

CS162 Operating Systems and Systems Programming Lecture 12

Address Translation

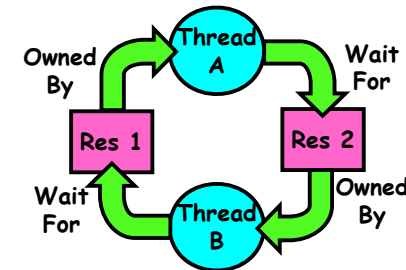
October 6th, 2015
Prof. John Kubiatowicz
<http://cs162.eecs.Berkeley.edu>

Recall: Starvation vs Deadlock



• Starvation vs. Deadlock

- Starvation: thread waits indefinitely
 - » Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
 - » Thread A owns Res 1 and is waiting for Res 2
 - » Thread B owns Res 2 and is waiting for Res 1



- Deadlock \Rightarrow Starvation but not vice versa
 - » Starvation can end (but doesn't have to)
 - » Deadlock can't end without external intervention

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.2

Recall: Four requirements for Deadlock

- **Mutual exclusion**
 - Only one thread at a time can use a resource.
- **Hold and wait**
 - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
 - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
 - There exists a set $\{T_1, \dots, T_n\}$ of waiting threads
 - » T_1 is waiting for a resource that is held by T_2
 - » T_2 is waiting for a resource that is held by T_3
 - » ...
 - » T_n is waiting for a resource that is held by T_1

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.3

Can Priority-Inversion cause deadlock?

- Technically not - Consider this example:
 - 3 threads, T_1 , T_2 , T_3 in priority order (T_3 highest)
 - T_1 grabs lock, T_3 tries to acquire, then sleeps, T_2 running
 - Will this make progress?
 - » No, as long as T_2 is running
 - » But T_2 could stop at any time and the problem would resolve itself... So, this is *not* a deadlock (it is a livelock)
 - Why is this a priority inversion?
 - » T_3 is prevented from running by T_2
- How does *priority donation* help?
 - Briefly raising T_1 to the same priority as $T_3 \Rightarrow T_1$ can run and release lock, allowing T_3 to run
 - Does priority donation involve taking lock away from T_1 ?
 - » NO! That would break semantics of the lock!

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.4

Recall: Important Aspects of Memory Multiplexing

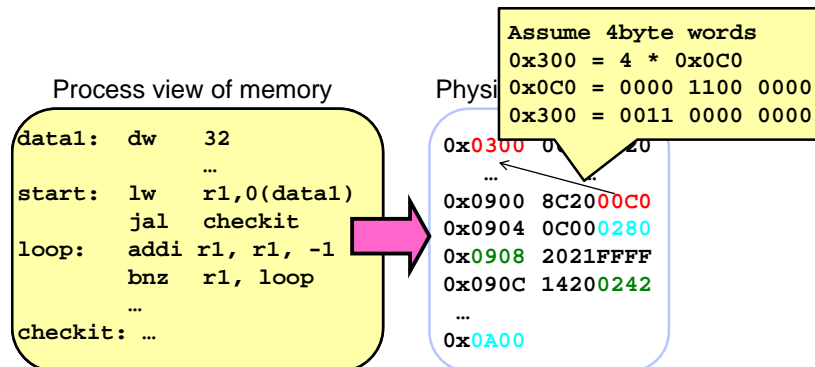
- **Controlled overlap:**
 - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
 - Conversely, would like the ability to overlap when desired (for communication)
- **Translation:**
 - Ability to translate accesses from one address space (virtual) 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.5

