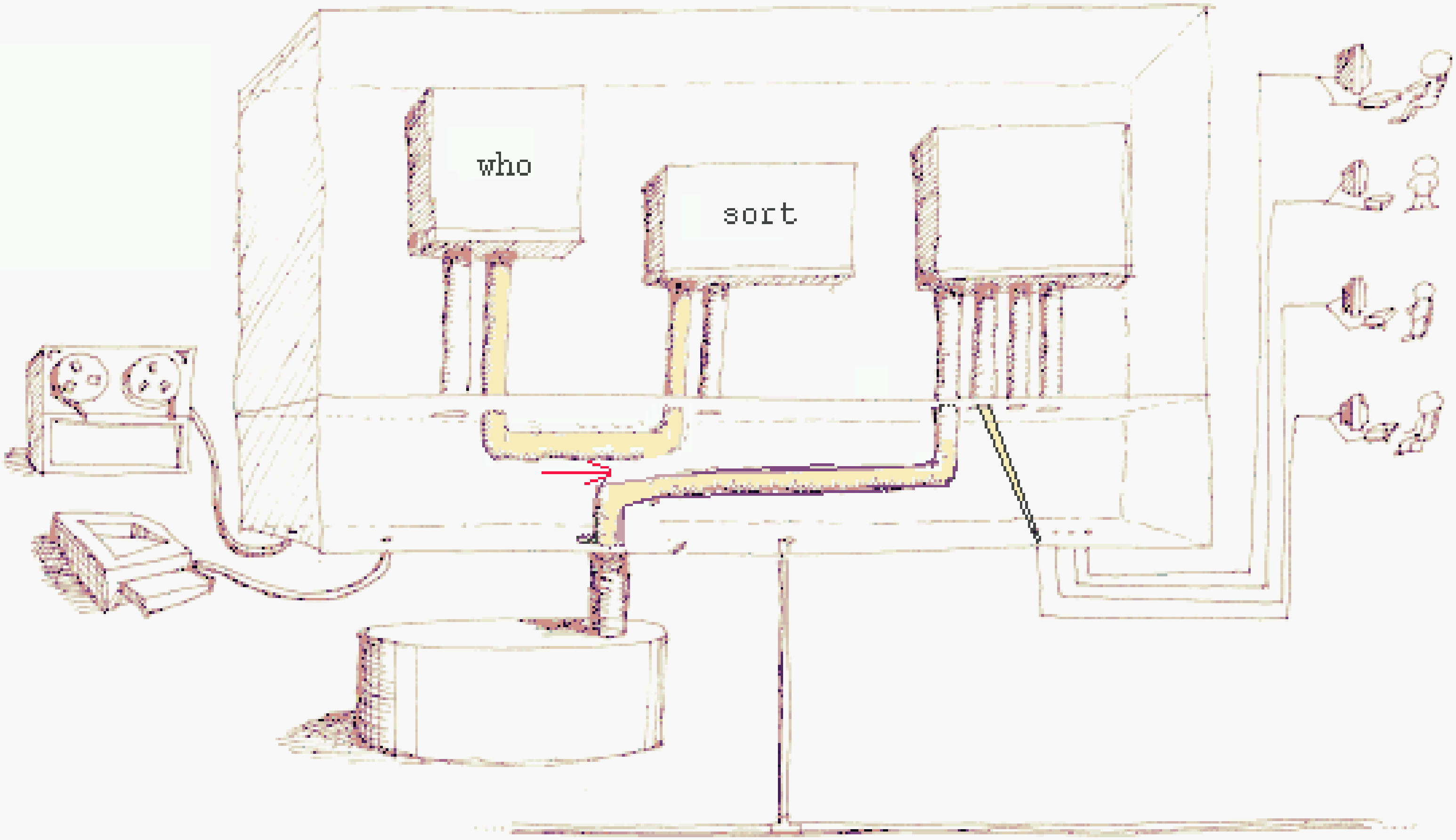
Lecture 10

More IPC: I/O Redirection and Pipes



Class 10: I/O Redirection and Pipes

- This week, we continue with two themes

  a. Features of the shell
      * running programs, programming
  b. Interprocess communication (ipc)
      * exec()-argv[], exit()~wait()
      * the environment

- We shall focus on:

  I/O redirection - a feature of the shell
  Pipes - a feature of the shell, AND
      another example of IPC

- Our method will (still and again) be

  (a) What does it do? (b) How does it do it?
  (c) Let's do it ourselves!

