


# COS 318: Operating Systems

## Protection and Virtual Memory


Jaswinder Pal Singh and a Fabulous Course Staff  
Computer Science Department  
Princeton University

(<http://www.cs.princeton.edu/courses/cos318/>)



## Outline


- ◆ Protection Mechanisms and OS Structures
- ◆ Virtual Memory: Protection and Address Translation



2

## Some Protection Goals

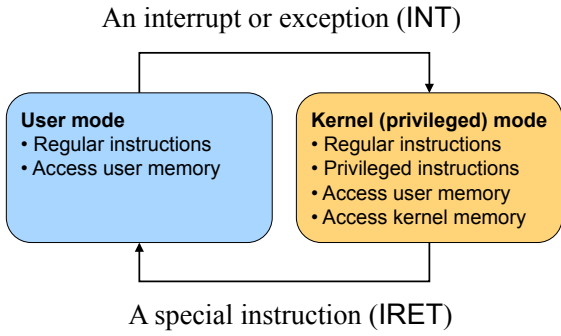
- ◆ CPU
  - Kernel has the ability to take CPU away from users to prevent a user from using the CPU forever
  - Users should not have such an ability
- ◆ Memory
  - Prevent a user from accessing others' data
  - Prevent users from modifying kernel code and data structures
- ◆ I/O
  - Prevent users from performing "illegal" I/Os
- ◆ Question
  - What's the difference between protection and security?



3

## Architecture Support for Processing/CPU Protection

- **Privileged Mode**



```


graph TD
    User[User mode] -- "An interrupt or exception (INT)" --> Kernel[Kernel (privileged) mode]
    Kernel -- "A special instruction (IRET)" --> User
  
```

**User mode**

- Regular instructions
- Access user memory

**Kernel (privileged) mode**

- Regular instructions
- Privileged instructions
- Access user memory
- Access kernel memory



4

## Privileged Instruction Examples

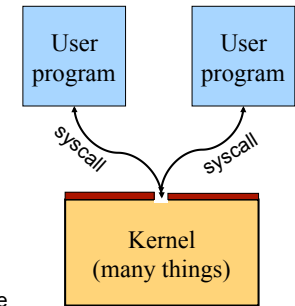
- ◆ Memory address mapping
- ◆ Flush or invalidate data cache
- ◆ Invalidate TLB entries
- ◆ Load and read system registers
- ◆ Change processor modes from kernel to user
- ◆ Change the voltage and frequency of processor
- ◆ Halt a processor
- ◆ Reset a processor
- ◆ Perform I/O operations



5

## Monolithic

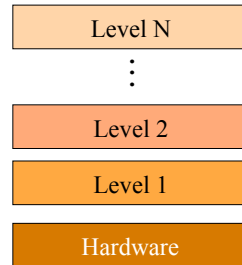
- ◆ All kernel routines are together, linked in single large executable
  - Each can call any other
  - Services and utilities
- ◆ Provides a system call API
- ◆ Examples:
  - Linux, BSD Unix, Windows, ...
- ◆ Pros
  - Shared kernel space
  - Good performance
- ◆ Cons
  - Instability: crash in any procedure brings system down
  - Unweildy / difficult to maintain, extend



6

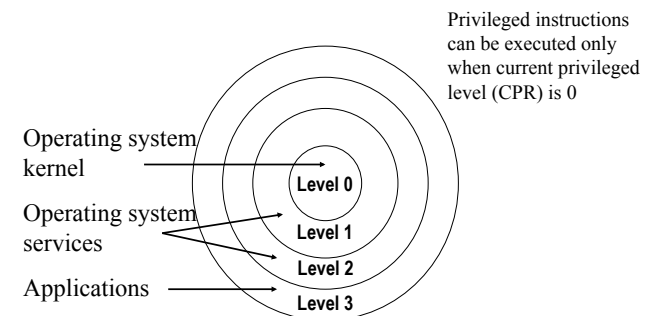
## Layered Structure

- ◆ Hiding information at each layer
- ◆ Layered dependency
- ◆ Examples
  - THE (6 layers)
    - Mostly for functionality splitting
  - MS-DOS (4 layers)
- ◆ Pros
  - Layered abstraction
  - Separation of concerns, elegance
- ◆ Cons
  - Inefficiency
  - Inflexible



