static int count = 5;

cout << " " << count << endl;

#include <iostream>
using namespace std;
class car {
public:
int a;
};

class human {
public:
static int count;
void display()
{
cout << count;
}
};

int human :: count = 0;

main()
{
car obj1, obj2;
obj1.a = 5;
obj2.a = 6;
}

human()
{
count++;
}

main()
{
cout << human :: count;

Static variable should be initialized outside the class.
Can be accessed

Static variables and functions

→ Static variable should be initialized first one that should be outside the class using scope resolution operator.

→ Static variables can be changed in the main using class name and object.

→ Static variables can be accessed inside the main using class name and object

→ Static functions can have only static variables, if we use non static variable it will show error.

→ Static variable can be changed and accessed inside class (using member function and static function)

~~Default value to a function~~

→ Default value to a function

```
void fun(int a, int b, int c
{                          =10)
}
main()
{
  fun(10, 20);
}
```

Encapsulation

→ Data hiding

→ To make other functions and objects not to access member variable & member functions

→ Only the member functions of the same class in which private members are declared can able to access

→ To hide we should use "private" access specifier

friend functions and friend class

```
class A{
  friend void fun
  private:
    int a;
};
void fun()
{
  A obj
  obj.a=5;
}
```

Creating object

```
class Human {
public:
    string name;
    void introduce()
    {
        cout << "hi" << name << endl;
    }
};

main()
{
    Human jagi;
    Human *ptr = new Human();
    {jagi.name = "Jagadeesh";
    jagi.introduce();
    {ptr -> name = "Jagi";
    ptr -> introduce();
```

memory will be in stack

memory will be in heap

Constructor

Constructor with no statements

→ It is a special function with no return type

→ It will be having the same name as that of class

→ It is called automatically when the object is created.

→ There will be a default constructor called when you won't write the constructor.

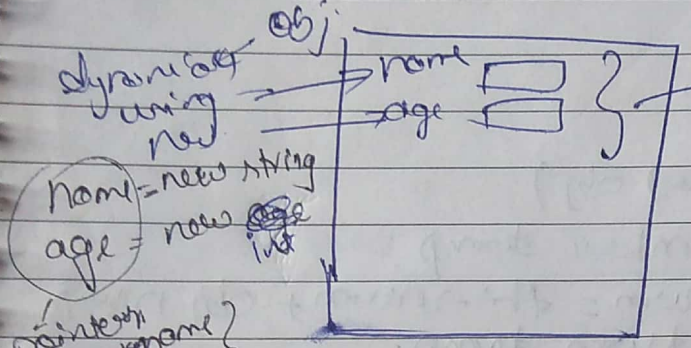
Use:

→ To initialize the variables for that particular object.

Constructor with parameter

Destructor → freed when object goes out of scope (or)
→ Destructor is used to release all the memories allocated to the members of the object.

when delete is called to a ptr of the object it is called



when we create object using "new" keyword, then we only need to destroy object using "delete" else the memory won't be freed and also destructor won't be called.

Function overloading

→ Same name but different no of parameters, data types of parameters, order of parameters.
→ It is used so that a programmer doesn't have to remember multiple function name.

Constructor overloading

operator overloading

n1 + n2
n1.operator+(n2);

n1 + n2 + n3
n1.operator(n2).

class

Binary operator overloading

```
class Number{
    int num;
    public:
        Number(){ }
        Number(int num)
        {
            this->num = num;
        }
    Number operator+ (Number obj)
        {
            of Number temp;
            temp.num = this->num + obj.num;
            return temp;
        }

    main()
    {
        Number obj1(10), obj2(20), obj(30), obj4;
        obj4 = obj1 + obj2 + obj3;
```

Post & Pre ~~increment~~ overloading

```
class Number{
    int num;
    public:
        Number(){ }
        Number(int num)
        { this->num = num; }

Number operator++(){
    Number temp;
    num = num+1;
    temp.num = num;
    return temp;
}
```

```
main()
{ Number obj1(10),
    obj2;
  obj2 = ++obj1;
  obj2 = obj1++;
}
```

```
Number operator++(int)
{ Number temp;
  temp.num = num;
  num = num+1;
  return temp;
}
```

Inheritance

class base {

};

class derived : public Base {

};

	Public	Protected	Private
Same class	✓	✓	✓
derived class	✓	✓	X
Outside class	✓	X	X

Public Inheritance:

→ All public members becomes public, private cannot inherited, protected becomes protected in derived class.

→ To assign any value to a private member just upcast (using functions)

Protected Inheritance

→ All public, protected members become protected, private cannot inherited in derived class.

→ Same.

Private inheritance.

→ public, protected becomes private and private members cannot inherited in derived class.

→ upcasting can be used if we want to initialize data in further derived classes.

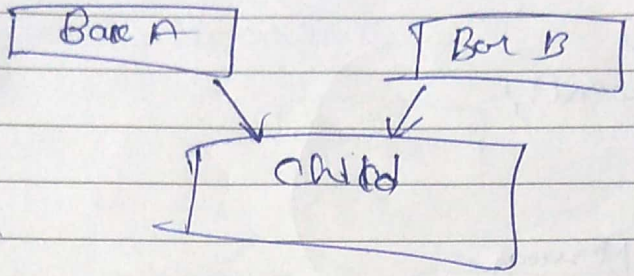
To change access level of Base class member in derived class

Eg:

```
class Person{
    protected:
        string name;
    public:
        void setname(string iname)
        {
            name = iname;
        }
};
class Student : private Person{
    public:
        Person::name;
        Person::setname;
};
```

That will become public in Student class i.e changing access level

Multiple inheritance



Order of Execution of Constructor & Destructor in Inheritance

Base construct → Derived Const → Derived Destruct
→ Base Destructor.

