Final Examination
Term: Winter Year: 2017

CS343
Concurrent and Parallel Programming
Sections 001
Instructor: Peter Buhr

Thursday, April 20, 2017
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: 6
Total marks available: 104
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED
1. (a) **7 marks** The following is an implementation of a binary semaphore. Indicate where the *conceptual baton* is picked up, put down, passed and received. Use the line numbers on the left to indicate where the baton actions occur.

```cpp
class BinSem {
    queue<Task> blocked;
    bool inUse;
    SpinLock lock;

public:
    BinSem( bool usage = false ) : inUse( usage ) {} 
    void P() {
        lock.acquire();
        if ( inUse ) {
            // add self to lock's blocked list
            yieldNoSchedule( lock ); // atomically block and release lock
        }
        inUse = true;
        lock.release();
    }
    void V() {
        lock.acquire();
        if ( ! blocked.empty() ) {
            // remove task from blocked list and make ready
        } else {
            inUse = false;
            lock.release();
        }
    }
};
```

(b) **3 marks** Explain the difference between barging *avoidance* and *prevention*. Does question 1a use avoidance or prevention?

(c) **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*.

(d) **2 marks** Given this 4-way stop, where for simultaneous arrival, the person on the right has the right-of-way (otherwise, the first arriver has the right-of-way), explain how starvation can occur.
(e) 6 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, 5, 8, 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. $(1, 2, 4, 2)$
   ii. $(1, 1, 5, 3)$

(f) 2 marks For deadlock detection-and-recovery, give two reasons why preemption is difficult?

2. (a) 2 marks Explain the two aspects of the dating-service problem preventing a straightforward implementation by external scheduling.
   
   (b) 2 marks Explain two situations where a `public _Nomutex` member is useful and give an example of each.

   (c) 5 marks Using $\mu$C++, write the shortest possible external-scheduling monitor using the following interface that implements a counting semaphore (you may add code anywhere).

   ```
   _Monitor semaphore {
   public:
   semaphore( int cnt = 1 );
   void P();
   void V();
   }
   ```

   (d) 2 marks Explain why automatic-signal monitors are easier to use than explicit-signal monitors but more expensive in terms of execution time.

   (e) 1 mark Independent of any performance benefit, why do many concurrent locks and monitors allow barging?

   (f) 2 marks Explain why it is impossible to construct a condition lock using a separate monitor.

3. (a) 2 marks What purpose does the `When` clause provide on an `Accept` clause?
   
   (b) 2 marks Explain why the `When` clause cannot be easily replaced by the `if` statement.

   (c) 2 marks On assignment 6, is the vending machine a `server` or an `administrator`? Justify your answer.

   (d) 3 marks A future can have 3 outcomes. Name each kind of outcome.

   (e) 2 marks What is cache thrashing?

   (f) 2 marks What does the C/C++ qualifier `volatile` do, and give an example where it prevents a problem in a concurrent program.

4. (a) 2 marks Why are operations on lock-free data-structures deadlock-free?
   
   (b) 1 mark Which aspect of mutual exclusion do lock-free data-structures violate?

   (c) 4 marks When pushing a node on a lock-free stack there is problem. Give the name of the problem and describe it.

   (d) 1 mark State the main hardware architectural difference between CPUs and GPUs that makes GPUs difficult to program.

   (e) 2 marks In the programming language Go, name the mechanism used for thread communication and explain how the communication mechanism is implemented.
5. A barrier lock performs synchronization on a group of \(N\) threads so they all proceed at the same time. A barrier is accessed by any number of threads. The barrier makes the first \(N - 1\) threads wait until the \(N\)th thread arrives at the barrier and then all \(N\) threads continue. After a group of \(N\) threads continue, the barrier resets and begins synchronization for the next group of \(N\) threads. A barrier is used in the following way by client tasks:

```cpp
Barrier b; // global declaration or passed to the client
... b.block(); // each client synchronizes, possibly multiple times
```

Write a barrier using \(\mu\)C++ monitors that implements a barrier lock using:

(a) 4 marks external scheduling,
(b) 4 marks internal scheduling,
(c) 4 marks implicit (automatic) signalling,
(d) 7 marks internal scheduling with barging.

