OPERATING SYSTEMS

Threads and Concurrency 05

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 2

UNIT 2: Threads and Concurrency

Introduction to Threads, types of threads, Multicore Programming, Multithreading Models, Thread creation, Thread Scheduling, PThreads and Windows Threads, Mutual Exclusion and Synchronization: software approaches, principles of concurrency, hardware support, Mutex Locks, Semaphores. Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts. Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads. Deadlocks: principles of deadlock, tools for detection and Prevention.

OPERATING SYSTEMS

Course Outline - Unit 2

13	4.1,4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3,5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, hardware support	6	
18	6.5,6.6	Mutex Locks, Semaphores	6	
19	6.7.1-6.7.3	Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts	6	
20	6.9	Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6	
21	Handouts	Demonstration of programming examples on process synchronization		
22	7.1-7.3	Deadlocks: principles of deadlock, Deadlock Characterization.	7	
23	7.4	Deadlock Prevention, Deadlock example	7	
24	7.6	Deadlock Detection	7	

Topics Outline

- Process Synchronisation
  - Background
  - The Producer Consumer Problem
  - The Producer Consumer Problem - Race Condition
  - The Critical-Section Problem

Background

- 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 both code and data** or be **allowed** to **share** data only through **files** or **messages**, former is usually achieved using **Threads** and the latter using **IPC**

- Concurrent access to shared data may result in data inconsistency

- There should be various **mechanisms** which ensure the **orderly** execution of **cooperating processes** that share a logical address space, so that data **consistency** is **maintained**

Background

- Processes can execute **concurrently** or in **parallel**
- As a result of **scheduling**, one process may only **partially** complete execution before **another** process is **scheduled**.
- **Concurrent** or **Parallel** execution can contribute to issues involving the integrity of data shared by several processes

The Producer Consumer Problem

- One of the most common task structures in concurrent systems is illustrated by the **producer-consumer problem**.

- In this problem, threads or processes are divided into two relative types:

  - A **producer thread / process** is responsible for performing an initial task that ends with **creating / producing** some result

  - A **consumer thread/process** that **takes / consumes** that initial result for some later task.

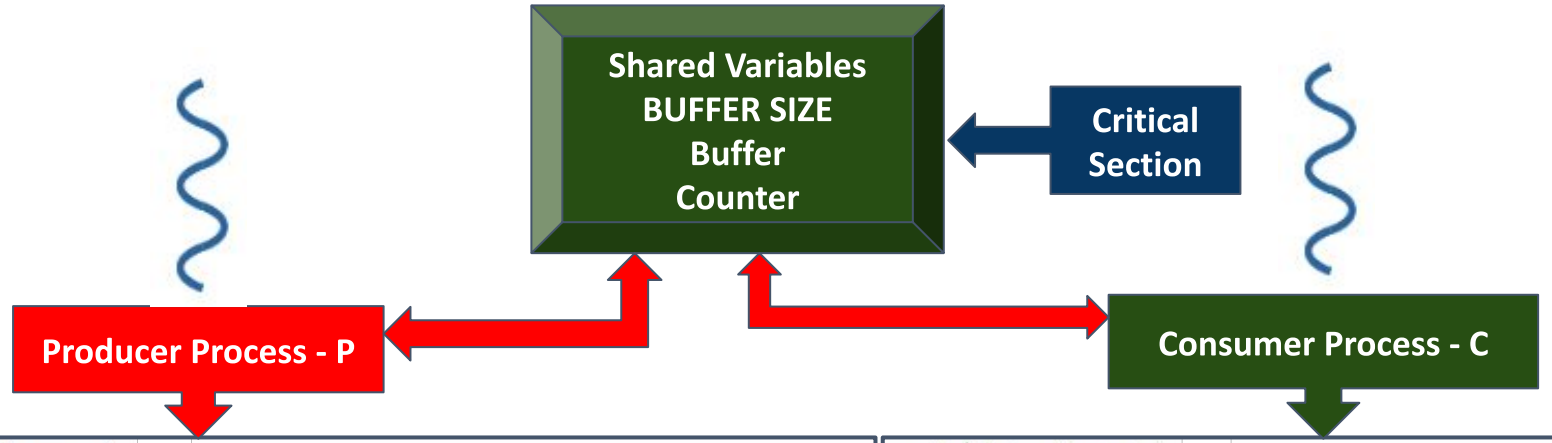
The Producer Consumer Problem

- Between the threads or processes, there is a **shared array** or **queue** that stores the results being passed.

- One **key feature** of this problem is that the **consumer** removes the data from the queue and "**consumes**" it by using it in some later purpose.

- There is **no** way for the consumer threads or processes to **repeatedly** access data in the queue.

