Synchronization Summary

Synchronization is all about ordering the execution of tasks (on a machine, in real life) in a way that suits you (as a programmer, as a person, …).

A good synchronization tool is one that enables its user to apply his/her synchronization logic to resolve a problem.

Having this in mind, let us work on a real synchronization problem, devise a human solution, then map it onto a solution that could work on our computers.

Problem : Manage how 5 computers can be used by 50+ users

Please, take some time and think about how you would handle this problem as a manager !

You would probably come up with the following solution.

1. At startup, all computers are available, and the first five requesters are going to get a computer
2. If another user requests a computer at this point, since there are none available, put him on a waiting list (physical, or simply write down his contact details)
3. When a computer becomes available, get a person (why not the first on the list, or the last one, or …) from the waiting list (if it's not empty) to use the computer

We have just devised a synchronization protocol (data + operations) that manages how the computers can be requested and used by many users correctly !

If this solution works for you as a person, the nit must be a good tool to implement and use in your programs to synchronize tasks.

Sempahores

The previous solution/protocol/tool can be mapped onto a programming tool ; this is what you've already seen as the definition of a semaphore. The colours in the following section identify the elements used to resolve the computer usage problem.

Structure

Cpt : number of available resources

Waiting List

Operations

```
ACQUIRE {
    If (resource is not available)
        Block task and add it to the waiting list

    // resource is available
    Cpt--;
}

RELEASE {
    Cpt++ ; // one more available resource
    If (waiting list is not empty)
        Wake up a blocked task and hand it the resource
}
```

As you can see, semaphores are something that you naturally use to handle synchronization problems.

As the semaphore is shared by multiple tasks in order to synchronize their executions, therefore, the functions ACQUIRE and RELEASE should run at a stretch (without any interruption). We say that they are atomic or indivisible.