7

## Possible Implementation: x86 Protection Rings



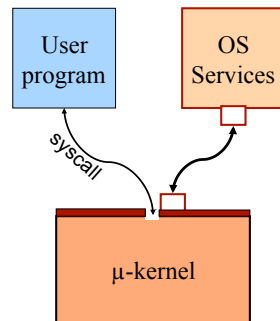
Privileged instructions can be executed only when current privileged level (CPR) is 0



8

## Microkernel Structure

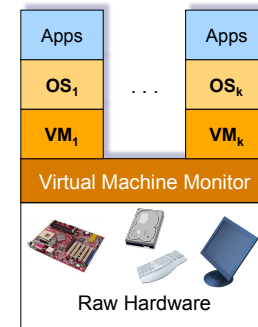
- ◆ Services implemented as regular processes
- ◆ Micro-kernel obtains services for users by messaging with services
- ◆ Examples:
  - Mach, Taos, L4, OS-X
- ◆ Pros?
  - Flexibility
  - Fault isolation
- ◆ Cons?
  - Inefficient (boundary crossings)
  - Inconvenient to share data between kernel and services
  - Just shifts the problem, to level with less protection, testing?



9

## Virtual Machine

- ◆ Virtual machine monitor
  - Virtualize hardware
  - Run several OSES
  - Examples
    - IBM VM/370
    - Java VM
    - VMWare, Xen
- ◆ What would you use virtual machine for?



10

## Memory Protection

- ◆ Kernel vs. user mode, plus
- ◆ Virtual address spaces and Address Translation

### Physical memory

No protection  
  
Limited size  
  
Sharing visible to programs

### Abstraction: virtual memory

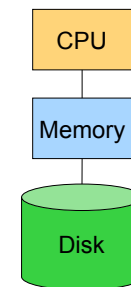
Every program isolated from all others and from the OS  
  
Illusion of "infinite" memory  
  
Transparent --- can't tell if physical memory is shared

Virtual addresses are translated to physical addresses



## The Big Picture

- ◆ DRAM is fast, but relatively expensive
- ◆ Disk is inexpensive, but slow
  - 100X less expensive
  - 100,000X longer latency
  - 1000X less bandwidth
- ◆ Our goals
  - Run programs as efficiently as possible
  - Make the system as safe as possible



13

## Issues

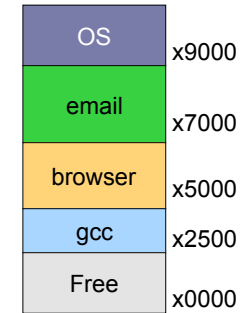
- ◆ Many processes
  - The more processes a system can handle, the better
- ◆ Address space size
  - Many processes whose total size may exceed memory
  - Even one process may exceed physical memory size
- ◆ Protection
  - A user process should not crash the system
  - A user process should not do bad things to other processes



14

## Consider A Simple System

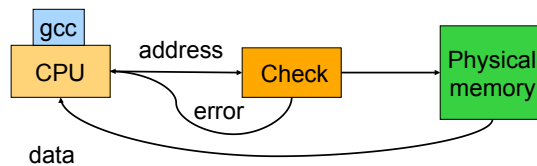
- ◆ Only physical memory
  - Applications use physical memory directly
- ◆ Run three processes
  - Email, browser, gcc
- ◆ What if
  - gcc has an address error?
  - browser writes at x7050?
  - email needs to expand?
  - browser needs more memory than is on the machine?



