L3: Operating Systems

Frans Kaashoek

kaashoek@mit.edu

6.033 Spring 2013

OS: New topic [4 lectures]

- Case study of widely-used interesting system
  - Virtual memory system, file system, processes,
- Illustrates ideas from first lectures:
  - OSs supports client/server computing within a single computer
  - OSs have a naming system
- Introduce new ideas and techniques: kernel, files, locks, etc.
- Example: UNIX (Linux, BSDs, etc.)

A Computer

how to make it to do something useful?

Inside the computer

OS goal

- Turn hardware into something usable
  - Multiplexing: run many programs
  - Isolation: enforce modularity between programs
  - Cooperation: allow programs to interact
  - Portability: allow same program to run on different hardware
  - Performance: help programs to run fast

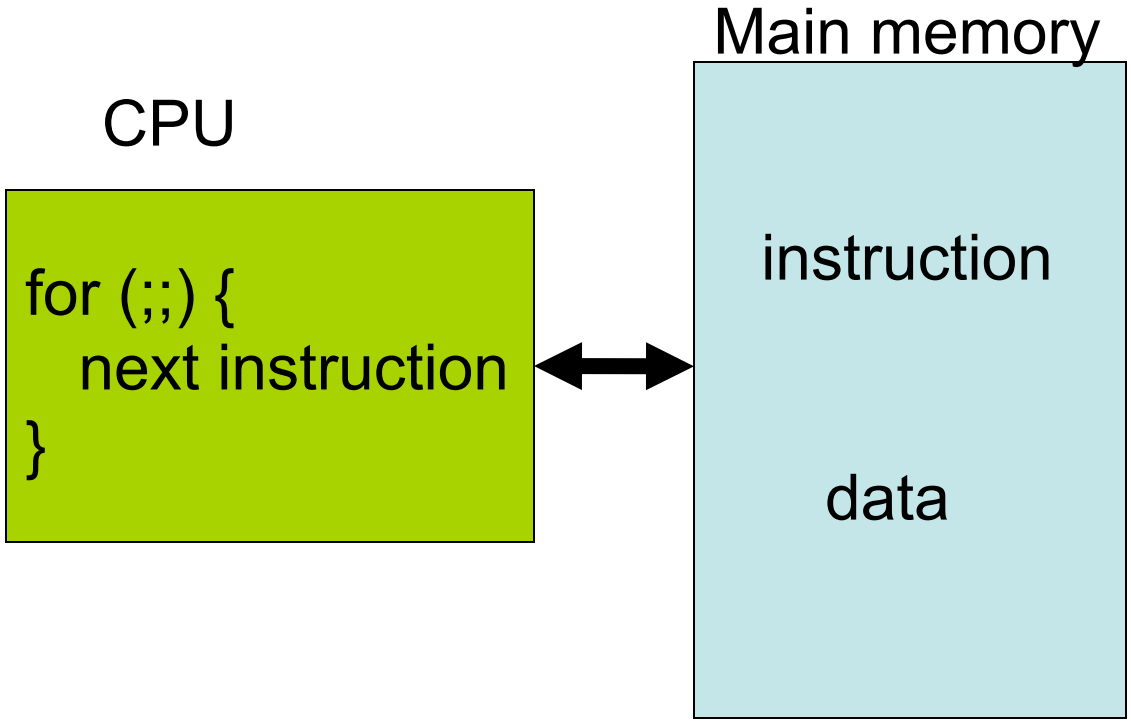
Main challenges

- Isolation:
  - Buggy or malicious program shouldn't crash other programs
- Controlled sharing:
  - Programs must be able to cooperate

