IPC – 2
Shared memory

System V shared memory :

- popular shared memory.
- proc. first creates shared memory seg.
  Have to make sys call
  shmget(IPC_PRIVATE,size,S_IRUSR | S_IWUSR)
  return segment id
- proc. that want acces have to be attached to it.
  shared_mem = (char *) shmat(id, NULL, 0);
  shared_mem : pointer to mem sef
- now proc can write t =o shared mem
- after done detach the shared mem from addr. Spc.
  shmdt(shared_mem)

** **example**

assign key to shmid
  shmid = shmget(key,IPC_CREAT | 0666)
  if 10 proc share the same seg, then have to **copy the key to all 10 proc code**.
  So it help us identify with kernel with mem seg we are referring.

Then attach to get ; returns a pointer to seg

wehave delete seg . Using shmdt

POSIX SHARED MEMORY:

POSIX (Portable OS interdace) compilant
family of standard
specified by IEEE
maintained compatibility between OS.
  API, shell, utility compacitbilty among unix and variants

1. shmopen
2. ftruncate
3. mmap

Message passing

In message passing, kernel makes a message queue which is shared by proc. (send or receive message)

- Message sys. : proc. communicate with each other using **send(), receive() like sys. call given by OS**
- IPC provide 2 ops
  1. send(msg) : fixed / varible size
  2. receive(msg)
- If proc want to communicate :
  - eastablish comm. link b/w them
- comm. link establish in multi. way.

***

In Write msg to queue:

1. sys. call : key = **ftok(progfile, 65)** : create a unique key
   if another want to share mem queue, then have to share the unique key

2. sys. call msgid = **msgget(key, 0666 | IPC_CREAT) :** get access to the message.
# returns a pointer to msg Q.
Message.mesg_type = 1 # **have to greater than 1**
3. sys. call **msgsnd(msgid, &msg, sizeof(msg), 0)**
4.

In Read msg from queue:

1. sys. call : key = **ftok(progfile, 65)** : create a unique key
if another want to share mem queue, then have to share the unique key
2. sys. call msgid = **msgget(key, 0666 | IPC_CREAT) :** get access to the message.
returns a pointer to msg Q.
M=
3. sys. call **msgrcv(msgid, &msg, sizeof(msg), 0)**
4. sys. call **msgctl(msgid, IPC_RMID, NULL); #** delete the msg queue

similar as socket in CN. Message passing

Message passing using "Naming":

// can be implemented as same form as post offices

1. Direct communcation with rcvr
    1. rcvr is identified by sndr using **name**
2. indirect comm with rcvr
    1. rcvr is identified by sndr in-directly by using "**location of receipt**"

1. Direct communication message passing:
send(P, msg) : send msg to proc P
receive(Q,msg) : receive a msg from proc Q.

Properties of comm link:
- links are established automatically
- link assoc with **exactly one pair of comm. process.**
- b/w each pair there is **exactly one link**
- Usually bidirectional (may be unidirectional)

e.g. firefox and gedit

2. Indirect communication message passing:
messges are directe ad rvecieved by mailbox(**ports**)
- each mailbox has **unique id**
- proc can comm if common mailbox is shared by them.

Properties of comm link:
1. link is established only if shares common mailbox
2. link assoc **with multiple proc**
3. pair of proc may **share several comm. links**
- Usually bidirectional or unidirectional

Operations :
- create a new mailbox
- snd or rcv msg through mailbox
- destroy mailbox

Primitives:
send(A,msg) : send msg to mailbox A
receive(B, msg): receive msg from mailbox B

in KERNEL example taken it was of **Indirect passing.**

Mailbox sharing:

if P1, p2, p3 share mailbox
if p1 send, p2 and p3 receives
but who gets msg ?

Solutions:

Allow **a link to associate to atmost two**
allow only one proc at time to execute receive
allow sys to select arbitraly choose receiver. Sender is notified whi receives message.

Message passing implementation : Synchronisation issues:

