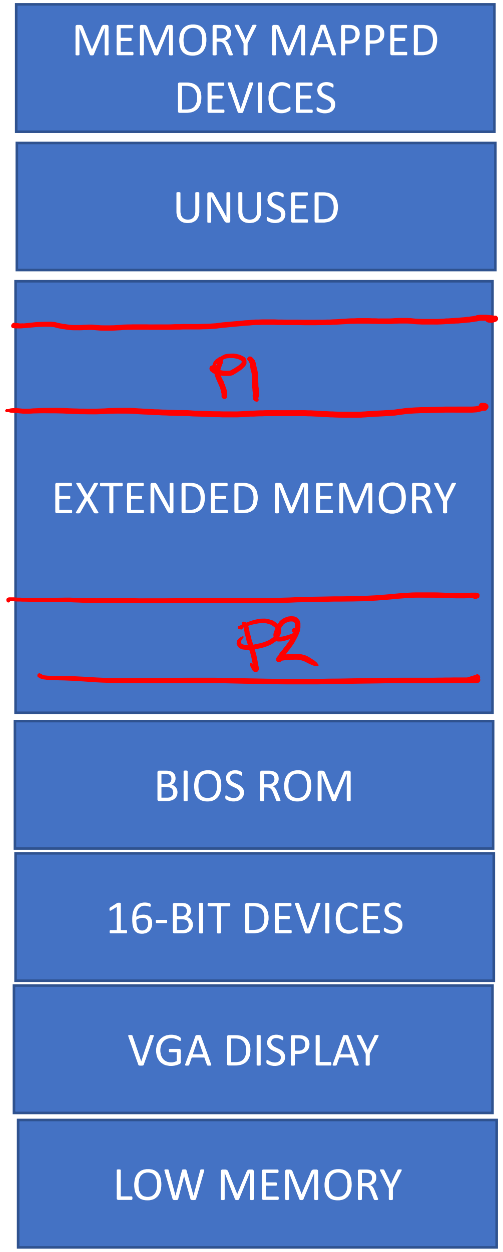
Processes

- Processes are containers of threads

- Each process start with one thread and can create more threads if needed

- Every process has its own quota in RAM

- A process cannot see other processes data



MEMORY MAPPED DEVICES

UNUSED

P1

EXTENDED MEMORY

P2

BIOS ROM

16-BIT DEVICES

VGA DISPLAY

LOW MEMORY

Application

- OS is a shared library

- The library exports some interfaces that application developers can call to use OS services

- The OS enforces that an application can only call these exported routines

Application

- Application is partitioned into two components
  - Application code (user programs)
  - OS library

- User programs are untrusted

- OS library is trusted

How user programs use the OS library?

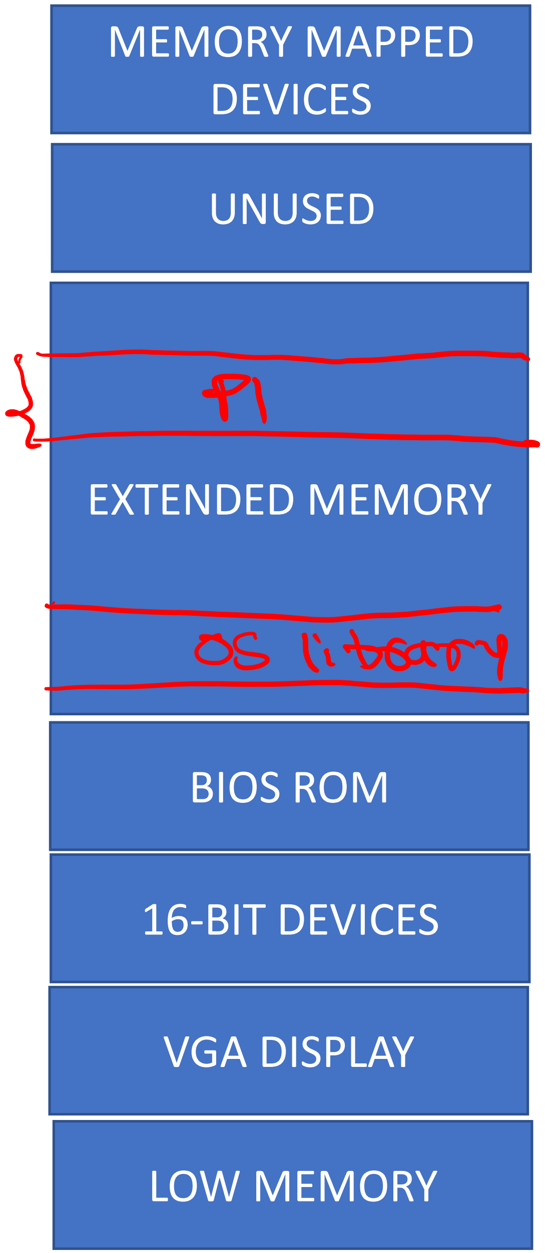
```
create_thread () {
    struct thread *t = malloc (sizeof(struct thread));
    t->esp = malloc (4096) + 4096;
    status = interrupt_disable ();
    add_to_ready_list (t);
    set_interrupt_status (status);
}
```

