UNIVERSITY OF WATERLOO

Final Examination
Term: Fall Year: 2015

CS343
Concurrent and Parallel Programming
Sections: 001, 002, 003
Instructors: Peter Buhr and Ashif Harji

Friday, December 18, 2015
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: 100
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

1. (a) **2 marks** Explain the *staleness* and *freshness* problems.
   (b) **3 marks** Can mutex/condition locks perform baton passing to prevent barging?
   (c) **1 mark** In *baton passing*, what value is assigned to the baton to indicate the baton is being passed?
   (d) **1 mark** What order should multiple threads acquire resources to prevent deadlock?
   (e) **1 mark** Describe the case where a cycle in a process graph does *not* imply a deadlock?
   (f) **2 marks** Once a deadlock has occurred, the only mechanism to recover is preemption. Explain why *preemption* is difficult.

2. (a) **6 marks** Solve the producer and consumer problem with a shared bounded buffer of integers using the shared declaration qualifier and the conditional critical-region, e.g.:

```
shared stack<int,10> s; // example syntax of critical region, not part of the solution
region s {
  await s.size() > 10;
    // access s
}
```

   Write only the declaration of the shared bounded buffer using any appropriate STL data-structure, and the mutual exclusion and synchronization code using the conditional critical-region. Include a comment indicating what buffer operation (insert/remove) is performed. Assume the size of the buffer is 10 items. Do not write the producer/consumer tasks.
   (b) **2 marks** Explain the difference between μC++ signal and signalBlock statements.
   (c) **4 marks** In the external scheduling version of the readers/writer problem, explain how the synchronization works in routine startWrite:

```
void startWrite() {
    if ( wcnt > 0 ) Accept( endWrite );
    else while ( rcnt > 0 ) Accept( endRead );
    wcnt = 1;
}
```

   (d) **2 marks** Explain why automatic signal monitors can be simulated without busy waiting?
   (e) **2 marks** Explain what it means when a task *barges* with respect to a monitor.
   (f) **1 mark** What is *spurious wakeup*?

3. (a) **2 marks** Explain why the long form of the **_Accept** statement is necessary:

short form	long form
_Accept(m1, m2) S1	Accept(m1) S1; **or** Accept(m2) S1;

   (b) **2 marks** Why is it important to have as little code as possible in the mutex members of a server task, and where should the code go?
   (c) **2 marks** Why is it necessary to use a flag variable when an administrator server needs to return an exception for a client call?
   (d) **2 marks** Why is it useful for a task to accept its destructor and what is unusual about accepting the destructor?
   (e) **1 mark** What is the general approach used to increase client-side concurrency?
   (f) **2 marks** What is the advantage of using a future to increase client-side concurrency?
   (g) **2 marks** Explain the purpose of the **_Select** statement. Show an example that requires a **_Select** statement.

4. (a) **3 marks** What is *false sharing*? What problem does it cause? Explain how to prevent it.
   (b) **2 marks** What aspect of cache consistency results in stale reads?
   (c) **1 mark** What optimization problem does the C++ declaration-qualifier **volatile** prevent for concurrent programming?
   (d) **1 mark** If a programming language does not have a memory model, can concurrency be implemented with a library approach. (Answer *yes* or *no*, do not explain.)
   (e) **2 marks** Give an advantage and disadvantage for lock-free programming.
   (f) **2 marks** Is the Ada requeue mechanism as powerful as internal scheduling? Explain.
   (g) **2 marks** The programming language Go does not support direct communication using an **_Accept** statement. Explain the similar high-level approach and construct Go uses for interaction among its threads and how direct communication is accomplished.

5. (a) **2 marks** What does an operating-system *address-space* have to do with distributed programming?
   (b) **1 mark** Why is there a problem passing *integer* values between different computers?
   (c) **2 marks** Message Passing Interface (MPI) uses an interesting trick with operations like Bcast, Scatter, Gather to make them work when multiple threads execute exactly the same code. Briefly explain this trick.
   (d) **2 marks** Where is message passing used in remote procedure call?

6. A *counting semaphore* lock performs synchronization and mutual exclusion.
   (a) Write a counting semaphore using μC++ monitors that implements the semaphore lock using:
      i) **3 marks** external scheduling;
      ii) **3 marks** internal scheduling;
      iii) **6 marks** internal scheduling with barging, using *only* routines wait() and signalAll() defined below;
      iv) **3 marks** implicit (automatic) signalling.
   (b) **2 marks** Does your solution in 6(a)ii, internal scheduling, work if the signal is changed to signalBlock? Explain, why it does or does not work.

