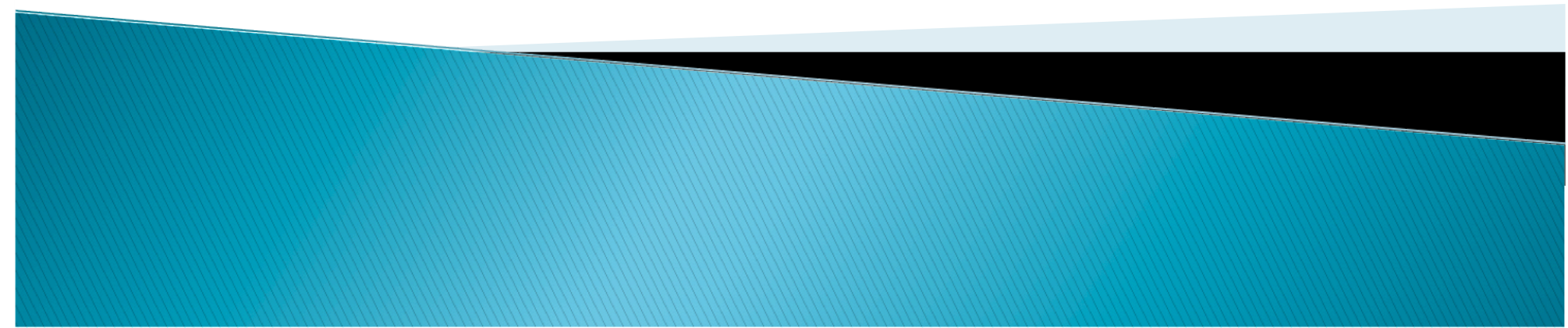


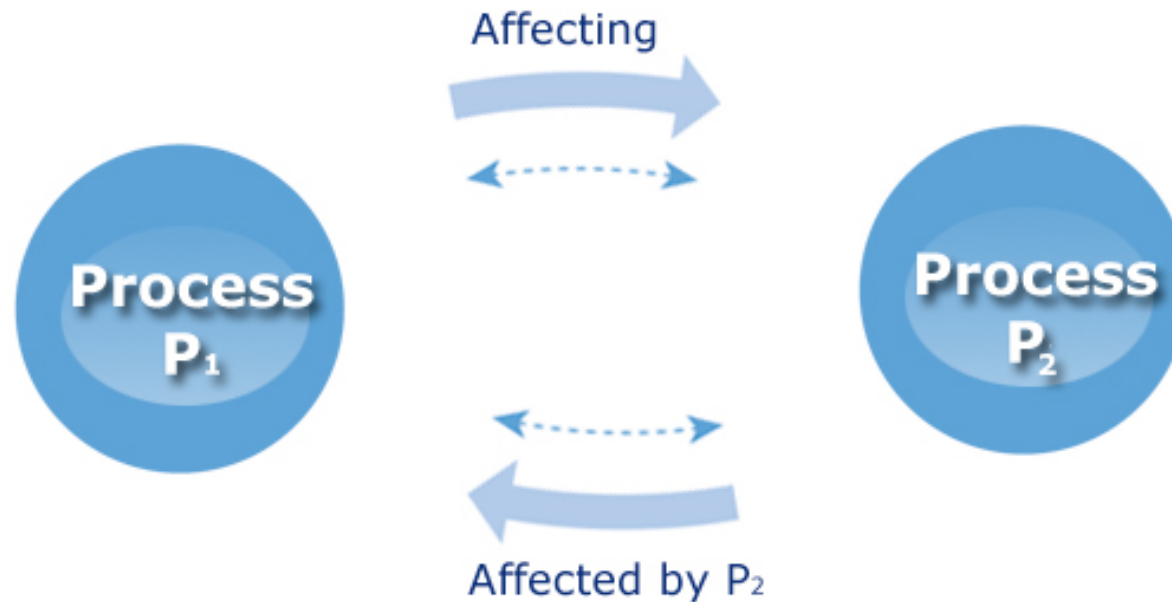
# Chapter 6: Process Synchronization

---

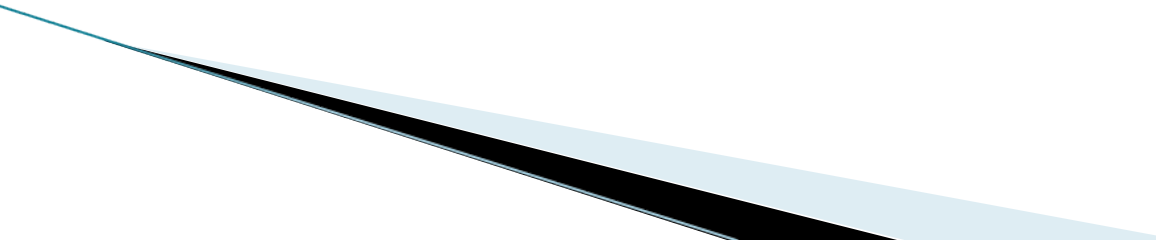


# Process Synchronization

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Concurrent access to shared data may result in data inconsistency.




