6.033 Computer System Engineering
Spring 2009

6.033 Lecture 5: Operating System Organization

```
Plan for next few lectures
  general topic: modularity
  just talked about one module per computer (c/s)
    more details on net-based c/s later in course
  now: modules within one computer
    L5: o/s organization -- tools for enforced modularity
    L6: concurrency
    L7: threads
  o/s interesting design artifact in itself

O/S Goals
  multiplexing: one computer, many programs
  cooperation: help programs interact / communicate / share data
  protection (bugs, privacy)
  portability (hide h/w differences and e.g. amt of RAM)
  performance (improve, and don't get in the way)

Key techniques to achieve those goals?
  virtualization
  abstraction

Virtualization
  computer has h/w resources
  a few CPUs, one mem system, one disk, one net i/f, one display
  but you want to run many programs (X, editor, compiler, browser, &c)
  idea: give each program a set of "virtual" resources
    as if it had its own computer
    CPU, memory

Virtualization example: CPU
  multiplex one CPU among many programs
  so each program thinks it has a dedicated CPU
  give the CPU to each in turn for 1/n-th of the time
  h/w timer that interrupts 100 times/second
  save registers of current program (CPU registers)
  load previously-saved registers of another program
  continue
  transparent to the programs!
  much more to say about this in the next few lectures

some systems virtualize all h/w
  "virtual machine monitors"
  give each subsystem (program?) a complete virtual computer
    precisely mimic complete hardware interface
  early IBM operating systems shared computer this way
    one computer, many VMs
    each user can run their own o/s
  VMWare, Parallels on Mac, Microsoft Virtual PC
  virtualization by itself makes cooperation hard
    compiler wants to see editor's output
    not so easy if each thinks it has a dedicated virtual disk

Abstraction
  abstraction: virtualize but change interface
```

change interfaces to allow sharing, cooperation, portability

Abstraction examples:
  really "abstract virtual resources"
  disk -> file system
  display -> window system
  cpu -> only "safe" instrs, e.g. no access to MMU, clock, &c
  ??? -> pipes

What does an abstraction look like?
  usually presented to programmer as set of "system calls"
  see slide (l5.ppt)
  you're familiar with FS from UNIX paper
  chdir asks the kernel to change dir
  rest of code reads a file
  system call looks like procedure, we'll see it's different internally

```
main() {
  int fd, n;
  char buf[512];

  chdir("/usr/rtm");

  fd = open("quiz.txt", 0);
  n = read(fd, buf, 512);
  write(1, buf, n);
  close(fd);
}
```

How to manage / implement abstractions?
  how to enforce modularity?
    e.g. want prog to use FS *only* via system calls
  mechanism: kernel vs user
  [diagram: kernel, programs, disk, &c]
  kernel can do anything it likes to h/w
  user program can *only* directly use own memory, nothing else
    asks kernel to do things via system calls
  w/ this arrangement, enforcement more clear, two pieces:
    keep programs inside their own address spaces
    control transfers between user and kernel

enforcing address space boundaries
  use per-program pagemap to prohibit writes outside address space
    kernel switches pagemaps as it switches programs
    that's how o/s protects one program from another
  program's address space split
    kernel in high addresses
    user in low addresses
  PTEs have two sets of access flags: KR, KW, UR, UW
    all pages have KR, KW
    only user pages have UR, UW
    convenient: lets kernel directly use user memory, to impl sys calls

CPU has a "supervisor mode" flag
  if on, Kx PTE flags apply, modify MMU, talk to devices
  if off, Ux PTE flags, no devices

```
  need to ensure supervisor mode = on only when exec kernel code

orderly transfer from user to kernel
  1. set supervisor flag
  2. switch address spaces
  3. jmp into kernel
  and prevent user from jumping just anywhere in kernel
  and can't trust/user user stack
  can't trust anything user provides, e.g. syscall arguments

system call mechanics
  App:
    chdir("/usr/rtm")
      R0 <- 12 (syscall number)
      R1 <- addr of "/usr/rtm"
      SYSCALL
  SYSCALL instruction:
    save user program state (PC, SP)
    set supervisor mode flag
    jump to kernel syscall handler (fixed location)
  Kernel syscall handler:
    save user registers in per-program table
    set SP to kernel stack
    call sys_chdir(), an ordinary C function
    ...
    restore user registers
    SYSRET
  SYSRET instruction:
    clear supervisor mode flag
    restore user PC, SP
    continue process execution

this is procedure call with enforced modularity
  though only protects kernel from user program
  user program can do only two things
    use just its own memory
    or jump into kernel, but then kernel is in charge
  SYSCALL combines setting of supervisor mode and jumping to known location
    would be a disaster to let user program specify target

how to use kernel's enforced modularity?
  how to arrange module boundaries for system services?
  two main camps: microkernel vs monolithic

the microkernel vision
  implement main o/s abstractions as user-space servers
    FS, net, windows/graphics
  apps, servers interact with messages
  kernel provides minimum to support servers
    inter-process communication (messages / RPC)
    processes -- address space + running program
  most interaction via messages
    only two main system calls: send() and recv()

why are microkernels attractive?
  elegant, simple
```

  direct support for client/server
  kernel is small, fewer bugs
  kernel can be optimized to do one thing well (messages)
  hard modularity among different servers, vs bugs
  uniform port scheme -> uniform security plan
    control access by giving (or not giving) port access rights
  easier to distribute (messages allows services on other hosts)

where did this idea go?
  implementations were slow around 1990 -- e.g. Mach
    many messages, esp if you have many servers
  tended to end up with a few huge servers, so little modularity win
  the message idea was influential, e.g. Apple's OSX

monolithic kernels -- the (so far) winning design
  it's too much of a pain to separate lots of servers
  easier and more efficient to have one big program

what's in the Linux kernel?

```
     FDs                processes

            FS

            disk
 term  net  cache

    drivers       alloc

   enet disk        RAM  CPU MMU
```

complex: 300+ sys calls, not just two+ as in microkernel
  much of complexity hidden beneath uniform interfaces -- OO style
  e.g. all storage devices look the same to the file system
    USB flash key, hard disk
  all network hardware looks the same to network code
  so not as complex as it looks

why have monolithic kernels mostly won vs microkernels?
  tradition, advantages of microkernels are subtle
  efficiency: procedure call or mem refs rather than messages
  efficiency: better algorithms via integration among modules
    balance between file system cache and amt of mem for processes
  easy to add microkernel-style messaging, user-level servers
    X, sshd, DNS, apache, database, printer

what doesn't work so well in [monolithic?] kernels?
  device drivers are buggy, cause crashes
  lots of rules e.g. for user pointers passed as arguments
    tricky bug-prone programming environment
  bad at shared resource management (mem, disk)
    compiler uses lots of mem -> laptop unuseable
    backup program uses disk -> laptop unuseable (so I don't back up...)
    modularity is soft for resource management

Summary

o/s virtualizes for sharing/multiplexing
abstracts for portability and cooperation
provides enforced modularity in two useful ways:
  program / kernel
  program / program
supports two kinds of c/s: program -> kernel, program -> program
next: specific techniques for client/server on single computer