Sweety Mam

Unit-3

Sequence control → control of execution of operation both primitive & user defined.

Sequence control in programming provide structure in a programming language provide framework within which operations and data are combined into programs and set of programs.

Types **Implicit Sequence control** (Defined by programming lang itself) :- Sequence control structure are those defined by language (default) to be in effect unless modified by programmer though some explicit structure.

[The built-in execution model in which the statements execute in the order in which they appear]

e.g $a+b*c$ → First we multiply then add.

Explicit Sequence control → The programmer uses staments to change the order of execution. e.g condition, loops.

e.g goto X

e.g (1) if Y goto X → Transfer control to the Statement labeled X if Y is true

(e.g 2)
if (condition)
Statement 1
else Statement 2

Levels of Sequence control

(1) **Expression** :- How data are manipulated using precedence rules and parantheses.

(2) **Statements** → conditional & iteration change the Sequential execution.

(3) **Declarative model** :- An execution model that does not depend on the order of the statements in source program e.g Prolog

(4) **Subprograms** :- Transfer control from one program to another program.

Sequence control within Expression:- Sequence control within expression determine the order of operations within this expression i.e. sequence of performing the operations

Hierarchy of operation → operators are that are occurs in expression are placed in hierarchy

Precendance & Associativity

→ Precendance concerns the order of applying operations e.g $a \times b + c$ → * Evaluated first then +

Associativity deals with the order of operations of same precedence. e.g $a - b - c$ → Left to Right associativity is most common implicit Rule So a-b-c treated as (a-b)-c

Precedence & Associativity are defined when lang. is defined with the semantic rules for expressions.

→ In case of exponentiation right to left → $a \uparrow b \uparrow c$ treated as $a \uparrow (b \uparrow c)$

Arithmetic operations

Linear representation of the expression tree

Prefix Notation → operator is prefixed to operands.

Postfix Notation → Also known as Reversed Polish Notation operator is postfix to operands.

Infix Notation →

Infix Notation	Prefix Notation	Postfix Notation
a+b	+ab	ab+
(a+b)*c	*+abc	ab+c*
a*(b+c)	*a+bc	abc+*

Infix Expression	Equivalent Prefix Expression	Equivalent Postfix expression	Order of Evaluation
a+b*c	+a*bc	abc*+	* will be executed first then +
(a+b)*c	*+abc	ab+c*	+ is Evaluated first due to parenthesis

Sequence control in expression

Rules

$19 + (10 + 3 * 7) / 20 - 3 ** 2$ →

⇒ Rules of operator precedence

Operator	Precedence & Associativity
( )	Evaluated first. If nested (embedded) innermost first. If on same level, left to Right
**	
*, /, %	Evaluated second. If there are several evaluated left to Right
+, -	Evaluated third. If there are several, evaluated left to Right.
=	Evaluated last, Right to left.

$3 + 5 * 4$ → 20, 23 (1st, 2nd)

$8 / (5 - 2)$ → 3, 2

$8 + (12 \% 5)$ → 2, 10

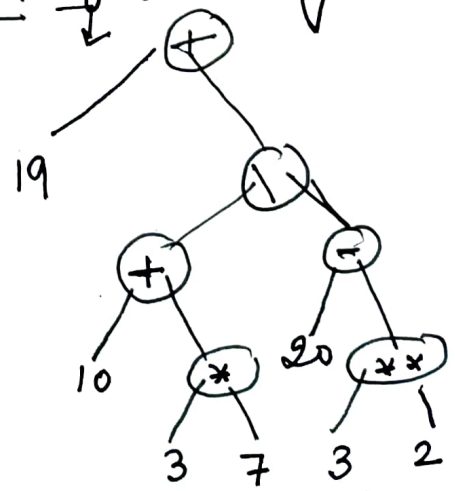
$5 * 2 \% (7 - 4)$ → 1st: 3; 2nd: 10, 1

Tree Representation of following Expression

Eg: $19 + (10 + 3 * 7) / (20 - 3 ** 2)$

$19 + (10 + 21) / (20 - 9)$

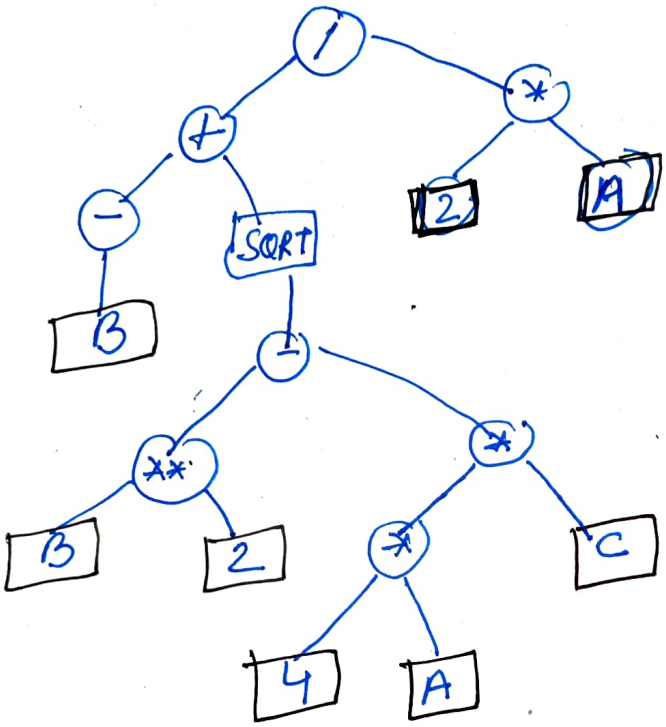
$19 + 31 / 11 \Rightarrow 19 + 2 = 21$



$$root = \frac{-B \pm \sqrt{B^2 - 4 \times A \times C}}{2 \times A}$$

(we can write it

root = (-B+ SQRT(B**2-4*A*C)) /2*A



Tree Representation

Sequence Control between Statements :- Basic mechanisms in use for controlling the sequence in which individual statements are executed within a program

Basic Statements These statements are written in a single line. Basic statements include assignment statements, subprogram calls, and input & output statements.

