OPERATING SYSTEM: CSET209

CONTENT

- **Software solutions to CSP**
- **Hardware solutions to CSP**

PRINTER SPOOLER DAEMON

In =3

0	F1.docx
1	F2.docx
2	F3.docx
3	
4	
5	
6	

Out=0

```
I1. LOAD  Ri  M[IN]
I2. STORE SD[Ri] "FILE_NAME"
I3. INCR Ri
I4. STORE M[IN] Ri
```

Problem: Data Loss

Let we have two process P1 and P2 in ready queue

Ready queue	P1 "f4.docx"	P2 "f5.docx"

P1 I
P1 II
P1 III
P2 I
P2 II

0	F1.docx
1	F2.docx
2	F3.docx
3	
4	
5	
6	

TURN VARIABLE OR STRICT ALTERNATION APPROACH

- Turn Variable or Strict Alternation Approach or Decker's algorithm is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock.

- This approach can only be used for only two processes. In general, let the two processes be Pi and Pj. They share a variable called turn variable. The pseudo code of the program can be given as following.

For Process Pi

```
Non - CS
while (turn ! = i);
Critical Section
turn = j;
Non - CS
```

For Process Pj

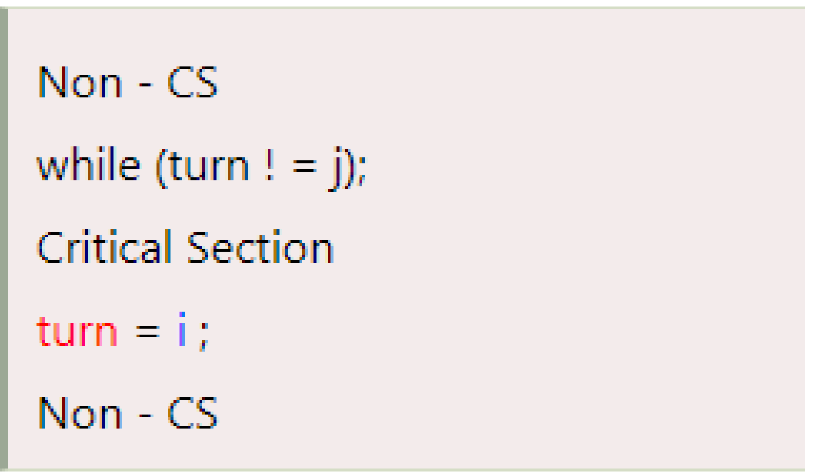
```
Non - CS
while (turn ! = j);
Critical Section
turn = i ;
Non - CS
```

TURN VARIABLE OR STRICT ALTERNATION APPROACH

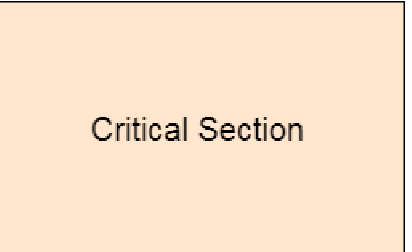
For Process Pi

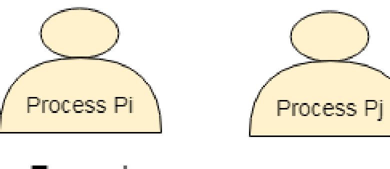
```
Non - CS
while (turn ! = i);
Critical Section
turn = j;
Non - CS
```

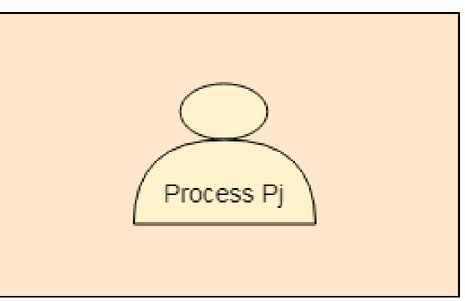
For Process Pj



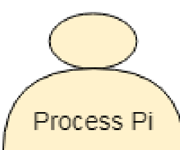
```
Non - CS
while (turn ! = j);
Critical Section
turn = i ;
Non - CS
```

