CS162
Operating Systems and Systems Programming
Lecture 12

Address Translation

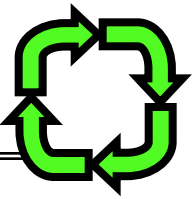
October 7th, 2019

Prof. David E. Culler

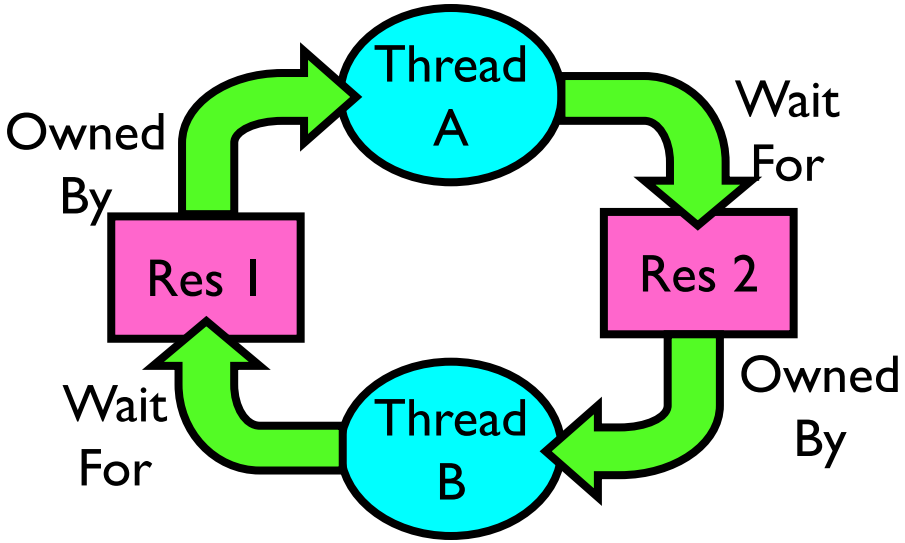
http://cs162.eecs.Berkeley.edu

Read: A&D Ch 8
MidTerm: Thurs

Recall: Starvation and 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 ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

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 $\{T_1, \ldots, T_n\}$ 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

- To prevent deadlock, make sure at least one of these conditions does not hold

Recall: How should a system deal with deadlock?

- Three different approaches:

1. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
2. Deadlock prevention: write your code in a way that it isn't prone to deadlock
3. Deadlock recovery: let deadlock happen, and then figure out how to recover from it

- Modern operating systems:
  - Make sure the *system* isn't involved in any deadlock
  - Ignore deadlock in applications
    - » "Ostrich Algorithm"

Recall: Ways of preventing deadlock

- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Example (x.P, y.P, z.P,…)
    - » Make tasks request disk, then memory, then…
- Banker's algorithm:
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, and grant request if result is deadlock free (conservative!)
  - Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Can Priority Inversion cause Deadlock?

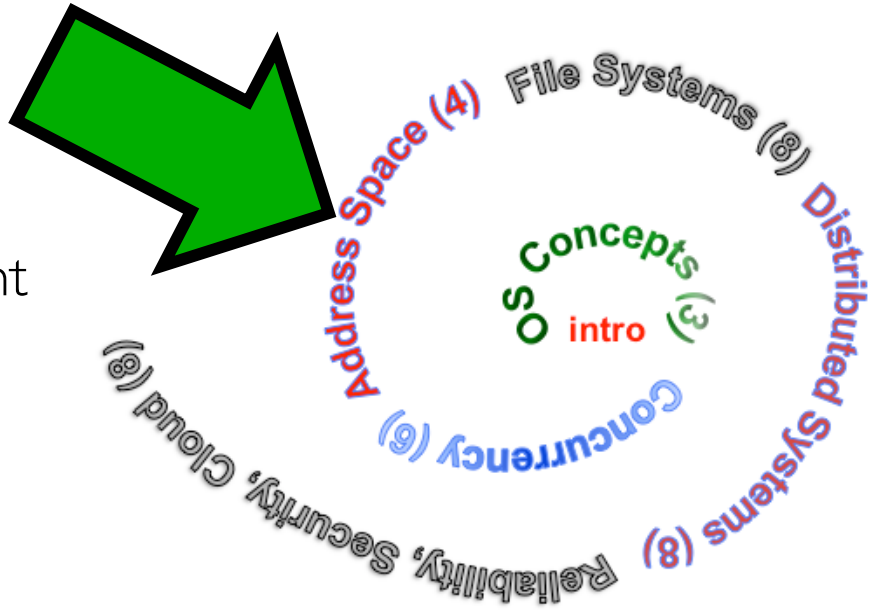
- Technically not – Consider this example:
  - 3 threads, T1, T2, T3 in priority order (T3 highest)
  - T1 grabs lock, T3 tries to acquire, then sleeps, T2 running
  - Will this make progress?
    - » No, as long as T2 is running
    - » But T2 could stop at any time and the problem would resolve itself… So, this is *not* a deadlock (it is a livelock)
  - Why is this a priority inversion?
    - » T3 is prevented from running by T2
- How does *priority donation* help?
  - Briefly raising T1 to the same priority as T3⇒T1 can run and release lock, allowing T3 to run
  - Does priority donation involve taking lock away from T1?
    - » NO! That would break semantics of the lock and potentially corrupt any information protected by lock!

Logistics and Such

- We have completed the first major segment of the course
- The Midterm will cover through last week
  - The basics of address translation of Lec. 2, not the new material in this lecture
- Homeworks 3 & 4, Project 2 will continue to exercise and cement what we have covered so far.
- Building on these foundations, we can now deepen and broaden

Next Objective

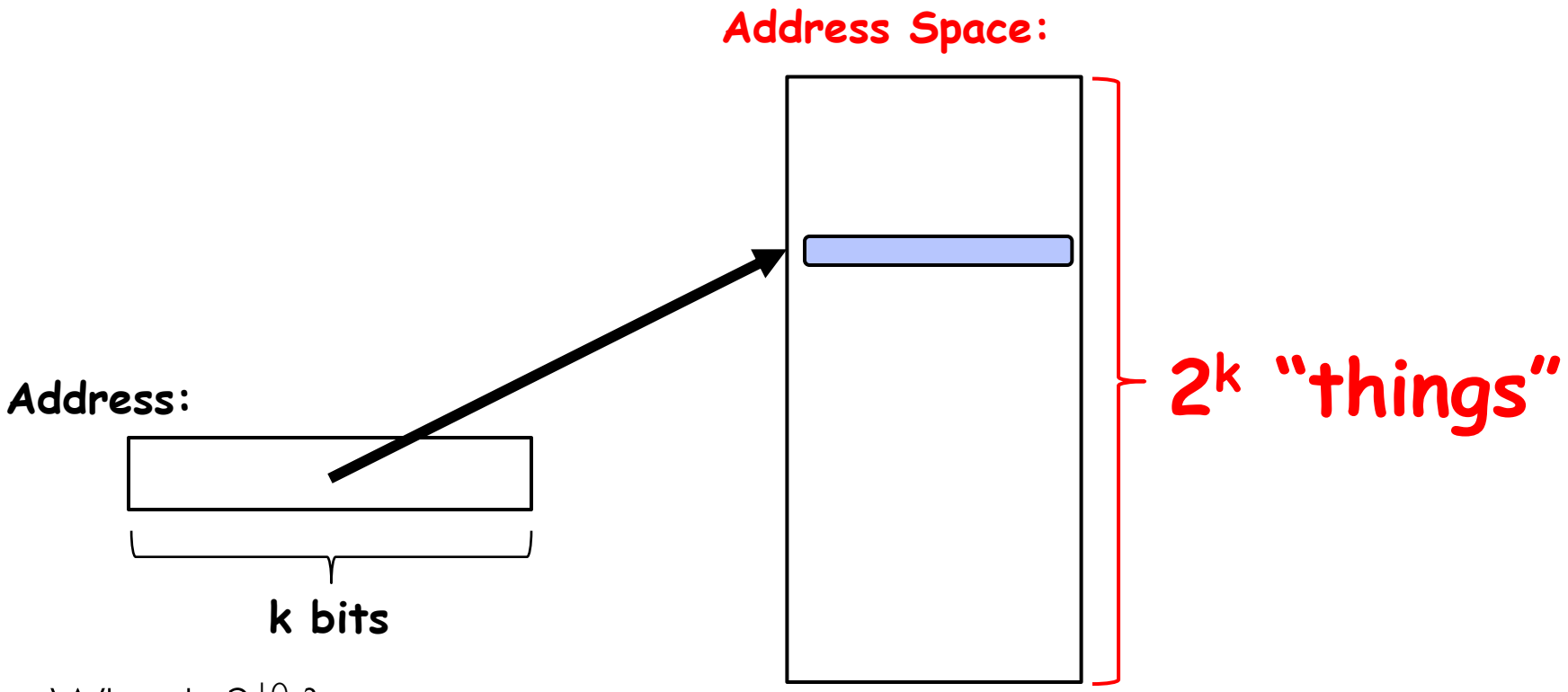
- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
  - Demand paging
- Today: Translation



Recall: Four Fundamental OS Concepts

- Thread: Execution Context
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with translation)
  - Program's view of memory is distinct from physical machine
- Process: an instance of a running program
  - Address Space + One or more Threads + …
- Dual mode operation / Protection
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

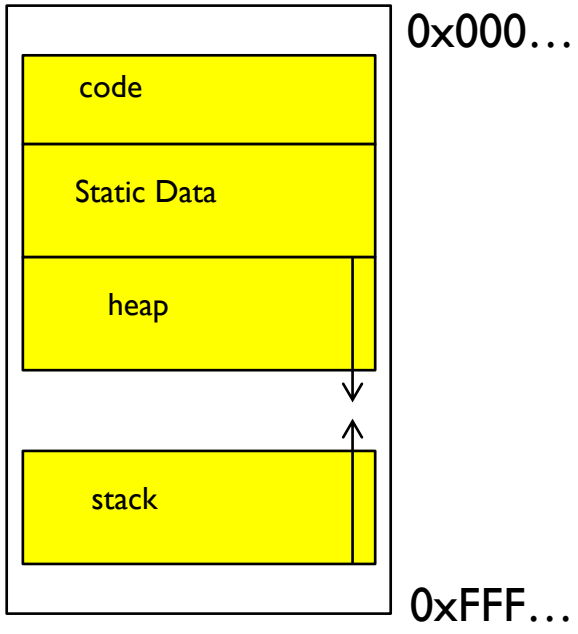
THE BASICS



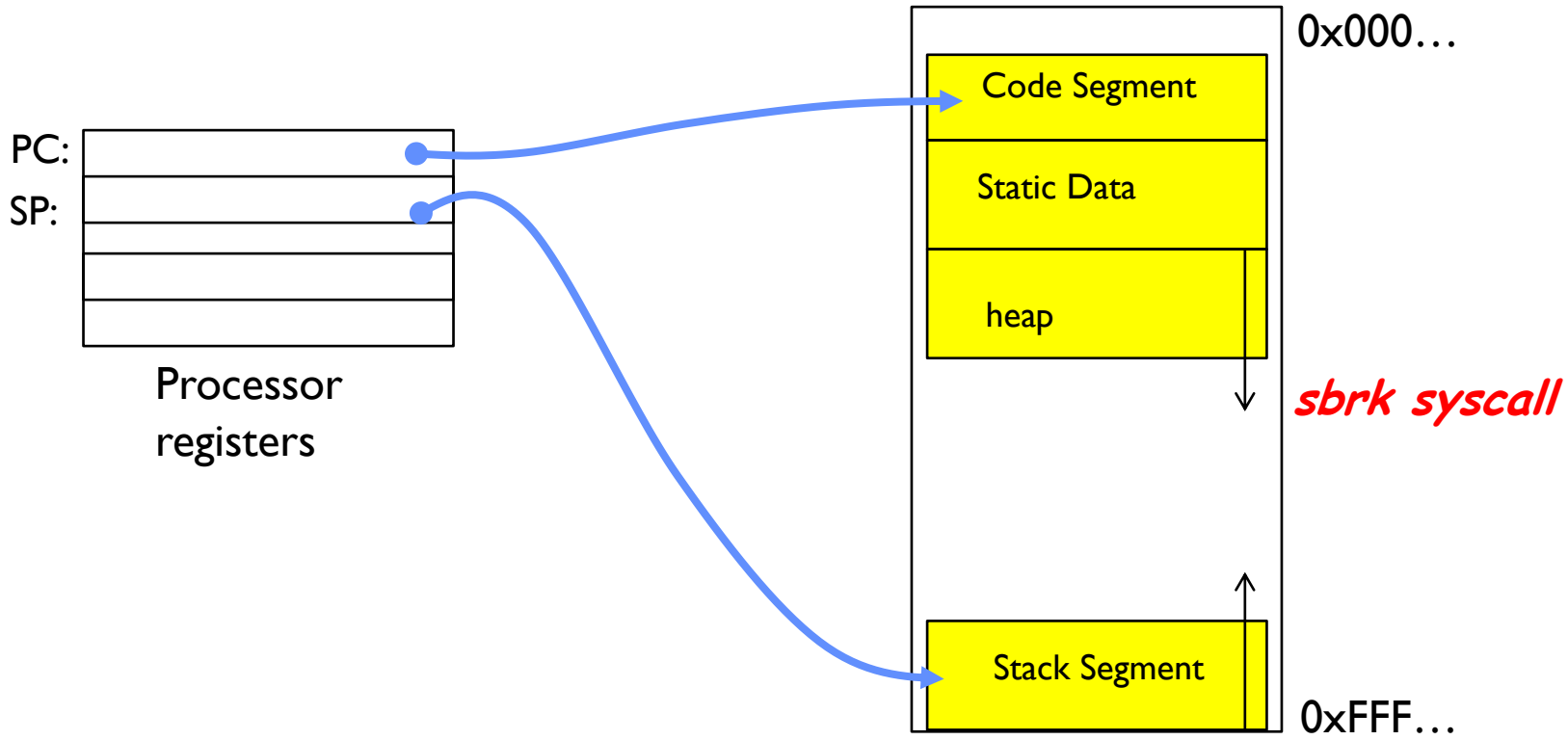
- What is 2^{10} ?
- How many bits to address each byte of 4kb page?
- How much memory can be addressed with 20 bits? 32 bits?
- 48 bits?

Address Space, Process Virtual Address Space

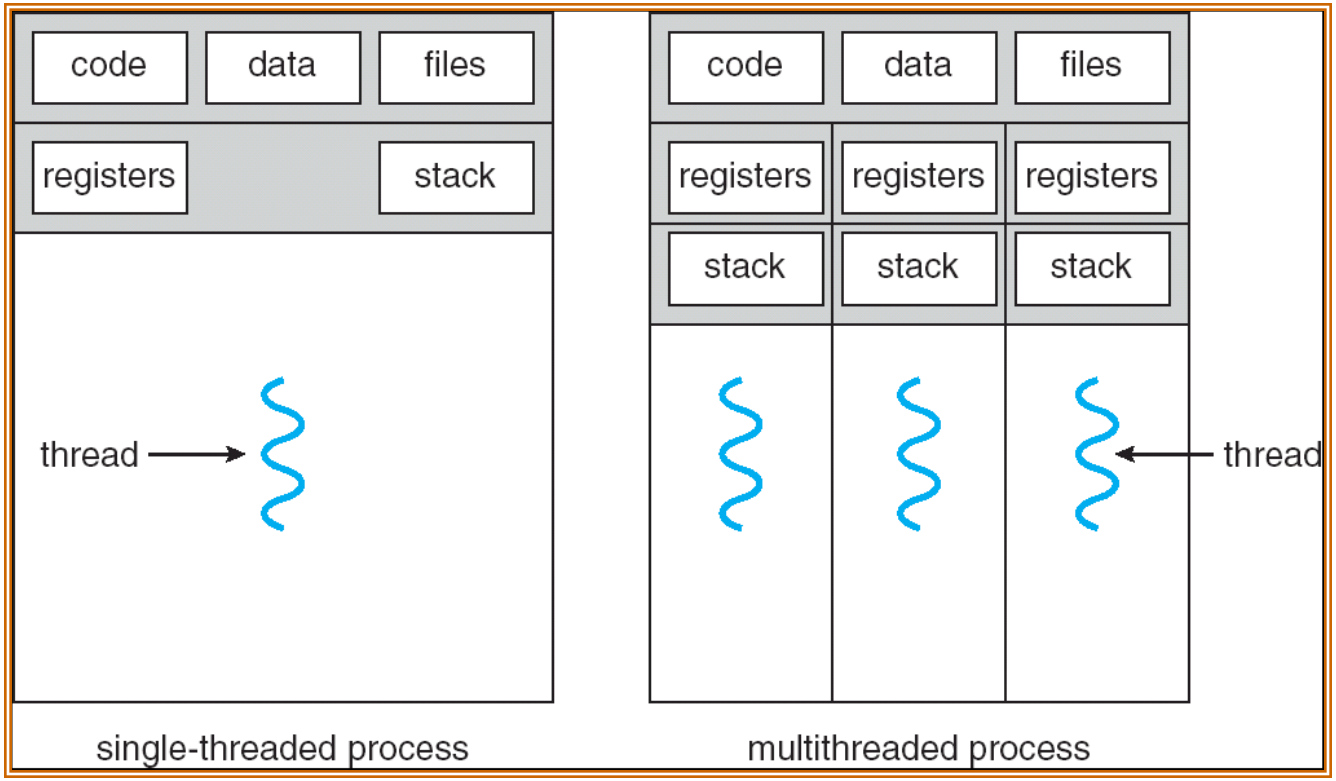
- Definition: **Set of accessible addresses and the state associated with them**
  - 2^{32} = ~4 billion on a 32-bit machine
- What happens when processor reads or writes to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - …



