CS210-001: Assignment 5
Fall 2019 (201930)

Due Date and Time: Tuesday, October 29, 2019 at 11:20 AM

(**100 marks**) Consider an "ordinary" program, whose statements are executed *sequentially* on a single processor such that, at any given time, only one statement is executing, and the next statement does not begin executing until the current statement has finished executing. Consider also, a *concurrent system*, consisting of a set of these ("ordinary") *sequential programs* executing *concurrently* (i.e., at the same time), where each sequential program is executing on a different processor. In a concurrent system, the execution of a statement in one of the sequential programs may "overlap" with the execution of a statement in one or more of the other sequential programs.

If the sequential programs in a concurrent system are executing in parallel, there will not be any order imposed on the execution of statements in the sequential programs if they are executing *independently*. However, in some concurrent systems, the sequential programs may need to exchange information occasionally. When that is the case, the sequential programs need to *synchronize*. Within the context of a concurrent system, there are *two* kinds of *synchronization primitives*: synchronous and asynchronous. Using a *synchronous* approach, the exchange of a message between two sequential programs (i.e., the *sender* and the *receiver*) is an *atomic* operation (i.e., once started, it must be run to conclusion) requiring the participation of both programs (i.e., the programs *rendezvous*). So, if the sender is ready to send but the receiver is not ready to receive, the sender is not allowed to continue executing (i.e., it is *blocked*) until the receiver is ready to receive. Then, the message is sent and received. Similarly, if the receiver is ready to receive but the sender is not ready to send, the receiver is blocked until the sender is ready to send. Then, the message is sent and received. In contrast, using an *asynchronous* approach, a sender is allowed to send a message without blocking and a receiver is allowed to try to receive without blocking. In this way, neither the sender nor the receiver will have to wait if the receiver or sender, respectively, is busy doing something else.

PART 1

In this problem, you are required to write a program that *simulates* an asynchronous communication protocol consisting of two sequential programs: (1) a *sender* and (2) a *receiver*. The sender will open the file to send (i.e., the input file) and the receiver will open the file to receive (i.e., the output file). The sender will have *two* queues: (1) an *array* containing data (i.e., *data* messages) that are ready to send to the receiver (call it the *sender's output queue*), and (2) a *linked list* containing acknowledgements (i.e., *ack* messages) received from the receiver (call it the *sender's input queue*). The receiver will also have *two* queues: (1) a *linked list* containing *data* messages received from the sender (call it the *receiver's input queue*), and (2) an *array* containing *ack* messages that are ready to send to the sender (call it the *receiver's output queue*). The queues should be defined as C++ classes. Your C++ classes should be implementations of the queue algorithms given on the CS210 Algorithms web page. Put the queue class specifications

and implementations in separate `.h` and `.cpp` files, respectively. Put the client program in separate `.h` and `.cpp` files. Do not waste your time implementing any algorithms that you don't actually need.

The sizes of the queues will be determined by the following *four* global constants: (1) `SENDERS_OUTPUT_QUEUE_SIZE = 32`, (2) `SENDERS_INPUT_QUEUE_SIZE = 4`, (3) `RECEIVERS_INPUT_QUEUE_SIZE = 8`, and (4) `RECEIVERS_OUTPUT_QUEUE_SIZE = 16`.

The `data` and `ack` messages have the same format and should be described by a `struct` containing *two* members: (1) a message number defined as an `int` and (2) a single character of data defined as a `char`.

The simulation will have a `globalClock` variable whose value will determine the order of send/receive events, and a time stamp will be generated to indicate when send and receive events should be initiated by the sender and receiver. Thus, *four* time stamp variables are required: (1) `senderTimeToSend`, (2) `senderTimeToReceive`, (3) `receiverTimeToReceive`, and (4) `receiverTimeToSend`. When the simulation begins, the `globalClock` will be set to `0` and the four time stamps will be initialized with random values between `1` and `100`, inclusive. The file `timeTilNextEvent.cpp` from the directory `/home/venus/hilder/cs210/assignment5/datafiles` contains a random number generator that is already set up to generate numbers between `1` and `100`, inclusive. Use it to create your own random number generator function.

