24/12/19

Computer Networks

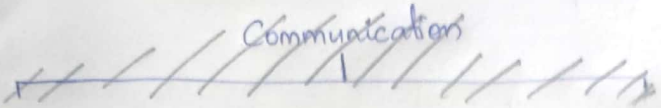
(communication purpose)

Computer — Creating
- Computation
- Co-ordination/ Communication

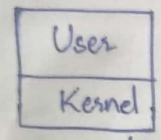
Real Life — Active / Passive

Active – implemented using a process

Passive – File id (file descriptor)

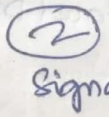


Notation:
- process ○
- Resource □
- File descriptor ○
- Direct comm. —— (two-way) ⟶ (one-way)
- Indirect comm. - - -
- Threads

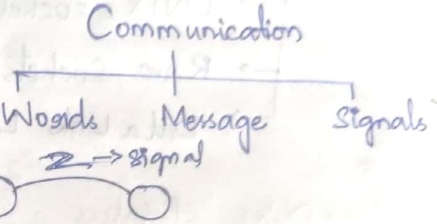


(remembers)

Communication system



Signal

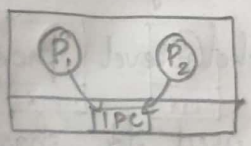


Processors = computer systems (client, server etc.)

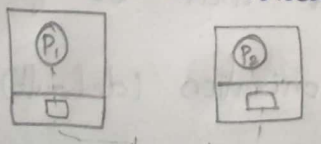
processes } – diff.

Types of process communication:

1) IPC
- All levels / versions of UNIX (multi-users system)
  - Pipes
  - FIFOs
- System V
  - Message queues
  - Semaphores
  - Shared memory



2) RPC (Remote Procedure Calls)



Two processes of different systems communicate but not directly.

P_1 will be able to call methods in remote P_2

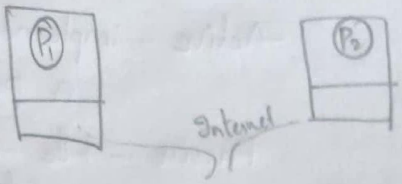
JDBC etc.

Original RPC: Sun RPC

Types
- Sun RPC
- JDBC (most used and commercial)

3) NPC (Networked Process Comm.)

Processes in diff comp. systems communicate through internet (far away)



BSD Sockets (Berkley s/w Distribution)

- Elementary sockets
- UNIX sockets (UDS)
- Raw Sockets
- Data Link Layer Interface (DLI)

TCI (Transport Layer Interface)

5 Hard Disks, 3 Printers, 2 Scanners → How many semaphores req.? (3)

One for each - semaphore

one Semaphore can represent all printers (Values: 0, 1, 2)

No. of semaphores = No. of types of resources

Initialize each semaphore with the values.

Semaphore: used for synchronization, resource sharing

→ Any lower level prog. is done in Socket-level prog.

→ In the above example, semaphores are used for sharing.

If seq. of processes is to be maintained,
↳ semaphores used for syn.

If semaphore is binary ⟹ synchronization (default)
(can be used for allow / not)

Counting semaphores ⟹ sharing semaphores
(if only two values, still not binary)

→ If a process enters critical section acquiring a resource, and leaves without releasing the resource, the process is locked forever.

→ So, monitors are used for maintaining the state of processes and resources and doesnot allow to lock a resource forever.

→ Queues are maintained for waiting processes.

1.
```
main()
{
  int c=0;
①—cout <<" This is test";
   c= fork();
   if (c>0){
②—— cout<<".....";
   wait();
③—— cout <<".....";
  }
  else{
④—— cout <<".....";
   exit();
  }
}
```

Output:

①②④③

①④②③

If no wait():

1 2 3 4

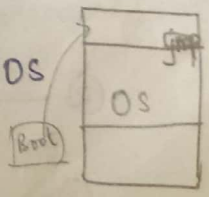
1 4 2 3

1 2 4 3

→ Hardware ckt will 256 byte boot code when system (from .sys to system files) is switched ON.

OS — Resident part (
└ Transient part

Last instruction of boot code is 'jmp' to OS



Interrupt Service Table (IST) — prepared by OS

↳ It overwrites boot code

It will have address to instruction in main memory

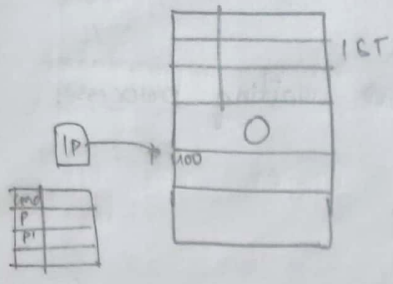
Process Table (will have command prompt instruction) (PT)

Process will address entered in PT

IP points to the process → automatically executed.

When c = fork() executed, image of process P (400-500) copied again. → P'

- All OS will use Round Robin Scheduling Alg.



P = 400 - 500
→ P' = 500 - 600

→ Person and process — equivalent (Simulation)

1. Create	1. fork()
2. executing	2. exec()
3. Wait / Sleep	3. wait(), sleep()
4. end	4. exit()

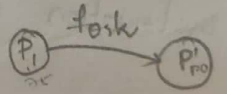
→ In s/w, child cannot wait for parent processes. If parent terminates before child, → orphan child

P1.cpp

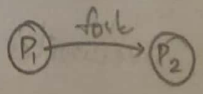
```
main(){
    int c=0;
①   cout<<"...";
    c=fork();
    if(c>0){
②       cout<<"...";
        wait();
③       cout<<"...";
    }
    else{
④       cout<<"...";
        exec("P2.exe");
        exit();
    }
}
```

P2.cpp

```
main(){
⑤   cout<<"Hello World";
}
```



after exec



① ② ④ ⑤ ③ ⑥

→ When 'exec' executed, entire code of P1' is erased and P2.exe is loaded.

when ⑤ cout after exec ("P2.exe");

exit;

}

① ② ④ ⑤ ③

① ④ ⑥ ② ③

① ④ ② ⑥ ③

Control is not returned back after ⑤ P2.exe

```
main()
{
① — cout
  c = fork()
  if (c>0){
      ① — cout
          exec("P2.exe");
      ② — cout
  }
  else{
  ④ — cout
  ⑤ — cout
      exit();
  }
```

P2.c

⑤ — cout

① ② ⑥ ④ ⑤

① ④ ⑤ ② ⑥

→ Threads are useful than fork.

Default: First write parent part, then child part.

When 2 processes are created,

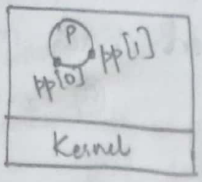
in parent process — child code is not executed (waste)

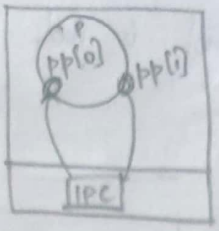
Communication

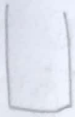
Signals — Words — Messages

Pipes

Real World







Kernel provides pipe

→ It asks using system call pipe( ) &, — arguments to store two value

Kernel will return 2 ends of pipes

```
int pp[2];
```

pipe(pp) — stored in array.

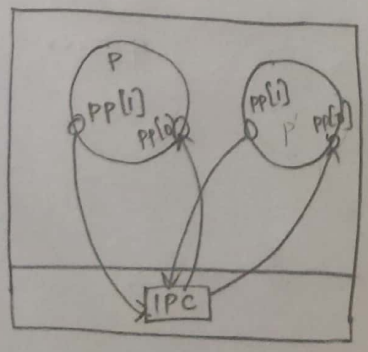
pipe is passive ⟹ identified by a file descriptor (fd)

pp[0] — reading ; pp[1] — writing

```
main(){
    char buf[20];
    int pp[2];
    pipe(pp);
    write(pp[1],".....");
    read(pp[0], buff, n);
    cout << buf;
}
```

→ We require pipe for comm. in between processes.

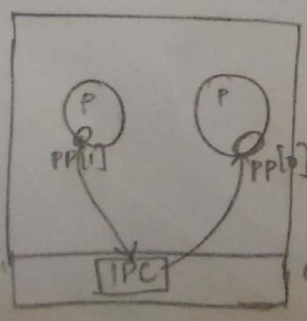
Create another process first




```
main(){
    char buf[20];
    int pp[2];
    pipe(pp);
    int c = fork();
    if(c>0){ // parent code
        close(pp[0]);
        while(write(pp[1]));
    }
    else{
        close(pp[1]);
        while(read(pp[0], buff, n));
```

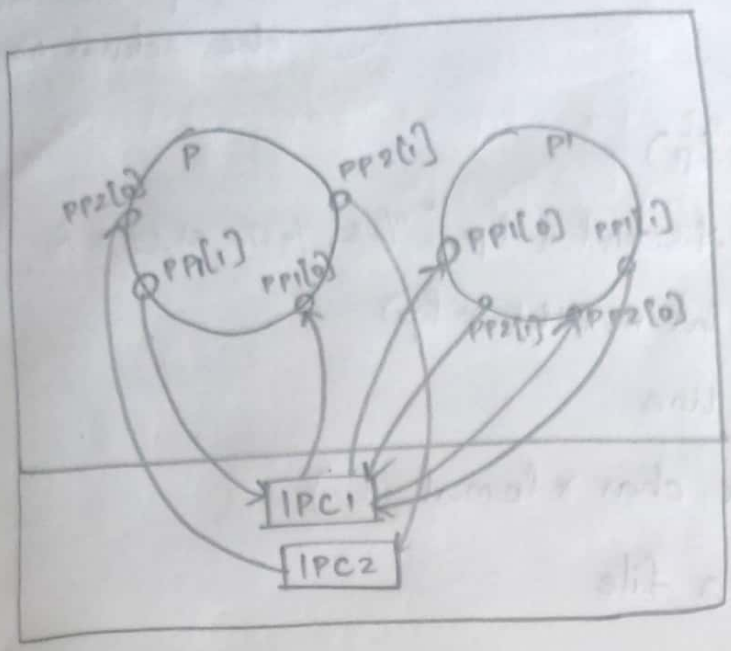
close reading end

close writing end

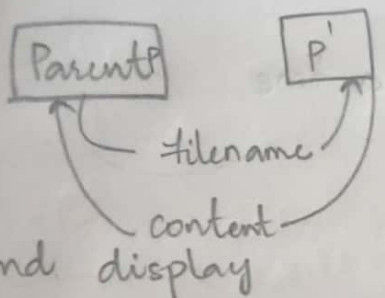


Max no. of bytes in pipe is 1024.

