

Operating System

Unit – 3

(Part-A)

Process Synchronization



Mayank Mishra

School of Electronics Engineering

KIIT-Deemed to be University

Introduction

- If multiple processes runs in our system, then there could be two modes: *serial mode and parallel mode*.
- In **serial mode**, processes will be executed one by one. Here one running process won't make any affect on other.
- Considering the scenario of parallel execution (**parallel mode**), our system these days uses multiprogrammed environment. The use time sharing environment. Many processes can run simultaneously at a time.

➤ Processes can be categorized into two types: *Cooperative Process and Independent process.*

➤ A cooperating process is one that can affect or be affected by other processes executing in the system. *Why it affects the other?*

- ❑ The reason is, they share something common.

- ❑ The common thing can be:

 - ✓ Variable

 - ✓ Memory/Buffer

 - ✓ Code

 - ✓ Resources (e.g. CPU, Printer, Scanner, etc.)

➤ **Independent Process:** The execution of one process does not affect the execution of other processes.

Example:

- Let us consider two processes P1 and P2 running on the same system (Let us assume P1 gets CPU first).

Int Shared = 5;

P1

X = Shared;

X++;

sleep (1);

Shared =X;

P2

Y = Shared;

Y--;

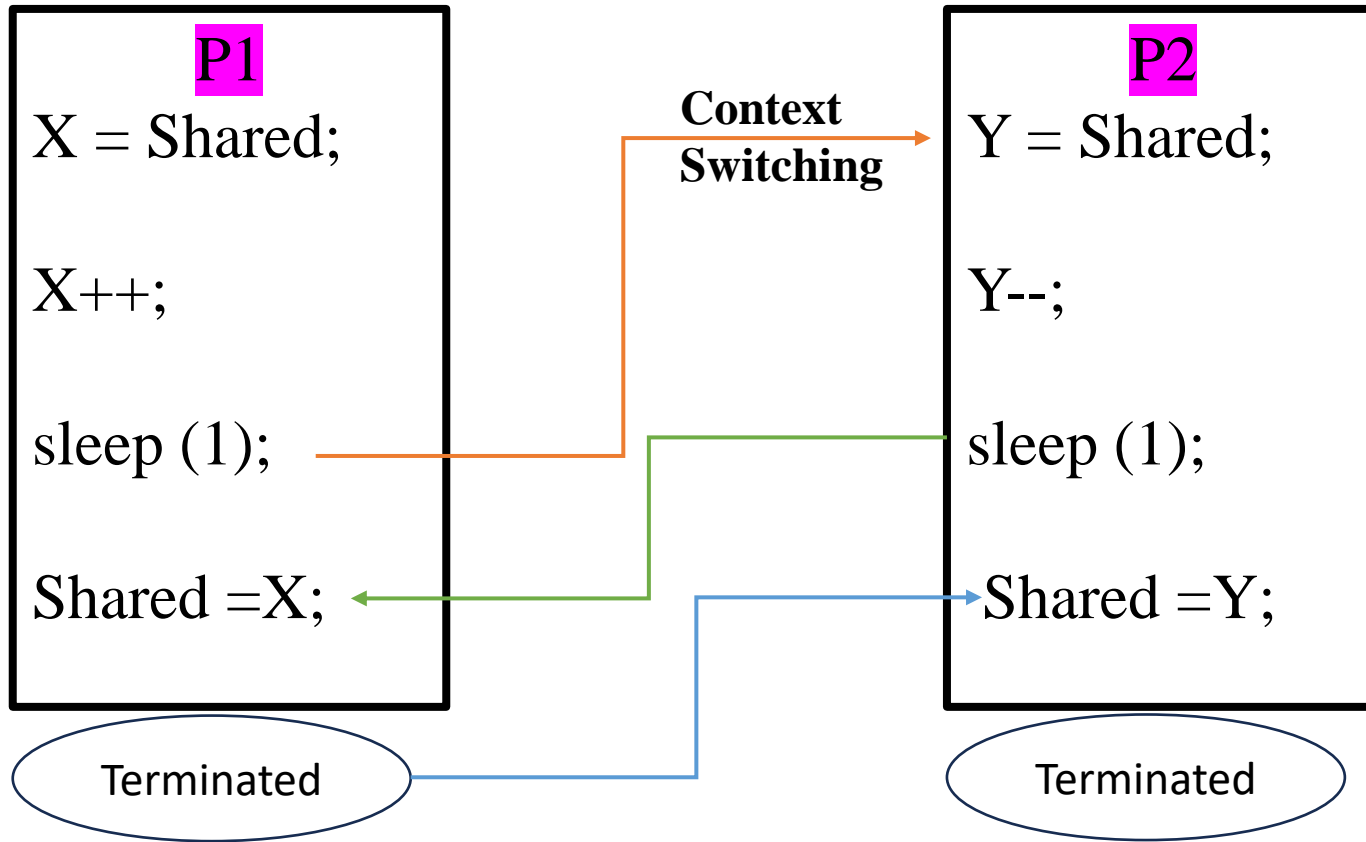
sleep (1);

