shm_atime → set the current time
shm_lpid → set the process ID
shm_nattch → incremented by one.

} shmat()

shm_dtime → set the current time.
shm_lpid → set to the process-ID.
shm_nattch → is decremented by one.

} shmdt()

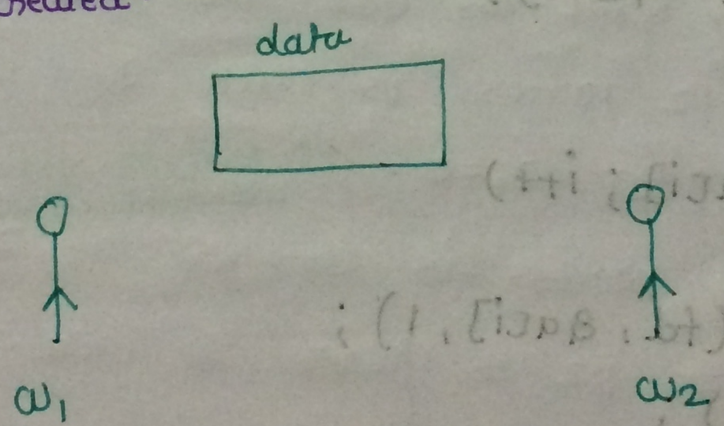
* disadvantages :-

① Synchronization

- both process are enter the data at a time Then data corrupt.

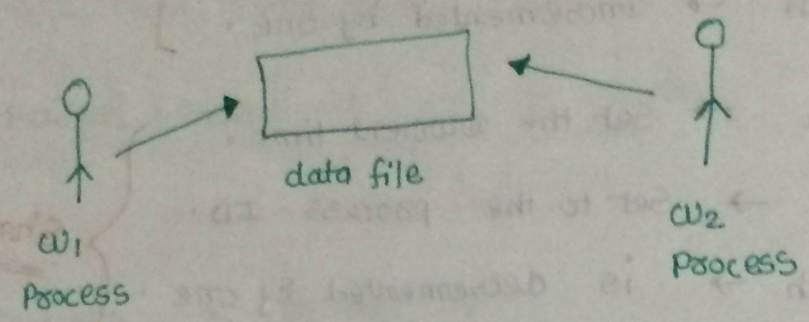
shmget ()

⇒ A new shared memory segment, with size equal to the value of size rounded upto multiple of PAGESIZE is created.



⇒ data is also shared because data file is access by any process.

Program :- both process W_1 and W_2 are write in data file.



Process : Sender W1.c

```
#include "header.h"
main()
{
    char a[] = " abcdefghijklmnopqr";
    int fd ;

    fd= open ("data", O_RDWR | O_APPEND | O_CREAT, 0644);
    if (fd<0)
    {
        Perror (" open");
        return ;
    }
    for (i=0 ; a[i] ; i++)
    {
        write (fd, &a[i], 1) ;
        Sleep (1) ;
    }
  Printf (" done ----\n") ;
}
```

Program : Receiver W2.c

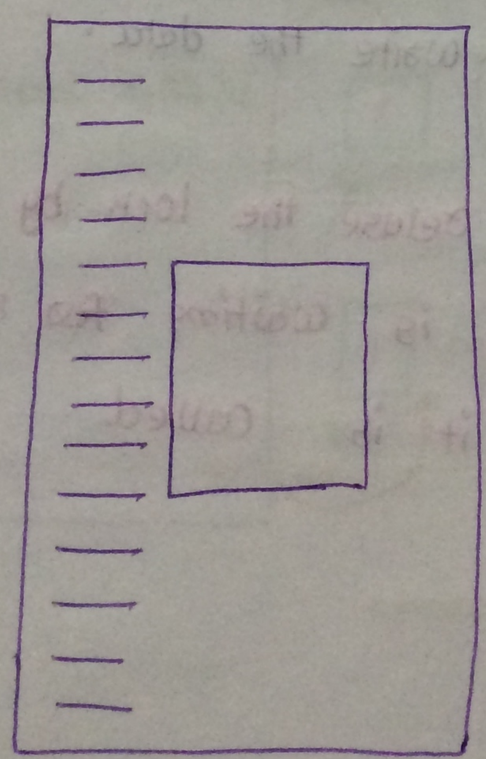
```
# include " header.h "
main()
{
```

```
Char a[] = " ABCDEFGHIJKLMNOPQR " ;
int fd ;
    fd = open( "data " , O_RDWR | O_APPEND | O_CREAT , 0644) ;
    if (fd<0)
    {
        Perror (" open ") ;
        return ;
    }
    for (i=0 ; a[i] ; i++)
    {
        write (fd , &a[i] , 1)
        Sleep(1) ;
    }
  Printf (" Done ----\n") ;
}
```

Defination :- Critical Section OF Code ....

