// ------------------------System file io------------------------

open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int fd;
fd=open("filename",O_RDONLY | O_WRONLY );
fd=open("filename",O_RDONLY | O_WRONLY | O_CREAT,0666);
if ( fd == -1 )    perror("open filename");   /* error */
```

read()

```
#include <unistd.h>
char buffer;      //  char buffer[20] → read(fd,buf,fer1)
read(fd,&buffer,1)  // read(file descriptor,buffer,nbytes) – reads nbytes bytes from the file w/ file
descriptor and stores into buffer   /* <0 error */
```

write()

```
#include <unistd.h>
write(fd,&buffer,1) // writes up to count bytes from buffer to fd /* <0 error */
                                        close(fd);     #include <unistd.h>
```

stat()   - get file status

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
main() {
      struct stat stats;
      stat("FilenameOrPath",&stats);  /* 0 success , -1 error */
      if (stats.st_mode & R_OK/W_OK/X_OK)       printf("read/write/execut.");
            File owner: %d              stats.st_uid
            File group: %d              stats.st_gid
            File size: %ld              stats.st_size
            ST_MODE VALUE: %ld   stats.st_mode

#include <time.h>
 int res;     struct stat stats;     res=stat(argv[1],&sBuf);
 printf("st_mtime: %ld    st_ctime: %ld  st_atime: %ld\n",stats.st_mtime,stats.st_ctime,stats.st_atime);
      // st_atime: time of last access
      // st_mtime: time of last data modification
      // st_ctime: time of last file status change

             struct tm tmp;
            tmp = *(gmtime(&stats.st_atime));
            printf("Last access time: %d-%d %d %d:%d:%d\n",
             tmp.tm_mday, tmp.tm_mon, tmp.tm_year + 1900,
             tmp.tm_hour, tmp.tm_min, tmp.tm_sec);
            tmp = *(gmtime(&stats.st_ctime));
            printf("Last change time: … )
            tmp = *(gmtime(&stats.st_mtime));
            printf("Last modification time: …)
```

// -------------------------Directories-------------------------

DIR- represents directory stream
struct dirent – members are d_name[] , d_ino

```
#include <sys/types.h>
#include <dirent.h>
main() {
    DIR *d;
    struct dirent *dir;
    d=opendir(“.”);     // current directory , also path
    if(d) {
        while( dir=readdir(d) != NULL ) {     /* list files in directory */
           if ( dir→d_name[0] != ‘.’ )    %ld    %s   dir→d_ino , dir→d_name  /* no hidden files */
        }
    closedir(d);
   }     }
```

// -------------------- -TIME- -----------------

```
#include <locale.h>
#include <time.h>
        char buffer[80];
        time_t currtime;
        struct tm *timer;
        time( &currtime );
        timer = localtime( &currtime );
        strftime(buffer,80,"%c", timer );
        printf("Date is: %s\n", buffer);
        // printf("Date is: %s\n",asctime(timer));
                                                      Date is: Thu Dec 20 09:12:07 2018
        printf("Locale is: %s\n", setlocale(LC_ALL, "en_US.UTF-8"));
        strftime(buffer,80,"%c", timer );
        printf("Date is: %s\n", buffer);
                                                 Date is: Thu 20 Dec 2018 09:12:07 AM +04

        struct lconv *lc;
        lc=localeconv();
        printf ("Currency symbol is: %s\n-\n",lc→currency_symbol);        Currency symbol is: $
```

//   type of file - Directory or Ordinary File

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
void printType(mode_t mode) {
    if ( S_ISDIR(mode) ) {
        printf("Directory\n");
        return;
    }
     if ( S_ISREG(mode) ) {
         printf("Ordinary File\n");
         return;
     }
```

```
int main () {

    struct stat stats;

    stat("foo.txt",&stats);

    // stat("../pw5",&stats);

    printType(stats.st_mode);

}
```

// ------------- Splits into array of strings and which to locate a command.------------------------

