CS 343 Fall 2021 – Assignment 3
Instructor: Peter Buhr
Due Date: Wednesday, October 27, 2021 at 22:00
Late Date: Friday, October 29, 2021 at 22:00

October 11, 2021

This assignment examines synchronization and mutual exclusion, and introduces locks in μC++. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution. (You may freely use the code from these example programs.) **(Tasks may *not* have public members except for constructors and/or destructors.)**

1. Given the C++ program in Figure 1, compare stack versus heap allocation in a concurrent program.

   (a) Compare the versions of the program and different numbers of tasks with respect to performance by doing the following:

   - Run the program after compiling with preprocessor variables DARRAY, VECTOR1, VECTOR2 and STACK. Use compiler flags -O2 -multi -nodebug.
   - Time the executions using the time command:

     ```
     $ /usr/bin/time -f "%Uu %Ss %Er %Mkb" ./a.out 2 20000000
     3.21u 0.02s 0:03.32r 4228kb
     ```

     (Output from time differs depending on the shell, so use the system time command.) Compare the *user* (3.21u) and *real* (0:3.32) 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.
   - Use the second command-line argument (as necessary) to adjust the real time into the range 1 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
   - Run the 4 experiments with the number of tasks set to 1, 2, and 4.
   - Include all 12 timing results to validate the experiments and the number of calls to malloc.

   (b) State the performance and allocation difference (larger/smaller/by how much) with respect to scaling the number of tasks for each version.

   (c) Very briefly (2-4 sentences) speculate on the performance scaling among the versions.

2. (a) Merge sort is one of several sorting algorithms that takes optimal time (to within a constant factor) to sort N items. It also lends itself easily to concurrent execution by partitioning the data into two, and each half can be sorted independently and concurrently by another thread (divide-and-conquer/embarrassingly concurrent).

   Write a concurrent merge-sort function with the following interface:

   ```
   template<typename T> void mergesort( T data[], unsigned int size, unsigned int depth );
   ```

   that sorts an array of non-unique values into ascending order. Note, except for include files, all sort code must appear within mergesort.

   Implement the concurrent mergesort using:

   i. COBEGIN/COEND statements

   ii. **_Task** type: Do not create tasks by calls to **new**, i.e., no dynamic allocation is necessary for mergesort tasks.

   iii. **_Actor** type: All information about the array must be passed to the actor in an initial message not via the actor's constructor.

```
#include <iostream>
#include <vector>
#include <memory>                                       // unique_ptr
using namespace std;

int tasks = 1, times = 20000000;                        // default values

_Task Worker {
    enum { size = 100 };
    void main() {
        for ( int t = 0; t < times; t += 1 ) {
#if defined( DARRAY )
            unique_ptr<volatile int []> arr( new volatile int[size] );
            for ( int i = 0; i < size; i += 1 ) arr[i] = i;
#elif defined( VECTOR1 )
            vector<int> arr( size );
            for ( int i = 0; i < size; i += 1 ) arr.at(i) = i;
#elif defined( VECTOR2 )
            vector<int> arr;
            for ( int i = 0; i < size; i += 1 ) arr.push_back(i);
#elif defined( STACK )
            volatile int arr[size] __attribute__ (( unused )); // prevent unused warning
            for ( int i = 0; i < size; i += 1 ) arr[i] = i;
#else
    #error unknown data structure
#endif
        } // for
    } // Worker::main
}; // Worker

int main( int argc, char * argv[] ) {
    bool nosummary = getenv( "NOSUMMARY" );             // print summary ?
    try {                                               // process command-line arguments
        switch ( argc ) {
          case 3:
            times = stoi( argv[2] ); if ( times <= 0 ) throw 1;
          case 2:
            tasks = stoi( argv[1] ); if ( tasks <= 0 ) throw 1;
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ tasks (> 0) [ times (> 0) ] ]" << endl;
        exit( 1 );
    } // try
    uProcessor p[tasks - 1];                            // add CPUs (start with one)
    {
        Worker workers[tasks];                          // add threads
    }
    if ( ! nosummary ) { malloc_stats(); }
} // main
```

Figure 1: Stack versus Dynamic Allocation

For actor mergesort, there is work on the front and back side of the recursion, i.e., partitioning on the front and merging on the back. Hence, the parent actor needs to know when its two children actors are finished before merging. This synchronization is straightforward for COBEGIN and tasks because both provide strong synchronization points.

However, actors do not provide any direct synchronization points, so synchronization must be done with more messages. This means the left/right child *must* send a message back to its parent to let the parent know when the partitions are sorted.

