

CS162
Operating Systems and
Systems Programming
Lecture 15

Memory 2: Segments, Page Tables,
Multi-Level Page Tables

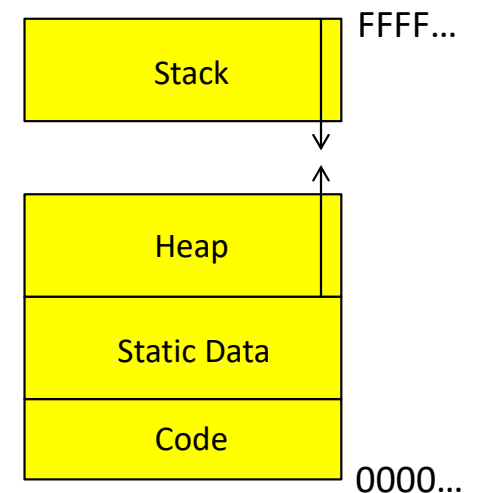
March 12th, 2026

Prof. John Kubiatowicz

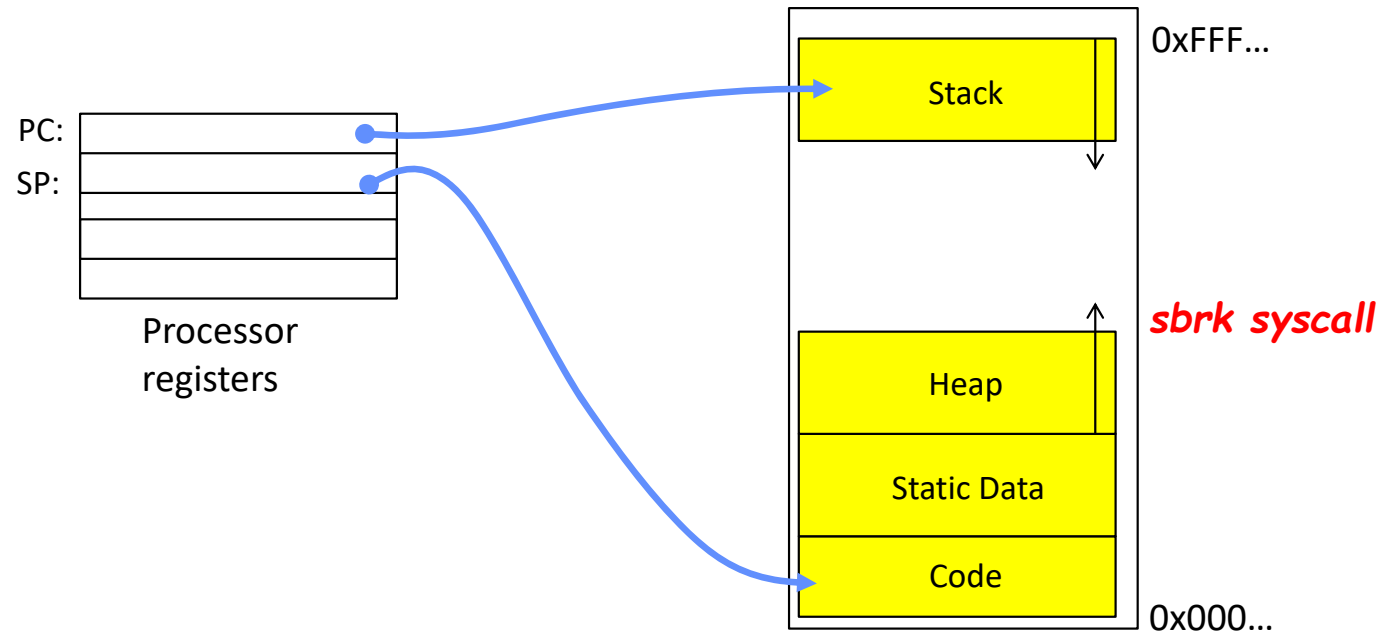
<http://cs162.eecs.Berkeley.edu>

Recall: Address Space, Process Virtual Address Space

- Definition: Set of accessible addresses and the state associated with them
 - $2^{32} = \sim 4$ billion *bytes* on a 32-bit machine
- How many 32-bit numbers fit in this address space?
 - 32-bits = 4 bytes, so $2^{32}/4 = 2^{30} = \sim 1$ billion
- 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
 - ...

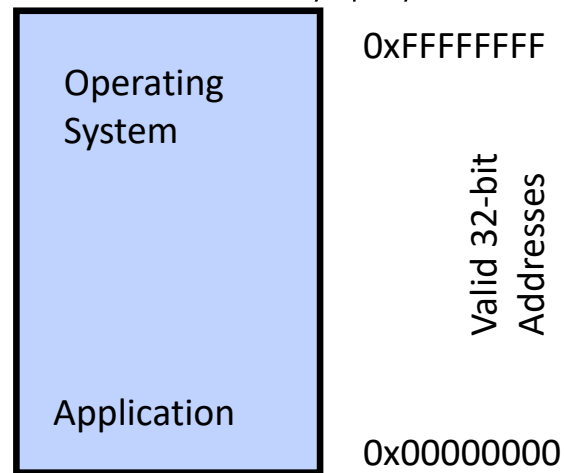


Recall: Process Address Space: typical structure



Recall: Uniprogramming

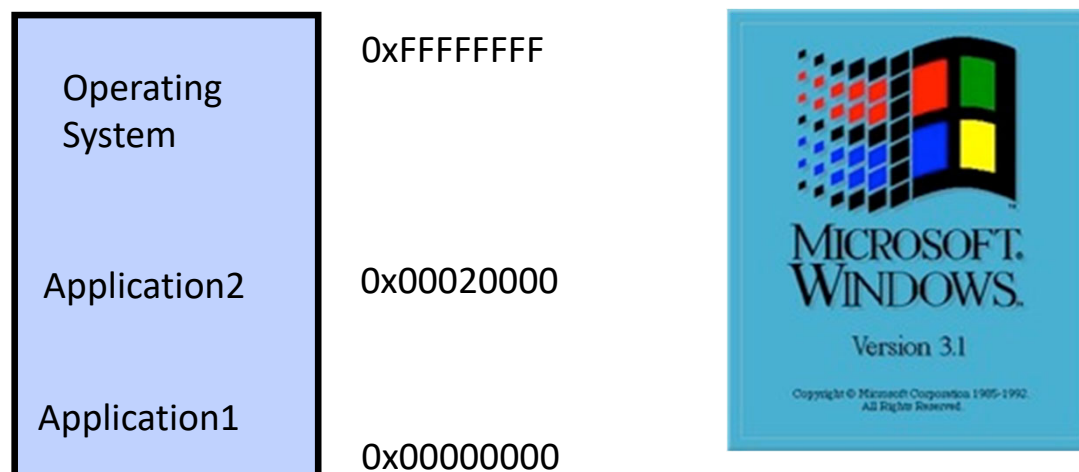
- Uniprogramming (no Translation or Protection)
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address



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

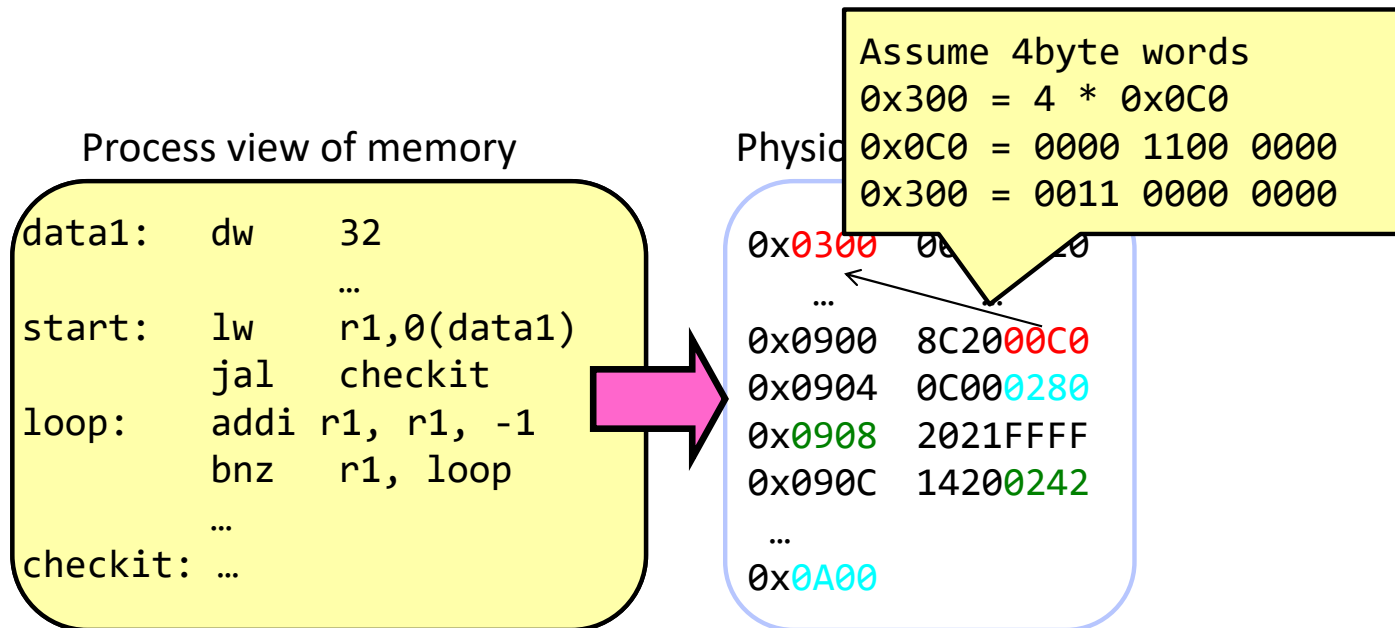
Primitive Multiprogramming

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

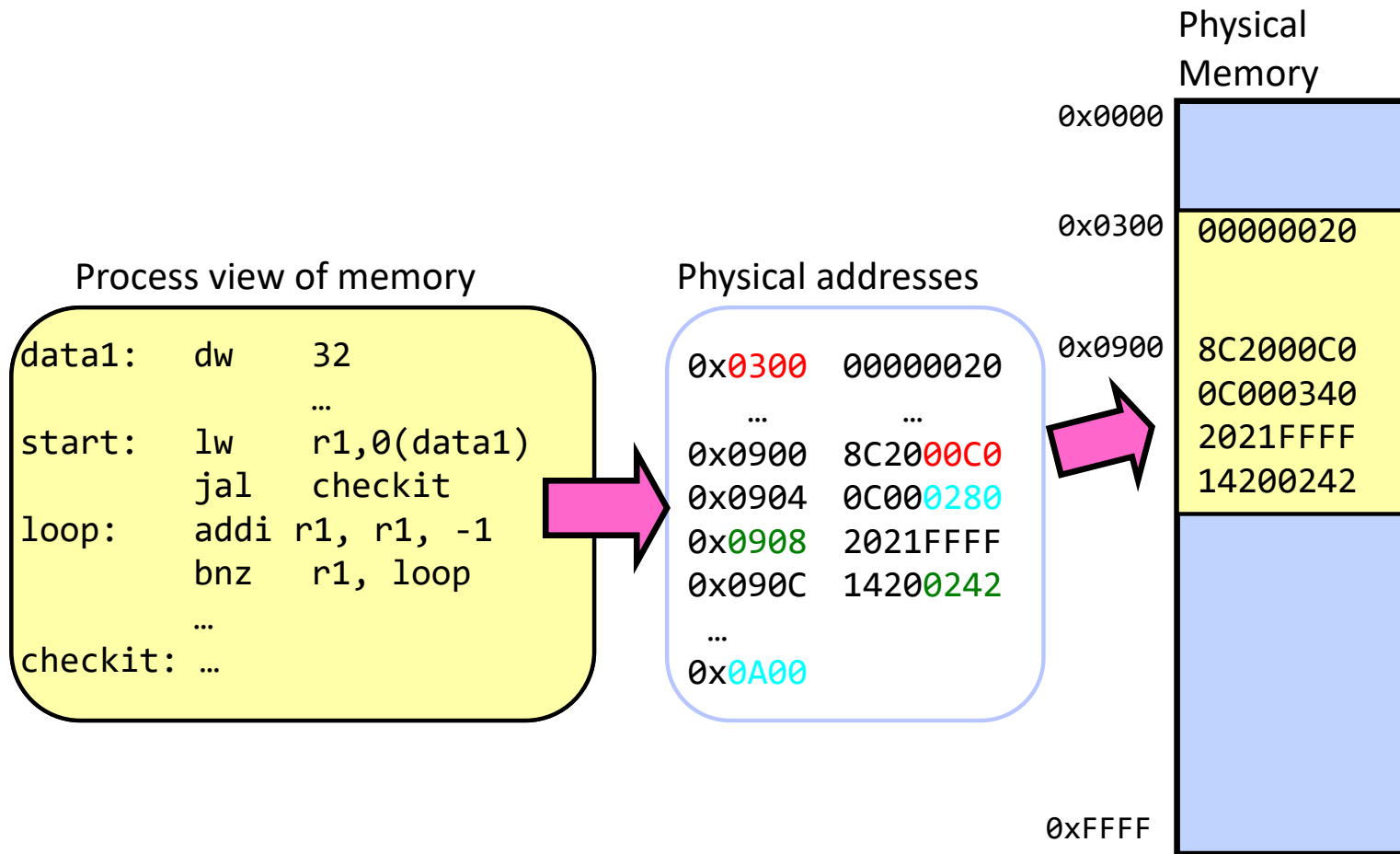


- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - » Everything adjusted to memory location of program
 - » Translation done by a linker-loader (relocation)
 - » Common in early days (... till Windows 3.x, 95?)
- With this solution, no protection: bugs in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

