
# OPERATING SYSTEM

## Synchronization Problems

**Dr Rahul Nagpal**
Computer Science

# OPERATING SYSTEM

## Synchronization Problems

**Dr. Rahul Nagpal**
Computer Science

## Outline

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

# OPERATING SYSTEMS

## Bounded Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

- 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);
```

## Reader Writer 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 **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

# Reader Writer 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);
```

## Reader Writer Problem Variation

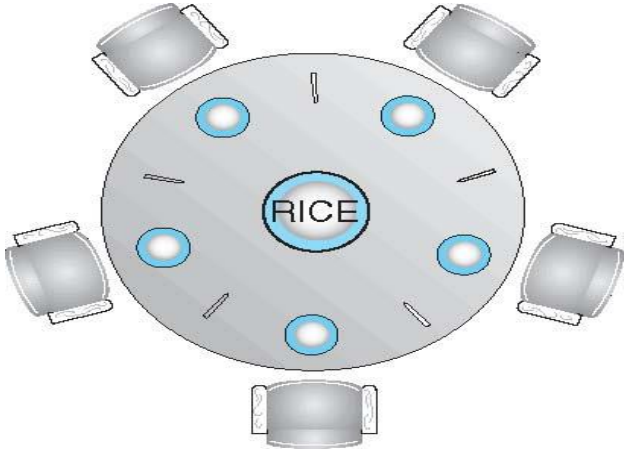
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer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

## Reader Writer Problem Variation

- ***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

# Dining Philosophers Problem (Contd.)

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (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 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);
```

## Dining Philosophers 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.

## Problem with Semaphors

- 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.


# THANK YOU

**Dr Rahul Nagpal**

Computer Science

**rahulnagpal@pes.edu**