Final Examination

Term: Fall Year: 2021

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002
Instructor: Peter Buhr

Thursday, December 9, 2021
Start Time: 16:00 Dec 9 End Time: 16:00 Dec 10
Duration of Exam: 3 hours including download and upload
Number of Exam Pages (including cover sheet): 6
Total number of questions: 2
Total marks available: 76

Click below to view your time remaining in the Final exam

https://student.cs.uwaterloo.ca/~cs343/cgi-bin/examTime.cgi

Instructions

The final exam duration is **3 hours** including download of the exam and uploading/submitting the programs. Your time begins at the exam download and is compared with the date-stamps on the files you submit.

The final consists of 5 *complete* running programs, which will be tested after the exam.

You are given the following files:

```
Makefile
MPRNG.h
AutomaticSignal.h
BarrierTime.cc
BarrierTimeAdmin.cc
```

Answer each question in the appropriate file using the given program main, Worker and Timer tasks, and starter code specific for each question. Do NOT subdivide a program into separate .h and .cc files. Use the Makefile to help simplify compilation, when a question has multiple implementations. Enter "make" in the given directory to see how the programs compile.

After writing and testing the 5 programs, submit all the files on the undergraduate environment using the submit command:

```
$ submit cs343 final directory-name
```

As a precaution, submit often, not just at the end of the final.

The following aids are allowed:

- computer to write, compile, and test the final programs.
- course notes
- course textbook
- your previous assignments
- μC++ Annotated Reference Manual
- man pages
- cppreference.com / cplusplus.com to look up C++ syntax

The following aids are NOT allowed:

- any answers from prior final exams because you did not create them
- any web searching, like stack overflow or Wikipedia
- any interaction with another person
- any use of another person's documents or programs

Basically, you are to complete the final by yourself using only course-related material or work you have created versus the work of others.

There is no way for us to fairly answer questions over the 24 hours of the exam, so state any assumptions with a comment in the program and press on.

Do not post on Piazza during the 24-hour exam period. If necessary, do a private post.

Monitors

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 an Nth 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:

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

A *time-barrier* has a timeout member to flush waiting tasks. That is, if less than N tasks have arrived at time T, the group of tasks ($< N$) is unblocked to make progress. If a timeout call occurs when no tasks are waiting in the barrier, the call does nothing. Hence, timeout needs to *trick* waiting tasks that called block to unblock by modifying variables and/or start unblocking. Finally, timeout ensures no quorum-failure deadlock because waiting tasks eventually unblock.

1. Using μC++ monitors, implement a time-barrier lock in file BarrierTime.cc using the given interface (you may add a public destructor and private members):

```
_Monitor Barrier {
    const unsigned int N;            // group size
    unsigned int count = 0,          // blocked tasks (< N)
                 total = 0,          // value sum
                 groupno = 0;        // group number
    // VARIABLES FOR EACH IMPLEMENTATION
  public:
    struct Result { unsigned int total, groupno; };
    Barrier( unsigned int N ) : N( N ) {
        // INITIALIZATION FOR EACH IMPLEMENTATION
    }
    void timeout() {
        // WRITE THIS MEMBER FOR EACH IMPLEMENTATION
        // PRINT TIMEOUT AND GROUP MESSAGE
    }
    Result block( unsigned int value ) {
        // WRITE THIS MEMBER FOR EACH IMPLEMENTATION
        // PRINT GROUP MESSAGE
    }
};
```

The group size, N, is passed to the constructor. Each task calling block passes a value (>0). The sum of these values for all the tasks in a group and the group number are returned in Result to each task. For a timeout, *the total returned is 0*, so a task knows if its group timed out. A timeout with no waiting tasks *does not* advance the group number.

Implement the time-barrier monitor in the following ways, where the solutions *must prevent* staleness and freshness.

