Poll() function

"It is a system call that select devices / files from the list of available files / devices that are ready for intented operation."

→ Select() is used for checking device for read, write and error, Poll() is extended version of select which also check a device is ready for block read, normal read, character read etc.

It has more flags than select.

```
int poll(struct pollfd poll fd1[], int n,
         int timeout)
```

→ Poll fd1[] has list of specific devices to be checked, select generally take a list of devices and check.

① Struct poll fd
```
{
    int fd;
    int events; → purpose
    int revents; → response
}
```
② poll fd1[];
→ list of fds to be checked, each device / file has one poll fd.

③ int n;
Total number of devices / files

④ int timeout;
→ timeout in seconds.

Q Write a program for two files bluetooth and printer to check for read and write, will check 2 read operations and 1 write operation.

Bluetooth → Read and Write
Printer → read
Total devices = 2, operations = 3

```
#include<poll.h>
int main()
{
  char buf[1000];
  int printer_fd = open("/dev/printer",
                        O_WRONLY);
  int bt_fd = open("/dev/bluetooth",
                   O_RDWR);

Struct pollfd pollfd1[2];
  poll fd1[0].fd = printer_fd;
  poll fd1[1].events = (poll fd1[0].
        events | POLLRDNORM | POLLWRNORM);

  poll fd1[1].fd = bt_fd;
  poll fd1[1].events = (poll fd[1] |
        POLLRDNORM | POLLWRNORM);

  int nfds = poll(pollfd1, 2, 30);
int (nfds = 0)
{
```

```
    printf("No device ready \n");
}
if (nfds = -1)
  {
    perror (errno);
  }

if (nfds > 0)
{
  if(poll fd1[0].revent & POLLWRNORM)
    {
      write (pwriter_fd, "HELLO", 5);
    }
  if (poll fd1[1].revent & POLLRDNORM)
    {
      read (bt_fd , buf, 1000);
    }
  if (poll fd1[1].revent & POLLWRNORM)
    {
      write (bt_fd, "Hello", 5);
    }
}

close (pwriter_fd);
close (bt_fd);
return 0;
}
```

⇒File Representation

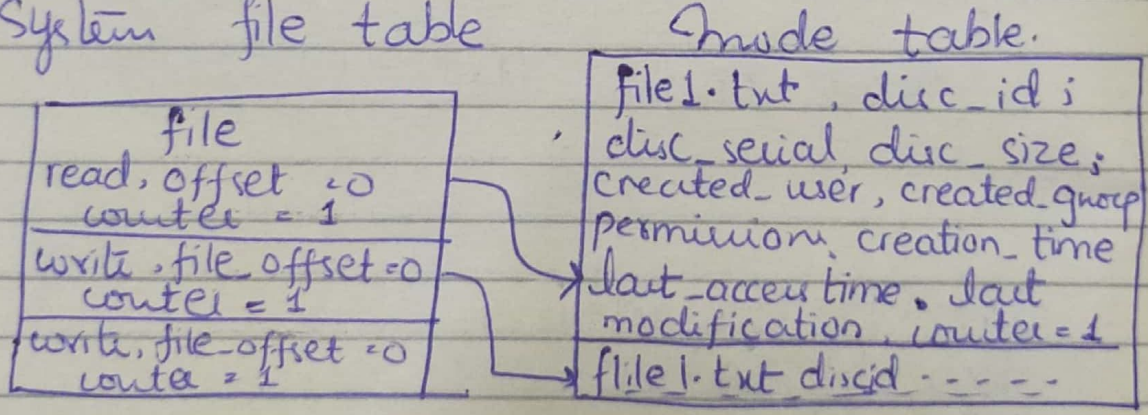
Process 1

```
int fd1 = open("file.txt", O_RDONLY);
int fd 2 = open("file.txt", O_WRONLY);
```

FDT - P1

STDIN	0
STDOUT	1
STDERR	2
fd 1	3
fd 2	4

→ Whenever a file is opened an entery is created in the system file table and file desciptor table.



file discriptor ↑ pointers

counter → tells that how many pointers are pointing towards the file.

file_offset → It is a pointer which points that how many characters have been read / write.