A shell Application: watch for users

The problem:

You have a list of buddies. You want a program to notify you when any of your buddies login or logout.

Solution:

You could write a C program to read utmp, but a shell script can use who and other tools:

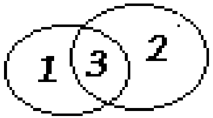
logic

```
get list of users (prev)
sleep
get list of users (curr)
compare lists
  in prev, not curr -> out
  in curr, not prev -> in
mv curr prev
```

shell code

```
who | sort  > prev
sleep 60
who | sort  >  curr

comm -23  prev  curr
comm -13  prev  curr
mv curr prev
```



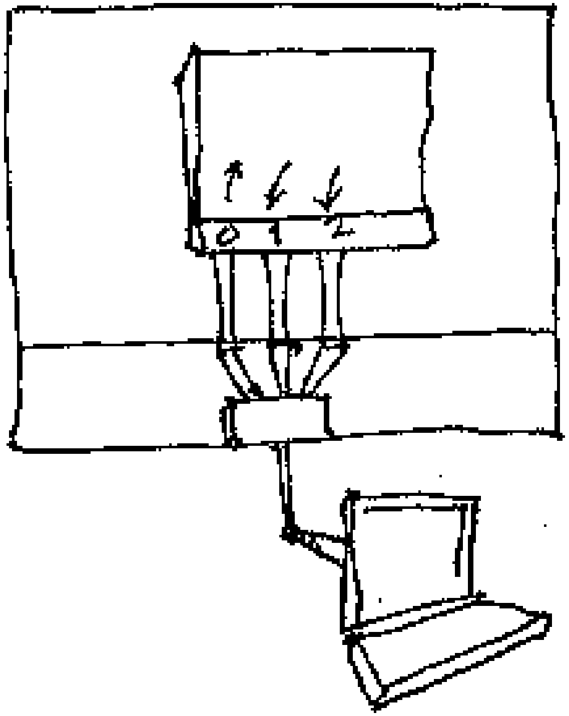
The shell version of watch demonstrates

a. power of shell scripts

b. flexibility of software tools

c. use and value of i/o redirection and pipes

II. Focus on Redirection and Pipes: Basic Facts



1. Every unix program gets three open file descriptors at startup:
0:stdin, 1:stdout, 2:stderr

2. These are often attached to the tty.

3. Most Unix tools send output to stdout and provide NO WAY to send output to a file.

4. If you want to send output to a files, use
cmd > filename and the shell redirects

In Fact: the tool is not aware of the
>filename
notation. example: listargs.c

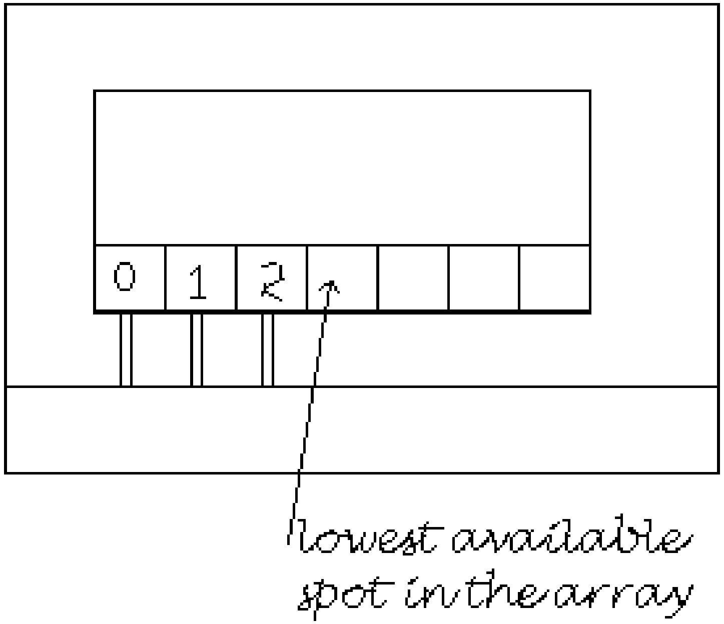
Goal: Understand how i/o redirection works AND learn how to write programs that use it.

