UNIVERSITY OF WATERLOO

Final Examination
Term: Winter Year: 2023

CS343
Concurrent and Parallel Programming
Section 001
Instructor: Caroline Kierstead

Thursday, April 13, 2023
Start Time: 19:30 End Time: 22:00
Duration of Exam: 2.5 hours
Number of Exam Pages (including cover sheet): 6
Total number of questions: 24 multiple-choice, 2 short-answer, 2 coding
Total marks available: 77
CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Part A – Multiple Choice

Elided for reuse.

Part B – Short Answer

1. (a) **2 marks** Explain why preventing *synchronization deadlocks* is not practical.

   (b) **2 marks** Briefly explain the difference between *deadlock prevention* and *deadlock avoidance*.

   (c) **1 mark** Name a technique for *mutual exclusion deadlock avoidance*.

   (d) **1 mark** What is the only reasonably practical method for preventing *mutual exclusion deadlock*?

2. Assume a programming language that provides blocking mutual exclusion locks and condition locks:

```
MutexLock m     CondLock c;
m.acquire()     c.wait();
m.release();    c.wait( m ); // atomically block and release m, no reacquire of m on unblock
                c.signal();
                c.empty();
```

Note, MutexLock maybe acquired and released by different threads.

Create a Monitor class that is capable of performing internal scheduling, similar to the one shown below, where methods foo and bar are mutually exclusive with themselves and each other (just like **_Mutex** public methods in μC++ monitors):

```
Monitor M {
    CondQueue bench; // blocking bench
    int value;
  public:
    void foo();
    int bar();
};
```

(a) **3 marks** Write the locking declaration in M and the entry and exit protocol needed to provide mutual exclusion for method foo.

(b) **3 marks** Write the entry and exit protocol needed to provide mutual exclusion for method bar.

(c) **3 marks** Assume a thread enters foo and performs bench.wait(m), where bench is a CondLock and m is the MutexLock providing the monitor mutual-exclusion. Assume a thread now enters bar, performs bench.signal() and immediately returns. Assume signal() returns a boolean value indicating true if a thread is signalled and false otherwise. Write *barging-prevention* code in foo and bar that guarantees the signalled thread acquires the monitor next.

Part C – Long Answer

1. (a) The taxi cab company, Maple Leaf Cabs, has N taxi cabs scattered throughout the city. The dispatcher's job is to take requests from clients for a taxi and to dispatch a taxi to the client for a pickup. The dispatcher also takes requests from taxis for a client at the start of the day and after each client is delivered to their destination.

The interface for the dispatcher monitor is the following (you may not change this interface; you may only add code in the designated areas L1, L2 and L3). **(Do not copy the starting code into your answer booklet.)**

```
_Monitor Dispatcher {
    // communication variables
    int xclient, yclient;                    // client (x,y) coordinates
    int taxiId, clientId;                    // identities of taxi/client pairing

    // L1: ANY VARIABLES NEEDED FOR EACH IMPLEMENTATION

  public: // common interface
    int getTaxi( int id, int x, int y ) {             // called by client
        // L2: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        return taxiId;
    }
    int getClient( int id, int & x, int & y ) {       // called by taxi, x,y in/out parameters
        // L3: ANY SYNCHRONIZATION NEEDED FOR EACH IMPLEMENTATION
        return clientId;
    }
};
```

A client calls the getTaxi routine to ask for a taxi to pick them up at an address given by parameter coordinates (x,y). getTaxi returns the id of the taxi picking up the client, taxiId.

A taxi calls the getClient routine to indicate it is available and tell the dispatcher the taxi's current (x,y) location. The taxi's call to getClient returns the client's (x,y) position in arguments x and y, and returns the id of the client being collected, clientId.

Do not write or create either the taxi or client tasks. You may assume for this question that the dispatcher never shuts down.

Implement the Dispatcher monitor using:

i. **10 marks** external scheduling,

ii. **14 marks** internal scheduling,

iii. **13 marks** implicit (automatic) signalling, using *only* the 3 macros defined below.

