UNIVERSITY OF WATERLOO

Final Examination
Term: Fall Year: 2018

CS343
Concurrent and Parallel Programming
Sections 001, 002
Instructor: Peter Buhr

Monday, December 17, 2018
Start Time: 12:30 End Time: 15:00
Duration of Exam: 2.5 hours
Number of Exam Pages (including cover sheet): 6
Total number of questions: 7
Total marks available: 112
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

1. (a) i. **5 marks** The fetch-and-increment instruction atomically reads and increments a memory location. The instruction has one parameter: a reference to a value. The action of this single atomic instruction is the same as the following C routine if it could be performed as a single atomic operation without interruption with respect to the given parameter:

```
int fetchInc( int &val ) {      // atomic execution
    int temp = val;             // read
    val += 1;                   // increment and write
    return temp;                // return previous value
}
```

Show how this atomic instruction can be used to build mutual exclusion by completing a class with the following public interface (you may only add a public destructor and private members):

```
class Lock {
  public:
    Lock();
    void acquire();
    void release();
};
```

which is used as follows:

```
Lock l;
l.acquire();
// critical section
l.release();
```

Your solution may use busy waiting, must deal with starvation, and **you may not assume assignment is atomic**.

ii. **1 mark** Is there any limitation (failure scenario) in your solution?

(b) i. **3 marks** For the *baton-passing* locking-technique, list the baton passing rules.

ii. **1 mark** How many bytes of storage does it take to implement a baton? Explain.

(c) i. **2 marks** What is the potential problem with this coding sequence, and what is the name of the problem?

```
... entry.V(); Xwait.P(); ...
```

ii. **2 marks** Declare a semaphore member to fix this problem and explain how it would work?

(d) **2 marks** Assume a solution to the readers/writer problem handling temporal order using a single FIFO bench for the readers/writers and a chair. Briefly, explain how the overall solution works.

2. (a) **5 marks** List the five conditions for a mutual-exclusion deadlock?

(b) **2 marks** Are the weeping angels around the Tardis in a livelock or deadlock? Explain.

(c) **2 marks** Explain the *basic* idea behind the *ordered-resource* allocation policy to prevent deadlock (be brief), and name the mutual-exclusion deadlock-condition it prevents.

(d) **2 marks** Suggest what a programmer should and should not do if the deadlock-avoidance Oracle says NO to a resource allocation.

3. (a) **2 marks** Explain why a solution for a bounded buffer with multiple producers/consumers can be more efficient using semaphores rather than a monitor.

   (b) **3 marks** Explain how the three parts of the conditional-critical region are incorporated into the object-oriented model.

   (c) **1 mark** Why is external scheduling easier than internal scheduling?

   (d) **2 marks** Write the code for an OR accept and an AND accept for mutex members M1 and M2.

   (e) **2 marks** Explain the difference between μC++ signal and signalBlock.

   (f) **3 marks** What is the *nested-monitor problem* and what concurrent error can it generate?

   (g) **2 marks** When monitors are classified using C (calling), W (signalled), and S (signaller) priorities, explain why monitors are rejected for W = S?

   (h) **2 marks** Excluding spurious wakeup, name two aspects of a Java monitor that makes it difficult to use. Do not describe them, i.e., very short answers.

   (i) **1 bonus mark** Do you believe in spurious wakeup? (yes or no)

4. (a) **2 marks** Explain why μC++ does not provide a type generator (language construct) composed of the execution properties: thread, stack, no synchronization/mutual-exclusion.

   (b) **4 marks** Write a code fragment that generates a RendezvousFailure and show how to deal with the failure.

   (c) **2 marks** When a server accepts a member routine and the client waits (blocks) in the member, what happens to the rendezvous and how does the server deal with it?

   (d) i. **2 marks** Explain why accepting a task's destructor should not work.

   ii. **2 marks** Explain how μC++ makes it work.

   (e) **2 marks** After *cancelling* a future, what happens when the future is accessed and why are aspects of cancellation difficult to deal with?

5. (a) **2 marks** Virtual memory increases performance between hardware components _____ and _____, while caching increases performance between _____ and _____.

   (b) **2 marks** Explain the concurrent *caching* problem that might occur in the following code:

```
thread 1                thread 2
int *x = new int        int *y = new int;
```

   (c) **2 marks** Sequential optimization allows reordering $Rx \rightarrow Wy$ to $Wy \rightarrow Rx$. Show how this reordering is possible when creating synchronization and explain the failure scenario.

```
while ( ! Insert );     // R
Insert = false;
data = Data;            // W
```

   (d) **2 marks** A GPU is a coprocessor of the CPU. Why does this architecture cause a bottleneck in processing data by the GPU?

   (e) **3 marks** Explain how the Ada requeue statement may be used to simulate internal scheduling using external scheduling.

   (f) **2 marks** In the Go programming language, threads are anonymous, precluding direct communication. Explain the alternative communication mechanism used by Go threads.

   (g) **1 mark** Are Java *executors* an implicit or explicit concurrency system?