Shared =Y;

Example:

- Let us consider two processes P1 and P2 running on the same system (Let us assume P1 gets CPU first).

Int Shared = 5;

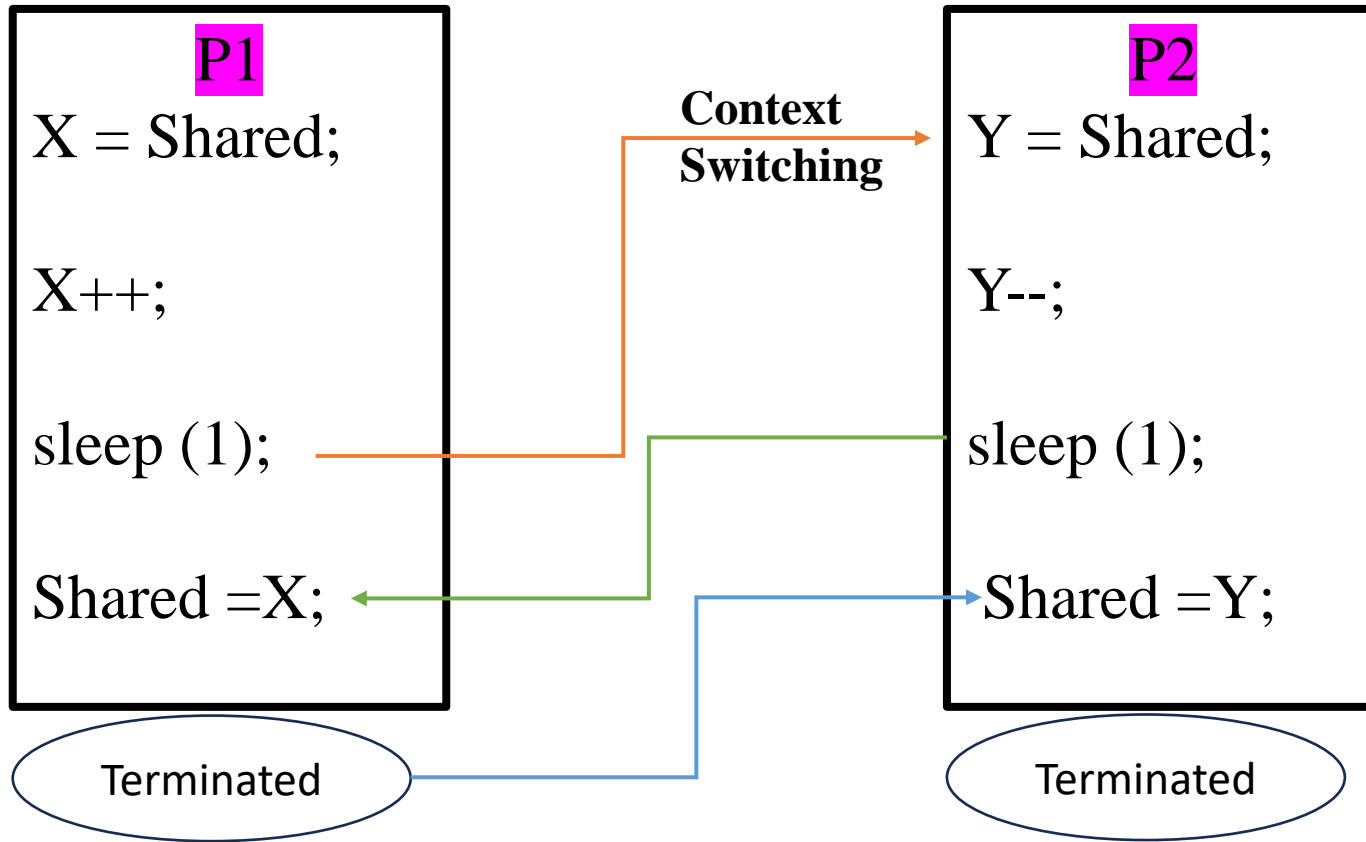


