

UNIT III

PROCESS SYNCHRONIZATION AND DEADLOCK

Operations on Processes, Cooperating Processes, Inter-process Communication: Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer's Problem, Dining Philosopher Problem etc.

Deadlocks: Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention, Deadlock Avoidance: Banker's algorithm, Deadlock detection and Recovery.

Operations on Processes:

1. Process Creation

- Processes may create other processes through appropriate system call, such as **fork**. The process which creates is termed the **parent** and the created process is termed the **child**.
- Each process is given an *integer identifier*, termed its **process identifier**, or **PID**. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed **sched**, and is given **PID 0**. The first thing it does at system startup time is to launch **init**, which gives that process **PID 1**. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure shows a typical process tree for a Linux system

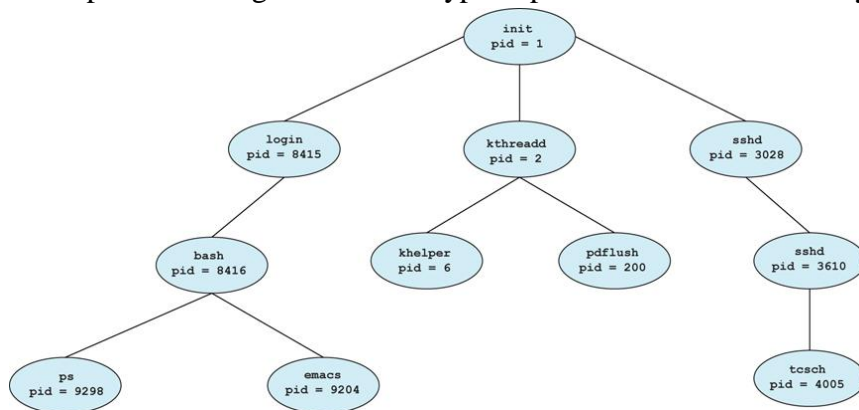


Figure - A tree of processes on a typical Linux system

- A child process may receive some amount of shared resources with its parent.
- Or dedicated resources may be allocated to Child processes.

- There are two options for the parent process *after creating the child*:
 1. **Wait for the child process to terminate** before proceeding. The parent makes a *wait() system call*, for either a specific child or for any child, which causes the parent process to block until the *wait()* returns.
 2. **Run concurrently with the child, continuing to process without waiting.** This is the operation seen when a UNIX shell runs a process as a background task.
- Two possibilities for the **address space** of the child relative to the parent:
 1. The child may be an **exact duplicate of the parent**, sharing the same program and data segments in memory. Each will have their own PCB. This is the behavior of the **fork** system call in UNIX.
 2. The child process may have a **new program loaded into its address space**, with all new code and data segments. UNIX systems implement this as a second step, using the **exec** system call.
- Figures below shows the fork and exec process on a UNIX system. The **fork** system call returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the *getpid()* and *getppid()* system calls respectively.

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}

```

Figure 3.10 C program forking a separate process.

Figure Creating a separate process using the UNIX fork() system call.

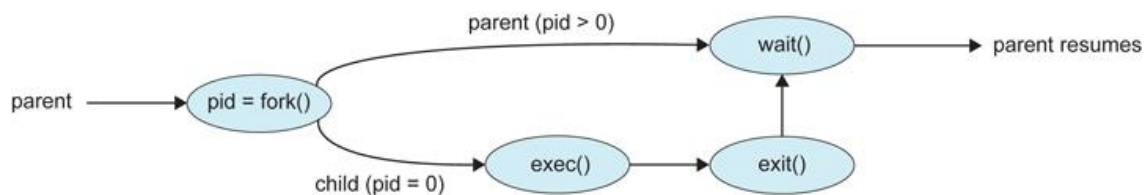


Figure - Process creation using the fork() system call

Summary of Process creation:

- ☐ Parent process create children processes, which, in turn create other processes, forming a tree of processes
- ☐ Generally, process identified and managed via a process identifier (pid)
- ☐ Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources

□ Execution

- Parent and children execute concurrently
- Parent waits until children terminate

□ Address space

- Child duplicate of parent
- Child has a program loaded into it

□ UNIX examples

- fork system call creates new process
- exec system call used after a fork to replace the process' memory space with a new program

2. Process Termination

- Processes may request their *own termination* by making the **exit()** system call and is typically **zero** on successful completion and some **non-zero** code in the event of problems.
 - code:

```
int exitCode;

exit( exitCode ); // return exitCode; has the same effect when executed from
                  main( )
```
- Processes may also be terminated by the system for a variety of *reasons*, including:
 - The **inability** of the system to deliver necessary **system resources**.
 - In response to a **KILL command**, or other un handled process interrupt.
 - **A parent may kill its children** if the task assigned to them is no longer needed.
 - If the parent exits, the system may or may not allow the child to continue without a parent.
- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc.

- An **orphan process** is a process that is still executing, but whose parent has died.
- A **zombie process** is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. [When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, whereupon the zombie is removed].

Summary of Process Termination:

- Process executes last statement and asks the operating system to delete it (*exit*)
 - Output data from child to parent (*via wait*)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (*abort*)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - Some operating systems do not allow child to continue if its parent terminates
 - All children terminated - *cascading termination*

Interprocess Communication

A process can be of two type:

- Independent process.
- Co-operating process.

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes nor be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes.

There are several reasons(advantages) why cooperating processes are allowed:

- **Information Sharing** - There may be several processes which need access to the same file.
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously
- **Modularity** - The most efficient architecture may be to break a system down into cooperating modules.
- **Convenience** - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Cooperating processes require some type of inter-process communication, which is most commonly one of two types: (Processes can communicate with each other using these two ways:)

- **Shared Memory systems**
- **Message Passing systems**

Figure illustrates the difference between the two systems:

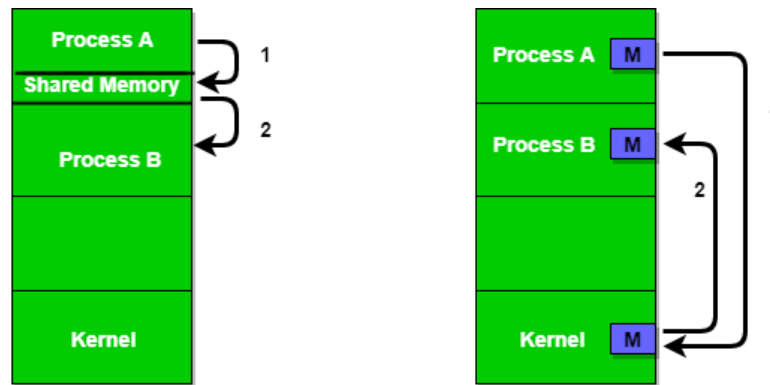


Figure 1 - Shared Memory and Message Passing

Figure - Communications models: (a) Shared memory (b) Message passing.

Shared memory Vs. Message passing:

- **Shared Memory**

It is *faster* once it is set up, because *no system calls* are required and access occurs at normal memory speeds. However it is *more complicated to set up*, and doesn't work as well across multiple computers. Shared memory is generally preferable when *large amounts of information* must be shared quickly on the same computer.

- **Message Passing**

It requires *system calls* for every message transfer, and is therefore *slower*, but it is *simpler to set up* and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of *data transfers is small*.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data.
- The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.

- This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.
- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

- Then the **producer process**. Note that the buffer is full when "in" is one less than "out" in a circular sense:

