UNIVERSITY OF WATERLOO

Final Examination
Term: Fall Year: 2022

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

Saturday, December 10, 2022
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: 123
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

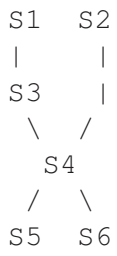
1. (a) **2 marks** When creating a cyclic barrier, an extra Coordinator task is often created. What is the advantage of the Coordinator?

   (b) **2 marks** In a barrier lock, explain the purpose of members block and last.

   (c) **3 marks** Explain how a semaphore is used to create the *synchronization pattern*, i.e., do S1 before S2. Use words not code.

   (d) **2 marks** Explain the difference between a *simple* and *complex* critical section?

   (e) **7 marks** Given the following precedence graph:



   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., $L1, L2, ...$, to simplify marking.

   (f) **1 mark** Is *staleness / freshness* a form of barging?

   (g) **2 marks** Explain the purpose of a *shadow queue* with respect to locks.

   (h) **2 marks** With respect to the readers/writer problem, there are two kinds of temporal reordering that can lead to incorrect behaviour. Describe these two reorderings.

2. (a) **2 marks** What two aspects of concurrency may be missing when there is a race condition?

   (b) **6 marks** Restructure the following code in two different ways so the deadlock is prevented.

```
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
```

   (c) 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 $(a_1a_2a_3a_4)$, 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 with the steps of the banker's algorithm and a concluding statement.

   i. **3 marks**

	P1	P2	P3	P4
Maximum Needed	2	5	8	8
Current Acquired	1	1	4	4
Needed to Max.	1	4	4	4

   ii. **3 marks**

	P1	P2	P3	P4
Maximum Needed	2	5	8	8
Current Acquired	0	1	2	7
Needed to Max.	2	4	6	1

iii. **4 marks**

	P1	P2	P3	P4
Maximum Needed	2	5	8	8
Current Acquired	1	2	4	2
Needed to Max.	1	3	4	6

3. (a) **2 marks** The following monitor simulation using a mutex lock has a problem returning v. Explain the problem and how to fix it. Use words not code.

```
class Mon {
    MutexLock mlock;
    int v;
  public:
    int x(...) {
        mlock.acquire();
        ...
        mlock.release();
        return v;
    }
};
```

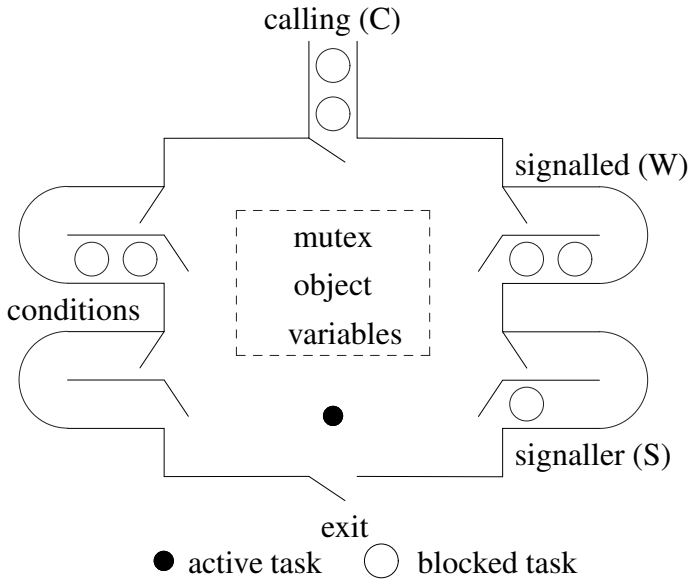
(b) **2 marks** What is unusual about signalling a condition in a monitor?

(c) **2 marks** Explain the difference between signal and signalBlock on a monitor condition variable.

(d) **1 mark** Does **_Accept** behave like a signal or signalBlock?

(e) **2 marks** Give two situation where a monitor uses a **_Nomutex** public member.

(f) **2 marks** Explain the transformation from this general monitor into the specific μC++ monitor.



4. (a) **2 marks** Explain two capabilities added with the *long form* of the **_Accept** / **_Select** statements.

(b) **2 marks** Explain a *rendezvous failure*.

(c) **2 marks** Explain how a *courier task* works.

(d) **1 mark** Does a future increase client or server side concurrency?

(e) **2 marks** Explain the semantics of the following **_Select** statement.

```
_Select( f1 || f2 && f3 );
```

5. (a) **1 mark** Are registers shared among different cores on a multiprocessor computer?

   (b) **2 marks** How does data replication work in a cache hierarchy?

   (c) **2 marks** How does **volatile** prevent problems when programming with race conditions?

   (d) **1 mark** What is the name of the problem that occurs if only the CAS instruction is used to build a lock-free stack?

   (e) **1 mark** How do goroutines communicate in the programming language Go?

   (f) Given the following Java monitor to implement a barrier:

```
class Barrier {                          // monitor
    private int N, count = 0;
    public Barrier( int N ) { this.N = N; }
    public synchronized void block() {
        count += 1;                      // count each arriving task
        if ( count < N )
            try { wait(); } catch( InterruptedException e ) {}
        else                             // barrier full
            notifyAll();                 // wake all barrier tasks
        count -= 1;                      // uncount each leaving task
    }
}
```

   i. **3 marks** Explain why the barrier is incorrect.

   ii. **1 mark** Explain a simple correction to make it work.

   iii. **2 marks** Explain what unusual phenomenon prevents the simple solution from working.

