UNIVERSITY OF WATERLOO

Midterm Examination
Fall 2023

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002

Duration of Exam: 2 hours
Number of Exam Pages (including cover sheet): 6
Total number of questions: 5
Total marks available: 110

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructor: Peter Buhr

November 2, 2023

1. (a) **5 marks** Rewrite the following code fragment to remove the exit from the middle of the loop.

```
for ( ;; ) {
    S1;
  if ( C ) { E; break; }
    S2;
}
```

   (b) **1 mark** Explain why a loop exit never needs an **else** clause.

   (c) **1 mark** Why are *flag variables* the variable equivalent to a **goto**?

   (d) **1 mark** Why is it good practice to always label exits (**break** statements)?

   (e) **2 marks** Normal and labelled **break** are a **goto** with what two limitations?

   (f) **1 mark** Explain the only situation where a **goto** is necessary.

   (g) **3 marks** Explain the purpose of uNoCtor and how it differs from unique_ptr.

   (h) **2 marks** What issue is removed by the $O(N)$ matching search during propagation versus:
      i. return codes exiting multiple call-levels.
      ii. fixup routines passed into a call.

   (i) **2 marks** What is the problem with a destructor raising an exception? Is there a way around this problem?

   (j) **1 mark** Explain why a resumption handler cannot perform a **break**, **continue**, **goto**, or **return**.

   (k) **1 mark** Why is it necessary to enable/disable non-local exceptions?

2. (a) i. **1 mark** When a coroutine's main returns, where does control transfer to?
      ii. **2 marks** Why is this transfer point important for full coroutining?

   (b) **1 mark** Explain why a programmer might put a resume in a coroutine's constructor.

   (c) **2 marks** When a coroutine fails to handle an exception, where does control transfer to? When a task fails to handle an exception, where does control transfer to?

3. (a) **2 marks** Explain the term *concurrent bottleneck*?

   (b) **1 mark** Explain the term *scheduler*?

   (c) **2 marks** Explain the term *non-linear program speedup*.

   (d) **2 marks** For each of the following indicate if the concurrency is *implicit* or *explicit*.
      i. COBEGIN/COEND
      ii. START/WAIT
      iii. actors
      iv. tasks

   (e) **4 marks** Explain how the following software mutual-exclusion code fails and what is the name of the failure.

```
me = WantIn;                    // entry protocol
while ( you == WantIn ) {}
CriticalSection();              // critical section
me = DontWantIn;                // exit protocol
```

   (f) **2 marks** Explain the term *reader-write safe*.

   (g) **1 mark** How many locks are needed for mutual exclusion?

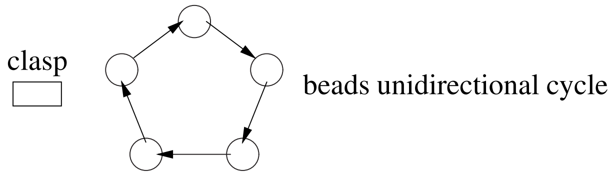
   (h) **1 mark** What property does an atomic instruction require to implement mutual exclusion?

   (i) **1 mark** Explain *barging*.

   (j) **2 marks** Explain what yieldNoSchedule(lock) does.

4. **16 marks**

The Halloween necklace game has a *clasp* to open a necklace and *beads* in a unidirectional cycle.



Each bead takes a turn generating a random key, prng(), and calling clasp.lock to see if the key opens the clasp. However, the clasp can raise a Zombie exception at a bead, after which that bead continues to play the game but cannot win. Now, when the clasp opens, it raises a Won exception at the bead, which prints its id and either "Won" or "Zombies cannot win", and the game ends. Use the following two exceptions and clasp coroutine:

```
_Event Zombie {};
_Event Won {};
_Coroutine Clasp {
    unsigned int key;
    void main() {
        for ( ;; ) {
          if ( key % 50 == 5 ) break;        // check if clasp opens
            if ( prng( 20 ) == 0 ) _Resume Zombie() _At resumer();
            suspend();
        }
        _Resume Won() _At resumer();      // clasp opened
        suspend();
    }
  public:
    void lock( unsigned int key ) {
        Clasp::key = key;
        resume();
    }
} clasp; // DECLARE Clasp OBJECT
```

Write a complete full-coroutine version for the bead and the program main using the following outline.

```
_Coroutine Bead {
    unsigned int id;
    Bead * part;
    void main() {
        // YOU WRITE THE CODE TO PLAY THE GAME.
    }
  public:
    Bead( unsigned int id, Bead * partner ) : id{ id }, part{ partner } { resume(); };
    void next() { resume(); }
};
int main() {
    // YOU WRITE THE CODE TO CREATE A CYCLE OF 5 BEADS AND START GAME.
}
```

The program main creates a cycle of 5 beads, where each bead immediately suspends back to the program main to set it as the coroutine's starter, and then the program main starts the game by calling next for one of the beads.

Note: Few marks will be given for a solution that does not take advantage of the capabilities of the coroutine, i.e., you must use the coroutine's ability to retain data and execution state.