For 2-way communication – 2 pipes are required for each process (one for reading & writing)



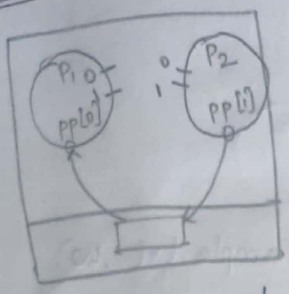

```
// parent code
   close(pp1[0])
   close(pp2[1])
// child code
   close(pp1[1])
   close(pp2[0])
```



Parent sends a filename and child sends back the content and parent displays it.

↳ First type of Client-Server

26/12/19.



Establish connection b/w 2 diff. processes

Give fd as command line arguments for P_2.

→ When 'exec' is called, P_2 erases child process in child but we require pp[1] for connection and fds

pp - variable in P_1 ; P_2 doesnot know

0 - Reading fd (Keyboard)

1 - Writing fd (Screen)

2 - Error

---

dup, dup2

~~dup2 dup(fd)~~

int nfd = dup(fd) — nfd is same as fd (points to same)

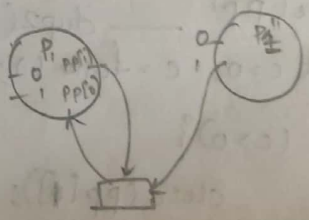
→ For every process, for every file, a file table is there

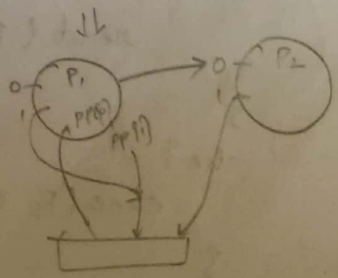
file table contains ① file descriptor(fd)
② Reading pointer
③ Writing pointer

dup2(old fd, new fd)

↳ old fd is duplicated to new fd.

For our program;



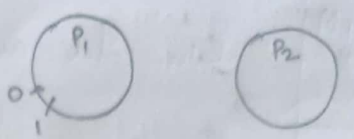



```
pipe(pp)
int f dup2(pp[i], i)
int c=0;
 c= fork()
 if (c>0){

}
else{
    exec("P2.exe");
}
```

Write 3 cout into file and then display on screen

```
main() {
    printf (fp,
     cout << " ";
     pipe (pp);
     dup2 (pp[1], 1);
     dup2 (fd, 1); fi
     cout << "   ";
     cout << "   ";
     cout << "   ";
     dup2 (nfd, 1);
      cout << " . . . ";
      cout << " . . . ";
      cout << " . . . ";
```




```
int fd = fopen ("sample.txt", w)
int nfd;
dup2 (1, nfd);
```

Read First store 1 into nfd

fd into 1 => cout on 1 (screen) becomes onto file

then returns back to screen

nfd into 1.

Output of P2 should go into P1

P1

```
pipe(pp) ____ dup2(pp[1], 1);
int c=0; c = fork ();
if (c>0){
        close (pp[0]);
        read ( pp[0], buff, n);
}
else{
        exec ("P2.exe");
}
}
```

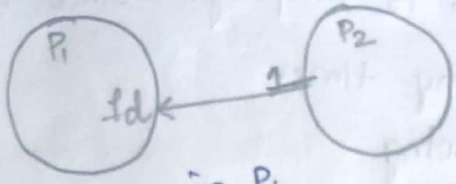
P2

```
cout<<" . . . ";
        (fd is on 1);
```

To get back again, special way is there

order: pipe, dup2, fork, exec

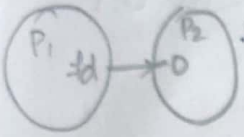
popen() — inside implements pipe, dup2, fork, exec



P1 req. standard o/p of P2
"
1

For this, write in P_1 `fd = popen("P2.exe", 'R');`
↳ reading type

P_1's output to fd should be standard i/p to P_2



`fd = popen("P2.exe", 'W');`

$ P1 ¦ P2 ¦ P3 ⟹ P1's standard o/p is P2's standard i/p.

for this `fd = popen("P2.exe", 'W');`
`dup2(fd, 1);`

P2's st. o/p is P1's st. i/p

For this, `fd = popen("P2.exe", 'R');`
`dup2(fd, 0);`

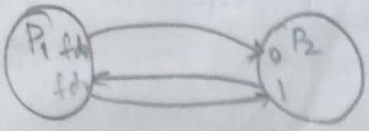
Q: P1's st. o/p to P2's st. i/p and vice-versa

```
fd1 = popen("P2.exe", 'W');
dup2(fd1, 1);
fd2 = popen("P2.exe", 'R');
dup2(fd2, 0);
```

H/w: $ tpipe $ P1 ¦ P2 ¦ P3 — implement prog.
semget
shmget

27/12/19

echo- server



P_1 sends, P_2 displays and reverse

↳ many times echoing

P_1 – reads from std i/p into fd1
output fd2 into std. o/p

P_2 – Create two fds and duplicate them to 0,1

* flush to duplicate and then do.

1. Echo P_1, P_2 with popen

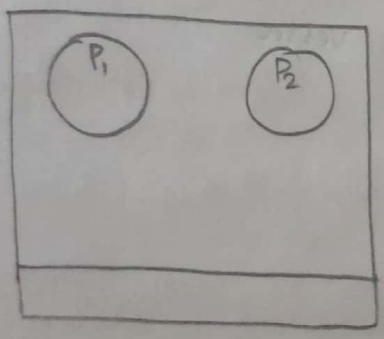
popen – expensive (includes 4 system calls)

1) pipe()
2) dup()
3) fork()
4) exec() — expensive

Disadv:

1) Comm. b/w related processes only
2) One-way comm.

FIFOs (Named Pipes) B/w
↳ Two not related processes

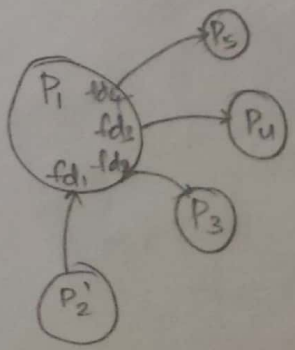


Communicating processes decide which named pipe to use.

→ Users name pipe
mkfifo ↳ pathname

→ With popen, one pipe – max 2 fds

2.



Try duplicating fd_3 with fd_2 and send to

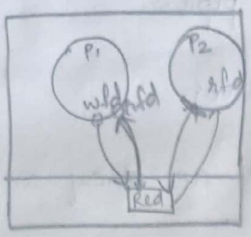
— Chat- Server

→ Can also be opened using 'mknode'
↳ opens as a device driver

mkfifo (char * argv, File mode)
pathname

mkfifo ("Red", —) — a pipe "Red" is reserved
↳ check.

wfd = open ("Red", 'W')



→ P_2 can again reserved
↳ OS checks before giving

P1 Code

```
mkfifo("Red", mode)
int wfd = open("Red", 'w')
w
cin>> buf;
write(wfd, ...)
```

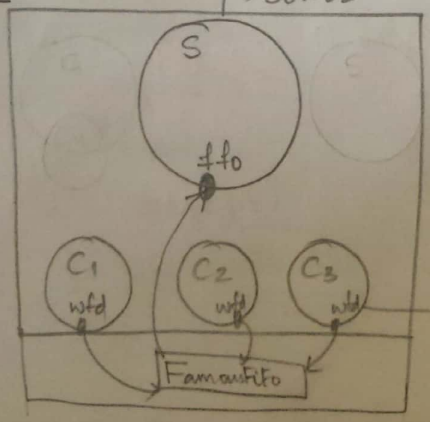
P2 Code

```
main(){
    rfd = open("Red", R)
```

P_1 can use it for reading purpose also
int rfd = open ...
But 'wait' because, synchronization is important.
Or else it reads whatever it wrote.

→ For synchronization, semaphores are to be used.

Chat - Server.



~~mkfifo~~ Let, pipe name = famous fifo
mkfifo ("Famous fifo", mode)
Clients int rfd = open ("Red", 'R')
Client assumes server already opens fifo, since it is req.

After reading by server, server should send to C_1, C_2 (msgs sent by C_3)

The server process has to send seperate FIFOs for messages of all clients.
No. of extra fifos = # clients.

No. of fifos for clients = 2

No. of fifos for server = # clients + 1

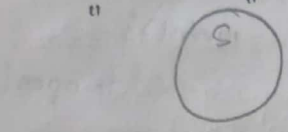
→ But for the extra fifos, the clients and server should agree on the name of the pipe.

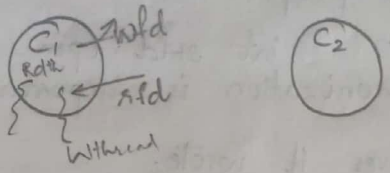
The client sends it 'pid' to server.

→ Server maintains 'pid' of all clients in an array and sends its pid through message.

③ Chat server using fifo

④ " " " " using threads





Code of client:

```
wfd = open ("ffifo ", 'w')
getpid();
// convert into string sts
mkfifo ("str", mode)
rfd = open ('sts', 'r')
while{
        cin >> buff;
        write (wfd, buf, n);
        read (rfd, buff, n); // If others do not send anything this will block
        cout << buff;
}
```

Client - should not assume continuous comm.

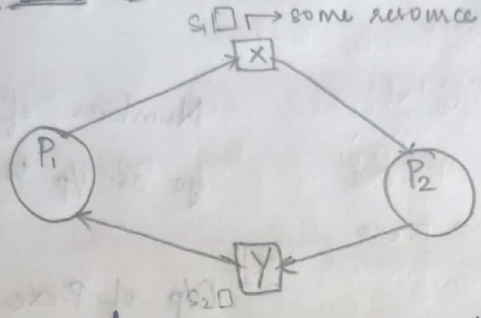
Two threads are to be used for the client

1) Reading
2) Writing

→ If any process has to do multiple things at a time, it requires threads.

→ If server wants to sends its own messages, threads are used. (donot use now)

Semaphores & Shared Memory



X and Y - not local to P_1 or P_2
↳ shared memory
√Sequence is to be maintained.

