Operating System

Unit – 5
Concurrency Control & Dead Lock

By: Vibhuti Patel
PICA - BCA

Race Condition

- 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 **race condition**.
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable/data.

Critical Section Problem

- Consider a system consisting of n processes {P0, P1,…,Pn-1}.
- Each process has a segment of code, called a **critical section**, in which,
  - the process may be changing common variables
  - updating a table
  - writing a file and so on

Critical Section Problem

- The important feature of the system is that, when one process is executing in its critical section, ***no other*** process is to be allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections ***at the same time***.
- The critical section problem is ***to design a protocol*** that the processes can use to cooperate.

Critical Section Problem

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section.**
- The remaining code is the **remainder section.**
- A solution to the critical section problem must satisfy the following three requirements:

1. Mutual exclusion
2. Progress
3. Bounded waiting

Critical Section Problem

```
do {

        entry section

            critical section

        exit section

            remainder section

} while (TRUE);
```

Figure 6.3 General structure of a typical process P_i.

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
  - some processes wish to enter their critical sections,

  then
  - only those processes that are ***not*** executing in their remainder sections can participate in the decision on which will enter its critical section next, and
  - this selection cannot be postponed indefinitely.

Critical Section Problem

- **Bounded waiting**: there exists a bound, or limit, 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.
- We assume that each process is executing at nonzero speed. However, we can make no assumption concerning the relative of the n processes.

Synchronization Hardware

- In general, we can state that any solution to the critical section problem requires a simple tool – a lock.
- Race conditions are prevented by requiring that critical regions be protected by locks.
- That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

Synchronization Hardware

- The critical section problem could be solved simply in a uni-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable. This approach is taken by non-preemptive kernels.

Semaphores

- A semaphore is a synchronization tool.
- A Semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait()** and **signal().**
- The definition of wait() is as follows:

```
wait (S) {
    while S < = 0
     ; // no – op
    S--;
 }
```

- The definition of signal() is as follows:

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

Semaphores

- When one process modifies the semaphore value, ***no other process*** can ***simultaneously*** modify that same semaphore value.
- In addition, in the case of wait(S), the testing of the integer value of S (S <= 0), and its possible modification (S--), must also be executed ***without interruption***.

Usage of Semaphores

- OS often distinguish between counting and binary semaphores.
- The value of a counting semaphore can range over an unrestricted domain.
- The value of binary semaphore can range only between 0 and 1.
- On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.
- We can use binary semaphores to deal with the critical section problem for multiple purpose.
- The n processes share a semaphore, mutex, initialized to 1.
- Each process Pi is organized as shown in following figure:

Mutual exclusion implementation with Binary Semaphores

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

Usage of Binary Semaphores

- We can also use semaphores to solve various synchronization problems.
- Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2.
- Suppose we require that ***S2 be executed only after S1 has completed***. We can implement this scheme readily by letting P1 and P2 share a common semaphore **synch,** initialized to 0, and by inserting the statements

```
S1;
signal(synch);
```

In process P1, and the statements

```
wait(synch);
S2;
```

In process P2.

- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(signal), which is after statement S1 has been executed.

Usage of Counting Semaphores

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

The Readers-Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database (Readers), whereas other may want to update the database(Writers).
- If two readers access the shared data simultaneously, no adverse affects will result.
- If a writer and some other thread (reader or writer) access the database simultaneously, chaos may cause.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.
- This synchronization problem is referred to as the readers-writers problem.

The Dining Philosophers Problem

- The dining philosophers problem is
  - a classic synchronization problem
  - involving the allocation of limited resources
  - amongst a group of processes in a deadlock-free and starvation-free manner

The Dining Philosophers Problem

- Consider five philosophers sitting around a table, in which,
    - there are five chopsticks evenly distributed and
    - an endless bowl of rice in the center, as shown in the diagram.
    - There is exactly one chopstick between each pair of dining philosophers
- These philosophers spend their lives alternating between two activities: **eating and thinking**.
- When it is time for a philosopher to eat, he must first acquire two chopsticks - one from their left and one from their right.
- When a philosopher thinks, he puts down both chopsticks in their original locations.

