23/11/22

⇒
```
main()
{
  int x=10;
 if(x<=10)
  {
    main();
    PF("%d\n", x);
  } x++;
}
```

⇒ This program goes into infinite loop by creating multiple stack frames.

⇒
```
main()
{
  int x=1;
 For(x=1; x<=10; x++)
  {
    int x=10;
    PF("%d\n", x);
     x++;
  }
}
```

⇒ It will print '10' 10 times.

⇒ Variable's memory will allocate & deallocate 10 times

(2) Register storage classes :-

⇒ Register storage class is applicable on local Variables only.

⇒ when we use register keyword, the variables memory is allocated in general purpose CPU registers. If the registers are not freely available then the memory is allocated in stack frame of corresponding fun^c. But this variable

⇒ But we can't print the address of a register variable. But we can access the content. If we try to print the address, it will generate compilation error.

⇒ This Register keyword is applicable to access variable's memory very fastly.

* Life time :- From the statement execution time to till the End of the fun^c.

* Scope :- Fun^c scope or Block scope.

* Default values :- Garbage values

* storage location :- General purpose CPU registers (or) stack frame of corresponding fun^c. (This can be verified using gdb tool).

④ we can't read the input from the keyboard for register variable (using scanf fun^c).

③ Extern Keyword :-

⇒ This keyword is applicable on global variables.

⇒ This is to inform to the compiler that variables memory is allocated globally, simply access that variable.

⇒ we can't initialize a variable with extern keyword.

⑤ Scope ⇒ visibility of a variable during its life time

1.c

```
main()
{
 display();
}
int x=10;
```

2.c

```
void display()
{
PF("%d\n", x);
}
```

⇒ To access x value in Fun definition which is in another file we should use extern keyword.

① solution

```
main()
{
 display();
}
int x=10;
```

Life & scope

```
extern int x;
void display()
{
  PF("%d", x);
}
```

② solution

Scope diff

```
void display()
{
 extern int x;
 PF();
}
```

* Life time :- Process loading time to till process termination time.

* scope :- We can increase the scope from file scope to process scope.

* Default values :- zero's

* uninitialized G.V :- .bss
* Initialized G.V :- .data

} storage location.

⇒ Extern keyword even it is applicable on Fun^c declarations when we construct the libraries.

24/11/22

Static Keyword :-

⇒ This static keyword is applicable on local variables, global variables & also functions.

* Local variables :-

~~Life time :- From the stmt execution time to till End of the Fun^c or block~~

~~:- Fun^c scope~~

Life time : From process loading time to process termination

scope : Fun^c scope (or) Block scope

Default values :- Uninitialized static local variables contains zero

Storage location : bss or data segment

⇒ For static local variables, compilers will do naming decoration or data mangling in second stage of compilation.

⇒ For static local variables, the memory sections (or) segments symbols are represented in small letters.

```
int x=10; //.data
main
{
  static int x=2; //.data
  printf("%d", x);
}
```

x.1269 ⇒ d (small letters)

(changes every time)

⇒ Naming decoration or data mangling is nothing but adding a number at the end of a variable as a suffix with dot (.) extension.

(*)
```
main()
{
  increment();
  increment();
  increment();
}
void increment()
{
  int x=10;
  x++;
  PF("%d", x);
}
PF("%p-%d\n", &x, x);
```

It will allocate memory separately for 3 increment functions.

scope

O/P:-

11, 11, 11

(*)
```
main()
{
  increment();
  increment();
  increment();
}
Void increment()
{
  static x=10;
  x++;
  PF("%d", x);
}
PF("%p-%d\n", &x, x);
```

scope

⇒ It will allocate memory only one time as it applied with static keyword.

O/P

11, 12, 13

Similarly global variable memory allocation

⇒ How do you verify that naming decoration is done for static local variables

(A) ① By looking into symbol table

② By stoping the compilation in 2nd stage (By seeing the 2nd stage compilation output).

⇒ This stmt gets executed only one time i.e., during process loading time.

// program

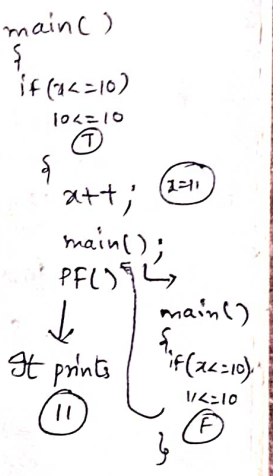
⇒
```
main()
{
  int x=10;
  if (x<=10)
  {
    x++;
    main();
    PF("%d\n", x);
  }
}
```

O/P Infinite loop

⇒
```
int x=10;
main()
{
 if (x<=10)
 {
   x++;
   main();
   PF("%d\n", x);
 }
}
```

O/P

11



⇒
```
main()
{
 static int x=10;
 if (x<=10)
 {
   x++;
  main();
  PF("%d\n", x);
 }
}
```

O/P

11

(static keyword applied variable's memory location is same as global variable. so it acts as global variable

⇒ static variable scope is fun^c scope or block scope, but global variable scope is file scope. This is the only difference b/w static keyword variable & global variable).

⇒
```
int x=10;
main()
{
  if (x<=10)
    {
      x++;
      main();
      display();
    }
}
void display()
{
  PF("%d\n", x);
}
```

O/P

11

⇒
```
main()
{
  static int x=10;
   if (x<=10)
    {
      x++;
      main();
      display();
    }
}
void display()
{
  PF("%d\n", x);
}
```

O/P

compilation error.

⇒
```
main()
{
  int x=1;
for (x; x<=10; x++)
  {
    static int x=10;
     PF("%d\n", x);
      x++;
  }
}
```

o/p

10
11
12
13
14
15
16
17
18
19

⇒
```
main()
{
  static int x=1;
  for (x; x<=10; x++)
   {
     Static int x=10;
      PF("%d\n",x);
       x++;
   }
}
```

o/p
10 11 12 13 14 15 16 17 18 19

o/p will not change

cuz both variables memory will allocate in .data segment with different suffix numbers.

Static Keyword on global Variables :-

⇒ Life time :- From process loading time to till process termination.

⇒ Scope :- strictly file scope only.

⇒ Default values :- Zeroes

⇒ storage location :- .data or bss segments.

⇒ No naming decoration (or) data mangling is done for static global variables.

⇒ Memory section or segment symbols are represented in Small letters.

⇒ Strictly it restricts the scope of global variable within the file from the location where it is declared or defined.

(*) How do you access a static global variable in another file?

(A) ⇒ We can access by using a global extern pointer variable.

1.c

```
static int x=10;
int *ptr=&x;
main()
{
  display();
}
```

2.c

```
extern int *ptr;
void display()
{
PF("%d\n", *ptr);
}
```

⇒ when we assign the address of a variable to a pointer & we can use extern keyword, we can access in another file

⇒ Even static keyword is applicable on functions

⇒ In general fun^c will have process scope or global scope

⇒ when we apply static keyword to a fun^c, the scope reduces to the same file where the fun^c is defined

⇒ In kernal programming every driver routine is applied with static keyword.

(*) How do you access (or) invoke a fun^c applied with static keyword from another file. fun^c.

(A)

1.c

```
static int x=10;
int *ptr=&x;
extern void (*fptr)();
main()
{
  fptr();
}
```

2.c

```
extern int *ptr;
static void display()
{
 PF("%d\n", *ptr);
}
void (*fptr)()=display;
```

(or) 3.c

```
int x;
x=10;
main()
{
 PF("%d", x);
}
```

compilation error.

2.C (or) 3.C

```
Static void display ( );
extern int *ptr;
void (*fptr)( ) = display;
Static void display ( )
{
   PF ("%d\n", *ptr);
}
```

→ we have to write fun^{c} declaration if we want to assign display fun^{c} to fun^{c} pointer before itself.

25/11/22

⊛ Structures :- (Heterogeneous)

⇒ Structures are used to store different type of data elements in continuous memory locations.

⇒ Structures comes under user defined data types.

⇒ we can define the data type by using struct keyword

⇒ we have to write our own structure definition to define the data type.

⇒ Structure name must be a meaningful name.

Student details

name → character array
id_no → integer variable
gen → character variable
grade → character variable

```
struct student
{
  char name[32];
  unsigned int id_no;
  char gen;
  char grade;
};
```

(struct student { ... }; : Structure definition; char name[32]; unsigned int id_no; char gen; char grade; : Structure members)

⇒ Structure definition always ends with semicolon (;).

⇒ This ~~def~~ Structure definition must be defined on the top (cuz to know the compiler for verification & allot size).

i.e., compiler should know the structure definition to decide the size and for verification purpose.

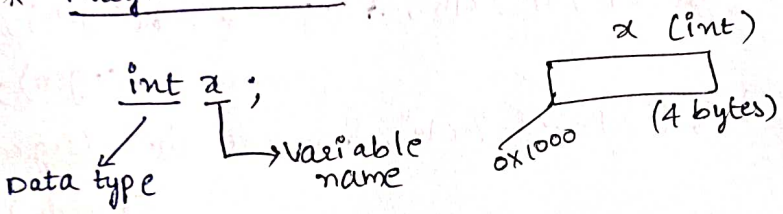
⇒ structure definition does not occupy any space/memory in an executable file

⇒ It is used by the compiler for verification purpose.

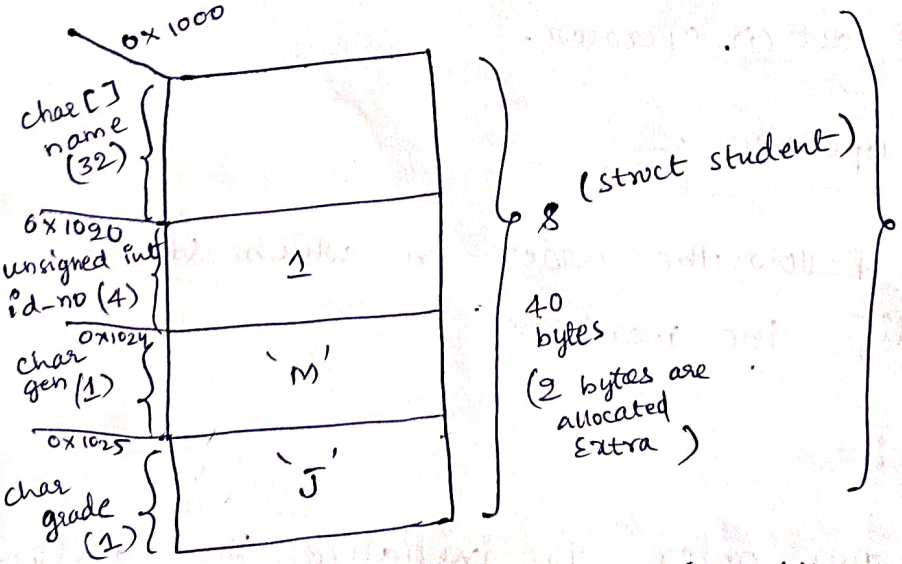
(*)
⇒ while defining the structure, we can't initialize the members inside the definitions.

⇒ we can access the structure members with the help of variable name by using dot (.) operator.

* integer variable



⇒ main( )
{
struct student s ;
Data type
Variable name



(same o/p) { printf("%d\n", sizeof(s));
printf("%d\n", sizeof(struct student));

Printf("%p\n", &s);

To read

32 bytes starting base address.

s.id-no = 1;

s.gen = 'M';

s.grade = 'J';

X (s.name = "kumar";)

s.name[0] = 'k';
s.name[1] = 'u';
s.name[2] = 'm';
s.name[3] = 'a';
s.name[4] = 'r';
s.name[5] = '\0';

(or) scanf("%s", s.name); // strcpy(s.name, "kumar");

(or) scanf("%d", &s.id-no);

(or) scanf("%c", &s.gen);

(or) scanf("%c", &s.grade);

To print

printf("%p-%d-%s\n", s.name, sizeof(s.name), s.name);

printf("%p-%d-%d\n", &s.id-no, sizeof(s.id-no), s.id-no);

Printf("%p-%d-%c\n", &s.gen, sizeof(s.gen), s.gen);

Printf("%p-%d-%c\n", &s.grade, sizeof(s.grade), s.grade);

✱ Structure Variable initialization :-

⇒ Structure Variable can be initialized in 2 ways:

① with using dot (.) operator &

② without using dot (.) operator.

✱✱ without using dot (.) operator :-

⇒ In this we have to follow the order in which the structure is defined to initialize the members.

✱ with dot (.) operator :-

⇒ No need to follow any order to initialize the structure members.

⇒ Need to follow order.

(I) Struct student s = { "kumar", 1, 'M', 'J' };

(II) Struct student s = { .id_no = 1, .grade = 'J', .gen = 'M', .name = "kum ar" };

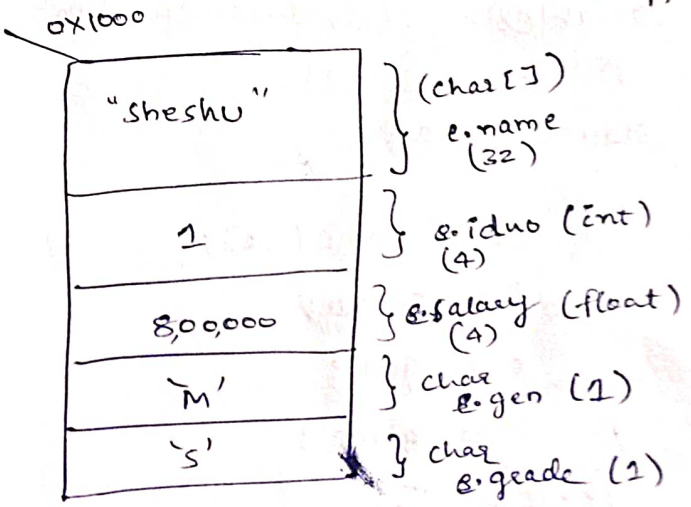
⇒ No need to follow order

⇒ Mostly used in kernal programming

⇒ **Update with new details**

```
struct employee
{
    char name[32];
    unsigned int id_no;
    float salary;
    char gen;
    char grade;
};
```

```
main()
{
struct employee e = {"sheshu", 1, 80000, 'M', 'S'};
```




```
PF("%p-%p-%p-%p-%p\n", e.name, &e.id_no, &e.salary, &e.gen, &e.grade);
```

```
PF("%s-%d-%f-%c-%c\n", e.name, e.id_no, e.salary, e.gen, e.grade);
```

To modify or update

manually

```
e.name[0] = 's';
e.name[1] = 'a';
e.name[2] = 'k';
e.name[3] = 'e';
e.name[4] = 't';
e.name[5] = 'h';
e.name[6] = '\0';
```

(or)

```
scanf("%s", e.name);
```

(or)

```
strcpy(e.name, "saketh");
```

```
e.id_no = 2;   // scanf("%d", &e.id_no);
e.salary = 10,00,000;   // scanf("%f", &e.salary);

PF("%p-%p-%p-%p\n", e.name, &e.id_no, &e.salary, &e.gen, &e.grade);
PF("%s-%d-%f-%c-%c\n", e.name, e.id_no, e.salary, e.gen, e.grade);
}
```

28/11/22

⇒ write a program to swap the content of two structure variables by using swap func^c, without swap func^c.

⇒
```
struct student
{
    char name[32];
    int id_no;
    char gen;
    char grade;
};

void main()
{
    struct student s1 = {"Ramu", 1, 'M', 'A'};
    struct student s2 = {"kumar", 2, 'M', 'A'};
    struct student temp;   // char str_temp[32];
                           // int temp_id_no;
                           // char temp_gen;
                           // char temp_grade;
```

All are of same data type.

```
temp = s1;
s1 = s2;
s2 = temp;
```

(or)

⇒ strcpy(temp.name, s1.name);
strcpy(s1.name, s2.name);
strcpy(s2.name, temp.name);

temp.id_no = s1.id_no;
s1.id_no = s2.id_no;
s2.id_no = temp.id_no;

temp.gen = s1.gen;
s1.gen = s2.gen;
s2.gen = temp.gen;

temp.grade = s1.grade;
s1.grade = s2.grade;
s2.grade = temp.grade.

// strcpy(str_temp, s1.name);
strcpy(s1.name, s2.name);
strcpy(s2.name, str_temp);

temp_id_no = s1.id_no;
s1.id_no = s2.id_no;
s2.id_no = temp_id_no;

temp_gen = s1.gen;
s1.gen = s2.gen;
s2.gen = temp_gen;

temp_grade = s1.grade;
s1.grade = s2.grade;
s2.grade = temp_grade;

PF( );
PF( );

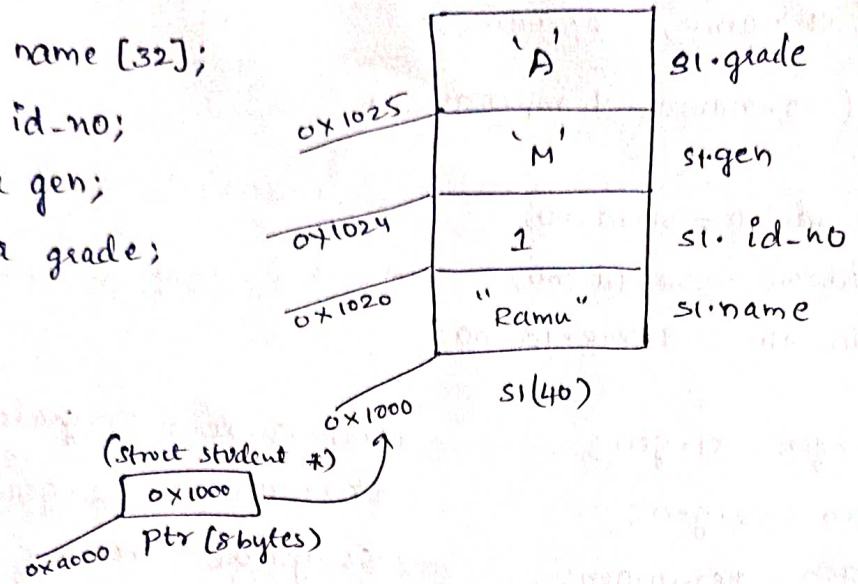
* **Structure pointer** :-

1. creating a structure pointer of same type
2. Assigning structure variable's base address of same type
3. Accessing structure members by using structure pointer with the help of arrow (→) operator.

* ⇒ →

⇒ struct student

```
{
    char name[32];
    int id_no;
    char gen;
    char grade;
};
```




```
main()
{
    struct student s1 = {"Ramu", 1, 'M', 'A'};
    struct student *ptr;
    ptr = &s1;
```

ptr = &s1; → 0xa000

```
PF("%p-%s-%d\n", ptr->name, ptr->name, sizeof(ptr->name));
PF("%p-%d-%d\n", &ptr->id_no, ptr->idno, sizeof(ptr->id_no));
PF("%p-%c-%d\n", &ptr->gen, ptr->gen, sizeof(ptr->gen));
PF("%p-%c-%d\n", &ptr->grade, ptr->grade, sizeof(ptr->grade));
}
```

* Call by value Mechanism :-

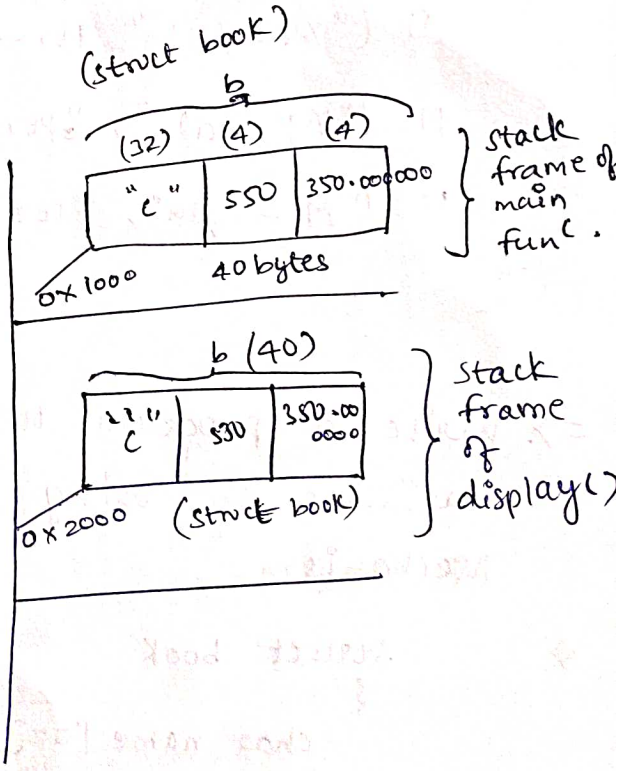
General example

```
main()
{
  int x = 10;
  display (x);
}

void display ( int y )
{
   PF (" %d\n", y);
}
```

⇒
```
struct book
{
   char name[32];
   int nop;
   float price;
};

main()
{
   struct book b = { "c", 550, 350 };
   display (b);
}
```

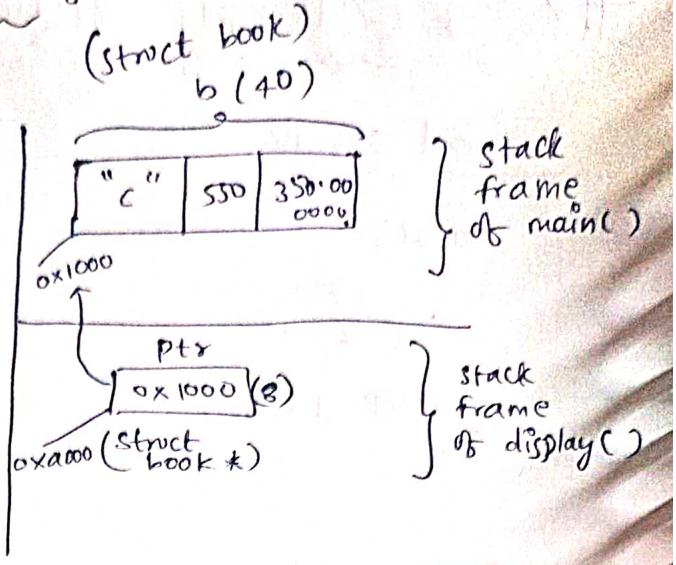
→ content of variable (b)

```
void display (struct book b)
{
   PF (" %s\n", b.name);
   PF (" %d\n", b.nop);
   PF (" %f\n", b.price);
}
```



* Call by Reference Mechanism :-

```
struct book
 {
    char name[32];
    int nop;
    float price;
 };

main()
{
   struct book b = {"c", 550, 350};
   display (&b);
}

void display (struct book *ptr)
 {
   PF ("%p-%s\n", ptr→name, ptr→name);
   PF ("%p-%d\n", &ptr→nop, ptr→nop);
   PF ("%p-%f\n", &ptr→price, ptr→price);
 }
```



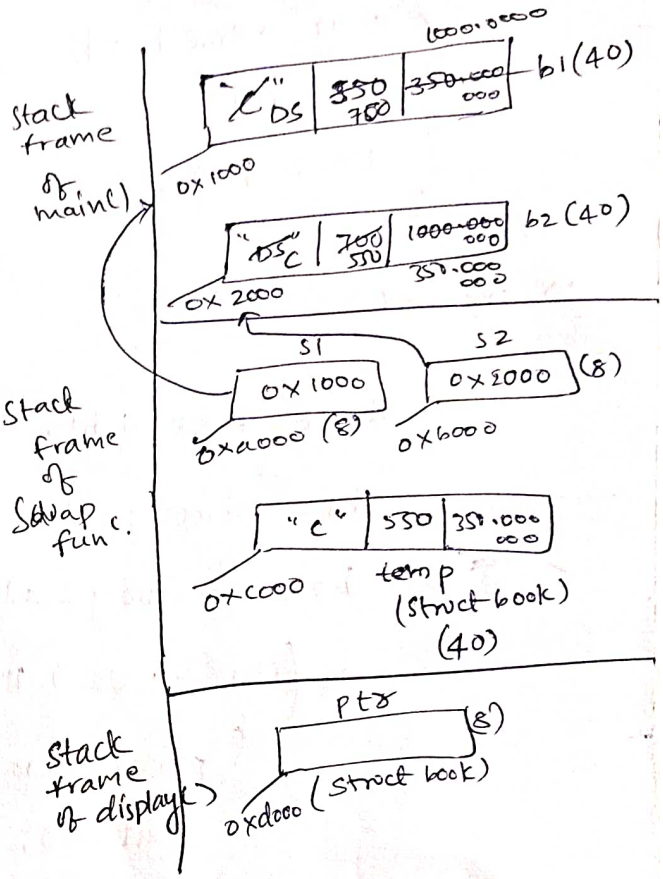
↑ passing the address of the variable

↑ receiving the address of b variable.

⇒ write a program to swap the content of two structure variables by using swap func^c through call by reference Mechanism.

⇒
```
struct book
 {
    char name[32];
    int nop;
    float price;
 };
```

```
main()
{
    struct book b1 = {"C", 550, 350};
    struct book b2 = {"DS", 700, 1000};
    swap (&b1, &b2);
    display (&b1);
    display (&b2);
}
```




```
⇒ void swap (struct book &s1, struct book *s2)
{
    struct book temp;
    strcpy (temp.name, s1→name);
    strcpy (name, s2→name);
    strcpy (s2→name, temp.name);

    temp.nop = s1→nop;
    s1→nop = s2→nop;
    s2→nop = temp.nop;

    temp.price = s1→price;
    s1→price = s2→price;
    s2→price = temp.price;
}
```

```
⇒ void display (struct book *ptr)
{
    PF ("%p - %s\n", ptr→name, ptr→name);
    PF ("%p - %d\n", &ptr→nop, ptr→nop);
    PF ("%p - %f\n", &ptr→price, ptr→price);
}
```

⇒ write a program to allocate memory dynamically using fun^c, swap fun^c, display fun^c & memory deallocation fun^c.

⇒
```
struct book
{
    char name[32];
    int nop;
    float price;
};
```

⇒
```
main()
{
    struct book *b1, *b2;

    b1 = memory_alloc ();

    b2 = memory_alloc ();

    if ((b1 == NULL) || (b2 == NULL))
    { printf ("failed to allocate memory in heap \n");
      exit (1);
    }

    read (b1);
    read (b2);

    swap (b1, b2);

    display (&b1);
    display (&b2);

    memory_dealloc (b1);
    memory_dealloc (b2);
}
```

→
```
struct book * memory_alloc ( )
{
    struct book *ptr = (struct book *) malloc (1 * sizeof (struct book));

    return ptr;
}
```

⇒ void read (struct book *ptr)

```
{
    PF ("enter book name\n");
    SF ("%s", ptr->name);
    --Fpurge (stdin);
    PF (" enter the nop \n");
    SF (" %d", &ptr->nop);
    --Fpurge (stdin);
    PF ("Enter the price\n");
    SF ("%f", &ptr->price);
    --fpurge (stdin);
}
```

⇒ void swap (struct book *s1, struct book *s2)

```
{
    struct book temp;
    strcpy (temp.name, s1->name);
    strcpy (s1->name, s2->name);
    strcpy (s2->name, temp.name);

    temp.nop = s1->nop;
    s1->nop  = s2->nop;
    s2->nop  = temp.nop;

    temp.price = s1->price;
    s1->price = s2->price;
    s2->price = temp.price;
```

⇒ void display (struct book *ptr)

```
{
    PF ("%p - %s\n", ptr->name, ptr->name);
    PF ("%p - %d\n", &ptr->nop, ptr->nop);
    PF ("%p - %f\n", &ptr->price, ptr->price);
}
```

⇒ void memory_dealloc (struct book *ptr)

```
{
    free (ptr);
}
```

29/11/22

⇒ some-times in Books they will refer Objects which are ntg but structures

⇒ Tables are ntg but structure arrays

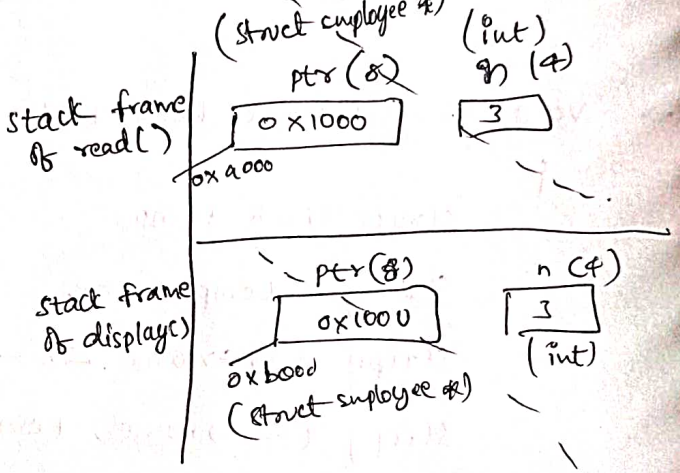
⇒ Buffers are ntg but character array.

(*). ⇒ →

[ ]. ⇒ →

* Structure Arrays :-

```
struct employee
{
    char name[32];
    int id;
    float salary;
    char grade;
    char gen;
};
```



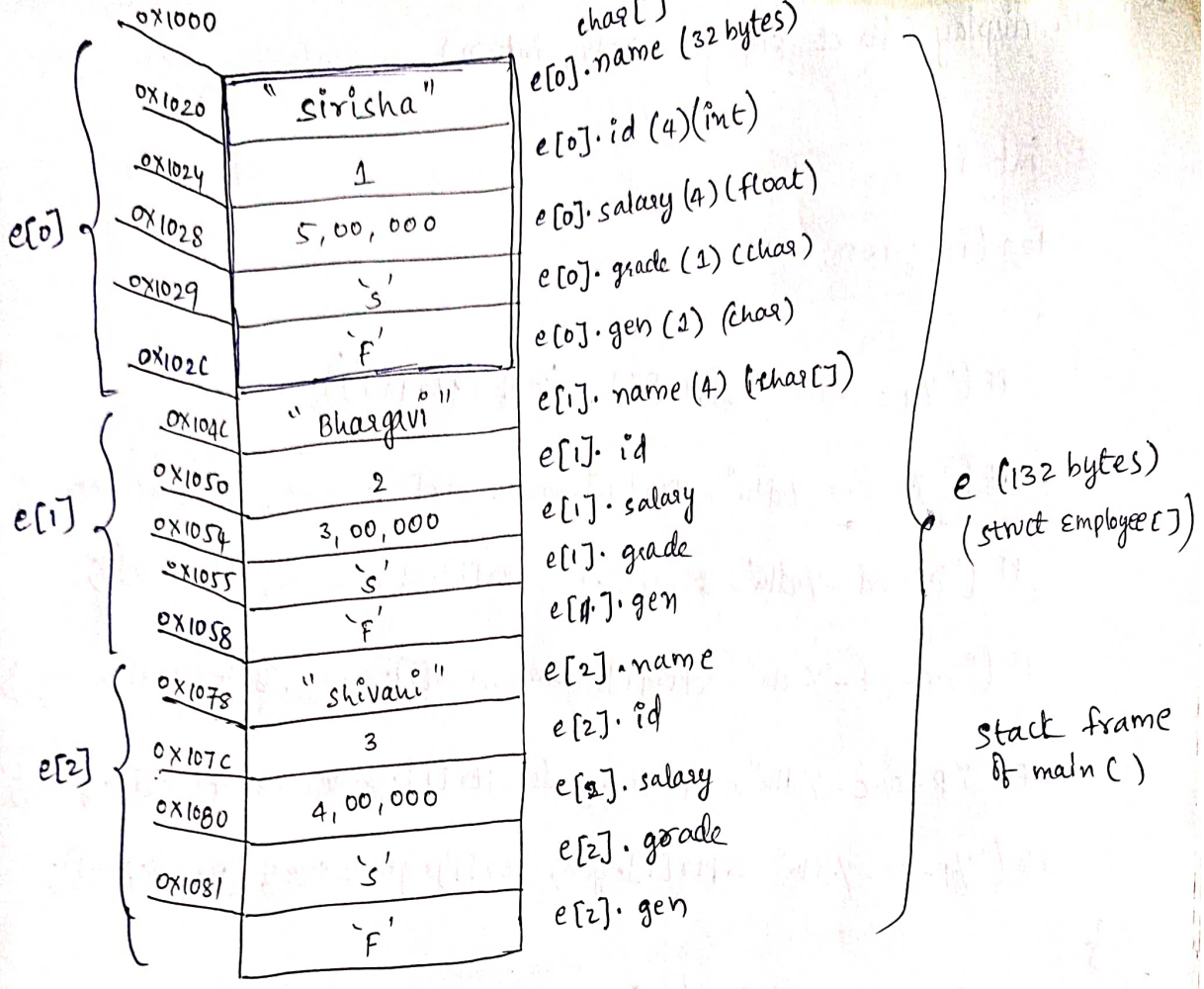
⇒
```
main()
{
    struct employee e[3];   // to request dynamically
    PF("%p - %d\n", e, sizeof(e));
    read(e,3);
    display(e,3);
}
```

index values (0,1,2)

↑ ⟶ no. of employees

data type — e ↓ array name

```
main()
{ struct employee *ptr = (struct employee *)malloc
                         (3*sizeof(struct employee));
  if(ptr == NULL)
  { PF("failed to allocate memory\n");
    exit(1);
  }
  read(ptr,3);
  display(ptr,3);
  free(ptr);
}
```



(0x1000) { 132 bytes array base add ③

⇒ void read (struct employee *ptr, int n)

```
{
    int i;
    for (i=0; i<n; i++)
    {
        PF("enter employee name:\n");
        SF("%s", ptr[i].name);
        --fpurge(stdin);
        PF("enter id:\n");
        SF("%d", &ptr[i].id);
        --fpurge(stdin);
        PF("enter salary\n");
        SF("%f", &ptr[i].salary);
        --fpurge(stdin);
        PF("enter grade:\n");
        SF("%c", &ptr[i].grade);
        --fpurge(stdin);
        PF("enter gender\n");
        SF("%c", &ptr[i].gen);
}
```

⇒ void display (struct Employee *ptr, int n)

```
{
    int i;
    for (i=0; i<n; i++)
    {
      PF("%p-%d\n", &ptr[i], sizeof(ptr[i]));
      PF("%p-%s-%d\n", ptr[i].name, ptr[i].name, sizeof(ptr[i].name));
      PF("%p-%d-%d\n", &ptr[i].id, ptr[i].id, sizeof(ptr[i].id));
      PF("%p-%f-%d\n", &ptr[i].salary, ptr[i].salary, sizeof(ptr[i].salary));
      PF("%p-%c-%d\n", &ptr[i].grade, ptr[i].grade, sizeof(ptr[i].grade));
      PF("%p-%c-%d\n", &ptr[i].gen, ptr[i].gen, sizeof(ptr[i].gen));
    }
}
```

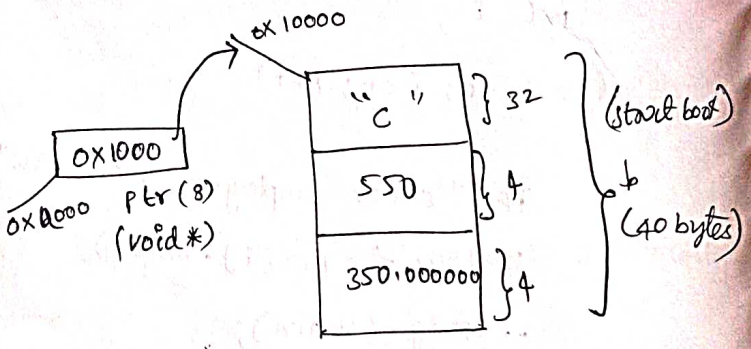
⊛ Void Pointer ⊛

⇒ How do you access structures by using Void pointers :-

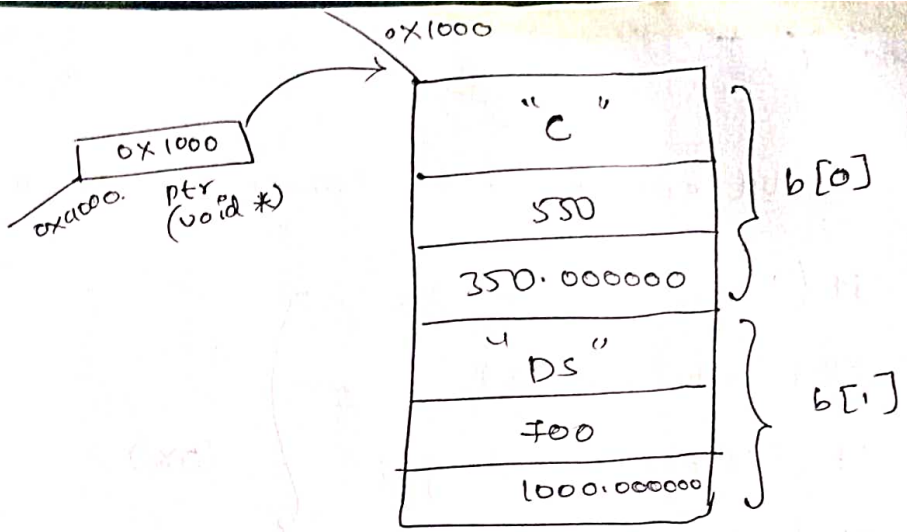
```
struct book
{
  char name[32];
  int nop;
  float price;
};

main()
{
  struct book b={"c", 550, 350};
  void *ptr;
  ptr=&b;
```

⇒ (or) `void *ptr = &b;`

```
  PF("%s\n", ((struct book*)ptr)->name);
  PF("%d\n", ((struct book*)ptr)->nop);
  PF("%f\n", ((struct book*)ptr)->price);
}
```



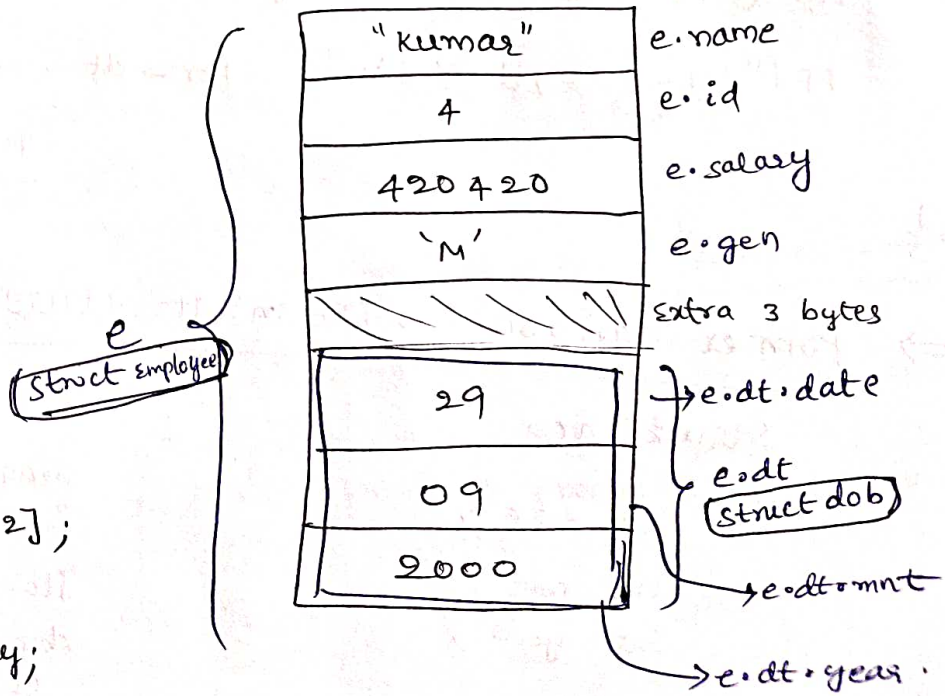
⇒ Array structure




```
struct book
{
    char name[32];
    int nop;
    float price;
};

main()
{
    int i;
    struct book b[2] = { {"C", 550, 350}, {"DS", 700, 1000} };
    void *ptr = b;
    for (i=0; i<2; i++)
    {
        PF("%.s\n", ((struct book *) ptr)[i].name);
        PF("%d\n", ((struct book *) ptr)[i].nop);
        PF("%f\n", ((struct book *) ptr)[i].price);
    }
}
```

⊛ structure within structure :-

```
struct dob
{
    int date;
    int mnt;
    int year;
};

struct employee
{
    char name[32];
    int id;
    float salary;
    char gen;
    struct dob dt;
};
```



```
main()
{
    struct employee e={"kumar", 4, 420420, 'M', 29, 09, 2000};

    PF("%s\n", e.name);
    PF("%d\n", e.id);
    PF("%f\n", e.salary);
    PF("%c\n", e.gen);
                                    (or)
    PF("%d-%d-%d\n", e.dt.date, e.dt.mnt, e.dt.year);

                    (or)
      display(&e);

}
```

```
⇒ void display ( struct employee *ptr)
  {
      Printf("%p - %s\n", ptr→name, ptr→name);
      PF("%p - %d\n", &ptr→id, ptr→id);
      PF("%p - %f\n", &ptr→salary, ptr→salary);
      PF("%p - %c\n", &ptr→gen, ptr→gen);
      PF("%p - %d - %d - %d\n", &ptr→dt, ptr→dt.date, ptr→dt.mnt,
                                ptr→dt.year);
  }
```

arrow operator to access structure pointer

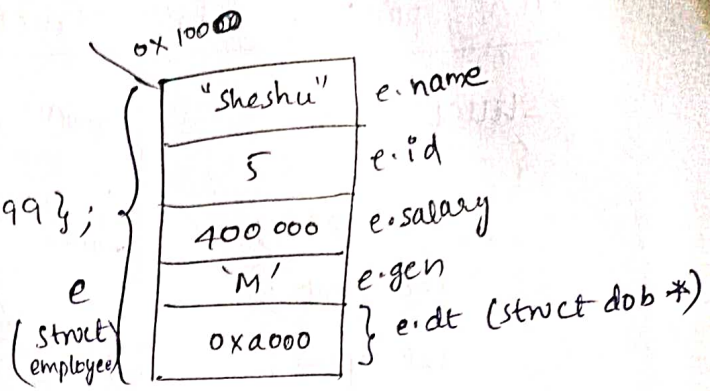
dot operator to access structure variable

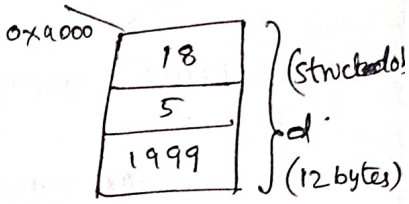
⇒ **Pointer Variable with in structure Variable :-**

```
struct dob
{
    int date;
    int mnt;
    int year;
};
```

```
struct employee
{
    char name[32];
    int id;
    float salary;
    char gen;
    struct dob * dt;
};
```

```
main()
{
    struct Employee e;
    struct dob d = {18, 5, 1999};
    strcpy (e.name, "sheshu");
    e.id = 5;
    e.salary = 400000;
    e.gen = 'M';
    e.dt = &d;
```






```
    PF ("%s\n", e.name);
    PF ("%d\n", e.id);
    PF ("%f\n", e.salary);
    PF ("%c\n", e.gen);
    PF ("%d-%d-%d\n", e.dt->date, e.dt->mnt, e.dt->year);
(or)
    display (&e);
}
```

⇒
```
void display (struct employee *ptr)
{
    PF ("%p-%s\n", ptr->name, ptr->name);
    PF ("%p-%d\n", &ptr->id, ptr->id);
    PF ("%p-%f\n", &ptr->salary, ptr->salary);
    PF ("%p-%c\n", &ptr->gen, ptr->gen);
    PF ("%p-%d-%d-%d\n", &ptr->dt, ptr->dt->date, ptr->dt->mnt,
                                   ptr->dt->year);
}
```

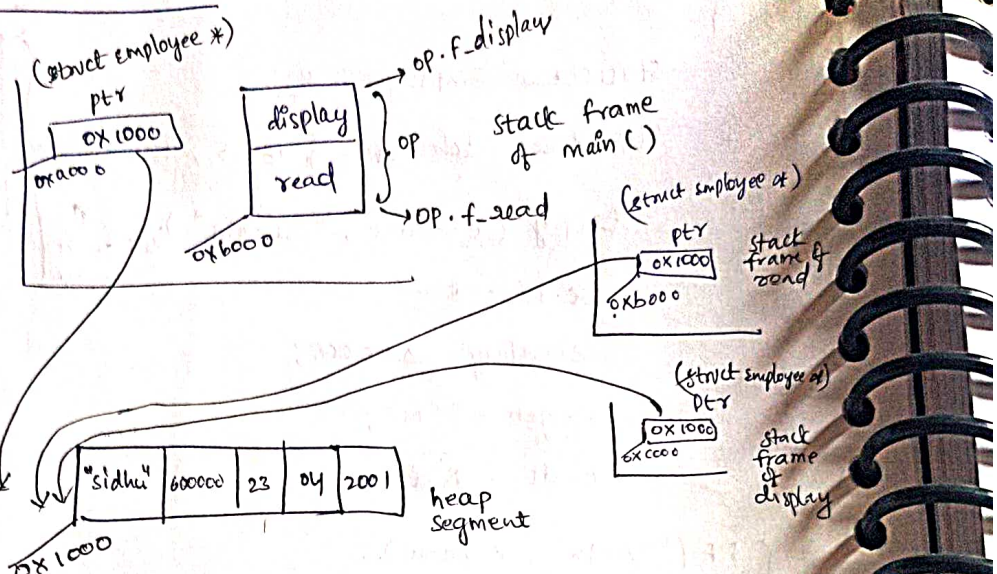
↓

arrow operator to ~~struct~~ access structure variable through structure pointer of struct employee type

* Fun^c pointers in structures :-

```
struct dob
{
    int date;
    int mnt;
    int year;
};

struct employee
{
    char name[32];
    float salary;
    struct dob dt;
};

struct employee_ops
{
    void (*f_read)(struct employee *);
    void (*f_display)(struct employee *);
};
```

} fun^c pointers



⇒
```
main()
{
    struct employee *ptr = (struct employee *) malloc(1 * sizeof(struct employee));
```

0x1000

```
    if (ptr == NULL)
        exit(1);

    struct employee_ops op;
```

① creating a fun^c pointer

```
    op.f_read = read;
    op.f_display = display;
```

② Assigning fun^c's base address

```
    op.f_read(ptr);
    op.f_display(ptr);
```

③ fun^c invocation by using fun^c pointer.

```
    free(ptr);
}
```

⇒ Void read (struct employee *ptr)

```
{
    PF ("Enter employee name \n");
    SF ("%s", ptr → name);
    -- fpurge (stdin);
    PF ("Enter salary \n");
    SF ("%F", &ptr → salary);
  -- Fpurge (stdin);
    PF ("Enter date /mnt/ year \n");
    SF ("%2d - %2d - %4d", & ptr → dt . date, &ptr → dt . mnt,
                                    & ptr → dt . year);
    -- fpurge (stdin);
}
```

⇒ void display (struct employee *ptr)

```
{
    PF ("%p - %s \n", ptr → name, ptr → name);
    PF ("%p - %f \n", &ptr → salary, ptr → salary);
    PF ("%p - %d \n", & ptr → dt, ptr → dt . date, ptr → dt . mnt,
                                    ptr → dt . year);
}
```

⇒ 30/11/22 * Structure Size *

⇒ The latest compilers by default follows structure padding or byte padding to decide the size of the structure

Because to access the structure members, CPU will take less no. of machine cycles that will improve the o/p efficiency (ntg but performance).

⇒ Structure padding will also depends upon machine type.

⇒ The size of structure varies from 32 bit to 64 bit machine.

32 bit machine → 32 bit compiler
64 bit machine → 64 bit compiler.

⇒ The wasted (or) unused bytes (extra bytes) of memory, in structure padding while allocating memory is known as "holes".

(*) How do you avoid the holes in structures?

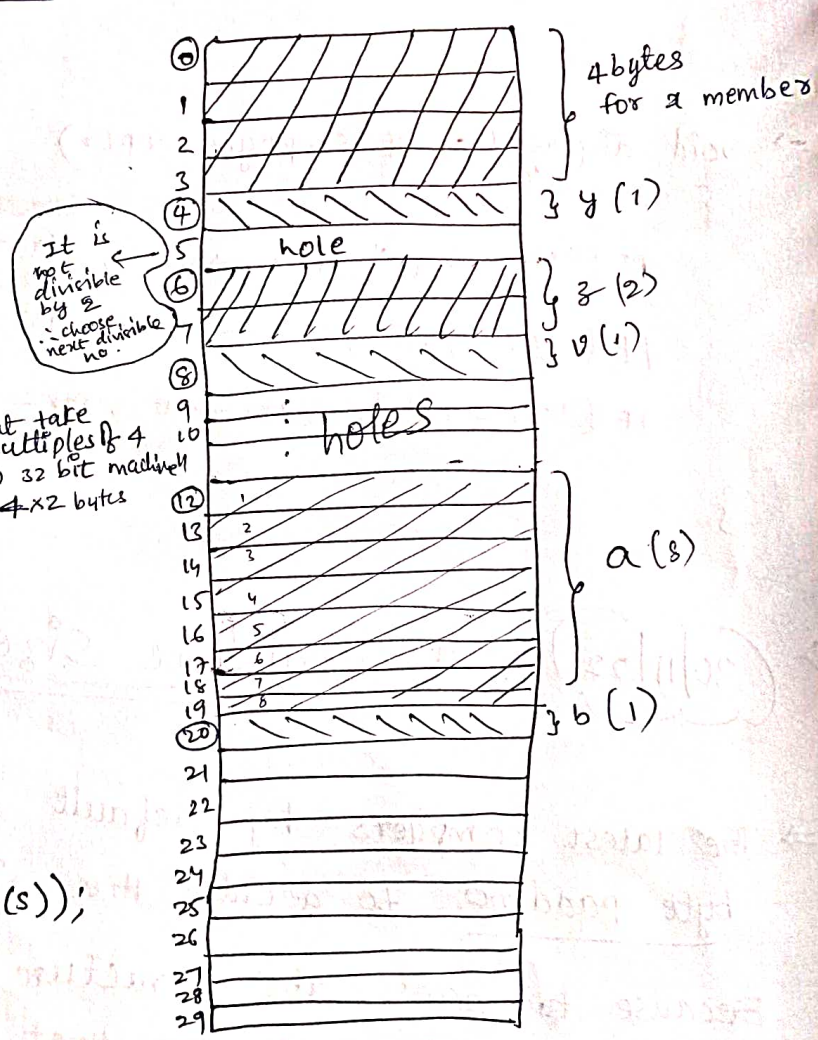
(A) By using ① `#pragma pack(1)` after inclusion of header files

②
```
struct s
{
    int i;
    char ch;
    double d;
} __attribute__((packed));
```

holes can be avoided.