```
#include<string.h>     #include <dirent.h>
      printf("PATH: %s\n",getenv("PATH"));  // "ROOT" , "HOME"
                              /* splits the PATH into arrays */
      char *buffer=getenv("PATH");     char *s=":";  char *token=strtok(buffer,s);
      char *array[20];        int i=0;
      while ( token != NULL ) {
             array[i]=token;  // po odnomu zapisivayu v array
             token=strtok(NULL,s);  // going to next
             i++;
      }
                               /* which ls -> /bin/ls */
      int length=strlen(buffer)+1;    char *command="ls";      int count = 0;
      while ( count < length ) {
           DIR *d = opendir(array[count]);       struct dirent *dir;
            if (d != NULL) {
               while ((dir = readdir(d)) != NULL) {
                  if (!strcmp(dir->d_name, command)) {
                     printf("\n%s is found in %s\n", command, array[count]);
                     exit(0);
                  }
            }
           closedir(d);
         }
        ++count;
     }
```

```
for(int i=0;i<19;i++) {

             printf("%s\n",array[i]);

      }
```

// ------------- Returns path name (ttyname) ------------------------

```
char *my_ttyname(int fd) {
    struct stat st;
    fstat(fd, &st);  // info about a file

    DIR *d = opendir("/dev/pts/");
    struct dirent *dir;

    if ( d == NULL ) {
          fprintf(stderr,"Directory does not exist\n");
          exit(2);
    }
    else {
        while ( (dir = readdir(d)) != NULL) {
            if ( dir->d_ino == st.st_ino ) {
                char *name = (char*) calloc(15, sizeof(char));
                strcpy(name , "/dev/pts/");
                strcat(name , dir->d_name);
                return name;
            }
        }
    }
```

```
int main() {

     printf("%s\n", my_ttyname(STDIN_FILENO));

// 0

   printf("%s\n", my_ttyname(STDOUT_FILENO));

 // 1

  printf("%s\n", my_ttyname(STDERR_FILENO));

 // 2

     return 0;

}
```

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
                        /* generate a child process using the system call fork */
int main () {
    int pid=fork();     /* child process generated  */
    int why;
    switch(pid) {
      case -1:          /* unsuccesful creation of process */
         perror("fork() error");       exit(-1);
      case 0:           /*   child process (=0)  */
         printf("child %d: parent %d\n",getpid(),getppid());         exit(getpid()%10);
      default:          /*   parent process (>0)   */
         printf("parent %d: child %d\n",getpid(),pid);
         wait(&why);          /* waiting for child process to terminate */
         // printf("Return code: %d\n", wait(&why));
         if ( WIFEXITED(why) )      printf("exitcode %d\n",WEXITSTATUS(why));
         if ( WIFSIGNALED(why) )   printf("stopped by signal %d",WTERMSIG(why));
     }
    return 0;
}
```

/* launch *n-child* processes */

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    int n;       printf("Enter n: ");     scanf("%d",&n);
    int pid;     int why;
    for(int i=0;i<n;i++) {      /* for n-child proccesed */
        pid=fork();
        if ( pid == -1 )   {    fprintf(stderr,"Error\n");     exit(-1);    }
        if ( pid == 0)     {
            printf("Child (%d) : %d\t",i,getpid());
            exit(i);      /* if you don't -> mnogo parents */
        }
        else {
            wait(&why);           /* child to terminate */
            printf("Parent : %d\t",getpid());
            if (WIFEXITED(why)) printf("Return value: %d\n", WEXITSTATUS(why));
        }
    }
}
```

```
Enter n: 3
Child (0) : 4493      Parent : 4492   Return value: 0
Child (1) : 4494      Parent : 4492   Return value: 1
Child (2) : 4495      Parent : 4492   Return value: 2
```