Memory map

- User program and OS (kernel) lives in different address spaces

- Users programs cannot directly access kernel memory

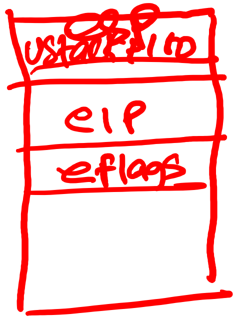
- However, the kernel can access the entire memory



Interrupt handling

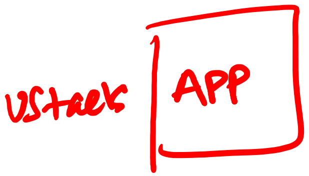
esp = Ustack + 100

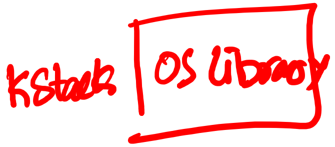
esp = Kstack

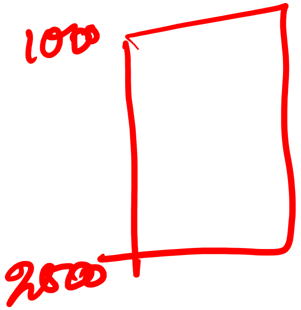


iret

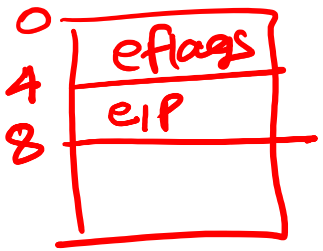
mov $8, %esp







mov $8, %esp



Why kernel stack is needed

Software interrupts

128

- int $100
- Linux uses vector 128 for system call
- The syscall id is passed in some registers or user stack

```
  idt[128] = syscall_handler
mov $0, %eax
int $128

  syscall_handler(int id)
  {
    switch (id) {
     case 0:
          create_thread();
          break;
     case 1: create_process();
          break;
    }
  }
```

