CS 343 Fall 2024 – Assignment 3
Instructor: Peter Buhr
Due Date: Wednesday, October 23, 2024 at 22:00
Late Date: Friday, October 25, 2024 at 22:00

September 22, 2024

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 v t to performance by doing the following:

   - Run the p AY, VECTOR1, and VECTOR2. Use comp
   - Time the

     $ /usr
     3.21u

     (Output f mand.) Compare the *user* (3.21u) a ed solely by the execution of user c he program.
   - Use the s me into the range 1 to 100 seconds. command-line values for all experi s necessary and scale the differenc
   - Run each with the number of tasks set to 1, 2
   - Include a calls to malloc.

   (b) State the perf with respect to scaling the number of tas

   (c) Very briefly (2 rsions.

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.

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

intmax_t tasks = 1, times = 200′000′000, asize = 10;    // default values

_Task Worker {
    void main() {
        for ( int t = 0; t < times; t += 1 ) {
        #if defined( STACK )
            volatile int arr[asize] __attribute__ (( unused )); // prevent unused warning
            for ( int i = 0; i < asize; i += 1 ) arr[i] = i;
        #elif defined( DARRAY )
            unique_ptr<volatile int []> arr( new volatile int[asize] );
            for ( int i = 0; i < asize; i += 1 ) arr[i] = i;
        #elif defined( VE
            vector<int> 
            for ( int i = 
            asm volatile
        #elif defined( VE
            vector<int> 
            for ( int i = 
            asm volatile
        #else
            #error unkno
        #endif
        } // for
    } // Worker::main
}; // Worker

int main( int argc, char *
    char * nosummary = 

    struct cmd_error {};
    try {                                                                  uments
        switch ( argc ) {
          case 4:
            asize = conv
          case 3:
            times = conv
          case 2:
            tasks = convert( argv[1] ); if ( tasks <= 0 ) throw cmd_error();
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ tasks (> 0) [ times (> 0) "
                "[ array size (> 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

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

- If the ion message to its parent indic e works, e.g., the builtin uActo
- If the waits for 2 messages from each wo sorted partitions from the cl e vertex node is complete and t

There is r the depth tree and starts it with a e sender value in the first start mes ase so no synchronization message

A naïve conc d of recursively invoking mergesort on e ct is created to handle each partition. (Fo nall partitions.) However, this approach $2 \times N$, where N is the number of dat he sort.

The number c ting the tree depth of the divide-and-co ow). Basically, the depth argument is d e 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 externally by the preprocessor variable STYPE (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.

  Assu nt and correctly specifies the n rectly formed so no error chec

- The s ted list, as in:

```
25 6
−5 6

1 −3
−3 1

blank
blank

9 8 7
0 1 2
60 59                                   39
  38                                  3 17
  16
0 1 2
  22                                    2 43
  44
```

  End e es after printing 22 values from

time mode: d ize..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[prng( 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();
mergesort( ... );
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" mergesort -t 200000000 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 CFOR, ACTOR, and TASK, run 5 experiments varying the value of depth from 0 1 2 3 4. In

ii. State the ect to scaling when using different

iii. Very brie sort time and the real time from the

3. (a) Implement a g with the following interface (you may

```
template<
public:
    _Exce
    Bound
    unsig
    void
    void
    T rem
};
```

which creates and consumers. Member blocks returns Member poison marks the buffer as *poiso* is poisoned and all values have been con n exception; similarly for any arriving c inate. 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 bu NOBUSY.

Insert the foll hese macros must appear in your submi

BCHECK_DE ntains variables and members need

PROD_ENTE rt.

INSERT_DON insert.

CONS_SIGNA on lock that is guaranteed to unbloc sumers blocked on it, it is NOT pre

CONS_ENTE ove.

REMOVE_DO r in remove.

PROD_SIGNA s guaranteed to unblock a producer d on it, it is NOT preceded by this m

Figure 2 sho ing preprocessor variable BARGING−CH ted, a message is printed and the progra hen the buffer is poisoned and waiting c ck) before any signalling caused by poi g during normal program operation and not during termination due to poisoning. 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, int &sum );
};
```