```
item nextProduced;

while( true ) {

    /* Produce an item and store it in nextProduced */
    nextProduced = makeNewItem( ... );

    /* Wait for space to become available */
    while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
        ; /* Do nothing */

    /* And then store the item and repeat the loop. */
    buffer[ in ] = nextProduced;
    in = ( in + 1 ) % BUFFER_SIZE;

}
```

- Then the **consumer process**. Note that the buffer is empty when "in" is equal to "out":

```
item nextConsumed;

while( true ) {

    /* Wait for an item to become available */
    while( in == out )
        ; /* Do nothing */

}
```



```

/* Get the next available item */
nextConsumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in nextConsumed
   ( Do something with it ) */

}

```

Message-Passing Systems

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are **three key issues** to be resolved in message passing systems as further explored in the next three subsections:
 - Direct or indirect communication (Naming)
 - Synchronous or asynchronous communication
 - Automatic or explicit buffering.

1. Naming

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
 - There is a one-to-one link between every sender-receiver pair.
 - For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared *mailboxes, or ports*.
 - Multiple processes can share the same mailbox or boxes.
 - Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
 - The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

2. Synchronization

- Either the sending or receiving of messages may be either **blocking** or **non-blocking**.

There are basically three most preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive

- **Blocking send** means the sender will be blocked until the message is received by receiver.
- **Blocking receive** has the receiver block until a message is available.
- **Non-blocking send** has the sender send the message and continue.
- **Non-blocking receive** has the receiver receive a valid message or null.

3. Buffering

- Messages are passed via queues, which may have one of three capacity configurations:
 1. **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
 2. **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
 3. **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.

Process Synchronization

On the basis of synchronization, processes are categorized as one of the following two types:

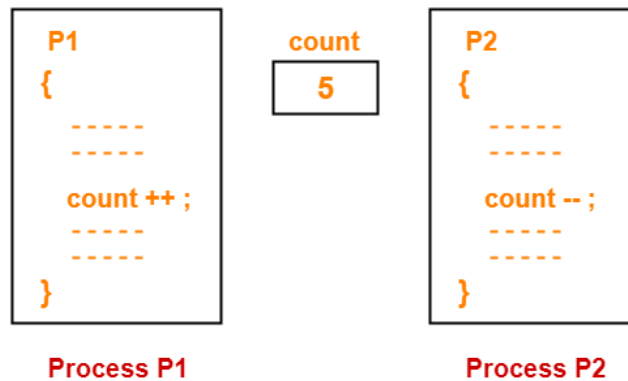
- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes. Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Need of Synchronization:

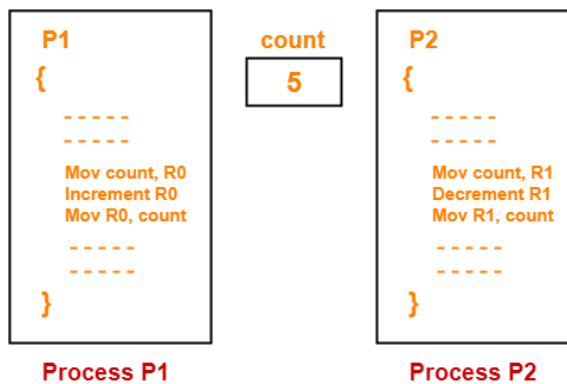
Process synchronization is needed-

- When multiple processes execute concurrently sharing some system resources.
- To avoid the inconsistent results.

Example:



In assembly language, the instructions of the processes may be written as-



Now, when these processes execute concurrently without synchronization, different results may be produced.

Case-01:

The execution order of the instructions may be-

$P_1(1), P_1(2), P_1(3), P_2(1), P_2(2), P_2(3)$

In this case,

Final value of count = 5

Case-02:

The execution order of the instructions may be-

$P_1(1), P_2(1), P_2(2), P_2(3), P_1(2), P_1(3)$

In this case,

Final value of count = 6

It is clear from here that inconsistent results may be produced if multiple processes execute concurrently without any synchronization.

Race Condition:

Race condition is a situation where-

- The final output produced depends on the execution order of instructions of different processes.
- Several processes compete with each other. The above example is a good illustration of race condition.

Critical Section :

Critical section is a section of the program, where a process access the shared resources during its execution.

Critical Section Problem:

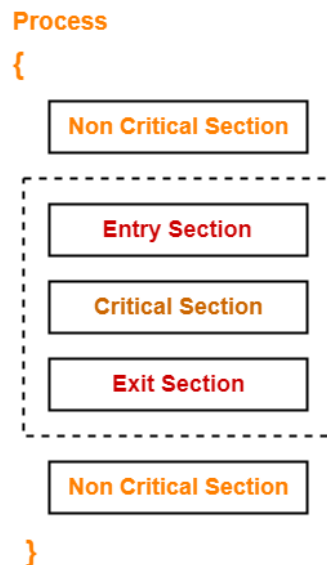
- If multiple processes access the critical section concurrently, then results produced might be inconsistent.
- This problem is called as **critical section problem**.

Synchronization Mechanisms

Synchronization mechanisms allow the processes to access critical section in a synchronized manner to avoid the inconsistent results.

For every critical section in the program, a synchronization mechanism adds-

- An entry section before the critical section
- An exit section after the critical section



Entry Section

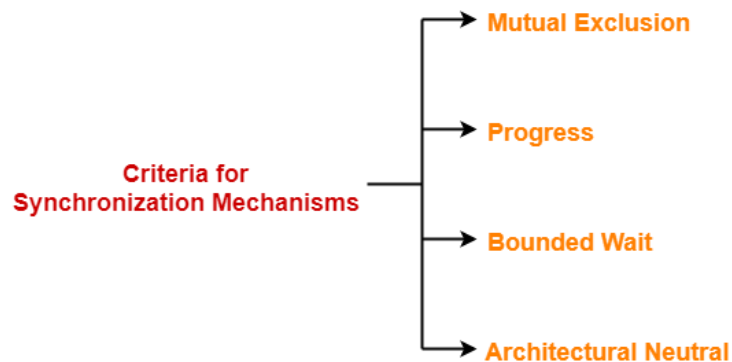
- It acts as a gateway for a process to enter inside the critical section.
- It ensures that only one process is present inside the critical section at any time.
- It does not allow any other process to enter inside the critical section if one process is already present inside it.

Exit Section

- It acts as an exit gate for a process to leave the critical section.
- When a process takes exit from the critical section, some changes are made so that other processes can enter inside the critical section.

Criteria For Synchronization Mechanisms

Any synchronization mechanism proposed to handle the critical section problem should meet the following criteria-



Any solution to the critical section problem must satisfy three requirements:

1. Mutual Exclusion
2. Progress
3. Bounded Wait

1. Mutual Exclusion

The mechanism must ensure-

- The processes access the critical section in a mutual exclusive manner.
- Only one process is present inside the critical section at any time.
- No other process can enter the critical section until the process already present inside it completes.

2. Progress

The mechanism must ensure-

- An entry of a process inside the critical section is not dependent on the entry of another process inside the critical section.
- A process can freely enter inside the critical section if there is no other process present inside it.
- A process enters the critical section only if it wants to enter.
- A process is not forced to enter inside the critical section if it does not want to enter.

3. Bounded Wait

The mechanism should ensure-

- The wait of a process to enter the critical section is bounded.
- A process gets to enter the critical section before its wait gets over.

Important Notes

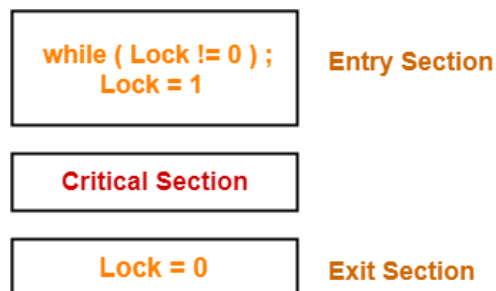
- Mutual Exclusion and Progress are the mandatory criteria.
- They must be fulfilled by all the synchronization mechanisms.
- Bounded waiting is the optional criteria.
- However, it is recommended to meet these criteria if possible.

Hardware Solution / Synchronization hardware:

1. Lock Variable:

- Lock variable is a synchronization mechanism.
- It uses a lock variable to provide the synchronization among the processes executing concurrently.
- However, it completely fails to provide the synchronization.

It is implemented as-



Initially, lock value is set to 0.

- Lock value = 0 means the critical section is currently vacant and no process is present inside it.
- Lock value = 1 means the critical section is currently occupied and a process is present inside it.

Working

This synchronization mechanism is supposed to work as explained in the following scenes-

Scene-01:

- Process P_0 arrives.
- It executes the $\text{lock} \neq 0$ instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop.
- The while loop condition breaks.
- It sets the lock value to 1 and enters the critical section.
- Now, even if process P_0 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P_0 completes and sets the lock value to 0.

Scene-02:

- Another process P_1 arrives.
- It executes the `lock!=0` instruction.
- Since lock value is set to 1, so it returns value 1 to the while loop.
- The returned value 1 does not break the while loop condition.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the lock value becomes 0 and its condition breaks.

Turn variable: Two processes P0 and P1.	
<p>P0 comes and sets turn=0</p> <p>Code: <code>while(1)</code> { <code>while(turn!=0);</code> //when <code>turn==0</code>, it breaks the while loop and enters C.S (This is the entry section which grants permission to enter C.S) critical section; <code>turn=1;</code> // gives turn to P1 remainder section; }</p>	<p>P1 comes and sets turn=1</p> <p>Code: <code>while(1)</code> { <code>while(turn!=1);</code> //when <code>turn ==1</code>, it breaks the while loop and enters C.S (This is the entry section which grants permission to enter C.S) critical section; <code>turn=0;</code> // gives turn to P0 remainder section; }</p>
In this case, each process gives turn to next process, so progress.	