(i) **Assignment Statement** → The primary purpose of an assignment is to assign to the l-value of a data object, the r-value of some expression. The syntax for an explicit assignment varies widely :

	language
A := B	Pascal, Ada
A = B	C, FORTRAN, PL/I, Prolog
MOVE B TO A	COBOL
A ← B	APL
(SETQ A B)	LISP

C have several form of assignment :

A = B (Assign r-value of B to l-value of A, return r-value)

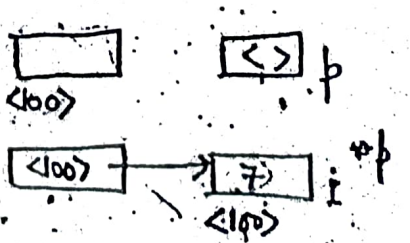
A += B (-=) Increment (or decrement) A by B (A = A+B or A = A-B) return new value

++A (--A) Increment (or decrement) A then return new value (eg. A = A+1, return r-value of A)

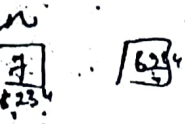
A++ (A--) Return value of A then increment (decrement) A (return r-value, then A = A+1).

In C, the unary * operator is an indirection operator that makes the r-value of a variable behave as if it were an l-value :

```
int i, *p;
p = &i;
*p = 7;
```

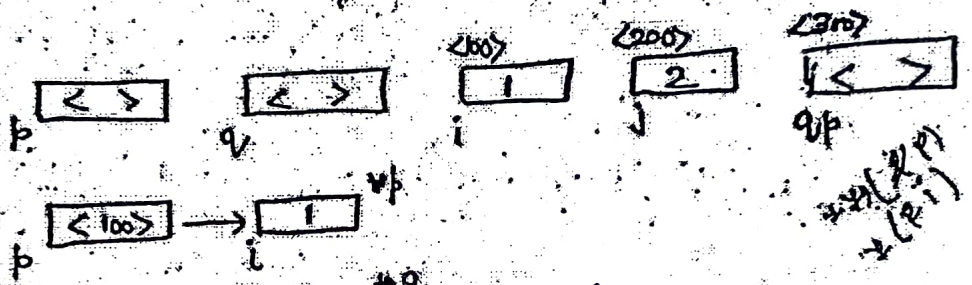


The unary & operator is an address operator that converts an l-value into r-value.

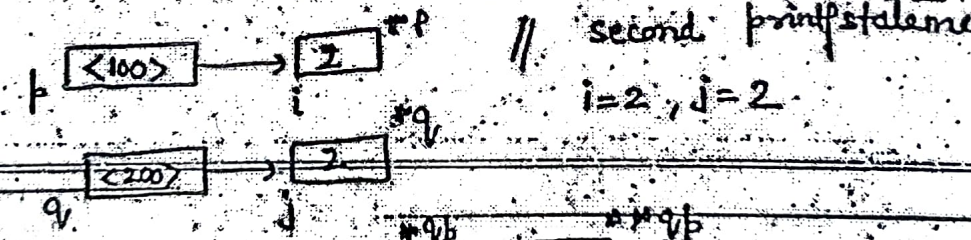


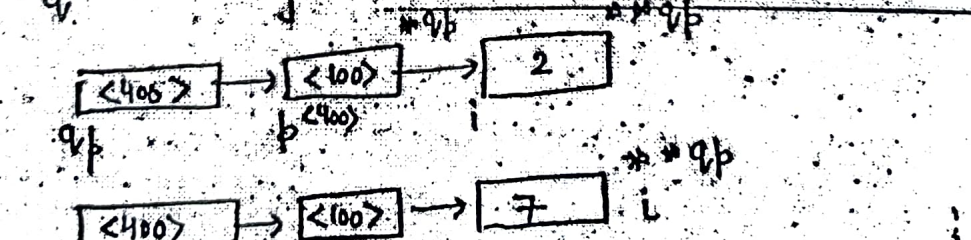
*(&i) = 7

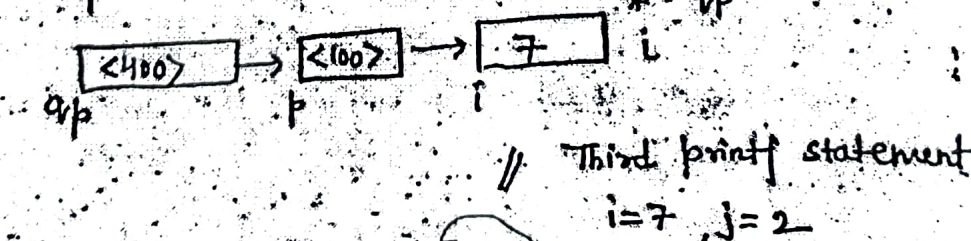
```
main()
{ int *p, *q, i, j;          // qp is pointer to pointer to int
  int **qp;
  i=1; j=2; printf("%d%d", i, j);     // i=1, j=2
  p = &i;
  q = &j;
  *p = *q; printf("%d%d", i, j);      // i=2, j=2
  qp = &p;
  **qp = 7;
  printf("%d%d", i, j);               // i=7, j=2
}
```



// second printf statement
i=2, j=2







// Third printf statement
i=7, j=2

**qq = 7
*(&p) **(&p)
&i = 7 *(&i) = 7

Three main forms of statement - [illegible]
usually distinguished:

① Composition :→ Statements may be placed in a textual sequence so that they are executed in order.

② Alternation → Two sequences of statements may form alternatives so that one or the other sequence is executed, but not both.

③ Iteration → A sequence of statements may be executed repeatedly, zero or more times.

Compound statements → A compound statement is a sequence of statements that may be treated as a single statement in the construction of larger statements. often a compound statement is written:

```
begin
  ...   - sequence of statements (one or more)
end
```

In C, C++, Java, it is written simply as { ... }.

Within the compound statement, statements are written in the order in which they are to be executed.

(i) Conditional statements → A conditional statement is one that expresses alternation of two or more statements, or optional execution of a single statement. The choice of alternative is controlled by a test on some condition, usually written as an expression involving relational & boolean operations. The most common forms of conditional statement are the if & case statements.

Single branch → if condition then statement endif

Two branch → if condition then $statement_1$ else $statement_2$ endif

Multibranch →
```
if condition1 then statement1
   elsif condition2 then statement2
   elsif conditionn then statementn
   else statementn+1 endif
```

(ii) Iteration Statements :- It provides the basic mechanism for repeated calculations in most programs. The basic structure of an iteration statement consists of a head and a body. The head controls the No. of times that the body will be executed, whereas the body is usually a (compound) statement that provides the action of the statement.