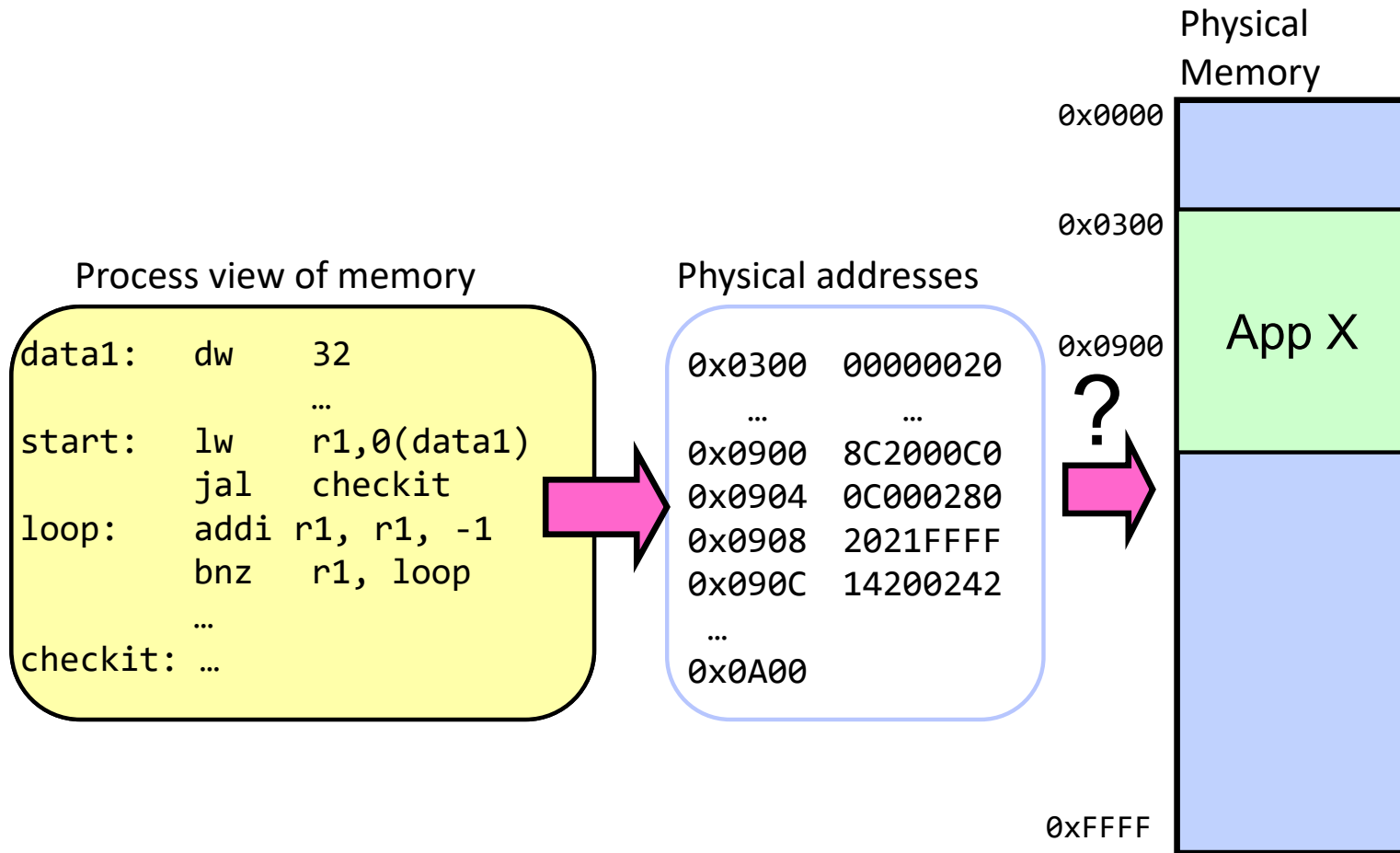
Binding of Instructions and Data to Memory



Binding of Instructions and Data to Memory

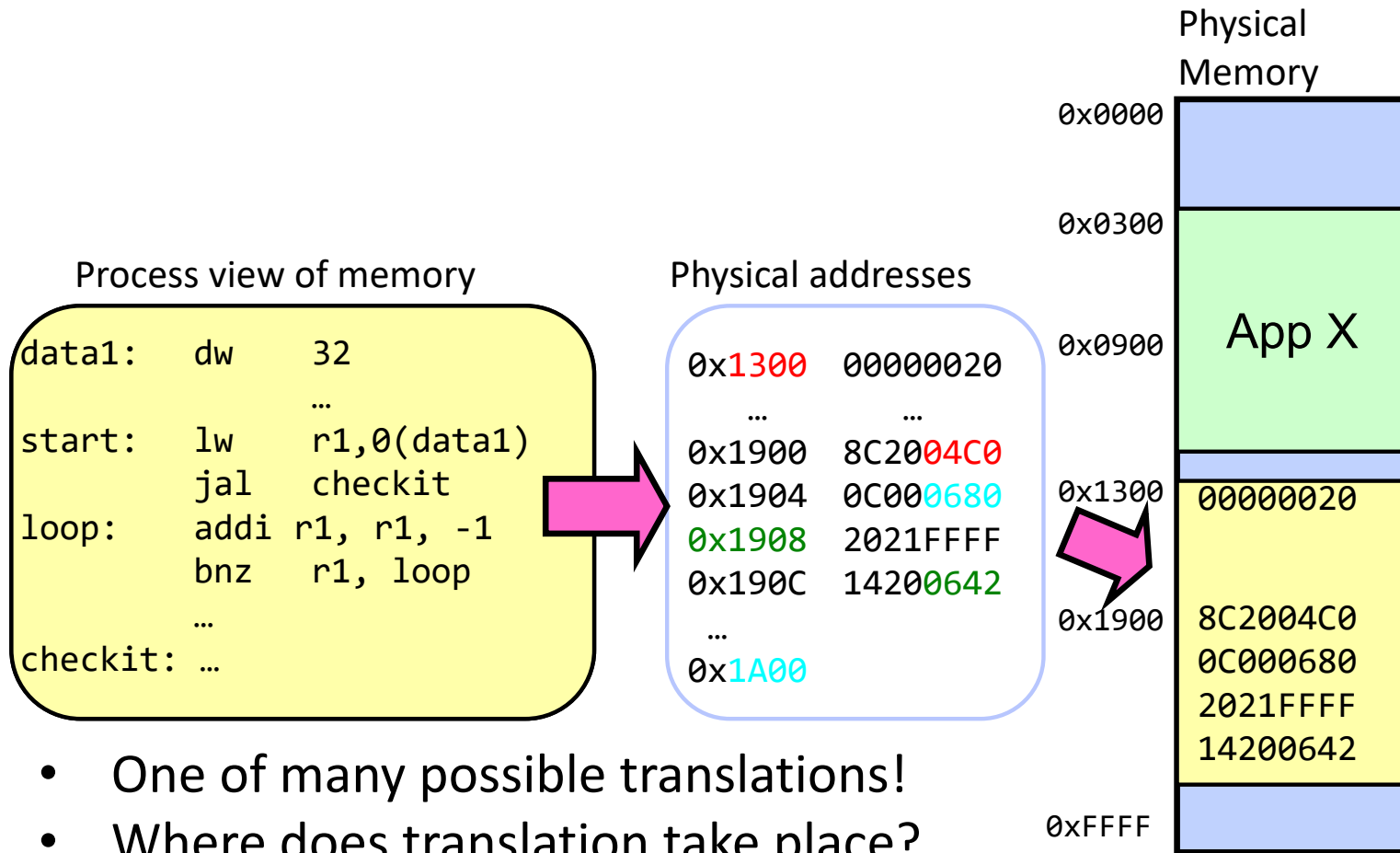


Second copy of program from previous example



Need address translation!

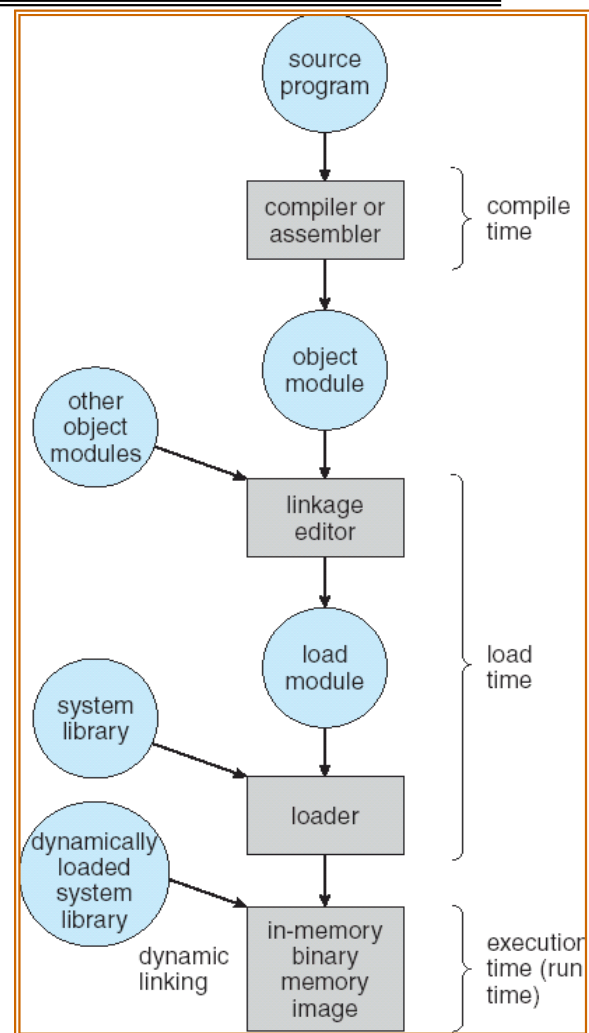
Second copy of program from previous example



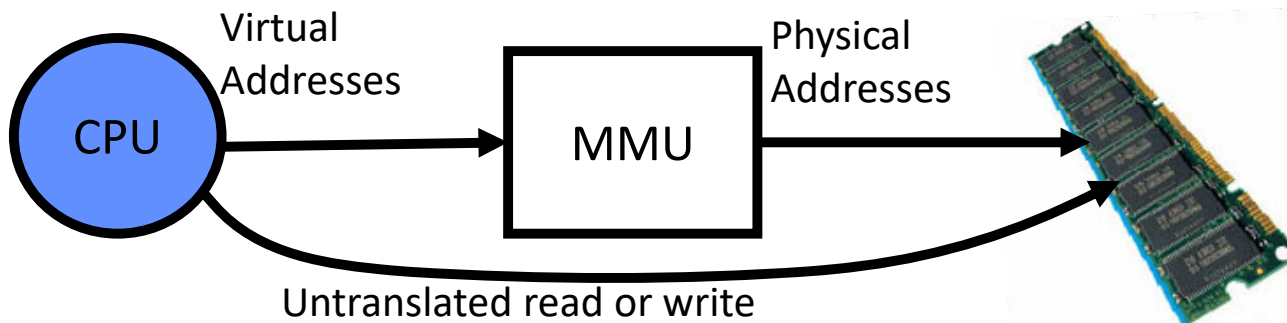
- One of many possible translations!
- Where does translation take place?
Compile time, Link/Load time, or Execution time?