Recall: Process Address Space: typical structure

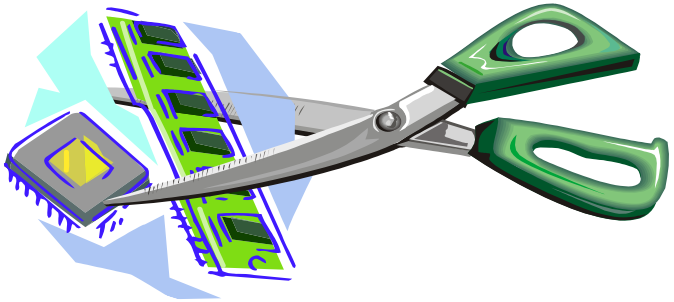


Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

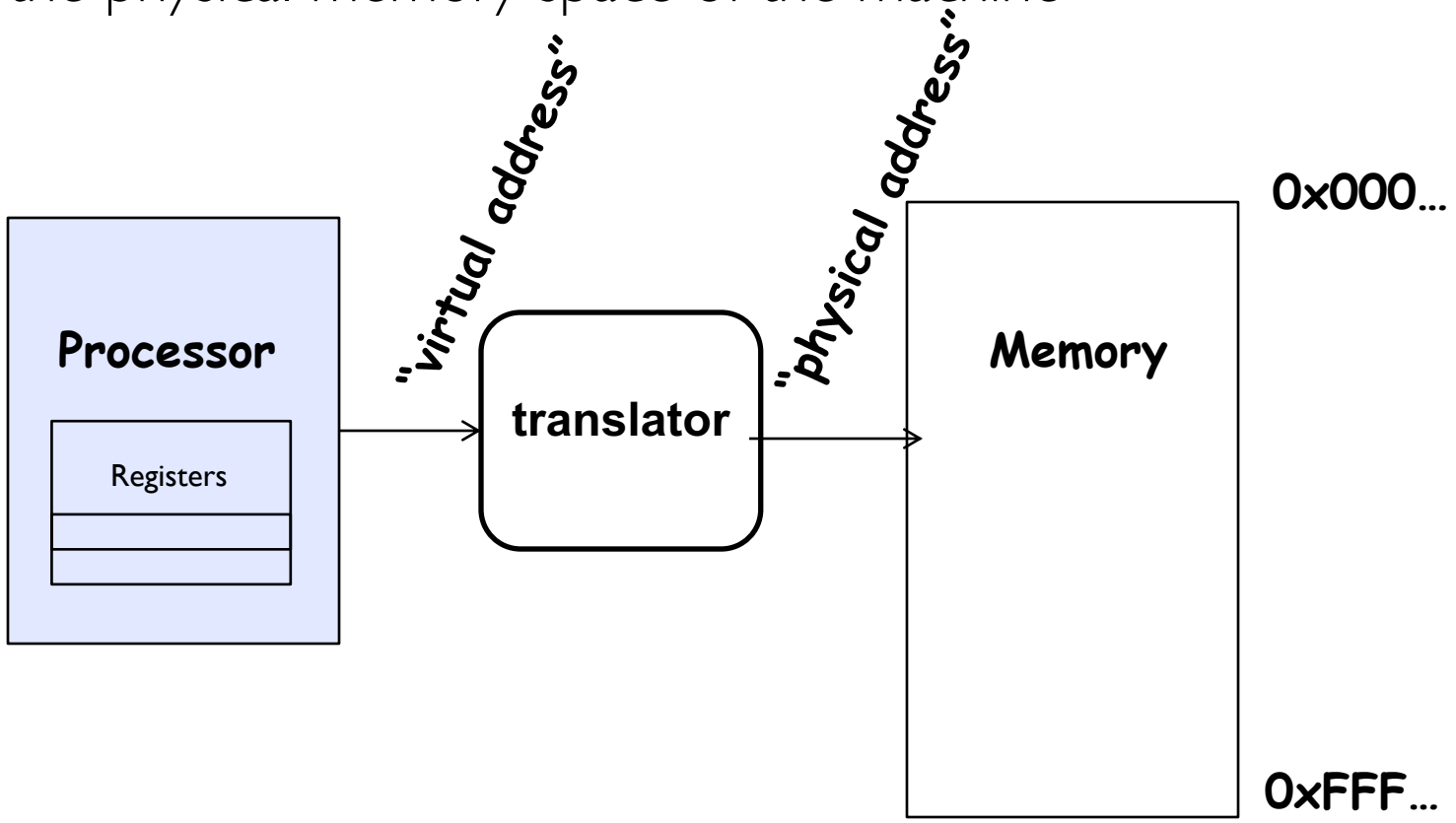
Virtualizing Resources



- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (starting today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    - » Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don't want different threads to even have access to each other's memory if in different processes (protection)

Recall: Key OS Concept: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine



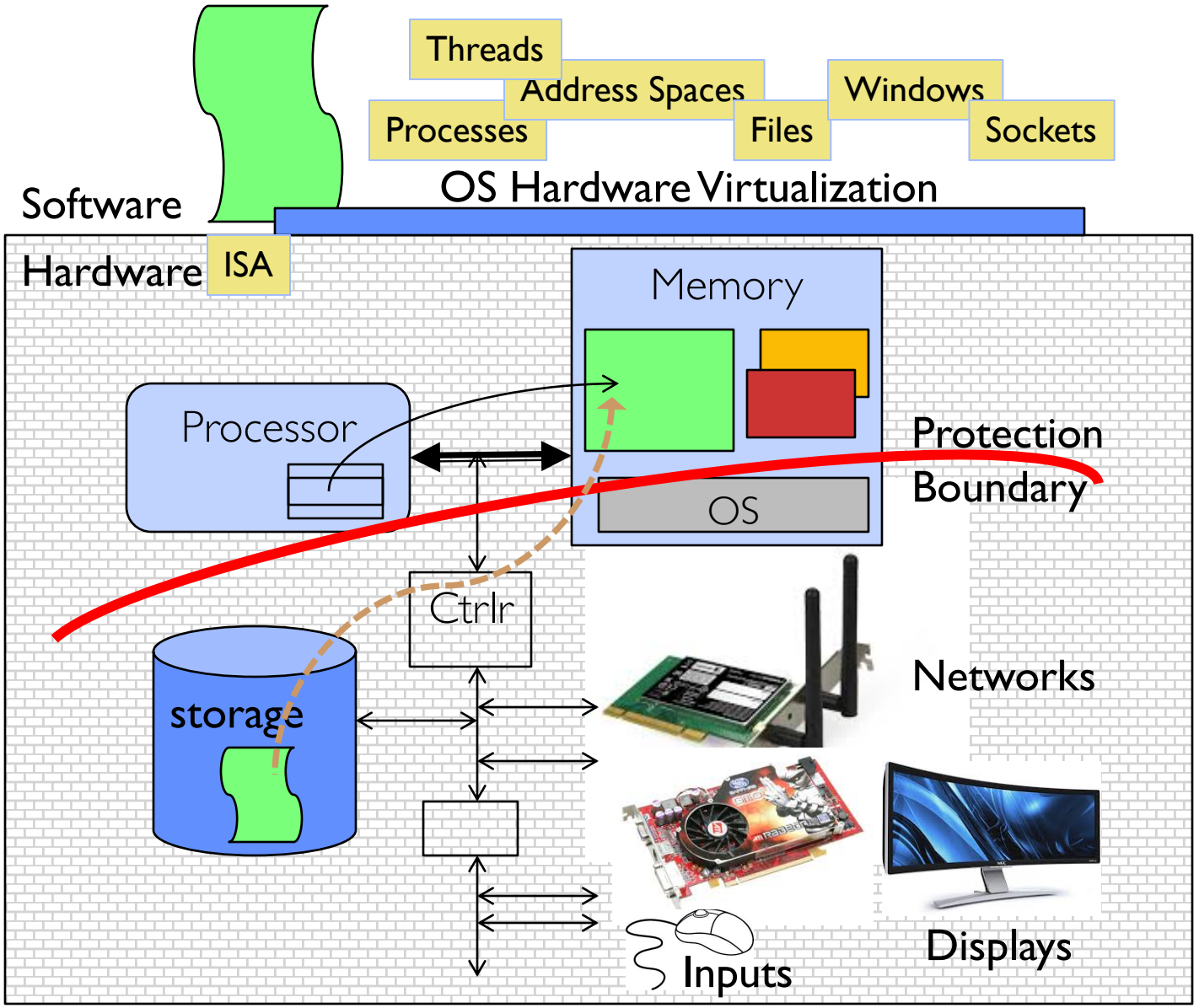
Important Aspects of Memory Multiplexing

- 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
- 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
  - Can be used to avoid overlap (isolation, protection)
  - Can be used to give uniform view of memory to programs
- Controlled overlap:
  - There are cases where overlap may be desired
    - » Read-Only data, Execute-Only shared libraries,
    - » Inter-process communication

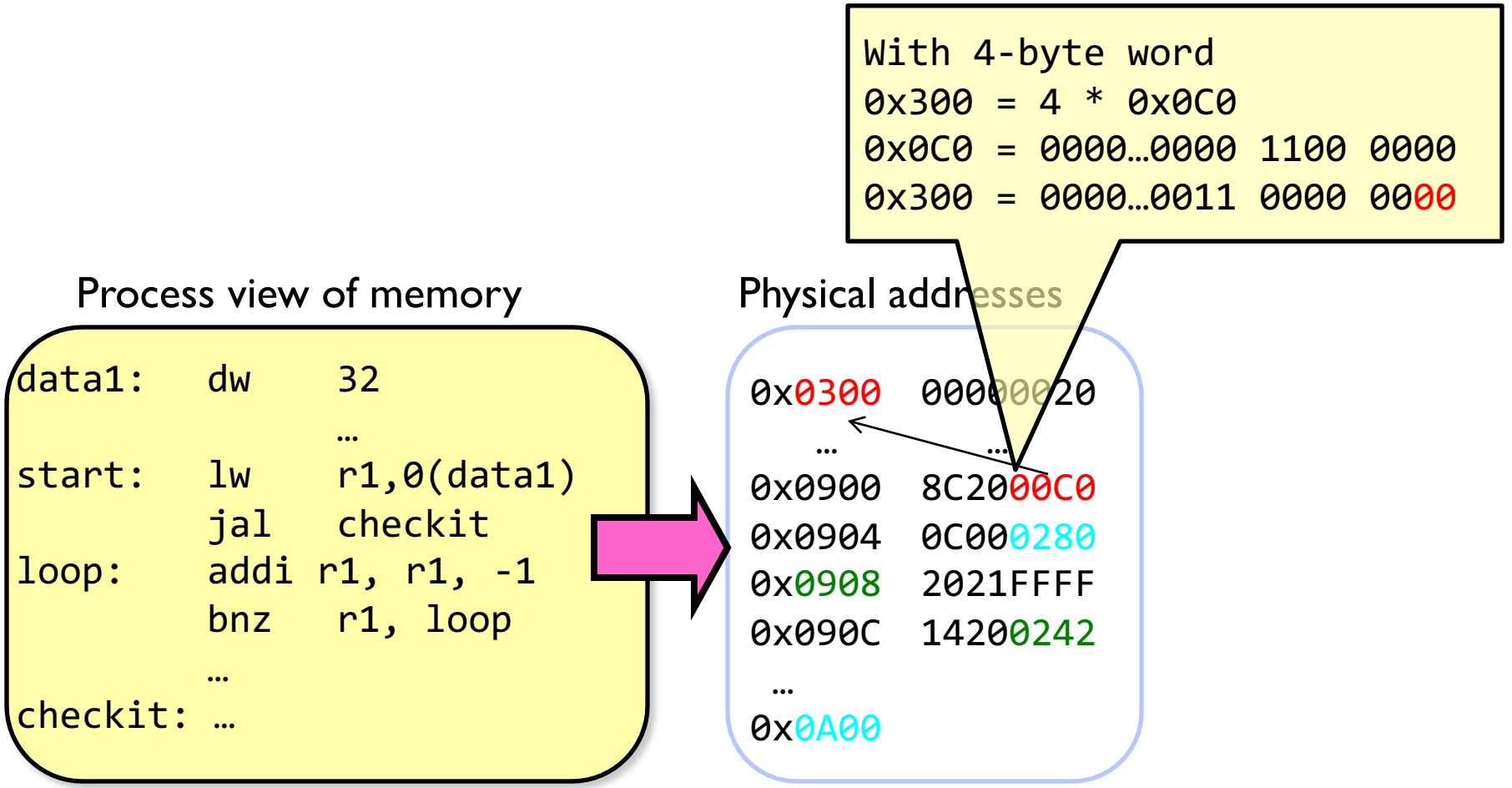
Approach

- Let's start simple and physical
- Work our way up to typical (page-based) virtual address translation

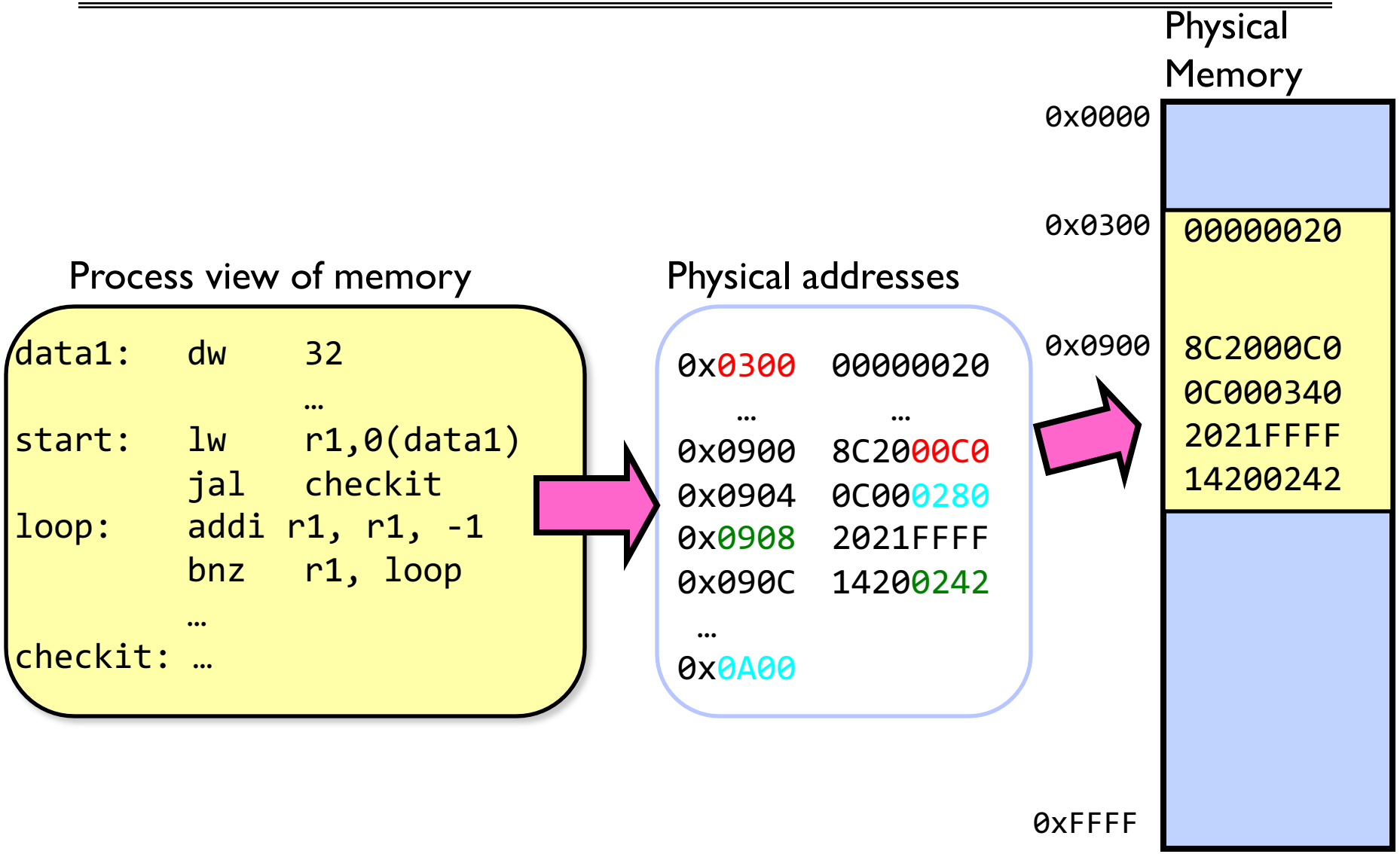
Recall: Load Pgm in Mem, Create Proc.



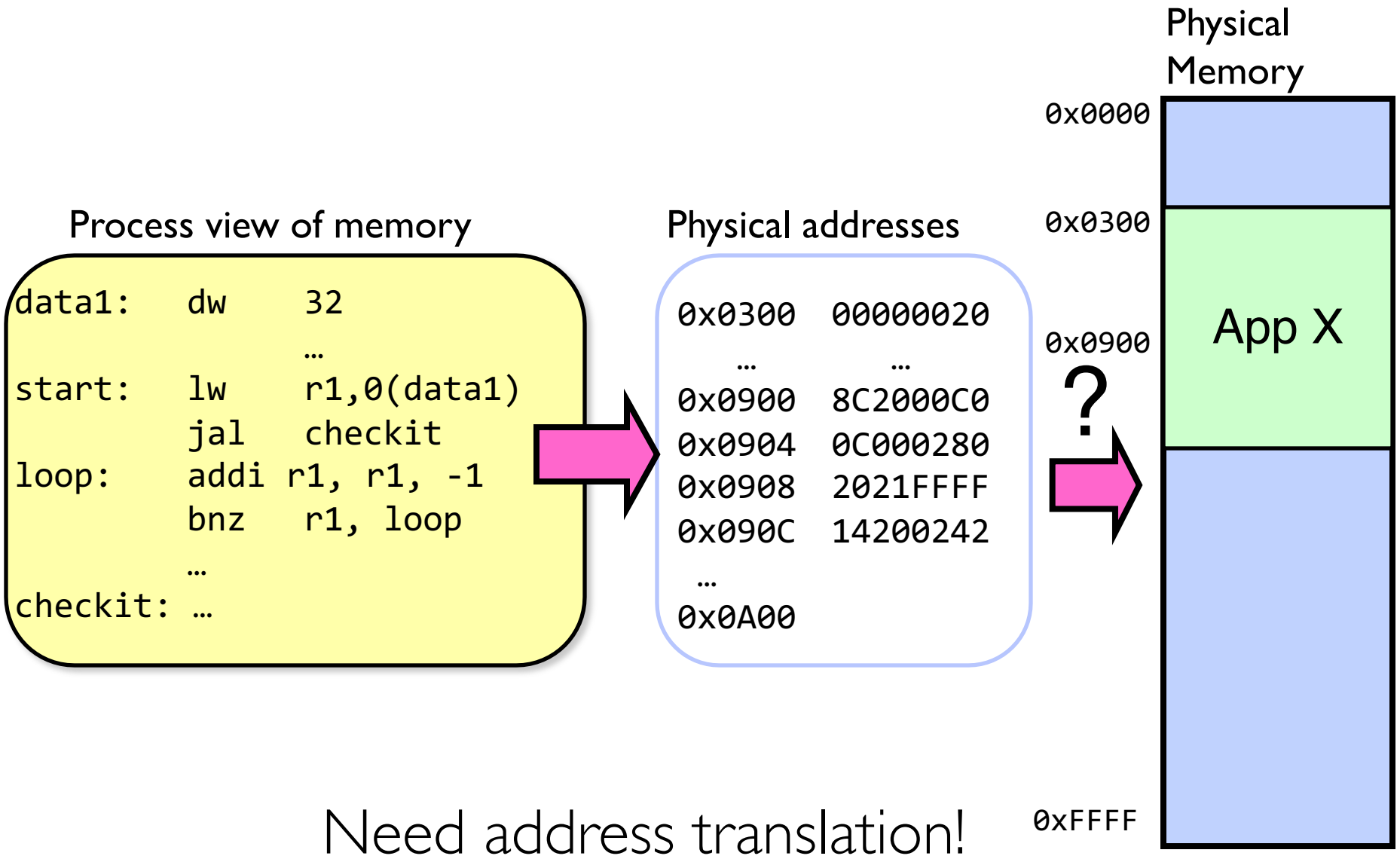
Binding of Instructions and Data to Memory



Binding of Instructions and Data to Memory



What about a 2nd instance of this program ?