Flag variable: Initially flag[0]= false and flag[1]= false. (No process is in Critical Section)	
P0 comes Code: <pre>while(1) { flag[0]=true; while (flag[1]); // if it is true means P1 is in C.S and do nothing. But in this case, flag[1]=false, hence while loop breaks and P0 enters C.S critical section; flag[0]=false; }</pre>	P1 comes Code: <pre>while(1) { flag[1]=true; while (flag[0]); critical section; flag[1]=false; }</pre>
In this case, deadlock may occur if both flag[0]=flag[1]=true	

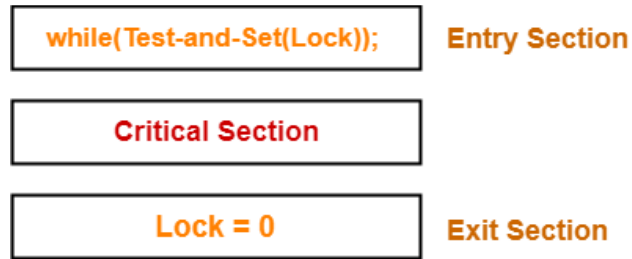
2. Test and Set Lock

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a ***test and set instruction*** to provide the synchronization among the processes executing concurrently.

Test-and-Set Instruction

- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

It is implemented as-



Initially, lock value is set to 0.

- Lock value = 0 means the critical section is currently vacant and no process is present inside it.
- Lock value = 1 means the critical section is currently occupied and a process is present inside it.

Working

This synchronization mechanism works as explained in the following scenes-

Scene-01:

- Process P_0 arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 1.
- The returned value 0 breaks the while loop condition.
- Process P_0 enters the critical section and executes.
- Now, even if process P_0 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P_0 completes and sets the lock value to 0.

Scene-02:

- Another process P_1 arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is now 1, so it returns value 1 to the while loop and sets the lock value to 1.
- The returned value 1 does not break the while loop condition.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the lock value becomes 0 and its condition breaks.

Characteristics

The characteristics of this synchronization mechanism are-

- It ensures mutual exclusion.
- It is deadlock free.
- It does not guarantee bounded waiting and may cause **starvation**.
- It suffers from **spin lock**. As it is a **busy waiting solution**, it keeps the CPU busy when the process is actually waiting.

Explanations

Point-01:

This synchronization mechanism guarantees mutual exclusion.

Explanation

- The success of the mechanism in providing mutual exclusion lies in the test-and-set instruction.
- Test-and-set instruction returns the old value of memory location (lock) and updates its value to 1 simultaneously.
- The fact that these two operations are performed as a single atomic operation ensures mutual exclusion.
- Preemption after reading the lock value was a major cause of failure of lock variable synchronization mechanism.
- Now, no preemption can occur immediately after reading the lock value.

Point-02:

This synchronization mechanism guarantees freedom from deadlock.

Explanation

- After arriving, process executes the test-and-set instruction which returns the value 0 to while loop and sets the lock value to 1.
- Now, no other process can enter the critical section until the process that has begin the test-and-set finishes executing the critical section.
- Other processes can enter only after the process that has begin the test-and-test finishes and set the lock value to 0.
- This prevents the occurrence of deadlock.

Point-03:

This synchronization mechanism does not guarantee bounded waiting.

Explanation

- This synchronization mechanism may cause a process to starve for the CPU.
- There might exist an unlucky process which when arrives to execute the critical section finds it busy.
- So, it keeps waiting in the while loop and eventually gets preempted.
- When it gets rescheduled and comes to execute the critical section, it finds another process executing the critical section.
- So, again, it keeps waiting in the while loop and eventually gets preempted.
- This may happen several times which causes that unlucky process to starve for the CPU

Point-04:

This synchronization mechanism suffers from spin lock where the execution of processes is blocked.

Explanation

Consider a scenario where-

- Priority scheduling algorithm is used for scheduling the processes.
- On arrival of a higher priority process, a lower priority process is preempted from the critical section.

Now,

- Higher priority process comes to execute the critical section.
- But synchronization mechanism does not allow it to enter the critical section before lower priority process completes.
- But lower priority process can not be executed before the higher priority process completes execution.
- Thus, the execution of both the processes is blocked.

Strict Alternation Approach

In strict alternation approach,

- Processes have to compulsorily enter the critical section alternately whether they want it or not.
- This is because if one process does not enter the critical section, then other process will never get a chance to execute again.

- It does ***not guarantee progress*** since it follows strict alternation approach.
- It ***ensures bounded waiting*** since processes are executed turn wise one by one and each process is guaranteed to get a chance.
- It ensures processes ***does not starve for the CPU***.
- It is ***deadlock free***.
- It is a ***busy waiting solution*** which keeps the CPU busy when the process is actually waiting.

Peterson's Solution [Software solution for synchronization]

This solution is for 2 processes to enter into critical section. This solution works for only 2 processes.

<p>It uses 2 variables turn and flag.</p> <p>Flag variable: Initially flag[0]= false and flag[1]= false. (No process is in Critical Section)</p>	
<p>P0 comes</p> <p><u>Code:</u> <pre>while(1) { flag[0]=true; turn =1; //gives turn to P1 while (turn==1 && flag[1]==True); // if it is true means P1 is in C.S and do nothing. critical section; flag[0]=false; }</pre></p>	<p>P1 comes</p> <p><u>Code:</u> <pre>while(1) { flag[1]=true; turn =0; //gives turn to P0 while (turn==0 && flag[0]==True); // if it is true means P0 is in C.S and do nothing. critical section; flag[1]=false; }</pre></p>
<p>In this case, deadlock will not occur.</p>	

Properties:

1. This is a *software mechanism*.
2. It used both TURN and FLAG variable.

It satisfies the following:

1. Mutual Exclusion: This condition is followed.
2. Progress: It is definitely followed as whichever process needs critical section, will make the flag value as true.

3. Bounded Waiting: This property is also followed as whichever process can make the TURN variable first, will get into critical section.

Disadvantage:

1. This solution **works for 2 processes**, but this solution is best scheme in user mode for critical section.
 2. This is also a **busy waiting solution** so CPU time is wasted. And because of that “SPIN LOCK” problem can come. And this problem can come in any of the busy waiting solution.
-

Semaphores: [Synchronization tool for multiple process]

A semaphore is a (integer) variable used to control access to a common resource by multiple processes in a concurrent system. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment.

[Just for understanding purpose -Library analogy: Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.

In the simplest implementation, the clerk at the front desk knows only the number of free rooms available. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario the front desk count-holder represents a **counting semaphore**, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, he is granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the current value of the semaphore is 0, he is forced to wait until a room is freed (when the count is increased from 0). If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO).] It's simply a way to limit the number of consumers for a specific resource. For example, to limit the number of simultaneous calls to a database in an application.

Semaphores are integer variables that are used to solve the critical section problem by using two **atomic operations**, *wait and signal* that are used for process synchronization.

This integer variable S is initialized to the **number of resources** present in the system. The `wait()` and `signal()` operation modifies the value of the semaphore S indivisibly or atomic, which means when a process is modifying the value of the semaphore, no other process can simultaneously modify the value of the semaphore.

The definitions of wait and signal are as follows:

1. Wait: $P(S)$

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
    while (S<=0);

    S--;
}
```