# Background

- Previously we studied **producer-consumer problem** to understand cooperating processes.
  - Our solution allowed at most `BUFFER.SIZE - 1` items in the buffer at the same time (refer to old slides).
  - To remedy this deficiency we can add an integer variable counter, initialized to 0.
  - Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- 

# Background

## □ **producer-consumer problem**

### Shared Data:

- `#define BUFFER_SIZE 10`
  - `int buffer [BUFFER_SIZE] ;`
  - `int in = 0 ,`
  - `int out = 0 ;`
  - `int count=0;`
- 

## Background(cntd)

### Producer

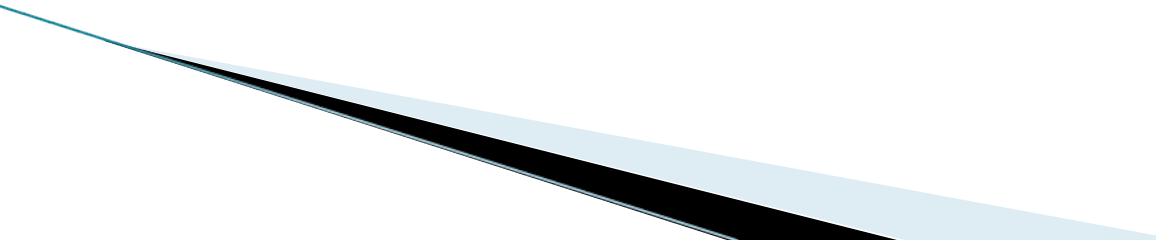
while (true)

```
/* produce an item and put in nextProduced  
while (count == BUFFER_SIZE)  
    ; // do nothing  
buffer [in] = nextProduced;  
in = (in + 1) % BUFFER_SIZE;  
count++;  
}
```

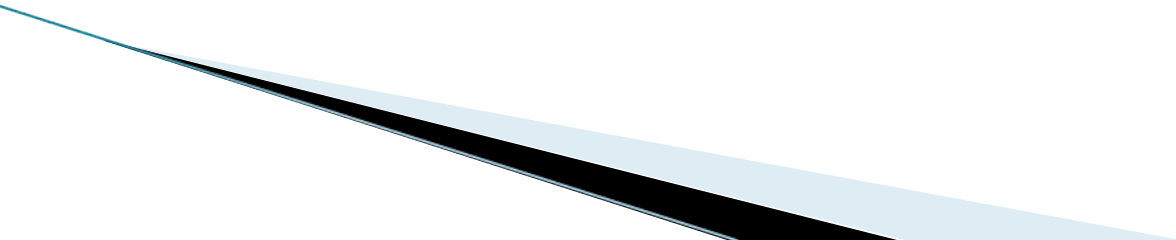
## Background(cntd)

### Consumer

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

A decorative graphic in the bottom-left corner consisting of a light blue triangle and a black triangle pointing towards the bottom-right.

# Problem

- Although both producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
  - For example the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter—" concurrently.
  - Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! .  
(correct result, though, is counter == 5)
- 

# Race Condition

- In machine language `count++` could be implemented as

```
T1 register1 = count  
T2 register1 = register1 + 1  
T3 count = register1
```

- `count--` could be implemented as

```
S1 register2 = count  
S2 register2 = register2 - 1  
S3 count = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
T1: producer execute register1 = count {register1 = 5}  
T2: producer execute register1 = register1 + 1 {register1 = 6}  
S1: consumer execute register2 = count {register2 = 5}  
S2: consumer execute register2 = register2 - 1 {register2 = 4}  
T3: producer execute count = register1 {count = 6}  
S3: consumer execute count = register2 {count = 4}
```

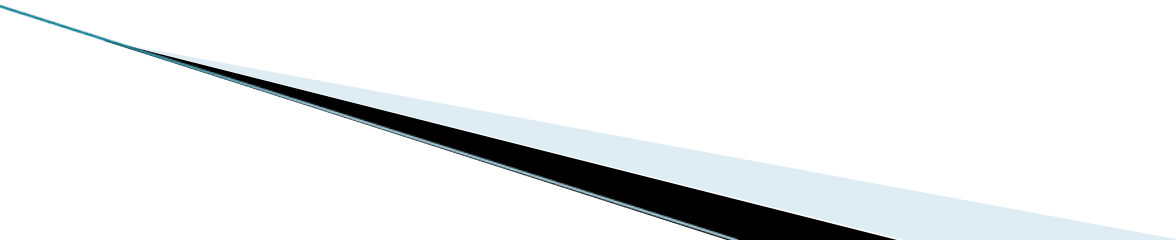
What will be the value of Count=?