Need address translation!

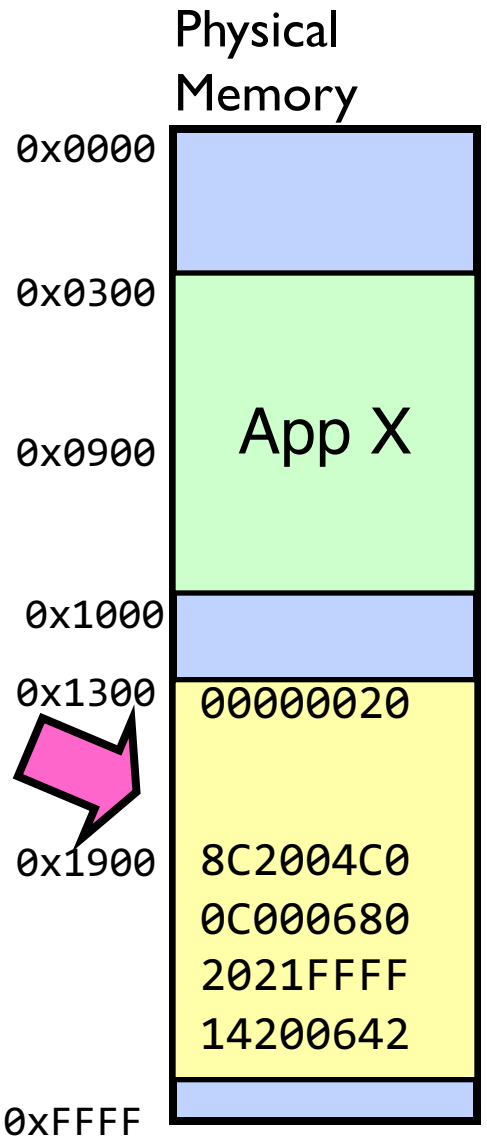
What about a 2nd instance of this program ?

Process view of memory

```
data1:   dw    32
         …
start:   lw    r1,0(data1)
         jal   checkit
loop:    addi  r1, r1, -1
         bnz   r1, loop
         …
checkit: …
```

Physical addresses

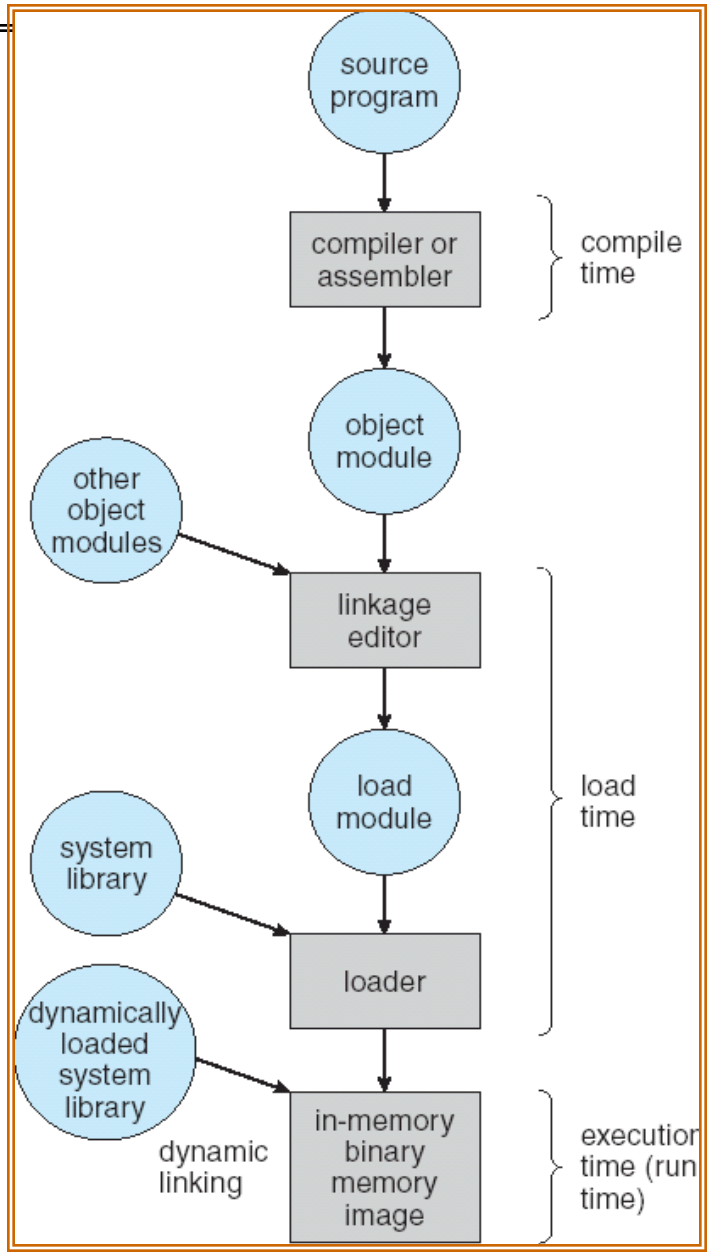
```
0x1300   00000020
  …        …
0x1900   8C2004C0
0x1904   0C000680
0x1908   2021FFFF
0x190C   14200642
 …
0x1A00
```



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

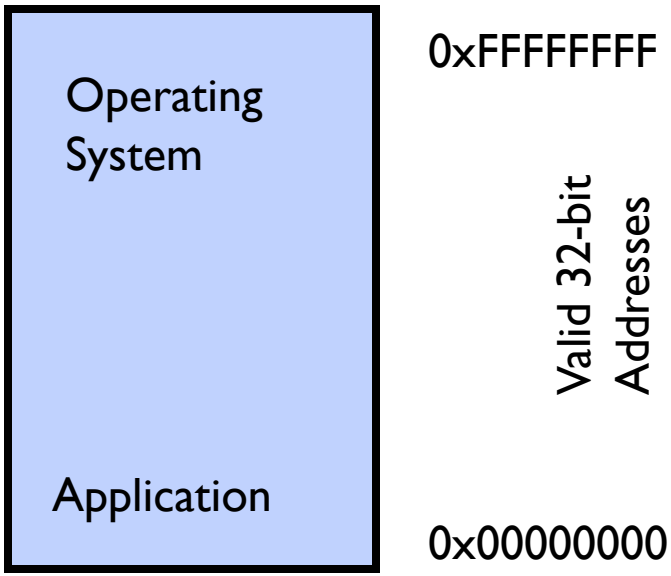
Multi-step Processing of a Program for Execution

- 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 (i.e. the *stub*), locates appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Uniprogramming – one process at a time

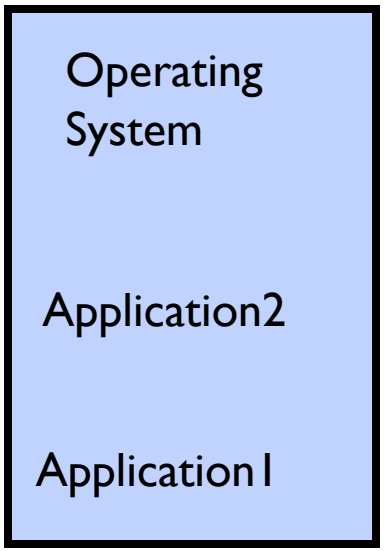
- no Translation or Protection
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address

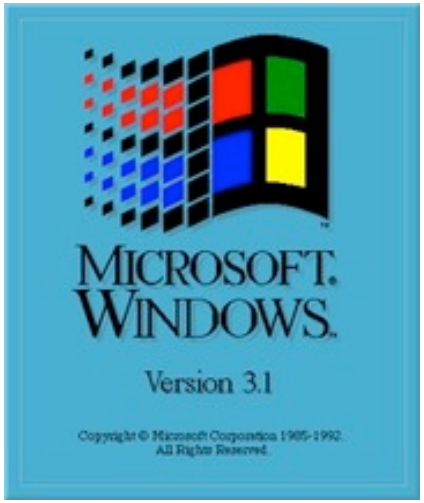


  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Multiprogramming (primitive stage)

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

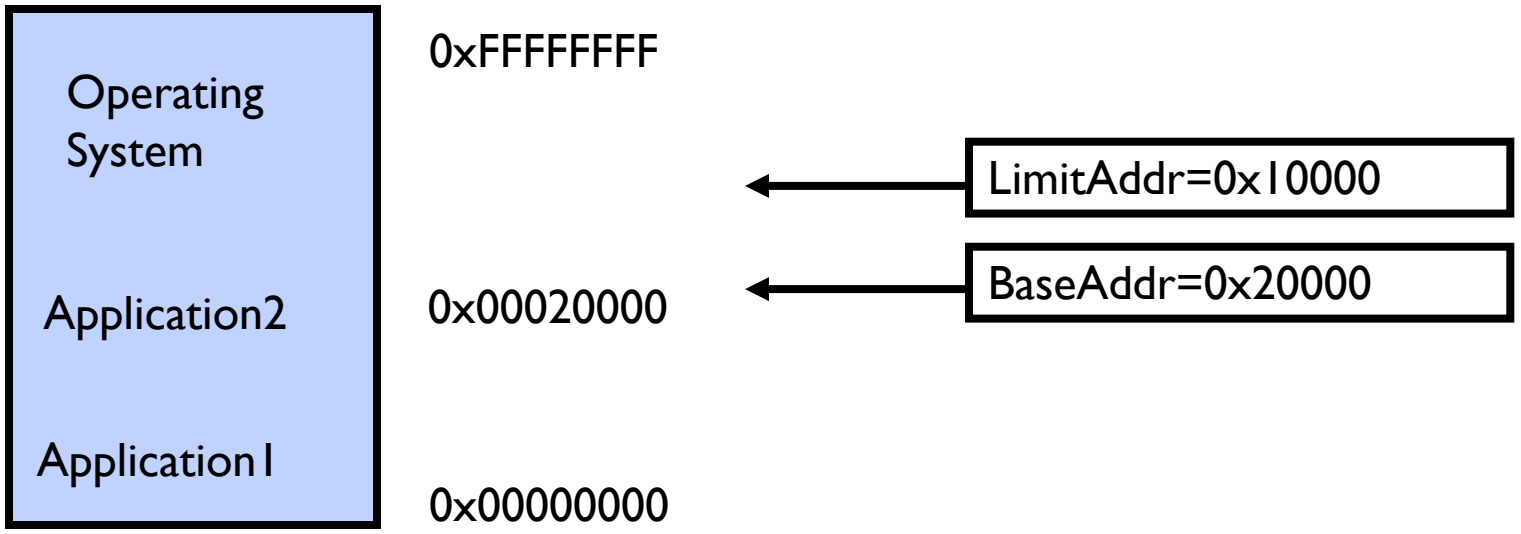




  - Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - » Everything adjusted to memory location where OS put program
    - » Translation done by a linker-loader (relocation)
    - » Common in early days (… till Windows 3.x, 95?)
- no protection: bugs in any program can cause others to crash or even the OS (and lots of hacks, e.g., TSR)

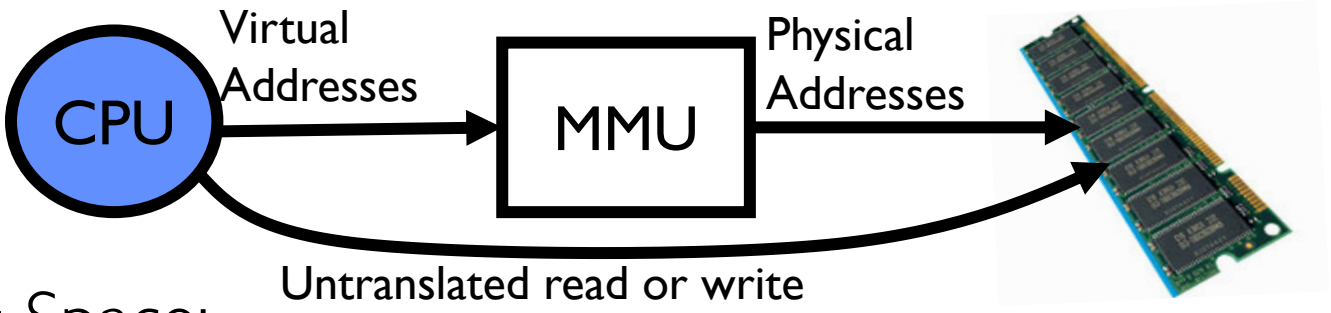
Multiprogramming - with Protection

- 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
    - » Cause error if user tries to access an illegal address
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    - » User not allowed to change base/limit registers

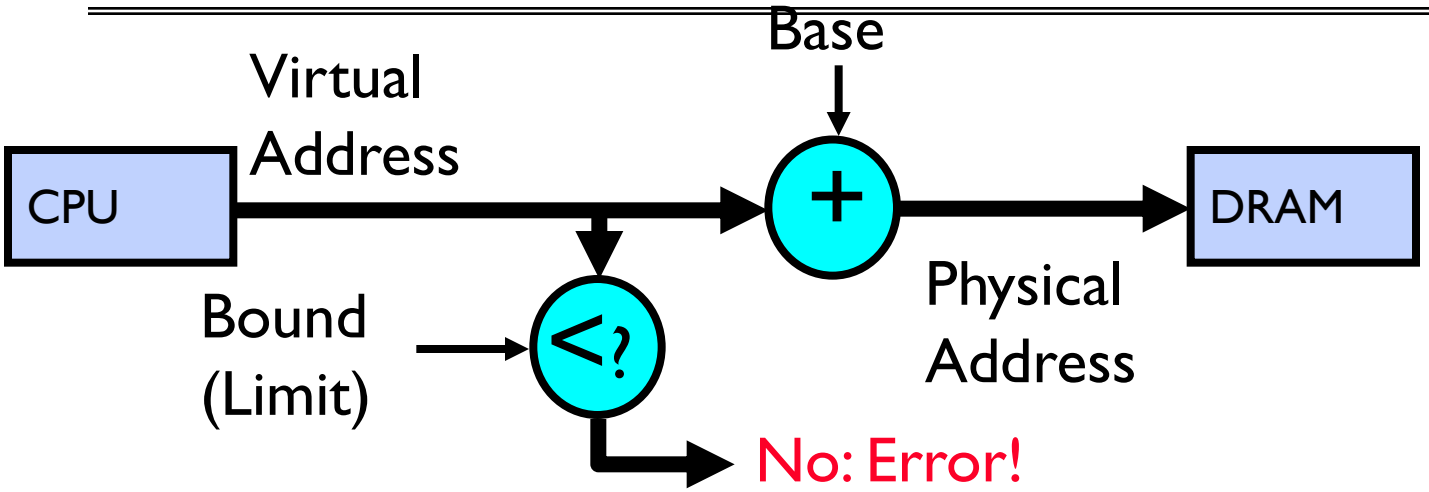
General Address translation

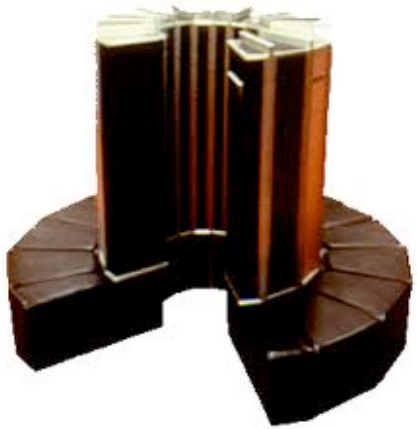


- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot 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 (no adjustments)

psst. MMU accesses Page Table

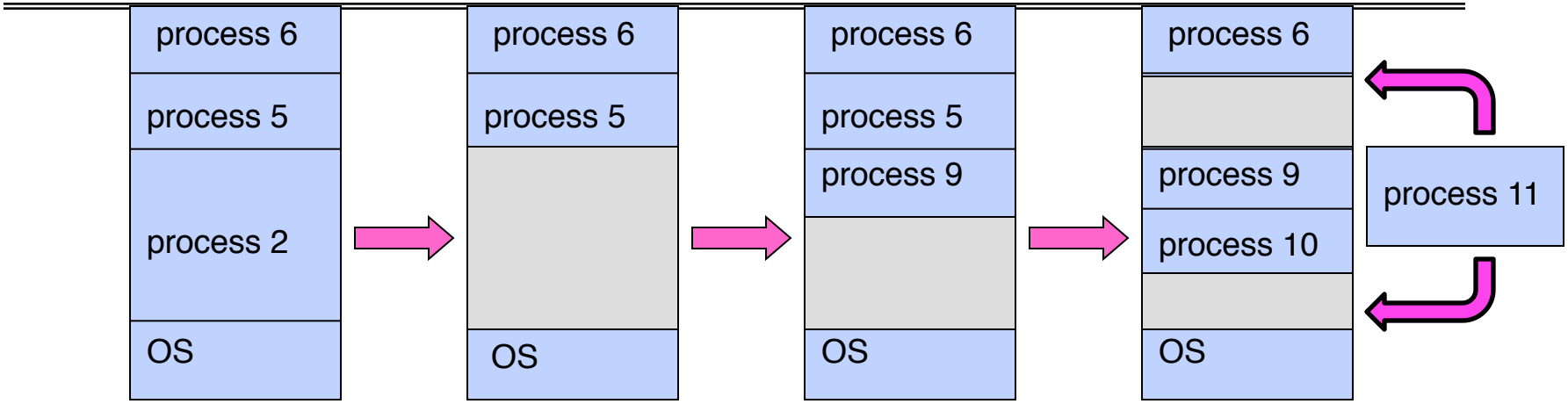
Recall: Base and Bounds (e.g., CRAY-1)





