Critical Section

Critical Section (C.S) is a part of the code or place or buffer, which is common between two or more processes like variable, or resource i.e. both want to use is common between two or more processes.

Non-Critical Section is a part of code or place or buffer, where there is no common variable or resource between two or more processes.

Synchronization Conditions or Rules

condition 1 and 2 are primary or mandatory conditions and 3 and 4 are secondary conditions for process synchronization.

1. **Mutual exclusion**

if process P1 enters the Critical Section (C.S) then at same time p2 cannot enters the C.S, as P1 is already in the C.S and no other process can enter at that time.

Before entering C.S, P1 first executes the entry section code and after that it enters the C.S, if P2 is late to execute entry section code, then it cannot allow to enter C.S after that.

There are many methods to enable "Process Synchronization" like **Semaphores, Monitors, Mutex, Peterson Solution** etc.

If both processes p1 and p2 enters the C.S at the same time, then we have “Race Condition”.

2. Progress

No Progress means for e.g. P1 wants to enter C.S but P2 is not interested to enter C.S, and not allowing P1 to enter C.S (i.e. through code written in P2 entry section avoiding it to enter in C.S), that is called No Progress. Thus no process can enter the C.S. But progress should be there. If either

P1 or P2 interested to enter C.S then it can enter the C.S without any restrictions.

3. Bounded Waiting

Bounded Waiting means for e.g. P1 enters the C.S and comes out after executing code in C.S, then P2 is interested to enter C.S, but before P2 entry, P1 again enters the C.S and comes out after executing code in C.S again for second times, then a bound must be existing on number of times any process to enter the C.S. For e.g. P2 can enter the C.S for first time only, and P1 enters the C.S for infinite times, then such thing could not be happening or acceptable for process synchronization, called as unbounded wait, that causes starvation.

If any process enters C.S and other process waits, then it is called bounded wait. Because any process can use C.S many times and some processes use C.S least times, but unbounded wait condition should not happen.

4. No assumption related to hardware or speed

Means if you design a process synchronization solution then it must be compatible with any hardware, computer specification, memory, processor, O.S or speed of processor should be 1GHz or 2GHz etc to run fast on that computer.

Critical Section Handling in O.S

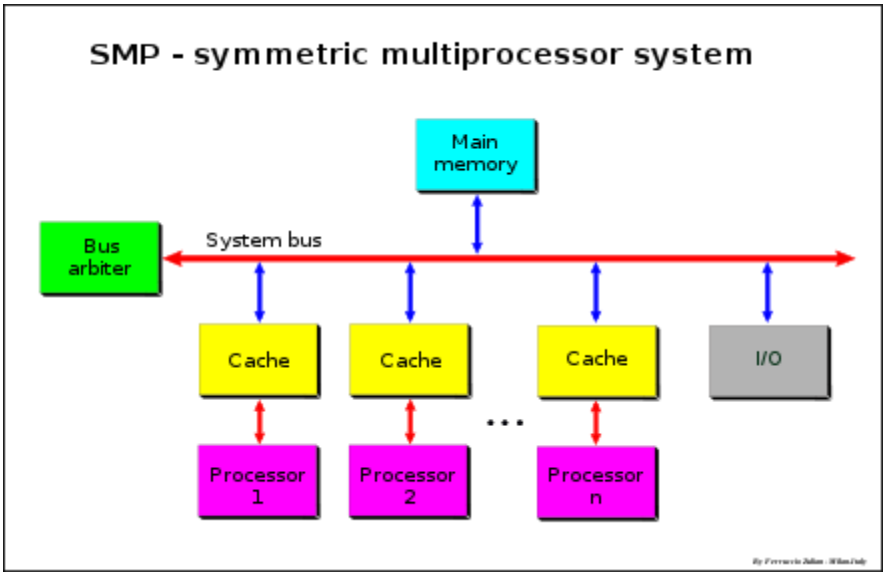
Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as

only one process is active in the kernel at a time. We cannot say the same about nonpreemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Furthermore, a preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. Of course, this effect can be minimized by designing kernel code that does not behave in this way.