6. A group of students go to the horse races to gamble their tuition money. Each race, a student bets a *fixed amount* of money on two different horses, which is subtracted from their tuition money. After the race is over, if a student picks a winner, the amount won is added to their tuition money. The students gamble until one (or more in the simultaneous case) loses enough money they cannot make the fixed bet. Once a student cannot bet, all the students leave the race track and stop betting.. Figure 1 shows an example day at the races. Consult this example to understand what your monitor needs to handle. **DO NOT print any output or compute any values needed only for printing!**

   Figure 2 shows the betting interface (you may only add code in the designated areas). **(Do not copy the starting code into your answer booklet.)** Write the tally-betting office using 3 kinds of monitors.

   - Member placeBet (you implement) is called by a student to place a bet on 2 horses using BetSlip.
   - Member done (you implement) is called by a student with insufficient money to bet. After which, arriving and waiting students receive a Leave exception indicating its time to stop betting.
   - Member race (you implement) is randomly called by the racetrack to indicate a race is run and deliver the winning horse.
   - Member tally (given) is called to compute the race results for each student's bet. Called to generate the return value from placeBet.

   Write only the code for the TallyBets monitors; do not write or create the student, racetrack, or program main. Assume the program main shuts down the racetrack after the students leave. No error checking is required.

```
fixed bet 2 x $3, 5 horses, betting odds 3 to 1
better 1 starts with $100
race 1: 1 bet, $6 bet, winning horse 2
better 0 starts with $100
better 3 starts with $100
better 2 starts with $100
better 4 starts with $100
better 5 starts with $100
race 2: 6 bets, $36 bet, winning horse 1
race 3: 3 bets, $18 bet, winning horse 4
race 4: 6 bets, $36 bet, winning horse 2
race 5: 4 bets, $24 bet, winning horse 2
race 6: 6 bets, $36 bet, winning horse 3
race 7: 5 bets, $30 bet, winning horse 0
race 8: 4 bets, $24 bet, winning horse 4
race 9: 6 bets, $36 bet, winning horse 3
race 10: 4 bets, $24 bet, winning horse 4
race 11: 5 bets, $30 bet, winning horse 3
race 12: 6 bets, $36 bet, winning horse 3
race 13: 5 bets, $30 bet, winning horse 0
race 14: 5 bets, $30 bet, winning horse 1
race 15: 4 bets, $24 bet, winning horse 0
race 16: 4 bets, $24 bet, winning horse 2
race 17: 5 bets, $30 bet, winning horse 4
race 18: 6 bets, $36 bet, winning horse 2
race 19: 5 bets, $30 bet, winning horse 3
race 20: 5 bets, $30 bet, winning horse 4
race 21: 5 bets, $30 bet, winning horse 1
race 22: 5 bets, $30 bet, winning horse 2
race 23: 5 bets, $30 bet, winning horse 3
race 24: 3 bets, $18 bet, winning horse 2
better 4 ran out of cash $4
better 3 finishes with $58
better 5 finishes with $31
better 0 finishes with $31
race 25: 1 bet, $6 bet, winning horse 0
better 1 ran out of cash $1
better 2 finishs with $34
students leave
```

Figure 1: Betting Example

Implement the betting monitor TallyBets using:

(a) **10 marks** external scheduling,

(b) **8 marks** internal scheduling,

(c) **8 marks** implicit (automatic) signalling, using *only* the following three macros.

```
#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 is called whenever control leaves the monitor.

7. **25 marks** Write an administrator task to perform the same TallyBets as question 6. The only interface changes are:

```
#include <list>
#include <uFuture.h>

_Task TallyBets {
    // YOU ADD DECLARATIONS
  public:
    TallyBets( unsigned int numStuds );          // number of students
    typedef Future_ISM<Payout> FPayout;          // future type
    FPayout placeBet( BetSlip slip );            // return future payout
    ...
  private:
    // YOU ADD DECLARATIONS
    ...
}; // TallyBets
```

```
_Monitor TallyBets {
#if defined( EXT )                          // external scheduling monitor solution
    // YOU ADD DECLARATIONS
#elif defined( INT )                        // internal scheduling monitor solution
    // YOU ADD DECLARATIONS
#elif defined( AUTO )                       // automatic-signal monitor solution
    // YOU ADD DECLARATIONS
#endif
  public:                                   // common interface
    enum { FixBet = 3, NumHorses = 5, BettingOdds = 3 };
    _Event Leave {};
    struct BetSlip { unsigned int bet1, horse1, bet2, horse2; };
    struct Payout { unsigned int winner, payout; };

    Payout placeBet( BetSlip slip ) {       // called by students
        // YOU WRITE THIS CODE
    }
    void done() {                           // called by students
        // YOU WRITE THIS CODE
    }
    void race( unsigned int winner ) {      // called by racetrack
        // YOU WRITE THIS CODE
    }
  private:
    // YOU ADD DECLARATIONS

    Payout tally( BetSlip & slip ) {        // called before returning from placeBet
        unsigned int payout = 0;
        if ( slip.horse1 == winner ) payout = slip.bet1 * BettingOdds;
        else if ( slip.horse2 == winner ) payout = slip.bet2 * BettingOdds;
        return { winner, payout };
    } // TallyBets::tally
}; // TallyBets
```

Figure 2: TallyBet Interface

where the constructor is passed the number of students in the group and placeBet returns a future to a payout, which is filled in after a race for each student. **(Do not copy the starting code into your answer booklet.)**

Ensure the TallyBets task does as much administration work as possible; a monitor-style solution will receive little or no marks. **Write only the code for the TallyBets task; do not write or create the student, racetrack, or program main. Assume the program main shuts down the racetrack after the students leave. No error checking is required.**

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

The 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