Binding of Instructions and Data to Memory

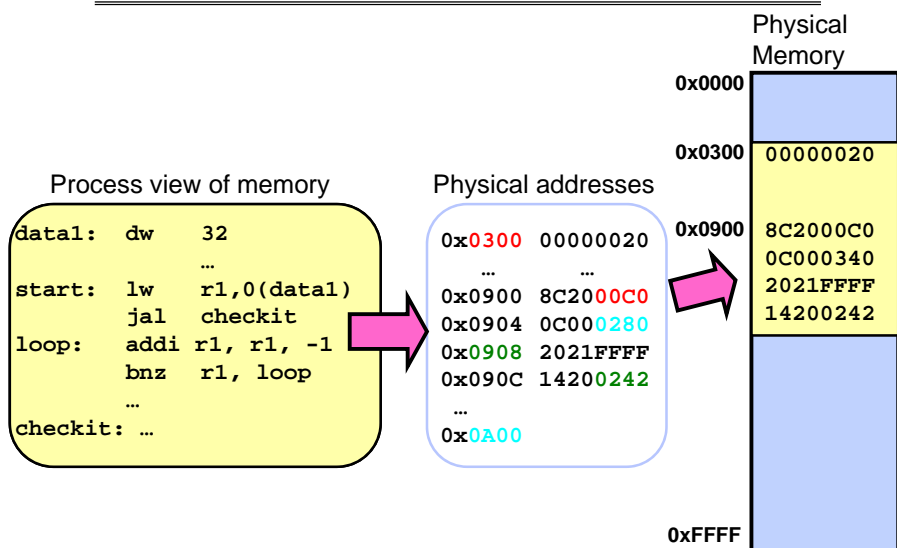


10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.6

Binding of Instructions and Data to Memory

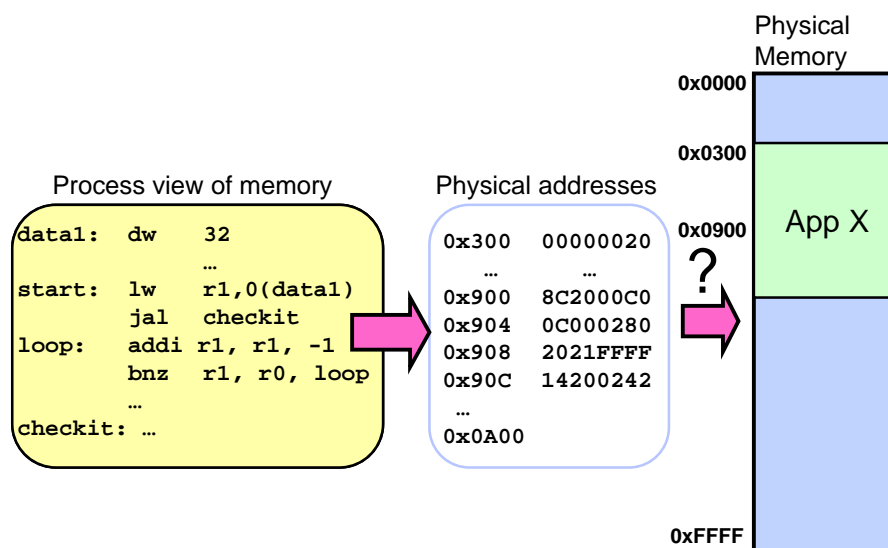


10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.7

Second copy of program from previous example



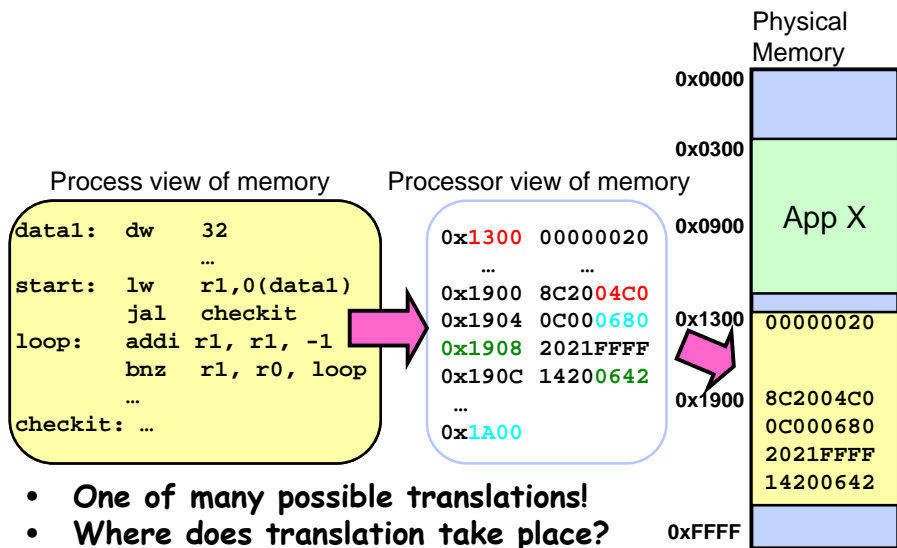
Need address translation!

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.8

Second copy of program from previous example



- One of many possible translations!
- Where does translation take place?
Compile time, Link/Load time, or Execution time?

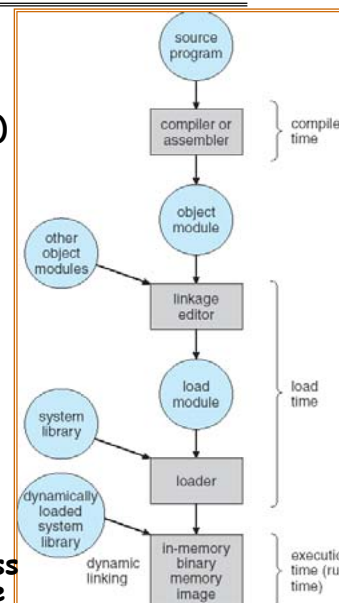
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.9

Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
 - Compile time (i.e., "gcc")
 - Link/Load time (UNIX "ld" does link)
 - Execution time (e.g., dynamic libs)
- Addresses can be bound to final values anywhere in this path
 - Depends on hardware support
 - Also depends on operating system
- Dynamic Libraries
 - Linking postponed until execution
 - Small piece of code, *stub*, used to locate appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine, and executes routine



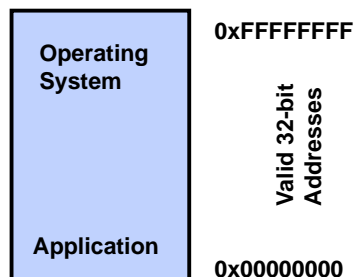
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.10

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address



- Application given illusion of dedicated machine by giving it reality of a dedicated machine

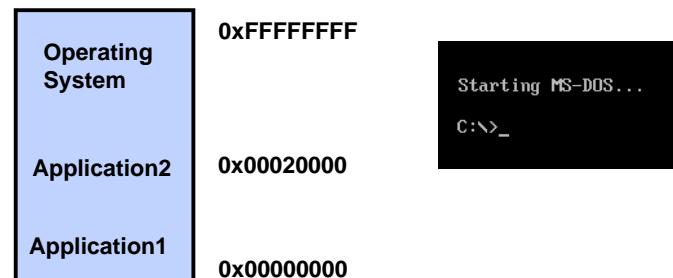
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.11

Multiprogramming (primitive stage)

- Multiprogramming without Translation or Protection
 - Must somehow prevent address overlap between threads



- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - » Everything adjusted to memory location of program
 - » Translation done by a linker-loader (relocation)
 - » Common in early days (... till Windows 3.x, 95?)
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

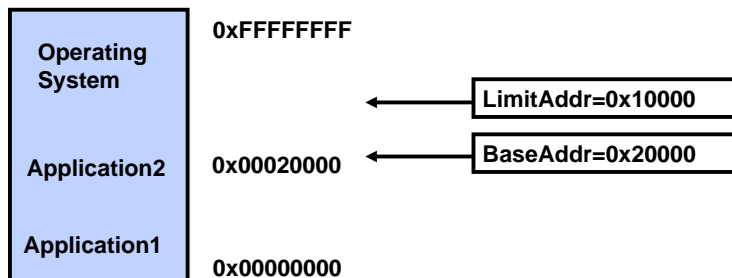
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.12

Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



- Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
 - » If user tries to access an illegal address, cause an error
- During switch, kernel loads new base/limit from PCB (Process Control Block)
 - » User not allowed to change base/limit registers

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.13

Administrivia

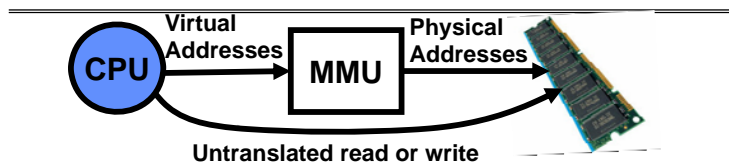
- Midterm coming up in 1 week!
 - Wednesday 10/14, 6:30-9:30pm
 - In 145/155 Dwinelle
 - No class that day, extra office hours
- Details
 - Intend this to be a 2-hour exam in 3 hour slot
 - 1 page of hand-written notes, both sides
 - Closed book
 - Topics will include the material from that Monday
- Review Session
 - This Sunday, 2-4 PM, 306 Soda Hall