Windows XP and Windows 2000 are nonpreemptive kernels, as is the traditional UNIX kernel. Prior to Linux 2.6, the Linux kernel was nonpreemptive as well. However, with the release of the 2.6 kernel, Linux changed to the preemptive model. Several commercial versions of UNIX are preemptive, including Solaris and IRIX.

Symmetric multiprocessing or shared-memory multiprocessing (SMP) involves a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single, shared main memory, have full access to all input and output devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. Most multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.



Peterson's Solution

Good algorithmic description of solving the critical section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress and bounded waiting.

- Works on two processes at the same time only.
- The two processes share two variables:
    - int turn;
    - Boolean flag[2]

- The variable turn indicates whose turn it is to enter the critical section.

For e.g.
If turn =i means process i turn to enter critical section, if turn=j means process j turn to enter critical section.

- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P is ready!.

For e.g.
If flag[i] = true means process i is ready to enter critical section.
If flag[j] = true means process i is ready to enter critical section.

Algorithm
Structure of Pi in Peterson' s solution

```
do {
flag [i] = true;      //means process i ready to enter C.S.
turn=j;               // process i humbly give turn to process j to enter C.S,
                      if process j wanted to enter C.S.

while (flag [j] && turn == [j]);      //means if both conditions are true
                                      then while loop is also true and the
                                      control remains in the block and never
                                      goes to the next statement and
                                      semicolon means no statement in the
                                      while block. If while condition
                                      becomes false, then control jumps to
                                      next statement after this condition.
```

Critical section

```
Flag [i] = false;

Remainder section

} while (true)
```

Algorithm

Structure of Pj in Peterson' s solution

```
do {
flag [j] = true;      //means process i ready to enter C.S.
turn=i;               // process i humbly give turn to process j to enter C.S,
                      if process j wanted to enter C.S.

while (flag [i] && turn == [i]);      //means if both conditions are true
                                      then while loop is also true and the
                                      control remains in the block and never
                                      goes to the next statement and
                                      semicolon means no statement in the
                                      while block. If while condition
                                      becomes false, then control jumps to
                                      next statement after this condition.
```

Critical section

```
Flag [j] = false;

Remainder section

} while (true)
```

Assumption that both algorithms of Pi and Pj runs simultaneous.
Let for e.g. Pi set the flag[i] = true.
And at the same time Pj set the flag [j] =true also.
Now in Pi algorithm, turn=j and in P2 algorithm it set turn=i (i.e. as both turn and flag variables are shared between Pi and Pj).

Now in Pi algorithm, while (flag[j] && turn==[j]) executes, then flag[j]=true but turn=[i] (i.e. as Pj lastly set turn=[i]), thus both are not true, because Pj lastly set turn=[i], thus while condition becomes false (and turn value is not equal to j, its i) so control comes to the next statement i.e. critical section. So Pi will be executing critical section and set flag [i] = false after exiting critical section, now Pi executing reminder section. During that time, we will see whether Pj is executing critical section or not, now check in P2 algorithm while condition. i.e.
while (flag[i] && turn==[i]) so what is flag of i, it is true and turn==[i] (as Pj lastly set turn=[i]) so both conditions are true, thus while condition becomes true and control does not come out, but control remains in the while loop. Now after Pi completes execution in the critical section, it set flag [i] = false, thus now when Pj again checks the while condition
while (flag[i] && turn==[i]), thus both are not true, because Pi lastly set after exiting critical section set flag[i]=false, while turn=[i] true, thus while condition becomes false, so control comes to the next statement i.e. critical section. Thus now Pj enters the critical section. After executing critical section, it set flag [j] = false after exiting critical section, now Pj is executing reminder section.
So after executing Peterson's algorithm, we satisfied all the three Process Synchronization conditions or rules.

Synchronization Hardware

Solution to Critical Section problem using Locks

```
do {
    acquire lock
                Critical section
    release lock
                Reminder section
  } while (true);
```

1. While (lock==1); // or while (lock!=0)
2. Lock =1;
3. Critical section
4. Lock=0;

Let suppose we have two processes P1 and P2. We take initially Lock=0 or Lock=false (means critical section is empty, otherwise it is full).