(*) 32 bit machine (*)

```
struct size
{
    int x;    → 4 bytes
    char y;   → 1  "
    short z;  → 2  "
    char v;   → 1  "
    double a; → 8 → But take multiples of 4 in 32 bit machine 4×2 bytes
    char b;   → 1 bytes
};

main()
{
    struct size s;
    printf("%d\n", sizeof(s));
}
```



overall size = 21 bytes (it is not divisible by highest mem size

i.e., double = 8 ⇒ 4×2

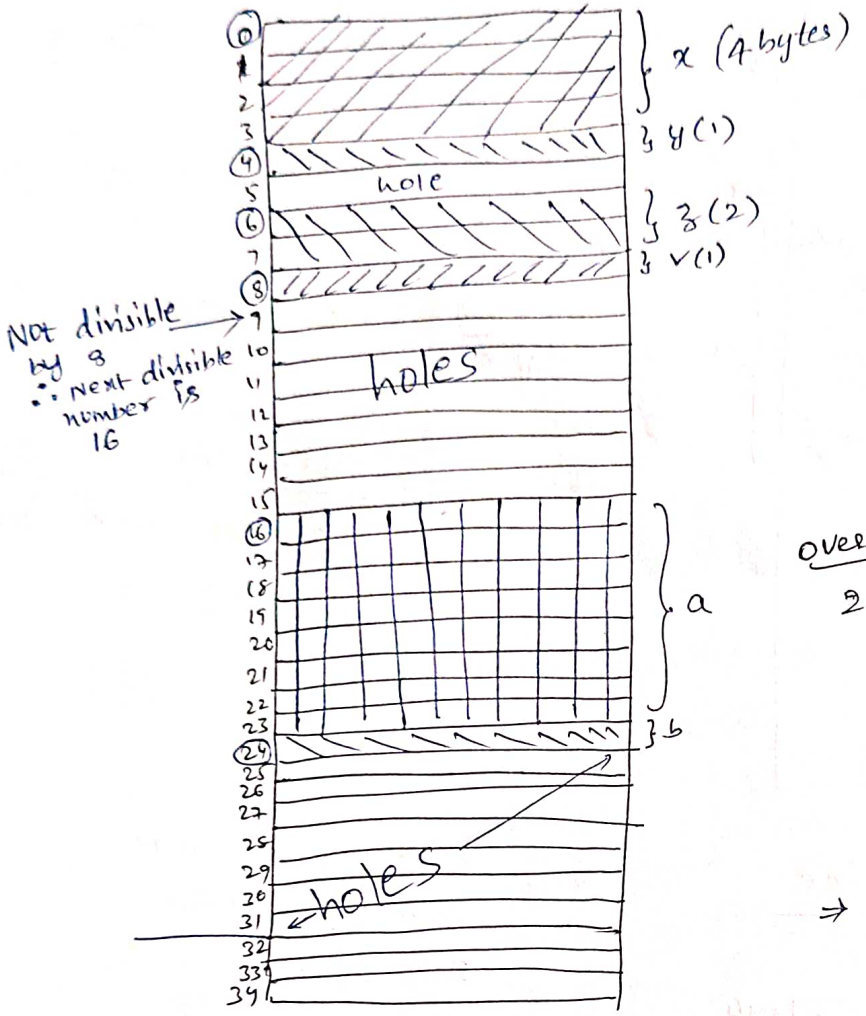
$\frac{21}{4}$ = Not divisible

∴ $21+3 = \frac{24}{4}$ = divisible

∴ overall size = 24 bytes

(*) 64 bit Machine (*)

```
struct size
{
    int x;    4 bytes
    char y;   1 "
    short z;  2 "
    char v;   1 "
    double a; 8 bytes (for 64 bit machine we consid 8 bytes directly
    char b;   1 "
};
```



overall
25 bytes (Not divisible by highest byte member)

25/double a

= 25/8 [Not divisible]

⇒ 25 + 7 = 32/8 [Divisible]

∴ overall size = 32 bytes

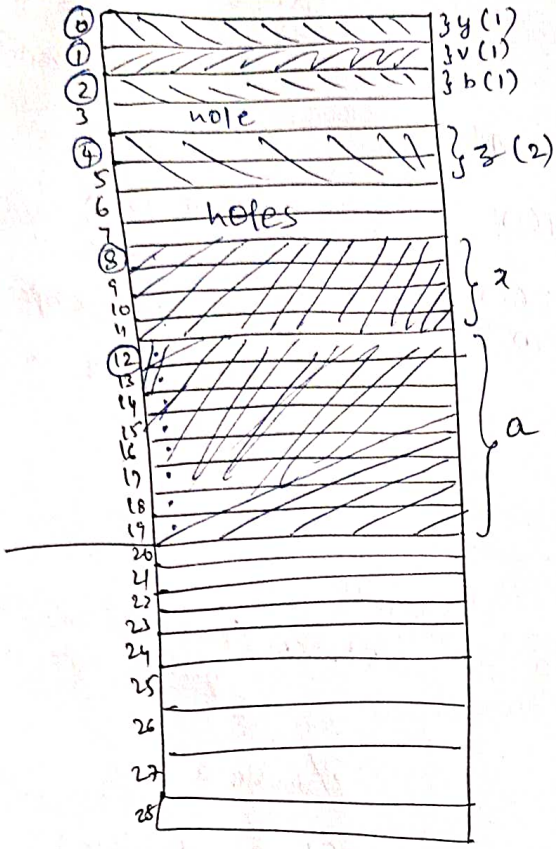
(**)
⇒ These holes can be minimized by rearranging the members

⇒ struct size (for 32 bit machine)

```
{
    char y;   1 byte
    char v;   1 "
    char b;   1 "
    short z;  2 "
    int x;    4 "
    double a; 8 " (But we consider multiples of 4 as it is 32 bit machine)
};
```

↳ 4×2

32 bit



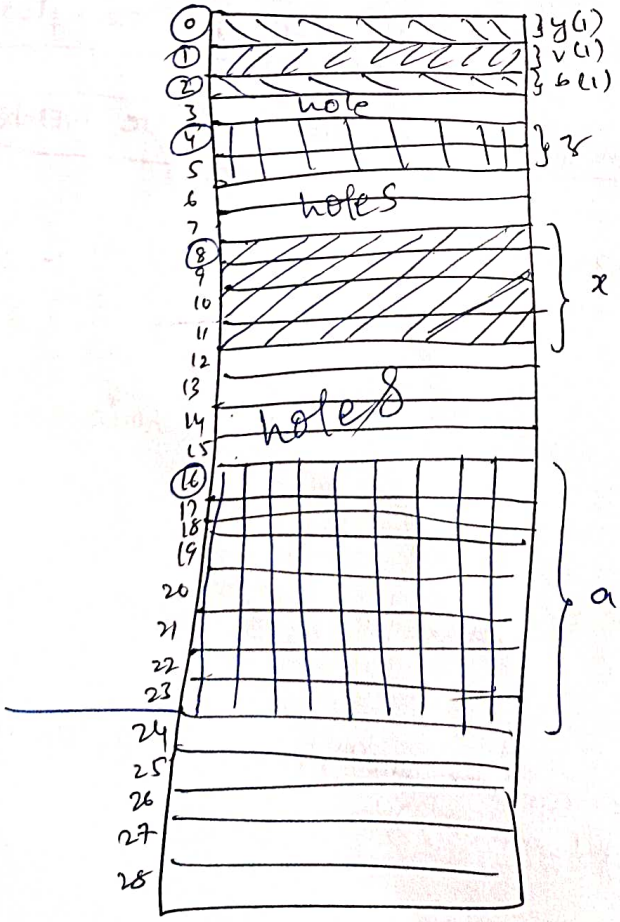
20 bytes = divisible

a (double) = 8 → 4

∴ overall size = 20 bytes

4 holes are reduced

64 bit machine



24 bytes (Divisible)

a (double) = 8 → 8

∴ overall size = 24 bytes

8 holes are reduced

```
struct size
{
    char y;  → 1 byte
    char v;  → 1 "
    char b;  → 1 "
    short z; → 2 "
    int x;   → 4 "
    double a; → 8 bytes
};
```

01/12/22

* Bit fields :-

⇒ Bit fields are created as the members of the structure.

⇒ when we use bit fields, the memory is allocated in terms of bits not bytes for bit fields.

⇒ we cannot apply storage classes for structure members.

⇒ But storage classes are applicable for structure variables, arrays etc.

⇒ sign Qualifiers are applicable for structure members.

```
const struct employee e = {"kumar", 1, 50,000, 'M', 'S'};
```

structure variable applied with const keyword declared globally & initialized it goes to ro.data segment. e can be found in symbol table & complete info can be found in ro.data segment.

⇒ we can't access the address of a bit field because it is byte addressable.

⇒ <sign Qualifiers> <data type> <Bit field name> : <no. of bits> ;

NOTE :-

⇒ In bit fields, bit field size should not exceed the size of data type.

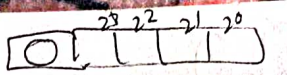
```
char x : 9 ;   ✗   char = 1 byte = 8 bits   (9 is more than 8 so it cannot exceed 8 bits)

Char x : 8 ;   ✓   char = 1byte = 8bits   (size should be less or equal to 8
```

⇒ we can't use scanf fun^c on bit fields (As it is unable to access the address of bit field variable).

date → 1 to 31

month → 1 to 12

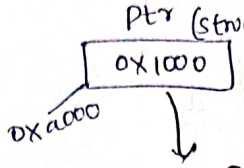
year → 4 digits



int date : 5; → -16 to +15

Unsigned int date : 5; → 0 to $2^5 - 1$

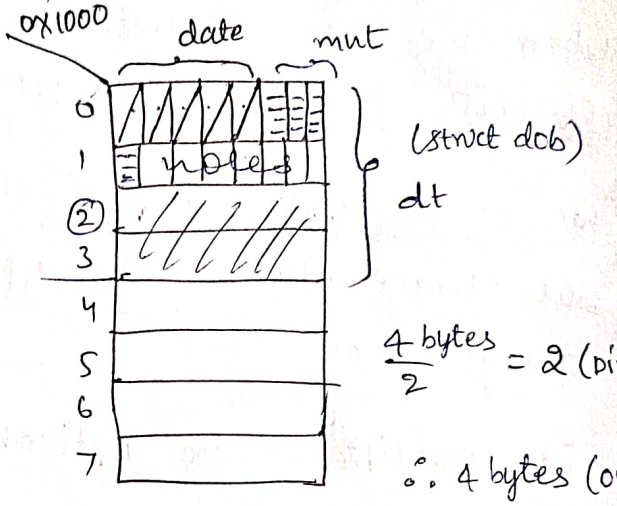
0 to 31.



⇒ struct dob

```
{
    unsigned int date : 5 ;
    unsigned int mnt : 4 ;
    unsigned short int year ;
};
```

②




```
main ()
{
    struct dob dt ;
    dt.date = 29;
    dt.mnt = 09;
    dt.year = 2000;
    struct dob *ptr = &dt;
    PF ("%p\n", &dt );
    PF ("%p\n", ptr);
    PF ("%d\n", ptr->date);
    PF ("%d\n", ptr->mnt);
    PF ("%d\n", ptr->year);
}
```

⇒ we can't print bit field variables directly. so, we are assigning the address of that variable to the pointer

⇒ 0X1000

0X1000

29

09

2000

* Data Structures *

⇒ Arranging the data in a particular format on which we are performing certain operations. that comes under data structures.

⇒ Data structures are constructed on abstract data types (user defined data types).

* Abstract data type :-

⇒ It is a mathematical concept based user defined data type or general user defined data type on which we are performing certain operations.

① Premitive data types = int, char, float, double etc.

② Non-premitive data types = user defined data types. (structures)

⇒ These abstract data types (abt) are.
Self referencial structures (a ~~structure~~ pointer variable of same type of structure is a member of that structure).

⇒
```
Struct employee
{   char name[32];
    int id;
    float salary;
    Struct employee *ptr ⟶ pointer with same data type should be there
};
```

⇒ Now it is called as self-referencial structure.

⇒ Data structures are of two types :-

① Linear DS

② Non-linear DS.

① In linear data structure, data is arranged linearly

② In Non-linear data structure, data is arranged non-linearly. (which is not arranged in linear).

⇒ Examples of linear DS :- Linked list (single & double), stack & Queue.

⇒ Examples of Non-linear DS :- Trees & graphs

⇒ whether it may be any type of data structure on which we are performing certain operations commonly they are insertion, deletion, sort, display, travel, search etc..

⇒ If we want to write or perform these operations we have to follow some set of rules (algorithms).

↳ step by step process to solve prblm.

⇒ Algorithm defines step by step approach to solve particular prblm or to find a solution for a particular problem.

⇒ ① Linear Data structures :-

① single linked list :- It is a collection of nodes.

⇒ Each node will have a data part & a link part.

⇒ Each link part contains next node base address. so, this collection of single nodes we will construct dynamically.

* structure definition for single linked list

```
struct node
{
    int data;
    struct node * link;
};
```

* operations on single linked list :-

1. add at beginning
2. delete at beginning
3. add at last
4. delete at last
5. display
6. search
7. delete particular node
8. delete list
9. create list
10. delete duplicate nodes
11. swap nodes (exchange by data, exchange by links)
12. add after a node
13. add before a node
14. sorting { selection, Bubble, Merge } → exchange by data / Exchange by link
15. reverse the list.
16. count.

9t is preffered when data is less

preffered when data is more

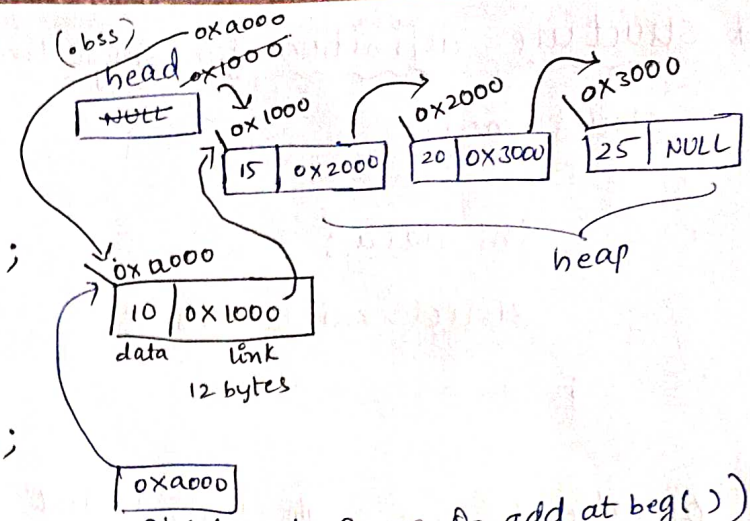
⇒ In single linked list, we should have one head pointer either globally or locally which will point the first element (or) node of the list.

⇒ when list is empty, head pointer should contain NULL.

⇒ when we are declaring 1st time, head pointer must be initialized with NULL.

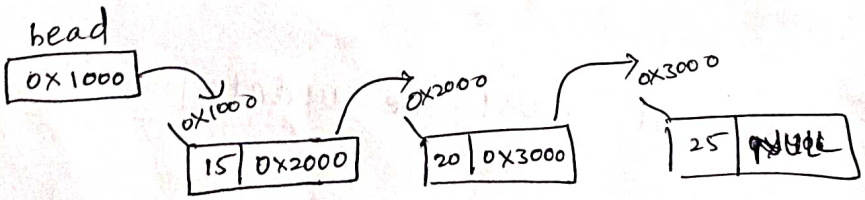
⇒ Last node link part must be updated with NULL.

⇒ struct node
{
int data;
struct node *link;
};



struct node *head = NULL;
main()
{
int num, pos, opt; int c, n;
while(1)
{
PF("0.exit \n 1. add at beg \n 2. delete at beg \n ----- \n");
SF("%d", &opt);
--fpurge(stdin);
switch(opt)
{
case 0: exit(0);
case 1: PF("Enter the data \n");
SF("%d", &num);
add at beg(num);
break;
case 2: delete at beg( );
break;

case-2



case 3: display( );
break;
case 4: c = count( );
PF("%d", c);
break;

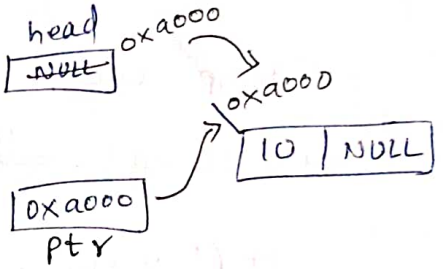
⇒ void addatbeg (int num)

```
{
    struct node *ptr = (struct node *) malloc (1*sizeof (struct node));
        if (ptr == NULL)
        {
            PF (" Failed to allocate memory in heap \n");
            exit (1);
        }
    ptr → data = num;
    ptr → link = head;
    head = ptr;
}
```

(corner cases if any)

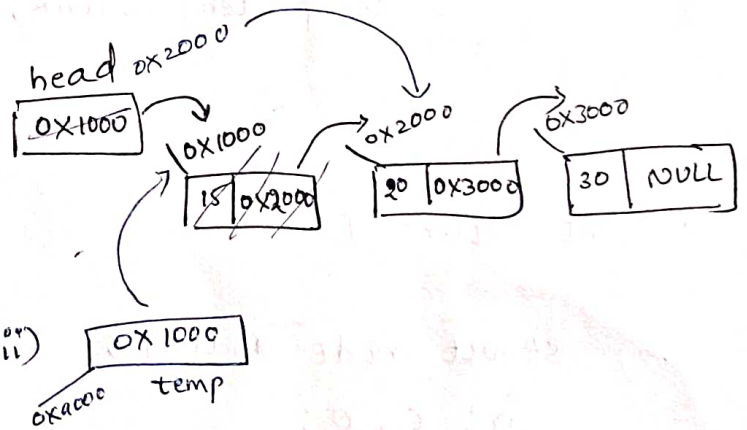
check corner cases

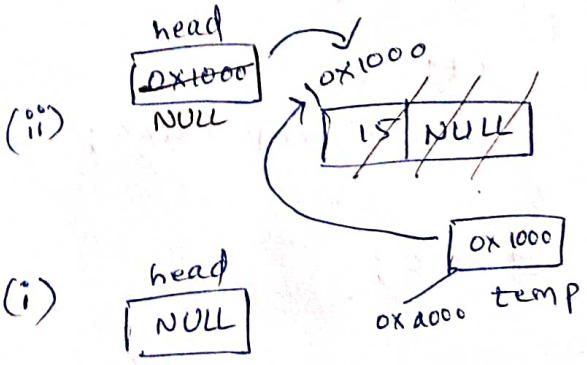
(i) list is empty



⇒ void deleteatbeg ( )

```
{
    struct node *temp;
case-1  if (head == NULL)
        {
            PF (" List is Empty");
            return;
        }
(case-2,3)  temp = head;
    head = head → link;
    free (temp);
}
```

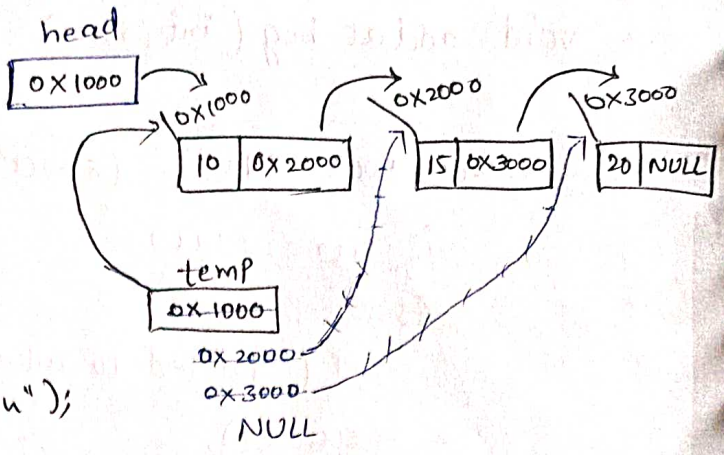




case 5 : pf (" enter no to be searched \n");
sf ("%d", &n);
pos = search (n);
break;

⇒ void display ( )

```
{
    struct node *temp;
    // list is empty :
    if (head == NULL)
    {
        PF(" list is empty\n");
        return;
    }
    temp=head;
  while (temp != NULL)
    {
      PF (" %d\n", temp→ data);
        temp = temp → link;
    }
}
```




```
10
15
20
```

⇒ int count ( )

```
{
  struct node * temp;
   int c=0;
   temp = head;
   while (temp != NULL)
   {
      c++;
      temp = temp → link;
   }
   return c;
}
```

⇒ int search (int num)

```
{
    int pos =0;
    struct node *temp;
    if (head == NULL) {
    {
      PF(" list is empty\n");
      return pos;
    }
    temp = head;
  while (temp != NULL)
    {
      pos++;
      if (temp → data == num)
        return pos;
      temp = temp → link;
    }
  PF(" Element not found\n");
  return 0;
}
```

```
case 6 : delete_entire_list ( );
         break;
```

⇒
```
void delete_entire_list ( )
{
    struct node * temp;
    if (head == NULL)
    {
        PF ("List is empty \n");
        return;
    }
    while (head != NULL)
    {
        temp = head;
        head = head → link;
        free (temp);
    }
}
```

02/12/22

```
case 7 : PF ("Enter the number \n");
         SF ("%d ", &num);
         ~~delete~~ add_at_last (num);
         break;

case 8 : delete_at_last ( );
         break;

case 9 : ~~add~~ ~~after~~ PF ("Enter the i/p's);
                  SF ("%d-%d ", &num, & val );
                  ↑ element to be searched
                  ↑ element to be added.

         add_after_a_node (num, val);
         break;
```

08/10/22

⇒ void add_at_last (int num)

{

struct node *temp;

struct node *ptr = (struct node *)malloc (1*sizeof(struct node)); // Memory req to add node

ptr→data = num;

ptr → link = NULL;

if (head == NULL) // (i) list is empty

{

head = ptr;

return;

}

temp = head; // when list contains nodes

while (temp→link != NULL)

{

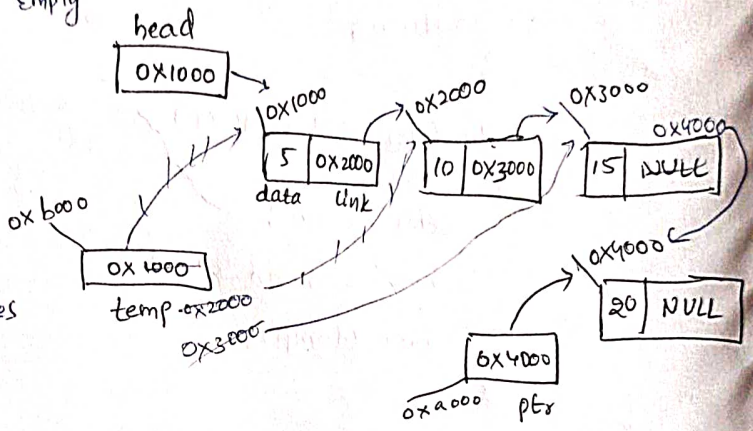
temp = temp→link;

}

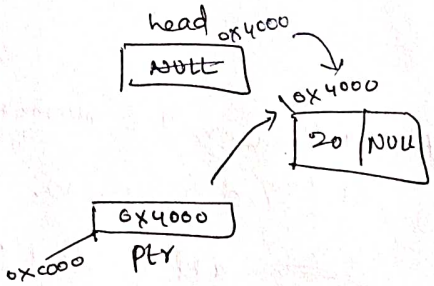
temp→link = ptr;

}

(ii) List contains nodes



(i) List empty



⇒ void del_at_last ( )

{

struct node *prev, *cur;

if (head == NULL)

{

PF("list is empty\n");

return;

}

if (head→link == NULL)

{

free(head);

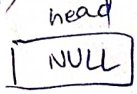
head = NULL;

return;

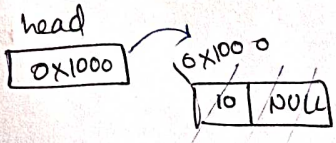
}

① // list is empty

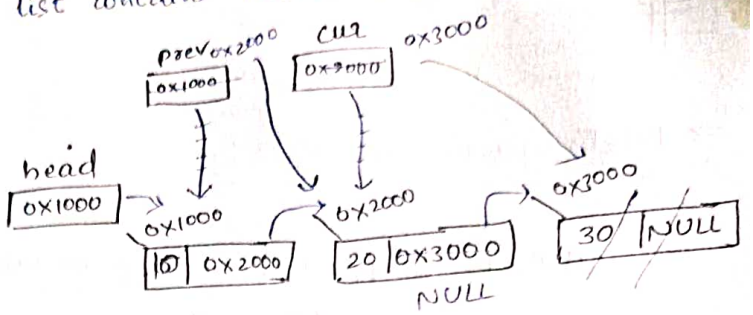
(i)



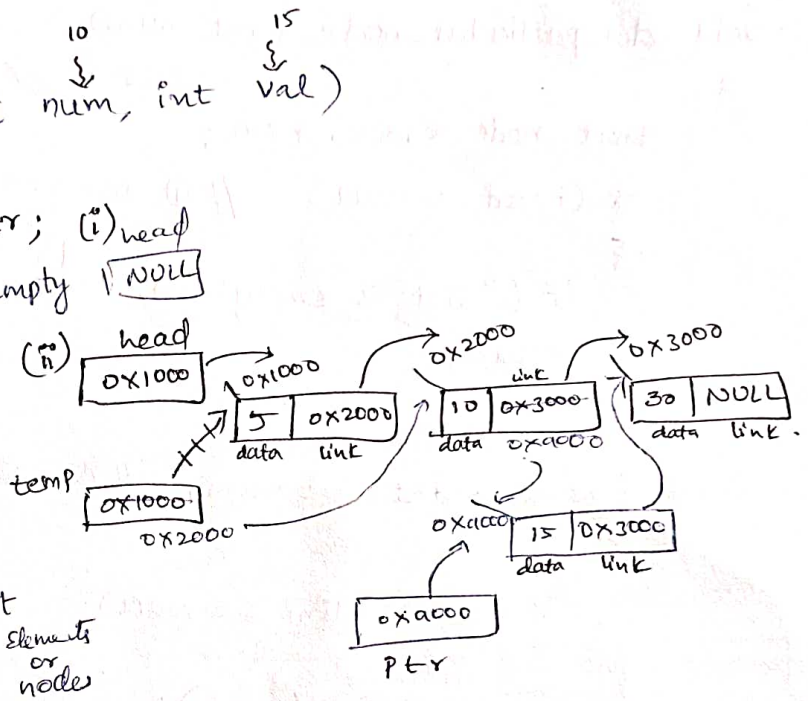
② // list contains single node & also verified 1st node.



```
        prev = head;          // when list contains nodes
        current = head → link;
        while (cur → link != NULL)
        {
            prev = cur;
            cur = cur → link;
        }
        free (cur);
        prev → link = NULL;
}
```




```
⇒ void add_after_a_node (int num, int val)
{
    struct node *temp, *ptr;
    if (head == NULL)  // list is empty
    {
        PF ("list is empty \n");
        return;
    }
    temp = head;
    while (temp != NULL)  // when list contains elements or nodes
    {
        if (temp → data == num)
        {
            ptr = (struct node *) malloc (1 * sizeof (struct node));
            ptr → data = val;
            ptr → link = temp → link;
            temp → link = ptr;
            return;
        }
        temp = temp → link;
    }
    Printf ("Element not found \n");
}
```

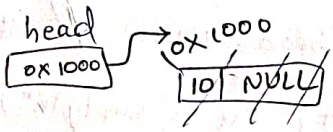


⇒ Delete particular node :-

```
case 10 : PF("enter the number \n");
          SF("%d", &num);
          del_particular_node(num);
          break;
```

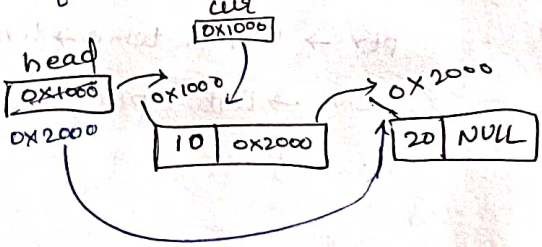
```
case 11 : PF("enter numbers\n");
          SF("%d-%d", &num, &val);
          add_before_node(num, val);
          break;
```

```
⇒ void del_particular_node (int num)
  {
    struct node *prev, *cur;
    if (head == NULL)    // ① list empty
    {
      PF("list is empty\n");
      return;
    }
```




```
    if (head -> data == num)   // ② single node & it is first node
    {
      if (head -> link == NULL)
      {
        free(head);
        head = NULL;
        return;
      }
      cur = head;
      head = head -> link;
      free(cur);
      return;
    }
```

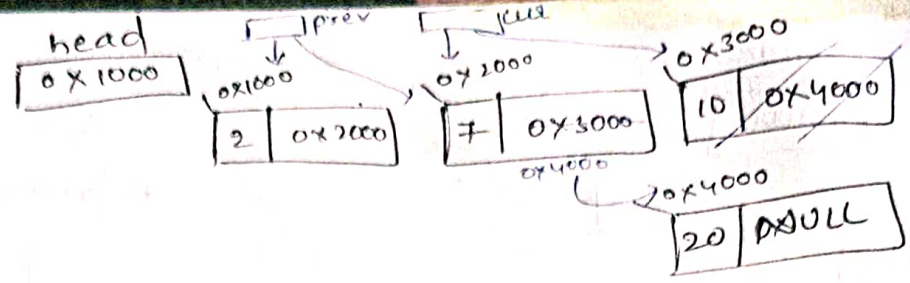
// ③ If it is 1st node but not single node



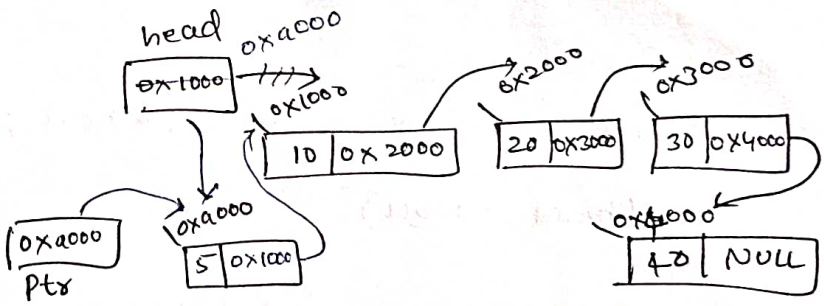

```
    prev = head;
    cur = head -> link;
    while (cur != NULL)
    {
      if (cur -> data == num)
      {
        prev -> link = cur -> link;
        free(cur);
```

// ④ multiple nodes with required element present in it.

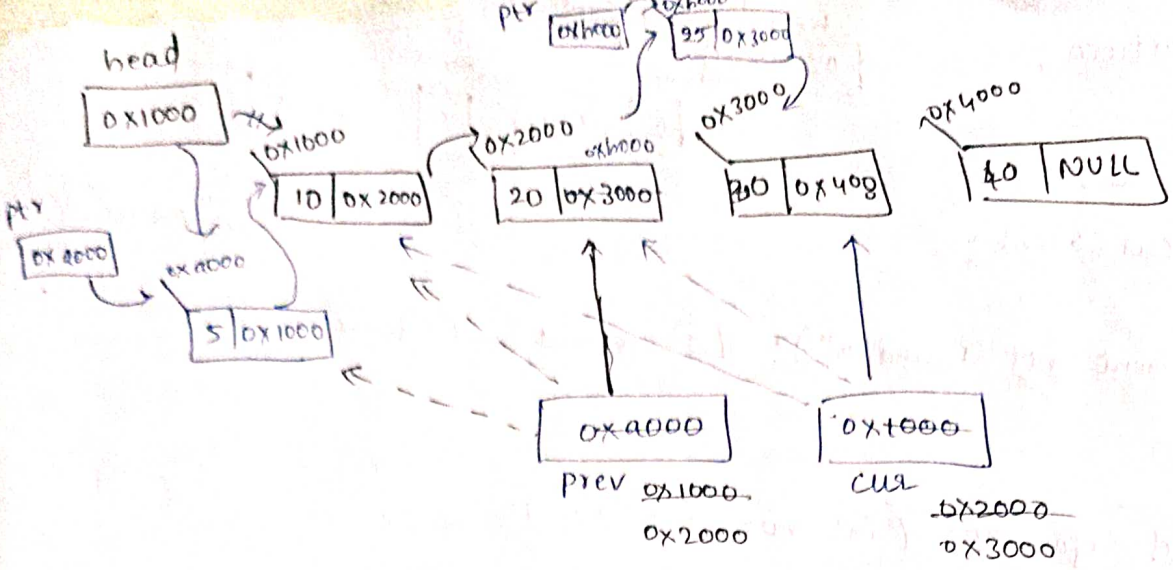
```
        return;
    }
    prev = cur;
    cur = cur -> link;
  }
  PF ("Element not found\n"); // ⑤ Element not found to delete.
}
```




```
⇒ void add_before_node (int num, int val)
{
    struct node *prev, *cur, *ptr;
    if (head == NULL)    // ① List is empty
    {
        PF ("list is empty\n");
        return;
    }
    if (head -> data == num)    // ② First node verification
    {
    ptr = (struct node *) malloc (1 * sizeof (struct node));
        ptr -> data = val;
        ptr -> link = head;
        head = ptr;
        return;
    }
    prev = head;
    cur = head -> link;    // ③ Element found at middle of the nodes.
    while (cur != NULL)
    {
        if (cur -> data == num)
        {
            ptr = (struct node *) malloc (1 * sizeof (struct node));
                ptr -> data = val;
                ptr -> link = cur;
                prev -> link = ptr;
                return;
        }
        prev = cur;
        cur = cur -> link;
```



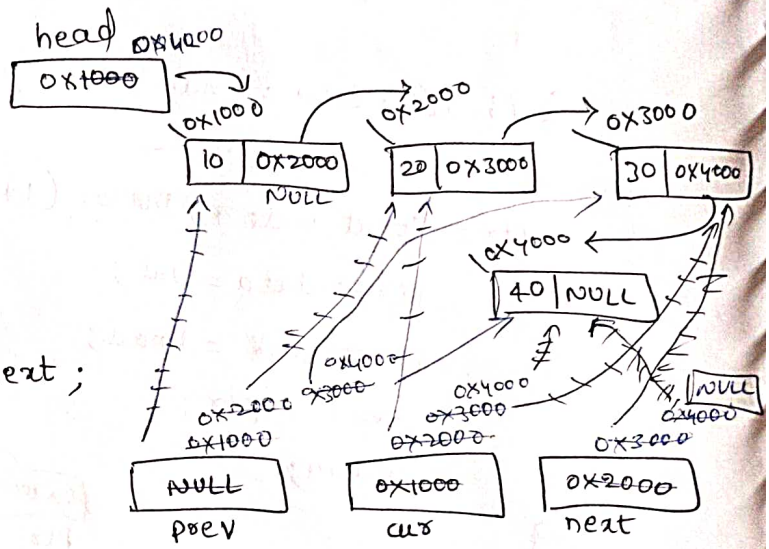

```
    if (ptr == NULL)
    {
        PF ("Failed to allocate memory in heap");
        exit (0);
    }
```



printf ("Element not found \n"); ④ Element not found

}

⇒ case 12: reverse_list();
break;



⇒ void reverse_list ( )
{

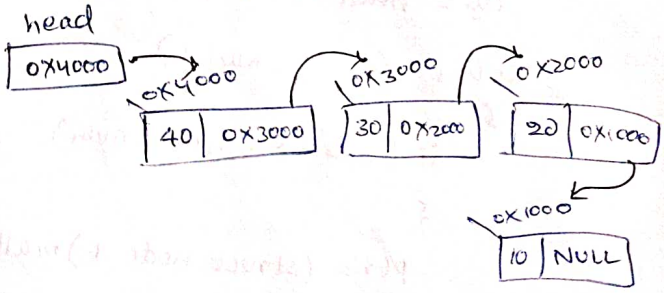
struct node *prev, *cur, *next;

if (head == NULL)
{
PF ("list is empty\n");
return;
}
prev = NULL;
cur = head;

while (cur != NULL)
{
next = cur → link;
cur → link = prev;
prev = cur;
cur = next;
}
head = prev;

}

O/P



03/12/22

case 13 : create list

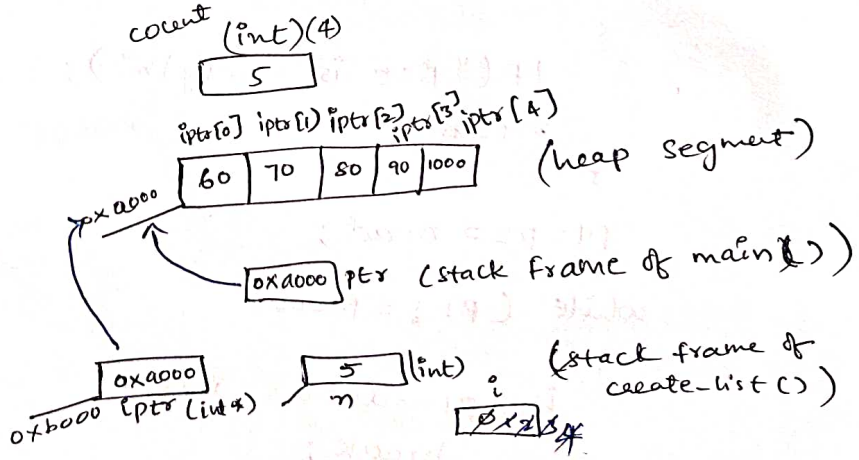
① List is empty

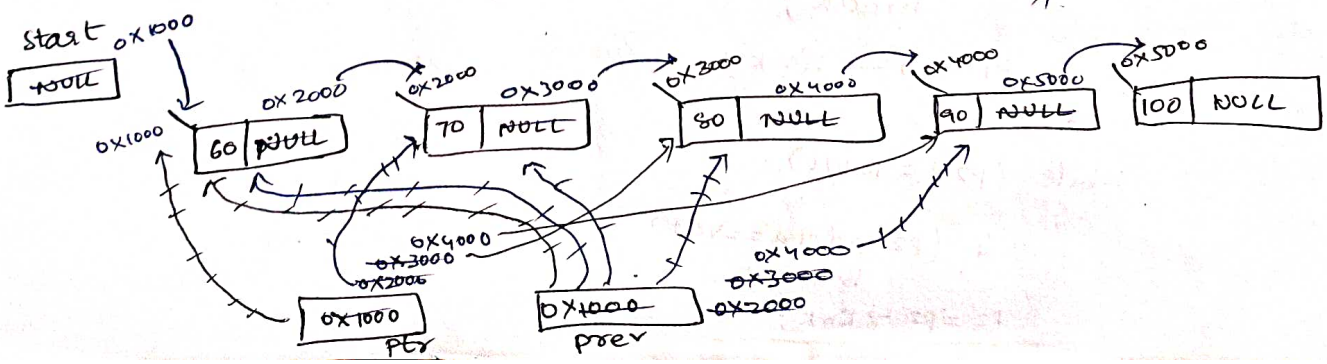
② contains nodes

```
int *ptr;
case 13: PF("Enter the count \n");
         SF("%d", &count);
         ptr = (int*)malloc(count * sizeof(int));
         if (ptr == NULL)
         {
           PF("failed to allocate \n");
           exit(1);
         }
         for (i=0; i<count; i++)
         {
           PF("Enter the i/p \n");
           SF("%d", &ptr[i]);
         }
         create_list(ptr, count);
         free(ptr);
         break;
```

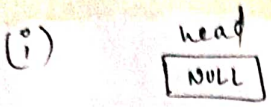
⇒

```
void create_list(int *iptr, int n)
{
    int i;
    struct node *ptr, *start = NULL, *prev, *temp;
    for (i=0; i<n; i++)
    {
     ptr = (struct node *)malloc(1 * sizeof(struct node));
     if (ptr == NULL)
     {
       PF("Failed to allocate \n");
       exit(2);
     }
     ptr -> data = iptr[i];
     ptr -> link = NULL;
     if (start == NULL)
         start = ptr;
     Else
         prev -> link = ptr;
     prev = ptr;
}
```



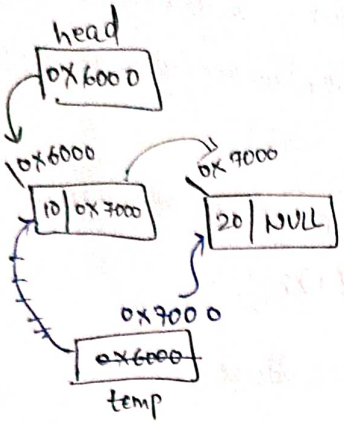


(i)




```
if (head == NULL)
{
    head = start;
    return;
}
temp = head;
while (temp -> link != NULL)
{
    temp = temp -> link;
}
temp -> link = start;
}
```

(ii) List contains nodes

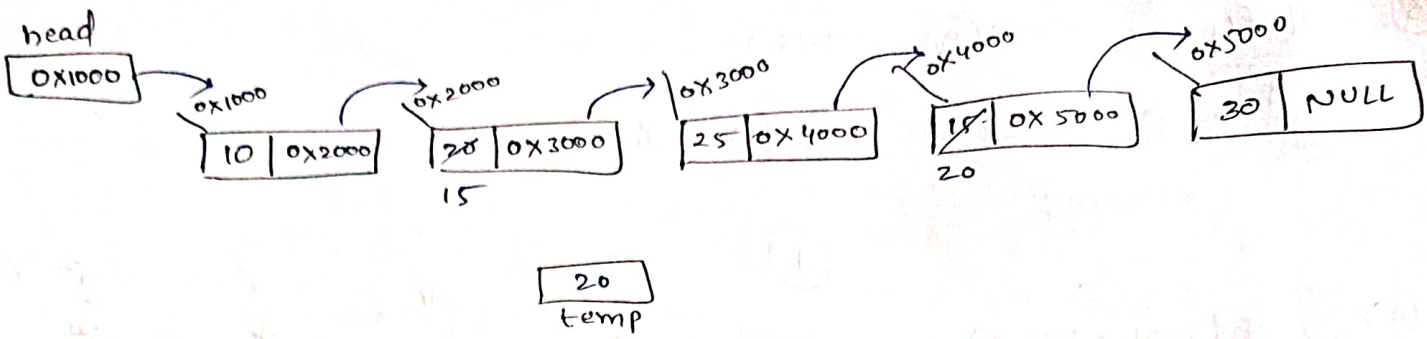


Case 14 : (swap)

```
PF("enter the elements to be swapped\n");
SF("%d-%d", &num, &val);
swap_data(num, val); // swap_link(num, val);
break;
```

⇒

```
void swap_data(int num, int val)
{
    int temp;
    struct node *P1, *P2;
    if (head == NULL)
    {
        PF("list is empty\n");
        return;
    }
    P1 = P2 = head;
    while (P1 != NULL)
    {
        if (P1 -> data == num)
            break;
        P1 = P1 -> link;
    }
    while (P2 != NULL)
    {
        if (P2 -> data == val)
            break;
        P2 = P2 -> link;
    }
```




```
if((p1==NULL) || (p2==NULL))
 {
   PF ("data not found\n");
   return;
 }
temp = p1->data;
p1->data = p2->data;
p2->data = temp;
}
```

```
⇒// void swap_link (int num, int val)
{
    int pos1=0, pos2=0;
    struct node *p, *q, *r, *s, *temp;
```

① ⇒ when p is pointing the current node, r will point its previous node.

① ⇒ when q is pointing the current node, s will point its previous node.

