UNIVERSITY OF WATERLOO

# Final Examination
## Term: Fall Year: 2016

**CS343**
**Concurrent and Parallel Programming**
**Sections 001, 002, 003, 004**
**Instructors: Peter Buhr and Aaron Moss**

**Saturday, December 17, 2016**
**Start Time: 9:00 End Time: 11:30**
**Duration of Exam: 2.5 hours**
**Number of Exam Pages (including cover sheet): 7**
**Total number of questions: 7**
**Total marks available: 127**
**CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED**

1. (a) **2 marks** Explain why the Angels in the Dr. Who video are in a livelock.

   (b) **2 marks** With respect to the readers/writer problem, what is a *shadow queue* and what problem is it solving.

   (c) The solutions of the readers/writer entry-protocols have the following code sequence:

   ```
   ... entry_q.V(); INTERRUPTED HERE X_q.P();
   ```

   i. **1 mark** What problem can occur because of the interruption?

   ii. **2 marks** Explain one possible solution to this problem.

   (d) **1 mark** What problem is the split-binary semaphore trying to solve.

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

   (f) 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.

2. **12 marks** Using a *counting semaphore* to its fullest (Zen), construct a solution to the bounded-buffer problem using the following outline allowing *multiple* producer and consumer tasks:

```
// global semaphore declarations used by producer and consumer

void Producer::main() {
    for ( ;; ) {
        // produce an item
        // add element to buffer
    }
    // produce a stopping value
}
```

```
void Consumer::main() {
    for ( ;; ) {
        // remove element from buffer
      if ( stopping value ? ) break
        // consume item
    }
}
```

   Write the global semaphore declarations shared by the producer and consumer tasks, and any synchronization and/or mutual exclusion necessary among the comments in the **for** loops inside the main routines of Producer and Consumer. Your solution must ensure the buffer does not overflow or underflow, and it must handle multiple producer and consumer tasks with maximum concurrency. Do not write any buffer declarations or code to manipulate it; just use the supplied comments to indicate where you want buffer operations to occur. Assume items can be copied into and out of the buffer, insertion and removal can occur concurrently, and the size of the buffer is 10 items.

3. (a) **5 marks** Show how to make the following class behave like a monitor using a MutexLock.

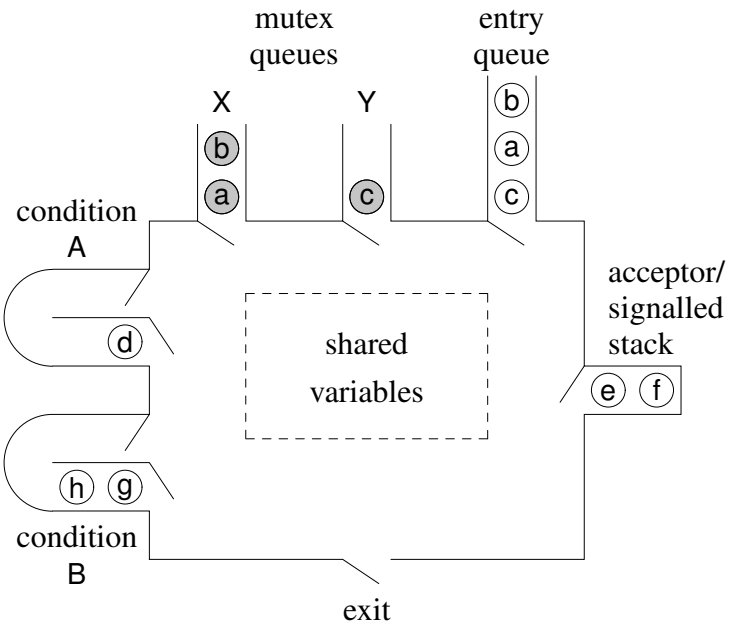
```
class M {
    int v;
  public:
    int x(...) {
        ... // compute v
        return v;
    }
};
```

(b) **2 marks** Explain monitor *internal* and *external* scheduling.

(c) **2 marks** Does a monitor condition-lock (uCondition) use *external* or *internal* locking? Explain your answer.

(d) **2 marks** When can barging produce a performance gain? What must be true for barging to be safe?

(e) **9 marks** Given the following snapshot of a $\mu$C++ 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. Explain two different scenarios that can result in tasks "e" and "f" to appear in the given order on the acceptor/signalled stack.

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

vi. 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?

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

viii. 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?

ix. 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?

4. (a) **3 marks**

i. When the main routine of a coroutine returns, the coroutine changes into a *put answer in booklet* .

ii. When the main routine of a coroutine-monitor returns, the coroutine-monitor changes into a *put answer in booklet* .

iii. When the main routine of a task returns, the task changes into a *put answer in booklet* .

(b) **2 marks** Given the following **_Select** statement:

```
Select( f1 ) S1;
or Select( f2 ) S2;
and Select( f3 ) S3;
```

and the futures become available in the order f2, f1 and f3. Which statements S1, S2, or S3 are executed and in what order?

(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** Java tasks must be explicitly started. Show how to accomplish the same behaviour in a $\mu$C++ task.

(e) **2 marks** What object-oriented feature does Java provide by explicitly starting a thread object? Explain.

(f) **2 marks** What is the purpose of a *courier worker* and how does it work?

(g) **2 marks** Why is it difficult to return a value from an asynchronous call?

5. (a) **3 marks** What is *false sharing*? What problem does it cause? Explain how to prevent it.

