OPERATING SYSTEM

CONTENT

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

TURN VARIABLE OR STRICT ALTERNATION APPROACH

- Turn Variable or Strict Alternation Approach 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
```

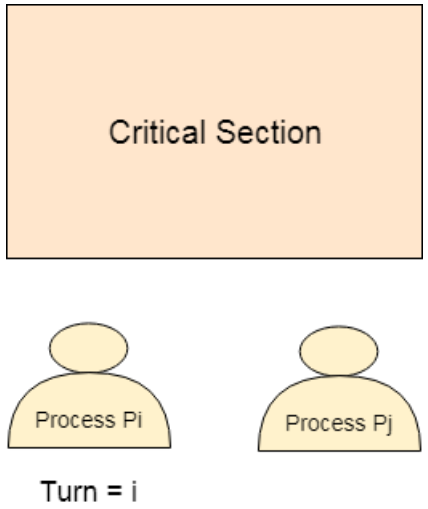
TURN VARIABLE OR STRICT ALTERNATION APPROACH

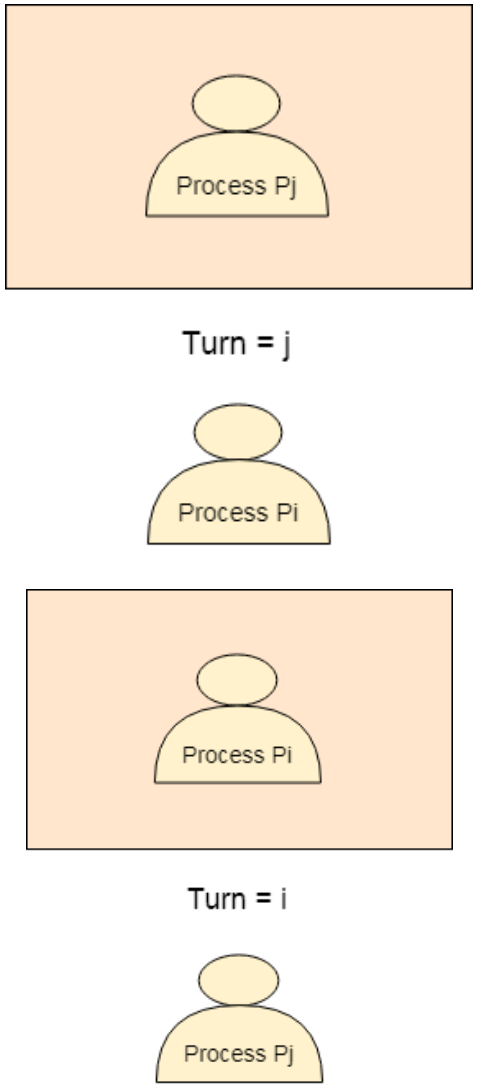
For Process Pi

```
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
```





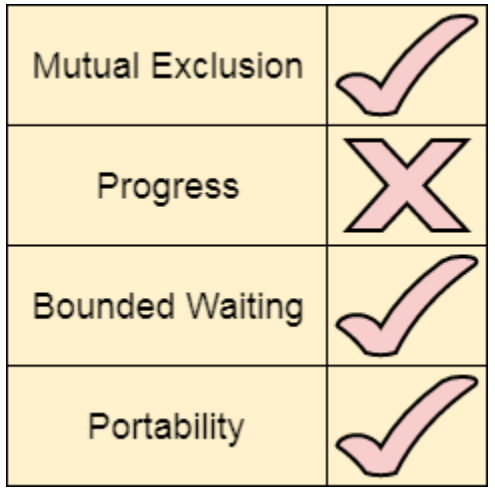
TURN VARIABLE OR STRICT ALTERNATION APPROACH

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
```



PETERSON'S SOLUTION

- 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 that are turn variable and interested variable.

```
# define N 2
# define TRUE 1
# define FALSE 0
int interested[N] = FALSE;
int turn;
voidEntry_Section (int process)
{
   int other;
   other = 1-process;
   interested[process] = TRUE;
   turn = process;
   while (interested [other] =True && TURN=process);
}
voidExit_Section (int process)
{
   interested [process] = FALSE;
}
```

ANALYSIS OF PETERSON SOLUTION

```
voidEntry_Section (int process)
{
    1. int other;
    2. other = 1-process;
    3. interested[process] = TRUE;
    4. turn = process;
    5. while (interested [other] =True && TURN=process);
}

Critical Section

voidExit_Section (int process)
{
    6. interested [process] = FALSE;
}
```

Mutual Exclusion	
Progress	
Bounded Waiting	
Portability	

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 procedure occurs in the cyclic order.

HARDWARE SOLUTION 1: DISABLE INTERRUPT

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption

```
do {
        ......
        DISABLE INTERRUPT
        critical section
        ENABLE INTERRUPT

        RemainderSection
} while (1);
```

What are the problems with this solution?