A child actor determines its parent from its start message because each received message contains the address of its sender.

```
uActor * parent;
...
... receive(...) {
        ...
        parent = msg.sender(); // who sent message, maybe nullptr
```

So each child can discover and "remember" its parent (along with any other information to do the merge from its children). After partitioning:

- If the actor is a leaf actor in the depth tree, it sends a synchronization message to its parent indicating its partition is sorted. Any message works, e.g., the builtin uActor::stopMsg, and the actor is done.
- If the actor is a vertex actor in the depth tree, it does not terminate, but waits for 2 messages from each child. When the second message arrives, the parent merges the two sorted partitions from the children, sends a synchronization message to its parent indicating the vertex node is complete, and the actor is done.

There is one more detail. The mergesort function creates the root actor for the depth tree and starts it with a message. However, the mergesort function is *not* an actor so the sender value in the first start message is the nullptr. Hence, a special check is necessary for this case so no synchronization message is sent back to the merge function.

A naïve concurrent mergesort partitions the data values as normal, but instead of recursively invoking mergesort on each partition, a new implicit or explicit concurrent mergesort object is created to handle each partition. (For this discussion, assume no other sorting algorithm is used for small partitions.) However, this approach can create a large number of concurrent objects, approximately $2 \times N$, where N is the number of data values. This large number of concurrent objects can slow down the sort.

The number of concurrent objects can be reduced to approximately N by limiting the tree depth of the divide-and-conquer where concurrent sort objects are created (see details below). Basically, the depth argument is decremented on each recursive call and concurrent sort objects are only created while this argument is greater than zero; after which sequential recursive-calls are use to sort each partition. For explicit threading objects, a further reduction is possible by only creating one new concurrent sort task for the left partition and recursively sorting the right partition using the current mergesort task.

To simplify the mergesort algorithm, use two dynamically sized arrays: *one* to hold the initial unsorted data and *one* for copying values during a merge. (Merging directly in the unsorted array is possible but very complex.) Both of these arrays can be large, so creating them on the stack can overflow the stack when there are stacks with limited size. Hence, the program main dynamically allocates the storage for the unsorted data, and the mergesort function dynamically allocates *one* copy array, passing both by reference to internal recursive mergesorts.

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

```
mergesort [ unsorted-file | 'd' [ sorted-file | 'd' [ depth (>= 0) ] ] ]
mergesort -t size (>= 0) [ depth (>= 0) ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no unsorted file is specified, use standard input. If no sorted file is specified, use standard output. If no depth is specified, use 0. **The input-value type is provided by the preprocessor variable TYPE (see the Makefile).**

The program has two modes depending on the command option -t (i.e., sort or time):

sort mode: read the number of input values, read the input values, sort using 1 processor (which is the default), and output the sorted values. Input and output is specified as follows:

- The unsorted input 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 5 lists with 8, 3, 0, 10, and 61 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 the program 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.

- The sorted output is the original unsorted 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 (this line not actually printed)
blank line from list of length 0 (this line 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
```

End each set of output with a blank line, and start a newline with 2 spaces after printing 22 values from a set of values.

time mode: dimension an integer array to size, initialize the array to values size..1 (descending order), randomize the values using:

```
unsigned int times = sqrt( size );
for ( unsigned int counter = 0; counter < times; counter += 1 ) {
    swap( values[0], values[rand() % size] );
} // for
```

sort using $2^{depth} - 1$ processors, and print no values (used for timing experiments).
Parameter depth is a non-negative number ($>= 0$). The default value if unspecified is 0.
Create the additional processors by placing the following declaration in the same block as the merge-sort call:

```
uProcessor p[ (1 << depth) - 1 ] __attribute__(( unused )); // 2^depth-1 kernel threads
```

Keep the number of processors small as the undergraduate machines have a limited number of cores and other students are using the machines.

