UNIVERSITY OF WATERLOO

Final Examination
Term: Winter Year: 2019

CS343
Concurrent and Parallel Programming
Sections 001
Instructor: Peter Buhr

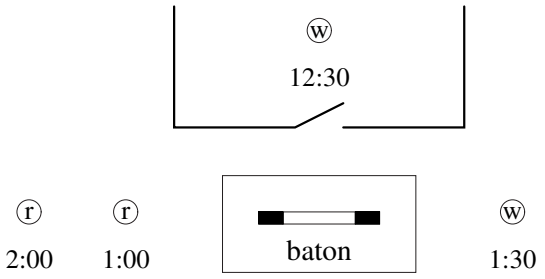
Monday, April 15, 2019
Start Time: 9:00 End Time: 11:30
Duration of Exam: 2.5 hours
Number of Exam Pages (including cover sheet): 8
Total number of questions: 7
Total marks available: 125
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

1. (a) **2 marks** Why is a buffer used in concurrent programming, e.g., producer/consumer problem, and what must be true about the execution performance of threads adding and removing elements for the buffer to be useful?

   (b) **2 marks** For a bounded buffer with multiple adding/removing threads, discuss the two kinds of locking that are required for correct behaviour.

   (c) **1 mark** What is the relationship between *baton passing* and *barging*?

   (d) **2 marks** Given the following readers/writer snapshot:



   and the 12:30 writer exits the critical section at 2:30, explain a scenario resulting in *staleness* and one resulting in *freshness*.

   (e) **2 marks** When solving the staleness/freshness problem, is *eventual progress* sufficient? If not, what is sufficient?

   (f) **2 marks** Split binary-semaphores have this pattern `Xdel += 1; entry.V(); INTERRUPTED HERE Xwait.P()`. Assume an arriving reader thread must block so it releases the entry lock (puts down the baton) but is interrupted before blocking on the reader bench. Assume a leaving writer thread acquires the entry lock (picks up the baton), sees the reader thread (`rdel > 0`), and V's the read bench. Explain why the reader does not block forever as it is not yet blocked on the read bench.

2. (a) **2 marks** What is a *race condition* and why is it difficult to locate?

   (b) **10 marks** Given a 4-way traffic intersection where cars drive on the right-hand side of the road:

   i. How many independent critical sections are there and explain what they are?

   ii. What is the maximum number of critical sections that can be used *simultaneously* without causing a problem and how does it happen?

   iii. For a single car, what is the shortest chain of *simultaneous* critical sections (holds > 1) and how does it happen?

   iv. For a single car, what is the longest chain of *simultaneous* critical sections (holds > 1) and how does it happen?

   v. What is the minimum number of critical sections that needs to be held simultaneously for a deadlock (gridlock) and how does it happen?

   (c) i. **1 mark** Is the following code fragment a potential *synchronization* or *mutual exclusion* deadlock?

```
uSemaphore L1(1), L2(1);                    // open
    task1                   task2
L1.P()                  L2.P()
    R1                      R2              // access resource
    L2.P()                  L1.P()
        R1 & R2                 R2 & R1     // access resources
```

   ii. **3 marks** Restructure the code fragment so it is deadlock-free.

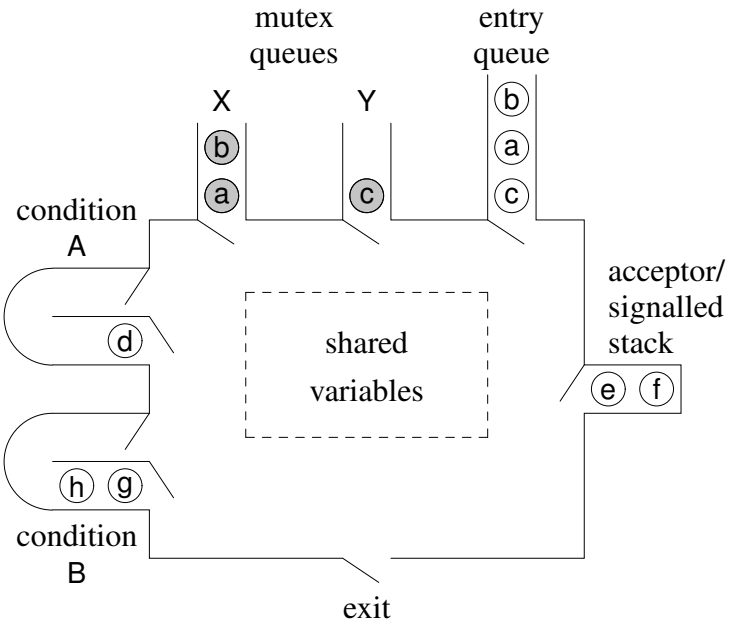
   iii. **2 marks** Why is the performance of the restructured code not as good as the original code.