Two main techniques

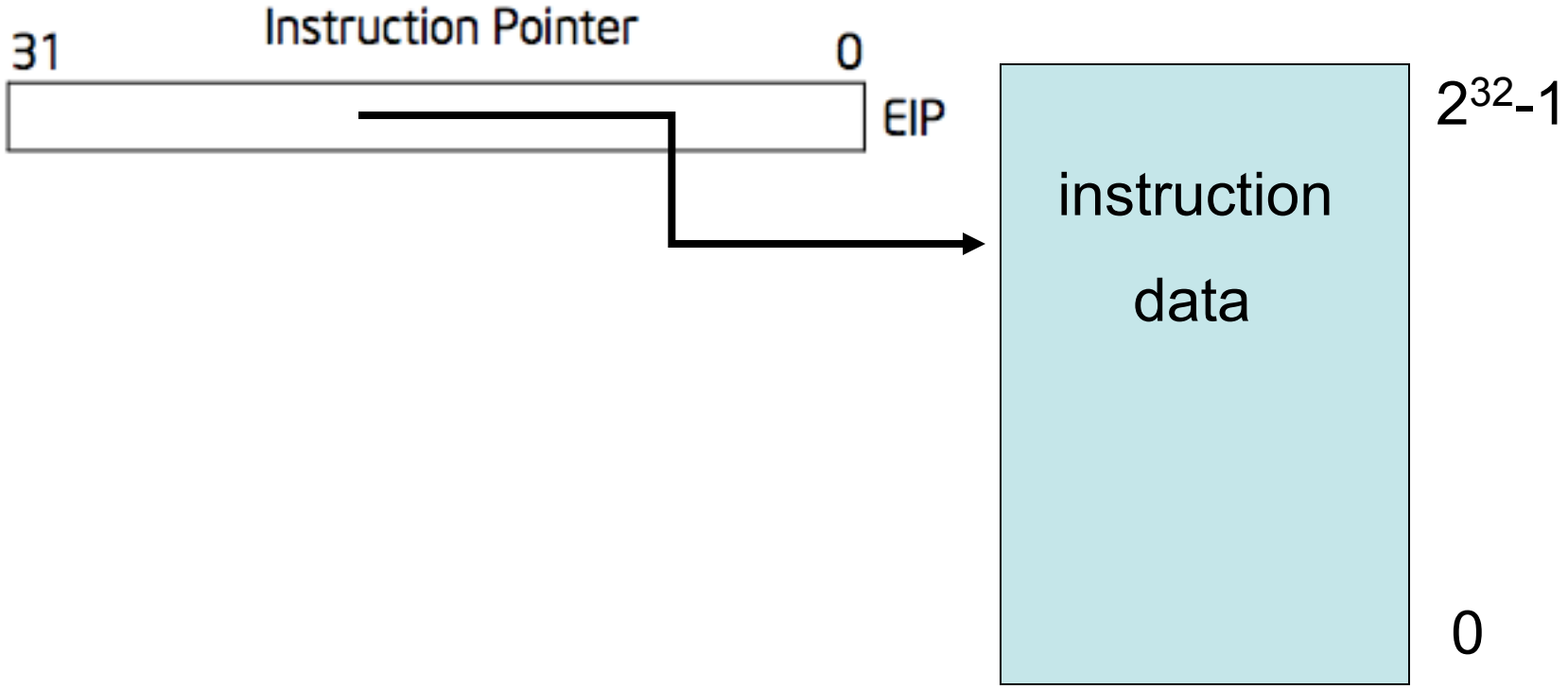
- *Virtualization*: interpose between program & hardware, but provide same interface
  - Give the program the illusion of its own memory
  - Give the program the illusion of its own CPU
- *Abstraction*: define new hardware-independent interfaces
  - Disk -> Files system
  - Display -> Window system