Two semaphores are required (S_1 & S_2) = (0, 0) - initially

```
P1()
{ while do{
    <compute A1>
    write (x)
    signal (S1);
    wait (S2);
    read (Y)
    <compute A2>
  }
}
```

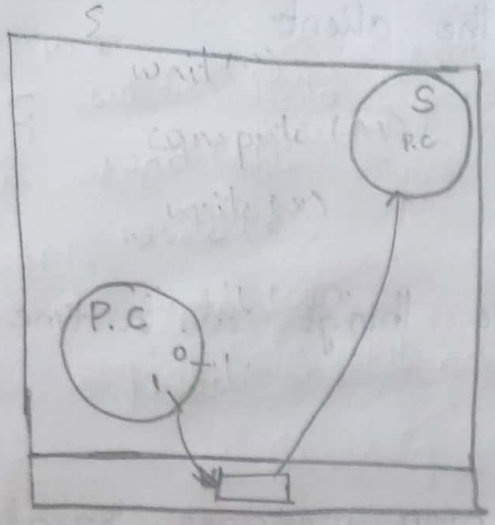
```
P2()
{
  wait (S1);
  Read (x);
  <compute B1>
  write (Y);
  signal (S2);
  <compute B2>
}
```

⑤ If S1=1, S2=0 — what should be the code?
↳ start with P_1 writing to X
S1 = 0; S2 = 1
S1 = 1; S2 = 1 — not possible synchronization

and implement:
P(S) = wait(s)
V(S) = signal(S)

S1 >1 ; S2 >0

P1()



Server has standard code PS outputs O_2

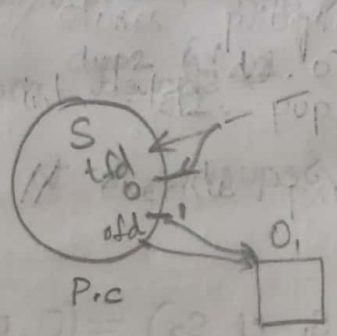
Test case = t_1

Output = O_1

checks O_1 & O_2 and evaluates

system ("cc p.c"); – generates P.exe.

```
int nfd1 = dup(1);
    nfd2 = dup(0);
int fd1 = fopen ("P.C", 'w');
int nfd = popen ("S.C",
```



PS.exe

P.exe

system ("cc P.c").

```
int nfd1 = dup(1);
    nfd2 = dup(0);
int fd1 = popen ("system(P.C)",
```

Numbers of t.txt should go as i/p to PS.exe

PC.exe

O/p of P.exe same in O_1

None fork and exec

P.exe will take i/p from t.t

o/p is given to Ofd

31/12/19

★ Signals

→ Notifications are because of signals.

30 signals / 60 signals – depending on OS

→ 28 out of 30 signals have predefined signals actions.

2 – can be modified (12, 14)

SIGUSR1, SIGUSR2 – user defined

SIGINT – Ctrl C

SIGALRM – Alarm

SIGKILL – Killing a process

→ A signal can be handled 3 ways: ① Ignored (SIG_IGN) ← function

② Default reaction (SIG_DFL)

③ Handler (special func. hdlfn())

Sender of signal:

① Kernel – sends to process (always) – $ kill -pid signo

② Process to process kill(pid, signo) kill -1 -pid signo ?

③ Self

↳ raise(signo) / raise(sno)

↳ function call

kill – it is not killing a process

↳ sending a signal to a process

↳ different from SIGKILL

→ $kill -L – displays all signals with signal numbers

Handling a signal:

1. write a signal handler function – hdlfn()

↳ can include anything, printf statements also

2. signal(signo, hdlfn)

3. If hdlfn() is for a group of processes, then

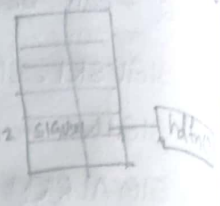
fctl() } – question in Minor 1.

ioctl()

```
void hdfn()
{
    cout<<"Hello Signal World\n";
}

main()
{
    signal (SIGUSR1, hdfn);
    cout<<" Checking signal";
    raise (SIGUSR1);
}
```

Interrupt Service table / Signal Handler table



O/p: Checking signal
Hello signal world

→ For SIGINT, we want to write hdfn()
But the actual code of Ctrl C which was pointed before is not pointing now.

```
void hdfn(){
    cout<<" You have pressed Ctrl C";
}
main(){
    signal (SIGINT, hdfn);
    cout<<" . . . ";
    raise (SIGINT); — If raise is not there.
}
```

Once, when ctrl C is pressed, cout statement appears.

If there is a while(1){} after cout<<" Checking". it is cout<<. . . forever in loop.

```
Void hdfn(){
    cout<<"Pressed 1";
    signal (SIGINT, hdfn2)
}
void hdfn2(){
    cout<<" Pressed 2";
    signal (SIGINT, hdfn3);
}
void hdfn3(){
    cout<<" Pressed 3";
    signal (SIGINT, SIG_DFL);
}
```

→ default action should be restored

→ Use a static variable to maintain count of no. of Ctrl C and Then call signal(SIGINT, SIG_DFL)

→ If nothing is to be done when pressed ↳ be careful

① write nothing in hdfn()

② write signal(SIGINT, SIG_IGN)

→ Two signals cannot be ignored.

Ctrl + / — SIGSTOP

→ Handler function can have 1 parameter — by value only.

```
void hdfn(int signo){

}
```

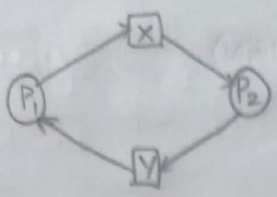
→ System calls — code executed in KERNEL.

→ Functions — executed in User space.

→ We have seen asynchronous I/O.

keystroke interrupt – write ^handler func.

Write a prog. to continue its process until you type something.



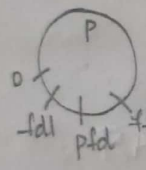
2 signals
KILL
shared memory
P1 should know P2's PID
Named pipe (FIFO) is required

① P1 will write into shared memory
P2 will read and write again
↳ Do this with signals.

kill all ().
↳ sends that signal to all groups

IO - Multiplexing:

└ Many to 1



→ Process can have i/p in any way (4 ways)
It can be solved by Mechanisms / Techniques
① fork
② Threads
③ Signals
④ poll()
↳ check everything

But signals are limited
└ not always possible

Create child processes
and each should look after one

→ But they are expensive.

event — Read, Write, reject
revent – reply event

```
struct pollfd{
    int fd; // which fd
    short event; // checking purpose
    short revents;
}
```

int poll (struct pollfd fds[],
n nfds,
struct time_t timeout

```
struct time_t {
    int sec;
    int microsec;
}
```

→ Poll returns something.

↳ Check for which fd

2. Try poll for keyboard and pipe

SIGCHLD – Kernel sends this signal to parent when child exits.

↳ By default – it ignores

02/01/2020

Computer-Networks

Computer contains CPU, Hard Disk, Main Memory, Screen, Mouse, KeyBoard

Essential parts: CPU, Main Memory

↳ it may contain OS

→ For a thing to connect to CN, it should have CPU, Main Mem

→ Story to software

Computer – independent and autonomous

Protocol – set of Conventions and Rules

Program = Data structures + Algorithms

→ http, ftp, tcp, udp – protocols – it is an executable program

→ It is called as protocol because it is special in which world-wide agreed set of conventions and rules are implemented in this for a purpose.

RFC – Reference For Comments

No.

↳ present for all protocols

People can suggest ideas

$ ftp in UNIX to send files

$ smtp – to send mails

↳ simple mail transfer protocol

1. If P_1 sends, P_2, P_3, P_4 should receive

Hint: fctl ioctl

2. Group chat server (use poll in server)

maintain 2D array with fd & pid

One Level Library

P.h

P1() P2() P3(){}{}

{ { {

{ { {

Two-level

D.h

d1{ d2{ d3{

include P.h P4()

P1() P2() P3()

{ } {

Three-level

n.h

n1()

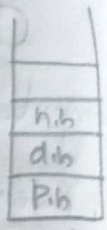
{

d1()

}

Can have any level libraries.

different layers —



→ We should have info. / reference to about the headers.

→ Each one is a protocols.

n.h can use d.h only.

It is a stack of protocols.

→ US defence came up with a model

ISOs, OSI Model.

They first named the model / reference book as

ARPANet (Adv. Research Proj. Agency Network

DARPANet (Defence ARPANet)

→ Computer Networking software is to be developed in 7 layers:

Application ↳ reference only, not implementation

Presentation

Session

Transportation

Network

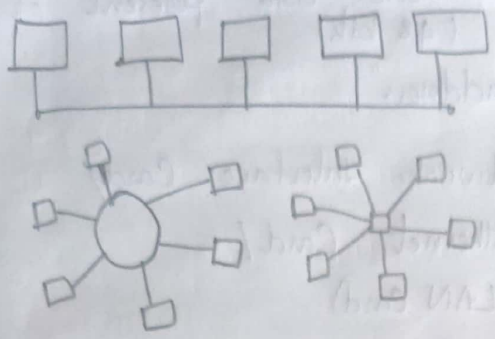
Data Link

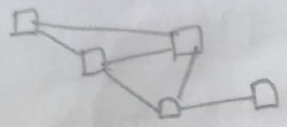
Physical

1. Physical Layer: (More of ECE part)

→ Connect two computers physically

1. Transmission medium (Twisted pair, Fibre Optic Cable)
2. Modem (Modulator, DeModulator)
3. Decide upon signals (Volt., Cur., Light, Freq. etc)
4. Encoding & Decoding (-0.5=0 ; +0.5=1) - prev. Now, -0.7=0 ; +0.7=1
5. Transmission mode (Simplex, Half duplex, Full duplex (99%))
6. Line Topology



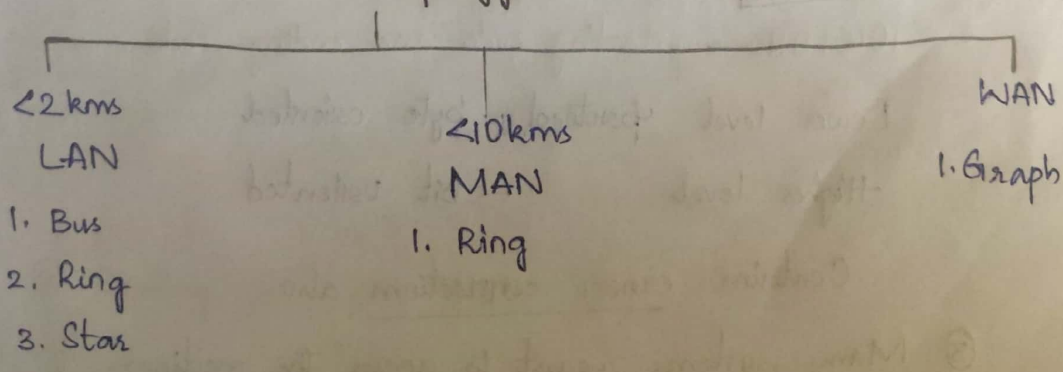


→ different types of connection

Different topologies

1. Bus
2. Ring
3. Star
4. Graph / Mesh / Irregular
5. Tree (Not used now)

CN Topology.