Process 2

int fd 3 = open ("file.txt", O_WRONLY);

FDT - P2

STDIN	0
STDOUT	1
STDERR	2
fd3	3

→ An entry is created in the system file table.
→ If there is already an entry in the inode table then no. new entry will be created. and file from the system file table points toward the same location.
→ Pointer updated
→ At a time there will be only one entry of a file in inode table.

→ When pant has opend a file and fork is called then the child has acces to that files as file discriptor table has bee copied.

→ Process 2 has created a child

FDT - Ch - P2

STDIN	0
STDOUT	1
STDERR	2
3	(fd 1)

→ Counter will be updated from 1 to 2 in fd1 and fd2 in Fd table.

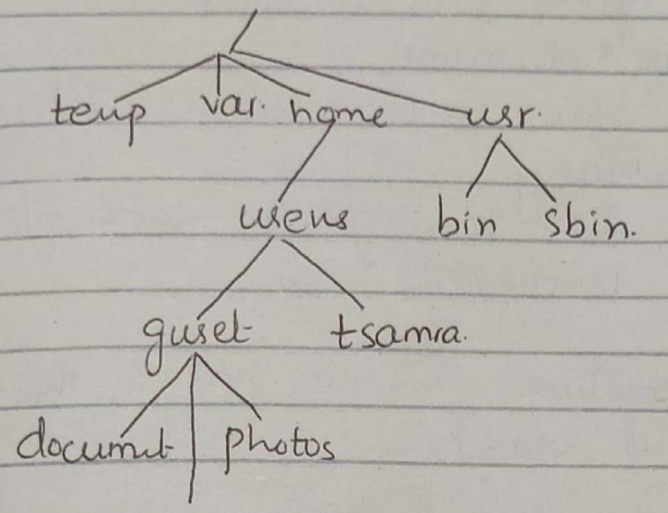
→ For each call of close system call → one entry is deleted from the "system file table" and one or more from "file descriptor table".

→ For each close system call an entry is deleted from inode table if entry is 1, otherwise counter is decremented.

Chapter #05

Files and Directories

"/" is the home directory
in this we have other directories



In unix system directory is actually a file.
In simple file we keep content but in directory file it consist of the other files stored in it.

"." is current working directory
".." is the previous directory.

→ current working directory is accessed by (pwd).

int getcwd (char* buf, int size)

↓

where to put that current working directory

int
float

→ change current working directory (cd)
chdir (char * dirname);

→ directory oper:
For this system call is opendir
→ DIR * opendir (char * dirname
For DIR return type we should include <dirent.h> library.
Now if i want to open the same file.
→ Struct dirent * readdir(DIR * dir);

dirnt in a structure and it has 3 things
→ size of file a directory
struct dirent
{ d_size // size;
d_name // name;
d_ctime // creation time
}
int close dir (DIR * dir)

⇒ Program to implement ls - command
report number of sub-directories count

Steps
① create/declare DIR and dirent pointers
② open the given directory
③ read the ~~given~~ from directory and list until we reach NULL
④ For each entry list and find its type
⑤ if type is directory update count.
⑥ Print ... t.

close directory.

Program

```
#include <dirent.h>
int main(int argc, char *argv[])
 DIR *dir ;   // point ①
 struct dirent *dirp;   // point ①
 int dcount=0;

if(dir = opendir(argv[1]) == NULL)   // point ②
   { perror(" Error in opening of dir");
     exit(0); }

else
   while((dirp = readdir(dir))!= NULL)
   {
      printf("%s", dirp->d_name);
       if(isDirectory(dirp->d_name))
          dcount++;          // if it return true
                                update counter.
   }

printf("Number of subdirectories : %d\n",
                              dcount);
closedir(dir);
 return 0;
}
```

which command —?
echo command —?