- Simple repetition :- Here body is to be executed some fixed Number of times

The COBOL PERFORM is typical the example of this

perform body k times

- Repeation while condition holds :-

while test do body

The test expression is reevaluated each time after the body has been executed.

- Repeatition while incrementing a counter :- Here an initial value, final value & increment are specified in the head & the body is executed repeatedly until the final value is reached.

foreg in C:

```
for (a=1 ; a<=10 ; a++)
{
  - - -
}
```

- Data based repetition → Sometimes the format of the data determines the repetition counter. Perl, for example, has a foreach construct:

foreach $x (@arrayitem) { - - }

for each pass through the loop, scalar variable $x will have a value equal to the next element of array @arrayitem. The size of the array determines how many times the program loops.

Indefinite Repetition :- foreg in C

for (expression$_1$; expression$_2$; expression$_3$) {body}

Here expression$_1$ is the initial value,

$expression_2$ is the repeat condition, & $expression_3$ is the increment. All of these expressions are optional, allowing for much flexibility in C iterations. Some C sample iteration loops can be specified as follows :

Simple counter from 1 to 10 : for (i=1 ; i<=10 ; i++) {body}

Infinite loop : for (; ;) {body}

Counter with exit condition : for (i=1 ; i<=100 && NOT End File ; i++) {body}

Disadvantages of goto statement

① Lack of hierarchical program structure

② Order of statements in the program text need not correspond to the order of execution

③ Group of statements may serve multiple purposes.

Sequence Control for Subprogram :→ There are two types of (12)
Subprogram. These are
① Simple Call-return
② Recursive calls.

Simple Call return Subprogram → A program is composed of a single main program, which during execution may call various subprograms, which in turn may each call other sub-subprograms & so forth, to any depth. Each subprogram at some point is expected to terminate its execution & return control to the program that called it. During execution of a subprogram, execution of the calling program is temporarily halted. When execution of the subprogram is completed, execution of the calling program resumes at the point immediately following the call of the subprogram.
Simple call return supports to the following properties :-

① No recursive calls → There should not be recursion in simple call-return

② Explicit calls → Each subprogram should be explicitly called.

③ Complete execution → Each subprogram should be executed completely. ie, from its beginning to its logical end.

④ Immediate control transfer → Immediate control transfer should be made as one subprogram can call to another subprogram.

⑤ Single Execution Sequence → Execution proceeds in single sequence from calling program to called subprogram & back to calling program. There should be a single execution sequence for given condition.

Implementation for Simple call return Control Structure :-

For Implementation we need following things :-

(1) There is a distinction b/w definition & subprogram activation. The definition is written part of the subprogram as we see. An activation is created each time a subprogram is called.

(2) An activation is implemented in two ways : a code segment containing the executable code and constants, & an activation record containing local data, parameters, & various other data items.

(3) The code segment is invariant during execution. It is created by the translator & stored statically in memory. During execution, it is used but never modified. Every activation of the subprogram uses the same code segment.

(4) The Activation record is created new each time the subprogram is called, and it is destroyed when the subprogram returns. While the subprogram is executing, the contents of the activation record are constantly changing as assignments are made to local variables & other data objects.

To keep track of the point at which a program is being executed, we need two pieces of data which we consider as storage in two system-defined pointer variables

(1) Current Instruction pointer → At any point during execution, there is some instruction in some code segment that is currently being executed by the h/w or s/w interpreter. This instruction is termed the current instruction, & a pointer to it is maintained in the variable called the current-instruction pointer (CIP).

(2) Current Environment Pointer :- Because all activations of the same subprogram use the same code segment, it is not enough simply to know the current instruction being executed; a pointer to the activation record being used is also needed. The pointer to the current activation record is maintained during execution in the variable coe known the current-environment pointer (CEP).

Working :→ An activation record for the main program is created. The CEP is assigned a pointer to it. The CIP is assigned a pointer to the first instruction in the code segment for the main program.

When a subprogram call instruction is reached, an activation record for the subprogram is created & a pointer to it is assigned to the CEP. The CIP is assigned a pointer to the first instruction of the code segment for the subprogram. The interpreter continues from that point, executing instructions in the subprogram. If the subprogram calls another subprogram, new assignments are made to set the CIP & CEP for the activation of that subprogram.

To return correctly from a subprogram call, the values of the CIP & CEP must be saved somewhere by the subprogram call instruction before the new values are assigned. When a return instruction is reached that terminates an activation of a subprogram, the old values of the CIP & CEP that were saved when the subprogram was called must be retrieved & reinstated. This reinstatement of the old values is all that is necessary to return control to the correct activation of the calling subprogram at the correct place so that execution of that subprogram may continue.

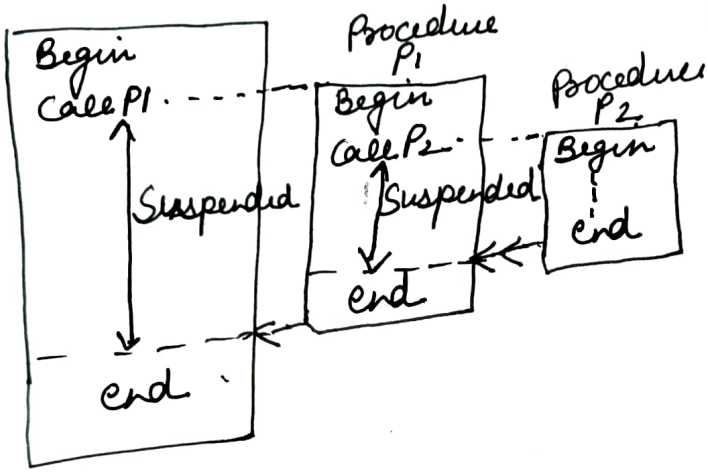
Sub Sequence / Subprogram control

↳ Simple call Return Subprogram
↳ Recursive Subprogram →

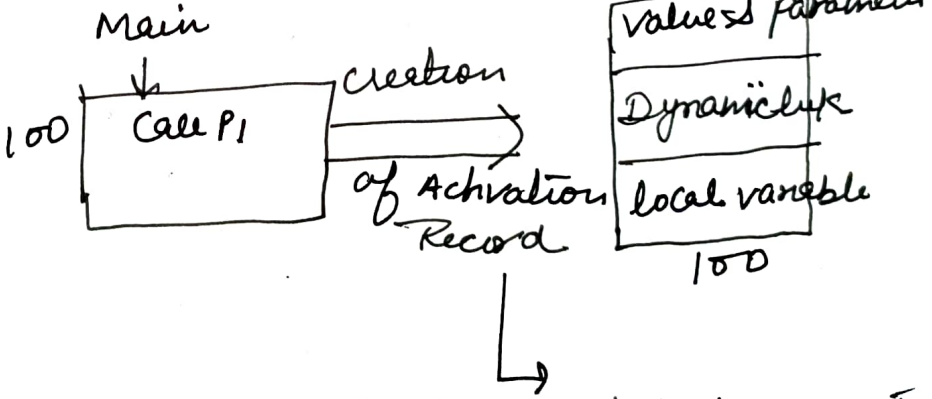
Activation Record { CIP, CEP } dynamic link

