Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

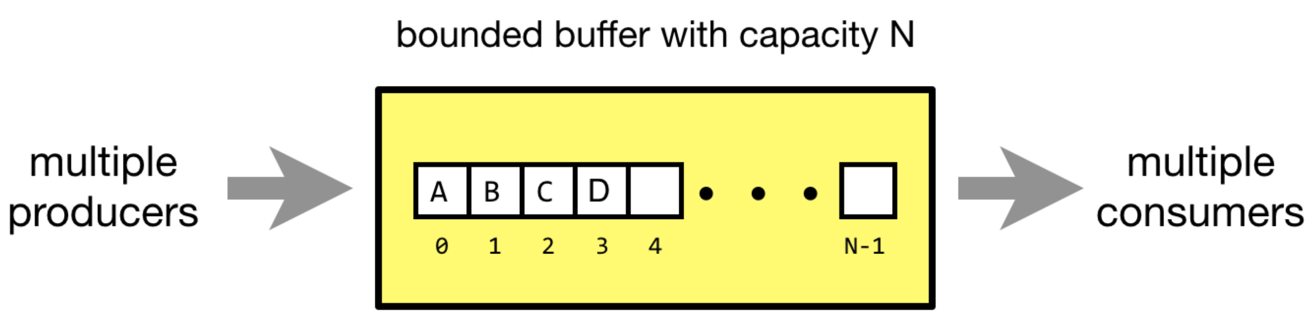
Classical Problems of Synchronization

Classical Problems of Synchronization

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

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



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
      ...
      /* produce an item in next_produced */




       ...
      /* add next produced to the buffer */
       ...



}
```

Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {



        ...
     /* remove an item from buffer to next_consumed */
        ...




       /* consume the item in next consumed */
        ...
   }
```

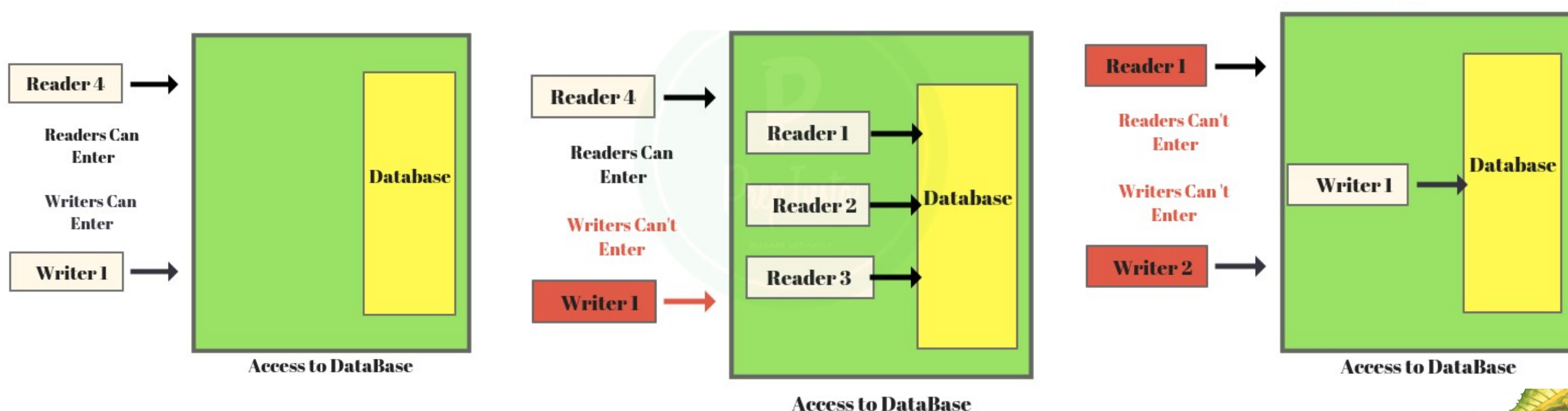
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

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



Readers-Writers Problem (Cont.)

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

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {



}
```

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){















}
```

Readers-Writers Problem (Cont.)

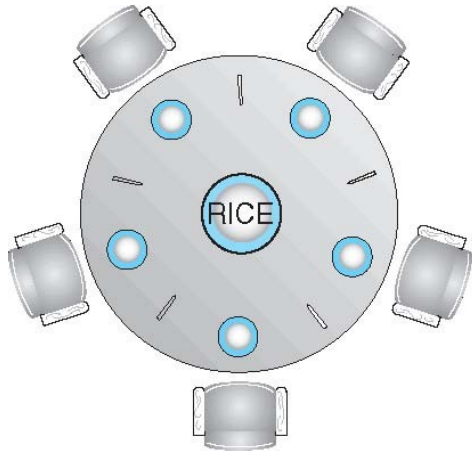
- The structure of a reader process

```
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);
}
```

Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
  - **Once a reader is ready to read, no “newly arrived writer” is allowed to read**.
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - **Once a writer is ready to write, no “newly arrived reader” is allowed to read**.
- Both the first and second may result in starvation. leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Classic problem: Dining-Philosophers Problem



- 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

Dining-Philosophers Problem Algorithm

- 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);
```

- 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.
    - Using a semaphore initialized to 4
  - 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.
  - But starvation is still possible with these solutions

Dining-Philosophers application

- What is the real application of dining philosopher problem?
  - A transaction between two accounts