Print an appropriate error message and terminate the program if unable to open the given files. Check command arguments size and 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 mergesort algorithm with respect to performance by doing the following:

- Bracket the call to the sort in the program main with the following to measure the real time for the sort *within* the program.

```
uTime start = uClock::currTime();
quicksort( ... );
cout << "Sort time " << uClock::currTime() - start << " sec." << endl;
```

- Compile with makefile option OPT="-O2 -multi -nodebug".
- Time the execution using the time command:

```
$ /usr/bin/time -f "%Uu %Ss %E %Mkb" quicksort -t 300000000 0
14.13u 0.59s 0:14.68 1958468kb
```

(Output from time differs depending on the shell, so use the system time command.) Compare the *user* (14.13u) and *real* (0:14.68) 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 the real time in the range 10 to 20 seconds for depth 0. Use the same array size for all experiments.
- For each of CBEGIN, ACTOR, and TASK, run 6 experiments varying the value of depth from 0 1 2 3 4 5. Include all 18 timing results to validate your experiments.

ii. State the performance difference (larger/smaller/by how much) with respect to scaling when using different numbers of processors to achieve parallelism.

iii. Very briefly speculate on the performance difference, i.e., why is it the same or different.

3. (a) Implement a generalized FIFO bounded-buffer for a producer/consumer problem with the following interface (you may add only a public destructor and private members):

```
template<typename T> class BoundedBuffer {
  public:
    BoundedBuffer( const unsigned int size = 10 );
    unsigned long int blocks();
    void insert( T elem );
    T remove();
};
```

which creates a bounded buffer of size size, and supports multiple producers and consumers. Member blocks returns the current number of calls to wait in both insert and remove. You may *only* use uCondLock and uOwnerLock to implement the necessary synchronization and mutual exclusion needed by the bounded buffer.

Implement the BoundedBuffer in the following ways:

i. Use busy waiting, when waiting on a full or empty buffer. In this approach, tasks that have been signalled to access empty or full entries may find them taken by new tasks that barged into the buffer. This implementation uses one owner and two condition locks, where the waiting producer and consumer tasks block on the separate condition locks. (If necessary, you may add more locks.) The reason there is barging in this solution is that uCondLock::wait re-acquires its argument owner-lock before returning. Now once the owner-lock is released by a task exiting insert or remove, there is a race to acquire the lock by a new task calling insert/remove and by a signalled task. If the calling task wins the race, it barges ahead of any signalled task. So the state of the buffer at the time of the signal is not the same as the time the signalled task re-acquires the argument owner-lock, because the barging task changes the buffer. Hence, the signalled task may have to wait again (looping), and there is no guarantee of eventual progress (long-term starvation).

ii. Use *no* busy waiting when waiting for buffer entries to become empty or full. In this approach, use *barging avoidance* so a barging task cannot take empty or full buffer entries if another task has been unblocked to access these entries. This implementation uses one owner and two condition locks, where the waiting producer and consumer tasks block on the separate condition locks, but there is (*no looping*). (If necessary, you may add more locks.) Hint, one way to prevent overtaking by bargers is to use a flag variable to indicate when signalling is occurring; entering tasks test the flag to know if they are barging and wait on an appropriate condition-lock. When signalling is finished, an appropriate task is unblocked. (Hint: uCondLock::signal returns true if a task is unblocked and false otherwise.)

Before inserting or removing an item to/from the buffer, perform an assert that checks if the buffer is not full or not empty, respectively. Both buffer implementations are defined in a single .h file separated in the following way:

```
#ifdef BUSY                          // busy waiting implementation
// implementation
#endif // BUSY

#ifdef NOBUSY                        // no busy waiting implementation
// implementation
#endif // NOBUSY
```

The kind of buffer is specified externally by a preprocessor variable of BUSY or NOBUSY.

Insert the following macros into the NOBUSY solution to verify correctness.

```
#ifdef NOBUSY                           // no busy waiting implementation
#include "BargingCheck.h"
template<typename T> class BoundedBuffer {
    ...                                 // regular declarations
    BCHECK_DECL;
  public:
    void insert( T elem ) {
        // acquire mutual exclusion
        PROD_ENTER;
        ...
        // buffer insert
        INSERT_DONE;
        ...
        CONS_SIGNAL( cond-lock );
        cond-lock.signal();             // signal consumer task
        ...
        PROD_SIGNAL( cond-lock );       // if necessary
        cond-lock.signal();             // signal producer task
        ...
    }
    T remove() {
        // acquire mutual exclusion
        CONS_ENTER;
        ...
        // buffer remove
        REMOVE_DONE;
        ...
        PROD_SIGNAL( cond-lock );
        cond-lock.signal();             // signal producer task
        ...
        CONS_SIGNAL( cond-lock );       // if necessary
        cond-lock.signal();             // signal consumer task
        ...
    }
};
#endif // NOBUSY
```

Figure 2: Barging Check Macros

BCHECK_DECL is placed once in the buffer class as a private member, and contains variables and members needed to implement the barging check.

PROD_ENTER is placed immediately *after* mutual exclusion is acquired in insert.