Running a single program



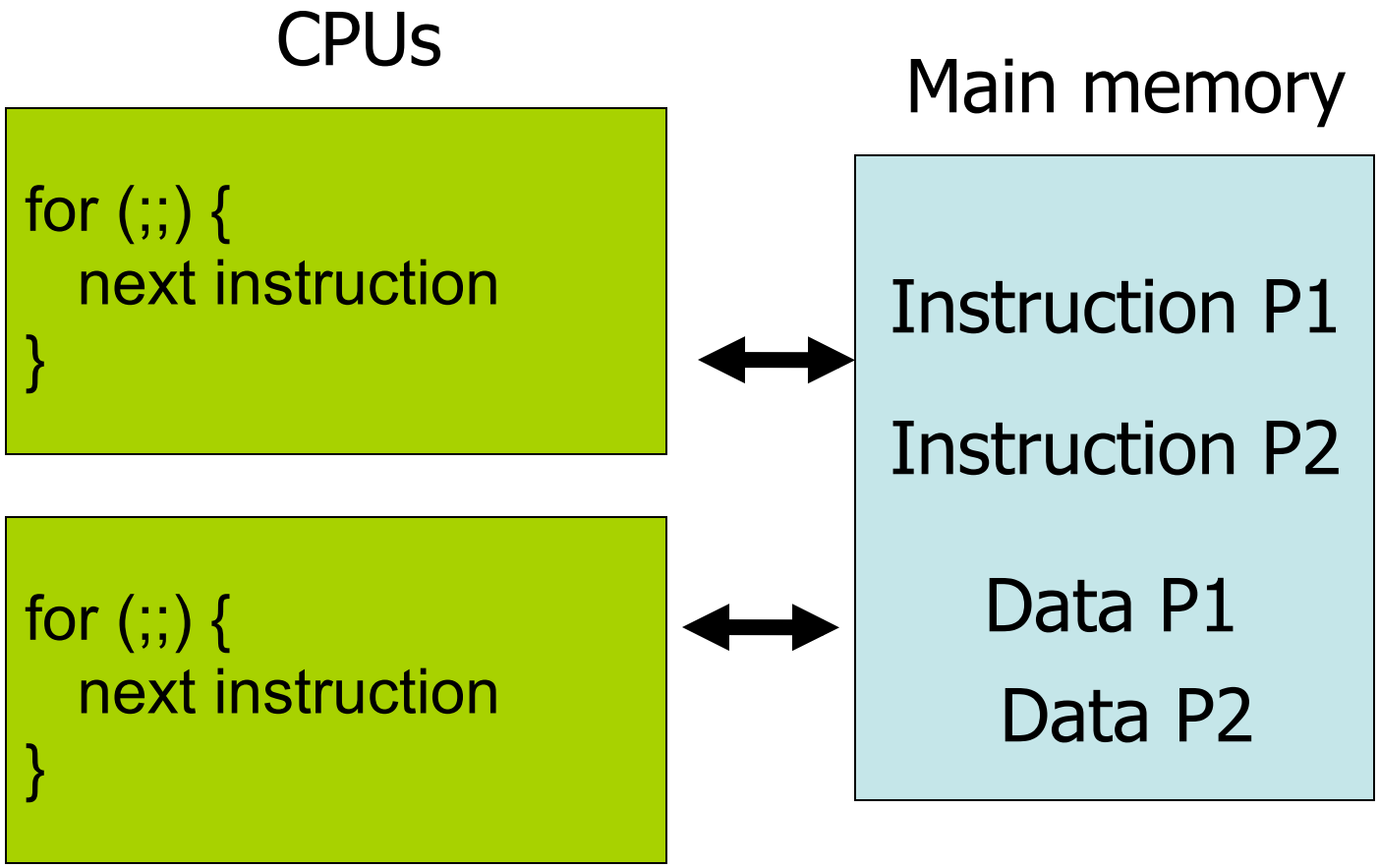
- Memory holds *instructions* and *data*
- CPU *interpreter* of instructions