File State Information

If we know the name of file or its hard link or we had opened it and we had its file discriptor and want to extract the information of that file in inode file, so we will we certain system calls() for it.

e.g which device, its size, its user etc type of information

the systemcalls used for this are:

stat(char* path, struct stat *buf)

lstat(char* link_path, struct stat *buf)

fstat(int fd, struct stat* buf) // we pass the file discriptore

the return type of all the three is integer. if failed returns -1.

```
Struct stat {
    dev_t st_dev;       // device id
    ino_t st_ino;       // what is inode number of file.
    mode_t st_mode;     // permissions
    uid_t st_uid;       // created by user.
    gid_t st_gid;       // user group
    off_t st_off        // size of file.
    nlink_t st_nlink    // hard links
    time_t st_atime.    // last access time
```

```
time_t st_ctime;        // correct time
time_t st_mtime;        // modification time.
}
```

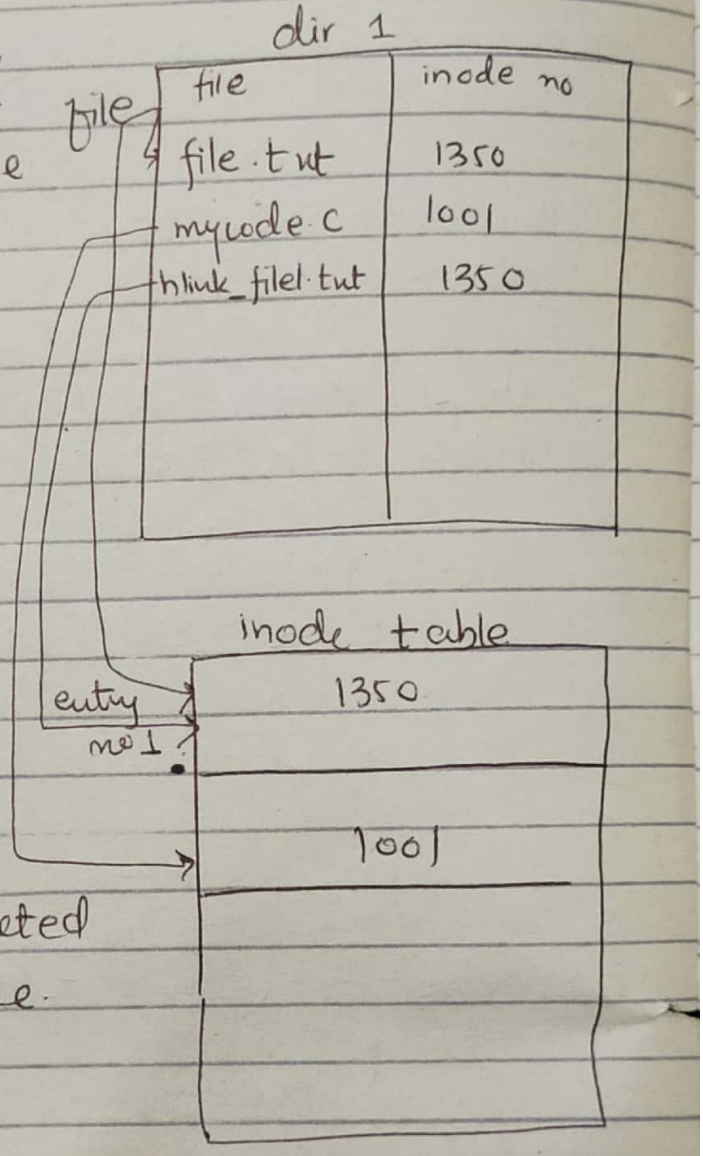
=> Program to find that which of the two files is ~~greater~~ larger.

```
#include <sys/stat.h>
int main()
{
struct stat *file1buf=(stat*) malloc (sizeof(stat));
struct stat * file2_buf=(Stat*) malloc ( sizeof (stat));

Stat("file1.txt", file1_buf);
stat ("file 2_txt", file 2_buf);

if (file1_buf -> st_size > file2_buf -> st_size)

    printf ("file1 is larger");

else if (file2_buf -> st_size > file1_buf -> st_size)
    {
      printf ("file 2 is larger");

  else
      printf ("Both are equal size");
  return 0;
}.
```

