OPERATING SYSTEM: CSET209

CONTENT

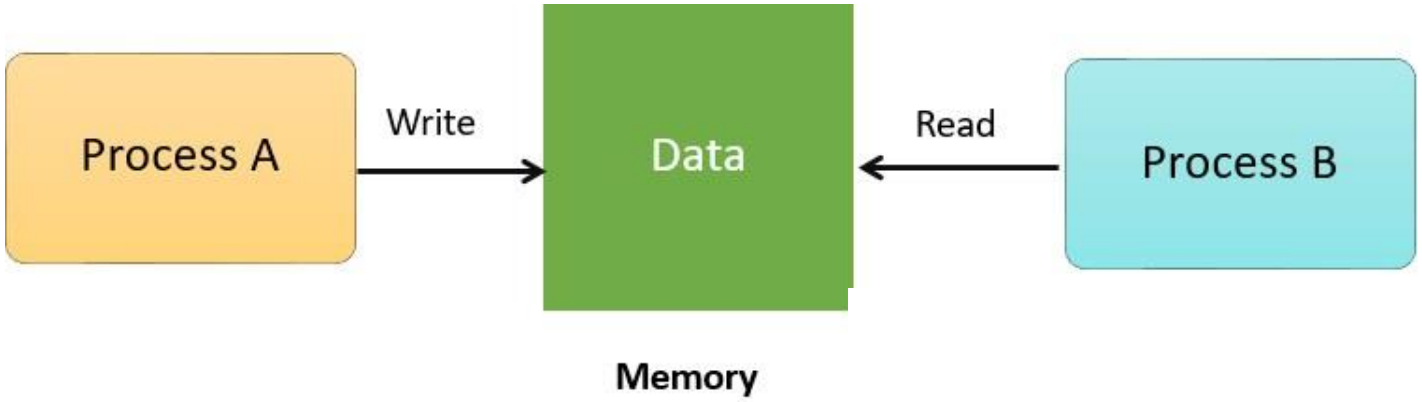
- Process Synchronization
- Critical section problem (CSP)
- Synchronization constructs

WHAT IS PROCESS SYNCHRONIZATION ?

- When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.
- On the basis of synchronization, processes are categorized as one of the following two types:
  - **Independent Process :** Execution of one process does not affects the execution of other processes.
  - **Cooperative Process :** Execution of one process affects the execution of other processes.

  - **Note:** Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

EXAMPLE:

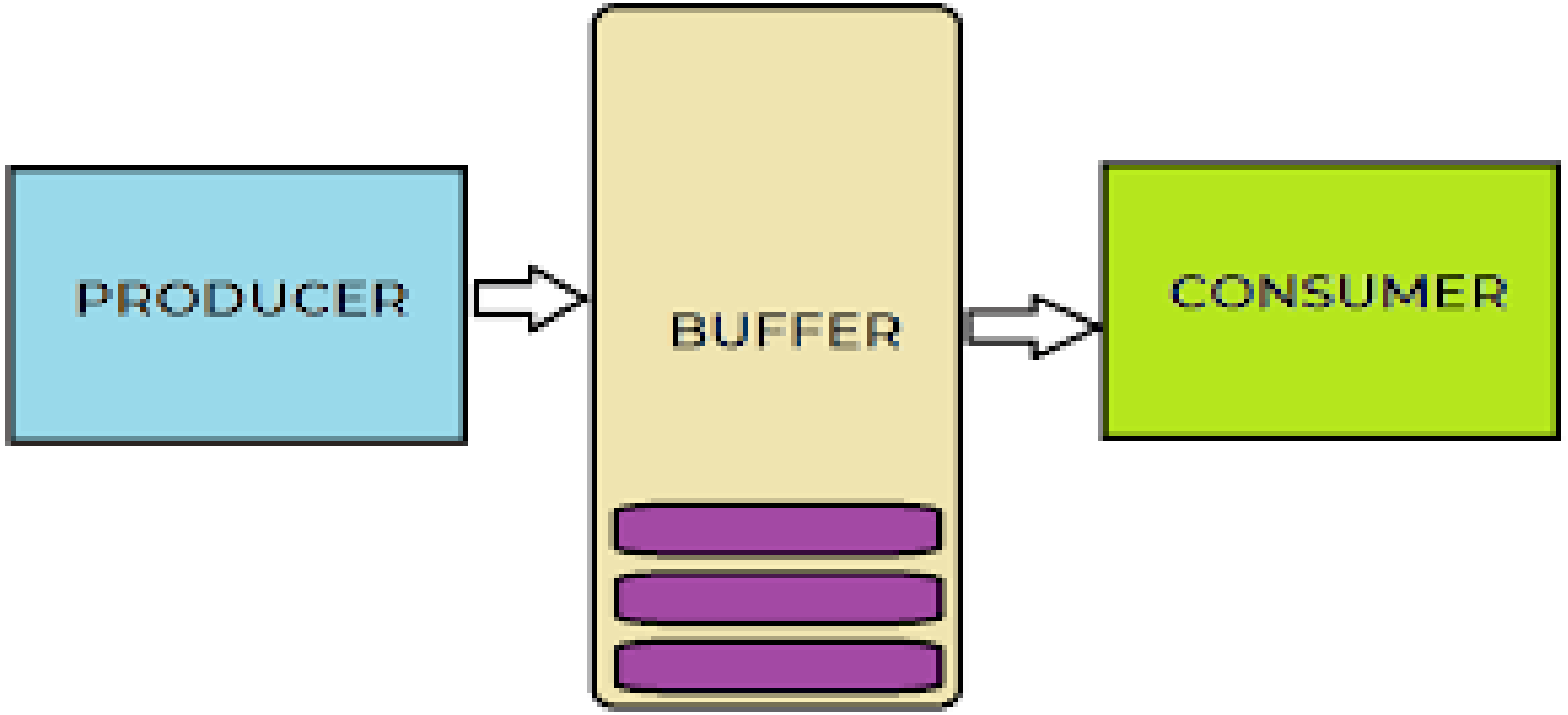


- A process A tries changing data in a particular memory location. At the same time another process B tries reading data from the same memory location. Thus, there is a high probability that the data being read by the second process is incorrect.