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

```cpp
_Monitor Barrier {
    const unsigned int N; // number in group
    unsigned int count; // number of arrived tasks
    // ANY ADDITIONAL VARIABLES NEEDED FOR EACH IMPLEMENTATION

    public:
    Barrier( int N ) : N( N ), count( 0 ) {
        // ANY ADDITIONAL INITIALIZATION NEEDED FOR EACH IMPLEMENTATION
    }
    void block(); // WRITE A VERSION FOR EACH IMPLEMENTATION
}
```

The group size, \(N\), is passed to the constructor. Do not write or create the client tasks.

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

```cpp
#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.

Assume the existence of the following routines for internal scheduling with barging:

```cpp
void wait() {
    bench.wait(); // wait until signalled
    while ( rand() % 5 == 0 ) {
        _Accept( block ) { // accept barding callers
            _Else {
                // do not wait if no callers
            }
        } // Accept
    }
}

void signalAll() {
    while ( ! bench.empty() ) bench.signal(); // drain the condition
}
```
6. 26 marks Write an administrator task for the Echo-River Rafting-Company, which offers rafting trips on the Tuolumne River composed of 2 guides and $R$ rafters. The company has ten guides with a sufficient number of rafts available for hire. Once a group of 2 guides and $R$ rafters form, a rafting trip can occur.

Figure 1 contains the starting code for the Echo-River Rafting-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 rafters to indicate their desire to take a rafting trip. A future lead-guide is immediately returned to the rafter so they do not have to wait for the lead-guide (e.g., they can get ready for the rafting trip). Eventually, the rafter accesses the lead-guide future to start the trip, which may block until a pair of guides and $R-1$ other rafters are available.

**onduty**: is called by guides to indicate their desire to supervise a rafting trip. The call blocks immediately until another guide and $R$ rafters are available; when this call returns it specifies the lead guide for the pair of guides.

The company administrator assigns $R$ rafters and two guides (on a first-come-first-served basis) to a raft, releases both guides indicating which is the lead guide, and informs the $R$ rafters about the lead guide via their future. The lead guide can be either of the two guides. (Then the non-lead guide would
contact the lead guide to select a raft, and the rafters would contact the lead guide to learn which raft to assemble at, which you do not have to write.

When the administrator’s destructor is called, the administrator may assume no new calls occur from rafters. However, the administrator must unblock rafters waiting for a lead guide by inserting exception \texttt{Closed} as the future value, and unblock any guides waiting for \(R\) rafters, returning \texttt{nullptr} as the address of the lead guide.

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 \texttt{Echo::main} based on the given outline, \textbf{do not write a rafter or guide or} \texttt{uMain::main}. Assume \texttt{uMain::main} creates the rafter.

\texttt{\textmu C++} future server operations are:

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

\texttt{\textmu C++} wait statement allows an integer/pointer value to be stored with a waiting task on a condition queue. The integer value can be accessed through the \texttt{uCondition} member routine \texttt{front}, e.g.:

\[
x = \text{guides.front();}
\]

\texttt{C++} list operations are:

\begin{verbatim}
int size()
bool empty()
T front()
T back()
void push_front( const T &x )
void push_back( const T &x )
void pop_front()
void pop_back()
void clear()
\end{verbatim}

| \texttt{int size()}               | list size               |
| \texttt{bool empty()}            | size() == 0             |
| \texttt{T front()}               | first element           |
| \texttt{T back()}                | last element            |
| \texttt{void push_front( const T \&x )} | add x before first element |
| \texttt{void push_back( const T \&x )} | add x after last element |
| \texttt{void pop_front()}        | remove first element    |
| \texttt{void pop_back()}         | remove last element     |
| \texttt{void clear()}            | erase all elements      |