PART 2

Let's look at an example of how the simulation should work. Assume `senderTimeToSend`, `senderTimeToReceive`, `receiverTimeToReceive`, and `receiverTimeToSend` are set to `2`, `10`, `2`, and `7`, respectively. The `globalClock` will then begin "ticking". Each time the `globalClock` ticks, the value of each time stamp variable will be compared to the `globalClock` to determine whether a send/receive event must be processed. So, when the `globalClock` value is `1`, no send/receive events will be processed because none are scheduled. When the `globalClock` value is `2`, there are two events scheduled: a send event by the sender and a receive event by the receiver. The receiver will attempt to dequeue a *data* message from its input queue. If the receiver was able to dequeue a *data* message from its input queue, it will write the character received to the output file and enqueue an *ack* message in its output queue. "Simultaneously", the sender will read a character from the input file and enqueue a *data* message in its output queue. Then, the sender will enqueue *data* messages in the receiver's input queue until its (i.e., the sender's) output queue is empty or until the receiver's input queue is full. Once all send/receive events scheduled to occur at time `2` have been processed, new values will generated for the `senderTimeToSend` and `receiverTimeToReceive` time stamps. A new time stamp will be equal to the current value of the `globalClock` *plus* a random value between `1` and `100`, inclusive.

Assume values of 12 and 15 are generated for the sender and receiver, respectively. Thus, the new senderTimeToSend is 14 and the new receiverTimeToReceive is 17. To continue with the example, each time the globalClock ticks for the values 3 to 6, no send/receive events will be processed because none are scheduled. When the globalClock value is 7, the receiver will enqueue *ack* messages in the sender's input queue until its (i.e., the receiver's) output queue is empty or until the sender's input queue is full. Then, a new value will be generated for the receiverTimeToSend time stamp (assume 20). Both times the globalClock ticks for the values 8 and 9, no send/receive events will be processed because none are scheduled. When the globalClock value is 10, the sender will attempt to dequeue an *ack* message from its input queue. If the sender was able to dequeue an *ack* message from its input queue, the receive event is done. Then, a new value will be generated for the senderTimeToReceive time stamp.

PART 3

In pseudocode, the general algorithm is given as follows.

```
AsynchronousProtocolSimulation (inputFile, outputFile)

// Initialize the global clock.
globalClock = 0

// Generate the times for the next send and receive events.
senderTimeToSend = GenerateTimeToNextEvent ()
senderTimeToReceive = GenerateTimeToNextEvent ()
receiverTimeToReceive = GenerateTimeToNextEvent ()
receiverTimeToSend = GenerateTimeToNextEvent ()

// Initialize flags.
senderSendStatus = NO_SEND
senderReceiveStatus = NO_RECEIVE
receiverReceiveStatus = NO_RECEIVE
receiverSendStatus = NO_SEND

// This loop will continue until all the messages have been
// sent and received.
while 1

    // If there are no more messages to send and all queues
are empty,
    // it is time to end the simulation.
    if !inputFile and senderOutputQueue.IsEmpty () and
receiverInputQueue.IsEmpty ()
            and receiverOutputQueue.IsEmpty () and
senderInputQueue.IsEmpty ()
```

```
        break

    // The global clock ticks once.
    globalClock ++

    // Determine whether any send/receive events are
scheduled.
    // If so, set the appropriate flag and generate the time
for
    // the next event.
    if globalClock == senderTimeToSend
        senderSendStatus = SEND
        senderTimeToSend = globalClock +
GenerateTimeToNextEvent ()
    if globalClock == senderTimeToReceive
        senderReceiveStatus = RECEIVE
        senderTimeToReceive = globalClock +
GenerateTimeToNextEvent ()
    if globalClock == receiverTimeToReceive
        receiverReceiveStatus = RECEIVE
        receiverTimeToReceive = globalClock +
GenerateTimeToNextEvent ()
    if globalClock == receiverTimeToSend
        receiverSendStatus = SEND
        receiverTimeToSend = globalClock +
GenerateTimeToNextEvent ()

    // The simulation needs to ensure that messages are not
received
    // before they are sent. So, we assume that if a message
is sent
    // at time t, then it cannot be received until, at a
minimum,
    // time t+1. That is, we assume that sending and
receiving a
    // message does not happen instantaneously.
Consequently, all
    // receive events are processed before any send events.

    // Step 1: We dequeue a data message from the receiver's
input queue.
    // Notice that we don't enqueue the acknowledgement
message yet. That
    // will be done in Step 5. Then the message is saved.
    if receiverReceiveStatus == RECEIVE
        if !receiverInputQueue.IsEmpty ()
            receiverDataMsg = receiverInputQueue.Dequeue ()
```

```
            outputFile.WriteOutputFile (receiverDataMsg)
        else
            receiverReceiveStatus = NO_RECEIVE

    // Step 2: We dequeue an ack message from the sender's
input queue.
    // This step and Step 1 could be interchanged because
they don't
    // affect each other in any way.
    if senderReceiveStatus == RECEIVE
        if !senderInputQueue.IsEmpty ()
            senderAckMsg = senderInputQueue.Dequeue ()
        senderReceiveStatus = NO_RECEIVE

    // Step 3: We read a message to send and enqueue a data
message in
    // the sender's output queue. Then, the sender tries to
enqueue as
    // many data messages as it can in the receiver's input
queue.
    if senderSendStatus == SEND
        if !senderOutputQueue.IsFull ()
            if inputFile
                senderDataMsg = inputFile.Read ()
                if inputFile
                    senderDataMsg = PrepareNextDataMessage
(senderDataMsg)
                    senderOutputQueue.Enqueue
(senderDataMsg)
        while !senderOutputQueue.IsEmpty () and !
receiverInputQueue.IsFull ()
            senderDataMsg = senderOutputQueue.Dequeue ()
            receiverInputQueue.Enqueue (senderDataMsg)
        senderSendStatus = NO_SEND

    // Step 4: The receiver tries to enqueue as many ack
messages as
    // it can in the sender's input queue. This step and
Step 3 could be
    // interchanged because they don't affect each other in
any way.
    if receiverSendStatus == SEND
        while !receiverOutputQueue.IsEmpty () and !
senderInputQueue.IsFull ()
            receiverAckMsg = receiverOutputQueue.Dequeue ()
            senderInputQueue.Enqueue (receiverAckMsg)
        receiverSendStatus = NO_SEND
```

```
    // Step 5: If a data message was received in Step 1,
enqueue an
    // acknowledgement in the receiver's output queue.
    if receiverReceiveStatus == RECEIVE
        receiverAckMsg = PrepareNextAckMessage
(receiverDataMsg)
        receiverOutputQueue.Enqueue (receiverAckMsg)
        receiverReceiveStatus = NO_RECEIVE

return
```