Subprogram control
→ Interaction among Subprogram
→ How Subprogram pass data among themselves

Main Program



Implementation



CIP → Address of next Statement to be executed
CEP → Pointer to Activation Record

Subprogram Activation

On call Instruction :- An Activation Record is created
→ CIP & CEP are saved in created Activation Record as return point

On return :- old value of CIP & CEP are Retrieved

Explain a) Recursive subprogram. (14)
b) LD - routines.

Recursive Subprogram :→ Recursion, in the form of recursive program calls, is one of the most important sequence-control notions in programming. A recursive function is a function whose definition is based upon itself. It means that a recursive function is that function which has a statement calling itself. A recursive function must have following properties :

- There must be a base criteria that is a condition for which the function doesn't call itself. It is necessary to avoid infinite recursion.
- Each time function call itself, it must be closer to base criteria. It means that with each recursive call, recursion moves closer to termination.

Specification :- The only difference b/w a recursive call and an ordinary call is that the recursive call creates a second activation of the subprogram during the lifetime of the first activation. If the second activation leads to another recursive call, then three activations may exist simultaneously, & so on. In general, if the execution of the program results in a chain of a first call of subprogram A followed by K recursive calls that occur before any return is made, then K+1 activations of A will exist at the point just before return from the k^{th} recursive call. The only new element introduced by recursion is the multiple activations of the same subprogram that all exist simultaneously at some point during execution.

Implementation :→ Because of the possibility of multiple activations, we need both the CEP & CEP pointers. At the time of each subprogram call, a new activation record is created, which is subsequently destroyed upon return.

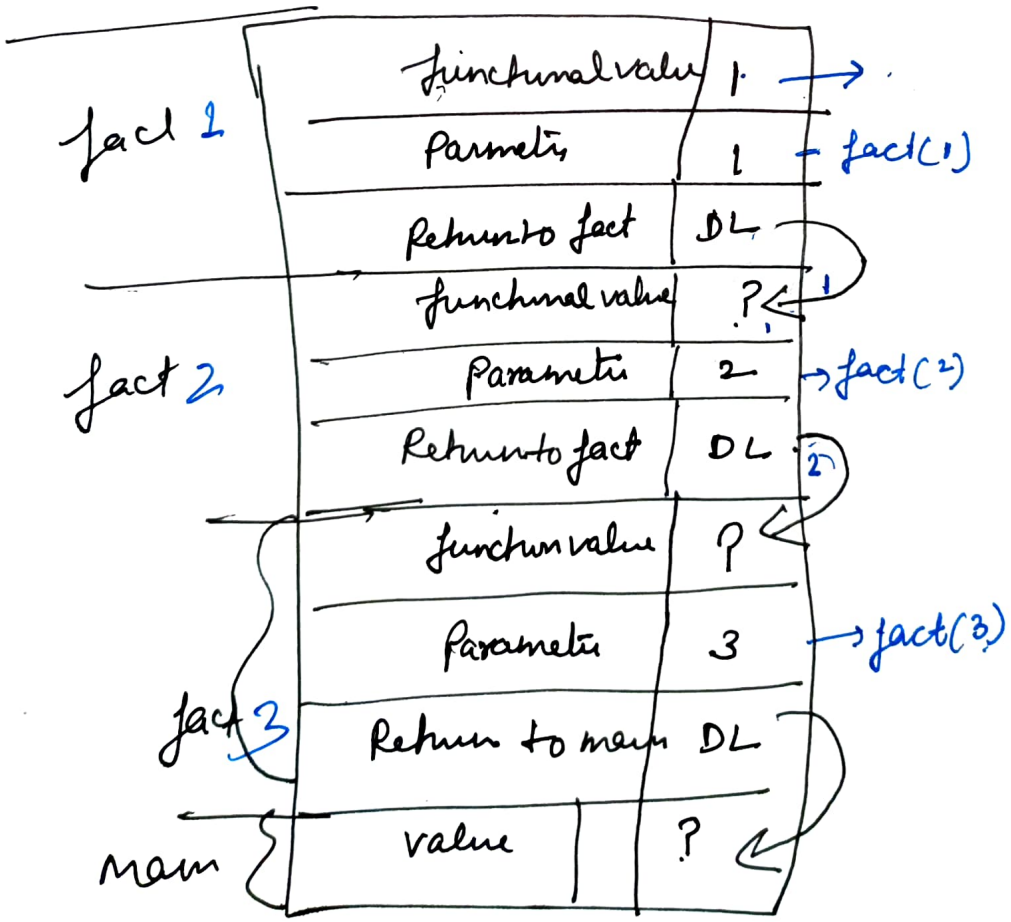
Each activation record contains a return point to store the values of the (ip, ep) pair used by call & return. The ep values form a linked list that links together the activation records on the central stack in the order of their creation. From the CEP pointer, the top activation record in the central stack is reached. From the ep value in

Recursive Subprograms

```
int fact (int n)
{
if (n<=1) return 1;
else
  return (n* fact(n-1));
}
void main()
{ int value;
value = fact(3);
printf ("%d", value);
}
```

3

3 * fact(2);
3*2 fact(1)
3*2*1



Exception :-

Exception is an unwanted event that interrupts the normal flow of the program.
when Exception occurs program execution gets terminated

e.g Array Element with Subscript out of bounds.
Arithmetic operation overflow.
class Not Found Exception

Exception handling :- is Block of code that is executed if an exception occurs during execution of code.

@ **Raising An Exception** :- Exception are raised when the program is syntactically correct, but the code resulted in an error.

Exception handling using Try & Catch block

Try Block :- → allows you to define a block of code to be tested for errors while it being executed.
It contain set of statements where exception can occurs.