15

## Handling Protection

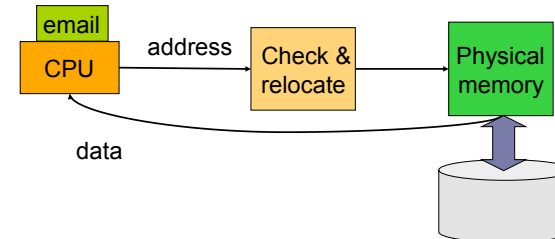
- ◆ Errors in one process should not affect others
- ◆ For each process, check each load and store instruction to allow only legal memory references



16

## Handling Finiteness: Relocation

- ◆ A process should be able to run regardless of where its data are physically placed or physical memory size
- ◆ Give each process a large, static “fake” address space that is large and contiguous and entirely its own
- ◆ As a process runs, relocate or map each load and store to addresses in actual physical memory



17

## Virtual Memory

- ◆ Flexible
  - Processes (and data) can move in memory as they execute, and can be part in memory and part on disk
- ◆ Simple
  - Applications generate loads and stores to addresses in the contiguous, large, “fake” address space
- ◆ Efficient
  - 20/80 rule: 20% of memory gets 80% of references
  - Keep the 20% in physical memory (a form of caching)



18

## Address Mapping and Granularity

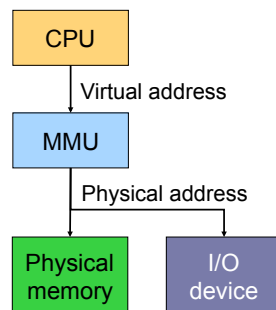
- ◆ Must have some “mapping” mechanism
  - Map virtual addresses to physical addresses in RAM or disk
- ◆ Mapping must have some granularity
  - Finer granularity provides more flexibility
  - Finer granularity requires more mapping information



19

## Generic Address Translation

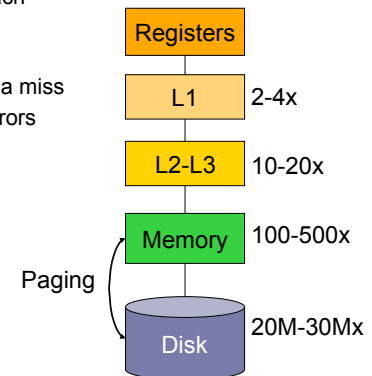
- ◆ Memory Management Unit (MMU) translates virtual address into physical address for each load and store
- ◆ Combination of hardware and (privileged) software controls the translation
- ◆ CPU view
  - Virtual addresses
  - Each process has its own memory space [0, high] – virtual address space
- ◆ Memory or I/O device view
  - Physical addresses



20

## Goals of Translation

- ◆ Implicit translation for each memory reference
- ◆ A hit should be very fast
- ◆ Trigger an exception on a miss
- ◆ Protected from user's errors



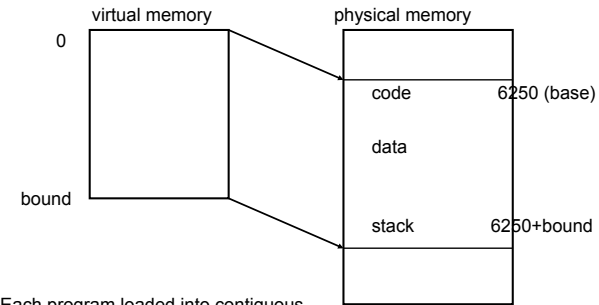
21

## Address Translation Methods

- ◆ Base and Bounds
- ◆ Segmentation
- ◆ Paging
- ◆ Multilevel translation
- ◆ Inverted page tables



## Base and Bounds

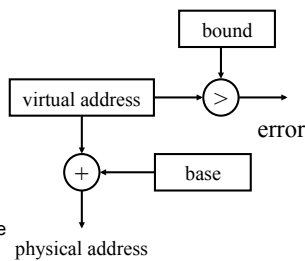


Each program loaded into contiguous regions of physical memory.  
Example on next slide