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.14

Recall: Address translation



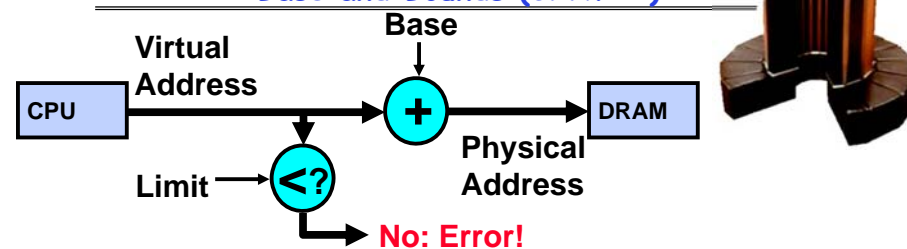
- Address Space:
 - All the addresses and state a process can touch
 - Each process and kernel has different address space
- Consequently, two views of memory:
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
 - Translation box (MMU) converts between the two views
- Translation essential to implementing protection
 - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.15

Recall Simple Example: Base and Bounds (CRAY-1)



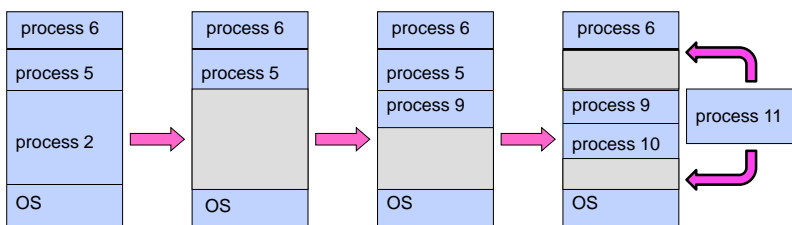
- Could use base/limit for **dynamic address translation** - translation happens at execution:
 - Alter address of every load/store by adding "base"
 - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.16

Issues with Simple B&B Method



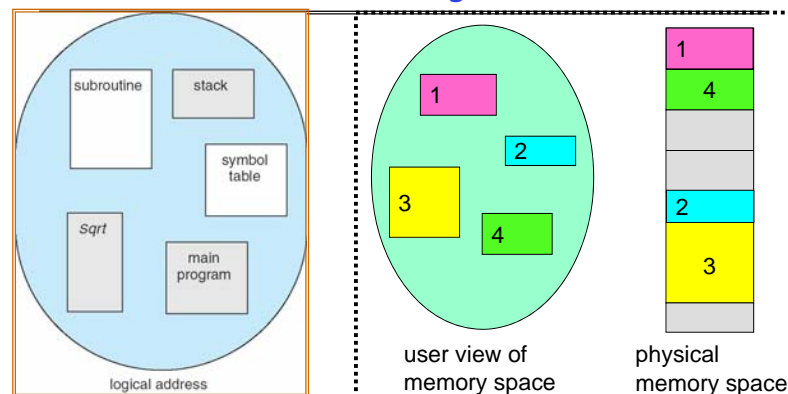
- **Fragmentation problem**
 - Not every process is the same size
 - Over time, memory space becomes fragmented
- **Missing support for sparse address space**
 - Would like to have multiple chunks/program
 - E.g.: Code, Data, Stack
- **Hard to do inter-process sharing**
 - Want to share code segments when possible
 - Want to share memory between processes
 - Helped by providing multiple segments per process

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.17

More Flexible Segmentation



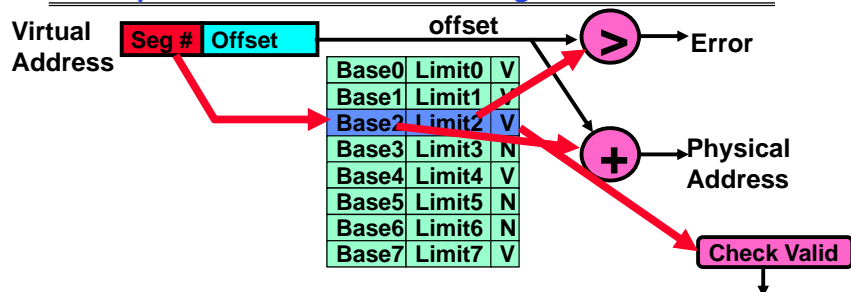
- **Logical View: multiple separate segments**
 - Typical: Code, Data, Stack
 - Others: memory sharing, etc
- **Each segment is given region of contiguous memory**
 - Has a base and limit
 - Can reside anywhere in physical memory

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.18

Implementation of Multi-Segment Model



- **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" (valid / not valid)?**
 - Can mark segments as invalid; requires check as well

10/6/15

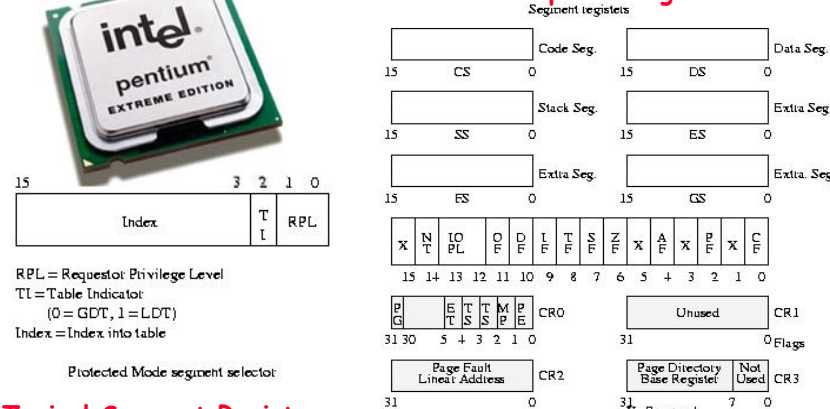
Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.19