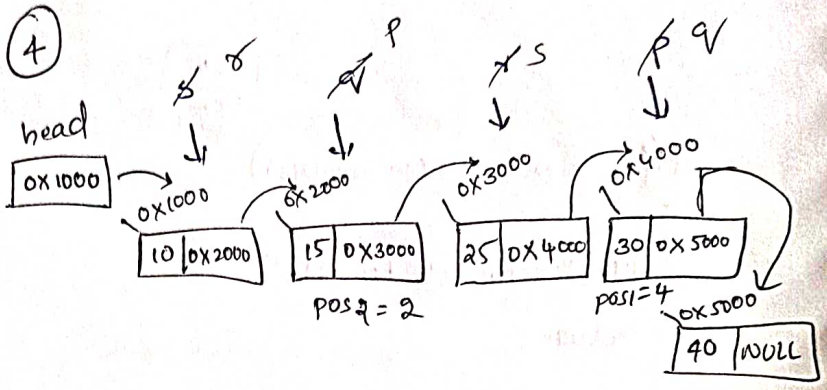
```
if (num==val)
  return;
```

②
```
if (head==NULL)
 {
   PF ("list is empty\n");
   return;
 }
p=q=head;
while (p!=NULL)
 {
   pos1++;
   if (p->data==num)
     break;
   r=p;
   p=p->link;
 }
```

```
while (q!=NULL)
 {
   pos2++;
   if (q->data==val)
     break;
   s=q;
   q=q->link;
 }
```

③ if ((p==NULL) || (q==NULL))
{
PF("Element not found\n");
return;
}

④ (4->2)
if (pos1 > pos2)
{
temp = p;
p = q;
q = temp;
temp = r;
r = s;
s = temp;
}

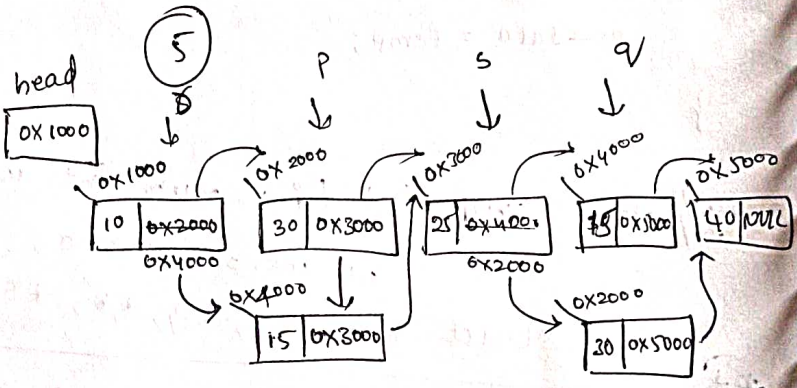


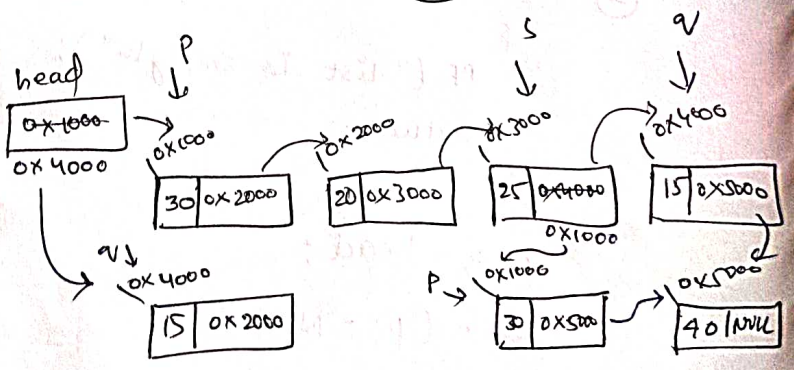
⑤ // Non-adjacent nodes

if (p->link != q)
{
temp = p->link;
p->link = q->link;
q->link = temp;

(i) if (p != head)
r->link = q;
Else
head = q;
s->link = p;
return;
}

// if it is 1st node (5i)

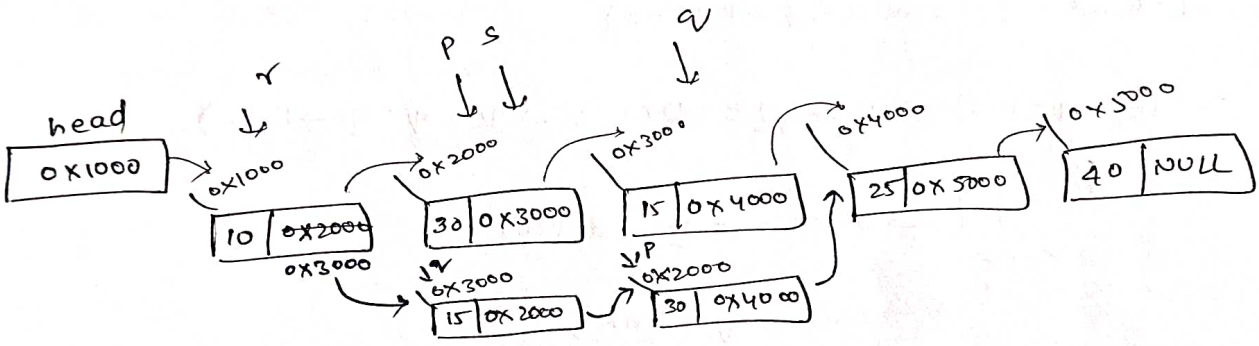




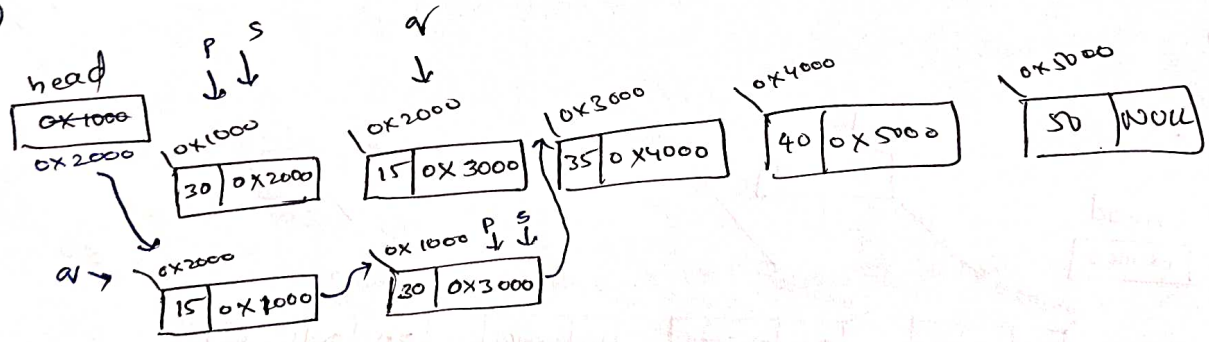
(6) // adjacent nodes

```
else
{
    p->link = q->link;
    q->link = p;
(i) if (p != head)
        r->link = q;
    else
        head = q;
    return;
}
}
```

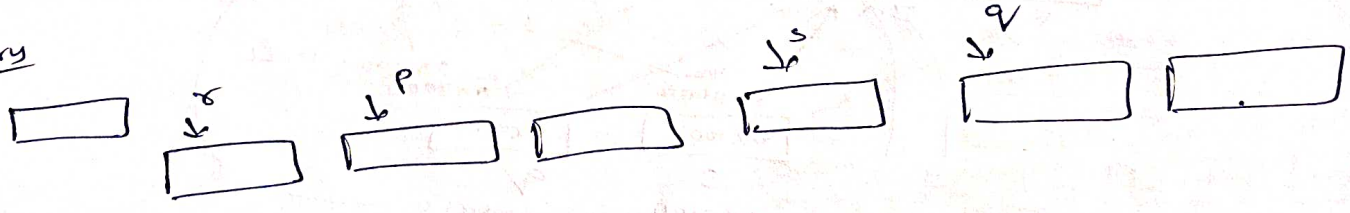
(6)



(6i)



Try

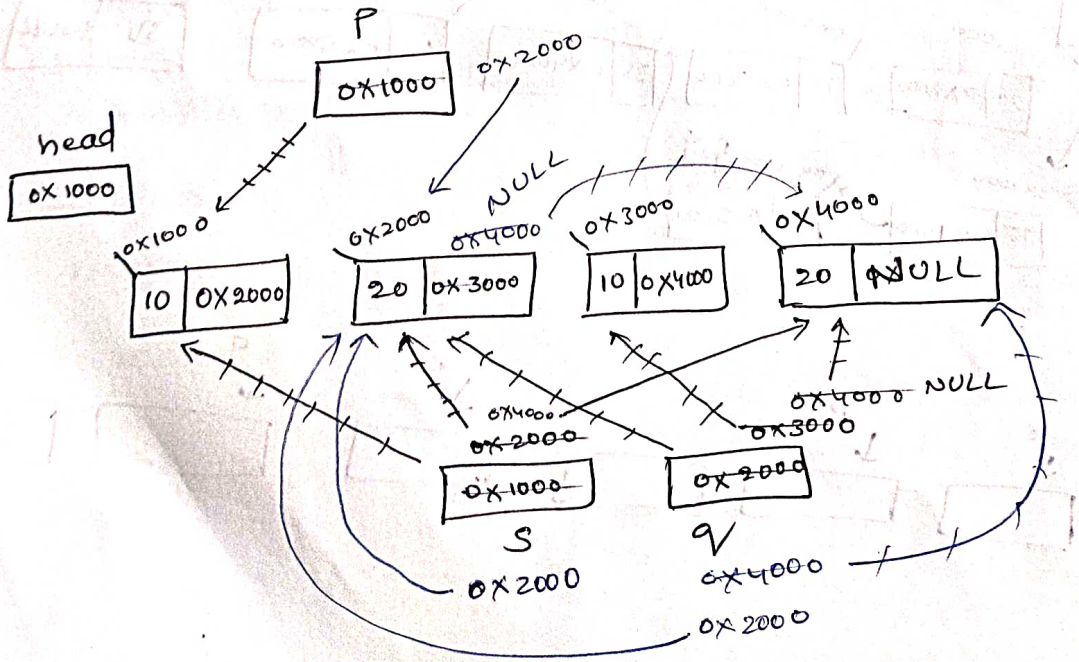


05/12/22

case 15 : delete_duplicate_nodes ( );
break;

⇒ void delete_duplicate_nodes ( )

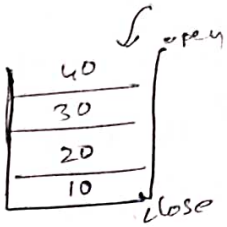
```
{
  struct node *p, *q, *s ;
 if (head==NULL)
  {
    PF("list is empty\n");
    return;
  }
for (p=head , p !=NULL ; p=p→link)
{
  for (s=p , q=p→link ; q!=NULL ; s=q , q=q→link)
      {
        if (p→data == q→data)
        {
          s→link = q→link ;
          free(q)
           q=s ;
        }
      }
}
}
```



Stack

Implementation of stack using single linked list

(Last in first out)



1. push → add at beginning
2. pop → del at ~~beginning~~ last
3. display
4. count
5. search
6. reverse
7. swap
8. delete duplicate
9. delete stack / delete entire list
10. delete particular node

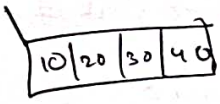
(different) 11. create list (stack)

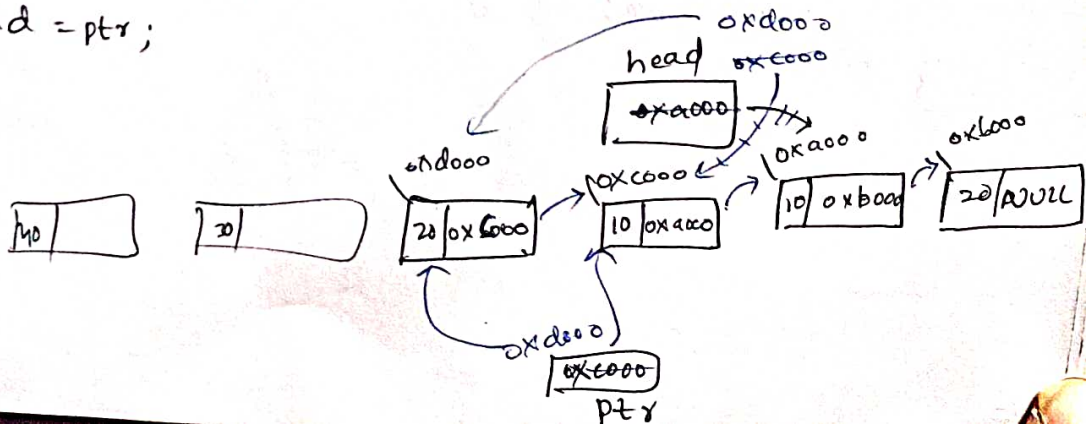
stack

```
case 6 :
        Create_stack (ptr, N);
        break;
```

⇒

```
void create_stack (int *iptr, int n)
{
    int i;
    struct node *ptr;
    for (i=0; i<n; i++)
    {
     ptr = (struct node*) malloc (1*sizeof (struct node));
         ptr->data = iptr[i];
         ptr->link = head;
         head = ptr;
    }
}
```





Queue

⇒ In Queue we have 2 pointers (front pointer & rear pointer)

⇒ By using the front pointer, we delete the nodes.

⇒ By using rear pointer, we will add the nodes.

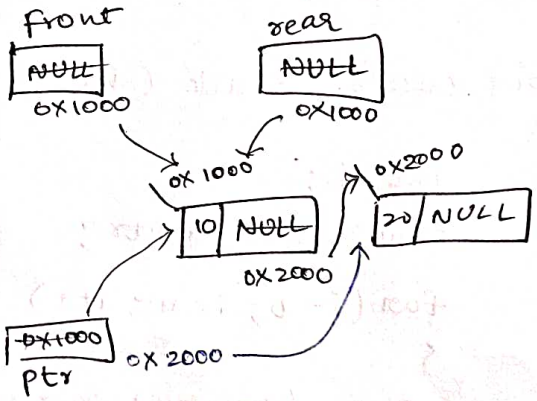
⇒ Both front & rear are NULL, when Queue is empty.

⇒ when Queue contains single node, both front & rear pointer's will Points the ~~Same~~ same node.

⇒ Front pointer will always points the 1st node, rear pointer will always points the last node.

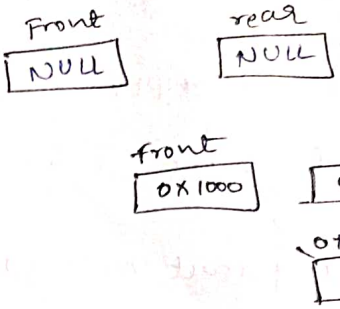
① ⇒ void insert (int val)

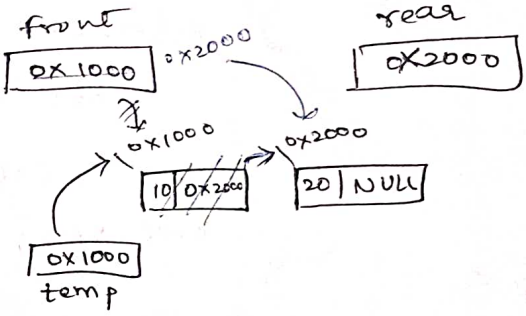
```
{
struct node *ptr = (struct node *) malloc (1*sizeof (struct node));
  if (ptr == NULL)
   {
     PF (" failed to allocate memory \n");
     exit (1);
   }
  ptr → data = val;
  ptr → link = NULL;
if (rear == NULL)
 {
   front = rear = ptr;
   return;
 }
rear → link = ptr;
  rear = ptr;
}
```



② void delete ( )

```
{
    struct node *temp;
    if (rear == NULL)   // queue is empty
    {
        PF("Queue is empty\n");
        return;
    }
    Else if (front -> link == NULL) // when Queue contains single Element
    {
        free (front);
        front = rear = NULL;
        return;
    }
    Else
    {                               // when queue contains more than single node.
        temp = front;
        front = front -> link;
        free (temp);
    }
}
```





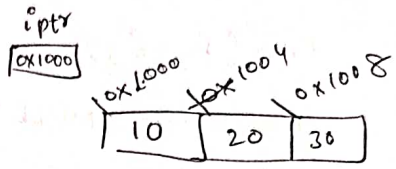
③ count ④ search ⑤ display

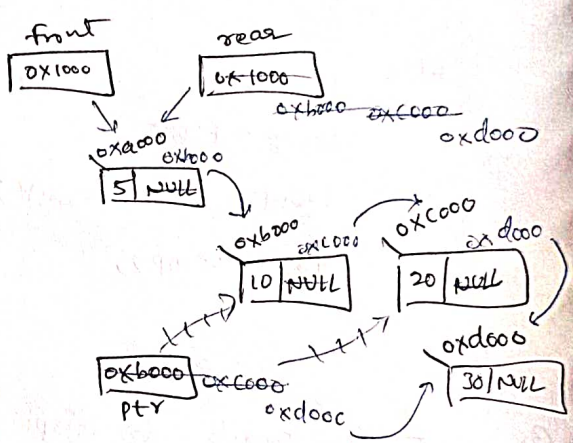
⑥ void delete_queue ( )

```
{   struct node *temp;
    if (rear == NULL)
    {
        PF("Queue is empty\n");
        return;
    }
    while (front != NULL)
    {
        temp = front;
        front = Front -> link;
        free (temp);
    }
    rear = NULL;
}
```

(7)

```
void create_queue (int *iptr, int n)
{
    int i;
    struct node *ptr
    for (i=0; i<n; i++)
    {
        ptr = (struct node *) malloc (1*sizeof(struct node));
        ptr->data = iptr[i];
        ptr->link = NULL;
        if (rear == NULL)
            rear = front = ptr;
        Else
            rear->link = ptr;
            rear = ptr;
    }
}
```





(8)

```
void reverse_queue ( )
{
    struct node * prev, * cur, * next;
    if (rear == NULL)
    {
        PF ("Queue is Empty \n");
        return;
    }
    prev = NULL;
    cur = Front;
    while (cur != NULL)
    {
        next = cur->link;
        cur->link = prev;
        prev = cur;
        cur = next;
    }
    rear = front;
    front = prev;
}
```

(or) we can take another temp pointer & we can swap the front & rear pointers base addresses.

(9) void delete_duplicate_queue ( )

```
{
    Struct node *p, *q, *s;
    if (rear == NULL)
    {
        PF ("Queue is empty \n");
        return;
    }
    for (p = front , p != NULL ; p = p->link)
    {
        for (s=p, q= p->link ; q != NULL ; s = q , q = q->link)
        {
            if (p->data == q->data)
            {
                s->link = q->link;
                if(q==rear)
                {
                rear = s;
                }
                free(q);
                q=s;
    }   }   }
}
```

duplicate element at last when it has ~~only~~ one

(10) void delete-particular_node

```
{
    Struct node * temp, * prev, * cur;
    if ( front == NULL)
    {
        PF ("queue is empty\n");
        return;
    }
    if ( front ->data == num)
    {
        if (front -> link == NULL)
        {
            free (front);
            front = rear = NULL;
            return;
        }
        temp = front;
        front = front -> link;
        free (temp);
    }   .   return;
```

```
        cur = front -> link;
        prev = front;
        while (cur != NULL)
        {
            if (cur -> data == num)
            {
                prev -> link = cur -> link;
                if (cur == rear)
                {
                    rear = prev;
                }
                free (cur);
                return;
            }
            prev = cur;
            cur = cur -> link;
        }
        PF ("element not found\n");
    }
```

(40) (15)

(11) void swap_link (int num, int val)

(40) (15)

```
    {
        int pos1 = 0, pos2 = 0;
        struct node *p, *q, *temp, *q, *s;
```

(1) if (num == val) // when both nos are same to swap

```
        {
            return;
        }
```

(2) if (front == NULL) // when queue is empty

```
        {
            PF ("queue is empty\n");
            return;
        }
    q = p = front;
    while (p != NULL)
        {
            pos1 ++;
            if (p -> data == num)
                break;
            q = p;
            p = p -> link;
        }
```

```
while (q != NULL)
 {
    pos2++;
    if (q->data == val)
       break;
    s=q;
    q=q->link;
 }
```

③ `if((p==NULL)||(q==NULL))` // when required swap element is not found

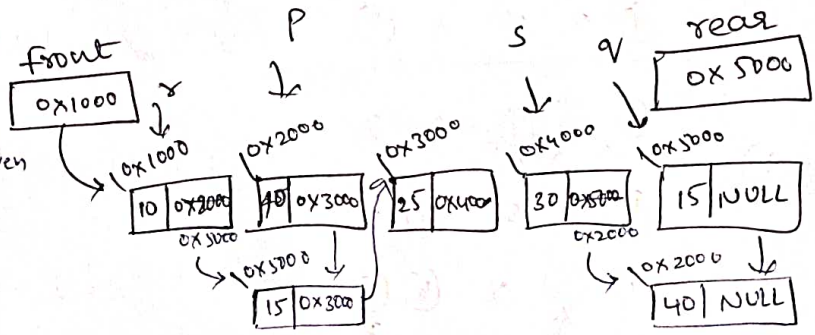
```
 {
   Pf("Element not found\n");
   return;
 }
```

④ `if (pos1 > pos2)`

```
 {
   temp=p;
   p=q;
   q=temp;

   temp=r;
   r=s;
   s=temp;
 }
```

// non-adjacent nodes

When positions are not in given order



⑤ `if (p->link != q)`

```
 {
   temp=p->link;
   p->link=q->link;
   q->link=temp;
```

(i) `if (p != front)` // when it is 1st node to swap.

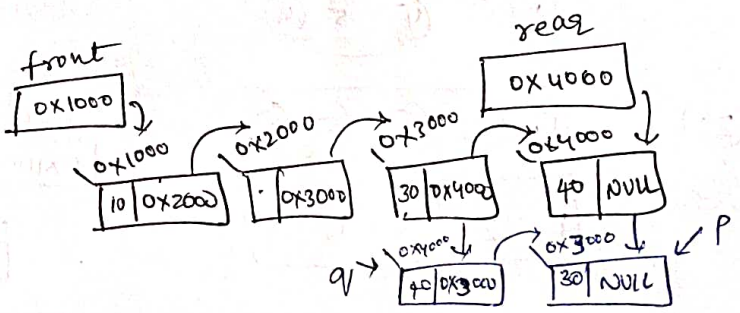
```
   r->link=q;
 Else
   front=q;
   s->link=p;
```

(ii) `if (q == rear)` // when q consists same as rear

```
   rear=p;
 }
```

Else

⑥ // adjacent nodes




```
Else
 {
   p->link=q->link;
   q->link=p;
```

(6)(i) `if (p != front)` // when it is first node & adjacent node to swap.

```
   r->link=q;
 Else
   front=q;
```

(ii) `if (q==rear)` // q & rear pointers consists of same address

```
   rear=p;
 }
}
```

6/12/22

* Sorting *

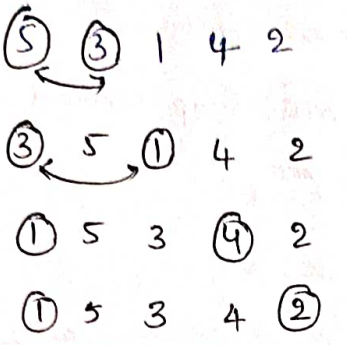
By using arrays

① selection sort (we will set 1st position)

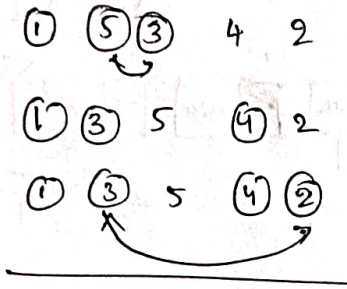
n elements (5)

(n-1) iterations

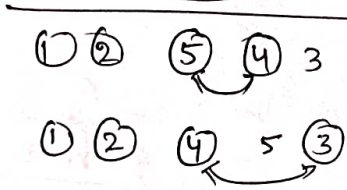
(5-1)



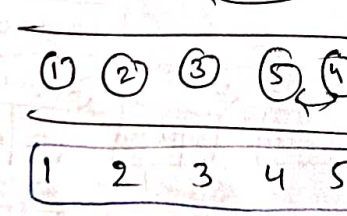
(4-1)



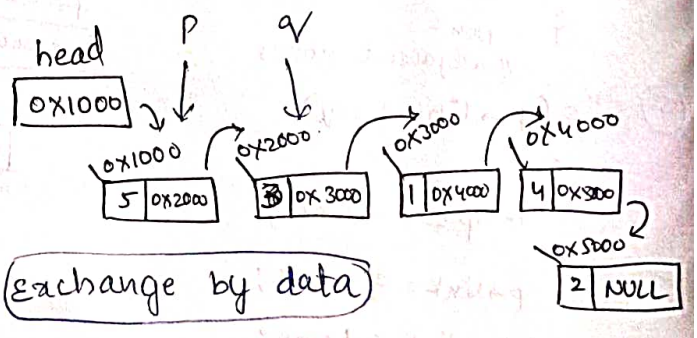
(3-1)



(2-1)



```
main()
{
  int arr[5] = {5,3,1,4,2};
  selection_sort (arr, 5);
  display (arr, 5);
}

Void selection_sort (int *ptr, int n)
{
  int i, j, temp ;
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      if (ptr[i] > ptr[j])
      {
        temp = ptr[i];
        Ptr[i] = ptr [j];
        Ptr [j] = temp ;
} } } }
```



⇒ exchange by data

```
⇒ void selection_sort_data ( )
  {
    int temp ;
    Struct node *P, *q ;
    if (head == NULL)
    {
      PF ("list is empty \n");
      return;
    }
    for (P=head ; P->link != NULL ; P=P->link)
    {
      for (q = P->link ; q != NULL ; q=q->link)
```

```
        if (p->data > q->data)
          {
              temp = p->data;
              p->data = q->data;
              q->data = temp;
          }
    }   }
}
```

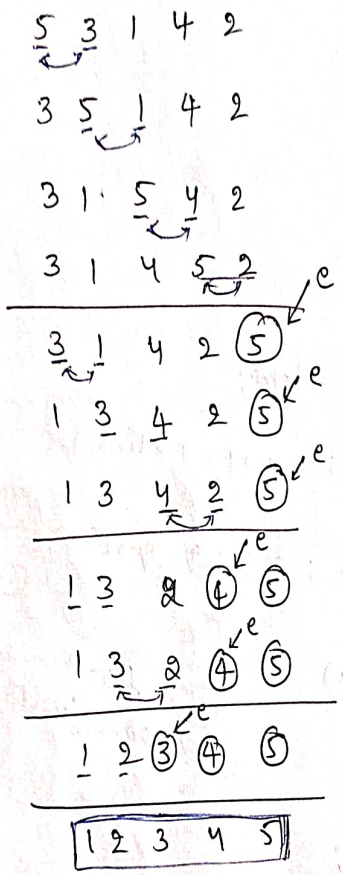
Exchange by link

```
=> void selection_sort_link (    )
   {
     struct node *p, *q, *r, *s, *temp;
     for(r=p=head; p->link != NULL ; r=p, p = p->link)
      {
        for(s=q= p->link; q != NULL ; s=q, q=q->link)
          {
            if (p->data > q->data)
              {
                temp =p->link;
                p->link = q->link;
                q->link = temp;
              if (p != head)
                r->link = q;
              else
                head=q;
              s->link = p;
            temp = p;
            p = q;
            q = temp;
          }
      }
   }
}
```

follows //non-adjacent swapping

✳ Bubble sort :-

⇒ In bubble sort, always we will set the last position (i.e., highest element).



For array elements

```
Void Bubble sort (int *ptr, int n)
{
  int i, j, temp, e;
 for (e=n; e>0; e--)
 {
  for (i=0; i<e; i++)
    {
      j=i+1;
    if (ptr[i]>ptr[j])
      {
        temp = ptr[i];
        ptr[i] = ptr[j];
        ptr[j] = temp;
      }
    }
 }
}
```

Exchange by data

⇒
```
Void bubble_sort_data ( )
{
        int temp;
  struct node *p, *q, * e, *temp;
   for (e=NULL; head→link != e ; e=q)
   {
     for (p=head ; p→link != e ; p=p→link)
     {
        q= p→link;
        if (p→data > q→data)
        {
          temp = p→data;
          p→data = q→data;
          q→data = temp;
        }
     }
   }
}
```

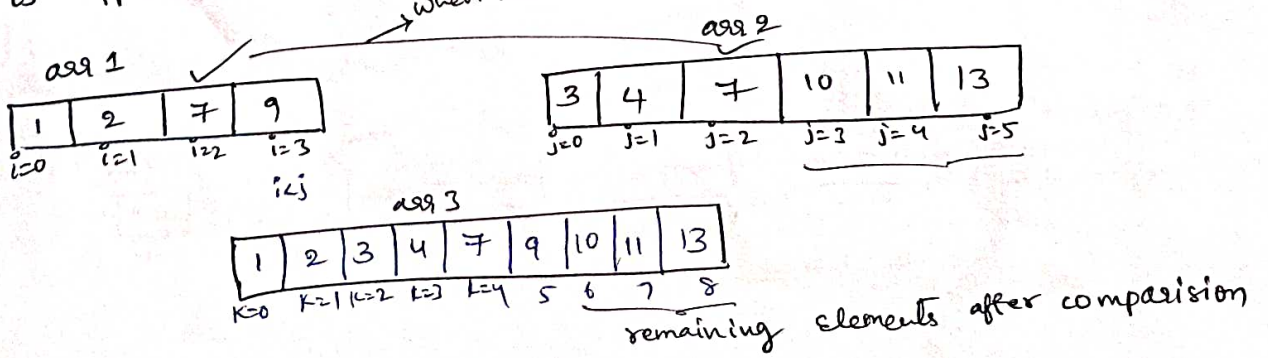
Exchange by link

```
⇒ void bubble_sort_link ( )
{
   struct node * p, *q, *r, *e, * temp ;
   for (e= NULL ; head→ link ! =e ; e=q)
   {
     for (p= head ; p→ link ! = e ; r=p , p= p→ link)
     {
        q = p→link ;
        if ( p→ data > q→ data)
        {
           p→ link = q→ link ;
           q→ link = p ;
           if (p ! =head )
            r→ link=q ;
           Else
                head=q ;
           temp = p ;
            p = q ;
            q = temp ;
        }
     }
   }
}
```

// follows adjacent swap by link.

(*) Merge sort :-

⇒ It is applicable on two sorted arrays



⇒
```
for (i=0, j=0, K=0 ; (i<N1) && (j<N2) ; K++)
  {
    if (arr1[i] == arr2[j])
      {
        arr3[K] = arr1[i];
        i++ , j++ ;
      }
  Else if (arr1[i] < arr2[j])
      {
        arr3[K] = arr1[i];
        i++;
      }
  Else
      { arr3[K] = arr2[j];
        j++;
      }
  }
for (i; i<N1; i++, K++)
      arr3[K] = arr1[i];
for (j ; j<N2 ; j++ , K++)
      arr3[K] = arr2[j] ;
}
```

09/12/22

* Double Linked list *

⇒ In double linked list, each node will have two link parts one link will point the previous node & another link will point the next node.

⇒ In single linked list, we can traverse only in one particular direction i.e., forward direction but we can't traverse in reverse.

But, In double linked list, we can traverse in forward as well as reverse direction.

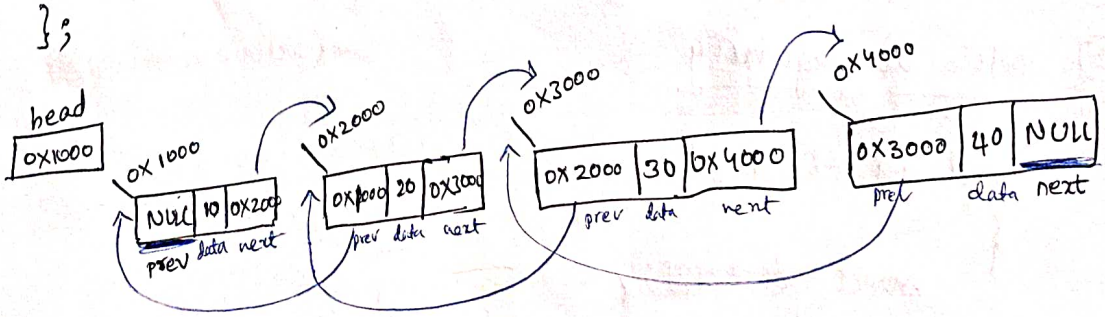
(Q) what are the differences blw single linked list & arrays.

(6) " " " " " single linked list & double linked list.

⇒ In double linked list, first node previous part contains NULL. Last Node, next part contains NULL.

* Structure definition for double linked list :-

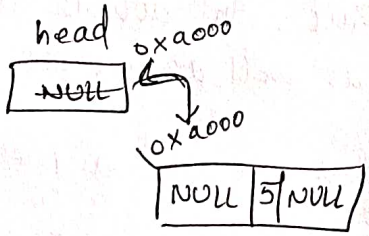
```
struct node
{
    struct node * prev;
     int data;
    struct node * next;
};
```



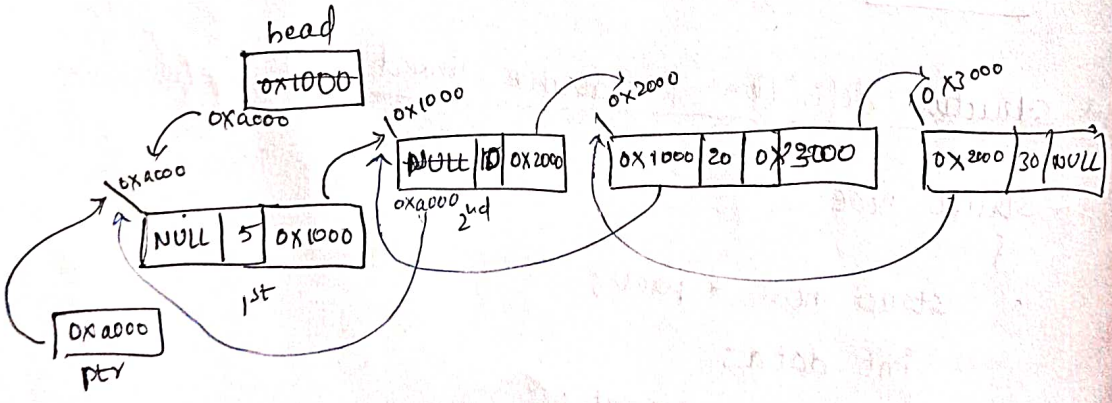
① void add_at_beg (int num)

```
{
    struct node *ptr ;

ptr = (struct node *)malloc ( 1*sizeof (struct node));
    if (ptr == NULL)
    {
      PF("Failed to allocate \n");
      exit(1);
    }

Ptr → data = num;
Ptr → prev = NULL;
Ptr → next = head;
if ( head != NULL)
    Ptr → next → prev = ptr;
    head = ptr;
}
```

(i) when list is empty



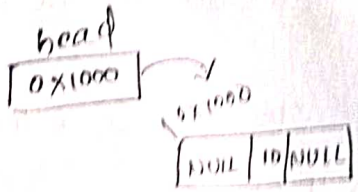
(ii) when list contains nodes



② Delete at beginning

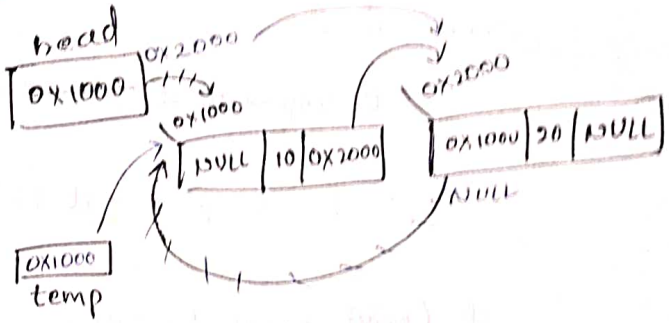
```
void del_at_beg( )
{
    struct node *temp;
  if (head == NULL)
  {
    PF ("list is empty \n");
    return;
  }
```

// when list contains single node.




```
if (head -> next == NULL)
 {
   free (head);
    head = NULL ;
   return;
 }
```

// when list contains multiple nodes



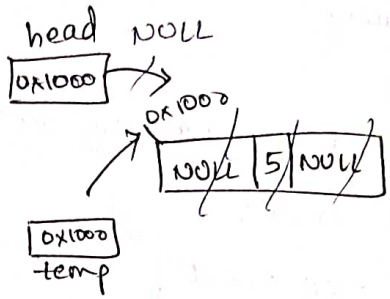

```
  temp = head ;
  head = head -> next ;
  head -> prev = NULL ;
   free (temp);
}
```

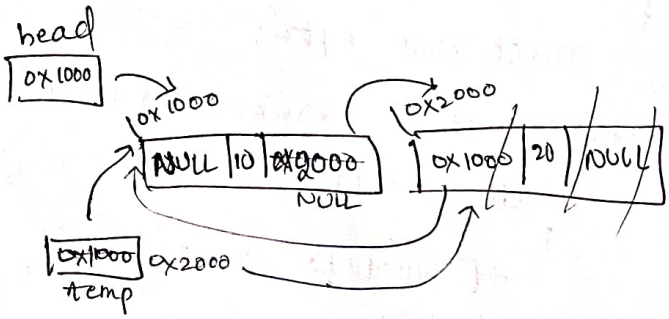
③ void add_at_last (int num)

```
{
    struct node *ptr ;
ptr = (struct node *)malloc (1*sizeof (struct node));
     if (ptr == NULL)
     {
       pf(" failed to allocate memory in heap\n");
       exit (1);
     }
    ptr -> data = num;
    ptr -> next = NULL;
 if (head == NULL)
  {
      ptr -> prev = NULL;
      head = ptr;
      return;
  }
 temp = head;
 while (temp -> next != NULL)
   {
      temp = temp -> next ;
   }
  temp -> next = ptr;
   ptr -> prev = temp;
}
```

④ void del_at_last ( )

```
{
    struct node *temp;
    if (head == NULL)      // when list is empty
    {
      PF ("list empty \n");
      return;
    }
    temp = head;
    while (temp -> next != NULL)
      {
        temp = temp -> next ;
      }
    if (head -> next != NULL)      // when list contains single node
        temp -> prev -> next = NULL;
    Else
        head = NULL;               // when list contains multiple nodes
    free (temp);
}
```



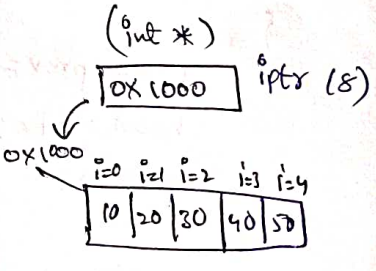


⑤ display ⑥ count ⑦ search // Assignment
⑧ delete entire list. ⑪ add after node
⑫ add before node

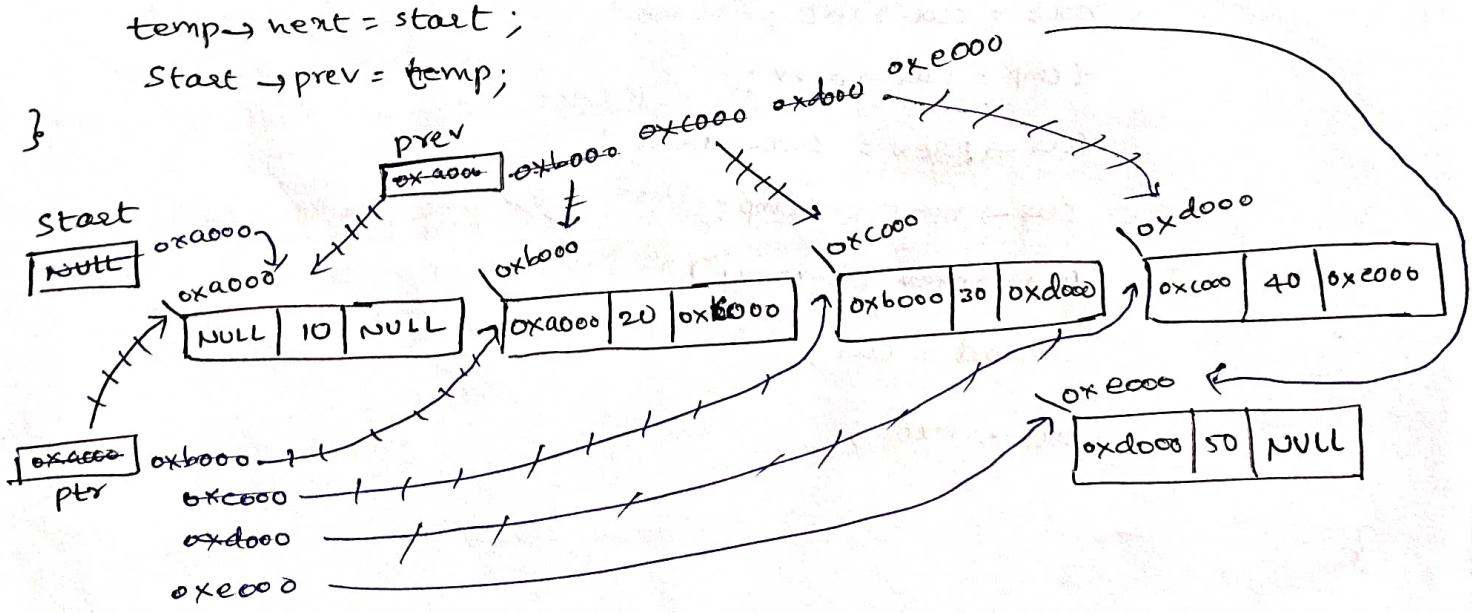
⑨ void create_list (int *iptr, int n)

{0X1000 {5

```
{
    int i;
    struct node * ptr, * start = NULL, *prev,
                                  * temp;
    for (i=0; i<n; i++)
    {
      ptr=(struct node *)malloc (1*sizeof (struct node));
          if (ptr == NULL)
            { PF("failed to allocate memory in heap\n");
            } return;
```

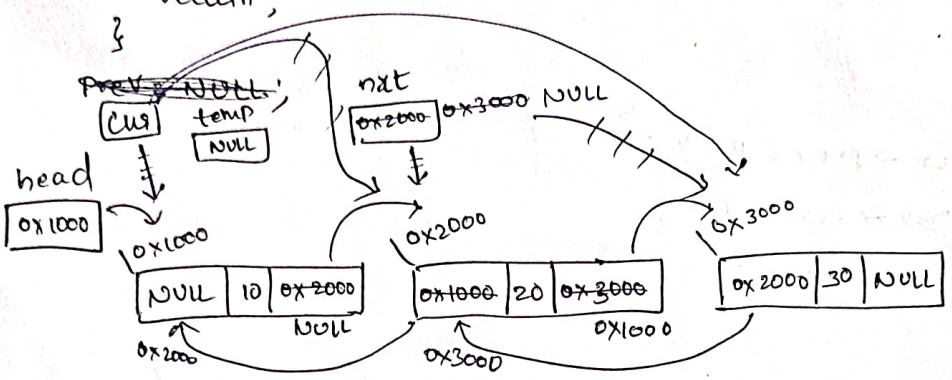


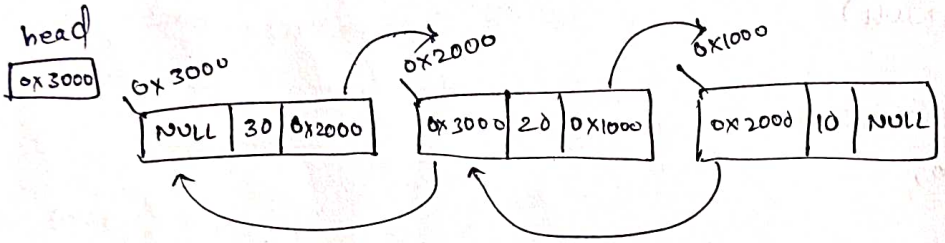
```
        ptr -> data = iptr[i];
        ptr -> next = NULL;
    if (start == NULL)
        {
            ptr -> prev = NULL;
            start = ptr;
            prev = ptr;
        }
    Else
        {
            ptr -> prev = prev;
            prev -> next = ptr;
            prev = ptr;
        }
}

if (head == NULL)
 {
    head = start;
    return;
 }
temp = head;
while (temp -> next != NULL)
    {
        temp = temp -> next;
    }
    temp -> next = start;
    start -> prev = temp;
}
```



(10) void reverse_list ( )

```
{
  struct node *prev, *cur, *nxt, *temp ;
  if (head == NULL)
   {
     Printf (" list is empty \n" );
     return ;
   }
```

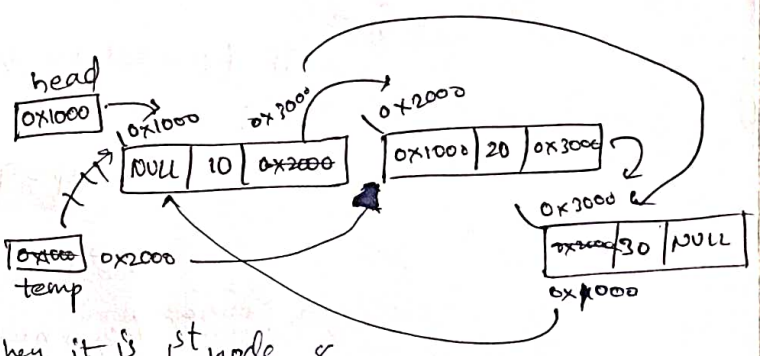





```
  cur = head ;
  while (cur != NULL)
  {
     nxt = cur → next ;
     temp = cur → prev ;
     cur → prev = cur → next ;
     cur → next = temp ;
   if (cur → prev == NULL)
      head = cur ;
     cur = nxt ;
  }
}
```

(13) void Delete particular node (int num)

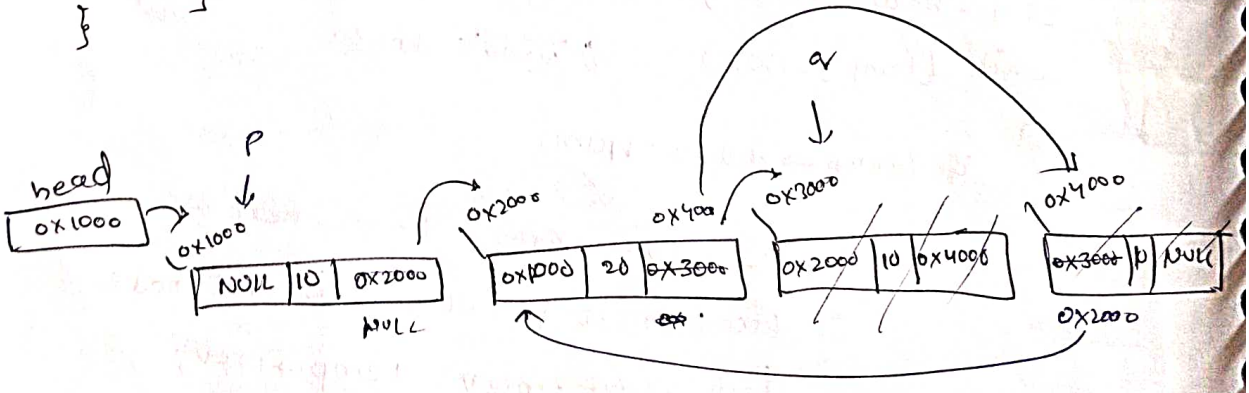
```
{
    struct node *temp;
if (head==NULL)                    // when list is empty
  {
    PF ("List empty \n");
    return;
  }
if (head->data==num)               // when it is 1st node
  {
    if (head->next==NULL)          // when it is single node.
      {
          free (head);
          head=NULL;
          return;
      }
    temp = head;                   // single node
    head = head->next;
    head->prev = NULL;
     free (temp);
    return;
  }
  temp = head->next;
  while (temp!=NULL)               // middle node.
   {
     if (temp->data==num)
       {
         temp->prev->next = temp->next;
         if (temp->next!=NULL)     // last node
           temp->next->prev= temp->prev;
          free (temp);
        return;
       }
    temp=temp->next;
  }
  PF ("element not found \n");     // not found no to delete.
}
```



// when it is 1st node & contains multiple nodes.

