CS 343 Fall 2012 – Assignment 2
Instructor: Martin Karsten
Due Date: Tuesday, October 9, 2012 at 22:00
Late Date: Thursday, October 11, 2012 at 22:00

September 26, 2012

This assignment introduces full-coroutines and concurrency in μC++. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution, i.e., writing a C-style solution for questions is unacceptable, and will receive little or no marks. (You may freely use the code from these example programs.)

1. A *token-ring algorithm* is used in a distributed system to control access to a shared transmission channel. Multiple nodes, called *stations*, communicate with each other using a shared channel, which forms a ring among stations. Stations can only communicate unidirectionally with each other using the ring, i.e, a station can only directly transmit data to its immediate successor in the ring. A *token* is passed between stations around the ring and a station can only send data when it holds the token. There is only one token in the system, which is passed around the ring until a station has data to send. Write a simulation where each station is represented by a full coroutine. Assume all of the coroutines are identical, distinguishable only by their identification number (id), so there is only one type of full coroutine. Coroutines are structured around the ring in increasing order by their id numbers.

   A token ring with 8 stations is illustrated in Figure 1. The ring represents the shared transmission channel. The unit of transmission is called a *frame*. The frame is currently being transmitted from Station 4 to Station 5, clockwise around the ring. There are three types of frames: *token*, *data*, and *ack* (for acknowledgment). A token frame represents the right to send, subject to the priority mechanism explained below. When a station receives the token frame, it can conceptually hold the token and then send a data frame. The data frame carries the source and destination id numbers of the sender and receiver stations. This data frame is then transmitted around the ring until the destination station is reached. The destination station conceptually removes the data frame and immediately generates an acknowledgment (ack) frame that is sent around the ring back to the sender by swapping the source and destination fields in the data frame. After receiving an ack frame, the source station must forward the token to its neighbour and wait for the token to return (after at least one transmission round), before transmitting another data frame or making another reservation (see below). A station generates data frames for transmission from external *send requests* that it receives at creation. A send request is *pending* when the request cannot be immediately transmitted around the ring.

   The token-ring algorithm has a priority mechanism to control traffic around the ring. All frames carry a priority field, which is used to reserve the channel for pending send-requests with a certain priority. Each send request

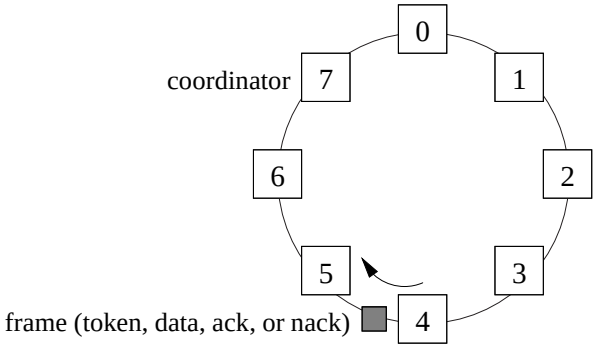

Figure 1: Token Ring with 8 Stations

arrives with a specified priority. When a station has a pending send-request, it compares the request priority with the current priority in the incoming frame and uses the following rules:

(a) If the frame is a data or passing ack frame (destined to another station) and the frame priority is less than the request priority, the station saves the frame priority value (including 0) and replaces it with its request priority when forwarding the frame.

(b) If the frame is a token frame and the frame priority is less or equal than the request priority, the station can send its data frame. The data frame is sent with the stored value (and the stored value is deleted). Note that this rule can never increase the frame priority.

(c) Otherwise, the station must not change the frame priority.

The rules for a pending request are summarized in the following table:

Frame Priority	Frame Type		
	Token	Data or Passing Ack	Ack To Self
higher than request	wait	wait	wait
equal to request	send	wait	wait
lower than request	send	reserve	wait

In this simulation, time is measured in transmission rounds, starting from 0. Each station increments its local round counter whenever a frame (or the combination of data + ack frame) makes a full transmission round and arrives back at the station. In combination with the priority mechanism, this means frames are not always sent during the requested round.

The executable program is named tokenring and has the following shell interface:

```
tokenring S [ F ]
```

S is the number of stations (coroutines) in the ring and must be a positive integer. Station Ids are numbered 0..S-1. The program must support 1-100 stations. F is an optional file name for reading send requests. If the file name is not present, input must be read from standard input.