2. Signal : V(S)

The signal operation increments the value of its argument S.

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

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores.

1. Counting Semaphores

In **Counting Semaphore**, the value of semaphore S is initialized to the number of resources present in the system. Whenever a process wants to access the shared resources, it performs **wait()** operation on the semaphore which **decrements** the value of semaphore by one. When it releases the shared resource, it performs a **signal()** operation on the semaphore which **increments** the value of semaphore by one. When the semaphore count goes to **0**, it means **all resources are occupied** by the processes. If a process need to use a resource when semaphore count is 0, it executes wait() and get **blocked** until a process utilizing the shared resources releases it and the value of semaphore becomes greater than 0.

2. Binary Semaphores

In **Binary semaphore**, the value of semaphore ranges between 0 and 1. It is similar to mutex lock, but mutex is a locking mechanism whereas, the semaphore is a signalling mechanism. In binary semaphore, if a process wants to access the resource it performs wait() operation on the semaphore and **decrements** the value of semaphore from 1 to 0. When process releases the resource, it performs a **signal()** operation on the semaphore and increments its value to 1. If the value of the semaphore is 0 and a process want to access the resource it performs wait() operation and block itself till the current process utilizing the resources releases the resource.

3. [**Just for understanding- Analogy:** Consider a variable *A* and a boolean variable *S*. *A* is only accessed when *S* is marked true. Thus, *S* is a semaphore for *A*.]

Advantages of Semaphores

Some of the advantages of semaphores are as follows:

1. Semaphores **allow only one process** into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
2. There is **no resource wastage because of busy waiting** in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
3. Semaphores are implemented in the machine independent code of the microkernel. So they are **machine independent**.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows:

1. Semaphores are complicated so the **wait and signal operations must be implemented in the correct order to prevent deadlocks**.
 2. Semaphores may lead to a **priority inversion** where low priority processes may access the critical section first and high priority processes later.
-

Mutex: (Mutual Exclusion)

A **mutual exclusion** object (mutex) is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

The **Mutex is a locking mechanism** that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.

This is shown with the help of the following example:

```
wait (mutex);  
  
.....  
Critical Section  
.....  
signal (mutex);
```

Mutex vs Semaphore

Mutex and semaphore are kernel resources that provide synchronization services. Mutex and Semaphore both provide synchronization services but they are not the same though there is a similarity in their implementation.

The producer-consumer problem:

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length (4KB). A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Difference between Mutex and Semaphore:

The purpose of mutex and semaphore are different. Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

Problem with Mutex:

Spin lock: (Busy waiting)

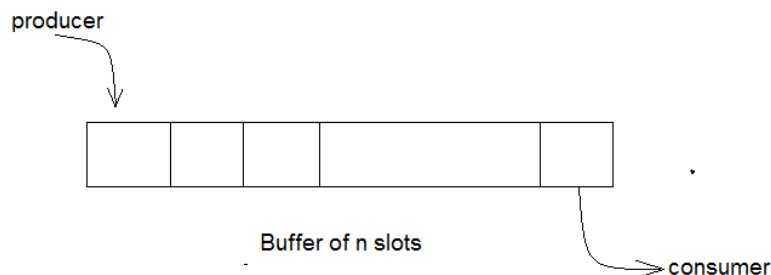
Say a resource is protected by a lock, a thread that wants access to the resource needs to acquire the lock first. If the lock is not available, the thread might repeatedly check if the lock has been freed. During this time the thread busy waits, checking for the lock, using CPU, but not doing any useful work. Such a lock is termed as a spin lock and the use of such a lock is a kind of **busy waiting**.

