CS343: Operating System

Synchronization: Semaphore, Monitors

Lect22 : 26th Sept 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Synchronization
  - Critical Section Problem
- Sync Hardware
  - CAS, TAS, LL-LC, BackupLock
- Semaphore and Monitor
- Classical Sync Problems

Semaphore

- Semaphore: Synchronization tool
  - Provides more sophisticated ways (than Mutex)
  - For process to synchronize their activities.
- Semaphore: Abstract data type
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - Used for controlling access, by multiple processes
  - Can be access by two atomic **Wait()** and **Signal()**

Semaphore Usage

- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- **Counting semaphore** – integer value can range over an unrestricted domain
- Can solve various synchronization problems

Counting/Bin Semaphore

```
S=50; // Initialized to 1 for Binary
// S =0 Locked;  S>=1 Available
void synchronized  wait(S) {
            while (S <= 0) ; // busy wait
            S--;
}
void synchronized signal(S) {
                    S++;
}
```

Counting Semaphore: Real life Example

- Counting Semaphores : Representation of a limited number of resources
- If a restaurant has a capacity of **50 people**
  - And nobody is there, the semaphore would be initialized to **50**

Counting Semaphore: Real life Example

- **As each person arrives at the restaurant**
  - They cause the seating capacity to decrease
  - So the semaphore in turn is decremented.
- **When the maximum capacity is reached**
  - The semaphore will be at zero
  - Nobody else will be able to enter the restaurant.
  - Instead the hopeful restaurant goers must wait until someone is done eating.
- **When a patron leaves**
  - The semaphore is incremented
  - And the resource becomes available again.

Semaphore Usage

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
  signal(synch);
```

```
P2:
    wait(synch);
      S2;
```

- Can implement a counting semaphore S as a binary semaphore

```
wait(){while(S<=0);S--;}
signal(){S++;}
```

Semaphore Implementation

- Guarantee that no two processes can execute
  - **wait()** and **signal()** on the same semaphore at the same time
  - void synchronized wait(); void synchronized signal();
- Implementation becomes the critical section problem
  - where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

Semaphore with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore with no Busy waiting

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
     add this process to S->list;
     block();
  }
}
```

```
typedef struct{
   int value;
   struct process *list;
   } semaphore;
```

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
     remove a process P from S->list;
     wakeup(P);
  }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Deadlock and Starvation

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems **used to test** newly-proposed synchronization schemes
  1. Bounded-Buffer Problem
  2. Readers and Writers Problem
  3. 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: Producer

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

```
wait(S) {while (S <= 0) ; Add(P,S); S--; }
signal(S) {S++; wakeup rem(S.front);}
// Initialization Full=0; Empty=n
```

Bounded Buffer : Consumer

```
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);
```

wait(S) {while (S <= 0) ; Add(P,S); S--; }
signal(S) {S++; wakeup rem(S.front);}
// Initialization Full=0; Empty=n

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
- Variation one: other variations…..
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
  - **No reader kept waiting unless writer has permission to use shared object**

Readers-Writers Problem

- 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

Writer Structure : RW problem

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

```
wait(S) {while (S <= 0) ; S--; }
signal(S) {S++;}
```