6. A *counting semaphore* lock performs synchronization and mutual exclusion. Using a μC++ monitor, write a counting semaphore implementing the P, tryP, P(s), and V semaphore operations using:

   (a) **8 marks** external scheduling

   (b) **7 marks** internal scheduling

   The semaphore class has the following interface (you may add a public destructor and private members):

```
_Monitor Semaphore {
    unsigned int cnt;              // semaphore counter
    // ANY ADDITIONAL VARIABLES NEEDED FOR EACH IMPLEMENTATION
  public:
    Semaphore( unsigned int cnt ) : cnt( cnt ) { // initialize semaphore counter
        // ANY ADDITIONAL INITIALIZATION NEEDED FOR EACH IMPLEMENTATION
    }
    void P();                      // YOU WRITE 2 VERSIONS OF THIS MEMBER
    bool tryP();                   // YOU WRITE 2 VERSIONS OF THIS MEMBER
    void P( Semaphore s );         // YOU WRITE 2 VERSIONS OF THIS MEMBER
    void V();                      // YOU WRITE 2 VERSIONS OF THIS MEMBER
}
```

7. **25 marks** Write an administrator task for the *Maple Leaf Taxi* company. At Maple Leaf Taxi, there are N taxis located throughout the city. The dispatcher's job is to take requests from clients for a taxi and to dispatch the closest taxi to the client for a pickup. The dispatcher also takes requests from taxis for the next client at the start of the day and after each client is delivered to their destination. Figure 1 contains the starting code for the dispatcher administrator task (you may not change this interface; you may add private members in the task). **(Do not copy the starting code into your answer booklet.)**

   A client calls the getTaxi routine to ask for a taxi to pick them up at an address given by parameter coordinates (x,y). A *future* taxi is returned immediate to the client, so the client can execute asynchronously (e.g., get ready to leave) before accessing the taxi. When the client accesses the future taxi, they may block because the taxi is not there; otherwise, the client gets into the taxi.

   A taxi calls the getClient routine to indicate they are available and tell the dispatcher the taxi's current (x,y) location. The taxi is dispatched to a client if there is an outstanding request from a client; otherwise, the taxi blocks until a client request is made. The taxi's call to getClient returns with the (x,y) arguments changed to the next client's location.

   The dispatcher creates a pool of 5 taxis, and dispatches the nearest taxi to the client's address to minimize client waiting time, if a taxi is available. The taxi constructor is

```
Taxi( MapleLeafTaxi & employer, int id );
```

   Routine nearestTaxi is used by the dispatcher to find the nearest available taxi address to the given client address. (Do not write nearestTaxi; just use the member interface in Figure 1.)

   When the dispatcher's close routine is called at the end of the day, it prints out the message "Closed for the day", and stops accepting calls to getTaxi. Then, the dispatcher must deal with outstanding futures to clients that cannot be serviced (no cab will come to service their request), and it must wait for all the taxis to check in before telling them to go home so they can be deleted. Any client with an outstanding taxi-future has the exception Closed inserted into the future, so the client gets this exception raised when it accesses the future. Any waiting or arriving taxi has the exception Closed raised on its stack. The dispatcher must delete any allocated storage before terminating.

Ensure the dispatcher task does as much administration works as possible; a monitor-style solution will receive little or no marks. Write the code for MapleLeafTaxi::main and any necessary declarations/initializations; do **NOT** write the client, taxi, or program main. Assume the program main creates and deletes all the necessary tasks, appropriately, and calls the dispatcher's close routine.

```
_Task MapleLeafTaxi {
  public:
    _Event Closed {};                                   // indicate MapleLeafTaxi closed
    typedef Future_ISM<int> Ftaxi;                      // future taxi
  private:
    struct LocnClient {
        int id, x, y;                                   // client location coordinates
        Ftaxi ftaxi;
        LocnClient( int id, int x, int y ) : id( id ), x( x ), y( y ) {}
    };
    struct LocnTaxi {
        int id, x, y;                                   // taxi location coordinates
        uCondition idle;
        LocnTaxi( int id, int x, int y ) : id( id ), x( x ), y( y ) {}
    };
    enum { NoOfTaxi = 5 };
    list<LocnClient *> clients;                         // client requests for taxis
    list<LocnTaxi *> taxis;                             // waiting taxis
    int xclient, yclient;                               // communication variables
    bool closed = false;
  public:
    Ftaxi getTaxi( int id, int x, int y ) {             // called by client
        LocnClient *client = new LocnClient(id, x, y);  // create work request
        clients.push_back( client );                    // add to request list
        return client->ftaxi;                           // return future request
    }
    void getClient( int id, int & x, int & y ) {        // called by taxi, x,y in/out parameters
        LocnTaxi taxi( id, x, y );                      // use the stack!
        taxis.push_back( &taxi );                       // add to waiting taxi list
        taxi.idle.wait();                               // taxi always blocks
        if ( closed ) _Throw Closed();
        x = xclient; y = yclient;                       // taxi returns client info
    }
    void close() {}                                     // called at closing time
  private:
    list<LocnTaxi *>::iterator nearestTaxi( LocnClient * node, list<LocnTaxi *> & alist ) {
        // Find the element in parameter alist whose address is closest
        // to the address in parameter node. "alist" must be non-empty.
        // ASSUME THIS ROUTINE IS WRITTEN; DO NOT WRITE IT.
    }
    void main() {                                       // YOU WRITE ONLY THIS ROUTINE!!!!
        // allocate taxi tasks
        // dispatch taxis to clients, until close called
        // print closed message
        // mark outstanding futures with Closed exception
        // tell each taxi to go home
        // delete taxi tasks
    }
};
```

Figure 1: Maple Leaf Taxi Administrator

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++ list operations are:

int size()	list size
bool empty()	size() == 0
T & front()	first element
T & back()	last element
void push_front(**const** T & x)	add x before first element
void push_back(**const** T & x)	add x after last element
void pop_front()	remove first element
void pop_back()	remove last element
void clear()	erase all elements
iterator erase(iterator position)	erase element from list