From Program to Process

- 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



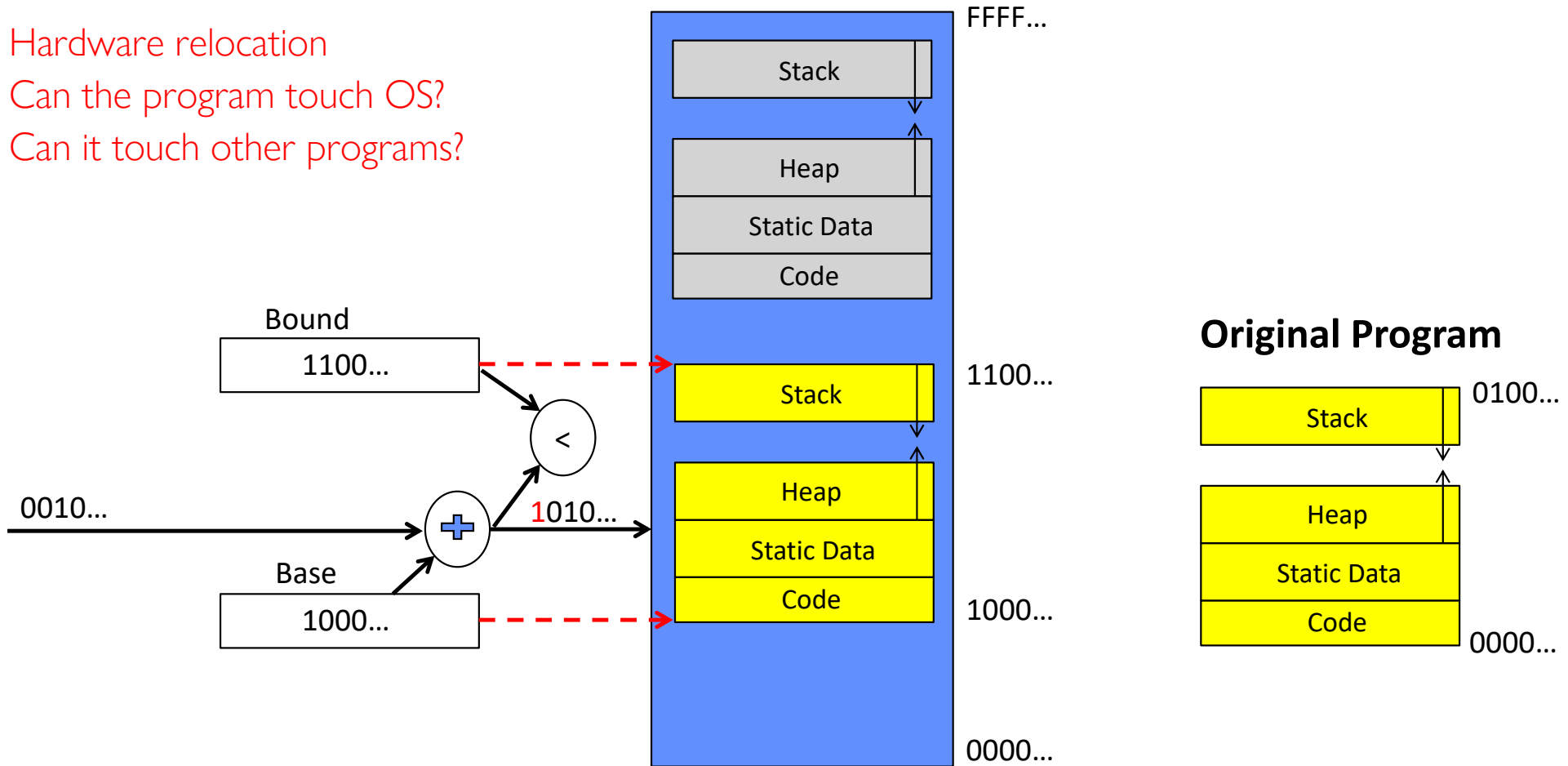
Recall: General Address translation



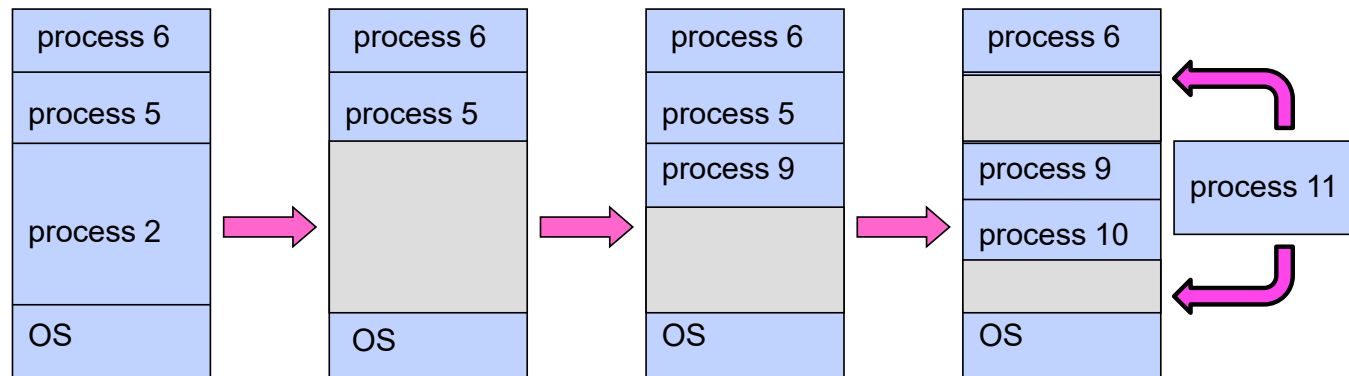
- Consequently, two views of memory:
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
 - **Translation box** (Memory Management Unit or MMU) converts between two views
- **Translation \Rightarrow much easier to implement protection!**
 - If task A cannot 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

Recall: Simple address translation with Base and Bound

- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

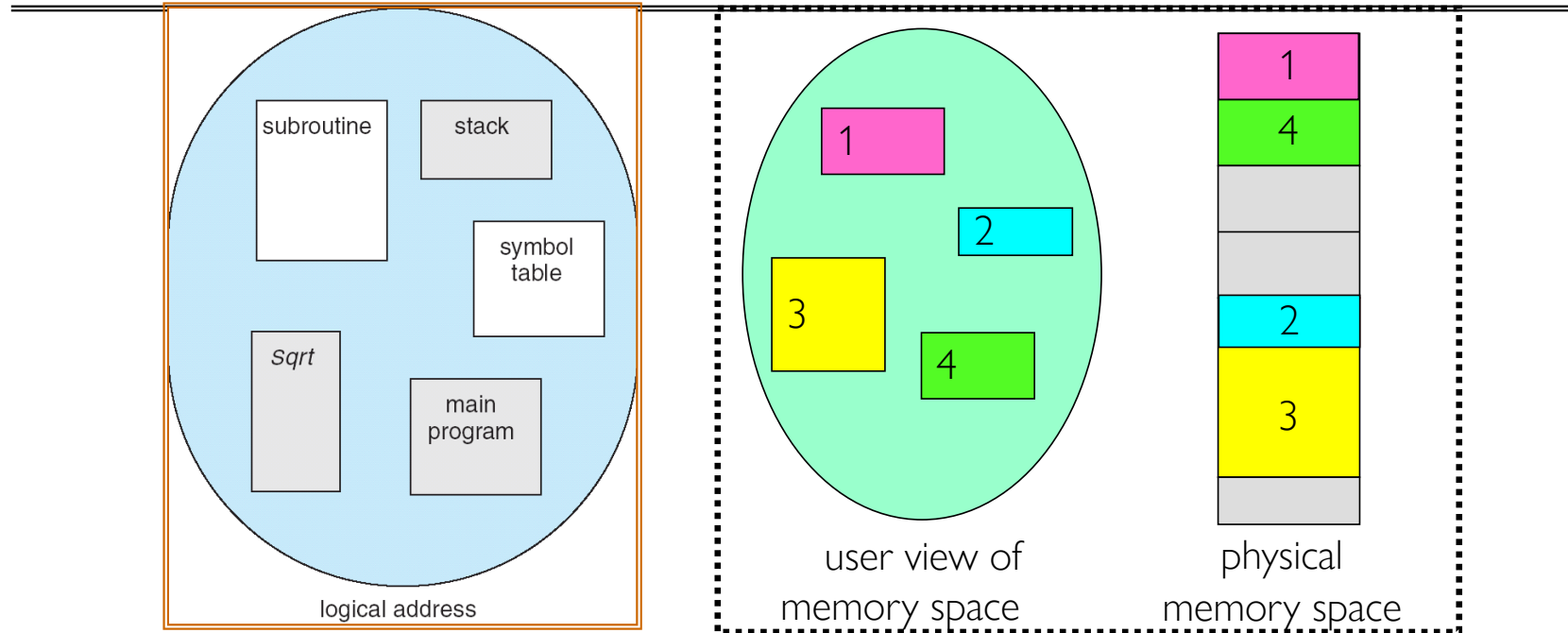


Issues with Simple B&B Method



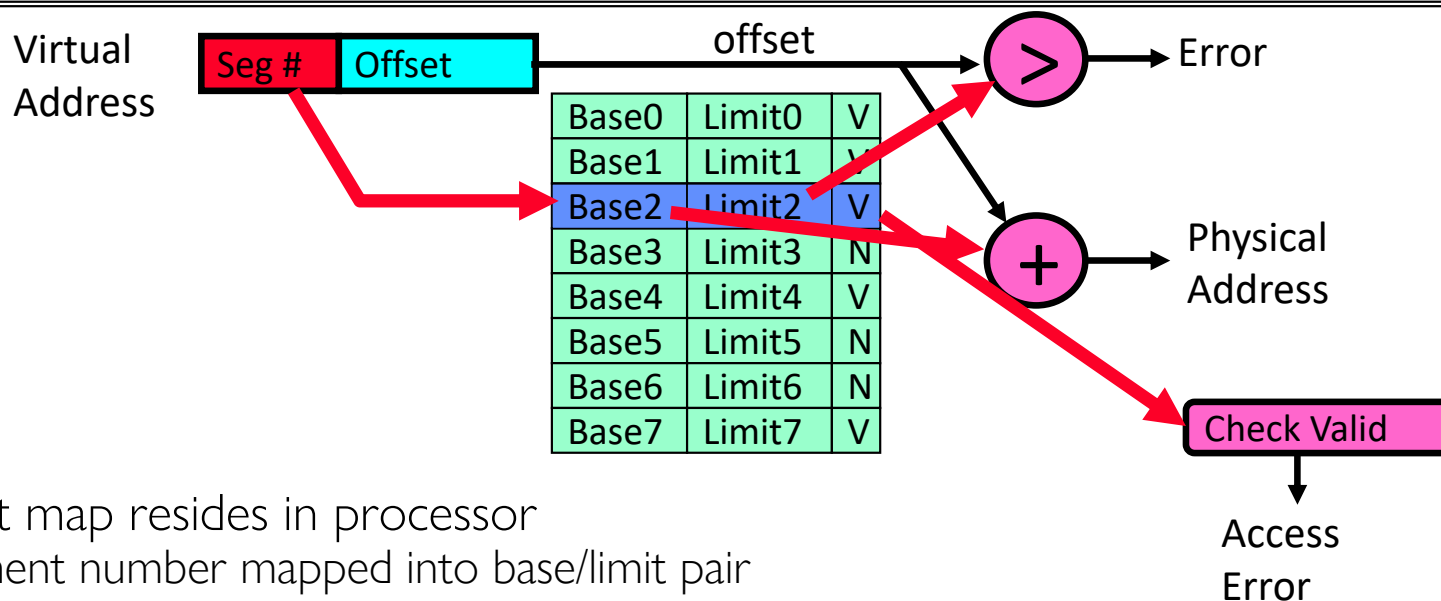
- Fragmentation problem over time
 - Not every process is same size \Rightarrow memory becomes fragmented over time
 - Fragmentation: wasted space both *external* (between blocks) and *internal* (inside blocks)
- Missing support for sparse address space
 - Would like to have multiple chunks/program (Code, 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



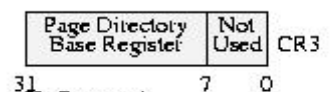
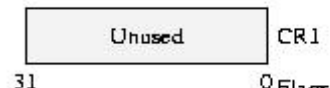
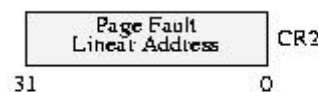
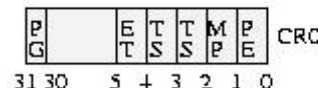
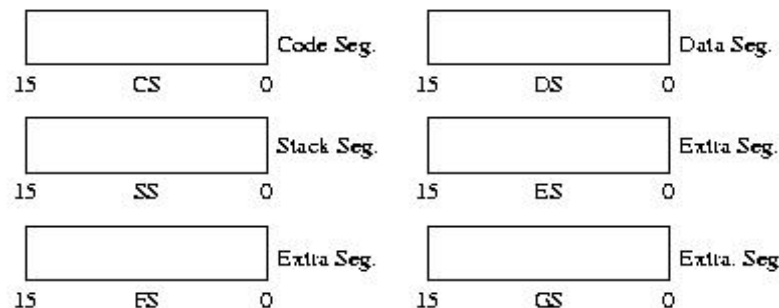
RPL = Requestor Privilege Level
 TI = Table Indicator
 (0 = GDT, 1 = LDT)
 Index = Index into table

Protected Mode segment selector



80386 Special Registers

Segment registers



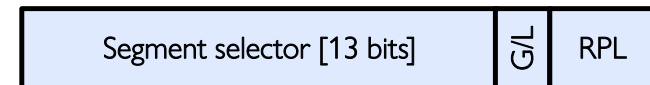
PG=Paging Enable
 ET=Emulation Type
 TS=Task Switched
 EM=Emulate Coprocessor
 MP=Math coprocessor present
 PE=Protected Mode enable

X=Reserved
 NT=Nested Task
 IOPL=I/O Privilege Level
 OF=Overflow Flag
 DF=Direction Flag
 IF=Interrupt Flag
 TF=Trap Flag
 SF=Sign Flag
 ZF=Zero Flag
 AF=Auxiliary Flag
 PF=Parity Flag
 CF=Carry Flag

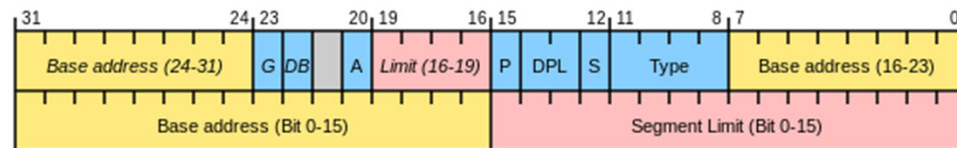
- Typical Segment Register
 - Current Priority is RPL of Code Segment (CS)
 - Index points into table (GDT or LDT) with actual segment descriptor
- Segmentation can't be just "turned off"
 - What if we just want to use paging?
 - Set base and bound to all of memory, in all segments

