```
char* cmd_args[] = {cmd, param, NULL};
```

x | y

```
pipes[2*# of pipes];
pipes[#%2==0]; → read
pipes[#%2==1]; → write
```

always close pipes!

```
dup2(pipes[a], a); open side of pipe
wait(&status); for # fork()
```

x < y > z

```
inFile = open(y, O_RDONLY);
outFile = open(z, O_WRONLY);
dup2(inFile, 0);
dup2(outFile, 1);
close(xFile);
```

```
for(int i=0; i<X; i++){
    fork();
}
```

2^{n+1} processes

$2^{n+1} - 1$ children

```
exec_(prog, cmd, NULL);
```

replaces address space with program

Control is never returned if error-free

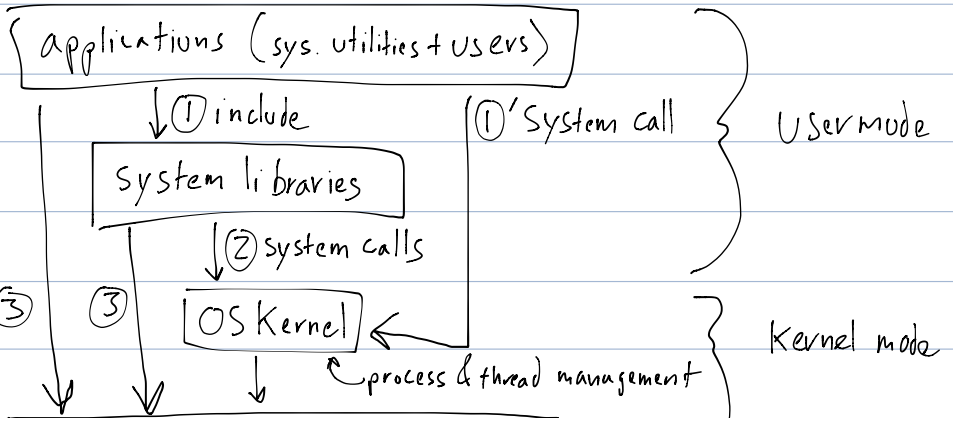
pid = fork(); child_pid = C parent_pid = P

```
child_process {
    pid = 0
    get_pid = C
}
parent_process {
    pid = C
    get_pid = P
}
```

fork() creates child process, which has a copy of parent's data

What is a computer system?

1) Hardware
2) Software
3) Data
4) Users



Hardware

① + ①′ : UNIX/LINUX

① + ② : Windows

③ : Assembly programming

privileged instructions = accessible only by Kernel

mode bit = check for if kernel/user (0/1)

abstraction = put one layer on top of another

virtualization = Use one physical entity to be shared by multiple processes/threads

Concurrency = have multiple processes try to access memory

resource management

memory management

file management

mass-storage management

System calls used to access system resources

Types of OS

1) batch systems = program treated as batch (one thing)

set up time is bottleneck

turnaround time = time when submit until everything comes back (PROBLEM)

2) time sharing systems = CPU can be switched rapidly between processes

multiprogramming = put multiple programs in memory (SOLUTION)

response time

3) real time systems = microprocessor

soft - miss 1-2 frames, it's fine

hard - must react instantly, can't miss frame

Computing environments

1) traditional (multi-processor systems)

2) networked (mobile, distributed DTOS NOS, client-server, peer-to-peer, cloud computing)
HTTP
distributed network

ftp://hostname pathname
DTOS NOS

Infrastructure/Software/Platform
as a Service
OS doesn't matter
more control of OS
OS does matter

2 ways to access system resources

(1) command-driven (interpreters-based)

MS-DOS

(2) menu-driven (GUI-based) (graphical user interface)

mouse

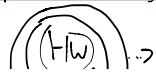
touchscreen

OS design methodologies

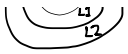
(1) monolithic approach - no structure, big mess, unidentified lot of files, good for small systems, hard to adapt
very efficient