(a) **10 marks** external scheduling

(b) **13 marks** internal scheduling

(c) **17 marks** internal scheduling with a Java-style monitor using a single condition variable and the 2 given routines:

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

Note, this implementation is tricky so do not spend too much time on it. Get something that starts to work and come back to it, if you have time at the end of the exam.

(d) **11 marks** implicit (automatic) signalling using the 3 given macros in file AutomaticSignal.h.

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( predicate, before, after ) ...
#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 mutex member of an automatic-signal monitor.

Use the given Makefile, which generates an executable called barrier, to compile and test the 4 time-barrier implementations using commands:

```
$ make barrier MIMPL=EXT
$ barrier ...
$ make barrier MIMPL=INT
$ barrier ...
$ make barrier MIMPL=INTB
$ barrier ...
$ make barrier MIMPL=AUTO
$ barrier ...
```

DO NOT CHANGE THIS Makefile!

Use the given program main for testing, which has the following shell interface:

barrier [workers (> 0 & <= group) | 'd' [group (> 0) | 'd' [times (> 0) | 'd' [seed (> 0) | 'd']]]]

workers is the number of tasks that enter the barrier (>0). If d or no value for workers is specified, the default is 6.

group is the size of a barrier group (>0). If d or no value for group is specified, the default is 4.

times is the number of times (>0) each worker calls the barrier. If d or no value for times is specified, the default is 30.

seed is the starting seed for the random-number generator (>0). If d or no value for seed is specified, the default is getpid(), so each run of the program generates different output. **Note, because the program has 2 kernel threads for parallelism, output is not repeatable, even with the same seed.**

