# Pipe System call

```
int main(int agrc , char* argv[])
{
    int fd[2];

    // fd[0] is for read instruction

    // fd[1] is for write instruction

    if (pipe(fd)== -1)
    {
        cout<<" Error occurred...";
    }

    int id = fork();
    if (id==-1)
    {
        cout << " Error... ";
    }
}
```

write( int fd , void* buf , size_t cnt)

fd = file descriptor
buf = buffer
cnt = length of buffer

**Example:**

```
char buf1[12]="hello world";
write( fd[0], buf1 , strlen(buf1));
write (1, buf2, read(fd[1], buf2, 12));

char buf2[12];
```

```
read (int fd, void* buf , size_t cnt)
```

Example:

```
read (fd, c , 10)
```

# Pipe

=> A pipe has two ends. One end is used for reading operation and the other end is used for writing operation.

write → Pipe → read

=> A pipe cannot read anything if nothing is written at the file. So, we close the read operation when we are writing and we close the write end of the file when we are reading.

=> A pipe is used in a child and parent process.

=> There is no restriction on writing being done in either the parent process or child process and same goes for reading.

=> The syntax for calling pipe is :

```
int pipe (int fds[2])
```

where this is the pipe system call

this is the array of fide descriptor table

=> fd[0] is used for reading and fd[1] is used for writing.

=) If the process returns 0 then it is a success and if it returns -1 then it is a failure.

=) 512 bytes can be written on a pipe where as pipe is able to read 1 byte.

=) To communicate b/w the two ends of the pipe we use the read and write system call.

=) At the ~~read~~ write system call, one of the arguments of the system call would be fd [1] while at the ~~write~~ read system call, one of the argume of the read system call would be fd [0].

=) After read system call is implemented successfully, it will return the numbers of bytes it had read.

=) After write system call is implemented successfully, it will return the number of bytes it had written.

# Dup system call

=> Dup stands for duplicate.

=> Dup is used to duplicate a file descriptor table. It has mainly two system calls dup() and dup2().

=> dup() takes one parameter which is the old file descriptor and it generates a new file descriptor on success. The new file descriptor will get the lowest numbered unused file descriptor as the new value.

Example:-

```
int main()
{
    int fd, fd1;
    fd = open("dup", O_RDONLY);
    printf("old file descriptor %d\n", fd);
    fd1 = dup(fd);
    printf("New file disc. %d\n", fd1);
}
```

=> The above program will simply print the value of old file descriptor and the value of new file descriptor.

# Process Synchronization

⇒ There are two types of process synchronization ① Serial mode ② parallel mode.

⇒ In serial mode, next process will starts not until the previous process has terminated and it goes on and on.

⇒ In parallel mode, more than one process can run simantaneously.

⇒ Parallel mode is further divided into two categories ① cooperative process and ② independent process.

⇒ In cooperative process, execution of one process affects the other process because they share something which is mutual. For example, variables, buffer/memory, code etc.

⇒ If any single process in cooperative process does not work. Then, it will generate error.

P1

```
int x = shared
   x++ ;
  Sleep(1);
```

at this point CPU moves to P2 as context switching occurs

```
     Shared = x ;
```

P2

```
int y = shared
  y-- ;
 sleep(1);
```

at this point CPU moves to P1 as context switching occurs

```
  shared = y;
```

=) The final answer will not be correct due to lack of synchronization and a condition called race condition would occur as two different answers would compete to become the final answer.