x86 implementation

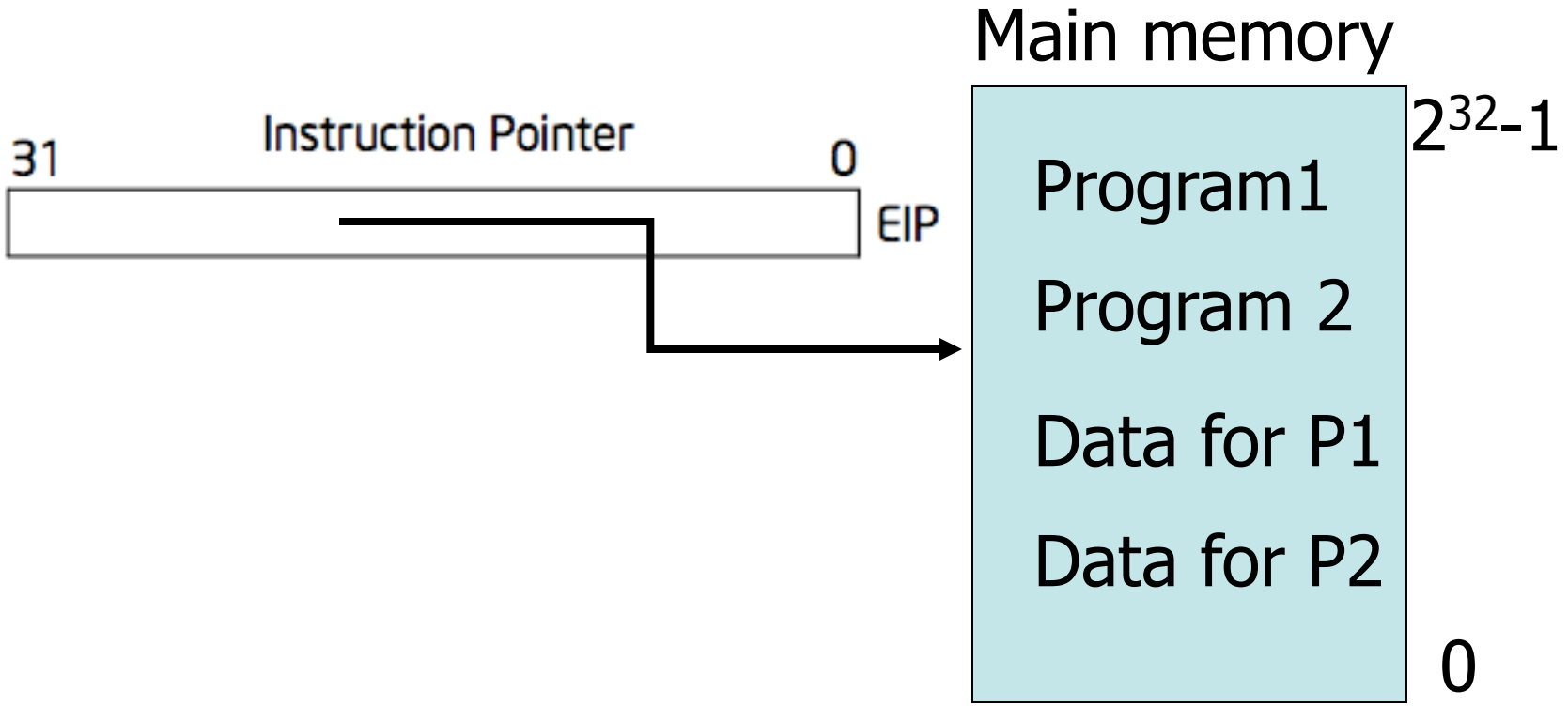


- EIP is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and conditional JMP

Several programs



Problem: no boundaries

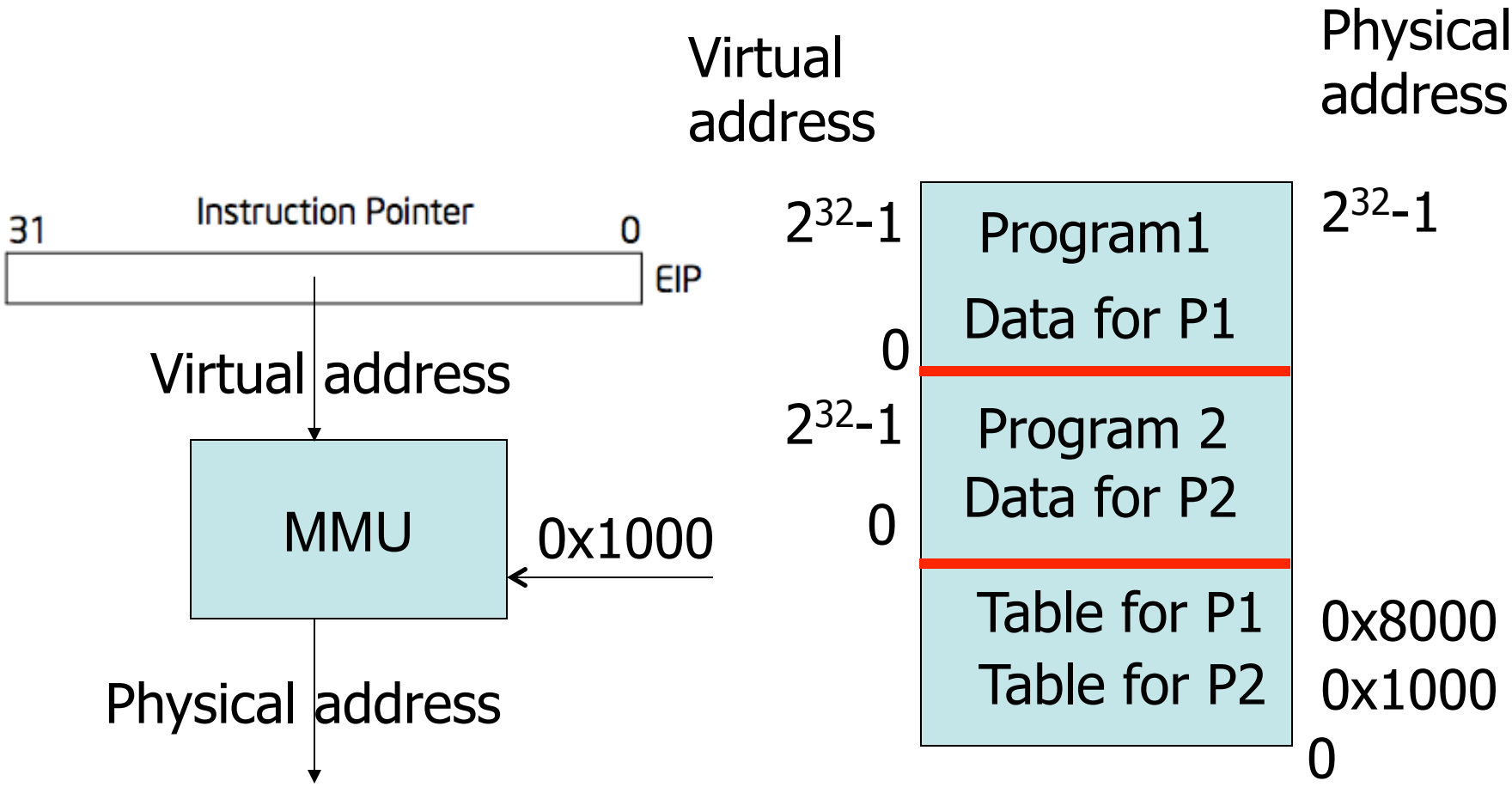


- A program can modify other programs data
- A program jumps into other program's code
