Assignment Part1 (Promela and Spin).

Name: Dushyanth

Roll No: 18CS01009

Q1. Model Dekker's algorithm presented above in Promela, the input language of the spin model-checker (http://www.spinroot.com/). Explain your model.

```
//Dekker's mutex algorithm, two parallel processes

bool flag[2] = {false};
byte turn = 0;

active [2] proctype process(){
    //printf("process: %d",_pid)
    byte i = _pid, j = 1 - _pid

    //infinite loop
    do
      ::    //non-critical section
            flag[i] = true;
            //trying section
            do
                :: flag[j] ->
                   if
                       :: turn==j ->
                          flag[i] = false;
                          if
                              :: turn != j -> flag[i] = true;
                          fi
                       :: else -> skip;
                   fi
                :: else -> break;
            od
            //critical section
            turn = j;
            flag[i] = false;
    od
}

```

dekker_q1.pml

Explanation:

Two active instances of a process are created. The special variable '_pid' uniquely identifies the process. Since there are only two processes, the possible values of '_pid' are 0 / 1. So, if '_pid' identifies the current process then '1-_pid' identifies other process. The outer infinite while is modelled in spin with 'do od' with only one alternative and no guard. The outer while loop of the trying section is also modelled using 'do od' with 2 alternatives, one with 'flag[j]' guard and the other with 'else' to break out of the loop when 'flag[j]' is false. The 'if' statement of the trying section is modelled with 'if fi' having 2 alternatives. The first one is executed when 'turn equals j' and 'else' part has 'skip' which is like 'nop' to avoid blocking of 'if fi' until other process makes the guard true. The inner while loop is implemented using 'if fi' since 'if fi' is blocking when all guards are false in spin. The inner 'if fi' blocks till 'turn equals j' and then assignment 'true' to 'flag[i]'.

Q2. Add assertion mechanism into the code which triggers when both processes are in the critical section at the same time. Explain. Check with Spin whether Dekker's algorithm guarantees mutual exclusion for two processes. Submit the verification output given by Spin and explain the result. Hint: Use additional variable(s) to model when a process is in the critical section.

```
//Dekker's mutex algorithm, two parallel processes

bool flag[2] = {false};
byte turn = 0, critical = 0;

active [2] proctype process(){
    //printf("process: %d",_pid)
    byte i = _pid, j = 1 - _pid

    //infinite loop
    do
    ::    //non-critical section
        flag[i] = true;
        //trying section
        do
            :: flag[j] ->
            if
                :: turn==j ->
                flag[i] = false;
                if
                    :: turn != j -> flag[i] = true;
                fi
                :: else -> skip;
            fi
            :: else -> break;
        od
        //critical section
        critical++;
        assert(critical<=1);
        critical--;
        turn = j;
        flag[i] = false;
    od
}

```

dekker_q2.pml

A variable 'critical' of data type byte is declared. At the critical section entry point 'critical++' is done and at the end of the critical section (exit point), 'critical- -' is done. The assert statement is return in a way that it get triggered only when both processes are in the critical section. The value of critical should always be less than 1.

Steps to execute the model:

1. spin –a dekker_q2.pml
2. gcc –o dekker_q2 pan.c
3. ./dekker_q2

   The above steps are followed to generate the verifier and this generated verifier in 'pan.c' is compiled and ran.

Output:

```
dush123@ubuntu:~/Desktop/se$ cd Q2
dush123@ubuntu:~/Desktop/se/Q2$ ls
dekker_q2.pml
dush123@ubuntu:~/Desktop/se/Q2$ spin -a dekker_q2.pml
dush123@ubuntu:~/Desktop/se/Q2$ gcc -o dekker_q2 pan.c
```

```
dush123@ubuntu:~/Desktop/se/Q2$ ls
dekker_q2  dekker_q2.pml  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t
```

```
dush123@ubuntu:~/Desktop/se/Q2$ ./dekker_q2

(Spin Version 6.4.9 -- 17 December 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 28 byte, depth reached 49, errors: 0
      166 states, stored
      156 states, matched
      322 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.009       equivalent memory usage for states (stored*(State-vector + overhead))
    0.290       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage


unreached in proctype process
        dekker_q2.pml:34, state 26, "-end-"
        (1 of 26 states)

pan: elapsed time 0 seconds
```

We can see that there are no errors reported by the spin verifier. Thus, the dekker's algorithm indeed guarantees the mutual exclusion for both processes.