(14) void del_duplicate_nodes ( )

```
{
    struct node *p, *q, *temp;
  if (head ==NULL)
   {
      PF ("list is empty \n");
      return ;
   }
  for (p = head ; p != NULL ; p= p->next)
   {
      for ( q = p-> next ; q ~~->next~~ != NULL ; q=q-> next )
        {
            if (p ->data == q -> data)
              {
                  temp =q;
                  q -> prev -> next = q -> next ;
                if (q -> next != NULL)
                    q -> next -> prev =q -> prev ;
                    q = q -> prev ;
                    free (temp) ;
              }
        }
   }
}
```



(15) sorting

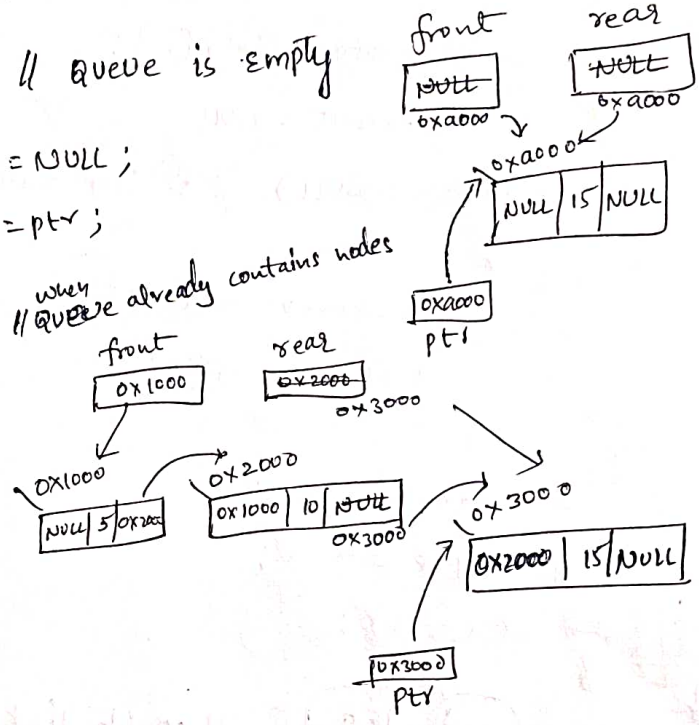
(16) Swap

13/12/22 * Implementation of Queue by double linked list

①

⇒ void insert (int num)

```
{
  struct node *ptr = (struct node *) malloc (1 * sizeof (struct node));

    ptr → data = num;
    ptr → next = NULL;

  if (rear == NULL)      // Queue is empty
  {
     ptr → prev = NULL;
     front = rear = ptr;
     return;
  }
  ptr → prev = rear;
  rear → next = ptr;
   rear = ptr;
}
```

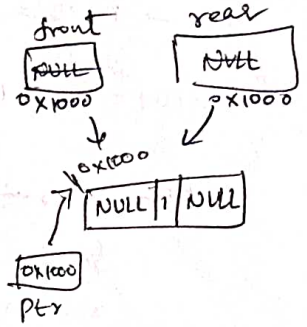


② void delete ( )

```
{
  struct node *temp;
  if (front == NULL)
  {
    pf ("Queue is empty\n");
    return;
  }
  if (front == rear)
  {
    free (front);
    front = rear = NULL;
    return;
  }
  temp = front;
  front = front → next;
    free (temp);
    front → prev = NULL;
}
```

③ Void Create_list (int iptr, int n)

```
{
    int i;
    Struct node *ptr;
    for (i=0; i<n; i++)
    {
    ptr = (struct node *)malloc (1* sizeof (struct node));
        ptr → data = iptr[i];
        ptr → next = NULL
    if (rear == NULL)  // 1st is 1st node
        {
            ptr → prev = NULL;
            front = rear = ptr;
        }
    rear → next = ptr;
    ptr → prev = rear;
      rear = ptr;
    }
}
```

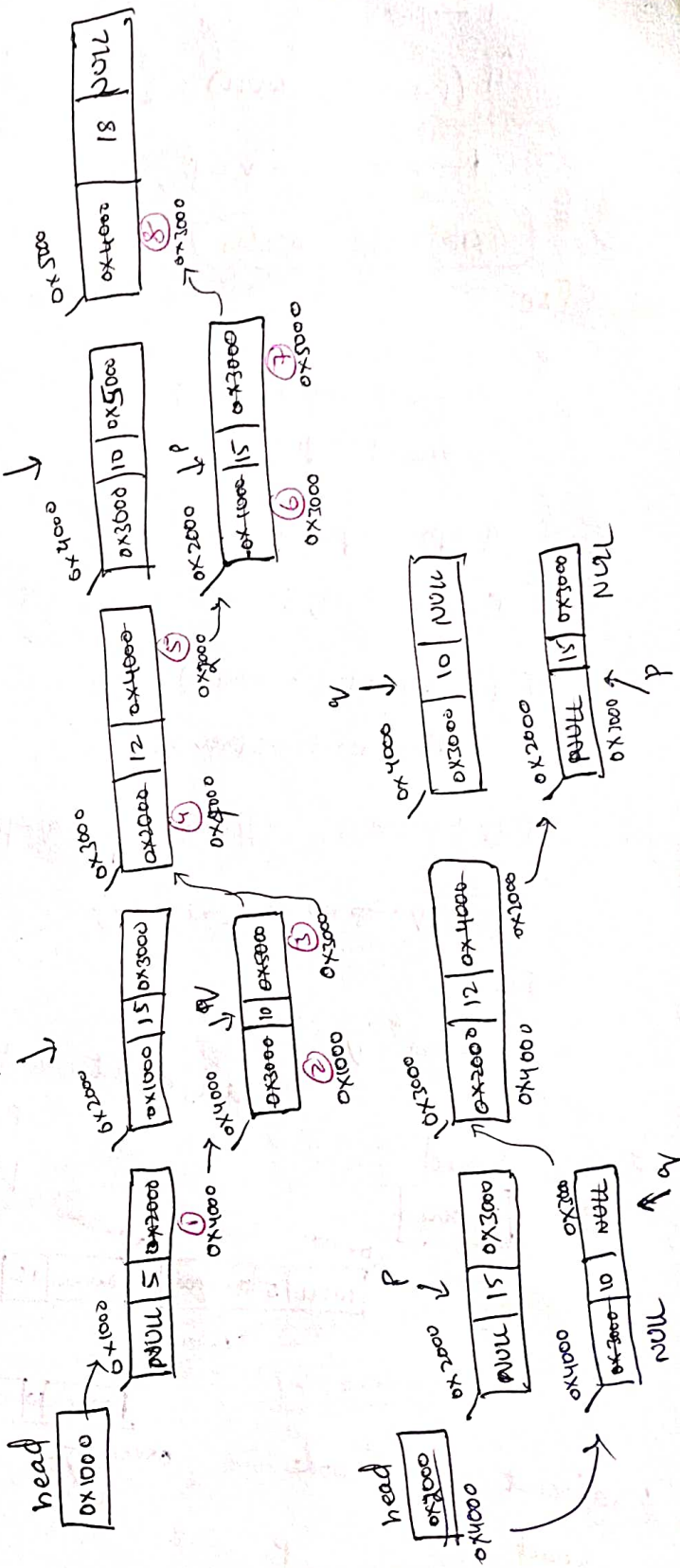


// double linked list

④ void swap_link (int num1, int num2)

```
{
    Struct node *P, *q, *temp;
      int pos1, pos2;
    if (head == NULL)
    {
        printf ("list is empty\n");
        return;
    }
    if (num1 == num2)
      return;
    pos1 = 0, pos2 = 0;
      P = q = head;
```

```
while (p != NULL)
{
    pos1++;
    if (num1 == p->data)
        break;
    p = p->next;
}
while (q != NULL)
{
    pos2++;
    if (num2 == q->data)
        break;
    q = q->next;
}
if ((p == NULL) || (q == NULL))
{
    PF ("data not found\n");
    return;
}
if (pos1 > pos2)
{
    temp = p;
    p = q;
    q = temp;
}
if (p->next != q)
{
    temp = p->prev;
    p->prev = q->prev;
    q->prev = temp;

    temp = p->next;
    p->next = q->next;
    q->next = temp;

    if (p != head) || (or) if (q->prev != NULL)
        q->prev->next = q;
    else
        head = q;
    q->next->prev = q;
```

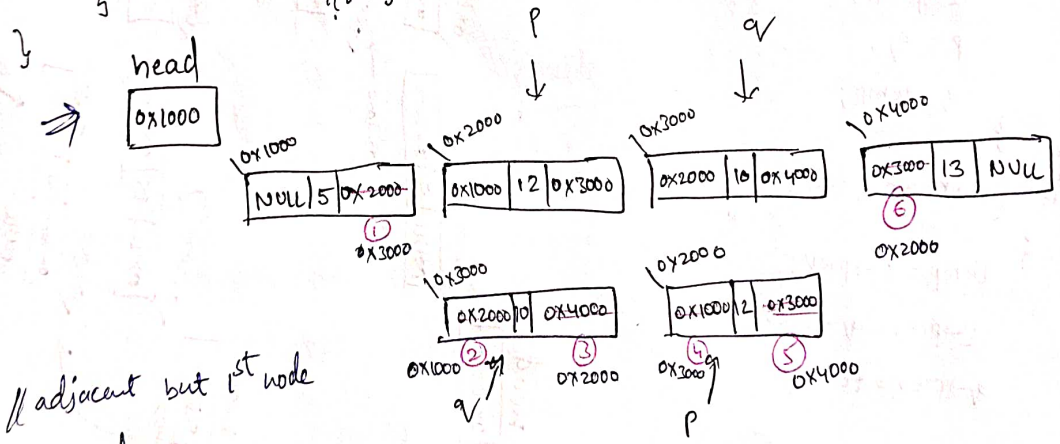


```
p->prev->next=p;
if (p->next != NULL)
    p->next->prev=p;
}
```

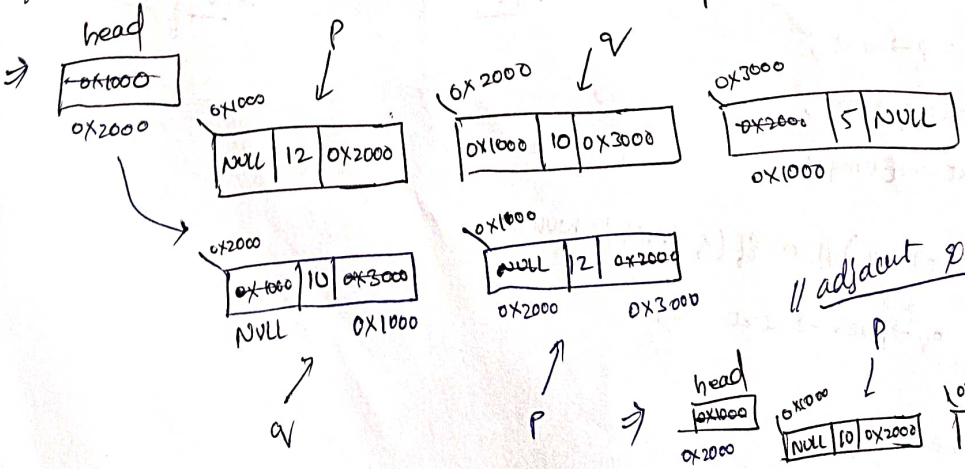
// Adjacent nodes

```
Else
{
    p->next=q->next;
    q->next=p;
    q->prev=p->prev;
    p->prev=q;
    if (p->next != NULL)
        p->next->prev=p;
    if (p!=head)    || (or if (q->prev != NULL)
        q->prev->next=q;
    Else
        head=q;
}
```

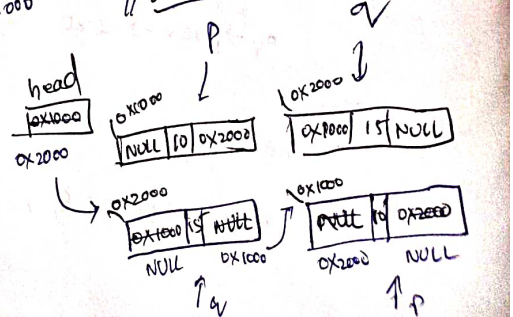
// adjacent but middle nodes



// adjacent but 1st node



// adjacent 2 only 2 nodes

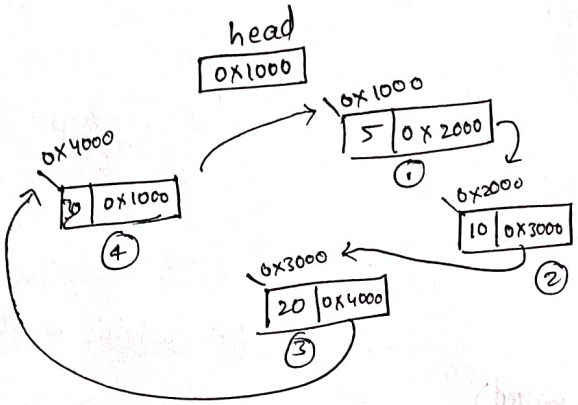


* Single circular list *

⇒ In circular linked list, last node link part contains 1st node's base address.

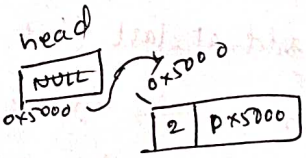
⇒ If the list is empty, head contains NULL.

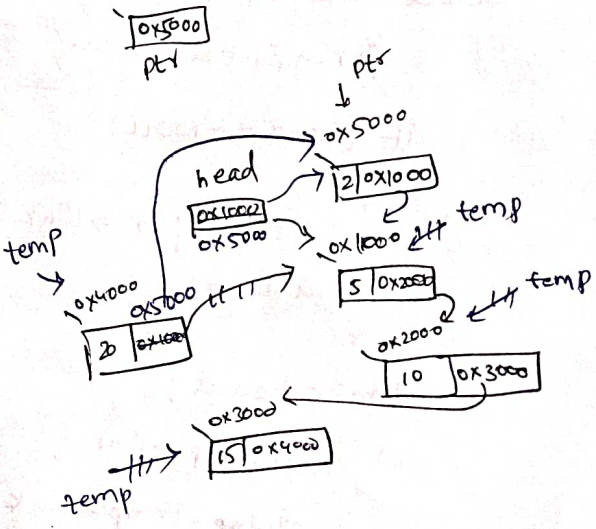
⇒ when list contains single node, it will point the same node.



① void add_at_beg(int num)

```
{
  struct node *temp;
  struct node *ptr = (struct node *) malloc (1 * sizeof(struct node));
    if (ptr == NULL)
    {
      pf("failed to allocate memory in heap \n");
      return;
    }
  ptr -> data = num;
  if (head == NULL)
  {
      head = ptr -> link = ptr;
      return;
  }
  temp = head;
  while (temp -> link != head)
        temp = temp -> link;
  ptr -> link = head;
  head = temp -> link = ptr;
}
```

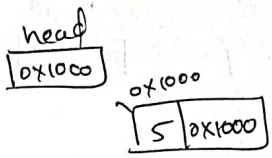




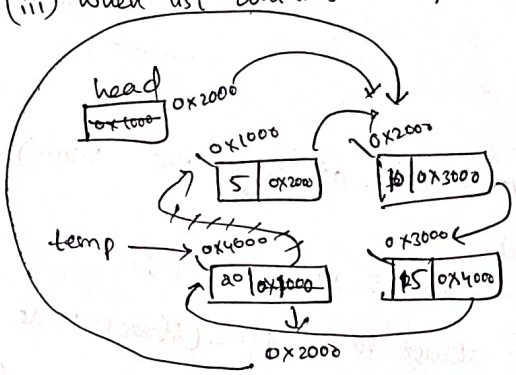
```
void del_at_beg ( )
{
    struct node * temp;
    if (head == NULL)
    {
        PF ("list is empty\n");
        return;
    }
    temp = head;
    if ( head -> link == head)
    {
        free (head);
        head = NULL;
        return;
    }
    temp = head;
    while (temp -> link != head)
        temp = temp -> link;
    temp -> link = head -> link;
    free (head);
    head = temp -> link;
}
```

(i)

(ii) when list contains single node

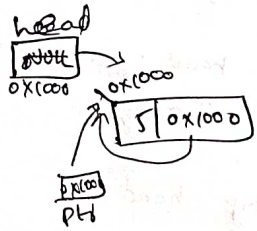


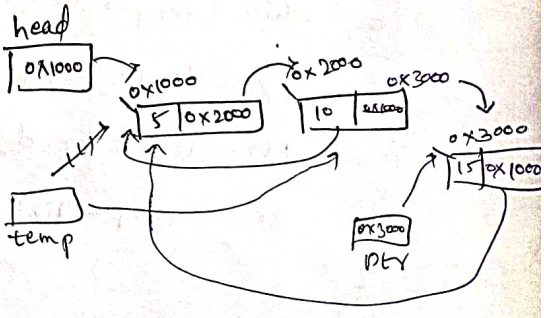
(iii) when list contains multiple nodes



(3)

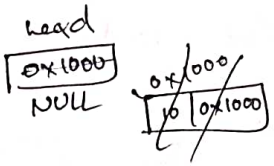
```
void add_at_last (int num)
{
    struct node *temp;
struct node *ptr = (struct node *) malloc ( 1*sizeof (struct node));
    ptr -> data = num;
    if (head == NULL)
    {
        head = ptr -> link = ptr;
        return;
    }
    temp = head;
    while (temp -> link != head)
        temp = temp -> link;
    temp -> link = ptr;
    ptr -> link = head;
}
```

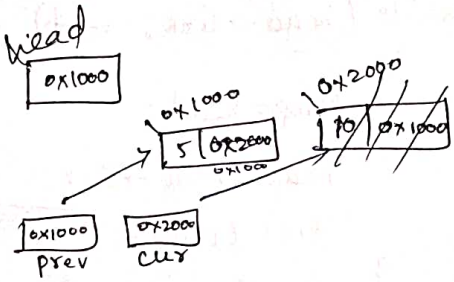




⇒ Void del_at_last ( )

```
{
    struct node *cur, *prev;
  if (head==NULL)
   {
     pf ("list empty\n");
     return;
   }
 if (head->link==head)
   {
     free (head);
     head=NULL;
   }
   prev=head;
   temp=head->link;
 while (temp->link!=head)
    {
      prev=cur;
      cur=cur->link;
    }
    prev->link=head;
     free (cur);
}
```





⇒ void display ( )

```
{
   struct node *temp;
   if (head==NULL)
   {
     pf ("list empty\n");
     return;
   }
   temp=head;
   do
   { pf ("%d\n", temp->data);
     temp=temp->link;
   } while (temp!=head);
```

⑥ ⇒ void del_Entire_list ( )

```
{
    struct node * temp, *P;
    if (head == NULL)
    {
        PF ("list empty\n");
        return;
    }
    if (head -> link == head)
    {
        free (head);
        head = NULL;
    }
    p = head;
    while (head -> link != head)
    {
        temp = head;
        head = head -> link;
        free (temp);
    }
    free (head);
    head = NULL;
}
```

⑦ count

⑧ search

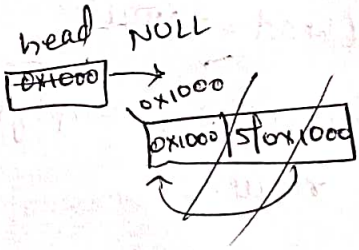
15/12/22

Circular Double linked list :-

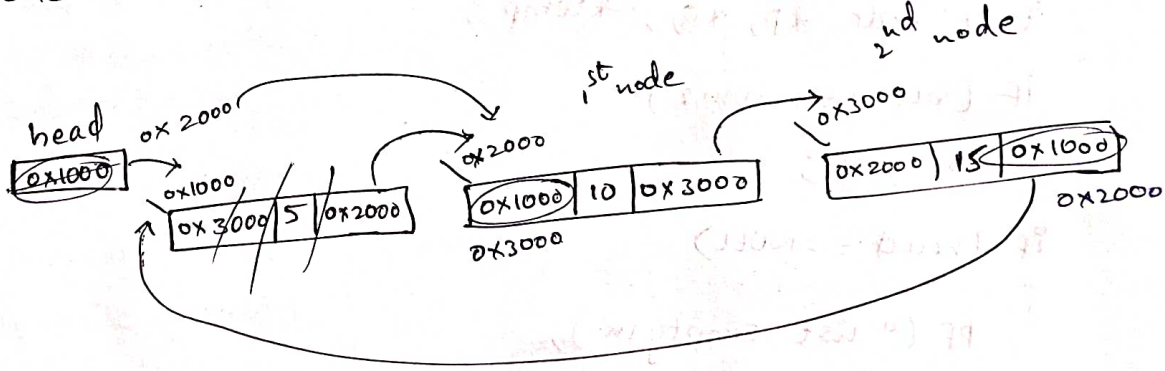
⇒ void del_particular_node (int num)

```
{
    struct node *temp;
    if (head == NULL)
    {
        PF ("list empty\n");
        return;
    }
    if (head -> data == num)    // first node
    {
        if (head -> next == head)    // single node
        {
            free (head);
            head = NULL;
            return;
        }
```

(or) if (head -> prev == head)



```
        temp = head;
    temp->next->prev = temp->prev;
    temp->prev->next = temp->next;
    head = temp->next;
    free (temp);
    return;
}

temp = head->next;
while (temp != head)   // except 1st node remaining all nodes
{
    if (temp->data == num)
    {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
        free (temp);
        return;
    }
    temp = temp->next;
}
PF ("Element not found \n");
}
```



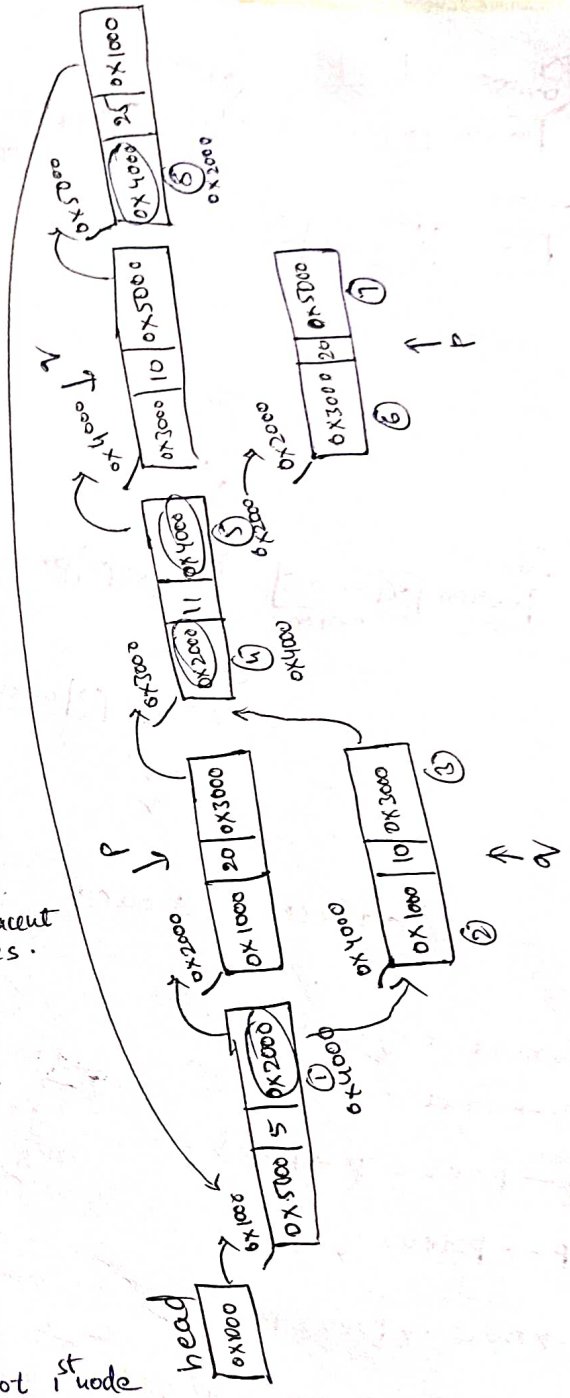

```
⇒ void create_list (int *ptr, int n)
  {
      int i;
      struct node *ptr;
    for (i=0; i<n; i++)
      {
          ptr = (struct node *) malloc (1 * sizeof (struct node));
          if (ptr == NULL)
          { PF ("Failed to allocate memory in heap \n");
            exit(1);
          }
```

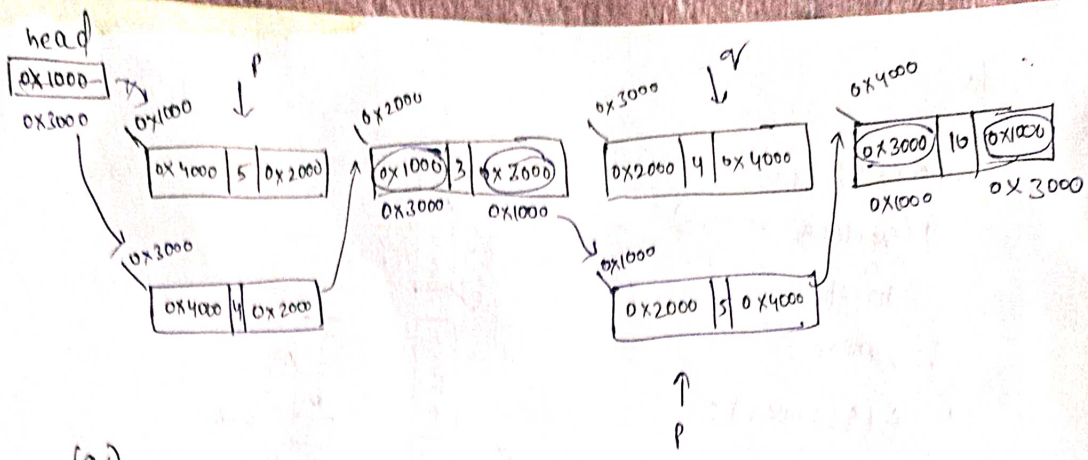
```
ptr -> data = iptr[i];
if (head == NULL)
  {
    ptr -> prev = ptr -> next = head = ptr;
  }
Else
  {
    Ptr -> prev = head -> prev;
    ptr -> next = head;
    ptr -> prev -> next = ptr;
    head -> prev = ptr;
  }
}
}
```

Swap

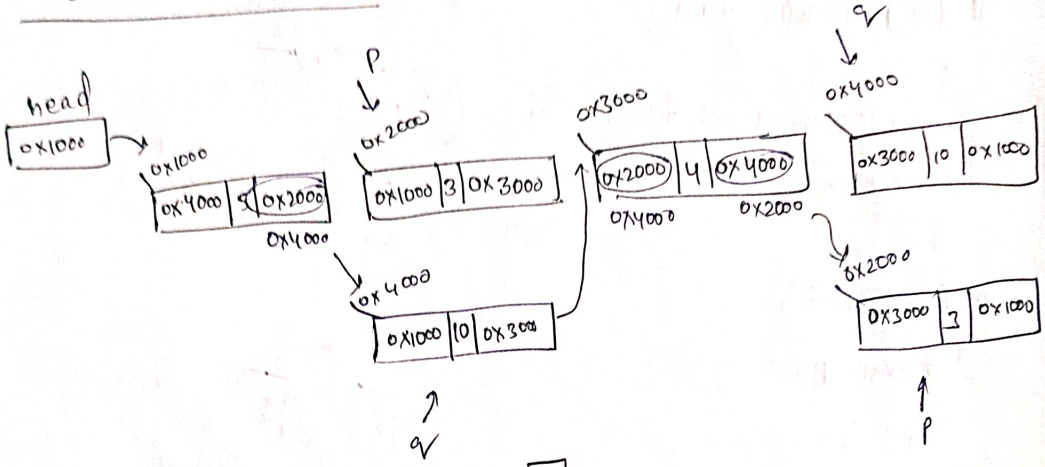
```
⇒ void swap_link (int num1, int num2)
  {
    int pos1=0, pos2=0, f1=0, f2=0
    Struct node *p, *q, *temp;
    if (num1 == num2)
      return;
    if (head == NULL)
    {
      PF("list empty\n");
      return;
    }
    p=q=head;
    do
    {
      pos1++;
      if (p -> data == num1)
      { f1=1;
        break;
      p = p -> next;
    } while (p != head);
```

```
do
{
    pos2++;
    if (q->data == num2)
    {  f2 = 1;
       break;
    }
    q = q->next;
} while (q != head);

if ( (f1 != 1) || (f2 != 1) )
{
    printf("Element not found\n");
    return;
}
if (pos1 > pos2)
{
    temp = p;
    p = q;
    q = temp;
}
if (p->next != q)        // For non-adjacent nodes.
{
    temp = p->prev;
    p->prev = q->prev;
    q->prev = temp;

    temp = p->next;
    p->next = q->next;
    q->next = temp;

    if (p != head)       // if it is not 1st node
        q->prev->next = q;
    else
    {
        head = q;
        head->prev->next = q;
    }

    q->next->prev = q;
    p->prev->next = p;
    p->next->prev = p;
}
```





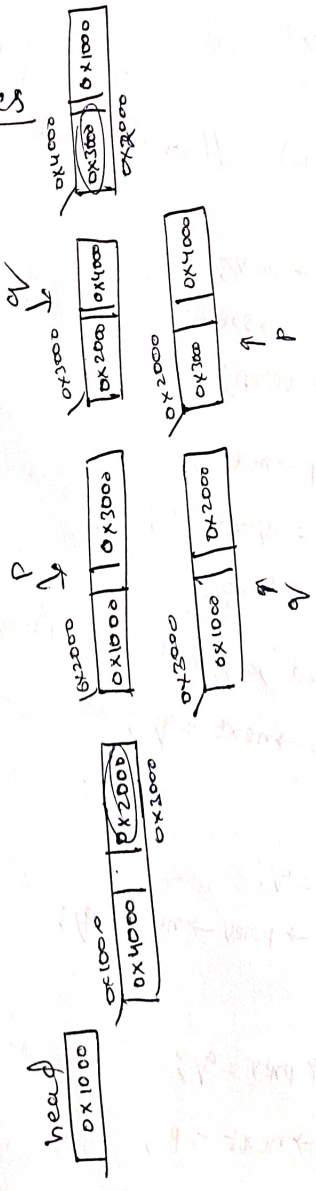
// (q) one node is last node



Else // adjacent nodes

{

```
p->next = q->next;
q->next = p;
q->prev = p->prev;
p->prev = q;
p->next->prev = p;
q->prev->next = q;
if (p==head)
    head=q;
}
}
```



16/12/22

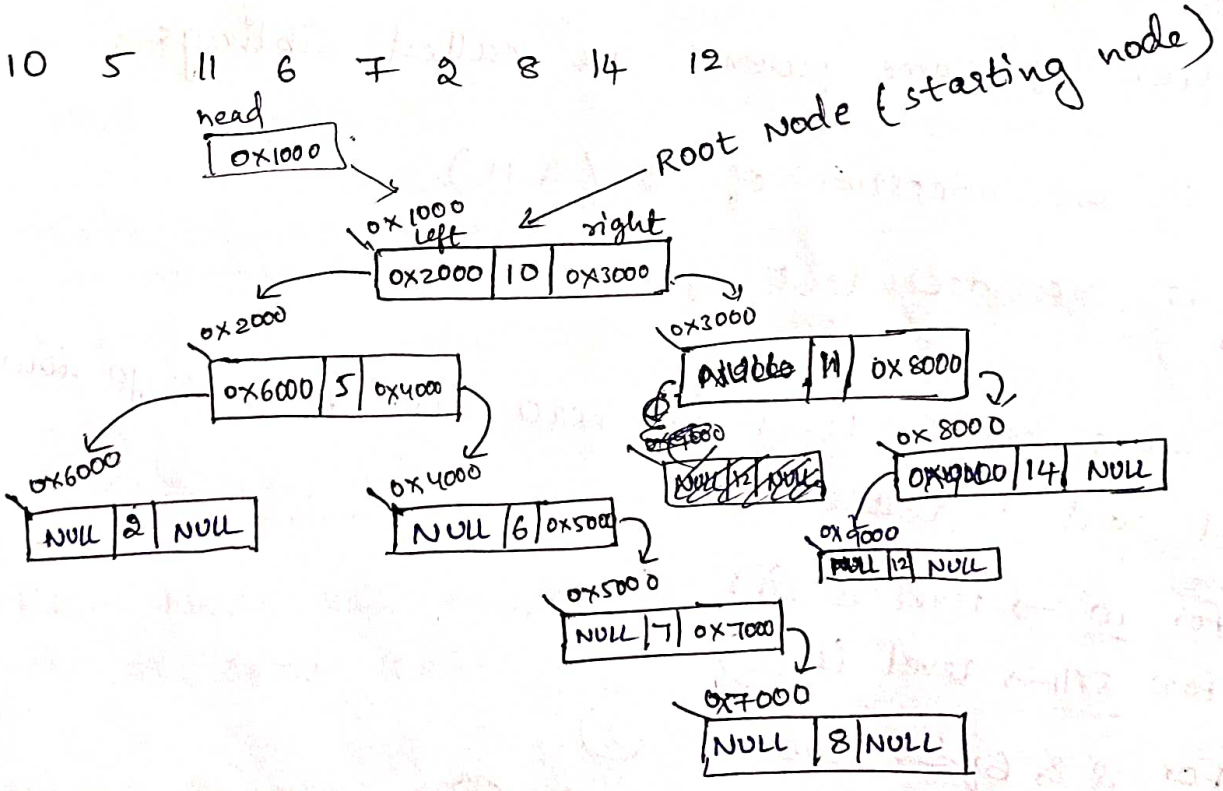
* Non-linear Data structures *

* Trees :-

⇒ Trees comes under Non-linear data structures.

```
Struct node
{
    Struct node *left;
    int data;
    Struct node *right;
};
```

10 5 11 6 7 2 8 14 12



① PRE ORDER - NLR = 10 5 2 6 7 8 11 14 12

② IN ORDER - LNR = 2 5 6 7 8 10 11 12 14

③ POST ORDER - LRN = 2 8 7 6 5 12 14 11 10

⇒ Each node contains multiple nodes

⇒ A tree structure is a way of representing the hirerachial nature of a structure in a graphical form.

⇒ A root node of a tree is a node which doesn't have parents.

⇒ Edge :- An edge refers to a link from parent to child (arrow marks)

⇒ Leaf node :- A node with no children is called leaf node. (2, 8, 12)

⇒ Children of same parent are called siblings

⇒ 10 is the ancestor of 5 (& 11)

⇒ 5 is the decendent of 10 (5 & 11)

⇒ The root node level is zero (0). As we go down, the level will increase.

For 10 → level is (0)
For 5 & 11 → level is (1)
For 2 & 6, 14 → Level is (2)
7, 12 comes under level (3)
8 comes under level (4)

⇒ The set of all nodes at a given depth is called a level of tree.

⇒ Height of a Node is the length of the path from that node to the deepest node.

⇒ Height of the tree is the maximum height among all the heights in the tree.

⇒ The size of the node is the no. of decendents it has, including itself.

⇒ If every node in a tree has only one (1) child that is called skew tree.

⇒ If every node has only left child, then we call it as left skew tree.

⇒ If every node has only right child, then we call it as right skew tree.

(*) Binary tree :-

⇒ A tree is called binary tree, if each node has zero (0) child, 1 child (or) 2 children.

(*) Strict Binary tree :-

⇒ A tree which node has ~~exactly~~ 2 children (or) NO children

(*) Full Binary tree :-

⇒ Each node has exactly 2 children & all leaf nodes are at same level.

⇒ Height of root node (10) is zero.

⇒ Height of 5 is 1. (level & height is same)

⇒ Measuring the height & measuring the level both are same.

⇒ In each level we can have maximum 2^{level} nodes only

for 10 → 2^0 ⇒ 1 node (max)

for 5, 11 → 2^1 ⇒ 2 nodes

⇒ To implement these Binary tree operations we have to use recursive functions.

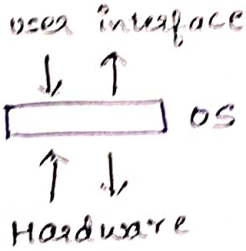
19/12/22

* System Programming *

* OS concepts :-

⇒ Operating system (OS) is an executable binary file which acts as a mediator between user interface & hardware interface.

⇒ The operating system has to handle the request coming from user interface as well as hardware interface.



⇒ User can send the request to the OS in two ways i.e.,

① GUI (through applications) → this applications can be developed by using (Java or python)

② CLI (commands).

shell (application) → commands are written/developed in 'c'

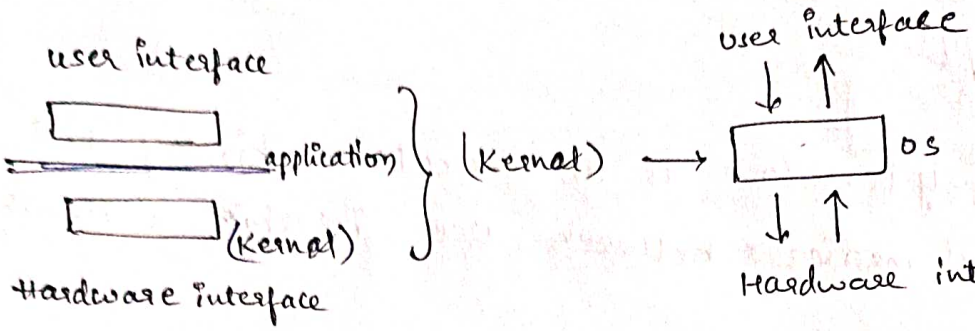
⇒ 'c' code is pre-compiled in OS executable file format

⇒ Even though the applications are developed in another programming languages, but they again internally interacts with the same set of commands (c).

⇒ This commands are ntg but an executable binary files which again exist in ELF format.

⇒ These commands can be developed in by 'c' library functions (or) System calls.

⇒ In an operating system (Linux based (or) Unix based) Kernal is the main core component which will service the request coming from user interface & hardware.



⇒ Every library function internally invokes one system call.

⇒ Kernal internally maintains sub system :-

5 sub components in Kernal
1. process management
2. File "
3. Memory "
4. Network "
5. I/O "
+ Device drivers

Eg :- 3-4 commands
↓
C-library func
↓
System call
↓
Sys - system call → Kernal equivalent call.

→ every 'c' library func internally invokes 1 system call

⇒ we have total 300+ system calls. These system calls are again divided based on Kernal sub components.

⇒ These 5 sub components internally interacts with each other

⇒ Each system call will have an equivalent Kernal function that starts with sys prefix.

⇒ Each system call will have a unique number.

⇒ Along with these sub systems we will understand

⑥ → Threads → mutexes → locking mechanism.
↳ conditional variables

⑦ → signals

⑧ → IPC mechanisms → system-V → we use system calls.
pipes
Named pipes (FIFOs)
message queues (~~FIFOs~~) semaphore
shared memory region, ~~semaphore~~

→ POSIX → we will use library functions

↳ Inter process communication.

⇒ A process cannot interact directly with another process. so they have to use either system-V or posix mechanism.

⟹ pipes, named pipes, message queues they have inbuilt synchronization.

⟹ shared memory, ~~semaphores~~, threads doesn't have any synchronization
⇒ for synchronization we will use semaphores or signalling.

⇒ The processes that are running under user space of RAM, their information is monitored & maintained by the kernal in proc virtual file system.

⇒ Signals are used to send the information from one Process to another process and to stop (or) terminate the process and to implement blocking mechanism.

⇒ Every signal will have a standard behaviour (or) execution, Even we can change the behaviour of that too...

⇒ Total we have 64 signals (under this 32 are pre-defined and remaining we can define) (user define)

20/12/22

* Process :- It is ntg but a program under execution.

⇒ Each process is associated with a process identifier (pid) (unique no (+ve))

⇒ When we create a new process, an entry gets created (or updated) in proc virtual file system, that entry is ntg but a folder with Pid name. In that folder it will maintain process specific information.

⇒ Every operating system will support a file system.

(*) For linux it will support ext2, ext3, ext4, FAT32, FAT16.

⇒ As soon as the process gets terminated, the entry gets destroyed from proc virtual file system.

(*) How do you see the currently running processes info.?

⇒ By using ps -ef command.

⇒ This command will read that information from proc virtual file system.

Assignment

(*) Check the various flags (or) options used with ps command & their importance (various flags)

⇒ From GUI interface we can see this info from system monitor.

⇒ Each process is created from another process. The newly created process is known as child process. The process which created is known as parent process. Both processes will have process id's.

⇒ PPid is known as parent process id.

⇒ Each process will have pid and ppid.

⇒ The first process that is loaded during the system boot up time is init process (pid-1). Before this one process gets loaded i.e., known as swapper (p-id-0).

⇒ Through this init process, multiple processes gets loaded.

⇒ This init process gets terminated when system shut downs

* This process is loaded under user space of RAM.
* In kernal space of RAM, we will have only kernal image.

⇒ This kernal has to handle the multiple requests coming from user space processes through system call interface.

⇒ Every library func used in a program will invoke a specific system call.

⇒ Each system call will have an equivalent kernal call, so the each system call will have an unique positive number. This system call number information is maintained in a header file.

⇒ These system calls & their numbering information is maintained in a table (i.e, system call table).

⇒ When a system call is invoked, the execution changes from user mode to kernal mode, again from kernal mode to user mode.

* ⇒ From user space, we can't access kernal data structures
⇒ Some kernal specific information we can acess through commands (or) system calls.

⇒ From Kernal point of view, each process is visualized as a Kernal thread.

⇒ For each process, Kernal maintains a data structure. (struct task_struct) type which is pre-defined func which is defined in sys call.h header file. path /usr/include/linux

⇒ This structure is known as process control block (or) task control block. This process control block's some of the membess information is maintained in proc virtual file System.

⇒ when a new process is loaded (or) created in user space, Kernal creates a new PCB in Kernal space of RAM. The newly loaded (or) created process info is maintained in that PCB.

⇒ The proc virtual file system entries changes dynamically based on process execution.

⇒ Kernal maintains all these PCB's in queue's. In general, Kernal maintains two queue's ⇒ ① Running queue
② waiting queue

⇒ The processes which are under ready for execution their PCB's are maintained Under Running queue.

⇒ The processes which are suspended (or) blocked & stopped (or) waiting for some event, their PCB's are maintained in waiting queue.

⇒ From Kernal point of view, each process request, it will trace (or) track through PCB. Each PCB is tracked by P-id.

⇒ The terminated process p-id is not assigned to the new process.

⇒ Switching the execution from User mode to Kernal mode, it will raises an interrupt by system call handler.

21/12/22

* System call :- A system call is a controlled entry point into the Kernal, allowing a process to request the Kernal to Perfoam some actions on behalf of the process.

⇒ System call is nothing but a function.

⇒ (API) ⇒ Application programming interface.

⇒ A system call will changes the processor (CPU) state from user mode to Kernal mode. so that the CPU can access protected Kernal memory.

⇒ The set of system calls are fixed. Each system call is identified by a unique number. Each system call may have set of arguments that specify information to be transferred from user space to Kernal space and vice. versa.

⇒ Invoking a system call is similar to calling a fun^c. The implementation of system call depends upon architecture

⇒ In X-86, 32 bit machine, the steps are as follows :-

① The application program makes a system call by invoking a wrapper fun^c in 'c' library

⇒ The wrapper fun^c must make all of the system argument available to the system call trap handling routine. These arguments are passed to the wrapper via stack but the Kernal expects them in the specific registers, the

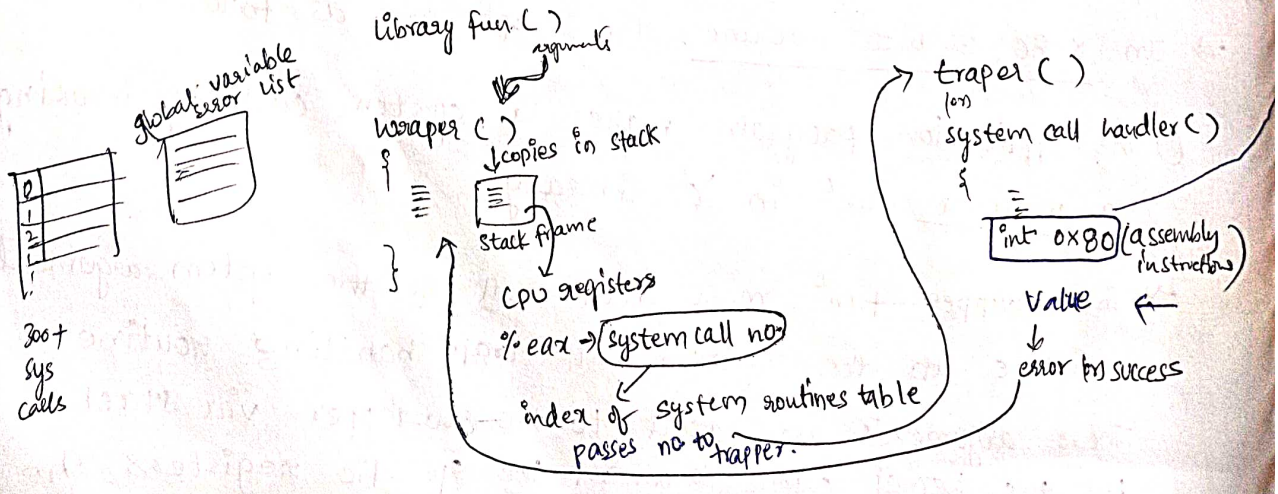
wrapper fun^{c} copies the arguments to these registers. To identify the system call, even it copies system call number in to a specific register eax.

⇒ Now, the wrapper fun^{c} executes a trap machine instruction. int 0x 80 (which it raises the interrupt) to switch the processor from user mode to kernal mode.

⇒ Now the kernal invokes system_call() routine to handle the trap. This handler saves the register values on the kernal stack & checks the validity of the system call number. and invokes appropriate system call service routine, which is found by using the system call number to index a table of all system call service routines.

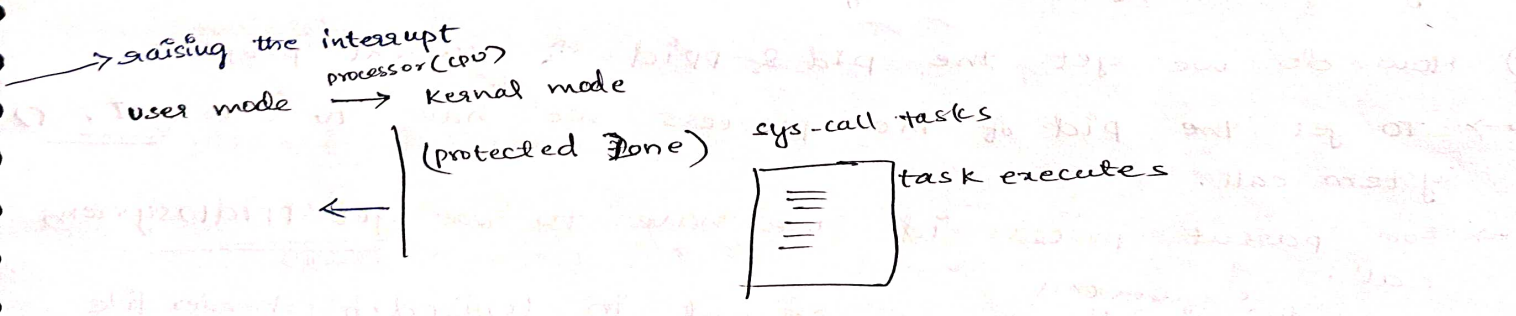
⇒ Finally, the service routine returns its result status to system call routine, restores register values from the kernal stack & places the system call return no. values on the stack. It returns to the wrapper fun^{c}, simultaneously returning to the user mode.

⇒ If the returned value of the system call service routine indicated an error, the wrapper fun^{c} sets the global variable error number using this value. The wrapper fun^{c} then returns to the caller, providing an integer return value indicating the success or failure of system call.



Assignment

⇒ How a system call is executed in ARM architecture.



22/12/22

⇒ PCB (Process control Block) contains P-id (process id), PPid (parent process id), Fd table (File descriptor), page table, process state, owner id, group id, context area, Signal disposition table (SDT), blocking signals info, Pending signals info, signal Mask, time stamps etc.

PCB's → nodes in a double linked list
Running queue.

⇒ FD table contains opened file information from the process.

⇒ page table contains process pages info which are loaded into RAM.

⇒ Process state → shows the status of process (Running, executing, terminated, etc)

⇒ context area → It will store CPU registers info when they are suspended / blocked / stopped.

⇒ Signal disposition table (SDT) → It maintains 64 signals behaviour

⇒ Blocking signals info → It maintains the info whether it is blocked / terminated (some blocked signals).

⇒ Signal Masks → (Blocked signals info).

⊛ How do we get the pid & ppid of current process

⇒ To get the pid of the process we have to use getpid() System call.

⇒ For parent process id we have to use getppid() system call.

⇒ These fun^c system call declarations are present in unistd.h header file

return type → pid_t getpid();
pid_t getppid(); → it is the typedef of signed int

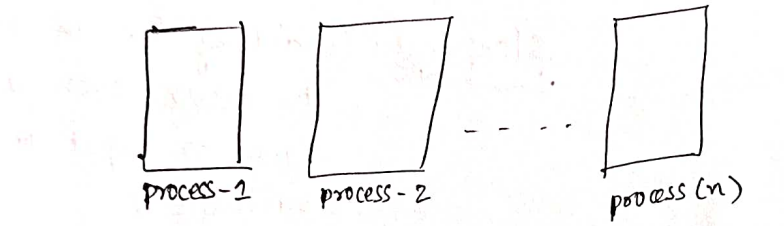
⇒ compiler doesn't know about any system call. We must include corresponding header file when we use the system call.

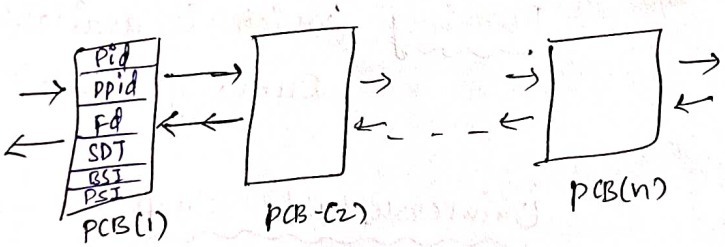
(unistd.h) If we don't include these header file, we will get compilation error.

* $ man 2 system call name

→ To know which system call we have to use
→ To know which header file is to add for particular system call.

⇒ type.h ⇒ It consists of type def 's info.





PCB are connected as double linked list.
maintained in " " " as Running Queue

```
#include <unistd.h>
main()
{
  int pid, ppid;
// pid_t  pid, ppid;
    pid = getpid();
    Printf(" process pid = %d\n", pid);
    ppid = getppid();
    Printf(" parent process pid = %d\n", ppid);
    while(1);
}
```

$ cat /proc/<pid>/status will displays the pid.

⇒ Running queues maintains the process PCB's which are ready for execution.

⊛ Process states :-

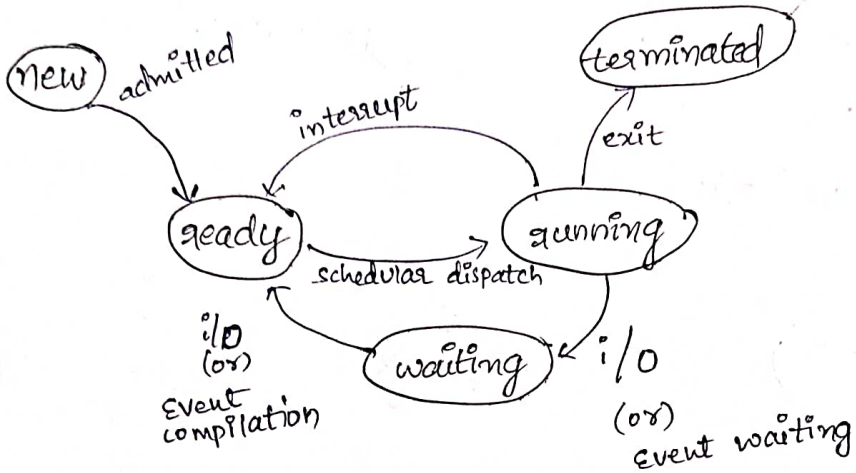
⇒ To see the currently running processes states information we have to use

$ ps ax command

Stat code	Description
S	Sleeping (waiting for an event to occur such as signal or input to become available)
R	Running (which is available on Running queue).
D	Uninterruptable sleep (usually waiting for input or output to complete)
T	Stop (usually stopped by shell job control or the process is under the control of a debugger.
Z	Zombine process.
N	Low priority task
s	process is a session leader
+ (plus)	process is in the foreground process group

2 process is multi threaded

< high priority task.



New :- The process is being created

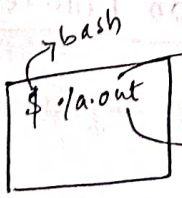
Running :- The instructions are being executed.

Waiting :- The process is waiting ~~from~~ for some event to occur
such as an I/O completion or reception of a signal

Ready :- The process is waiting to be assigned to a processor

terminated :- The process has finished execution

⊛ How to create a new process (identical process)



→ It uses fork() to create new process

→ A process can be created by using fork. (or) V_fork system call.

→ fork system call is also used to create child process as new process itself is a child process.

fork() → invokes / V_fork()

↓ It invokes

do_fork( )

↓ formal arguments will changes

It creates under clone process

Find out the status of process

①
```
#include<unistd.h>
main()
{
    int x;
    PF("enter i/p\n");
    SF("%d", &x);
    PF("%d", x);
}
```

②
```
#include<unistd.h>
main()
{
    while(1)
    PF("Hi");
}
```

⇒ 24/12/22

⇒ we can use fork(), v_fork() & clone() to create a new process.

⇒ fork(), v_fork() & clone() internally invokes do_fork() system call.

⇒ Return type of fork() is pid_t

```
pid_t fork(void);
```

⇒ Process id is 32 bit value as it is signed int type. (4 bytes)

NOTE:-

⇒ MOST of the system calls will return on failure -1. So, we must verify the return values of system calls.

⇒ Fork() system call will return -1 on failure & on success it will return twice.

① → one is zero (i.e., in child process)

② → second is pid of child process (in parent process)

⇒ The newly created child process execution starts after fork() system call.