INSERT_DONE is placed immediately *after* a value is inserted into the buffer in insert.

CONS_SIGNAL(cond-lock) is placed immediately *before* signalling a condition lock that is guaranteed to unblock a consumer task. If a condition lock has both producers and consumers blocked on it, it is NOT preceded by this macro.

CONS_ENTER is placed immediately *after* mutual exclusion is acquired in remove.

REMOVE_DONE is placed immediately *after* a value is removed from the buffer in remove.

PROD_SIGNAL(cond-lock) is placed before signalling a condition lock that is guaranteed to unblock a producer task. If a condition lock has both producers and consumers blocked on it, it is NOT preceded by this macro.

Figure 2 shows the macro placement in a buffer implementation, and defining preprocessor variable BARGINGCHECK triggers barging testing (see Makefile). If barging is detected, a message is printed and the program continues, possibly printing more barging messages. To inspect the program with gdb when barging is detected, set BARGINGCHECK=0 to abort the program.

Test the bounded buffer with a number of producers and consumers. The producer interface is:

```
_Task Producer {
    void main();
  public:
    Producer( BoundedBuffer<int> & buffer, const int Produce, const int Delay );
};
```

The producer generates Produce integers, from 1 to Produce inclusive, and inserts them into buffer. Before producing an item, a producer randomly yields between 0 and Delay times. Yielding is accomplished by calling yield(times) to give up a task's CPU time-slice a number of times. The consumer interface is:

```
_Task Consumer {
    void main();
  public:
    Consumer( BoundedBuffer<int> & buffer, const int Delay, const int Sentinel, int &sum );
};
```

The consumer removes items from buffer, and terminates when it removes a Sentinel value from the buffer. A consumer sums all the values it removes from buffer (excluding the Sentinel value) and returns this value through the reference variable sum. Before removing an item, a consumer randomly yields between 0 and Delay times.

The program main creates the bounded buffer, the producer and consumer tasks, and an array of subtotal counters, one for each consumer. After all the producer tasks have terminated, the program main inserts an appropriate number of sentinel values (the default sentinel value is -1) into the buffer to terminate the consumers. The partial sums from each consumer are totalled to produce the sum of all values generated by the producers. Print this total in the following way:

```
total: ddddd...
```

The sum must be the same regardless of the order or speed of execution of the producer and consumer tasks.

Create a global MPRNG random-number generator in the program main and initialize it with getpid() (monitors are discussed shortly). Share this global variable with the producers and consumers using an external (**extern**) declaration.

The shell interface for the boundedBuffer program is:

```
buffer [ Cons | 'd' [ Prods | 'd' [ Produce | 'd' [ BufferSize | 'd' [ Delay | 'd'
    [ Processors | 'd' ] ] ] ] ] ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.)

Cons is the positive number of consumers to create. If d or no value for Cons is specified, assume 5.

Prods: is the positive number of producers to create. If d or no value for Prods is specified, assume 3.

Produce: is the positive number of items generated by each producer. If d or no value for Produce is specified, assume 10.

BufferSize: is the positive number of elements in the bounder buffer. If d or no value for BufferSize is specified, assume 10.

Delays: is the positive number of times a producer/consumer yields *before* inserting/removing an item into/from the buffer. If d or no value for Delays is specified, assume Cons + Prods.

Processors: is the positive number of kernel threads to create for parallelism. If d or no value for Processors is specified, assume 1. Use this number in the following declaration placed in the program main immediately after checking command-line arguments but before creating any tasks:

```
uProcessor p[Processors - 1] __attribute__(( unused )); // create more kernel thread
```

The program starts with one kernel thread so only N - 1 additional kernel threads are added.

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. The type of the buffer element is **int** throughout the program, and the integer sentinel value is specified externally by preprocessor variable SENTINEL.

(b) i. Compare the busy and non-busy waiting versions of the program with respect to *uniprocessor* performance by using 1 kernel thread:

- Use the μC++ -nodebug flag in all the experiments.
- Time the executions using the time command:

