

CRITICAL SECTION PROBLEM

- CSP Solution needs to satisfy three important properties of **MUTEX**, **PROGRESS** and **BOUNDED WAIT**
- Meaning of each property follows:
- **MUTEX** - Only one process at a time can be executing in their critical section.
- **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then **only the processes not in their remainder sections** can participate in the decision, and the decision cannot be postponed indefinitely .
- **processes cannot be blocked forever** waiting to get into their critical section

CRITICAL SECTION PROBLEM

- **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management.



CRITICAL SECTION PROBLEM

- Kernels can be of two types:
- Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions,
- but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.
- Preemptive kernels allow for real-time operations, but must be carefully written to **avoid race conditions**.
- This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

Peterson's Solution

Peterson's Solution is a classic software-based solution to the critical section problem

```
do {
```

```
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = FALSE;
```

remainder section

```
} while (TRUE);
```

Process P_i structure in Peterson's Solution

Peterson's Solution

do {

```
flag[j] = TRUE;  
turn = i;  
while (flag[i] && turn == i );
```

critical section

```
flag[j] = FALSE;
```

remainder section

} while (TRUE);

Process P_j structure in Peterson's Solution

- **boolean flag[2]** - Indicates when a process **wants to** enter into their critical section.
- When process i wants to enter their critical section, it sets **flag[i]** to true.
- Peterson's solution is based on two processes, P_0 and P_1 , which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is P_i , and the "other" process is P_j . (I.e. $j = 1 - i$)
- Peterson's solution requires two shared data items:
 - **int turn** - Indicates whose turn it is to enter into the critical section. If $turn = i$, then process i is allowed into their critical section.

In the entry section,

- process i sets the flag indicating a desire to enter the critical section.

Then turn is set to *j* to allow the ***other*** process to enter their critical section ***if process j so desires.***

The while loop is a busy loop

(notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.

Process i resets the flag[i] in the exit section, allowing process j to continue if it has been waiting.

In the entry section,

- process *i* sets the flag indicating a desire to enter the critical section.

Then **turn** is set to ***j*** to allow the ***other*** process to enter their critical section ***if process j so desires.***

The while loop is a busy loop


(notice the semicolon at the end), which makes process *i* wait as long as process *j* has the turn and wants to enter the critical section.

Process *i* resets the flag[*i*] in the exit section, allowing process *j* to continue if it has been waiting.

Peterson's Solution – Proof of Correctness

- **Mutual exclusion** - If one process is executing their critical section ;
- when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.

- **Progress -**
- Each process can only be blocked at the while if the other process wants to use the critical section ($\text{flag}[j] == \text{true}$), AND it is the other process's turn to use the critical section ($\text{turn} == j$).
- If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, **will set $\text{flag}[j]$ to false**, releasing process i .
- The shared variable turn assures that **only one process at a time can be blocked**, and the flag variable allows one process to release the other when exiting their critical section.

- 
- **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn.
 - Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.
 - Note that the instruction "turn = j" is **atomic**, that is it is a single machine instruction which cannot be interrupted