```
          /* parent and child read characters one at a time using read and display them on the screen */
#include <sys/types.h>   #include <sys/stat.h>    #include <fcntl.h>     #include <unistd.h>
int main(int argc,char *argv[]) {
          if ( argc != 2 ) {   printf("Wrong number of arguments\n");    exit(1);      }
          int fd;
          char buf;
          fd=open(argv[1],O_RDONLY); // argv[1] = filename
          int pid=fork();
          if (pid == -1) {     fprintf(stderr, "Error\n");     exit(-1);      }
          if ( pid == 0 ) {
                    while( (read(fd,&buf,1) > 0 )) {
                    printf("Child reads: [ %c ]\n",buf);
                    sleep(1);
                    }     }
          else {
                    while( (read(fd,&buf,1) > 0 )) {
                    printf("Parent reads: [ %c ]\n",buf);
                    sleep(1);
                    }      }
close(fd);
}
```

```
/*** another way parent and child

          ssize_t read_n = 0;

          while (1) {

          char c;

          read_n = read(fd, &c, 1);

          if (read_n == 0) break;

          printf("%c", c);

          }
***/
```

/** displays the system time, using seconds and micro-seconds (system call: **gettimeofday**) ;
exec(ls -l directory)
display the time (as before) and the time the command **ls took to run **/**.

```
int main(int argc,char *argv[]) {
          if ( argc != 2 ) {  }

          struct dirent *dir;    DIR *d;     d=opendir(argv[1]);    if ( d == NULL ) {  }

          struct timeval tv1,tv2;
          if ( gettimeofday(&tv1, NULL) == 0 ) { // success , -1 failure
                    printf("Seconds: %ld\n",tv1.tv_sec); // seconds
                    printf("Micro-seconds: %ld\n",tv1.tv_usec); // micro-seconds
          }
….
case 0:
          execlp("ls","ls","-l",argv[1],NULL);
          exit(0);
default:
          wait(&why);
          if(WIFEXITED(why))      printf("parent : exit code is %d\n", WEXITSTATUS(why));
          gettimeofday(&tv2, NULL);
          printf("Seconds: %ld   Micro-seconds: %ld   tv2.tv_sec,   tv2.tv_usec); // sec + microsec
          printf("\nInterval: %ld , %ld   ,tv2.tv_sec-tv1.tv_sec,      tv2.tv_usec-tv1.tv_usec);
          }
}
```

/* ls with the option -R on the directory; ;
redirects the standard output of ls to /dev/null ;
displays the sum of each processor time used by the ls command in seconds (primitive times).**/.

```
int main(int argc,char *argv[]) {
        int pid1=fork();
        int why;                        struct tms buf;
        switch(pid1) {
                                        int hz;
                case -1:    fprintf(stderr,"fork() error");    exit(-1);
                case 0:
                        execlp("ls","ls","-R",argv[1],NULL);
                        exit(0);
                default:
                        wait(&why);
                if ( WIFEXITED(why) ) printf("exitcode %d\n",WEXITSTATUS(why));
                if ( WIFSIGNALED(why) ) printf("stopped by signal %d",WTERMSIG(why));
                int pid2=fork();
                switch(pid2) {
                        case -1:   fprintf(stderr,"fork() error");     exit(-1);
                        case 0:
                        execlp("ls","ls > /dev/null",argv[1],NULL);
                        exit(0);
                default:
                        wait(&why);
                if ( WIFEXITED(why) ) printf("exitcode %d\n",WEXITSTATUS(why));
                if ( WIFSIGNALED(why) ) printf("stopped by signal %d",WTERMSIG(why));
                times(&buf);
                hz=sysconf(_SC_CLK_TCK);
        printf("Time spent: %f %f\n",(double)buf.tms_cutime/hz,
                                        (double)buf.tms_cstime/hz);
```

the parent reads data from the standard input and transmit them to its child through a pipe, the child then prints them on the standard output. The parent then waits for the termination of the child process.