X86 Segment Descriptors (32-bit Protected Mode)

- Segments are implicit in the instruction (e.g. code segments) or 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)
 - RPL: Requestor's Privilege Level (**RPL of CS \Rightarrow Current Privilege Level**)
- Two registers: GDTR/LDTR hold pointers to global/local descriptor tables in memory
 - Descriptor format (64 bits):

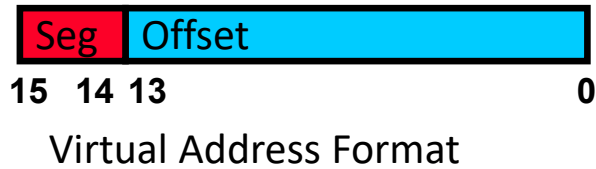


Segment Register

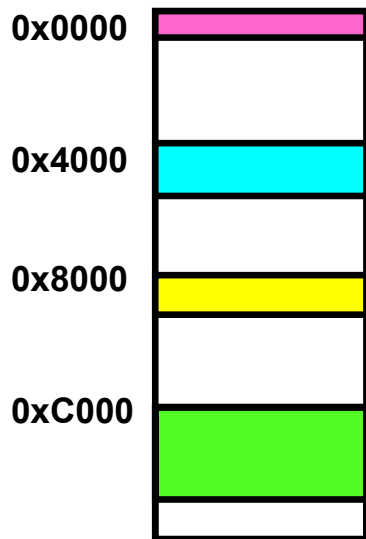


- G: Granularity of segment [Limit Size] (0: bytes, 1: 4KiB unit)
- DB: Default operand size (0: 16bit, 1: 32bit)
- A: Programmer definable (no hardware meaning)
- P: Segment present
- DPL: Descriptor Privilege Level: Access requires $\text{Max}(\text{CPL}, \text{RPL}) \leq \text{DPL}$
- S: System Segment (0: System, 1: code or data)
- Type: Code, Data, Segment

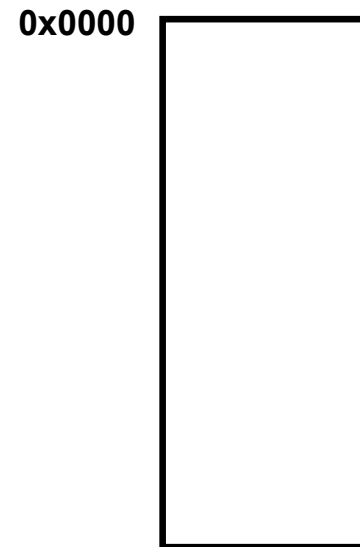
Example: Four Segments (16 bit addresses)



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

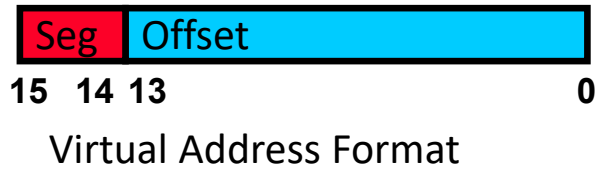


Virtual
Address Space

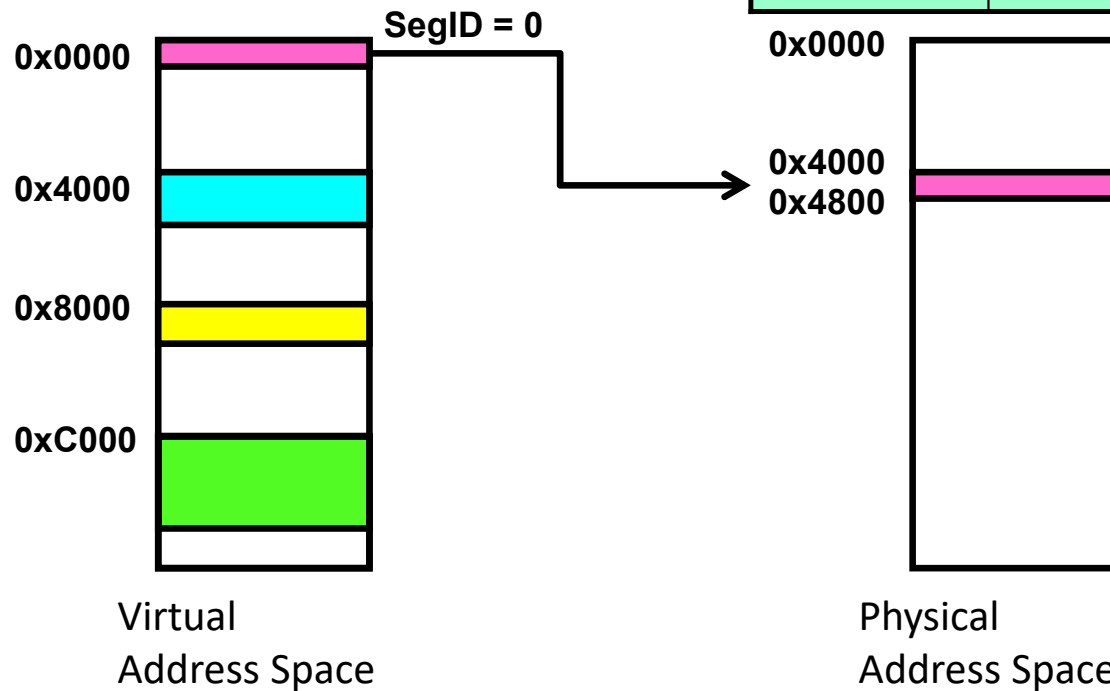


Physical
Address Space

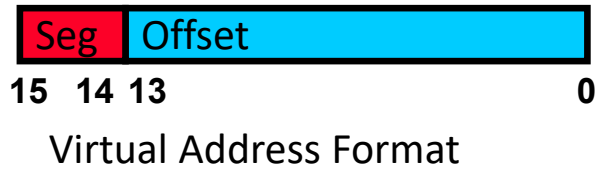
Example: Four Segments (16 bit addresses)



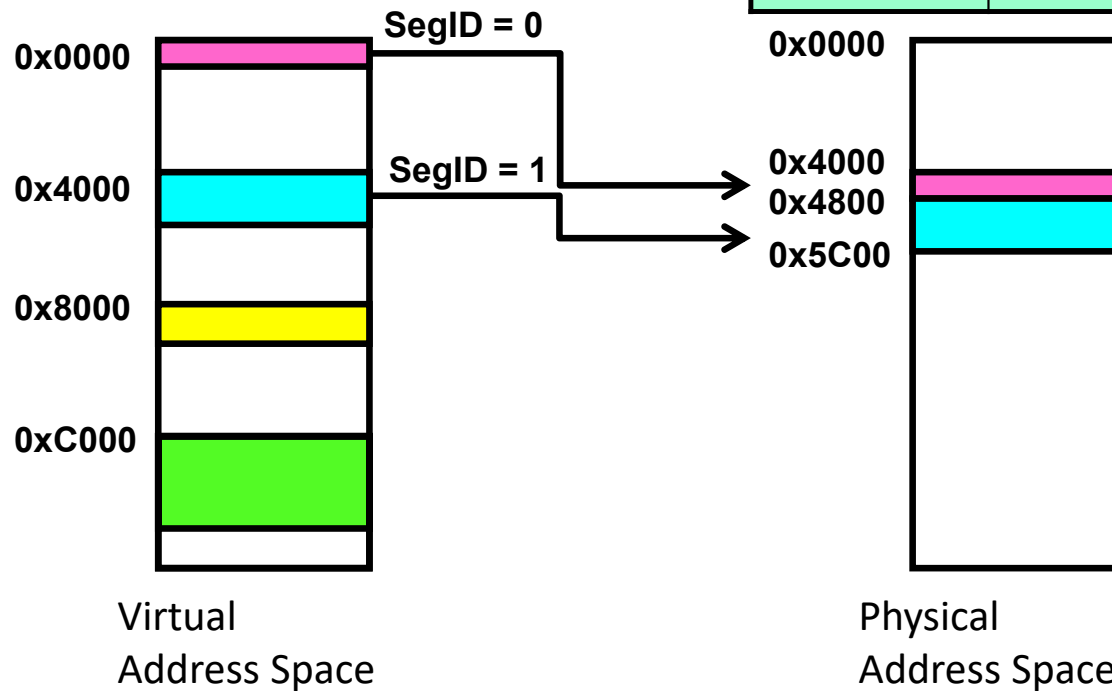
Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



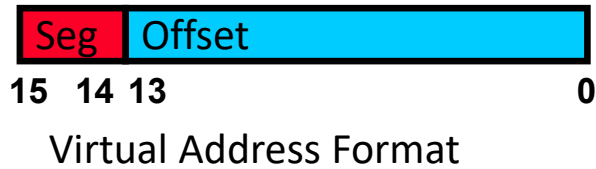
Example: Four Segments (16 bit addresses)



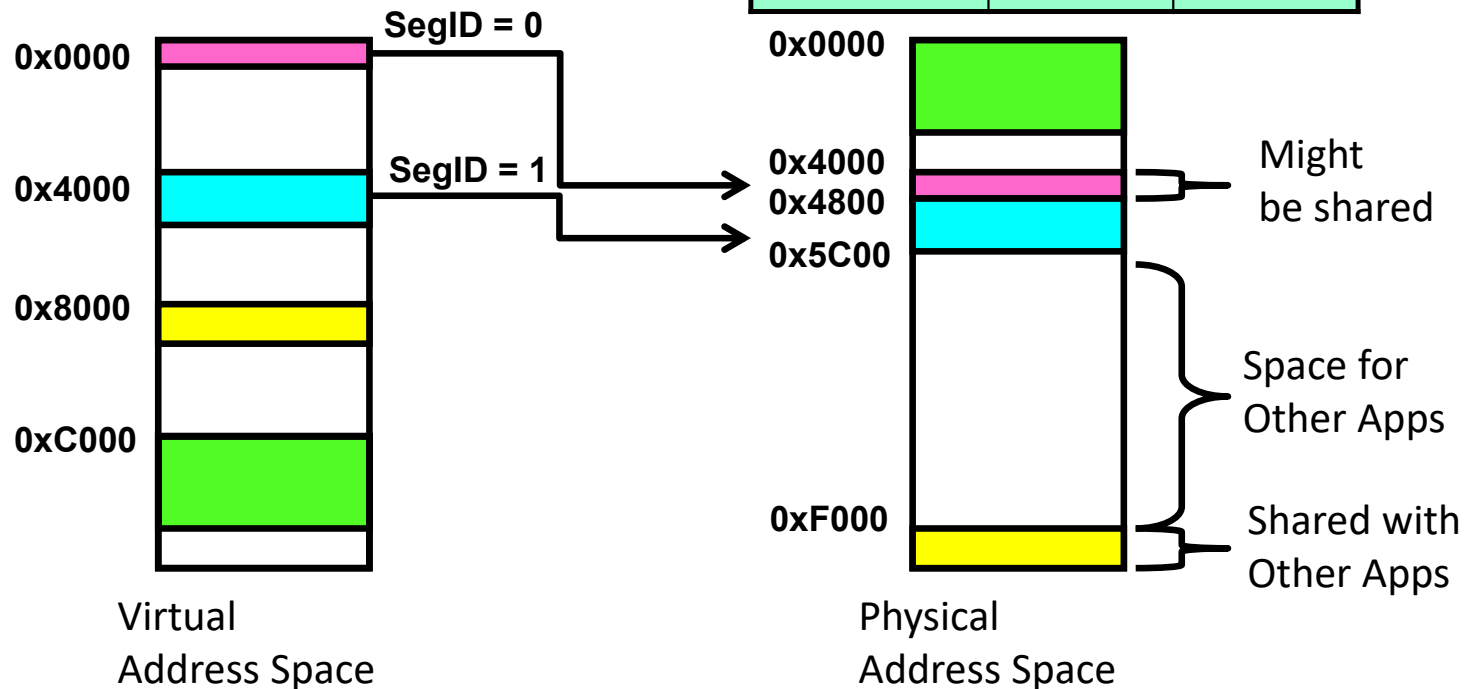
Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



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



Administrivia

- Midterm 2: Tuesday 3/31 from 7-10PM
 - A week from today!!!
 - All material up to Lecture 17 technically in bounds
 - Closed book: with two double-sided handwritten sheets of notes
- Project 2 design document due this Saturday.
- Starting next week – will have an opportunity to get extra credit participation points by attending lecture
 - Details to follow