- A program may get into an infinite loop

Goal: enforcing modularity

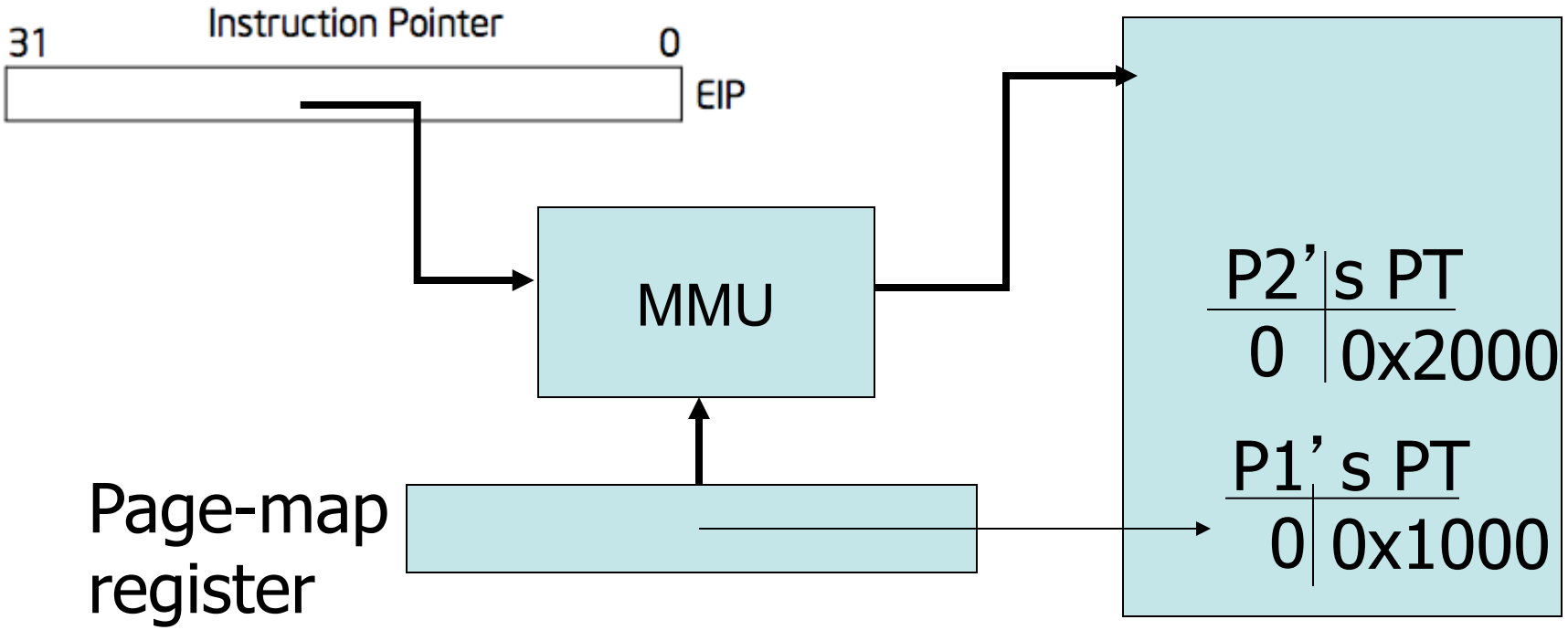
- Give each program its private memory for code, stack, and data
- Prevent one program from getting out of its memory
- Allowing sharing between programs when needed
- Force programs to share processor [next week]