Intel x86 Special Registers



80386 Special Registers



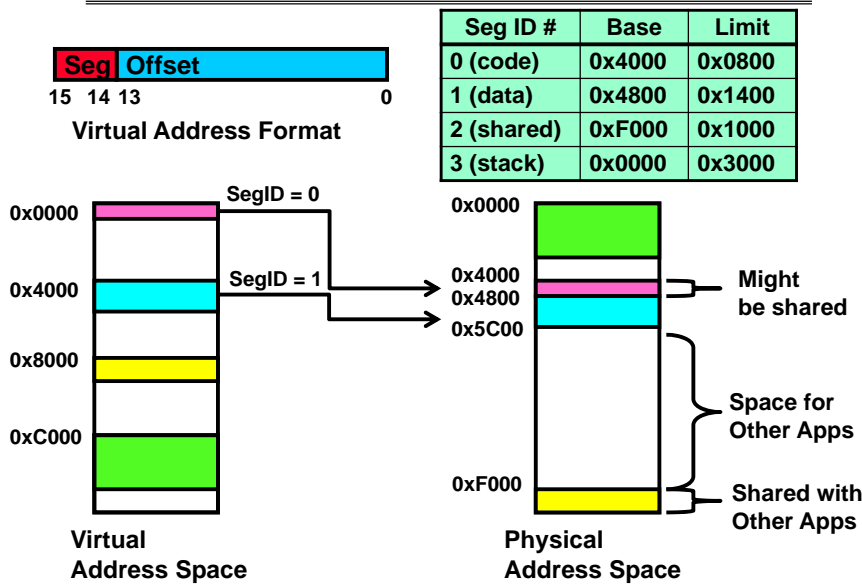
Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.20

Example: Four Segments (16 bit addresses)



10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.21

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):

- 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
- Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
- Fetch 0x360. Translated to Physical=0x4360. Get "li \$v0, 0"
Move 0x0000 → \$v0, Move PC+4 → PC
- 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.22

Observations about Segmentation

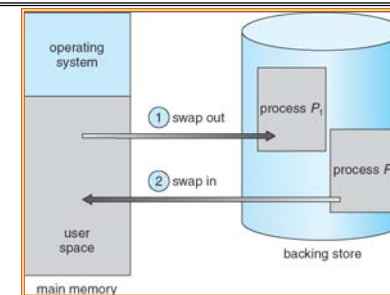
- Virtual address space has holes
 - Segmentation efficient for sparse address spaces
 - A correct program should never address gaps (except as mentioned in moment)
 - » If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
 - This is how the stack and heap are allowed 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")

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.23

What if more segments than will fit into memory?



- 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.24

Problems with Segmentation

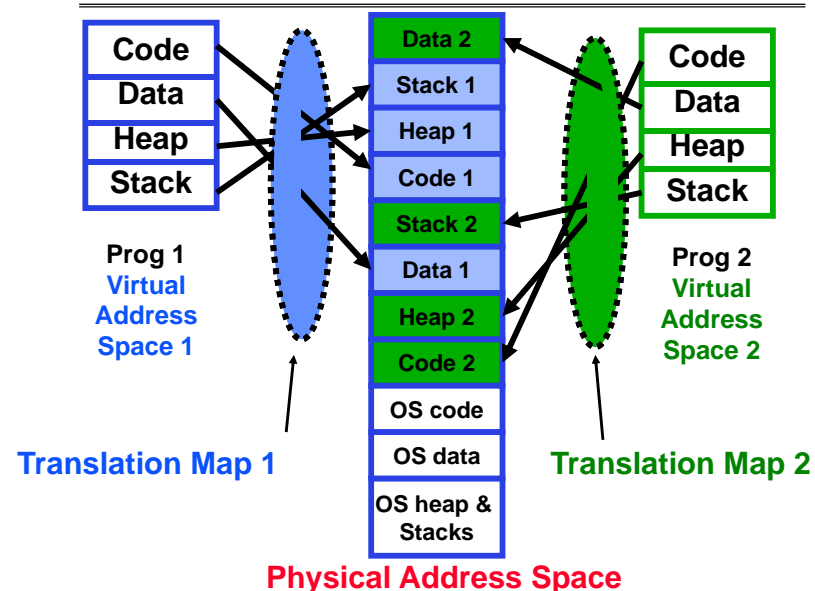
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- **Fragmentation**: wasted space
 - **External**: free gaps between allocated chunks
 - **Internal**: don't need all memory within allocated chunks

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.25

Recall: General Address Translation



10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.26

Paging: Physical Memory in Fixed Size Chunks

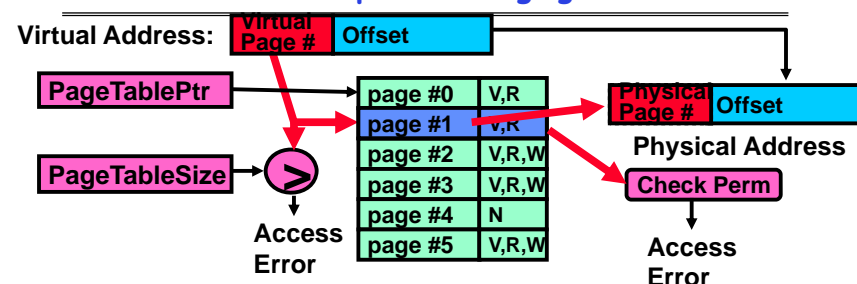
- **Solution to fragmentation from segments?**
 - Allocate physical memory in fixed size chunks ("pages")
 - Every chunk of physical memory is equivalent
 - » Can use simple vector of bits to handle allocation: 00110001110001101 ... 110010
 - » Each bit represents page of physical memory
1⇒allocated, 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.27

How to Implement Paging?