The part OF the program where we trying to access Shared Resource is called Critical Section of the Code.

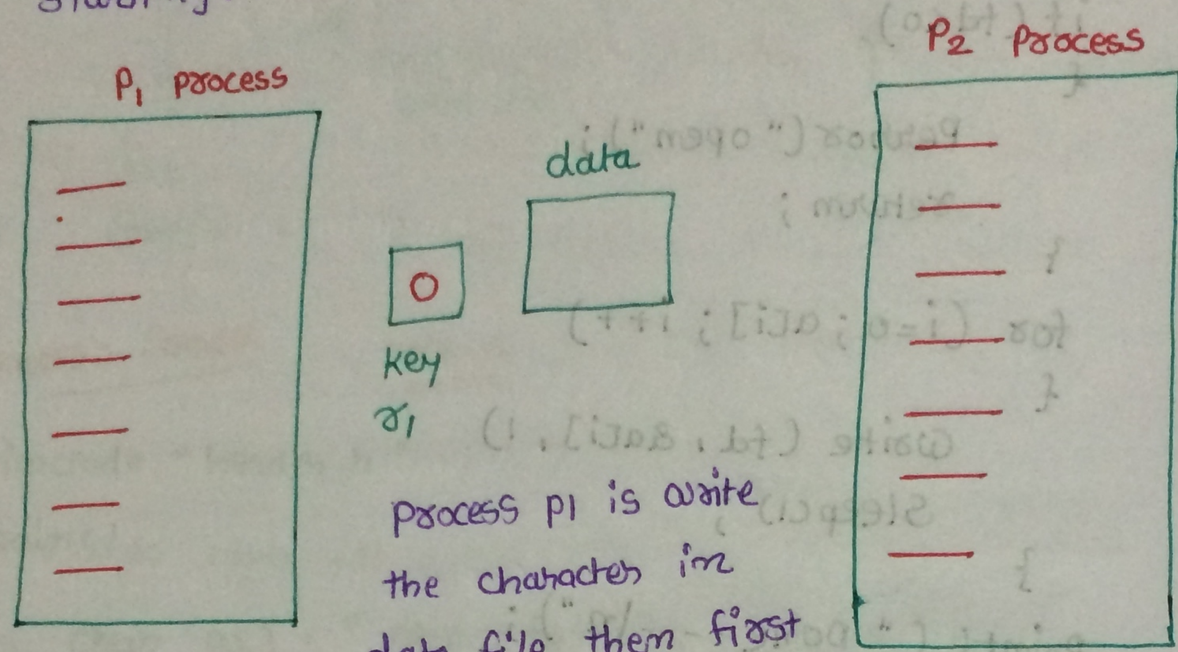
In big project :-



Only access this part it is called Critical Section of the code.

⇒ Before enter the critical section first check the shared part is free or not.