The Producer Consumer Problem:

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner/synchronized manner.

Here's a Solution: (Using semaphores and Mutex in Producer-Consumer Problem)

One solution of this problem is to **use semaphores**. The semaphores which will be used here are:

- **mutex**, which is used to acquire and release the lock.
- **empty**, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a **counting semaphore** whose initial value is **0**.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
    // wait until empty > 0 and then decrement 'empty'
    wait(empty);
    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
    signal(mutex);
    // increment 'full'
    signal(full);
}
while(TRUE)
```

- The producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
    // wait until full > 0 and then decrement 'full'
    wait(full);
    // acquire the lock
    wait(mutex);

    /* perform the remove operation in a slot */

    // release the lock
    signal(mutex);
    // increment 'empty'
    signal(empty);
}
while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Monitors:

Definition of Monitor

To overcome the timing errors that occurs while using semaphore for process synchronization, the researchers have introduced a high-level synchronization construct i.e. the **monitor type**. A monitor type is **an abstract data type** that is used for process synchronization.

A monitor is a **set of multiple** procedures/routines which are protected by a mutual exclusion lock. None of the routines in the monitor can be executed by a process/thread until that process/thread acquires the mutex lock. This means that only ONE process/thread can execute within the monitor at a time. Any other processes/threads must wait for the thread that's currently executing to give up control of the lock.

Monitors are supposed to be used in a multithreaded or multiprocess environment in which multiple threads/processes may call the monitor procedures at the same time asking for service. Thus, a monitor guarantees that ***at any moment at most one thread can be executing in a monitor!*** If a thread calls a monitor procedure, this thread will be blocked if there is another thread executing in the monitor. When the monitor becomes empty (*i.e.*, no thread is executing in it), one of the threads in the entry queue will be released and granted the permission to execute the called monitor procedure.

Being an abstract data type monitor type contains the **shared data variables** that are to be shared by all the processes and some programmer-defined **operations/procedures** that allow processes to execute in mutual exclusion within the monitor. A process **can not directly access** the shared data variable in the monitor; the process has to access it **through procedures** defined in the monitor which allow only one process to access the shared variables in a monitor at a time.

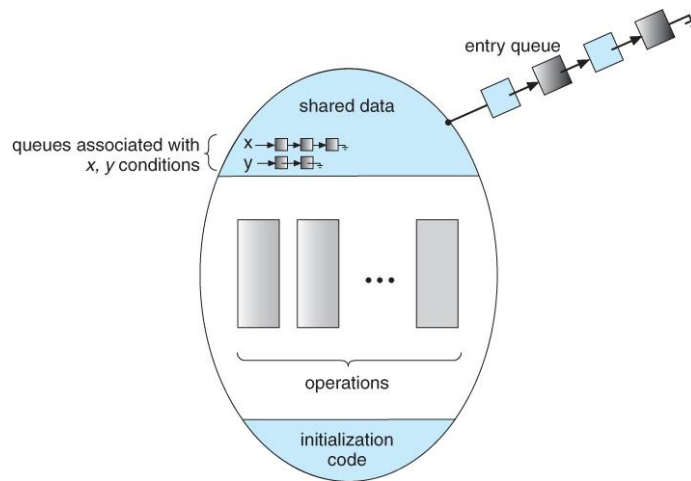
The syntax of monitor is as follows:

```
1. monitor monitor_name
2. {
3. //shared variable declarations
4. procedure P1 ( . . . ) {
5. }
6. procedure P2 ( . . . ) {
7. }
```

```

8. procedure Pn ( ... ) {
9. }
10. initialization code ( ... ) {
11. }
12. }

```



A monitor is a construct such as only one process is active at a time within the monitor. If other process tries to access the shared variable in monitor, it gets blocked and is lined up in the queue to get the access to shared data when previously accessing process releases it.

Conditional variables are introduced for additional synchronization mechanism. It allows a thread/process to block if a condition is not met. The conditional variable **allows a process to wait inside the monitor until some condition P holds true** [Eg: **while not(P) do skip**] and allows a waiting process to resume immediately when the other process releases the resources.

The **conditional variable** can invoke only two operation **wait()** and **signal()**. Where if a process **P** invokes a **wait()** operation it gets suspended in the monitor till other process **Q** invoke **signal()** operation i.e. a **signal()** operation invoked by a process resumes the suspended process.

Difference Between Semaphore and Monitor in OS:

Semaphore and Monitor both allow processes to access the shared resources in mutual exclusion. Both are the process synchronization tool. Instead, they are very different from each other.

BASIS FOR COMPARISON	SEMAPHORE	MONITOR
Basic	Semaphores is an integer variable S .	Monitor is an abstract data type .
Action	The value of Semaphore S indicates the number of shared resources available in the system	The Monitor type contains shared variables and the set of procedures that operate on the shared variable.
Access	When any process access the shared resources it perform wait() operation on S and when it releases the shared resources it performs signal() operation on S.	When any process wants to access the shared variables in the monitor, it needs to access it through the procedures .
Condition variable	Semaphore does not have condition variables .	Monitor has condition variables .

Monitors are easy to implement than semaphore, and there is little chance of mistake in monitor in comparison to semaphores.

Event Counter

It is another data structure that can be used for process synchronization. Like a semaphore, it has an integer count and a set of waiting process identifications. Unlike semaphores, the count variable only increases. It is similar to the "next customer number" used in systems where each customer takes a sequentially numbered ticket and waits for that number to be called.

[Just for understanding- Analogy:

Each customer entering the coffee shop takes a single "ticket" with a number from a "sequencer" on the counter. The ticket numbers dispensed by the sequencer are guaranteed to be unique and sequentially increasing. When a server is ready to serve the next customer, it calls out the "eventcount", the next highest unserved number previously dispensed by the sequencer. Each customer waits until the eventcount reaches the number on its ticket.]

Classical IPC Problems:

1. Bounded buffer problem (or Producer-Consumer problem)
2. Reader's & Writer's Problem
3. Dining Philosopher Problem

1. Bounded buffer problem (or Producer-Consumer problem)

Discussed earlier. Refer to that section

2. Reader's & Writer's Problem

In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading, or instead of reading it.

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows:

Reader Process

The code that defines the reader process is given below:

```
wait (mutex);
```

```
rc ++;  
if (rc == 1)  
    wait (wrt);  
signal(mutex);  
. READ THE OBJECT  
.  
wait(mutex);  
rc --;  
if (rc == 0)  
    signal (wrt);  
signal(mutex);
```

In the above code, **mutex** and **wrt** are **semaphores** that are initialized to **1**. Also, **rc** is a variable that is initialized to **0**. The **mutex semaphore ensures mutual exclusion** and **wrt handles the writing mechanism** and is common to the reader and writer process code.

The variable **rc** denotes the **number of readers** accessing the object. As soon as *rc* becomes 1, wait operation is used on *wrt*. This means that a writer cannot access the object anymore. After the read operation is done, *rc* is decremented. When *rc* becomes 0, signal operation is used on *wrt*. So a writer can access the object now.

Writer Process

The code that defines the writer process is given below:

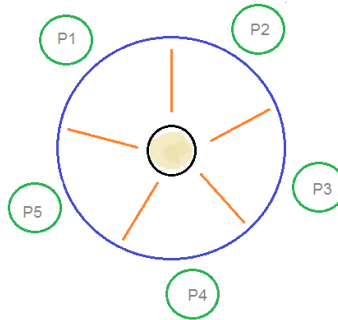
```
wait(wrt);  
.  
WRITE INTO THE OBJECT  
.  
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on *wrt*. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on *wrt*.