(d) **7 marks** Consider a system in which there is a single resource with 11 identical units. The system uses the banker's algorithm to avoid deadlock. Suppose there are four processes P_1, P_2, P_3, P_4 with maximum resource requirements of 2, 7, 5, and 8 units, respectively. A system state is denoted by $(a_1\ a_2\ a_3\ a_4)$, where a_i is the number of resource units held by $P_i, i = 1,2,3,4$. Which of the following states are safe? Justify your answers.

i. (2 2 0 7)

ii. (2 4 1 4)

3. (a) **11 marks** Given the following snapshot of a μC++ monitor during execution:



i. The monitor is empty, which task next enters the monitor?

ii. Which tasks have called **_Mutex** members and which **_Mutex** member did each call ?

iii. Which tasks have executed a wait and on which condition variables?

iv. Why do some tasks appear twice in the diagram?

v. Explain two different scenarios that can result in tasks "e" and "f" to appear in the given order on the acceptor/signalled stack.

vi. Assume a task in the monitor does a signal on condition A, which task continues in the monitor?

vii. Assume a task in the monitor does a signal on condition A, exits the monitor, *and barging is allowed*, which tasks could next enter the monitor?

viii. Assume a task in the monitor does a signal-block on condition A, which task continues using the monitor?

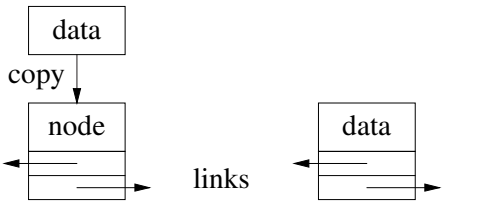
ix. Assume a task in the monitor does a signal-all (notifyall) on condition B, which task continues using the monitor and where do the other tasks go?

x. Assume a task in the monitor does an **_Accept**(X), which task continues using the monitor, and which tasks are on the acceptor/signalled stack?

(b) **2 marks** Can a *monitor* have a RendezvousFailure? Explain how it can or cannot happen.

(c) **2 marks** μC++ monitors do not have barging. How does μC++ achieve this property.

(d) **3 marks** C++ STD containers copy data into nodes and link the nodes. μC++ containers require the link fields to be part of the data, called *intrusive lists*. Name 2 advantages of intrusive lists over copying lists and one disadvantage.



4. (a) **1 mark** Explain why μC++ does not provide a type generator (language construct) composed of the execution properties: thread, *no stack*, synchronization/mutual-exclusion.

   (b) **2 marks** Write two distinct but equivalent **_Accept** statements, one in short form and one in long.

   (c) **3 marks** Java has a start member in the base task to start the task's thread, while μC++ does not have a start member. Show the μC++ pattern to simulate the Java start pattern.

   (d) **1 mark** When a task's thread ends, it turns into what kind of object?

   (e) **1 mark** Explain why you are told over and over again to move code from a task's mutex members into the task main.

   (f) **1 mark** What property makes an administrator different from a normal server?

   (g) **2 marks** What is the connection between asynchronous call and futures?

5. (a) **1 mark** Explain why modern computers have a *cache*.

   (b) **1 mark** Once there is a cache, why is *cache coherence* necessary?