User written program

pid :- 1000
(Parent process)

```
#include <unistd.h>
main()
{
    pid_t pid;
    pid = fork();
}
```

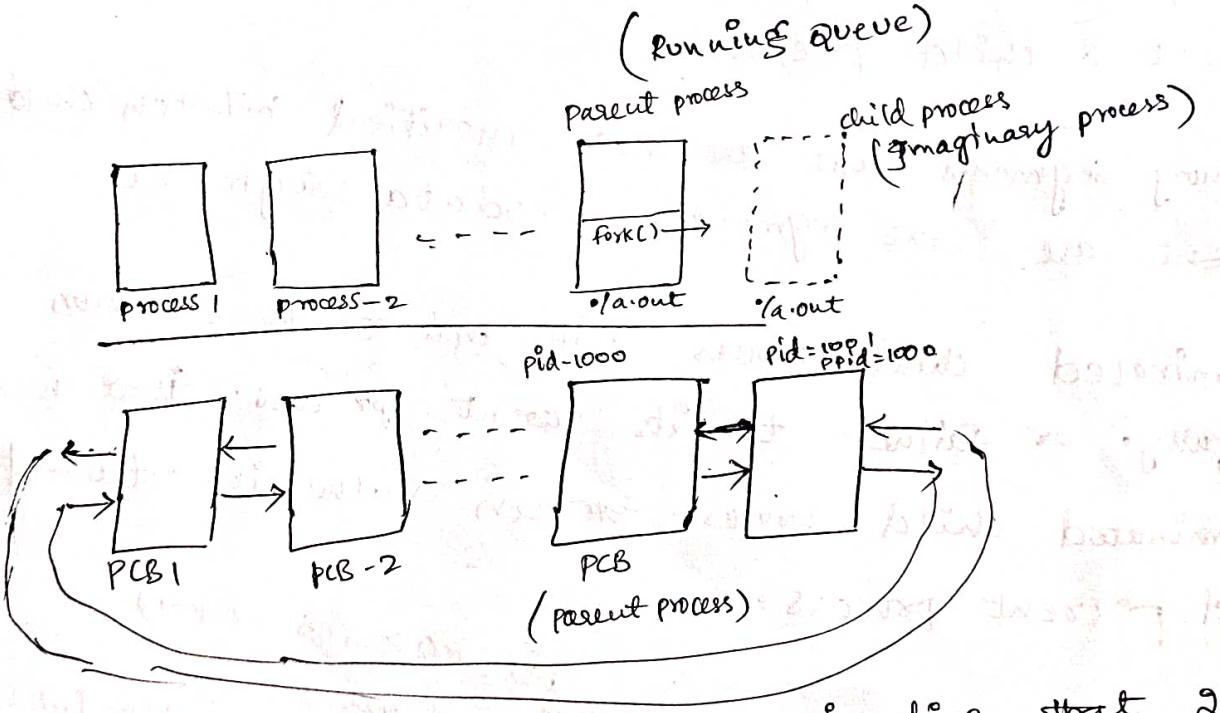
Single process

→ succesful ⇒

pid :- 1000 > (Greater than 1000)
(assume :- 1001)
(child process)
⇒ identical process created by fork()

```
main()
{
    pid_t pid;
    pid = fork();
}
```

← child process execution will start after fork()



⇒ After fork, Now we have to visualize that 2 processes are running. Both the processes are present in running queue.

⇒ we can't say which process will terminate First. Both the processes will not execute the same code. That differentiation we are creating based on return value of the fork

⇒ In user space, initially both parent & child will share same set of pages that means both parent & child will share same memory segments initially. But, some part of the code is executed under parent & some part of the code is executed under child

⇒ In Kernal space, both processes will have different PCB's.

⇒ As soon as, if one of the process modifies the data, it will create a duplicate page for that particular process based on copy on write (or) write on copy technique. And the page table entry gets modified in PCB of that particular process. But will get same virtual address for both parent & child process.

⇒ The memory segments that are not modified between child & parent are text segment & rodata segment

⇒ Each terminated child process will update its execution successfully or failure to its parent process. That means the terminated child process return value is returned to parent process.

```
pid = 1000
(parent process)
#include <unistd.h>
main()
{
  pid_t pid;
  pid = fork();
```

pid = 1001 → child process pid

→ sucess

```
pid: 1009 (assume: 1001)
(child process)
identical process created by fork()
main()
{
  pid_t pid;
  pid = fork();
```

pid = 0

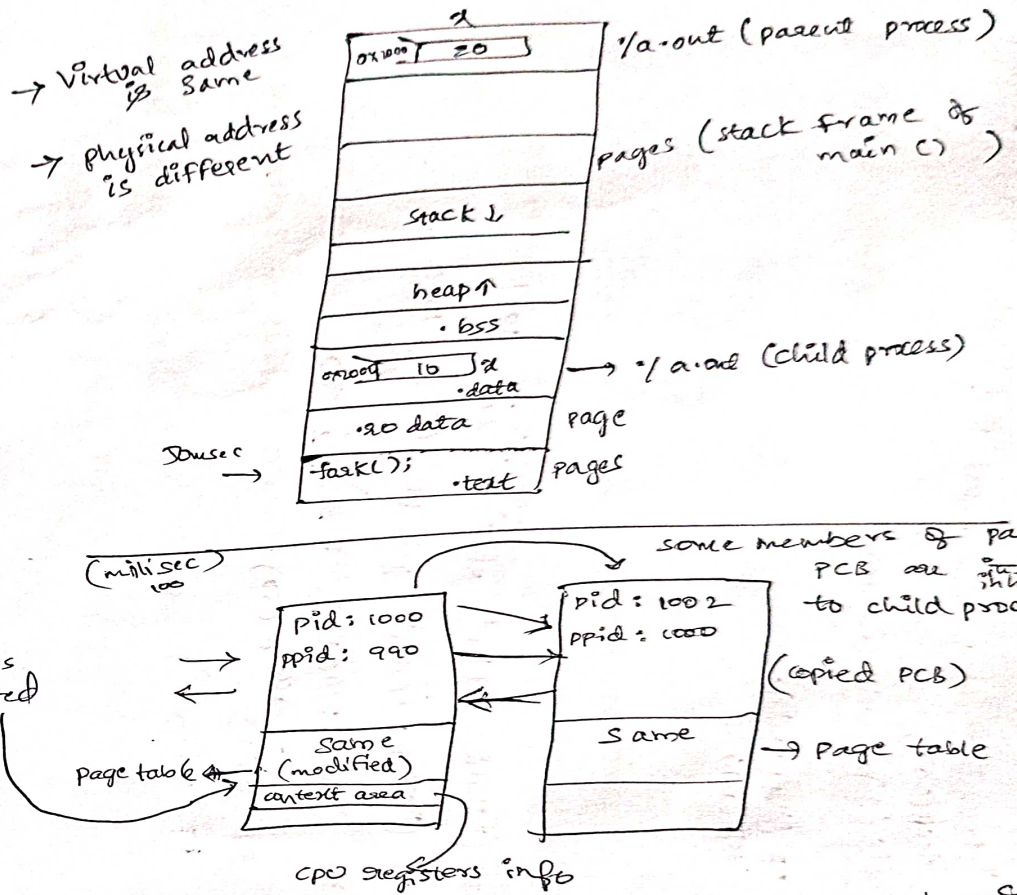
← child process execution will start after fork()

```
if (pid>0)
{
        int i=10;
PF("Parent : %d \n", getpid());          1000
PF("parent :child : child proces pid : %d\n", pid));   1001
    sleep(20);
while(1) // while (i<=100)
{
  PF("1111");
  sleep(1);
}
}
else
{
PF("child : %d \n", getpid());          1001
PF("child : %d\n", getppid());          1000
PF("child : %d \n", pid);               0
  while(1) --> sleep(50);
  PF("2222");
}
}
```

This is executed under parent process (Task-1)

This is executed under child process (TASK-2)

```
int x=10;
main()
{
  pid_t pid;
  Printf("%p-%d\n", &x, x);
  pid = fork();
  if (pid<0)
  {F}
    Pf("failed to create child\n");
    exit(1);
  }
  if (pid>0)
  {P}
    Pf("Parent: %p-%d\n", &x, x);
(100 msec) ⇒ x=20; → (cow) → parent process → new page allocated
    Pf("Parent: %p-%d\n", &x, x);
    exit(0);
  }
  else
  {
    sleep(3);
    Pf("child: %p-%d\n", &x, x);
    exit(3);
  }
}
```



⇒ some members of PCB is copied into child process PCB.

⇒ After copy on write (cow) technique, the modifications are done in parent process are not reflected in child process & vice versa.

⇒ Suppose, if any of the process uses blocking call such as wait ( ) [wait event calls] paused ( ), sleep ( ) and blocking calls like scanf ( ) etc.

⇒ That process PCB is moved from running queue to waiting queue. After some events are occured or the time is expired, After providing the input that process PCB is again moved from waiting queue to running queue.

⇒ The process PCB's which are present in waiting queue. Scheduler will not assign CPU time.

⇒ Scheduler will assign the CPU time for with processes (PCB's) which are present in running queue.

⇒ Before Moving PCB from waiting queue to running queue, Kernel uses signalling mechanism internally to ~~wake up~~ activate the processes. (Running state).

⇒ **Pre emption :-**

Suspending the current process execution & switching to the next process.

(*) In what scenario's pre emption will occur?

① when given CPU time for the process is expired (or) elapsed.

② when we use Blocking calls in current process.

③ when exceptions are raised or when signal handlers is executed.

④ when an interrupt raised

⑤ when signal handler is executing, the current working process is suspended.

⇒ When an exception (or) interrupt is raised, CPU will suspend the current process execution & executes the exceptional handler (or) interrupt handler after that it will continue the remaining execution.

⇒ Before pre-emption (or) before switching to the next process the instant where the execution is suspended is ntg but this CPU registers information is copied to the context area of the PCB.

⇒ When signal handler func is executing, the process (parent) will get suspended.

⇒ When we create a new child process, that new child process is added to the running queue. Meanwhile, if parent process gets terminated the PCB ~~which~~ belongs to the parent process and pages are belongs to parent process (which are not common b/w child process are destroyed.

⇒ Now, the child process becomes orphan process. In this case, the ppid of the child process will change. then init process will become the parent for that orphan process.

(*) verify the orphan process ppid and parent process name ?

(A) ⇒ orphan process parent name is init

) when you create a new child process, tell me the members of PCB that are inherited from parent to child process.

① Real user id
② Real group id
③ Effective user id
④ Effective group id

(2) Suplementary group ids
(3) process group id.
(4) Session id
(5) The set user id, set group id flags ~~current~~
(6) current working directory.
(7) file mode creation mask
(8) Page table.
(9) FD table
(10) SDT table (signal disposition table).
(11) Environment
(12) ~~Attached~~ shared memory segments.
(13) Memory Mappings
(14) Resovre limits.

⇒ There are different members b/w parent & child ~~PCB~~'s are

(1) Retuan value's from fork() are different.
(2) Process id's are different.
(3) Even parent processes are different.
(4) The child utilization time, set time are set to zero. current time & current set time are set to zero.
(5) The file locks set by Parent are not inherited by the child.
(6) Pending alarm's are cleared for the child.
(7) The pendings signals information is set to Empty.

⇒

Sleep System call :-

⇒ Header file is <unistd.h>

⇒ return type is unsigned int.

⇒ fun^c name is sleep.

⇒ It will take unsigned int seconds

single i/p → unsigned int sleep (unsigned int secs);

⇒ This will return zero (or) unslept seconds.

⇒ (0) on success

⇒ unslept seconds on failure.

⇒ This fun^c causes the calling process to be suspended until either (1) The amount of wall clock time specified by secs has - elapsed (expired).

(2) A signal is caught by the process & the signal handler returns.

⇒ When we use this system call the process execution is suspended until the time is expired. During this time that process PCB is moved from running queue to waiting queue. As soon as the time is expired, kernal will send the signal to wake up the process after that, the PCB is moved from waiting queue to running queue.

26/12/22

Differences b/w fork() & V-fork()

⊛ V_fork() system call :-

→ It is similar to fork system call.

→ It will return the value twice after succesful execution

→ In parent process, it will return child process pid, In child process it will return zero (0). On failure this system call will return -1.

⇒ The child process created by V-fork, the parent process execution is suspended (waiting queue) until the child process completes its execution.

⇒ Both parent & child will have same set of pages ntg but, both processes will have same set of memory segments. so, these NO copy on write (cow) technique is applied that means both parent & child will have same set of pages. The modification done in child is reflected to parent.

⇒
```
int x=10;
main()
{
  pid_t pid;
  pid = fork();
  if (pid<0)
  {
    PF("failed to create child\n");
    exit(1);
  }
  else if(pid>0)
  {
    PF("%p-%d\n", &x, x);
    exit(0);
  }
  else
  {
    sleep(2);
    x=20;
    PF("%p-%d\n", &x, x);
    exit(0);
  }
}
```

pid = V-fork();

child process execution will start after fork();

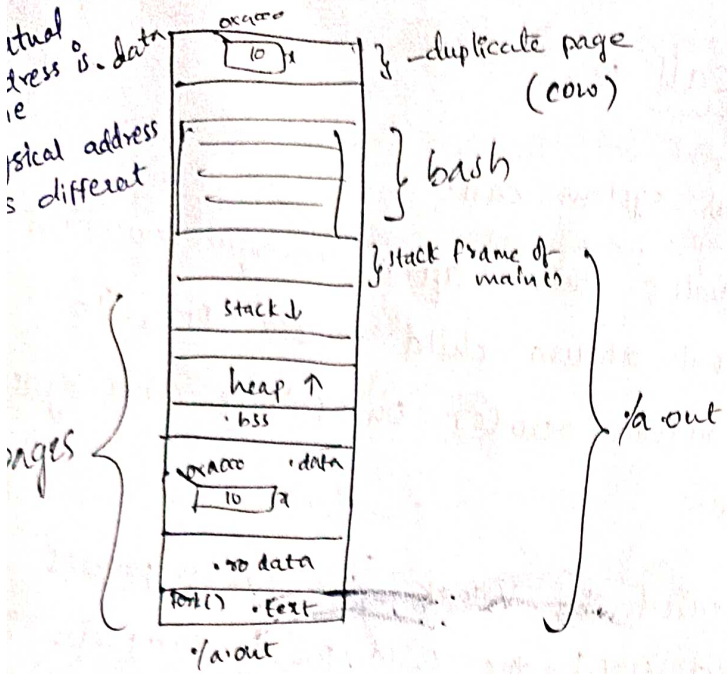
(100 msec) parent process (given cpu time expired)

0xa000 - 10

Blocking call (child process execution is suspended for 2 sec)

0xa000 - 20

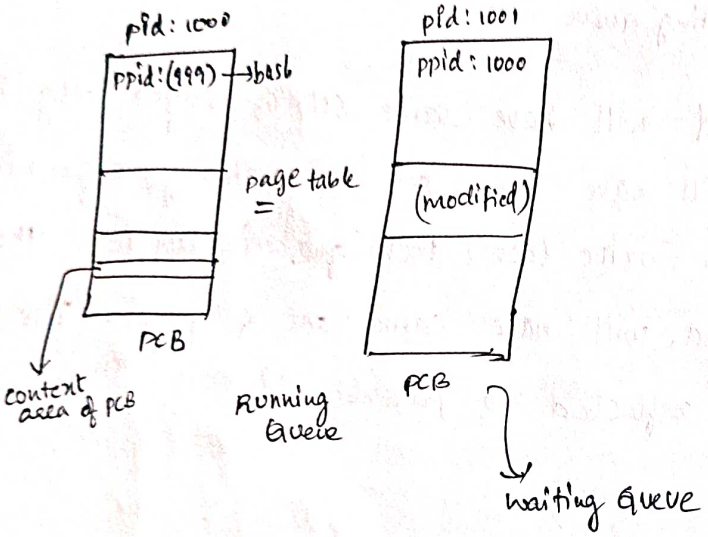
fork() system call



sleep in child

parent = 0xa000 - 10

child = 0xa000 - 20,

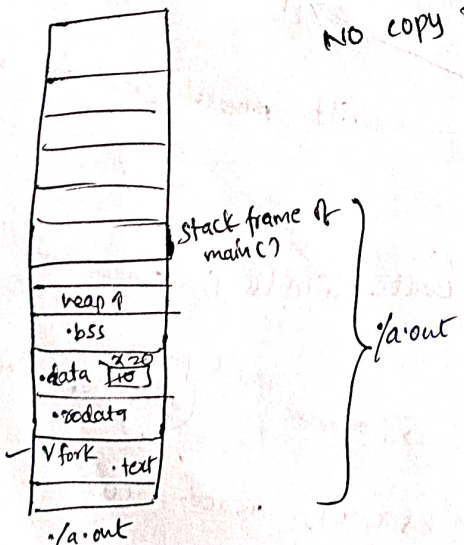


sleep() in parent

child process = 0xa000 - 20

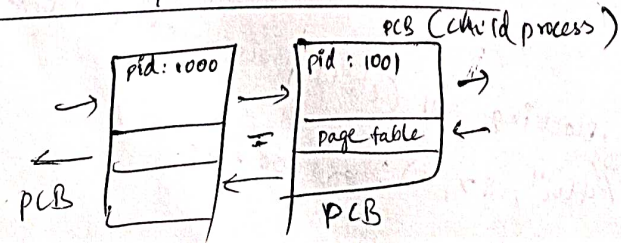
parent proces = 0xa000 - 10

Vfork() sys call



No copy on write (COW) technique is applied

both parent & child will have same set of pages



o/p

child proces = 0xa000 - 20

parent proces = 0xa000 - 20

(*) In how many ways we can terminate a process?

Process termination

⇒ when a process terminates, the Kernal will releases the resources owned by the process & notifies the child's termination to its parent.

⇒ we can see normal termination & abnormal termination

(*) Normal termination we can see in (5) ways:-

(1) Return from main func (If we use return stmt in the main func).

(2) calling exit func in main or any func. (It will terminate the complete process).

(3) calling _exit func/system call from a process

(4) Return of last thread from its start routine

(5) calling pthread_exit from the last thread.

* Abnormal termination occurs in (3) ways:-

(1) calling abort func from a process which will generate core dump file.

(2) ~~After~~ receiving a signal or After caught the signal.

(3) Response of the last thread to a cancellation request.

```
#include <stdlib.h>
void exit (int status);
```

⇒ The status is return to its parent.

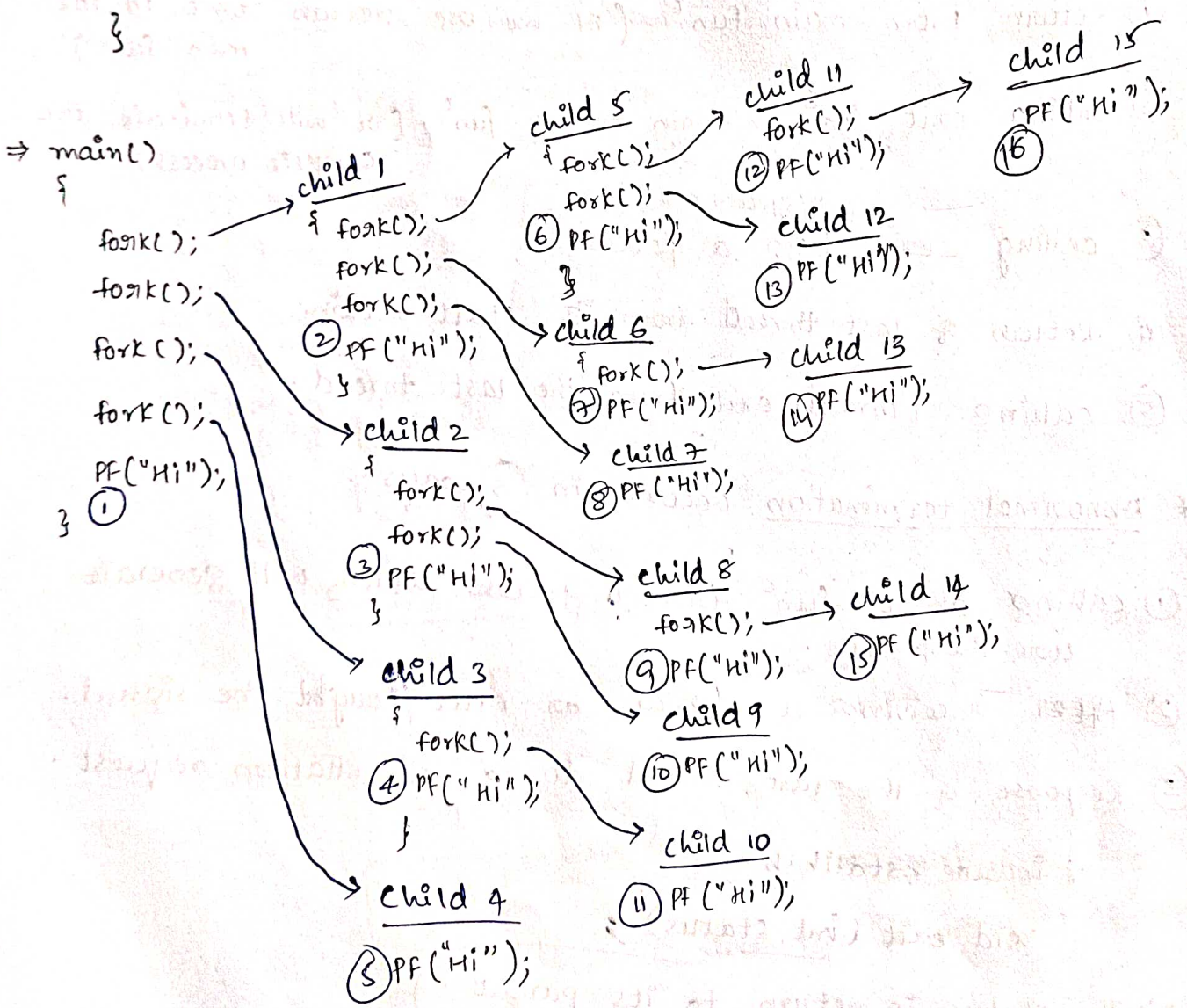
current process is terminated
&
status value is returned to its parent

27/12/22

⇒ main()
{
fork();
fork();
fork();
fork();
Printf("Hi\n");
}

⇒ If we use fork() → 'n' times,
it will create $2^n - 1$ child processes.

⇒ For 'n' times → it will print 2^n times



⇒ It will print Hi 16 times when we use fork() system call for 4 times.

(*) Signals :-

⇒ Signals are nothing but software interrupts.

→ we have total 64 signals. Out of 64, 32 are pre defined signals (which will have pre-defined behaviour).

⇒ These signals are used in a process for various purposes

① sometimes To stop the process execution ② sometimes to continue the stopped process execution.

③ sometimes to terminate the process.

④ Sometimes to send the event or signal to a process.

⑤ Sometimes to perform the specific operation on behalf of that signal.

⇒ when a process receives the signal, CPU will stops the current process execution & executes the behaviour of that signal. After that, it will continue the process execution.

⇒ The default behaviour of most of the signals is to terminate the process.

⇒ Signal is an event generated by the system in response to some condition upon receiving the signal, the process will intern perform an action.

⇒ So, Here we have 2 questions :-

① How signals are generated ?

② How signals are delivered to a process ?

1A We can generate a signal in 3 ways

① Key combination.
② By using command.
③ By using system call.

① Key combination :-

① **ctrl + C** :- It will generate SIGINT signal. So, this will terminate the process.

② **ctrl + Z** :- It will generate SIGSTOP signal. So, it will suspend or stops the process execution.

③ ctrl+Q :-
⇒ If we use the same key combination again, it will generate SIGCONT (signal continue) signal. so, it will resume the stopped process execution.

④ ctrl+z ⇒ SIGTSTP (terminate) → tty

⇒ ① main()
{
int x=1;
while(x>0)
{
PF("Hi");
x++;
}
}

ctrl+C ⇒ terminates the process.

kill -19
ctrl+Z → stop (suspend)

ps -ax ⇒ Check status
Stop

again

ctrl+Q → RUN (continue)

ps -ax ⇒ RUN

kill -18

③ **ctrl + ** :- It will generate SIGQUIT signal. It will abort & create core dump file.

⇒ Each signal have signal number & signal name, the signal name will start with SIG prefix. we can see the signal number & signal name by using kill -l command

⇒ These signal names & signal numbers are defined in a header file (signal.h).

⇒ All these signals are sent to your process, which are currently running in your terminal application.

④ Verify using gdb tool :-

① 1/0 ⇒ SIGFPE (Floating point exception) (core dumped)

② Access invalid memory ⇒ SIGSEGV (segmentation fault)

③ Call abort(); ⇒ SIGABRT

④ prgm. (verified)

⇒ Even, we can send a signal by using a command

```
*$ kill -signal name <pid>
        (or)
        number
```

⇒ kill is a process which will send the corresponding signal to the process mentioned by pid.

⇒ when CPU executes some invalid instructions or previlaged instructins, Kernal will send the corresponding signals to ~~that particular~~ terminate ~~that~~ process.

Eg:-

⇒ Invalid instructions :-

① ⇒ Dividing 1 with 0 ⇒ SIGFPE signal will be received

② ⇒ Instructions which are operating on invalid instructions.

⇒ For dereferencing the invalid address we will get segmentation fault.

⇒ when we try to over write stack

⇒ when we try to modify read only memory segments

the process will receive SIGSEGV signal

⇒ In these Scenario's, exceptional handlers will come into picture.

⊛ Exceptions Vs Interrupts :-

⇒ Whenever, an exception (or) interrupt occured, in both the cases, the process execution gets suspended.

⇒ Interrupts are generated by external hardware peripherals. These peripherals are connected to the CPU through system bus & IRQ line numbers

⇒ System bus is used only for data transfer, the IRQ lines are used to raise the interrupt.

⇒ whenever these peripherals wants to send the request to the CPU. they activates the IRQ line number.

⇒ when an interrupt is raised, CPU will suspend the current process execution & executes a special function called interrupt handler.

⇒ If devices wants to request the CPU for some work, they are going to raise interrupts. To handle these interrupts, in device drivers will have special fun^c called interrupt handler.

⊛ Exceptions :-

⇒ Exceptions are generated by CPU, when CPU executing invalid instructions or previlaged instructions.

⇒ when CPU identifies the exception, it suspends the current process execution & executes the exception handler. Exception handler will do 3 things:-

① Identifying the process from where the invalid instructions are executed.

② Get the pid of that process

③ Send particular signal to that process.

⇒ Kernal also sends signals to the process based on some Software conditions.

ⓐ Kernal uses signalling mechanism to bring parent process out of wait() system call blocking state. For this, kernal will send a signal to the parent process i.e, SIGCHLD

ⓑ When we use alarm() system call in a process, after reaching the timer value to zero, the process will receive SIGALRM signal

ⓒ In IPC mechanisms :-

⇒ No two processes will share the data directly, they have to use IPC mechanisms

⇒ When we use pipes IPC mechanism in a process, wrong handling of pipes may generate a signal called SIGPIPE.

(*) In how many ways a user can send a signal to process

② Through terminal by using command (kill)

③ Kill() system call

⇒ Kill command & kill() sys call requires ~~pid~~, signal name (or) signal number along with pid of the process.

⊛ Kill ( ) system call :-

⇒ We have to include signal.h header file.

⇒ Return type is int.

⇒ Func name is Kill

⇒ It receives 2 inputs

```
int kill ( pid_t pid , int signal no. ) ;
                          (or)
                       signal name
```

⇒ on success it will return 0 & on failure it will return -1.

⇒ If pid > 0 , the signal is sent to the process, whose process id is pid.

⇒ If pid == 0 , the signal is sent to all processes, whose process group id is equals to the process group id of the sender.

⇒ If pid < 0 , the signal is sent to all processes, whose process group id equals the absolute value of pid & for which the sender has permission to send the signal.

⇒ If pid == -1 , the signal is sent to all processes of the system, for which the sender has permission to send the signal.

⊛ Raise() system call :-

⇒ If a process wants to send a signal to the same process, or by itself, we have to use raise() system call.

```
int raise (int signal no) ;
```
⟺
```
int kill (get pid(), int signal no);
```

```
⇒ main()
{
  int ret ;
  pid_t pid;
  scanf ("%d", &pid);
  ret = kill (pid, 2);
}
```

pid no.

```
⇒ main()
{
  while(1)
  pf ("%d\n", get pid());
}
```

28 | 12 | 22

⇒ wait and wait pid system calls.

⇒ we have to include `#include <sys/wait.h>` header file.

⊛
```
pid_t wait (int *stat loc) ;
```

⊛
```
pid_t waitpid (pid_t pid, int *stat loc, int options);
```

⇒ Both system calls, on success, they will return pid of the terminated child process.

⇒ on failure, they will return -1.

⇒ If parent wants to wait (or) block for any one of its child termination, it has to use wait() system call.

⇒ If parent wants to block particular child termination, we have to use waitpid() system call.

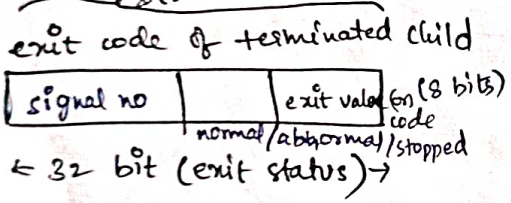
⇒ wait() & waitpid() behaves as a blocking calls.

⇒ In int *statloc variable, Kernal will update terminated child process exit status.

⇒ ~~bool~~ wait if exited (WIFEXITED)

if (WIFEXITED(stat)) ⇒ checking whether it is normal termination or not

(normal termination)



⇒ if it is normal termination of child, extracting the exit code of the child, (return value)

⇒ This is the 8 bit lower order value

⇒ PF ("%d\n", WEXITSTATUS(stat));

⇒ This will print the exit code of terminated child process

⇒ If the child process is terminated abnormally, that is verified by if (WIFSIGNALED(stat)) (Abnormal termination.)

⊛ we can extract the signal no. by using

⇒ PF ("%d\n", WTERMSIG(stat));

core dump
SIGABRT
SIGQUIT

⇒ If core dump file is generated, that can be verified by `PF("%d\n", WCOREDUMP(stat));`

⇒ If the child process is ~~suspended~~ (or) stopped currently, that can be verified by using

`if (WIFSTOPPED(stat))`

& the parent can ~~continue its execution by using~~ get the stopped signal no. by using

`PF("%d\n", WSTOPSIG(stat));`

⊛ Zombie State :-

⇒ Child process is already termated, but the parent is busy in performing other activity & is not ready to take the exit status of the child (Not blocking or wait or wait pid calls) so, in this case, the child process state changes to Zombie state. This process is known as Zombie process. This process memory segments are already removed from user space but the PCB still exist in kernal space until parent reads the exit status of the child. Once, the parent reads the exit status of the child, the PCB of child process is destroyed.

Wait pid() System call :-

⇒ In wait pid() system call, if pid == -1, waits for any child process, it is same as wait,

⇒ if pid > 0, waits for particular child

⇒ if pid == 0, waits for any child, whose process group id equals to the calling process.

⇒ If pid < -1, waits for any child, whose process group id equals to the absolute value of pid.

⇒ int options (third argument)

⇒ This argument can be zero (or) we can use predefined macros they can be combined by using bitwise OR (|) also.

⇒ (*) WCONTINUED

⇒ The status of any child specified by pid that has to be continued after being stopped. But, whose status has not yet been reported is returned.

(*) WNOHANG

⇒ The wait pid func will not block, if a child specified by pid is not immediately available. In this case, it will return zero (0).

(*) WUNTRACED

⇒ The status of any child specified by pid that has stopped & whose status has not been reported since it has been stopped is returned.

⇒ In this case, we have to use WIFSTOPPED to determine the return value corresponds to stopped child process.

⇒ wait pid provides 3 features. so ① wait pid func will blocks for a particular specific child.

② wait pid func provides a Non-blocking version of wait that depends upon 3rd argument.

③ wait pid func provides support for job control with the macros wait untraced & WAIT CONTINUED options.

⇒ WCONTINUED → ① • if child → running (or) execution → block
② • child → stopped → continue

⇒ WNOHANG → ③ • NO blocking } child is in execution
return zero } ④ if particular child is terminated, it gets the status of terminated child.

⇒ WUNTRACED → ⑤ If child execution stopped it will trace

(*) Exec family of calls :-

⇒ To replace a process image (newly created) with a specific process will use exec family of calls.

Ex :- Bash terminal application.

⇒ so, this process/application internally uses fork + exec family of calls. After reading the i/p from the terminal, it will create a new child process image by using fork() & replaces a child process with new process image after executing exec family of calls.

⇒ The pid, ppid remains same. Some of the members of PCB gets updated

⇒ The newly loaded process image execution will starts from the beginning.

⇒ The statements written after exec family of calls are not executed.

⇒ The environmental (path) variables are inherited from parent to child. They are stored above stack along with command line arguments.

LIBRARY_PATH =
environmental variable.

⇒ Default paths for commands (executable binary files) ⇒ /bin
/sbin
/usr/bin
/usr/sbin

All these paths are set by ENVIRONMENTAL VARIABLES.

⇒ Default paths for header files ⇒ /usr/include.

⇒ Libraries, shell commands & default paths for header files are set by ENVIRONMENTAL VARIABLES.

⇒ command line arguments ⇒ Inputs to the executable file while loading the process are known as CLA.

Eg:- gcc ① -o ② sample ③ sample.c ④

⇒ We are passing 4 command line arguments to gcc
⇒ Executable file name itself is the 1st CLA.

⇒ main ( ) ← 'n' no. of i/p's

{
=
}

⇒ main (void) ← no i/p's

{
=
}

⇒ main (int argc , char * argv[ ], char ** env)

argc ↓ no. of CLA

argv[ ] ↓ CLA base address

env ↓ ~~CLA for default library~~. ENVIRONMENTAL VARIABLES.

⇒ To compile .c file directly ~~from~~ home / different directory.

```
$ gcc /home/usr/desktop/New folder/sample.c
```

⇒
```
main()
{
  pid_t pid, cpid;
  int stat=0, i=0;
  pid = fork();
  if (pid<0)
  {
    PF("Failed to create child process \n");
    exit(1);
  }
  else if (pid>0)
  {
      // sleep(10);
    cpid = wait (&stat);
    PF ("%d\n", cpid);

  if (WIFEXITED(stat))
      PF (" %d\n", WEXITSTATUS (stat));

  if (WIFSIGNALED(stat))
      PF("%d\n", WTERMSIG(stat));

  if (WIFSTOPPED (stat))
  } exit(20);  PF("%d\n", WSTOPSIG (stat));
```

```
else
{
    PF("child : pid : %d\n", getpid());
    while (i<=10000)
    {
        PF("1");
        i++;
    }
    exit(10);
}
```

29/12/22

* For exec family of calls we have to use #include <unistd.h>

⇒ we have to include unistd.h header file.

⇒ int execl(const char * path, const char * arg 0, ..., (char *) 0);
(we have to mention executable file along with path.)
(command line arg to the executable file)

⇒ int execlp(const char * file, const char * arg 0, ..., (char *) 0);
(/bin, /usr/bin, /usr/local/bin)

⇒ int execv(const char * path, char * const argv[ ]);

⇒ int execvp(const char * file, char * const argv[ ]);

⇒ int execle(const char * path, const char * arg 0, ..., (char *)0,
char * const envp[ ]);

⇒ int execve (const char * path, char * const argv[ ],
char * const envp[ ]);

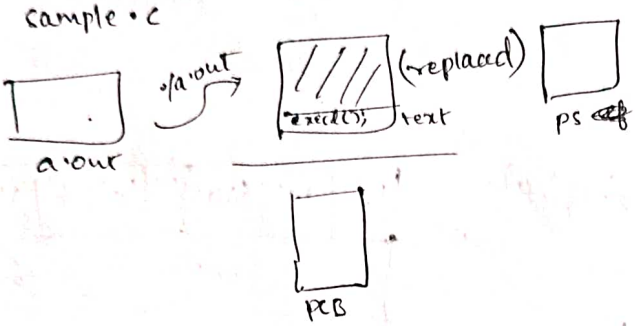
Sample.c

ps → /bin

① main()
{
PF(" process image is replaced \n");
execl("/bin/ps", "ps", "-ef", 0); // execlp("ps", "ps", "-ef", 0);
}

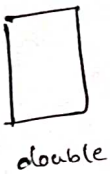
②

gcc sample.c



(*) Ex

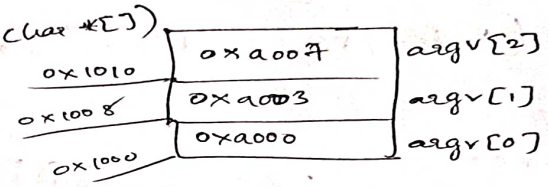
⇒ gcc -o double doublelinkedlist.c



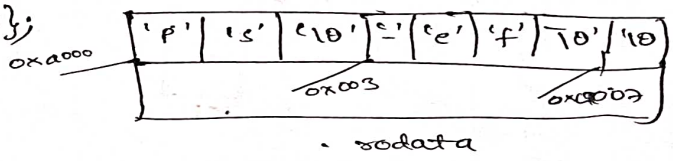
execl("/home/usr/Desktop/double", "double", 0);

③
main()
{
pid_t pid;
pid = fork();
if (pid<0)
{
PF(" Failed to create new child \n");
exit(1);
}
else if (pid==0)
{ usleep(10);
child → execlp("ps", "ps", "-ef", 0);
}
Else
{ sleep();
PF("parent : %d - %d \n", getpid(), pid);
parent → exit(0);
}
}

④
```
main()
{
    char * const argv[] = {"ps", "-ef", 0};
⇒ execvp(argv[0], argv);
}
```



⑤
```
main()
{
  char * const argv = {"/bin/ps", "ps", "-ef", 0};
⇒ execv(argv[0], &argv[1]);
}
```



31/12/22

⇒ The Members belongs to PCB which stores signals info are:-

① SDT → signal Disposition table
② signal mask
③ pending signal's info
④ Blocking signals info.
⑤ Delivered " "

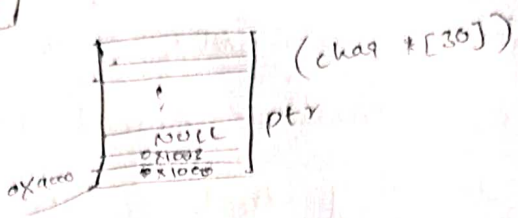
① <u>signal disposition table</u> :-

⇒ It maintains 64 signals behaviour

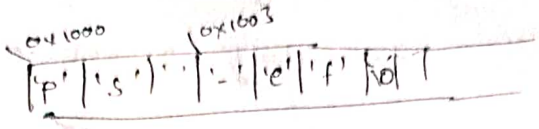
② <u>signal mask</u> maintains blocking signals information.

Assignment

```
char arr[100];
char *ptr[30];
gets(arr);
fork();
execvp(ptr[0], ptr);
```



↳ ps -ef



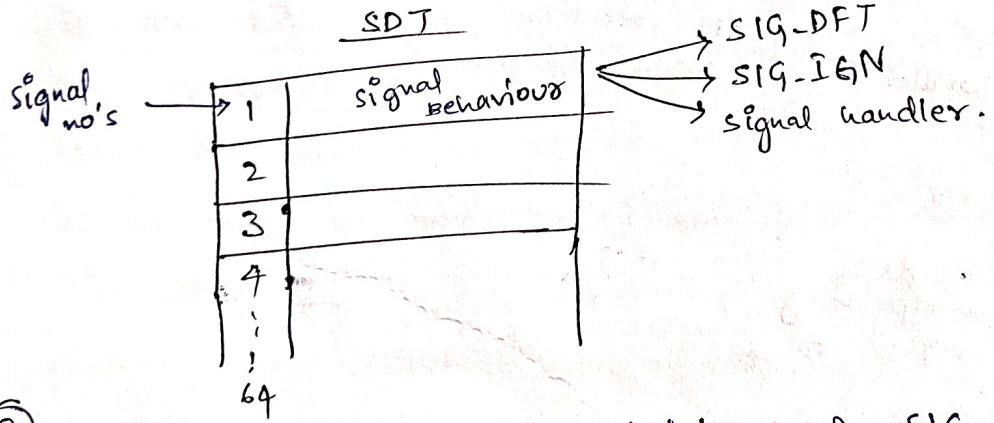
(*) How a signal is delivered to the process?

A⇒ Signal's disposition & signal's behaviour both are same.

⇒ On receipt of the signal (or) After receiving the signal what action should be performed is defined in SDT.

⇒ SDT contains total 64 entries.

⇒ signal numbers are indexed to the SDT. In second Entry we can find the behaviour of the signal. This can be SIG_DFT (signal default), SIG-IGN (signal ignore) (or) signal handler (defined by user).



(Q) (*) Find out the MACRO definitions of SIG_DFT & SIG_IGN

* Case (i) :- If Behaviour of the signal is SIG_DFT. It will execute the default behaviour of the signal. i.e., predefined action. Most of the signals default behaviour is terminating the process.

⇒ For a newly created process always contains default behaviour of the signals (SIG_DFT) in SDT.

Case (ii) :- If behaviour of the signal is SIG-IGN, that particular signal is ignored. That means, on receiving that particular signal, the process doesn't have any effect.

* Case (iii) :- If Behaviour is defined by the User by installing a signal handler func. it will executes the user defined behaviour on receiving that signal (itg but installed signal handler func is executed.

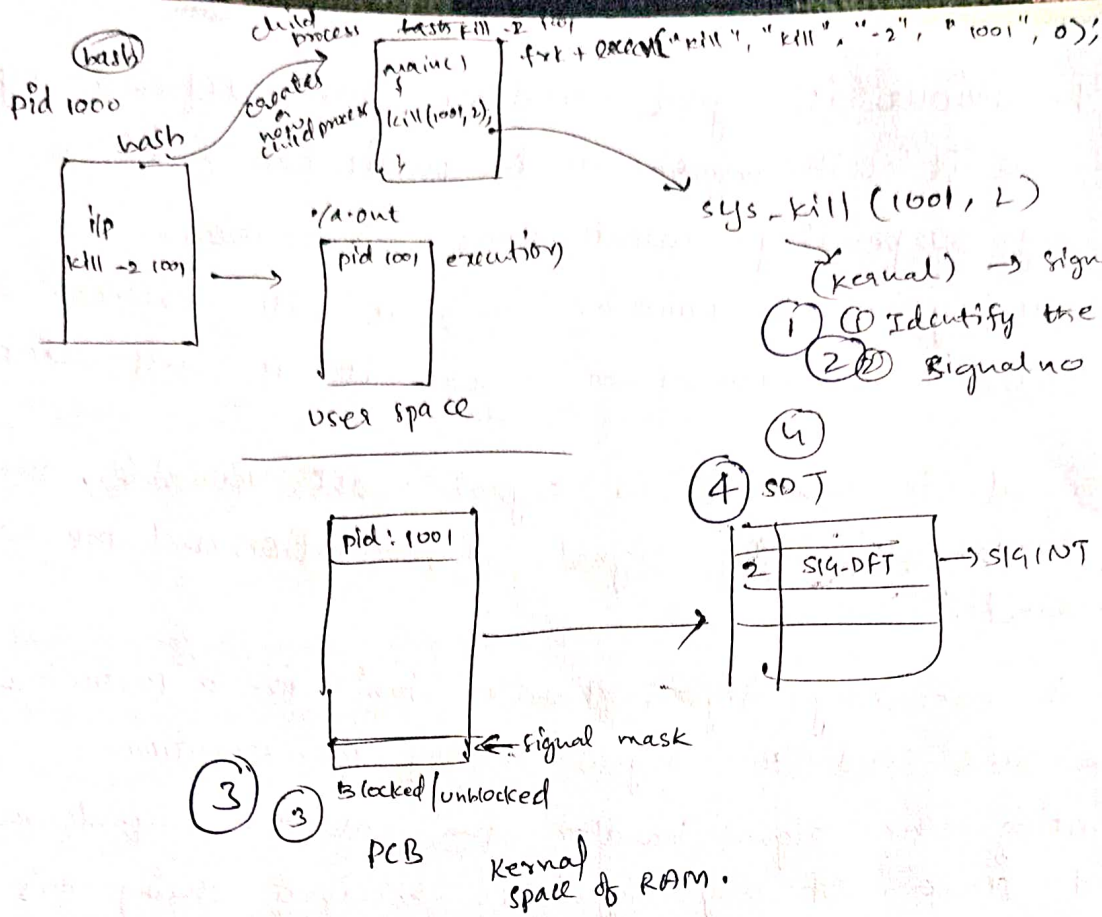
exception:-

invalid instructions execute
↓
CPU → exception is raised (CPU will suspend the current process execution)
↓
Kernal
↓
Invokes exception handler func (CPU execution mode will change)
↓
identify pid
&
sends corresponding signal to process

(2) Signal mask :-

⇒ In signal mask, it will store the blocking signals info of 64 signals.

⇒ In signal mask variable, each bit represents a particular signal.



⇒ when we use kill command under bash application, so it will create a new child process by using fork() and replaces the process image by kill by using exec family of calls. SO, this kill process internally invokes kill system call and this to which we are passing pid and signal no. this will invoke an equivalent Kernal fun^c sys_kill(). Now Kernal will identify the PCB based on pid (the current process execution is suspended.) After accessing the PCB, it will check the ~~whether it is blocked/unblocked~~ signal mask variable whether the particular signal is blocked/unblocked (particular corresponding bit is verified).

⇒ If the signal is unblocked in signal mask variable, now it will access signal disposition table & executes the behaviour defined in the corresponding entry based on signal number.

⇒ If the behaviour is SIG-DFT, it will terminate the process (most of the signal's behaviour)

⇒ If the behaviour is SIG-IGN, the signal is ignored & the process will continue the execution as it is.

⇒ If the behaviour is signal handler fun^c defined by the user, so it will invoke that particular fun^c & executes by suspending current process execution.
After executing that handler fun^c, it will resumes the current process execution from where it is suspended.

⇒ If the signal is blocked in signal mask variable, that is updated in to pending signals information and the suspended process is resumed.

⇒ when cpu is executing signal handler fun^c for a particular signal. we can't send the signal during the execution.
while executing the signal handler fun^c, all the signals are in blocked state. If any signal is received during this time period, that will updated in to pending signals.

⇒ The signals that are blocked in signal mask variable they are updated in Blocking signals information.

⇒ we can't ignore two signals i.e., SIGKILL & SIGSTOP.
⇒ we can't change the behaviour of these 2 signals.

⇒ Changing the Behaviour of a Signal:-

```
#include <signal.h>
typedef void (*sighandler_t) (int);
sighandler_t signal(int signo, sighandler_t handler);
```

signo → Signal no. (or) Signal name
↳ If we use signal name it will substitute its no.

handler → SIG_DFL (or) SIG_IGN (or) signal handler fun^c defined by user

⇒ On success, it will return the previous disposition of the signal i.e., zero. On failure, it will return -1.

* Alarm system call ( )

⇒ Alarm fun^c allows us to set a timer that will expire at a specified time in future.

⇒ when the timer expires, SIGALRM is generated. The default behaviour of the signal is to terminate the process.

⇒ We have to include #include <unistd.h>.

⇒ Return type is unsigned int.

⇒ fun^c name is alarm ( )

⇒ It will take only 1 i/p

unsigned int alarm (unsigned int sec);

⇒ Returns '0' or no. of seconds until previously set alarm.

⇒ The seconds value is no. of clock seconds in the future when the signal should be generated. When the time occurs or when the time expires, the SIGALRM signal is generated by the kernal.

- If when we call alarm( ), a previously registered alarm clock for the process has not yet expired, the no. of seconds left for that alarm is returned as the value of this fun^c. That previously registered alarm clock is replaced with new value.

Pause ( )

#include <unistd.h>

int pause (void);

This process fun^c will suspends the current process execution until it receives any one of 64 signals. After

receiving, it will execute the behaviour of the signal.

alarm(seconds)

⇒ If we want to set a timer from the current process, it should use alarm system call.

⇒ For every one sec, value is decremented by 1. After reaching to zero (0). Kernal will send SIGALRM signal. The default behaviour is to terminate the process.

⇒ When we set the alarm, the process execution is continued, it will not be suspended. That means alarm is not a blocking call

⇒ If we use alarm system call multiple times in a process, if the ~~set~~ time is not expired again if it executes alarm func, it will replace the old value with new value & the remaining time of old time is returned.

sleep(sec)

⇒ Sleep is a blocking call. When we use sleep(), the current process execution is suspended until the set time is expired.

⇒ Calling multiple sleeps will delay the execution of the process.

⇒ When we use sleep(), the PCB will moved from Running queue to waiting queue until the time gets expired.

⇒ When we change the behaviour of the signal by using signal func the behaviour is a function. That func will receive signal no, as the input and executes the behaviour i.e., invokes the func after receiving that particular signal.

⇒ We should not call exit function from the signal handler

⇒ When signal handler func is executed, the current process is suspended

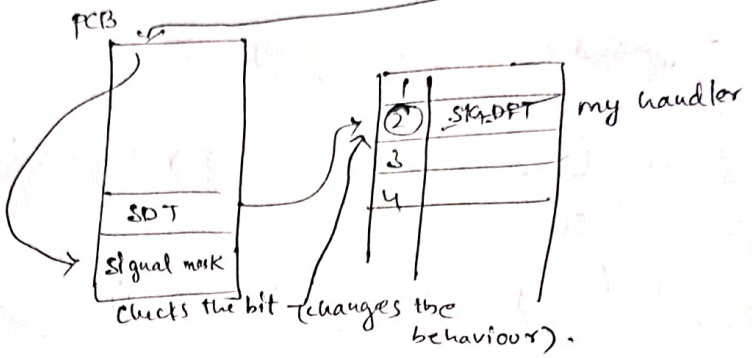
pid: 1000

```
main()
{
   int i=0;
   signal (SIGINT, (SIG_IGN) myhandler);
   while (i<=1000)
   {
      sleep(1);
      PF("%d\n", i);
      i++;
   }
}
```

kill (1000, 2);

equivalent sys call
sys-kill()
kernal identifies according to pid

Equivalent Kernal call
↓
sys-signal()




```
(2)
{
⇒ void myhandler (int signo)
   {
      PF("myhandler : (%d) signo\n", signo);
      sleep(5);
   }
   PF("myhandler is done\n");
```

⇒ when this process is under execution, from another terminal if we send the signal no 2 for this process ($ kill -2 1000) After receiving that particular signal, CPU will suspend the process execution & executes the myhandler func. Once execution is over, it will continue the process execution from where it is suspended.

Assignment

(2) Check the behaviour of the signal by writing the prgms. verify by SIG_DFT, SIG_IGN & myhandler func.

(4)
```
main()
{
   alarm(5);
   PF("process is in execution\n");
   while(1);
}
```
→ infinite loop
↑ after '5' sec it receives SIGALRM. Now the process is terminated.

