CS162
Operating Systems and Systems Programming
Lecture 12

Protection (continued)
Address Translation

October 11th, 2010

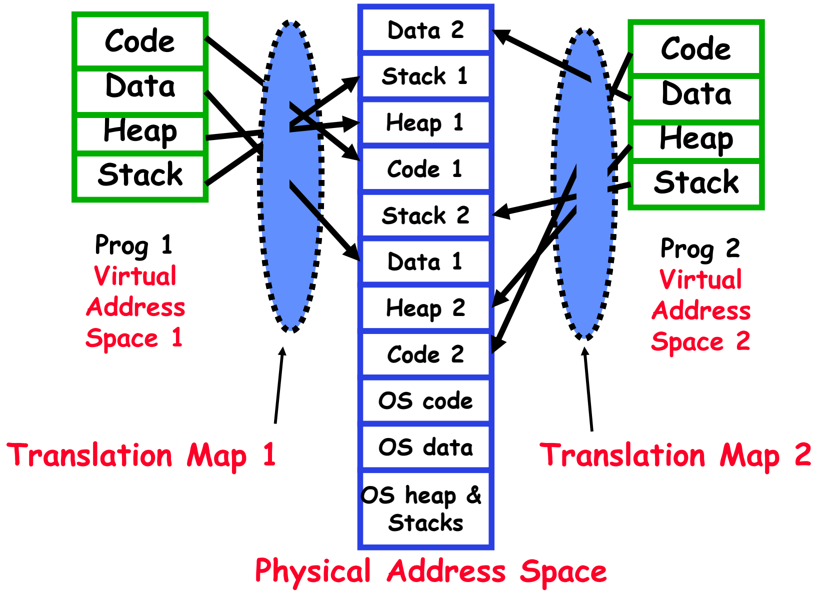
Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

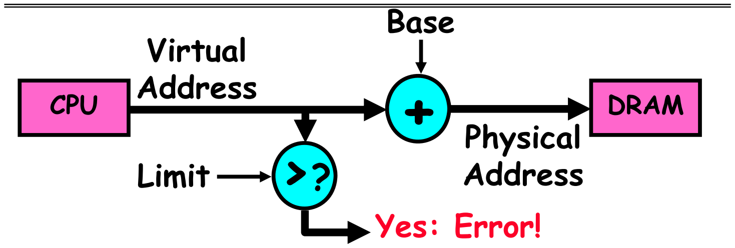
Review: Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - » Can be used to avoid overlap
    - » Can be used to give uniform view of memory to programs
- Protection:
  - Prevent access to private memory of other processes
    - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - » Kernel data protected from User programs
    - » Programs protected from themselves

Review: General Address Translation



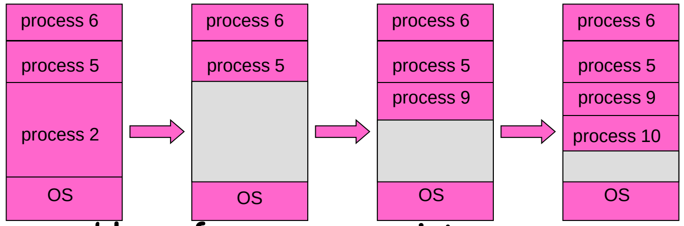
Review: Simple Segmentation: Base and Bounds (CRAY-1)



- Can use base & bounds/limit for dynamic address translation (Simple form of "segmentation"):
  - Alter every address by adding "base"
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Review: Cons for Simple Segmentation Method

- Fragmentation problem (complex memory allocation)
  - Not every process is the same size
  - Over time, memory space becomes fragmented
  - Really bad if want space to grow dynamically (e.g. heap)