Passing value to Base constructor through Derived class.

Belongs to child class
Belong to Father class

child(int x): Father(x)

Overriding Base class method in Derived class

```
class Person {
  public:
    void introduce()
    {
      cout << "hi" << endl;
    }
};
class Student : public Person
  { public:
      void introduce()
      {
        cout << "hello" << endl;
      }
};
main()
{
  Student obj;
  obj.introduce();
}
```

→ overriding base class method introduce

→ o/p hello

Accessing the overridden base class method

```
class Person {
  public:
    void introduce()
    {
      cout << "hi" << endl;
    }
};
class Student : public Person
  {
  public:
    void introduce()
    {
      cout << "hello" << endl;
      Person::introduce();
    }
};
main()
{
  Student obj;
  obj.introduce();
  obj.Person::introduce();
}
```

→ o/p hello

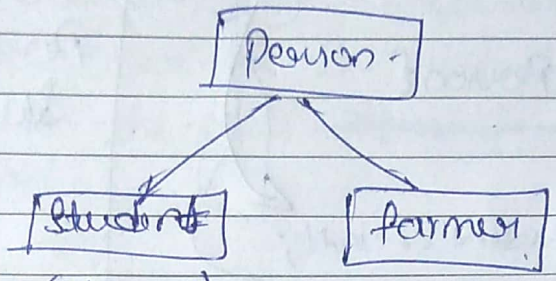
→ o/p hi

→ o/p hi

Dynamic polymorphism

↓ having many forms

That will happen when there is an heirarchical of classes using inheritance:



Scenario 1 (Normal)

```
class Person{
 Public:
    void introduce()
    {
    cout << "from Person" << endl;
    }
 };
class Student: Public Person {
   Public:
       void introduce()
       {cout << "from Student" << endl;
       }
   };
class farmer: Public Person {
    Public:
       void introduce()
       {
        cout << "from farmer" << endl;
       }
   };
void whoisthis (Person p)
   {
     p.introduce();
   }
main()
{ Student obj1;
  whoisthis(obj1); //o/p: from Person
  farmer obj2;
  whoisthis(obj2); //o/p: from Person
```

→ Base type

Scenario 2 (Dynamic Polymorphism)

```
class Person {
  Public:
   virtual void introduce()
   {
    cout << "from Person" << endl;
   }
};
```

→ Same

```
main()
{Student obj1;
whoisthis(obj1); // from Student
farmer obj2;
whoisthis(obj2); // from farmer
}
```

Inheriting Virtual nature

```
class Person{
public:
virtual void introduce()
    {
    cout<< "from person" << endl;
    }
};
class Student : public Person{
public:
    void introduce()
    {
    cout << "from Student" << endl;
    }
};
class GStudent : public Student {
public:
    void introduce()
    {
    cout<< "from GStudent" << endl;
    }
};
void whoisthis(Person p)
{
    p.introduce();
}
main()
{
    GStudent obj;
    Person P;
    Student S;
    GStudent G;
    P.introduce(); -> from person
    S.introduce(); -> from Student
    G.introduce(); -> from GStudent
    whoisthis(P);
    whoisthis(S);
    whoisthis(G);
}
```

Even if virtual nature will be inherited from Person class to Student class

If it is not there the o/p of cout will be "from Student"

Pure virtual function, abstract class.

Pure virtual function

→ When base class not knowing to define the virtual function

→ When all the derived classes from the base class "must" override the virtual function

→ In both the cases ~~virtual func()~~ Pure virtual functions are used.

```
class Person{
public:
    virtual void introduce() = 0;
};
```

→ Pure virtual function

```
class Student : public Person{
public:
    void introduce()
    {
        cout << "from Student" << endl;
    }
};
```

→ If we don't override introduce method, then we will get error.

```
main()
{
    Student s;
    s.introduce();
}
```

→ o/p from Student.

We can also define pure virtual function definition (body) like as follows

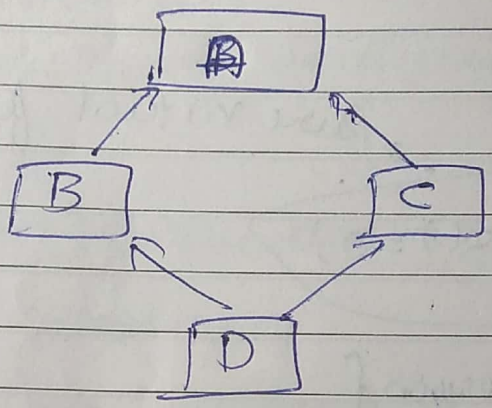
→ It ~~should~~ can be declared outside of the class using scope resolution operator

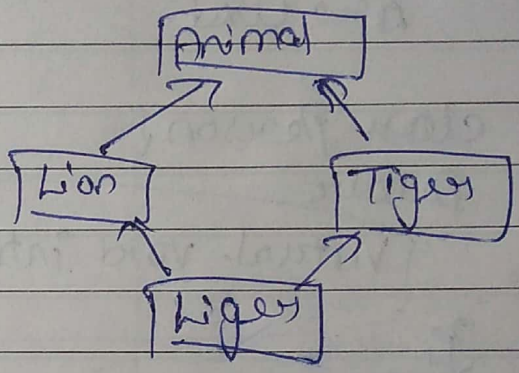
```
void Person::introduce()
{
    cout << "from Person" << endl;
}
```

Abstract class:

- If a class has one or more pure virtual functions then it is called as abstract class.
- We cannot create an object for abstract class.