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=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)

```
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=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 0x0244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → 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=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 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

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

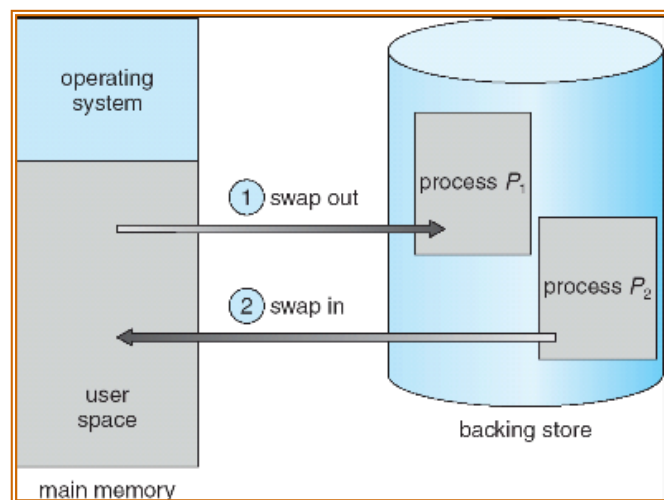
Let's simulate a bit of this code to see what happens (PC=0x0240):

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
- 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)
- 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 in memory?

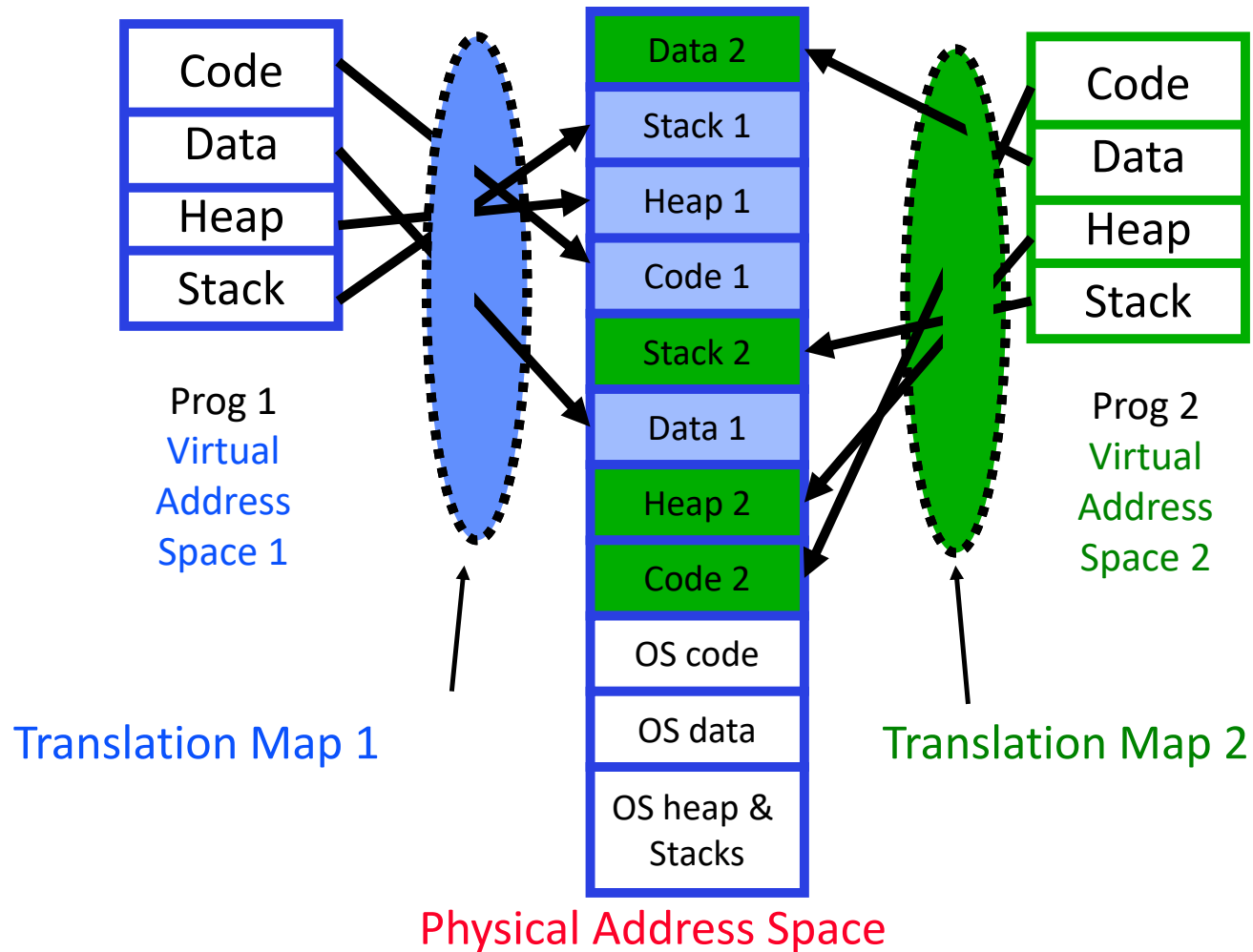


- Extreme form of Context Switch: **Swapping**
 - To make room for next process, some or all of the previous process is moved to disk
 - » Likely need to send out complete segments
 - This greatly increases the cost of context-switching
- What might be a desirable alternative?
 - **Some way to keep only 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

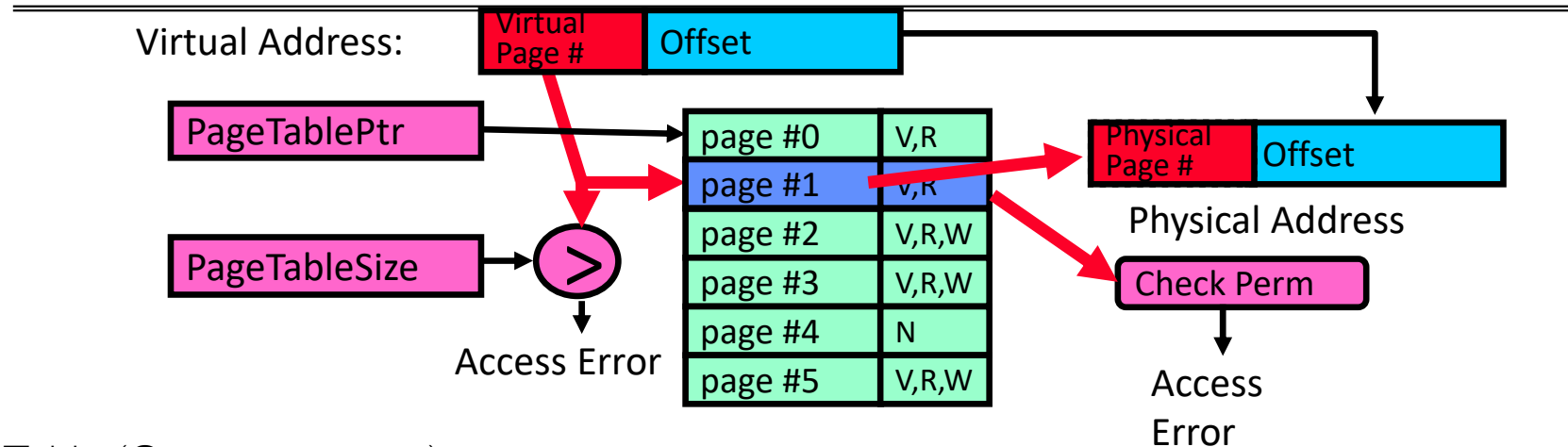
Recall: General Address Translation



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

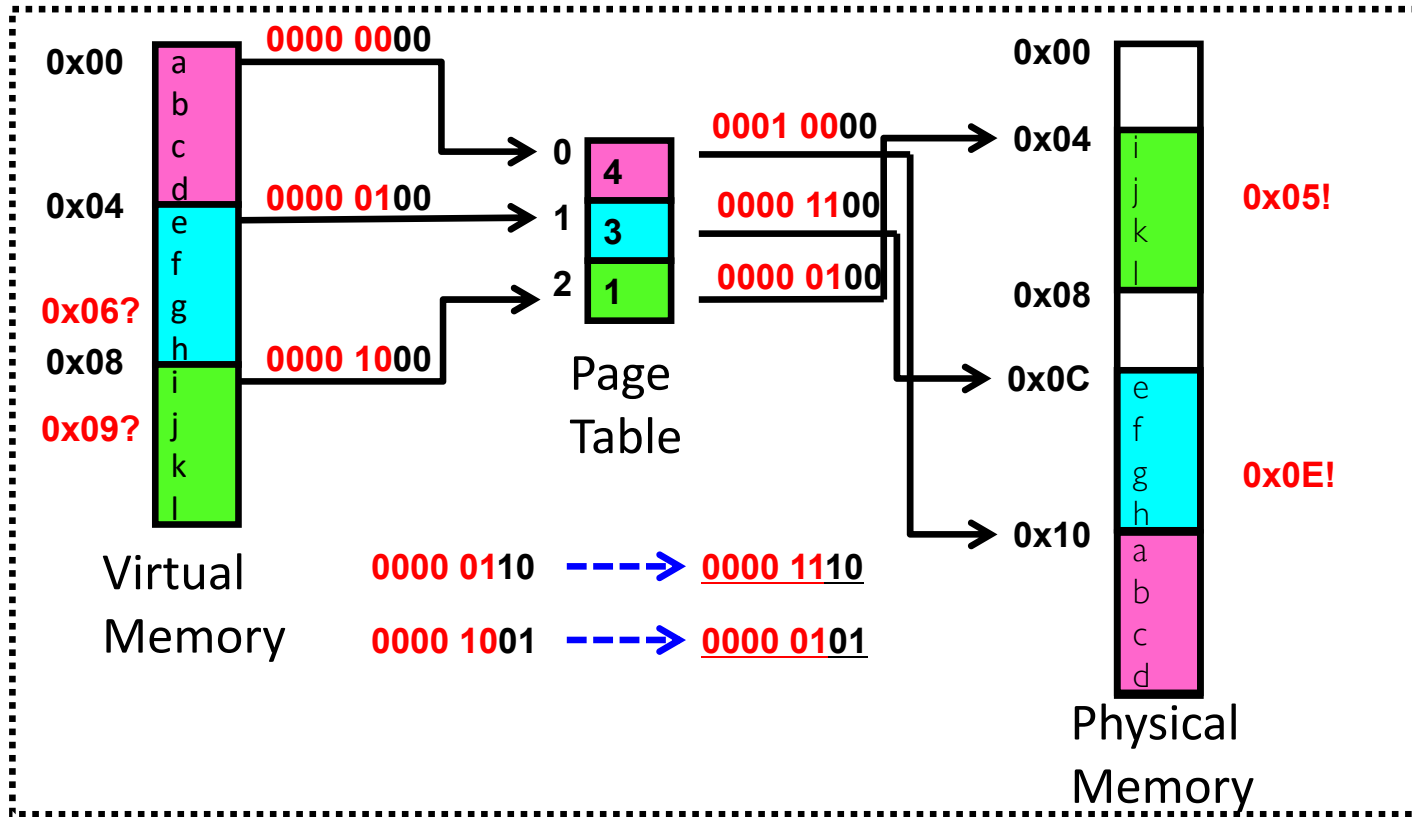
How to Implement Simple Paging?



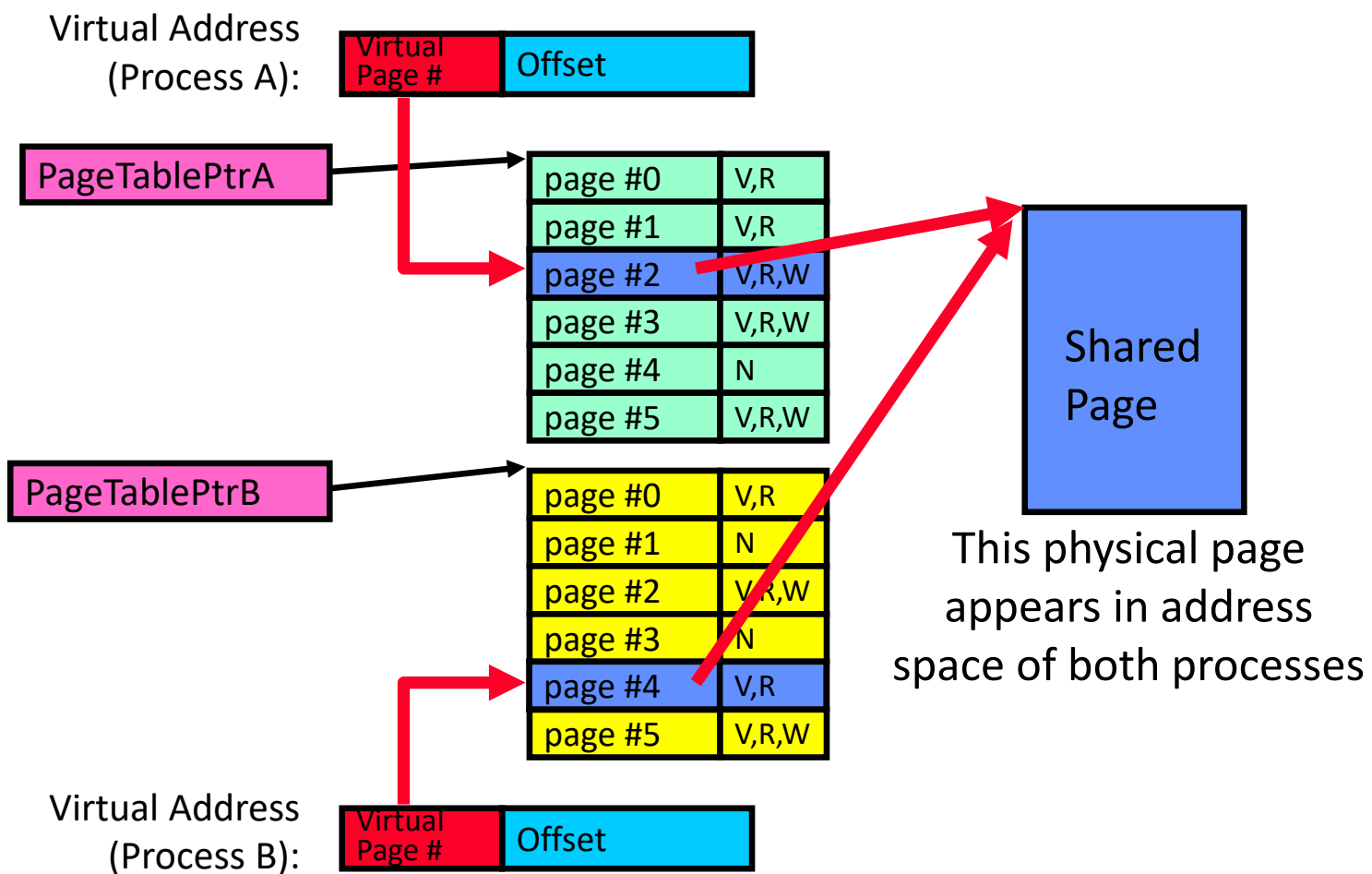
- Page Table (One per process)
 - Resides in physical memory
 - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- Virtual address mapping
 - Offset from Virtual address copied to Physical Address
 - » Example: 10 bit offset \Rightarrow 1024-byte pages
 - 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)



