# CS 343 Winter 2012 – Assignment 2
# Instructor: Peter Buhr
# Due Date: Monday, January 30, 2012 at 22:00
# Late Date: Wednesday, February 1, 2012 at 22:00

January 27, 2012

This assignment introduces full-coroutines and concurrency in $\mu$C++. 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. One of the stations in the ring is designated as the *coordinator*; the coordinator is used to implement various centralized work, such as managing deleted resources.

   A token ring with 8 stations is illustrated in Figure 1; station 7 is the coordinator. The ring represents the shared transmission channel. The unit of transmitted is called a *frame*. The frame is currently being transmitted from Station 4 to Station 5, clockwise around the ring. There are four types of frames: *token*, *data*, *ack* (acknowledgment) and *nack* (negative acknowledgment). A token frame represents the right to send. When a station receives the token frame, it can conceptually hold the token and then send a data frame. The data frame carries a value, and 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.

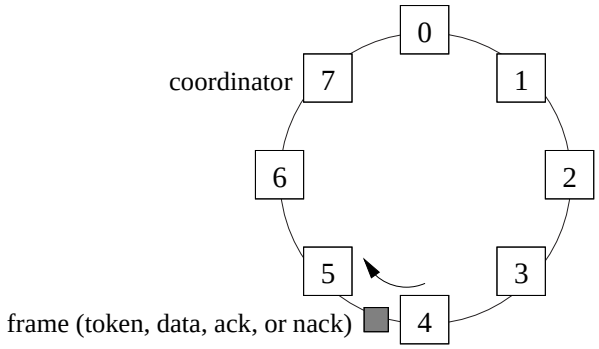


Figure 1: Token Ring with 8 Stations

   A station has a random chance of failing or becoming disconnected. When a station determines its partner has failed, it passes the address of the failed station to the coordinator. The coordinator deletes the failed station, and returns the address of the failed station's partner. (This delete routine is simple, and hence, does not resume the coordinator.) As well, if a frame's destination is to a failed partner:

   - converted to a token ~~the frame is dropped~~, if it is an ack or nack (i.e., no station to receive it),

- converted to a nack if it is a data frame to a source other than the failed partner by swapping the source and destination fields (i.e., data cannot be delivered so let the source know),
- converted to a token if it is a data frame where the source is equal to the failed destination (i.e., frame sent from source to itself), so the entire communication is over and a new one starts.

Finally, if a station receives a data frame but the frame's source is the failed partner, there is no reason to send an ack so the frame is converted to a token. Should the coordinator fail, a new coordinator is elected among the remaining stations. An election involves circling the ring to find the remaining station with the highest id; this station becomes the new coordinator.

Write a *full-coroutine* simulation of a token ring where each station is represented by a coroutine with the following interface (you may add only a public destructor and private members):

```
_Coroutine Station {
    _Event Election {
      public:
        Station &candidate;
        Election( Station &candidate ) : candidate( candidate ) {}
    };
    static unsigned int value;                  // global variable, increasing transmission value
    static std::vector<Station *> active;       // global variable, active stations
    Station *remove( Station *victim );         // delete failed object and return new partner
    void vote( Station &candidate );            // perform election voting
    // ADD OTHER PRIVATE MEMBERS
    void main() {
        try {
            _Enable {                           // allow delivery of nonlocal exceptions
                suspend();                      // establish starter for termination
                ...                             // YOU WRITE THIS CODE
            } // _Enable
        } _CatchResume( Election &election ) ( ADD NECESSARY ARGUMENTS ) {
            ...                                 // YOU WRITE THIS CODE
        } // try
    }
  public:
    static Station *coordinator;                // global variable, shared among all instances
    struct Frame { YOU WRITE THIS CODE };
    Station( unsigned int id, unsigned int failure );
    void start( Station *partner );             // supply partner
    void transmit( const Frame &f );            // pass frame
    unsigned int id() const;                    // station id
    Station *partner() const;                   // station partner
    bool failed() const;                        // station status
};
```

Assume all of the stations are identical, distinguishable only by their identification number (id), so there is only one station type. Coroutines are structured around the ring in increasing order by their id numbers. To simplify the implementation, three global variables are used, which cannot exist in a distributed system as data cannot be shared except through the communication channel. (Without these global variables, the communication protocol must be significantly more complex.) The global variable value is the value a station places in its data frame; value starts at zero and is increment by a station *before* placing it in a data frame for transmission; hence, the values passed around the ring are just increasing positive integers. The global variable active is a vector of the active (non-failed) stations. This array allows a station to always select an active station as the destination of a data frame; otherwise, more a complex protocol is necessary to inform all stations about a failed station. A station adds itself to active when created, and then only reads the vector to find a valid destination; the coordinator is the only station allowed to remove a failed station from the vector. The global variable coordinator is the station currently acting as the coordinator. This variable allows a station to communicate directly with the coordinator; otherwise, a more complex protocol is necessary to interact with the coordinator. The variable coordinator is updated after the coordinator fails and an election selects a new coordinator.