Unix operating system

- “shell” is the first user program created by the Unix

- “shell” can create more processes

Fork

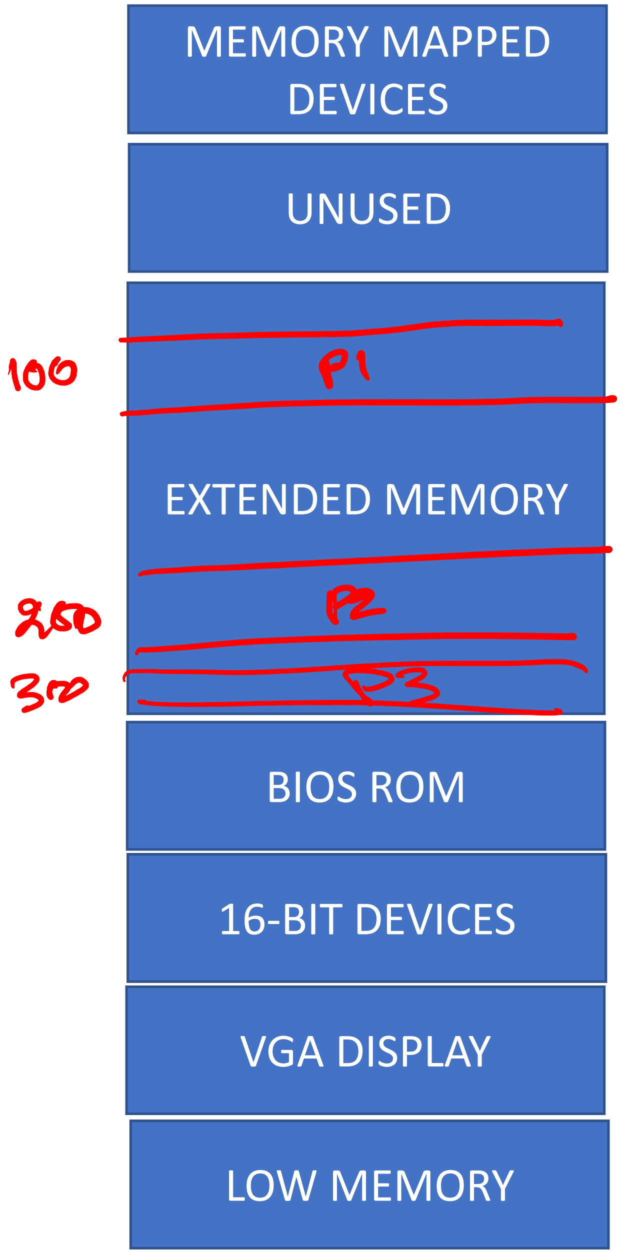
```
main () {
    int pid;
    pid = fork();
    if (pid == 0) {
       /* child process */
    } else if (pid > 0) {
      /* parent process */
   }
}
```

pid= fork()

300
0

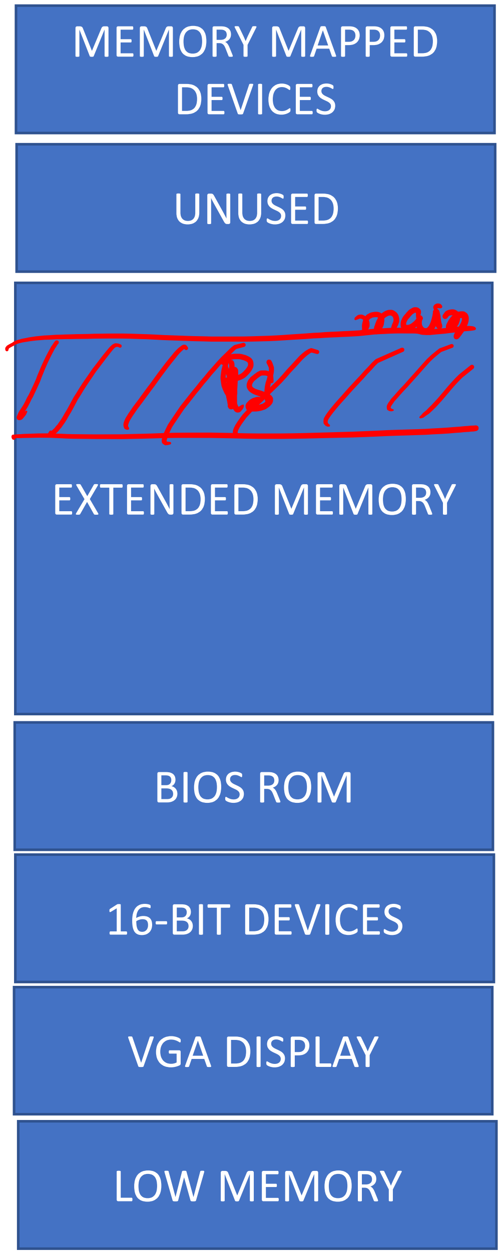
200 ↑ Parent

child
0



Exec

```
main (int argc, char *argv[]) {
    exec ("ls", argv, 0);
}
```



System calls

- int creat (pathname, mode)
- write (fd, buf, len)
- read (fd, buf, len)
- close (fd)

```
fd = creat ("tmp.txt", 0666)

write( fd, "Hello world\n", sizeof("Hello...."));

char buf[64];
len = read (fd, buf, 64);

close (fd);
```

System calls

- fds 0, 1, 2 have special meaning

- 0 points to standard input
  - e.g., keyboard

- 1 points to standard output
  - e.g., terminal

- 2 points to standard error
  - e.g., terminal

Shell

```
while (1) {
    write (1, “$ “, 2);
    readcommand (0, command, args);
    if ((pid = fork ()) == 0) {
        exec (command, args, 0);
    } else if (pid > 0) {
        wait (0);
    } else
       printf (“Failed to fork\n”);
}
```

exit (int status)