   (c) **2 marks** Sequential optimization allows reordering $Wx \rightarrow Wy$ to $Wy \rightarrow Wx$. Show this reordering for the following code and explain how this reordering causes a failure scenario for the producer when creating synchronization.

```
Data = i; // W
Insert = true; // W
```

   (d) **2 marks**

      i. What optimization problem does the C++ declaration-qualifier **volatile** prevent for concurrent programming.

      ii. How does the optimization cause problems?

   (e) **2 marks** Explain how the double-wide compare-and-set instruction solves the ABA problem for stack pop.

   (f) **2 marks** In C++11 concurrency, explain the unusual connection between task creation and termination.

6. Consider the following situation involving a tour group of V tourists. The tourists arrive at the Louvre museum for a tour. However, a tour group can only be composed of G people at a time, otherwise the tourists cannot hear the guide. As well, there are 3 kinds of tours at the Louvre: pictures, statues and gift shop. Therefore, each group of tourists must vote among themselves to select the kind of tour to take. Voting is a *ranked ballot*, where each tourist ranks the 3 tours with values 0, 1, 2, where 2 is the

highest rank. Tallying the votes sums the ranks for each kind of tour and selects the highest ranking. Assume the following declarations and member routines are available to handle voting:

```
struct Ballot { unsigned int picture, statue, giftshop; };
enum TourKind { Picture = 'p', Statue = 's', GiftShop = 'g' };
unsigned int sumpic, sumstat, sumshop;              // rank sums
void reset() {                                      // reset rank sums
    sumpic = sumstat = sumshop = 0;
}
void add( Ballot & ballot ) {                       // accumulate rank
    sumpic += ballot.picture;   sumstat += ballot.statue;   sumshop += ballot.giftshop;
}
TourKind tally() {                                  // compute vote with priorities for ties
    return sumshop >= sumpic && sumshop >= sumstat ? GiftShop :
           sumpic >= sumstat ? Picture : Statue;
}
```

Do not copy this code into your answer booklet.

Figure 1 shows the interface for the vote-tallier class with the given code for the public members and private declarations (you may only add code in the designated areas). Denote the location of added code with the labels L1, L2, L3, L4, L5, L6. Depending on the implementation, some label sections are empty. **Do not write or create the tourist tasks.**

The vote routine does not return until group votes are cast. The groups are formed based on voter arrival; e.g., for a group of 3, if voters 2, 5, 8 cast their votes first, they form the first group, etc. The tour size G may not evenly divide the number of tourists, resulting in a *quorum* failure when the remaining tourists is less than G. Note, even when V *is* a multiple of G *and* tourists take multiple tours, a quorum failure can occur. When a tourist finishes taking tours and leaves the Louvre Museum, they *always* call done (even if it receives a Quorum exception). TallyVotes detects a quorum failure when the number of remaining voters is less than the group size. When a tourist calls done, they may cooperate if there is a quorum failure by helping to unblock waiting voters. At this point, any new calls to vote immediately raise exception Quorum, and any waiting voters must be unblocked so they can raise exception Quorum.

Implement a vote-tallier for G-way voting using:

(a) **8 marks** external scheduling;

(b) **6 marks** internal scheduling;

(c) **7 marks** implicit (automatic) signalling, using *only* the 3 macros defined below.

Assume the existence of the following preprocessor macros for implicit (automatic) signalling (6c):

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( cond ) ...
#define RETURN( expr... ) ...      // gcc variable number of parameters
```

Macro AUTOMATIC_SIGNAL is placed only once in an automatic-signal monitor as a private member, and contains any private variables needed to implement the automatic-signal monitor. Macro WAITUNTIL is used to delay until the cond evaluates to true. Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value.

