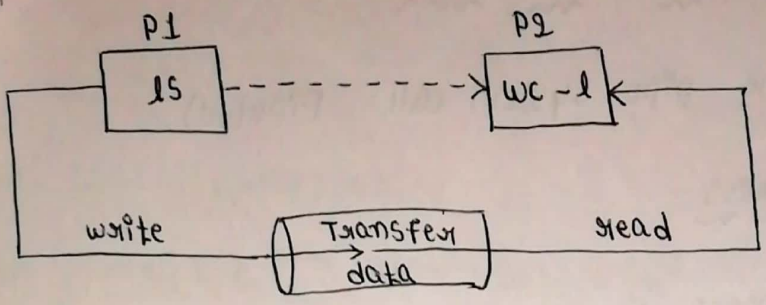
Pipes



How pipes provide communication between two processes.?

→ pipes provides communication between parent and child or inter related processes.

* Creation of a pipe:

Syntax: `int pipe (int fields);`

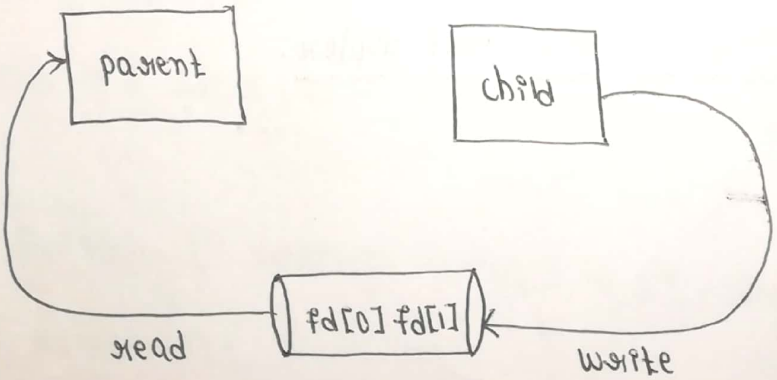
→ file discriptor (fields) is used to identify read end and write end.

→ Usually file discriptor we can create like integer array.

int fd[2]; Here fd = fields

- fd[0] → read
- fd[1] → write

* procedure for creating pipe:



→ If one end is open the other end should be closed.

→ only one operation can be done at a time.

* procedure for Half duplex

1. create a pipe using pipe system call pipe(fd).

Procedure for child process

2. create a child process so that ps

```
pid = fork();
if (pid==0)
child created successfully
if (pid!=0)
  " fork error"
```

3. After creation of pid if (pid==0) then if it is true close(fd[0]);

use write system call write(fd[1], string, strlen(string));

Procedure for parent process.

4.
```
if (pid>0)
then
close (fd[1]);
read (fd[0], buff, sizeof(buff));
```

Program for Half duplex.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
```

```
.in (void)

int fd[2], nbytes;
char pid-t childpid;
char string[] = "Hello world!\n";
char readbuffer[80];
Pipe (fd);
if ((childpid = fork()) == -1)
{
    Perror ("fork");
    exit (1);
}
if (childpid == 0)
{
    close (fd[0]);
    write (close fd[0] fd[1], string, (strlen(string)+1));
}
else
{
    close (fd[1]);
    nbytes = read (fd[0], readbuffer, sizeof (readbuffer));
}
    Printf ("received string : %s", read buffer);
    return (0);
}
    output :
        received string : Hello world!
```

FIFO

1. Create file

   mknod ("fifo1", 0666)

2. Open file
   - read → fd1 = open (fifo1, O_RDONLY);
   - write → fd2 = open (fifo2, O_WRONLY);

3. read on write operations.

4. close all the file discriptors.

Program :

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
int main()
{
    char s[100] = "  n ;
    char s1[100] = "  n ;
    int fd , fd1 ;
    fd = open ("fifo1", O_WRONLY);
    fd1 = open ("fifo2", O_RDONLY);
   Printf ("In enter the file name : n");
   scanf ("%s", s);
   write (fd, s, strlen(s));
   while (read (fd1, s1, 1000) != 0)
   {
      Printf ("file content : %s", s1);
    }
}
```