- The process terminates with a given status
- Wipe out all the memory occupied by the process
- Release all resources

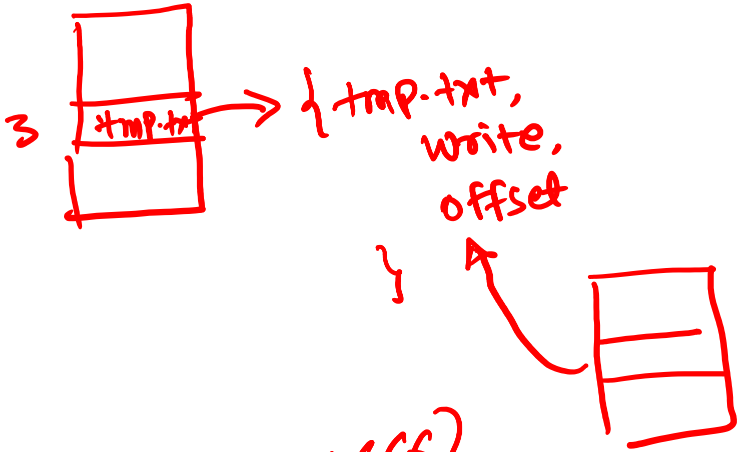
int wait ()

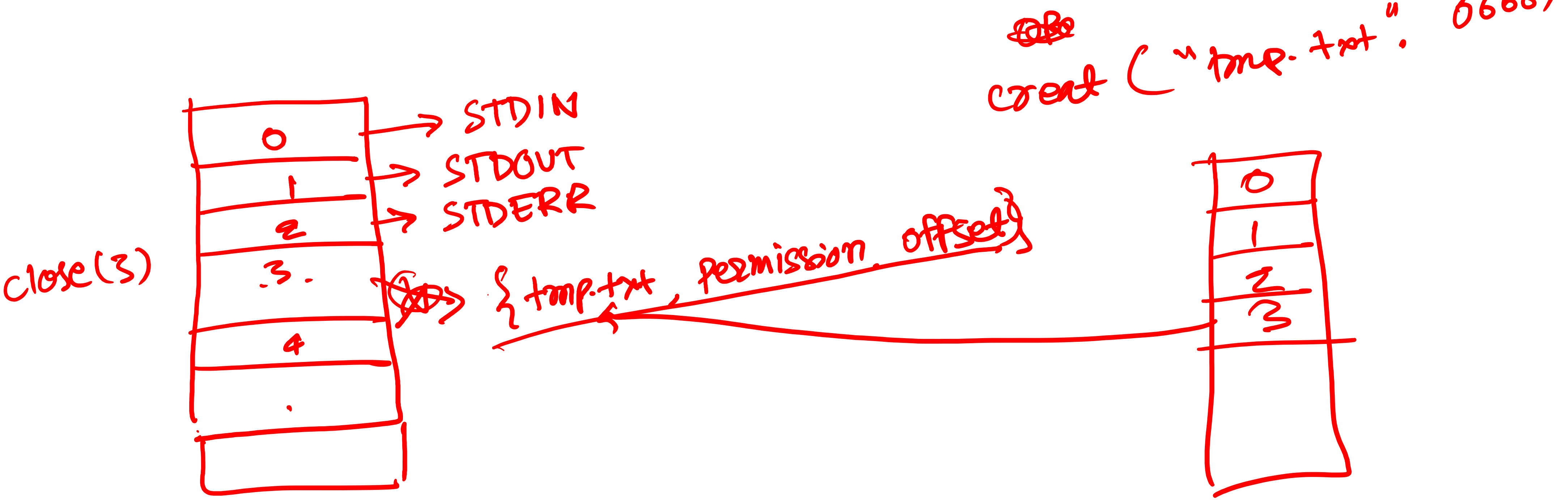
- wait system call waits until the child terminates

