OPERATING SYSTEMS

Threads and Concurrency 10

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

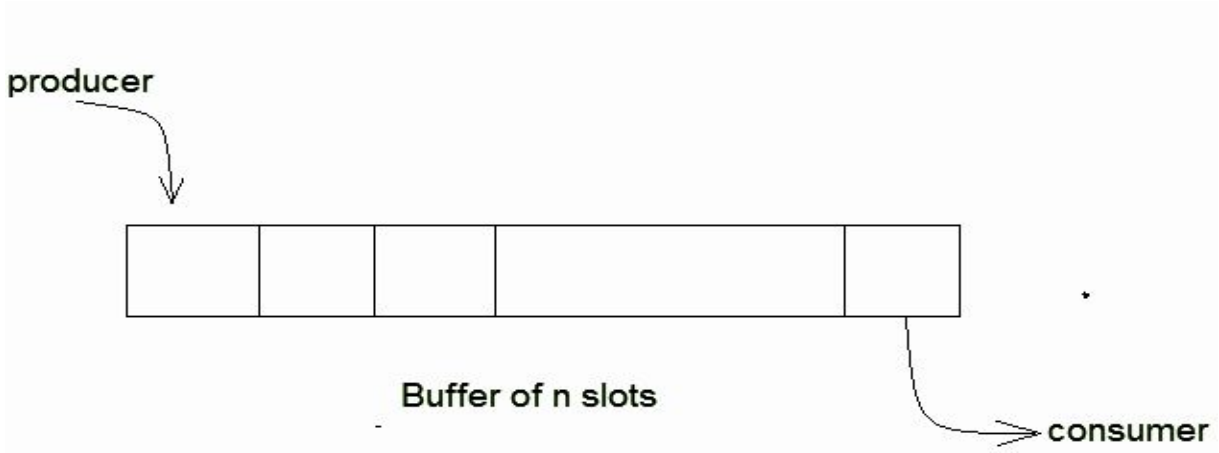
13	4.1,4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3,5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, 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

- Bounded Buffer Problem
- The Readers – Writers Problem
- The Dining Philosopher Problem

Classical Problems of Synchronization: Bounded Buffer Problem

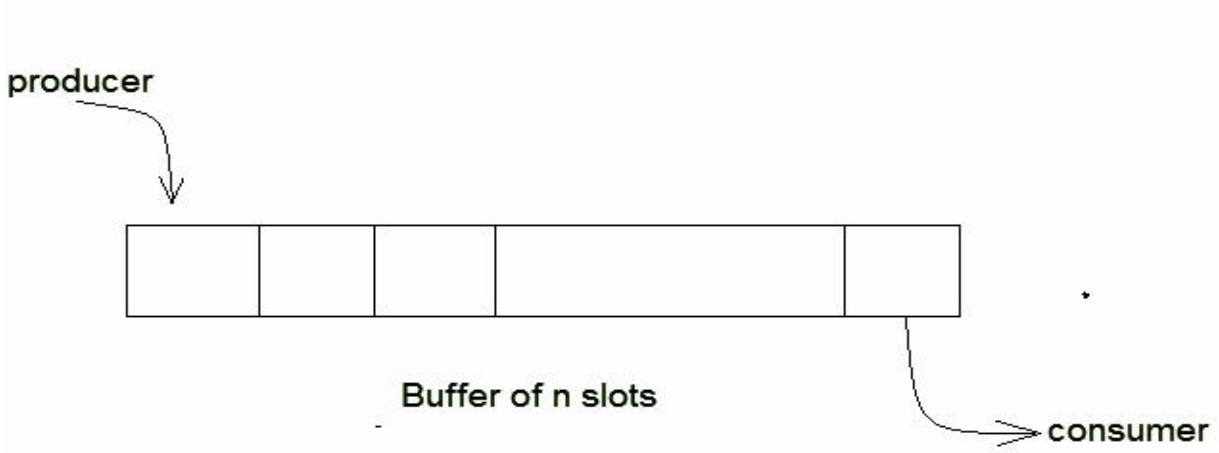
- Bounded buffer problem, which is also called **producer consumer problem**, is one of the classical problems of synchronization.
- There is a buffer of n slots and each slot is capable of storing one unit of data.
- There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Classical Problems of Synchronization: Bounded Buffer Problem

- A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As discussed earlier, those two processes won't produce the expected output if they are being executed concurrently.