The Producer Consumer Problem




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

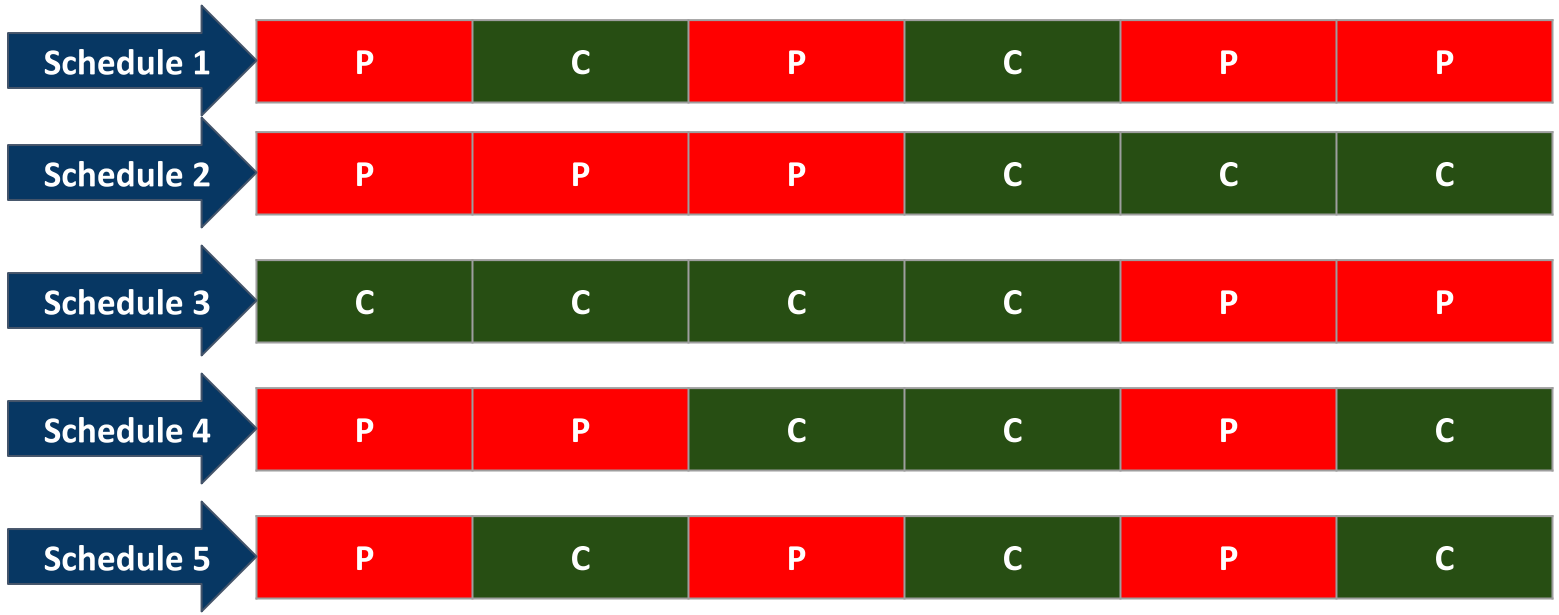
```
while (true)
{
    while (counter == 0)
          ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
     counter--;
    /* consume the item in next consumed */

}
```

The Producer Consumer Problem

- Both Producer and Consumer can run Concurrently either with True concurrency or Pseudo Concurrency or both as deemed fit

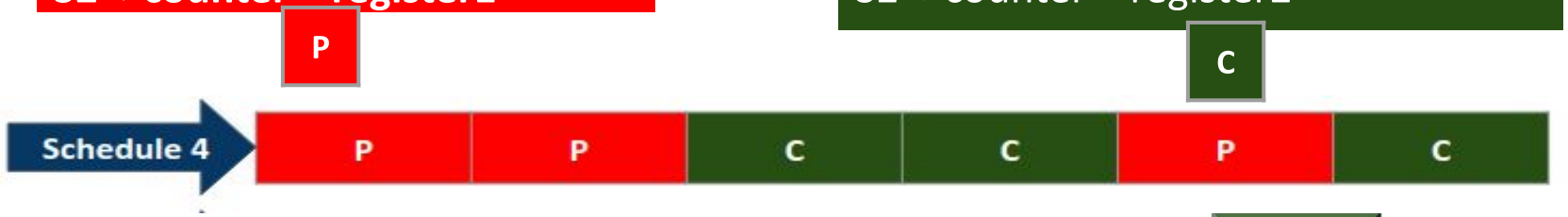
- The P and C order of execution is not guaranteed



The Producer Consumer Problem: The Race Condition

- **counter++**
could be implemented as
S0=>register1 = counter
S1=>register1 = register1 + 1
S2=>counter = register1

