OPERATING SYSTEMS

Classic problems of Synchronization

Chandravva Hebbi
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

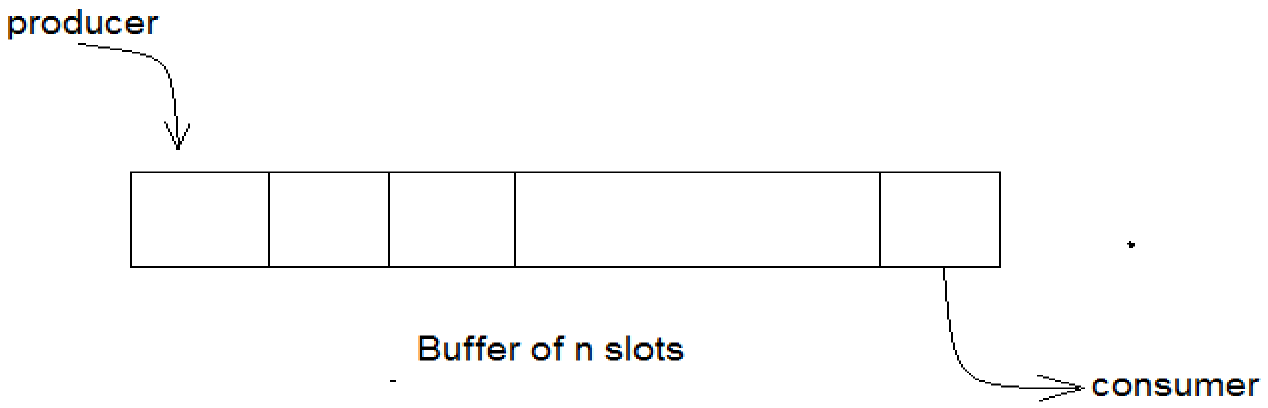
OPERATING SYSTEMS

- Classic problems of Synchronization

Chandravva Hebbi
Department of Computer Science

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

```
do {
    ...
      /* produce an item in next_produced */
    ...
  wait(empty); //wait until empty > 0 and then decrement 'empty'
  wait(mutex); ///acquire lock
     ...
      /* add next produced to the buffer */
     ...
  signal(mutex); //release a lock
  signal(full); //increment full
} while (true);
```

Bounded-Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
wait(full); // wait until full > 0 and then decrement 'full'
 wait(mutex); // acquire the lock
   ...
/* remove an item from buffer to next_consumed */
   ...
signal(mutex); // release the lock
signal(empty); // increment 'empty'
   ...
/* consume the item in next consumed */
   ...
} while (true);
```

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

- **solution**
- 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)
  - Semaphore **mutex** initialized to 1 (mutex)
  - Integer **read_count** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

Readers-Writers Problem (Cont.)

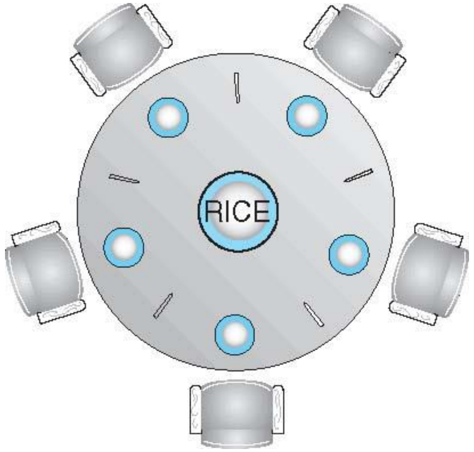
- The structure of a reader process

```
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);
```

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

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.
  - 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.

THANK YOU

Chandravva Hebbi
Department of Computer Science Engineering
chandravvahebbi@pes.edu