- **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

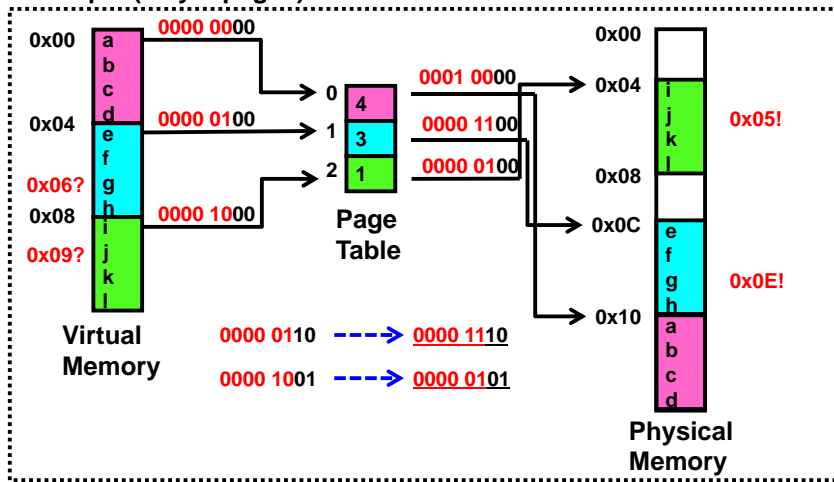
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.28

Simple Page Table Example

Example (4 byte pages)

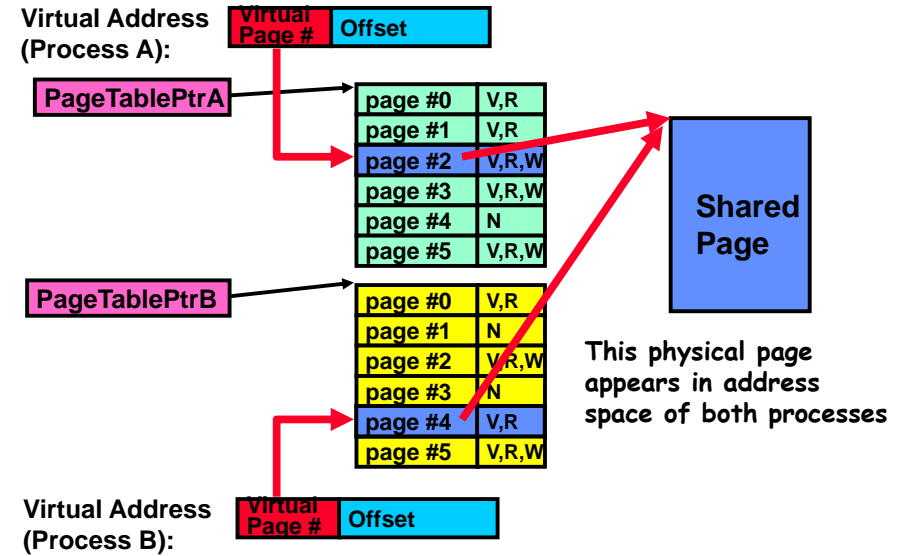


10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.29

What about Sharing?

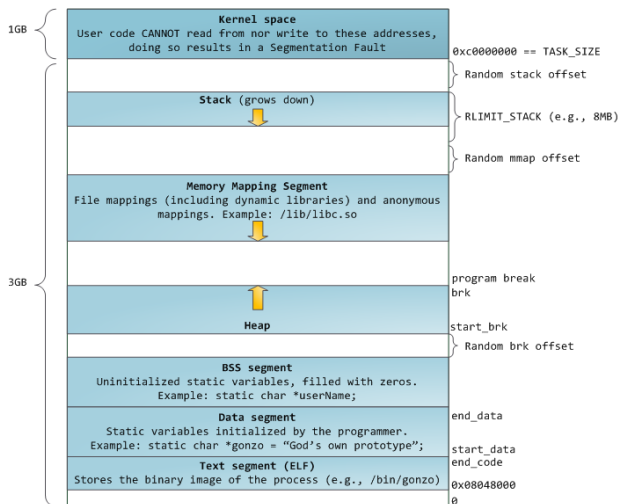


10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.30

Memory Layout for Linux 32-bit



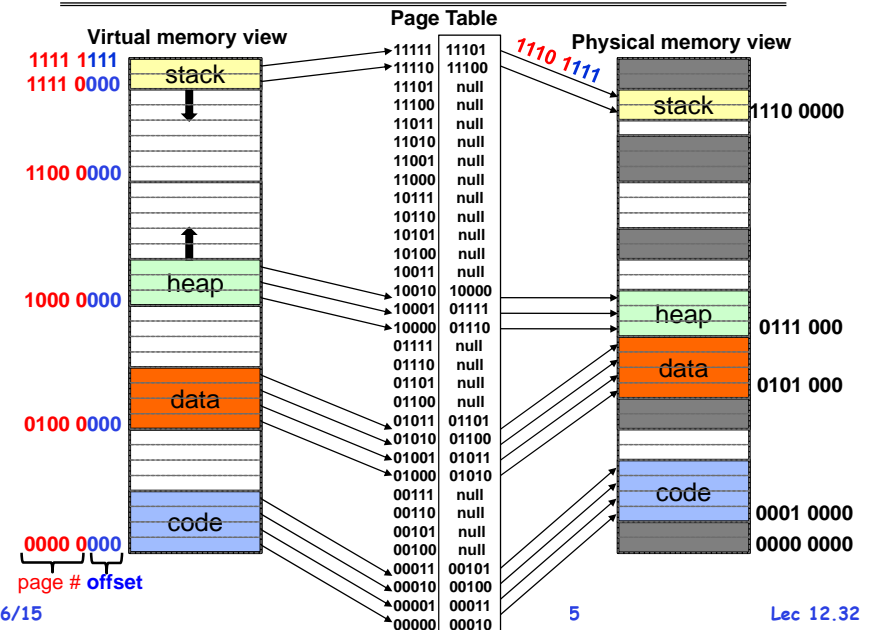
<http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png>