Server

```
mknod (fifo1, 0666);
mknod (fifo2, 0666);
readfd = open (fifo1, O-RDONLY);
writefd = open (fifo2, O-WRONLY);
read (readfd, fname, 100);
fd = open (fname, 0666);
while (read (fd, buff, 1000)<0)
{
  write (writefd, buff, 1000);
}
close (fd);
close (readfd);
close (writefd);
```

client

```
1. writefd = open (fifo1, O-WRONLY)
2. readfd = open (fifo2,
                  O-RDONLY);
3. write (fd, fname, sizeof (file
                  name));
4. while (read (readfd1 buff1 1000)
                  <=0)
{
   pf ("%s", buff);
}
```

Message Queues:- Server and client can interact with each other by sending the messages using message queues.

→ IPC key can be created using the function ftok()

Syntax:- ① key = ftok ("path", A2);

→ ftok() function has mode, GID, UID and CID.

② msgid = msgget (key, flags);

- Point-to-point
  - 0400 → MSG_R
  - 0200 → MSG_W
- multicasting
  - 0040 → MSG_R >>3
  - 0020 → MSG_W >>3
- broadcasting
  - 0004 → MSG_R >>6
  - 0002 → MSG_W >>6

→ If msgid = -1 then it fails to create message queue id.

③ msgsnd():

msgsnd (int msgid, char *ptr, int length);

(or)

msgsnd (int msgid, struct msgbuf *ptr, int length);

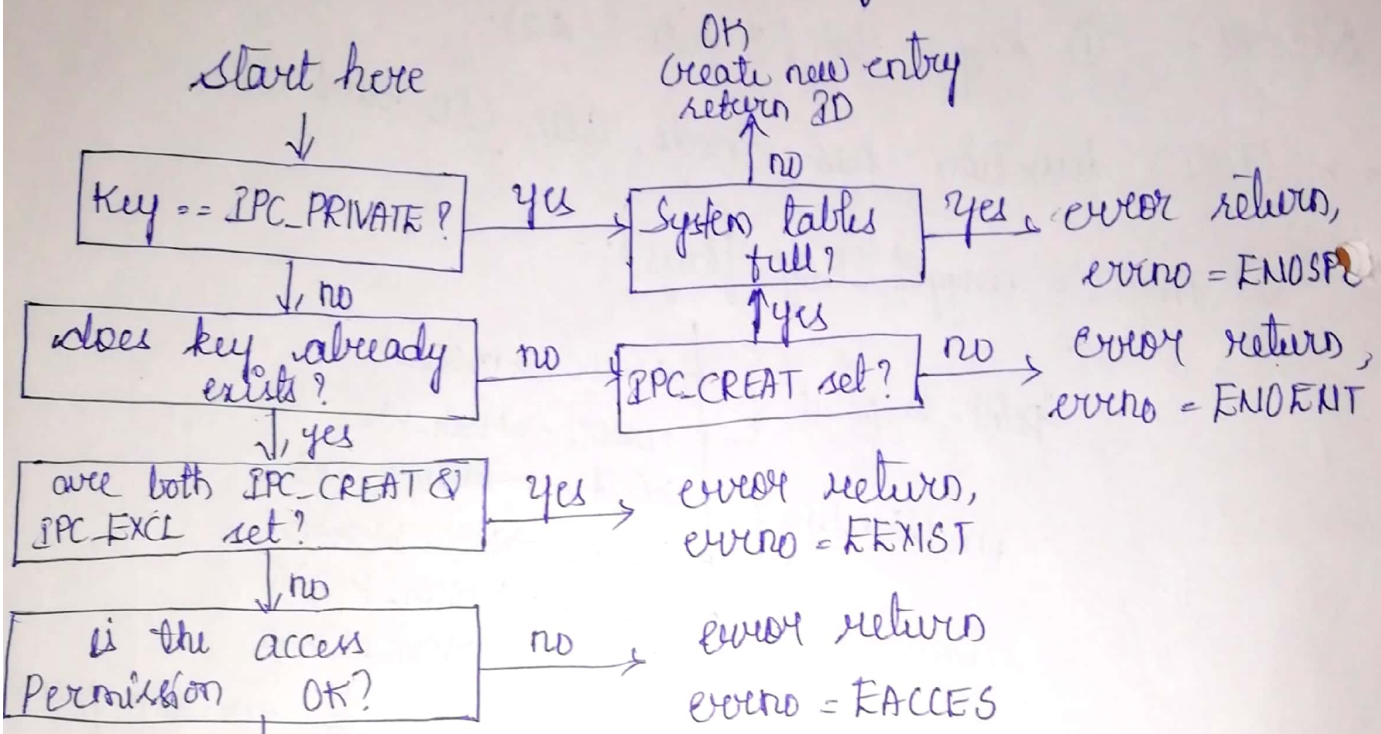
```
Struct msgbuf
{
    int msgtype;
    char msg[];
}
```

