1. (a) 5 marks

```cpp
class Lock {
  unsigned int tickets, serving;
public:
  Lock() : tickets(0), serving(0) {} // entry protocol
  void acquire() {
    int ticket = fetchInc(tickets); // obtain a ticket
    while (ticket != serving) {} // busy wait
  }
  void release() { // exit protocol
    fetchInc(serving);
  }
};
```

ii. 1 mark If 4+ billion (assume 4 byte integers) tasks arrive simultaneously, the tickets overflow and threads get the same ticket.

(b) i. 3 marks

- there is exactly one baton
- nobody moves in the entry/exit code unless they have it
- once the baton is released, cannot read/write variables in entry/exit

ii. 1 mark 0 bytes, there is no actual baton

(c) i. 2 marks A time-slice between the V and P can result in a task barging or staleness so waiting is in non-temporal (non-FIFO) order.

ii. 2 marks Member Xwait.P(entry), which atomically blocks on Xwait and unlocks entry.

(d) 2 marks When the chair is empty, tasks at the front of the bench are unblocked until there is a writer that cannot enter. This writer waits in the chair, and the chair is always unblocked (priority) before the bench.

2. (a) 5 marks

i. There exists more than one shared resource requiring mutual exclusion.

ii. A process holds a resource while waiting for access to a resource held by another process (hold and wait).

iii. Once a process has gained access to a resource, the runtime system cannot get it back (no preemption).

iv. There exists a circular wait of processes on resources.

v. These conditions must occur simultaneously.

(b) 2 marks The angels are in a livelock because, after the humans leave, and a cardboard is used to cover one of the angels 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).

(c) 2 marks Order resources and allocate resources in that order to prevent a hold-and-wait cycle.

(d) 2 marks should release some resources, should not block or busy wait.
3. (a) 2 marks A monitor solution cannot allow *simultaneous insert/remove* to an appropriate buffer because of the *mutual exclusion property*.

(b) 3 marks SHARED declarations become private monitor declarations, REGION statements become public monitor members, AWAIT clauses become \_Accept or signal/wait statements.

(c) 1 mark External scheduling is simpler because unblocking (signalling) is implicit.

(d) 2 marks
\[
\text{\_Accept( M1, M2 ); // OR}
\text{\_Accept( M1 ); \_Accept( M2 ); // AND}
\]

(e) 2 marks For signal the signalling task continues execution until it waits or exits, and the signalled task is delayed (on the A/S stack).
   For signalBlock the signalling task is delayed (on the A/S stack), and the signalled task continues execution until it waits or exits.

(f) 3 marks A task calls into monitor M1 and monitor M2, and waits in M2, releasing M2’s monitor lock but not M1’s monitor lock. Because M1’s monitor lock is not released, a signalling task may not be able to enter M2 to signal the waiting task, leading to a **deadlock**.

(g) 2 marks Too confusing because either the signalled or signaler task can *randomly continue* in the monitor.

(h) 2 marks only one condition variable, barging

(i) 1 mark No!!

4. (a) 2 marks Without mutual-exclusion, multiple thread can enter the type object, including the task thread, which means the type’s data members must be protected by explicit locks.

(b) 4 marks
\[
\text{void mem() {}
1 \quad \ldots \_Throw E(); \ldots \text{ cause RendezvousFailure}
\}
\text{void main() {}
1 \quad \text{try {}
1 \quad \quad \ldots \_Accept( mem ); \ldots
1 \quad } \text{ catch( uMutexFailure::RendezvousFailure ) {} // deal with RendezvousFailure}
\}
\]

(c) 2 marks The rendezvous is postponed or subdivided, and the server must fulfill the rendezvous later and unblock the client.

(d) i. 2 marks \_Accept should block, run the destructor, and then unblock, but the object is gone (deleted).
ii. 2 marks \_Accept continues running and the destructor call is postponed on the A/S stack.

(e) 2 marks Accessing a cancelled future raises an exception.
   There is race condition between the canceller and the processing/accessing of the future.
5. (a) 2 marks disk/memory, memory/registers

(b) 2 marks The heap memory-allocator may place variables x and y on the same cache line resulting in false sharing.

