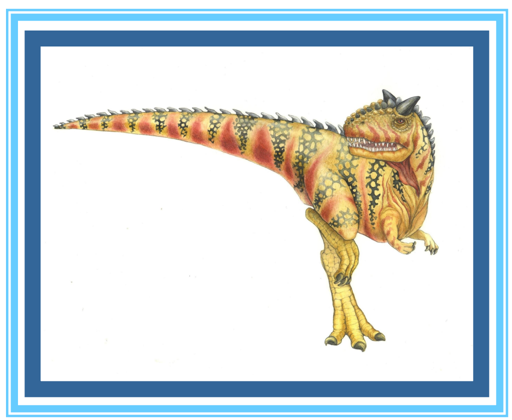
# Chapter 5: Process Synchronization

## Lecture # 15



# Previous Lecture

- **Semaphores**
- **Classical problems used to test newly-proposed synchronization schemes**
  - **Bounded-Buffer Problem**
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - **Readers and Writers Problem**
  - **Dining-Philosophers Problem**

# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - **Readers and Writers Problem**
  - Dining-Philosophers Problem

# 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
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`S`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`readcount`** initialized to 0

# Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(wrt);
          ...
        /* writing is performed */
          ...
     signal(wrt);
} while (true);
```

```
wait(S) {
     while (S <= 0)
        ; // busy wait
     S--;
}

signal(S) {
     S++;
}
```

| wrt | mutex | readcounter |
|---|---|---|
| 1 | 1 | 0 |

# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        readcount++;
        if (readcount == 1)
        wait(wrt);
    signal(mutex);
         ...
        /* reading is performed */
         ...
    wait(mutex);
        readcount--;
        if (readcount == 0)
    signal(wrt);
    signal(mutex);
} while (true);
```

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}

signal(S) {
    S++;
}
```

| wrt | mutex | readcount |
|---|---|---|
| 1 | 0 | 0 |

# Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

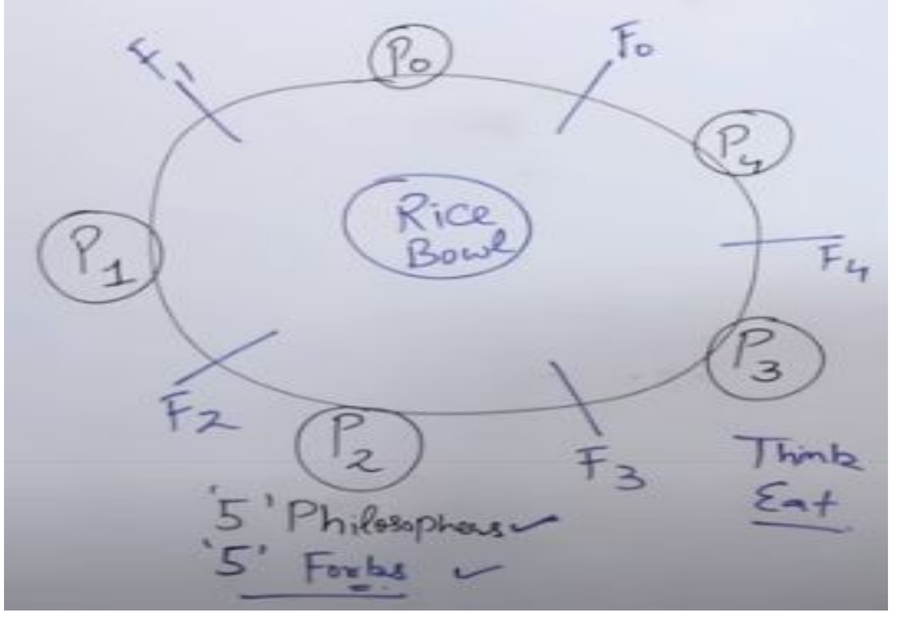
# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - **Bounded-Buffer Problem**
  - **Readers and Writers Problem**
  - Dining-Philosophers Problem

# Classical Problems of Synchronization

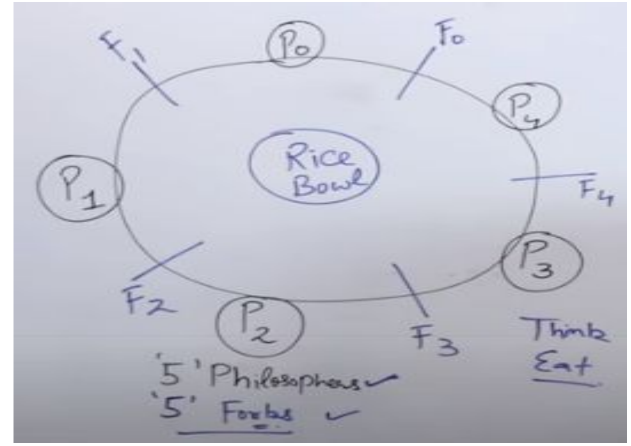
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - **Dining-Philosophers Problem**

# Dining-Philosophers Problem




- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 forks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore S [5] initialized to 1

# Dining-Philosophers Problem Algorithm




**Case-I**

- The structure of Philosopher *i*:

void philosopher(void)

```
{
While(true)
{
Thinking();
     (take-fork[i]);
     (take-fork[ (i + 1) % 5] );

                    //  eat

      (put-fork[i] );
      (put-fork[ (i + 1) % 5] );

                     //  think
}}
```

What is the problem with this algorithm?

| $P_0$ | $F_{0,}F_1$ |
|---|---|
| $P_1$ | $F_{1,}F_2$ |
| $P_2$ | $F_{2,}F_3$ |
| $P_3$ | $F_{3,}F_4$ |
| $P_4$ | $F_{4,}F_0$ |

# Dining-Philosophers Problem Algorithm

**Case-II**

- The structure of Philosopher *i*:

void philosopher(void)

```
{
While(true)
{
Thinking();
     Wait(take-fork[i]);
     wait(take-fork[ (i + 1) % 5] );

               //  eat

      signal(put-fork[i] );
      signal(put-fork[ (i + 1) % 5] );

                //  think
}}
```

- What is the problem with this algorithm?

| $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |

| $P_0$ | $S_{0,}S_1$ |
|---|---|
| $P_1$ | $S_{1,}S_2$ |
| $P_2$ | $S_{2,}S_3$ |
| $P_3$ | $S_{3,}S_4$ |
| $P_4$ | $S_{4,}S_0$ |

# Dining-Philosophers Problem Algorithm

| $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |

**Case-III**

- The structure of Philosopher *i*:

void philosopher(void)

```
{
While(true)
{
Thinking();
     Wait(take-fork[i]);
     wait(take-fork[ (i + 1) % 5] );

                // eat

      signal(put-fork[i] );
      signal(put-fork[ (i + 1) % 5] );

                 // think
}}
```

| $P_0$ | $S_0, S_1$ |
|---|---|
| $P_1$ | $S_1, S_2$ |
| $P_2$ | $S_2, S_3$ |
| $P_3$ | $S_3, S_4$ |
| $P_4$ | $S_0, S_4$ |

What is the problem with this algorithm?

# Dining-Philosophers Problem Algorithm (Cont.)

- 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.

# End of Chapter 5