PART 4

This section describes the characteristics of the output from your simulation program.

Prior to processing any send and receive events, the simulator should output to the screen the current value of the global clock. For example,

```
Global Clock: 99
```

After the sender enqueues a *data* message in the receiver's input queue, the simulator should output to the screen the contents of the message sent. For example,

step 3

```
Sender: Sent [10,d]
```

After the receiver dequeues a *data* message from its input queue, the simulator should output to the screen the contents of the message received. For example,

step 5

```
Receiver: Received [10,d]
```

After the receiver enqueues an *ack* message in the sender's input queue, the simulator should output to the screen the contents of the message sent. For example,

step 4

```
Receiver: Sent [10,d]
```

After the sender dequeues an *ack* message from its input queue, the simulator should output the screen the contents of the message received. For example,

step 2

```
Sender: Received [10,d]
```

The following sequence of *data* and *ack* message exchanges between the sender and receiver should give you some idea about how to format your output.

.
.
.

```
Global Clock: 99
    Sender: Sent [10,d]
  Receiver: Sent [9,n]
Global Clock: 102
  Receiver: Received [10,d]
Global Clock: 103
  Receiver: Sent [10,d]
Global Clock: 108
    Sender: Received [9,n]
    Sender: Sent [11, ]
Global Clock: 110
  Receiver: Received [11, ]
  Receiver: Sent [10,d]
Global Clock: 117
    Sender: Received [10,d]
.
.
.
```

PART 5

When you are ready to demonstrate that your program works, use the file `wordCount.gauss` from the directory `/home/venus/hilder/cs210/assignment5/datafiles` as input to your program. After your simulation is complete, compare the contents of the input file read by the sender to the output file written by the receiver. At the command prompt, enter

`diff` *your_input_file_name your_output_file_name*

The `diff` command compares two files to see where they differ. If there is no difference, the command prompt will appear. If there is a difference, the lines that differ will be indicated before the command prompt appears.

WHAT TO SUBMIT

If you are working alone, submit to UR Courses: (1) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles` and (2) a script file called `part5script` showing the compilation and execution of your program.

If you are working with a partner, *one of the partners* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%), (2) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles`, and (3) a script file called `part5script` showing the compilation and execution of your program. *The other partner* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative

contribution of the partners (should total 100%). Note that *both* partners need to submit the `partners` file.