```
_Monitor TallyVotes {
    unsigned int voters;                                    // voter size
    const unsigned int group;                               // group size
    TourKind talliedResult;                                 // vote result
    unsigned int numVotes = 0;                              // number of voted tasks
    // L1: ANY VARIABLES NEEDED FOR EACH IMPLEMENTATION
  public:
    _Event Quorum {};
    TallyVotes( unsigned int voters, unsigned int group )
            : voters( voters ), group( group ) {
        reset();                                            // reset for next vote
    }
    TourKind vote( Ballot ballot );
        // L2: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        numVotes += 1;
        add( ballot );
        if ( numVotes < group ) {                           // all but last voter
            // L3: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        } else {                                            // last group voter
            talliedResult = tally();                        // compute vote from ballot sums
            reset();                                        // reset for next vote
            numVotes = 0;
            // L4: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        }
        // L5: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        return talliedResult;                               // POSSIBLY UPDATE
    }
    void done() {
        voters -= 1;
        // L6: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
    }
}
```

Figure 1: Tally Votes **(Do not copy this code into your answer booklet.)**

7. **29 marks** Consider the tally vote as described in question 6, and assume the declarations and member routines to handle voting are directly available. Write an administrator task to coordinate voting to form a tour group and match a tour group with a guide. Figure 2 shows the interface for the vote-tallier with the given code for the public members and private declarations (you may NOT add any private variables or public routines; **private helper routines are allowed**).

   At creation, a vote-tallier is passed the number of voters, size of a voting group, *and number of guides*. The vote-tallying members are:

   vote: is called by a Tourist task passing a ranked ballot. A Vote work request is created and pushed at the end the votes vector. The vote member returns immediately with a future tour so the tourist can go to the "restroom" before the tour starts. The kind of tour will be delivered to all the future tours after a group has formed and a guide is available. If there is a quorum failure, a Quorum exception is delivered in the future tour.

   done: is *always* called by a Tourist task when the tourist finishes taking tours and leaves the Museum (even if it receives a Quorum or Closed exception).

   tour: is called by a Guide task to indicate it is ready to take a group on a tour. Each guide blocks until a complete group is available and is returned the kind of tour for a group.

```
_Task TallyVotes {
  public:
    _Event Quorum {};                                   // Quorum failure
    _Event Closed {};                                   // Museum closed
    typedef Future_ISM<TourKind> Ftour;                 // future tour

    TallyVotes( unsigned int voters, unsigned int group, unsigned int guides ) :
            voters( voters ), group( group ), guides( guides ) {
        reset();                                        // reset for next vote
    }
    Ftour vote( Ballot ballot ) {
        Vote * v = new Vote( ballot );                  // create work request
        votes.push_back( v );                           // add to votes list
        return v->ftour;                                // return future tour
    }
    void done() {}                                      // tourist leaves museum

    TourKind tour() {
        wguides.wait();                                 // block until group available
        if ( closed ) _Throw Closed();                  // tell guide closed
        return gtour;                                   // tour kind
    }
  private:
    struct Vote {                                       // work request
        Ballot ballot;
        Ftour ftour;
        Vote( Ballot ballot ) : ballot( ballot ) {}
    };
    vector<Vote *> votes;                               // client requests for vote
    unsigned int voters;
    const unsigned int group, guides;
    uCondition wguides;                                 // bench for waiting guides
    // communication
    bool closed = false;
    TourKind gtour;

    void main();                                        // YOU WRITE THIS MEMBER
};
```

Figure 2: Tally Votes Administrator **(Do not copy this code into your answer booklet.)**

When the vote-tallier's destructor is invoked it closes for the day and delivers exception Closed into all outstanding future tours and informs all waiting and returning guides of the closure by raising exception Closed in member tour.

Ensure the TallyVotes task does as much administration works as possible; a monitor-style solution will receive little or no marks. Assume the program main creates the TallyVotes, Tourist and Guide tasks. Write only the TallyVotes::main member, **do not write the program main or the tourists and guide tasks**.

μC++ future server operations are:

- delivery(T result) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.
- exception(uBaseEvent * cause) – copy a server-generated exception into the future, and the exception cause is thrown at clients accessing the future.

C++ vector operations are:

bool empty()	size() == 0
int size()	vector size
T & front()	first element
T & back()	last element
T & **operator**[ ](size_type pos)	subscript element
void push_back(**const** T & x)	add element x after last element
void pop_back()	remove last element
void clear()	erase all elements
iterator erase(iterator first, iterator last)	erase elements from first to last
iterator begin()	iterator for first element
iterator end()	iterator for last element