④ msgrcv (int msqid, struct msgbuf *ptr, int
int msgtype);

(or)

msgrcv (int msqid, struct msgbuf *ptr, int length, 0);

Logic for opening or creating an IPC channel



Server → message → client

① key = ftok ("path", 84);

② msqid = msgget (key, 0200);

③ msgsnd (msqid, "hai", 10);

① key = ftok ("path", 84);

② msqid = msgget (key, 0400);

③ msgrcv = (msqid, buff, 10, 0);

④ Printf ("%.s", buff);

q client.c :-

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <string.h>
#include <stdio.h>

int main (void)
{
    key_t ipckey;
    int mq_id;
    Struct { long type; char text[100]; } mymsg;
    int received;
    ipckey = ftok ("/tmp/abc", 42);
    Printf ("My key is %d \n", ipckey);
    /* set up the message Queue */
    mq_id = msgget (ipckey, 0);
    Printf ("message identifier is %d", mq_id);
    received = msgrcv (mq_id, &mymsg, sizeof (mymsg),0,
    Printf ("%s (%d) \n", mymsg.text, received);
}
```

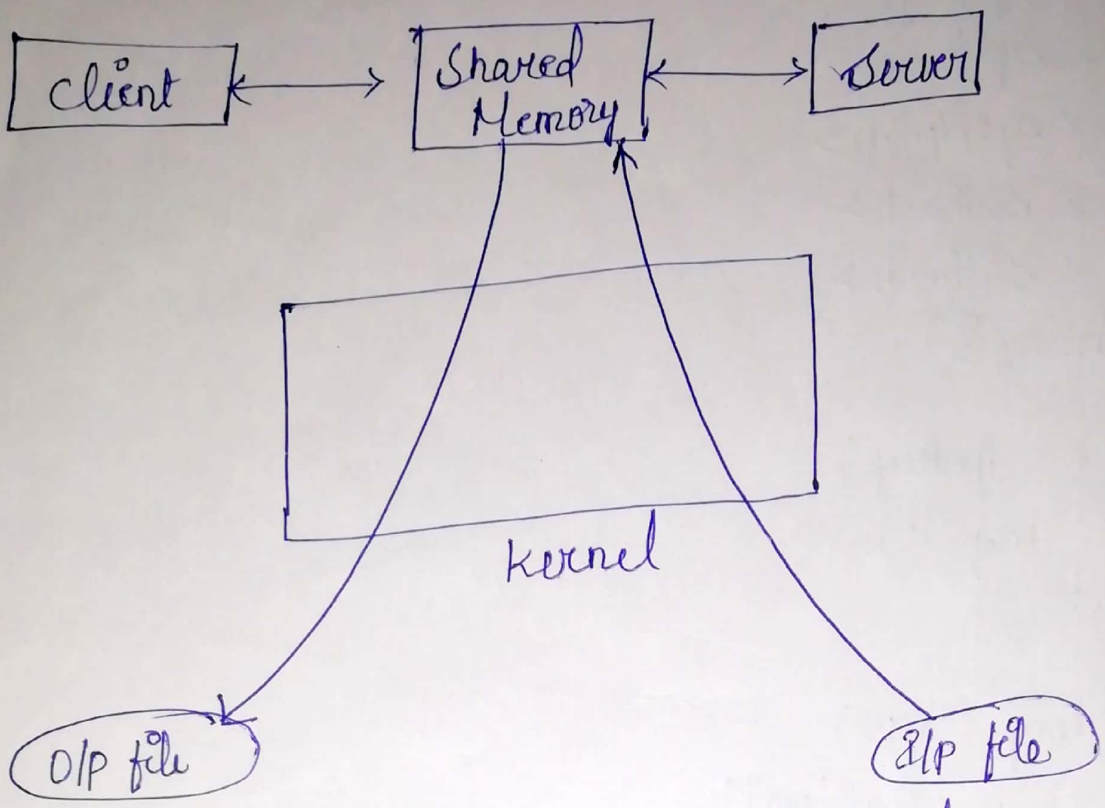
output :-