Assume the existence of the following preprocessor macros for implicit (automatic) signalling (1(a)iii):

```
#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 **every** public routine of an automatic-signal monitor.

The μC++ uCondition operations are available at the end of the exam.

(b) **25 marks** Write an administrator task to handle the dispatcher's role for the Maple Leaf Taxi company. Figure 1 contains the starting code for the dispatcher-Administrator (you may add only a public destructor and private members). **(Do not copy the starting code into your exam booklet.)**

A client calls the getTaxi routine to ask for a taxi to pick them up at an address given by parameter coordinates (x,y). A *future* taxi is returned immediate to the client, so the client can execute asynchronously (e.g., get ready to leave) before accessing the taxi. When the client accesses the future taxi, they may block because the taxi is not there; otherwise, the client gets into the taxi whose id is in the future.

A taxi calls the getClient routine to indicate it is available and tell the dispatcher the taxi's current (x,y) location. The taxi is dispatched to a client if there is an outstanding request from a client; otherwise, the taxi blocks until a client request is made. The taxi's call to getClient returns with the (x,y) arguments changed to the next client's location.

The dispatcher creates a pool of 5 taxis, and dispatches the nearest taxi to the client's address to minimize client waiting time, if a taxi is available. The taxi constructor is

```
Taxi( MapleLeafTaxi & employer, int id );
```

Routine nearestTaxi is used by the dispatcher to find the nearest available taxi address to the given client address. (Do not write nearestTaxi; just use the member interface in Figure 1.)

When the dispatcher's close routine is called at the end of the day, it prints out the message "Closed for the day", and stops accepting calls to getTaxi. (Assume no outstanding client requests for a taxi after this point.) Then, the dispatcher must deal with outstanding futures to clients that cannot be serviced (no cab will come to service their request), and it must wait for all the taxis to check in before telling them to go home so they can be deleted. Any client with an outstanding taxi-future has the exception Closed inserted into the future, so the client gets this exception raised when it accesses the future. Any waiting or arriving taxi has the exception Closed raised on its stack. The dispatcher must delete any allocated storage before terminating.

Ensure the dispatcher task does as much administration works as possible; a monitor-style solution will receive little or no marks. Write the code for MapleLeafTaxi::main and any necessary declarations/initializations; do **NOT** write the client, taxi, or program main. Assume the program main creates and deletes all the necessary tasks, appropriately, and calls the dispatcher's close routine. 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:

μC++ uCondition operations	C++ list operations
bool empty() *// true if nobody blocked*	**int** size() *// list size*
void wait() *// wait on condition*	**bool** empty() *// size() == 0*
void wait(**int** info) *// wait with info*	T front() *// first element*
bool signal() *// signal condition*	T back() *// last element*
bool signalBlock() *// signal condition*	**void** push_front(**const** T & x) *// add x before first element*
int front() *// return front element info*	**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*

```
_Task MapleLeafTaxiDispatcher {
  public:
    _Event Closed {}; // indicate MapleLeafTaxi closed
    typedef Future_ISM<int> Ftaxi; // future taxi
  private:
    struct LocnClient {
        int id, x, y; // client id and location coordinates
        Ftaxi ftaxi; // future returned to client
        LocnClient( int id, int x, int y ) : id( id ), x( x ), y( y ) {}
    };
    struct LocnTaxi {
        int id, x, y; // taxi id and location coordinates
        uCondition idle;
        LocnTaxi( int id, int x, int y ) : id( id ), x( x ), y( y ) {}
    };
    enum { NoOfTaxi = 5 };
    list<LocnClient *> clients; // client requests for taxis
    list<LocnTaxi *> taxis; // waiting taxis
    int xclient, yclient; // communication variables
    bool closed = false;
  public:
    Ftaxi getTaxi( int id, int x, int y ) { // called by client
        LocnClient *client = new LocnClient( id, x, y ); // create work request
        clients.push_back( client );     // add to request list
        return client->ftaxi; // return future request
    }
    void getClient( int id, int & x, int & y ) { // called by taxi, x,y in/out parameters
        LocnTaxi taxi( id, x, y ); // use the stack!
        taxis.push_back( &taxi ); // add to waiting taxi list
        taxi.idle.wait(); // taxi always blocks
        if ( closed ) _Throw Closed();
        x = xclient; y = yclient; // taxi returns client info
    }
    void close() {} // called at closing time
  private:
    list<LocnTaxi *>::iterator nearestTaxi( LocnClient * node, list<LocnTaxi *> & alist ) {
        // Find the element in parameter alist whose address is closest
        // to the address in parameter node. "alist" must be non-empty.
        // ASSUME THIS ROUTINE IS WRITTEN; DO NOT WRITE IT.
    }
    void main() {
        // YOU WRITE ONLY THIS ROUTINE!!!!
        // allocate taxi tasks
        // dispatch taxis to clients, until close called
        // print closed message
        // mark outstanding futures with Closed exception
        // tell each taxi to go home
        // delete taxi tasks
    }
};
```

Figure 1: Maple Leaf Taxi Dispatcher-Administrator