Catch Block → allows you to define a block of code to be executed, if an error occurs in the try block.
It handle the exceptions.

Syntax

```
try
{
    Block of code to try (that may cause an Error)
}
catch (Exception e)
{
    Block of code to handle Error
}
```

If an exception occurs in try block then control of execution is passed to corresponding catch block

```
try
{
num1 = 0;
num2 = 62 / num1;
System.out.println(num2);
}
Catch (Arithmetic Exception e)
{
System.out.println("you should not divide a number by zero");
}
```

Try Block:- Block of statements where we expecting the Exception

Catch Block:- Block of statement handles exception raised in Try Block

Why we need Exception handling?

Avoiding abnormal termination of Programs. bcoz when exception take place in 'Try Block' it is handled in 'catch Block' so that rest of code of program executed without termination.

Types of Exceptions

↳ Checked Exception:- Compile time Exception

↳ Unchecked Exception:- Run time Exception

(a) Arithmetic Exception:- Divide by zero eg int a = 8/0;

(b) Null pointer Exception:- e.g String Str = null;
System.out.println(Str.length());

(c) Number Format Exception
↳ String Str = "hello";
int num = Integer.ParseInt(Str);

(d) Array Index out of Bound Exception:- int a[] = new int[5];
If we want to access a[7] = 15;

Co-routines :→ Coroutines allow subprograms to return to their calling program before completion of execution. When a coroutine receives control from another subprogram, it executes partially & then is suspended when it returns control. At a later point, the calling program may resume execution of the coroutine from the point at which execution was previously suspended.

If A calls subprogram B as a coroutine, B executes a while & returns control to A, just as any ordinary subprogram would do. When A again passes control to B via a resume B, B again executes awhile & returns control to A, just as an ordinary subprogram. Thus, to A, B appears as an ordinary subprogram. However, the situation is similar when viewed from subprogram B. B, in the middle of execution, resumes execution of A. A executes awhile & returns control to B. B continues execution awhile & returns control to A. A executes awhile & return control to B. From subprogram B, A appears much like an ordinary subprogram. The name coroutine derives from this symmetry. Two programs appears more as equals - two subprograms swapping control back & forth as each executes, with neither controlling the other.

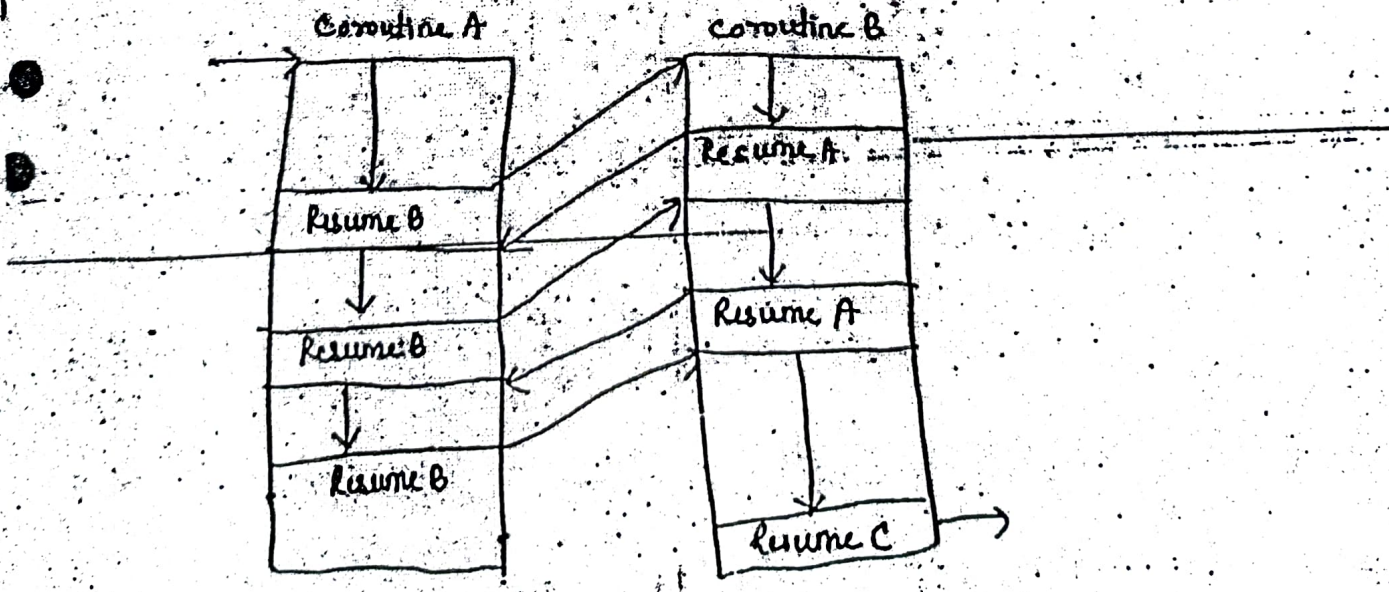


fig :- Control transfer b/w coroutines

Coroutine :- It is special kind of Subprogram.
It have multiple entry points which are controlled by themselves. They also have mean to maintain their status between various activation.

- only one coroutine executes at a given time.
- coroutines partially executes and transfer control to other coroutines.

- Coroutine is similar to threads. The only difference is once a caller invoked a thread, it will never returns back to caller function. But in case of ~~Sub~~ coroutine can return to caller after executing a few piece of code

* Coroutine:- has many entry point & many exit point.

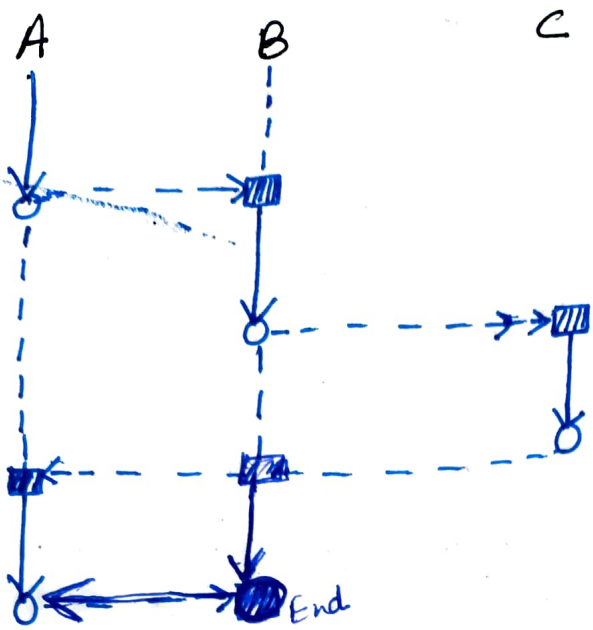
Example of coroutine

The 'yield' Statement in Python. when 'yield' is encountered the current state of function is saved & control is returned to calling function.