```
$ cc msgserver.c
$ ./a.out
        my key is -1
        message identifier is -1
$ cc msgclient.c
$ ./a.out
        my key is -1
        message identifier is 0
        Hello, world!
```

ogram for Message queues: msg server.c

```
nclude <sys/types.h>
include <sys/msg.h>
#include <sys/ipc.h>
#include <string.h>
#include <stdio.h>
int main(void)
{
    key_t ipckey;
    int mq_id;
    struct
    {
        long type;
        char text[100];
    } mymsg;
    /* Generate the IPC key */
    ipckey = ftok("/tmp/abc", 42);
    Printf("My key is %d\n", ipckey);
    /* Set up the message queue */
    mq_id = msgget(ipckey, IPC_CREAT|0666);
    Printf("Message identifier is %d\n", mq_id);
    /* Send a message */
    memset(&mymsg.text, 0, 100); /* clear out the space*/
    strcpy(mymsg.text, "Hello, world!");
    mymsg.type = 1;
    msgsnd(mq_id, &mymsg, sizeof(mymsg), 0);
}
```

Shared Memory :-



1. Server get access to a shared memory
2. Server reads i/p file into the shared memory segment.
3. client write that data from the shared memory segment to the o/p file.

→ Creating a shared memory segment
- allocated in bytes amounts

→ shared memory ~~segment~~ operations
- create
- attach
- detach

→ shared memory control
- remove

start

① Creating a shared memory segment

int shmget(key_t key, int size, int shmflg);

② Shared memory control

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

③ Shared memory operations

a) void *shmat(int shmid, const void *shmaddr, int shmflg);

b) int shmdt(const void *shmaddr);

Program for shared Memory :-

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#define SHMSZ 27
main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;
/* we'll name our shared memory segment "5678" */
```

```
key = 5678;
/* Create the segment */
if ((shmid = shmget (key, SHMSZ, IPC_CREAT | 0666)) < 0)
{
        perror ("shmget");
        exit (1);
}
/* Now we attach segment to data space */
if ((shm = shmat (shmid, NULL, 0)) == (char *) -1)
{
        perror ("shmat");
        exit (1);
}
/* Now put something into memory */
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;
/* Finally, we wait until the other process changes the first
character of memory to '*' */
    while (*shm != '*')
        sleep (1);
    for (s = shm; *s != NULL; s++)
        putchar (*s);
}
```

Client Program

```
include <sys/types.h>
include <sys/ipc.h>
include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
# define SHMSZ 27
main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    key = 5678;
    if ((shmid = shmget (key, SHMSZ, 0666)) < 0)
    {
        perror ("shmget");
        exit(1);
    }
    if ((shm = shmat (shmid, NULL, 0)) = (char *) -1)
    {
        perror ("shmat");
        exit(1);
    }
    for (s = shm; *s != NULL; s++)
        putchar (*s);
    putchar ('\n');
    *shm = '*';
    exit(0);
}
```

D

Output :-

```
$ cc shmserver.c
$ ./a.out
$ bcdef.....
$ xyz
```

```
$ cc shmclient.c
$ ./a.out
abcdefghijklmnopq
rstuvwxyz.
```

maphore :- Semaphores are a synchronization primitive.

As a form of IPC, they are not only used for exchanging the data between process and also for synchronize their operations.

→ consider a semaphore as an integer valued variable that is a resource counter.

→ our use of semaphore is to provide resource synchronization between different processes.

→ The actual semaphore value must be stored in the kernel.