⑤

```
main()
{
    signal(14, myhandler);
    alarm(5);
    pf("process is in execution\n");
    while(1);
}
```

```
void myhandler(int signo)
{
    pf("my handler : (%d) signo\n", signo);
    sleep(5);
    pf("my handler done\n");
}
```

when we send signal 14 by using another terminal

kill -14 <pid>

it will print

→ my handler : 14 signo

→ my handler done

instead of alarm - signal cuz we have changed the behaviour of alarm signal by using myhandler func

⑥

```
main()
{
    signal(14, myhandler);
    alarm(2);
    while(1)
    {
        pf("going to sleep for 2 sec\n");
        sleep(2);
        pf("woke up\n");
        alarm(2);
    }
}
```

The process will never gets terminated, even after sending alarm signal externally.

So, to stop it we need to send kill -2 <pid>.

02/1/23

⑦

```
main()
{
    alarm(2);
    while(1)
    {
        alarm(2);
        pause();
    }
}
```

02/01/23

* Signal Set Variables :- (we have to create) temporary

(User level executable file)

⇒ sigset_t

signal no

63 ---------- 2nd 1st 0th bit

0 ---------- 0 0 0

⇒ each signal is in unblocked state

⇒ If the bit is 1 (one), that particular signal is blocked, that means, On receiving that particular signal, it cannot access signal disposition table & cannot executes the behaviour of the signal.

(At kernal level, in PCB)

(*) signal mask (*)

⇒ It is the member of PCB which contains blocking signals information

⇒ It is total 64 bit variable

⇒ By default this variable is initialized with zero. All the bits are set to zero, that means each bit represents a signal

⇒ If bit is zero (0), that particular signal is not blocked. On receiving that particular signal it can access signal disposition table it can executes the behaviour.

⇒ when CPU is executing critical code section, that means a group of 'c' statements which are Perfoaming some important task it should not be effected (or) disturbed by receiving a set of signals from other process (or) Kernal. For this reason, we will block the signals, this can be done by updating into signal mask variable.

⇒ Steps to block the signals :-

① Declare a variable which is of type sigset_t.

② Add the blocking signals information to this variable by using signal set system calls.

③ Pass this information to the Kernal to modify signal mask variable. This can be done by using a system call SIGPROC MASK. When this is done (or) on succesful of this

Call, blocking signals info gets updated.

⇒ Set of functions :-

```
#include <signal.h>
```

① int sigemptyset (sigset_t *set);

② int sigFillset (sigset_t *set);

③ int sigaddset (sigset_t *set, int signo);
(To block)

④ int sigdelset (sigset_t *set, int signo);
(To unblock)

⇒ MACROS (or) fun' definitions.

⊛ Find out definitions

⇒ We cannot block 2 signals i.e., SIGKILL & SIGSTOP.

① sigempty set :-

⇒ The signal set variable is updated / initialized with zeroe's (0)

⇒ To add for update Blocked signals info.

② sigfill set :-

⇒ It initializes signal set variable's corresponding bits are updated with one (1)

To update add unblocked signals info

③ sigadd set :-

⇒ particular signal corresponding bit is set without modifying other bits

④ sigdel set :-

⇒ Particular signal corresponding bit is cleared without modifing other bits.

⇒ On success they will return 0.
⇒ On failure they will return -1.

④ sigprocmask :-

int sigprocmask (int how, const sigset_t *restrict set,
sigset_t *restrict oset);

⇒ on success, it will return 0.
⇒ on failure, it will return -1.

⇒ By using this system call, we not only we can block or unblock, even we can read the signal mask variable info.

① If oset is a non-NULL pointer, the current signal mask for the process is returned through oset.

② If the set is non-NULL pointer, the how argument indicates how the signal is modified. how is a predefined macro.

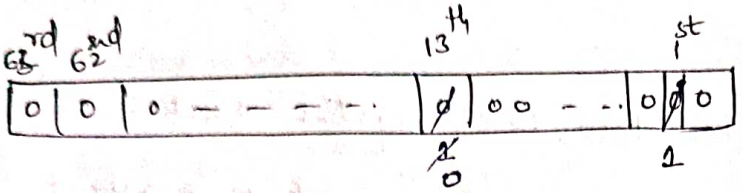
(i) SIG_BLOCK ⇒ The signal mask for the process is the union of its current signal mask & the signal set pointed to by set. i.e, set contains the additional signals that we want to block.

(ii) SIG_UNBLOCK ⇒ The new signal mask for the process is the intersection of the current signal mask & the complement of the signal set pointed to by set. That is the set contains the signal that we want to unblock.

(iii) SIG_SETMASK ⇒ The new signal mask for the process is replaced by the value of the signal set pointed to by set.

⇒ If set is a NULL pointer, the signal mask of the process is not changed & how is ignored.

sigset_t signal set;



→ User space of RAM
→ It is called signal set variable.

sigemptyset (&signal set);

unblocking/blocking signal info.
2,14 is added

sigaddset (&signalset, 2);
sigaddset (&signalset, 14);

sigprocmask (SIG_BLOCK, &signal set, NULL);

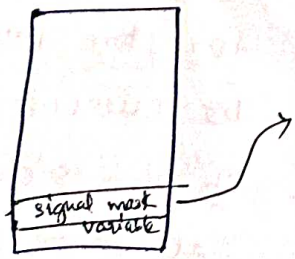
blocked (2,14) signals bits are modified by signal mask

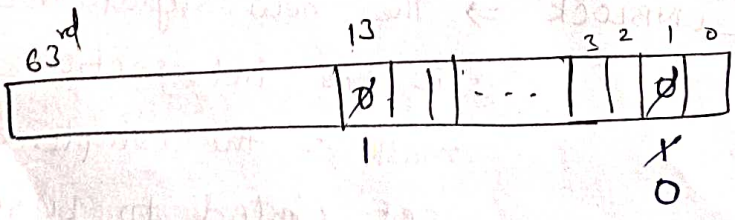
```
while(1)
{
sleep(1);
PF("%d", i);
i++;
}
```

remove 14 signal info from signalset variable ← sigdelset (&signal set, 14);

sigprocmask (SIG_UNBLOCK, &signalset, NULL);

→ 2nd signal is unblocked in signal mask variable but 14th signal is blocked (only verify 13th bit)





→ In kernal level it is known as signal mask variable

sigset_t osignalset,
sigemptyset (&osignalset);
sigprocmask (0, NULL, &osignalset);

⇒ cat /proc/<pid>/status (or)
↳ It shows the status of 64 bit

⇒ what is inline fun^c ?

⇒ what are its uses ? & How it is different from normal fun^c ?

⇒ Differences b/w inline fun^c & normal fun^c ?

⇒ How do you see Blocking signals info

① Ps -C$ (2) cat /proc/<pid>/status.

✱ Sigaction system call :-

```
# include <signal.h>

int sigaction (int sig no., struct sigaction *restrict act,
                              struct sigaction *restrict oact);
```

⇒ on success, it will return 0

⇒ on failure, it will return -1.

⇒ struct sigaction

```
{
    void (*sa_handler)(int);   // we can pass signal handler fun^c base add (or) SIG_DFT (or) SIG_IGN
    sigset_t  sa_mask;         // blocking signals info.
    int sa_flags;
    :
};
```

⇒ Even, we can change the behaviour of signal by using sigaction() as well as signal system call.

```
main()
{
   int i=0;
   struct sigaction act;
   act.sa_handler = myhandler;
   act.sa_flags=0;
   sigemptyset (&act.sa_mask);
   Sig addset (&act.sa_mask, 14);
   sig addset (&act.sa_mask, 2);
   sigprocmask (SIG_BLOCK, &act.sa_mask, 0);
   sigaction (2, &act, 0);
   while (i<=100)
   {
      sleep (1);
      PF ("%d\n", i);
      i++;
   }
sigdelset (&act.sa_mask, 14);
sigprocmask (SIG_UNBLOCK, &act.sa_mask, 0);
alarm(3);
Sleep (4);
 i=0;
while (i<=100)
{
   sleep (1);
   PF ("%d\n", i);
   i++;
}
```

$ps -cs
(or)
$cat /proc/<pid>/status.

→ it will change the behaviour of signal (2).

← send signals 2 & 14 (Blocked untay)

← send 2 & 14 (2 unblocked, 14 Blocked)

```
void myhandler (int sig no)
{
  PF ("signal-%d is received", signo);
}
```

03/01/23

⊛ File Management ⊛

⇒ From linux point of view, everything it is a file.

⇒ Every os will have a file system.

⇒ Linux will support ext2, ext3 & ext4 file system. These file system is mounted under root directory. So, it is also known as root file system. This file system gets loaded into ram during the boot up time.

⊛ Types of files :-

① textual files ⇒ understandable by users.

Eg:- .txt / .c / .cpp / .py / .java ----

⇒ These are user understandable (ASCII character format)

⇒ To read this contents we need editors. Eg:- geditor (or) Vim Editor.

⇒ In General, each file is applicable with three types of permissions i.e, owner, group & others.

⇒ The textual files doesn't have executable permission

	owner	group	other
Type of directory ←	rw-	rw-	r--
	1 1 0	1 0 0	1 0 0
0	6	4	4

(verify this by creating new file) by using ls -l. (or) File followed by file name.

⇒ The file permissions are represented in octal format.

② Object files :& executable files :-

⇒ These are special type of files. So, we cant use textual editors. we have to use special tools like objdump & readelf tools.

⇒ These files contains binary data which contains 0 & 1.

⇒ We have different types of executable files. so, we must use specific tools (or) players to execute them.
(or) open

Eg:-

⇒ Audio files can be played/opened only in Audio player.

⇒ Video files can be played/opened only in Video player.

⇒ Different types of executable files will have different file format.

Eg:- .mp3, .exe, .o, a.out, .mp4, .jpg, .PNG. — —

⇒ Executable files will have executable permissions.

	owner	group	others
	rwx	r-x	r-x
	1 1 1	1 0 1	1 0 1
o	7	5	5

⇒ To open .KO (Kernel object files) we can use objdump & readelf tool.

③ Directories :-

⇒ It is also a ~~folder~~ one type of file, where we can store multiple files & sub directories.

⇒ Directories will have executable permissions. Here, the type of file is represented with d.

	owner	group	others
d	rwx	r-x	r-x

⇒ Even we have special files

① device files (or) device nodes :-

⇒ These files are used by the Kernal device drivers for communication from user space to Kernal space & vice-versa.

⇒ These are present in /dev directory & the type of file is also represented

c → character device files

b → block " "

l → symbolic link file

⇒ Even they have permissions (verify).

(*) Find the differences b/w soft link & Hard link?

⇒ These files can be accessed by root user only.

⇒ As a normal user we can't open, modify (or) delete these files. So, we require super user permissions.

⇒

(2) Sockets :-

⇒ Even sockets are also one type of special files.

⇒ They are used (only) during N/w communication

(3) IPC objects :-

⇒ These are used during the inter process communication.

Eg:- pipes, named pipes & message Queues.

⇒ To check the type of file.

⇒ ls -l

⇒ stat <File name>

⇒ File <File name>

⇒ Any type of file (whether it may be), it has file attributes

File name, type of file, size, owner name, owner id, group name, group id, others other id

Permissions, time stamps, location / path & so-on. all thes

comes under file attributes.

⇒ These file attributes are stored in a predefined kernal object by the Kernal i.e., inode object (struct inode).

(4) Find out the header file in which the struct inode is defined

⇒ /usr/include/linux/fs.h

⇒ The inode object info some of its members we can access from user space by using stat command or ls -l.

~~File~~

* Differences between Hard link & soft link

Comparison Parameters	Hard link	Soft link
① Inode number *	Files that are hard linked take the same inode number.	Files that are soft linked takes a different inode number.
② Directories	Hard links are not allowed for directories	soft links can be used for linking directories
③ File system	It cannot be used across file systems	It can be used across file system.
④ Data	Data present in the original file will still be available in the hard links	soft links only point to the file name, it does not retain data of the file.
⑤ Speed	Hard links are comparatively faster	soft links are comparatively slower
⑥ original's file deletion.	If the original file is removed, the link will still work as it accesses the data the original was having access to	If the original file is removed, the link will not work as it doesn't access the original file's data.

Hard links

⇒ It is a copy of the original file that serves as a pointer to the same file, allowing it to be accessed even if the original file is deleted or reallocated.

⇒ It has a similar inode no. to the target file.

⇒ It is not allowed the relative path.

⇒ It cannot be established outside the file system.

⇒ It is faster

⇒ The "ln" command is used to make a hard link in linux

⇒ It has an additional name for the original file that references to the target file through inode

⇒ It may only link to a file

⇒ It remains valid even if the target file is deleted.

Soft links

⇒ It is a short pointer file that links a filename to a path name. It's nothing more than a shortcut to the original file, much like the windows OS's shortcut option

⇒ It has a different inode number.

⇒ It allows both relative & absolute paths.

⇒ It may be established in the file system.

⇒ It is slower.

⇒ The "ln -s" command is used to make a soft link in linux.

⇒ It is different from the original file & is an alternati for it, but it does not use inode.

⇒ It may link both to a directory or a file.

⇒ It becomes invalid when the originating file is deleted.

⇒ The Kernal objects / ~~Kernel~~ that are associated with file System. These objects are associated with virtual file system. Mainly, we will discuss about four primary objects

① super block object which represents a specific mounted file system

② inode object which represents a specific file exist in hard disk.

③ The D-entry object which represents a directory entry which is a single component of a path.

④ File object which represents an open file associated with a process.

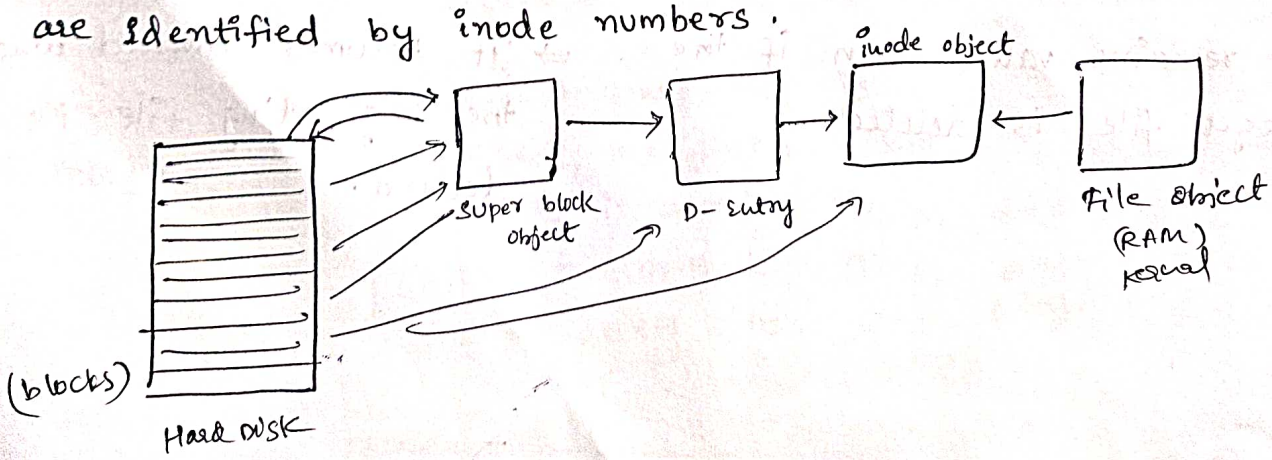
⇒ On these objects also, we perform some operations.

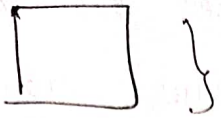
⇒ when we open, read, write, close & remove files (or) directories On these corresponding kernal objects will perform operations.

⇒ These operations are executed under Kernal.

⇒ The inode object exist in your hard disk which represents a particular file from Kernal point of view.

⇒ Each file is associated with an inode object. These objects are identified by inode numbers.





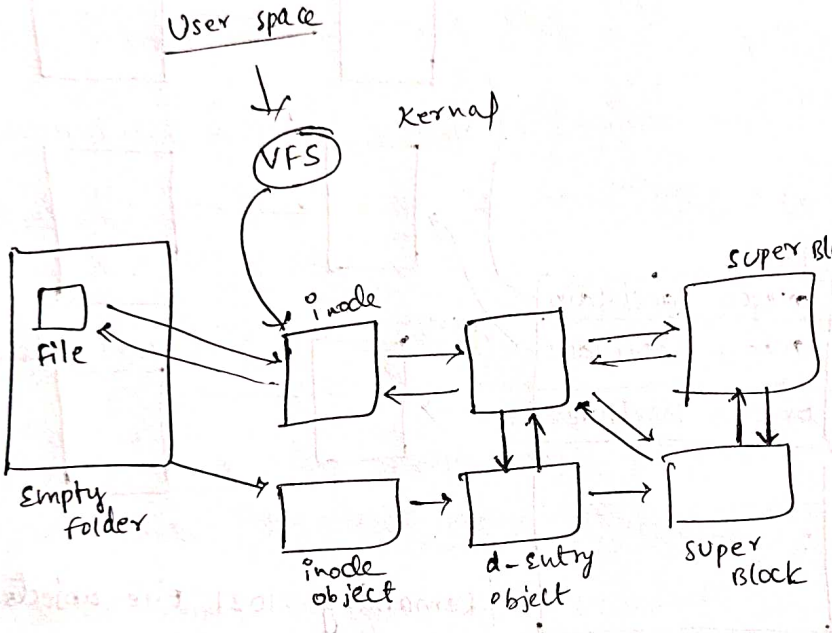
Sample.c

File attributes

- → inode no.
- → Name
- → size
- → permissions
- → path
- → group lowner
- → Time stamps

inode object → it is the member of D-entry & also it is an separate object.

ide (Hard disk)



⇒ creates when size of file gets increased.

⇒ super blocks may vary as it depends on the size of file.

04/1/23

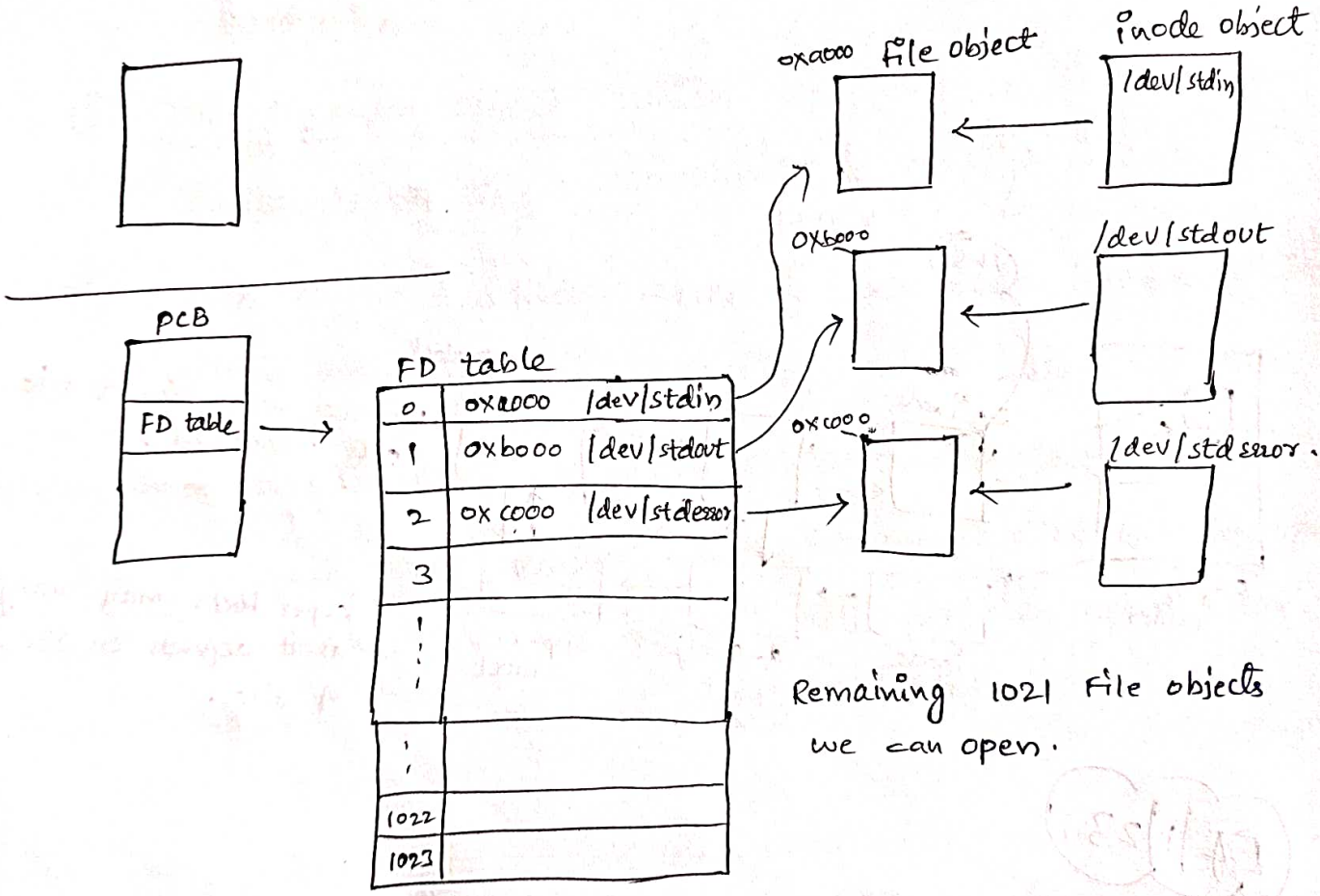
File Descriptor table FD

⇒ This is the member of PCB which maintains the no. of file opened.

⇒ FD table is also pre-defined kernal object which ~~has~~ can maintain maximum 1024 entries only, that means from a particular process we can open maximum of 1024 files. so, In these Entries it stores the opened

file objects base address. The FD values are the index of these FD tables. The FD values starts from 0 & ends at 1023. As you close the file, the entry is deleted from FD table.

⇒ In this FD table, First 3 entries are updated by default with stdin, stdout, stderror file objects for every process. (device files)



⇒ System calls :-

- open ()
- close ()
- read ()
- write ()
- ioctl ()
- ⋮

} Basic I/O calls (or) Universal I/O calls.

(wz we can open any type of file by using this system calls). & can perform operations

⇒ These file operation system calls are applicable on normal textual files, device files, executable files, IPC objects & sockets.

⇒ By using file management we can perform below operations:

① we can create a new file/folder.

② we can open a file which is already exist.

③ perform read & write operations to modify the data ④ Once modification is done we can close the file.

⑤ we can remove the file

⇒ Even directory/folder is also one type of file.

⇒ Even we can read & modify the some of file attributes

⊛ open() system call :-

⇒ To open a file from the process which is already exist we have to use open() system call.

⇒ Even open() sys call is used to create a new file.

⇒ It can be seen with two type of arguments

open (2 arguments) ⇒ (To open)

open (3 arguments) ⇒ (To create)

⇒
```
#include <fcntl.h>
int open(const char * path, int oflags, ....., mode_t mode);
```

- const char * path → file name along with the path as a string constant.
- int oflags → To open an existing file: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_TRUNC ---; O_CREATE → It is used only to create a new file
- mode_t mode → 0664 (octal), 0775, 0775

⇒ This call on success it will return FD value on failure, it will retun -1.

⇒ open() system call will fails, when the file doesn't exist in the specified path. In this case, it will return -1.

⇒ This open() system call, internally invokes Sys_open kernal call.

⇒ when we want to perfoam read (or) write operation, first we need to open the file

⇒ If we want to perform read only operations, open the file under read only mode. (O_RDONLY)

⇒ If we want to perform write opeeation (or) if we want to modify the content, open the file under write only mode (O_WRONLY)

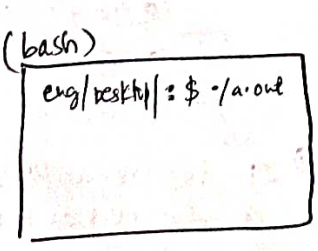
⇒ If we want to perfoam both read & write opeeations, open the file under read & write mode (O_RDWR)

```
int fd;
fd = open("sample.txt", O_RDONLY);
```

(3) It will return after an entry gets updated.

↓ it searches for the file in current working directory (By the help of ENV. VARIABLE) cuz it contains paths.

/home/eng/Desktop/sample.txt.



size 4 bytes

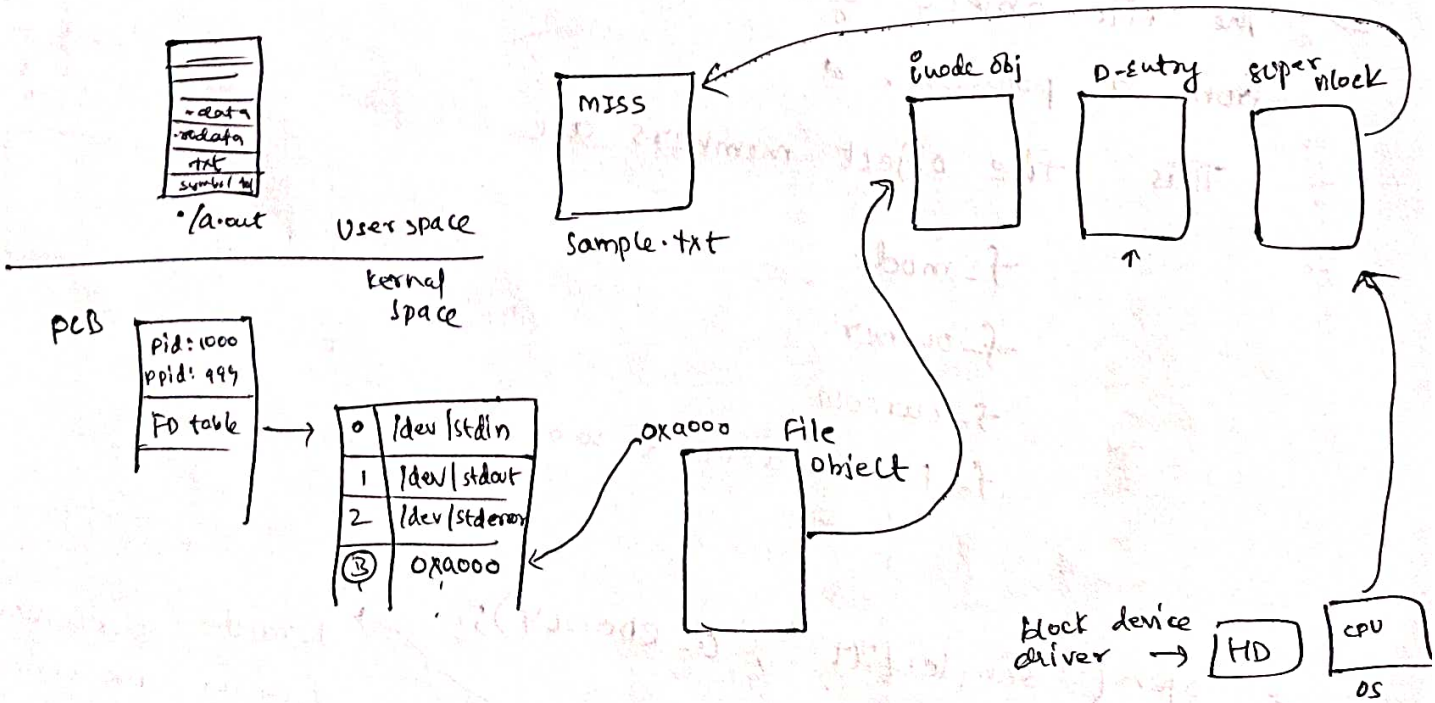
⇒ open() system call internally invokes sys-open kernall call. This will search for the file in the specified path

⇒ It search for the file, if it exist it will create a new file object (in kernal space of RAM) & links the file object with inode object. This file object base address gets updated into FD table. This index is return by the open() system call.

⇒ The opened file from a process can be accessed through FD value.

⇒ In the next sequent sub system calls like read(), write(), close(), fcntl(), ioctl() and so-on. The return FD is used. (open system call). Once the link is created, we can read the data, write the data & other operations can be performed on the file.

⇒ when we close the file, the complete link will gets destroyed. The file object & entry in FD table gets destroyed. But inode object, D-entry object & super block will remains.



Read system call :-

⇒ If we want to read the data from an open file, we need to use read() system call.

⇒ #include <unistd.h>

⇒ size_t read (int fd, void * buf, size_t nbytes);

- fd returned by open() systemcall
- empty buffer base address
- buffer size

⇒ On success it will return no. of bytes read from the file. when the file reaches to the end it returns zero.

⇒ On failure, it will return -1.

⇒ This function (or) call will fails when the file is opened under write only mode.

File object

⇒ The file object gets created when the file is ppened from a process.

This file object members are

- f_mode
- f_owner
- f_cursor
- f_private data & so on.

⇒ open ("sample.txt", O_RDONLY);

- f_mode = O_RDONLY
- f_owner = member name
- f_cursor = 0 (starting position)
- f_private data = — (used by device driver)

⇒ In a file we don't store any NULL character.

⇒ At the end of each line, it stores a New line character (\n)

⇒ read() system call fails, if we try to read the data without opening a file.

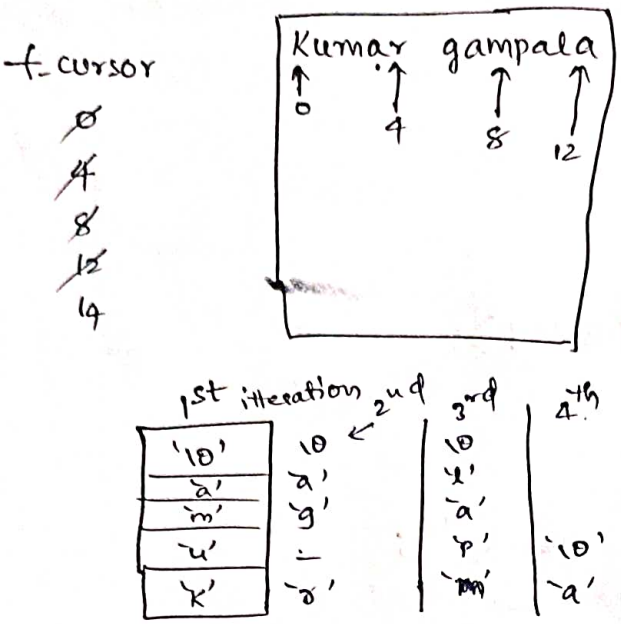
⇒ For every read & write operation f_cursor gets updated.

```
buf[ret] = '\0';
PF("%s\n", buf);
ret = read(fd, buf, 99);        // Kumar Gampala
PF("%d\n", ret);                // 0
close(fd);
}
```

(curr already reached to EOF)

⇒ If buffer size is less than total size of the file.

```
⇒ main()
{
    int fd, ret;
    char buf[5];
    fd = open("sample.txt", O_RDONLY);   (3)
    if(fd<0)
    {
        PF("failed to open the file\n");
        exit(1);
    }
    while(ret = read(fd, buf, 4))
    {
        if(ret<0)
        {
            printf("failed to read\n");
            exit(2);
        }
        buf[ret]='\0';
        PF("%s\n", buf);
    }
```



o/p

Kuma
r ga
mpal
a

(*) Assignment

⇒ Implement your own cat command.

* close() system call

⇒ we need to include #include <unistd.h>.

⇒ int close (int fd);

⇒ This fun^c/call on success it will return '0'
on failure it will return -1.

⇒ This call is used to close the open file from a process

⇒ This call will take the input fd returned by open() system call

⇒ If the reference count is not zero, the file will not closed. (This is only for drivers)

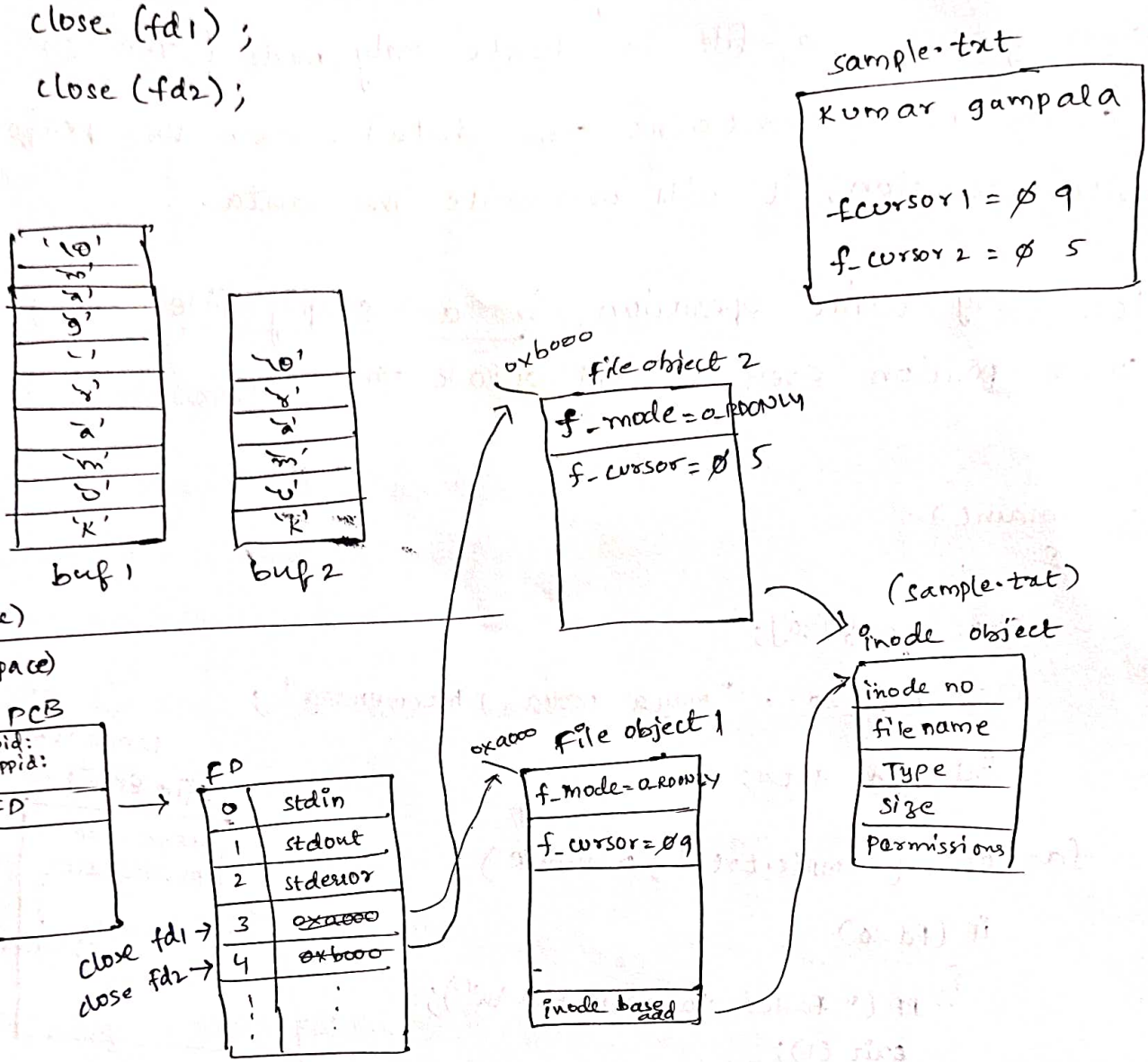
⇒ This reference count is verified in close driver routine. (Fun^c/method)

⇒ when this call is successful, this will destroy the link created b/w file & corresponding objects. The file object is destroyed when the reference count is zero.

⇒ main()
{

```
int fd1, fd2, ret1, ret2;
char buf1[10], buf2[10];
fd1 = open ("sample.txt", O_RDONLY);
if (fd1<0)
  {
    PF ("Failed to open \n");
    exit(1);
  }
ret1 = read (fd1, buf1, 9);
if (ret1<0)
  {
    PF ("Failed to read \n");
    exit(2);
  }
buf1[ret1]='\0';
printf ("%s\n", buf1);        // Kumar gam
fd2 = open ("sample.txt", O_RDONLY);
if (fd2<0)
  {
    PF ("failed to open\n");
    exit(3);
  }
ret2 = read (fd2, buf2, 5);
if (ret2<0)
  {
    PF ("failed to read\n");
    exit (4);
  }
buf2[ret2]='\0';
PF ("%s\n", buf2);            // Kumar
```

```
close (fd1);
close (fd2);
}
```



⇒ write ( ) system call :-

⇒ #include <unistd.h>

⇒ ssize_t write (int fd, ~~const~~ void *buf, int ~~nbytes~~);

fd returned by open system call

buffer address (or) source address

length (length depends on 2nd argument)

lengths of the data.

⇒ This Func (or) call will return no. of bytes written on success

⇒ On failure, it will return -1.

⇒ When you open the file in read only mode & you perform write operation, it will fail.

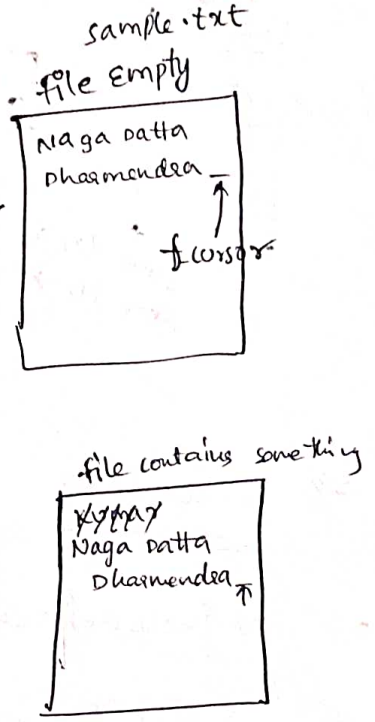
⇒ for every write operation, the cursor position gets updated

⇒ ~~When~~ If ~~the file~~

⇒ when you open a file in write only mode (the file already exists which contains some data). when we perform write operation, it will over write the data.

⇒ For every write operation, in an empty file along with the cursor position even it will update the size.

⇒
```
main()
{
  char rbuf[30];
  char buf[30] = "Naga Datta Dharmendra";
  int fd, ret;
fd = open("sample.txt", O_RDWR);
  if (fd<0)
  {
    Pf("failed to open file \n");
    exit(1);
  }
ret = write(fd, buf, strlen(buf));   // 21
  // sleep(30);   → verify size
ret = read(fd, rbuf, 30);   // 0
  rbuf[ret] = '\0';
  Pf("%s \n", rbuf);
  close(fd);
}
```



Eg:

file size	cursor	write mode
20	0	10
remains same → 20	10	10
changes to new size (30)	20	10
	30	

05/01/23

⇒ lseek() system call :- depends based on 3rd argument; pre-defined macro.

⇒ off_t lseek(int fd, off_t offset, int whence); (fd returned by open())

⇒ #include <unistd.h>.

⇒ If we want to change the cursor position from current position to either forward or backward or beginning or end of the file, we need to use lseek().

⇒ This system call will modify the member of file object i.e., f_cursor.

⇒ on success, this system call will return new offset.
on failure, it will return -1.

⇒ whence are SEEK_SET
SEEK_CUR
SEEK_END

⇒ when we use SEEK_SET ⇒ the cursor will change from beginning to the offset set value.

⇒ SEEK_CUR ⇒ The current position the offset value will be added from forward or backward.

⇒ SEEK_END ⇒ The cursor position will change from END to the set value either forward or backward.

⇒ If it is backward, then subtract the offset value from end of the position value.

⇒ If it is forward, then add the offset value from end of the position. In this case, size of file will increases.

Sample.txt

Naga Datta Dharmendra
....↑

```
fd = open("sample.txt", O_RDONLY);

lseek(fd, 5, SEEK_SET);      ---> fcursor ↓ 5
// lseek(fd, -5, SEEK_CUR);  ---> 0
lseek(fd, +5, SEEK_CUR);     ---> 5 10
lseek(fd, +5, SEEK_END);     ---> Add 5 spaces at the end.
```

⇒ Create() system call

⇒ `#include <fcntl.h>`

⇒ `int create(const char *path, mode_t mode);`

⇓ (equivalent create call)

`open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);`

- path ↙ File name along with path
- O_WRONLY | O_CREAT | O_TRUNC ↓ we need to provide permissions
- mode ↓ 0665 (Etc) 0775

⇒ O_TRUNC is used in open() system call as a second (2^{nd}) argument with ~~OR operation~~ other macros by using OR operator.

⇒ O_TRUNC it will set the file size to zero.

⇒ This system call will invoke an equivalent kernal call & it creates a new file (if it doesn't exist)

so, when a new file is creates that means an inode object is created & now opens the file in write only mode. and the file object is created & entry is updated in FD table & returns the index of the FD table.

```
⇒ main()
  {
    int fd, ret, nfd;
    char buf[20];
(3)
fd= open("sample.txt", O_RDWR);
(5)
  ret = read(fd, buf, 5);

      buf[ret] = '\0';

    PF("%s", buf);
? (4)
  nfd = create("sample.txt", 0664);
? (2)
  ret = write(nfd, "Hi", strlen("Hi"));
? (0)
  ret = read(fd, buf, 5);
? (5)
  ret = write(fd, "Hello", 5);
  }
```

Sample.txt

Naga Balla
Dharmendra

⇓ replaced by

hi---hello

⇒ O_APPEND() system call :-

⇒ This is also used in open() system call as 2nd argument

⇒ This is used when we want to write the data from the End of the file.

⇒ when we use this call, the cursor position will changes to the End of the file.

⊛ Assignment :-

⇒ Implement own my copy command.

Standard input & output file descriptors

Stdin → 0
Stdout → 1
Stderror → 2

⇒ These three file descriptors are referred to three device files /dev/stdin
/dev/stdout
/dev/stderror

⇒ By default ~~these 3 files~~ for every process, these 3 files are automatically open, we can see these 3 entries in FD table

⊛ stdin (file descriptor) is used to read the data from the keyboard.

⊛ stdout & stderror is used to display the o/p on the terminal screen

⇒ stdin is used with read() system call

⇒ stdout is used with write() system call.

⇒ when we use standard c library input functions, they internally invokes read() system call with standard input (stdin) file descriptor.

⇒ when we use standard c library output functions, they internally invokes write() system call with stdout file descriptor.

⇒ read() system call with normal textual files opens fds it behaves as a non-blocking call.

⇒ read system call with stdin file descriptor (0) behaves as a blocking call.

NOTE :-

⇒ Read system call on IPC objects & sockets, it behaves as a blocking call (until user enter's the data).

⇒ Read with 0 behaves as a blocking call until user enters the no. of characters specified in 3rd argument & also enter key.

read (0,);

⇒
```
main()
{
    char buf[20];
    int fd, ret;
    fd = open("sample.txt", O_RDWR);
    if (fd<0)
    {
        PF("Failed to open file\n");
        exit(1);
    }
    ret = read(fd, buf, 20);
    if (ret<0)
    {
        PF("Failed to read\n");
        exit(2);
    }
    ret = write(1, buf, ret);
    ret = write(3, buf, ret);
    close(3);
}
```

sample.txt

```
kumar gampala
kumar gampala
```

Terminal.

```
$ ./a.out
kumar gampala
```

⇒ The standard c library input & output functions internally uses the buffer. Whereas, these system calls doesn't use any buffer. Directly the data is read or displayed.

⇒ c libraries are in user mode

⇒ system calls are used in Kernal mode.

06/1/23

⇒ Dup() system call : & ~~DUP2()~~

⇒ whenever we want to duplicate the file in the FD table, we have to use dup() (or) dup2().

⇒ we have to include <unistd.h> header file.

⇒ int dup (int fd);

⇒ we have to use this call after opening a file.

⇒ on success it will retuen new file descriptor. (dup file) & on Failure it will return -1.

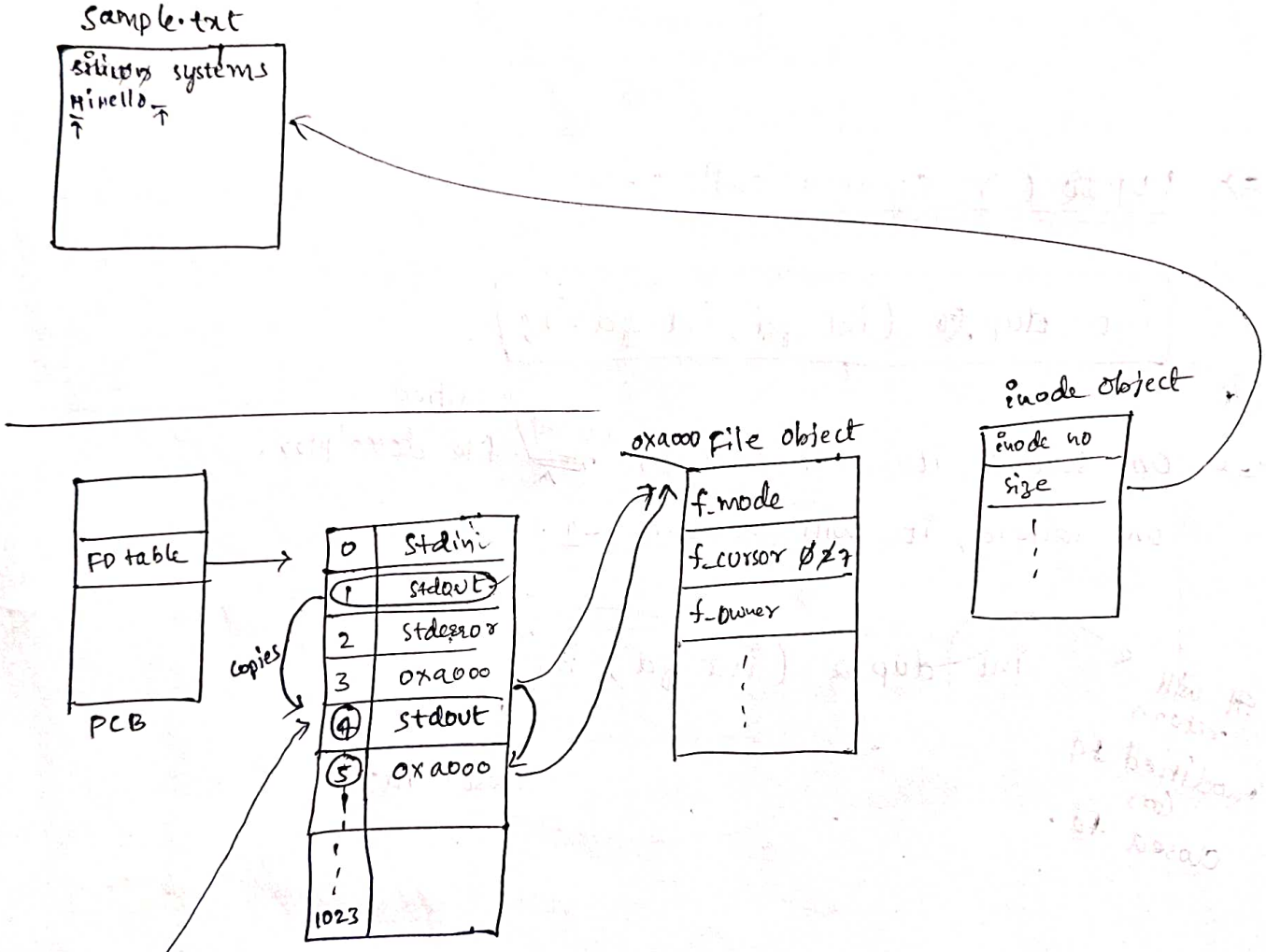
⇒ This dup() will not create a separate file object, jst it will duplicates the entry present in corresponding fd table in freely available next entry.

⇒
```
main()
{
    int fd, ret, fd1, fd2;
    fd = open("sample.txt", O_RDWR);   (3)
    fd1 = dup(1);                      (4)
    printf("%d\n", fd1);
    write(1, "silicon systems", strlen("silicon systems"));
```

```
write (fd1, "silicon systems", strlen ("silicon systems"));
```

⑤ ④

```
fd2 = dup (fd);

PF ("%d\n", fd2);

write (fd, "Hi", strlen ("Hi"));

write (fd2, "Hello", strlen ("Hello"));
}
```



verify

welcome
silicon

```
main()
{
    int fd;
(?) ← close (1);
    Printf ("welcome\n"); → ?
? ←
    fd = open ("sample.txt", O_RDWR);
    PF("fd = %d\n", fd);
    PF ("silicon \n"); → ?
}
```

→ It will not print on terminal screen as we closed 1st entry

⇒
```
main()
{
    fd = open ("sample.txt", O_RDWR);
    close(1);
    dup(fd);
    printf("systems\n");
}
```

(?) (?) (?) (?)

⇒ DUP() system call :-

```
int dup (int fd, int fd2);
```

→ on success, it will return closed/modified file descriptor.