Write a *full coroutine* with the following interface (you may add only a public destructor and private members):

```
_Coroutine Station {
  private:
    struct Frame {
        enum { Token, Data, Ack } type;          // type of frame
        unsigned int src;                        // source id
        unsigned int dst;                        // destination id
        unsigned int prio;                       // priority
    } frame;

    void data( Frame frame );                    // pass frame
    void main();                                 // coroutine main
  public:
    Station( unsigned int id );
    void setup( Station *nexthop );              // supply next hop
    void sendreq( unsigned int round, unsigned int dst,
                  unsigned int prio );           // store send request
    void start();                                // inject token and start
}; // Station
```

To form the ring of stations, uMain::main calls the setup member for each station to pass a pointer to the coroutine's next hop. After uMain::main initializes the ring, it reads all send requests from the given input source and calls sendreq to store send requests at the respective source station. Each station must store all its send requests at the beginning and work through them during the simulation. After processing the input file, uMain::main calls the start member of Station 0 to start the simulation. The start member creates a token frame with priority 0 and activates (resumes) the coroutine. Each station then performs the protocol processing by calling the data

member of its successor with a frame, and the data member must resume the coroutine for processing the frame (i.e., protocol processing must be done on each coroutine's stack).

The input is provided in the following format:

```
<round> <src> <dst> <prio>
```

with

```
<round> transmission round
<src>   source id
<dst>   destination id
<prio>  priority
```

An example is given below:

```
1 1 3 0
1 2 1 0
1 3 1 4
10 4 2 7
```

For Round 1, Station 1 has a send request to Station 3 with priority 0. For Round 1, Station 2 has a send request to Station 1 with priority 0. For Round 1, Station 3 has a send request to Station 1 with priority 4. For Round 10, Station 4 has a send request to Station 2 with priority 7.

Because the request at Station 3 has a higher priority than the request at Station 2, it is served first.

Output is written to standard output and shows a dynamic log of transmitted messages (one message per station per line):

```
round <round> station <id> sends with prio <prio> <type> [from <src>] [to <dst>]
```

with

```
<round> transmission round
<id>    station id
<prio>  priority
<type>  'token', 'data', or 'ack'
<src>   source id (only printed for 'data' and 'ack' types)
<dst>   destination id (only printed for 'data' and 'ack' types)
```

The output for the above input example with 4 stations is:

```
round 0 station 0 sends with prio 0 token
round 0 station 1 sends with prio 0 token
round 0 station 2 sends with prio 0 token
round 0 station 3 sends with prio 0 token
round 1 station 0 sends with prio 0 token
round 1 station 1 sends with prio 0 data from 1 to 3
round 1 station 2 sends with prio 0 data from 1 to 3
round 1 station 3 sends with prio 4 ack from 3 to 1
round 2 station 0 sends with prio 4 ack from 3 to 1
round 2 station 1 sends with prio 4 token
round 2 station 2 sends with prio 4 token
round 2 station 3 sends with prio 0 data from 3 to 1
round 3 station 0 sends with prio 0 data from 3 to 1
round 3 station 1 sends with prio 0 ack from 1 to 3
round 3 station 2 sends with prio 0 ack from 1 to 3
round 3 station 3 sends with prio 0 token
round 4 station 0 sends with prio 0 token
round 4 station 1 sends with prio 0 token
round 4 station 2 sends with prio 0 data from 2 to 1
round 4 station 3 sends with prio 0 data from 2 to 1
round 5 station 0 sends with prio 0 data from 2 to 1
round 5 station 1 sends with prio 0 ack from 1 to 2
round 5 station 2 sends with prio 0 token
round 5 station 3 sends with prio 0 token
```

Devise a mechanism to properly hand back control to uMain::main after the last acknowledgment has been received by the last sender. Control must be handed back to uMain::main during the next transmission round after receiving the last ack frame. uMain::main must delete all dynamically allocated memory.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. Use of execution state variables in a coroutine usually indicates that you are not using the ability of the coroutine to remember execution location. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 4.3.1 in *Understanding Control Flow: with Concurrent Programming using µC++* for details on this issue.

2. Multiplying two matrices is a common operations in many numerical algorithms. Matrix multiply lends itself easily to concurrent execution because data can be partitioned, and each partition can be processed concurrently without interfering with tasks working on other partitions (divide and conqueror).

   (a) Write a concurrent matrix-multiply with the following interface:

   ```
   void matrixmultiply( int *Z[], int *X[], int xr, int xc, int *Y[], int yc );
   ```

   which calculates $Z_{xr,yc} = X_{xr,xcyr} \cdot Y_{xcyr,yc}$, where matrix multiply is defined as:

   $$X_{i,j} \cdot Y_{j,k} = \left(\sum_{c=1}^{j} X_{row,c} Y_{c,column} \right)_{i,k}$$

   Create a task to calculate each row of the Z matrix from the appropriate X row and Y columns. All matrices in the program are variable sized, and hence, allocated dynamically on the heap.