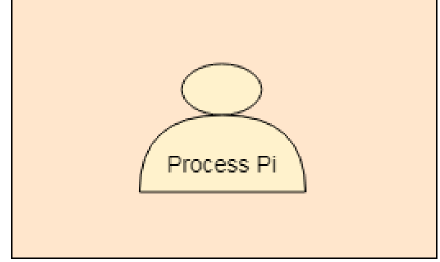




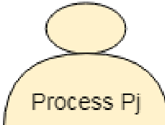


Turn = j





Turn = i

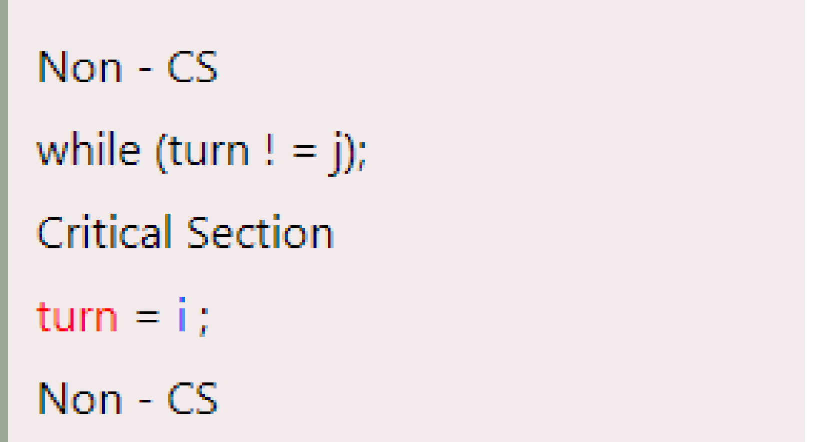


TURN VARIABLE OR STRICT ALTERNATION APPROACH

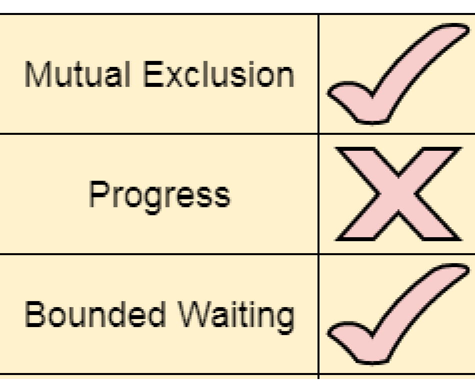
For Process Pi

```
Non - CS
while (turn ! = i);
Critical Section
turn = j;
Non - CS
```

For Process Pj



```
Non - CS
while (turn ! = j);
Critical Section
turn = i ;
Non - CS
```



Mutual Exclusion	✓
Progress	✗
Bounded Waiting	✓

PETERSON'S SOLUTION

globally defined

```
p2_wants_to_enter, p1_wants_to_enter, favored
```

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
            favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

Process 2

```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
            favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

- This is a software mechanism implemented at user mode.
- It is a busy waiting solution can be implemented **for only two processes**.
- It uses **two variables** which are globally defined.

PETERSON'S SOLUTION

```
globally defined
p2_wants_to_enter, p1_wants_to_enter, favored
```

Process 1

```
while(1){
   p1_wants_to_enter = True
   favored = 2

   while (p2_wants_to_enter AND
           favored = 2);
   critical section
   p1_wants_to_enter = False
   other code
}
```

Process 2

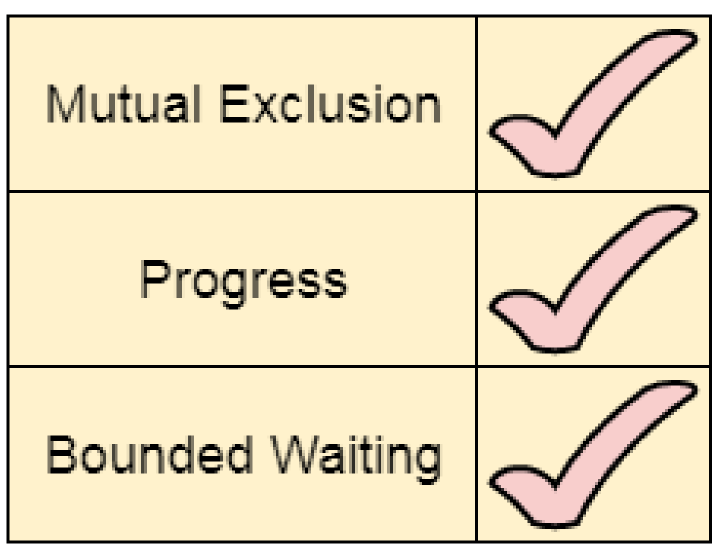
```
while(1){
   p2_wants_to_enter = True
   favored = 1

   while (p1_wants_to_enter AND
           favored = 1);
   critical section
   p2_wants_to_enter = False
   other code
}
```

Same as

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

ANALYSIS OF PETERSON SOLUTION



Mutual Exclusion	✓
Progress	✓
Bounded Waiting	✓

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
           favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

Process 2

