6.033 L3: Operating Systems 2/12/2014

Recap: modularity, naming

Saw in first lecture how we can use client-server to isolate running programs from each other

But how can enforce modularity on one machine?

[slide: overall plan, next few lectures]

Operating System: make it easier to run applications on hardware
- **Multiplexing**: run many programs.
- **Isolation**: enforce modularity between programs.
  - Buggy or malicious program shouldn't crash other programs.
- **Cooperation**: allow programs to interact, share data.

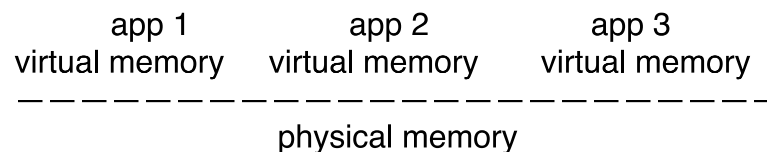
These three are about making one machine do what multiple machines in C/S did

- **Portability**: allow the same program to run on different hardware.
- **Performance**: help programs run fast.

(Show slide)

Focus in this lecture on first three.
[Demo: wilddemo]
Two programs on one computer
But they are isolated:
Two programs writing to same memory address but can’t crash each other
They may crash themselves
(Sharing terminal for output!)

Two main techniques to achieve these goals:
Virtualization: interpose between program & hardware, but provide same interface.
One physical memory, but give each program the illusion of its own memory.
Same for CPUs (next week's lecture) -- for now assume one CPU per program.
Diagram:



Abstraction: define new (typically hardware-independent) interfaces.
Only one disk, display, network card, etc.

But each app is able to use disk (e.g., file system directory), display (windows), etc.

Virtualizing memory.

Start with a single program.

(Slide)

CPU, DRAM, 32-bit addresses. DRAM stores code, data.

CPU loads and runs instructions (based on program counter).

CPU loads / stores other memory locations (based on instructions).

What if we store code and data for two programs?

One program's instructions can load / store other program's data!

Enforcing modularity: **virtual memory addressing**.

Idea: Create a *virtual memory*, where application has its own 32-bit address space; apps cannot refer to other apps memory!

OS manages translation from an apps virtual address to a real physical address

Uses a special piece of hardware called a *memory management unit*, or MMU.

Slide: CPU, MMU, DRAM with code & data for two programs.

Inside the MMU is a logical table from virtual address to physical address.

CPU points to a "virtual" address, entry in MMU.

MMU points to a "physical" address, location in DRAM.

No way for CPU to refer to a physical address -- must go through MMU.

(slides) Plan: maintain a separate MMU table for each program.

Program 1's table only contains physical addrs for program 1's code + data.

Program 2's table only contains physical addrs for program 2's code + data.

Benefit: programs don't need to be written to avoid each other's addresses!

Both programs can use virtual address 0x0000, if they want.

Each program's MMU table will map it to its own physical address.

How to represent this mapping?

Naive approach: one entry for each possible address (2^32).

Table would be too big: 4 bytes * 2^32 entries = 16GB.

Better approach: page tables (review from 6.004).

Instead of mapping individual addresses, map groups of addresses.

Typically, map 4KB of memory (i.e., 2^12 addresses) at a time.

This group is called a "page" of memory.

On a 32-bit system, this results in 2^20 different pages.

Table size is now manageable: 4 bytes * 2^20 entries = 4MB.

Example of page-level translation (show slide)
  Diagram: page table contains the following entries:
    VA   PA
    0 -> 3
    1 -> 0
    2 -> 4
    3 -> 5
  Processor issues LD R0, 0x00002148.
  CPU sends 0x000002148 to MMU.
  MMU uses top 20 bits to compute page number (0x00002).
  Translates virtual page number 2 into physical page number 4.
  Page offset in low 12 bits (0x148) remains the same.
  Physical address becomes 0x00004148, sent to DRAM.

<u>Where to store this page table?</u>
  Page table is typically itself stored in memory.
  The MMU has a register that stores the address of the current page table.
  Note, as with DNS, need to bootstrap -- otherwise, chicken-and-egg problem.
  MMU's page table register must have the <u>physical</u> address of page table.

<u>Naming view of virtual memory.</u>
  Virtual addresses are names.
  Physical addresses are values.
  MMU performs lookup in a table.
  Context is the pointer to the current table in the MMU.

  Benefits of naming:
    - Hiding: program cannot access physical memory that's not in its page table.
      Simply can't name it!
    - (Controlled) sharing: two page tables can map same physical page.
    - Indirection: can control where/how a program's memory is stored.
      If two programs use same library, map library pages read-only in both PTs.
      If we run out of physical memory, temporarily write pages to disk.
      Program doesn't need to know about any of this.

Page table typically stores additional information. [slide: x86 page table entry]
  Present bit: is this virtual address valid?
  Writable bit: are writes to this virtual address allowed?
  User bit: can the program access this virtual address, or only the OS?
    (A little more about this later..)