- counter--
could be implemented as
S0=>register2 = counter
S1=>register2 = register2 - 1
S2=>counter = register2



Consider this execution interleaving with "count = 5" initially:

P=>S0: Producer execute register1 = counter {register1 = 5}

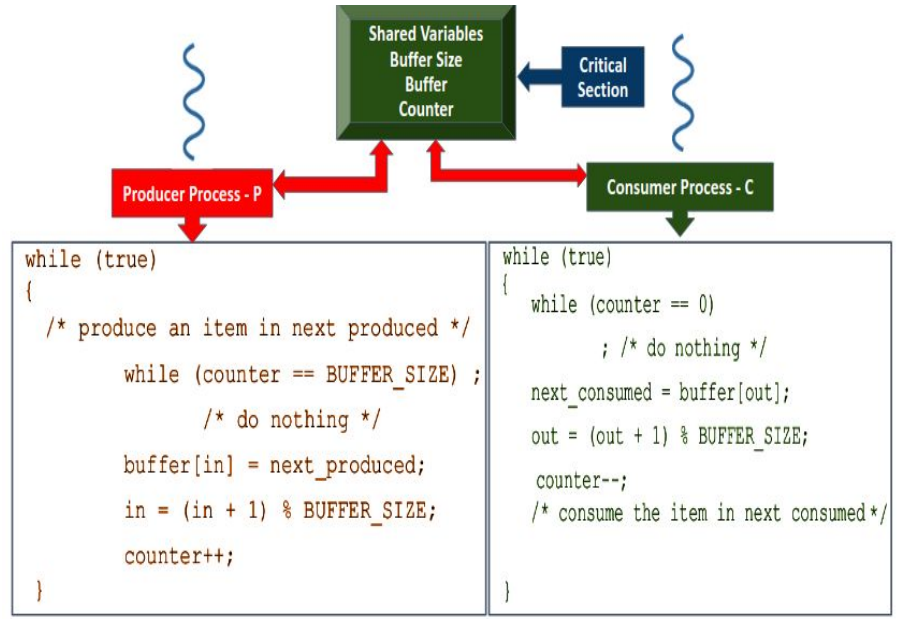
P=>S1: Producer execute register1 = register1 + 1 {register1 = 6}

C=>S0: Consumer execute register2 = counter {register2 = 5}

C=>S1: Consumer execute register2 = register2 – 1 {register2 = 4}

P=>S2: Producer execute counter = register1 {counter = 6 }

C=>S2: Consumer execute counter = register2 {counter = 4}



The Producer Consumer Problem: Non-Occurrence of Race Condition by luck

- **counter++**
could be implemented as
S0=>register1 = counter
S1=>register1 = register1 + 1
S2=>counter = register1

- counter--
could be implemented as
S0=>register2 = counter
S1=>register2 = register2 - 1
S2=>counter = register2

P

C

Consider this execution interleaving with "count = 5" initially:

P=>S0: Producer execute register1 = counter {register1 = 5}

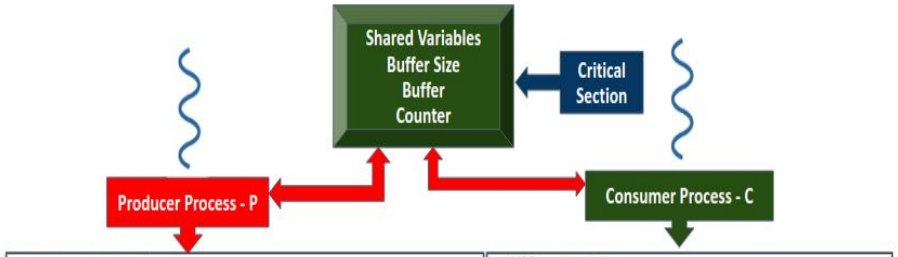
P=>S1: Producer execute register1 = register1 + 1 {register1 = 6}

P=>S2: Producer execute Counter = register1 {Counter = 6}

C=>S0: Consumer execute register2 = Counter {register2 = 6}

C=>S1: Consumer execute register2 = register2- 1 {register2 = 5}

C=>S2: Consumer execute counter = register2 {counter = 5}




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

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

}
```

The Critical Section Problem

- **Critical Section** is the **part** of a program which **tries** to **access shared resources**.
- That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.
- The critical section **cannot** be executed by **more** than **one** process at the **same time**
- Operating System faces the difficulties in allowing and disallowing the processes from entering the critical section.
- The **solution** to critical section problem is used to design a set of protocols which can ensure that the race condition among the processes will **never arise**.

Topics Uncovered in this Session

- Process Synchronisation
  - Background
  - The Producer Consumer Problem
  - The Producer Consumer Problem - Race Condition
  - The Critical-Section Problem

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com