→ LAN in labs is a star model

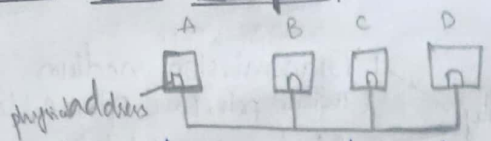
No. diff. b/w bus & star (technology - noise)

better and safe

03/02/19.

Data-Link Layer:

All of us with different faces decide upon one single lang, how talk, wake, talk, control flow & error control.



→ A system doesnot action any permission.

→ Likes names for peoples, addresses are present for systems (48 bits)

It is called "physical address"

① Physical addressing (48 bits) (Network Interface Card/ Ethernet Card/ LAN Card)

Eg: MAC address
Ethernet address
(IEEE address)
Hardware address

↳ unique for every computer.

Let physical address be 'A', 'B', 'C', 'D'

② Decide language (contains 0,1 and words with them)

People talk in sentences
computers talk in frame

Framing is done

10101011 — starting code and ending code

Lower level protocol — byte oriented
Higher level — bit oriented

Contains error connections also.

③ Many systems want to access the medium

Access control methods — decide which to access

- CSMA/CD — used for ethernet — IEEE 802.3 Ethernet
- Reservation — IEEE 802.4 — transmission ckt + sensing ckt — checks the medium
- Token Ring — used for real time LAN

Algo: Binary exponential back-off alg. — waited till everyone stops talking

Reservation was previously used by IBM.

But if out of 120 people, 2 and 119 wants to talk

2 has to wait for 118 sec

119 has to wait for 1sec + 2 mins } - after reservation

④ Synchronization

⑤ Flow control

⑥ Error control

⑦ Result: Node-to-node data transfer

→ Computers are called nodes.

~~Address~~ — identify uniquely.

If 202.141.49.75

Asia | India | NITW | Computer Number

0–63 - South Numbers

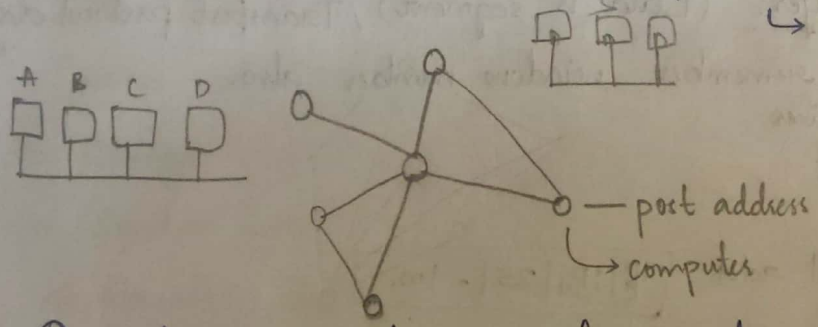
Network Layer: (data is packets)

① Logical Addressing (32 bits)

IP Address / Network Address / Internet Address / Sub Net Address / WAN (wide)

It contains 32 bits — First 8 bits, next 8 . . .

→ heirarchy is present



② Routing (Collects & classifies based on addresses)

③ Congestion control

④ Inter-networking

⑤ Address translation (ARP - Add. Resolution Protocol)

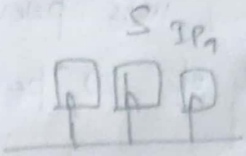
⑥ Multiplexing / Demultiplexing

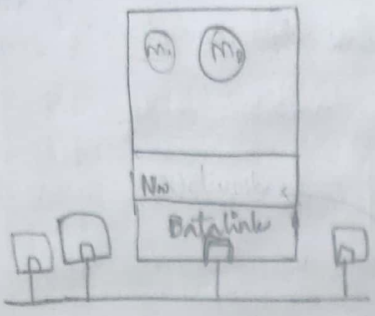
→ Physical addressing should be LA on letters.

⑦ Source-to-dest. data transfer

IP_5 - IP address of C

IP_7 - IP address of S

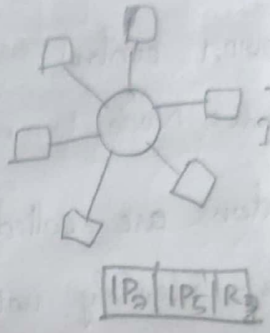




IP_7	IP_5	R_3

(m_1) message will come to n/w layer, it will prep. packet. to IP_7

(m_2) to IP_7



IP_3	IP_5	R_2

IP_7	IP_5	R_3
IP_3	IP_5	R_2

IP_5	IP_7	m_1

IP_3	IP_7	m_2

Message server will send / receive messages from systems.

Now, replies are present in n/w layers.

It should know which reply belongs to which

~~Transport~~

Transport Layer: (Data is segment) / Transpat protocol data unit

Message will remember window number also.

25	-	m_1

Receiver

Sender

Network layer will add

IP_5	IP_7	25	-	m_1

IP_5	IP_7	40	-	m_2

Now, replies will be:

to n/w layer

IP_7	IP_5	-	25	R_3
IP_3	IP_5	-	40	R_2

Now, n/w layer will cut off the head and sends

-	25	R_3

to transport layer

and then transport layer disperses.

① Port addressing (16-bits)

2^{16} - max length of data that can be sent over internet.

③ Segmentation and Reassembling

② Connection-oriented / Connection-less

④ Flow control

⑤ Multiplexing / Demultiplexing

⑥ Error control

⑦ End-to-end data transfer

→ Any connection in the world is uniquely identified by:

5 tuples / 4 tuples

① Local IP & Local Port, Foreign IP, Foreign Port

Local & Foreign ⇓ Source & dest.

From this window, system to specific window, system

5th one – Protocol

Heart of WAN, CN software is Network & Transport.

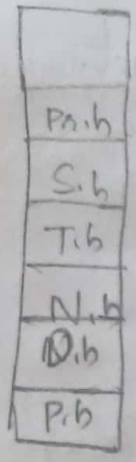
Session Layer: SPDU

1. Session management
2. Dialogue control
3. Synchronization
4. Graceful close

Presentation Layer:

1. Encryption & Decryption
2. Compression and decompression
3. Translation
4. Security

Application Layer:



Program is req. to send messages

This whole program can be written as command line argument (mail id, file name)

this is application program

```
#include "Ps.h"
main(){
  :
  :
}
port("mail ID");
prog();
prog(. . . .){
}
```

1. Electronic mail
2. Hyper media applications
3. File transfer
4. Network File System (NFS)
5. Remote LOGIN
6. Network management
7. Directory Services

Hostel – Host

Boys, Girls – Processes

Rooms – Port

Street – LAN

Leaders – Transport Layer

Postal service – WAN

Postman – Email server

Letter – Message

Post – Routers

Cost – Ro

HNo

07/01/19.

ISO's OSI Model (Reference)

→ Network & Transport Layer – important
(IP) (TCP)

Named s/w as TCP/IP
↓
Renamed as Internet → developed initially by US defence

Implemented model TCP/IP protocol stack

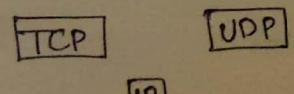
TCP/IP model has 5 layers.

A, P → Application
S, T → Transport
N Network IP
D Data → Ethernet IEEE 802.3 | Token R 802.5 | DQDB | Wireless 802.11
P Physical → ECE part.

Network Layer
RIP → Routing Info. Protocol – Routing
OSPF → Open Shortest Path First
ICMP → Internet Congestion Message Protocol
↳ Congestion control
BGP → Border Gateway Protocol
ARP → Address Resolution Protocol (IP Address → Physical)
RARP → Reverse Address Resolution Protocol (Physical → IP)
DHCP → Dynamic Host Configuration Protocol

Transport Layer
↳ Connection oriented (TCP)
↳ Connection less (UDP)
TCP → Transmission Control Protocol
UDP → User Datagram Protocol

IP Multiplexing



When packet received sent to TCP or UDP, IP has to decide
↳ IP Multiplexer.

Application Layer

(TCP) SMTP → Simple Mail Transfer Protocol

(TCP) FTP → File Transfer Protocol

65,536 → max. size sent through Internet (2^{16})

(TCP) TFTP → Trivial File Transfer Protocol

(TCP) HTTP → Hypertext Transfer Protocol

NFS → Network File Server (developed from RPC)

(UDP) RPC → Remote Procedure Call

(TCP) TELNET → #Remote login / Teletype network

(UDP) SNMP → Simple Network Management Protocol

(TCP & UDP) DNS → Domain Naming Service (both)

(ASCII name to IP-Address)

HTTP:

Connection-oriented (TCP)

→ Maintain connection b/w links & hyperlinks

FTP:

Large files need to be maintained

DNS:

Iterative & Recursive DNS

↓ (near) ↓ for far (connectionless)

TFTP → Trivial File one attempt it can send all files

So, UDP

SMTP → when email sent, if email wrong message received because connection not established

→ Attachments sent through mail needs connection

→ SMTP extended with 5 new header and renamed as MIME

Multipurpose Internet Mail Extension

→ Internet user need only

App. layer → All

Transport layer → All

Network Layer → IP ARP

only these required

↳ Client part
Answering part

Related to router and all not required

i.e., why transport layer has only 2 protocols needed remaining loaded onto other parts of system.

Message Queues

IPC_Permission (IPC-permission)

```
{
    time_t c
    msgq_hdr
}
```

msgq_hdr

```
{
    int msg_id;
}
```

msg

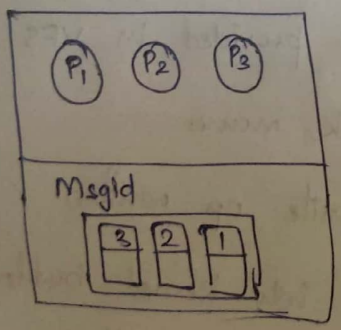
```
{
```

of msg in msgqs

① type ② Length ③ Msg Address ④ Pointer

type → to identify a particular message

①



creating a message queue:

```
key_t key;
key = ftok(" ");
msgid = msgget(key, ' ');
msgsnd(msgid, buffer, n, flag)
msgrcv(msgid, buf, n, type, flag)
```

if(type==0) first message receive

+ve → First message of that type received

-ve → first message of type value less than absolute value of type received

```
struct msg_buf
{
    int type;
    char msg[105];
}
```

→ To print no. of messages, use msg_qnum

type value → Best value can be process ids

→ Basically a queue, a msg added at last based on 'type', FIFO can be done

② Chat Server using message queue

S → Reads with type value 0

C → put msg with their IDs.