⇒ In linux we had both soft-link and hard-link.

relation with inode table:

hard-link points to the inode number of that file
soft-link points to the name of that file.



if i change the name of file the hard-link to it will not break but the soft link to it will break.

if we delete a file it not actually deleted from the disk only its data is deleted from the inode-table.

hard link

int link (char* path1, char * path2);

path1 ↓: original file name & path.

path2 ↓: name and path of hard link to be created.

int symlink (char *path1, char *path2);

pointer to the

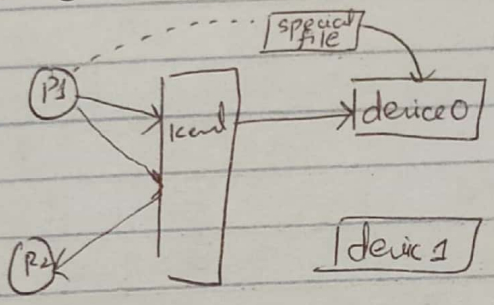
int unlink (char * path);

x ——— x ——— x ——— x ——— x

Chapter # 06

~~Fifo~~ Pipes (Interprocess communication)

but we have two process P1 and P2 and two devices d0 and d1. Now these process can access through kernel



P1 access to device0 and open it, when we open a device actually a special file is opened.

Now if the two process wants to communicate through each other e.g to send data it will be done through interprocess communication using kernel

One method is two used socket method in this we must have a network card on our computer.

2nd method is to use shared libraries.

Another method used by UNIX is "pipes".

In this type if two process wants to communicate kernel creates a "special file" one process read to it and other write to it in this way data is shared.

But this shared file is always created by kernel so the process should send a request through a system call.

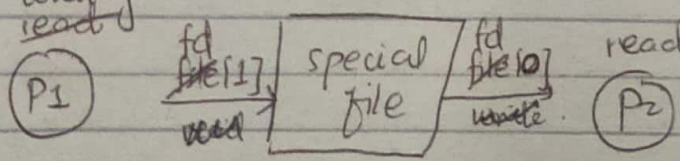
int pipe(int fd[]) → // sending request to kernel that i want to get a pipe IPC.

As a result of this call. kernel creates a special file

Now how reading writing will be done in special file

so the

int fd[] array of size 2 will be used to read and write a data by making handles



Disadvantage:

the disadvantage is that we cannot know its name so we cannot use it for another process except for the child process.

Program to create a process which will create a pipe will write the data and then will create a child process and the child will display the parent data.

Points

→ create int array size 2.
→ create pipe
→ write "Hello" via fd[1] to the pipe.
→ create child process.
→ Child process read data from pipe via via fd[0].
→ child process display the read data.

library used is unistd.

```
# include <unistd.h>
int main()
{
  int fd[2];
  int pipe(~~int~~ ~~fd[2]~~ Fd);

  write(fd[1], "Hello from your paret process", 29);
  int pid = fork();
  char * buf[29]; // to read the data in buffer.
  read (fd[0], buf, 29)
  printf ("%s", buf);
  return 0;
```

Another System Call used for IPC

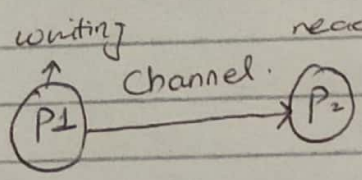
FIFOs (named pipes)

mkfifo(char* pipename, int mode);

loggers are the separate applications in the system and we can communicate through it using mkfifo.

For using this channels (special files) will be created one for reading and one for writing.
On previous the special files were opened by the kernel.
But in this we had to open them.

Program to write so that two process communicate using fifos.



For P1.

```
#include<unistd.h>
int main()
{
    int mkfifo("chat1", S_IRUSR|S_IWUSR);
    int fd = open("chat1", O_WRONLY);
    write(fd, "Hello", 5);
    write(fd, "How are u doing", 12);
    return 0;
}
```

For P_2 reading

```
#include <unistd.h>
int main().
   int fd = open("chat1", O_RDONLY);

   char buf[50];
   read(fd, buf, 50);
   printf("%s", buf);
    return 0;
   }
```

1 ROTHER

* Pipelines:
lets you have on terminal and you run commands.
lets from a file i want to know that how many are are there in a file.
So we will use.