- The return value is the exit status of the child

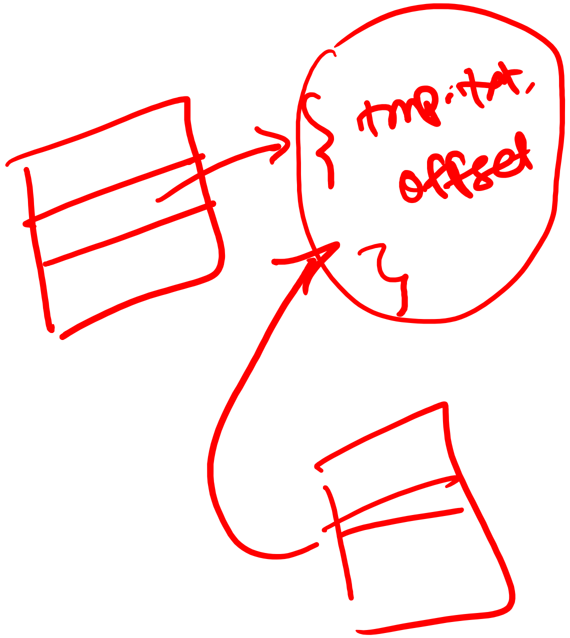
File descriptor table

- File descriptors are inherited by the child process

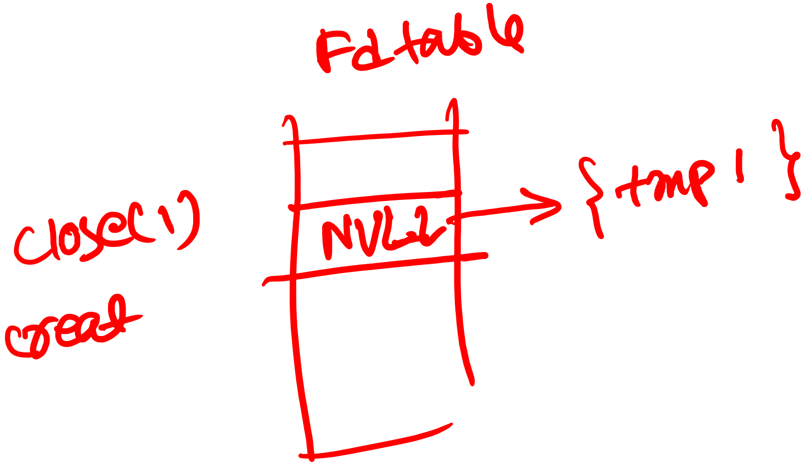


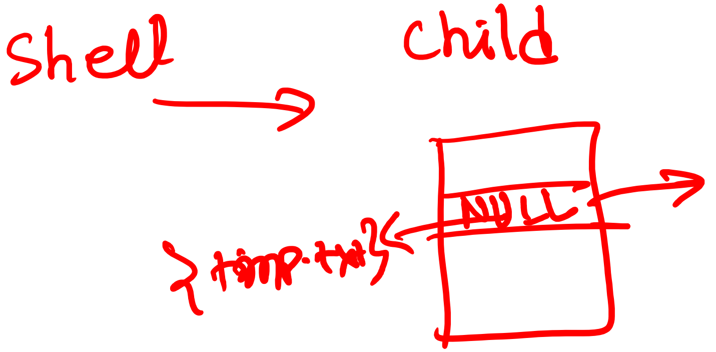


How does shell implement “ls > tmp1”



```
pid = fork()
if (pid == 0 )
{
    close (1);
1 = creat ( tmp1, .. );
    exec ( "ls" command);
}

write(1, "  ")
```





STDOUT
close(1);
open (tmp.txt)
exec(ls);

sh < script > tmp1

```
          close (0);
<         open (script);
          close (1);
>         creat ( "tmp1);
```