3. Dining Philosopher Problem

The dining philosophers problem is a classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes in a deadlock-free and starvation-free manner.

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a **semaphore to represent a chopstick**. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below:

```
semaphore chopstick [5];
```

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random **philosopher i** is given as follows:

```
do {  
    wait( chopstick[i] );  
    wait( chopstick[ (i+1) % 5] ); //after 5, the next chopstick is 1, hence mod function is used  
    .  
    .  
}
```

```
. EATING THE RICE
.
signal( chopstick[i] );
signal( chopstick[ (i+1) % 5] );
.
. THINKING
.
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher *i* has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher *i* has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows:

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

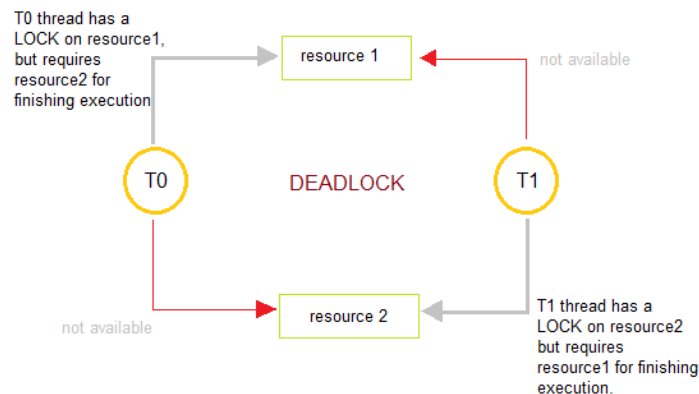
Deadlocks

In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

1. **Request** - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available.
2. **Use** - The process uses the resource, e.g. prints to the printer or reads from the file.
3. **Release** - The process relinquishes the resource. so that it becomes available for other processes.

What is a Deadlock?

Deadlock is a situation, where a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



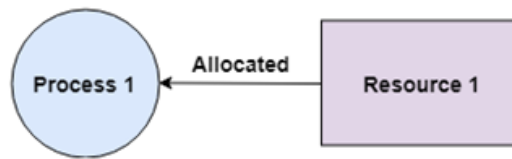
How to avoid Deadlocks

Deadlocks can be avoided by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause deadlock.

Necessary conditions for deadlock : (Characteristics of deadlock)

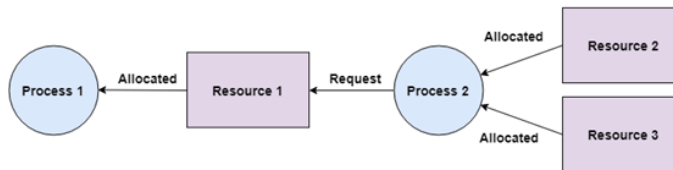
1. Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 which is held by Process 1 only. It implies, two processes cannot use the same resource at the same time.



2. Hold and Wait

A process is holding multiple resources and still requesting more resources held by other processes. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



3. No Preemption

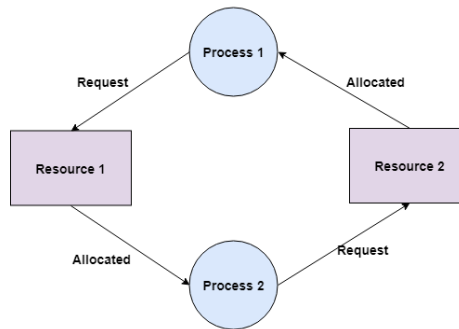
A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4. Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated

Resource 2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:
 - A set of resource categories, $\{ R_1, R_2, R_3, \dots, R_N \}$, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)
 - A set of processes, $\{ P_1, P_2, P_3, \dots, P_N \}$
 - **Request Edges** - A set of directed arcs from P_i to R_j , indicating that process P_i has requested R_j , and is currently waiting for that resource to become available.
 - **Assignment Edges** - A set of directed arcs from R_j to P_i indicating that resource R_j has been allocated to process P_i , and that P_i is currently holding resource R_j .
 - Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)
 - For example:

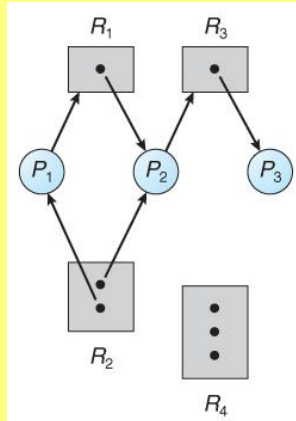


Figure - Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are *directed* graphs.) See the Figure above.
- If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains **more than one instance**, then the presence of a cycle in the resource-allocation graph indicates the *possibility of a deadlock*, but **does not guarantee one**. Consider, for example, Figure below:

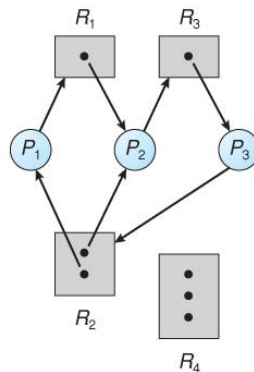


Figure - Resource allocation graph with a deadlock

Strategies or methods for handling Deadlock

1. Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

2. Deadlock avoidance

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.

In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

3. Deadlock detection and recovery

This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

Deadlock Prevention:

If we violate one of the four necessary conditions, and don't let them occur together then we can prevent the deadlock.

Let's see how to prevent each of the conditions.

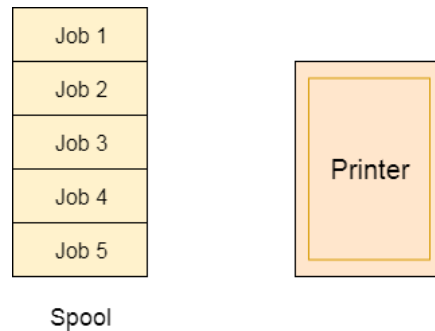
1. Mutual Exclusion

Mutual exclusion is where a resource can never be used by more than one process simultaneously, which is fair enough but that is the main reason behind the deadlock. If a resource can be used by more than one process at the same time, then the process would never wait for any resource.

Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one

of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This **cannot be applied to every resource**.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We **cannot force a resource to be used by more than one process** at the same time since it will not be fair enough and some serious **problems may arise in the performance**. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process **either doesn't hold any resource or doesn't wait**. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a **process can't determine necessary resources initially**.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. **Practically not possible.**
2. Possibility of getting **starved** will be increasing due to the fact that **some process may hold a resource for a very long time.**

3. No Preemption





Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock. Preempt resources from process when resources required by other high priority process.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now will become inconsistent.

4. Circular Wait

Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.

For Example, if P1 process is allocated R5 resource, next time if P1 ask for R4, R3 which are lesser than R5 such request will not be granted. Only request for resources more than R5 will be granted.

Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	