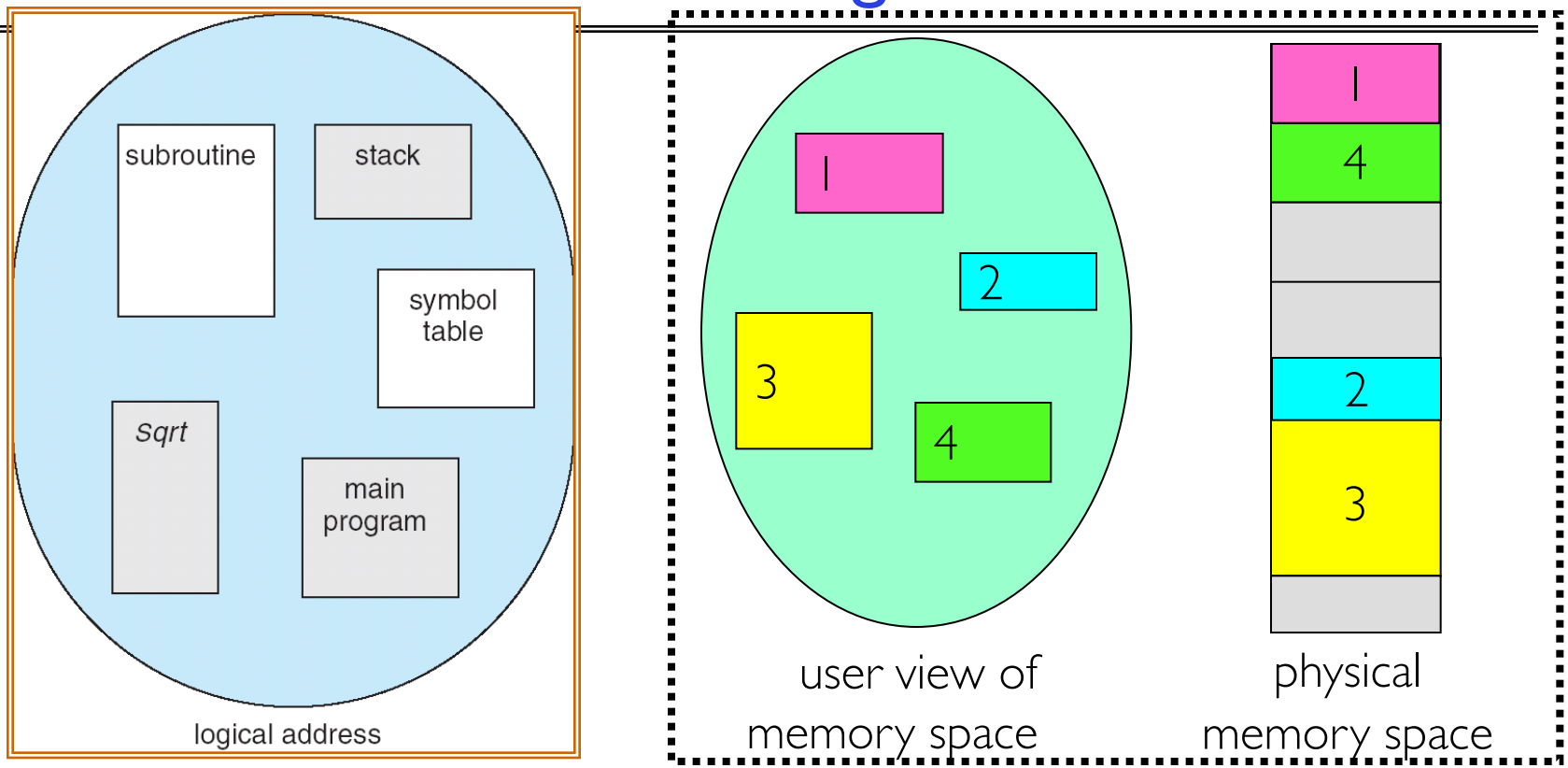
- Could use base/bounds for dynamic address translation – translation happens at execution:
  - Alter address of every load/store by adding "base"
  - Generate error if address bigger than limit
- Gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
 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

Issues with Simple B&B Method



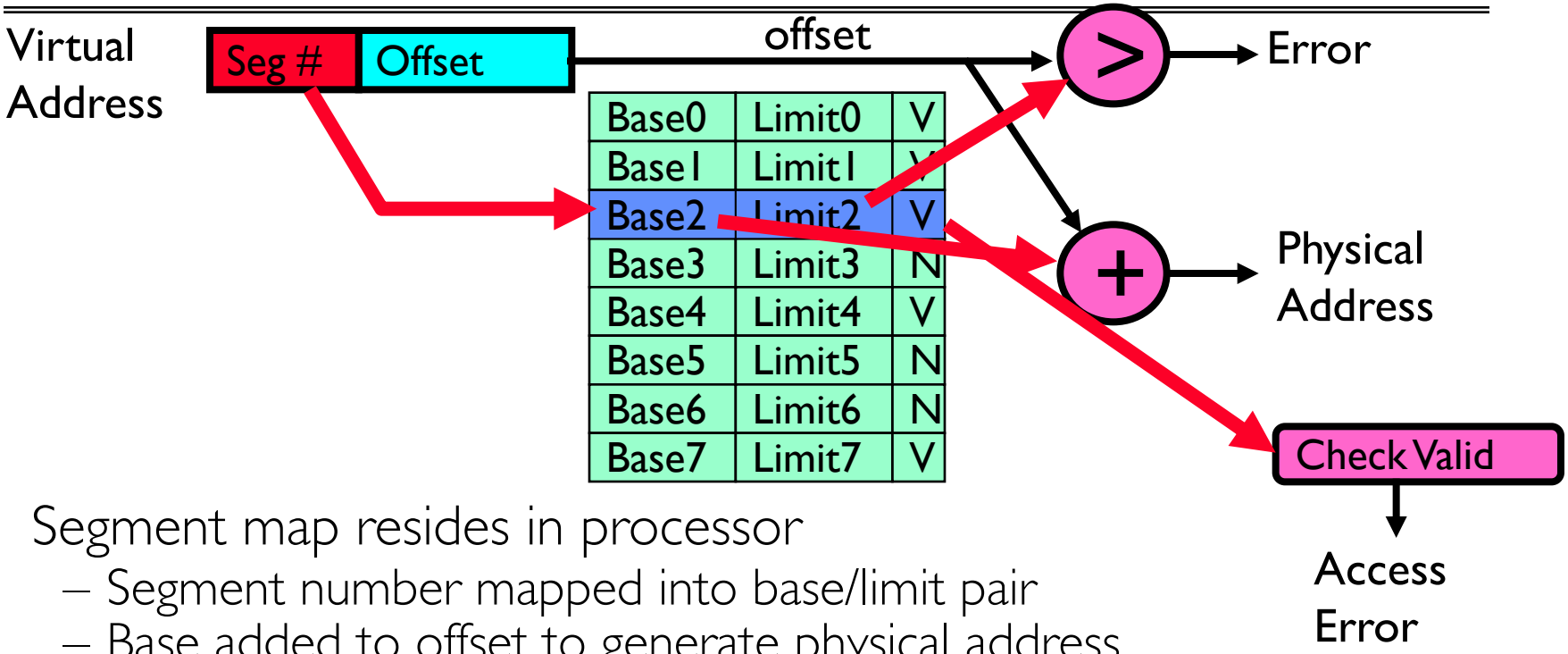
- Fragmentation problem over time
  - Not every process is same size ⇒ memory becomes fragmented over time
- Missing support for sparse address space
  - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

More Flexible Segmentation



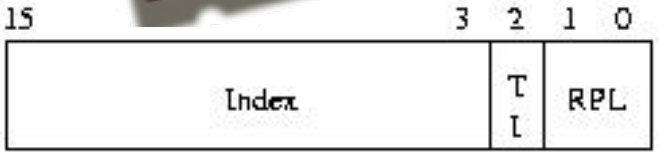
- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

Implementation of Multi-Segment Model



- 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

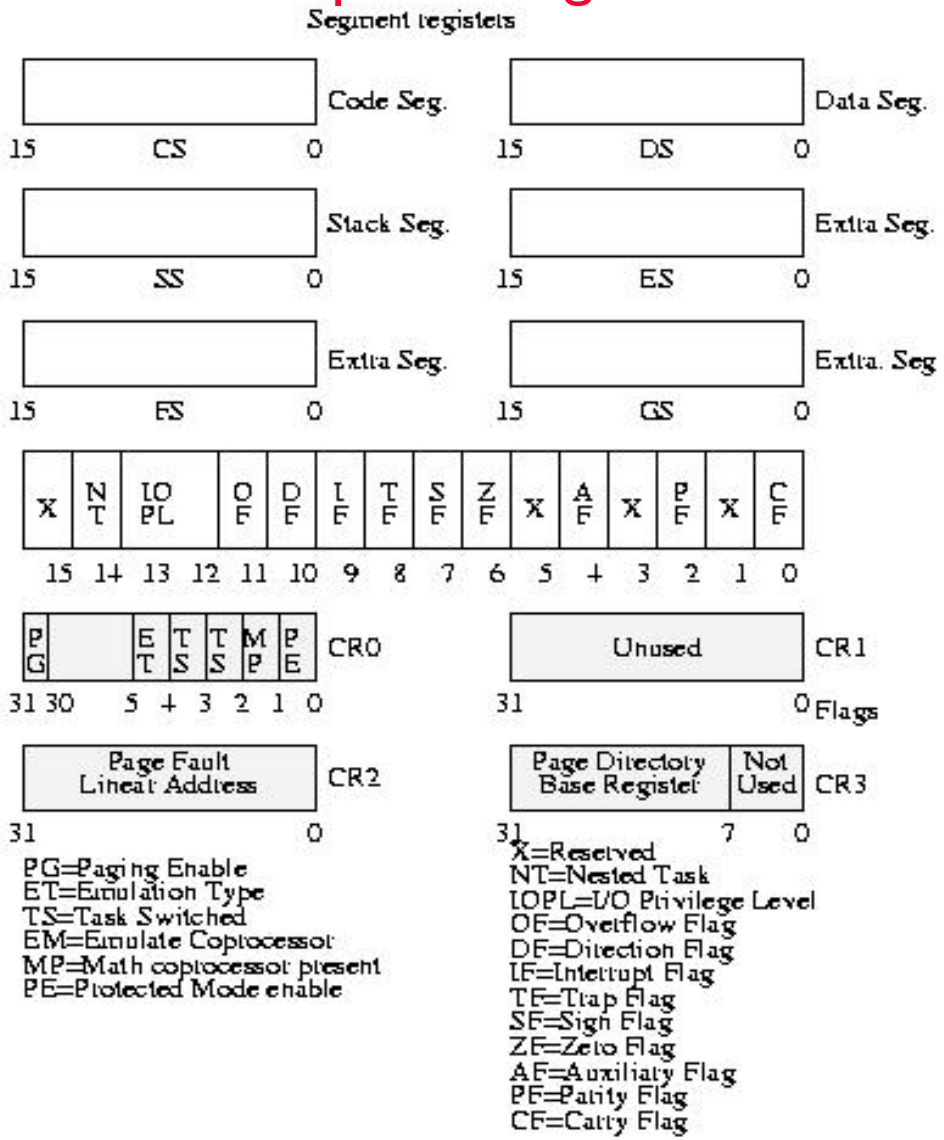
Intel x86 Special Registers



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

Protected Mode segment selector

80386 Special Registers

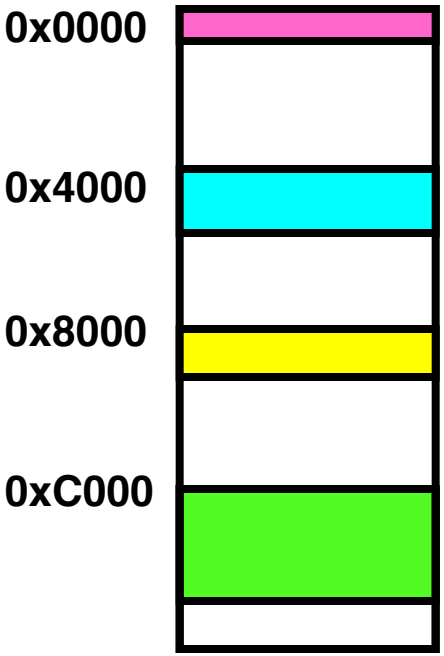


Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

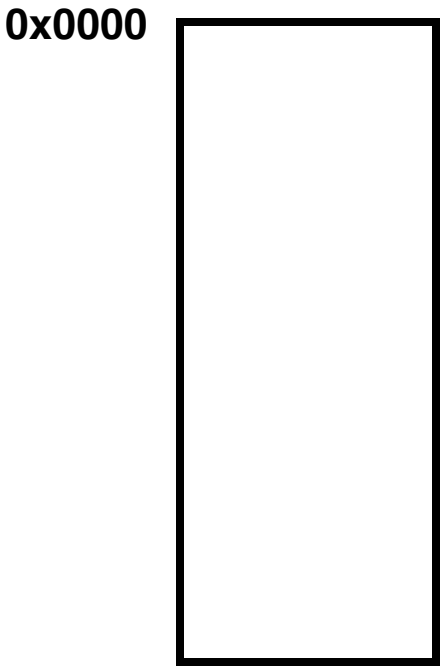
Example: Four Segments (16 bit addresses)

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

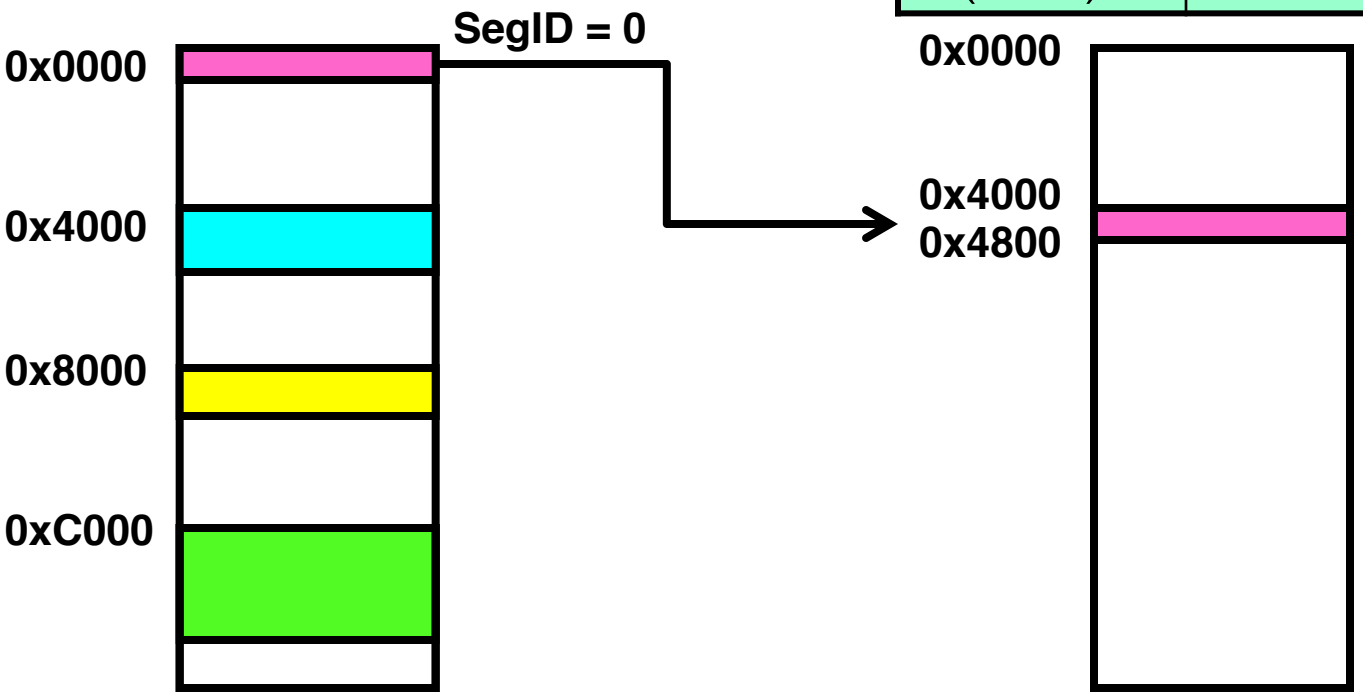


Physical
Address Space

Example: Four Segments (16 bit addresses)

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

Virtual Address Format



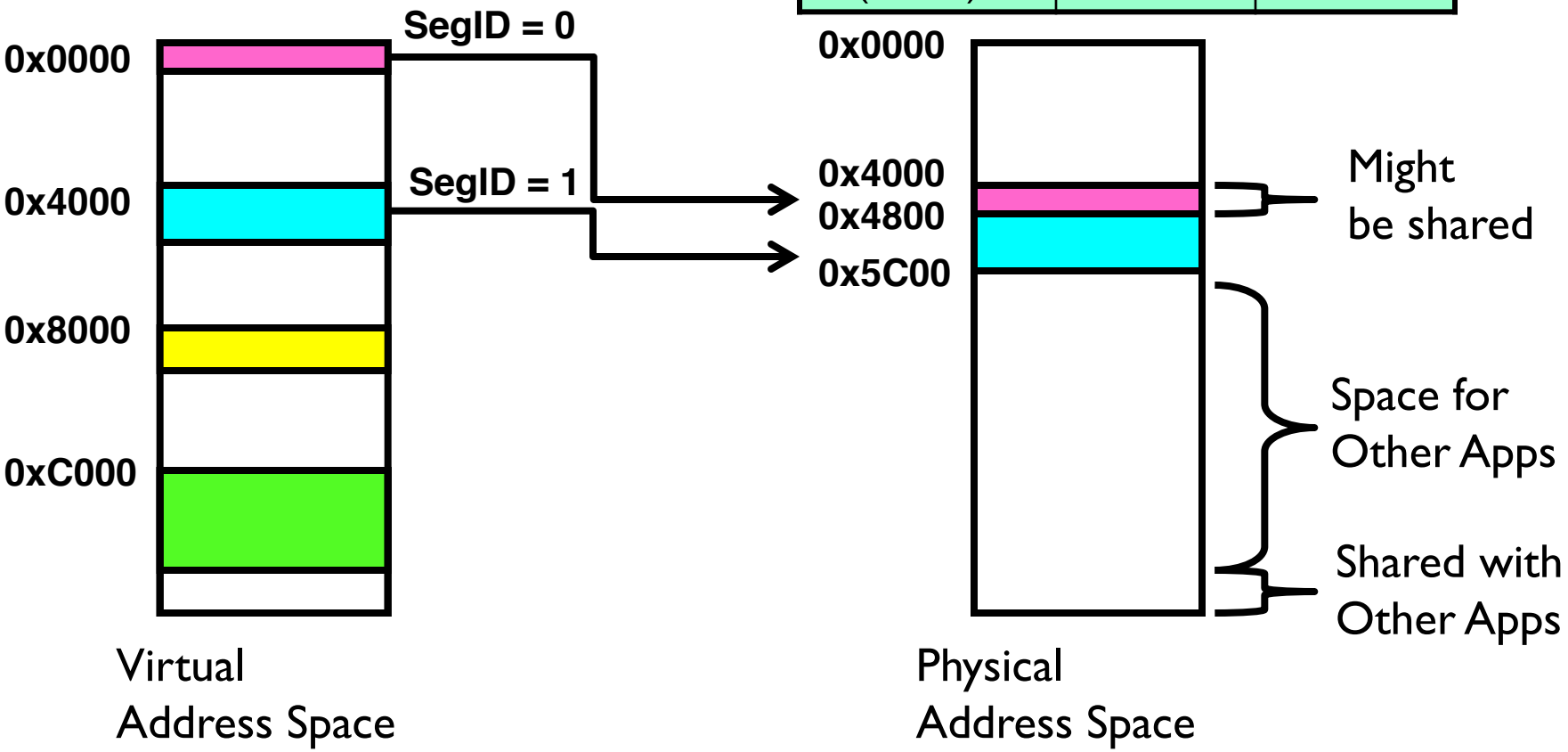
Virtual Address Space

Physical Address Space

Example: Four Segments (16 bit addresses)