The Dining Philosophers Problem

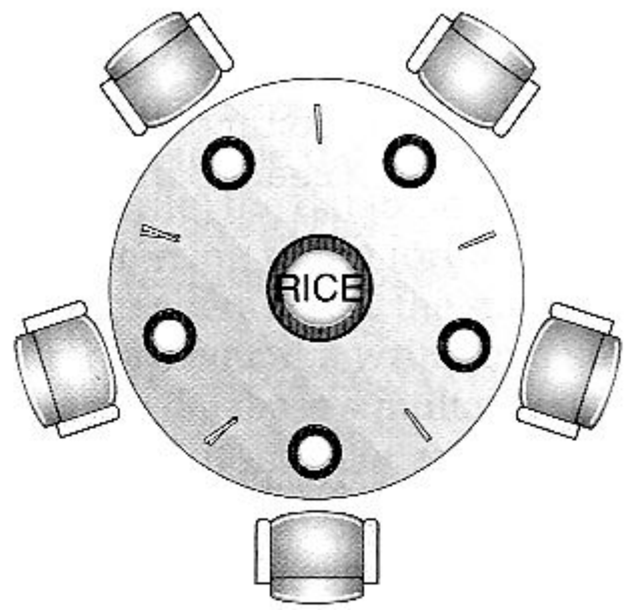
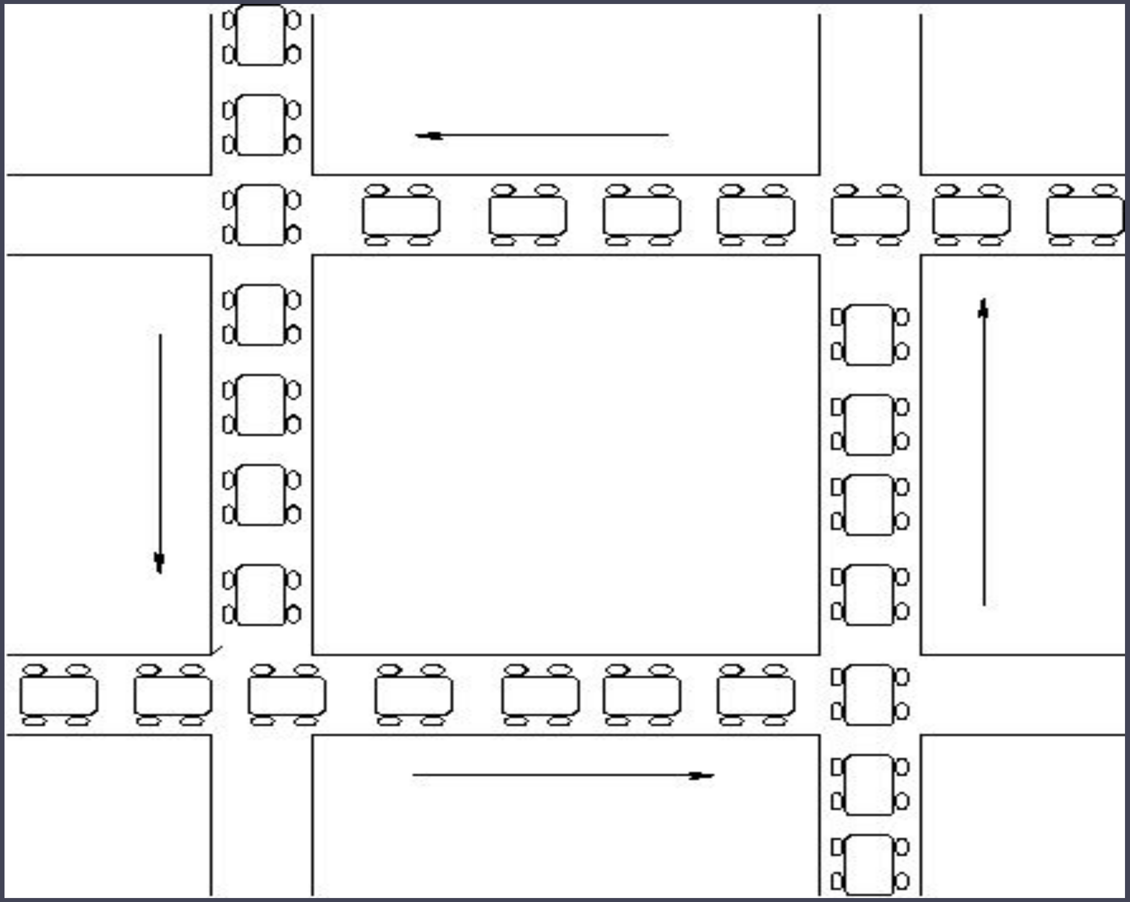


Figure 6.14 The situation of the dining philosophers.

Deadlock



Deadlock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; and if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- This situation is called a deadlock.