sh

Read_command (0,

1 → tmp1

ls f1 f2 nonexistent-f3 > tmp1 2>&1

STDERR
↓
2

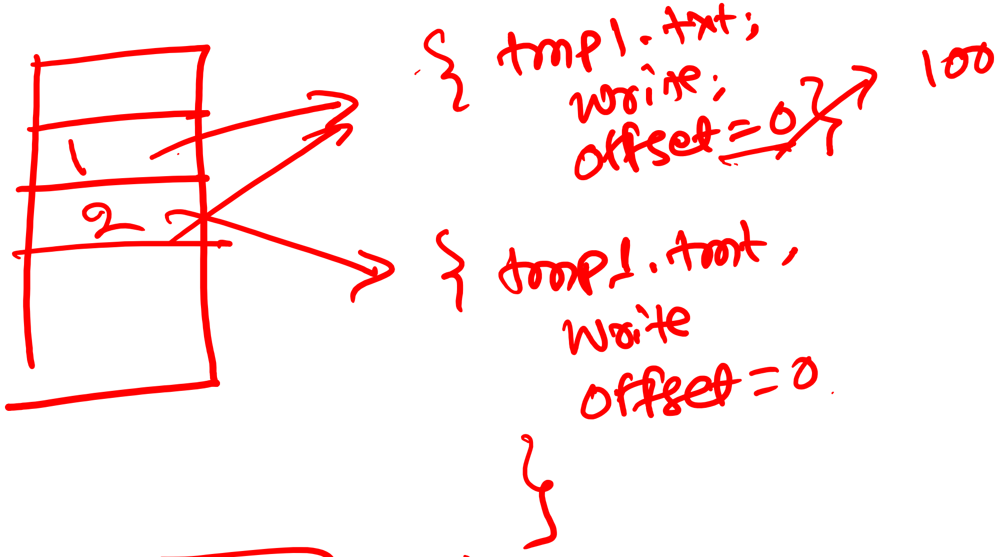
fd1 = open(tmp.txt)
fd2 = open (tmp.txt)

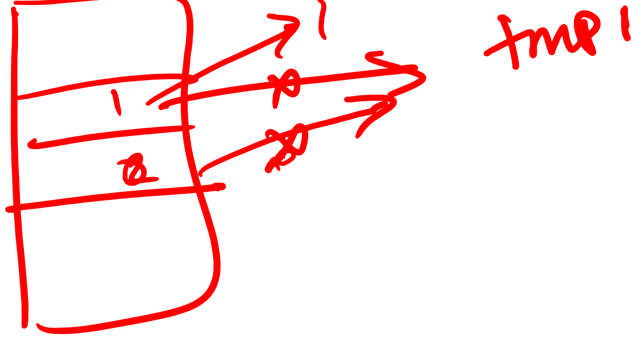
```
close(1);
creat(tmp1);
```

```
~~close(2)~~
creat(tmp1);
```

```
dup(1);
```

```
close(1);
creat(tmp1)
close(2);
dup(1);
```