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

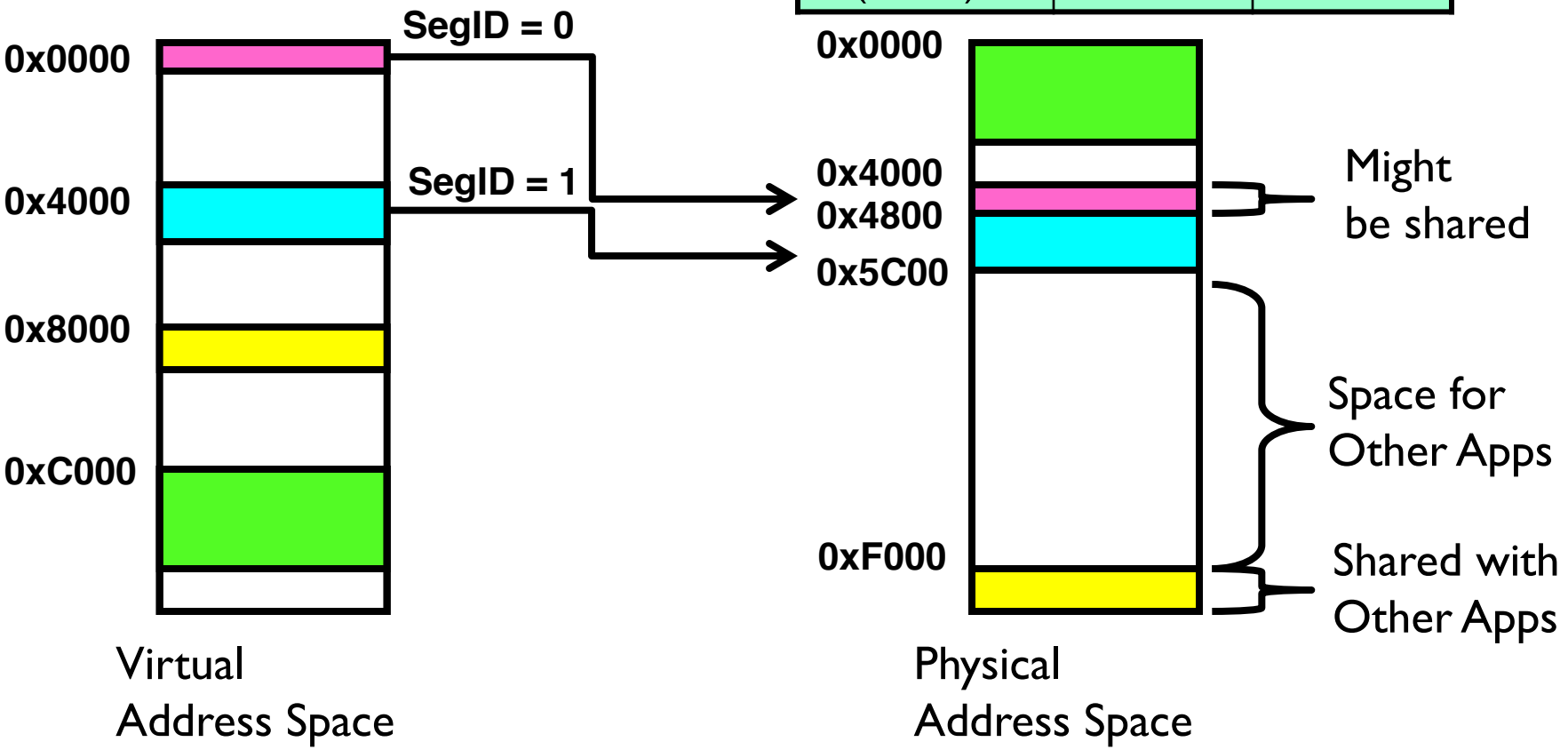
Virtual Address Format



Example: Four Segments (16 bit addresses)

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

Virtual Address Format



Example of Segment Translation (16bit address)

```
0x240    main:    la $a0, varx
0x244             jal strlen
  …                  …
0x360    strlen:  li   $v0, 0  ;count
0x364    loop:    lb   $t0, ($a0)
0x368             beq  $r0,$t0, 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 0x0240 (0000 0010 0100 0000). 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

Example of Segment Translation (16bit address)

```
0x240    main:    la $a0, varx
0x244             jal strlen
  …                  …
0x360    strlen:  li   $v0, 0  ;count
0x364    loop:    lb   $t0, ($a0)
0x368             beq  $r0,$t0, 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 0x0240 (0000 0010 0100 0000). 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

Example of Segment Translation (16bit address)

```
0x240    main:    la $a0, varx
0x244             jal strlen
  …                  …
0x360    strlen:  li   $v0, 0  ;count
0x364    loop:    lb   $t0, ($a0)
0x368             beq  $r0,$t0, 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 0x0240 (0000 0010 0100 0000). 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

Example of Segment Translation (16bit address)

```
0x0240   main:    la $a0, varx
0x0244            jal strlen
  …                  …
0x0360   strlen:  li   $v0, 0  ;count
0x0364   loop:    lb   $t0, ($a0)
0x0368            beq  $r0,$t0, 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

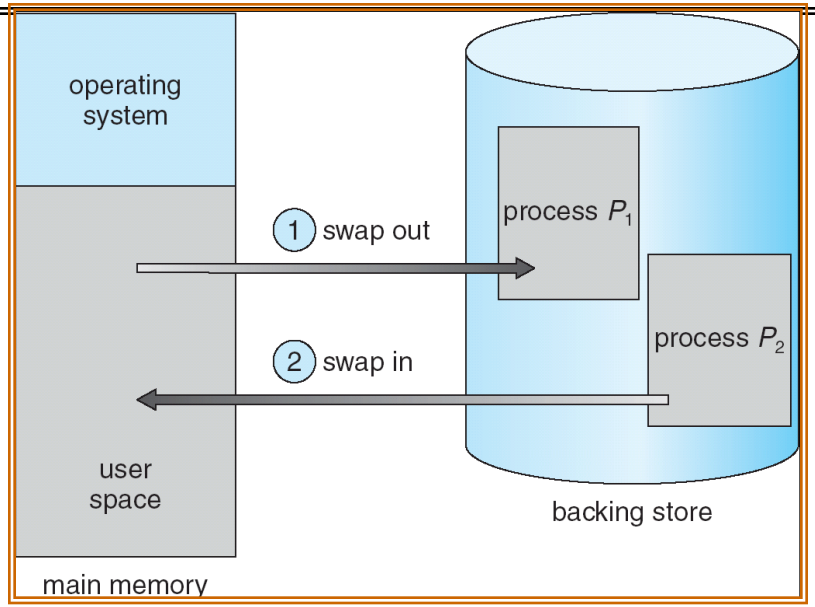
Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). 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 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x0360. Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x0364. Translated to Physical=0x4364. Get "lb $t0, ($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Observations about Segmentation

- Translation on *every* instruction fetch, load or store
- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never access 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 ?) 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")

What if not all segments fit into memory?

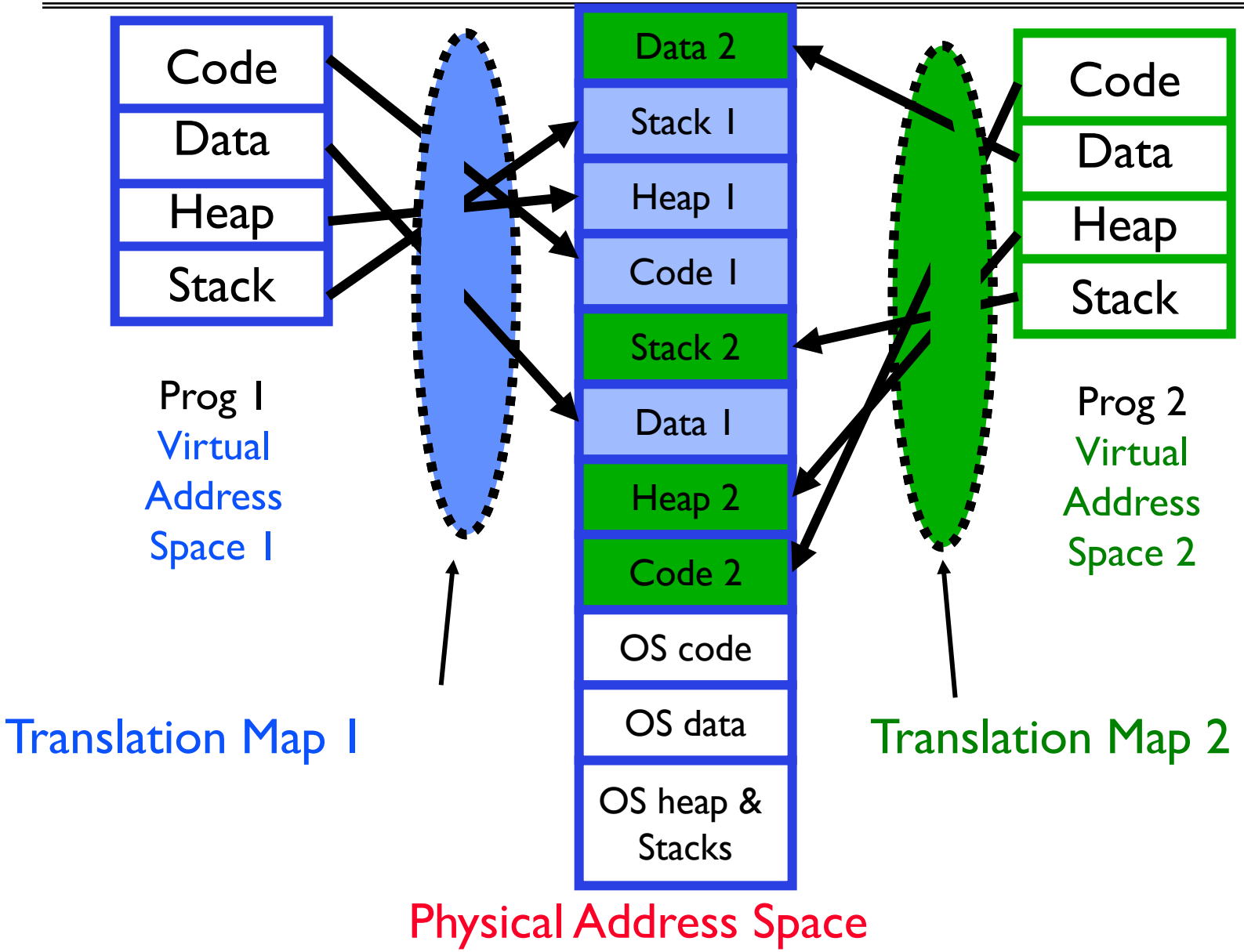


- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way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

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

Recall: General Address Translation

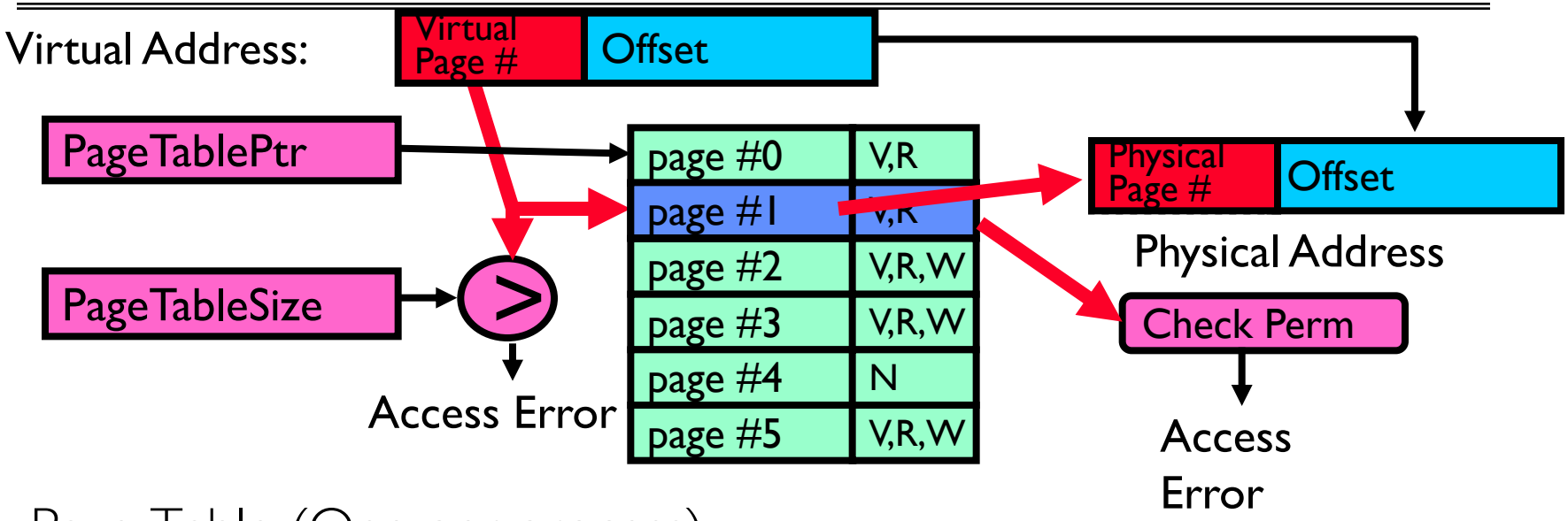


Paging: Physical Memory in Fixed Size Chunks

- 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

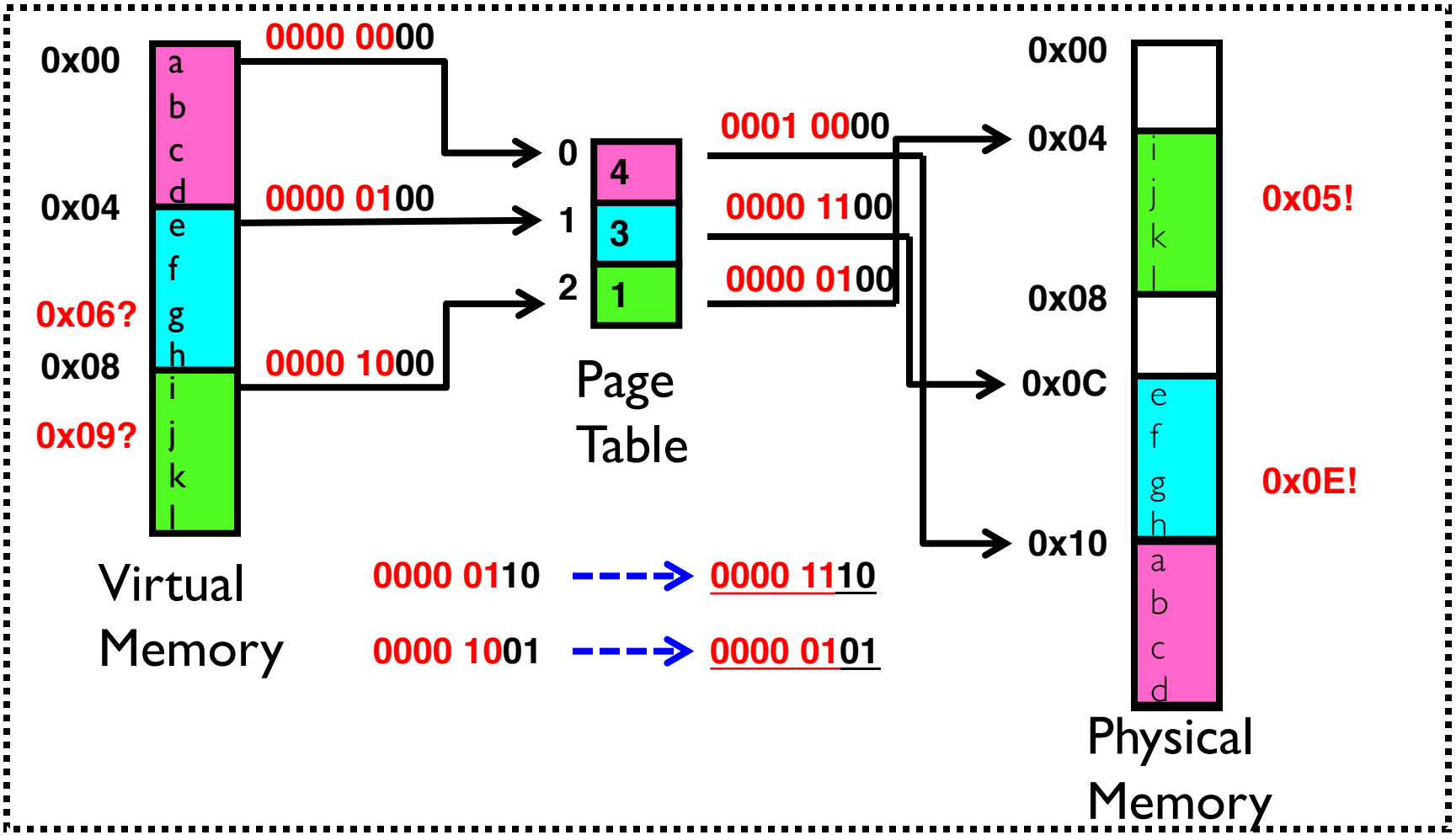
How to Implement Paging?