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

```
_Monitor Semaphore {
    unsigned int counter;                          // semaphore counter
    // ANY ADDITIONAL VARIABLES NEEDED FOR EACH IMPLEMENTATION
  public:
    Semaphore( unsigned int counter ) : counter( counter ) {
        // ANY ADDITIONAL INITIALIZATION NEEDED FOR EACH IMPLEMENTATION
    }
    void P() {
        // YOU WRITE 4 VERSIONS OF THIS MEMBER
    }
    void V() {
        // YOU WRITE 4 VERSIONS OF THIS MEMBER
    }
}
```

   The starting value of the semaphore counter is passed to the constructor. **Do not write or create the client tasks.**

Assume the existence of the following routines for internal scheduling with barging (6(a)iii):

```
uCondition bench;
void wait() {
    bench.wait();                                   // wait until signalled
    while ( rand() % 5 == 0 ) {                     // multiple bargers allowed
        _Accept( P ) {                              // accept barging callers
        } _Else {                                   // do not wait if no callers
        } // Accept
    } // while
}
void signalAll() {
    while ( ! bench.empty() ) bench.signal();       // drain the condition
}
```

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

```
#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.

7. **23 marks** Write an administrator task to perform the job of a central coordinator for a shuttle service. A *waiting* shuttle makes a run as soon as it is either full, or a 15 minute window expires and there is at least one client waiting. Hence, if there is no shuttle waiting, a timer signal is lost. Figure 1 contains the starting code for the Coordinator administrator (you may add only a public destructor and private members). **(Do not copy the starting code into your exam booklet.)** The coordinator members are as follows:

   timeUp: is called by a Timer task every 15 minutes. If there is at least one client waiting and one shuttle waiting, the coordinator sends the shuttle off to deliver the client(s). The member returns **true** if the coordinator is in the process of shutting down.

   checkIn: is called by a Shuttle to indicate it is ready to make a run. A shuttle waits until it can be filled or a 15 minute window expires and there is at least one client waiting. The member returns **true** if the coordinator is in the process of shutting down.

   getRide: is called by a Client to indicate they want a ride. A client is immediately returned a future so the client can shop until a shuttle can be filled, or a shuttle is available and its 15 minute window has expired. The future contains the identity of the Shuttle that is giving the Client a ride or the exception Closed when the coordinator is in the process of shutting down.

   When the coordinator's destructor is invoked, it shuts down for the day and notifies timer, shuttle and client tasks. Ensure the Coordinator task does as much administration works as possible; a monitor-style solution will receive little or no marks. Write only the Coordinator task, **do not write uMain::main or the timer, client and shuttle tasks**. Assume uMain::main creates the Coordinator, Timer, Shuttle and Client tasks, and deletes them at the end of the day.

   For your reference, 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.

```
_Task Coordinator {
  public:
    typedef Future_ISM<unsigned int> Fshuttle; // future shuttle identifier
    _Event Closed {};                          // too late for a ride
  private:
    const unsigned int ShuttleSize, NumShuttles, NumClients;
    uCondition shuttles;                       // shuttles wait here
    unsigned int shuttleId;                    // current shuttle leaving
    bool shuttingDown;                         // returns false if not currently shutting down
    unsigned int numClientsWaiting;
    list< Fshuttle > clients;                  // client futures
    ...                              // YOU WRITE DECLARATIONS AND HELPER ROUTINES
  public:
    Coordinator(const unsigned int ShuttleSize, // maximum number of people on shuttle
                const unsigned int NumShuttles, // number of shuttles in the fleet
                const unsigned int NumClients): // number of people using the shuttles
            ShuttleSize( ShuttleSize ), NumShuttles( NumShuttles ), NumClients( NumClients ) {
        numClientsWaiting = 0;  shuttingDown = false;
    }
    bool timeUp() { return shuttingDown; }
    bool checkIn( unsigned int id ) {
        shuttles.wait();                       // wait for clients
        shuttleId = id;                        // indicate leaving
        return shuttingDown;
    }
    Fshuttle getRide( unsigned int id ) {
        Fshuttle fshuttle;
        numClientsWaiting += 1;
        clients.push_back( fshuttle );         // store future
        return fshuttle;
    }
  private:
    void main() {                    // YOU WRITE THIS ROUTINE
        // coordinate timer, shuttles, and clients
        // tell clients, shuttles, and timer to shutdown
    }
};
```

Figure 1: Coordinator Administrator

- exception(uBaseEvent *cause) - copy a server-generated exception into the future, and the exception cause is thrown at clients accessing the future.

Also for your reference, 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