   (b) i. **5 marks** Show the code for the double-check locking singleton-pattern.

   ii. **2 marks** Why is it possibly unsafe?

   (c) **3 marks** The TSO memory-model allows disjoint reads to be moved before writes. Show a coding example that illustrates this optimization and explain how it can cause a concurrent program to fail.

   (d) **2 marks** Why does the MIPS LL and SC instructions not have the ABA problem like the CAA instruction?

   (e) **3 marks** Can an Ada monitor implement the dating-service problem. If not, why not.

   (f) **2 marks** How does the Linda *tuple space* provide concurrency and communication? (Do not describe the specific Linda operations.)

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 can only be composed of $G$ people at a time, otherwise the tourists cannot hear what the guide is saying. As well, there are 2 kinds of tours available at the Louvre: picture and statue art. Therefore, each group of $G$ tourists must vote amongst themselves to select the kind of tour they want to take. During voting, tasks must block until all votes are cast, i.e., assume a secret ballot. Once a decision is made, the tourists in that group proceed on the tour.

   To simplify the problem, the program only has to handle cases where the number of tourists in a group is odd, precluding a tie vote, and the total number of tourists is evenly divisible by the tour-group size so all groups contain the same number of tourists.

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

   (a) **1 mark** external scheduling;

   (b) **3 marks** internal scheduling;

   (c) **11 marks** internal scheduling with barging, using *only* routines wait() and signalAll() defined below;

   (d) **6 marks** implicit (automatic) signalling, using *only* the macros defined below.

The tally voters class has the following interface and code structure (you may not add any new member routines):

```
_Monitor TallyVotes {
    enum Tour { Picture, Statue };                      // tour kind (0 or 1)
    const unsigned int group;                           // group size
    unsigned int sum = 0, numVotes = 0;                 // counters
    Tour talliedResult;                                 // vote result
    // L1: ANY VARIABLES NEEDED FOR EACH IMPLEMENTATION
  public:
    Tour vote( unsigned int id, Tour ballot ) {
        // L2: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        sum += ballot;                                  // sum 0 or 1
        numVotes += 1;
        if ( numVotes < group ) {                       // all but last group voter
            // L3: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
            numVotes -= 1;
        } else {                                        // last group voter
            // L4: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
            talliedResult = (Tour)(sum > group / 2);    // compute vote result
            numVotes -= 1;
            sum = 0;                                    // reset for next vote
        }
        // L5: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        return talliedResult;
    }
}
```

Denote the location of your added code with the labels L1, L2, L3, L4, L5. Depending on the implementation, some label sections are empty. **Do not write or create the voter tasks.**

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

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

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

```
#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.

```
_Task Maid;                                          // forward declaration
_Task DollyMaid {
  public:
    struct Maids { Maid *maids[2]; };                // maid pair
    typedef Future_ISM<Maids> FMaids;                // future maid
    _Event Closed {};                                // DollyMaid closed
  private:
    enum { MaxHires = 50 };                          // maximum hires er day
    const unsigned int NoOfMaids, NoOfClients;
    uCondition unhired;                              // maids wait here
    list<FMaids> clients;                            // clients waiting for maid
    unsigned int money;                              // communication variable
    bool closed = false;                             // DollyMaid closed
  public:
    DollyMaid( const unsigned int NoOfMaids, const unsigned int NoOfClients ) :
        NoOfMaids( NoOfMaids ), NoOfClients( NoOfClients ) {}
    FMaids hire() {                                  // called by client
        FMaids mf;                                   // maid future
        clients.push_back( mf );                     // store maid future
        return mf;                                   // return maid future to client
    }
    void checkIn( unsigned int money ) {             // called by maid
        DollyMaid::money = money;                    // transfer money to administrator
        unhired.wait( (uintptr_t)&uThisTask() );     // wait for next job with task id
        if ( closed ) _Throw Closed();               // maid goes home
    }
  private:
    void main();      // YOU WRITE THIS ROUTINE
};
```

Figure 1: Dolly Maid Administrator

7. **25 marks** Write an administrator task to manage the *Dolly Maid* (DM) cleaning-service company. Clients contact DM with requests to hire maids (male or female) for cleaning jobs. When two maids (for efficiency and safety) are available, they are sent out to perform each cleaning job. Maids may pair up with any waiting maids. After the cleaning is completed, each maid is paid separately by the client, and the maids return to DM with the money and wait for the next cleaning job. DM accumulates the money from the maids after they return for more work. DM is restricted to N hires from clients per day, after which it closes for the day, and prints out the total earned.

   Figure 1 contains the starting code for the Dolly Maid administrator (you may add only a public destructor and private members). **(Do not copy the starting code into your exam booklet.)** The administrator's members are as follows:

   **hire:** is called by clients to get a pair of maid for cleaning duties. A future maid-pair is immediately returned to the client so they do not have to wait for the maids to arrive and form a pair. The future contains the address of the two maids or the exception Closed when DM is in the process of closing down.

   **checkIn:** is called by a maid to return money from a client for cleaning and to get a new client. The first call by a maid passes a value of zero for the money.

   After MaxHires (N) calls to member hire, DM closes for the day and must send futures containing the Closed exception to each client and raise the Closed exception to each maid.

Ensure the administrator task does as much administration works as possible; a monitor-style solution will receive little or no marks. Write only the code for DollyMaid::main based on the given outline, **do not write a client or maid or uMain::main**. Assume uMain::main creates the DollyMaid, Maid, and Client tasks, and deletes them at the end of the day.

For your reference, $\mu$C++ 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.

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 |