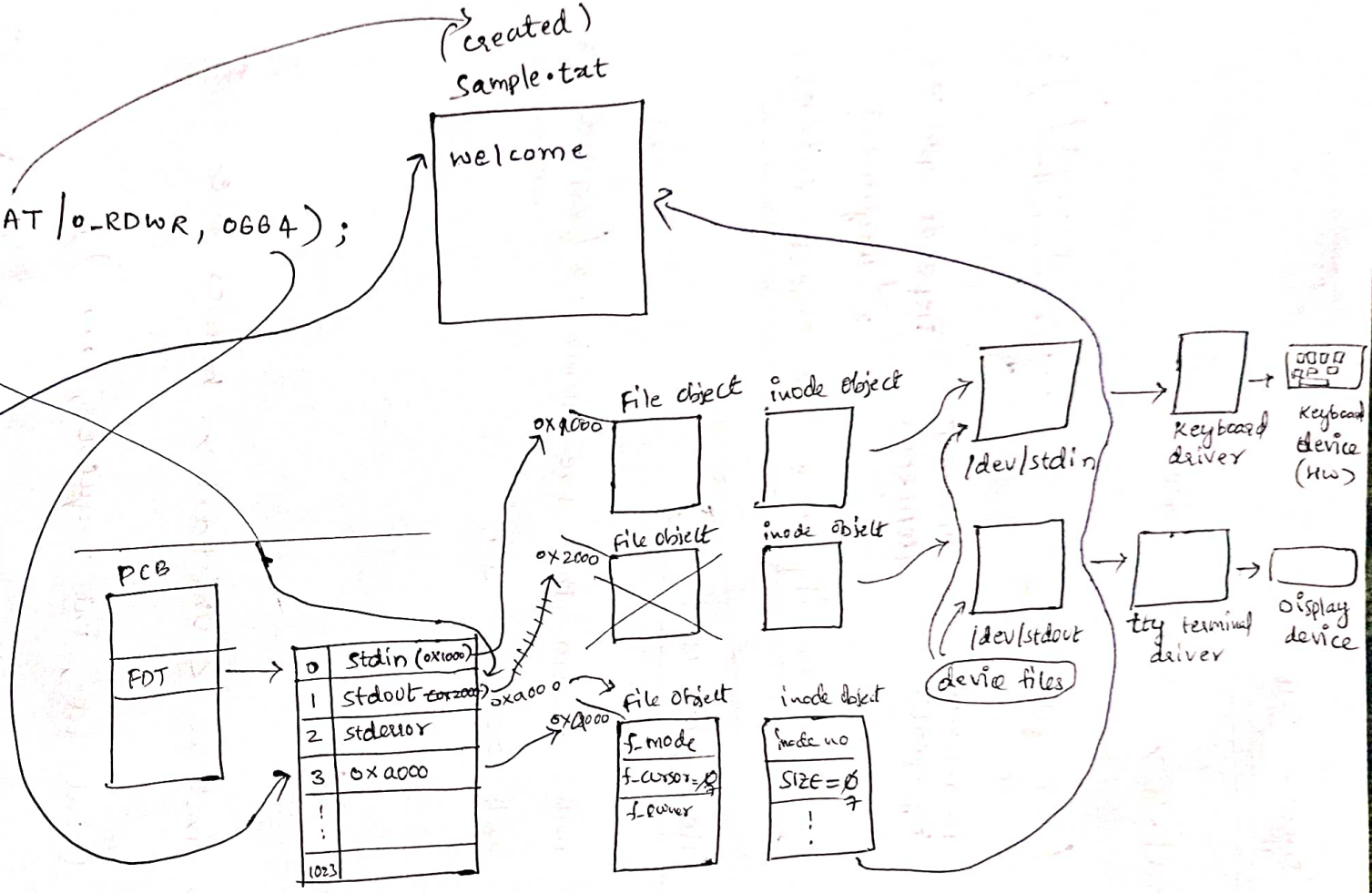
on failure, it will return -1.

int dup (int fd, int fd2);

- fd: It will return modified fd (or) closed fd.
- fd2: copies → close. fd2

```
main()
{
   int fd,ret;

   fd = open("sample.txt", O_CREAT|O_RDWR, 0664);

   ret = dup2(fd,1);

   Printf("welcome\n");
}
```

Internally invokes write(1,); System call



⇒ fcntl ( ) system call :-

⇒ #include <fcntl.h> header file

⇒ int fcntl (int fd, int cmnd, ... /*int arg*/);

- fd: ↑ fd returned by open()
- cmnd: ↑ command (MACROS)
- ... /*int arg*/: ↑ Depends upon the 2nd argument

⇒ This fun^c is used to modify the file object entries.

fcntl ⇒ file object control.

⇒ on success, it will return the +ve numbers (depends on the command)
on failure, it will return -1.

⇒ ioctl ( ) system call

⇒ we need to include <unistd.h> header file

⇒ int ioctl (int fd, int request, ...);

⇒ This fun^c is used to perform input output operations. This will be understand in device drivers clearly.

⇒ If we want to control particular parameters of device i/p & o/p and its operations from user space can be done by using ioctl.

⇒ on success, it will return +ve value.
on failure, it will return -1.

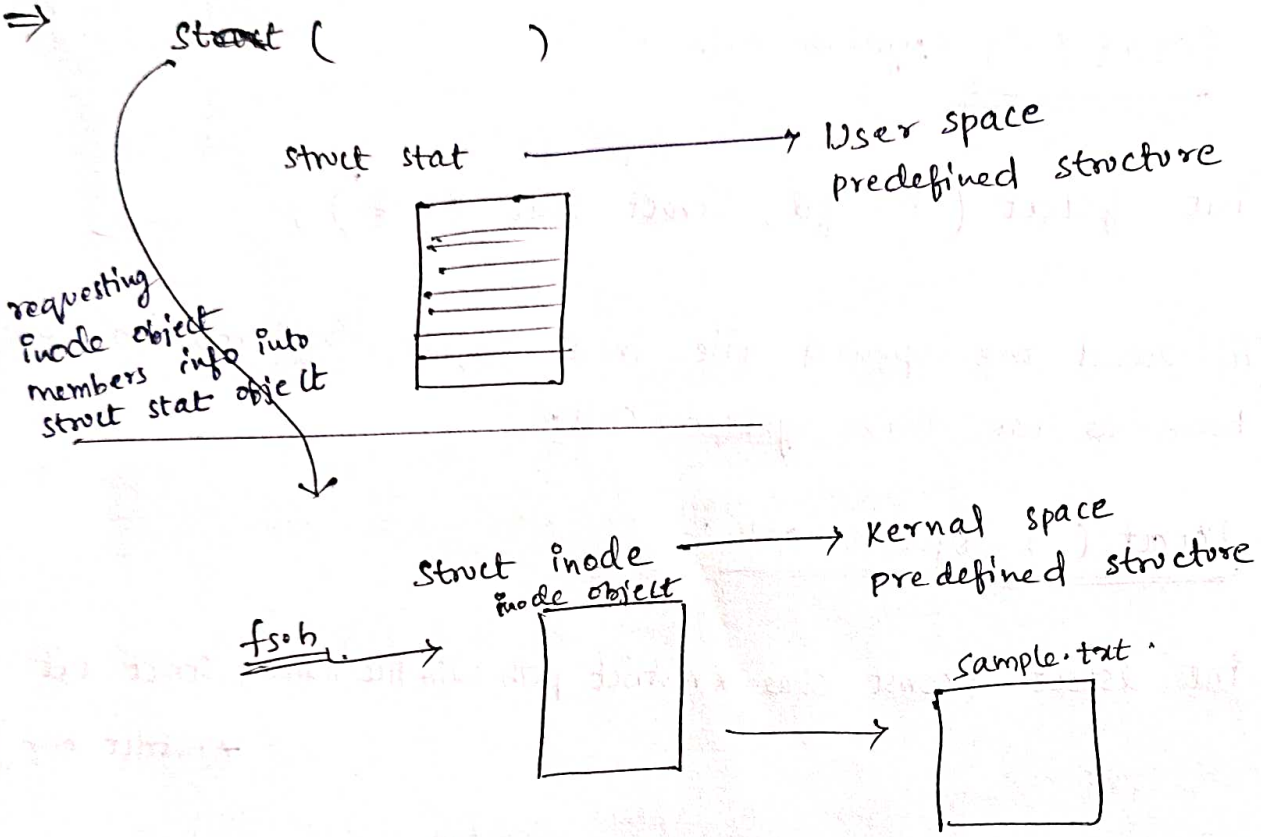
⇒ Brightness, contrast, display resoultion, size etc can be control by using ioctl system call from user space by making request to driver.

⇒ If we want to read some of the inode object members information, we can use stat, fstat, lstat (for symbolic link)

(for unopened file) (for opened file)

⇒ stat command internally invokes stat system call.

⇒



⇒ File object is created by open() system call.

⇒ File object can be accessed by fcntl().

⇒ stat() system call

⇒ #include <sys/stat.h> we have to include

⇒ int stat (const char * restrict path File name, struct stat *restrict buf);

↑ file name along with the path

↑ (Empty buffer) struct stat's variable's Base address

⇒ On success, it will return '0'.

⇒ On failure, it will return -1.

Assignment

(*) Implement your own stat command.

(1) understand struct stat structure members. check the definition in corresponding header file

(2) Based on members & their type use the corresponding format specifiers.

(*) fstat( ) system call :-

⇒ int fstat (int fd, struct stat *buf);

⇒ To read the opened file inode object information, we have to use this system call.

(*) lstat( ) system call :-

⇒ int lstat (const char *restrict path with file name, struct stat *restrict buf);

⇒ It is used on a symbolic link file.

⇒ To change the file access permissions, we have to use below system calls. <sys/stat.h> header file.

(1) (*) chmod( ) system call :-

int chmod (const char * path name, mode_t mode);

⇒ This will change in inode object

⇒ Even we have an equivalent call for chmod

chmod (file name along with path, permissions);
0664

⇒ Even we can use predefined macros as well.

⇒ on success, it will return 0.
on failure, it will return -1.

a.out

r-- r-- r--

sudo chmod 777 /a.out.

↓ For kernal space

↓

rwx rwx rwx

↳ This command internally invokes chmod() system call.

② fchmod()

⇒ For an opened file, if we want to change the permission we have to use fchmod()

⇒ int fchmod (int fd, mode_t mode);

⇒ To change the owner name (or) group name (or) owner id (or) group id we have to use below system calls.

#include <unistd.h>.

① chown()

int chown (const char * file with path name, uid_t owner, gid_t group);

⇒ used for closed file.

② fchown()

int fchown (int fd, const char * file with path, uid_t owner, gid_t group);

⇒ Used for opened file.

→ For super user permission

$ sudo chown 1000:1000 <file name> // command

chown ("file name", 1000, 1000) // chown() sys call.

⇒ On success, it will return 0
on failure, it will return -1.

lchown ( )

⇒ To change permissions for link file.

truncate ( )

⇒ If we want to change the file size with (or) without opening a file we have to use below functions-

#include <unistd.h>.

int truncate (const char * path with file name, Off_t length

⇒ ftruncate ( )

⇒ int ftruncate (int fd, Off-t length);

⇒ On success, it will return 0,
on failure, it will return -1.

⇒ ln is used to create symbolic link.

⇒ ln will use link system call internally.

⇒ If we want to change the file name (or) directory name, we have to use below functions.

#include <stdio.h>

⊛ rename()

⇒ int rename (const char *old name, const char * new name);

⇒ To delete the file, we have to use remove func^c (library func^c).

⊛ remove() :-

⇒ #include <stdio.h>

→ int remove (const char * file name);

⇒ on success, it will return 0,
on failure, it will return -1.

⇒ To create a new directory in current working directory we have to use mkdir() system call.

⊛ mkdir()

⇒ #include <sys/stat.h>.

⇒ int mkdir (const char * path with directory name, mode_t mode);

⇒ on success, it will return 0,
on failure, it will return -1.

⊛ rmdir() :-

⇒ To delete empty directory, rmdir system call is used.

⇒ #include <unistd.h>

int rmdir(const char * path with file name);

⇒ on success, it will return 0
on failure, it will return -1.

⊛

⇒ To change the directory from a calling process, we have to use below system calls.

⇒ #include <unistd.h>.

⊛ chdir()

⇒ int chdir (const char * path ~~with~~ name);

⇒ If it is opened we can use fchdir().

⇒ int fchdir (int fd);

⇒ on success, it will return 0.
on failure, it will return -1.

07/1/23

* Scheduler :-

⇒ It is an algorithm or a program which will decide the CPU time for each & every process.

⇒ This will be loaded during system boot up time as a first process. It is a demon process which will run in the background or foreground process.

⇒ In linux, this process name is swapper.

⇒ ~~Linux~~ operating system will follow round robin scheduler, according to this scheduler each process is scheduled with equal amount of CPU time.

⇒ Each process is executed with a given CPU time.

⇒ scheduled processes are maintained in ready queue or running queue.

⇒ If the process execution is not completed within given CPU time, then it is again rescheduled

⇒ If the process execution is completed within given CPU time, that process address space is deallocated along with PCB. & CPU will jumps to the next task.

⇒ Changing (or) switching the control from one process to another process execution is known as context switching.

⇒ Context switching will happen due to following reasons:-

1. Given CPU time expired
2. process completed within given CPU time.

③ When we use Blocking calls.

④ When interrupts or exceptions are raised

⑤ When high priority task is executed.

⇒ The scheduler algorithm depends upon operating system. In real time operating system, we see priority based Schedulers. In this, equal amount of CPU time is assigned based on priority. So, CPU will execute the processes based on priority values.

⇒ In some OS, least value will have high priority.

⇒ In some OS, highest value will have highest priority.

09/01/23

* Threads :- (32-bit machine → 2MB or 32MB is allocated for stack area of threads & depends on architecture). (64-bit ↑)

⇒ Threads are ntg but light weight processes which are created within the process itself.

⇒ Each process will have atleast 1 thread i.e., main thread.

⇒ Each thread will have one entry point i.e., function from where the thread execution will start.

⇒ A thread within the process utilizes the existing resources of the process, thats why it is known as light weight process. Even no kernal objects are created like processes.

⇒ By using threads, the given CPU time (or) time given for the process is utilized efficiently

⇒ (*) when we try to create thread, try to create independent

⇒ Even for parallel execution of the code, we will use threads.

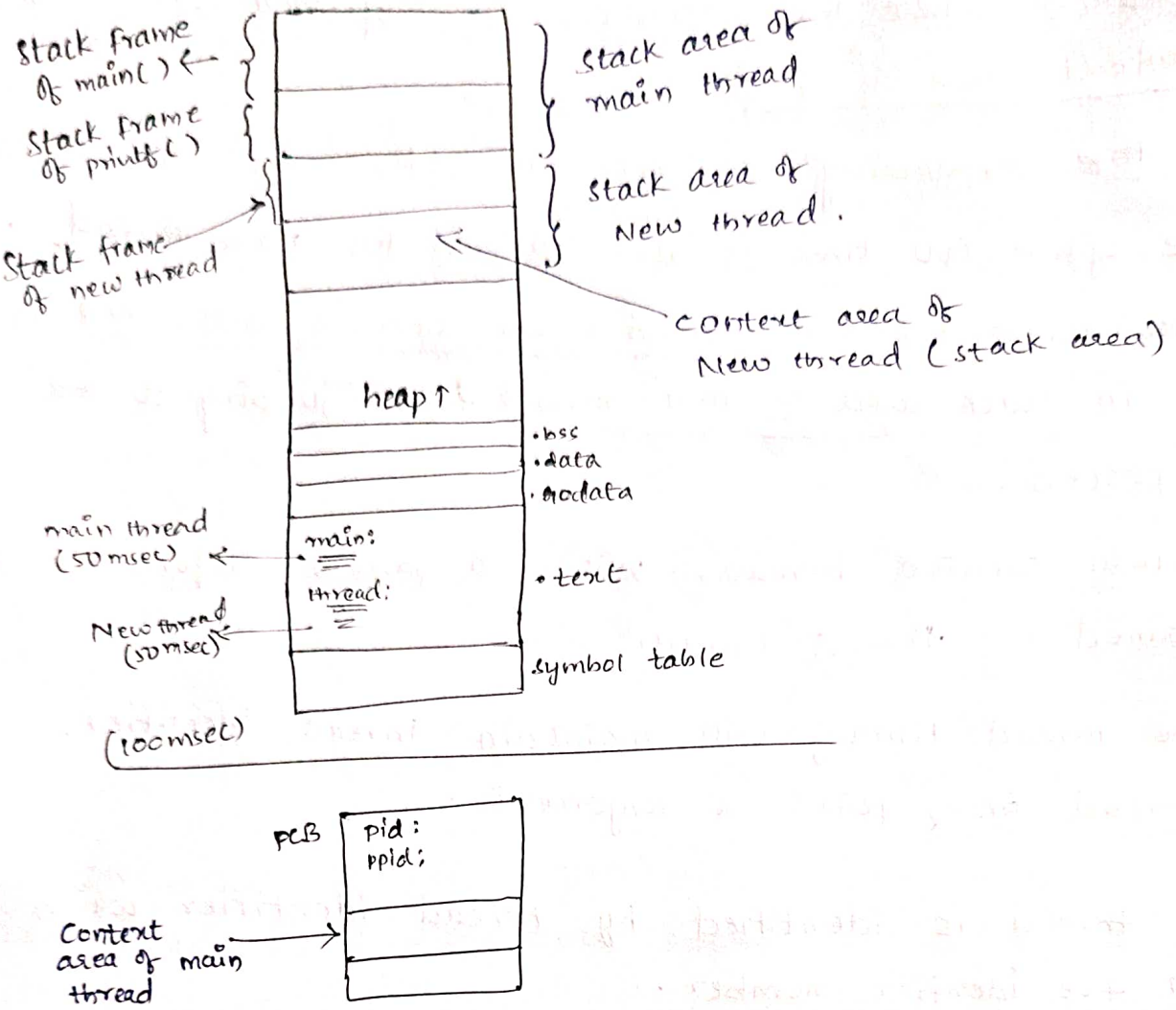
⇒ The sequential execution of 'c' statements by utilizing the process resources in the given CPU time by using threads

⇒ The main thread entry point is main().

⇒ For each thread, internally it will allocate (or creates) stack area. (a block of memory gets allocated).

⇒ The functions which are invoked from the threads, there stack frames are created within that stack area.

⇒ The given CPU time for a process is equally shared by multiple threads within a process.



⇒ User space threads are different from Kernal space threads.

⇒ In this process, we can see two threads,

① main thread
② New thread

① ⇒ main thread execution starts from main()

② ⇒ New thread execution stats from new thread entry point

⇒ For Both the threads internally it will allocate separate stack area.

⇒ For main thread & new thread, the cpu time is shared Equally initially (50msec – 50 msec)

⇒ The main thread is executed for 50msec. If the time is expired before ~~the~~ starting the execution of new thread

the instance where the execution is suspended is stored in context area of PCB.

⇒ Now, the remaining 50msec (CPU time) is utilized by New thread if the given CPU time is also expired for new thread, the instance where the new thread's execution is suspended is stored in stack area of that thread before jumping to the next process.

⇒ So, newly created threads within a process, information is maintained by thread library.

① The thread library will maintain thread identifier, thread entry point & arguments.

⇒ Each thread is identified by thread identifier which is a +ve idenfier number.

⇒ All these New threads are created from main thread.

⇒ If we want to access the thread from the process it has to use thread id.

⇒ The threads created within the process can't be accessed by other processes.

⇒ To create a thread, we have to use thread library functions. compiler doesn't know about thread library functions. so, we must link the library during compilation time. The thread library name is p-thread (posix thread)

$ gcc -lpthread. (to link pthread library)

⇒ If we have multiple threads within a process, that is known as multi threaded process.

⇒ If we don't have a new thread within the process, then control goes to next process after the process termination (or) CPU time expired

⇒ To create a new thread (from a process), we have to use below library function

pthread_create :

⇒ We have to include #include <pthread.h>

```
int pthread_create (pthread_t *thread, const pthread_attr_t
                    *attr, void *(*start)(void *), void *arg);
```

- thread: empty thread identifier
- attr: thread attributes
- start: thread entry point (func base address)
- arg: passed the input to the thread func.

⇒ On success, it will return '0'.
On failure, it will return error.

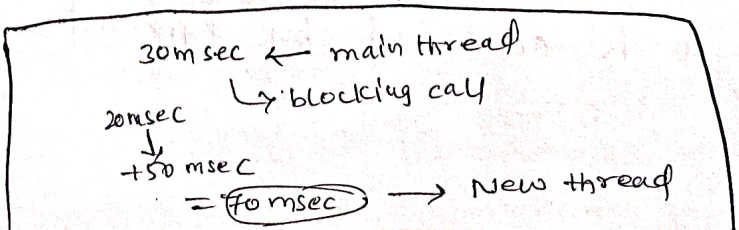
⇒ The pthread_create library func internally invokes clone system call. When this call is successful, the first input gets updated with thread identifier & the entry gets created in thread table maintained by the library. (pthread library)

Ti	entry	arg	thread attributes

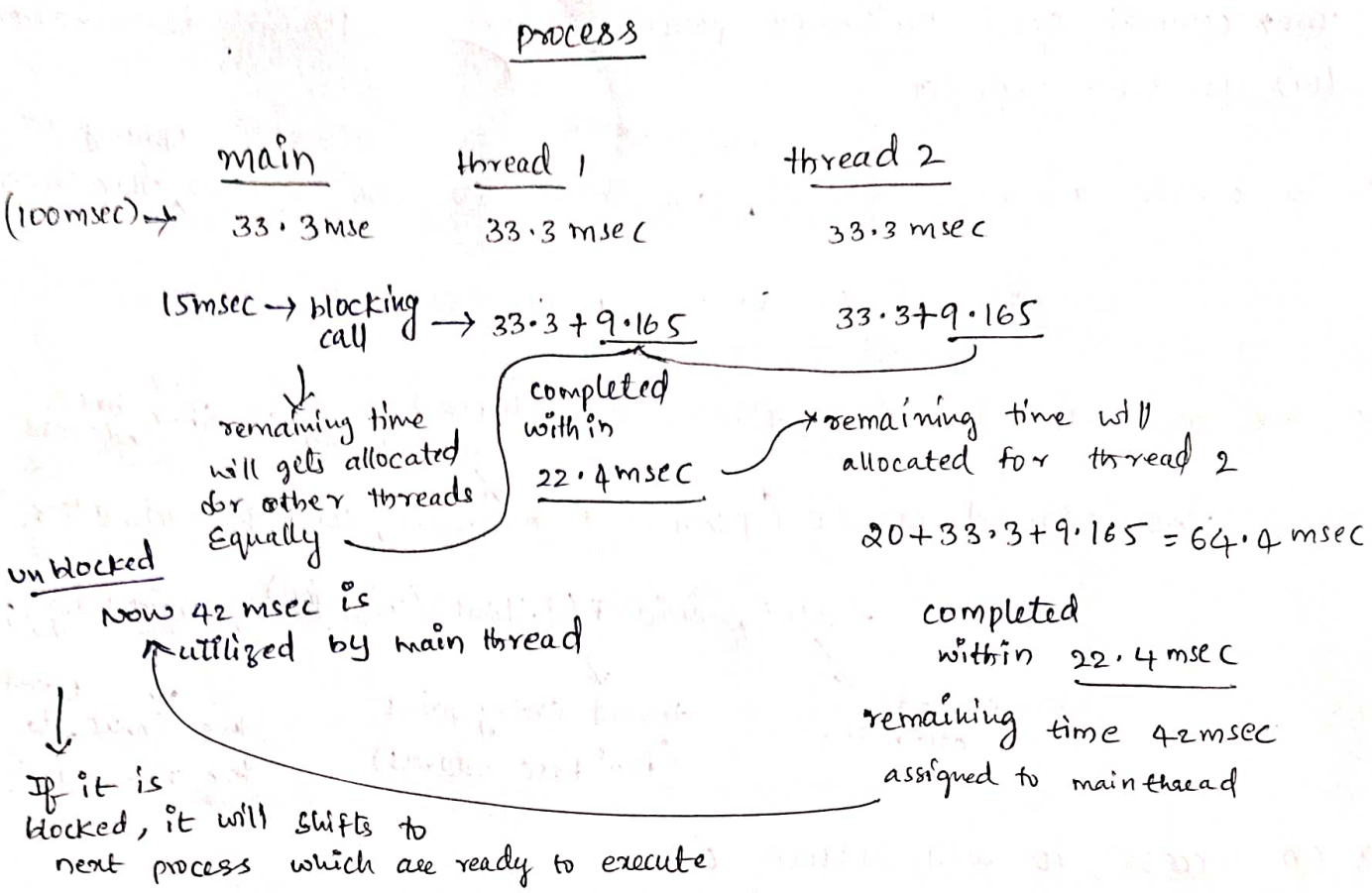
⇒ Thread entry func should be same type as mentioned in third (3rd) argument.

⇒ The thread library will decide the CPU time for the thread

⇒ If main thread uses a blocking call after 30 msec, the remaining 20 msec is assigned to the new thread.



Example :-



⇒ The context switching within the threads will happen multiple times

⇒ we should not call exit func from other than main threads, cuz that will effect entire process

⇒ main thread should be alive, till other threads completes its execution. cuz --- as soon as the main() completes its execution the entire process gets terminated. that means the threads within the process also terminated automatically.

Threads termination :-

⇒ 1) Threads Entry point func uses return statement.

2) If it uses pthread_exit.

3) If the thread is cancelled by pthread_cancelled.

* pthread_exit :-

#include <pthread.h>

⇒ void pthread_exit (void * ret val);

⇒ This fun' terminates the calling thread & specifies a return value can be obtained in another value by calling pthread.

⇒ Each thread within a process is identified by thread identifier

(*) How do we get the thread id from the same thread ?

⇒ we have to use pthread_self Function.

⇒ pthread_t pthread_self (void);

⇒ On success, it will return thread id.

⇒
```
#Include <pthread.h>
void main()
{
    pthread_t ti;
    int i, ret, loop = 1000;
    ret = pthread_create (&ti, 0, mythread, &loop);
        sleep(5);   // sleep(1);
    // exit(0);
}
```
↓ func' name

(1) (2) Verify with & without sleep

⇒
```
void * mythread ( void * ptr)
{
    int i;
    PF("my thread is invoked\n");
    for(i=0; i<= *(int *)ptr ; i++)
        Printf("%d\n", i);
}
```
&loop

(3) → use exit(0) & verif the o/p.

(*) ⇒ main()
{
pthread_t ti; → thread identifier
int i, ret, loop=1000;
ret = pthread_create(&ti, 0, mythread, &loop);
Pthread_exit(&ret);
}

⇒ void * mythread (void *ptr)
{
int i;
PF("my thread is invoked\n");
for(i=0; i<= *(int*)ptr; i++)
PF("%d\n", i);
}

$gcc filename -lpthread

⇒ when we call pthread_exit from main() (or) main thread, the main thread gets terminated & remaining threads continue their execution

(*) How do u wait for a particular thread termination ?

⇒ It should use pthread_join().

thread identifier on which we want to block

int pthread_join (Pthread_t ti, void **ret val);

address of the pointer variable

⇒ On success, it will return 0.
on failure, it will return error

⇒ if ret val is a non-NULL pointer, then it receives a copy of terminated threads return value.

⇒ In the above example, instead of sleep we can use pthread_join, we can make main thread block until new thread completion.

⇒ The size of stack area in 32 bit machine is 2 MB
" " " 64 bit machine is 32 MB

⇒ stack area is created for threads

⇒ stack frames are created for functions.

⇒ we can change the size of stack area by using a library call pthread_attr_set stack size

⇒ Even we can change the stack size (or) by using a command ulimit.

⇒ we can change the location of stack area by using a library call pthread_attr_set stack.

⊛ what are the segments that are shared by multiple threads within a process.?

⇒ symbol table, text, rodata, bss, data, heap will shared by multiple threads.

⇒ stack will not shared by multiple threads.

⇒ These multi threaded applications we can see in client to server communication.

⊛ Rules for client to server communication :-

⇒ you should have only one server & can have multiple clients.

⇒ server wait or blocks until it gets request from client.

⇒ First server should run, then only clients can send the request.

⇒ once server receives the request it process the data & sends the response back.

Eg:- chatting applications, Gmail web applications etc.

* For every client request, server will create one thread.

⇒ Instead of threads, even server can create a process.

Both are having their own advantages based on their requirement.

✱ Advantages of threads :-

① Creation & maintaining the threads is easy.

② Creation of threads uses less memory resources.
(or) requires

③ Creation of threads will consumes less time.

④ context switching b/w the threads are faster.

⑤ Given CPU time is utilized efficiently.

⑥ All the threads within the process can share the memory segments commonly .text, .rodata, .data, heap, symbol table etc except stack.

✱ Disadvantages of threads :-

① Sharing the data between multiple threads, there may be synchronization (or) mutual exclusion issues (updation issues)

To over this, we have to use ① mutex locks (or) mutual exclusion locks
These are synchronization techniques. ② conditional variables.

② A bug (or) error in one thread that effects the all other threads.

③ User space threads are completely different from kernal space threads.

④ Kernal space threads are similar to child process.

⑤ From user space, we can see multiple threads that are running in kernal by ps -elf // ps -elf

⇒ The processes which are highlighted in square brackets that are Kernal threads.

* Mutual exclusion (or) updation issues :-

⇒ when multiple threads within a process trying to access critical section (global variables, heap segments, shared memory region) we can see updation issues or mutual exclusion issues.

```
int glob = 0;
main()
{
    void *ret;
    Pthread_t t1, t2;
    int loop = 2000;
    pthread_create(&t1, 0, thread_fun1, &loop);   → thread 1
    pthread_create(&t2, 0, thread_fun2, &loop);   → thread 2
    pthread_join(t1, &ret);                       ← blocked
    pthread_join(t2, &ret);
    printf("%d\n", glob);
}                                  → 3000   // but the o/p should be 4000.
                                            This is known as updation issues
```

① Information where it is suspended is stored in context area of PCB (main thread)

② context area of stack frame of corresponding threads. (thread 1 & 2)

⇒ void *· thread_fun1 (void *arg) ← entry point

```
{
    int loop, i, local;
    loop = *(int *) arg;
    for(i=0; i<loop; i++)
    {
        local = glob;
        local++;
        glob = local;
    }
}
```

critical section

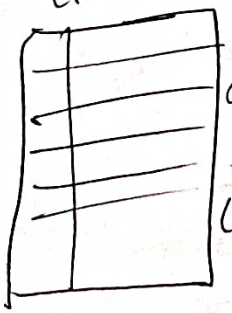
→ 9t is suspended as CPU time given to thread is expired

1st time

glob = 999
local = 1000
glob = local (1000)
i = 1000
iterations = 1000

→ 44 msec is expired

CPU registers



2nd time

glob = 1000, local = 1001
Pending iteration = 1000
continues the execution & completed within 40msec
i = 2001, local = 1999, glob = 2000

void * thread_fun2 (void * arg)

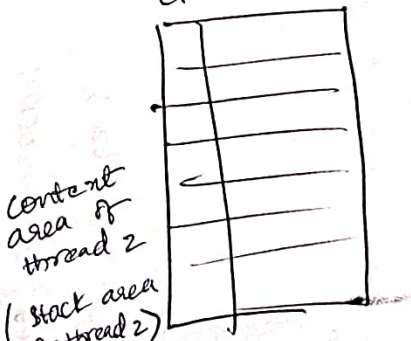
```
{
    int loop, i, local;
    loop = *(int *) arg;
    for(i=0; i<loop; i++)
    {
        local = glob;
        local++;        → suspended
        glob = local;
    }
}
```

local = 1998
(1999)
local++;

1st time

i = 1000
glob = 1998
iterations = 1000

CPU registers



2nd time

glob = 1999 i = 1001, local = 2000
Pending iterations = 1000
continues & completes with in 40 msec

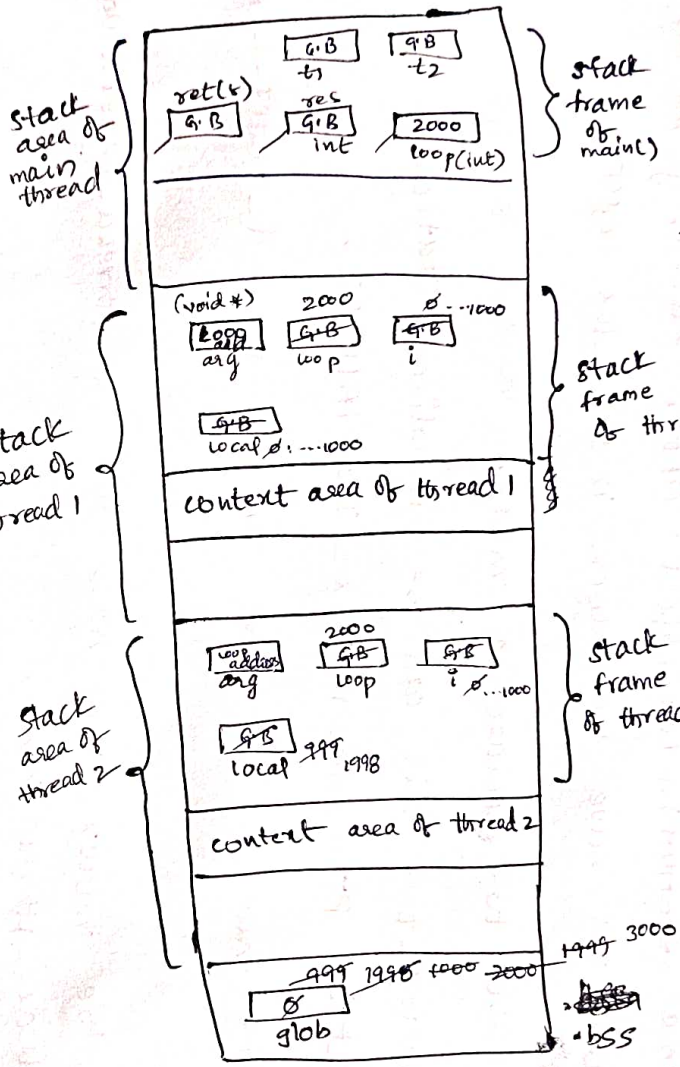
i = 2001, local = 3000, glob = 3000.

100 msec glob = 0

main	thread 1	thread 2
10msec ⇒ rem = 90sec	30msec	30msec
30msec		
2msec → pthread_join()	14msec	14msec
→ blocking state ← blocking call → until thread1 completes	30+14	30+14
(remaining time) 28 msec	44msec	44msec

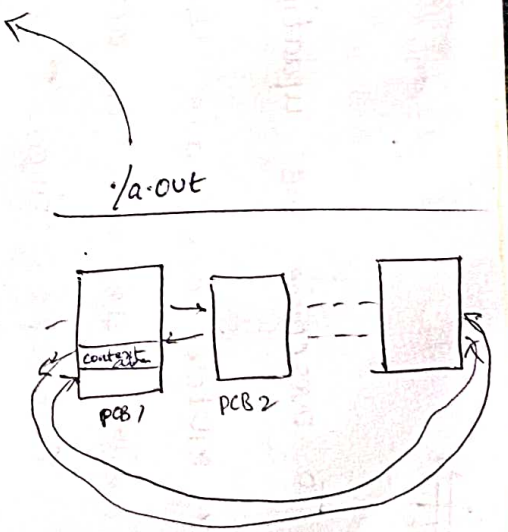
100 msec

main	thread 1	thread 2
33.3	33.3	33.3
3.3 msec ⇒ 30msec verified	33.3+15	33.3+15
(blocking state)	48.3	48.3 msec
	Completed - 2000 loops in 40 msec	48.3+8.3
		⇓
		56.6msec - 40msec
		= 16.6msec
		⇓
		(gives to main thread)



thread library table

	t1	attributes	Entry point	inputs
thread1 →	pthread1	0	thread_fun 1	loop = 2000
thread2 →	pthread2	0	thread_fun	loop = 2000



* To overcome the updation issues, we have to use mutex-locks.

⇒ Mutex locks are used to protect the shared resources. The code that is accessing the global resources is ntg but critical section. Before accessing the critical section we have to lock the section mutex object. After executing the critical section we have to unlock the mutex object.

⇒ First we have to declare a mutex variable of type pthread_mutex_t. This can be initialised statically by using a mactro PTHREAD_MUTEX_INITIALIZER.

⇒ To lock this mutex, we have to use pthread_mutex_lock function

⇒ To unlock the mutex, we have to use pthread_mutex_unlock

```
# include <pthread.h>
```

⇒ int pthread_mutex_lock (pthread_mutex_t *mutex);

↑ we need to pass mutex variable's base address to this fun^c.

⇒ int pthread_mutex_unlock (pthread_mutex_t *mutex);

⇒ on success, they will return 0.

⇒ on failure, they will return Error.

⇒ lock & unlock is also known as aquire & release.

⇒ The thread which locks the mutex will become the owner of that mutex, only that particular thread can unlock the mutex.

(only owner can unlock)

⇒ This mutex variable internally maintains a counter on which these locks or unlocks are performed.

⇒ By default when we initialize the mutex, it will be in unlock state.

⇒ when one thread locks the mutex, when other thread trying to release the mutex that threads goes into blocking state until the owner releases the lock.

⇒

```
int glob=0;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void main()
{
  pthread_t t1, t2;
  int res, loop=2000;
  void *ret;
  pthread_create(&t1, 0, thread_fun1, &loop);
  pthread_create(&t2, 0, thread_fun2, &loop);
  pthread_join(t1, &ret);
  pthread_join(t2, &ret);
  printf("%d\n", glob);
}
```

→ main() gets blocked. & the info is stored in context area of PCB.

Now the thread starts executing for its time being.

After executing this statement the threads entry will gets updated in thread library table.

ti	attributes	Entry point	i/p's
pthread_1	0	thread_fun 1	loop=2000
pthread_2	0	thread_fun 2	loop=2000

⇒ void * thread_fun1 (void *arg) (§ 2000 loop)

```
{
  int local, i, loop;
  loop = *(int *)arg;          (2000)
  for(i=0; i<loop; i++)        (1) (2000)
  {
    pthread_mutex_lock(&mtx);
    local = glob;
    local++;
    glob = local;
    pthread_mutex_unlock(&mtx);
  }
}
```

critical section

① Time is expired

30 msec

thread1 ← thread2

40 msec

1st → 1000, i=1000, local=1000, glob=999

30 msec

1000 + 8000 i=18000, local=1800, glob=1800

i++ local 2000 glob =2000

⇒ void * thread_fun2 (void *arg)

```
{
  int local, i, loop;
  loop = *(int *)arg;
  for(i=0; i<loop; i++)
  {
    pthread_mutex_lock(&mtx);
    local = glob;
    local++;
    glob = local;
    pthread_mutex_unlock(&mtx);
  }
}
```

critical section

⇒ pthread_mutex_lock behaves as a blocking call, if it is already locked by other threads.

main thread1 thread 2

100msec

10msec

30msec 30msec 30msec

10msec

Blocking call

30+10 = 40msec 30+10 = 40msec

10 msec

---

100 msec

33.3 33.3 33.3

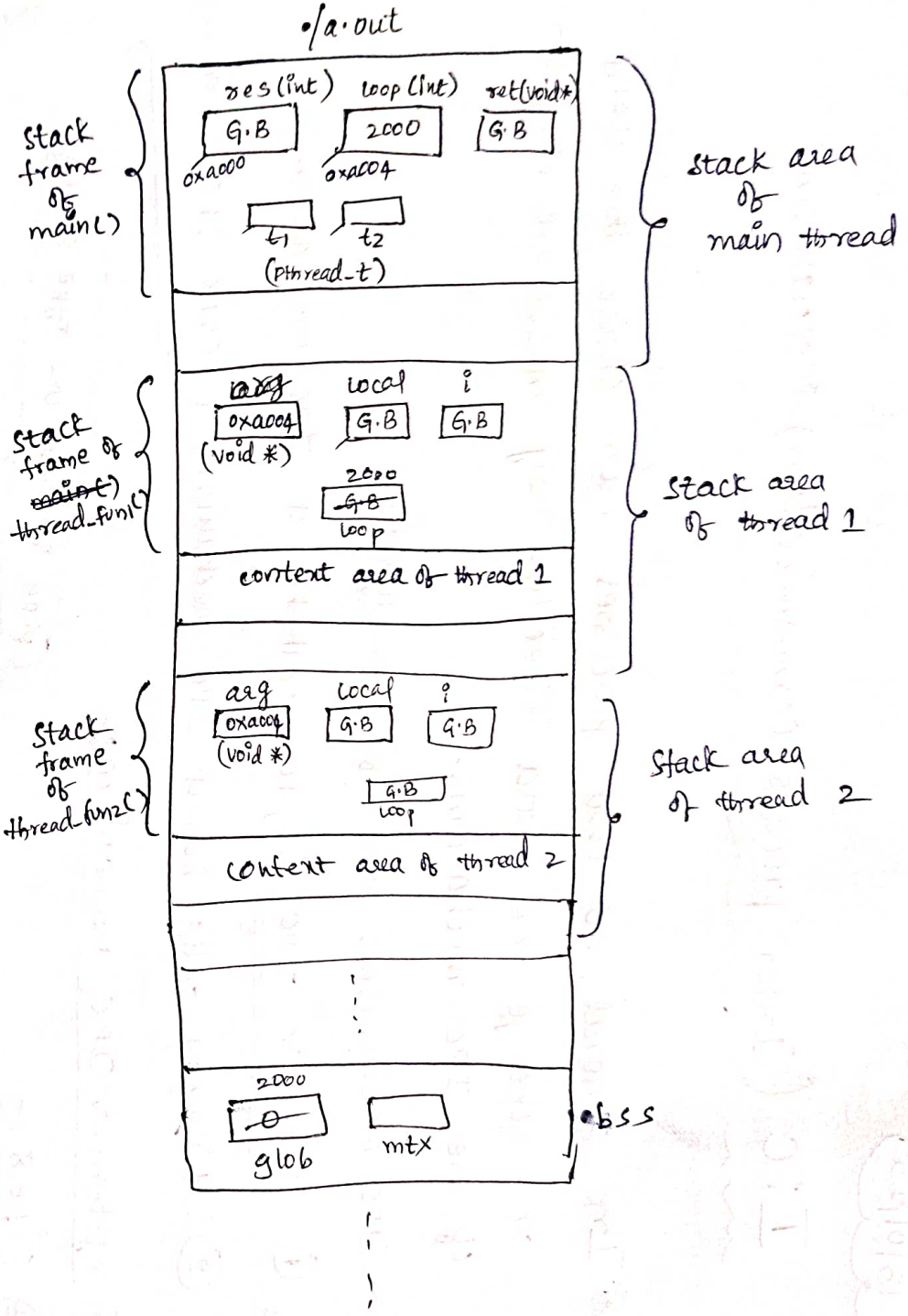
(blocking) Verified 3.3 msec

33.3+15 = 48.3 msec

10msec completed → 38.3

33.3+15 = 48.3

48.3 → 48.3+38.3 = 86.6 msec



13/01/23

IPC (Inter process communication) Mechanisms

⇒ In general, no two processes can share the data or exchange the data directly. They must use any one of the IPC mechanisms.

⇒ IPC mechanisms are of two types :-

① system-V IPC (system call based IPC mechanism).

② Library calls based IPC mechanism (POSIX IPC mechanism)

① system-V IPC mechanism :-

pipe is also one type of file

Pipes :-

⇒ pipes are the oldest form of IPC mechanism. They are half duplex. Data flows only in one direction (i.e., one process to another process)

⇒ pipes can be used only between the two related processes (b/w parent & child communication).

⇒ pipe is ntg but a kernal buffer created in kernal space through which parent & child can communicate the data.

⇒ Every time we type a sequence of commands in a pipe line for shell to execute. The shell will create a separate process for each command & links the standard output of one process to the standard input of next process by using pipes.

⇒ A pipe is created by calling a pipe system call

⇒ we have to include <unistd.h> header file

(*) int pipe (int fd[2]);

⇒ on success, it will return 0
on failure, it will return -1.

⇒ when this call is succesful it will create 2 file objects & this file objects entries are updated in FD table. These entries are updated in your input argument

⇒ so fd[0] will gets updated with file descriptors
&
Fd[1]

⇒ Fd[0] is open for reading.

⇒ Fd[1] is open for writing.

⇒ By using the pipe, we can send the data from parent to child (or) from child to parent.

⇒ read() system call on these Fd's behaves as a blocking call until parent or child writes the data in to the buffer.

⇒ Improper handling of pipes will generate SIGPIPE signal to terminate the processes.

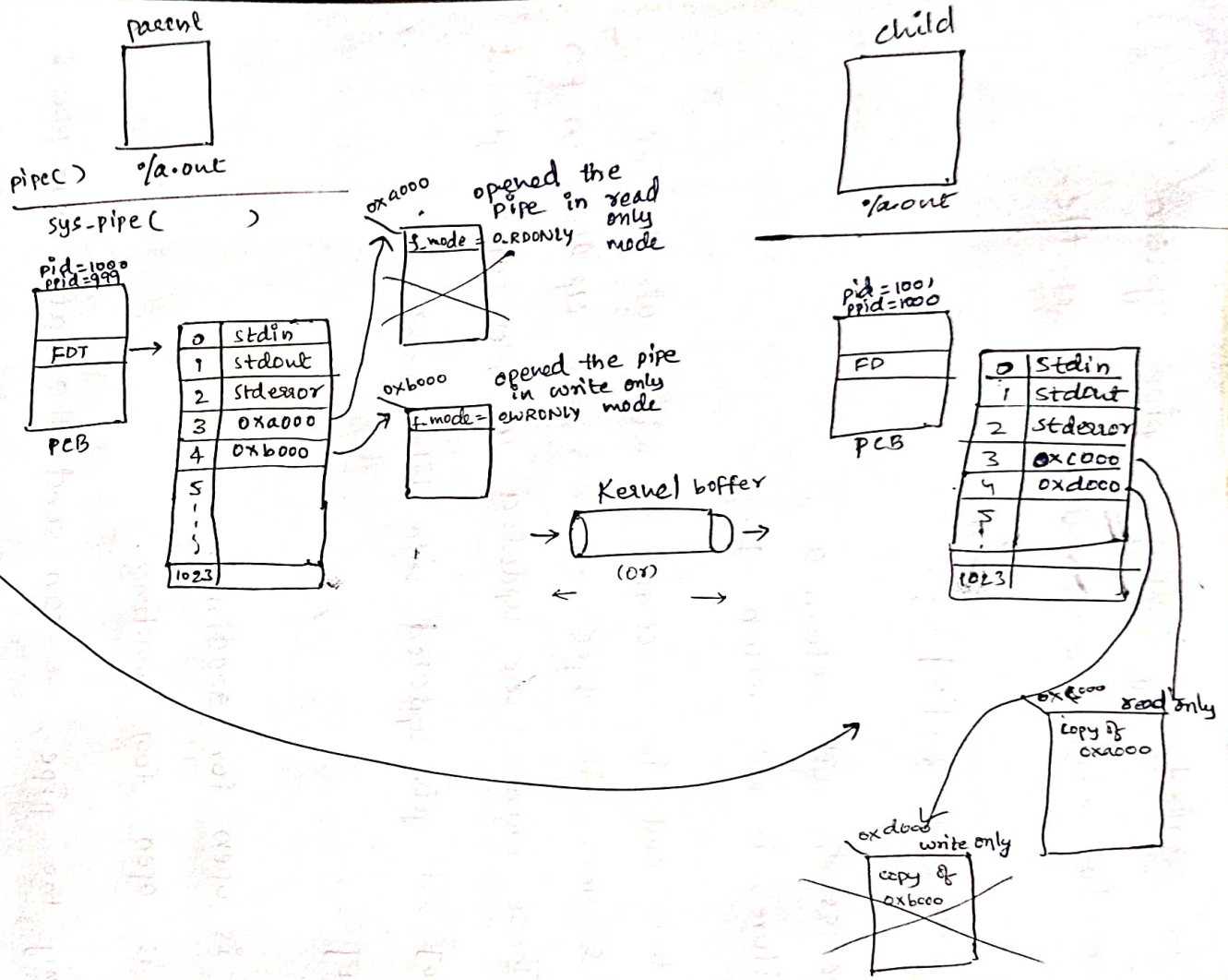
```
⇒ main()
{
   pid_t pid;
   int fd[2];
   char buf[100];
   int ret;
   ret = pipe(fd);
   pid = fork();
   if (pid<0)
   {
      PF ("Failed to create child\n");
      exit(1);
   }
   else if (pid>0)
   {
      close (fd[0]);
      ret = write (fd[1], "Hi", strlen("Hi"));
      close (fd[1]);
      exit(0);
   }
   Else
   {
      close (fd[1]);
      ret = read (fd[0], buf, 100);
      write (1, buf, ret);
      close(fd[0];
   } exit(1);
```

3 fd[0] } fd
4 fd[1] }

parent process

child process

blocking call until parent writes the data into pipe



Limitations

⇒ Pipes can be used only b/w two related processes

⇒ These are unidirectional & half duplex. As soon as the process terminates the pipe & corresponding objects are destroyed.

⊛ Named pipes :-

⇒ It is also ntg but FIFO's

⇒ These are used in b/w unrelated processes. ntg but different processes.

⇒ To understand this we have to use server to client communication.

⇒ To create these FIFO's object we have to use mkfifo() system call

⇒ include<sys/stat.h>. we have to include.

⇒ int mkfifo (const char * path, mode_t mode);

17/01/23

⇒ This call on success, it will return 0
on failure, it will return -1.

⇒ It creates a fifo object in current working directory.

⇒ Each process should have its own FIFO object in current working directory.

⇒ Same file permissions (or) modes are applicable on FIFO objects. But, it is not a normal textual file, it is a special file.

⇒ open() & read() behaves as blocking calls on FIFO objects ~~until~~

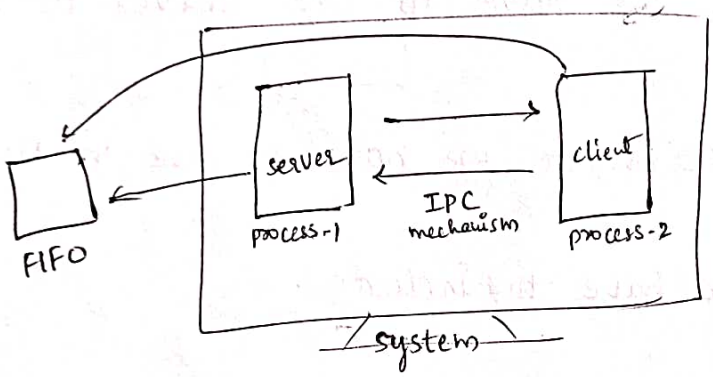
⇒ open() behaves as a blocking call until other process opens the same FIFO object.

