The Critical-Section Problem

- N processes competing to use some shared resource or data
- Each process has a code segment, called *critical section*, in which the shared resource or data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Solution – establish an access protocol to enter the critical section in mutual exclusion.

Access Protocol to the Critical Section

- A process execute a “reservation” code before entering its critical section.
- This code – *prologue* to access - blocks the process while another process is in its critical section.
- A process leaving its critical section executes a code – *epilogue* to access – to release its critical section, and to inform other processes that the critical section is no more busy.

Access Protocol to the Critical Section

while (TRUE) {	**while (TRUE) {**
non critical ops	non critical ops
Reservation	***Reservation***
Critical Section 1	**Critical Section 2**
Release	***Release***
non critical ops	non critical ops
}	**}**

Mutual exclusion - specifications

- The solution is symmetric: the access decision does not depend on the relative priority of the processes.
- The solution does not depend on the relative speed of the processes.
- The solution allows a process to access its critical section even if another process is blocked _outside_ its critical section.
- The solution is deadlock free
- The solution is starvation free

Mutual exclusion: Solution 1

Process i:	Process j:
while (TRUE) {	**while (TRUE) {**
while (turn == j);	**while (turn == i);**
P_i ***critical section***	P_j ***critical section***
turn = j;	**turn = i;**
non critical section	***non critical section***
}	**}**

One at a time

Mutual exclusion: Solution 2

Process i:	Process j:
while (TRUE) {	**while (TRUE) {**
while (flag[j]);	**while (flag[i]);**
flag[i] = TRUE;	**flag[j] = TRUE;**
P_i ***critical section***	P_j ***critical section***
flag[i] = FALSE;	**flag[j] = FALSE;**
non critical section	***non critical section***
}	**}**

Both processes inside their critical region

Mutual exclusion: Solution 3

Process i:	Process j:
`while (TRUE) {`	`while (TRUE) {`
`  flag[i] = TRUE;`	`  flag[j] = TRUE;`
`  while (flag[j]);`	`  while (flag[i]);`
P_i *critical section*	P_j *critical section*
`  flag[i] = FALSE;`	`  flag[j] = FALSE;`
non critical section	*non critical section*
`}`	`}`

Deadlock

Mutual exclusion: Solution 4

Process i:	Process j:
`while (TRUE) {`	`while (TRUE) {`
`  flag[i] = TRUE;`	`  flag[j] = TRUE;`
`  turn = j;`	`  turn = i;`
`  while(flag[j] &&`	`  while(flag[i] &&`
`        turn == j);`	`        turn == i);`
P_i *critical section*	P_j *critical section*
`  flag[i] = FALSE;`	`  flag[j] = FALSE;`
non critical section	*non critical section*
`}`	`}`

Drawbacks of Solution 4

- Valid for two processes, but can be generalized
- Complex prologue and ineffective solution: busy form of waiting (spin lock).
- The complexity is due to the possibility that a process changes or tests a variable, and this operation is “invisible” to the other processes.
  – That means that a process can react to a value of a variable that meanwhile has been changed

Other Solutions to Mutual Exclusion

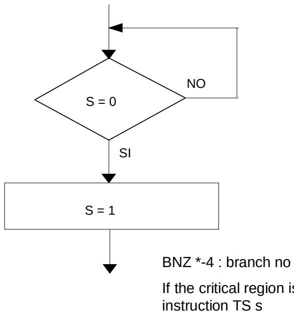
- Single processor systems:
  - **Disable interrupt** as prologue to the Critical Section.
  - **Enable interrupt** as epilogue at the end of the Critical Section.
- Multiprocessor systems with common memory:
  - **Test-and-set** special instruction on a lock variabile.
    - If lock is 0 the Critical Section is free
    - If lock is 1 the Critical Section is busy
  - The instructionTest-and-set tests the content of a variable and set it to 1 <u>in a single atomic cycle</u>

Test-and-set pseudo code

Tests and modifies atomically the content of a byte

```
char Test-and-Set(char *target){
    val = *target;
    *target = 1;
    return val;
}
```


Lock Implementation with IBM370 TS instruction



LOCK(s):
TS s
BNZ *-4

BNZ *-4 : branch no zero 4 bytes back

If the critical region is busy loop back to instruction TS s

Mutual exclusion with Test-and-Set

char *s* = 0;   // Initialization

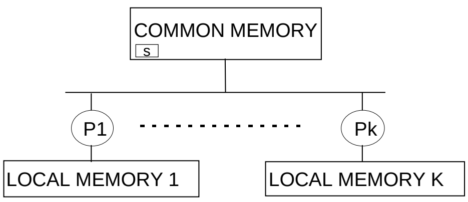
∀ Process P_i

```
while (TRUE) {
   while Test-and-Set (s);   // LOCK(s)
   critical section
   s = 0;                    // UNLOCK(s)
   non critical section
   }
```


Mutual exclusion without starvation

```
while (TRUE) {
  waiting[i] = TRUE;
  key = TRUE;
  while (waiting[i] && key) key = test_and_set(lock);
  waiting[i] = FALSE;
   critical section of Pi
  j = (i+1) % N;
  while ((j <> i) and (! waiting[j])) j = (j+1) % N;
  if (j == i) lock = 0;
  else waiting[j] = FALSE;
  non critical section
}
```


Multiprocessor Architeture



Spin Lock Drawbacks

- Busy form of waiting
- Scheduling cannot be controlled by the programmer
- Bus occupation