## Base and Bound (or Limit) Example: Cray-I

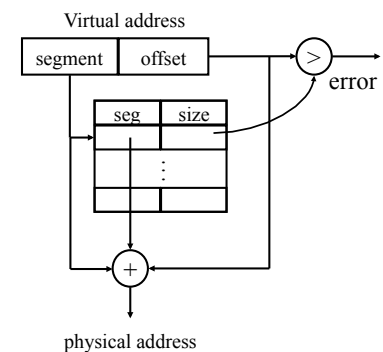
- ◆ Protection
  - A process can only access physical memory in [base, base+bound]
- ◆ On a context switch
  - Save/restore base, bound regs
- ◆ Pros
  - Simple
  - Inexpensive (Hardware cost: 2 registers, adder, comparator)
- ◆ Cons
  - Can't fit all processes in memory, have to swap
  - Fragmentation in memory
  - Relocate processes when they grow
  - Compare and add on every instruction



24

## Segmentation

- ◆ Every process has table of (seg, size) for its segments
- ◆ Treats (seg, size) as a finer-grained (base, bound)
- ◆ Protection
  - Every entry contains rights
- ◆ On a context switch
  - Save/restore table in kernel mem
- ◆ Pros
  - Provides logical protection: programmer knows program and so segments
  - Therefore efficient
  - Easy to share data
- ◆ Cons
  - Complex management
  - Fragmentation



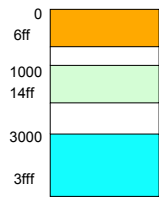
25

## Segmentation Example

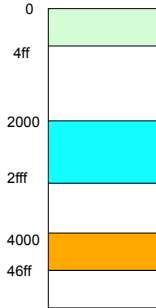
(assume 2 bit segment ID, 12 bit segment offset)

v-segment #	p-segment start	segment size
code (00)	0x4000	0x700
data (01)	0	0x500
- (10)	0	0
stack (11)	0x2000	0x1000

virtual memory



physical memory



## Segmentation Example (Cont'd)

Virtual memory for strlen(x)

```
Main: 240      store 1108, r2
      244      store pc+8, r31
      248      jump 360
      24c      ...
      ...
      strlen: 360  loadbyte (r2), r3
      ...
      420      jump (r31)
      ...
      x: 1108    a b c \0
      ...
```

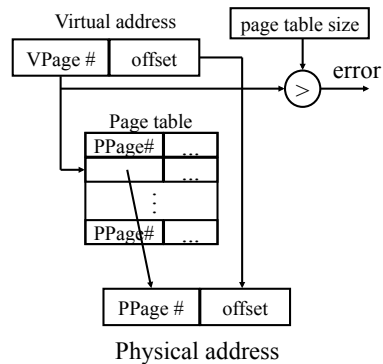
physical memory for strlen(x)

```
x: 108      a b c \0
      ...
Main: 4240   store 1108, r2
      4244   store pc+8, r31
      4248   jump 360
      424c   ...
      ...
      strlen: 4360 loadbyte (r2), r3
      ...
      4420   jump (r31)
      ...
```



## Paging

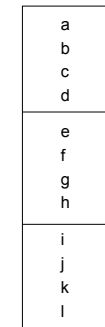
- ◆ Use a fixed size unit called page instead of segment
- ◆ Use page table to translate
- ◆ Various bits in each entry
- ◆ Context switch
  - Similar to segmentation
- ◆ What should page size be?
- ◆ Pros
  - Simple allocation
  - Easy to share
- ◆ Cons
  - Big table
  - PTEs even for big holes in memory



28

## Paging example

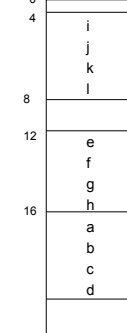
virtual memory



4
3
1

page size: 4 bytes

physical memory



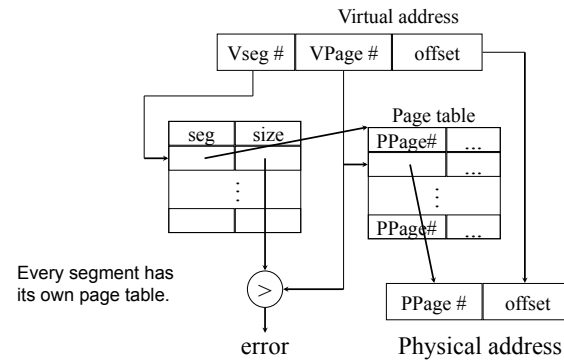
## How Many PTEs Do We Need?

