IPC
inter process communication

- process within sys (independent / coop)
- coop proc can be affected or affect other data
- Reason for coop proc
    - info sharing
    - computation speed
    - modularity
    - convenience

Co-op proc need IPC's
2 models :
    - shared mem : access common same shared memory intelligently between proc
    - message passing : proc don't have anything in common . **Send request to kernel to send or recive msg**

in shared mem : kernel doesn't do much work . Only makes available shared mem.
in msg pass : kernel also handles data

Each requires OS to make sys calls for
1. Create IPC mechanism
2. rd/wr using IPC mech
3. delete ipc mech.

Proc communicate with each other with help of OS

Producer Consumer problem

Paradigm for coop proc:

1. Producer proc produce info that is consumed by consumer

* unbounde buffer : no limit on buffer size

* bounded buffer : fixed

Example of producer consumer : *grep*

Pipes

types :

1. unnamed / ordinary pipes

2. named

1. Ordinaty pipes // Unnamed:

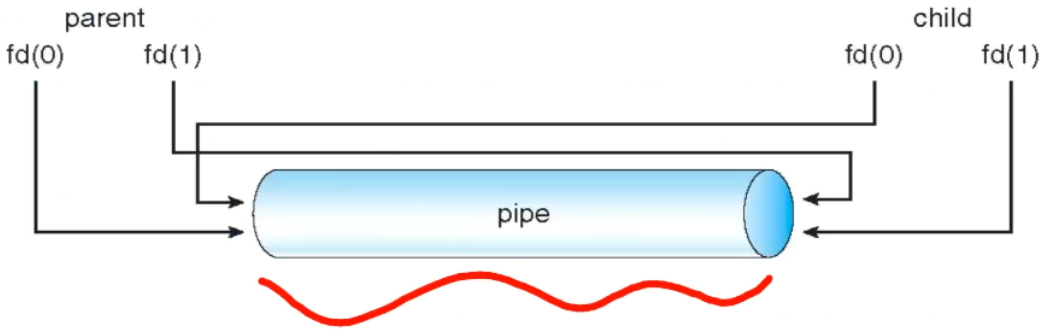
allow communcation in std producer-consumer style

prod write at one end (write-end of pipe)

consumer read from other end (read-end of pipe)

* ordinary pipes are unidirectionl

Limitation : **requires parent-child / sibling kind of relation ship b/w communi* process**



it gets pair of fd for each communication

***** pipe inner work**

when pip(pfd) is called . // int pfd[2]

OS allocates a internel buffer to the kernel. Create a notion of wr end and read end of buufer and made available as fd (file descriptor).

Pfd[0] : read fd

pfd[1] : write fd

in PCB:

1. kernel create memory region (buffer)
2. create struct file (read, write)
3. return fd as indices after pipes

after proc calls fork

4. pcb gets duplicated

5. its fd will also point to fd

6. buffer shared by two proc

As pipes are unidirectional, we have to close unused end end of the pipe using close(pfd[0])

if we close stdin file by close(0)

then read a file then fd for file = 0

then scanf command will read on file instead of keyboard.

called as redirection of std input

Similary

if we close stdout file by close(1)
then read a file then fd for file = 1

then printf command will write on file instead of output.

Also works in cat < /etc/passwd

called as redirection of std output

DUP : duplicates a file descriptor

two fd pointing to same file. Copy of pointer.

In dup2(old,new) : make new fd point to old fd

****

int pfd[2]

pipe(pfd) // **OS creates buffer for pipe data. Return 2 fd**

pfd[0] : reading pfd[1] : writing

pid = fork()

if pid ==0 : //**child**

close(0) //**std input closed**

dup(pdf[0]) // duplicate pdf[0] // in fd[0] as lowest available

close(pfd[1]) //as we don't want to write to file. //close unwanted pfd

execl("/usr/bin/head","/usr/bin/head","-2",NULL) //execute head -2

if pid ==1 : //**parent**

close(1) //**std output closed**

dup(pdf[1]) // duplicate pdf[12] // in fd[1] as lowest available

close(pfd[0]) //as we don't want to read from file. //close unwanted pfd

execl("/usr/bin/cat","/usr/bin/cat","/etc/passwd",NULL) //execute cat /etc/passwd

here parent runs cat and child runs head. This is how proc talk to each other.

Here parent was over written by "cat"

****

##########

when we want to use pipe in shell, while preserving parent, we

1. call fork

2. in first child

close(0) //**std input closed**

dup(pdf[0]) // duplicate pdf[0] // in fd[0] as lowest available

close(pfd[1]) //as we don't want to write to file. //close unwanted pfd

execl("/usr/bin/head","/usr/bin/head","-2",NULL) //execute head -2

3. in parent

again call fork

in child (2nd)

close(1) //**std output closed**

dup(pdf[1]) // duplicate pdf[12] // in fd[1] as lowest available

close(pfd[0]) //as we don't want to read from file. //close unwanted pfd

execl("/usr/bin/cat","/usr/bin/cat","/etc/passwd",NULL) //execute cat /etc/passwd

4. in parent don't do anything

close(pfd[0])

close(pfd[1])

##########

pipe is not redirection to a file by single process. Two process coexist at same time (continous writing nad continous writing)

for 3 proc, 3 are required

2. Named :

- called as FIFO (unnamed are also FIFO)
- pipeline can create “file” that act as pipe. Multiple proc share file to read/write as a FIFO
- more powerful than ordinary pipes
- communication is bidirectional
- No parent-child relation is needed.
- several proc can use pipe for comm.
- provided on unix and windows
- **not automatically deleted by OS. Can delete by rm.**

mkfifo : command in linux for named pipeline

mkfifo file : create named file “file” : **size 0 p start begining during ls *Special type of file**

mkfifo(file,permision)

open file for writing then write on it

similar another proc can use same file

**** example : chat code**

Code 1:

```
mkfifo(file,0666)
while loop {
open named file in readonly
print()
close(fd)
open named in writeonly
get input
write to file
}
```

Code 2:

```
mkfifo(file,0666)
while loop {
```

open named in writeonly

get input

write to file

open named file in readonly

print()

close(fd)

}