CS 6110, Spring 2022, Assignment 3
Given 2/3/22 – Due 2/10/22 by 11:59 pm via your Github

NAME: Tripti Agarwal UNID: u1319433

1. (8 marks) Murphi code

- Downloaded rumur and run the make file available on Github for the code npretenson.m
- Following outputs are obtained for N=3 and 5.
  - **N=3**
    BFS, 35 bits (rounded up to 5 bytes)
    172 states, 516 rules fired in 0s.
  - **N=5**
    BFS, 63 bits (rounded up to 8 bytes)
    770 states, 33850 rules fired in 0s.
- Using sym1 and sym2 flags
  - **N=3, sym1**
    BFS, sym1
    35 bits (rounded up to 5 bytes) 172 states, 516 rules fired in 0s
  - **N=3, sym2**
    BFS, sym2
    35 bits (rounded up to 5 bytes)
    172 states, 516 rules fired in 1s.
  - **N=5, sym1**
    BFS, sym1
    63 bits (rounded up to 8 bytes)
    6770 states, 33850 rules fired in 0s
  - **N=5, sym2**
    BFS, sym2
    63 bits (rounded up to 8 bytes)
    6770 states, 33850 rules fired in 0s.
- **N=9**
  BFS, sym1 runs for 2871372 states, 25842348 rules fired in 19s.
  BFS, sym2 runs for 2871372 states, 25842348 rules fired in 18s.

2. (2 points) Photoshop bug

- To improve the improvement of computation in photoshop, designers adopted parallelism.
- Success with parallelism involved basic image-processing routines. This made parallelism involved in the app in a way that did not complicated implementations for the bottleneck-routine algorithms.
- This lead to error-prone synchronization primitives. The critical section to enter and leave were very fast in Windows but were highly error prone.
- This basically explained how tricky can parallel programming can be. As the cores for the processors increases, the complexity will grow and we need more tools or better programming primitives for hiding the complexity for developers.
- As the number of cores increases, the image chunks being processed, called tiles, are sliced into greater number of smaller pieces, resulting in increased synchronization overhead.
- The effect the Amdalh's wall, was tried to be fixed by increasing the tile size, which made the sub pieces even larger. Engineers tried fixing it by using another algorithm but it led to memory-bandwidth limitations. This problem caused by the fact that Photoshop cannot interrupt pixel processing operations, until an entire tile is complete. Hence this will lead to latency issues.

3. (90 points)

The code had the error in the handle proctype. This can be seen by looking at the cycle in the error trace as shown the figure below:

```
<<<<<START OF CYCLE>>>>>
63:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
64:     proc  8 (handle:1) locking-buggy.pml:124 (state 5) [((po[me]!=me))]
65:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
2 66:   proc  8 (handle:1) locking-buggy.pml:124 (state 6) [request[po[me]]!requester]
67:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
68:     proc  8 (handle:1) locking-buggy.pml:134 (state 17) [mutex[me] = 0]
69:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
70:     proc  2 (handle:1) locking-buggy.pml:134 (state 17) [mutex[me] = 0]
71:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
72:     proc  2 (handle:1) locking-buggy.pml:118 (state -) [1]
72:     proc  2 (handle:1) locking-buggy.pml:118 (state 1) [(((mutex[me]==0)&&request[me]?[requester]))]
72:     proc  2 (handle:1) locking-buggy.pml:120 (state 2) [mutex[me] = 1]
73:     proc  2 (handle:1) locking-buggy.pml:121 (state 3) [request[me]?requester]
74:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
75:     proc  2 (handle:1) locking-buggy.pml:124 (state 5) [((po[me]!=me))]
76:     proc  - (never_0:1) locking-buggy.pml:188 (state 35) [(((!((acquire[pid_acq[0]]._p==release))&&!((acquire[pid_acq[1]]._p==release)))&
&!((acquire[pid_acq[2]]._p==release)))&&!((acquire[pid_acq[3]]._p==release)))]
```

We observe that the cycle is between line 118 and 124. This is because the code in lines 117 to 122 should run once for a particular value of me and then execute line 124 and should loop back to 117. The reason can be seen more clearly from the example, let say po of 2 and 3 is 1. The request channel of 1 consists of 2 and 3. Once the requester is 2 then the handle should make the mutex of po as 1 and 2 should acquire the resources. In the buggy implementation, since the cycle exists, once the requester is 2 it is possible that 3 can also be the requester, and hence we need to stop this issue.

The fix is provided in the code and shown in the snippet below:

```
proctype handle(int me)
{
    PID requester;
    byte count = 0;
    do
    :: atomic {
            mutex[me] == 0 && request[me]?[requester] && count == 0
                ->
            mutex[me] = 1;
            request[me] ? requester
            count = 1;
        };
        if
        :: po[me] != me && count != 0 -> request[po[me]] ! requester
        :: po[me] == me && locked[me] ->
                    waiters[me] ! requester;
                    qlen[me] = qlen[me] + 1;
                    assert (qlen[me] < Nprocs)
        :: po[me] == me && !locked[me] ->
                    q_len_ch[requester] ! 0;
                    assert (qlen[me] == 0);
                    po[me] = requester
        fi;
        mutex[me] = 0
    od
}
```

We see that putting an extra guard of count, which makes the code from line 117 to 122 to run only once and then putting a guard to see whether the code has run that atomic section, resolves the issue and removes the cycle. The output without any error is shown below:

```
State-vector 244 byte, depth reached 228, errors: 0
   755393 states, stored (801544 visited)
 11428127 states, matched
 12229671 transitions (= visited+matched)
   322516 atomic steps
hash conflicts:    166874 (resolved)
```