Co → Cooperative

Routines → Function.

```
AFunction()
{
 AnotherFonction();  ——→  AnotherFunction()
                          {
                            1     // I am done now. you can continue.
 OK, Print A  ←——
 I am done, you start  ——→  2
                            3     // I am done. your Turn now.
 Scan A  ←——
         ——→
```



Coroutine Sequence control

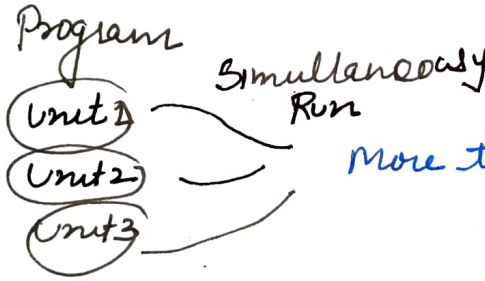
Topic Name

⇒ **Concurrency Computing** – is simultaneous execution of multiple interacting computation tasks.

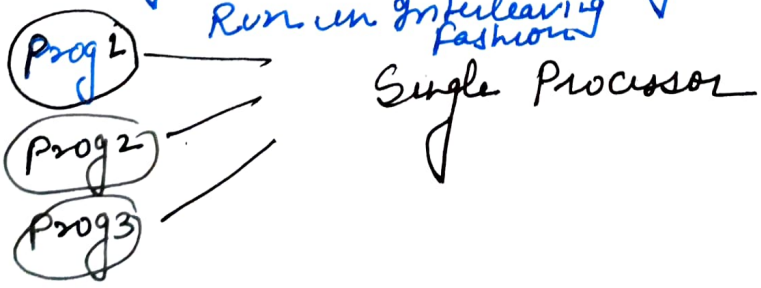
Task (or Process) → Single unit of control. Each task in program can provide one thread of control

Category of Concurrency — physical, Logical

(I) **Physical concurrency**: – The physical concurrency occurs when several program units from the same program execute simultaneously on more than one processor.
↳ Program code executed in parallel on multiple processors.



(II) **Logical concurrency**: – The logical concurrency occurs when execution of several programs take place in interleaving fashion in a single processor.



⇒ **Concurrency Levels**

(I) **Instruction level**: – is execution of two or more machine instructions simultaneously.

(II) **Statement level**: – is execution of two or more machine statement simultaneously.

(3) Unit level or Subprogram level → is execution of two or more subprograms units simultaneously.

(4) Program Level :- is the execution of two or more program simultaneously

The interaction between process-take 2 forms:-

(I) Communication → involves exchange of data between processes either by an explicit message or through the value of shared variables.

(II) Synchronization :- Exchange of control information between processes.

Next Topic

⇒ Subprogram Level concurrency & Synchronization

Task :- is Sugle unit of work.

Task can be one of following stage :-

New → created but not yet started

Runnable/Ready → Ready to run

Running →

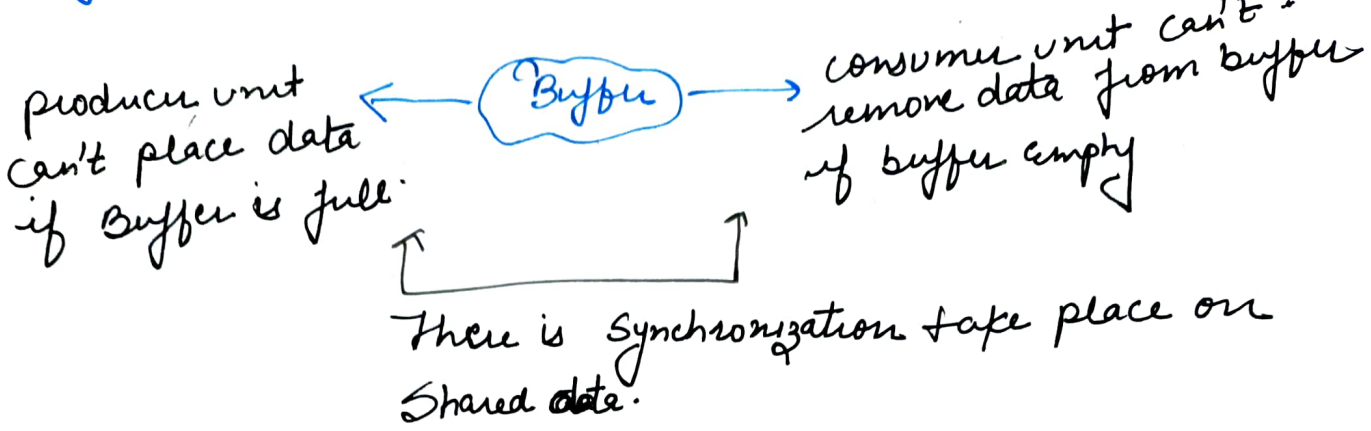
Blocked → usualy waiting some event to occur

Dead

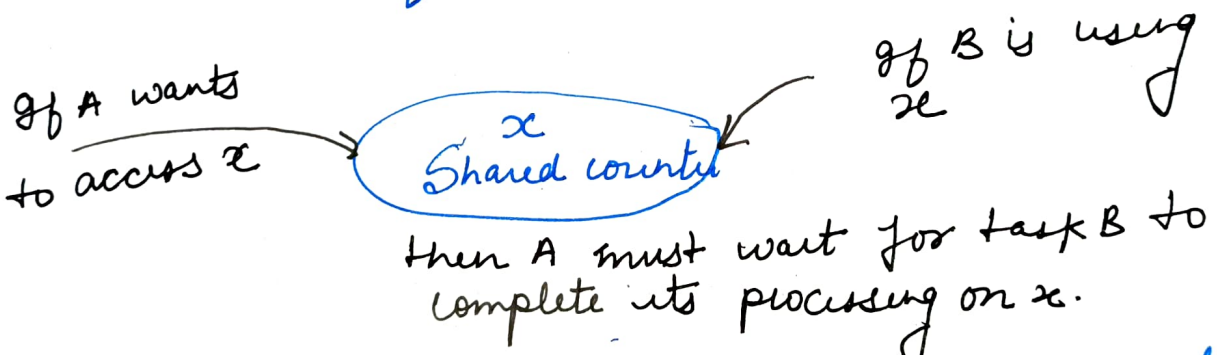
Synchronization :- is a mechanism that control the order in which tasks execute. Two type of synchronization required when tasks share data.