Hold and Wait	Request for all the resources initially	
No Preemption	Snatch all the resources	
Circular Wait	Assign priority to each resources and order resources numerically	

Among all the methods, violating Circular wait is the only approach that can be implemented practically.

Deadlock Avoidance:

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.

A state of the system is called **safe** if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfill the request of all processes then the state of the system is called **unsafe**.

The key of **Deadlock avoidance** approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

Deadlock avoidance algorithms:

- Resource allocation algorithm
- Banker's Algorithm

i. Resource-Allocation Graph Algorithm:

- If resource categories have only single instances of their resources, then deadlock states can be **detected by cycles in the resource-allocation graphs**.
- In this case, unsafe states can be recognized and avoided by *augmenting the resource-allocation graph with **claim edges***, noted by ***dashed lines***, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established

claim edges, and claim edges cannot be added to any process that is currently holding resources.)

- When a process makes a request, the **claim edge $P_i \rightarrow R_j$ is converted to a request edge**. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by **denying requests that would produce cycles in the resource-allocation graph**, taking claim edges into effect.
- Consider for example what happens when process P_2 requests resource R_2 :

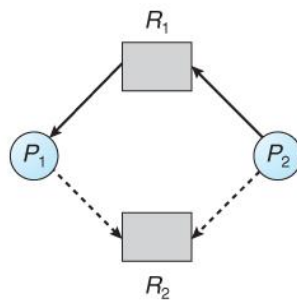


Figure 7.7 - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

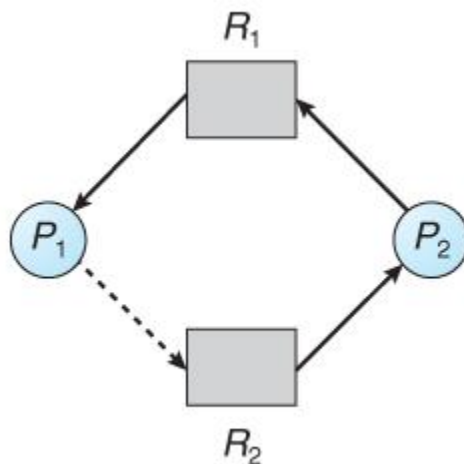


Figure 7.8 - An unsafe state in a resource allocation graph

ii. Banker's Algorithm:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker's Algorithm:

Let '**n**' be the number of processes in the system and '**m**' be the number of resource types.

Available :

- It is a 1-d array of size '**m**' indicating the number of available resources of each type.
- $\text{Available}[j] = k$ means there are '**k**' instances of resource type **R_j**

Max :

- It is a 2-d array of size '**n*m**' that defines the maximum demand of each process in a system.
- $\text{Max}[i, j] = k$ means process **P_i** may request at most '**k**' instances of resource type **R_j**.

Allocation :

- It is a 2-d array of size '**n*m**' that defines the number of resources of each type currently allocated to each process.
- $\text{Allocation}[i, j] = k$ means process **P_i** is currently allocated '**k**' instances of resource type **R_j**

Need :

- It is a 2-d array of size '**n*m**' that indicates the remaining resource need of each process.
- $\text{Need}[i, j] = k$ means process **P_i** currently need '**k**' instances of resource type **R_j** for its execution.
- $\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$

Allocation_i specifies the resources currently allocated to process **P_i** and Need_i specifies the additional resources that process **P_i** may still request to complete its task.

Banker's algorithm consist of Safety algorithm and Resource request algorithm

i. Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let *Work* and *Finish* be vectors of length '*m*' and '*n*' respectively.

Initialize: $\text{Work} = \text{Available}$

$\text{Finish}[i] = \text{false}$; for $i = 1, 2, 3, 4, \dots, n$

2) Find an *i* such that both

a) $\text{Finish}[i] = \text{false}$

b) $Need_i \leq Work$
 if no such i exists goto step (4)
 3) $Work = Work + Allocation[i]$
 $Finish[i] = true$
 goto step (2)
 4) if $Finish[i] = true$ for all i
 then the system is in a safe state

ii.Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i . $Request_i[j] = k$ means process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1) If $Request_i \leq Need_i$
 Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
 2) If $Request_i \leq Available$
 Goto step (3); otherwise, P_i must wait, since the resources are not available.
 3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
 $Available = Available - Request_i$
 $Allocation_i = Allocation_i + Request_i$
 $Need_i = Need_i - Request_i$

Example:

Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	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			

Question1. What will be the content of the Need matrix?

$Need[i, j] = Max[i, j] - Allocation[i, j]$

So, the content of Need Matrix is:

Process	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

Question 2. Is the system in safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,

step 1: First, consider P₀. Check whether the Need of P₀ (7,4,3) \leq Currently Available (3,3,2). This fails.

step 2: Next, consider P₁. Check whether the Need of P₁ (1,2,2) \leq Currently Available (3,3,2). This is true. Hence, P₁ can be allocated with the resources. After P₁ finishes its execution, it releases the resources allocated to it (2,0,0). Hence, the currently available = (3,3,2) + (2,0,0) = (5,3,2)

step 3: Next, consider P₀ again. Check whether the Need of P₀ (7,4,3) \leq Currently Available (5,3,2). This fails.

step 4: Next, consider P₂. Check whether the Need of P₂ (6,0,0) \leq Currently Available (5,3,2). This fails.

step 5: Next, consider P₃. Check whether the Need of P₃ (0,1,1) \leq Currently Available (5,3,2). This is true. Hence, P₃ can be allocated with the resources. After P₃ finishes its execution, it releases the resources allocated to it (2,1,1). Hence, the currently available = (5,3,2) + (2,1,1) = (7,4,3)