- There needs to be a way to make the producer and consumer work



Classical Problems of Synchronization: Bounded Buffer Problem

- One solution of this problem is to use semaphores. The semaphores which will be used here are:
  - mutex - a **binary semaphore** which is used to acquire (wait) and release(signal) the lock.
  - empty a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
  - full, a **counting semaphore** whose initial value is 0.

- At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

Classical Problems of Synchronization: Bounded Buffer Problem

- The structure of the producer process

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

The structure of the consumer process

```
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);
```

Demonstration of the Concept by writing the Simulation: Bounded Buffer Problem

Classical Problems of Synchronization: Bounded Buffer Problem

- The structure of the producer process

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

The structure of the consumer process

```
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);
```

Demonstration of the Concept by writing the Simulation: Bounded Buffer Problem

Classical Problems of Synchronization: Readers Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write

- Problem=> allow multiple readers to read at the same time

- Only one single writer can access the shared data at the same time

Classical Problems of Synchronization: Readers Writers Problem

- Several variations of how readers and writers are considered => all involve some form of priorities

- Shared Data
  - Data set
  - Semaphore rw_mutex initialized to 1
  - Semaphore mutex initialized to 1
  - Integer read_count initialized to 0

Classical Problems of Synchronization: Readers Writers Problem

- The structure of a reader process

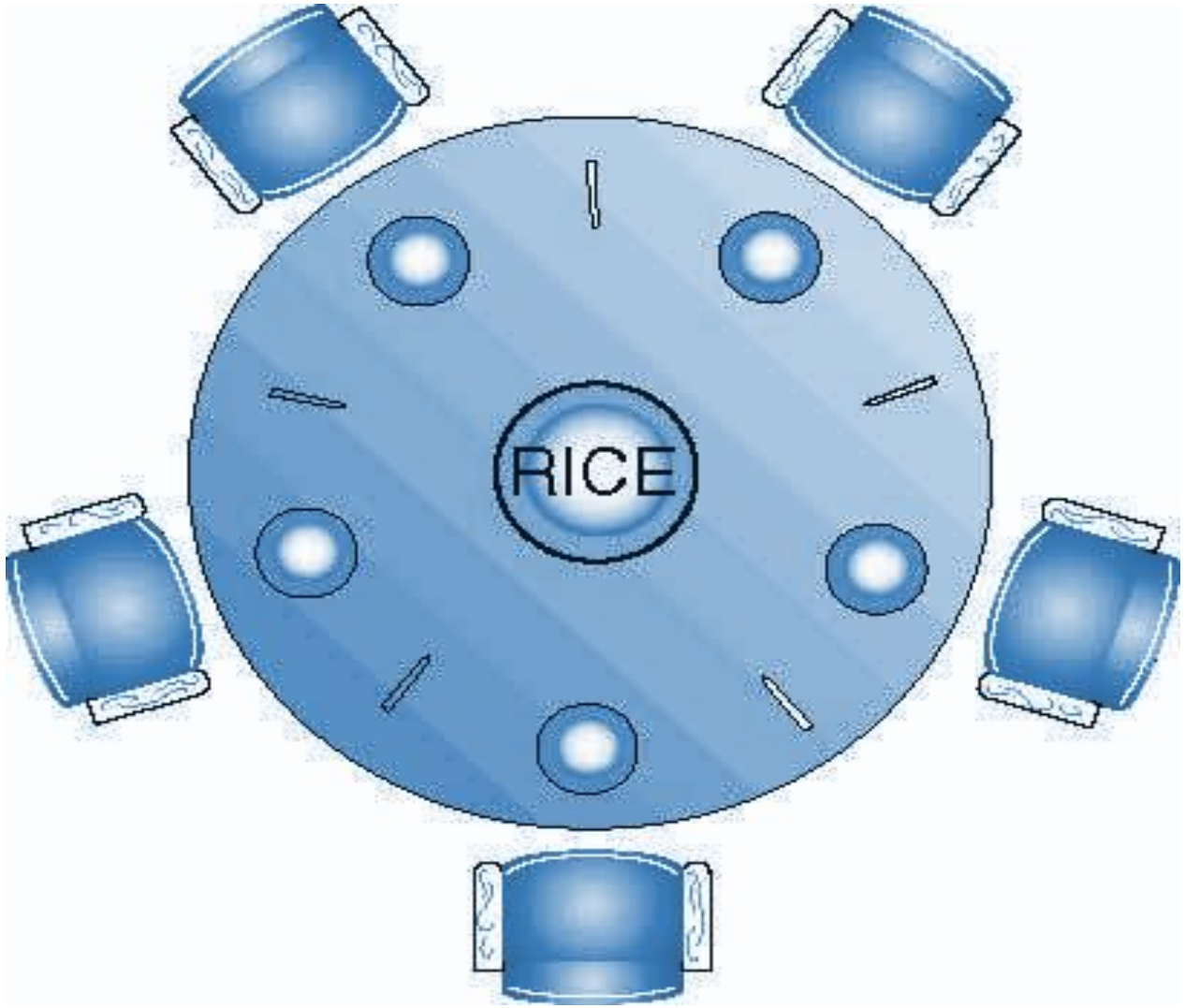
```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
          wait(rw_mutex);
      signal(mutex);

        ...
        /* reading is performed */
        ...

      wait(mutex);
        read_count--;
        if (read_count == 0)
      signal(rw_mutex);

      signal(mutex);

} while (true);
```