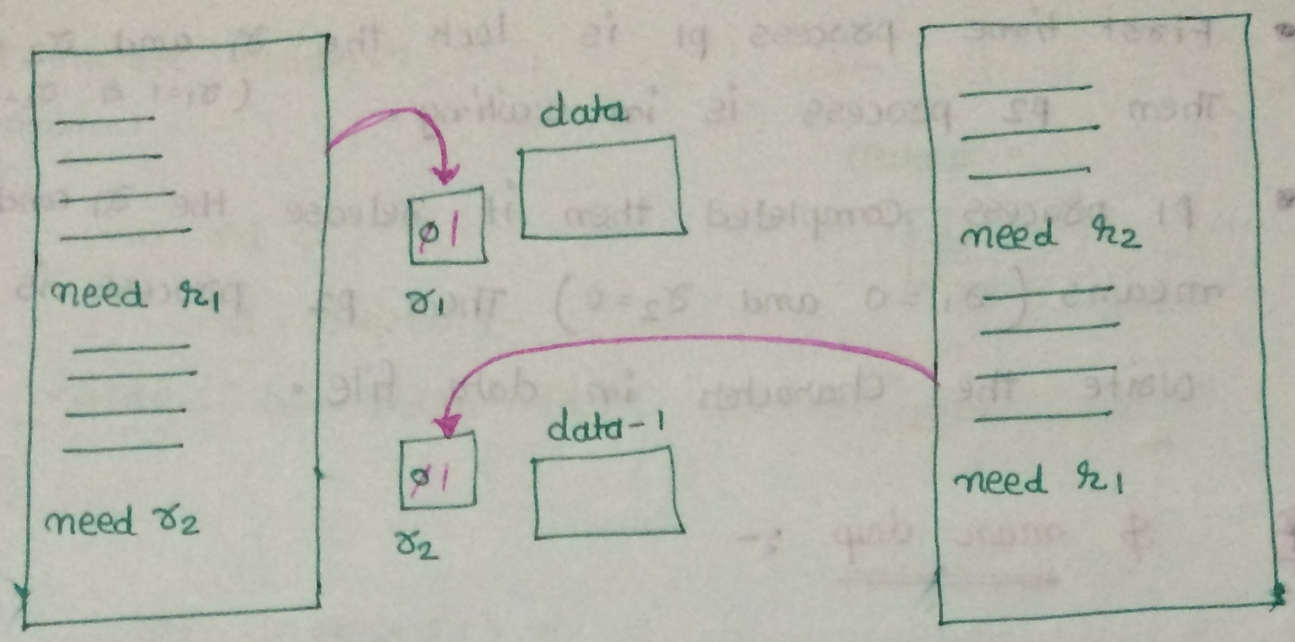
⇒ It is free than lock it and after other process starting.



process P1 is write the character in data file then first it is check r_1

- r_1 is 0 then it is write the data (but first change $r_1 = 1$) after writing all the data again change $r_1 = 0$.
- P_1 is write the data that time $r_1 = 1$ Then P2 process is check $r_1 = 1$ so it is wait for $r_1 = 0$.
- P_1 process is complete the $r_1 = 0$ Then P2 is write the data.

* P_1 process is waiting for release the lock by P_2 Process and P2 process is waiting for release the lock by P1 process it is called Dead process.