Q3. Add LTL propertie(s) outside the model (processes) that expresses that always at most one process can be in the critical section. Check with Spin whether Dekker's algorithm satisfies the LTL property. Submit the verification output given by Spin and explain the result.

```
//Dekker's mutex algorithm, two parallel processes
bool flag[2] = {false};
byte turn = 0, critical = 0;

active [2] proctype process(){
    //printf("process: %d",_pid)
    byte i = _pid, j = 1 - _pid

    //infinite loop
    do
    ::    //non-critical section
        flag[i] = true;
        //trying section
        do
            :: flag[j] ->
            if
                :: turn==j ->
                    flag[i] = false;
                    if
                        :: turn != j -> flag[i] = true;
                    fi
                :: else -> skip;
            fi
            :: else -> break;
        od
        //critical section
        critical++;

        critical--;
        turn = j;
        flag[i] = false;
    od
}
ltl atMostOne{
    [](critical<=1)
}
```

dekker_q3.pml

The LTL property is specified as [](critical<=1) which says that always at-most one process is in critical section.

Steps for execution:

1. spin –a dekker_q3.pml
2. gcc –o dekker_q3 pan.c
3. ./dekker_q3 –N atMostOne

Output:

```
dush123@ubuntu:~/Desktop/se$ cd Q3
dush123@ubuntu:~/Desktop/se/Q3$ ls
dekker_q3.pml
dush123@ubuntu:~/Desktop/se/Q3$ spin -a dekker_q3.pml
ltl atMostOne: [] ((critical<=1))
dush123@ubuntu:~/Desktop/se/Q3$ ls
dekker_q3.pml  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t  _spin_nvr.tmp
dush123@ubuntu:~/Desktop/se/Q3$ gcc -o dekker_q3 pan.c
```

```
dush123@ubuntu:~/Desktop/se/Q3$ ls
dekker_q3  dekker_q3.pml  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t  _spin_nvr.tmp
dush123@ubuntu:~/Desktop/se/Q3$ ./dekker_q3 -N atMostOne
warning: only one claim defined, -N ignored
warning: never claim + accept labels requires -a flag to fully verify

(Spin Version 6.4.9 -- 17 December 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (atMostOne)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     - (not selected)
        invalid end states      - (disabled by never claim)

State-vector 40 byte, depth reached 89, errors: 0
      144 states, stored
      134 states, matched
      278 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.009       equivalent memory usage for states (stored*(State-vector + overhead))
    0.289       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage


unreached in proctype process
        dekker_q3.pml:34, state 25, "-end-"
        (1 of 25 states)
unreached in claim atMostOne
        _spin_nvr.tmp:8, state 10, "-end-"
        (1 of 10 states)

pan: elapsed time 0 seconds
```

The verifier is generated and ran. It reports no errors. Thus, the property ‘atMostOne’ is satisfied in all the behaviours of the system.