Approach: memory virtualization



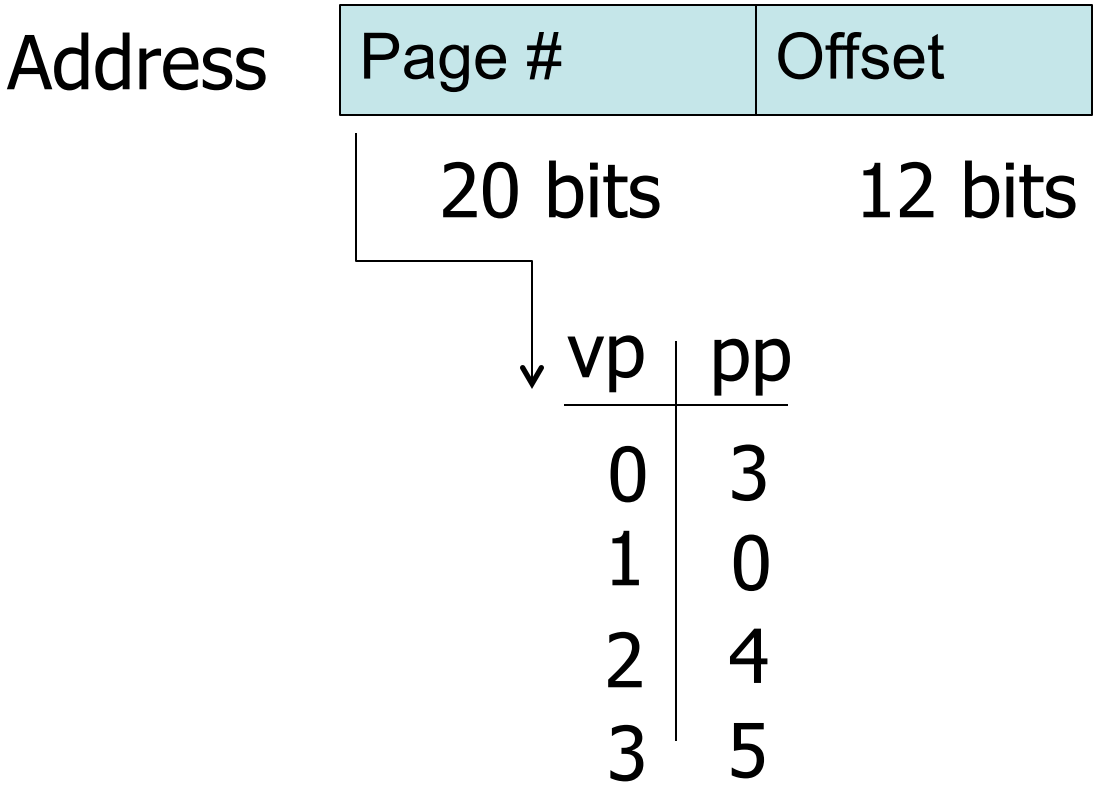
- P1: LD r0, 0x0000 translated with Table 1
- P2: LD r0, 0x0000 translated with Table 2