```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
           favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

ANALYSIS OF PETERSON SOLUTION

- This is a two process solution. Let us consider two cooperative processes P1 and P2. The entry section and exit section are shown below. Initially, the value of interested variables and turn variable is 0.
- Initially process P1 arrives and wants to enter into the critical section. It sets its interested variable to True (instruction line 3) and also sets turn to 1 (line number 4). Since the condition given in line number 5 is completely satisfied by P1 therefore it will enter in the critical section.
- P1 → 1 2 3 4 5 CS
- Meanwhile, Process P1 got preempted and process P2 got scheduled. P2 also wants to enter in the critical section and executes instructions 1, 2, 3 and 4 of entry section. On instruction 5, it got stuck since it doesn't satisfy the condition (value of other interested variable is still true). Therefore it gets into the busy waiting.
- P2 → 1 2 3 4 5
- P1 again got scheduled and finish the critical section by executing the instruction no. 6 (setting interested variable to false). Now if P2 checks then it are going to satisfy the condition since other process's interested variable becomes false. P2 will also get enter the critical section.
- P1 → 6
- P2 → 5 CS
- Any of the process may enter in the critical section for multiple numbers of times. Hence the progress condition is satisfied.

QUESTION ON PETERSON

Question: If both process P_1 and P_2 are trying to enter into the critical section at the same time then which process will go to the critical section first.

globally defined

```
p2_wants_to_enter, p1_wants_to_enter, favored
```

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
            favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

Process 2

```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
            favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

HARDWARE SOLUTION 1: DISABLE INTERRUPT

Process 1

```
        while(1){
lock ---> disable interrupts ()
          critical section
unlock -> enable interrupts ()
           other code
        }
```

Process 2

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
     other code
}
```

- **Simple:** When interrupts are disabled, context switches won't happen
- **Requires privileges:** User level programs cannot disable interrupts. Only kernel level programs can.
- **Not suited for multicore systems. Why?**

HARDWARE INSTRUCTION: TEST AND SET LOCK

Does this scheme provides mutual exclusion?

lock=0

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Process 2

```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No

```
lock = 0
-----------------------------
P1: while(lock != 0);
-----------------------------   → context switch
P2: while(lock != 0);
P2: lock = 1;
-----------------------------
P1: lock = 1;
.... Both processes in critical section
```

HARDWARE INSTRUCTION: TEST AND SET LOCK

Only when the operation shown can be made atomic.

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Make atomic

HARDWARE INSTRUCTION: TEST AND SET LOCK

1. Executed **atomically**.
2. Returns the **original value of passed parameter**.
3. Set the new value of passed parameter to "TRUE".

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

HARDWARE INSTRUCTION: TEST AND SET LOCK USAGE

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

Process 1

```
while(1){
   while(test_and_set(&lock) == 1);
   critical section
   lock = 0;  // unlock
   other code
}
```

Process 2

```
while(1){
   while(test_and_set(&lock) == 1);
   critical section
   lock = 0;  // unlock
   other code
}
```

1. "lock" initialized to 0.
2. So, if two CPUs execute test_and_set at the same time, hardware ensures one test_and_set completes its execution before the other starts.
3. But the implementation shown here does not satisfy the **Bounded-waiting requirement.**

ANOTHER HARDWARE INSTRUCTION: COMPARE_AND_SWAP

1. Executed atomically.
2. Returns the **original value of passed parameter “value”.**
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

```
int compare _and_swap (int *value, int expected, int new_value)
{
       int temp = *value;

       if (*value == expected)
          *value = new_value;
    return temp;
   }
```

COMPARE_AND_SWAP: USAGE

```
do {
       while (compare_and_swap(&lock, 0, 1) != 0) ;  /* do
nothing */

        /* critical section */

      lock = 0;

        /* remainder section */
     } while (true);
```

```
int compare _and_swap (int *value, int
expected, int new_value)
 {
        int temp = *value;

        if (*value == expected)
          *value = new_value;
     return temp;
    }
```

- Shared integer "lock" initialized to 0.

SUMMARY OF SOLUTIONS TO CRITICAL SECTION PROBLEM

- Software based solution
  - Peterson’s solution
- Hardware based solutions
  - Disable interrupts
  - Atomic instructions: Test_and_Set and Compare_and_Swap
- OS solutions
  - Semaphore

THANK YOU

?