The value of Shared variable is ?

Example:

- Let us consider two processes P1 and P2 running on the same system (Let us assume P1 gets CPU first).

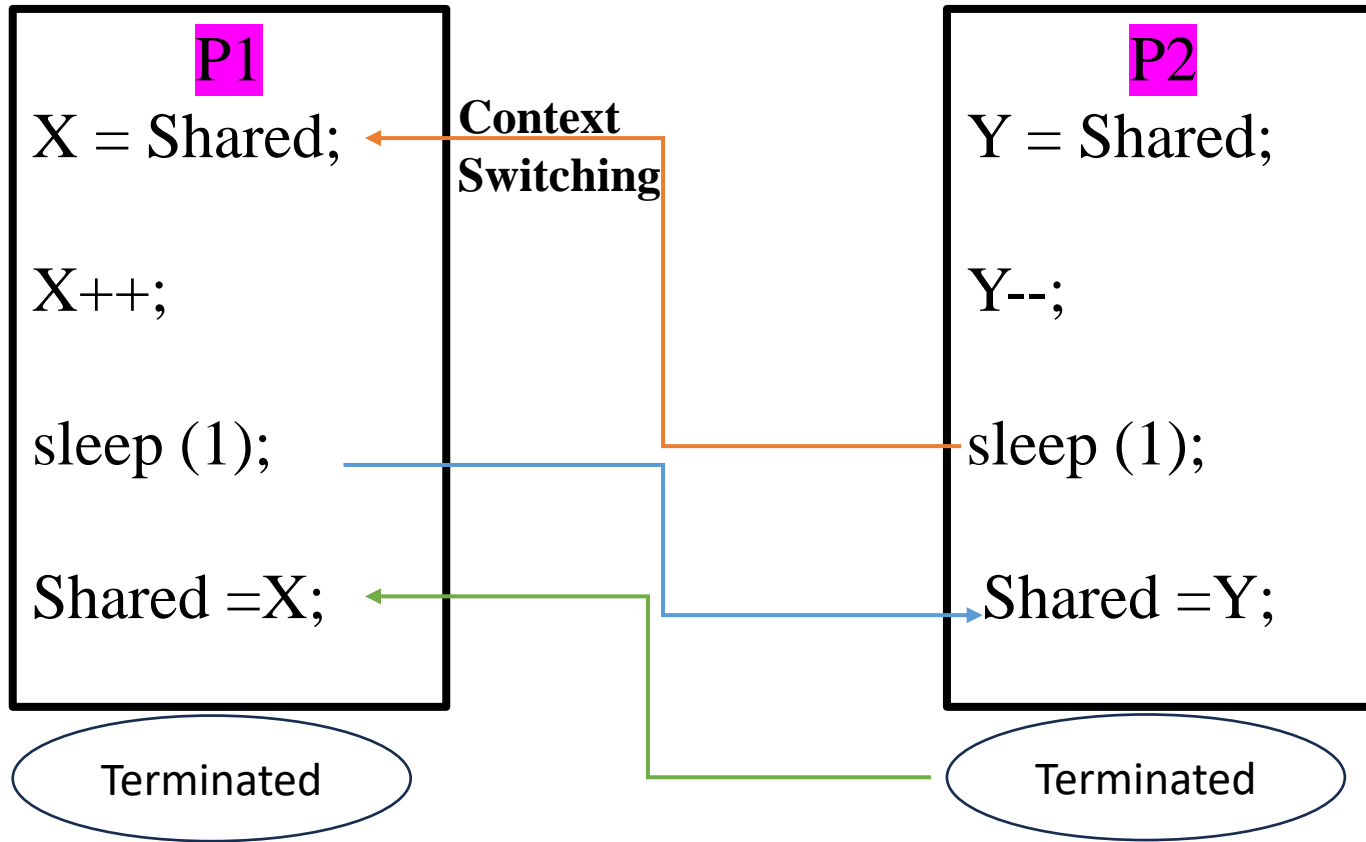
Int Shared = 5;