Table records mapping



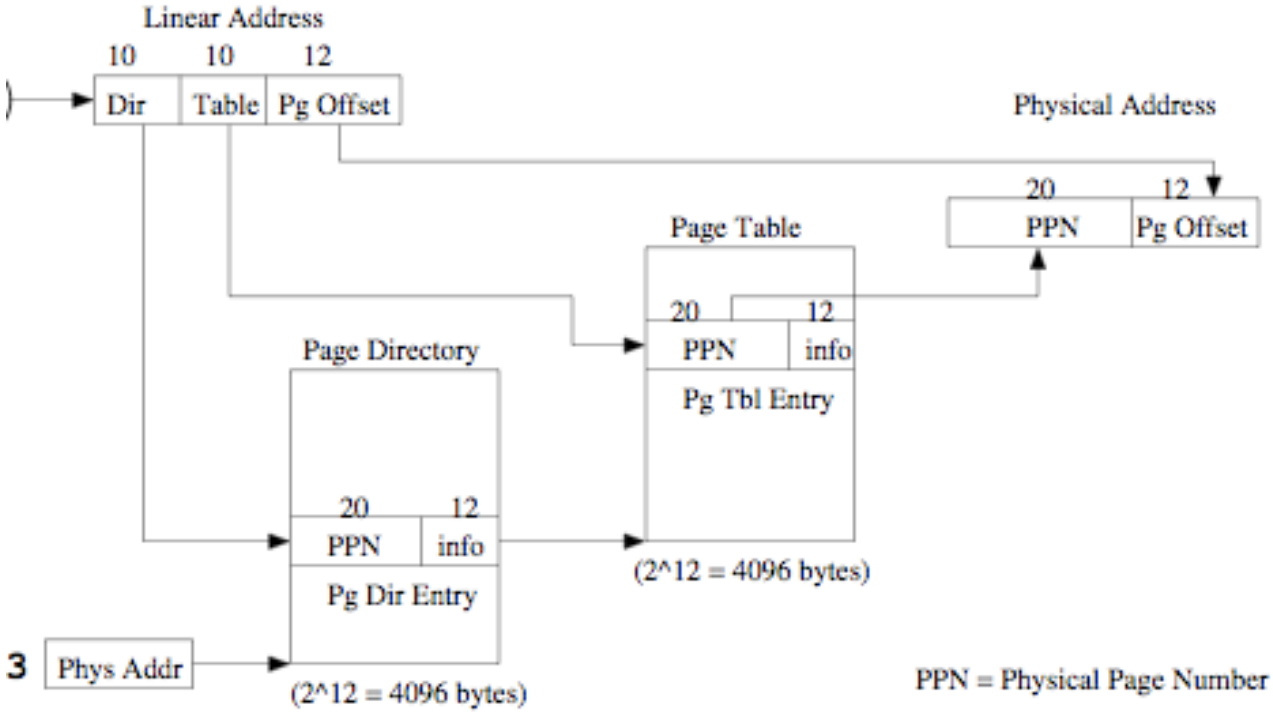
- Each program has its own translation map
  - Physical memory doesn't have to be contiguous
- P1: 0x0000 -> 0x1000
- P2: 0x0000 -> 0x2000

Space-efficient map



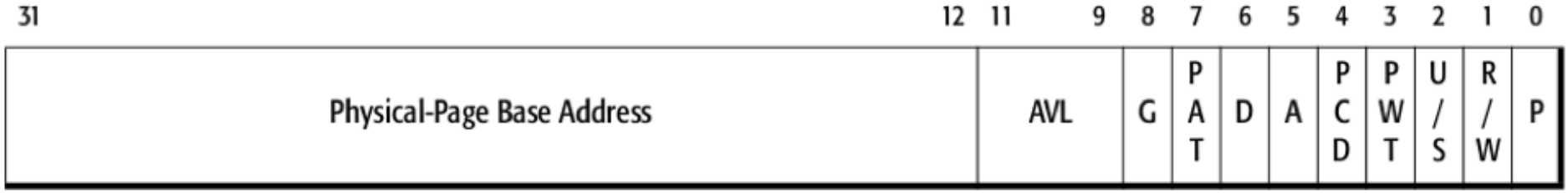
- 0x02020 -> 4 * 4096 + 0x20 = 0x4020

Intel x86-32 two-level page table



- Page size is 4,096 bytes
  - 1,048,576 pages in 2^{32}
  - Two-level structure to translate

x86 page table entry



- R/W: writable?
  - Page fault when W = 0 and writing
- U/S: user mode references allowed?
  - Page fault when U = 0 and user references address
- P: present?
  - Page fault when P = 0

Page fault

- Switches processor to a predefined handler in OS software
  - Handler can stop program and raise error
  - Handler can update the page map and resume at faulted instruction
- Allows OS to change page tables while program is running

Naming view

- Apply naming model:
  - Name = virtual address
  - Value = physical address
  - Context = Page map
  - Lookup algorithm: index into page map
- Naming benefits
  - Sharing
  - Hiding
  - Indirection (demand paging, zero-fill, copy-on-write, …)

Problem: how to protect tables?