The constructor is passed the station id and the range of random values for computing a station failure. A station

receives its partner in the ring via the start member. A station receives a frame for analyse via the transmit member. A station returns its id via the id member. A station returns its partner via the partner member. A station returns its status via the failed member, where **true** indicates the station has failed.

***After*** a station is resumed by transmit, it first checks if its partner has failed. If the partner failed, the frame is first analyzed and possibly changed. Then, if the failed partner is not the coordinator, the coordinator's remove member is called, via coordinator, which removes the station from active, deletes the failed station, and returns that station's partner. Note, this new partner may also be failed, so the check may be repeated. (Assert in your program, during each iteration of the search for a new partner, that the searching task is not failed.) If the failed partner is the coordinator, an election is performed. An election is composed of each station performing a resumption **_Resume** raise of an Election exception at the station's partner (skipping the coordinator since it is failed) and then calling the partner's vote member. (You decide if the exception is raised inside or outside of the vote member, which may make the parameter to vote necessary or unnecessary.) The vote member ***must*** perform a resume for the Election exception to be propagated. When a station receives an election exception, it compares its id with the id of the station in the exception, and the station with the larger id is placed in an exception that is raised at the next station. During an election, a failed station does not participates in the election but still raises the current election status at its partner. Once the exception control-flow has completely circulated the ring, the station with the highest id is assigned as the new coordinator. (Assert in your program that an elected coordinator is not failed.)

***After*** dealing with all failed partners, the station computes if it will fail by generating a random value between 0 and failure - 1, where 0 implies failed; this new failed status is returned from member failed. Regardless of the result of the new failed status, a station continues normally, i.e., it processes the frame passed to it. A station's failure is *only* detected by its partner the *next* time around the cycle, who then performs the appropriate action.

Finally, a station processes the frame it received, which may have changed when processing the failed partners. When a station receives the token, it must decide to hold the token and pass data or pass the token. This decision is simulated by each station having a counter, which is randomly initialized to 0 or 1 at the start of Station::main before the first suspend). When the counter is great than 0, the station decrements the counter and then forwards the token frame to its partner. When the counter is equal to 0, the station resets the counter to a random value between 1 and 3, and creates a data frame. To create a data frame, a station randomly chooses a destination from the remaining stations using the vector active, i.e., a random index between 0 and active.size() - 1, and increments the global value variable for the frame's value.

To form the ring of stations, uMain::main calls the start member for each station to link the stations together in reverse order; i.e., N-2 links to N-1, N-3 links to N-2, ... 0 links to 1, and the cycle is closed by linking N-1 to 0. The start routine also resumes the station to set uMain as its starter (needed during termination). The main routine of each station suspends back immediately so the next station can be started. After uMain initializes the ring, it sets the coordinator to the station with id 0, and transmits the token frame to it to start the simulation.

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

```
tokenring  S [ F [ Seed ] ]
```

S is the number of stations in the ring ($> 1$). F is a 1 in F chance of a station failing ($> 1$). If F is unspecified, use the value 15. Seed is a value for initializing the random number generator using routine ~~srand~~ PRNG::seed ($> 0$). When given a specific seed value, the program must generate the same results for each invocation. If the seed is unspecified, use a random value like the process identifier (getpid) or current time (time), so each run of the program generates different output. All random rolls (1 in *N* chance) are generated using prng(N-1) == 0. Check all command arguments for correct form (integers) and range; print an appropriate usage message and terminate the program if a value is missing or invalid.

To obtain repeatable results, all random numbers are generated using class PRNG and a single global instance of this class. There are a total of four calls to obtain random values all in Station::main: initialize the token counter, compute the failed flag, reset the token counter, compute the data destination (in that order).