# Problem (cntd)

- Notice that we have incorrect state "count == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- This happened because we allowed both processes to manipulate the variable count concurrently.
- When 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**.
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter, hence the need for processes synchronization.

# The Critical-Section Problem

- In concurrent programming, a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
  - 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 Critical-Section Problem(cntd)

```
do{  
    entry section  
    critical section  
    exitsection  
    remainder section  
} while (TRUE);
```

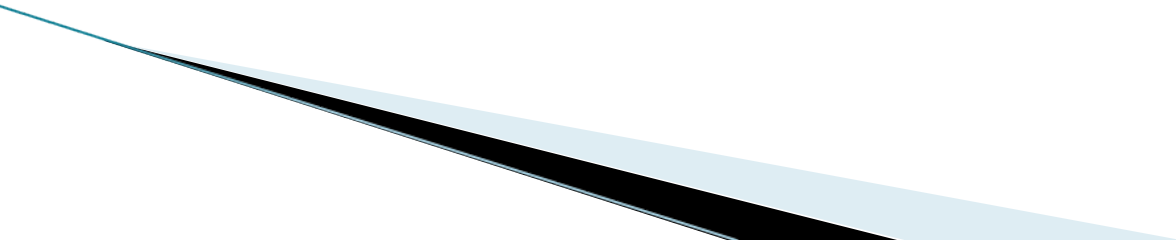
**Figure 6.1** General structure of a typical process  $P_i$ .

# Solution to Critical-Section Problem

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** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

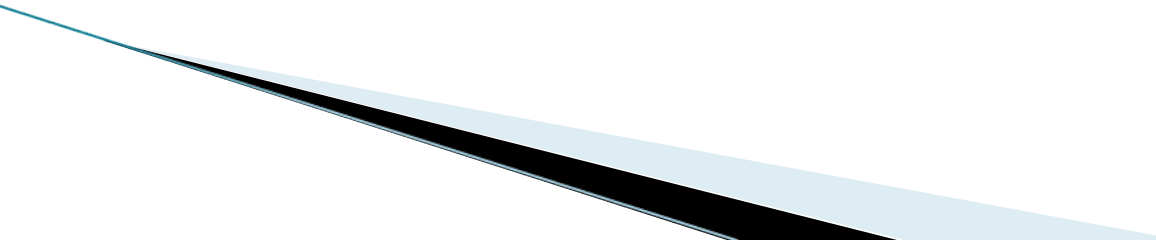


## Solution to Critical-Section Problem(cntd)

**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.



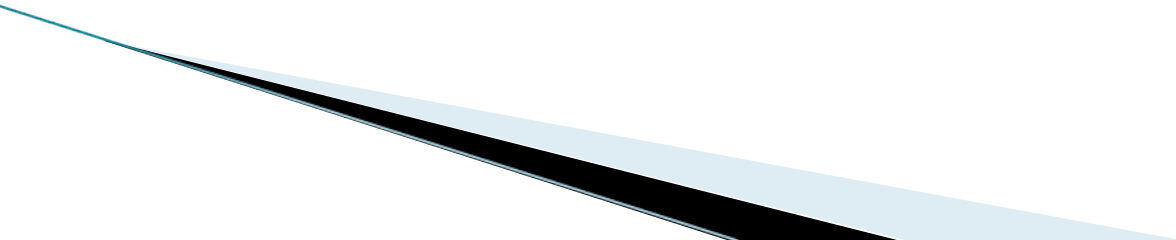
# Peterson's Solution to Critical Section Problem

- A software-based solution to the critical-section problem is known as Peterson's solution.
  - It is **restricted to two processes** that alternate execution between their CS and remainder sections.
  - Peterson's solution may not work correctly on modern architectures because of machine-language instructions (like example of variable count in producer-consumer in previous slides).
  - We present the solution because it provides a good algorithmic description of solving the critical-section problem and addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.
- 

# Peterson's Solution

- Processes are numbered  $P_0$  and  $P_1$ .
- Peterson's solution requires two data items to be shared between the two processes:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process  $P_i$  is ready!

# Peterson's Solution

- To enter the critical section, process  $P_i$  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.
- 



# Algorithm for Process $P_i$

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

CRITICAL SECTION

```
    flag[i] = FALSE;
```

REMAINDER SECTION

```
} while (TRUE);
```



# Algorithm for Process $P_0$ and $P_1$

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);  
  
        CRITICAL SECTION  
  
    flag[0] = FALSE;  
  
        REMAINDER SECTION  
  
} while (TRUE);
```

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);  
  
        CRITICAL SECTION  
  
    flag[1] = FALSE;  
  
        REMAINDER SECTION  
  
} while (TRUE);
```

# Peterson's Solution (cntd)

- We now prove that this solution is correct. We need to show that:
    - 1. Mutual exclusion is preserved?
    - 2. The progress requirement is satisfied?
    - 3. The bounded-waiting requirement is met?
- 