Diamond problem





To solve diamond problem we need to use virtual inheritance.

```
class Animal {
public:
    void walk()
    {
    cout << "Animal" << endl;
    }
};
class Lion: public Animal {
public:
class Lion: virtual public Animal {
{
};
class Tiger: virtual public Animal {
};
class Liger: public Lion, public Tiger {
};
```

```
main()
{
Liger obj;
obj.walk()
}
```

main points

* All objects share the same copy of the member function but maintains a separate copy of the member variables.

Macro, function, inline function, typedef, structure, union, enum, Static and Dynamic ~~library and~~ linking

Static and Shared libraries

FCFS

P1	0	7
P2	0	4
P3	0	5
P4	0	3

###

Static linking v/s Dynamic linking

✓

Compiler → To convert to an executable code so that program can be run on our machine.

Static libraries

.a → Linux.

#define fun(a, b) a+b (function like)

~~...~~ undef ...

#define MAX 3 (object like)

get(a) ...

get (...)

main()
{ int
~~pf("%d", MAX)~~
a = MAX;

→ Two types

→ Any datatype of variables can be there (arguments are not checked for data type)

✓ #define INCREMENT(x) ++x

✓ pf("%d", INCREMENT(pt)) ++ptr;

✓

→ #define

concat(a, b) a##b

#undef ...

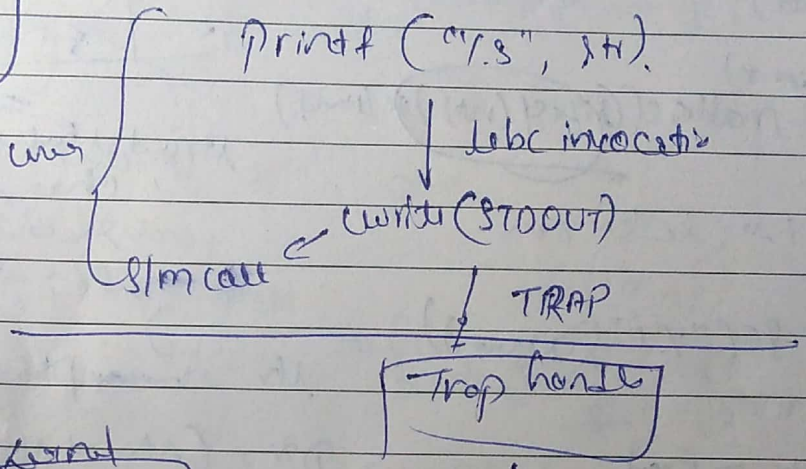
#a

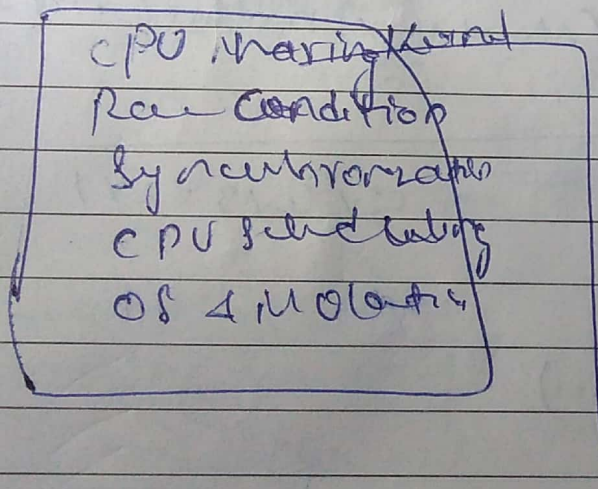
3, 4

34

1) preprocessing
2) data type not check
3) argument not evaluated
4) undef
5) ~~...~~ #

for multiple process, the lower part will be different, whereas the upper part will be same (i.e the lower part will be replaced)



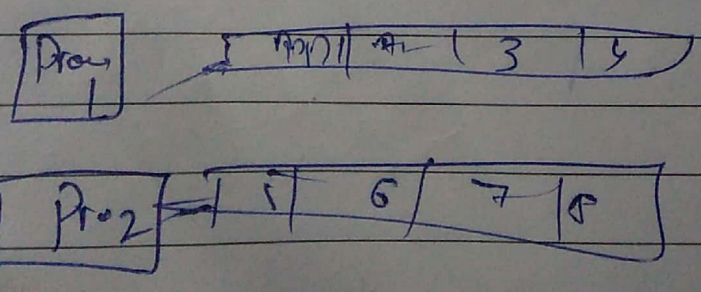


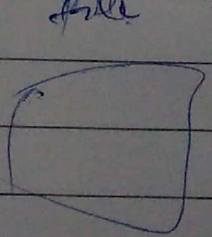
Invokes a function to display str

Decoder

3 core

CPU time is divided into time slice / time quanta.
Each slice of time is given to a each process



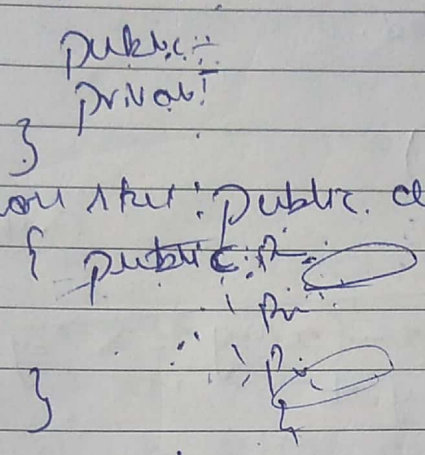


Working of Virtual memory

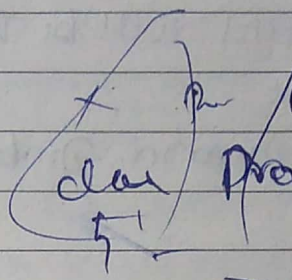
1. First when we run ./a.out, OS creates an empty page table.
2. Then control is transferred to main()
3. corresponding virtual address of main() in the text segment is taken by CPU (processor)
4. Then the virtual address is intercepted by MMU.
5. MMU checks for the corresponding page frame in the page table (check the value of present bit in page table)
6. If the value of present bit is 0 it triggers OS by rising page fault trap.
7. Then OS check for the corresponding page location on disk.
8. Then identifies a page frame to be used in RAM,
9. Triggers loading of the new page from disk to RAM
10. Then updates the page table
11. CPU resumes instruction and sends the same virtual address to the MMU.
12. Then MMU computes physical address of the first instruction to be fetched by the CPU.