Need of Synchronization-

- When multiple processes execute concurrently sharing some system resources.
- To avoid the inconsistent results.

PRODUCER –CONSUMER PROBLEM TO SHOW RACE CONDITION



RACE CONDITION

Producer Process:

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

Consumer Process:

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

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

RACE CONDITION

- **A race condition is a condition when there are many processes and every process shares the data with each other and accessing the data concurrently, and the output of execution depends on a particular sequence in which they share the data and access.**
- A race condition is a situation that may occur inside a **critical section**.
- Race conditions in critical sections can be avoided if the **critical section** is treated as an atomic instruction.

RACE CONDITION

- To guard against the race condition, we need to ensure that only one process at a time can manipulate the counter variable (shared variables). To make such a guarantee, we require that the processes be synchronized in some way.

THE CRITICAL SECTION PROBLEM

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other process is to be allowed to execute in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

SECTIONS OF A PROGRAM IN OS

Critical Section: This allows a process to enter and modify the shared variable.

Entry Section: Each process must request permission to enter its critical section. The section of code implementing this request is called entry section.

Exit Section: This makes sure that the process is removed through critical section once it's done executing. And allows other process waiting in the Entry Section, to enter into their Critical Sections .

Remainder Section: Parts of the Code, not present in the above three sections are collectively called Remainder Section.

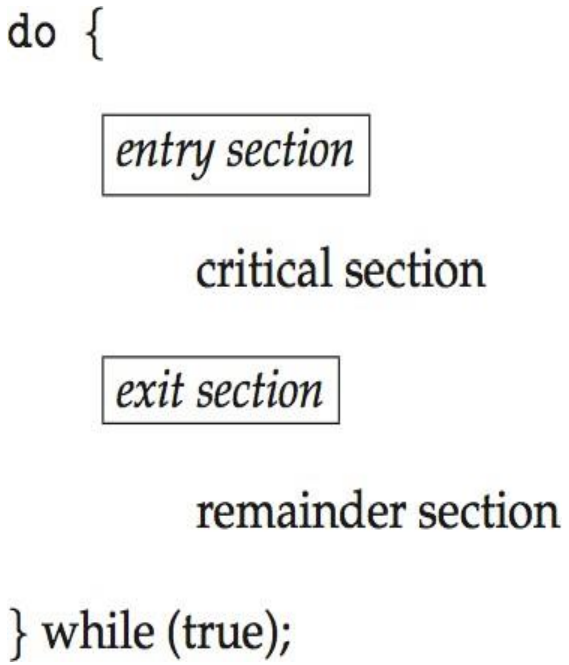
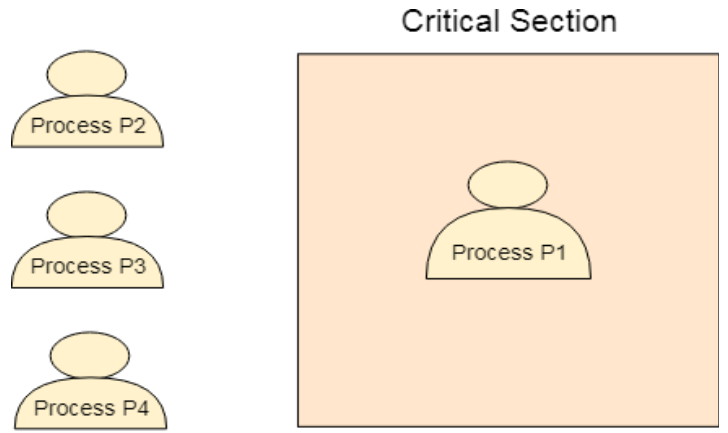
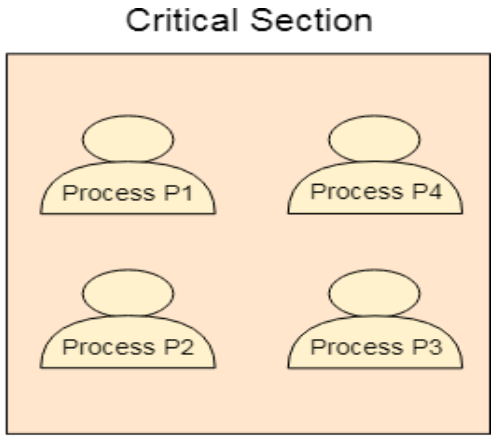


Figure: General structure of process P_i

REQUIREMENTS OF SOLUTION TO CRITICAL-SECTION PROBLEM

1. **Mutual Exclusion** - If process P_i 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 there exist some processes that wish to enter their critical section, must be permitted without any delay.
3. **Bounded Waiting/No starvation** - A bound must exist on the number of times that other processes are allowed to enter their critical sections while another is waiting. If bounded waiting is not satisfied then it is possible for starvation.





SOFTWARE BASED SOLUTION: A SIMPLE ATTEMPT

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

Analysis:

- Achieves **mutual exclusion.**
- **Busy waiting**: waste of CPU power, and
- Allows only **alternate execution** in critical section. Hence, **progress** condition is not satisfied.

process1 →process2 →process1 →process2

THANK YOU
?