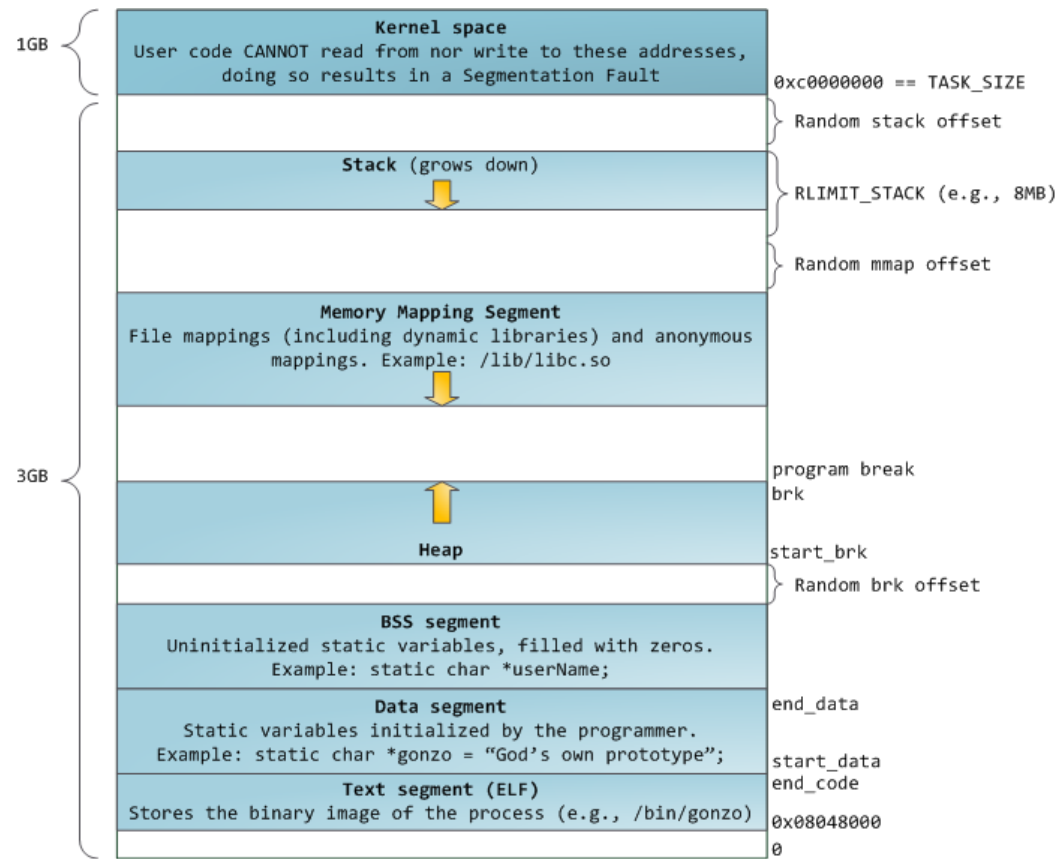
What about Sharing?



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?
- Different processes running same binary!
 - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
 - Can actually share objects directly between processes
 - » Must map page into same place in address space!
 - This is a limited form of the sharing that threads have within a single process

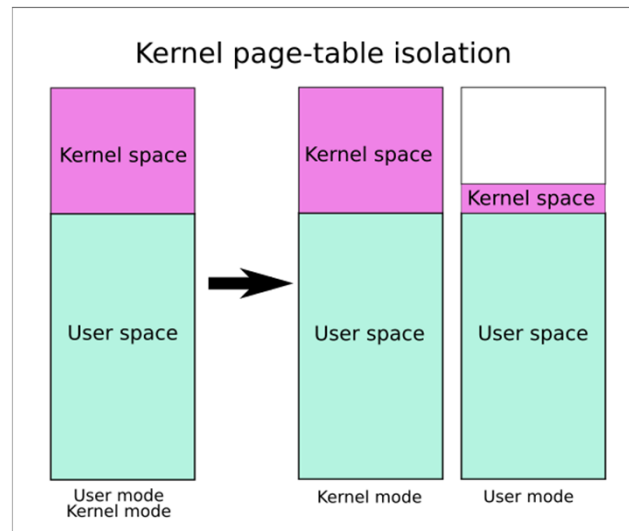
Recall: Memory Layout for Linux 32-bit (Pre-Meltdown patch!)



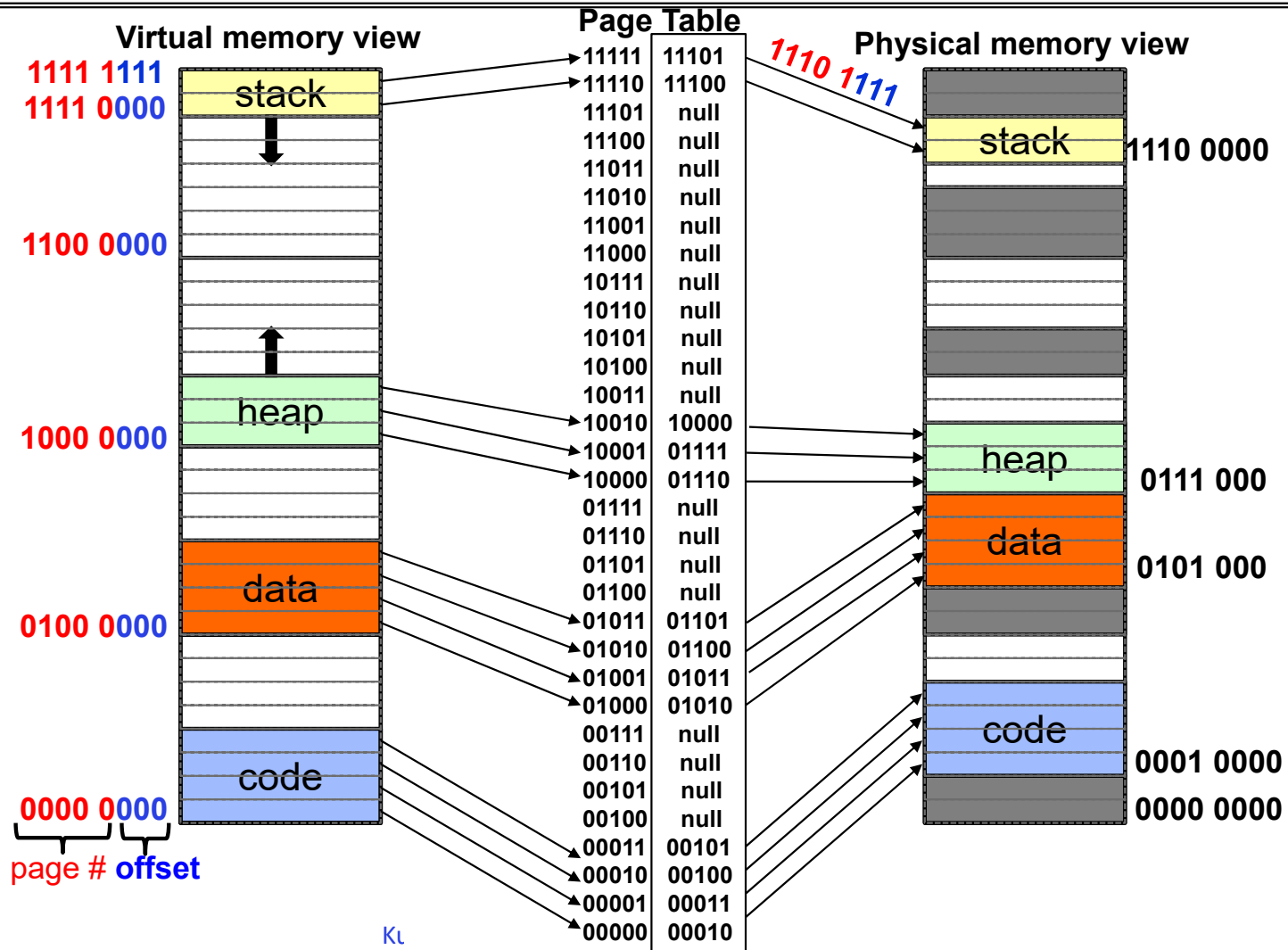
<http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png>

Some simple security measures

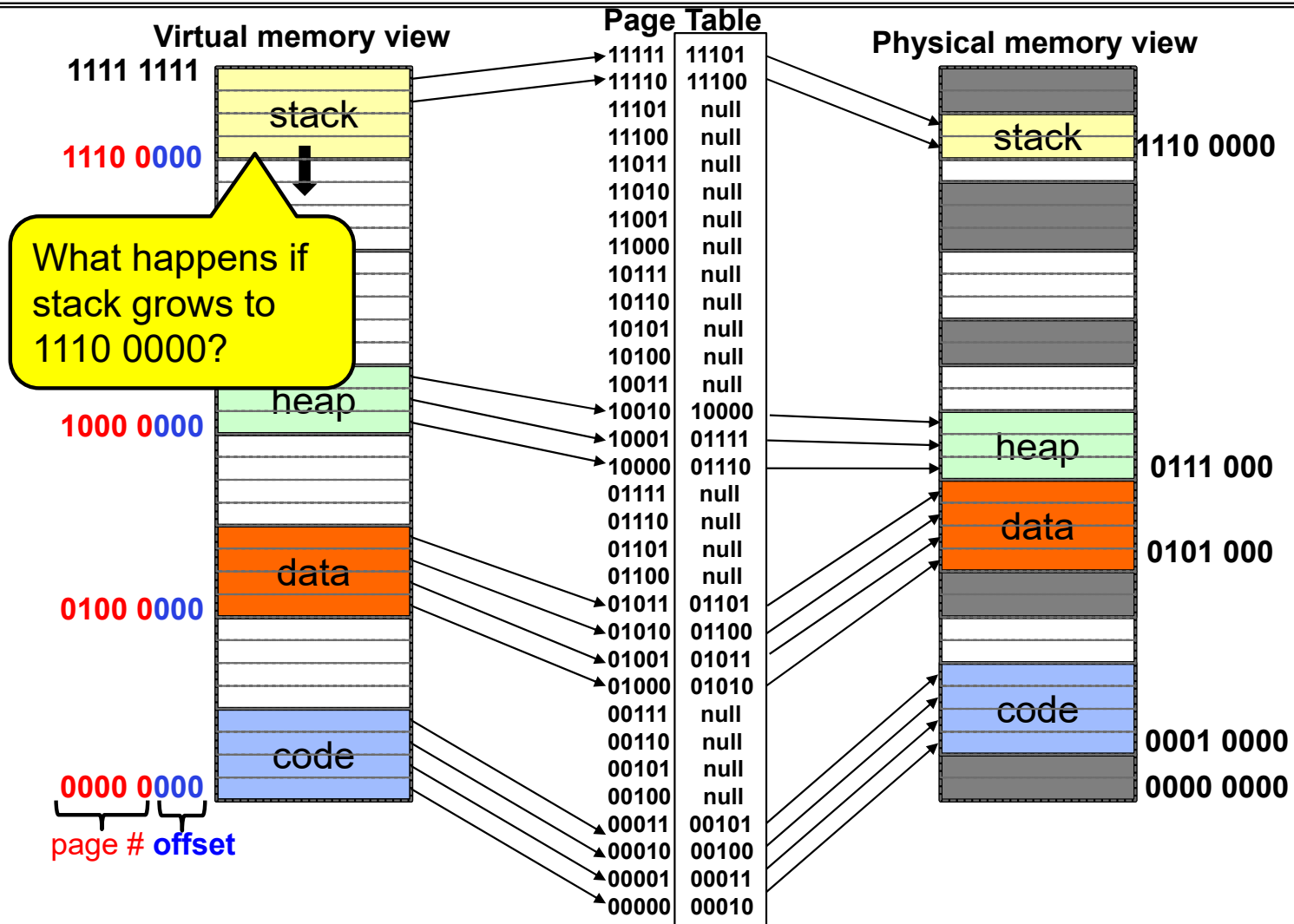
- Address Space Randomization
 - Position-Independent Code \Rightarrow can place user code anywhere in address space
 - » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
 - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
 - Don't map whole kernel space into each process, switch to kernel page table
 - Meltdown \Rightarrow map none of kernel into user mode!



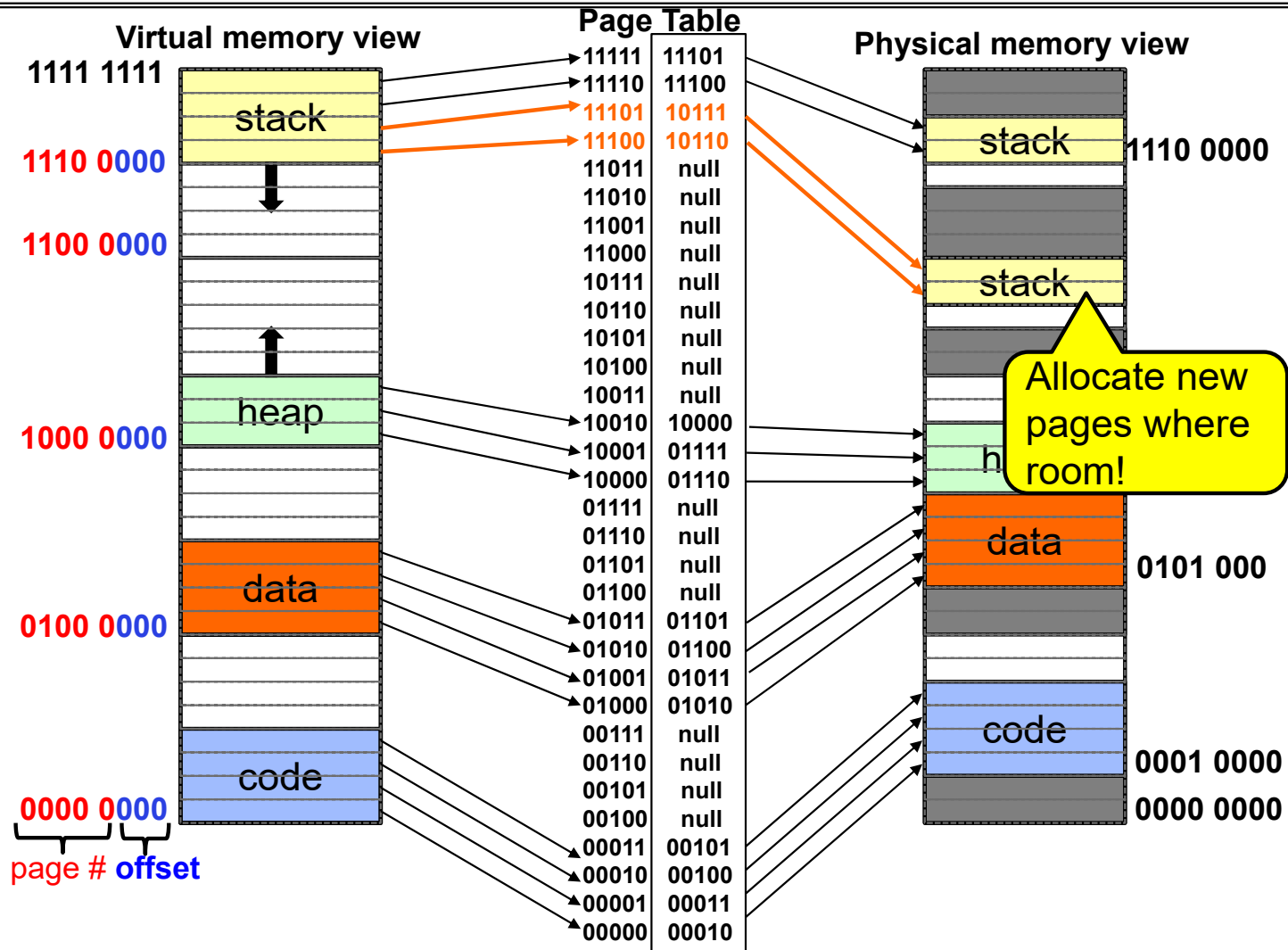
Summary: Paging



Summary: Paging



Summary: Paging



How big do things get?

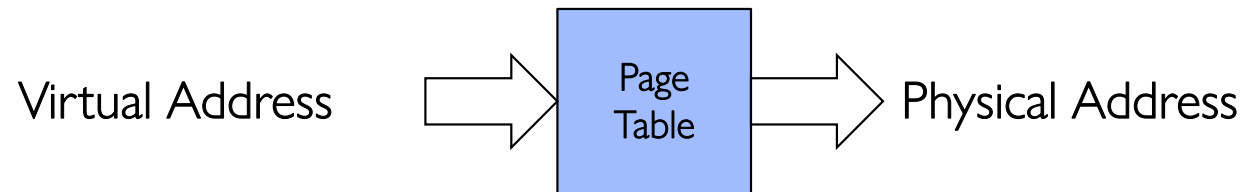
- 32-bit address space $\Rightarrow 2^{32}$ bytes (4 GB)
 - Note: “b” = bit, and “B” = byte
 - And *for memory*:
 - » “K”(kilo) = $2^{10} = 1024$ $\approx 10^3$ (But not quite!): Sometimes called “Ki” (Kibi)
 - » “M”(mega) = $2^{20} = (1024)^2 = 1,048,576$ $\approx 10^6$ (But not quite!): Sometimes called “Mi” (Mibi)
 - » “G”(giga) = $2^{30} = (1024)^3 = 1,073,741,824$ $\approx 10^9$ (But not quite!): Sometimes called “Gi” (Gibi)
- Typical page size: 4 KB
 - how many bits of the address is that ? (remember $2^{10} = 1024$)