10/1/2020

Advanced Advantages

Linux Device Drivers

LINUX Device Driver (640 pages)

Disclaimer: Kernel modules are dangerous and can render a system useless by corrupting hardware as well as software, not responsible for any damage.

DE Shaw - Madhukiran (Junior)
Deputy CEO

① Srikanth,
② Ranjan (Nvidia)
③ Rishikesh Goyal (File device driver)
④

Kernel → open a device read

VFS → Virtual File System

↓

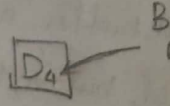
provide generic names, generic operations that can be defined on a device is provided in VFS

VFS

H/w D1 D2 D3

Devices — printers, keyboards, hard disk, mouses.

If a device (D4) added, read & write op written



Write into kernal buffer and it goes to device.

Users are not allowed to use kernel buffers.

So, a kernel module is to be written.

↳ this is to be added to kernel.

Write read, write into module

↳ VFS knows only these

Generic 'read' should behave as you wish.

Open the device ⇒ module active

Clock – put a device driver to it

↓

it will take time and updates it.

Linux File System:

→ Almost everything is file in LINUX.

→ Modules: (LINUX drivers written outside kernel)

Virtual → (loop devices)

Physical → H/w component

Bus-based

Types:

① Character devices

② Block devices

③ Network devices:

– users can't directly transfer data to n/w devices

– comm. indirectly by opening a connection to kernel's networking system.

Device driver interface

→ 21 sys. calls in VFS

File op. structure: Not all initialized, only those corresponding by device driver.

```
struct file_operations fops = {
        .read = device_read
        .write = device_write
        .open = device_open
        .release = device_release
};
```

① User asks kernel to execute a module once (Hello world prog.)

② Clock (set diff time)

H/w clock, s/w clock

s/w clocks copies time from n/w clock.

Header code.

*read = readme;

↳ doesnot have any functionality

'readme' is assigned

↳ kernel module

→ Every device will have 'major' numbers.

static int opendev (struct inode *, struct file *);

↳ user written kernel module.

→ All harddisks have same 'major' device

→ We assume, unique devices.

```
static init Tick_init(void){
        Major= register_chrdev( 0, DEVICE_NAME, &fops);
```

↳ charac. device is register.

0 — give some no.

if 25 — give 25

```
printk(" ... ", Major);
```

↳ printkernel

```
}
```

↳ overloading init

```
module_init ( Tick_Init);
```

↳ overloading init

```
static void Tock_exit (void)
{
    unregister_chrdev (Major, DEVICE_NAME);
}
module_exit (Tock_exit);
```

↓ should be written for sure

```
MODULE_LICENSE ("GPL");

static int opendev (struct inode* a, struct file*b){
    printk(" ");
    return SUCCESS;
}
// same for closedev
module_open( opendev);
```

Random utility func:

int pow(int a, int b);

Clock gives charac. (not numbers)

static int itoa(int val, char** ca);

int atoi(char * buff)

static ssize_t vreadme (struct file *flp, char *buff, size_t length, loff_t *offset){

do_gettimeofday(&time)

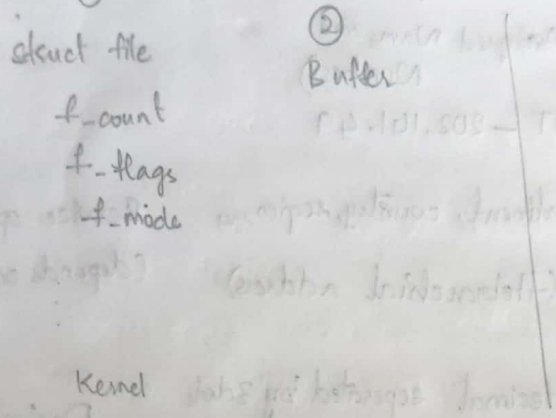
↳ kernel defined func.

char *b

copy_to_user(buff, b, len);

↳ copies kernel to user.

copy, read



ticktock.ko — generates kernel object file (by Make file)

↳ should be inserted into kernel module area

↓

for this, go to super user su

Insmod ./TickTock.ko — Insertmodule

Dmesg | tail — Kernel shows the printk

↓

this is req. as all printk will be written into kernel output buffer

Mknod /dev/TickTock.c Major 0

↳ should go into the given path

Now, write a user prog. and use the overwritten module

with drivers

cell drivers

To print "Hello World" in init, printk("Hello World");

3 files: TickTock.c -module

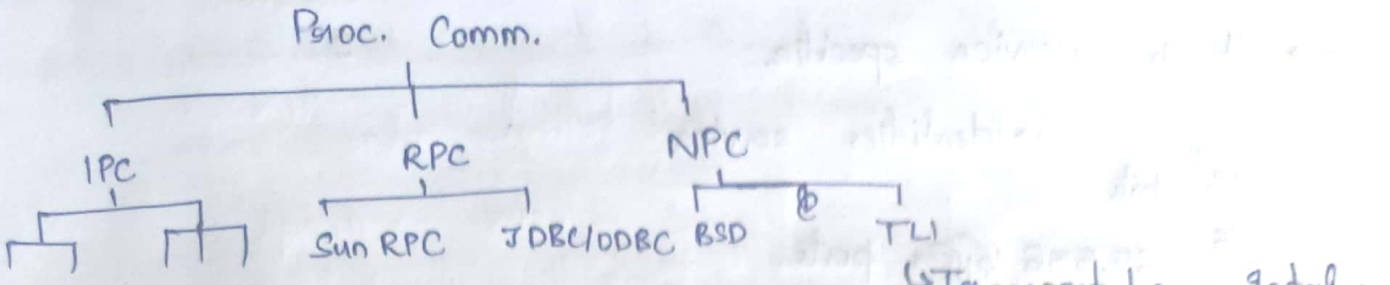
Makefile

User prog.c

Addresses

	Physical	Logical	Port
	(LAN, IEEE, h/w, MAC, machine)	(IP, Subnet, Internet)	(Service number)
Size:	48 bits	32 bits	16 bits
Layer:	Data Link	Network	Transport
Owner:	IEEE (gives first 24 bits) A company can manufac. 2^{24} numbers	US Defence (prev.) ICANN (Internet Corporation for Assigned Names and Numbers) NIT – 202.141.49	Transport layer
Nature:	Nothing can be made out of it (Flat address)	Continent, country, region, yaa (Heiranchial address)	Service specific (depends on email, http (sc
Expressed:	12 hex digits	4 decimal seperated by 3 dots Dotted Decimal notation	Decimal
Storage Area:	In the NIC (Network Interface Card) Burnt onto the chip	Hard Disk	Main memory
Nature:	Permanent	Semi-permanent	Temperosy

17/01/2020



↳ Transport Layer Interface

Networked Process Comm. (NPC):

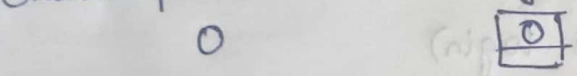
Two types: ① BSD Sockets

- BSD sockets
- Domain Unix Sockets (UDS)
- Raw Sockets
- DataLink Layer Interface (DLI)

Internet => Transport Layer

Two types of connections: Connection-oriented
(in BSD) Connectionless

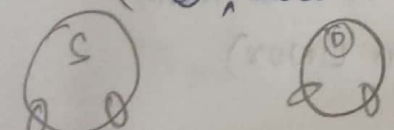
Client process & client system



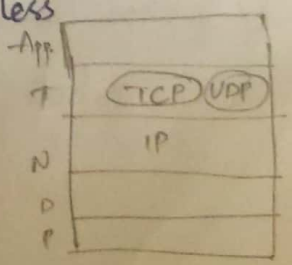
A connection b/w sys. is uniquely identified by:

(Protocol ①, Local IP, Local Port, Foreign IP, Foreign Port)

Local, Foreign
" "
Source, Dest.



Protocol — determines conn.-oriented / conn-less



App. Layer prog. — if worthy becomes protocol.

Source codes: http, smdp

App. prog. written is TCP / UDP? — determined by Protocol ①

PORTS

→ It is service specific.

↳ identifies service

16 bits

2^{16} = 65,536 ports

Networking software – receiptionist like in Star-Hotel

Some windows are reserved for some special purposes (0–511)

25 – SMTP (Simple Mail Transfer Protocol)

80 – HTTP

69 – Finger (enquiry)

20,21 – FTP

22 – TELNET (for remote login)

512–1023 – can be reserved

↳ for general purpose not personal

↳ Protocols exist

Two ways of allocating : Give a particular room

Give some random room

Similarly, 1024–6500 – check (depends on LINUX)

↳ allotted automatically

6500 – 65,535

↳ asked by user

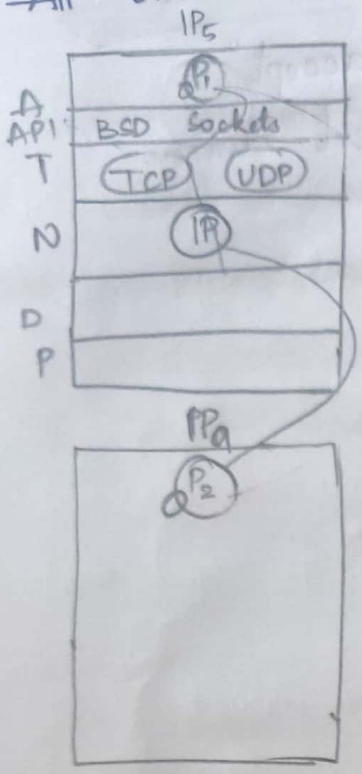
↓

Use roll no in LAB

BSD Sockets

Micro Systems, IBM – doesnot use this

All these do n/w prog. in this.



World's largest complex s/w – TCP
contains lower level n/w code

To understand better,

to get connected from one port of P_1
to another port of P_2.
(connect – systemcall)

P_2 – should accept (accept – sys. call)

send, receive – sys. calls

Some API (Application Prog. Interface) should provide functions (connect, accept, send, receive)

→ This API is BSD sockets. (s/w functions)

In P_1 – #include <sockets.h>

⇒ Learn all BSD sockets func. with arguments

These are executed in Kernel ⇒ so called system calls.