5. The half-way problem concurrently performs work but stops when the work is half complete. Concurrent workers compute 1000 units of work in batches of 100 units at a time. Each batch is reported to the manager, who inspects the units and randomly reduces a worker's completed work by 50; i.e., the manager finds a problem with the batch of 100 units, where 50 of them are defective. The worker has to recompute more work, still in units of 100, to make up for the 50 defective units removed by the manager. Hence, a worker might generate 100, 200, 300, 400, 450, 550, 650, 750, 800, 900, 950, 1050, and then be told the work is complete by the manager because $1050 \geq 1000$. After 5 workers complete their assigned units, the manager tells the remaining 5 workers to stop. Note, it is possible for a worker to have completed some or all of the 1000 work units and be told to stop.

   (a) **1 mark** Explain why the half-way problem cannot be implemented using COFOR.

   (b) **24 marks** Write a complete version of the half-way problem using 10 Halfway worker actors and 1 Manager actor. Use the following messages and code outline:

```
// Sent from the worker to the manger to indicate it has completed 100 work units towards
// its total. The message contains the worker id and a reference to the total work
// completed so far by the worker. The manager uses the reference to randomly deduct
// 50 units from the completed work.
struct Progress : public uActor::Message {
    unsigned int id, & complWork;
    Progress( unsigned int id, unsigned int & complWork ) :
                Message{ uActor::Delete }, id{ id }, complWork{ complWork } {}
};

// Sent from the manager to the worker to indicate it has completed >= 1000 work units.
struct Complete : public uActor::Message {
} complete; // DECLARE Complete MESSAGE

_Actor Halfway {
    unsigned int id;
    Manager & manager;
    unsigned int complWork = 0;

    Allocation receive( Message & msg ) {
        Case ( uActor::StartMsg, msg ) {                    // continue working
            complWork += 100;
            manager | *new Progress( id, complWork );
            return Nodelete;
        } else Case( Complete, msg ) {                      // complete work
            osacquire( cerr ) << id << " complete work" << endl;
        } else Case( uActor::StopMsg, msg ) {               // force stop
            osacquire( cerr ) << id << " told to stop" << endl;
        }
        return Finished;
    }
  public:
    Halfway( unsigned int id, Manager & manager ) : id{ id }, manager{ manager } {};
};
```

Write a Manager actor and program main.

```
_Actor Manager {
    // YOU WRITE DECLARATIONS.

    Allocation receive( Message & msg ) {
        // YOU WRITE THE CODE TO MANAGE 10 Halfway WORKERS.
    }
  public:
    Manager() {
        // YOU WRITE THE CODE TO CREATE 10 Halfway WORKERS.
    }
};

int main() {
    // YOU WRITE DECLARATIONS AND CODE TO CREATE THE MANAGER
}
```

After receiving the Progress message from a Halfway worker actor, the Manager

- with probability 1 in 5, reduces this worker's completed work by 50.
- checks if a worker should continue working, and if so, sends it a uActor::startMsg message,
- checks if a worker has completed its work, and if so, sends a complete message,
- checks if 5 workers have completed, and if so, sends the remaining 5 workers a uActor::stopMsg message.

The manager's constructor creates the 10 Halfway actors and sends each a uActor::startMsg message.

The program main

- starts the actor system and creates the Manager actor.

No documentation or error checking of any form is required.

(c) **25 marks** Write a complete version of the half-way problem using 10 Halfway worker tasks, with management provided by the program main. Use the following exception types and code outline:

```
// Sent from the worker to the manger to indicate it has completed 100 work units towards
// its total. The exception contains the worker id, a reference to the total work completed
// so far by the worker, and a shared spin variable for synchronization with the manager.
// The manager uses the reference to randomly deduct 50 units from the completed work
// and reset the spin variable.
_Event Progress {
  public:
    unsigned int id, & complWork;
    bool & spin;                                    // synchronization
    Progress( unsigned int id, unsigned int & complWork, bool & spin ) :
                id{ id }, complWork{ complWork }, spin{ spin } {}
};

// Sent from the manager to the worker to indicate it has completed >= 1000 work units.
_Event Complete {};

// Sent from manager to worker to indicate the worker must stop execution,
// i.e., the manager has 5 completed workers.
_Event Stop {};
```

Write a Halfway worker-task and program main.

```
_Task Halfway {
    unsigned int id;
    uBaseTask & pgmMain;
    bool spin = true;

    void main() {
        // YOU WRITE THIS CODE.
    }
  public:
    Halfway( unsigned int id, uBaseTask & pgmMain ) : id{ id }, pgmMain{ pgmMain } {};
};
int main() {
    // YOU WRITE THE CODE TO MANAGE 10 Halfway WORKERS.
}
```

Note, yield() polls for non-local exceptions.

The Halfway worker-task loops:

- starting with 100 work units,
- raises a non-local Progress exception at the manager for each batch of 100 work units,
- performs a *yielding busy-wait* until the manager sets spin to **false**,
- advances the total work by 100,
- stops looping after a:
  - Complete exception from the manager indicating work total $\geq$ 1000,
  - Stop exception indicating the worker's units are no longer needed.

The program main:

- allocates 10 worker tasks, and initializes each with a worker id (0-9) and the task address of program main.
- loop performing a yielding busy wait receiving Progress exceptions until 5 workers complete, then loop over the remaining 5 workers raising a Stop exception at each.
- and in the Progress exception handler,
  - with probability 1 in 5, reduces the worker's completed work by 50,
  - checks if a worker should stop working, and if so, sends a Complete exception at the worker.
  - tells the worker to proceed by resetting the shared spin-variable.

No documentation or error checking of any form is required.