Computing environments

1) traditional (multiprocessor systems)

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

HTTP

ftp://hostname pathname

DTOS NOS

Infrastructure/Software/Platform

↓ as a ↓

more control of OS — Service — OS doesn't matter — 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

API (Application Programming Interface)

(1) process management / thread management

CreateProcess(...) Windows

child → fork(), wait(); Unix/Linux

synchronize

thread = basic unit within a process

CreateThread()

posix? ↑ pthread_create,

exit(); execvp();

↑ family of 6 commands

(2) memory management

malloc(); memory allocation

free();

(3) file management

open(); read; write();

(4) communications

Socket programming

streams UNIX (solaris)

shake_jack? check it phone

Socket - end point structure containing addresses / bind(); listen(); accept(); send(); receive();

port # = extension, exact process on host — IP & Port # — bidirectional

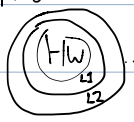
IP = telephone #

OS design methodologies

(1) monolithic approach - no structure, bigness, amidst alot of files, good for small systems, hard to adapt

MS-DOS, early UNIX — very efficient

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



not very efficient to access resources

THE ← operating system - 6 layers, batch system, multiprogramming, not multi-user

layer 5 - the user

layer 4 - user programs

layer 3 - device I/O management

layer 2 - communication between OS and console ← operator

layer 1 - memory and drum management

layer 0 - processor location and multiprogramming

enforce system security

layered system

(multics) multiple rings

(3) micro kernel approach — WINDOWS NT

minimum set of functions in kernel ← won't have to be changed

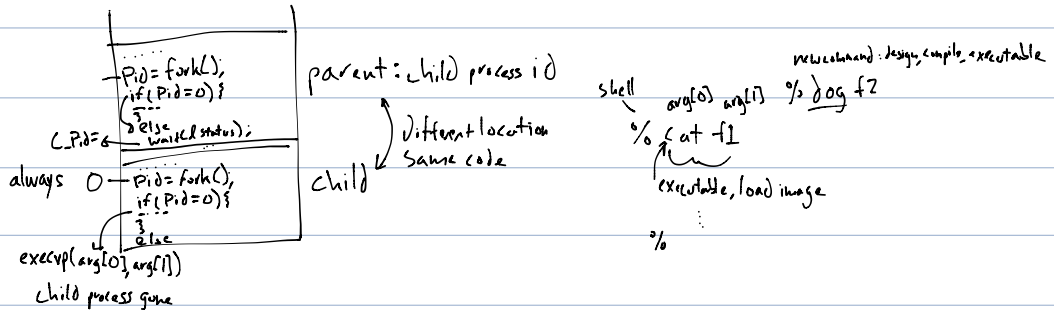
- Setting device registers ← need to have devices do something, addresses
- Switch CPU between processes
- manipulate MMU virtual → physical
- capture HW interrupts — Mach
- pass system calls ← ~~shared memory~~ message passing — TRUSTI

(4) Object-oriented approach — classes, inheritance, metaobjects, introspection, encapsulation
dynamic linking & dynamic loading

process - independent entity, program + execution environment — registers, ID, priority

posix API

Pid = fork(); system call to create child process, no parameter, returns pid (process)
Parent & child Pid 0-65,535



runcom — int runcom(char *cmd)
{ char *argv[max_args];
Pid_t child_pid, C_Pid;
int child_status;
parsecmd(cmd, argv);

```
child_pid = fork();
if (child_pid==0) {
execvp( argv[0], argv);   <- shouldn't see this
  print("unknown command \n")
exit(1);
  }
else{
  C_pid= wait (&child_status)
print("child process = %d \n", c_pid);
 return(child_status);
 }
}
```