Output shows the dynamic display of the ring behaviour:

```
1  $ tokenring 5 7 17844
2  0(skip) 1(send 1, to 0) 2(for 0, 1) 3(for 0, 1) 4(for 0, 1) 0(rec 1, from 1) 1(ack)
3  2(skip) 3(send 2, to 0) 4(for 0, 2) 0(rec 2, from 3) 1(ack 3) 2(ack 3) 3(ack)
4  4(skip) 0(send 3, to 4) 1(for 4, 3) 2(for 4, 3) 3(for 4, 3) 4(rec 3, from 0) 0(fail 1, part 2) 0(ack)
5  2(send 4, to 3) 3(rec 4, from 2) 4(fail 0, part 2) 4(elec 4?2 4?3 win 4) 4(ack 2) 2(ack)
6  3(skip) 4(send 5, to 2) 2(rec 5, from 4) 3(ack 4) 4(ack)
7  2(skip) 3(send 6, to 2) 4(for 2, 6) 2(rec 6, from 3) 3(fail 4, part 2) 3(elec 3?2 win 3) 3(ack)
8  2(skip) 3(skip) 2(skip) 3(skip) 2(send 7, to 3) 3(rec 7, from 2) 2(ack)
9  3(send 8, to 2) 2(rec 8, from 3) 3(ack)
10 2(skip) 3(skip) 2(send 9, to 2) 3(for 2, 9) 2(rec 9, from 2) 3(ack 2) 2(ack)
11 3(fail 2, part 3) 3(send 10, to 3) no more partners
```

Each action begins with the station id followed by parenthesis information about the action occurring at that station:

| | |
|---|---|
| send *value*, to *destination* | send the value to the destination id |
| for *destination*, *value* | forward to the destination id the value |
| rec *value*, from *source* | received value from source id |
| ack *[ value ]* | sending to or received Ack from destination |
| nack *[ value ]* | sending to or received Nack from destination |
| skip | skip this round |
| fail *old-partner*, part *new partner* | failure of old-partner id, now using new-partner id |
| drop Ack | delivery of Ack to failed source |
| drop Nack | delivery of Nack to failed source |
| drop Data | delivery of Data to failed destination |
| elec *id*?*id* *id*>*id* ... win *id* | run an election showing the pairs of stations at each vote and the winner, where "?" indicates a vote and ">" indicates skipping a failed station |
| no more partners | only one station |

Start a newline after a transmission is finalized by: Ack, drop Ack, drop Data if Nack cannot be returned, no more partners. For example, line 5 in the example output reads:

**2(send 4, to 3)** : station 2 sends value 4 to station 3

**3(rec 4, from 2)** : station 3 receives value 4 from station 2 and sends an Ack to the message source, station 2.

**4(fail 0, part 2)** : station 4 detects its partner, station 0, has failed and makes station 0's partner its new partner, which is station 2

**4(elec 4?2 4?3 win 4)** : station 4 detects that failed partner, station 0, was the coordinator so it starts an election, which has two votes between stations 4 and 2, and 4 and 4, where station 4 wins the election

**4(ack 2)** : station 4 sends the acknowledgement on to station 2

**2(ack)** : station 2 receives the acknowledgement from the destination station 3

**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. (a) Quicksort is one of the best sorting algorithms in terms of execution speed on randomly arranged data. It also lends itself easily to concurrent execution by partitioning the data into those greater than the pivot and those less than the pivot so each partition can be sorted independently and concurrently by another task.

   Write an in-place concurrent quicksort with the following public interface (you may add only a public destructor and private members):

```
template<typename T> _Task QuickSort {
  public:
    QuickSort( T array[], int low, int high, unsigned int depth );
};
```

that sorts an array of values into ascending order. Choose the pivot as follows:

```
pivot = array[low + ( high - low ) / 2];
```

A naïve conversion of a sequential quicksort to a concurrent quicksort partitions the data values as normal, but instead of recursively invoking quicksort on each partition, a new quicksort task is created to handle each partition. (For this discussion, assume no other sorting algorithm is used for small partitions.) However, this approach creates a large number of tasks: approximately $2 \times N$, where $N$ is the number of data values. The number of tasks can be reduced to approximately $N$ by only creating a new quicksort task for one partition and recursively sorting the other partition in the current quicksort task. In general, creating many more tasks than processors significantly reduces performance (try an example to see the effect) due to contention on accessing the processors versus any contention in the program itself. The only way to achieve good performance for a concurrent quicksort is to significantly reduce the number of quicksort tasks via an additional argument that limits the tree depth of the quicksort tasks. The depth argument is decremented on each recursive call and tasks are only created while this argument is greater than zero; otherwise sequential recursive-calls are use to sort each partition. Use a starting depth value of $\lfloor \log_2 depth) \rfloor$.

Recursion can overflow a task's stack, since the default task size is only 32K or 64K bytes in $\mu$C++. To check for stack overflow, call verify() at the start of the recursive routine, which checks if the call is close to or exceeds the task's stack-size. If warnings or an error is generated by verify, globally increase the stack size of all tasks by defining routine:

```
unsigned int uDefaultStackSize() {
    return 512 * 1000;          // set task stack-size to 512K
}
```

Finally, to maximize efficiency, quicksort tasks must not be created by calls to **new**, i.e., no dynamic allocation is necessary the quicksort task.

