MTE241, Spring 2020, Final Assessment

August 1 – August 15

Summary

The final assessment is a uVision5 project that solves the Byzantine Generals Problem. You will implement the part of the OM algorithm that distributes the values to the generals. You do not need to calculate majority values (i.e. each general does not have to reach a decision). Your program will output the values received in the OM(0) stage by only one of the generals. To do the assessment, download the starter project, final.zip. It includes three source files:

- final.c – the test code (details below),
- general.h – macro definitions and function declarations,
- general.c – the file in which you implement your solution.

Submit only general.c to the Learn dropbox. The given test cases will be used to do the grading with only these variations: changing which general is the commander, changing what command is sent, or changing which general generates output.

Test Thread

```
void testCases(void *arguments) {
  for(int i=0; i<N_TEST; i++) {
    printf("\ntest case %d\n", i);
    if(setup(tests[i].n, tests[i].loyal, tests[i].reporter)) {
      startGenerals(tests[i].n);
      broadcast(tests[i].command, tests[i].commander);
      cleanup();
      stopGenerals();
    } else {
      printf(" setup failed\n");
    }
  }
  printf("\ndone\n");
}
```

The test thread iterates through 6 test cases, increasing in complexity. For each test case, it calls your setup() function first to record the test parameters and create any OS resources or other resources that will be needed by your solution. Setup() should calculate m, the number of traitors, and use c_assert() to test that n > 3*m and return false if fails The test thread then starts n general threads with IDs 0 … n-1. The general threads run your general() function. Then it invokes your broadcast() function which sends a command from the commanding general to the other generals. Your broadcast() function should return when the algorithm completes and the expected output has been printed to UART #1. When your broadcast() function returns, the test thread invokes your cleanup() function to delete or free any resources and then the test threads deletes the n general threads.

Test Cases

Case	# generals	Loyal?*	Reporter	Command	Commander
0	3	T, T, F	1	R	0
1	3	T, T, T	2	R	0
2	3	T, T, T	2	A	1
3	4	T, F, T, T	2	R	0
4	4	T, F, T, T	2	A	1
5	7	T, F, T, T, F, T, T	6	R	0

*T = true, F = false

Test case 0 is n=3, m=1 so asserting n > 3*m should fail.
Test case 1 is n=3, m=0. General 0 sends command R. General 2 reports what it receives in OM(0).
Test case 2 is n=3, m=0. General 1 sends command A. General 2 reports.
Test case 3 is n=4, m=1. General 0 sends command R. General 2 reports. General 1 is a traitor.
Test case 4 is n=4, m=1. General 1 sends command A. General 2 reports. General 1 is a traitor.
Test case 5 is n=7, m=2. General 0 sends command R. General 6 reports. Generals 1 and 4 are traitors.

Traitors

A treacherous commander will broadcast ‘R’ to even numbered generals and ‘A’ to odd numbered generals.
In all subsequent steps, a treacherous lieutenant will send ‘R’ to all other generals if it is even numbered and ‘A’ otherwise.
Loyal generals will send the same command that they receive.

Expected Output

```
test case 0
general.c,22: assertion 'n > 3*m' failed
 setup failed

test case 1
 0:R

test case 2
 1:A

test case 3
 1:0:A 3:0:R

test case 4
 0:1:R 3:1:A

test case 5
 2:1:0:A 3:1:0:A 4:1:0:R 5:1:0:A 1:2:0:A 2:3:0:R 3:2:0:R 4:2:0:R
5:2:0:R 5:3:0:R 1:3:0:A 4:3:0:R 3:4:0:R 5:4:0:R 1:4:0:A 2:4:0:R
4:5:0:R 1:5:0:A 2:5:0:R 3:5:0:R

done
```

The order that output values are listed for each test case is not important. The values themselves are important. For example, if the test case 5 output above was sorted and put in table form, it would correspond to:

```
-:-:-:- 1:2:0:A 1:3:0:A 1:4:0:A 1:5:0:A -:-:-:-
2:1:0:A -:-:-:- 2:3:0:R 2:4:0:R 2:5:0:R -:-:-:-
3:1:0:A 3:2:0:R -:-:-:- 3:4:0:R 3:5:0:R -:-:-:-
4:1:0:R 4:2:0:R 4:3:0:R -:-:-:- 4:5:0:R -:-:-:-
5:1:0:A 5:2:0:R 5:3:0:R 5:4:0:R -:-:-:- -:-:-:-
-:-:-:- -:-:-:- -:-:-:- -:-:-:- -:-:-:- -:-:-:-
```

Grading out of 20

Passed cases	1	2	3	4	5	6
Grade	10	12	14	16	18	20

If your solution passes 0 cases it will be hand marked for a maximum of 8 out of 20.

Tips

- You can solve it ***<u>however you want</u>*** as long as the general threads do the solving. e.g. the OM algorithm is recursive but your solution doesn't have to be. Section 13.5.2 of the course text gives a different way to think about the solution.
- Work on one test case at a time, in order. They get increasingly difficult.
- Check the OS function return values, especially when creating new resources (semaphores, mutexes, message queues, event flags, etc), and when using them (get, put, acquire, release, etc). Also check the return value from malloc().
- I've set up the starter project with 18 kiB for Global Dynamic Storage (used to create semaphores, mutexes, etc), 8 kiB heap for malloc(), 512 B for the main() stack. Thread stacks come from the Global Dynamic Storage and are 1 kiB each. This should be sufficient for most solutions. The debugger is configured to show RTX data such as the amount of Global Dynamic Storage consumed and the percentage full of each thread's stack. You can also examine local variables in the bottom-right window of the debugger (same window as the UART#1 output).