The administrator signals the blocked worker, which moves the worker to the acceptor-signalled stack. Since the administrator continues to execute, the worker is prevented from leaving to perform its work.
   ii. **2 marks** Change `bench.signal()` to `bench.signalBlock()`, so the administrator blocks and the worker executes.

(c) **3 marks**
   i. The caller can perform other work while the server is computing the call.
   ii. A buffer is implicitly inserted between client and server to hold the arguments of the call.
   iii. The protocol for picking up returned values from the server.

(d) **1 mark** Because multiple tasks may need to read the value of the future.

(e) **2 marks** _Accept_ is used on the server side to block for client requests, while _Select_ is used on the client side to block for server results.

5. (a) **2 marks** compiler can reorder declarations, heap provides no control on storage placement

(b) **2 marks** It is a list of sequential optimizations performed implicitly by the compiler and/or hardware.

(c) **2 marks**
   ```
   temp = you; // R
   me = WantIn; // W
   while ( temp == WantIn ) {
   }
   ```
   both threads read `Dont WantIn`, both set `Want In`, both see `DontWantIn`, and proceed

(d) **2 marks** Because variables must be rotated through the registers frequently when there is only a few of them.

(e) **1 mark** spurious wakeup

(f) **1 mark** GPUs are SIMD.
6. 22 marks

```cpp
_Monitor Bridge {
    const unsigned int N;
    bool direction = true;
    unsigned int start = 0, finish = 0;
    uCondition bench[2];

    _Mutex void begin( bool dir ) {
        if ( start == finish ) { // bridge empty ?
            direction = dir; // set new direction
        } else if ( direction == dir && start < N ) { // traffic in my direction ?
            bench[dir].wait(); // wait turn
        } else {
            start += 1; // start across
        } // Bridge::begin
    }

    _Mutex void end() {
        finish += 1; // complete trip
        if ( start == finish ) { // bridge empty ?
            start = finish = 0; // reset
            if ( ! bench[ ! direction ].empty() ) { // opposite direction waiting ?
                direction = ! direction; // change direction
            } // if
            for ( int i = 0; i < N; i += 1 ) // unblock at most N cars
                bench[direction].signal(); // may signal empty condition
        } // if
    } // Bridge::end

public:
    _Nomutex void cross( bool direction ) {
        begin( direction );
        uThisTask().yield( rand() % 10 ); // pretend to cross bridge
        end();
    } // Bridge::cross
}; // Bridge
```
7. 26 marks

_Task CouplesAreToo {

    private:
    void deliver() {
        locn += 1;  // advance location for new date
        pairs.front().delivery(locn);  // deliver location to pair at front of list
        pairs.pop_front();
        pairs.front().delivery(locn);
    }
    chapers.signalBlock();  // wakeup chaperone
}

    void closing( list<Flocn> & person ) {
        while ( ! person.empty() ) {
            person.front().exception(new Closed);  // raise exception
            person.pop_front();
        }  // while
    }

    void main() {
        for ( ;; ) {
            _Accept( ~CouplesAreToo ) {  // time to close ?
                break;
            } or _Accept( chaperone ) {  // chaperone arrived ?
                if ( ! pairs.empty() ) deliver();  // couple waiting ?
            } or _Accept( date ) {  // girl/boy arrived ?
                if ( ! people[!sex][ccode].empty() ) {
                    Flocn fl = people[0][ccode].front();  // move pair to other list
                    people[0][ccode].pop_front();
                    pairs.push_back( fl );
                    fl = people[1][ccode].front();
                    people[1][ccode].pop_front();
                    pairs.push_back( fl );
                    if ( ! chapers.empty() ) deliver();  // chaperone available ?
                }  // if
            }  // _Accept
        }  // for
        locn = -1;  // indicate closed
        while ( ! chapers.empty() ) chapers.signalBlock();  // unblock waiting chaperones
        closing( pairs );  // unblock waiting pairs
        for ( int j = 0; j < 2; j += 1 ) {
            for ( int i = 0; i < NoOfCCodes; i += 1 ) {
                closing( people[j][i] );
            }  // for
        }  // for
    }
};

-6 starvation, i.e., lose marks for not using pair.