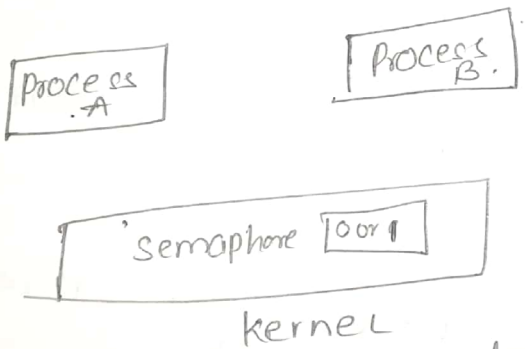


fig:- Semaphore value stored in kernel.

→ To obtain a resource that is controlled by a semaphore, a process needs to test its current value.

→ If the current value is ~~grater~~ greater than zero, decrement the value by zero.

→ To release a resource that is con by a semaphore, a process increments th semaphore value.

→ If some other process has been waiting for the semaphore value to become greater than zero, that other process can now obtain the semaphore.

Pseudocode :-

```
for(;;)
{   if (semaphore_value > 0)
    {   semaphore_value--;
        break;
    }
}
```

System V Implementation expands semaphore as follows :-

1. A semaphore is not a single value but a set of nonnegative integer values.
2. Each value in the set is not restricted to zero and one. Instead each value in the set can assume any nonnegative value, up to a system defined maximum value.

Kernel support for semaphore :-

(10)

The every set of semaphore kernel maintains the following structure :

```
#include <sys/types.h>
#include <sys/ipc.h>

struct semid_ds
{
    struct ipc_perm sem_perm;
    struct sem *sem_base;
    ushort sem_nsems;
    time_t sem_otime;
    time_t sem_ctime;
};
```

System V Semaphore system call :-

All these system calls will be available in <sys/sem.h>

1) semget :- A semaphore is created, or an existing semaphore is accessed with the semget system call.

Syntax :-

```
int semget (key_t key, int nsems,
            int semflag);
```

The value returned by semget is the semaphore identifier, semid, or -1 if an error occurred.

Semflag values for semget system call:-

Numeric	Symbolic	Description
0400	SEM_R	Read by owner
0200	SEM_A	Alter by owner
0040	SEM_R >> 3	Read by group
0020	SEM_A >> 3	Alter by group
0004	SEM_R >> 6	Read by world
0002	SEM_A >> 6	Alter by world
	IPC_CREAT	
	IPC_EXCL	

(2) semop :- once a semaphore set is opened with semget, operations are performed on one or more of the semaphore values in the using the semop semop system call.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Syntax

```
int semop (int semid, struct sembuf *opsb,
           unsigned int nops);
```

```
struct sembuf
{
    ushort  sem_num;
    short   sem_op;
    short   sem_flag;
};
```

1) if sem_op is positive :- The value of sem_op is added to the semaphore's current value.

2) if sem_op is zero :- The caller has to wait until the semaphor's value becom zero

3) if sem_op is negative :- The caller has to wait until the semphore's value becomes grater than or equal to the obsolute value of sem_op.

* The return value of semop is zero if ok or -1 if error

(3) Semctl :- This system call provides various control operations on a semaphore.

Syntax :-

```
int semctl (int semid, int semnum, int cmd,
            union semun arg);
```

1) cmd :- IPC_RMID → remove a semaphore
IPC_GETVAL } to fech and set the
SETVAL } semaphore values.

2) semnum :- to specify one number of the semaphore

3) semid :- id of the semaphore creatid by semget system call

File locking with semaphore :-

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 123456L /* key value for semget() */
#define PERMS 0666

static struct sembuf op_lock[2] = { 0, 0, 0,
                                    0, 1, 0 };

static struct sembuf op_unlock[1] = { 0, -1,
                                      IPC_NOWAIT };

int semid = -1 /* semphore id */

my_lock (fd)
int fd;
{
    if (semid < 0)
    {
        if ((semid = semget (SEMKEY, 1,
                             IPC_CREAT | PERMS)) < 0)
            err_sys ("semget error");
    }
    if (semop (semid, &op_lock[0], 2) < 0)
        err_sys ("semop lock error");
}

my_unlock (fd)
int fd;
{
    if (semop (semid, &op_unlock[0], 1) < 0)
        err_sys ("semop unlock error");
```