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.31

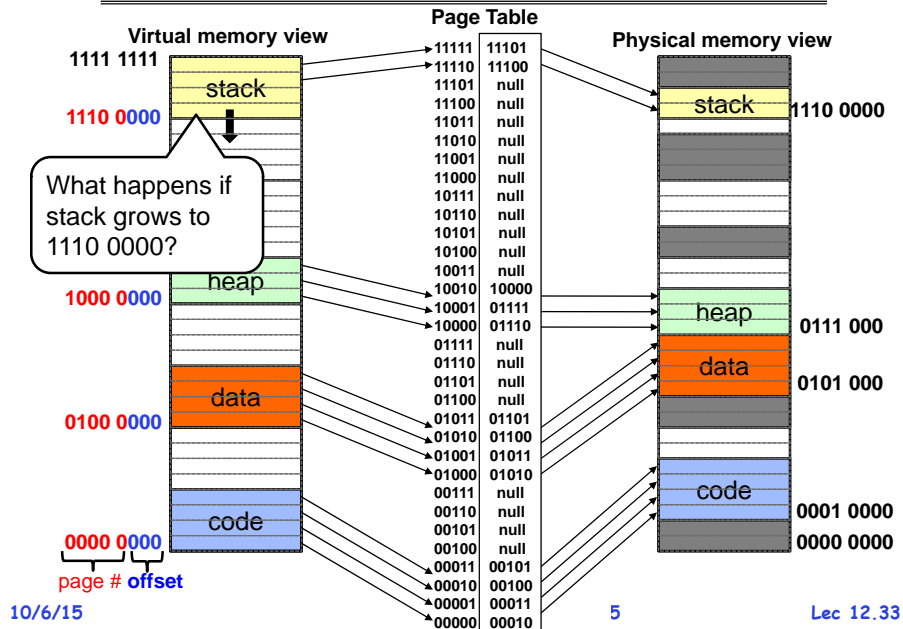
Summary: Simple Page Table



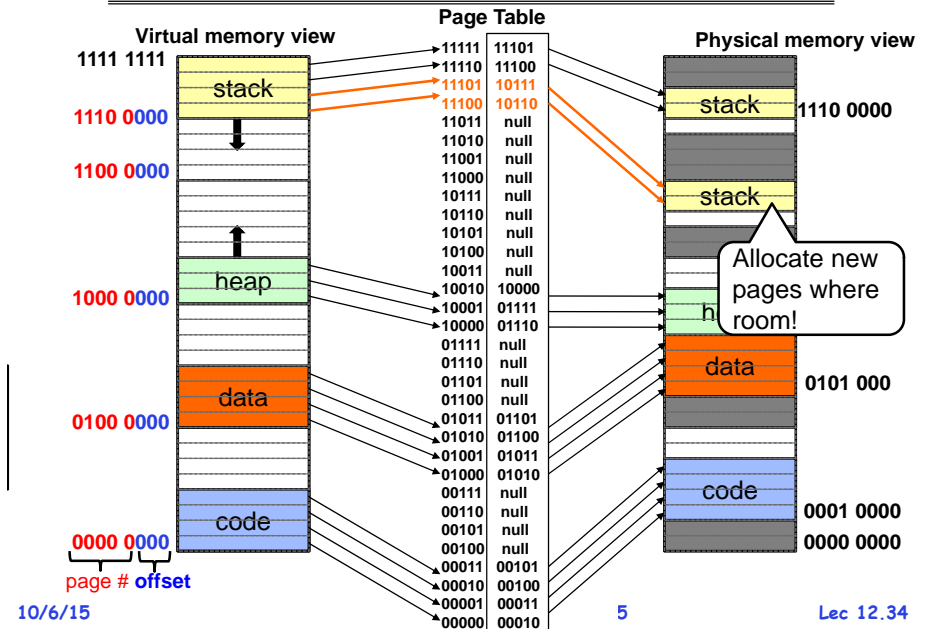
10/6/15

Lec 12.32

Summary: Simple Page Table



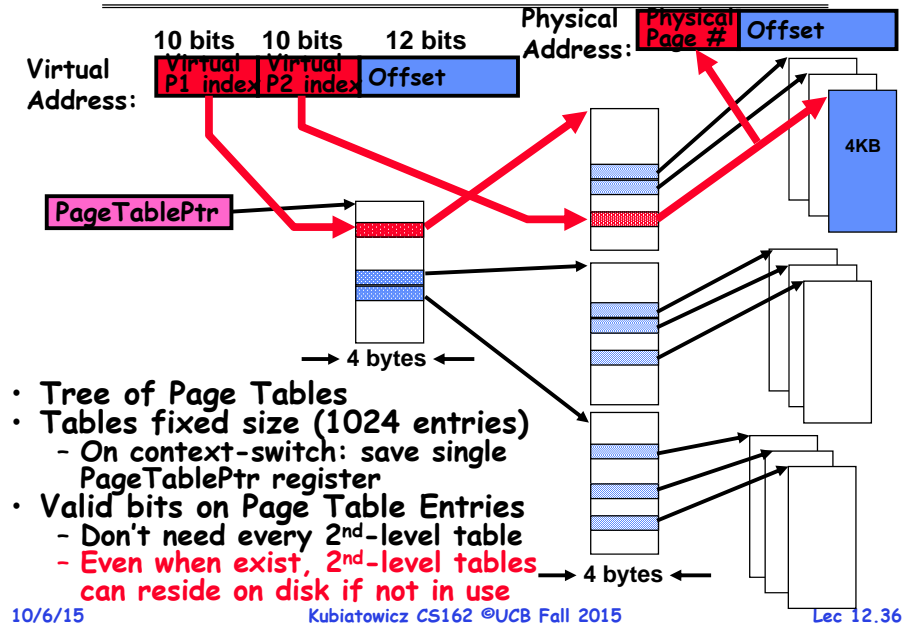
Summary: Simple Page Table



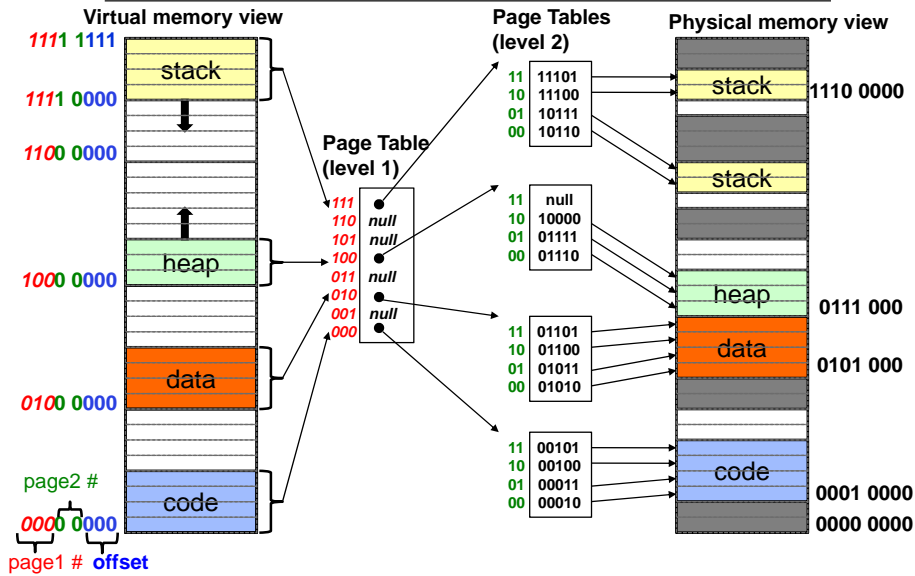
Page Table Discussion

- What needs to be switched on a context switch?
 - Page table pointer and limit
- 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 (231-1).
 - » With 1K pages, need 2 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 can we handle sparseness?
 - Multi-level page table
- How about combining paging and segmentation?
 - Segments with pages inside them?
 - Also need some sort of multi-level translation