Method: write programs that do

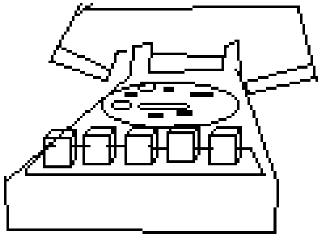
sort < data	attach stdin to a file
who > userlist	attach stdout to a file
who | sort	attach stdout to stdin

Essential Fact for Redir and Pipes

Every process has an array of open files. A file descriptor is an index into that array.

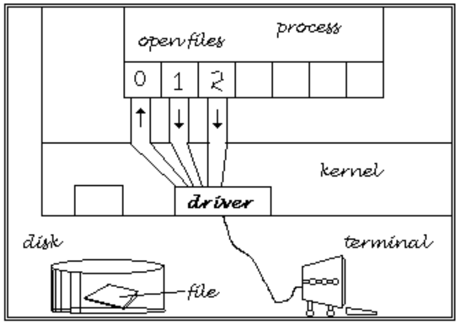


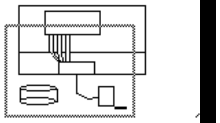
FACT: when you open a file, you ALWAYS GET the lowest available spot in the array



III. How to Attach stdin to a file: 3 methods

method 1: close ... open

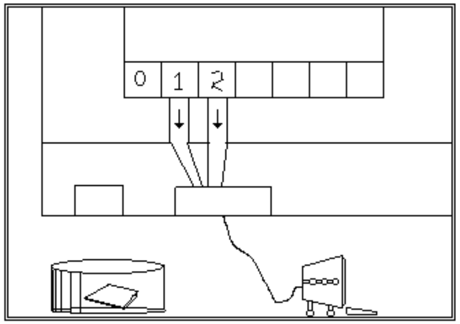




1. Standard Plumbing

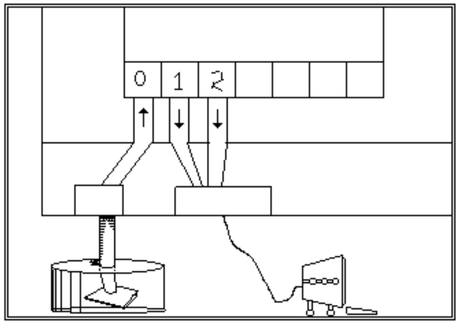
File descriptors 0,1,2
attached to /dev/tty

0 for reading
1 for writing
2 for writing



2. close(0)

If the process closes file descriptor 0, that entry in its array of i/o channels is free.



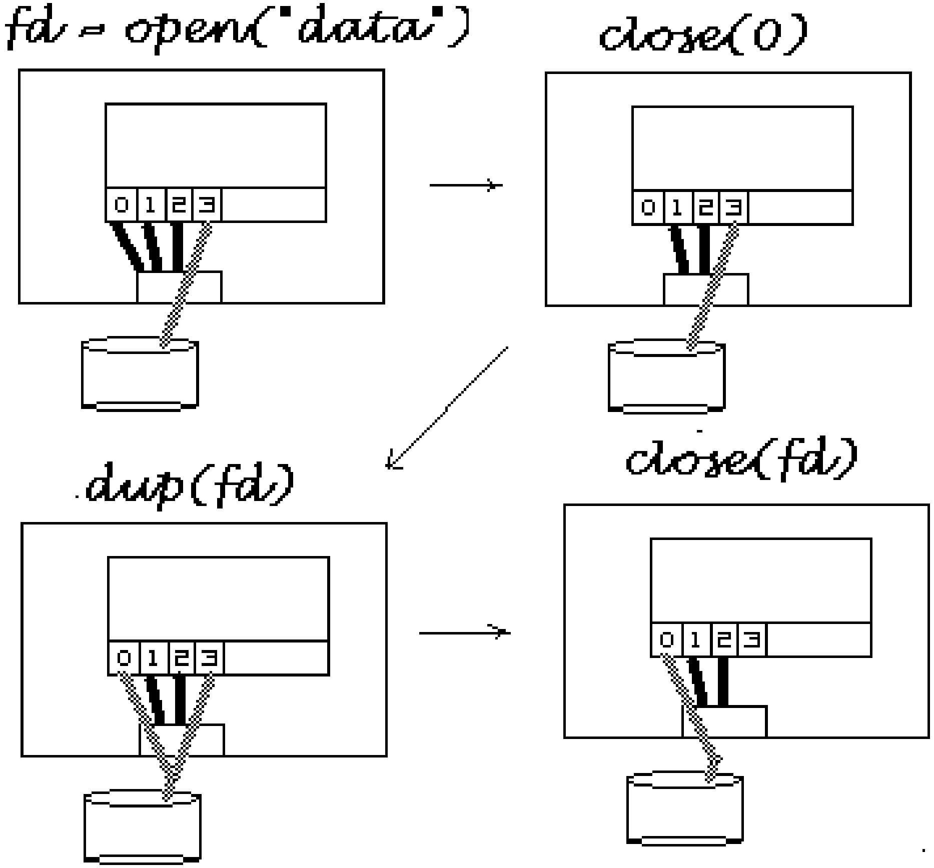
3. fd = open("file",0)