```
int main(int argc, char *argv[]) {
        int tube[2];
if ( pipe(tube) != 0 ) {
        fprintf(stderr,"Pipe failed\n");
        exit(1);
}
int pid=fork();                         void copy(int fdsrc,int fddst) {
if ( pid == -1 ) { }
if ( pid == 0 ) {   // child                    int length;
        close(tube[1]);
        copy(tube[0],1);                        char buf;
        close(tube[0]);
        exit(0);                          while((length=read(fdsrc,&buf,1)) > 0 ) {
}
else { // parent                                        write(fddst,&buf,1);
close(tube[0]);       copy(0, tube[1]);          close(tube[1]); }              }

                                        }
```

ps eaux > foo ; grep "^$1 " < foo > /dev/null && echo "$1 is connected"

```
int pid1 = fork();
switch (pid1) {
case -1:
case 0: // child process
        int pid2 = fork();
        switch (pid2) {
                case -1:
        case 0: // child process - ps eaux > foo
                int fd = open("foo", O_WRONLY | O_CREAT | O_TRUNC, 0666);
                dup2(fd, 1);
                close(fd);
                execlp("ps", "ps", "eaux", NULL);
                exit(2);
        default: // parent process
                wait(NULL);
            }

        int pid3 = fork();
        switch (pid3) {
        case -1:
                fprintf(stderr, "Something went wrong\n");
                exit(-1);
        case 0: // child process - grep "^$1 " < foo > /dev/null
                int fd1 = open("foo", O_RDONLY);
                int fd2 = open("/dev/null", O_WRONLY | O_CREAT | O_TRUNC, 0666);
                dup2(fd1, 0);
                close(fd1);
                dup2(fd2, 1);
                close(fd2);
                char *t=malloc(strlen(argv[1])+3);
                sprintf(t,"^%s",argv[1]);
                execlp("grep", "grep", t, NULL);
                exit(0);
        default:     // parent - echo "$1 is connected"
                execlp("echo", "echo", argv[1], " is connected", NULL);
        }
        default:
                wait(NULL);
}

        char c;
        int fd = open("foo", O_RDONLY);
        while(read(fd, &c, 1) > 0) {
                write(1, &c, 1);    /* the final result using the primitive write   */
        }
        return 0;
}
```

ps eaux | grep "<name>" | wc -l   // name == USER

```
main() {
          int tube1[2];
          int pid1 = fork();
           if (pid1 == 0) {
                    int tube2[2];
                    int pid2 = fork();

          if (pid2 == -1) {   exit(4);   }
          else if (pid2 == 0) {           // ps eaux
                    close(tube2[0]);
                    dup2(tube2[1], 1);
                    close(tube2[1]);
                    execlp("ps", "ps", "eaux", (char*)0);
                    exit(5);
           }
                    // grep $USER
          close(tube2[1]);
          dup2(tube2[0], 0);
          close(tube2[0]);

          dup2(tube1[1], 1);
          close(tube1[1]);
          execlp("grep", "grep", argv[1] : getenv("USER"), NULL);
          exit(6);
          }
                    // "wc -l"
          close(tube1[1]);
          dup2(tube1[0], 0);
          close(tube1[0]);
          execlp("wc", "wc", "-l", (char*)0);
          return 0;
}
```

ps eaux | grep "^$1 "

```
main(int argc, char *argv[]) {
          int tube[2];              int pid;                    pipe(tube);              pid=fork();
          if (pid==-1) { exit(2); }
          if (pid==0) { // child ps eaux
                    close(tube[0]);
                    dup2(tube[1],1);
                    close(tube[1]);
                    execlp("ps", "ps", "eaux", NULL);
          }
          else   { // parent grep "^$1 "
                    char *t= malloc( strlen(argv[1])+3 );
                    close(tube[1]);
                    dup2(tube[0], 0);
                    close(tube[0]);
                    sprintf(t, "^%s ", argv[1]);
                    execlp("grep", "grep", t, NULL) ;    }         return 0;                  }
```

ps eaux | grep "^$1 " > /dev/null