The value of Shared variable is 4

- Let us consider two processes P1 and P2 running on the same system (Let us assume P2 gets CPU first).

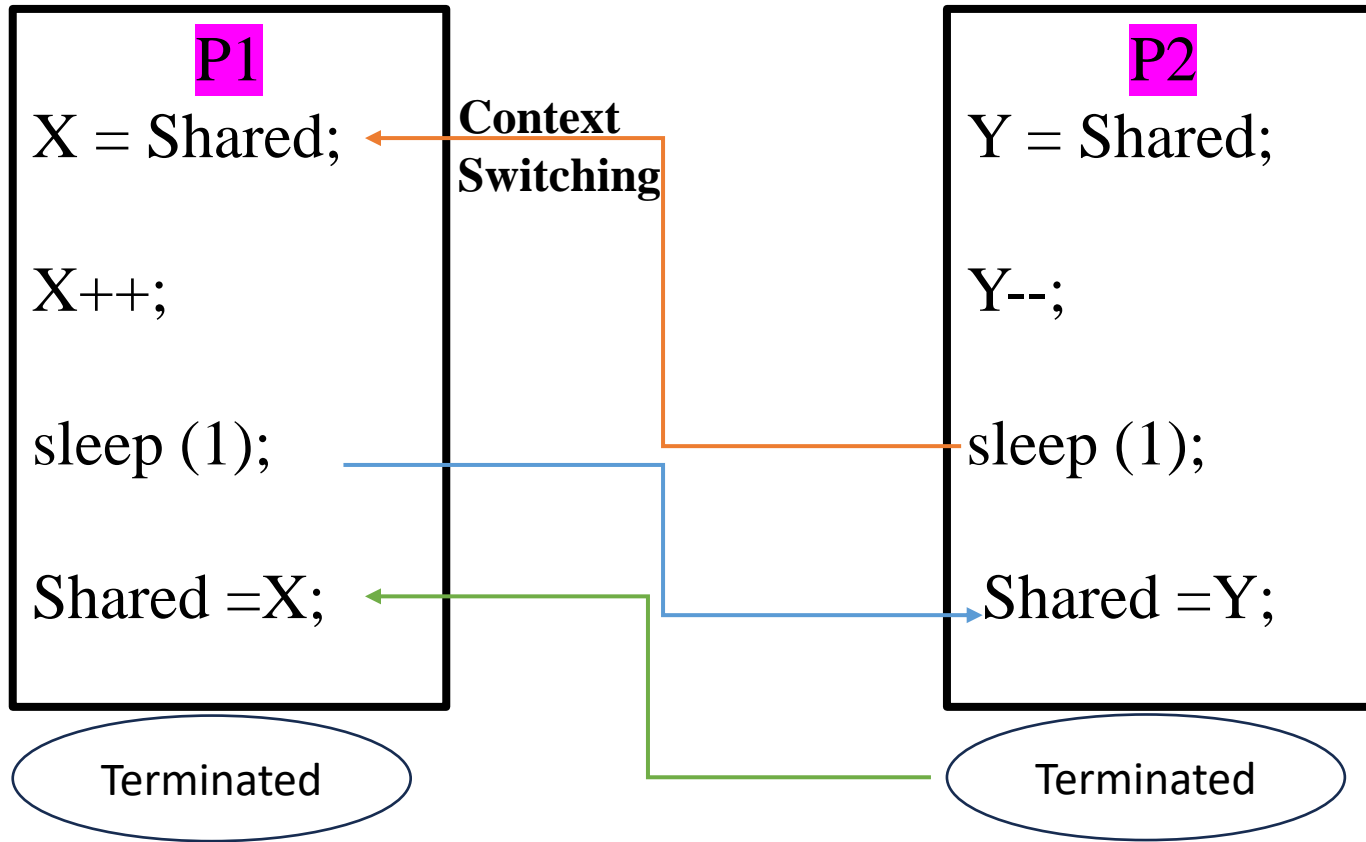
Int Shared = 5;



