UNIT III

SYNCHRONIZATION & DEADLOCK

Synchronization & Deadlock

Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization

Deadlock

- System Model
- Deadlock characterization
- Methods for handling deadlocks
- Deadlock prevention
- Deadlock avoidance
- Deadlock Detection
- Recovery from deadlock.

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

     /*  produce an item and put in nextProduced  */
    while (count == BUFFER_SIZE)
          ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer

```
while (true)  {
     while (count == 0)
          ; // do nothing
          nextConsumed =  buffer[out];
          out = (out + 1) % BUFFER_SIZE;
          count--;

            /*  consume the item in nextConsumed
}
```

Race Condition

- count++ could be implemented as

  register1 = count

  register1 = register1 + 1

  count = register1

- count-- could be implemented as

  register2 = count

  register2 = register2 - 1

  count = register2

Race Condition

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}

  S1: producer execute register1 = register1 + 1 {register1 = 6}

  S2: consumer execute register2 = count {register2 = 5}

  S3: consumer execute register2 = register2 - 1 {register2 = 4}

  S4: producer execute count = register1 {count = 6 }

  S5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section.

  flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
do {
      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);
                critical section
      flag[i] = FALSE;
                remainder section
} while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- **Uniprocessors** – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

TestAndndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
 {
     boolean rv = *target;
     *target = TRUE;
     return rv:
 }
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
    while ( TestAndSet (&lock ))
          ;   // do nothing
            //   critical section
    lock = FALSE;
          //     remainder section

} while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
 {
       boolean temp = *a;
       *a = *b;
       *b = temp:
 }
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );
                   //    critical section

         lock = FALSE;

                 //      remainder section

 } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet( )

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

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 semaph 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

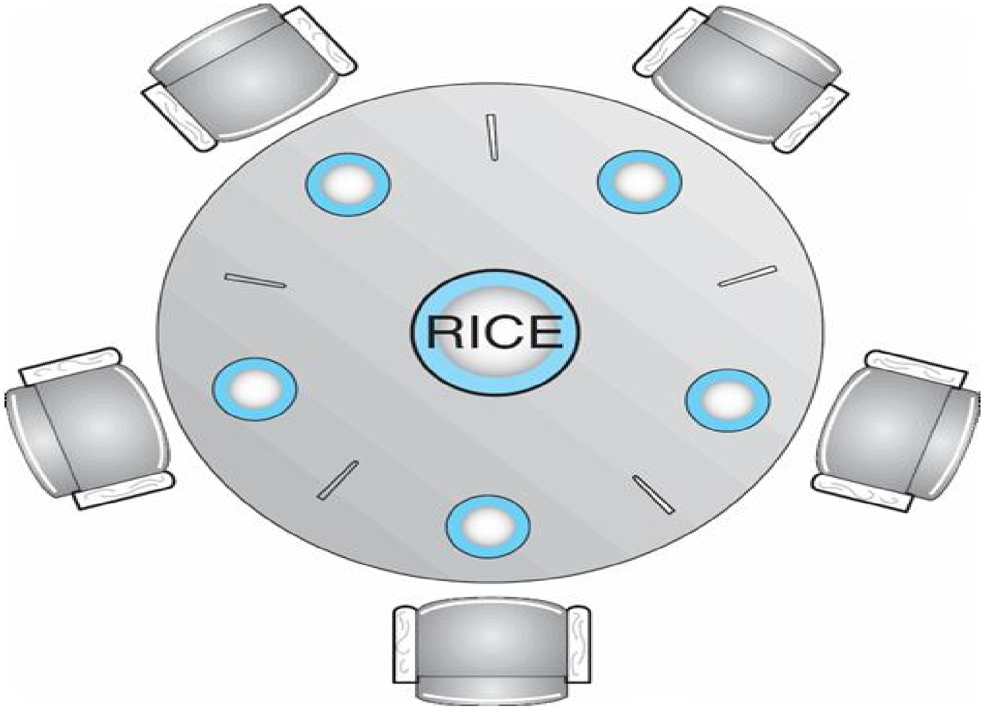
```
do {
        wait (wrt) ;

            //    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)

TEXT BOOK

1. Abraham Silberschatz, Peter B. Galvin, "Operating System Concepts", 10th Edition, John Wiley & Sons, Inc., 2018.

2. Jane W. and S. Liu. "Real-Time Systems". Prentice Hall of India 2018.

3. Andrew S Tanenbaum, Herbert Bos, Modern Operating Pearson , 2015.

REFERENCES

1. William Stallings, "Operating Systems: Internals and Design Principles",9th Edition, Prentice Hall of India., 2018.

2. D.M.Dhamdhere, "Operating Systems: A Concept based Approach", 3rd Edition, Tata McGraw hill 2016.

3. P.C.Bhatt, "An Introduction to Operating Systems–Concepts and Practice",4th Edition, Prentice Hall of India., 2013.

THANK YOU