(c) 2 marks

\[
\text{data} = \text{Data}; \quad // W \\
\text{while} \ (\neg \text{Insert}) ; \quad // R \\
\text{Insert} = \text{false};
\]

Allows reading of uninserted data.

(d) 2 marks All data to be processes must be copied from the CPU to the GPU, and all results copied back.

(e) 3 marks requeue cancels the current call (request) to a task, reschedules the call on another (usually non-public) mutex member of the task, and accepts it later.

(f) 2 marks Go uses channels to support direct communication. Go uses a `select` statement to choose among a number of channels for data or block until data arrives.

(g) 1 mark implicit concurrency system

6. (a) 8 marks There is duplicate code, which is only counted once across the solutions.

\[
P() \\
3 \quad \text{if} \ ( \text{cnt} == 0 ) \ \text{for} \ (\_\_\_\_\_\_\_) \ _\text{Accept}( \ V \ ) \ \text{break}; \ \text{or} \ _\text{Accept}( \ \text{tryP} \ ); \\
1 \quad \text{cnt} -= 1; \\
\text{tryP}() \\
1 \quad \text{if} \ ( \text{cnt} == 0 ) \ \text{return} \ \text{false}; \\
1 \quad \text{cnt} -= 1; \\
1 \quad \text{return} \ \text{true}; \\
P( \ \text{Semaphore s} ) \\
1 \quad \text{s.V}(); \\
1 \quad \text{P}(); \quad // \text{or duplicate P() code} \\
\text{V}(); \\
1 \quad \text{cnt} += 1;
\]

(b) 7 marks

\[
\text{uCondition bench;} \\
P() \\
1 \quad \text{if} \ ( \text{cnt} == 0 ) \ \text{bench.wait}(); \\
- \quad \text{cnt} -= 1; \\
\text{tryP}() \\
1 \quad \text{if} \ ( \text{cnt} == 0 ) \ \text{return} \ \text{false}; \quad // \text{or same as for external} \\
- \quad \text{cnt} -= 1; \\
- \quad \text{return} \ \text{true}; \\
P( \ \text{Semaphore s} ) \\
1 \quad \text{s.V}(); \\
- \quad \text{P}(); \quad // \text{or duplicate P() code} \\
\text{V}(); \\
- \quad \text{cnt} += 1; \\
1 \quad \text{bench.signal}();
\]
7. 25 marks

```cpp
void MapleLeafTaxi::main() {
    Taxi * taxitasks[NoOfTaxi];

    for ( int id = 0; id < NoOfTaxi; id += 1 ) {
        taxitasks[id] = new Taxi( *this, id ); // allocate taxis
    }

    for ( ;; ) {
        _Accept( close ) {
            break;
        } or _Accept( getClient, getTaxi ) {
            if ( taxis.size() > 0 && clients.size() > 0 ) {
                LocnClient *n = clients.front();
                xclient = n->x; yclient = n->y;
                list<LocnTaxi*>::iterator nearest = nearestTaxi( n, taxis ); // find closest taxi
                n->ftaxi.delivery( (*nearest)->id );
                delete n; // allocated in getTaxi
                (*nearest)->idle.signalBlock();
                taxis.erase( nearest );
            }
        }
    }
    osacquire( cout ) << "Closed for the day." << endl;
    for ( int i = 0; clients.size() != 0; i += 1 ) { // notify potentially waiting clients
        LocnClient *client = clients.front();
        clients.pop_front();
        client->ftaxi.exception( new Closed ); // raise exception
        delete client; // allocated in getTaxi
    }
    closed = true; // tell taxi tasks to go home
    for ( int i = 0; i < NoOfTaxi; i += 1 ) {
        if ( taxis.empty() ) _Accept( getClient ); // wait for taxi
        taxis.front()->idle.signalBlock();
        taxis.pop_front(); // unblock with closed
    }
    for ( int i = 0; i < NoOfTaxi; i += 1 ) delete taxitasks[i]; // delete taxis
}
```