Public | Protected | Private

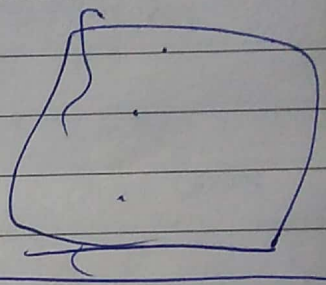
class clgl
{ protected:
public:
private:
}
class stud : public clgl
{ public:
}



class Protected



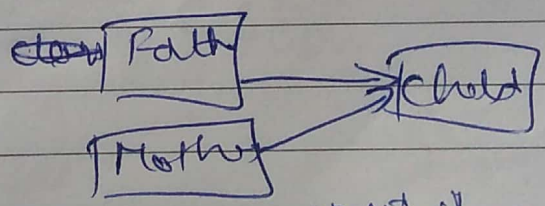
class Private
Pr:



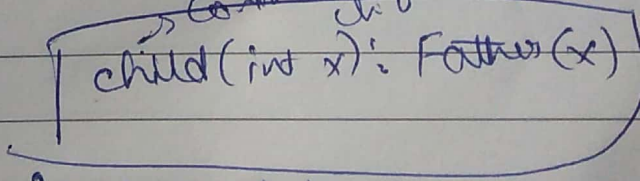
Extra Concepts ~~to be known~~

1) Access declaration. → classname :: name of the mv or mf
Eg: Person :: name;
Person :: SetName;

2) Multiple inheritance.



3) Binding values to the base class constructor from the child class



4) Overriding base class method in derived class
5) To get original method which has been overridden for the base class

```
Derivedobj.classname::memfunc();
obj.Person::introduce();
```

6) Using of virtual keyword (This will come into picture during Dynamic Polymorphism. We should make intructors, overriding etc

7) Even virtual will be inherited.

8) Pure virtual functn

9) Virtual class (Diamond problem)

(a+b)

#define swap(a,b) ~~a=x1y=x1y~~ x=(x^=y^=x^=y)

int t; t=x; x=y; y=t

Real time system: (RTS)

→ A system that responds to an external event in a guaranteed amount of time.

OS vs RTOS

- A GPOS is used for system / appln^ns that are not time critical • Eg: Windows, Linux, Unix etc. An RTOS is used for time critical systems. Eg: VxWorks, uCos etc

- In GPOS task scheduling is not based on "priority" always. Whereas in an RTOS scheduling is always priority based.

Type of RTS

1) Hard RTS 2) Soft RTS

~~RTOS~~ RTOS

OS → A set of programs that acts as an intermediary b/w a user of a computer & the computer hardware.

→ It is a set of prgms that designed to manage all the resources of the computer.