If the process opens another file, that connection is attached to the FIRST FREE entry in the array of i/o channels.

stdinreader1.c

Method 2: open .. close .. dup .. close

dup() creates a second (duplicate) connection to the same file.



fd = open("data")

close(0)

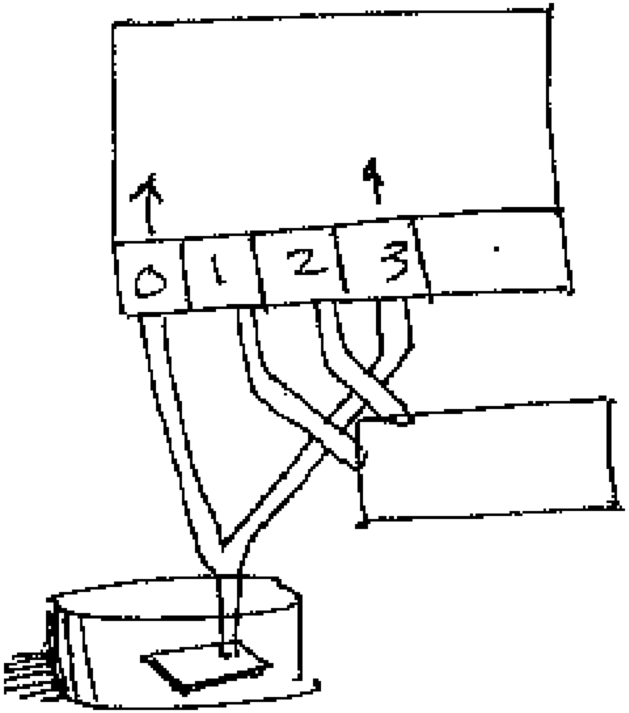
dup(fd)

close(fd)

stdinreader2.c

Method 3: uses dup2(origfd, destfd)

