Final Examination
Term: Fall Year: 2019

CS343
Concurrent and Parallel Programming
Sections 001, 002, 003
Instructor: Caroline Kierstead and Peter Buhr

Saturday, December 7, 2019
Start Time: 12:30 End Time: 15:00
Duration of Exam: 2.5 hours
Number of Exam Pages (including cover sheet): 7
Total number of questions: 7
Total marks available: 120
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED
1. (a) **7 marks** Given the following precedence graph:

```
S1  S2
  ↓  ↓
S3  S4  S5
  ↓  ↓
S6  S7
```

construct an *optimal* solution, i.e., minimal threads and locks, using COBEGIN and COEND in conjunction with *binary* semaphores using P and V to achieve the precedence graph. Use BEGIN and END to make several statements into a single statement and show the initial value (0/1) for all semaphores. Name your semaphores \( Ln \), e.g., \( L_1, L_2, \ldots \), to simplify marking.

(b) **4 marks** The readers/write solution 3 prevents starvation using the following protocols:

```
reader
if ( wcnt > 0 || wdel > 0 ) { // entry
    if ( rcnt == 0 && wdel > 0 ) {
        wwait.V(); // pass baton
    }
    else {
        entry.V(); // put baton down
    }
}
else if ( wdel > 0 ) {
    wwait.V();
    entry.V(); // put baton down
}
else if ( wdel > 0 ) {
    wwait.V();
    entry.V(); // put baton down
}
else {
    entry.V(); // put baton down
}
```

```
writer
if ( rcnt > 0 || wcnt > 0 ) { // entry
    if ( rdel > 0 ) {
        rwait.V();
    }
    else if ( wdel > 0 ) {
        wwait.V();
    }
    else {
        entry.V();
    }
}
```

Explain how this code causes alternation of readers and writers when there are simultaneous readers and writers waiting.

(c) Given the following readers-writer snapshot:

```
12:30
```

and the 12:30 writer exits the critical section at 2:30,

i. **2 marks** Explain a scenario resulting in *staleness* and one resulting in *freshness*.

ii. **1 mark** What is the high-level (general) approach to eliminate staleness/freshness?

iii. **1 mark** What is the general problem with this approach?

(d) A solution that deals with both barging and staleness/freshness for the readers-writer problem relies on having a semaphore method \( P \) of the form \( s.P(m) \).

i. **2 marks** Explain the semantics of \( s.P(m) \), i.e., how it works.

ii. **2 marks** Explain the problem that occurs without this semantics.

iii. **2 marks** How do these semantics solve the problem?
2. (a) 2 marks Explain the difference between live lock and starvation.

(b) 3 marks If the Banker’s algorithm denies a resource request, does that guarantee the tasks would deadlock if the request is granted? Explain.

(c) i. 2 marks Briefly explain synchronization and mutual exclusion deadlock.
    ii. 1 mark Explain the approach used to prevent mutual exclusion deadlock.

(d) 4 marks Does the following resource allocation graph produce a deadlock? Explain your answer by using graph reduction.

![Resource Allocation Graph]

3. (a) 2 marks Monitor scheduling is categorized as either implicit or explicit scheduling. Describe what is meant by these terms.

(b) 2 marks Describe one advantage and one disadvantage of the immediate return monitor.

(c) The following diagram shows a snapshot of a task implementing a bounded buffer using external scheduling.

![Bounded Buffer Diagram]

The task labelled BB is the buffer and the entry queue contains tasks P1 calling insert and task C2 calling remove.

i. 1 mark After BB executes _Accept( insert, remove ), where is BB in the diagram?

ii. 1 mark Which task executes next?

iii. 1 mark Assume the buffer is full. What action does the task that executes next do?

iv. 1 mark After executing the action, where does it go in the diagram?
v. 1 mark Which task executed next?
vi. 1 mark After the task that executed next exits, which task executes next?
vii. 1 mark After the task that executed next exits, which task executes next?

(d) Implicit monitor scheduling can select from the calling (C), signalled (W), and signaller (S) queues. Assigning different relative priorities to these queues creates different monitors.

<table>
<thead>
<tr>
<th>relative priority</th>
<th>1 &lt; C &lt; W &lt; S</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>C &lt; S &lt; W</td>
</tr>
<tr>
<td>3</td>
<td>C = W &lt; S</td>
</tr>
<tr>
<td>4</td>
<td>C = S &lt; W</td>
</tr>
<tr>
<td>5</td>
<td>C = W = S</td>
</tr>
<tr>
<td>6</td>
<td>C &lt; W = S</td>
</tr>
<tr>
<td>7</td>
<td>S = W &lt; C</td>
</tr>
<tr>
<td>8</td>
<td>W &lt; S = C</td>
</tr>
<tr>
<td>9</td>
<td>W &lt; C &lt; S</td>
</tr>
<tr>
<td>10</td>
<td>S &lt; W = C</td>
</tr>
<tr>
<td>11</td>
<td>S &lt; C &lt; W</td>
</tr>
<tr>
<td>12</td>
<td>W &lt; S &lt; C</td>
</tr>
<tr>
<td>13</td>
<td>S &lt; W &lt; C</td>
</tr>
</tbody>
</table>

i. 1 mark Explain why relative priorities 7-13 are rejected.
ii. 1 mark Explain why relative priorities 5-6 are rejected.
iii. 1 mark Explain why relative priorities 3-4 need barging avoidance.
iv. 1 mark Explain why relative priorities 1-2 have barging prevention.

