❖ The Critical-Section Problem

- The critical section is a code segment that accesses shared variables and has to be executed as a single action.
- Consider a system consisting of *n* processes: {*P*0, *P*1, ..., *Pn*-1}.
- Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- When one process is executing in its critical section, ***no other process is allowed to execute in its critical section.*** (That is, no two processes are executing in their critical sections at the same time.)
- The ***critical-section problem*** is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section in **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section** (non critical-section code).

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

❖ Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion:** If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only the processes not in their remainder section can select the process that will enter its critical section next; the selection cannot be postponed indefinitely.
3. **Bounded waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

❖ **Peterson's Solution**

- Assume that the LOAD and STORE instructions are atomic; i.e., cannot be interrupted.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered *P*0 and *P*1 or (*Pi and Pj).*
- The two processes share two data items:

int turn;
boolean flag[2];

- The variable **turn** indicates whose turn it is to enter its critical section. (if turn == i, then process *Pi* is allowed to execute its critical section.)
- The **flag** array is used to indicate if a process is ready to enter its critical section. (if flag[i] is true, it indicates that *Pi* is ready to enter its critical section.)

```
do {

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

          critical section

     flag[i] = false;

          remainder section

} while (true);
```

To enter the critical section, process *Pi* first sets **flag**[i] to be **true** and **turn** to the value **j**, (if the other process wishes to enter the critical section, it can do so).

```
// code for process 0

do {

flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1);

(critical Section)

flag[0] = false;

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

```
// code for process 1

do {

flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0);

(critical Section)

flag[1] = false;

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

We need to prove that this solution is correct by showing that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

- **Property 1: (Mutual exclusion)**: the turn value can be either 0 or 1 but cannot be both values at the same time, as a result, mutual exclusion is preserved.
- **Property 2: (Progress)**: when *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section, hence, the progress requirement is satisfied.
- **Property 3: (Bounded-waiting)**: *Pi* will enter the critical section after at most one entry by *Pj,* that is, bounded-waiting requirement is met.

❖ Mutex Locks

- **Mutex Locks** is a software tool to solve the critical-section problem. (In fact, the term ***mutex*** is short for ***mut***ual ***ex***clusion.)
- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The **acquire( )** function acquires the lock, and the **release( )** function releases the lock.
- A mutex lock has a boolean variable ***available*** whose value indicates if the lock is available or not.

- If the lock is available, a call to **acquire( )** succeeds, and the lock is then considered unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

```
do {

      acquire lock

            critical section

      release lock

            remainder section

} while (true);
```

The definition of acquire( ) is as follows:

```
acquire( )
{
  while (!available)  ;              /* busy wait */

  available = false;
}
```

The definition of release( ) is as follows:

```
release( )
{
        available = true;
}
```

- Calls to either **acquire( )** or **release( )** must be performed atomically.
- The main disadvantage of the implementation is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to **acquire( )**.