Classical Synchronization Problem Examples

Bounded-Buffer, Readers-Writers

&

Dining-Philosophers

Classical Problems of Synchronization

- ✔ Bounded-Buffer Problem
- ✔ Readers and Writers Problem
- ✔ Dining-Philosophers Problem

Bounded-Buffer Problem

- There is a buffer of **n slots** and each slot is capable of storing **one unit of data**.
- There are two processes running, Producer and Consumer.
- Producer and Consumer are operating on the buffer.
- The producer tries to **insert** data into an **empty** slot of the buffer.
- The consumer tries to **remove** data from a **filled** slot in the buffer.
- The producer must **not insert** data when the **buffer is full**.
- The consumer must **not remove** data when the **buffer is empty**.
- The producer and Consumer should not insert and remove data simultaneously.

Bounded-Buffer Problem

Three semaphores are used:

1. **Mutex**: a binary semaphore which is used to acquire and release the lock.
2. **Empty:** a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. **Full:** a counting semaphore.

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

Bounded-Buffer Problem

- The structure of the producer process

```
while (true) {
wait(empty);
   wait(mutex);
       ...
     /* add next produced to the buffer */
       ...
    signal(mutex);
    signal(full);
}
```

Bounded-Buffer Problem

- The structure of the consumer process

```
     while (true) {
          wait(full);
          wait(mutex);
               ...
 /* remove an item from buffer to next_consumed
 */
               ...
          signal(mutex);
          signal(empty);
}
```

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 – If a writer and some other (either a reader or a writer) access the data set simultaneously.
- Only one single writer can access the shared data at the same time.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared data set.

Readers-Writers Problem

- We will make use of two semaphores and an integer variable:
- **`mutex`** : a semaphore which is used to ensure mutual exclusion when **`read_count`** is updated i.e. when any reader enters or exit from the critical section.
- **`rw_mutex`** : a semaphore common to both reader and writer processes.
- **`read_count`** : an integer variable that keeps track of how many processes are currently reading the data.
- Shared Data
  - Data set
  - Semaphore **`mutex`** initialized to 1
  - Semaphore **`rw_mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem

- The structure of a writer process

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

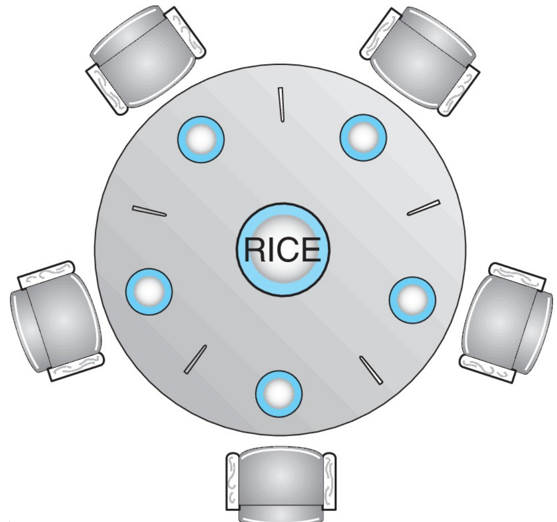
Readers-Writers Problem

- The structure of a reader process

```
while (true){
  wait(mutex);
     read_count++;
  if (read_count == 1) /* first reader */
    wait(rw_mutex);
          signal(mutex);
        ...
        /* reading is performed */
        ...
     wait(mutex);
        read count--;
    if (read_count == 0) /* last reader */
        signal(rw_mutex);
     signal(mutex);
}
```

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.



- They spend their lives alternating thinking and eating.
- They do not 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, the shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

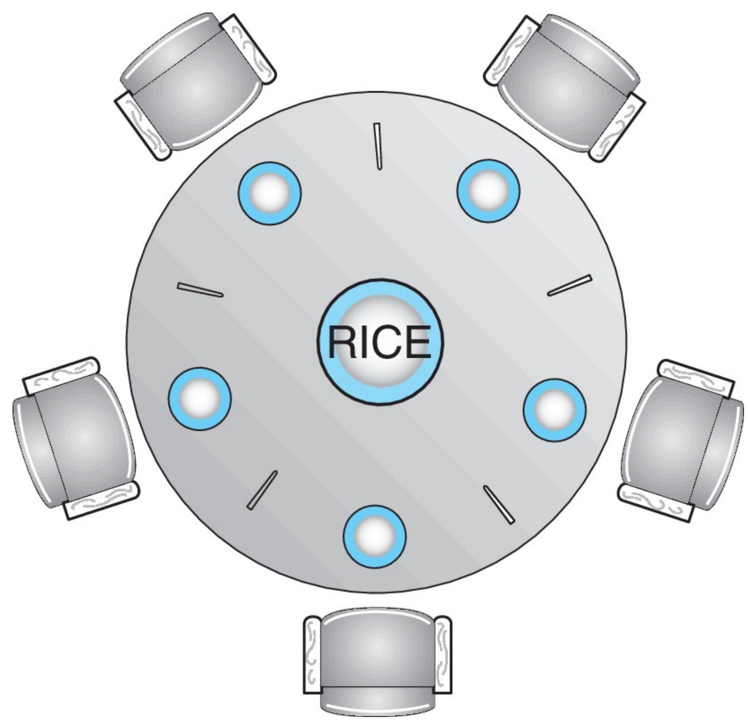
- Semaphore Solution
- The structure of Philosopher *i*:

```
while (true){
     wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

     /* think for awhile */

}
```

- What is the problem with this algorithm?

Dining-Philosophers Problem

- Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick.
- All the elements of chopstick will now be equals to 0.
- When each philosopher tries to grab his right chopstick, he will be delayed forever.



Dining-Philosophers Problem

- Several possible remedies to the deadlock problem are the following:

✔ Allow at most four philosophers to be sitting simultaneously at the table.

✔ Allow a philosopher to pick up his chopsticks only if both chopsticks are available.

✔ Use an asymmetric solution: an odd philosopher picks up first his left chopstick and then his right chopstick whereas an even philosopher picks up his right chopstick and then his left chopstick.