CS 340
Lecturer: Dr. Simina Fluture

Lecture # 9

Read: **Class Notes**
Web Lecture
Textbook

Topics:
The Critical - Section Problem
Two Process-Software Incomplete Solutions
Peterson Solution for two-process Software

Two Process-Software Solution

Note: the order of the algorithms given in the notes is different than the one in the textbook.

No support of the hardware, operating system or programming language level is assumed.

1st Attempt: (similar with *algorithm 1* of the textbook)
Processes communicate using common variable ***turn***.
If *turn* = *i* then *Pi* is allowed to execute.
Initially, turn = 0; (or 1)

```
P0                              P1
While (true) {                   while (true) {
  while (turn != 0) do no-op;      while (turn != 1) do no-op;
  CS                             CS
  turn = 1;                        turn =  0;
  remainder;                       remainder;
}                                }
```

Processes wait for access to CS by *busy waiting*.

Mutual Exclusion - Satisfied

Negative aspects (for the first attempt):
Processes must strictly alternate in the use of their CS. => the pace of the execution is dictated by the slower of the two processes. This violates the **Progress Condition**.

Failing possibility
If one process fails, the other process is permanently blocked.
Note: in general when a solution to a Critical - Section problem is discussed, the failing possibility is not considered.

Second Attempt:
Each process should have its own key to the CS so that if one process is executing outside the CS the other one is able to get in to its CS.

replace the turn variable with a **shared global** variable initialized to false.

```
Boolean[] flag = new Boolean[2];
```

flag is initialized to 'false'

```
P0                                  P1
while(true) {                       while(true){
  while (flag[1]) do no-op;           while (flag[0]) do no-op;
  flag[0] = true;                     flag[1] = true;
      CS                                  CS
  flag[0] = false;                    flag[1] = false;
 remainder section;                  remainder section;
}                                   }
```

Mutual Exclusion is violated (why? Prove it)

3rd Attempt (similar with *2nd algorithm* of the textbook)
We might try to fix the 2nd attempt by changing the first 2 lines.
First the process sets its flag to true to signal that is ready to enter the CS. Next, the process checks if the other process is not also ready to get in the CS.

flag is initialized to 'false'

```
P0                                  P1
while (true) {                          while (true) {
  flag[0] =  true;                        flag[1] = true;
  while (flag[1]) do no-op;               while (flag[0]) do no-op;
    CS                                                CS
  flag[0] = false;                        flag[1] = false;
  remainder section;                      remainder section;
}                                               }
```

Mutual Exclusion condition - Satisfied (why? Prove it)

Progress condition is not satisfied (why? Prove it)

Correct Solution (Peterson Solution) (similar with the 3rd attempt of the textbook)
The state of both processes is provided by the global array variable 'flag'.
Also we need to impose some order on the activities of the two processes. The variable 'turn' solves the simultaneity conflicts.

Peterson's solution

```
                        turn = 0;
                        flag[0] =  false;
                        flag[1] =  false;
P0                                          P1
while(true){                                while(true){
  flag[0] = true;                             flag[1] = true;
  turn =1;                                    turn = 0;
  while (flag[1] and turn ==1) do no-op;        while (flag[0] and turn ==0) do no-op;
    CS                                          CS
  flag[0] = false;                            flag[1] = false;
  remainder section;                          remainder section;
}                                           }
```