The value of Shared variable is ?

- Let us consider two processes P1 and P2 running on the same system (Let us assume P2 gets CPU first).

Int Shared = 5;



The value of Shared variable is 6.

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable **Shared** concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- When more than one process is either running the same code or modifying the same memory or any shared data, there is a risk that the result or value of the shared data may be incorrect because all processes try to access and modify this shared resource. Thus, all the processes race to say that my result is correct. This condition is called the **race condition**. Since many processes use the same data, the results of the processes may depend on the order of their execution.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable **Shared**. To make such a guarantee, we require that the processes be synchronized in some way.

- **Process Synchronization** is a technique which is used to coordinate the process that use shared Data.
- **Process Synchronization** is used in a computer system to ensure that multiple processes or threads can run concurrently without interfering with each other.
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

Lack of Synchronization in Inter Process Communication Environment leads to following problems:

- ❑ **Inconsistency:** When two or more processes access shared data at the same time without proper synchronization. This can lead to conflicting changes, where one process's update is overwritten by another, causing the data to become unreliable and incorrect.
- ❑ **Loss of Data:** Loss of data occurs when multiple processes try to write or modify the same shared resource without coordination. If one process overwrites the data before another process finishes, important information can be lost, leading to incomplete or corrupted data.
- ❑ **Deadlock:** Lack of Synchronization leads to Deadlock which means that two or more processes get stuck, each waiting for the other to release a resource. Because none of the processes can continue, the system becomes unresponsive and none of the processes can complete their tasks.

The Critical-Section Problem

- Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$.
- Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process.
- 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 synchronize their activity so as to cooperatively share data.

The Critical-Section Problem (Contd.)

- The general structure of a typical process is shown in Figure. 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**. (As shown in Fig, the entry section and exit section are enclosed in boxes to highlight these important segments of code.)