 - Ans – $4KB = 4 \times 2^{10} = 2^{12} \Rightarrow 12$ bits of the address
- So how big is the simple page table for each process?
 - $2^{32}/2^{12} = 2^{20}$ (that’s about a million entries) $\times 4$ bytes each $\Rightarrow 4$ MB
 - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory
- How big is a simple page table on a 64-bit processor (x86_64)?
 - $2^{64}/2^{12} = 2^{52}$ (that’s 4.5×10^{15} or 4.5 exa-entries) $\times 8$ bytes each = 36×10^{15} bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
 - This is really a lot of space – for only the page table!!!
- The address space is *sparse*, i.e. has holes that are not mapped to physical memory
 - So, most of this space is taken up by page tables mapped to nothing

Page Table Discussion

- What needs to be switched on a context switch?
 - Page table pointer and limit
- What provides protection here?
 - Translation (per process) *and* dual-mode!
 - Can't let process alter its own 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 2 million page table entries!
 - Con: What if table really big?
 - » Not all pages used all the time \Rightarrow would be nice to have working set of page table in memory
- Simple Page table is way too big!
 - Does it all need to be in memory?
 - How about multi-level paging?
 - or combining paging and segmentation

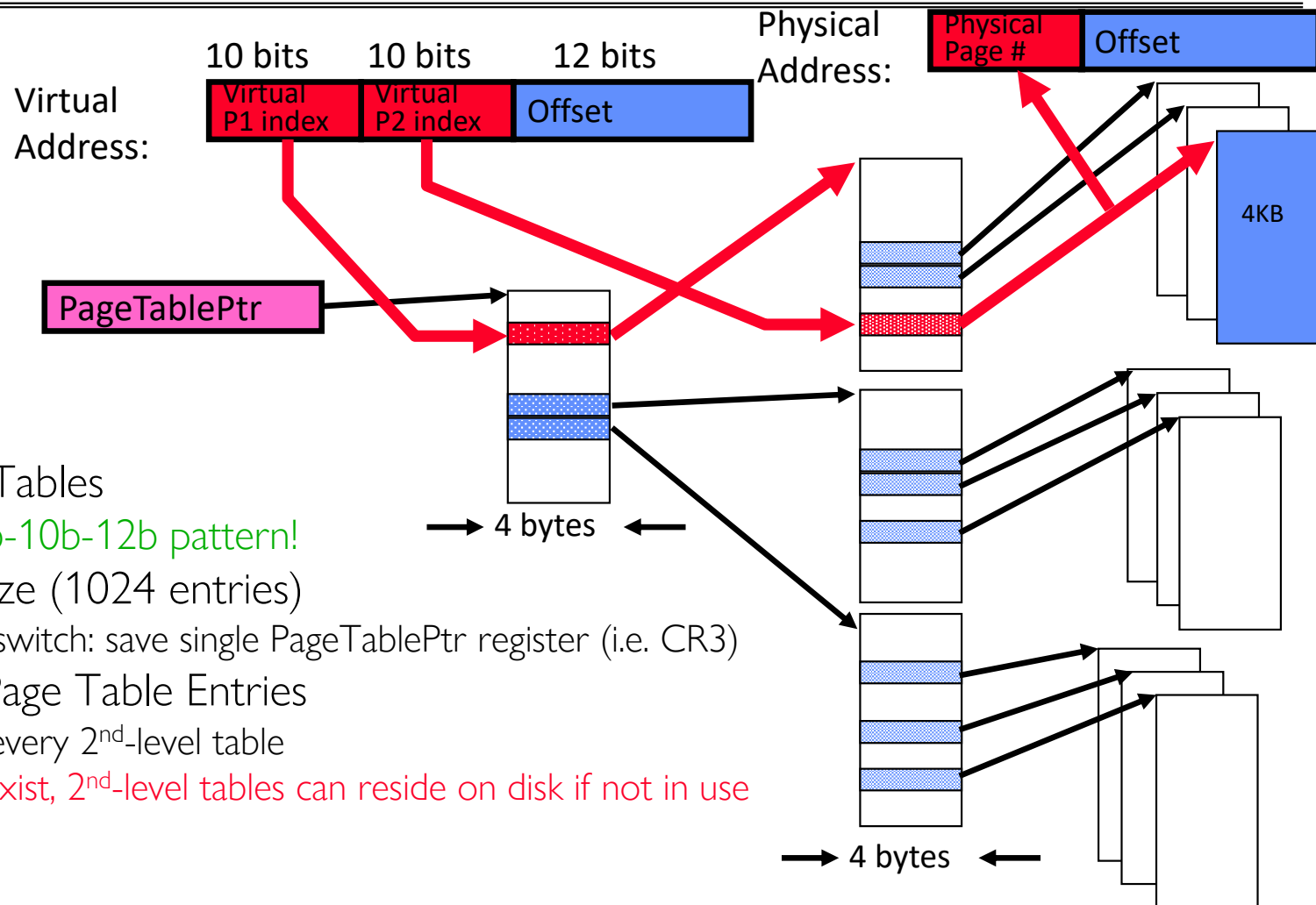
How to Structure a Page Table

- Page Table is a *map* (function) from VPN to PPN



- Simple page table corresponds to a *very large* lookup table
 - VPN is index into table, each entry contains PPN
- What other map structures can you think of?
 - Trees?
 - Hash Tables?

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



- Tree of Page Tables