Q4. In the book "M. Raynal. Algorithms for mutual exclusion (1986)" a simpler variant of the inner loop (trying section) is suggested. Does the modified algorithm work? Explain the result.

```
//Dekker's mutex algorithm, two parallel processes
bool flag[2] = {false};
byte turn = 0, critical = 0;

active [2] proctype process(){
    //printf("process: %d",_pid)
    byte i = _pid, j = 1 - _pid

    //infinite loop
    do
      ::        //non-critical section
            flag[i] = true;
            //trying section
            if
                :: flag[j] ->
                    if
                        :: turn==j ->
                            flag[i] = false;
                            if
                                :: turn != j -> flag[i] = true;
                            fi
                        :: else -> skip;
                    fi
                :: else -> skip;
            fi
            //cirtical section
            critical++;

            critical--;
            turn = j;
            flag[i] = false;
    od
}
ltl atMostOne{
    [](critical<=1)
}
```

dekker_q4.pml

The variant is modelled using Promela.

Steps for execution:

1. spin –a dekker_q4.pml
2. gcc –o dekker_q4 pan.c
3. ./dekker_q4 –N atMostOne

   It is observed that the LTL property specified is violated by the modelled system
4. spin –t –p –l –g dekker_q4.pml > output.txt

   The above command makes the use of 'q4.pml.trail' and outputs the system behaviour in which the property gets violated. This output is redirected to 'output.txt' which is zipped and submitted.

Output:

```
dush123@ubuntu:~/Desktop/se$ cd Q4
dush123@ubuntu:~/Desktop/se/Q4$ ls
dekker_q4.pml
dush123@ubuntu:~/Desktop/se/Q4$ spin -a dekker_q4.pml
ltl atMostOne: [] ((critical<=1))
dush123@ubuntu:~/Desktop/se/Q4$ ls
dekker_q4.pml  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t  _spin_nvr.tmp
dush123@ubuntu:~/Desktop/se/Q4$ gcc -o dekker_q4 pan.c
```

Verifier is generated – 'pan.c' and compiled

```
dush123@ubuntu:~/Desktop/se/Q4$ ls
dekker_q4  dekker_q4.pml  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t  _spin_nvr.tmp
dush123@ubuntu:~/Desktop/se/Q4$ ./dekker_q4 -N atMostOne
warning: only one claim defined, -N ignored
warning: never claim + accept labels requires -a flag to fully verify
pan:1: assertion violated  !( !((critical<=1))) (at depth 186)
pan: wrote dekker_q4.pml.trail

(Spin Version 6.4.9 -- 17 December 2018)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (atMostOne)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     - (not selected)
        invalid end states      - (disabled by never claim)

State-vector 40 byte, depth reached 207, errors: 1
      146 states, stored
       82 states, matched
      228 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.009       equivalent memory usage for states (stored*(State-vector + overhead))
    0.289       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage



pan: elapsed time 0 seconds
dush123@ubuntu:~/Desktop/se/Q4$ ls
dekker_q4  dekker_q4.pml  dekker_q4.pml.trail  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t  _spin_nvr.tmp
```

We can observe that an error is being reported by the verifier. It says that there exists a system behaviour where the property gets violated.

```
dush123@ubuntu:~/Desktop/se/Q4$ spin -t -p -l -g dekker_q4.pml
ltl atMostOne: [] ((critical<=1))
starting claim 1
Never claim moves to line 4     [(1)]
  2:    proc  1 (process:1) dekker_q4.pml:13 (state 1)  [flag[i] = 1]
                flag[0] = 0
                flag[1] = 1
  4:    proc  1 (process:1) dekker_q4.pml:25 (state 13) [else]
  6:    proc  1 (process:1) dekker_q4.pml:25 (state 14) [(1)]
  8:    proc  1 (process:1) dekker_q4.pml:28 (state 17) [critical = (critical+1)]
                critical = 1
 10:    proc  1 (process:1) dekker_q4.pml:30 (state 18) [critical = (critical-1)]
                critical = 0
 12:    proc  1 (process:1) dekker_q4.pml:31 (state 19) [turn = j]
 14:    proc  0 (process:1) dekker_q4.pml:13 (state 1)  [flag[i] = 1]
                flag[0] = 1
                flag[1] = 1
 16:    proc  1 (process:1) dekker_q4.pml:32 (state 20) [flag[i] = 0]
                flag[0] = 1
                flag[1] = 0
 18:    proc  1 (process:1) dekker_q4.pml:13 (state 1)  [flag[i] = 1]
                flag[0] = 1
                flag[1] = 1
 20:    proc  1 (process:1) dekker_q4.pml:16 (state 2)  [(flag[j])]
 22:    proc  1 (process:1) dekker_q4.pml:18 (state 3)  [((turn==j))]
 24:    proc  1 (process:1) dekker_q4.pml:19 (state 4)  [flag[i] = 0]
                flag[0] = 1
                flag[1] = 0
 26:    proc  0 (process:1) dekker_q4.pml:25 (state 13) [else]
 28:    proc  0 (process:1) dekker_q4.pml:25 (state 14) [(1)]
 30:    proc  0 (process:1) dekker_q4.pml:28 (state 17) [critical = (critical+1)]
                critical = 1
 32:    proc  0 (process:1) dekker_q4.pml:30 (state 18) [critical = (critical-1)]
                critical = 0
```