<u>What is a page fault?</u>
  If access is not permitted, MMU triggers a "page fault".
    Forces processor to execute pre-defined “page fault” handler.
    Can either stop program & raise error, or update page table & resume.

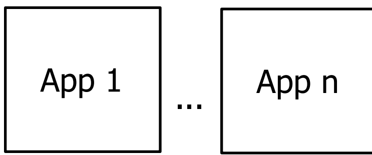
(For example, pages may be spilled to disk and read in; allows more virtual memory than physical memory)

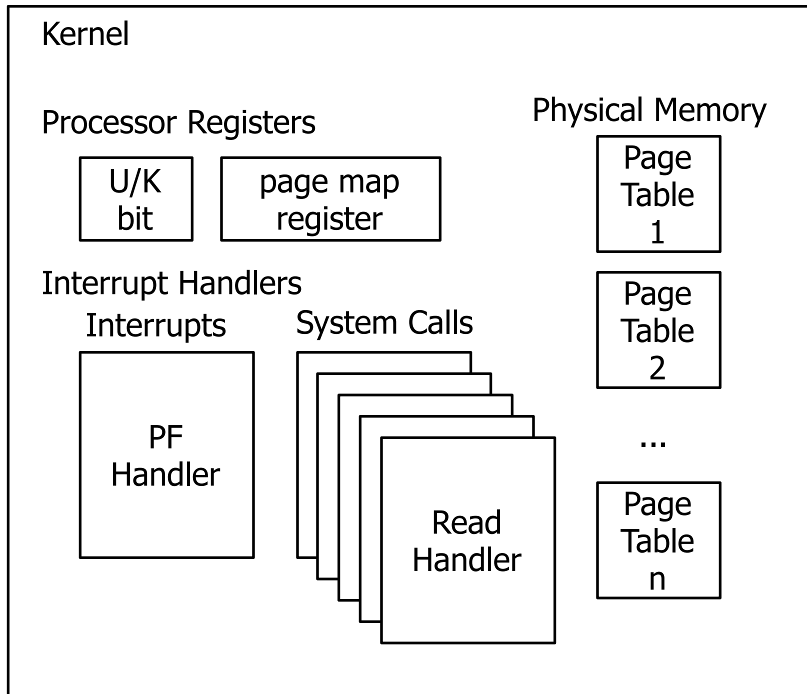
<u>Protecting Page Tables</u>

Can a determined program still access another program's memory?
- So far, yes, because it could change the page table address register in MMU.
- Could also modify page fault hanlder
- To prevent this, need to change the processor.
  - Two modes of execution: "user" mode (U) and "kernel" mode (K).
  - Processor maintains a "U/K" (U/S) bit that stores current execution mode.
  - In user mode, processor does not allow modifying page table register or U/K bit
  - In kernel mode, processor does allow it.
  - Code that executes in kernel mode is called the *OS kernel.*

Kernel is trusted piece of software that sets page maps, U/K register, and interrupt handlers

Diagram (show proc bits, handlers)





How does the transition happen between user and kernel mode (and back)?
(show slide)
If app access memory ("page fault") it doesn't have access to, or calls
Int x : special instruction that calls interrupt x
  set u/k bit to k
  lookup x in table of handlers
  run handler
  call iret instruction to unset bit U/K bit and return to app

Kernel code must be paranoid about anything that user-mode code has access to.
E.g., cannot assume user-mode code has a valid stack, etc.

How to protect the kernel's memory from user programs?
Strawman: switch page table pointer when switching to kernel mode.
In practice, kernel often needs to access memory of user program.

Alternative solution: use the "U" bit in page table entries.
Page table entries for kernel code & data don't have the "U" bit.
Those pages only accessible when executing in kernel mode.
Typically, kernel code & data located at the top of virtual address space.

OS abstractions.
Virtualization works well for memory (& CPU).
Does not work as well for disk, display, network.
Why not? Not portable, does not allow sharing, ...

OS kernel typically provides new abstractions for these devices.
Disk: file system.
Display: windows.
Network: TCP/UDP connections.

[slide: example Unix application using the file system]

Major challenge: designing good abstractions!
Next hands-on and recitations will discuss Unix's choice of OS abstractions.

Abstractions presented as a set of function calls, called "system calls"

[Demo: system calls]
strace ls cp rm

Kernel code will implement the abstraction's functions (open, read, ...).
Hide raw interface from apps; e.g., apps cannot read/write directly to disk

How to implement these abstractions?

Can we write functions like open() and read() in a library? No!

Implementation of these functions requires direct access to the disk.

If kernel allows programs to access disk, buggy programs can corrupt disk.

Instead, we want programs to access disk _only_ via FS abstractions.

system calls implemented via interrupts

Approach: make each system call a separate interrupt into the kernel.

Kernel recap:

- kernel enforces modularity
  - virtualizes memory (and cpu)
  - abstractions (e.g., file system for disk, windows for display)
- has to be coded careful ==> bugs introduce modularity holes!

[slide: summary]