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

```
quicksort unsorted-file [ sorted-file | -depth ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Add the following declaration to uMain::main after checking command-line arguments but before creating any tasks:

```
uProcessor processors[ depth - 1 ] __attribute__(( unused )); // use "depth" kernel threads
```

to increase the number of kernel threads to access multiple processors.

The program has two modes depending on the second command-argument (i.e., file or depth):

i. For the first mode: input number of values, input values, sort using 1 processor, output sorted values. Input and output is specified as follows:

- If the unsorted input file is not specified, print an appropriate usage message and terminate. The input file contains lists of unsorted values. Each list starts with the number of values in that list. For example, the input file:

  ```
  8 25 6 8 -5 99 100 101 7
  3 1 -3 5
  0
  10 9 8 7 6 5 4 3 2 1 0
  61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33
  32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  ```

  contains 4 lists with 8, 3, 0 and 10 values in each list. (The line breaks are for readability only; values can be separated by any white-space character and appear across any number of lines.) Since the number of data values can be (very) large, dynamically allocate the array to hold the values, otherwise the array can exceed the stack size of uMain::main.

  Assume the first number in the input file is always present and correctly specifies the number of following values; assume all following values are correctly formed so no error checking is required on the input data.
- If no output file name is specified, use standard output. Print the original input list followed by the sorted list, as in:

```
25 6 8 -5 99 100 101 7
-5 6 7 8 25 99 100 101

1 -3 5
-3 1 5

blank line from list of length 0 (not actually printed)
blank line from list of length 0 (not actually printed)

9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9

60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
  35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11
  10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
  25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
  50 51 52 53 54 55 56 57 58 59 60
```

for the previous input file. End each set of output with a blank line, and start a newline with 2 spaces after printing 25 values from a set of value.

ii. For the second mode: input array size $S$, initialize array to values $S..1$ (descending order), sort using depth processors, and print no values (used for timing experiments). The input file contains only one value, which is an array size. Parameter depth is a positive number ($> 0$). The default value if unspecified is 1. This mode is used to time the performance of the quicksort over a fixed set of values in descending order using different numbers of processors.

Print an appropriate error message and terminate the program if unable to open the given files. Check command argument depth for correct form (integer) and range; print an appropriate usage message and terminate the program if a value is invalid.

(b) i. Compare the speedup of the quicksort algorithm with respect to performance by doing the following:

- Time the execution using the time command:

```
% time quicksort size -1
16.100u 0.470s 0:04.24 390.8%
```

(Output from time differs depending on your shell, but all provide user, system and real time.) Compare the *user* (16.1u) and *real* (0:04.24) time among runs, which is the CPU time consumed solely by the execution of user code (versus system) and the total time from the start to the end of the program.

- Adjust the array size to get execution times in the range 5 to 20 seconds. (Timing results below 1 second are inaccurate.) Use the same array size for all experiments.
- After establishing an array size, run 7 experiments varying the value of depth from 1 2 4 8 16 32 64. Include all 7 timing results to validate your experiments.

ii. State the observed performance difference with respect to scaling when using different numbers of processors to achieve parallelism.

iii. Very briefly (2-4 sentences) speculate on the program behaviour.

## 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. PRNG.h, 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. q2quicksort.h, q2*.{h,cc,C,cpp} – code for question question 2a, p. 4. **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. q2quicksort.txt – contains the information required by question 2b.

5. Use the following Makefile to compile the programs for question 1, p. 1 and 2a, p. 4:

```
TYPE:=int
OPT:=

CXX = u++                                       # compiler
CXXFLAGS = -g -multi -Wall -Wno-unused-label -MMD ${OPT} -DTYPE="${TYPE}" # 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 = quicksort                               # 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 $@

-include ImplType

ifeq (${IMPLTYPE},${TYPE})                      # same implementation type as last time ?
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${TYPE},)                                 # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                            # implementation type has changed
.PHONY : ${EXEC2}
${EXEC2} :
    rm -f ImplType
    touch q2quicksort.h
    ${MAKE} TYPE="${TYPE}"
endif
endif

ImplType :
    echo "IMPLTYPE=${TYPE}" > ImplType

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

```
${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 quicksort TYPE=int
$ quicksort . . .
$ make quicksort TYPE=double
$ quicksort . . .
$ make quicksort TYPE=char
$ quicksort . . .
$ make quicksort OPT="-02" TYPE=int
$ quicksort . . .
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make tokenring or make quicksort 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!**