step 6: Next, consider P₀ again. Check whether the Need of P₀ (7,4,3) \leq Currently Available (7,4,3). This is true. Hence, P₀ can be allocated with the resources. After P₀ finishes its execution, it releases the resources allocated to it (0,1,0). Hence, the currently available = (7,4,3) + (0,1,0) = (7,5,3)

step 7: Next, consider P₂ again. Check whether the Need of P₂ (6,0,0) \leq Currently Available (7,5,3). This is true. Hence, P₂ can be allocated with the resources. After P₂ finishes its

execution, it releases the resources allocated to it (3,0,2). Hence, the currently available= (7,5,3) + (3,0,2)= (10,5,5)

step 8: Next, consider P4. Check whether the Need of P4 (4,3,1) \leq Currently Available (10,5,5). This is true. Hence, P4 can be allocated with the resources. After P4 finishes its execution, it releases the resources allocated to it (0,0,2). Hence, the currently available= (10,5,5) + (0,0,2)= (10,5,7).

The final (10,5,7) is same as the original number of resources (10,5,7)

The safe sequence is $\langle P1, P3, P0, P2, P4 \rangle$.

Deadlock detection and recovery

Deadlock Detection:

If a system does not employ some mechanism that ensures deadlock freedom, then a detection and recovery scheme must be used.

An algorithm that examines the state of the system is **invoked periodically** to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock.

To do so, the system must:

- Maintain information about the current allocation of data items to transactions.
- Provide an algorithm that to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection:

wait-for graph:

A simple way to detect a state of deadlock is with the help of **wait-for graph**. This graph is constructed and maintained by the system. One node is created in the wait-for graph for each transaction that is currently executing.

Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j , a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph.

When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph.

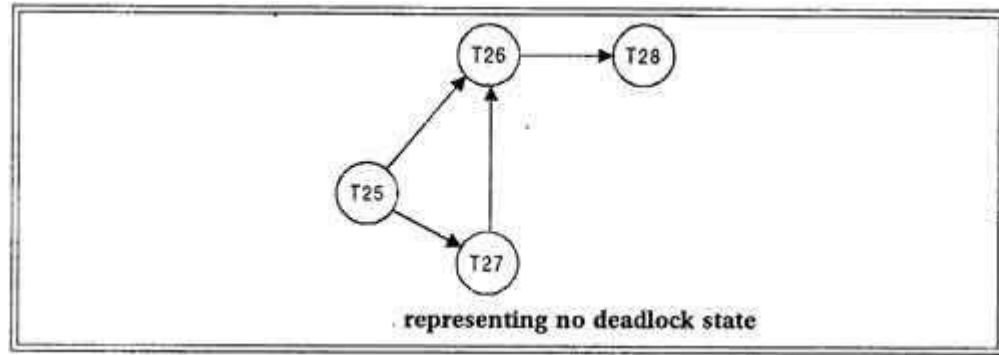
We have a state of **deadlock if and only if the wait-for graph has a cycle**. Then each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait for graph, and periodically to invoke **an algorithm that searches for a cycle in the graph**.

To illustrate these concepts, consider the following wait-for graph in figure. Here:

Transaction T_{25} is waiting for transactions T_{26} and T_{27} .

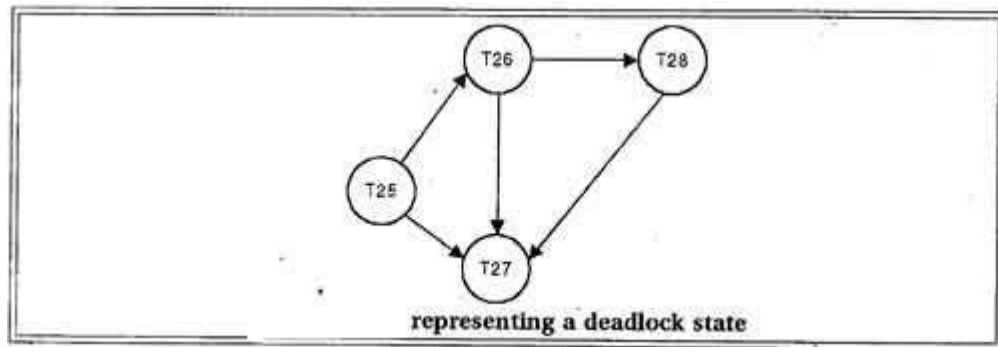
Transactions T_{27} is waiting for transaction T_{26} .

Transaction T_{26} is waiting for transaction T_{28} .



This wait-for graph has **no cycle**, so there is **no deadlock state**.

Suppose now that transaction T28 is requesting an item held by T27. Then the edge T28-->T27 is added to the wait -for graph, resulting in a new system state as shown in figure.



This time the graph contains the cycle.

T26-->T28-->T27-->T26

It means that transactions T26, T27 and T28 are all deadlocked.

Invoking the deadlock detection algorithm:

The invoking of deadlock detection algorithm depends on two factors:

- How often does a deadlock occur?
- How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

Recovery from Deadlock:

- There are three basic approaches to recover from deadlock:
 1. Inform the system operator, and allow him to take manual intervention.
 2. Terminate one or more processes involved in the deadlock
 3. Preempt resources.

1. Process Termination

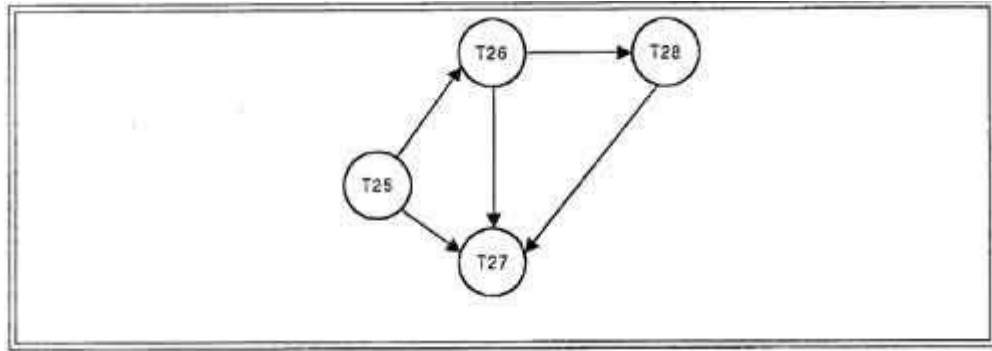
- Two basic approaches, both of which recover resources allocated to terminated processes:
 - ***Terminate all processes*** involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
 - ***Terminate processes one by one until the deadlock is broken***. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
 1. Process priorities
 2. How long the process has been running, and how close it is to finishing
 3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
 4. How many more resources does the process need to complete.
 5. How many processes will need to be terminated

2. Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
 1. **Selecting a victim** - When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to ***roll back one or more transactions to break the deadlock***. Choosing which transaction to abort is known as ***Victim Selection***.

Choice of Deadlock victim

In below wait-for graph transactions T26, T28 and T27 are deadlocked. In order to remove deadlock one of the transaction out of these three transactions must be roll backed.



We should ***roll back those transactions that will incur the minimum cost.*** When a deadlock is detected, the choice of which transaction to abort can be made using following ***criteria:***

- The transaction which have the ***fewest locks***
 - The transaction that has ***done the least work***
 - The transaction that is ***farthest from completion***
2. **Rollback** - Once we have decided that a particular transaction must be rolled back, we must determine **how far this transaction should be rolled back**. The simplest solution is a **total rollback; Abort the transaction and then restart it**. However it is more effective to roll back the transaction only as far as necessary to break the deadlock. But this method requires the system to maintain additional information about the state of all the running system.
 3. **Starvation** - In a system where the selection of victims is based primarily on cost factors, it may happen that the **same transaction is always picked as a victim**. As a result this transaction never completes so, a solution may be selecting a victim only a (small) finite number of times. The most common solution is to include the **number of rollbacks** in the cost factor.