↳ Cooperation Synchronization → between two tasks A and B is required when task A must wait for task B to complete some specify activity before task A can continue its execution.

e.g Producer – consumer problem



II Competition Synchronization :- The competition Synch. between two task A & task B is required when both require use of some resources that can't be simultaneously used then one of them wait.



* competition is usually provided by mutually exclusive access.

Methods for Synchronization

Synchronization Defination :- → is a mechanism that controls the order in which tasks execute.

issues of Synchronization :- - used for providing for mutual exclusive access to shared resources.
- used for cooperating process.

→ 3 Methods :-

(I) Semaphores (II) Monitors (III) Message Passing

Why we need Synchronization ? → To avoid Race condition

P_1

```
{
  =
  =
count++
  =
  =
}
```

P_2

```
{
  =
  =
count--
  =
  =
}
```

Count
10

count ++
① R_1 = count
② R_1 ++
③ count = R_1

count --
① R_2 = count
② R_2 --
③ count = R_2

Race condition

(case 1) If we execute P first P_1 completely & then P_2 completely.
then value of count = 10

(case 2) If we execute P_1 statement ① & ② then P_2 statement ① ② ③ then
value of count = 11

(case 3) If we execute P_1 statement ① & ② then P_2 statement ① & ② then P_1 Statement ③ & then P_2 statement ③
value of count = 9

(1)

⇒ Synchronization through Semaphores

Semaphores was devised by Edsger Dykstra in 1965. The concept of Semaphores was first implemented in ALGOL 68.

what is Semaphore: - A semaphore is an integer variable shared among multiple processes. The main aim of using semaphore is process synchronization and access control for a common resource in a concurrent environment.

Types

Binary Semaphore: - This is also known as mutex lock. It can have only two values → 0 and 1.

counting Semaphore: - Its value can range over an unrestricted domain.

2 Primitive operation in Semaphore

wait(): - The wait() function is used to decrement the value of the Semaphore variable "s" by one if the value of the semaphore variable is positive If value of the semaphore variable is 0, then no operation will be performed.

wait() operation also called down or P():

```
int s;
wait(s)
{ while(s==0);
  s--;
}
```

Signal(): - The signal() function is used to increment the value of semaphore variable by one.

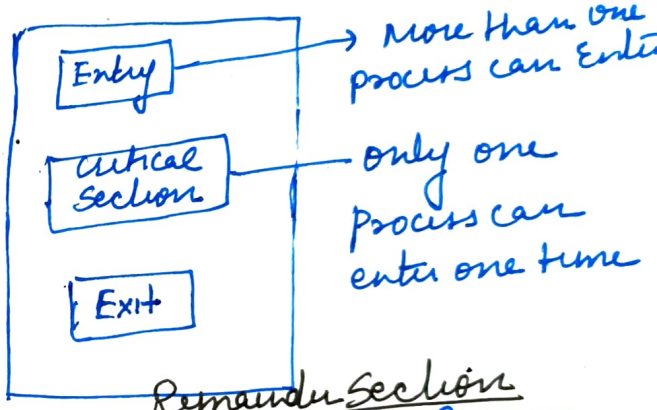
```
int S;
Signal(S)
  {
  S++;
  }
```

Signal(S) :- also known as up or V();

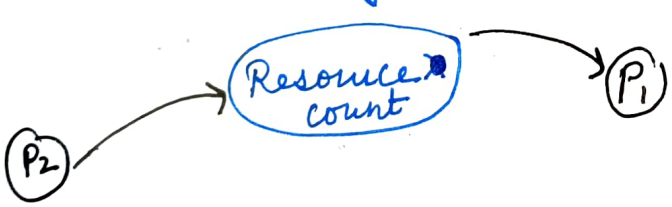
Critical Section → is a segment of code for each process where the process may be changing common variables, updating tables, writing a file & so on. e.g count variable when one process is executing in its critical section no other process is to be allowed to execute in its critical section.

entry section
critical section
Exit section
Remainder section

OR



Critical Section → only one Process can access the Resource



only P_i using X count

Sol^n of critical Section (where 2 competing Process)

(I) Mutual Exclusion
(II) Progress
(III) Bounded wait

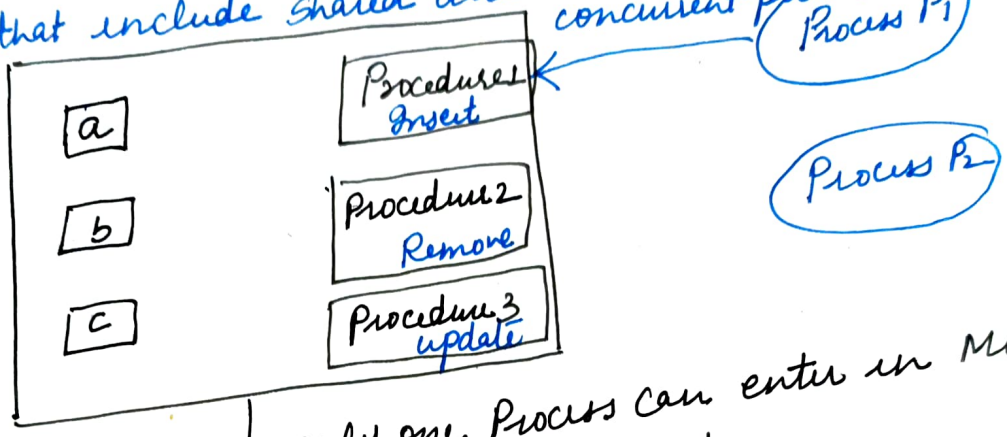
Advantages & Disadvantages of Semaphore

Advantages : (I) Semaphore provides cooperation synchronization.
(II) " " Mutual Exclusion.
(III) " Avoids Busy waiting.

(2) Synchronization through monitors

Monitor is a module that contains (1) shared data
(2) procedure that operate on shared data

* A monitor can be considered as an abstract data type (ADT) that include shared data & all the operation various concurrent process can perform.