Deadlock

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Memory space, CPU cycles, files, and I/O devices are examples of resource types.
- If a system has two CPUs, then the resource type CPU has two instances.
- If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request.

Deadlock

- A process must request a resource before using it and must release the resource after using it.
- A process may request as many resources as it requires to carry out its designated task.
- The number of resources requested may not exceed the total number of resources available in the system
- Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: if the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. Use: the process can operate on the resource.
3. Release: the process releases the resource.

Necessary Conditions for Deadlock

- A deadlock situation can arise if the following four conditions ***hold simultaneously*** in a system:

1. **Mutual exclusion**: at least one resource must be held in a non-sharable mode; that is,
   - only one process at a time can use the resource
   - If another process requests that resource, the requesting process must be delayed until the resource has been released
2. **Hold and Wait**: a process ***must be holding*** at least one resource and waiting to acquire additional resources that are ***currently being held by other processes***.

Necessary Conditions for Deadlock

3. **No preemption**: Resources cannot be preempted; that is, a resource can be released ***only voluntarily*** by the process holding it, after that process has completed its task.
4. **Circular wait:** a set { $P_0, P_1, \ldots, P_n$} of waiting processes must exist such that P0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2,…., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

- We emphasize that all four conditions must hold for a deadlock to occur.

Deadlock Prevention

- Deadlock prevention provides a set of methods for ensuring that ***at least one of the necessary conditions for deadlock cannot hold***.
- **Mutual Exclusion**: the mutual-exclusion condition must hold for ***non-sharable resources***.
- Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Read only files are good example of a sharable resource.
- If several processes attempt to open it at same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.
- In general, we ***cannot prevent*** deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold And Wait

- To ensure that the hold and wait condition never occurs in the system, we must guarantee that, ***whenever a process requests a resource, it does not hold any other resources***.
- One protocol that can be used requires each process to request and be allocated all its resources ***before it begins execution.***
- An alternative protocol allows a process to request resources ***only when it has none***. A process may request some resources and use them.
- Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- Both these protocols have two main disadvantages:

1. Resource utilization may be low, since resources may be allocated but unused for a long period.
2. Starvation is possible. A process that needs several popular resources may have to wait indefinitely.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- To ensure that this condition does not hold, we can use the following protocol.
- If a process is holding some resources and requests another resource ***that cannot be immediately allocated*** to it, then ***all resources currently being held are preempted***.
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

No Preemption

- Alternatively, if a process requests some resources, we first check whether they are available.
- If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
- If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

No Preemption

- If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them.
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.
- It cannot generally be applied to such resources as printers and tape drives.

Circular Wait

- The fourth and final condition for deadlocks is the circular-wait condition.
- One way to ensure that this condition never holds is to impose a ***total ordering of all resource types*** and to require that each process requests resources in ***an increasing order of enumeration.***
- Let R = { R1,R2,…,Rm } be the set of Resource types.
- We assign to each resource type a ***unique integer number***, which allows us to compare two resources an to determine whether one precedes another in our ordering.

Circular Wait

- Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any number of instances of resource type say Ri.
- After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri).
- **Alternatively**, we can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri, such that F(Ri) >= F(Rj)

Deadlock Avoidance

- Possible ***side effects of preventing deadlocks*** by deadlock prevention algorithms are ***low device utilization*** and ***reduced system throughput.***
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- The various algorithms that use this approach differ in the amount and type of information required.

Deadlock Avoidance

- The simplest and most useful model requires that ***process declare the maximum number of resources of each type that it may need***.
- Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- **A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition never exists**.

Example of Deadlock Avoidance

To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4 tape drives, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2 tape drives, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process P_0 can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P_2 can get all its tape drives and return them (the system will then have all 12 tape drives available).

Deadlock Detection

- If a system does not employ either a deadlock-prevention or deadlock avoidance algorithm, then a deadlock situation may occur.
- In this environment, the system must provide:

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred
2. An algorithm to recover from the deadlock

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  1. To abort one or more process to break the circular wait.
  2. To preempt some resources from one or more of the deadlocked processes.

Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

1. **Abort all deadlocked processes**: This method clearly will break the deadlock cycle, but at great expense.
2. **Abort one process at a time until the deadlock cycle is eliminated**: This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Resource Preemption

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- If the preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim**: which resources and which processes are to be preempted?
2. **Rollback**: if we preempt a resource from a process, what should be done with that process?
3. **Starvation**: How do we ensure that starvation will not occur? [we must ensure that a process can be picked as a victim only a finite number of times. We can count number of rollbacks and use them.]

Thank You....