open .. dup2 .. close



```
fd = open( "data" )
dup2( fd, 0 )
close( fd )
```

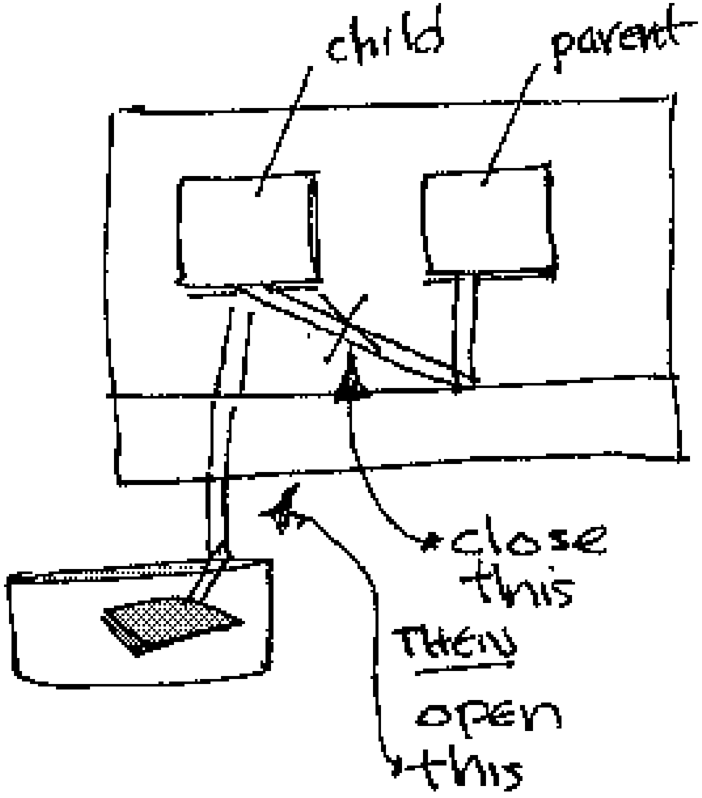
dup2(fd, 0)
  closes 0
AND
  dups fd to 0

stdinredir2.c

IV. Redirecting I/O for Another Program

A more typical example is a shell command like:

```
who > userlist
```



Logic:

fork

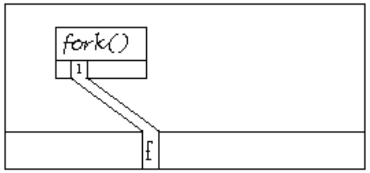
child

```
close(1)
creat("userlist")
 exec("who")
```

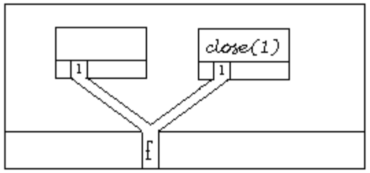
Then the who program runs sending its output to stdout, i.e. fd 1.

whotofile.c

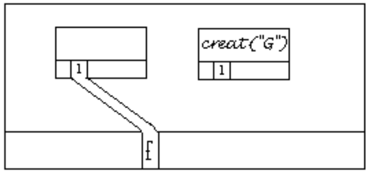
redirect stdout of a child, then run a program



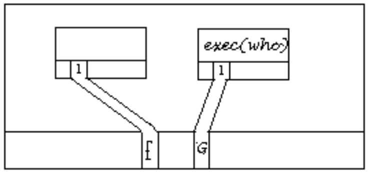
1) A process executes the fork() system call. (note: other open files are not shown to increase clarity.)



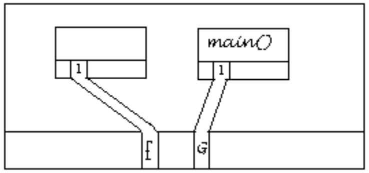
2) The child process inherits the open files of its parent. Here, both send stdout to the same file. The child then closes fd 1 .



3) The parent now has the sole connection to file f.
The child opens a different file, G.



4) In the child, fd 1 is now attached to the file G.
Any write()s to 1 go to that file. The child now exec()s a program



5) That program is loaded into the child. Its fd 1 is still attached to file G.

thus is explained: who > G

Questions to Wrap Up I/O Redirection

1) How to implement >>

example:

```
who >> userlog
```

answer:

for class discussion

2) How to redirect standard input for a program

example:

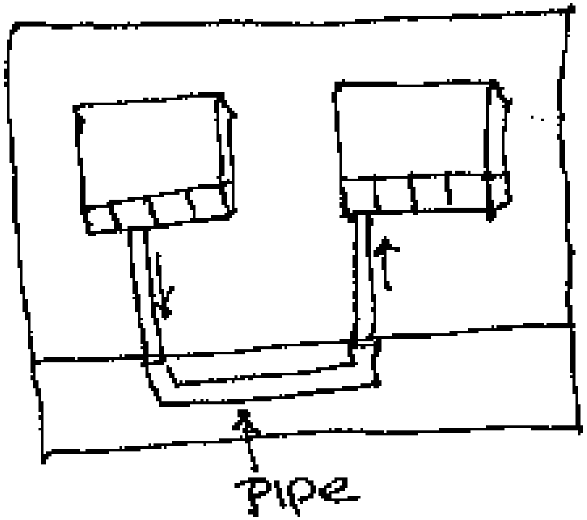
```
sort < data
```

answer:

for class discussion

Programming Pipes: coding who | sort

Why? Pipes allow one to combine software tools into practical, special-purpose programs.



