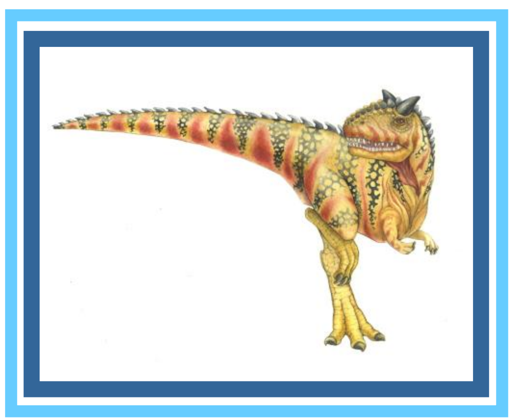
UNIT 3 : Process Synchronization and Deadlocks



Unit 3

Process Synchronization

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

Deadlocks:

System Model,

Deadlock Characterization,

Methods for handling Deadlocks,

Deadlock Prevention,

Deadlock Avoidance,

Deadlock detection and recovery from deadlock

Objectives

2.	To study and analyze various scheduling algorithms and **process synchronization techniques**
3.	To develop an **understanding about deadlocks and deadlock recovery techniques.**

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  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 **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** 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.

Consumer

```
while (true) {
while (counter == 0)
    ; /* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
        counter--;
/* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

S0: producer execute	`register1 = counter`	{register1 = 5}
S1: producer execute	`register1 = register1 + 1`	{register1 = 6}
S2: consumer execute	`register2 = counter`	{register2 = 5}
S3: consumer execute	`register2 = register2 - 1`	{register2 = 4}
S4: producer execute	`counter = register1`	{counter = 6 }
S5: consumer execute	`counter = register2`	{counter = 4}

A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Only one process at a time can be manipulating the variable counter.

So it is required that the processes be synchronized in some way.

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- It is a classical s/w based solution to CS problem.
- It is restricted to Two process solution
- 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);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock ie locks that provide mutual exclusion**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

```
do{
      wait(mutex);    ------→wait(1){
                                  while(1<=0); false
      //critical section          s - - }

      signal(mutex);   ------→ signal(0) {
                                  s++; }
      //reminder section

      }while(TRUE);
```

- Can implement a counting semaphore S as a binary semaphore

- i.e n processes share a semaphore mutex initialized to 1, Each process is organized as shown in above code.
- if value of s=0 that means some process is already executing in CS & requesting process has to wait.
- If value of s is 1, that means CS is free for currently requesting process to enter CS.
- When s completes using CS, it will call signal() operation and increment value of s ,so semaphore can be used by some other processes.

- **Counting semaphore** (integer value can range over an unrestricted domain)
- It is used to control access to a given resource consisting of a finite no of instances.
- Semaphore is initialized to number of resource available.
- Each process that wishes to use a resource performs wait() operation(by decrement the value 1 of semaphore)
- When a process release a resource, it performs a signal() operation (increment the count)
- When count of semaphore reaches 0, all resources are being used.
- After that processes that wish to use a resource will block until count becomes >0.

Semaphore Implementation

- Main disadvantage of semaphore is busy waiting.
- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the 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
- 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
- ```
  typedef struct{
     int value;
     struct process *list;
     } semaphore;
  ```
  ```

Implementation with no Busy waiting (Cont.)

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

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

Deadlock and Starvation

Implementation of a semaphore with a waiting queue may result in a Situation called Deadlock.

- **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);
```

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

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

1. 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(mutex);
         ...
        /* add next produced 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 next_consumed */
         ...
     signal(mutex);
     signal(empty);
         ...
        /* consume the item in next consumed */
         ...
     } while (true);
```

2. 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
- 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

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

Readers-Writers Problem (Cont.)

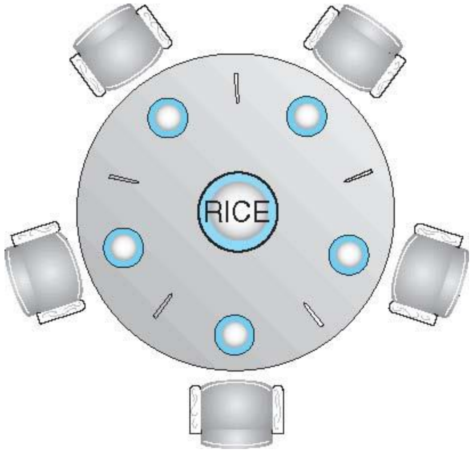
- The structure of a reader process

```
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

3. 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 chopsticks 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.

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)
- Deadlock and starvation are possible.