- The structure of a writer process

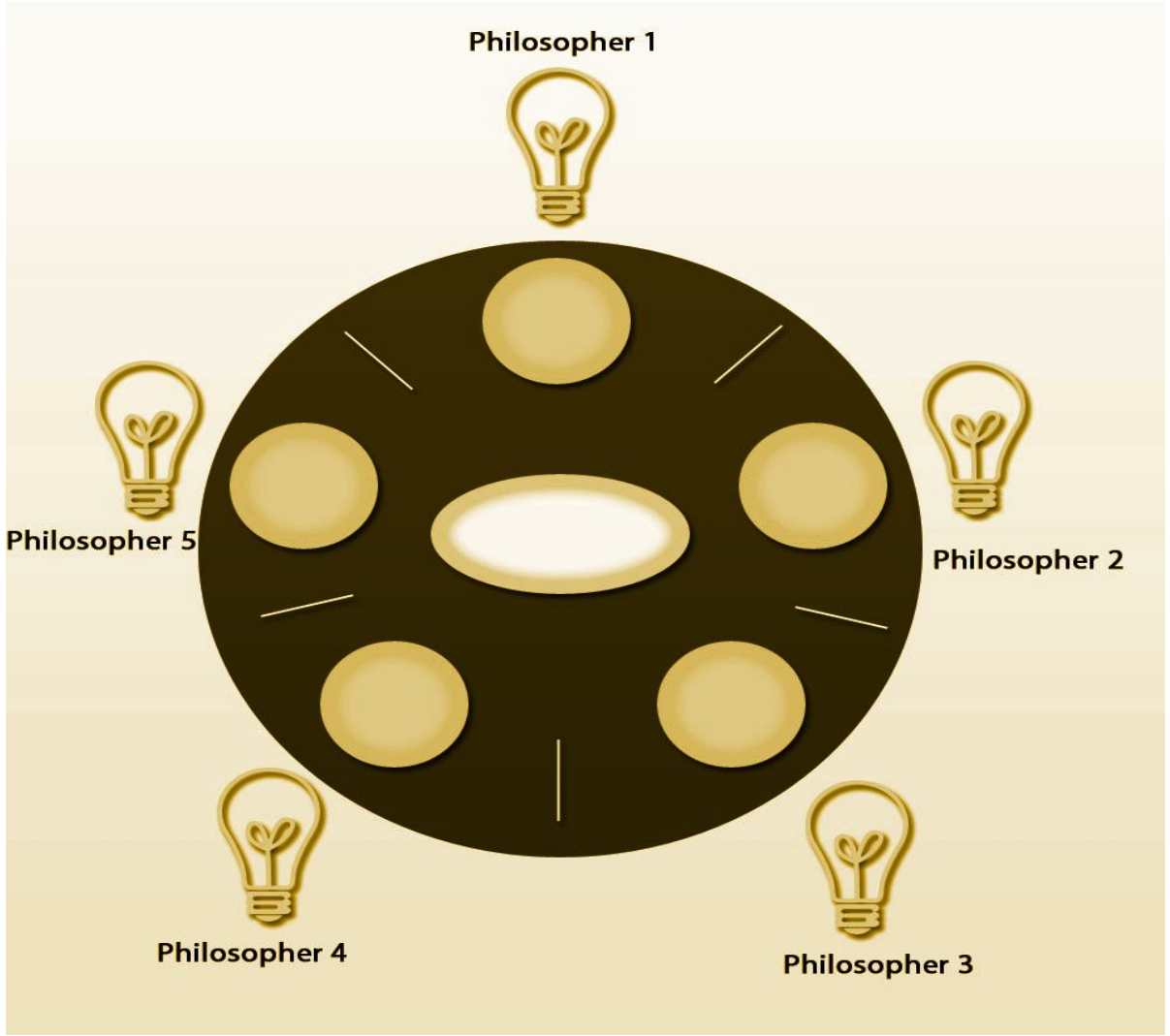
```
do {
      wait(rw_mutex);

        ...
      /* writing is performed */

        ...

    signal(rw_mutex);

} while (true);
```