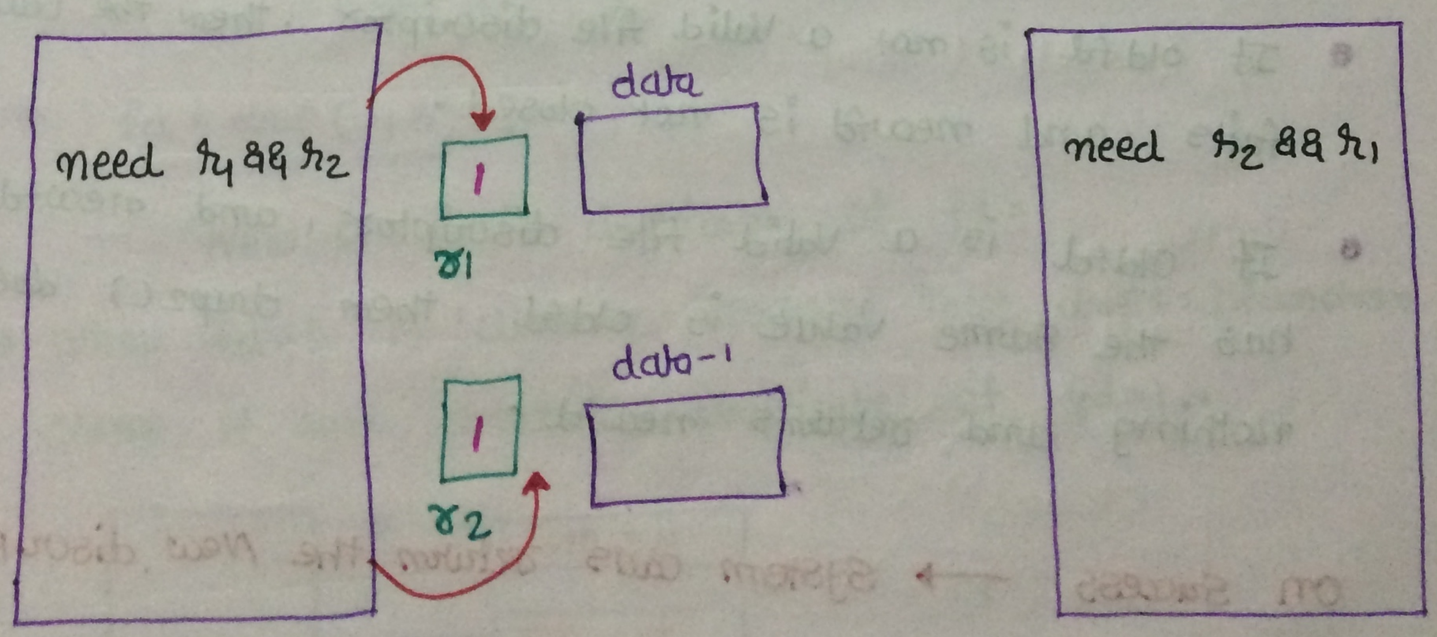


Dead process :-

OR Dead lock :-

One process is waiting for releases the lock by another process and another process is waiting for release the lock by this first process this situation is called the "dead process" or "dead lock".

Overcome it :-



- First time process p1 is lock the r_1 and r_2 ($r_1=1$ & $r_2=1$) Then p2 process is in waiting.
- p1 process completed then it release the r_1 and r_2 means ($r_1=0$ and $r_2=0$) Then p2 process is write the character in data file.

⊛ $ man dup :-

dup, dup2, dup3 → duplicate a file discriptor

```
# include <unistd.h>
```

① int dup (int oldfd);

② int dup2 (int oldfd , int Newfd);

③ int dup3 (int oldfd, int Newfd, int flags);

→ dup() uses the lowest numbered unused discriptor for the New discriptor.

→ dup2() makes newfd be the copy of oldfd, closing newfd first if necessary, but not the following.

- If oldfd is not a Valid file discriptor, then the call fails, and newfd is not closed.
- If Oldfd is a Valid file discriptor, and newfd has the same Value is oldfd, then dup2() does nothing and returns newfd.

On Success → System calls return the New discriptor

On error → -1 is returned.

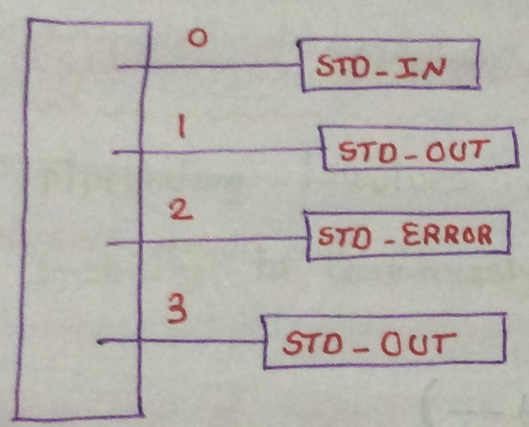
```
# include " header.h "
main()
{
    int fd ;
    fd = dup(1) ;
  Printf (" fd = %d\n", fd) ;
}
```

Output =

$\boxed{fd = 3}$

```
fd = dup(1) ;
```

New fd (Stdout) — Old fd (Stdout)