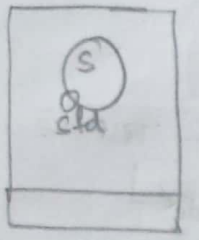
Socket system calls (<10 – Normal coding
<30 – Top Coder)

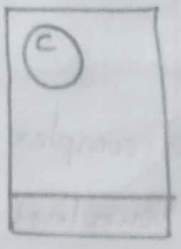
P_1, P_2 → client, server

↳ client can be server also.

Server Client

3 ways:
Diag. — ques. given
Flowchart
Pseudo code
↳ Mini 1





client contains "connect"
Server → "accept"

Counselling — easiest business

sfd - socket fd Min. inv. — Table & chair
↳ these are types of allotting a table

socket — sys. call for t p

int socket (int family, int type, int protocol)

↓ It is sfd
represents a conn.

family
↓ company
Internet (AF_INET)
AF_UNIX

Conn.-oriented — SOCK_STREAM
Conn.less — SOCK_DGRAM

Protocol depends on type
↳ so default (based on type)
automatically selected
So, put NULL / 0.

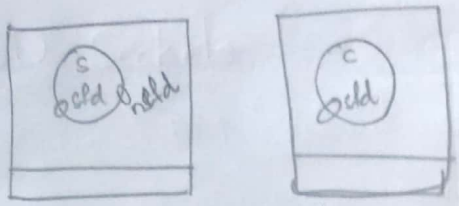
int bind (int sfd, struct sock_addr *myadd, int size) ~~int flag~~

Taking ↓ almost in everything
table & put in room in building
Port IP

struct sock_addr
↳ system defined socket address
↳ contains IP and port
Assign IP and port initially

Now Protocol, Local IP, Local Port completed

int listen (int sfd, int n)
↳ reserving waiting area
↳ reserves 'n' waiting buffers



Socket is waiting for clients.

Client can also receive services from many.

Client should establish "connect"

connect(int sfd, struct sock_addr *servaddr, int size)

↳ contains serveraddress.

In client side, protocol filled - through socket

foreign IP, foreign port - through connect

Local IP, Local port - already known

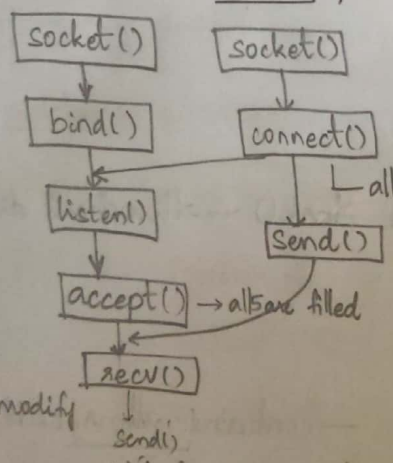
↓

Networking slw will take IP from Hardware

Iterative TCP allocate port numbers.

Connection - Oriented Client-Server

(Sbla) Server (sc) Client



all 5 are filled

=> write + flag

→ all 5 are filled

Net_func.() - contains nsfd

When server accepting the client. The Assume TV repair.

The server takes TV and puts that on another table where repairs are done. This will happen on "accept".

So, accept sys. call returns a new sfd (nsfd)

Now, connection / TV is on nsfd

int nsfd; int accept(int sfd, struct sock_addr *clientaddr, int *size); (cpp &size)

Here foreign IP, foreign port filled in Server. (c)

It will get the address of client and the size is determined by client address

So "& size" (c - *size

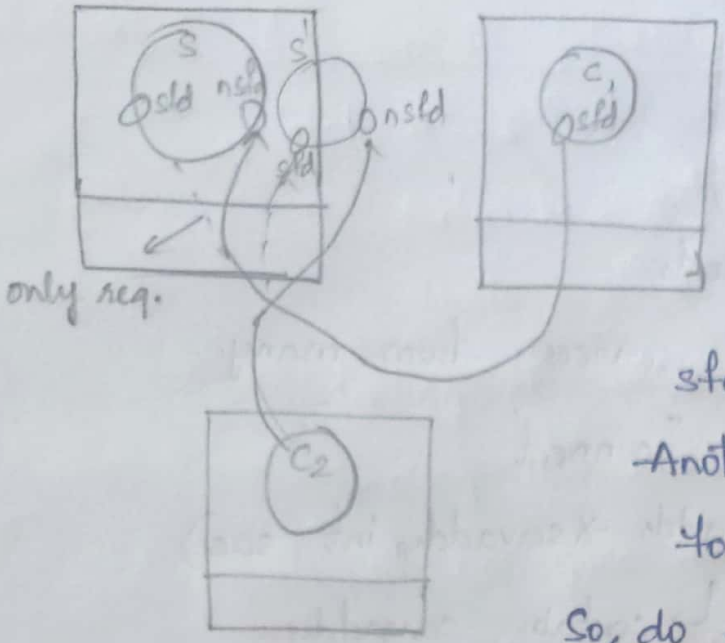
↳ everytime we say get.

send(int sfd, char *buff, int n, int flag)

recv(int nsfd, char *buff, int n, int flag)

↓

It is your choice of how much to read.

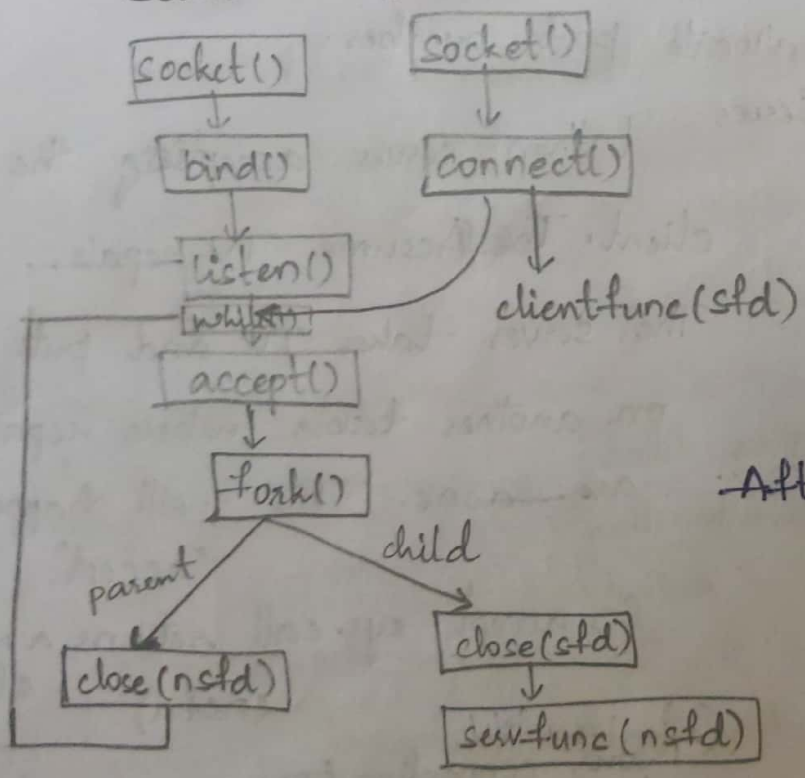


sfd is only working on "nsfd".
Another client has to wait
for sfd to complete working on c1

So, do fork()

```
if (c>0) {
        close (nsfd);
}
else    close ( sfd);
```

Concurrent connection oriented client-server.



Fork() should be after accep

After fork() — flowcharts divide

— contains all receive, send...

If 'n' client, then 'n+1' process

1 sfd : all nfsds client serving.

21/01/2020.

202.141.49.75

IP Addresses

32 bit – heirarchial address
uniquely identifies computer system.
Dotted decimal notation ____.____.____.____

32 bit Bits 1010 1111 0101 1000 0110 0111 0110 1101
8 Hex digits A F 5 8 6 7 6 D
AF.58.67.6D
10+16+15
Decimal Number 175.88.103.109 (4 decimal seperated by 3 dots)

Telephone no. | Line No. | Telephone No. |

⇩

| LAN no. | Host |

IP address contains 2 parts (① Network address
② Host Address)

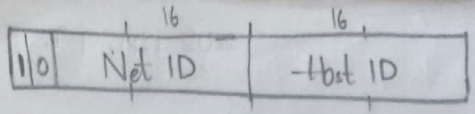
ICANN – Internet Corporation Assigned Names and Numbers.
(US – non profitable)

IP address classes:

Class A: | 0 | Net ID | Host ID | 32 bits (always)

Network ID – 8 bits (starting with 0)
Host ID – 24 bits
2^7-1 – class A type networks (starting with 0)
(All 0's excluded)
With each network – $2^{24}-2$ hosts (all 0's & all 1's)
Starting address – 0.0.0.0
Ending address – 127.255.255.255
It is domain IP address (.com, .org, .edu, .mil
It is generic domain IP address
Total IP address – 2^{31}

Class B:



Total IP = 2^{30}

Networks = 2^{14}

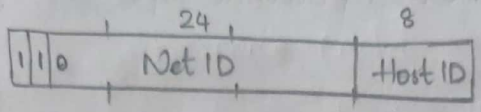
Hosts = $2^{16} - 2 = 65,534$

Starting IP = 128.0.0.0

Ending IP = 191.255.255.255

It will be given to world famous large companies, online companies (Microsoft, Sun Network, Satyam Online

Class C:



No. of networks = 2^{21}

no. of hosts = $2^8 - 2$

Total IP = 2^{29}

Starting = 192.0.0.0

Ending = 223.255.255.255

It is given to world famous universities & world famous small companies.