```
main() {
        int tube[2];   int pid=fork();
        if(pid == 0 ) {  // child process -  ps eaux
                close(tube[0]);
                dup2(tube[1],1);
                close(tube[1]);
                execlp("ps","ps","eaux",NULL);
                exit(0);
                }
        else{   // parent
                close(tube[1]);
                dup2(tube[0],0);
                close(tube[0]);

                int fd = open("/dev/null",O_WRONLY);
                dup2(fd,1);
                close(fd);
                char *t=malloc(strlen(argv[1]+3);
                sprintf(t, "^%s", argv[1]);
                execlp("grep","grep",t,NULL);
                exit(0);
}
```

ps eaux > toto ; grep "^$1 "< toto > /dev/null

```
int fifo = mkfifo("toto", 0666);
int pid = fork();
 if (pid == 0) {
        int fd = open("toto", O_WRONLY);
        dup2(fd, 1);
        close(fd);
        execlp("ps", "ps", "eaux", (char*)0);
        exit(4);
        }
int fd = open("toto", O_RDONLY);
dup2(fd, 0);
close(fd);

int out = open("/dev/null", O_CREAT | O_WRONLY | O_TRUNC, 0666);
dup2(out, 1);
close(out);

char *t= (malloc((strlen(argv[1]) + 3);
sprintf(t,”^%s”,argv[1]);
execlp("grep", "grep", t, (char*)0);
```

ps eaux | grep "^$1 "> /dev/null && echo "$1 is connected"

```
int main(int argc, char *argv[]) {
        int pid1 = fork();
        if (pid1 == 0) {   // CHILD PROCESS
                int tube[2];
                int pid2 = fork();
        else if (pid2 == 0) {
                close(tube[0]);
                dup2(tube[1], 1);
                close(tube[1]);
                execlp("ps", "ps", "eaux", NULL);
                exit(4);
        }
        else {
                close(tube[1]);
                dup2(tube[0], 0);
                close(tube[0]);

        int out = open("/dev/null", O_CREAT | O_WRONLY | O_TRUNC, 0666);
        dup2(out, 1);
        close(out);

        char *t = malloc((strlen(argv[1]) + 2)
        sprintf(t, "^%s ", argv[1]);
        execlp("grep", "grep", in, NULL);
        exit(4);
        }
}

int why;    wait(&why);
if (  WEXITSTATUS(why) == 0  ) {
        char *out = (char*) malloc((strlen(argv[1]) + 16) * sizeof(char));
        strcpy(out, argv[1]);
        strcpy(out + strlen(argv[1]), " is connected\n");
        write(1, out, strlen(out));   /* display the final result using the primitive write  */
}
```

displays a counter each time a SIGINT signal is received

```
#include<stdio.h>    #include<stdlib.h>       #include <signal.h>        #include<unistd.h>

void counter(int signal) {
        int count; count++;
        if (signal == SIGINT)
        printf("received SIGINT \n");
        if ( count == 5 ) exit(0);
}
```

```
int main() {

        printf("pid: %d\n",getpid());

        signal(SIGINT,counter);

        while(1);       return 0;

}
```

SIGQUIT

```
void count(int signal) {
        static int count=0; printf("%d\n",count);
        int pid=fork();
        if ( pid == -1 ) { exit(0); }
        else if ( pid == 0 ) {
                execlp("ulimit","ulimit","-c","unlimited",NULL);
                exit(0);
        }
        count++;
        if ( count >= 5 ) exit(0);
}
```

```
int main() {

        printf("pid: %d\n", getpid());

        signal(SIGQUIT, count);

        while(1);

        return 0;

}
```

```
void getsignal(int signal) {
        char buffer[100];
        psignal(signal,buffer); // prints the descr of signal
        // printf("%s\n",buffer);
        exit(0);
}
```

```
int main() {

        printf("pid: %d\n",getpid());

        signal(SIGINT,getsignal);

        while(1);

}
```