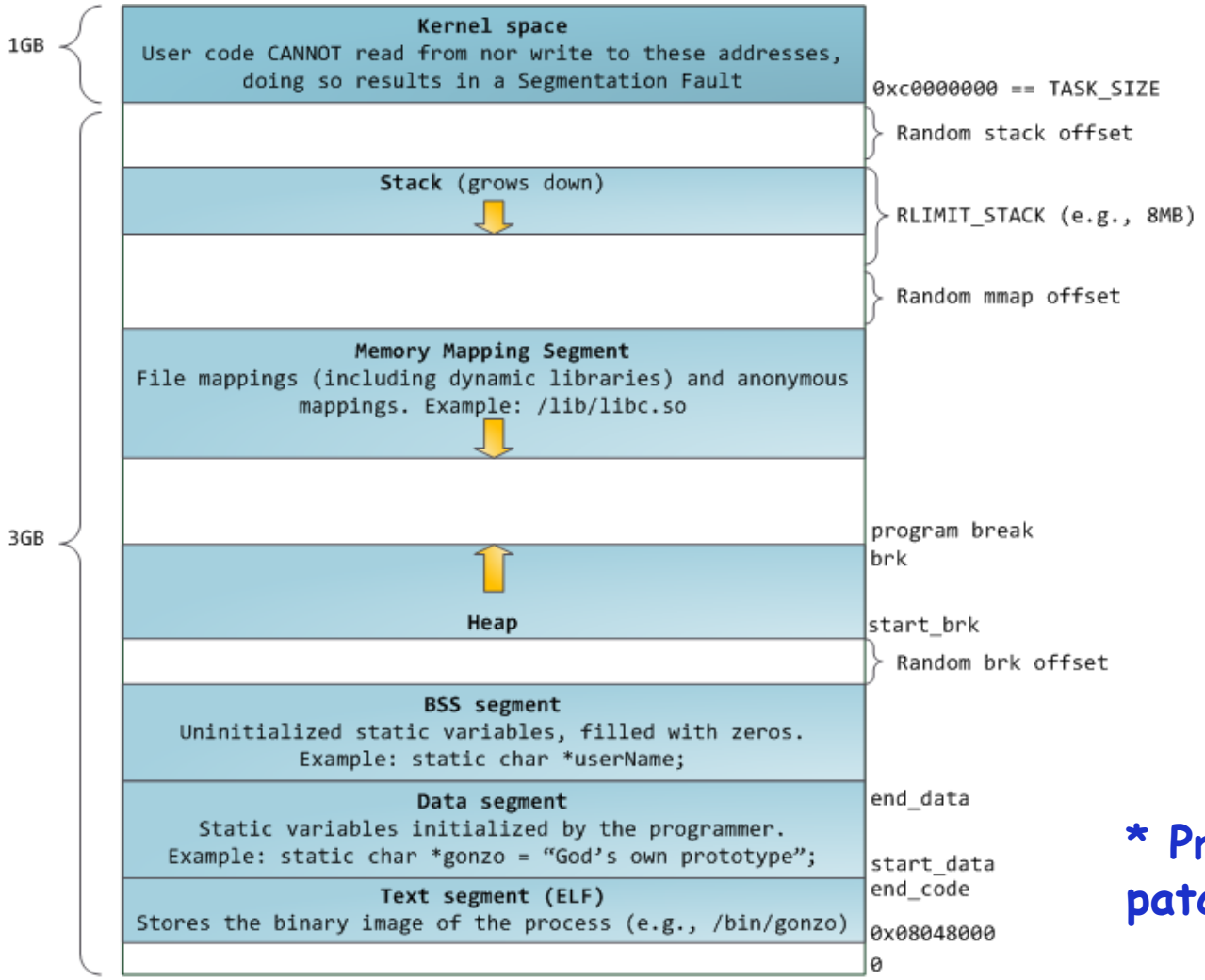
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 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

Simple Page Table Example

Example (4 byte pages)



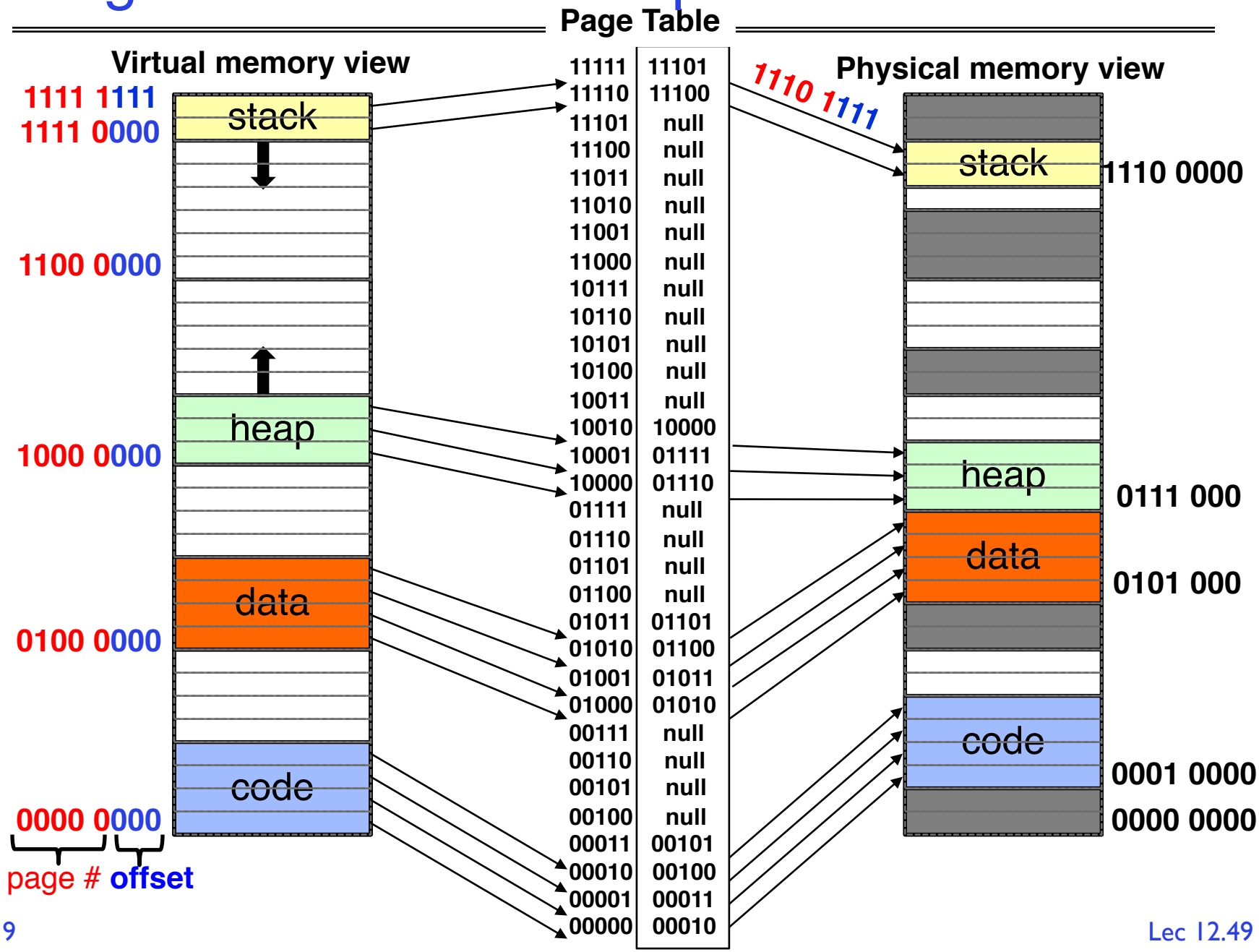
Example: Memory Layout for Linux 32-bit *



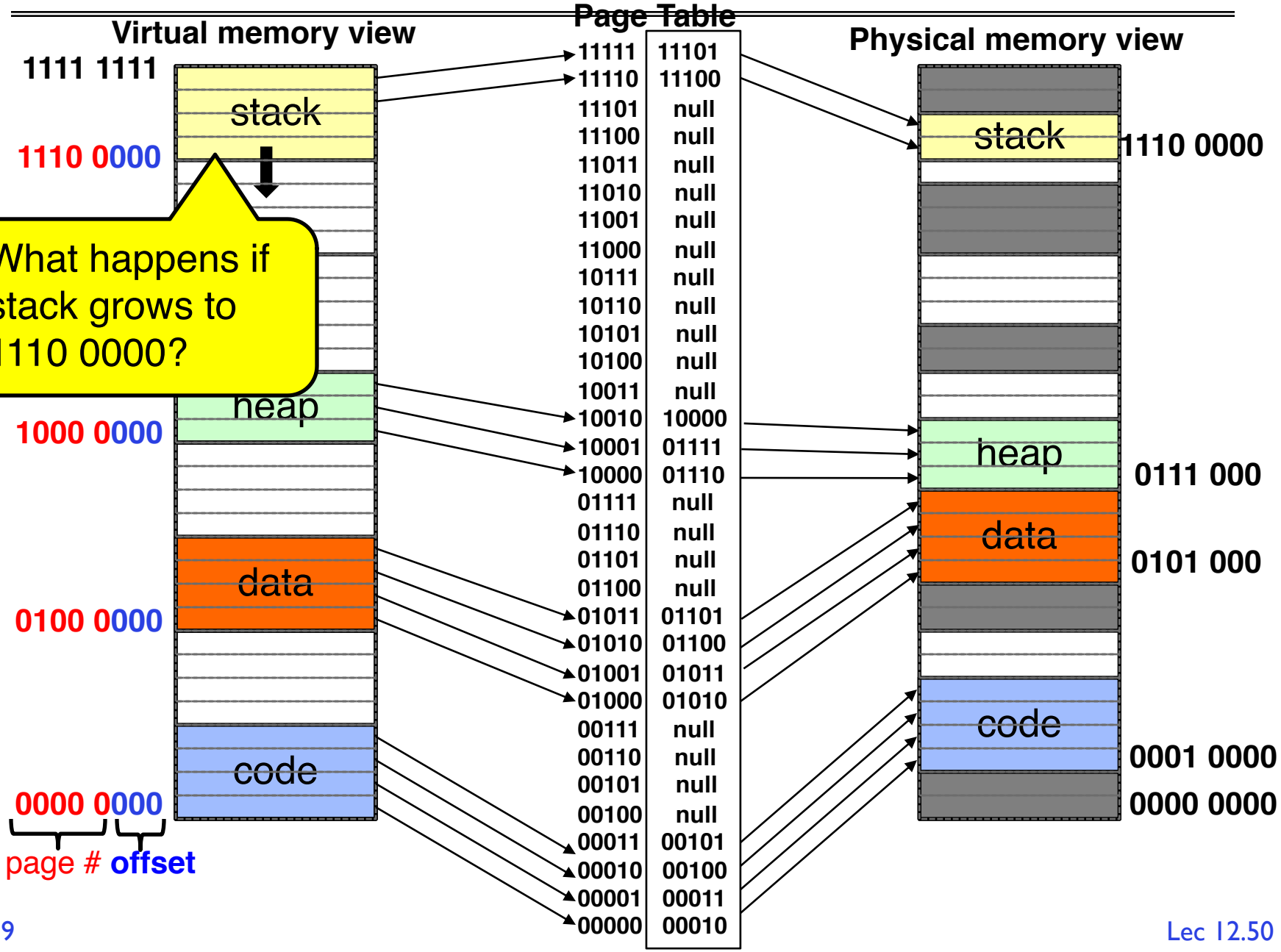
*** Pre-Meltdown patches, more later**

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

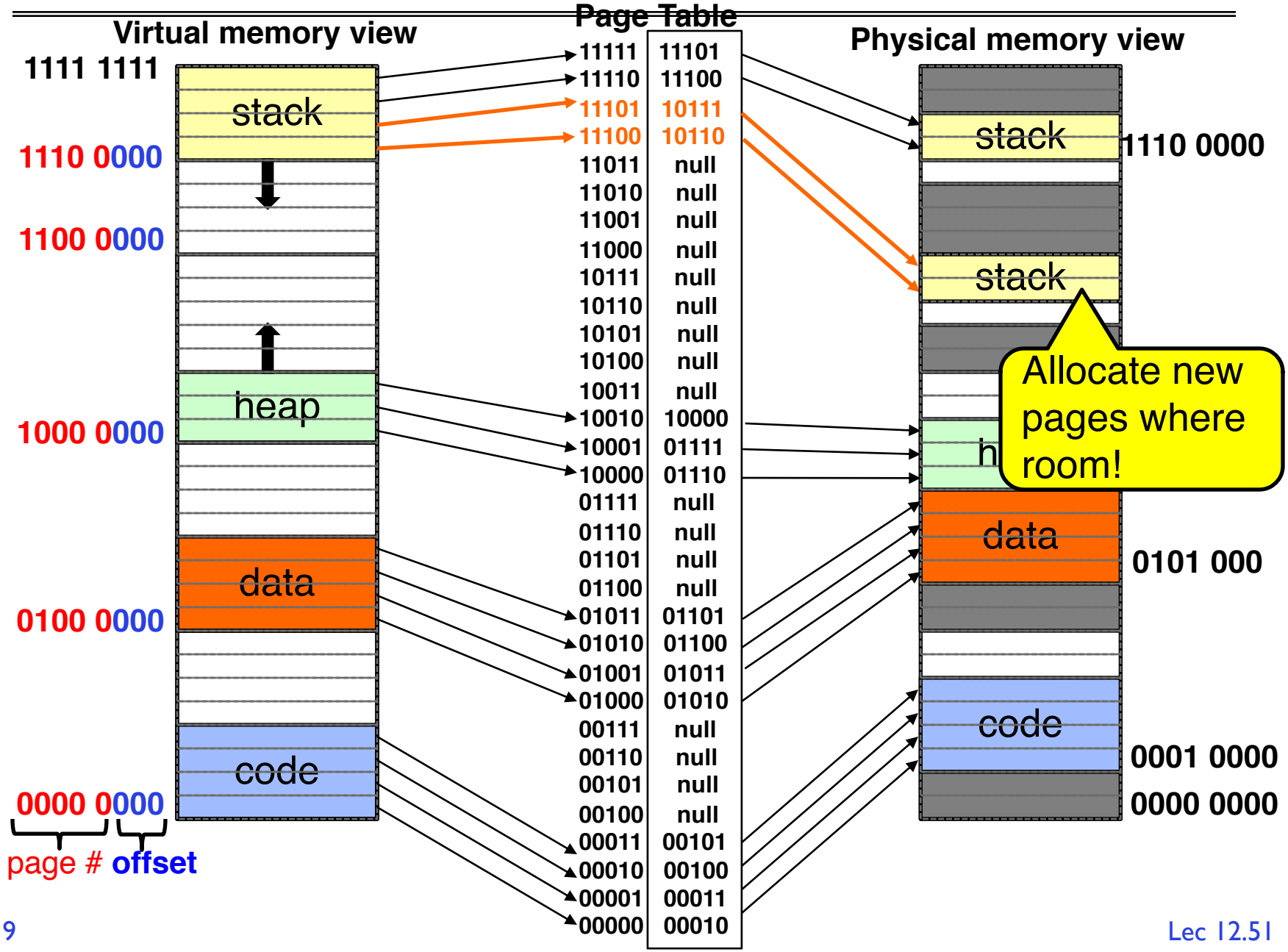
Paged virtual address space



Summary: Paging



Summary: Paging



x86 classic 32-bit address translation

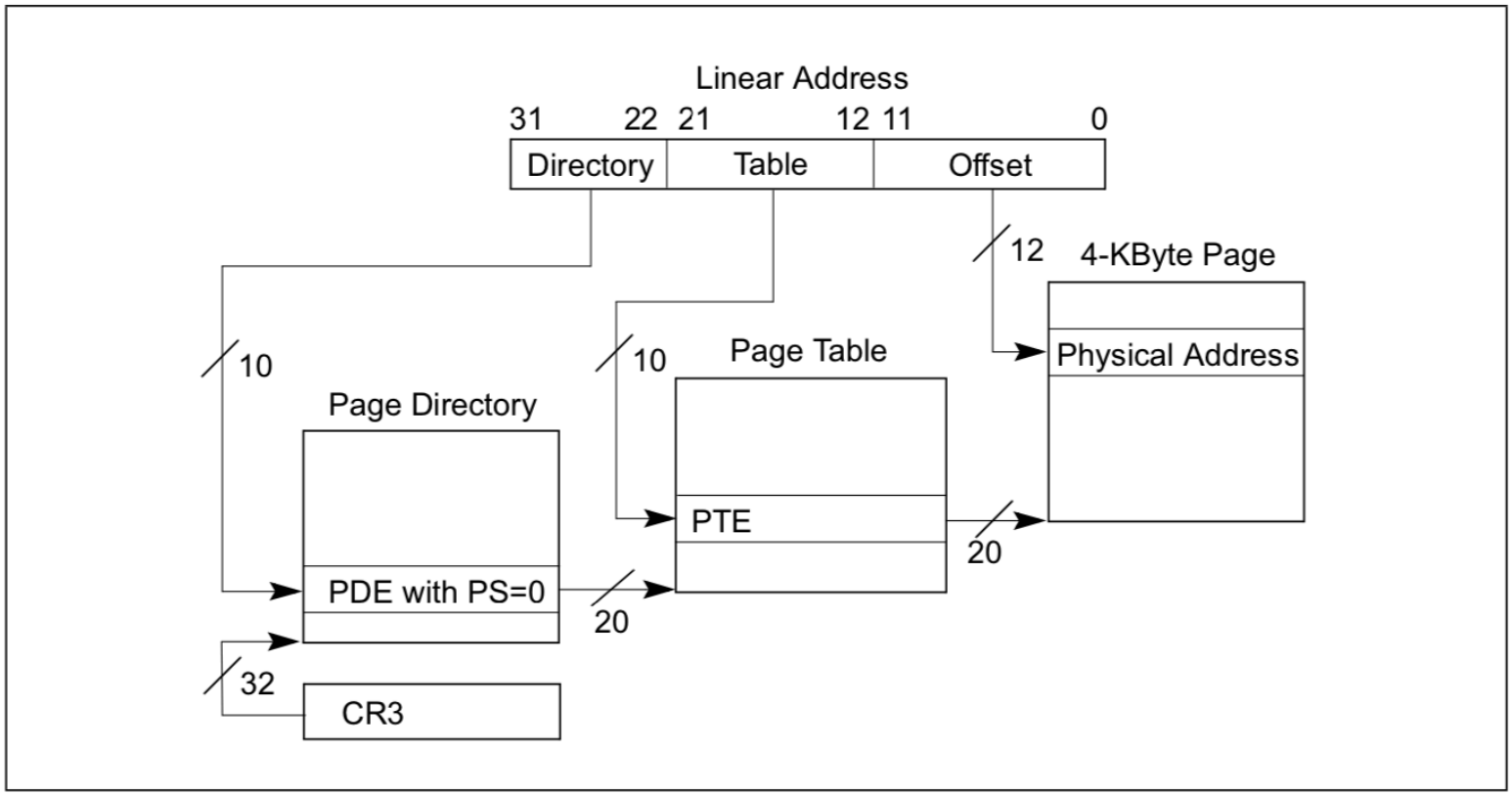


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

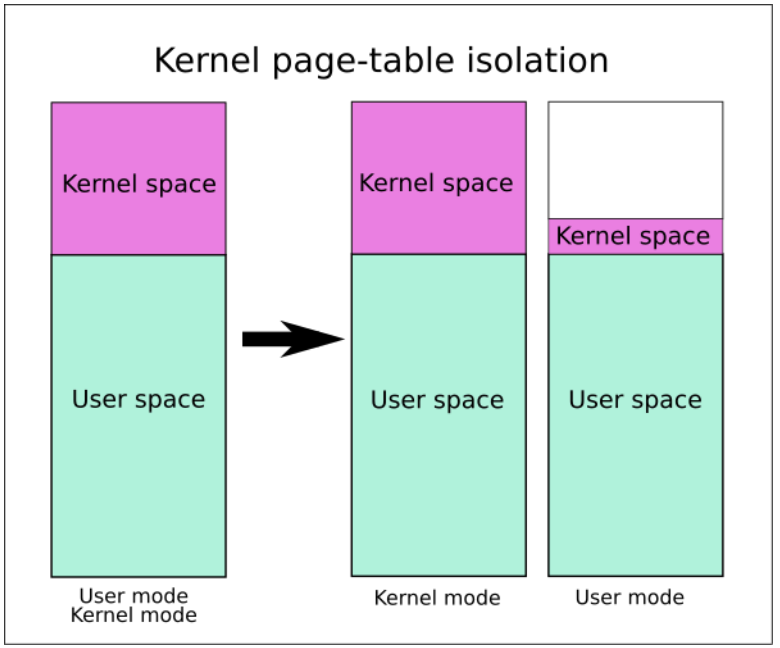
- CR3 provides physical address of the "Page Directory", i.e., first level of 2-level translation

