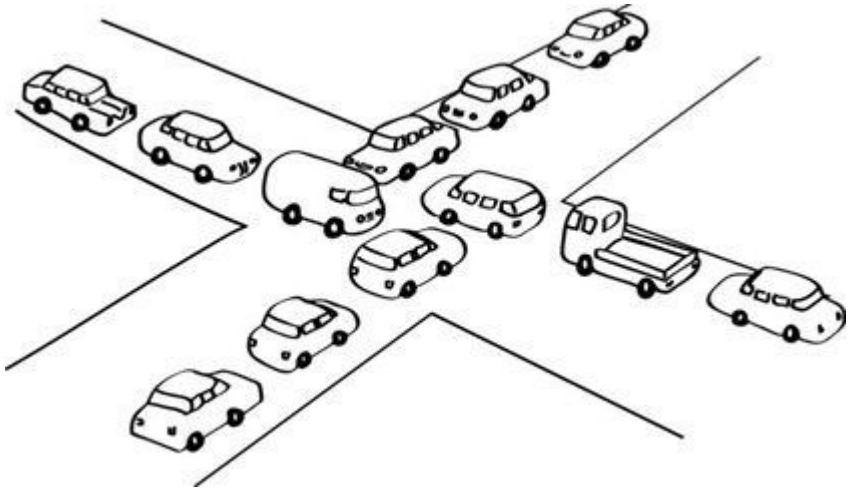
Deadlock & Process Synchronization

Jayesh Deep Dubey
Technical Officer
C-DAC Patna

Deadlock

- Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

Resource Allocation Graph

- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system
- Two types of edges:
  - request edge – directed edge Pi Rj
  - assignment edge – directed edge Rj Pi

Basics for deadlock detection in RAG

- If graph contains no cycles no deadlock
- If graph contains a cycle
- If only one instance per resource type, then deadlock
- If several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX also known as Ostrich method)

Deadlock Prevention

Prevent all or atleast one of the 4 conditions of deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.

Deadlock Prevention

Prevent all or atleast one of the 4 conditions of deadlock:

- **No Preemption –** If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- **Circular Wait –** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

- Requires that the system has some additional a priori information available
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock Avoidance Basic Rules

- If a system is in safe state => no deadlocks
- If a system is in unsafe state => possibility of deadlock
- Avoidance ensure that a system will never enter an unsafe state.

Banker’s Algorithm

- Banker's algorithm is a deadlock avoidance algorithm used to avoid deadlock and for resource allocation safely to each process in the system
- It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Banker's Algorithm Data Structure

Let n = number of processes, and m = number of resources types

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type Rj available
- **Max**: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj
- **Allocation**: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
- **Need**: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Example of Banker's Algorithm

- 5 processes P0 through P4;
- 3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

- Snapshot at time T0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example of Banker's Algorithm

- The content of the matrix Need is defined to be Max – Allocation

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence **< P1, P3, P4, P2, P0>** satisfies safety criteria

Banker’s Algorithm Question 1

- Let us consider the following snapshot of a system:

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Banker’s Algorithm Question 2

- Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Process Synchronisation

- 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

Critical Section

- Critical section refers to a segment of code that is executed by multiple concurrent threads or processes, and which accesses shared resources.
- The critical section must be executed as an atomic operation, which means that once one thread or process has entered the critical section, all other threads or processes must wait until the executing thread or process exits the critical section.
- The purpose of synchronization mechanisms is to ensure that only one thread or process can execute the critical section at a time.

Race Condition

- A race condition is a problem that occurs when two or more processes or threads are executing concurrently.
- The outcome of their execution depends on the order in which they are executed.
- In a race condition, the exact timing of events is unpredictable, and the outcome of the execution may vary based on the timing. This can result in unexpected or incorrect behavior of the system.

Example: If two threads are simultaneously accessing and changing the same shared resource, such as a variable or a file, the final state of that resource depends on the order in which the threads execute. If the threads are not correctly synchronized, they can overwrite each other's changes, causing incorrect results or even system crashes.

Solution to Critical Section Problem

- **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- **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
- **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

Semaphore

- Synchronization tool that provides a method for process to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
    - wait() and signal()

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

Mutex Locks

- A simple method to solve critical section problem
- Protect a critical section by first acquire() a lock then release() the lock
- Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic

Mutex Locks

Acquire a Lock

```
acquire() {
        while (!available)
           ; /* busy wait */
        available = false;
}
```

Release a lock

```
release() {
        available = true;
}
```

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

Producer Consumer Problem

- Producer-Consumer problem is a classical synchronization problem in the operating system.
- Producer is a process which is able to produce data/item
- Consumer is a Process that is able to consume the data/item produced by the Producer.
- Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.

Producer Consumer Problem (Rules)

1. Producer Process should not produce any data when the shared buffer is full.
2. Consumer Process should not consume any data when the shared buffer is empty.
3. The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.

Producer Consumer Problem (Solution)

```
int count = 0;
void producer( void )
{
 int itemP;
 while(1)
 {
  Produce_item(item P)
  while( count == n);
  buffer[ in ] = item P;
  in = (in + 1)mod n
  count = count + 1;
 }
}
```

Producer Code

```
int count;
void consumer(void)
{
 int itemC;
 while( 1 )
 {
  while( count == 0 );
  itemC = buffer[ out ];
  out = ( out + 1 ) mod n;
  count = count -1;
 }
}
```

Consumer Code

Deadlock and Starvation

- **Deadlock** occurs when two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- **Starvation** is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time.In starvation resources are continuously utilized by high priority processes. Problem of starvation can be resolved using Aging. In Aging priority of long waiting processes is gradually increased.

Thank You