→ H/w resources: Memory, I/o devices, Comm" devices etc

S/w resources: file system, virtual memory, security

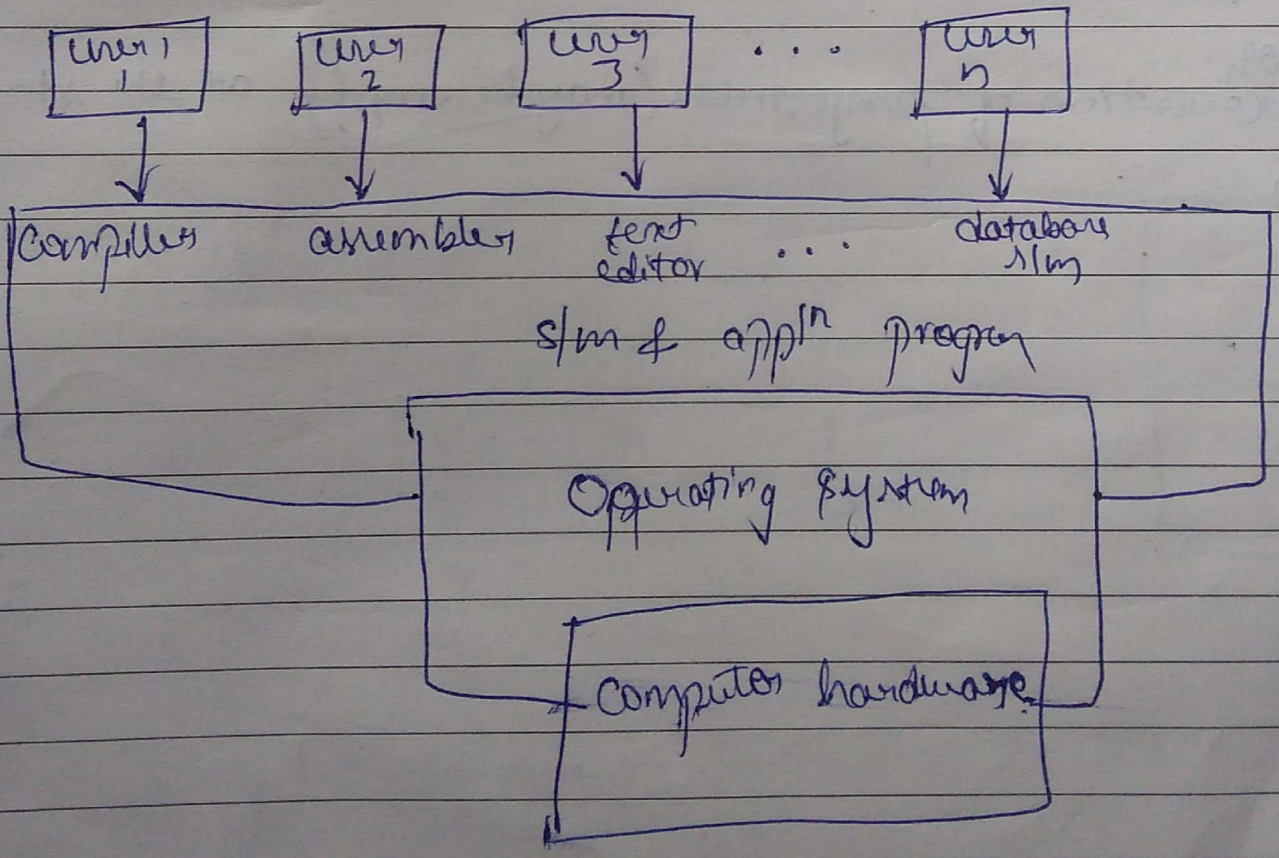
Need of OS

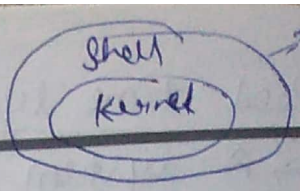
→ Without an OS, a user cannot run an appln prgm on their computer, unless the appln prgm is self booting

→ Execute user prgms & make solving user problems easier

→ Use the computer h/w in an efficient manner.

Abstract view of S/m Components





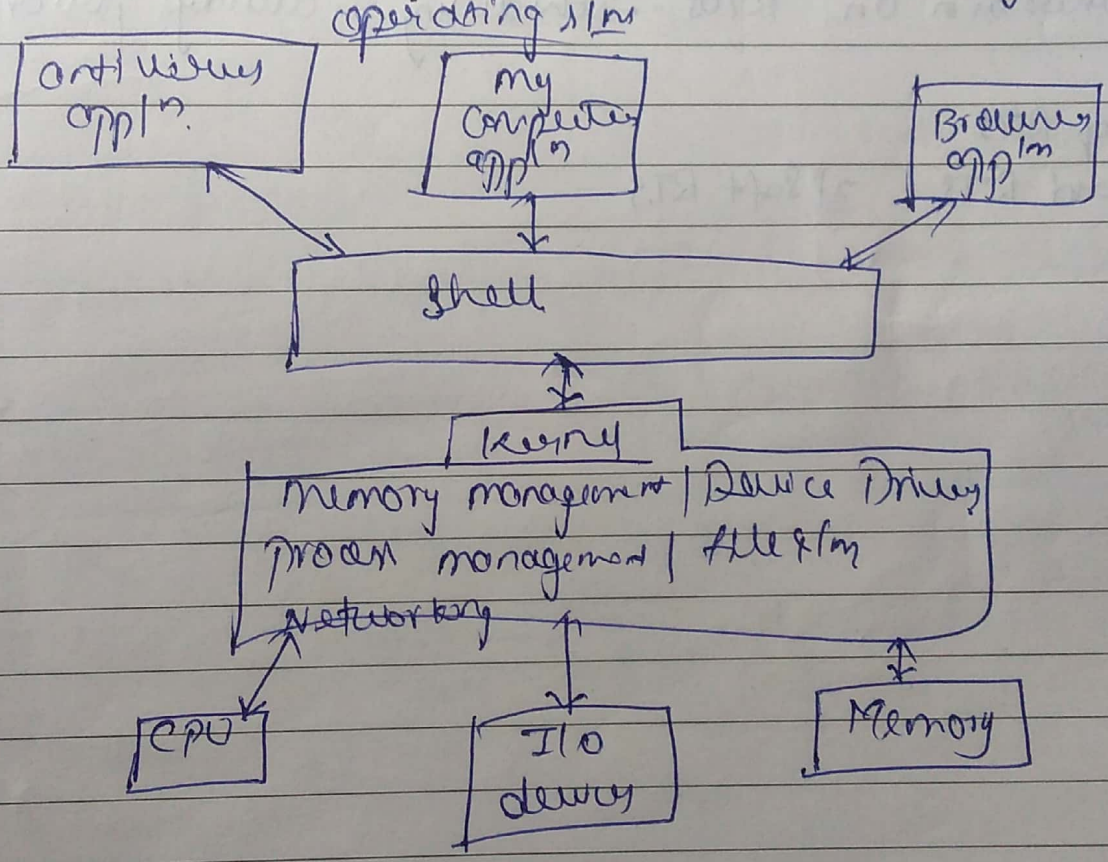
O/S is divided into 2 parts.

1) Shell

It is a program which handles all other programs
Eg: • K-shell or Born shell in unix
• Explorer.exe in Windows
• Command prgm in DOS

2) Kernel.

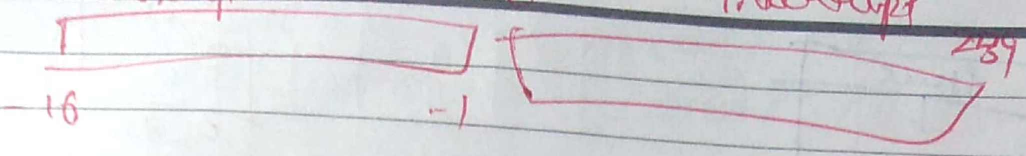
It provides the most basic level of control over all of the computer h/w devices • Time mngmnt, task scheduling, memory management, file s/m. etc all are monitored by the Kernel.