   The executable program is named matrixmultiply and has the following shell interface:

   ```
   matrixmultiply xrows xcols-yrows ycols [ X-matrix-file Y-matrix-file ]
   ```

   - The first three parameters are the dimensions of the $X_{xr,xcyr}$ and $Y_{xcyr,yc}$ matrices.
   - If specified, the next two parameters are the X and Y input files to be multiplied. Each input file contains a matrix with appropriate values based on the dimension parameters; e.g., the input file:

     ```
     1   2   3   4
     5   6   7   8
     9  10  11  12
     ```

     is a 3×4 matrix. Assume there are the correct number of input values in each matrix file and all matrix values are correctly formed. After reading in the two matrices, multiply them, and print the product on standard output using this format:

     ```
     % matrixmultiply 3 4 3 xfile yfile
                                          |       1       2       3
                                          |       4       5       6
                                          |       7       8       9
                                          |      10      11      12
     -----------------------------------------*---------------------------
           1       2       3       4      |      70      80      90
           5       6       7       8      |     158     184     210
           9      10      11      12      |     246     288     330
     ```

     Where the matrix on the bottom-left is X, the matrix on the top-right is Y, and the matrix on the bottom-right is Z.
   - If no input files are specified, create the appropriate X and Y matrices with each value initialized to 37, multiply them, ***but print no output***. This case is used for timing the cost of parallel execution.

   Print an appropriate error message and terminate the program if there are an invalid number of arguments, the dimension values are less than one, or unable to open the given input files.

   (b) i. Test for any benefits of concurrency by running the program in parallel:
   - Put the following declaration after the arguments have been analysed:

     ```
     uProcessor p[xrows - 1] __attribute__(( unused ));  // number of CPUs used
     ```

This declaration allows the program to access multiple virtual CPUs (cores). One virtual CPU is used for each task calculating a row of the Z matrix. The program starts with one virtual CPU so only xrows - 1 additional CPUs are necessary. Compile the program with the μC++ -multi flag and no optimization.

- Run the program on a multi-core computer with at least 16 or more actual CPUs (cores) (e.g., undergraduate machines linux024, linux028, and linux032 each have 48 CPUs), with arguments of xrows in the range [1,2,4,8,16] with xcols-yrows of 5000 and ycols of 10000.
- Time each execution using the time command:

```
% time ./a.out
3.21u 0.02s 0:03.32 100.0%
```

  Output from time differs depending on your shell, but all provide user, system and real time. (Increase the number of xcols-yrows and ycols if the timing results are below .1 second.)
- Include all 5 timing results to validate your experiments.

ii. Comment on the user and real times for the experiments.

Submission Guidelines

Please follow these guidelines very carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text file, i.e., *.*txt file, must be ASCII text and not exceed 500 lines in length, where a line is 120 characters.** Programs should be divided into separate compilation units, i.e., *.{h,cc,C,cpp} files, where applicable. Use the submit command to electronically copy the following files to the course account.

1. q1*.{h,cc,C,cpp} – code for question 1, p. 1. **Program documentation must be present in your submitted code. Output for this question is checked via a marking program, so it must match exactly with the given program.**

2. q1tokenring.testtxt – test documentation for question 1, p. 1, which includes the input and output of your tests, documented by the use of script and after being formatted by scriptfix. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

3. q2*.{h,cc,C,cpp} – code for question question 2a. **Program documentation must be present in your submitted code. No test, user or system documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

4. q2matrixmultiply.txt – contains the information required by question 2b.

5. Use the following Makefile to compile the programs for question 1, p. 1 and 2a:

```
OPT:=

CXX = u++                                       # compiler
CXXFLAGS = -g -multi -Wall -Wno-unused-label -MMD ${OPT} # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = # object files forming 1st executable with prefix "q1"
EXEC1 = tokenring                               # 1st executable name

OBJECTS2 = # object files forming 2nd executable with prefix "q2"
EXEC2 = matrixmultiply                              # 2nd executable name

OBJECTS = ${OBJECTS1} ${OBJECTS2}               # all object files
DEPENDS = ${OBJECTS:.o=.d}                      # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC2}                       # all executables

#############################################################################
```

```
.PHONY : all clean

all : ${EXECS}                                  # build all executables

${EXEC1} : ${OBJECTS1}                          # link step 1st executable
    ${CXX} ${CXXFLAGS} $^ -o $@

${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@

#############################################################

${OBJECTS} : ${MAKEFILE_NAME}                   # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}                             # include *.d files containing program dependences

clean :                                         # remove files that can be regenerated
    rm -f *.d *.o ${EXECS} ImplType
```

This makefile is used as follows:

```
$ make tokenring
$ tokenring . . .
$
$ make matrixmultiply
$ matrixmultiply . . .
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make tokenring or make matrixmultiply in the directory to compile the programs. This Makefile must be submitted with the assignment to build the program, so it must be correct. Use the web tool Request Test Compilation to ensure you have submitted the appropriate files, your makefile is correct, and your code compiles in the testing environment. **If the makefile fails or does not produce correctly named executables, or if a program does not compile, you receive zero for all "Testing" marks.**

Follow these guidelines. Your grade depends on it!