Purpose of Semaphores:

The problem of Critical section

To establish a communication between two processes there must be shared memory among them which they both can access; these are applicable to both competing and cooperative processes. The part of the code segment where shared memory is accessed is known as critical section. With the memory being accessed between multiple processes there arises some issues like data inconsistency due to simultaneous read and write because of different processes. The problem is known as race problem where the result is dependent on the order of execution of processes. The critical section problem revolves around designing a protocol so to allow process to use shared memory and coordinate their activities.

The following conditions must be met for solving the critical section problem:

1. Mutual exclusion: While one process is accessing the shared memory, other process isn’t allowed to access it
2. Progress: No process should block/restrict other process from accessing the critical section from accessing the critical section if it is not using it.

Semaphore:

The Dutch scientist E.W. Dijkstra proposed a software solution for the critical section for control synchronization using semaphore. Semaphore is integer variable which can be accessed only using special operations, wait and signal which are atomic in nature. Semaphore is a non-busy waiting solution and makes the process go in a sleep state when it can't access the shared memory and wakes up when it can. Semaphore also supports bounded waiting.

There are two types of semaphores:

1. Binary Semaphore: The semaphore variable can have value only 0 and 1 and allows only one process in the critical section at a time.
2. Counting semaphore: The value ranges over an unrestricted domain which allows to allocate resources from the pool of them.

Working of Semaphores:

To synchronize the processes using semaphore, the following steps are done

1. To go into the critical section the process must first call the wait/P(S),
2. Execute the critical section code,
3. after using the shared memory call signal/V(S).

Entry Section: P(S);

Critical Section:

Exit Section: V(S);

The two functions wait and signal also known as P(S) and V(S) are atomic function and are implemented as system calls. Here S is the semaphore variable. The function definition is as follows:

```
P(S) {
        S--;
        If(S<0) {
                //add process to queue L;
                block();
                }
}

V(S) {
S++;
        If(S<=0){
                //remove process p from queue L;
                wakeup(p);
        }
}
```

As semaphore is a non-busy waiting solution it has a queue of processes which have a list of sleeping processes. These processes are put in the queue whenever the value of semaphore is <=0 by using P(S) otherwise it allows it to access the shared memory. A process is taken out from the queue when a process has called the V(S) and when there is a process in the queue. Here block and wakeup are system calls which are used to put in the process in the waiting state and place it in the queue and for taking it out of the queue.

Problems with semaphores:

1. Deadlocks: The semaphore can be thought of a resource and thus can result in a deadlock if the condition of deadlock are satisfied, that are mutual exclusion, hold and wait, circular wait, no preemption.
2. Priority Inversion: Priority inversion occurs when a higher priority process wants to access the resource that is already accessed by a lower priority. This is solved with priority inheritance where the lower priority is inherits the highest priority till, they have accessed the resource. [i]

Other mechanisms for solving critical section problem:

1. **Test set and lock:**

   The TSL solution provides us with an atomic operation of testing and setting the lock to locked state which ensures mutual exclusion and progress. As this is a busy waiting solution this approach suffers from spin lock. This is not an architectural neutral solution as the TSL statement must be compatible with the hardware.

2. **Peterson's Solution:**

Before implementation of Peterson's solution these solutions were thought of, but they came with their own limitation.

a. **Lock variable**

   Lock variable is software approach in which a lock variable in which a variable is used to mark if the shared memory can be accessed by the process or not. As the operation of locking and unlocking of the lock variable are not atomic operation mutual exclusion is not fulfilled here.

b. **Strict alteration**

   This is also a software approach for synchronization between two processes in which a turn variable is used to denote turn of a process to access the shared memory. If the turn is set to 0 the P0 process accesses and if 1 P1 access. This approach satisfies the mutual exclusion property but can’t satisfy progress

Peterson’s solution uses two variables, namely bool array flag variable and turn variable, to solve the critical section problem. The flag array variable is initially set to 0 meaning right now no variable is interested to access the shared memory. This solution is only applicable to two processes and is a busy waiting solution. This solution also supports bounded waiting.

Pseudo code:
```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   /* critical section */
   flag[i] = false;
   /* remainder section */
}
while (true);
```

3. **Disabling Interrupts:**
   In this approach interrupts are disabled whenever going into the critical section and enable it after. This approach also provides us with mutual exclusion and progress but isn’t architectural neutral.

4. **Swap:**

   This is also a hardware assisted approach for process synchronization where swap() function is provided as an atomic function which swaps the contents of the parameters using pass by address. Two variables namely turn, and lock are used. If lock is false, the critical section can be accessed otherwise not. Lock is initialized with false.

Pseudo code:

```
do{
   while (compare_and_swap(&lock, 0, 1) != 0) ;
   /* critical section */
   lock = 0;
   /* remainder section */
}
while (true);
```

[ii]

5. **Monitor:**

A Monitor is a high-level process synchronization construct which abstracts away all the timing information. It holds conditionals, shared memory, and timing information all under the same hood. A Monitor class is an abstract data type which contains shared data variables and procedures. The variables are private and cannot be accessed from outside of the construct, only its procedures can access the variables. Only one thread can access a monitor class object at one time.[iii]

References:

1. https://medium.com/@angadsharma1016/process-synchronization-monitors-in-go-d31f4c42fce7
2. https://www.tutorialspoint.com/monitors-vs-semaphores
3. https://doc.lagout.org/operating%20system%20/linux/Understanding%20Linux%20Kernel.pdf
4. https://www.classes.cs.uchicago.edu/archive/2017/winter/510811/LabFAQ/lab7/Semaphores.html
5. http://index-of.es/Varios-2/Modern%20Operating%20Systems%204th%20Edition.pdf
6. http://perugini.cps.udayton.edu/teaching/courses/cps346/lecture_notes/classical.html
7. https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L8_SemaphoreMonitor_v2.pdf
8. http://www2.cs.uregina.ca/~hamilton/courses/330/notes/synchro/node3.html

[i] https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab7/Semaphores.html

[ii] http://www2.cs.uregina.ca/~hamilton/courses/330/notes/synchro/node3.html

[iii] https://medium.com/@angadsharma1016/process-synchronization-monitors-in-go-d31f4c42fce7