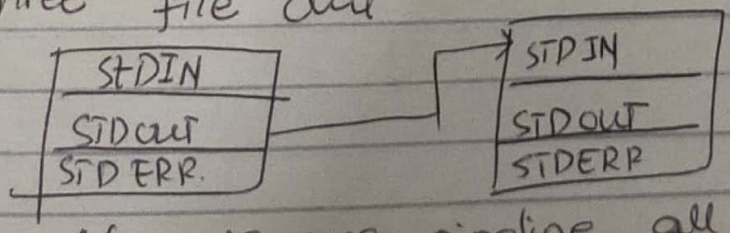
$ grep error output
but this command is not enough we want to know the number of lines to be created.

(P_1) (P_2)

$ grep error output | wc -l

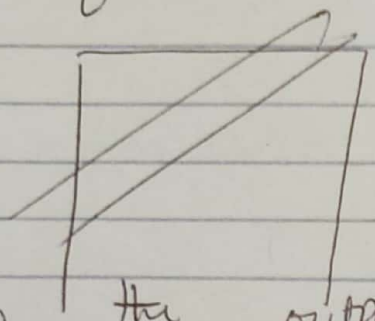
this is pipeline.

for every process there are atleast three file dis



but if we use pipeline all the data

and commands of P1 will be transferred to P2.

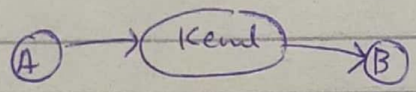


so the output of one process will become the input of another process.

Chapter # 08
Signals

Since them are interrupts that are alway directed towards the CPU.
But signals are thes short notification or messages send to the Process as a result of an event.
They are genrally genrated from one process to another but kernl is involved in it.
e.g sleep(10)
send a signal to kernl to suspend the Process of a process for 10 seconds.

signal from A to B throug kernl



SIGABRT → About the process → implementation dependent.
SIGBUS → Acess invalid memory → implementation dependent.
SIGFPE → Division by zero → "
SIGKILL → Kill the process → not implemetation dependent
SIGALRM → we pass time in it, when the time is complemented it kill the process.
SIGCHLD → on child tenination → implementation dependent.
SIGINT → Abort → implementation dependent.
SIGUSR1 → unused
SIGUSR2 → unused.
SIGSTOR → stop execution → not implementation dependent.

How signals are generated with shortcuts like ctrl+z, ctrl+c is is told by a command.

$ stty -a → list of signals that are generated with shortcuts.

$ kill -l (list) → gives you the list of all signals generate.

⇒ How to send a signal from terminal to a process:

$ kill -s signal name (e.g SIGABRT) pid (0034) → send signal to process.

A process can receive a signal only if it is in execution state on the CPU.
if the process is not in execution state the signal should wait for it.

A signal is delivered only when the process is in execution.
But the signal delivered will be in ordered like the signal delivered first will be receive by the process first.

⇒ If a process wants to send signal to another process.

int kill(pid_t pid, int sig_no); → defination
kill(3256, SIGINT); call.

kill(getppid(), SIGKILL);

⇒ If a process wants to sent signal to itself:

int raise(int sig_no);

raise(SIGKILL);

⇒ If we want to delay a signal this is known as 'blocking signal'.

if we want to block a signal we had to made a list of these signal which we want to ~~set~~ block.

sigset_t set;

⇒ int sigaddset(sigset_t * set, int signo);

⇒ if we want to deletete a signal from set.

↓

⇒ int sigdelset(sigset_t * set, int sig_no);

⇒ ~~int emptyset~~ int sigemptyset(sigset_t * set);

⇒ int sigfullset(sigset_t * set);

⇒ int sigismember(sigset_t * set, int sig_no);

⇒ header file for all of them is <signal.h>.

