PROCESS SYNCHRONIZATION

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages Concurrent access to shared data may result in data inconsistency, hence various mechanisms has to be employed to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Example: Producer – Consumer Problem / Bounded Buffer Problem

Producer : produces data item to buffer

Consumer : consumes data item from buffer

Let consider a shared buffer of size 5

Buffer :

- Initially Buffer is empty (consumer is not suppose to execute as buffer is empty).
- When producer produces a data , it will be added to buffer till buffer is full. If Buffer is full producer is not supposed to execute as empty space is not available.
- To keep track of all these we use a variable called Counter
- Shared variable is : Counter
- Counter is initialized to 0
- If Producer access counter, it increments counter value - counter++
- If Consumer access counter, it decrements counter value - counter—

- Let consider in buffer we have 3 items (i.e., 10,20,30)

30	20	10		

- Counter size = 5
- Counter value = 3 (3 data items are added)

✔ **<u>Case 1:</u>** if producer produce one more item i.e., 40 then counter value will be incremented, i.e., counter value= 4

Buffer :

40	30	20	10	

Then consumes one data item from buffer, then counter value will get decremented i.e., counter value = 3

Buffer :

40	30	20		

Synchronization will be there as they are executing one after the other.

✔ **<u>Case 2:</u>** vice versa (i.e., fist consumer executes and then producer- here also sunchronization will be there.)

✔ **<u>Case 3</u>** : If both Producer and consumer access counter concurrently then it leads to inconsistent data i.e., produer increments counter value by 1 and consumer decrements counter value by 1 concurrently which leads to inconsistent data.

- This problem lead to a definition called **<u>as race condition.</u>**

- **Race Condition** : Situation where several processes access and manipulates some data concurrently and the outcome of execution depends on the particular order in which access takes place.

1. **The Critical-Section Problem**
   - Consider a system consisting of *n* processes $\{P0, P1, ..., Pn-1\}$.
   - 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.
   - The important feature of the system is that, 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. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.
   - The general structure of a typical process *Pi* is shown in Fig a.
   - The entry section and exit section are enclosed in boxes to highlight these important segments of code.

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

Figure a: General structure of a typical process *Pi* .

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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this 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.

a. **Two Process solutions :**

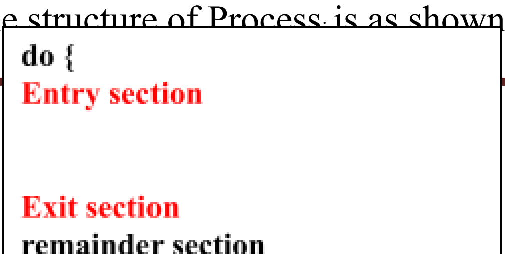
- It is applicable only for 2 processes at a time
- Processes are numbered as P0 & P1

Algorithm 1:

✔ Processes share a common variable called as Turn initialized to 0 or 1.

✔ If Turn == I, then Process$_i$ is allowed to execute in its critical section.

✔ The structure of Process$_i$ is as shown below



```
do {
Entry section


Exit section
remainder section
```

- ✔ This solution ensures that only one process at a time can be inits critical section.
- ✔ It doesnot satisfy the progress requirement.

<u>Algorithm 2 :</u>

- ✔ To overcome the problem in algorithm 1 , we can replace the variable turn with the following array

Boolean flag[2];

- ✔ The elements of the array are initialized to false
- ✔ If flag[i] is true, this value indicates that p_i is ready to enter the critical section.
- ✔ The structure of process p_i is

```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

- ✔ Process p_i first sets flag[i] to be true, signaling that it is ready toenter its critical section.
- ✔ Then p_i checks to verify that process p_j is not also ready to enter its critical section.
- ✔ If p_j were ready , then p_i would wait until p_j had indicated that it no longer neede to be in critical section i.e., until flag[j]= false
- ✔ At this point p_i would enter the critical section.
- ✔ On exiting the critical section p_i would set falg[i] to be false allowing other process to enter its critical section.
- ✔ In this solution mutual exclusion requirement is satisfied, but progress requirement is not met.

Algorithm 3: (Peterson's Solution)

- ✔ By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical section problem, where all three requirements are met.
- ✔ The process share two variables :

  Boolean flag[2];
  Int turn;
- ✔ Initially flag[0]=flag[1]=false
- ✔ The value of turn is immaterial (either 0 or 1)
- ✔ The variable turn indicates whose turn it is to enter its critical section. That is,
- ✔ if turn == i, then process *Pi* is allowed to execute in its critical section. The
- ✔ flag array is used to indicate if a process is ready to enter its critical section.
- ✔ For example, if flag[i] is true, this value indicates that *Pi* is ready to enter
- ✔ its critical section.
- ✔ Structure of process $\mathbf{p_i}$ **is:**

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

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

- ✔ To enter the critical section, process *Pi* first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- ✔ If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- ✔ Only one of these assignments will last; the other will occur but will be overwritten immediately.
- ✔ The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
- ✔ We now prove that this solution is correct.
- ✔ We need to show that:
  - o Mutual exclusion is preserved.
  - o The progress requirement is satisfied.
  - o The bounded-waiting requirement is met.

To prove property 1,

- ✔ we note that each *Pi* enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.
- ✔ These two observations imply that *P*0 and *P*1 could not have successfully executed their while statements at about the same time, since the —say, *Pj*—must have successfully executed the while statement, whereas *Pi* had to execute at least one additional statement ("turn == j").
- ✔ However, at that time, flag[j] == true and turn == j, and this condition will persist as long as *Pj* is in its critical section; as a result, mutual exclusion is preserved.

✔ **To prove properties 2 and 3**, we note that a process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible.

✔ If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section. If *Pj* has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then *Pi* will enter the critical section.

✔ If turn == j, then *Pj* will enter the critical section. However,once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section.

✔ If *Pj* resets flag[j] to true, it must also set turn to i.

✔ Thus, since *Pi* does not change the value of the variable turn while executing the while statement, *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

SEMAPHORES :

- A **semaphore** S is an integer variable that is accessed only through two standard atomic operations**: wait() and signal().**
- The wait() operation was originally termed P , signal() was originally called V
- The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // busy wait
S--;
}
```