- ◆ Assume 4KB page
  - Needs “low order” 12 bits to address byte within page
- ◆ Worst case for 32-bit address machine
  - # of processes  $\times 2^{20}$
  - $2^{20}$  PTEs per page table (~4Mbytes), but there might be 10K processes. They won't fit in memory together
- ◆ What about 64-bit address machine?
  - # of processes  $\times 2^{52}$
  - A page table cannot fit in a disk ( $2^{52}$  PTEs = 16PBytes)!

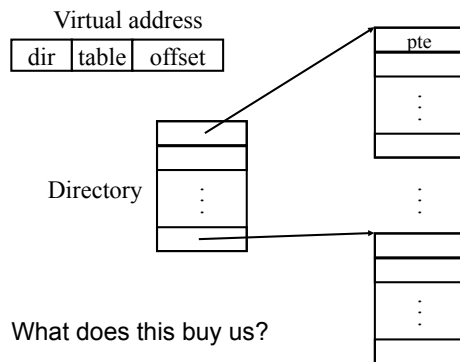


30

## Segmentation with paging

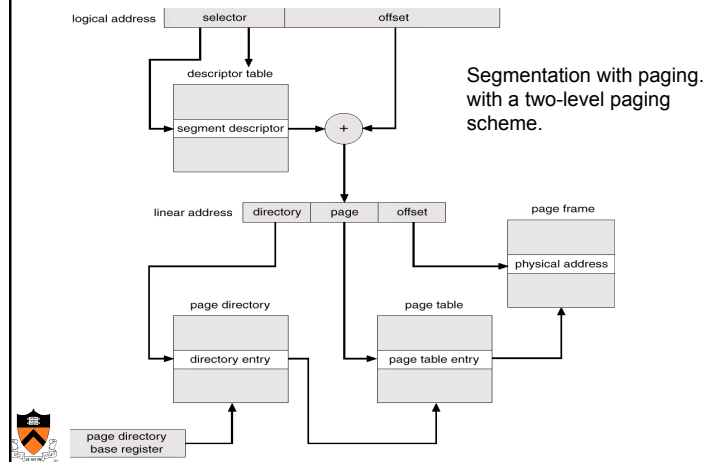


## Multiple-Level Page Tables



32

## Intel 30386 address translation



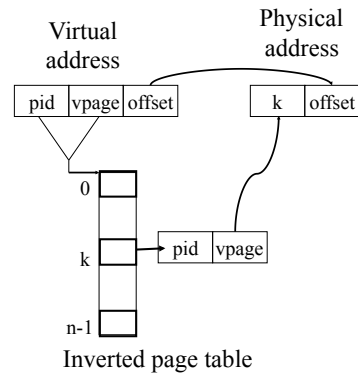


## Inverted Page Tables

- ◆ Main idea
  - One PTE for each physical page frame
  - Hash (Vpage, pid) to Ppage#