Only one Process can enter in Monitor using Procedure

A monitor is an interface between concurrent user process and provides :-

(i) A Set of Procedure callable by processes.

(ii) A mechanism for enforcing mutual exclusion one these procedure → only one process can execute.

(iii) A mechanism for scheduling calls to these procedure if other concurrently executing process request usage before the procedure has terminated.

(iv) A mechanism for suspending a calling procedure until a resources is available (delay) & then continue the process.

⇒ The procedure inside monitor determine an initialization operation access rights, and synchronization operations

A monitor has the form.

```
monitor monitor name
{
    Shared variable declarations

Procedure P1 (...)
 {
  ...
 }
Procedure P2 (...)
{
  .
  .
 }
        .
        .
Procedure Pn (....)
 {
    . . .
    . .
 }
{ intialization code
}
```

* Note
1. Shared variable are accessible through the monitor's procedure.
2. only one process can run inside the monitor.
3. All critical section are placed inside the monitor procedure.
4. one process can be active in monitor at any given time.

Monitor condition variable & operations

Condition variable:- condition variable are used to add additional synchronization behaviors beyond basic mutual exclusion. The condition variable are declared as condition x.

The only operations that can be performed on condition variable are wait() & signal().

1. x.wait():- behaves like a sleep.
2. x.signal() → behaves like a wakeup.

In addition to operation defined on monitors, there are two special operations

(1) Delay(): - It is analogous to semaphore wait operation and execution of delay enequene a process.

(11) continue(): - It is analogous to semaphore's signal operation and continue operation dequenes the first waiting process and allow it to enter the monitor

Advantages

(1) A monitor provides interface between concurrent user processes.
(2) A monitors allow concurrent process to prevent from accessing the same data item simultaneously.
(3) Monitors encapsulates the shared data structure with their operations.
(4) All the code for buffer appears together.
(5) Better way to provide competition synchronization
(6) Monitors are nice way to write multitasking programs

Disadvantages

(1) The monitors cannot provide synchronization of units in a distributed system which each processor has its own memory.
(2) Monitors are structure oriented.
(3) Monitors are not even in some most commonly used language like C, C++ etc.
(4) There is no exchange of data using monitors.

(3) Synchronization through message Passing

Message Passing models allows multiple process to read and write data to the message queue without being connected to each other.

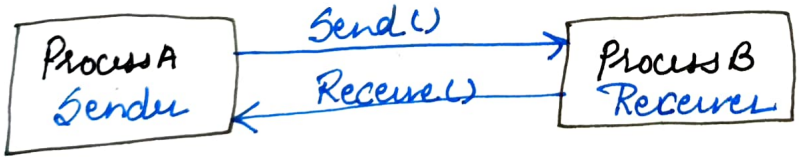
IPC (Interprocess communication) is used by cooperating processes to exchange data as information.

In this a sender or a source process send a message to a non receiver or destination process. Message has predefined structure

Message Passing uses 2 system calls

(i) Send() (ii) Receive()

↳ Send(name of destination process, message)
Receive(name of source procss, message)



In this calls, the sender and receiver processes address each other by names. Mode of communication between two process can take place through two methods

(1) Direct Addressing :-

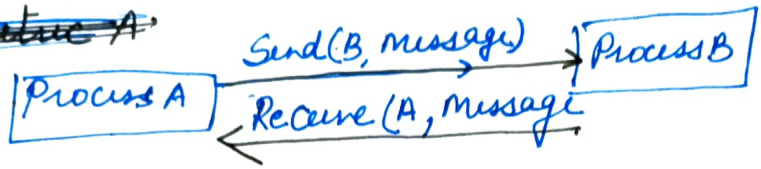
In this type that two processes need to name other to communicate. This process become easy if they have same parent

E.g:- If procss A sends a msg to procss B

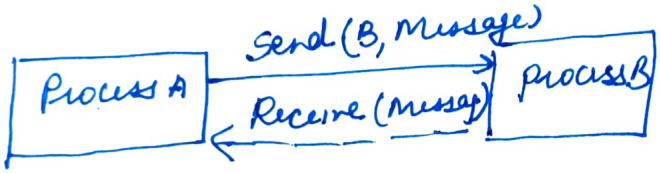
Send(B, message);
Receive(A, message);

By message passing a link is established Between A and B. Here the receiver knows the identity of sender message destination. This type of arrangement in direct communication is known as Symmetric Addressing. or Synchronous.

Asymmetric A



Asymmetric Addressing :— Another type of addressing where receiver does not know the ID of sending process in advance.



Asynchronous or Asymmetric addressing

Indirect Addressing :-

In this message send and receive from a mailbox. A mailbox can be abstractly viewed as an object into which messages may be placed and from which messages may be removed by processes

The sender and receiver processes should share a mailbox to communicate

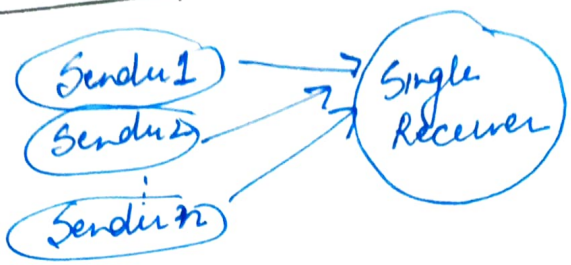
The following type of communication link are possible through mailbox.

one to many link

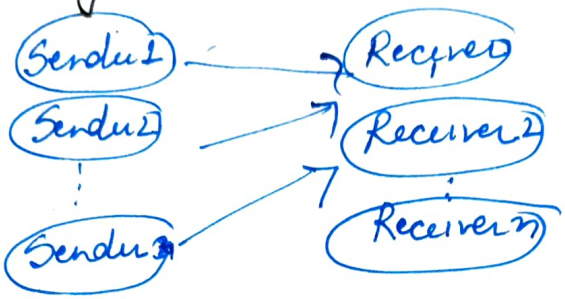
e.g Broadcast

one to one link

Many to one link



Many to Many link



Advantages of Message Passing

(i) There is no need to share memory.
(ii) Message passing is appropriate for distributed system because shared memory is not required.
(iii) No critical section or regions are required to maintained.
(iv) Message Passing is extensively used by many system like mach the basis of Apple's new Macintosh as.

Disadvantages

Reliability: - If Process are in different machines, a protocol must ensure that message are delivered in order & none is lost.