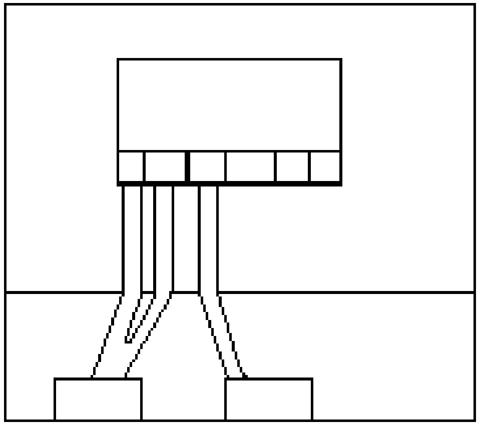
What?

A pipe is a one-way data channel in the kernel with a reading end and writing end.

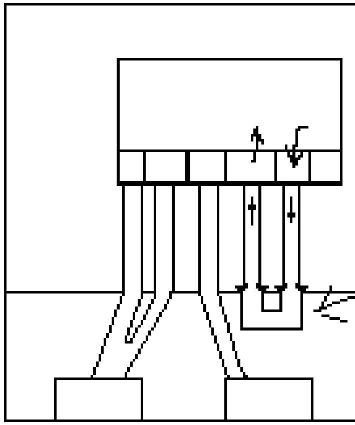
How? The system call pipe(int a[2]) creates a pipe and connects it to two file descriptors. a[0] is the file descriptor of the reading end, and a[1] is the file descriptor of the writing end.

```
in this example:
a[0] = 3, a[1] = 4

write(a[1],"hi!",3);
```



