• Four conditions to have a “good solution” to mutual exclusion
  o No two processes may be simultaneously in their critical regions (see above)
  o No assumptions may be made about speeds or the number of CPUs
  o No process running outside its critical region may block other processes
  o No process should have to wait forever to enter its critical region

• Busy Waiting Mutual Exclusion Techniques
  o Introduction
    ▪ Look at some of the approaches taken to provide mutual exclusion.
      ▪ Software-based lock variable
        • Technique
          o Shared lock variable that is initially set to zero
          o When process wants to enter, it checks the lock.
          o If lock is zero, it sets it to one and enters the region.
          o If it’s already one, it waits until it becomes zero
        • Flaw – race condition on check/modify of the lock
  o Disabling Interrupts – attacking preemption
    ▪ Technique
      • Just after entering the critical region, disable all interrupts
      • Right before leaving critical region, enable them again
    ▪ Implications
      • With interrupts disabled, preemption is disabled
      • Since another process can’t run, mutual exclusion is guaranteed
    ▪ Problems
      • Giving a user the ability to disable interrupts is very dangerous
        o They might never turn them back on
      • Could be useful for kernel though
        o Avoids race conditions for scheduler, etc.
      • Only works on single-CPU systems
        o A process can only disable interrupts on its own CPU
        o With multi-core chips, this is becoming a much less likely solution
  o Strict alternation – modified version of software lock
    ▪ Technique
      • *turn* variable (initially zero) keeps track of whose turn it is to enter the critical region

```c
while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}
```
Implication

- Because each process is waiting for a different condition, the race condition is avoided
- The processes strictly alternate between access to the critical region

Problems

- Uses busy waiting: continuously testing a variable until some value appears
  - Wastes CPU cycles
  - Definition: Spin lock – any locking mechanism that uses busy waiting
- Strict alternation violates condition three above
  - Suppose process 0 and 1 are both in their noncritical regions
  - Process 0 quickly enters and exits the critical region, setting turn to 1.
  - Process 0 finishes its non-critical region again and wants to enter the critical region while process 1 is still in its non-critical region.
- Example: With print spooler, process 0 couldn’t print another file because process 1 hasn’t printed a file yet.

Peterson’s Solution – software solution that avoids strict alternation

```c
#define FALSE 0
#define TRUE 1
#define N 2    /* number of processes */

int turn;    /* whose turn is it? */
int interested[N];    /* all values initially 0 (FALSE) */

void enter_region(int process);    /* process is 0 or 1 */
{
    int other;    /* number of the other process */
    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE;    /* show that you are interested */
    turn = process;    /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)    /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```
- **Technique**
  - Consists of two C functions that should be called before/after using shared variables
  - If other processes aren’t interested, immediately allows access
  - Uses busy waiting if both processes are interested

- **Implications**
  - Doesn’t violate condition 3 above any more
  - If \( turn == process \), but the other process is interested, the current process clearly changed \( turn \) last and should wait until the other process, which knows nothing of it, completes

  o TSL Instruction – hardware solution
    - **Technique**
      - “Test and Set Lock” instruction is provided on some computers
      - Read contents of memory word \( lock \) into a register and store a non-zero value at the memory address \( lock \)
        - Guaranteed to be indivisible
        - CPU locks the memory bus to prohibit access to the memory until it is done
        - JNE – jump on inequality

    ```
    enter_region:
    TSL REGISTER,LOCK | copy lock to register and set lock to 1
    CMP REGISTER,#0 | was lock zero?
    JNE enter_region | if it was nonzero, lock was set, so loop
    RET | return to caller; critical region entered
    
    leave_region:
    MOVE LOCK,#0 | store a 0 in lock
    RET | return to caller
    ```

    - Because the read/modify instruction is atomic, there can be no race condition

  - Another hardware technique
    - XCHG – exchanges the contents of two locations atomically
      - Can be used on a register and memory word
      - x86 CPUs use XCHG for low-level synchronization
Provides essentially the same functionality as TSL

**Implications**
- Locking memory bus works for multi-processor systems
  - Interrupt disabling on a single CPU wouldn’t prevent others from accessing memory

**General Problem – Busy Waiting**
- Wastes CPU time
- Other unexpected effects
- Priority Inversion Problem
  - Assume two processes that share storage: H with high priority and L with low priority
  - Assume a scheduler that runs H whenever it is in the ready state
  - Situation: L in critical region when H becomes ready (I/O completes)
    - H begins busy waiting
    - L never gets a chance to run because it isn’t scheduled
    - H loops forever
  - Priority inversion violates condition 4