Now if we want to do the above tasks with the system call.

sigprocmask(int how, sigset_t * newset, sigset_t * oldset)

↓ syscall used to block or unblock signals or setup signals

↓ 3 options SIG_BLOCK SIG_UNBLOCK SIG_MASK.

↓ If you block a signal the above argument to us the signals that were blocked previously

Lecture

Handeling of Signals

sigaction(int sig_no, struct sigaction* act, struct sigaction *oact)

sigaction* act → action to be applied on the receival of the signal.

There is a signal and you want to say the signal that i want to change the default action of that signal.

sigaction*oact → old action of the signal.

We use that oact that if we restore the signal action back, we should have it.

```
struct sigaction{
    void (sa_handler*)int; // handler 1
    sigset * sa_mask;
    int sa_Flags.
```

void (sa_sigaction_handler)(int, siginfo_t *, void)
// handler 2.
Real time.
}

Function of all these.

① Void (sa_handler*) int // normal POSIX ~~define~~ Function
the handler you passed in this will be given as an action to the signal.
e.g
DEL // give default action.
IGN // ignore signal.
handler function // user define function which we pass in it.

② sig set *samask:
// signals to be blocked during process
i-e while handling some signals we want some other coming signals to be blocked, so we will use it.

③ int sa_Flags:
// select handler function.
Here we have two type of handles by passing 0 and 1 in flags we can select either of them.
about ✗
SIGFPE → division by zero
passing this signal need the signal to be aborted so we we want to change its default action.

```
#include<signal.R>
int main()
{
    struct sigaction act, oact;
    sigset FPE_mask,
    // lets we want to block two
        4 signals so we will add them
        // to set
    sigaddset(&FPE_mask, SIGINT);
    sigaddset(&FPE_mask SIGUSR1);
    act.sa_handler = FPE_signal_handler;
    act.sa_mask = FPE_mask;
    act.sa_flags = 0;

    sigaction(SIGPE, &act, &oact)
      return 0;
}

// function:
  void FPE_signal_handler(int sig_no)
  {
  printf("Division by zero was attempted. I am
          avoiding the about power my
          custom handler.

  }
```

Waiting For a Signal

e.g We use sleeps to sleeps a signal for a certain time.

int sleep(int seconds);

→ int pause(void); ⟶ suspend the process, until a signal is received.

The signal received (ta k hm ko nend se jaga) we dont know its number.

so what we do is:

→ int sigsuspend(sigset * set1);

We will a set of 3 signals jo k hmen jagen gy.

→ int sigwait(sigset * set1, int * sig_no)

we will use a set of signal but we will also have its number.

Chapter # 09

Times and Timers

Timers is a running entity, it increments.
e.g we want to note the time b/w two events or we want to know that how much time a process take:
<time.h>
lets use a syscall:

① time_t time(time_t * t1) → return the time elapsed since epoch.

⇒ epoch is actually the time after which we want to know that how much seconds passed.
e.g Jan 01, 1970, 12AM. UTC.

② double difftime(time_t t1, time_t t2);
it gives the time difference b/w two functions or process
int is of 4 byte
long int is of 8 byte.
double is of 10 byte.

The issue here is that the above function give us time in seconds. with no year, no day,

③ cha* ctime(time_t t1)
we passed the time returned by syscall 1 to it and it return the time into string format.

A better syscall than this is

struct tm* localtime (time_t t1);
convert time into tm structure

```
Struct tm{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_year;
    int tm_mday;
    int tm_month;
    int tm_wday;
    int tm_yday;
    int tm_isdst; → is day light saving
                    time on or off.
}
```

wday means e.g since monday how many days passed.

④ struct tm* gmtime(time_t t1);
// convert time into structure (UTC).

⑤ char* asctime (struct time.* tm1);

// convert time from from tm into string.

CPU can execute 1 billion instruction at one second, so if the process takes time less than 1 second so the above ~~instruc~~ syscall return 0.
To solve this problem, we have another syscall

↳ this is for future use

⑥ int gettimeofday(struct timeval* tv, void* tzp);

```
struct timeval {
    time_t  tv_sec;
    time_t  tv_usec;
};
```

returns time since epoch in seconds and microseconds.