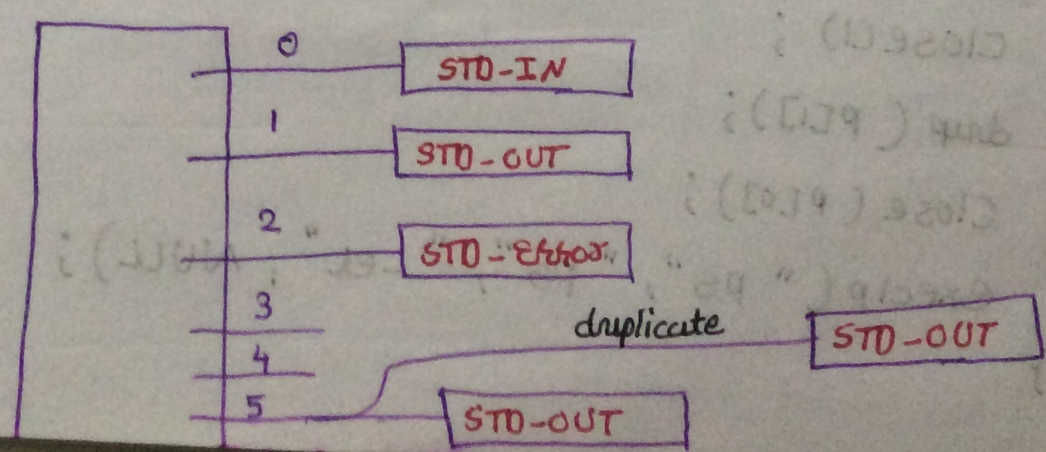
→ **fd = dup(1) :-**

New fd=3 is duplicate of fd=1.
(first priority is lowest New fd).

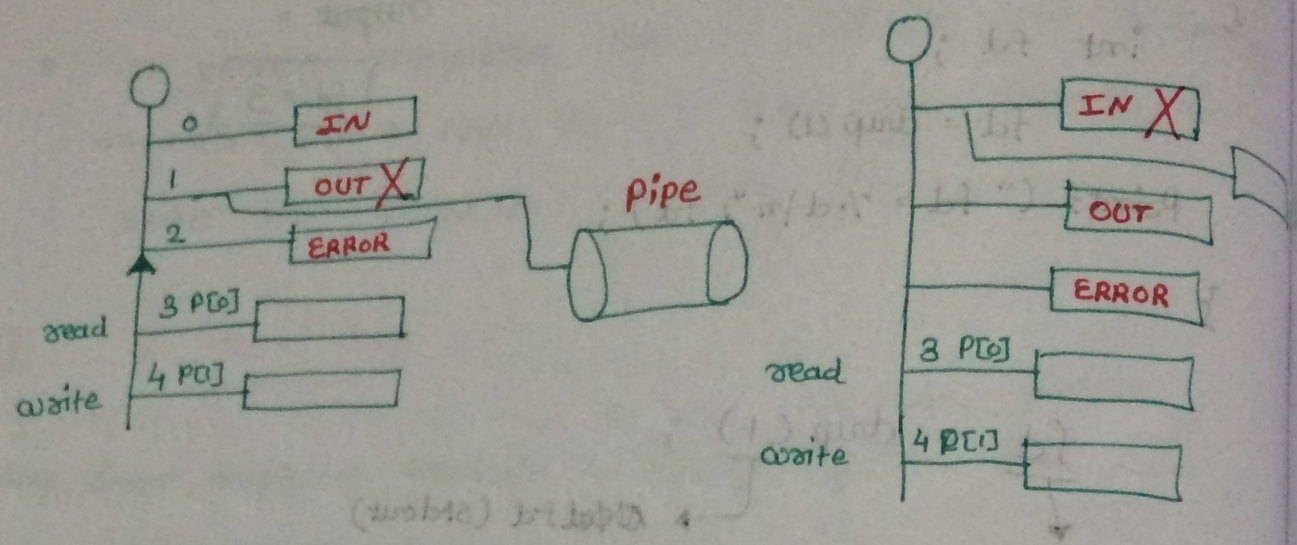
→ **fd = dup(1,5) :-**

New fd = 5 is duplicate of fd=1

→ when fd=5 is already reserved then dup2() function close it and creat the duplicate of fd=1.



Program ps -el | grep pts/0




```
#include "header.h"
main()
{
    int p[2];
        pipe(p);
  if (fork() == 0)
  {
      // grep pts/0  (Child---)
      close(0);
      dup(p[0]);
      close(p[1]);
      execlp("grep", "grep", "pts/0", NULL);
  }
  else
  {
      // Parent
      // ps -el
      close(1);
      dup(p[1]);
      close(p[0]);
      execlp("ps", "ps", "-el", NULL);
  }
}
```