Demonstrate the Concept by writing the Simulation: Readers Writers Problem

Classical Problems of Synchronization: The Dining Philosopher Problem



Classical Problems of Synchronization: The Dining Philosopher Problem



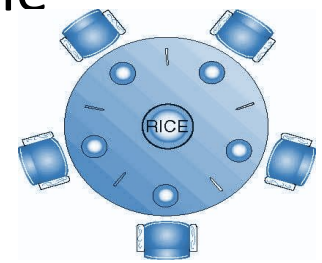
Classical Problems of Synchronization: The Dining Philosopher Problem

- The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

- Philosophers spend their lives alternating thinking and eating

- Don’t interact with their neighbors, occasionally try to pick up 2 Forks (Chopsticks) (one at a time) to eat from bowl

- Need both chopsticks to eat, then release both when done

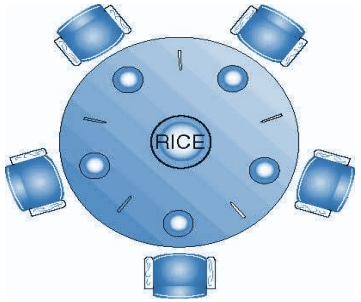


Classical Problems of Synchronization: The Dining Philosopher Problem

- In the case of 5 philosophers
- Shared data
  - Bowl of rice (data set)
  - Semaphore Fork [5] or Chopstick[5] initialized to 1

The structure of Philosopher i:

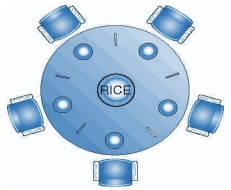
```
do {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );
        // eat

        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        //Think
} while (TRUE);
```



Classical Problems of Synchronization: The Dining Philosopher Problem

Deadlock handling

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
- Use an asymmetric solution => an odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
- Even-numbered philosopher picks up first the right chopstick and then the left chopstick.



Classical Problems of Synchronization: The Dining Philosopher Problem

- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

Topics Uncovered in this Session

- Bounded Buffer Problem
- The Readers – Writers Problem
- The Dining Philosopher 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