1. Mutual exclusion is preserved but efficiency of execution is degraded
   - while in CS, we cannot interleave execution with other processes that are in RS
2. On a multiprocessor, mutual exclusion is not preserved
3. Tracking time is impossible in CS

HARDWARE INSTRUCTION: TEST AND SET LOCK

- The *TestAndSet* instruction tests and modifies the content of a word **atomically** (non-interruptible)
- **Keep setting the lock to 1 and return old value.**

```
bool TestAndSet(bool *target){
  boolean m = *target;
  *target = true;
  return m;
}
```

```
do {
   … …

while(TestAndSet(&lock));
   critical section
   //free the lock
   lock = false;
   remainder section
} while(true);
```

What's the problem?

1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

ANOTHER HARDWARE INSTRUCTION: SWAP

- Swap contents of two memory words

```
void Swap (bool *a, bool *b){
    bool temp = *a;
   *a = *b;
   *b = temp:
}

bool lock = FALSE;
While(true){
    bool key = TRUE;
    while(key == TRUE) {
       Swap(&key, &lock) ;
    }
    critical section;
    lock = FALSE; //release permission
```

What's the problem?

1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

LOCK == FALSE

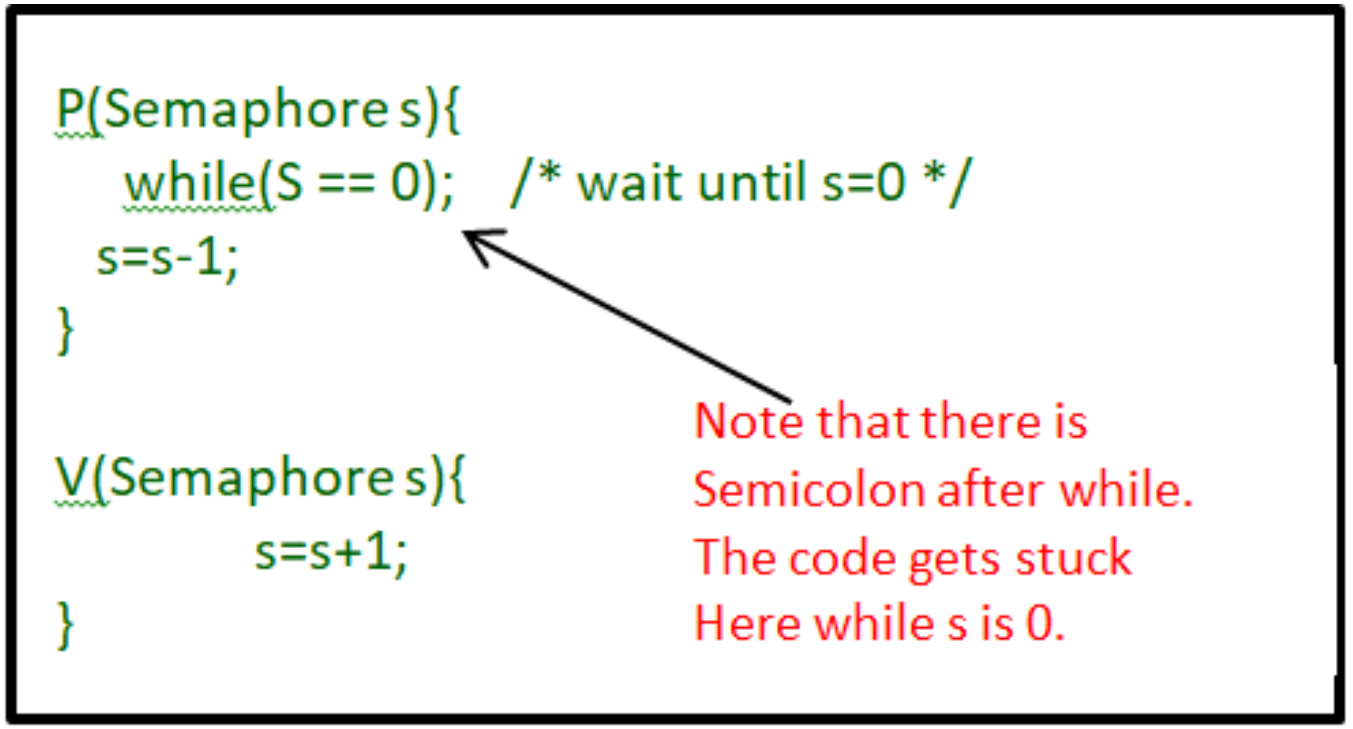
SEMAPHORES

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

Wait

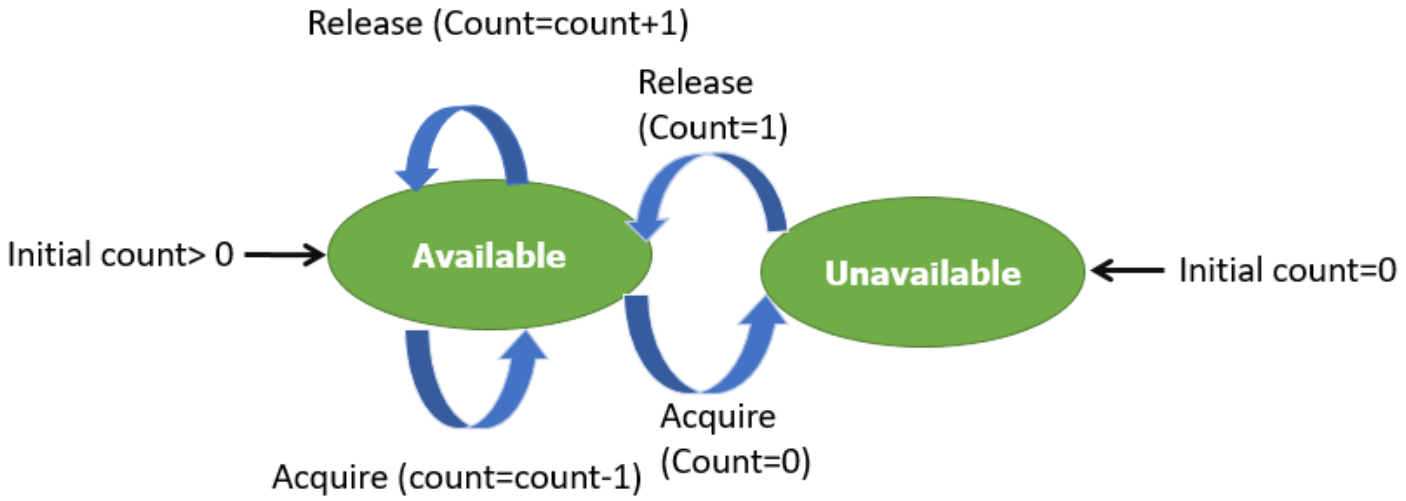
- **The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.**

- **Signal**
- **The signal operation increments the value of its argument S.**



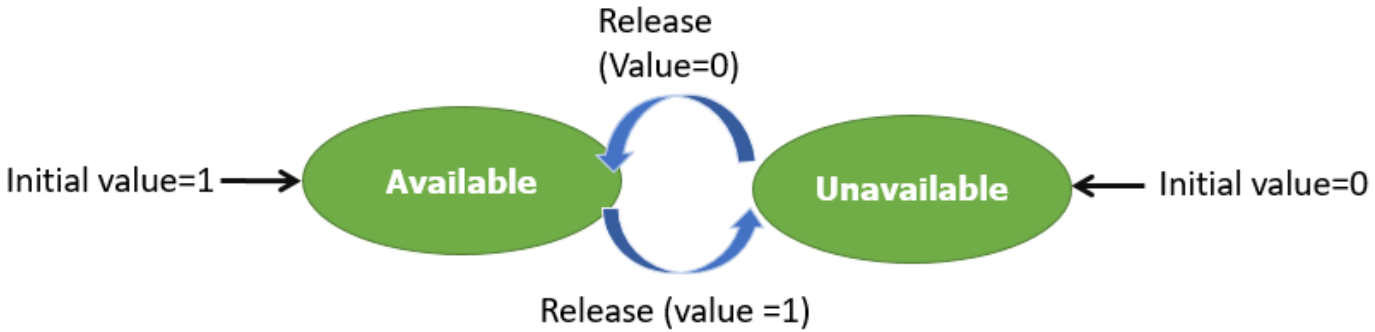
TYPES OF SEMAPHORES