- Malicious program could change page table address register
- Malicious program could change page fault handler

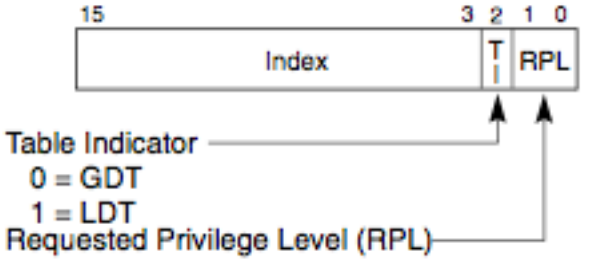
User and kernel mode

U

Code in U mode

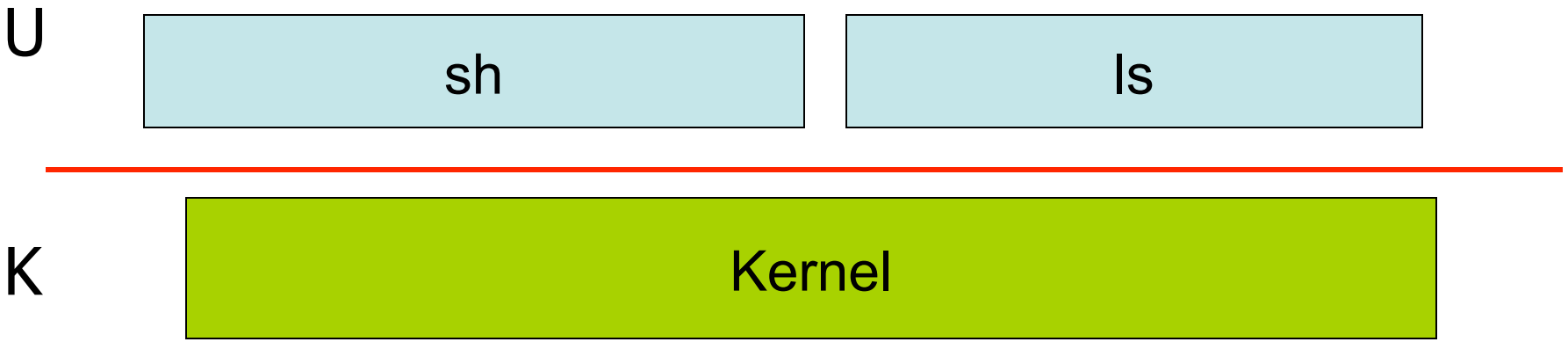
K

Code in K mode



- Processor maintains U/K bit to distinguish between user and kernel mode
- Code in K mode can set page table register and U/K bit

What is a kernel?



- The code running in kernel mode
  - Trusted program: e.g., sets page-map, U/K register
  - All interrupt handlers (e.g. page fault) run in kernel mode

How transfer from U to K, and back?

- Special instruction: e.g., int #
- Processor actions on int #:
  - Set U/K bit to K
  - Lookup # in table of handlers
  - Run handler
- Another instruction for return (e.g., reti)
  - Kernel sets U/K bit to U
  - Calls reti

How to protect kernel's memory from user programs?

- Straw man plan: switch page table pointer when changing modes
- Alternative: use U bit in page table entry
  - Pages for kernel code don't have U bit
  - Kernel code and user code in a single address space
  - Kernel can refer to user data, but not the other way around

Abstractions

- Pure virtualizing is often not enough
  - E.g., Portability
  - E.g., Cooperation
- Example OS abstractions:
  - Disk -> FS
  - Display -> Windows
  - DRAM -> heap w. allocate/deallocate
- Design of abstraction important
  - Study UNIX abstractions

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

  chdir("/usr/kaashoek");

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

Where do abstractions live?

- Library in user space?
  - No! Program must use disk only through abstraction
- Use kernel to enforce abstractions
- Each OS call changes into the kernel
- Kernel code implements abstractions

Kernel enforces modularity

- Kernel code must set page tables correctly
- Kernel code is responsible for enforcing abstractions
- Any bug in kernel can lead to hole in our enforced modularity
- [More about this in 2 weeks]

Summary

- Two key OS techniques
  - Virtualization allows programs to share hardware
  - Abstractions provide portability, cooperation
    - See Unix paper
- OS kernel enforces modularity
  - Program vs program
  - Program vs kernel