UNIT II

SEMAPHORES – MUTEX, CLASSICAL PROBLEMS OF SYNCHRONIZATION

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait( ) and signal( ) , Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

    while S <= 0

    ; // no-op

    S--;

    }
  - signal (S) {

    S++;

    }

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
     signal (mutex);
       // remainder section
} while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait( ) and signal( ) on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation 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 Implementation with no Busy waiting (Cont.)

- Implementation of wait:

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

- Implementation of signal:

```
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

Classical Problems of Synchronization

- 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

```
do  {

        //   produce an item in nextp

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
 } while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

        //  remove an item from  buffer to nextc

    signal (mutex);
    signal (empty);
      //  consume the item in nextc

} while (TRUE);
```

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
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

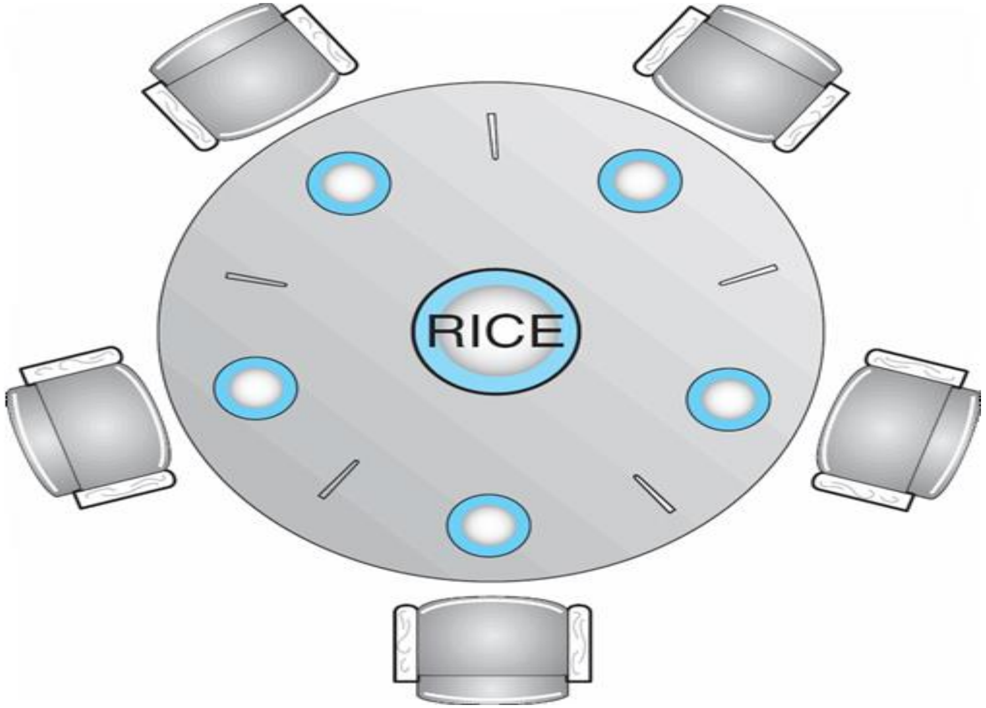
```
do {
    wait (wrt) ;

        //    writing is performed

    signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
            // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- 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);
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)