4. (a) 2 marks Explain why a task that has called into a monitor or another task may NOT block on a semaphore inside the monitor/task. What kind of lock should be used to block in a monitor/task?

(b) 1 mark Why it is it sometimes necessary to accept an explicit stop method rather the destructor to terminate a task?

(c) i. 3 marks Name the three elementary properties that make up a task. (just the names)
   ii. 1 mark When a _When clause is false, what does it do to the _Accept clause?
   iii. 1 mark What is the reason buffering in a server task produces little additional concurrency? (two words)
   iv. 1 mark Suggest a change to a server task so that buffering produces additional concurrency. (short answer)
   v. 1 mark If a language supports asynchronous call, how is the call implemented? (one word)

5. (a) 2 marks Explain why is it necessary to eagerly move reads up the memory hierarchy but a good idea to lazily move writes down the memory hierarchy.

(b) 2 marks The following is the entry protocol for the client in the N-thread software-solution for the arbiter algorithm.

```c
intents[me] = true; // entry protocol
while (!serving[me]) {} // busy wait
```
Explain why this entry protocol does not work if the compiler/hardware allows disjoint reads to be moved before writes.

(c) 2 marks Explain how the load-locked/store-conditional (LL/SC) instruction pair work to provide atomic execution in the hardware.

(d) 2 marks Give two similarities between the messaging passing in the programming language Go with the message passing in Actors.

6. Some students have decided to hold a “Peanut Butter and Jelly Sandwich Social” picnic as a way for people to get to know each other. Since students are poor, they have decided to pool their resources. They form pairs (you may assume the total number of students is a multiple of 2) where each member brings one of the two ingredients needed to make a peanut butter and jelly sandwich. Each matched pair of students receive the identity of their sandwich partners so that they can arrange a meeting.

The interface for the exchange class is the following (you may only add code in the designated areas L1, L2 and L3).

```cpp
_Monitor Exchange {
    int pb, jy; // exchange
    // L1: ANY VARIABLES NEEDED FOR EACH IMPLEMENTATION
    public:
        // common interface
        int PB(int id) { // called for peanut butter
            // L2: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        }
        int JY(int id) { // called for jelly
            // L3: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        }
    }
};
```

The PB and JY members are called by the students bringing peanut-butter/jelly, respectively. These members do not return until both students for a picnic have met up and filled in their identities. Class variables pb and jy are used to exchange the student ids. The pairs are formed based on order of arrival; e.g., if students 2 and 5 are the first to bring in each of the two needed ingredients, they form the first group, etc. You must write both members PB and JY; you may NOT say member JY is the same as member PB with everything reversed. Do not write or create the student tasks.

Implement the exchange monitor Exchange using:

(a) 10 marks external scheduling,
(b) 11 marks internal scheduling,
(c) 10 marks implicit (automatic) signalling, using only the 3 macros defined below.

   Hint: you may use a std::list to simplify the implicit (automatic) signalling solution.

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

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( predicate ) ...
#define EXIT() ...
```

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 wait until the predicate evaluates to true. Macro EXIT must be called on return from a public routine of an automatic-signal monitor.

The C++ list operations are:
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 answer booklet.)*

```
_Task Coordinator {  
    public:  
        typedef Future_ISM<Shuttle *> Fshuttle;      // future shuttle identifier  
        _Event Closed {};          // too late for a ride  
    private:  
        const unsigned int ShuttleSize, NumShuttles, NumClients;  
        uCondition shuttles;      // shuttles wait here  
        bool shuttingDown = false; // true => shutting down  
        list< Fshuttle > clients; // client futures  
    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 ) {  
        }  
        bool timeUp() { return shuttingDown; }  
        bool checkIn() {  
            shuttles.wait( (uintptr_t)&uThisTask() );  // wait for clients  
            return shuttingDown;  
        }  
        Fshuttle getRide( unsigned int id ) {  
            Fshuttle fshuttle;  
            clients.push_back( fshuttle );  // store future  
            return fshuttle;  
        }  
    private:  
        // ANY ADDITIONAL PRIVATE DECLARATIONS  
        void fillShuttle( unsigned int noOfClients ); // HINT: consider writing this helper routine  
        void main() {  // YOU WRITE THIS ROUTINE  
            // coordinate timer, shuttles, and clients  
            // tell clients, shuttles, and timer to shutdown  
        }  
};
```

*Figure 1: Coordinator Administrator*
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 work as possible; a monitor-style solution will receive little or no marks. Assume the program main creates the Coordinator, Timer, Shuttle and Client tasks, and deletes them at the end of the day.

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.