Case 1:

P1 starts executing statement 1
as Lock=0 so while (0==1), condition becomes false, so control goes to statement 2.
P1 set Lock =1 and comes in critical section.
At that time for example P2 also want to go to critical section, as already lock = 1, so when P2 executes statement 1, so while (1==1), so P2 stuck in infinite loop, and could not go to critical section.
In the meanwhile, when P1 comes out of the critical section, it execute statement 4 and set Lock=0, now if P2 wants to enter so, P2 can enter as while (0=1), condition becomes false, so control goes to statement 2.
P2 set Lock=1 and comes in critical section.

Case 2:

P1 successfully executes statement 1, in the meantime, P1 preempt, if P2 want to enter the critical section in the meantime, so it get Lock=0, so P2 execute statement 1 successfully and goes to statement 2 and set Lock=1 and comes in critical section. In the meantime, P1 comes back and resumes from statement 2 as it already executed statement 1, so it set lock=1 (i.e. whatever, the value of lock is it, it set to 1) and comes in critical section, as P2 is already in the critical section, so mutual exclusion fails and violates, so Locks does not handle guarantee mutual exclusion in case 2.

Solution using Test and Set Instruction

As we discussed problem in Case 2 (Locks), such problem is solved by Test and Set Instruction by merging statement 1 and statement 2 into one (or atomic). i.e.

```
while (test_and_set (&lock))
/* do nothing */
        /* critical section */
Lock=false
        /*reminder section */
} while (true);
```

Solution using Compare and Swap

```
Do {
        while (compare_and_swap (&lock, 0, 1) !=0);
        /*do nothing */
                /*critical section */
Lock=0;
        /*reminder section*/
} while (true);
```

Mutex Locks

acquire() and release()

Semaphore

A semaphore is an integer variable which is used in mutual exclusion. Counting (-infinity to infinity) and binary (0,1).

P() – called Down or wait operations

V() – called as Up or signal or Post or Release operations

Down means entry section. S is a semaphore, it is an integer variable. Let S (can vary from –infinity to infinity), let S=3 (initialized), Let P1 wants to enter critical section, so it has to execute entry section, so
S.value = S.value -1, so S.value =3-1 = 2. In statement 3, (S.value <0), so the condition returns false, so control goes to else part means P1 comes into critical section. Let suppose P2 wants to come to critical section, so as final value of S is 2, so P2 executes S.value=S.value-1, so S.value=2-1 = 1. Now check (S.value < 0), so the condition return false, control goes to else part means P2 also comes into critical section. Now P3 wants to come to critical section, so P3 executes S.value=S.value-1, so S.value=1-1 = 0. Now check (S.value < 0), so the condition returns to false, and P3 also comes to critical section. Now if P4 want to comes to critical section, so P4 executes S.value=S.value-1, so S.value=0-1 = -1. Now check (S.value < 0), so the condition returns to true, and P4 executes block code inside if condition and CPU puts the P4 process PCB into RAM (suspend list) or call Sleep() or block the process. If now P5 wants to comes, so it wouldn't come in critical section and CPU also puts P5 process PCB into suspend list and call Sleep() or block the process. The current value of S = -2.
If we set S=0 then no process will be able to come inside critical section. If P1, P2 or P3 wants to come out of the critical section, so it has to run exit code or Up code.

```
Down (Semaphore S)
{
S value= S value - 1;
If (S value < 0)
{
Put process (PCB) in the Suspend list and Sleep ();
}
else
return;
```

```
}

Up (Semaphore S)
{
S value= S value + 1;
If (S value <= 0)
{
Select the process from the Suspend list and Wakeup ();
}
else
return;
}
```

As S= -2 , when P1 executes S.value = S.value +1, S.value = -2+1= -1. Now if (S.value <=0), true or yes. Now select the process from the Suspend list and Wakeup() (i.e. pick one by one process and wakeup means put into ready state), Now CPU pick P4 and call wakeup. P2 can also come out and will execute entry section code, so S.value = -1+1= 0, now pick another process which was P5 and put into ready state. Now both P1 and P2 has comeout of the critical section, but P3 is already there in critical section. Now P3 wants to come out of the critical section, so S.value =0+1 = 1, so control will come out of if condition and P3 will also come out of the critical section.

For e.g. If value of S=10, then 10 processes can come inside the critical section (so 10 successful operations will be performed), but if S<=0 then all the other processes will not come inside the critical section.