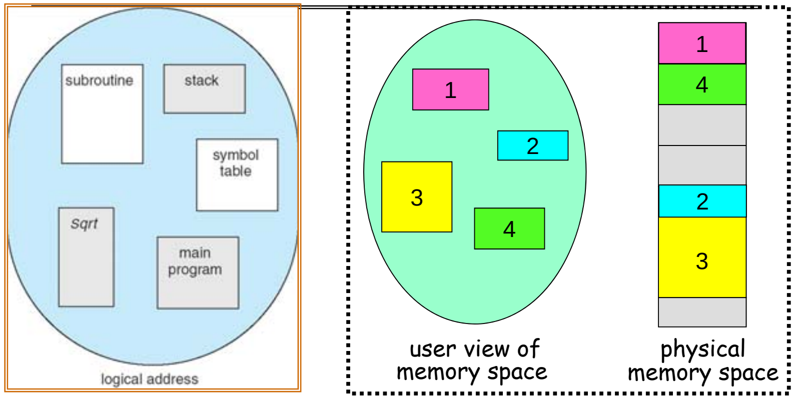
- Other problems for process maintenance
  - Doesn't allow heap and stack to grow independently
  - Want to put these as far apart in virtual memory space as possible so that they can grow as needed
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes

Goals for Today

- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Inverted page tables
- Discussion of Dual-Mode operation
- Comparison among options

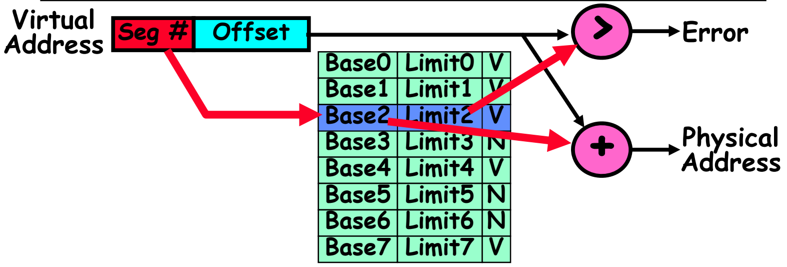
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

More Flexible Segmentation



- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

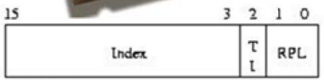
Implementation of Multi-Segment Model



Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: `mov [es:bx],ax.`
- What is "V/N"?
  - Can mark segments as invalid; requires check as well

Intel x86 Special Registers

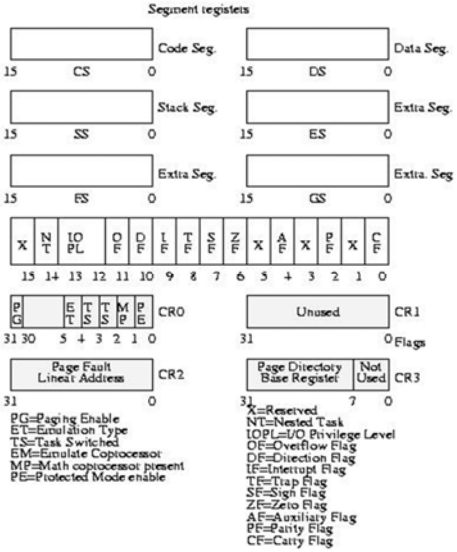


RPL = Requestor Privilege Level
TI = Table Indicator
(0 = GDT, 1 = LDT)
Index = Index into table

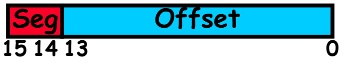
Protected Mode segment selector

Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

80386 Special Registers

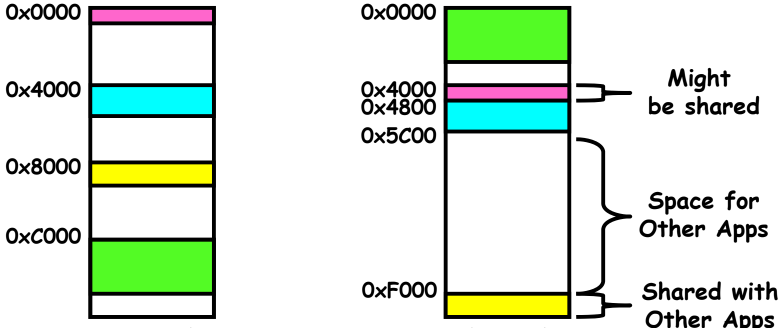


Example: Four Segments (16 bit addresses)



Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Virtual Address Space

Physical Address Space

Example of segment translation

```
0x240   main:    la $a0, varx
0x244            jal strlen
  …               …
0x360   strlen: li   $v0, 0  ;count
0x364   loop:   lb   $t0, ($a0)
0x368           beq  $r0,$t1, done
  …               …
0x4050  varx    dw   0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

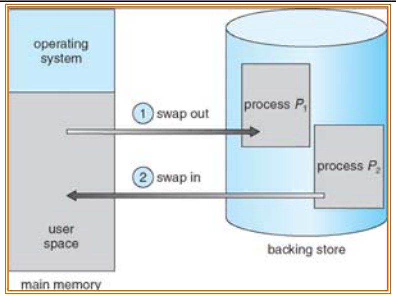
Administrivia

- Midterm I coming up in 1 week:
  - Monday, 10/18, 6:00-9:00pm, 155 Dwinelle
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)
- No class on day of Midterm
  - Extra Office Hours: Mon 2:00-5:00. Perhaps.
- Midterm Topics
  - Topics: Everything up to Wednesday 10/13
  - History, Concurrency, Multithreading, Synchronization, Protection/Address Spaces, TLBs
- Make sure to fill out Group Evaluations!
- Project 2
  - Initial Design Document due Friday 10/15
  - Look at the lecture schedule to keep up with due dates!

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    - » If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called "swapping")

Schematic View of Swapping

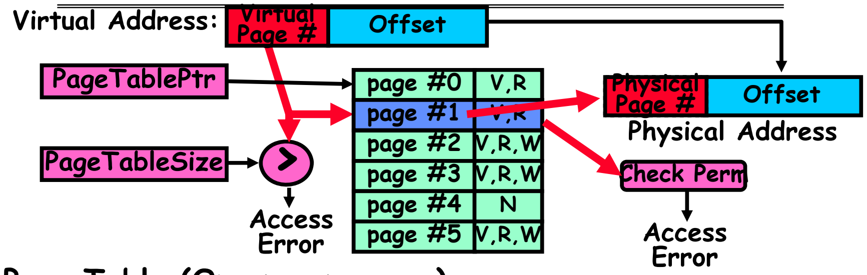


- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

Paging: Physical Memory in Fixed Size Chunks

- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      00110001110001101 … 110010
    - » Each bit represents page of physical memory
      1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

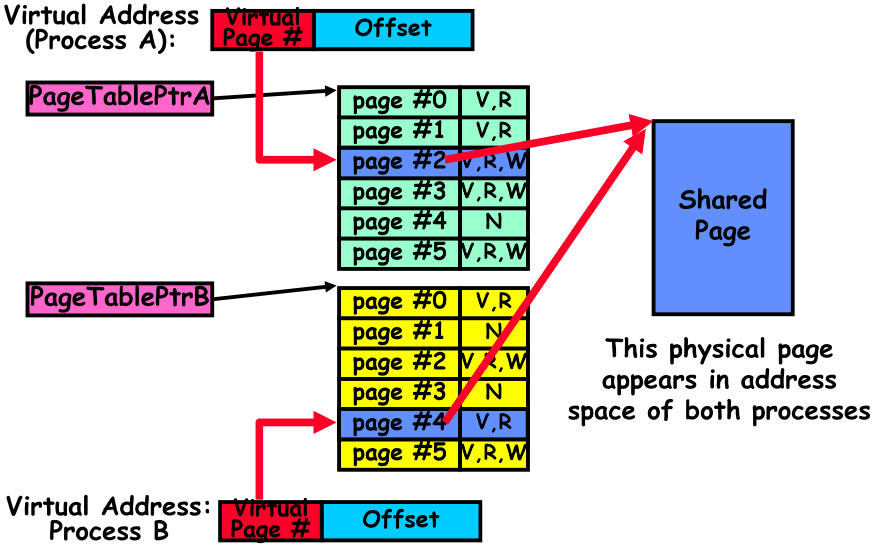
How to Implement Paging?



page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?



page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

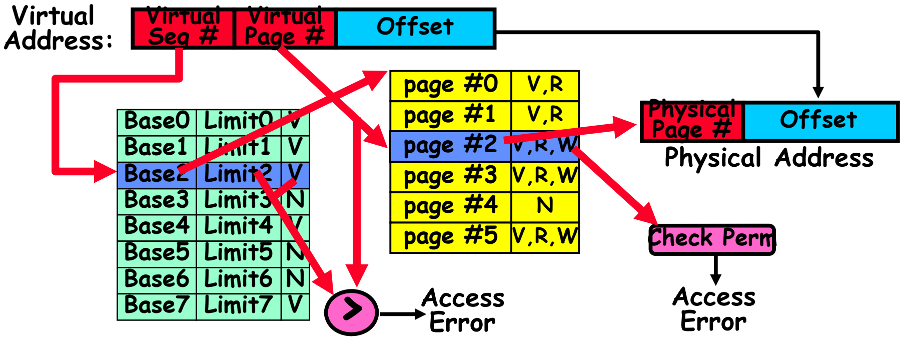
This physical page appears in address space of both processes

Simple Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Simple Page Table Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?
  - Segments with pages inside them?
  - Need some sort of multi-level translation

Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