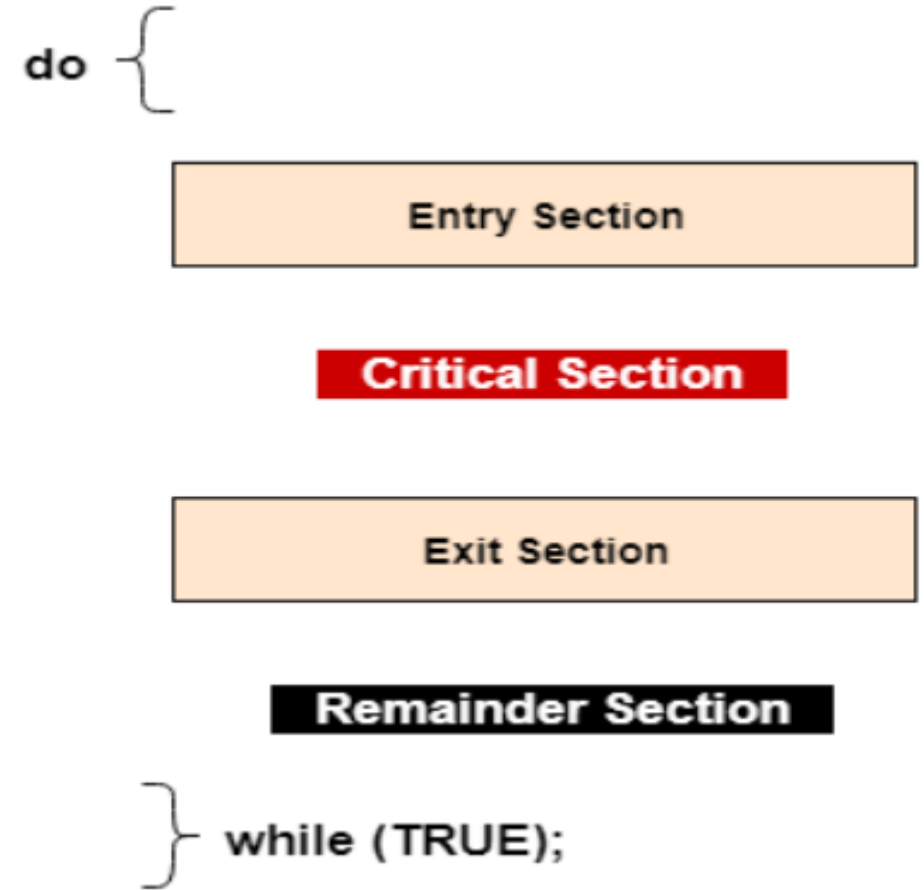


Fig. General structure of a typical process

A solution to the critical-section problem must satisfy the following three requirements:

- **Mutual exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress:** Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
- **Bounded Waiting:** Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a 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 requirements.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- It's simple and effective for solving synchronization issues in two-process scenarios.

Peterson's Solution (Contd.)

- The processes are numbered P0 and P1. For convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals $1 - i$. Let's call the process P_i and P_j .
- Peterson's Solution requires the two data items to be shared between the two processes.

```
int turn;  
boolean flag[2];
```

int turn

→ Indicates whose turn it is to enter its critical section.

boolean flag [2]

→ Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

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

    flag[i] = false;

    //remainder section
}while(true);
```

Structure of process P_j in Peterson's solution

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

    flag[j] = false;

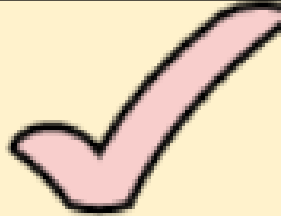


    //remainder section
}while(true);
```

Peterson's Solution (Contd.)

- To prove this solution is correct, we need to show that:
 - ❑ Mutual exclusion is preserved.
 - ❑ The progress requirement is satisfied.
 - ❑ The bounded-waiting requirement is met.

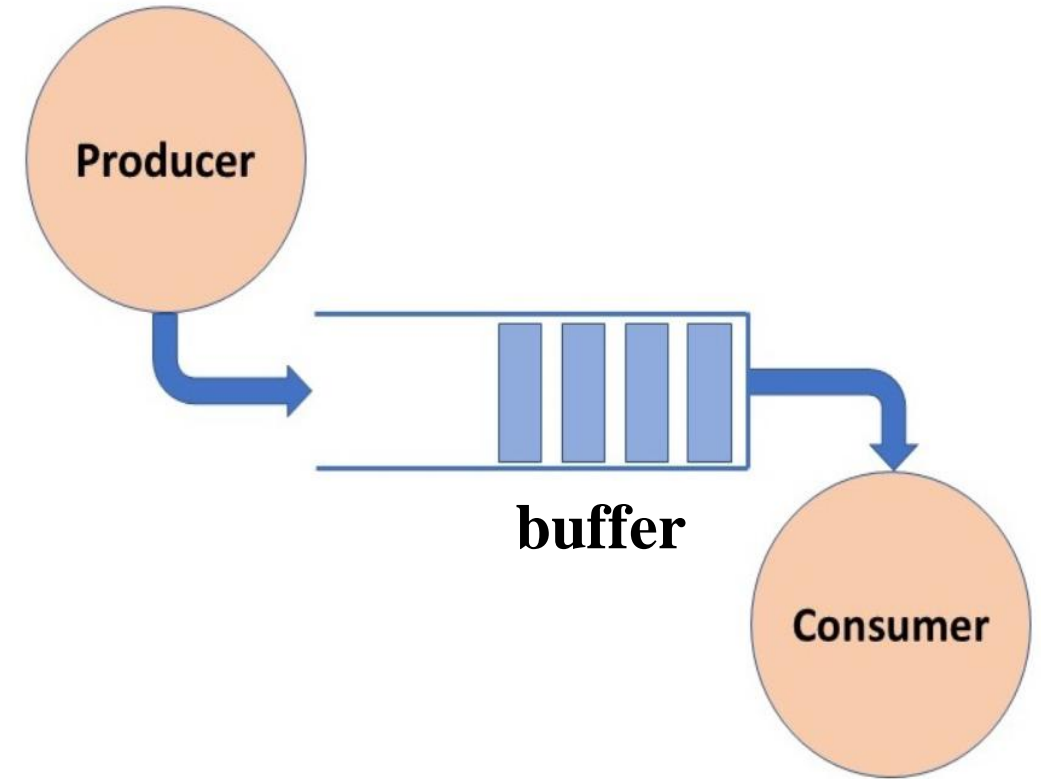
To prove property 1, we note that each P_i enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes—say, P_j —must have successfully executed the while statement, whereas P_i 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 P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i 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 P_j is not ready to enter the critical section, then `flag[j] == false`, and P_i can enter its critical section. If P_j 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 P_i will enter the critical section. If `turn == j`, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since P_i does not change the value of the variable `turn` while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Mutual Exclusion	
Progress	
Bounded Waiting	

