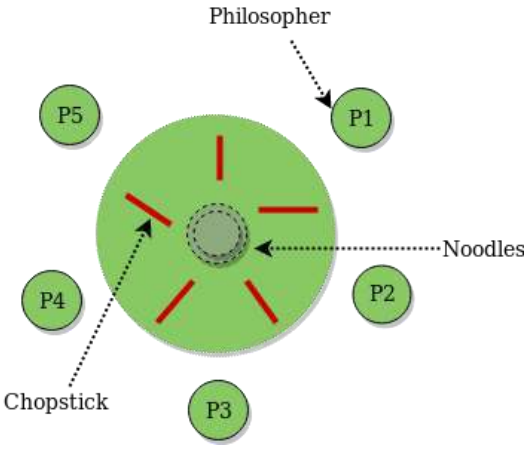
Dining Philosopher Problem Using Semaphores

Prerequisite – Process Synchronization, Semaphores, Dining-Philosophers Solution Using Monitors

The Dining Philosopher Problem – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Semaphore Solution to Dining Philosopher –

Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
  {  THINK;
     PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
     EAT;



     PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

There are three states of philosopher : **THINKING, HUNGRY and EATING**. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

Readers-Writers Problem

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**
Problem parameters:

One set of data is shared among a number of processes

- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer
Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for sempahore :
– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do {

   // writer requests for critical section

   wait(wrt);


   // performs the write


   // leaves the critical section

   signal(wrt);


} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:

- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
- It then, signals mutex as any other reader is allowed to enter while others are already reading.
- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

```
do {


  // Reader wants to enter the critical section
  wait(mutex);


  // The number of readers has now increased by 1
  readcnt++;


  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
  if (readcnt==1)
    wait(wrt);

  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);

  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;

  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);        // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

Thus, the semaphore '**wrt**' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

Source: https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/