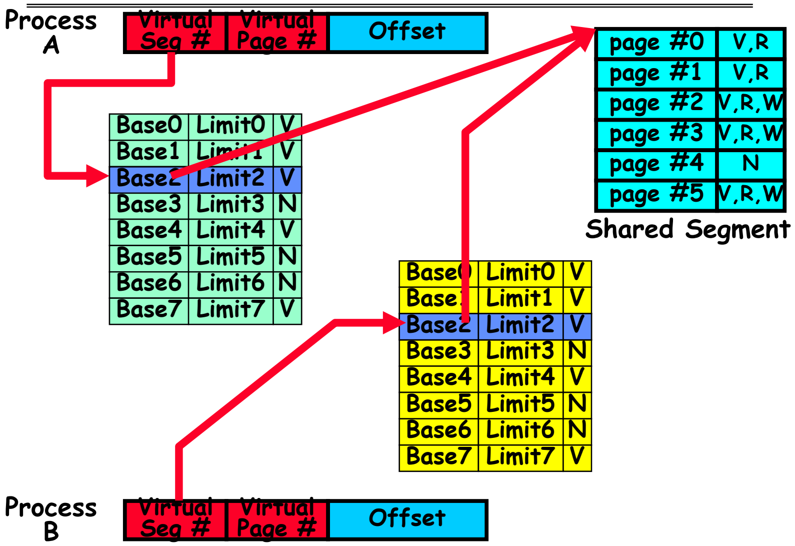


Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

What about Sharing (Complete Segment)?

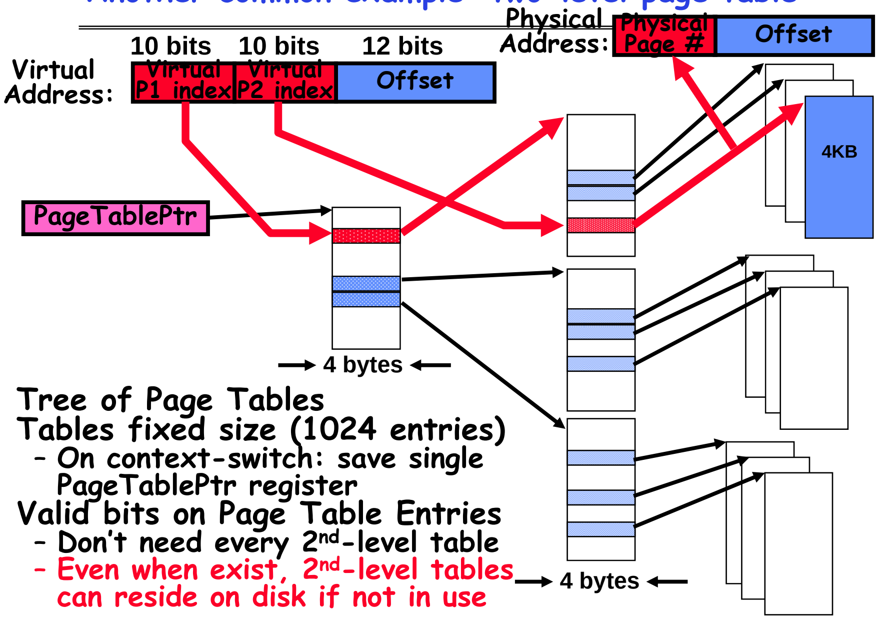


Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

Another common example: two-level page table



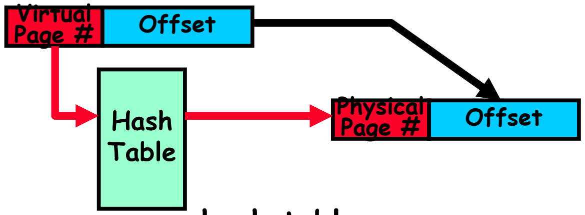
- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

Dual-Mode Operation

- Can Application Modify its own translation tables?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Priviledge Level) from 0 – 3
    - » PL0 has full access, PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel processors only use PL0 ("kernel") and PL3 ("user")

For Protection, Lock User-Programs in Asylum

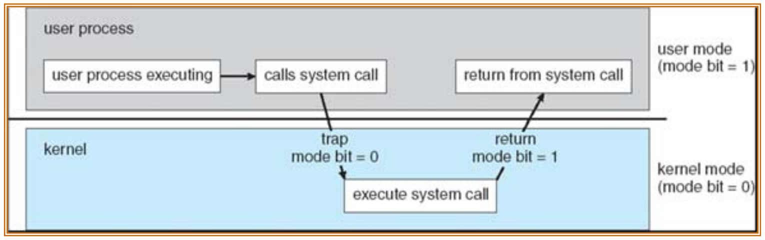
- Idea: Lock user programs in padded cell with no exit or sharp objects
  - Cannot change mode to kernel mode
  - User cannot modify page table mapping
  - Limited access to memory: cannot adversely effect other processes
    - » Side-effect: Limited access to memory-mapped I/O operations (I/O that occurs by reading/writing memory locations)
  - Limited access to interrupt controller
  - What else needs to be protected?
- A couple of issues
  - How to share CPU between kernel and user programs?
    - » Kinda like both the inmates and the warden in asylum are the same person. How do you manage this???
  - How do programs interact?
  - How does one switch between kernel and user modes?
    - » OS → user (kernel → user mode): getting into cell
    - » User→ OS (user → kernel mode): getting out of cell

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - » No! Only specific ones.
  - System call ID encoded into system call instruction
    - » Index forces well-defined interface with kernel

System Call Continued

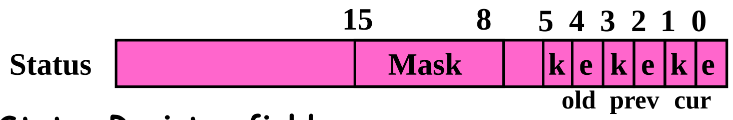
- What are some system calls?
  - I/O: open, close, read, write, lseek
  - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  - Process: fork, exit, wait (like join)
  - Network: socket create, set options
- Are system calls constant across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