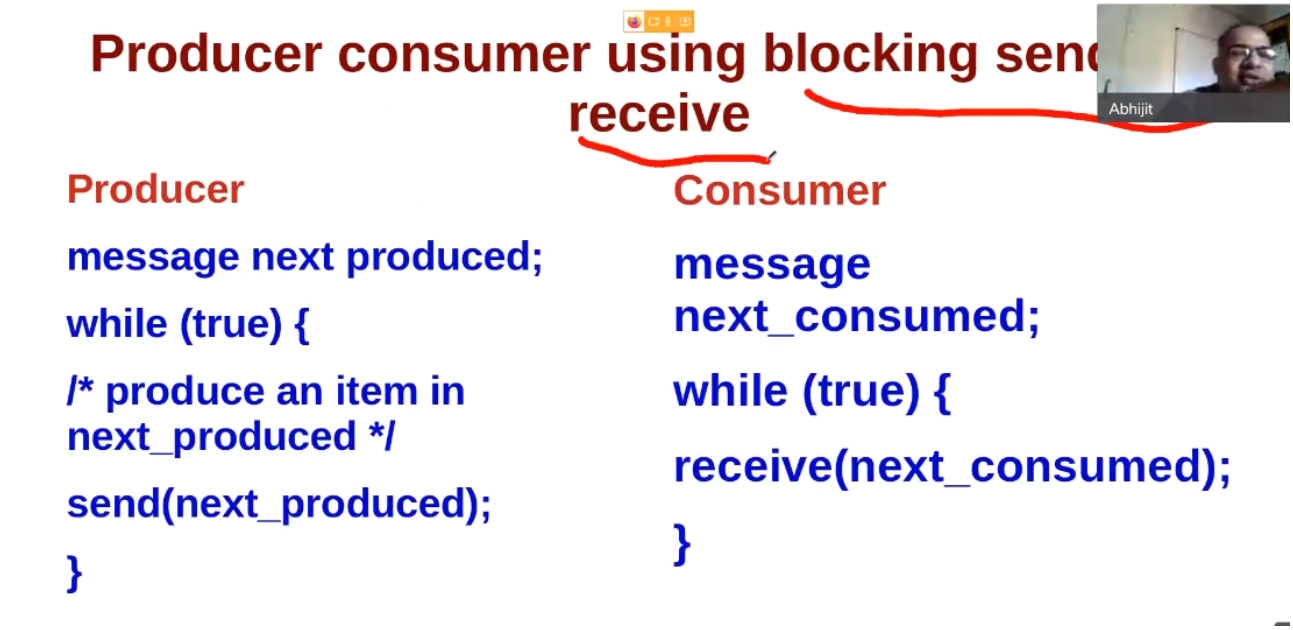
- message pasing may be **blocking or non-blocking**

A. Blocking
- blocking is considered synchronous

1. blocking send has to block sender until msg is received
2. blocking receice has to black receiver until msg is available.

B. Non Blocking
- non-blocking is considered asynchronous

1. non-blocking send has sender send the message and continue // function returns immediately
2. non-blocking receice has receiver receive a valid msg or null.



We have to consider in non-blocking
sndr without receiver can happen
full buffer
reeiver return w/o msg

Message passing implementation : Choice buffering :

- queue of message attached to link

Implemented in 3 ways :

1. Zero capacity : 0 messages
    - sender has to wait for receiver (rendezous)
    // Blocking
2. Bounded capacity – finite length of n messages
    - sender has to wait if link is full
3. Unbounded capacity – infinite length
    - sender never waits

argptr : is used to fetch arg passed by user to sys. call.
pipealloc(&rf, &wf) : struct files will be modified to point fd array
fdalloc(rf) : allocate a index in fd and make pointer to rf

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};
```

in pipealloc :

```
    kalloc a 4kb size to pipe p
    p->readopen = 1;
    p->writeopen = 1;
    p->nwrite = 0;
    p->nread = 0;

    (*f0)->type = FD_PIPE;
    (*f0)->readable = 1;
    (*f0)->writable = 0;
    (*f0)->pipe = p;
    (*f1)->type = FD_PIPE; //enum in file struct : 1
    (*f1)->readable = 0;
    (*f1)->writable = 1;
    (*f1)->pipe = p;
```

in file struct :
if pipe : struct pipe points to pipe
if file : inode points to file
in optimized both should be in union

****
in sys_read:

```
    we get fd by argfd // gets user passed args in arg*something*
    calls fileread(fd,location,number_of_byte)
```

in fileread:

```
    checks fd->type
    based on it called read
        if type == FD_PIPE:
            call piperead(fd->pipe, addr, number_of_byte)
```

```
                if type == FD_INODE:
                        ilock(fd->ip)
                        call piperead(fd->pipe, addr, number_of_byte)
in piperead():
        check if pipe empty
                if number of read == number of write and is writeopen:
                        return -1
                wait in while loop // sleep(p->nread, p->lock)
        else:
                in a for loop of number of byte
                read content of pipe to addr
                then wake up the writer //wakeup(p->nwrite)

in pipewrite():
        check if pipe full
                if number of read == number of write and is writeopen:
                        return -1
                wait in while loop
                // sleep(p->nwrite, p->lock)
        else:
                in a for loop of number of byte
                write content of pipe from addr
                then wake up the writer //wakeup(p->nread)

**********
```

// **Pipe is created**

ipcs : shows list of all mem mechanism eg msg queue