192-196 – America

-200 – Europe

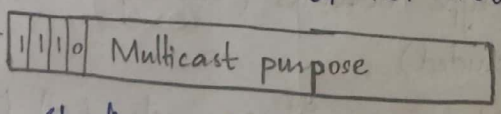
-210 – Asia

141 – India

49 – Region (0-63 - South

64-127 - North)

Class D:

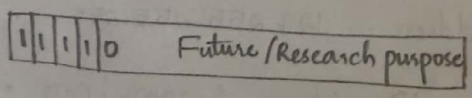


– used for US Defence

Starting – 224.0.0.0

Ending – 239.255.255.255

Class E:



Starting – 240.0.0.0

Ending – 247.255.255.255

Convert IP ____ → to binary

0 – Class A

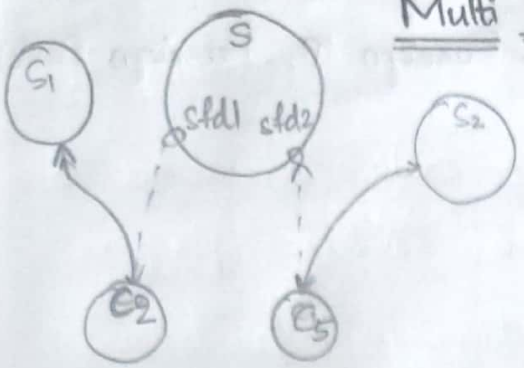
10 – B

110 – C

1110 – D

11110 – E

Multi Service Server



Many – One => poll

Server

socket()

Server

~~socket (sfd1)~~ int sfd1 = socket()

~~socket (sfd2)~~ sfd2 = socket()

~~bind~~

~~[int sfd1 = socket()~~

~~int sfd2 = socket()]~~

S

sfd1 = socket

bind

sfd2 = socket

bind

listen

while{

poll();

if (sfd1){

nsfd = accept()

fork()

if (c>0){

close(nsfd);

}

else{

close (sfd1);

close (sfd2);

dup2(nsfd, 1);

dup2(nsfd, 0);

exec(s1);

} – Read & write

}

}

}

MINOR 1
MID

(Protocol, Local IP, Local Port, Foreign IP, Foreign Port)

S	C
socket()	socket
bind	connect
listen	getsockname
accept	

? = char * getsockname (sfd, sock_addr) — returns socket name.

can be used after bind & connect. S & C Local IP & local port

To get from which client, which port request has come,

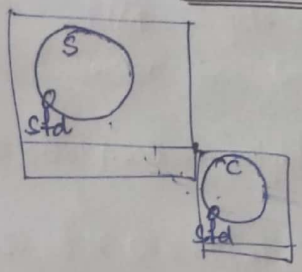
getpeername (nsfd, sock_addr)

↳ returns foreign IP, foreign port

↳ after accept (server side)

connect (client side)

Connection-less Client-Server



Without connection established,

something is sent, received.

"sendto"

int sfd = socket(AF_INET, SOCK_DRAM, 0) — protocol filled

bind(sfd, struct sock_addr *myaddr, int size)

sendto (sfd, char * buff, int size, struct sock_addr *to

↓ Foreign

sizeof(sockaddr),

_____, _____, flag)

Local IP Local port

Client side

sfd = socket (AF_INET, sock_DRAM, 0);

bind (sfd, sock_addr, size)

recvfrom (sfd, char *buff, int n, sock_addr *fromaddr, int &addrsize, int flag)

getpeername — after recvfrom

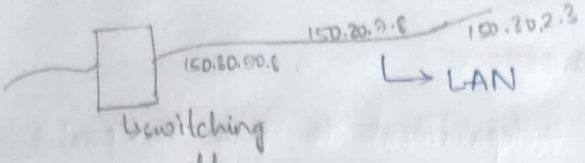
22/01/2010

Subnets

Class B IP Address:

150.40.—.—

↳ given by ICANN



Checks for first 16 bits and sends the connection.

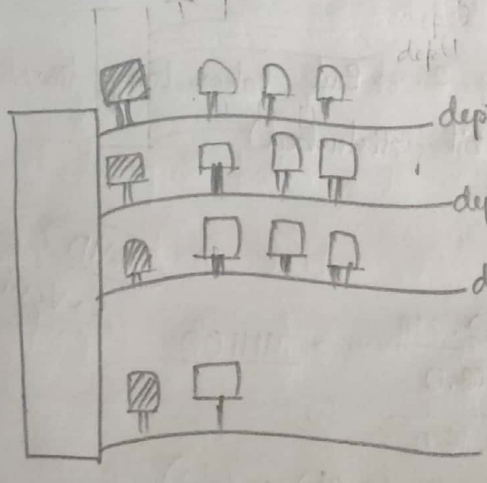
→ For a LAN connection, in a single line — max. computers that can be connected is 1000.

⇒ Through ethernet — max. 10000.

For class B, 65,534 ($2^{16}-2$) systems are present.

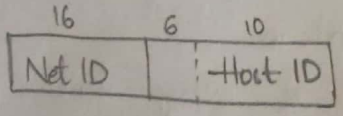
Checking 65,534 everytime — time consuming.

→ So, instead of connecting all systems onto a single cable, connect to more than 1 (for class B IP Addresses)



Every dept. will have special switching device

Main switching element takes care of the first 16 bits and will checks the department.



58 dept. → taken by you

Dept ID - 6 bits (∵ 58 dept. $2^6 = 64$)

In general, "subnet" is Dept ID. = 6 bits

64 may be there. (Take nearest number)

1024 systems present in a single dept.

⇒ Dept1 starts from 150.30.4.0

150.30.0.0 to 150.30.3.255 — (∵ 4 256's are there = 1024)

4.0 to 7.255

8.0 to 11.255

⋮

150.30.228.0 to 150.30.231.255

Interface — present in every department to pass the IP packet

Starting Address	↳ Comparator
150.30.0.0	Interface 1
4.0	Interface 2
8.0	Interface 3
⋮	
228.0	Interface 57.

150.30.17.25 — goes to interface 4

17/4 ⇒ gets the interface

But division is time consuming.

150.30.17.25 — right shift by 2 ⇒ But takes long time

↓ (all 32 bits disturbed)

150.30.16.0 — should be changed

AND, OR, XOR — simple

MID
↳ Version 3.

So, to obtained this, 150.30.17.25 → 11111100

AND 150.30.[illegible].0

150.30.16.0

Subnet Mask — 255.255.252.0 (22(1's) 10(0's))

∵ NetID + Subnet ID = 22 bits

Host ID = 10.

→ Comparator will have starting IP address of every line.

For 29 depts ⇒ 2^5 = 32

subnet address = 5 bits

16 + 5 = 21 1's — NetID + Subnet ID

11 0's — Host ID

NOTE: If no. of systems in a subnet is 'x', then the IP addresses are multiples of 'x'.

150.30.0.0/22 — representation of subnet IP address

↳ no. of bits used for representing networking

If subnet mask number is 24 ⇒ 24 bits for networking

2^8 – hosts ID.

→ Sub-groups can be present for subnets – Version 2.

→ If subnet mask is 255.255.252.0

a) 150.30.75.48 1024

b) 150.30.60.20

c) 150.30.47.22

d) 150.30.78.70 /22 ⇒ a, d - same subnet

21 ⇒ subnet = 8 5 bits

Host = 11 bits ⇒ $2^{11} = 2048$

. — . — ↳ 256 ⇒ 11 − 8 bits ⇒ 3

$2^3 = 8 \times 256$ – each size of subnet

/19 ⇒ each group 2^{13}

⇒ Multiples of 32.

∴ b, c ; a, d — same groups.

Supernet: CIDR – Classless Internet Domain Routing.

→ Class C – has only 256 host IDs – very less

Class B — 65,536 – very large

Some special universities (CMU) require 2000 departments.

Stanford	4000
MIT	1000
NIT	500
Boston university	1500

So, on request from the universities:

Class B – 160.70.0.0 – free (not allocated)

So, allocation is done in multiples of 256

CMU – 2000 req. — 2048 allocated ⇒ 160.70.0.0 to 160.70.7.255

Now, from 160.70.8.0 — available

Stanford – 4000 req. – 4096 allocated ⇒ 160.70.16.0 to 160.70.32.255

∴ $16 \times 256 = 4096$ i.e. 16 256's are present, so starting address is multiple of 16.

Though 8.0 – available – not used

∵ Only multiple of 16 required for starting address.

MIT – 1000 req. – 1024 allocated → 4×256 ⇒ multiple of 4.

160.70.8.0 – 160.70.15.255 } – available

160.70.32.0 – onwards

∴ Allocated are 160.70.8.0 – 160.70.11.255

↳ multiple of 4.

Available: 160.70.12.0 – 160.70.15.255

16.70.32.0 – onwards

NIT – 500 req. – 512 allocated ⇒ 2×256 – multiple of 2

∴ 160.70.12.0 – 160.70.13.255

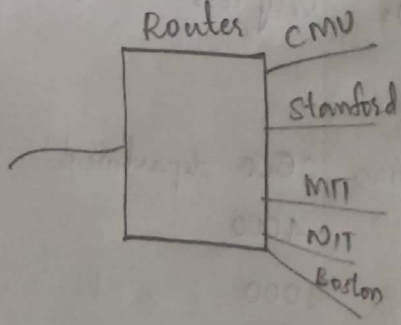
Available: 160.70.14.0 – 160.70.15.255

16.70.32.0 – onwards

Boston – 1500 req. – 2048 allocated – 8×256 ⇒ multiple of 8

∴ 160.70.32.0 – 160.70.39.255

∵ The allocated addresses are not of same length, then it cannot be called "subnet". It is called "Supernet".



So, subnet masking should be there for the departments.

CMU – 2048 = 2^{11}

11 – zeros

21 – 1's

↳ Starting Address

— 160.70.0.0 /21 — Subnet masking

Stanford — 160.70.16.0 /20

MIT — 160.70.8.0/22

NIT — 160.70.12.0 /23

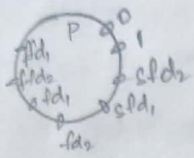
Boston — 160.70.32.0/21

I/O Multiplexing

I/O multiplexing

Asynchronous I/O → (signal)

Asynchronous I/O: It allows a process to tell the kernel to notify it whenever a prescribed file descriptor is ready for input-output I/O.

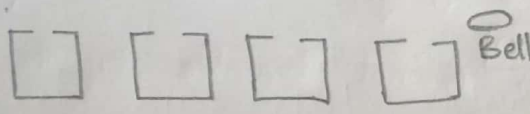


I/O multiplexing ways:

① fork()
② threads()
③ signals
④ poll()
⑤ select()

reduces disadvantages of above

Limitation of poll – checks continuously even if no work done in that. It is a waste of time.