    - » On entry to kernel, sets system to kernel mode
    - » Handler address fetched from table/Handler started
- System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel mem
    - » User addresses must be translated!w
    - » Kernel has different view of memory than user
  - Every Argument must be explicitly checked!

User→Kernel (Exceptions: Traps and Interrupts)

- A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

Additions to MIPS ISA to support Exceptions?

- Exception state is kept in "Coprocessor 0"
  - Use mfc0 read contents of these registers:
    - » BadVAddr (register 8): contains memory address at which memory reference error occurred
    - » Status (register 12): interrupt mask and enable bits
    - » Cause (register 13): the cause of the exception
    - » EPC (register 14): address of the affected instruction



- Status Register fields:
  - Mask: Interrupt enable
    - » 1 bit for each of 5 hardware and 3 software interrupts
  - k = kernel/user: 0⇒kernel mode
  - e = interrupt enable: 0⇒interrupts disabled
  - Exception⇒6 LSB shifted left 2 bits, setting 2 LSB to 0:
    - » run in kernel mode with interrupts disabled

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

Summary (1/2)

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Dual-Mode
  - Kernel/User distinction: User restricted
  - User→Kernel: System calls, Traps, or Interrupts
  - Inter-process communication: shared memory, or through kernel (system calls)
- Exceptions
  - Synchronous Exceptions: Traps (including system calls)
  - Asynchronous Exceptions: Interrupts

Summary (2/2)

- Segment Mapping
  - Segment registers within processor
  - Segment ID associated with each access
    - » Often comes from portion of virtual address
    - » Can come from bits in instruction instead (x86)
  - Each segment contains base and limit information
    - » Offset (rest of address) adjusted by adding base
- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table
  - Size of page table related to physical memory size