Page Table Discussion

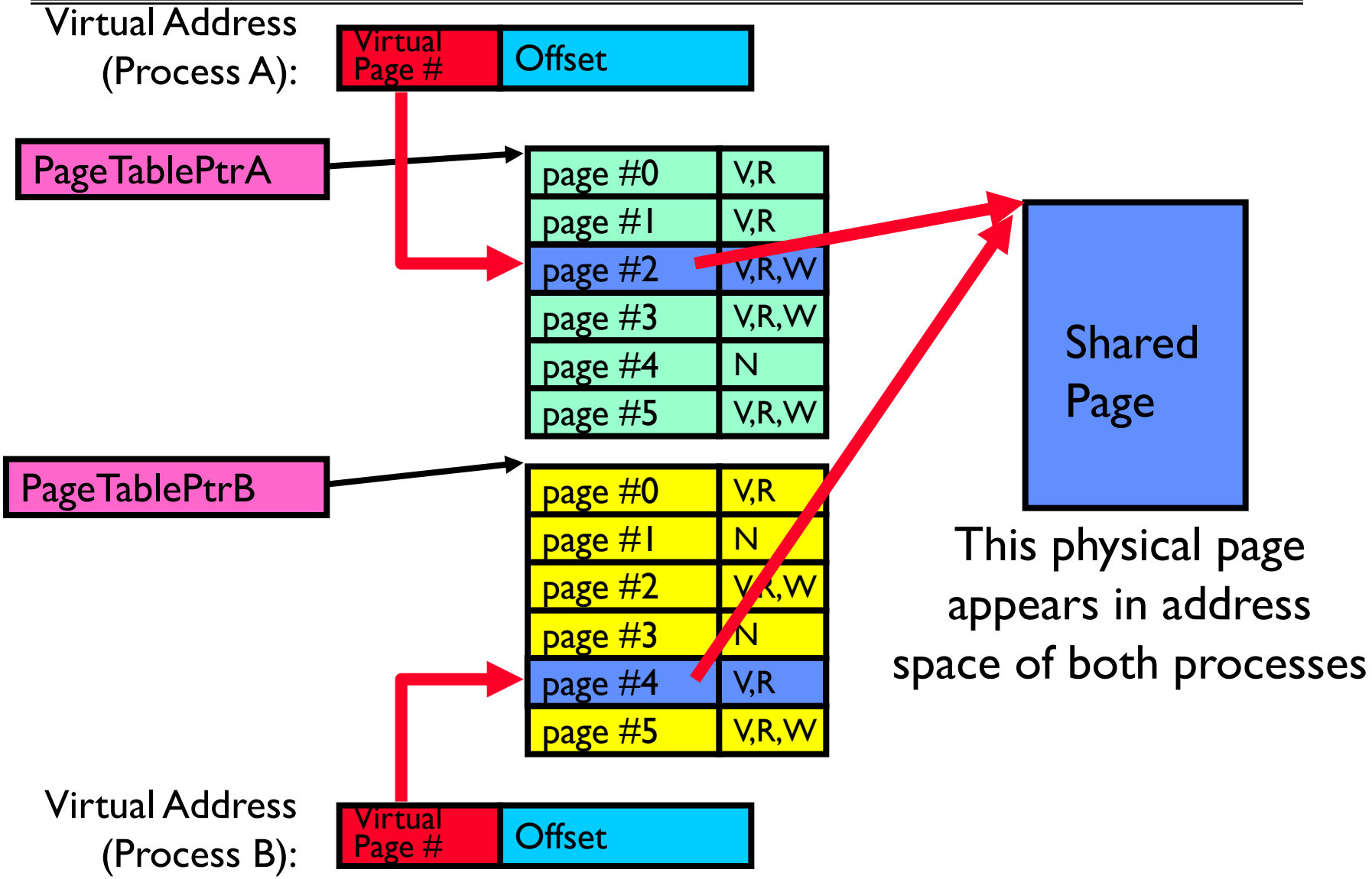
- What provides protection?
- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros: Simple memory allocation & management, Simple uniform process address space?
  - Con: What if address space is sparse?
    - » e.g., on UNIX, code starts at 0, stack starts at $(2^{31}-1)$
    - » With 1K pages, need 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
- Could processes constructively share memory?
- How big is the page table? Does it all need to be in memory?
  - How about multi-level paging
  - or combining paging and segmentation?

Simple mechanism – Profound Implications

- Address Space Randomization
  - Position-Independent Code => can place user code region anywhere in the address space
    - » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don't map whole kernel space into each process, switch to kernel page table



What about Sharing?



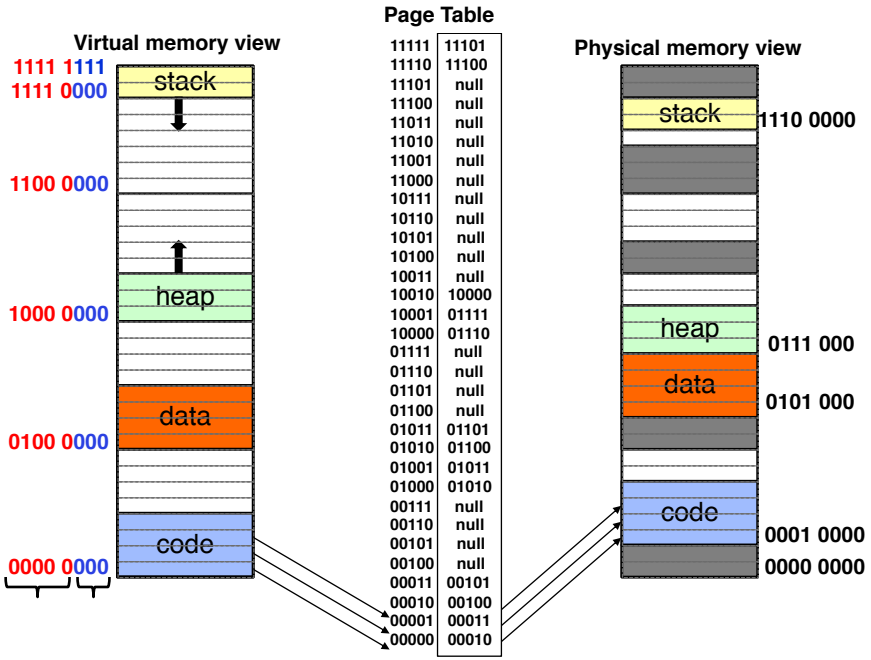
Where is page sharing used ?

- The "kernel region" of every process has the same page table entries
- The process cannot access it at user level
- But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
  - What does the kernel need to do to access other user processes?

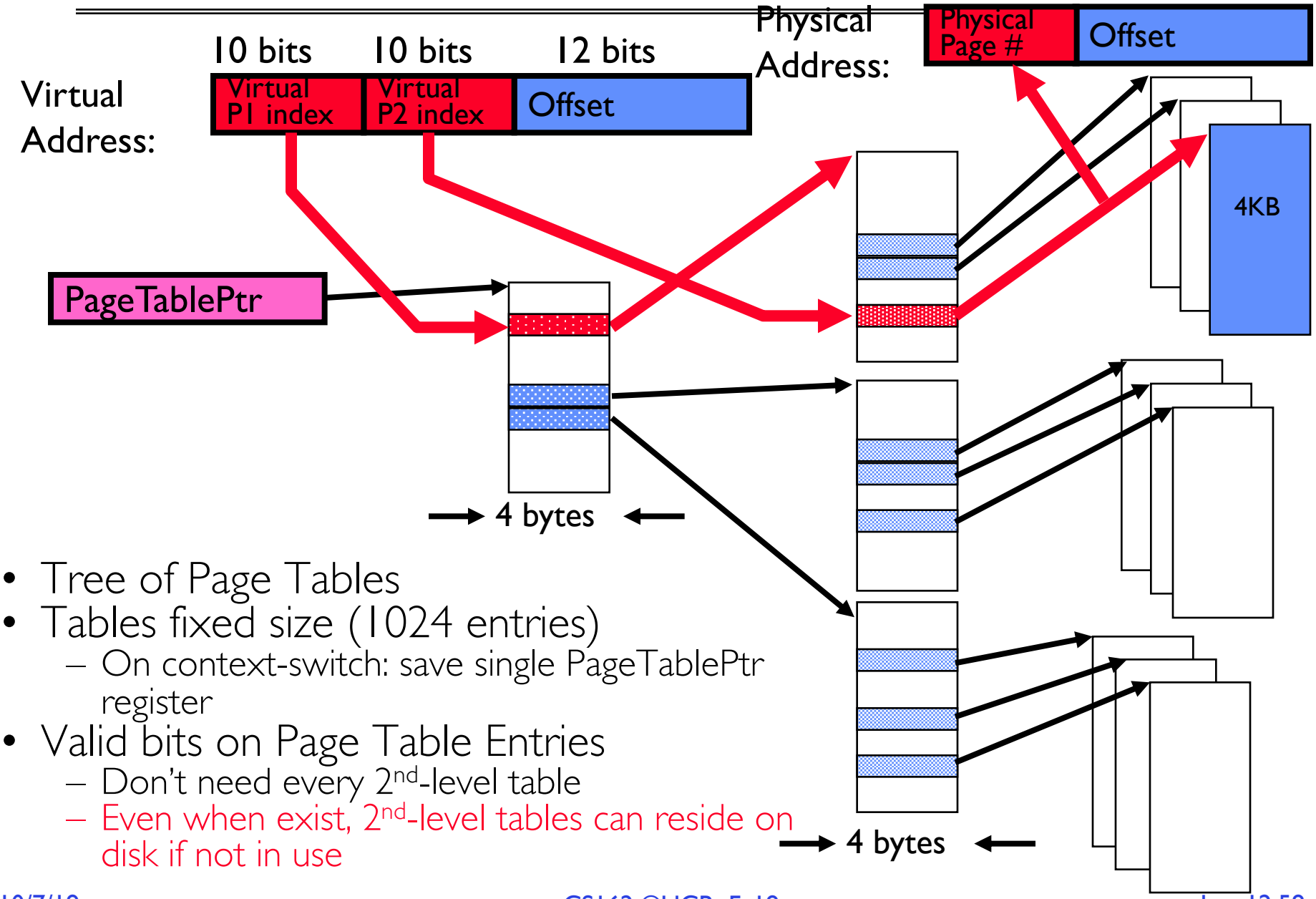
- User-level system libraries (execute only)

How big do things get?

- 32-bit address space => 2^{32} bytes (4 GB)
- Typical page size: 4 kb
  - how many bits of the address is that ? (hint 2^{10} = 1024)
- So how many entries in the page table of *each* process?
  - 2^{20} (that's about a million) x 4 bytes each => 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386) 16 mb was a LOT of memory
- **It has huge holes in it.**