Reader Structure : RW problem

- Multiple reader are allowed: **one is reading then wait for write to complete**

```
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

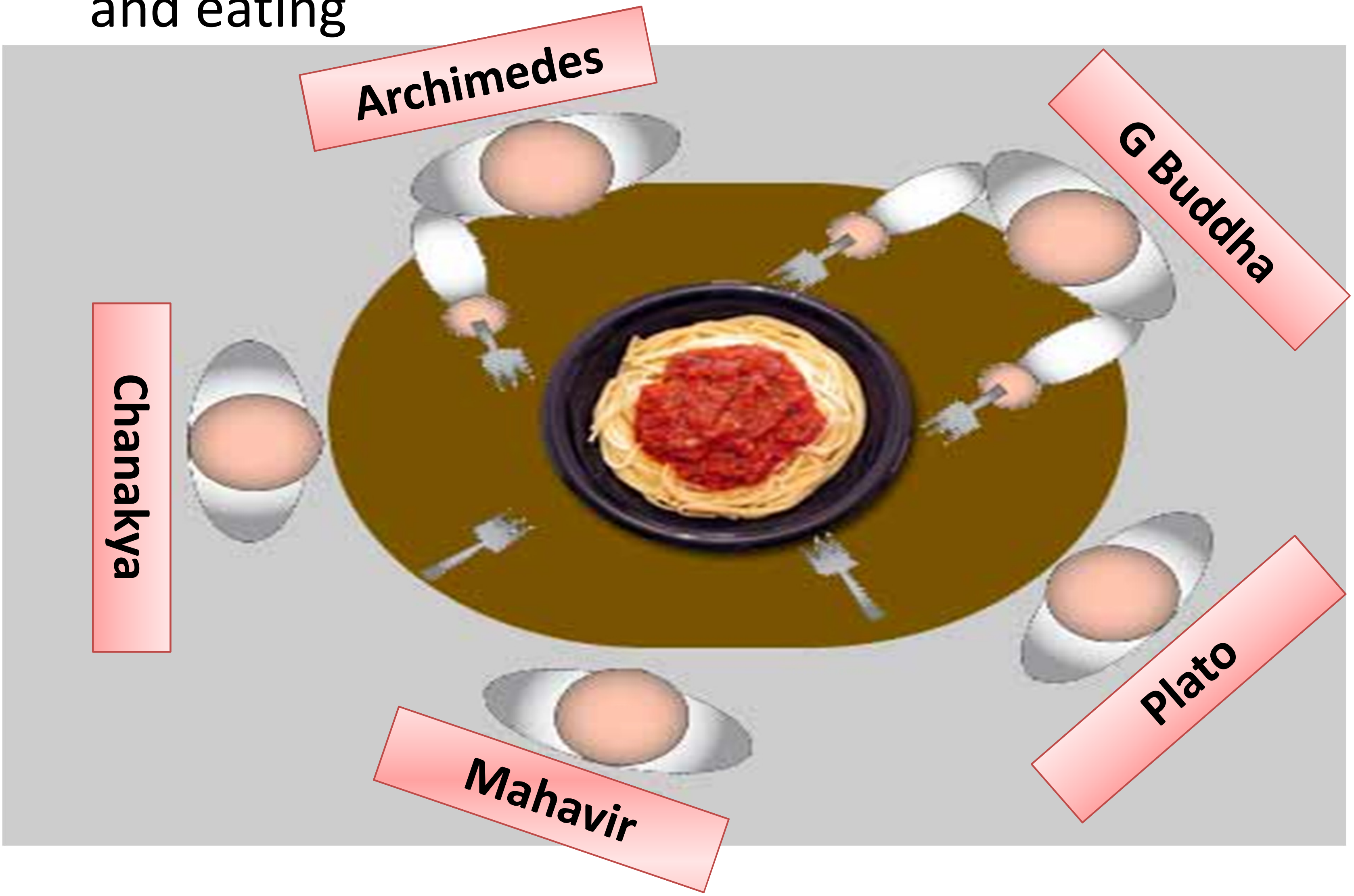
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
  - Writer have very high priority
- Both may have starvation leading to even more variations

Readers-Writers Problem Variations

- ***Third variation***: No thread shall be allowed to starve
  - Operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.
  - Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating



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,
    - 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?

Problem in Dining-Philosophers Algorithm

- May be deadlock
  - Every one is Holding one Fork and requesting for other
  - Form a circular wait

Problem in Dining-Philosophers Algorithm

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
    - 5 chop sticks, 4 people: pigeon-hole principle at least one can easily acquire 2 chop sticks , so no deadlock
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
    - Allow both or none, so only two person can get chance and there will not be any deadlock

Problem in Dining-Philosophers Algorithm

- Deadlock handling: Use an asymmetric solution
  - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick. **Left then Right**
  - Even-numbered philosopher picks up first the right chopstick and then the left chopstick. **Right then Left**
  - **As neighbor are different and circular fashion : They will allow you to pickup (if all try at same time)**