...

...

...

```
172:    proc  0 (process:1) dekker_q4.pml:25 (state 13) [else]
174:    proc  0 (process:1) dekker_q4.pml:25 (state 14) [(1)]
176:    proc  1 (process:1) dekker_q4.pml:13 (state 1)  [flag[i] = 1]
                flag[0] = 1
                flag[1] = 1
178:    proc  1 (process:1) dekker_q4.pml:16 (state 2)  [(flag[j])]
180:    proc  1 (process:1) dekker_q4.pml:23 (state 9)  [else]
182:    proc  1 (process:1) dekker_q4.pml:23 (state 10) [(1)]
184:    proc  1 (process:1) dekker_q4.pml:28 (state 17) [critical = (critical+1)]
                critical = 1
186:    proc  0 (process:1) dekker_q4.pml:28 (state 17) [critical = (critical+1)]
                critical = 2
spin: _spin_nvr.tmp:3, Error: assertion violated
spin: text of failed assertion: assert(!(!((critical<=1))))
Never claim moves to line 3     [assert(!(!((critical<=1))))]
spin: trail ends after 187 steps
#processes: 2
                flag[0] = 1
                flag[1] = 1
                turn = 1
                critical = 2
187:    proc  1 (process:1) dekker_q4.pml:30 (state 18)
                process(1):j = 0
                process(1):i = 1
187:    proc  0 (process:1) dekker_q4.pml:30 (state 18)
                process(0):j = 1
                process(0):i = 0
187:    proc  - (atMostOne:1) _spin_nvr.tmp:2 (state 6)
2 processes created
dush123@ubuntu:~/Desktop/se/Q4$ spin -t -p -l -g dekker_q4.pml > output.txt
dush123@ubuntu:~/Desktop/se/Q4$ ls
dekker_q4  dekker_q4.pml  dekker_q4.pml.trail  output.txt  pan.b  pan.c  pan.h  pan.m  pan.p  pan.t  _spin_nvr.tmp
```

Using spin flags '-t –p –l –g' we can obtain the system behaviour in which the property gets violated.

The reason behind the violation of the property - atMostOne is:

We have 1 'do od' (outer most) and 3 'if fi'. There are two processes in the system, process-0 and process-1. Two global variables – flag and turn are declared. The **flags** of both processes are initialized to FALSE and **turn** to process-ZERO.

Consider the following scenario:

The process-1 executes first, and it arrives at critical section since flag[process-0] is false. Before this process-1 leaves the critical section (before the statements: turn = j and flag[i] = false), context switch happens and process-0 starts execution. Since flag[process-1] is true and turn = process-0 the location control shifts out of the outermost 'if fi' block (Note: this could be mitigated by using 'do od' in-place of 'if fi') and into the critical section. In this way both the processes are in the critical section simultaneously.

There might be other system behaviours where the property gets violated. The verifier found one such run of the system that violates the specified property.