⇒ read() comes out of the blocking state until other process writes the data on the same FIFO object.

Inter process communication

related process ⇒ pipes

unrelated process ⇒ other IPC mechanisms.



⇒ Let us consider, server to client application.

⊛ Rules :-

1) ⇒ Server has to start first. and we should have only 1 Server which can communicate with multiple files.

2) ⇒ server has to wait (or) block until client sends the request. After receiving the request, server processes the request & sends the response back.

3) ⇒ For two way communication both server & client should create their own FIFO's & they must open it.

* Differences b/w pipes & Named pipes *

Pipes	Named pipes
① pipes are used between related processes	① Named pipes are used b/w Unreleated processes
② pipe is created as a kernal buffer in kernal space	② A special file is created as FIFO Object in current working directory.
③ Two Entries are updated in FD entry table.	③ only one entry gets updated in FD table.
④ read() behaves as a blocking call	④ Both open() & read() system calls behaves as a blocking calls.
⑤ when process terminates, the pipe, corresponding objects & buffer gets destroyed.	⑤ when process terminates, the FIFO objects still exist in current working directory.

=> If we try to create & open, the server fifo & client fifo simultaneously, the both processes will goes to infinite dead lock scenario. so we need to open one after the other.

(*) Assignment :-

① Toggle the characters

② Implement own server to client chatting.

server

```
main()
{
  int ret, fd;
  char wbuf[20] = "Hi";
  char rbuf[20];

fd = open("server fifo", O_RDONLY);
 if (fd<0)
  {
    ret = mkfifo("server fifo", 0660);
     if (ret<0)
     {
       PF(" Failed to create fifo obj\n");
       exit(0);
      }
    else
     {
       fd = open("serverfifo", O_RDONLY);
      if (fd<0)
      {
        PF("Failed to open\n");
        exit(2);
       }
   }
ret = read(fd, rbuf, 20);
  if (ret<0)
  {
    PF(" Failed to perform read operation\n");
    exit(3);
  }
ret = write(1, rbuf, ret);
   if (ret<0)
   {
     PF(" Failed to perform write\n");
     exit(4);
   }
  close(fd);
}
```

→ it will be in blocking state until same fifo is opened by client

→ blocking

→ blocking

client

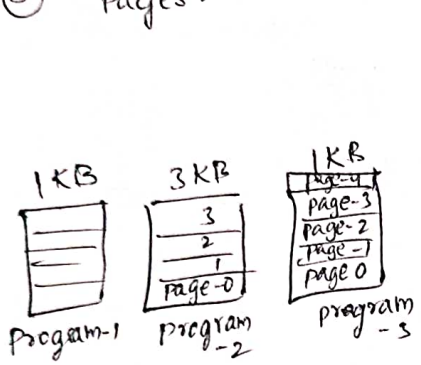
```
main()
{
  int ret, fd;
  char rbuf[20], wbuf[20] = "request";

fd = open("server fifo", O_WRONLY);
 if (fd<0)
 {
   PF(" failed to open\n");
   exit(0);
 }
ret = write(fd, wbuf, strlen(wbuf));
  if (ret<0)
  {
    PF(" Failed to write\n");
    exit(2);
  }
 close(fd);
}
```

once write is done this read will comes out of blocking state.

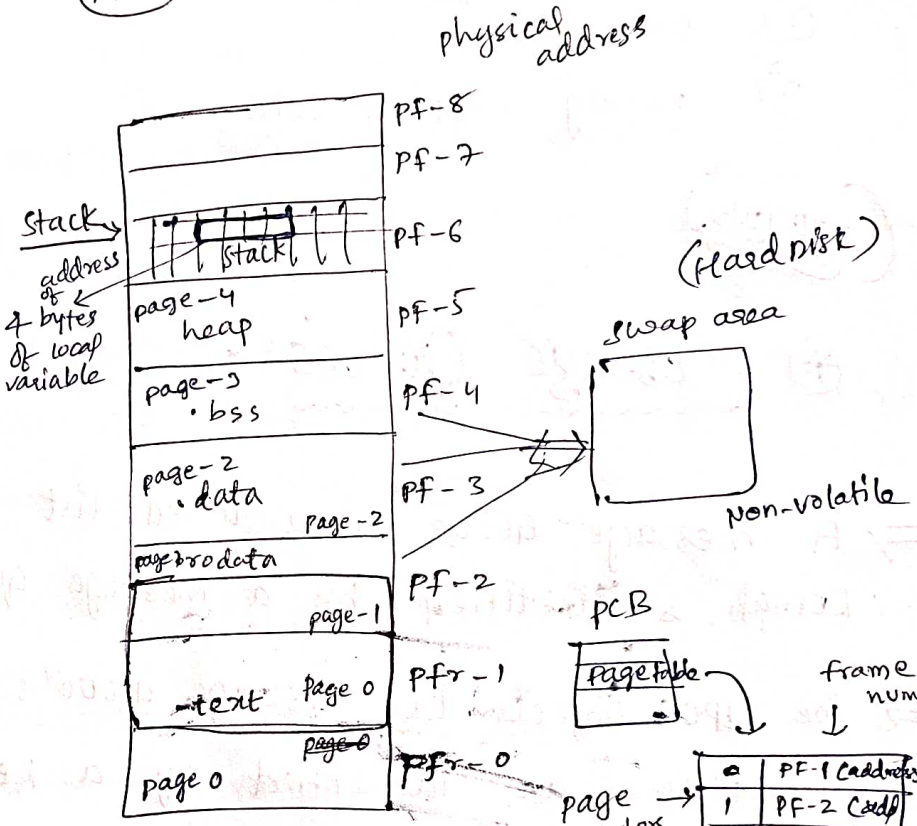
* Virtual Memory *

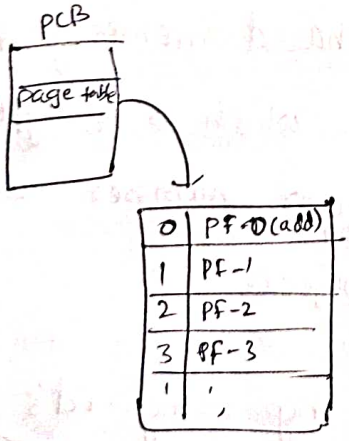
① Direct prgm is loaded
② segmentation
③ pages.

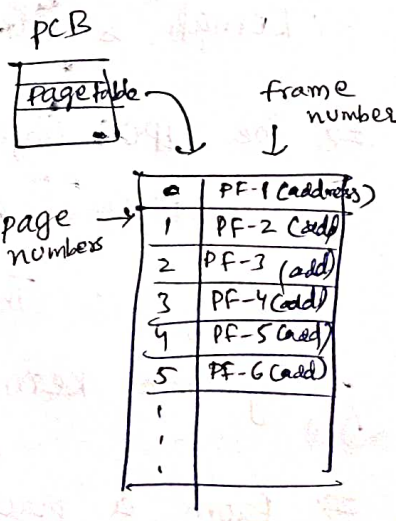
architecture dependent
2 KB / 4 KB / 8 KB



process ⇒ virtual address







physical address → may be modified (if it is swapped) (execution time)

virtual address ⇒ same (MMU)

Virtual address = page no * Pframe size + offset = physical address

Additional pages are created when

① function is invoked
② Recursive functions
} Stack frames will create

③ New threads are created → stack area will gets created
④ alloc a( )
⑤ shared memory
⑥ DMA calls
⑦ memory mapping calls.

18/01/23

⊛ Message Queues :-

⇒ A message queue is a linked list of messages stored in a kernal & identified by a message queue identifier.

⇒ The IPC objects like message queue's, shared memory & semaphores are identified by a key which is of type key_t. This key is a positive integer number used by the kernel to identify the IPC objects

⇒ From a process, we can access them by using id's (message queue id, sem id, SHM id). This key can be generated by dynamically by using a system call ftok()

⇒ It is defined in an header file #include<sys/ipc.h>

⇒ Each IPC object is associated with a permission structure which is of type struct IPC_perm

⇒ It is defined in an header file. <sys/ipc.h>

⇒ This structure gets created & updated automatically when we create IPC object.

⇒ These IPC object structures are alive until they removed Even though process is terminated.

⇒ This structure contains modes, effective user id, effective group id & so on - - -

⇒ Each message queue object is identified by a structure struct msqid_ds.

⇒ The ipc_perm structure is the member of these objects

⊛ Find out the structure definition & its members of msqid_ds.

⇒ This structure even maintains no. of msgs, message size & last msg sent pid, last msg received pid, last msg sent time, last msg received time & last msg change time.

⇒ Maximum size of message you can send according to linux 3.2.0 is 8192 bytes.

⇒ we can create (or) open a message queue object by using msgget( ) system call.

⇒ we have to include # include <sys/msg.h> header file.

⇒ int msgget (key_t key, int flags); (key should be more than 2 bytes value; flags → IPC_creat | 0660)

⇒ On success, it will return message queue id &.
On Failure, it will return -1

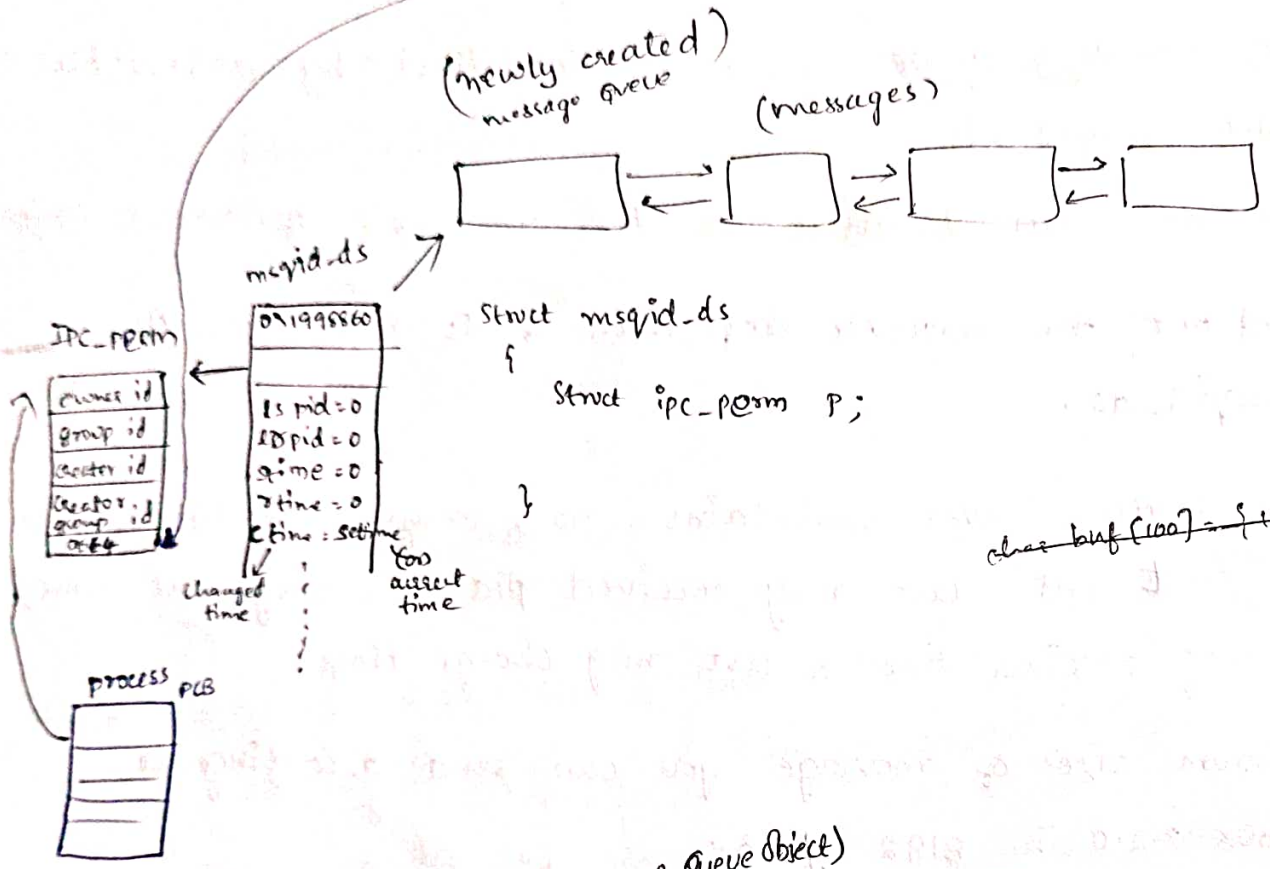
⇒ If we want to create new IPC object, we use IPC_CREAT|0660

⇒ If the IPC object is already exist, second argument will be zero.

define KEY 0x1998860

IPCS command

msgget (KEY, IPC_CREAT | 0664);



Struct msqid_ds
{
Struct ipc_perm p;
}

~~char buf[100] = { long msgtype, long pid, };~~

(message queue object)

⇒ when we create this object, some of the members of msqid_ds are initialized, creater id, creater group id & mode gets updated in IPC_PERM.

lspid, lrpid, last send, Key, Last received, ~~change time~~ are initialized with zeroes, change time initialized with current time in msqid_ds.

⇒ message queue bytes are updated with system limit.

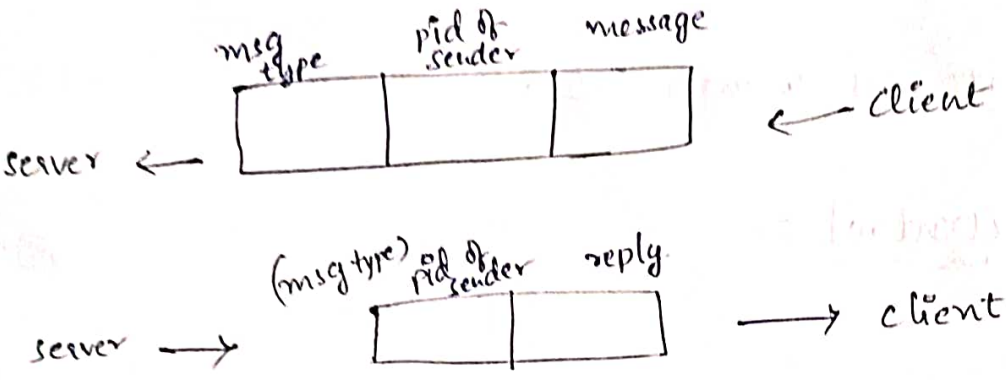
⇒ To send the message we have to use msgsnd()

system call.

msqid returned by msgget

⇒ msgsnd(int msqid, const void * ptr, size_t nbytes, int flag);

⇒ To send the message we must follow the format/pattern



→ The pid of the sender is used as the msg type to send the response back to the client from server.

⇒ msg type should be long

⇒ Creating struct members (or) data types in a buffer

```
char buf[30];
void *ptr = buf;
((long int *) ptr)[0] = MSG-TYPE;
((long int *) ptr)[1] = getpid();
strcpy (&buf[2 * sizeof (long int)], "Hello");
```

⇒ Message receive :-

→ It behaves as a blocking call until message type matches from the sender side.

⇒ #include <sys/msg.h>

⇒ ssize_t msgrcv (int msqid, void *ptr, size_t nbytes, long type, int flag);

(msqid returned by msgget; buffer base address; size of buffer; msg type)

→ Generally, msgrcv behaves as a blocking call, To avoid that blocking 5th argument should be IPC_NOWAIT.

⇒ On success, it will return the size of data portion of the message.

⇒ on Failure, it will return -1.

* Message control :-

⇒ msgctl

⇒ #include <sys/msg.h>

⇒ int msgctl(int msqid, int command, struct msqid_ds *buf);

(command: it is a pre defined macro; msqid_ds: dependent on command)

⇒ This func is used to control msqid ds structure

⇒ By this func, we can update, read, delete

⇒ IPC_STAT ⇒ It is used to fetch the msqid_ds structure info in this case, 3rd argument should be structure variable's base address (msqid_ds type).

⇒ IPC_SET ⇒ In this case we can update very few structure variables members like mode, user id, group id's, effective user id, effective group id's & length of the message.

⇒ After updating this, the structure variables base address should be passed as 3rd argument.

⇒ IPC_RMID ⇒ This will delete the message queue along with its objects. The third argument should be zero (0) or NULL.

⇒ IPCS is the command used to see the IPC object information.

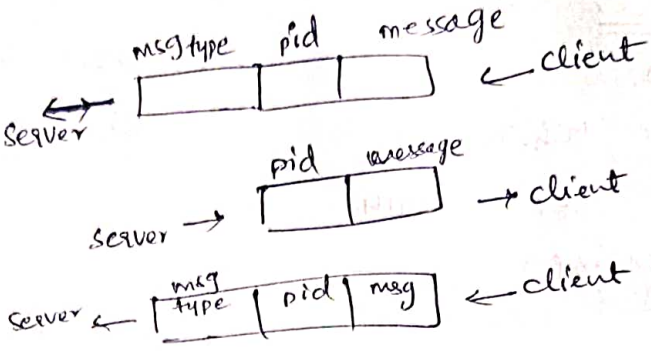
⑥ Assignment

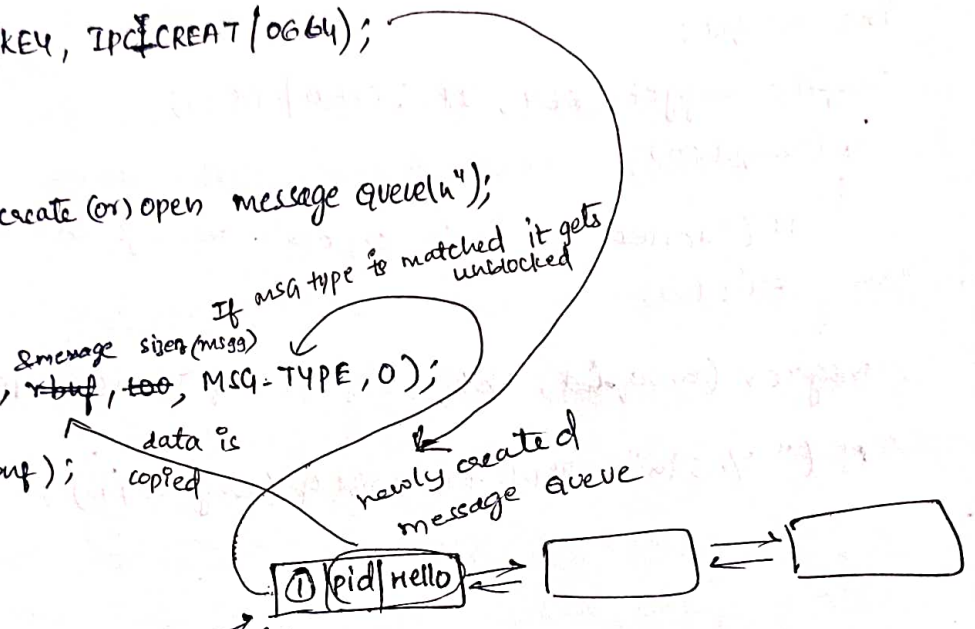
⇒ toggle

⇒ posix message queue

Server

```
# define MSG-TYPE  1
# define KEY   0x1998860
        struct msg
main()    {
          };
{
 char rbuf[100];
  int msqid;
 msqid = msgget (KEY, IPC_CREAT | 0664);
  if (msqid < 0)
  {
   PF(" failed to create (or) open message queue\n");
    exit(1);
  }
blocking state
    msgrcv ( msqid, rbuf, too, MSG-TYPE, 0);
     PF(" %s\n", rbuf);
}
```





Client

```
struct msg
{
   long msgtype;
   long pid;
   char buf[100];
};
# define MSG-TYPE  1
# define  KEY    0x1998860
 main()
 {
   int msqid;
   struct msg message;
  msqid = msgget ( KEY, 0);
  if (msqid < 0)
  {
   PF ("failed to open\n");
   Exit (1);
  }
  message.msgtype = MSG-TYPE;
   message.pid = getpid();
    Strcpy (message.buf, "Hello");
msgsnd(msqid, &message, (2 * sizeof (long int)) + strlen(message.buf) + 1, 0);
```

(or)

Server

(without using struct msg structure)

Using type casting of buffer.

```
#define MSG_TYPE 1
#define KEY 0x414

main()
{
    char rbuf[100];
    int msqid;
    msqid = msgget(KEY, IPC_CREAT|0664);
    if(msqid<0)
    {
      PF("failed to create or open message queue\n");
      exit(1);
    }
    msgrcv(msqid, rbuf, sizeof(rbuf), MSG_TYPE, 0);
    PF("%s\n", rbuf+(2*sizeof(long int)));
}
```

Client

```
#define MSG_TYPE 1
#define KEY 0x414

void main()
{
    int msqid;
    char rbuf[100];
    msqid = msgget(KEY,0);
    if(msqid<0)
    {
      PF("failed to open msg queue\n");
      exit(1);
    }
    void *ptr = rbuf;
    ((long int *)ptr)[0] = MSG_TYPE;
    ((long int *)ptr)[1] = getpid();
    strcpy(&rbuf[2*sizeof(long int)], "hello");
    msgsnd(msqid, &rbuf, (2*sizeof(long int))+strlen(rbuf),0);
}
```

19/01/23

⊛ Shared Memory :-

⇒ This is the block of memory which is shared between multiple processes

⇒ This is the fastest ipc mechanism because the data doesn't flow from user space to Kernal space & Kernal space to user space.

⇒ Here, the data is accessed in between two processes in user space only.

⇒ The shared memory is created between freely available physical frame in between stack & heap. This entry gets updated into the page table when they attached to the process.

⇒ when we create (or) request this shared memory, Kernel creates Kernel objects which is of type struct shmid_ds.

⇒ Even this shared memory region is identified by a key from the Kernal point of view. From a process this can be accessed by shared memory identifier.

⇒ struct ipc_perm is the member of the struct shmid_ds structure.

⇒ It maintains the size of the shared memory or segment & pid of the creator & no. of current attaches and also last attached time, last detached time, last changed time & so-on.

⇒ We can access (or) request the shared memory by using shmget() system call.

⇒ we must include <sys/shm.h> header file.

⇒ int shmget (key_t key, size_t size, int flag);
↓
IPC_CREAT | 0666 → To create
0 / NULL → To access

⇒ On success, it will return shared memory id.
On Failure, it will return -1.

⇒ when the shared memory is created that means this call is succesful, the IPC_perm structure is initialized.

⇒ In shmid_ds structure, no. of attaches, attached time, deattached time are set to zero (0), last change time is updated with current time. so, the segement size is updated with the requested size.

⇒ If other process wants to access this created memory, the third (3rd) argument should be zero (or) NULL.

⇒ These entries are updated in (kernal) shmid_ds table maintained by the kernal which is of type again shmid_ds.

⇒ To access this shared memory, it should be attached to the process, this can be done by using a system call shmat()

NOTE
(*) when we use this system call, an entry is created in page table

#include <sys/shm.h>

Void * Shmat (int shmid, const void *addr, int flags);

⇒ If the flags are zero, the shared memory is attached to the process, so the Kernal will generate the address (V.A) (virtual address) & that is returned by this system call.

↳ then the 2nd argument should be NULL.

⇒ If we want to attach to the specific address, ~~the correspond.~~ that address should be passed as 2nd argument & corresponding MACRO should be used in 3rd argument.

⇒ when this call is succesful, it will return virtual address of that memory. that means the virtual address created after updating the entry in page table (or) gets updated

⇒ Once, we get the virtual address, we can update the data.

⇒ Other processes also should follow the same steps but instead of creating again they just need to access.

⇒ To detach this memory we have a system call shmdt()

```
#include <sys/shm.h>
```

⇒ int shmdt (const void *addr);

⇒ On success, it will return 0
failure, it will return -1.

⇒ To this system call, we are passing the address returned by shmat which means the page table entry is deleted

⇒ when this memory gets attached & detached, the member of shmid_ds structure i.e, no. of attachments gets increased (or) decreased.

⇒ The Effective use of this shared memory between these (2) processes there should be a synchronization mechanism to be implemented.

⇒ When 1 process accessed this shared memory, other processes should be in waiting state. To avoid this waiting state or updation issues

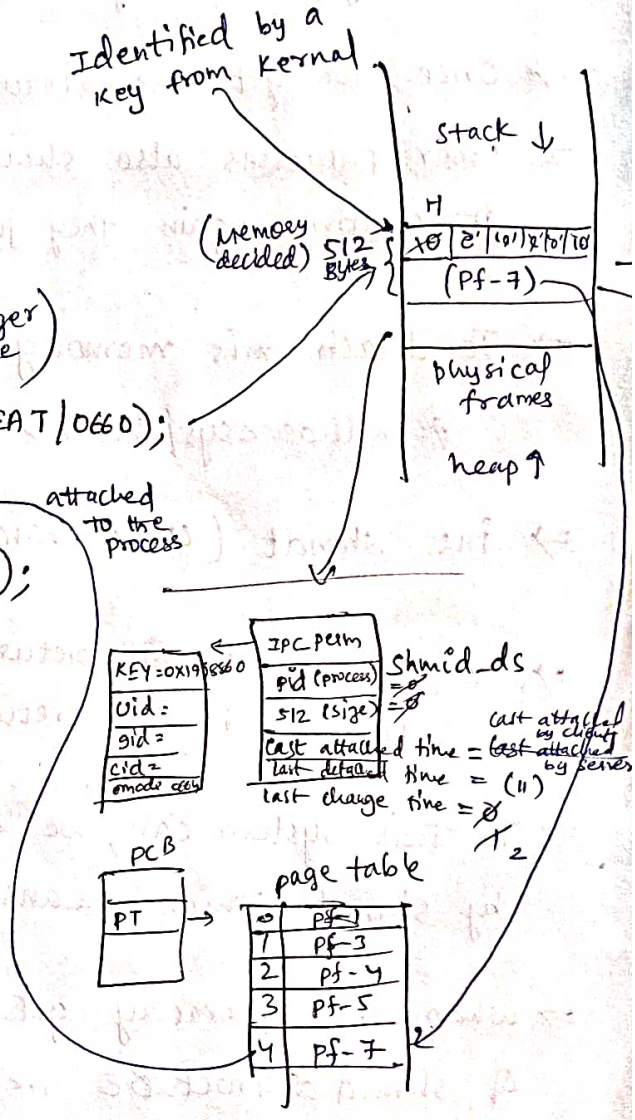
⇒ The synchronization mechanism implemented here is through Semaphoreses.

⇒ After detaching from the process, when the process terminates this memory is still alive, it is not destroyed. This can be deallocated or destroyed by using shmctl() system call or by using command IPCRM with flag & key (or) id.

Server

```
#define key 0x1998860
void main()
{
  char *ptr;
  int shmid, ret;
  shmid = shmget(KEY, 512, IPC_CREAT|0660);
  ptr = shmat(shmid, NULL, 0);
  ptr[0] = '10';
  while(ptr[0]=='10');
  Sleep(1);
  Pf("%s\n", ptr);
  ret = shmdt(ptr);
}
```

This is not the effective method of implementing synchronization (cpu does ntg simply verifies) performance of the system degrades

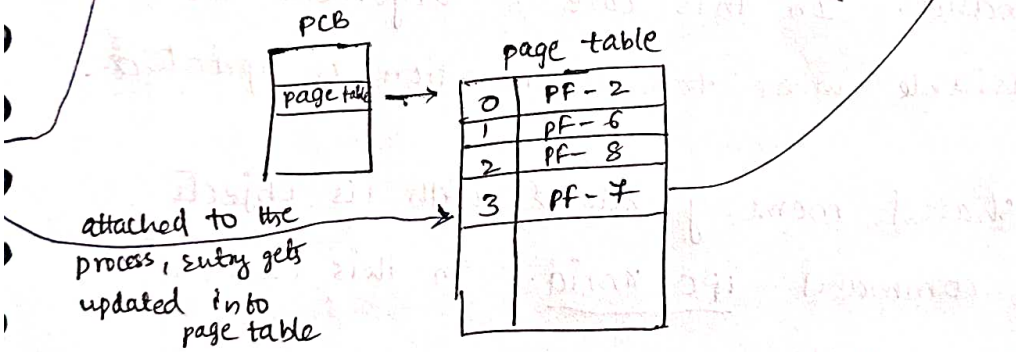


Client

```
#define KEY 0X1998860
void main()
{
  int shmid, ret;
  char *ptr;
  shmid = shmget (KEY, 512, 0);
  ptr = shmat (shmid, NULL, 0);
  strcpy (ptr, "Hello");
  ret = shmdt (ptr);
}
```

virtual address

for accessing



It is a synchronization

⇒ shmcontrol :-

⇒ #include <sys/shm.h>

⇒ int shmctl (int shmid, int cmd, struct shmid_ds *buf);

⇒ This is used to destroy (or) modify the shared memory & its objects ~~info~~

⇒ Based on command, we have to use 3rd argument.

⇒ command ipc_stat ⇒ To fetch shmid_ds structure info

⇒ we need to pass shmid_ds variable's base address.

⇒ To modify the members information, we have to use ipc_set command. In this case 3rd argument should be address of the variable where the information is updated.

⇒ To delete the shared memory along with its objects we have to use command ipc_rmid. In this case, 3rd argument should be NULL.

⊛ Semaphoreses :-

⇒ It is not an ipc mechanism, it is the synchronization mechanism used between ipc shared memory & threads.

⇒ A semaphore is a counter used to provide access to a shared data object for multiple processes.

⇒ These semaphores sets are identified by the kernal by using a structure semid_ds.

⇒ Even this semaphore sets are identified by a key.

⇒ Each semaphore set is associated with the counter.

⇒ Kernel will track these semaphore sets based on key.

⇒ From a process, these are accessed by using semaphore id's.

⇒ semid_ds structure maintains ipc_perm structure member key, last sem operation time, last change time, no. of semaphore sets & so on...

⇒ These semaphore sets can be created by using a system call semget( ). #include <sys/sem.h>

⇒ int semget (key_t key, int nsem, int flag);
(nsem: no. of semaphore sets.)

⇒ when this call is successful, it will return semaphore id On Failure, it will return -1.

⇒ when this call is succesful, struct semid_ds is created, its members are initialized with default values.

⇒ If these semaphore sets are operated on 0's & 1's they are called Binary semaphoreses

⇒ To perform the operations on these semaphore sets, we need to understand a predefined structure i.e., struct sem buf.

```
Struct sem buf
{
    unsigned short sem_num;
    Shoat  sem_op;
    Shoat  Sem_flg;
};
```

⇒ On this structure, we are going to perform semaphore operations by using semop system call.

⇒ #include <sys/sem.h>

⇒ int semop (int semid, struct sembuf semop array[],
size_t nops);

⇒ On success → 0
On failure → -1

⇒ This system call is used to perform the operations of these semaphore sets. (increment / decrement operations).

⇒ To initialize / to set the counter values & to perform the operations on semaphore sets we have to use semcontrol semctl( ) system calls.

⇒ #include <sys/sem.h>

⇒ int semctl (int semid, int semnumber, int cmnd, union semnum arg);

index of semaphore set

depends on 3rd argument

ipc_stat

It is a predefined union

Base address of variable (arg.buf)

semid_ds structure type

⇒ ipc_stat ⇒ To read semid_ds structure info
In this case, 4th argument should be arg.buf.

⇒ ipc_set ⇒ To modify the members of semid_ds

⇒ ipc_rmid ⇒ To delete the semaphore sets & objects

⇒ Union semnum

```
{
    int val;        // TO SET_VAL
    struct semid_ds *buf; // IPC_SET / IPC_STAT
    unsigned short *array; // SET_ALL / GET_ALL
};
```

~~getval~~ ⊛ GET_VAL :-

⇒ To return the value of ~~sem val~~ counter (or) for the member semnum

~~set val~~ : ⊛ SET_VAL :-

⇒ To set the value of counter / sem val for the particular Semnum

⊛ GET_ALL :-

⇒ It is used to read all the semaphore values / counters values for the semaphore set.

⊛ SET_ALL :-

⇒ To set all semaphore values / counter values

⇒ A semaphore counter value cannot be negative

For example:- If the counter value is initialized with zero (0) & decrement operation performed on it by using semop( ) system call, the counter value will become -1. The counter value cannot be negative, so, this will behaves as a blocking call. Meanwhile, client access the shared memory & updates the data. After that, until client performs the increment operation ~~initializes~~ the same semaphore counter with a value. (The operation performed) at server side.

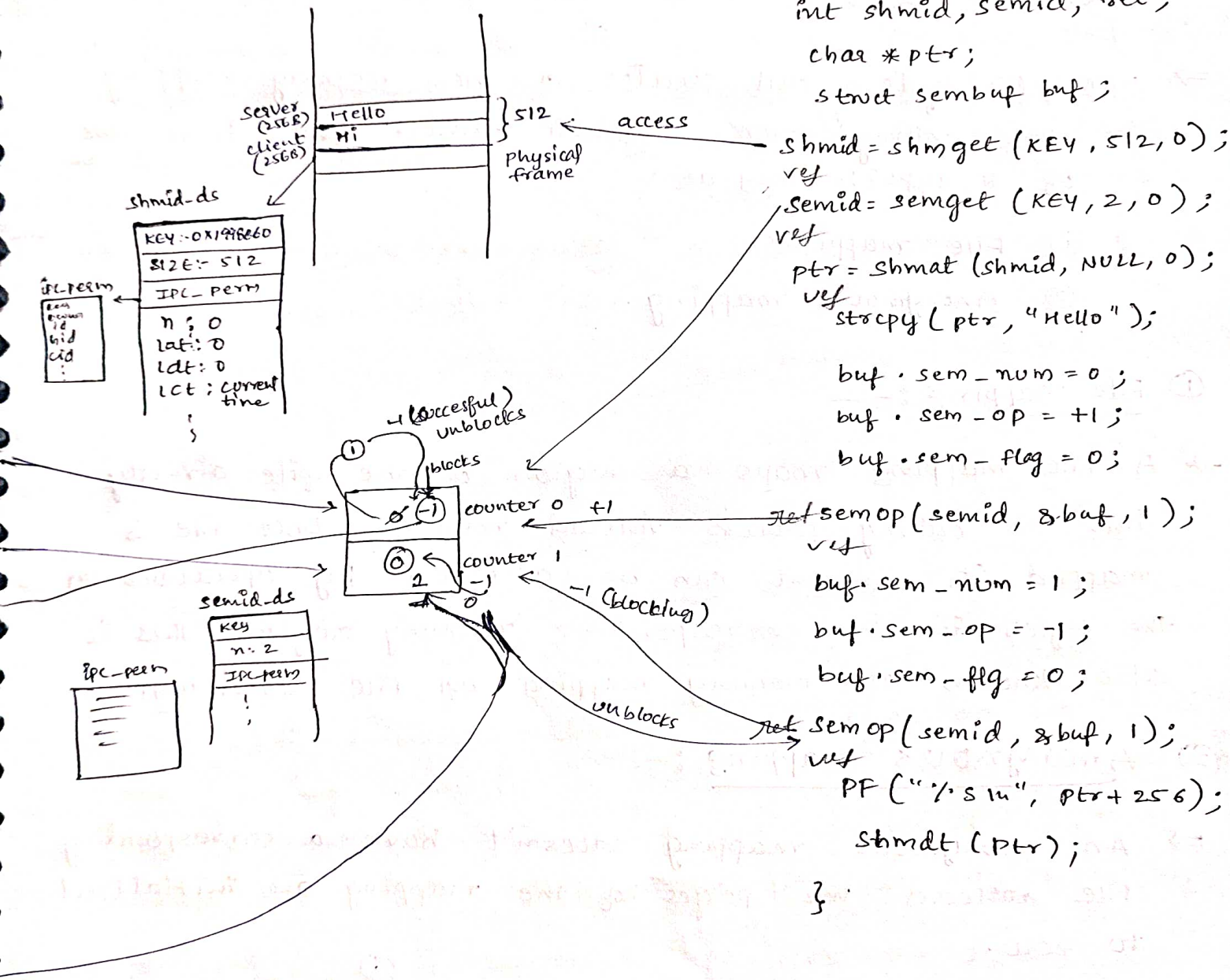
So, this makes semop() system call blocking at server comes out of blocking state by performing the decrement operation

Server

```
#define KEY 0x1998860
void main()
{
   int shmid, semid, ref;
   char *ptr;
   struct sembuf buf;
   shmid = shmget (KEY, 512, IPC_CREAT|0660);
   ref
   semid = semget ( KEY, 2, IPC_CREAT|0660);
   ref                      → NO. of semaphore sets
   ret = semctl( semid, 0, SET_VAL, 0);
   ref                    index
   ret = semctl (semid, 1, SET_VAL, 0);
   ref
    buf.sem_num = 0;
    buf.sem_op = -1;
    buf.sem_flg = 0;
                                  blocking call
  ret = semop ( semid, &buf, 1);        +1 (unblocks)
ref  integral address                   (success)
   ptr = shmat ( shmid, NULL, 0);
    pf ("%s\n", ptr);
//→  strcpy (ptr, "hi");
                256
    buf.sem_num = 1;
    buf.sem_op = +1;
    buf.sem_flg = 0;
    ret = semop(semid, &buf, 1);
    shmdt (ptr);
}
```

client

```
#define KEY 0x1998860
void main()
{
  int shmid, semid, ret;
  char *ptr;
  struct sembuf buf;
  shmid = shmget (KEY, 512, 0);
  ver
  semid = semget (KEY, 2, 0);
  ver
  ptr = shmat (shmid, NULL, 0);
  ver
    strcpy (ptr, "Hello");

    buf.sem_num = 0;
    buf.sem_op = +1;
    buf.sem_flag = 0;

  ret semop (semid, &buf, 1);
    ver
    buf.sem_num = 1;
    buf.sem_op = -1;
    buf.sem_flg = 0;
  ret semop (semid, &buf, 1);
    ver
    PF ("%s\n", ptr+256);
    shmdt (ptr);
}
```



21/01/23

✱ Memory Mappings :- (mmap)

⇒ mmap() system call creates a new memory mapping in the calling process virtual address space. There are of 2 types;- They are :-

① File mapping

② Anonymous mapping.

① File Mapping :-

⇒ A file mapping maps a region of the file directly into the calling process virtual memory. Once file is mapped, its content can be accessed by operations on the bytes in the corresponding memory region. This is also known as memory mapping or file based mapping.

② Anonymous mapping :-

⇒ An anonymous mapping doesn't have a corresponding file instead the pages of the mapping are initialized to zero.

⇒ In this mappings also we can see ① private mapping & ② shared mapping.

⇒ In private mapping MAP-PRIVATE should be used

⇒ The content modified in that memory region is not reflected in another process (or) file.

⇒ A MAP-PRIVATE mapping is sometimes referred to as a private copy on write mapping. Eg:- fork()

⇒ In shared mapping MAP_SHARED macro should be used

⇒ The modifications done in one process (or) file, they are reflected in other file (or) process. Eg:- library

⇒ Even this shared mapping is used in IPC Mechanisms.

⇒ mmap() system call creates a new mapping in the calling process virtual space.

⇒ #include <sys/mman.h> → (memory management)

⇒ void * mmap (void * addr, size_t length, int prot, int flags, int fd, off_t offset);

⇒ On success, it will return the address.
On failure, it will return the error.

⇒ The addr argument indicates the virtual address at which the mapping is to be located. If we specify addr as NULL (i.e., 1st argument as NULL). The kernel chooses the suitable address for mapping. This is the preffered way of mapping.

⇒ The 2nd argument length, it specifies the size.

⇒ 3rd argument is a bit mask, specifying the protection to be placed on the mapping. These are predefined macros

PROT_NONE ⇒ The region may not be accessed.

PROT_READ ⇒ The contents of the region can be read.

PROT_WRITE ⇒ The contents " " " " modified.

PROT_EXEC ⇒ The contents of the region can be executed.

⇒ 4th argument is flags. These are also pre defined macros

It is a bit mask of options controlling a various aspects of the mapping operation.

MAP_PRIVATE (or) MAP_SHARED.

⇒ offset, it can be zero. (6th argument).

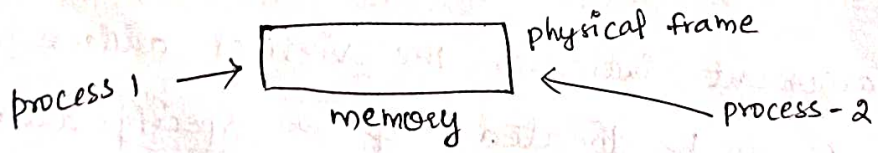
⇒ For the file mapping the 5th argument should be fd.

⇒ memory unmap → Unmapping a mapped memory region. i.e., munmap()

⇒ int munmap(void *addr, size_t length);

⇒ On success, it will return 0.

⇒ On failure, it will return -1.



mmap() ⟶ page table entry is created

MAP_PRIVATE ⟶ duplicate page is created → Ex:- fork()

MAP_SHARED ⟶ NO duplicate page → Ex:- vfork().

⇒ main()
{
int fd;
struct stat buf;
char * ptr;

(cat)

fd = open ("file.txt", O_RDWR);

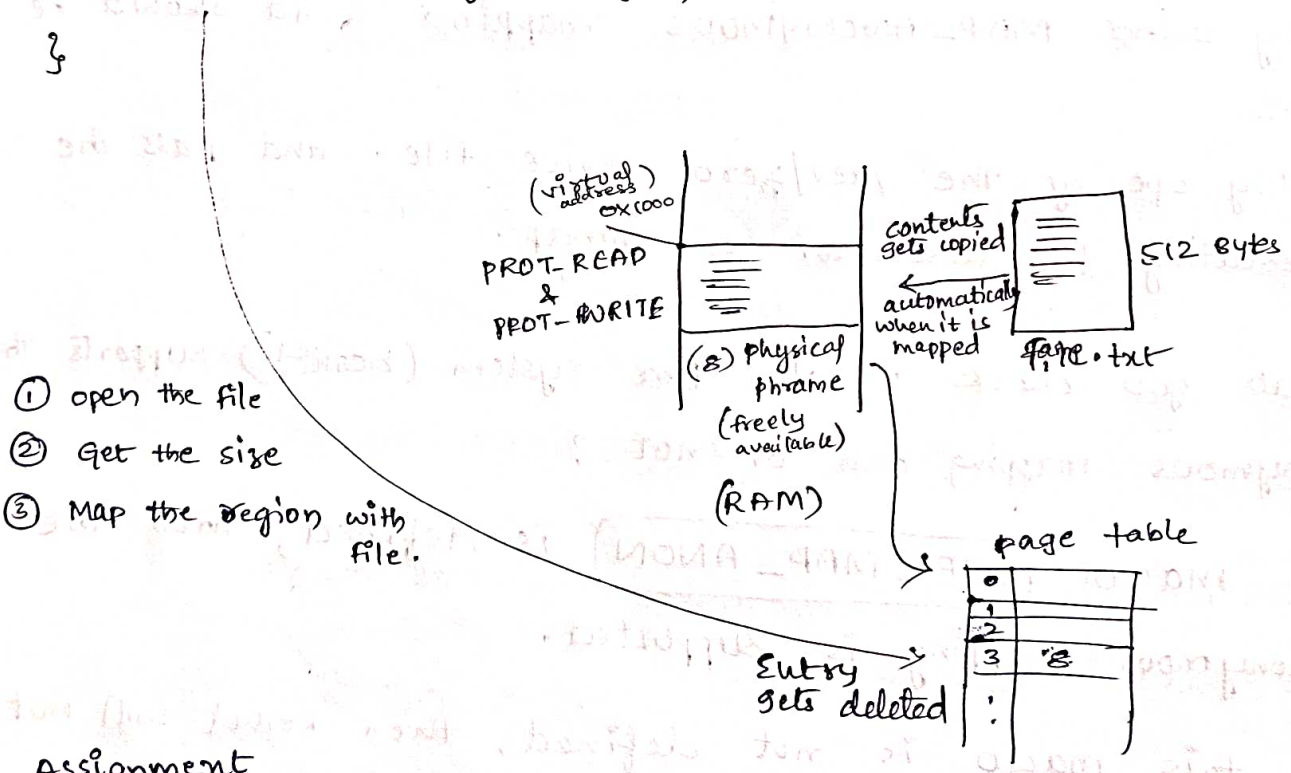
fstat (fd, &buf); ——→ reading inode object info

0x1000 (virtual add)
Ptr = mmap (NULL, buf.st_size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, 0);

write (1, ptr, buf.st_size);

munmap (ptr, buf.st_size);

}



Assignment

* Implement your own cat command by using file mapping
* " " " copy " " " " " "

Anonymous Mapping :-

⇒ A block of memory is created in the process & it is not mapped to any file & this memory is initialized to zero. This is used b/w related processes. So, parent & child can share the data by using ~~maps~~ pipes & even they can use anonymous mapping technique. So, in this case no copy on write technique is applied. So, this can be done in 2 ways. They are :-

① By using MAP_ANONYMOUS (macro) mapping & fd should be -1

② By opening the /dev/zero device file. and pass the resulting file descriptor to mmap.

⇒ How do you check whether the system (kernel) supports the anonymous mapping call or not ?

(A) ⇒ If macro USE_MAP_ANON is defined, then the anonymous mapping is supported.

⇒ If this macro is not defined, then kernel will not support anonymous mapping.

⇒ This can be implemented by using file mapping technique with NULL files. (/dev/zero, /dev/NULL) ⇒ These are called NULL files.

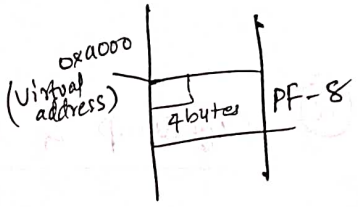
⇒ These NULL files are virtual files, they don't occupy any real memory in the Hard Disk. Whatever the data, we write into these files, they are completely lost.

⇒ whenever, we read from these files, they will return zeroe's.

⇒ If we dont want to display the output on terminal screen & if we " " the output to be saved in separate file. We must redirect the output to NULL files.

./a.out ⇒ /dev/zero (or) /dev/NULL.

Q How can we use these NULL files in our file mapping technique.



A

```
main()
{
    int *ptr;
#ifdef USE_MAP_ANON
    Ptr = mmap(NULL, sizeof(int), PROT_READ|PROT_WRITE,
                MAP_SHARED|MAP_ANONYMOUS, -1, 0);
# endif
    int fd = open("/dev/zero" (or) NULL, O_RDWR);
    Ptr = mmap(NULL, sizeof(int), PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, 0);
    close(fd);
int pid = fork();
  if(pid > 0)
  {
     sleep(2);
     PF("parent process: %d\n", *ptr);
     exit(0);
  }
Else
  {  *ptr = 4;
     PF("child process: %d\n", *ptr);
     Exit(1);
  }
}
```

✱ Tools :-

High profiling tools

① Gdb tools :- To debug the False outputs, run time errors and to understand the flow of execution of the process (applicable only for user level processes & which which have debug information)

② objdump & read elf tools :-

⇒ These are used to see the content of executable file & object file information in user understanding format by using various flags & options.

⇒ Generating 'c' code (or) source code from an executable file is known as reverse engineering. This is possible by some tools. i.e., strace, ltrace, mtrace, ptrace, dtrace & so on.

⇒ From assembly code we can't get complete 'c' statements we can generate only partial 'c' code. i.e, what are the library functions & system calls we have used in that process.

Q) what is meant by profiling of a process ?

A) ① profiling of a process is ntg but knowing the sequence of usage of system calls & library functions.

② How many times the library calls & system calls are invoked (count) & the arguments to the library function

& system calls and what are the return values.

Strace:- It is used to capture and display the info for every system call invoked by a process or application.

strace ./a.out

(strace path along with file name)

⇒ This command will displays system calls related to process, file, memory, signals, IPC mechanisms & network related calls.

⇒ If we want to display (or trace) the system calls which accepts the file name as arguments we have to use below command

strace -e trace=file ./a.out executable file with path

⇒ If we want to display (or trace) the system calls related to process management. we have to use below command.

strace -e trace=process ./a.out

⇒ If we want to display (or trace) system calls related to signals

strace -e trace=signal ./a.out

⇒ If we want to display (or trace) system calls related to ipc mechanisms.

strace -e trace=ipc ./a.out

⇒ If we want to display (or trace) system calls which accepts fd's as arguments

strace -e trace=desc ./a.out

⇒ If we want to trace (or) display system calls which are related to memory mapping (or) memory management

```
strace -e trace=memory ./a.out
```

⇒ If we want to trace (or) display system calls related to network management

```
strace -e trace=network ./a.out
```

⇒ High profiling is also known as process profiling is also known as stastics of system calls (or) functions used in ur process or application

```
strace -c ./a.out
```

⇒ To see this commands information with various flags, we can see manual pages.

```
man strace
```

ltrace :-

⇒ ltrace, tool used to display (or) trace the library functions that are invoked from your process or application

⇒ `ltrace ./a.out` ⇒ It gives complete info.

⇒ Statistics of library funcs invoked from your application (or) process can be seen by using below command.

```
ltrace -c ./a.out
```

⇒ To check the memory leaks in an executable file. we have to use valgrind tool.

⇒ To see the symbol table information we have to use nm tool.