Fix for sparse address space: The two-level page table



Summary: Two-Level Paging

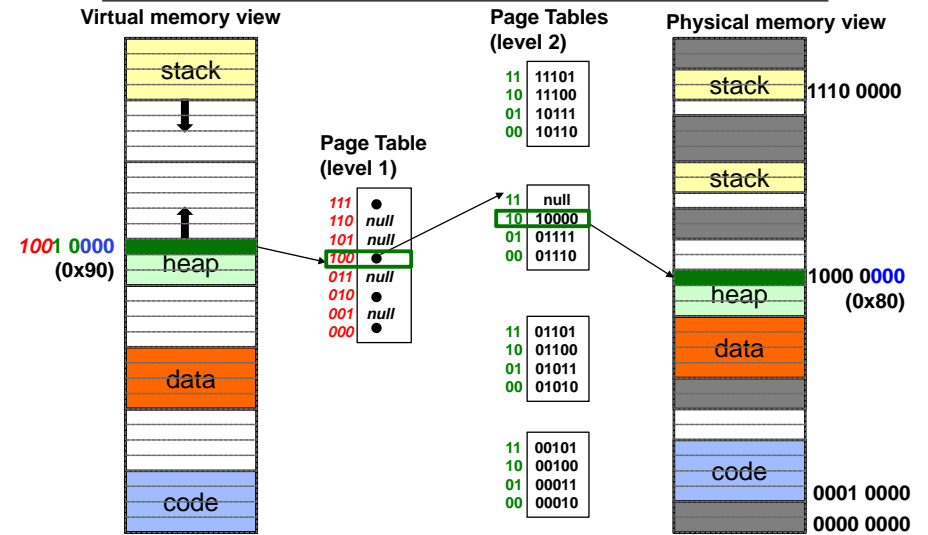


10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.37

Summary: Two-Level Paging



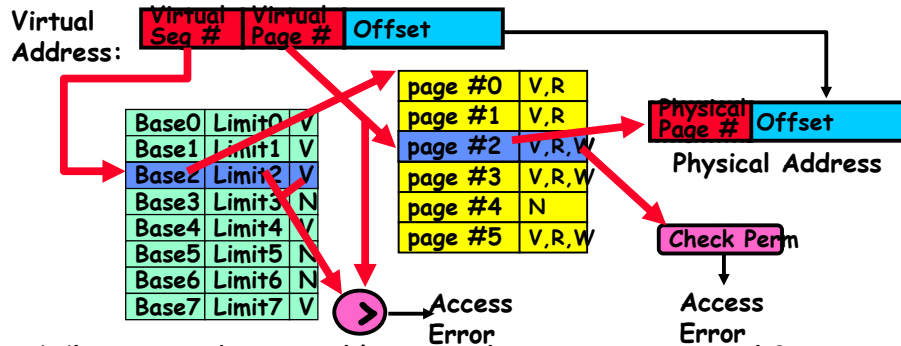
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.38

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):



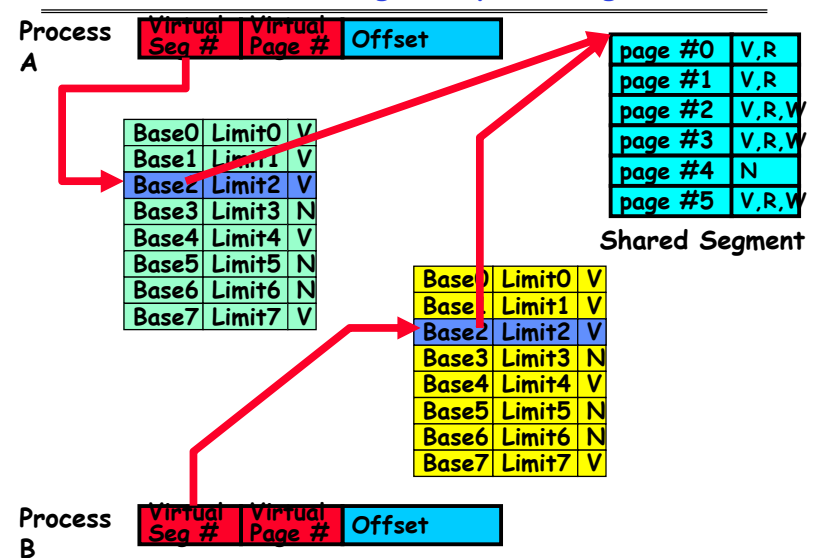
- What must be saved/restored on context switch?
 - Contents of top-level segment registers (for this example)
 - Pointer to top-level table (page table)

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.39

What about Sharing (Complete Segment)?



10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.40

Multi-level Translation Analysis

- **Pros:**
 - Only need to allocate as many page table entries as we need for application
 - » In other words, 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!