Producer-Consumer Problem

- The **Producer-Consumer** problem is a classic synchronization problem in operating systems.
- The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process. The **Producer** process creates an item and adds it to the shared buffer. The **Consumer** process takes items out of the shared buffer and “consumes” them.



Producer-Consumer Problem

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (count == BUFFER_SIZE)  
        ; /* do nothing */  
  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

Producer Process

```
while (true) {  
    while (count == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
  
    /* consume the item in next_consumed */  
}
```

Consumer Process

- **Producer** process produces an item, then the corresponding item will be placed in the **buffer** as well as it increments the **count** value by 1. **Consumer** process consumes an item from **buffer** and it decrements the **count** variable by 1.

- **in** : next empty buffer (used in producer process), **out**: first filled buffer (used in consumer process)

- Although the producer and consumer routines shown are correct separately, they may not function correctly when executed concurrently.
- As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements “count++” and “count--”.
- Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is $\text{count} == 5$, which is generated correctly if the producer and consumer execute separately.

- We can show that the value of count may be incorrect as follows. Note that the statement “count++” may be implemented in machine language (on a typical machine) as follows:

```
register1 = count  
register1 = register1 + 1  
count = register1
```

where register₁ is one of the local CPU registers.

- Similarly, the statement “count--” is implemented as follows:

```
register2 = count  
register2 = register2 - 1  
count = register2
```

where again register₂ is one of the local CPU registers.

- The concurrent execution of “**count++**” and “**count--**” is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is the following:

T_0 :	<i>producer</i>	execute	$register_1 = count$	$\{register_1 = 5\}$
T_1 :	<i>producer</i>	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2 :	<i>consumer</i>	execute	$register_2 = count$	$\{register_2 = 5\}$
T_3 :	<i>consumer</i>	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4 :	<i>producer</i>	execute	$count = register_1$	$\{count = 6\}$
T_5 :	<i>consumer</i>	execute	$count = register_2$	$\{count = 4\}$

- Notice that we have arrived at the incorrect state “ $count == 4$ ”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5 , we would arrive at the incorrect state “ $count == 6$ ”.

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable **Shared** concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- When more than one process is either running the same code or modifying the same memory or any shared data, there is a risk that the result or value of the shared data may be incorrect because all processes try to access and modify this shared resource. Thus, all the processes race to say that my result is correct. This condition is called the **race condition**. Since many processes use the same data, the results of the processes may depend on the order of their execution.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable **Shared**. To make such a guarantee, we require that the processes be synchronized in some way.

References

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Operating System Concepts,” Eleventh Edition (Wiley).
2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition (Pearson Publications), 2014.
3. <https://www.geeksforgeeks.org/>
4. <https://www.javatpoint.com/>
5. <https://www.tutorialspoint.com/>
6. <https://www.nesoacademy.org/>
7. <https://www.baeldung.com/>
8. <https://www.educative.io/>