MS-DOS, early UNIX

(2) layered approach - good to control complexity, condition: can't change functions between layers



not very efficient to access resources



(3) microkernel approach Windows NT

minimum set of functions in kernel ↑ doesn't have to be changed

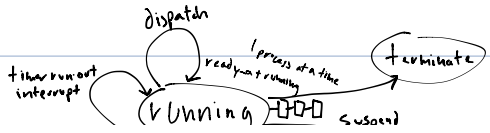
- Setting device registers ← need to have devices do something, addresses
- switch CPU between processes
- manipulate MMU virtual → physical
- capture HW interrupts — mach
- pass system calls ↖ shared memory / message passing — TRU64

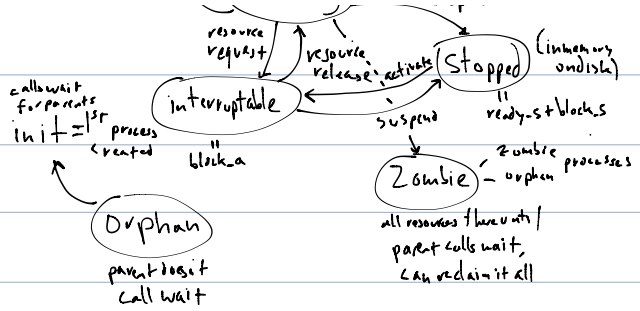
(4) Object-oriented approach classes, inheritance, metaclasses, info-hiding, encapsulation

dynamic linking & dynamic loading

process-control-block PCB

Linux approach





context:

(1) uniqueID of the process (pid)

(2) pointer to parent process

(3) pointer to child processes

(4) priority of the process (for CPU scheduling)

(5) a register save area (area used to save register contents) shared by all processes

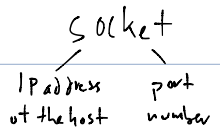
(6) the processor it's running

(7) list of open files

(8) current position of stack pointer

(9) current position of heap pointer

(10) housekeeping information



install a phone jack = create a socket

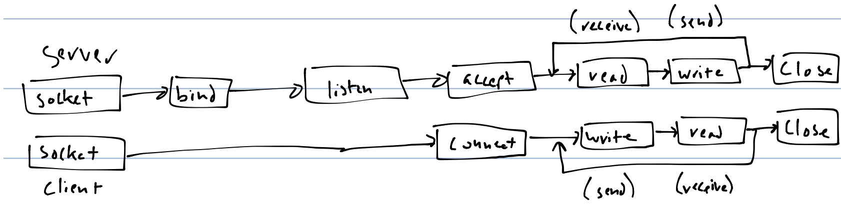
get a phone number = bind an address to the socket

listen to the phone see if it rings = listen to the address for an incoming request (LISTEN)

pick up the phone = accept the incoming request (ACCEPT)

talk on the phone = communicate with the requester (SEND/RECEIVE)

hang up the phone = disconnect the communication (CLOSE)



thread = independent unit running concurrently in a process

(1) PC
(2) stack
(3) registers

} allocated to each

$C(I) = A(I) + B(I), \; I = 1..100$

C1: 1..20

C2: 21..40

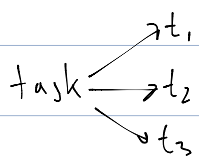
C3: 41..60

C4: 61..80

C5: 81..100

SIMD Model
single instruction string
multiple data string

(1) specialist paradigm



(2) client-server paradigm

client 1 → server 1

client 2 → server 3

client 3 → server 2

word processor

user input thread

spelling or grammar checking thread

document layout thread

etc.

(3) assembly line paradigm

a task → thread1 → thread2 → thread3

compiler (2 passes)

pass 1: symbol table

pass 2: object code

(1) many-to-one approach (multiple threads on 1 kernel thread) (Solaris)

(2) One-to-one (Linux, Unix, Windows) = kernel does a lot of work

(3) many-to-many

Run Time Scheduler (User-mode)