Types of OS

1) Batch ~~operating~~ processing OS
2) Time sharing
3) RTOS
4) Embedded OS

S/m exceptions — peripheral interrupt

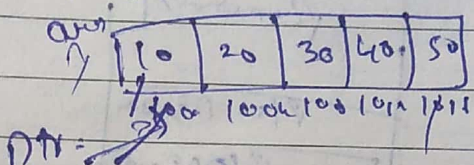


CMSIS

```
char str[] = "hello"
char *ptr = str.
*(*(ptr+1)+1)

int arr[] = {10,20,30,40,50};
int *ptr = arr
ptr++;
```

arr -> base ad
&arr -> whole *ptr.

```
*(ptr+1)
*(ptr+2)
*(ptr+3)
*(ptr+4)
```




```
for(i=0; i<sizeof(arr)/sizeof(a); i++)
{
    printf("%d", *(ptr+i));
}
```

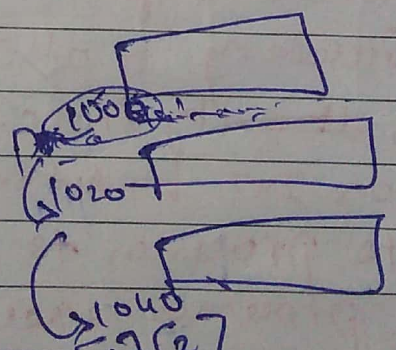
```
int arr[2][3] = {{1,2,3},{4,5,6}};
int *ptr[] = arr;
int (*ptr)[5] =
```

Pointer to array
Array of pointers

arr[][]; &arr;

size of column.

```
int (*ptr)[5] = arr.
ptr++;
int *ptr[5];
int arr[][3]={}
int *ptr;
int (*ptr)[5] = arr;
int (*ptr)[5]=
int arr[][3]=
```



ptr[0][0].
&ptr[0][0]

arr
arr+1
arr+2

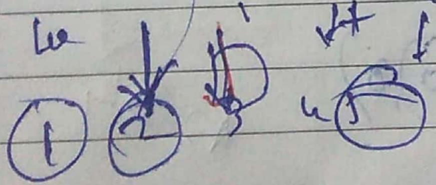
ptr[1][2]
((ptr+1)+2)

Scanned by CamScanner

```
for(i=0; i<3; i++)
{
    for(j=0; j<3; j++)
    {
        pf("%d", arr[i][j]);
    }

int arr[][3] = { 1, 2, 3, 4, 5, 6, 7, 8 }
fun(arr);

void fun(int arr[][3])
int (*arr)[3];
sizeof(arr) // 4
sizeof(*arr) // 12

int *arr[3];

int *ptr[3];
ptr[0]     ptr[0] 2000    1000    arr+0    &a
ptr[1]     ptr[1] 2012    1004    arr+1    &b
ptr[2]     ptr[2] 2024    1008    arr+2    &c
                          1012

for(i=0; i<3; i++)
    pf("%d", p[i]);

ptr[0][1]

int (*ptr)[5];
int *ptr[5]

char *arr[5];
char *arr[5] = {"hello",
}

int (*ptr)[3] = arr;

int *ptr[2];
ptr[0] = arr[0]
ptr[1] = arr[1]

for(i=0; i<2; i++)
ptr[i] = arr+i

char *arr[2] = {"I", "II"}; read only memory
char **ptr = arr
char **ptr = arr;
&a[1]
char **b = &a[1]
ptr+0
ptr+1
*+ptr
```

Linear

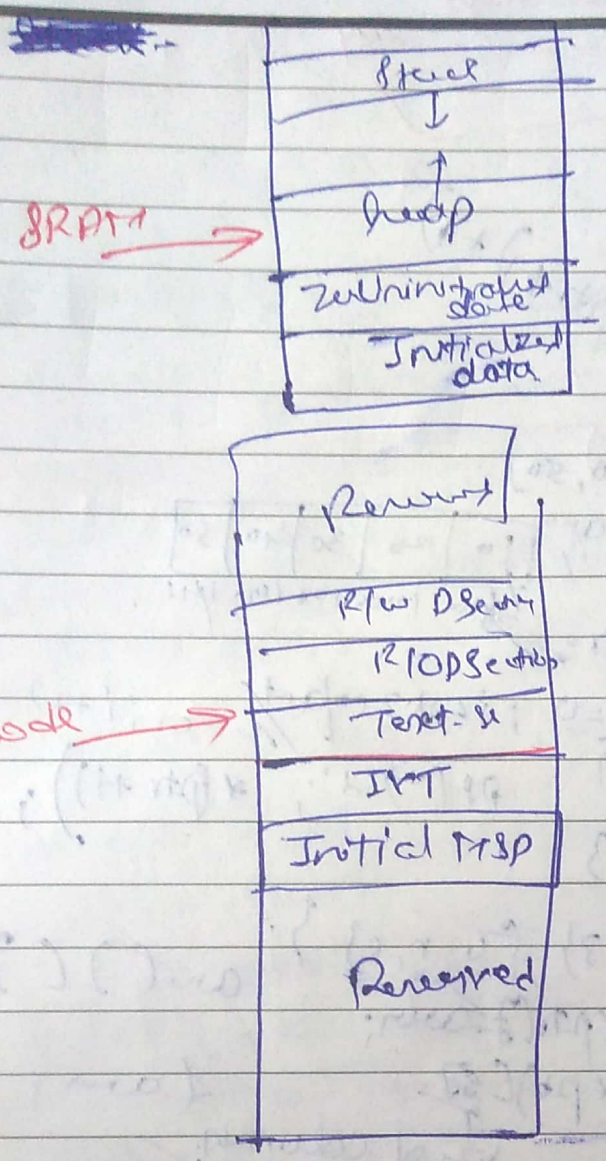
```
K = 10.
for(i=0; i<n; i++)
{
    if(arr[i] == k)
    {
        pf("Ele fo\n");
        return i+1;
    }
}
return 0;
```

