1. (a) 7 marks
   P: line 8 pickup, line 11 put down, line 11/12/13 received, line 14 put down
   V: line 17 pick up, line 19 pass, line 22 put down
(b) 3 marks Full marks if reverse terms avoidance and prevention but otherwise answer correctly.
   • Barging avoidance allows bargers to enter the mutual-exclusion entry-protocol but flags/tickets immediately block them.
   • Barging prevention does not release the spin lock so bargers cannot enter the mutual-exclusion entry-protocol.
   • barging prevention
(c) 2 marks
   • both readers enter ⇒ 2:00 reader reads data that is stale; should read 1:30 write
   • writer enters and overwrites 12:30 data (never seen) ⇒ 1:00 reader reads data that is too fresh (i.e., missed reading 12:30 data)
(d) 2 marks There is a continuous stream of identical cars on the right, and every time the left driver looks, it sees the same car and thinks it is still the simultaneous arrival and defers.
(e)  • 4 marks

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Maximum Needed</td>
<td>Current Acquired</td>
<td>Needed to Max.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The state is safe as this particular sequence of execution is safe.

• 2 marks

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>11 Total Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-10 Used</td>
</tr>
<tr>
<td>Maximum Needed</td>
<td>Current Acquired</td>
<td>Needed to Max.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>1 Available for allocation</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The state is NOT safe as no sequence of execution after this point is safe.
(f) **2 marks** Any two of:
- must arbitrarily select one of tasks to remove from the deadlock cycle and prevent immediate restarting.
- preempted victim must be restarted, either from beginning or some previous checkpoint state.
- difficult/impossible to guarantee preempted victim has not changed external resource before restarting.

2. (a) **2 marks**
- 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 fulfills cooperation, e.g., if boy accepts girl, she may not match (and boys cannot match).

(b) **2 marks**
- for read-only routines so that writers are not slowed during reading, e.g., a buffer size for drawing a graph of buffer usage.
- for combining a multi-step protocol into a single routine, where some of the steps need to release the monitor lock, e.g., allowing multiple readers for a readers-writer lock.

(c) **5 marks**
```cpp
_Monitor semaphore {
  int cnt;
  public:
  semaphore(int cnt = 1): cnt(cnt) {} 
  void V() {
    cnt += 1;
  }
  void P() {
    if (cnt == 0) _Accept(V);
    cnt -= 1;
  }
};
```

(d) **2 marks** Automatic signal monitors rely on predicates, and hence, condition variables or signal statements are unnecessary to build synchronization cooperation.
It is expensive to reevaluate the predicates of all waiting tasks in the monitor after monitor variables change and the monitor becomes empty.

(e) **1 mark** assume programmers cannot perform cooperation, and barging forces them to give up quickly on attempting cooperation.

OR
spurious wakeup

(f) **2 marks** Must acquire a monitor to determine if blocking is necessary, and then must acquire the condition monitor and wait, which only releases the condition monitor-lock (nested-monitor problem)
3. (a) **2 marks** The _When_ clause allows an _Accept_ clause to conditionally accept a call depending on the state of the mutex object (task).

(b) **2 marks** The _When_ clause checks all combinations of the accepts with linear syntax, while the if statements takes exponential syntax.

(c) **2 marks** The vending machine is a server because it makes calls (name server/WATCard) and has no workers, i.e., it does the work itself.

(d) **3 marks**
   - value
   - exception (server generated)
   - cancelled (exception)

(e) **2 marks** Threads continually read/write same memory locations, invalidating duplicate cache lines, resulting in excessive cache updates.

(f) **2 marks** Force variable loads and stores to/from registers, and prevents this scenario:

   Task\(_1\) ... Task\(_2\)

   flag = false // write

   register = flag; // one read, auxiliary variable

   while (register); // cannot see change by T1

4. (a) **2 marks** Lock-free operations have no ownership duration so there is no hold required for deadlock wait-and-hold.

(b) **1 mark** individual progress (allow starvation), rule 5

(c) **4 marks** ABA problem

   First thread loads the current top of stack A and next node B, and then it is interrupted. Second thread pops node A, pops node B, and pushes node A, so it points at C. First thread restarts, verifies node A is still the top, removes A, but sets stack top to B rather than C.

(d) **1 mark** GPUs are SIMD.

(e) **2 marks** Go threads communicate using channels, which are bounded/unbounded buffers.
5. (a) 4 marks
    count += 1;
    if ( count < N )
    Accept( block );
    else
    count = 0;

(b) 4 marks
    uCondition bench;
    count += 1;
    if ( count < N )
    bench.wait();
    else
    count = 0;
    bench.signal();

(c) 4 marks
    AUTOMATIC_SIGNAL;
    count += 1;
    if ( count < N )
    WAITUNTIL( count == 0, , );
    else
    count = 0;
    RETURN();

(d) 7 marks
    unsigned int generation = 0;
    unsigned int mygen = generation;
    count += 1;
    if ( count < N )
    while ( mygen == generation )
    wait();
    else {
    count = 0;
    generation += 1;
    signalAll();
    }
    unsigned int tickets = 0, serving = N;
    unsigned int myticket = tickets;
    tickets += 1;
    if ( myticket >= serving ) wait();
    count += 1;
    if ( count < N ) {
    wait();
    if ( count == 1 ) {
    serving += N;
    signalAll();
    } // if
    } else {
    signalAll();
    } // if
    count = 1;
6. 26 marks

```c
void main() {
    unsigned int rafterCnt = 0, guideCnt = 0;
    Guide *guidetasks[NoOfGuides];

    // allocate guide tasks
    for ( unsigned int i = 0; i < NoOfGuides; i += 1 ) guidetasks[i] = new Guide( ~this );

    for ( ;; ) {
        // fill rafts until destructor called
        _Accept( ~Echo ) { // time to close ?
            break;
        } or _Accept( hire ) { // rafter called
            rafterCnt += 1;
        } or _Accept( onduty ) { // guide called
            guideCnt += 1;
        } // _Accept

        if ( rafterCnt >= R && guideCnt >= G ) { // raft holds R rafters and G guides
            leadGuide = (Guide *)guides.front(); // either of the 2 guides
            guideCnt -= G;
            for ( unsigned int i = 0; i < R; i += 1 ) {
                rafters.front().delivery( leadGuide );
                rafters.pop_front();
            } // for
            rafterCnt -= R;
        } // if
    } // while

    leadGuide = nullptr;
    for ( unsigned int i = 0; i < NoOfGuides; i += 1 ) {
        if ( guides.empty() ) _Accept( onduty ); // wait for guide to return
        guides.signalBlock(); // restart guide
    } // for

    // delete guide tasks
    for ( unsigned int i = 0; i < NoOfGuides; i += 1 ) delete guidetasks[i];
} // Echo::main
```