- **Counting Semaphores:** This type of Semaphore uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state.
- **However, If the count is > 0, the semaphore is created in the available state, and the number of tokens it has equals to its count.**



TYPES OF SEMAPHORES

- **Binary Semaphores:**
  - The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1.
  - In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0. It is easy to implement than counting semaphores



COUNTING SEMAPHORE VS. BINARY SEMAPHORE

Counting Semaphore	Binary Semaphore
• No mutual exclusion	• Mutual exclusion
• Any integer value	• Value only 0 and 1
• More than one slot	• Only one slot
• Provide a set of Processes	• It has a mutual exclusion mechanism.

PRACTICE PROBLEMS BASED ON COUNTING SEMAPHORES IN OS-

A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

Solution-

We know-
•P operation also called as wait operation decrements the value of semaphore variable by 1.
•V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,
Final value of semaphore variable S
= 10 – (6 x 1) + (4 x 1)
= 10 – 6 + 4
= 8

PRACTICE PROBLEMS BASED ON COUNTING SEMAPHORES IN OS-

A counting semaphore S is initialized to 7. Then, 20 P operations and 15 V operations are performed on S. What is the final value of S?

Solution-

We know-
•P operation also called as wait operation decrements the value of semaphore variable by 1.
•V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,
Final value of semaphore variable S
= 7 – (20 x 1) + (15 x 1)
= 7 – 20 + 15
= 2

THANK YOU
?