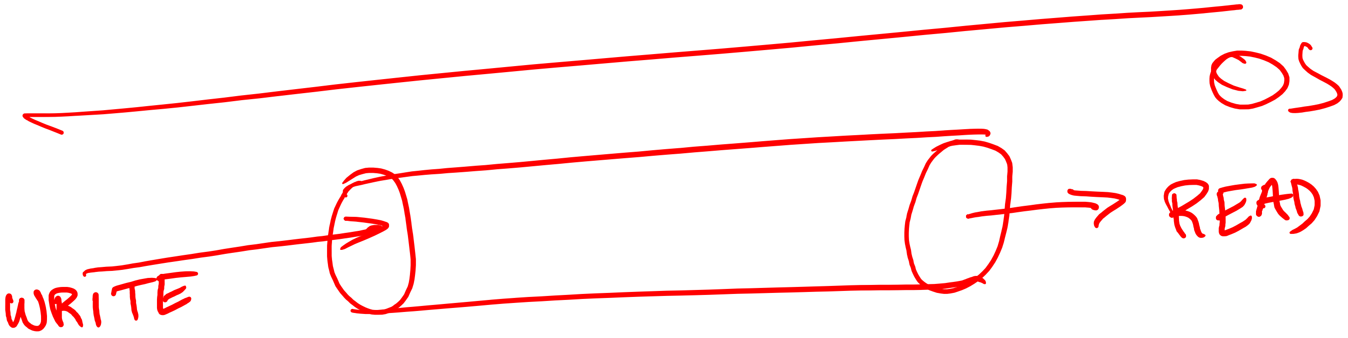




dup system call

Inter process communication

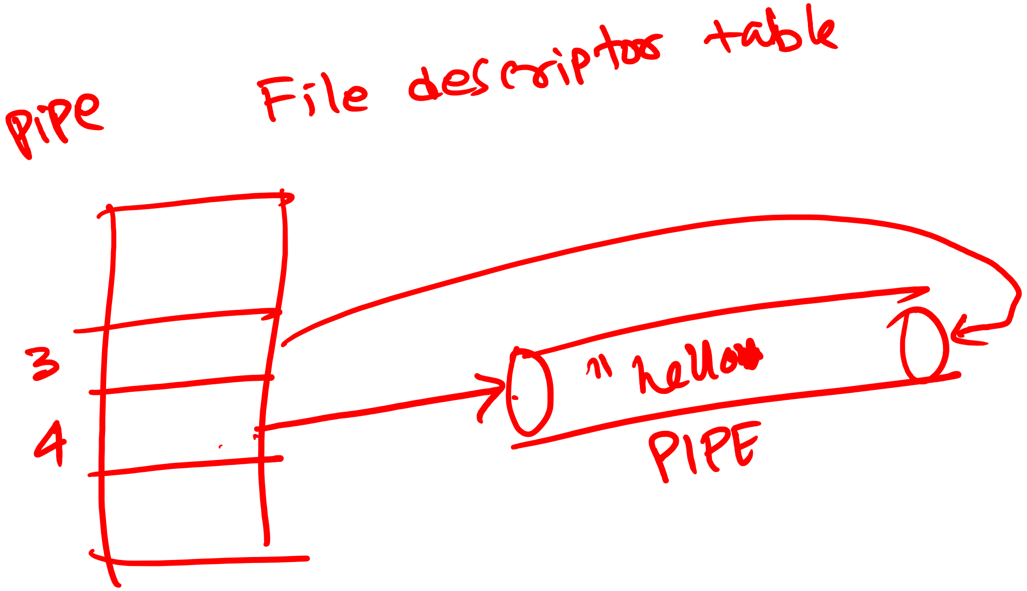
- Pipe
  - A pipe has two end
  - Data are written to the input end
  - Data are fetched from the output end



Pipes

```
int fd[2];
char buf[512];
int n;

pipe (fd);
write (fd[1], “hello”, 5);
n = read (fd[0], buf, sizeof(buf));
// buf[] now contains ‘h’, ‘e’, ‘l’, ‘l’, ‘o’
```