⇒ To know about the execution time of two Processes

```
int main()
{
    time_t t1, struct timeval *t1, t2;
    int gettimeofday(tv1, NULL);
    // function whose time we want to know
    int fact = factorial(15);
    gettimeofday(tv2, NULL);

    long int execution_time_in_sec =
        (tv2->tv_sec - tv2->tv_sec)*
        1000000 + (tv2->usec - tv2->tv_usec)
                    tv_
```

upto now we are taking time since epoch it was giving the time of execution. When we run process on the CPU. For sometime it run on the CPU and for sometime it is removed from the CPU. So the time of execution of a function differs from computer to computer. Now how much time a process have taken on CPU is found using syscall

clock_t time (struct tms * t1);

```
struct tms {
    clock_t  tms_utime;  // ① user process time spent of user code
    clock_t  tms_stime;  // ② system time on behalf of user process
    clock_t  tms_cutime; // user process time + terminated child process time
    clock_t  tms_cstime; // system time co behalf of process and terminated childrn
};
```

So when we use this the ① and ② will tell that cpu will tell that this much time is take by your code and this much time it had worked from you.

In system we also have

⇒REAL TIME CLOCK:

This starts when our system starts and this given the time since epoc in seconds and nano seconds. it has three system calls.

① int clock_getres(clockid_t clockid, Struct timespec* res);
② int clock_gettime(clockid_t clockid, struct timespec* res);
③ int clock_settime(clockid_t clockid, struct timespec* res);

```
Struct timespec{
    time_t  tv_sec
    long    tv_sec;
}
```

⇒ gettime() gives us the time from which the system has been started. i-e

⇒ getres() through this we reset the clock. Usually the clock start itself and then increment it but the user can reset it according to its needs.

⇒ clock_settime is used e.g i tell the system that i dont know from which time clock is started, but i give a time in clock_settime() function, which state that start the clock from that given time but for this we also need a root user.

Now we will go to two other calls

① int sleep(time_t t);
the time given in it will make a signal sleep upto that time.

② int nanosleep(struct timespec *tm, struct timespec * rtm).

It is a modified version of sleep.
it can take the time in seconds as well as in nano-seconds. and rtm is the resolution time.

Timers:-
one ~~constant~~ concept is of stopwatch i-e we send it certain time.

→int getitimer(int which, struct itimerval * value);

→int setitimer(int which, struct itimerval * value, struct itimerval * oval)

```
Struct itimerval
{
   Struct timeval value;
   struct timval interval;
   }
```

There are three type of timers.
① ITIMER_REAL; // decrements according to real time clock and generate SIGALRM when timer expires.
ITIMER_VIRTUAL; // decrements according to process time in CPU and generate SIGVTALRM when Timer expires.
ITIMER_PROF // decrements according to process time and kernel time and generate SIGPROF when timer expires.

the int which parameter in the above function take any of these three timers that which timer do you want.

```
struct time val
    {  time_t  tv_sec;
       time_t  tv_usec;
    }
```

then if you given it a counter time you will pass struct timeval *value;

⇒ then it asks that if this Timer expires what to do?
There will be two possibities either to send the alaram and then the clock time will take some further time after that again ALAR will ring.
For this we use struct timerval intend (interval for reset).

Write a C system Program that creates a child process everytime after 30 sec.

```
#include <time.h>
int main()
{
  struct sigaction * act
  sigset mask
  sig struct itimerval;
     struct timeval;

  act.sa_handler = sigvthandler();
  sigaddset(&mask, SIGUSR1);
  act.sa_mask = mask;
  act.sa_flag = 0

  sigaction(SIGVTALRM, &act, NULL)
     it.value.tv_sec = 30;
     it.value.tv_usec = 0;
     it.interval.tv_sec = 30;
     it.interval.tv_usec = 0;
  setitime(ITIMER_VIRTUAL, it, NULL);
  return 0; }
void sigvthand (int sig_no)
{
    fork();
}
```