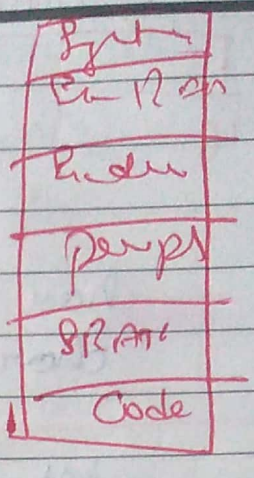


Binary

```
low = 0, high = sizeof(arr)/sizeof(arr[0])
while(low <= high)
{
    mid = (lo+hi)/2
    if(arr[mid] == key)
        return mid+1;
    else if(key > arr[mid])
        low = mid+1;
    else high = mid-1;
    mid = (l+h)/2
```

Callback functions





When an interrupt occurs (assume we need to set Enable & Priority). Stacking of all the registers will happen in the stack, ~~then~~ by NVIC. Then NVIC obtains the ISR address from IVT based on IRQ. Then set PC. ~~After completion~~ Then it gives to process, to execute that ISR. After execution processor executes BX LR to unstack.

How Interrupt works (Normal)

→ ~~Right~~ the peripheral sends an IRQ ~~signal~~ request signal to the PIC (Programmable interrupt controller)

→ PIC converts IRQ to an ~~interrupt~~ interrupt vector number. & then asserts ~~the~~ INT line to inform the processor

→ The CPU saves its context & sends an INTP to the PIC expecting an interrupt vector number

→ Then PIC sends an interrupt vector number to the CPU via system bus & the PIC deasserts the INT line.

→ Then CPU executes the ISR of ~~that~~ Interrupt vector number

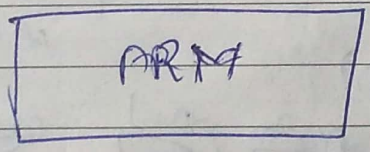
---

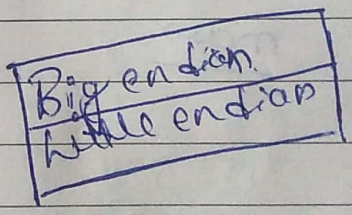
Interrupts

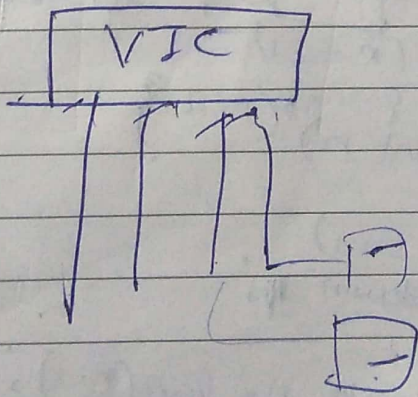
Recursion

Big endian & little endian

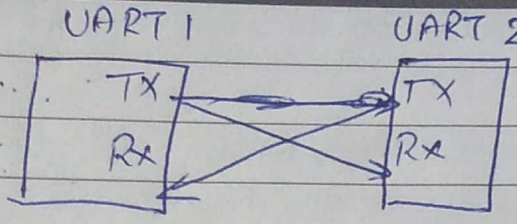
RTOS







UART (Universal Asynchronous Receiver / Transmitter)



Stand alone IC

why clock is used ??

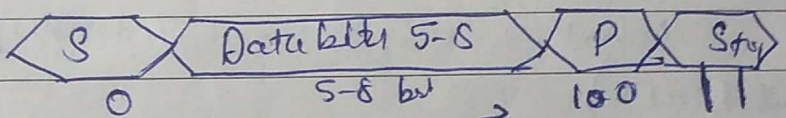
Baud rate

→ TX UART converts parallel data from the CPU through data bus into serial form, transmits it in serial form.

→ Asynchronous

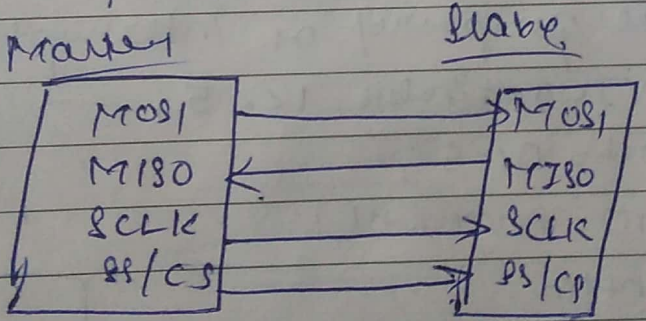
→ Baud rate → measure of the speed of the data transfer.