The following example output is for the EXT monitor (the seed is not helpful because of parallelism). Note, when a worker task, W, continues after its call to block, it prints its group number, group total returned from block, and its running total of these group totals: group-total,running-total. Because of parallelism, the output order is very interleaved.

```
$ a.out 4 2 4
TIMEOUT 1 waiting
GROUP 1
GROUP 2
W2 group 1 totals 0,0
W0 group 2 totals 2,2
W3 group 2 totals 2,2
GROUP 3
W2 group 3 totals 3,3
GROUP 4
W1 group 3 totals 3,3
TIMEOUT 0 waiting
GROUP 5
W0 group 4 totals 4,6
W3 group 4 totals 4,6
TIMEOUT 0 waiting
W1 group 5 totals 5,8
GROUP 6
W2 group 5 totals 5,8
GROUP 7
W2 group 7 totals 7,15
TIMEOUT 0 waiting
W1 group 7 totals 7,15
W3 group 6 totals 6,12
W0 group 6 totals 6,12
Worker 2 finish: total 15 timeouts 1
GROUP 8
W1 group 8 totals 8,23
Worker 1 finish: total 23 timeouts 0
TIMEOUT 1 waiting
GROUP 9
W3 group 8 totals 8,20
Worker 3 finish: total 20 timeouts 0
W0 group 9 totals 0,12
Worker 0 finish: total 12 timeouts 1
TIMEOUT 0 waiting
TIMEOUT 0 waiting
Timer finish
```

```
$ a.out 4 3 6
GROUP 1
W3 group 1 totals 3,3
W0 group 1 totals 3,3
W2 group 1 totals 3,3
GROUP 2
TIMEOUT 1 waiting
GROUP 3
TIMEOUT 0 waiting
W0 group 2 totals 5,8
W1 group 2 totals 5,5
W3 group 2 totals 5,8
W2 group 3 totals 0,3
TIMEOUT 2 waiting
GROUP 4
W0 group 4 totals 0,8
W2 group 4 totals 0,3
GROUP 5
W3 group 5 totals 9,17
W1 group 5 totals 9,14
W0 group 5 totals 9,17
GROUP 6
W1 group 6 totals 11,25
W2 group 6 totals 11,14
W3 group 6 totals 11,28
GROUP 7
W3 group 7 totals 14,42
W0 group 7 totals 14,31
GROUP 8
W1 group 7 totals 14,39
W3 group 8 totals 17,59
Worker 3 finish: total 59 timeouts 0
W0 group 8 totals 17,48
Worker 0 finish: total 48 timeouts 1
W2 group 8 totals 17,31
TIMEOUT 2 waiting
GROUP 9
W2 group 9 totals 0,31
Worker 2 finish: total 31 timeouts 3
W1 group 9 totals 0,39
TIMEOUT 0 waiting
TIMEOUT 1 waiting
GROUP 10
W1 group 10 totals 0,39
Worker 1 finish: total 39 timeouts 2
TIMEOUT 0 waiting
Timer finish
```

```
$ a.out 5 4 7
GROUP 1
W3 group 1 totals 4,4
W2 group 1 totals 4,4
W0 group 1 totals 4,4
W1 group 1 totals 4,4
GROUP 2
W2 group 2 totals 7,11
W3 group 2 totals 7,11
W0 group 2 totals 7,11
W4 group 2 totals 7,7
GROUP 3
W0 group 3 totals 11,22
W1 group 3 totals 11,15
W3 group 3 totals 11,22
GROUP 4
W2 group 3 totals 11,22
W1 group 4 totals 13,28
W0 group 4 totals 13,35
W4 group 4 totals 13,20
W3 group 4 totals 13,35
GROUP 5
W0 group 5 totals 18,53
TIMEOUT 2 waiting
GROUP 6
W3 group 5 totals 18,53
W0 group 6 totals 0,53
W2 group 5 totals 18,40
W1 group 5 totals 18,46
W4 group 6 totals 0,20
GROUP 7
W1 group 7 totals 23,69
W0 group 7 totals 23,76
W3 group 7 totals 23,76
W2 group 7 totals 23,63
Worker 0 finish: total 76 timeouts 1
GROUP 8
W3 group 8 totals 23,99
W1 group 8 totals 23,92
W2 group 8 totals 23,86
W4 group 8 totals 23,43
Worker 3 finish: total 99 timeouts 0
TIMEOUT 3 waiting
GROUP 9
W2 group 9 totals 0,86
W1 group 9 totals 0,92
W4 group 9 totals 0,43
Worker 2 finish: total 86 timeouts 1
Worker 1 finish: total 92 timeouts 1
TIMEOUT 1 waiting
GROUP 10
W4 group 10 totals 0,43
TIMEOUT 1 waiting
GROUP 11
W4 group 11 totals 0,43
Worker 4 finish: total 43 timeouts 4
TIMEOUT 0 waiting
Timer finish
```

Tasks

2. **25 marks** Using a μC++ task, implement a *time-barrier administrator* in file BarrierTimeAdmin.cc using the given interface (you may add a public destructor and private members):

```
_Task Barrier {
  public:
    struct Result { unsigned int total, groupno; };
    typedef Future_ISM<Result> Fresult; // future type
  private:
    const unsigned int N;           // group size
    unsigned int count = 0,         // blocked tasks (< N)
                 total = 0,         // value sum
                 groupno = 0;       // group number
    // ADDITIONAL VARIABLES
  public:
    Barrier( unsigned int N ) : N( N ) {
        // ADDITIONAL INITIALIZATION
    }
    void timeout() {
        // WRITE THIS MEMBER
    }
    Fresult block( unsigned int value ) {
        // WRITE THIS MEMBER
    }
  private:
    void main() {
        // WRITE THIS MEMBER
        // PRINT TIMEOUT AND GROUP MESSAGE
    }
};
```

The group size, *N*, is passed to the constructor. Each task calling block passes a value (>0). The sum of these values for all the tasks in a group and the group number are returned to each task in Result. For a timeout, *the total returned is 0*, so a task knows if its group timed out. A timeout with no waiting tasks *does not* advance the group number.

Use the given Makefile, which generates an executable called barrieradm, to compile and test the time-barrier administrator using commands:

```
$ make barrieradm
$ barrieradm ...
```

The program main, shell interface, and output are the same as for the monitors in question 1.