The request can be put in "box":
if it is to be immediately handled – press bell => like signal
else just drop request and leave
So, no time wasted like "poll"

→ It can behave depending on the situation.
read, write, error – things to be handled

→ It has more than 6 supplementary calls are to be there.
To use "select", first 'boxes' should be there

fd_set → set of fds

FD_SET(sfds, .name)

sfd wfd – read & write fds

First, boxes are to be set free => FD_ZERO – initialize to zero

int select (int nfds, fd_set *rfd, fds *wfd, fd *
struct timeval *timeout)

if timeval >0 – poll
=0 – signal
<0 – come if check

if "select" returns true, go and check

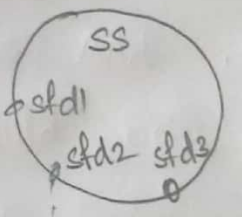
FD_ISSET – checks if request come
(not reading / writing)

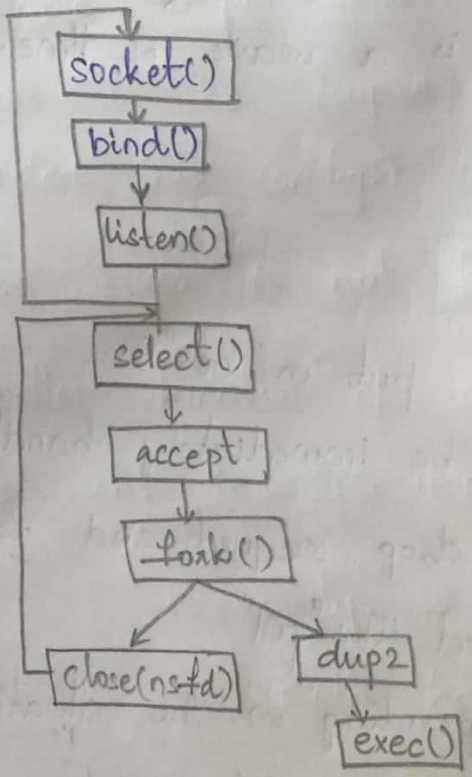
After reading – make FD_ZERO

At a time, one request can be present.

Prepare boxes – FD_SET – rfds, wfds
FD_ZERO
IS_ZERO –?

"Superserver" – if a server serves more than one
(SS) else called "Service server".





28/01/2020.

inetd - Super Server.

inter-net demon program. Mid

↳ runs in the background.

→ If system is connected to internet, inetd process will run in the background.

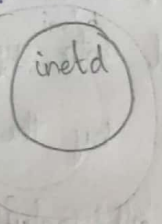
Super server — it will have executable code for all services.

Similar to super server prog. (multiple services)

/etc/services /etc/config

	Portno.	Protocol	func.type	exe file
0-511 — reserved port no.s services	25	TCP	external	/.../smtp.exe
	21	TCP	external	/.../ftp.exe
	80	TCP	external	/.../http.exe
	22	UDP	extern	/.../tfdp.exe
	69	UDP	extern	/.../finger.exe

func. type — extern, intern, thread



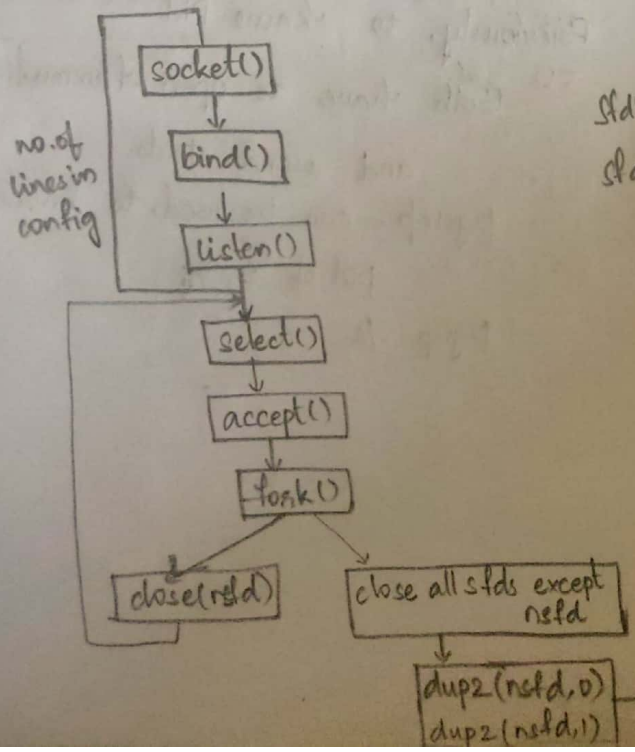
512 services present. ⇒ 512 sfds required.

But all are not created.

No. of sockets created = no. of lines in config file

Based on SOCK_STREAM (or) SOCK_DGRAM ⇒ TCP, UDP decided

If many IP address present, bind with main IP-Address.



extern – when request comes, it will look and external process created which takes care of it

thread – thread func. is available and it will serve. netd is passed

Mid.

→ Creating seperate process for every service is not optimal. Sometimes, super server will ask the service server to be threaded.

It is called "Threaded Service server". —✗

Internet services will be "threaded services".

→ For some small connections (where a client asks for time) the super server will take of it (no need to create a seperate process / threads for it).

inetd – already exists

If "thread" s_2, the super server should present "service thread"

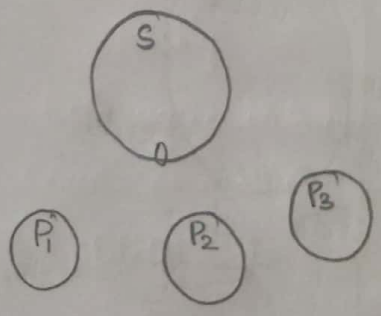
Dynamically Configurable Super Server:

To Add services to the super server, send signal to (should be appended into config.) add into superserver.

→ Duplicating super server is not bearable by the KERNEL.

The signal handler function of inetd will include the ser

The signal to be sent is ~~SIG_INET~~ SIG_CONFIG (special)

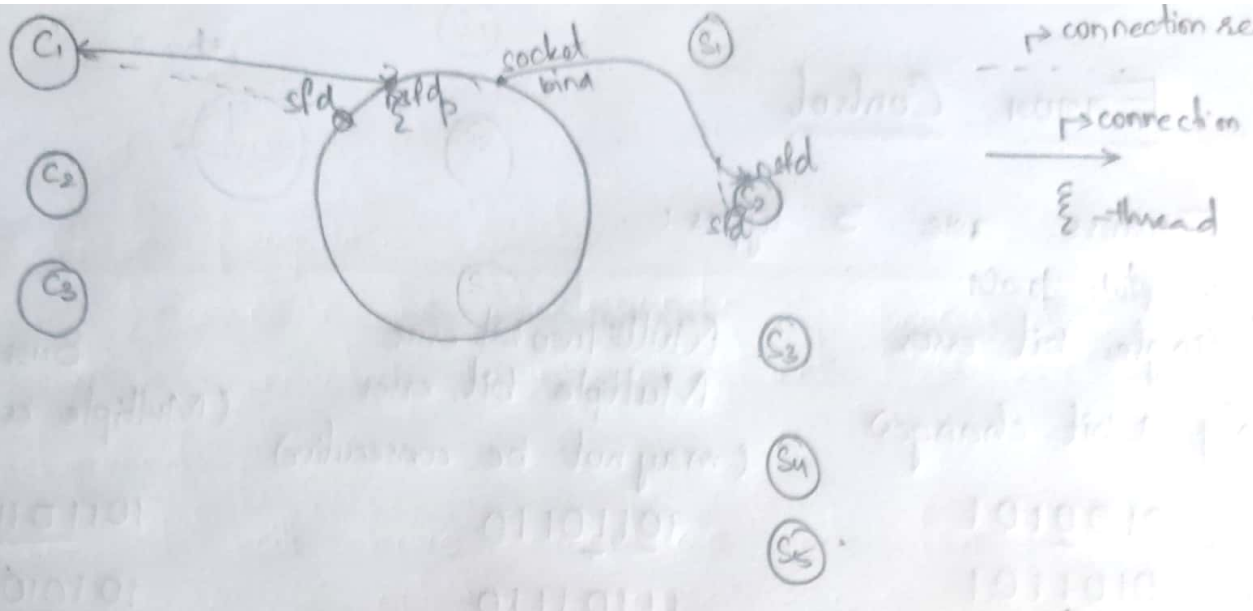


Previously, to know Pid

Both have to open famous fifo and send pids.

pgrep – can be used to know the pid of s by

pgrep ./s



→ Every time the proxy server has to send data that it got from server to client ⇒ connection-oriented.

A thread is created (nsfd) and asks for the service to be done.

Here, thread is created for every client.

But, if only 5 threads for 5 services.

Proxy itself will ask which server is required and the nsfd is added to the array and poll.

NOTE: Dup2 doesnot work with sfd

Error Control

Errors are 3 types:

- Single bit error (only 1 bit changes)
  - 0100101
  - 0101101
- ~~Two bit error~~ Multiple bit error (may not be consecutive)
  - 1011 0110
  - 1110 1110
- Burst error (Multiple errors in consecutive)
  - 1011 01110
  - 1010 10010

Error Control

- Error Detection (Some errors can be detected but cannot be corrected) they have to be asked for sender to send again
- Error Correction (Correction of detected errors)

m - message in bits

→ For any error control mechanisms, along with message bits, redundant bits are to be added.

Error Detection (Methods)

1. LRC
2. VRC
3. CRC
4. Checksum

~~Based on: Parity~~

Method: Linear Redundancy Check

1. LRC

Based On: Parity

Method: Both agree on Parity (even - normally)

10110110 1

Redundancy: 1

Errors: Odd no. of errors detected

Agree upon: even #1s / odd #0s

Vertical Redundancy Check

2. VRC

2D LRC

110	10	1
0110	1	1
110	11	0
0	11	00

m+n+1

Cyclic Redundancy Check

3. CRC — used in all layers level

1101) 100100 000 (

↳ append 3 0's (∵ degree of poly)

1111011

1101) 1001000000

1101

1000

1101

1010

1101

0110

1101

011

Polynomial

x^3+x^2+1 represented as

1101

$x^3\ x^2\ x^1\ x^0$

Remainder = 001 — CRC

Sender has to send original data appended with CRC

100100 001 — message sent
↳ CRC

The receiver checks for errors by dividing the message received by 1101 (already agreed) and remainder should be 000.

redundancy bits = degree of polynomial

errors : Detects all errors.

Based on : Modulo 2 division.

4. Checksum:

→ Higher layer protocols use it

→ It is based on 1s complement.

Data / message: 1011|0101|1101|1001

Message is divided into blocks of 4 and add now.

```
  1011 — 1
  0101 — 2
  ----
  10000
1 <- overflow
  1101 — 3
  ----
  1101
  1001 — 4
  ----
  0110  → do 1's complement [1001] — checksum
```

Sender has to send message and checksum (appended)

Receiver should do this process again (all 0's obtained ☹). ↳ with 1's compliment

Based on: 1's compliment

Method — summing

Redundancy bits: # blocks size

Errors: All types

Agreed upon: block size (normally, in internet block size = 16 bits)