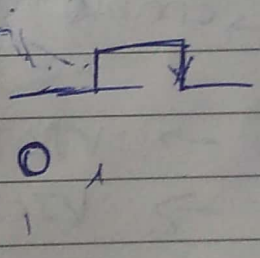
Packet



→ Both UART's must be configured to transmit & receive the same data structure

SPI (Serial peripheral interface)

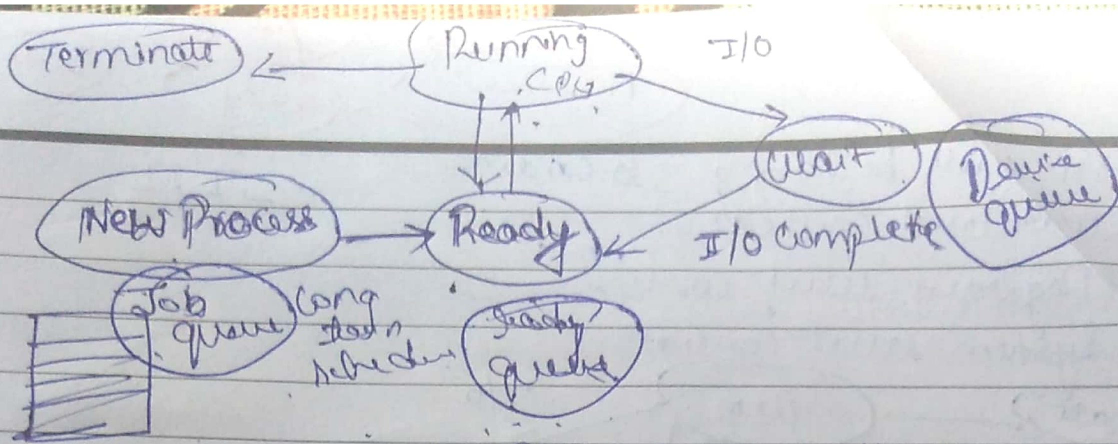




→ Synchronous

→ Clock polarity (CPOL) → CPHA

Clock s/g can be modified using properties of CPOL & CPHA



Enqueuer → Update PCB relating to the process state.

Dispatcher: It will do loading & unloading of context of a process

~~Forex~~ fork() → To create a new process

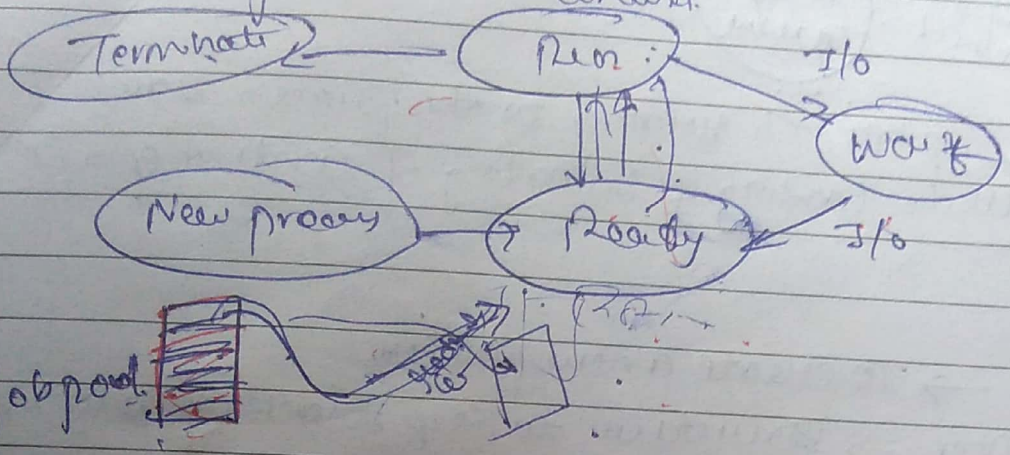
process descriptor → Structure to keep track of process attributes and information. The kernel stores all the process descriptors in a circular double linked list. Kernel uses a global variable "current" for reference of currently running process.

Foreground process → Runs in front but we cannot run any other commands (can't start p
eg: cat command

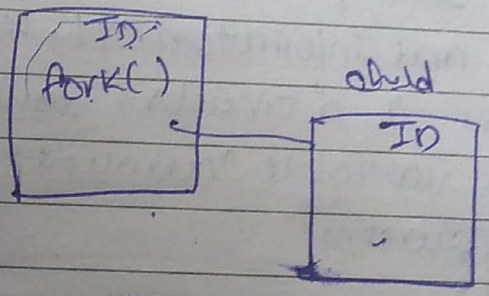
background process
↓
Runs in background. If we give '&' to the executable file then it runs in background.

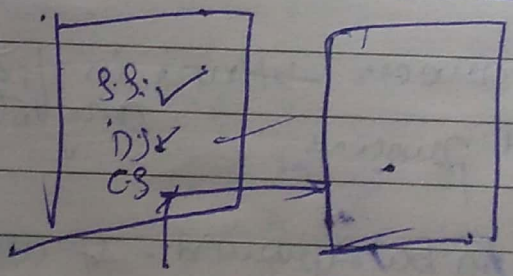
Daemon process runs in background

Process will be having 3 level contents
→ User level context
→ Register level context
→ System level context



PCB →
Process ID
Process state
Intq open file





Software bit
COW
Copy on write

Q) what will happen when fork() is called??
A) 1) Related to memory
2) Related to return value

```
int ret = fork();
if (ret != 0)
{
    parent
    getpid();
    getppid();
}
else
{
    child
    getpid();
    getppid();
}
```

= 8123209642

Encapsulation is the packing of *data* and *functions operating on that data* into a single component and restricting the access to some of the object's components.
Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

Abstraction is a mechanism which represent the essential features without including implementation details.

Encapsulation:-- **Information hiding**.
Abstraction:-- **Implementation hiding**.

Example:

```
class foo{
    private:
        int a, b;
    public:
        foo(int x=0, int y=0): a(x)

        int add(){
            return a+b;
        }
}
```

Internal representation of any object of `foo` class is hidden outside the class. --> Encapsulation.
Any accessible member (data/function) of an object of `foo` is restricted and can only be accessed by that object only

Encapsulation :- Information hiding
Abstraction :- Implementation hiding

Abstraction :- is a Mechanism which represents the Essential feature without including Implementation details

Encapsulation :- is packing of data & function operations on that data into a single component and restricting the access to some of the objects component.

- it means that the Internal representation of an object is generally hidden from view outside of the objects definition.