- The definition of signal() is as follows:

```
signal(S) {
S++;
    }
```

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- Operating systems often distinguish between counting and binary semaphores.
- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
- Counting semaphores can be used to control access to a given resource Consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait () operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

```
do {

    wait(mutex);

        critical section

    signal(mutex);

        remainder section

} while (1);
```

Deadlocks & Starvation:

✔ The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

✔ To illustrate this, consider a system consisting of two processes, *P*0 and *P*1,each accessing two semaphores, S and Q, set to the value 1:

```
 P0                  P1
wait(S);            wait(Q);
wait(Q);             wait(S);
  .                   .
  .                   .
  .                   .
signal(S);          signal(Q);
signal(Q);           signal(S);
```

✔ Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked.

Solution : Priority Inversion

<u>Classic Problems of Synchronization</u>

1. Bounded – Buffer Problem :

- In our problem, the producer and consumer processes share the following data structures:

  int n;
  semaphore mutex = 1;
  semaphore empty = n;
  semaphore full = 0
- We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- Structure of Producer process is :

```
do {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```

- Structure of Consumer Process is:

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

2. Readers – Writers Problem :

- ✔ Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- ✔ We distinguish between these two types of processes by referring to the former as ***readers*** and to the latter as ***writers***.
- ✔ If two readers access the shared data simultaneously, no adverse effects will result.
- ✔ If a writer and some other process (either a reader or a writer) access the database simultaneously, it affect the result.

- ✔ To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- ✔ This synchronization problem is referred to as the **readers–writers problem**.
- ✔ Readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object or no reader should wait for other readers to finish simply because a writer is waiting.
- ✔ The ***second*** readers –writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- ✔ A solution to either problem may result in starvation.
- ✔ In the first case, writers may starve; in the second case, readers may starve.
- ✔ For this reason, other variants of the problem have been proposed.
- ✔ Next, we present a solution to the first readers–writers problem.
- ✔ In the solution to the first readers–writers problem, the reader processes share the following data structures:

  semaphore rw mutex = 1;
  semaphore mutex = 1;
  int read count = 0;
- ✔ The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0.
- ✔ The semaphore rw mutex is common to both reader and writer processes.
- ✔ The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.

✔ The read count variable keeps track of how many processes are currently reading the object.

✔ The semaphore rw mutex functions as a mutual exclusion semaphore for the writers.

✔ It is also used by the first or last reader that enters or exits the critical section.

✔ It is not used by readers who enter or exit while other readers are in their critical sections.

✔ **The code for a writer process is:**

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

✔ **The code for a reader process is:**

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

- ✔ Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on rw mutex, and $n - 1$ readers are queued on mutex .
- ✔ The readers–writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems.
- ✔ Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access.
- ✔ When a process wishes only to read shared data, it requests the reader–writer lock in read mode.
- ✔ A process wishing to modify the shared data must request the lock in write mode.
- ✔ Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

3. The Dining-Philosophers Problem

- ✔ Consider five philosophers who spend their lives thinking and eating.
- ✔ The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Fig a).
- ✔ When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- ✔ A philosopher may pick up only one chopstick at a time.
- ✔ Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour.

- ✔ When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.
- ✔ When she is finished eating, she puts down both chopsticks and starts thinking again.
- ✔ It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. One simple solution is to represent each chopstick with a semaphore.
- ✔ A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- ✔ She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- ✔ Thus, the shared data are

  semaphore chopstick[5];

  where all the elements of chopstick are initialized to 1.
- ✔ The structure of philosopher *i* is shown below
- ✔ The structure of philosopher *I* :

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
. . .
/* eat for awhile */
. . .
signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
. . .
/* think for awhile */
. . .
} while (true);
```

Figure : The structure of philosopher *i*.

- ✔ Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- ✔ Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0.
- ✔ When each philosopher tries to grab her right chopstick, she will be delayed forever.
- ✔ Several possible remedies to the deadlock problem are replaced by:
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
  - Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.