 - “Magic” 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
 - On context-switch: save single PageTablePtr register (i.e. CR3)
- 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

Example: x86 classic 32-bit address translation

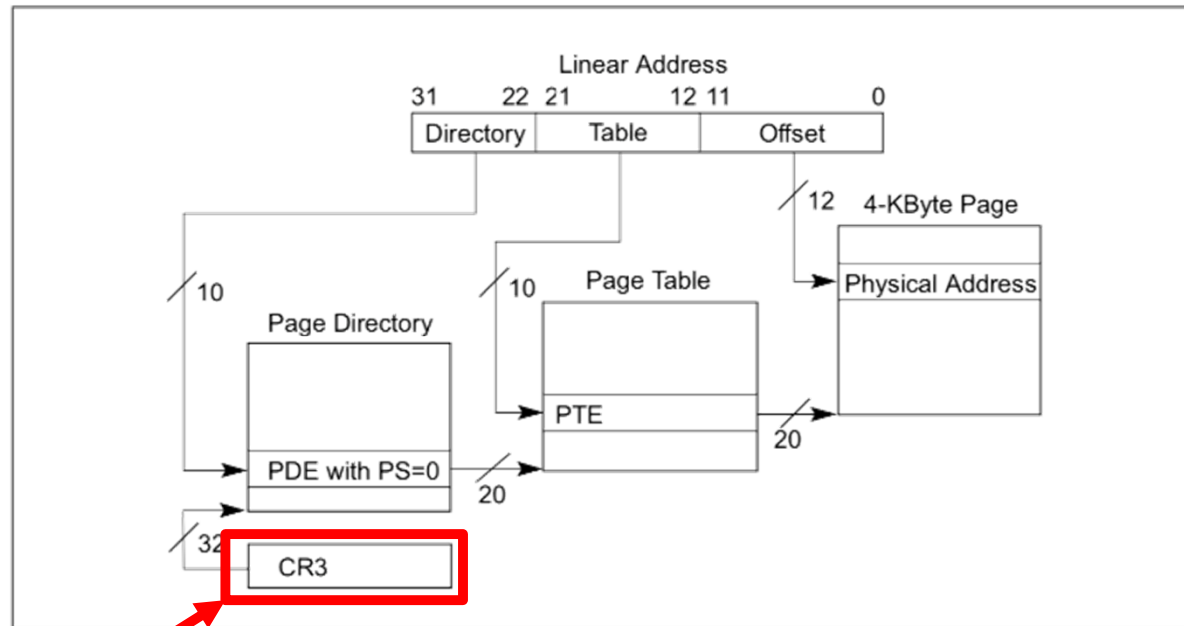
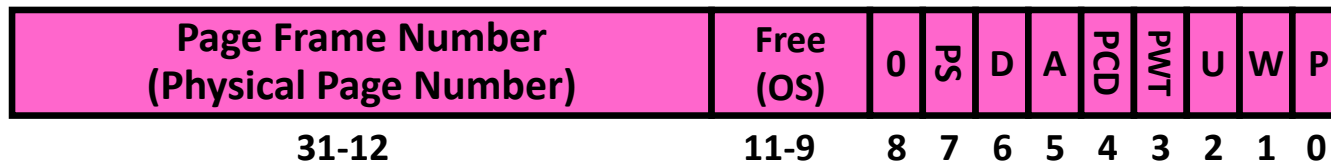


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

- Intel terminology: Top-level page-table called a “Page Directory”
 - With “Page Directory Entries”
- CR3 provides physical address of the page directory
 - This is what we have called the “PageTablePtr” in previous slides
 - Change in CR3 changes the whole translation table!

What is in a Page Table Entry (PTE)?

- 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”



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

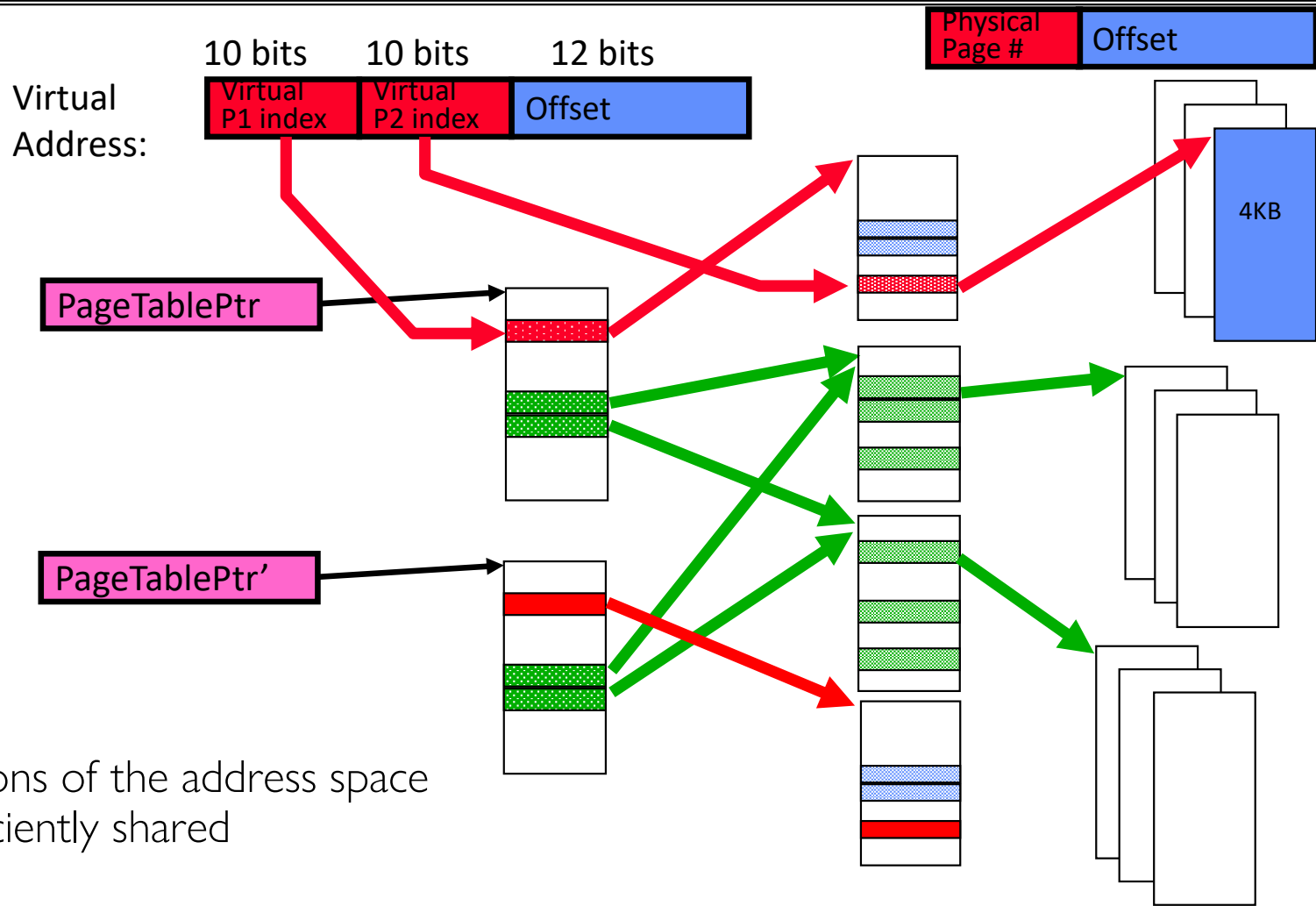
D: Dirty (PTE only): page has been modified recently

PS: Page Size: PS=1 ⇒ 4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

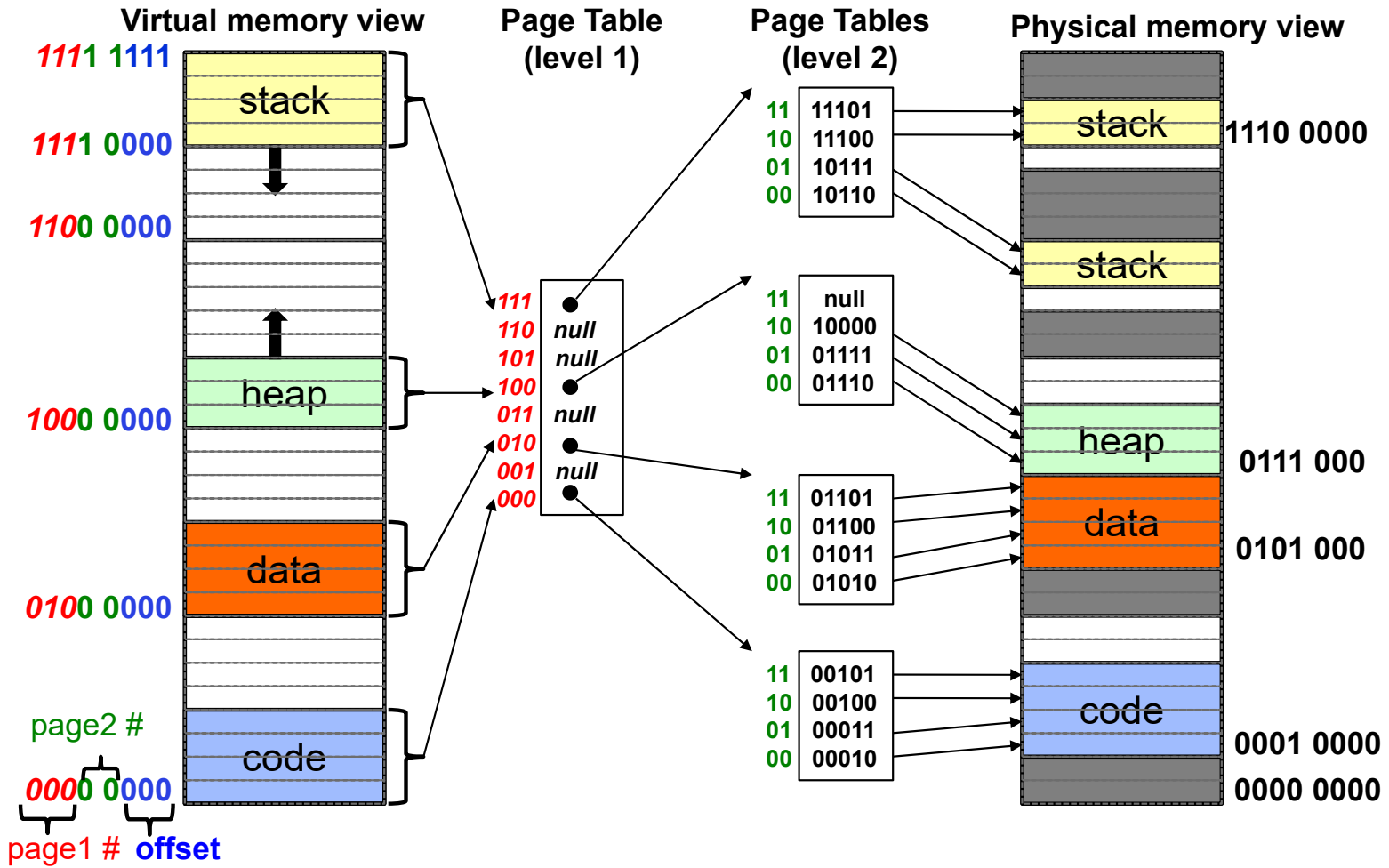
- How do we use the PTE?
 - Invalid PTE can imply different things:
 - » Region of address space is actually invalid or
 - » Page/directory is just somewhere else than memory
 - Validity checked first
 - » OS can use other (say) 31 bits for location info
- Usage Example: **Demand Paging**
 - Keep only 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

Sharing with multilevel page tables

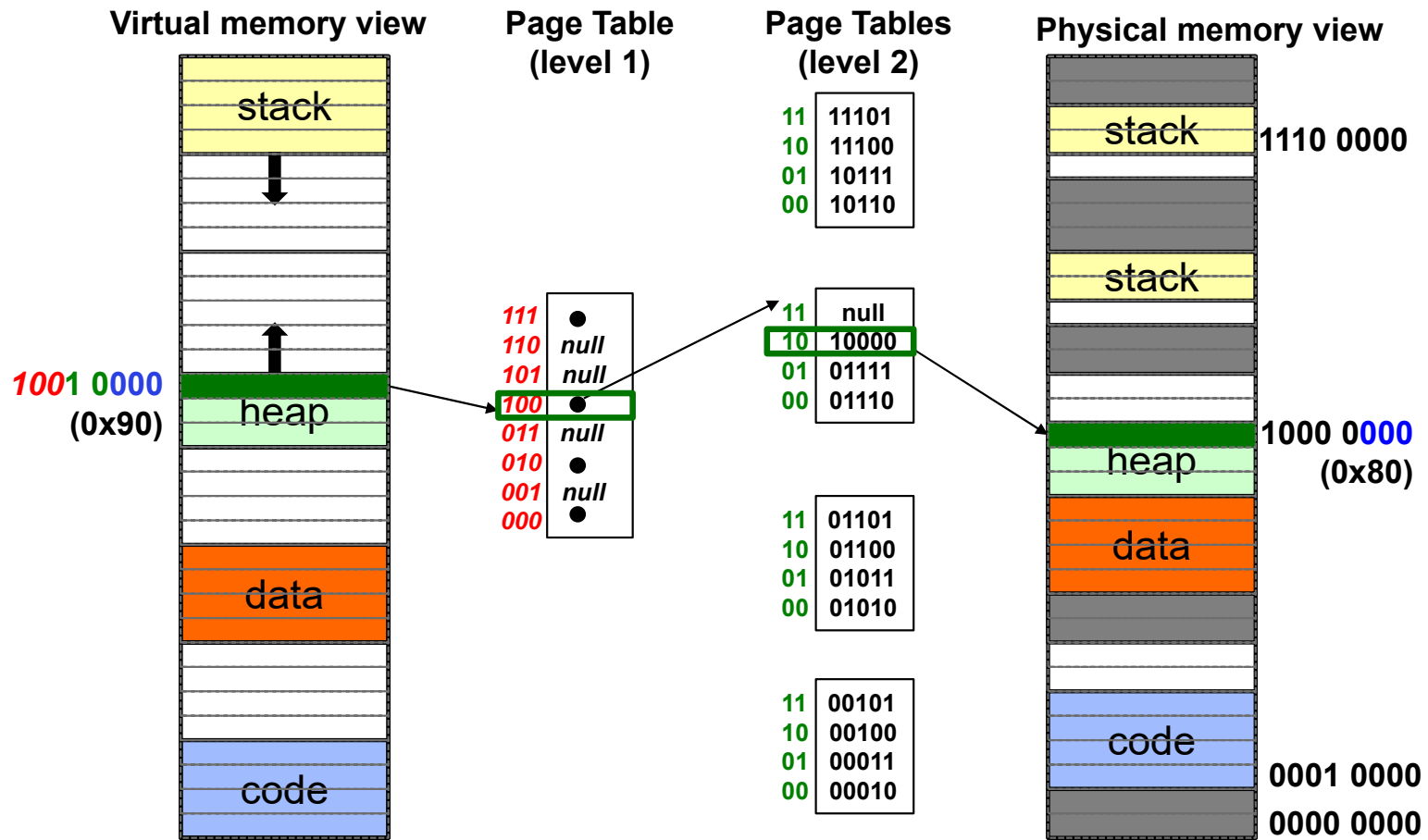


- Entire regions of the address space can be efficiently shared

Summary: Two-Level Paging



Summary: Two-Level Paging



Conclusion

- 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
 - Use of hash-table to hold translation entries
 - Size of page table ~ size of physical memory rather than size of virtual memory