before



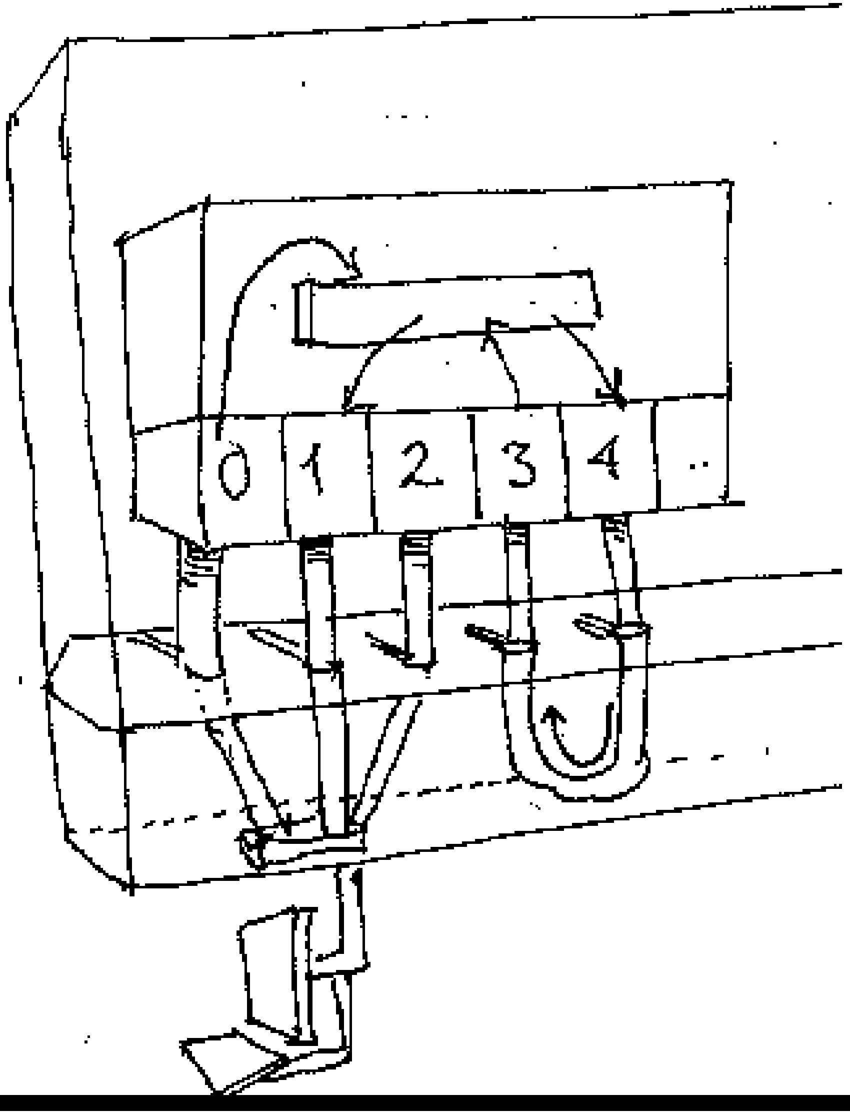
after

pipedemo.c

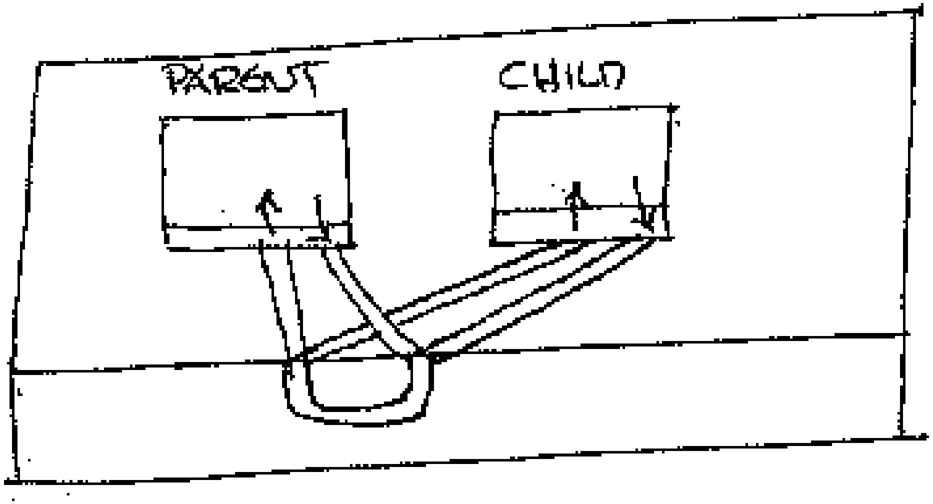
This program creates a pipe then uses that pipe to send data to itself.

It is an odd example, but it shows how a pipe works.

Typically, a pipe is used to send data from one process to another



pipedemo2.c using fork() to share a pipe



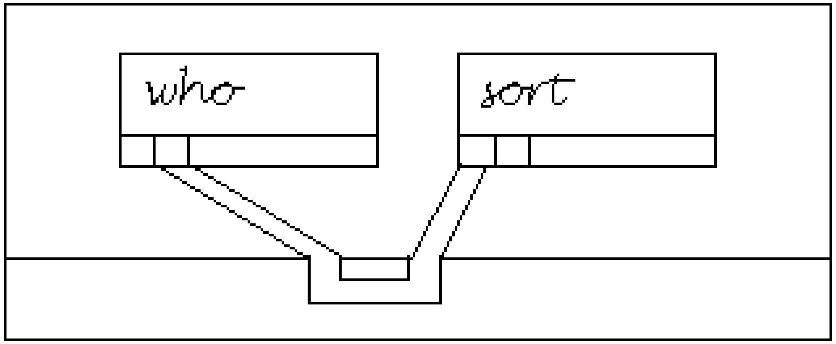
if child writes
into the pipe,
the parent
can read
those bytes.

Notes: (1) Multiple writers are ok
(2) Multiple readers cause trouble

We Are Almost at who | sort:
(1) need to redirect 0 and 1
(2) need to exec those programs

Coding who | sort

Goal:



Logic:

class exercise

Technical Details about Pipes

1) read() on a pipe blocks until data appear

2) write() on a pipe blocks until space is available in the pipe

3) When all writers close the writing end, read returns 0 (i.e. eof)

4) When all readers have closes the reading end, then write() causes SIGPIPE