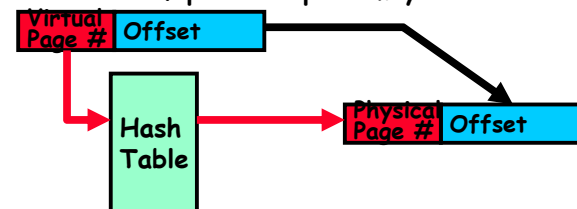
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.41

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



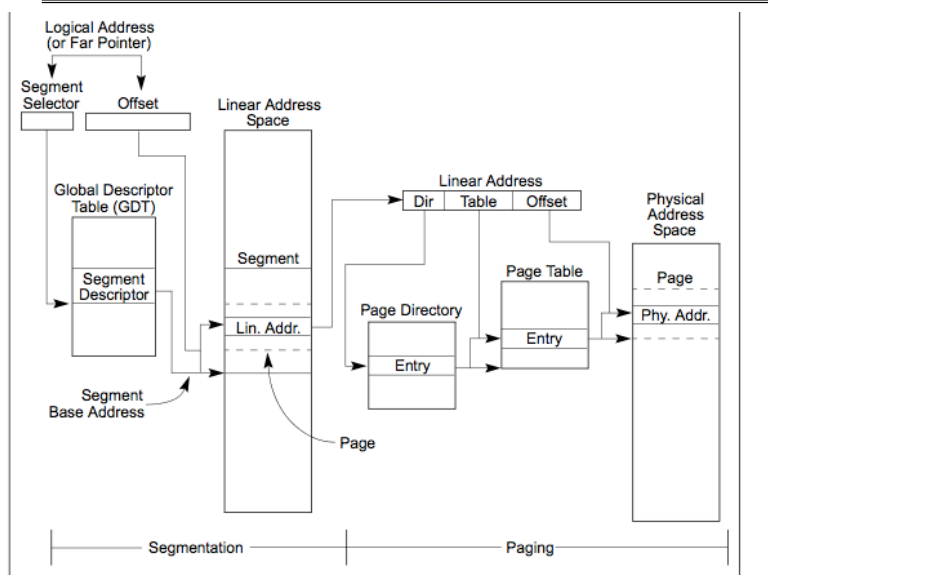
- **Answer: use a hash table**
 - Called an "Inverted Page Table"
 - Size is independent of virtual address space
 - Directly related to amount of physical memory
 - Very attractive option for 64-bit address spaces
- **Cons: Complexity of managing hash changes**
 - Often in hardware!

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.42

Making it real: X86 Memory model with segmentation (16/32-bit)



10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

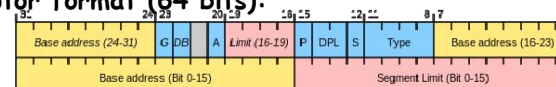
Lec 12.43

X86 Segment Descriptors (32-bit Protected Mode)

- **Segments** are either implicit in the instruction (say for code segments) or actually part of the instruction
 - There are 6 registers: SS, CS, DS, ES, FS, GS
- What is in a segment register?
 - A *pointer* to the actual segment description:



- G/L selects between GDT and LDT tables (global vs local descriptor tables)
- Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
 - Includes length of table (for 2^{13} entries)
- Descriptor format (64 bits):



- G: Granularity of segment (0: 16bit, 1: 4KiB unit)
- DB: Default operand size (0: 16bit, 1: 32bit)
- A: Freely available for use by software
- P: Segment present
- DPL: Descriptor Privilege Level
- S: System Segment (0: System, 1: code or data)
- Type: Code, Data, Segment

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.44

What is in a Page Table Entry?

- What is in a Page Table Entry (or PTE)?
 - Pointer to next-level page table or to actual page
 - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
 - Address same format previous slide (10, 10, 12-bit offset)
 - Intermediate page tables called "Directories"

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

- P: Present (same as "valid" bit in other architectures)
- W: Writeable
- U: User accessible
- PWT: Page write transparent: external cache write-through
- PCD: Page cache disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty (PTE only): page has been modified recently
- L: L=1 ⇒ 4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.45

Examples of how to use a PTE

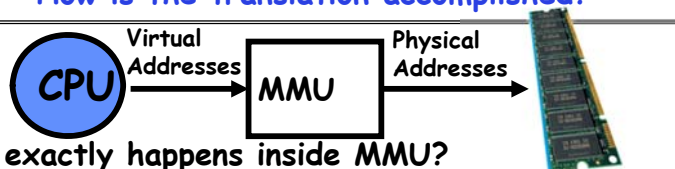
- How do we use the PTE?
 - Invalid PTE can imply different things:
 - » Region of address space is actually invalid or
 - » Page/directory is just somewhere else than memory
 - Validity checked first
 - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
 - Keep only active pages in memory
 - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
 - UNIX fork gives *copy* of parent address space to child
 - » Address spaces disconnected after child created
 - How to do this cheaply?
 - » Make copy of parent's page tables (point at same memory)
 - » Mark entries in both sets of page tables as read-only
 - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
 - New data pages must carry no information (say be zeroed)
 - Mark PTEs as invalid; page fault on use gets zeroed page
 - Often, OS creates zeroed pages in background

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.46

How is the translation accomplished?



- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
 - For each virtual address, takes page table base pointer and traverses the page table in hardware
 - Generates a "Page Fault" if it encounters invalid PTE
 - » Fault handler will decide what to do
 - » More on this next lecture
 - Pros: Relatively fast (but still many memory accesses!)
 - Cons: Inflexible, Complex hardware
- Another possibility: Software
 - Each traversal done in software
 - Pros: Very flexible
 - Cons: Every translation must invoke Fault!
- **In fact, need way to *cache* translations for either case!**

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.47

Recall: Dual-Mode Operation

- Can a process modify its own translation tables?
 - **NO!**
 - 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 (Privilege Level) from 0 - 3
 - » PLO 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 PLO ("kernel") and PL3 ("user")

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.48

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)

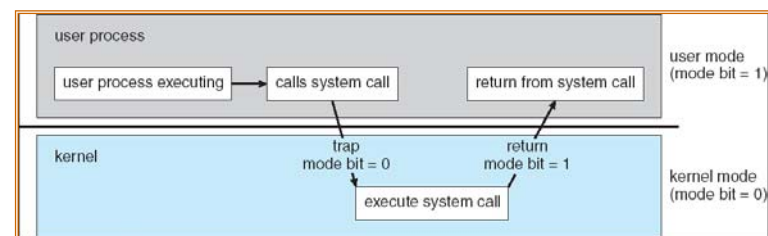
10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.49

Recall: 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**

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.50

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!
 - » Kernel has different view of memory than user
 - Every Argument must be explicitly checked!

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.51

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 ("Trap")**:
 - 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.52

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)

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.53

Summary (1/2)

- 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.54

Summary (2/2)

- PTE: Page Table Entries
 - Includes physical page number
 - Control info (valid bit, writeable, dirty, user, etc)
- 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

10/6/15

Kubiatowicz CS162 ©UCB Fall 2015

Lec 12.55