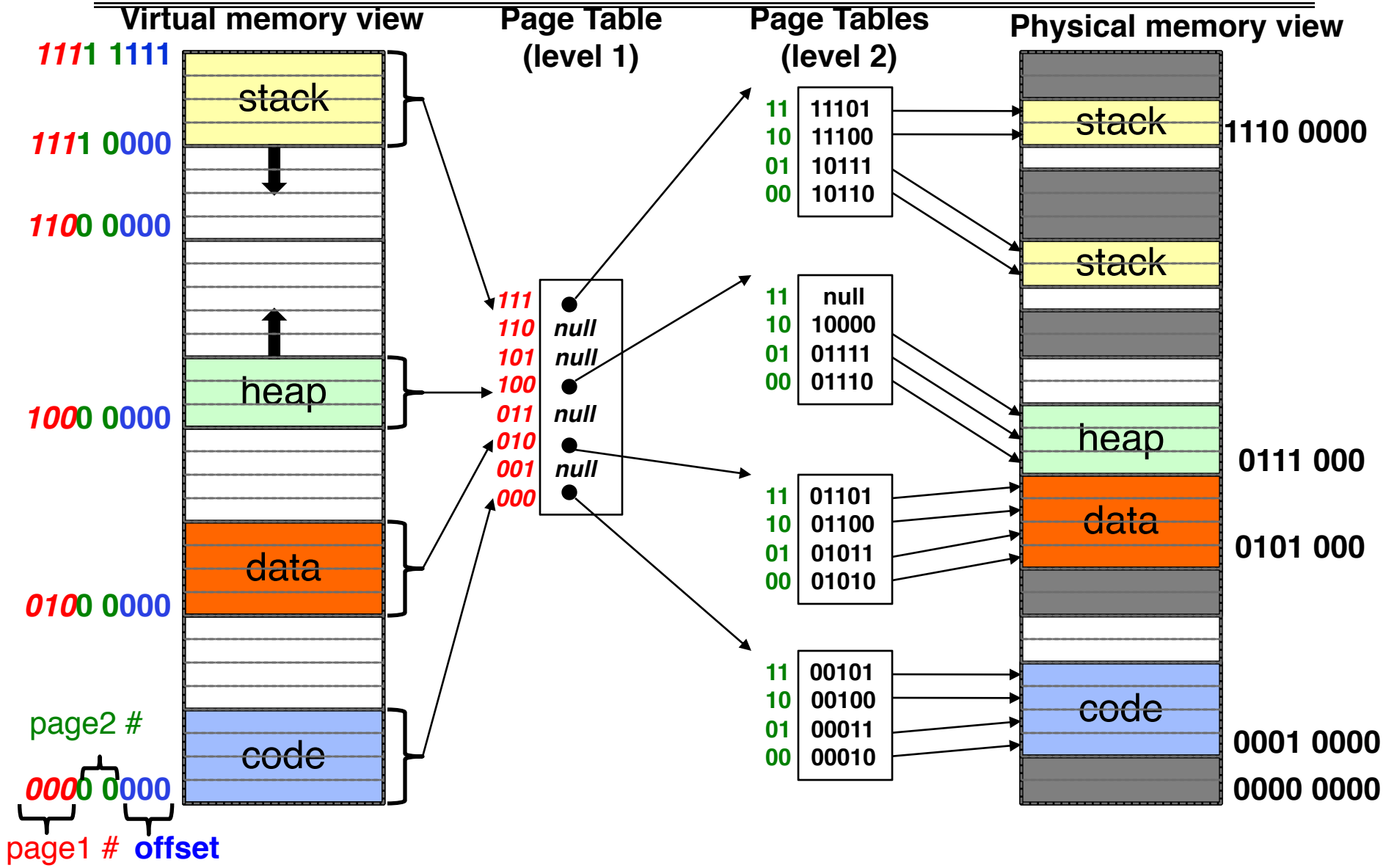


For sparse address space: two-level page table

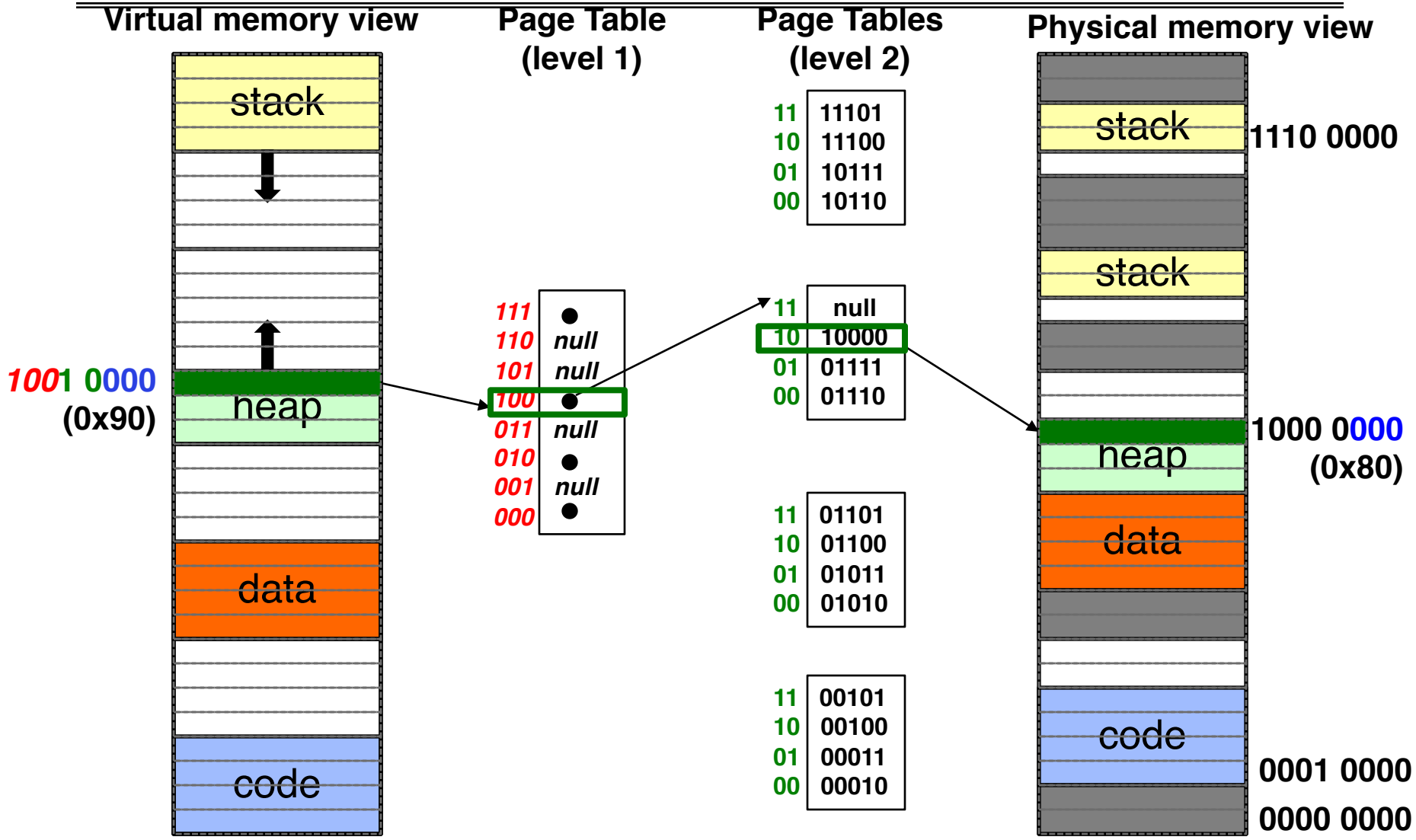


- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

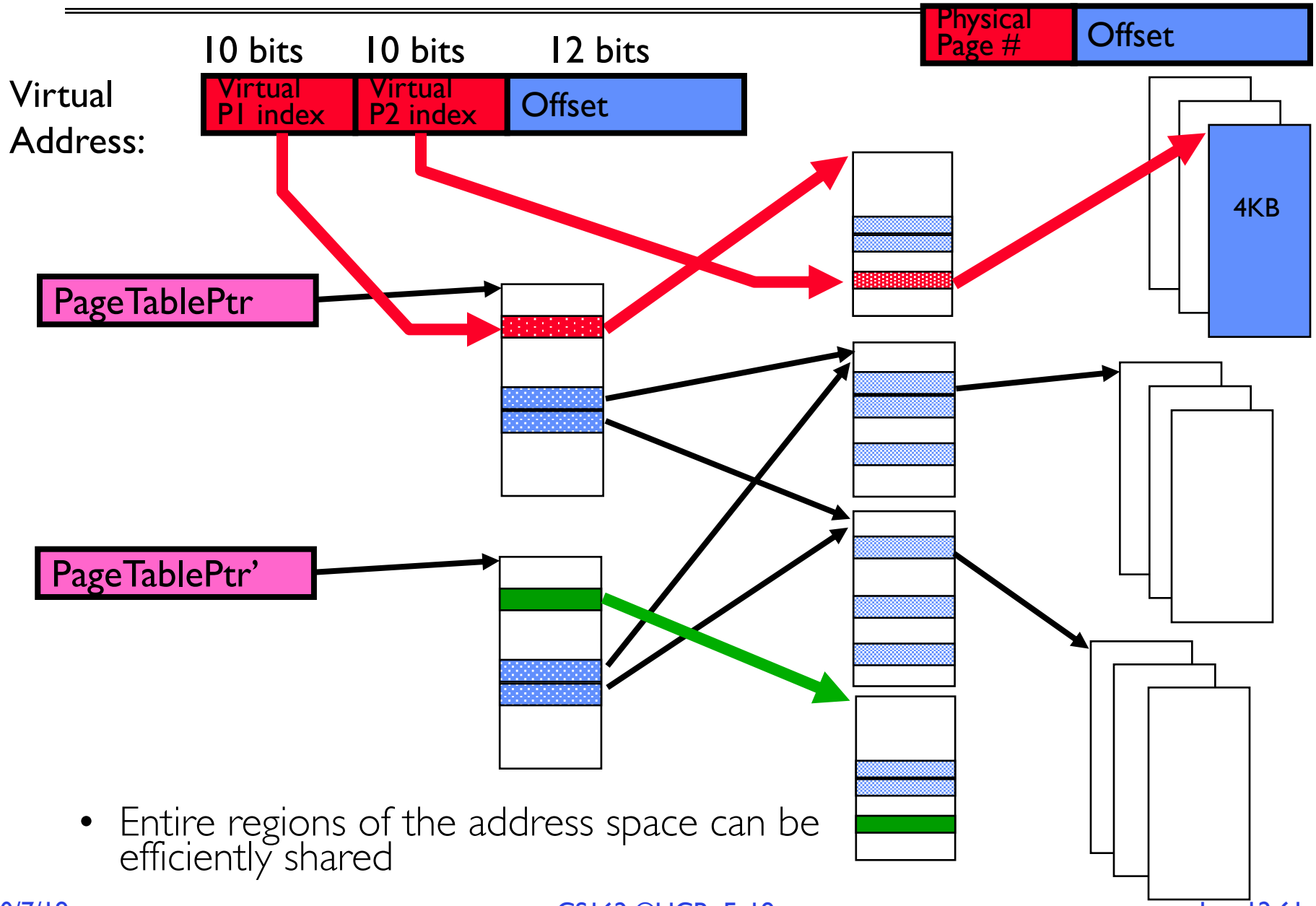
Summary: Two-Level Paging



Summary: Two-Level Paging



Sharing with multilevel page tables



- Entire regions of the address space can be efficiently shared

What about a large address space

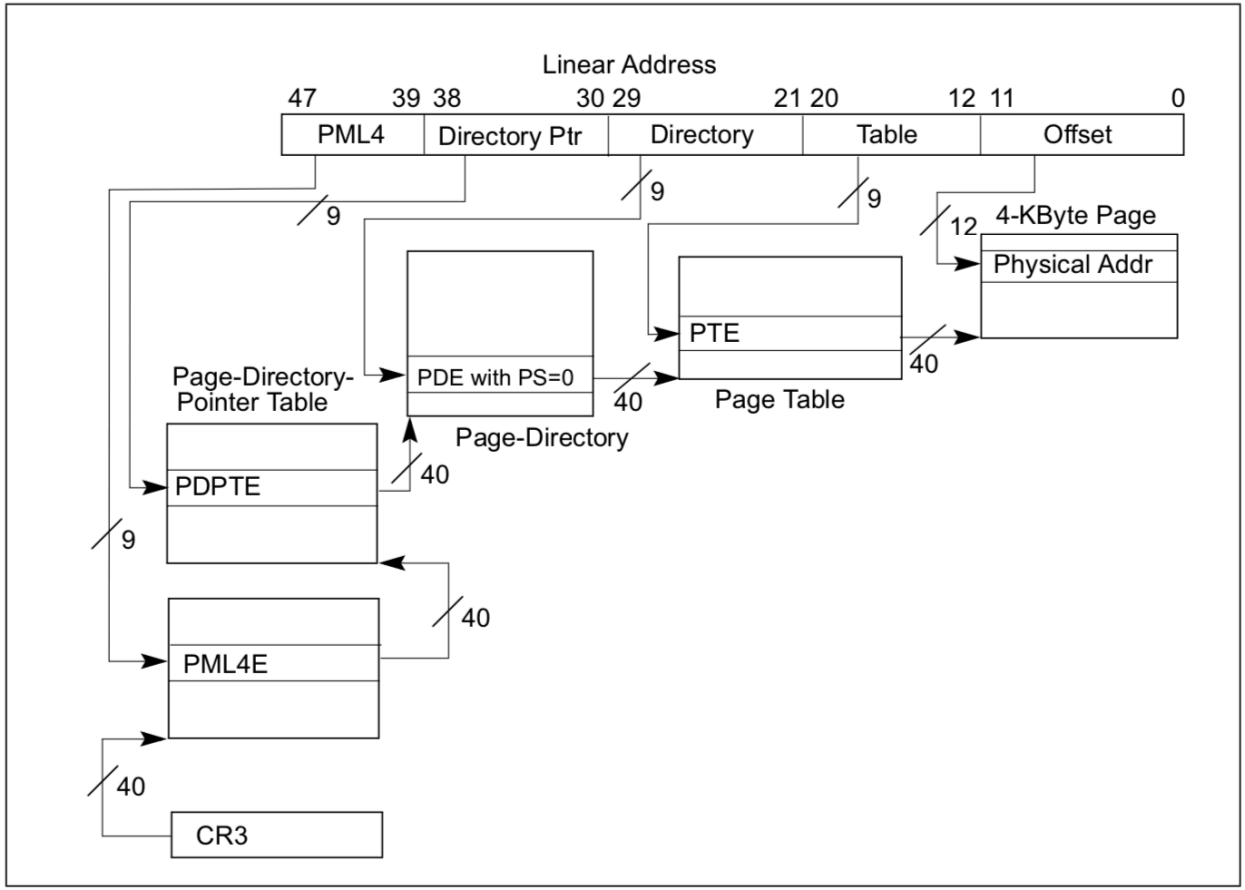


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

- All current x86 processor support a 64 bit operation
- 64-bit words (so ints are 8 bytes) but 48-bit addresses

What about a large address space

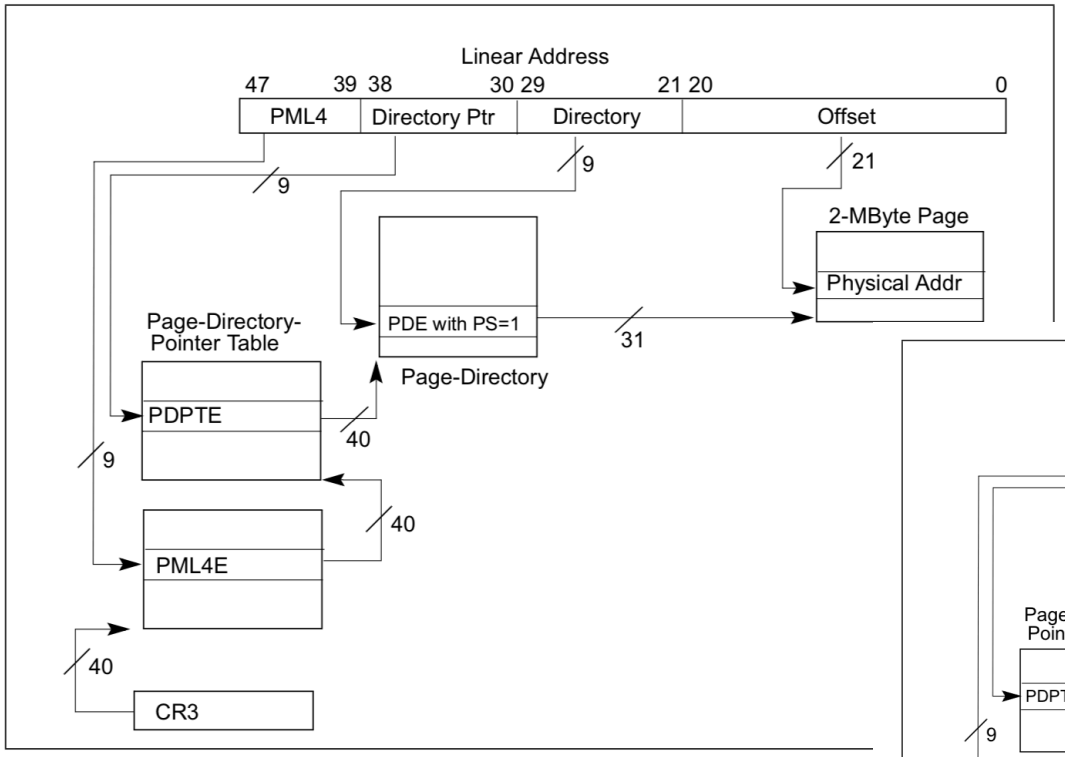


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-L

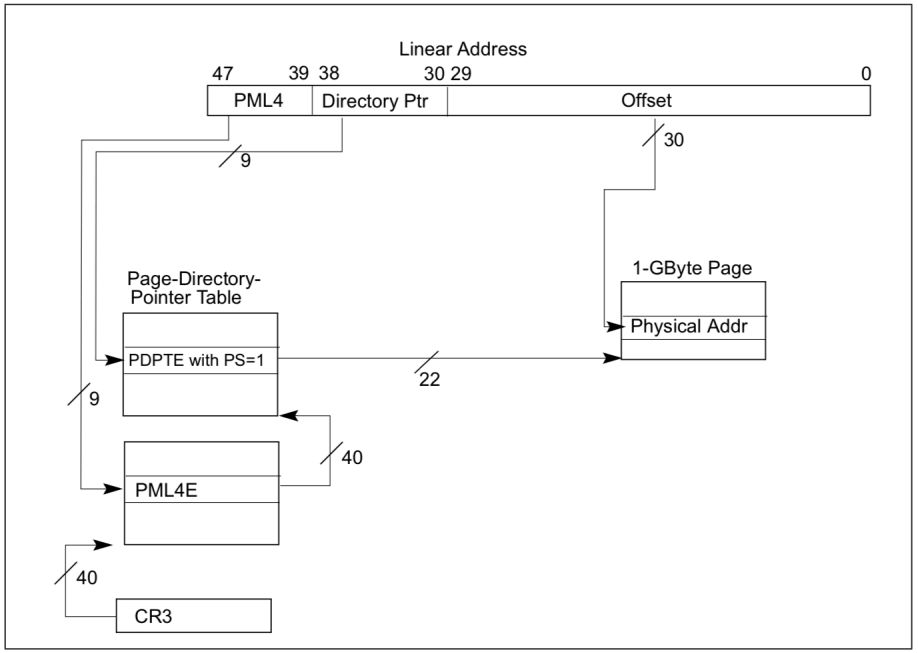


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Or larger page sizes, memory is now cheap

That MMU

- For every memory reference – instruction fetch, load or store:
  - the virtual address is presented to the MMU
  - It indexes through as many levels of page table entries as necessary to translate it to a physical address
  - At any stage, it may encounter a "Fault" if the page (or page table page) is not resident or not accessible
- Tricky to make this go extremely fast
  - But that's all in hardware

Multi-level Translation Analysis

- Pros:
  - Allocate (roughly) just page table entries needed for process
    - » Dense regions (code, static, heap, stack) with holes between
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - Minimum unit of page table allocation is a page
    - » 1024 entries x 4 KB => 4 MB region of address space
  - Two (or more) lookups per reference
    - » Seems very expensive!
    - » This is where the MMU gets fancy

So what needs to be in that PCB?

- Pointer to the page table for the process
- Thread Structures (regs, stacks, scheduling)
- File descriptors for open files
  - Buffers?
- Kernel lock objects
- Other scheduling data

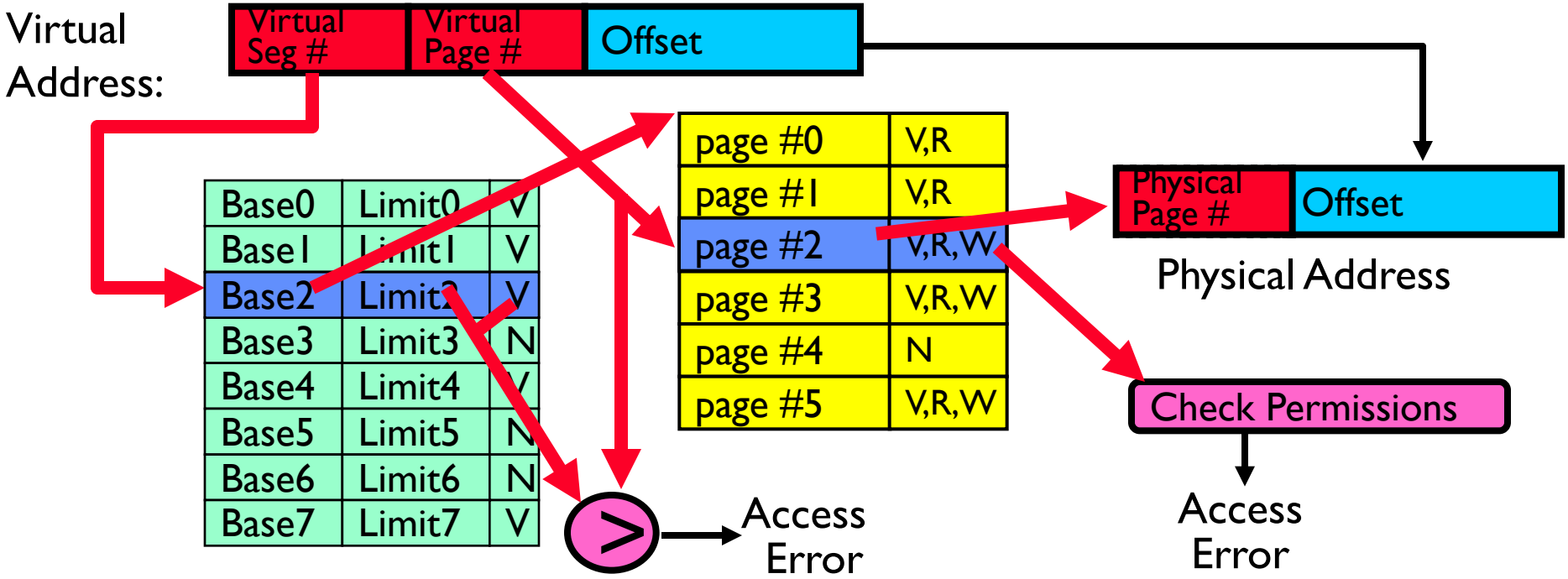
Summary

- Segment Mapping
  - Segment registers within processor
  - Segment ID associated with each access
    - » Can come from fields in instruction instead (x86)
  - Each segment contains base and limit information
    - » Offset (rest of address) adjusted by adding base
  - Closely approximate by sparse regions of flat virtual address space
- Paged Virtual Address Space & Page Tables
  - Address space 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 Address space => Large page tables
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space

Extras

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)

What about Sharing (Complete Segment)?