  ```
  $ /usr/bin/time -f "%Uu %Ss %Er %Mkb" ./a.out
  3.21u 0.02s 0:03.32r 33640kb
  ```

  (Output from time differs depending on the shell, so use the system time command.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).
- Use the program command-line arguments 55 50 20000 20 10 1 and adjust the Produce amount (if necessary) to get program execution into the range 1 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
- Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2).
- Include 4 timing results to validate the experiments.

ii. State the performance difference (larger/smaller/by how much) between uniprocessor busy and nobusy waiting execution, without and with optimization.

iii. Compare the busy and non-busy waiting versions of the program with respect to *multiprocessor* performance by repeating the above experiment with 4 kernel threads.
- Include 4 timing results to validate the experiments.

iv. State the performance difference (larger/smaller/by how much) between multiprocessor busy and nobusy waiting execution, without and with optimization.

v. Speculate as to the reason for the performance difference between busy and non-busy execution. Use the total number of times a producer/consumer blocks in the bounded buffer to help understand differences.

vi. Speculate as to the reason for the performance difference between uniprocessor and multiprocessor execution.

Submission Guidelines

Follow these guidelines carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text or test-document file, e.g., *.{txt,doc} file, must be ASCII text and not exceed 500 lines in length, using the command fold -w120 *.doc | wc -l.** 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*.txt – contains the information required by question 1, p. 1.
2. q2mergesort.h, q2*.{h,cc,C,cpp} – code for question question question 2a, p. 1. **Program documentation must be present in your submitted code. No user, system or test 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.**
3. q2*.txt – contains the information required by questions 2b, p. 4.
4. BargingCheck.h – barging checker (provided)
5. MPRNG.h – random number generator (provided)
6. q3buffer.h, q3*.{h,cc,C,cpp} – code for question question 3a, p. 5. **Program documentation must be present in your submitted code. No user, system or test 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.**
7. q3*.txt – contains the information required by questions 3b.

8. Modify the following Makefile to compile the programs for question 2a, p. 1 and 3a, p. 5 by inserting the object-file names matching your source-file names.

```
OPT:= # -O2 -nodebug
SIMPL:=CBEGIN
TYPE:=int
SENTINEL:=-1
BIMPL:=NOBUSY
BCHECK:=NOBARGINGCHECK

CXX = u++                                       # compiler
CXXFLAGS = -g -multi ${OPT} -Wall -Wextra -MMD -D"${SIMPL}" -DTYPE="${TYPE}" \
        -D"${BIMPL}" -DSENTINEL="${SENTINEL}" -D"${BCHECK}" # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = # object files forming 1st executable with prefix "q2"
EXEC1 = mergesort                               # 1st executable name

OBJECTS2 = # object files forming 2nd executable with prefix "q3"
EXEC2 = buffer                                  # 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

-include MergeImpl
```

```
ifeq (${shell if [ "${IMPLTYPE}" = "${TYPE}" -a "${IMPLSIMPL}" = "${SIMPL}" ] ; \
            then echo true ; fi },true)
${EXEC1} : ${OBJECTS1}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
.PHONY : ${EXEC1}
${EXEC1} :
    rm -f MergeImpl
    touch q2mergesort.h
    ${MAKE} ${EXEC1} SIMPL="${SIMPL}" TYPE="${TYPE}"
endif

MergeImpl :
    echo "IMPLSIMPL=${SIMPL}\nIMPLTYPE=${TYPE}" > MergeImpl
    sleep 1

-include BufImpl

ifeq (${shell if [ "${BUFIMPL}" = "${BIMPL}" -a "${SENTIMPL}" = "${SENTINEL}" -a \
          "${BCHECKIMPL}" = "${BCHECK}" ] ; then echo true ; fi },true)
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                            # implementation type has changed => rebuilt
.PHONY : ${EXEC2}
${EXEC2} :
    rm -f BufImpl
    touch q3buffer.h
    sleep 1
    ${MAKE} ${EXEC2} BIMPL="${BIMPL}" SENTINEL="${SENTINEL}" BCHECK="${BCHECK}"
endif

BufImpl :
    echo "BUFIMPL=${BIMPL}\nSENTIMPL=${SENTINEL}\nBCHECKIMPL=${BCHECK}" > BufImpl
    sleep 1
#################################################################

${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} MergeImpl BufImpl
```

This makefile is used as follows:

```
$ make mergesort SIMPL=CBEGIN
$ mergesort ...
$ make mergesort SIMPL=ACTOR TYPE=int
$ mergesort ...
$ make mergesort SIMPL=TASK TYPE=double
$ mergesort ...
$ make buffer BIMPL=BUSY BCHECK=BARGINGCHECK # use SENTINEL=-1
$ buffer ...
$ make buffer BIMPL=NOBUSY SENTINEL=0 OPT="-O2" # switch to SENTINEL=0
$ buffer ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make mergesort or make buffer 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.

Follow these guidelines. Your grade depends on it!