Final Exam Answers – CS 343 Winter 2017

Instructor: Peter Buhr

April 20, 2017

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

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**

	P1	P2	P3	P4
Maximum Needed	2	5	8	8
Current Acquired	1	2	4	2
Needed to Max.	1	3	4	6

11	Total Resources
-9	Used
2	Available for allocation
P1	1
	3
P2	0
	5
P3	1
	9
P4	3
	11

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

   • **2 marks**

	P1	P2	P3	P4
Maximum Needed	2	5	8	8
Current Acquired	1	1	5	3
Needed to Max.	1	4	3	5

11	Total Resources
-10	Used
1	Available for allocation
P1	0
	2

   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 fulfils 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**

```
_Monitor semaphore {
1   int cnt;
  public:
1   semaphore( int cnt = 1 ) : cnt( cnt ) {}
    void V() {
1       cnt += 1;
    }
    void P() {
1       if ( cnt == 0 ) _Accept( V );
1       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 an administrator because it never makes calls.

   (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:

$Task_1$	$Task_2$
...	**register** = flag; *// one read, auxiliary variable*
flag = **false** *// write*	**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**

```
1   count += 1;
1   if ( count < N )
1       _Accept( block );
    else
1       count = 0;
```

(b) **4 marks**

```
1   uCondition bench;
-   count += 1;
-   if ( count < N )
1       bench.wait();
    else
1       count = 0;
1   bench.signal();
```

(c) **4 marks**

```
1   AUTOMATIC_SIGNAL;
-   count += 1;
-   if ( count < N )
1       WAITUNTIL( count == 0, , );
    else
1       count = 0;
1   RETURN();
```

(d) **7 marks**

```
1   unsigned int generation = 0;
1   unsigned int mygen = generation;
-   count += 1;
-   if ( count < N )
1       while ( mygen == generation )
1           wait();
    else {
1       count = 0;
1       generation += 1;
1       signalAll();
    }
```

```
1   unsigned int tickets = 0, serving = N;
1   unsigned int myticket = tickets;
1   tickets += 1;
-   while ( myticket >= serving ) wait();

-   count += 1;
-   if ( count < N ) {
1       wait();
1       if ( count == 1 ) {
-           serving += N;
-           signalAll();
        } // if
    } else {
1       signalAll();
    } // if
1   count -= 1;
```

6. **26 marks**

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

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

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

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

1           while ( ! rafters.empty() ) {                   // mark futures closed
1               rafters.front().exception( new Closed );
1               rafters.pop_front();
            } // while

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

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