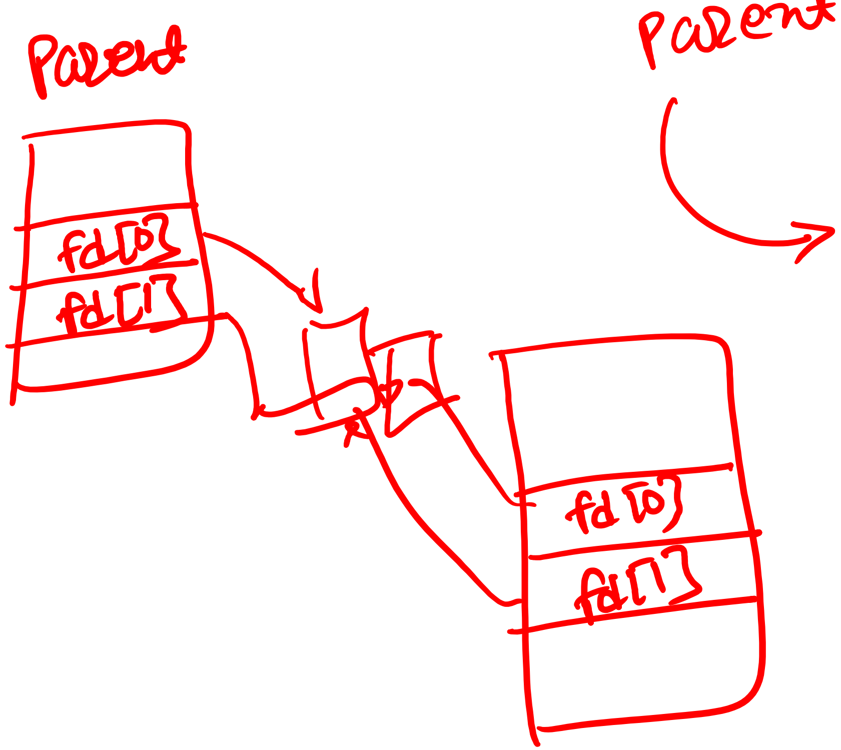
fd[0]=3
fd[1]=4

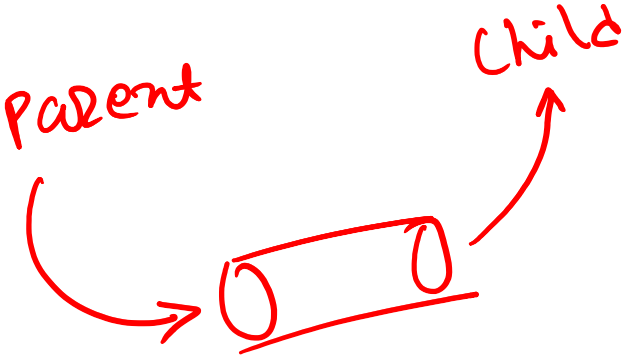


fd[0] -> Read
fd[1] -> write

Inter process communication

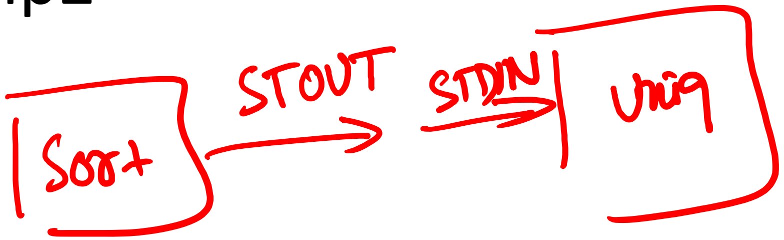
```
int fd[2];
char buf[512];
int n, pid;

pipe (fd);
pid = fork ();
if (pid > 0) {
    write (fd[1], “hello”, 5);
} else {
    n = read(fd[0], buf, sizeof (buf));
}
```





How to run a series of programs?

```
sort < file.txt > tmp1
uniq <tmp1 > tmp2
wc < tmp2
rm tmp1 tmp2
```

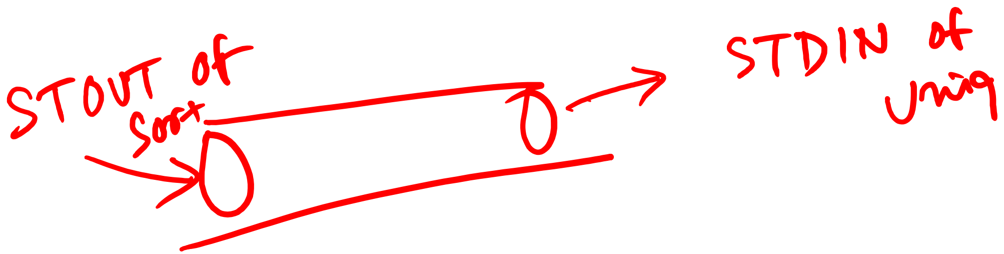


Sort < file.txt | uniq | wc

How to run a series of programs?

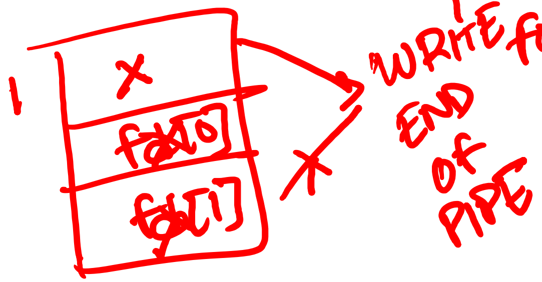
```
sort < file.txt | uniq | wc
```

Pipes

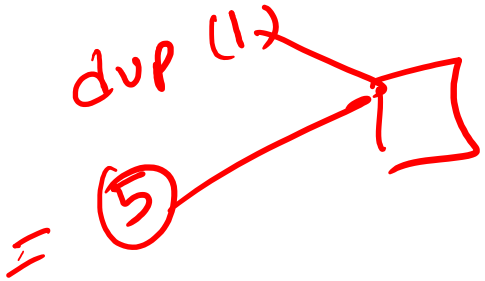


```
pipe (fd) ;  ✓
if ((pid == fork()) == 0) {
   close (1);
   tmp = dup (fd[1]);
   close (fd[0]);
   close (fd[1]);
                     sort
   exec (command1, args1, 0);
}  else if (pid > 0)  {
   close (0);
   tmp = dup (fd[0]);
   close (fd[0]);
   close (fd[1]);
                     uniq
   exec (command2, args2, 0); }
```

Shell
{



fork
{



```
fork ();
if (Pid == 0) {
    exec ("sort");
}
else {
    exec ("uniq");
}
}
```