- ◆ Pros
  - Small page table for large address space

- ◆ Cons
  - Lookup is difficult
  - Overhead of managing hash chains, etc



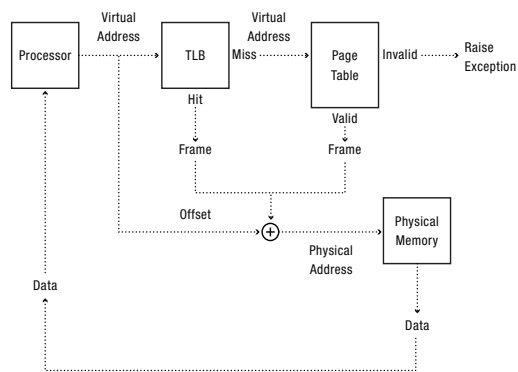
34

## Virtual-To-Physical Lookups

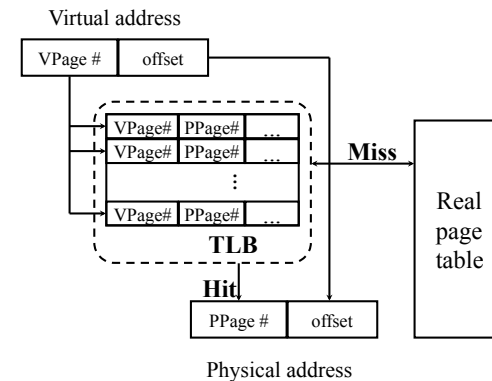
- ◆ Programs only know virtual addresses
  - Each program or process starts from 0 to high address
- ◆ Each virtual address must be translated
  - May involve walking through a hierarchical page table
  - Since the page table is in memory, a program memory access may require several actual memory accesses
- ◆ Solution
  - Cache recent virtual to physical translations, i.e. “active” part of page table, in a very fast memory
  - If virtual address hits in TLB, use cached translation
  - Typically fully associative cache, match against entries

35

## TLB and Page Table Translation



## Translation Look-aside Buffer (TLB)



37

## Bits in a TLB Entry

- ◆ Common (necessary) bits
  - Virtual page number
  - Physical page number: translated address
  - Valid bit
  - Access bits: kernel and user (none, read, write)
- ◆ Optional (useful) bits
  - Process tag
  - Reference bit
  - Modify bit
  - Cacheable bit



38

## Hardware-Controlled TLB

- ◆ On a TLB miss
  - If the page containing the PTE is valid (in memory), hardware loads the PTE into the TLB
    - Write back and replace an entry if there is no free entry
  - Generate a fault if the page containing the PTE is invalid, or if there is a protection fault
  - VM software performs fault handling
  - Restart the CPU
- ◆ On a TLB hit, hardware checks the valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction



39

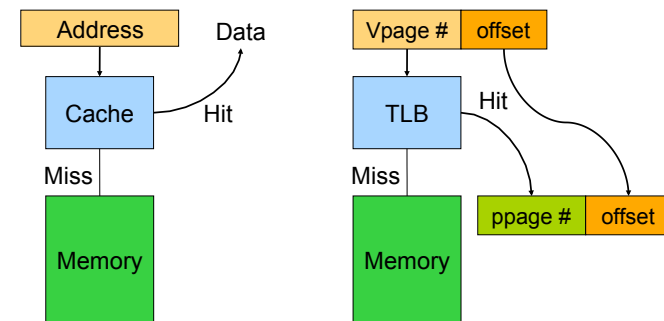
## Software-Controlled TLB

- ◆ On a miss in TLB, software is invoked
  - Write back if there is no free entry
  - Check if the page containing the PTE is in memory
  - If not, perform page fault handling
  - Load the PTE into the TLB
  - Restart the faulting instruction
- ◆ On TLB hit, same as in hardware-controlled TLB



40

## Cache vs. TLB



- ◆ Similarities
  - Cache a portion of memory
  - Write back on a miss

- ◆ Differences
  - Associativity
  - Consistency



42

## TLB Related Issues

- ◆ What TLB entry to be replaced?
  - Random
  - Pseudo LRU
- ◆ What happens on a context switch?
  - Process tag: invalidate appropriate TLB entries
  - No process tag: Invalidate the entire TLB contents
- ◆ What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry



43

## Summary: Virtual Memory

- ◆ Virtual Memory
  - Virtualization makes software development easier and enables memory resource utilization better
  - Separate address spaces provide protection and isolate faults
- ◆ Address Translation
  - Translate every memory operation using table (page table, segment table).
  - Speed: cache frequently used translations
- ◆ Result
  - Every process has a private address space
  - Programs run independently of actual physical memory addresses used, and actual memory size
  - Protection: processes only access memory they are allowed to

