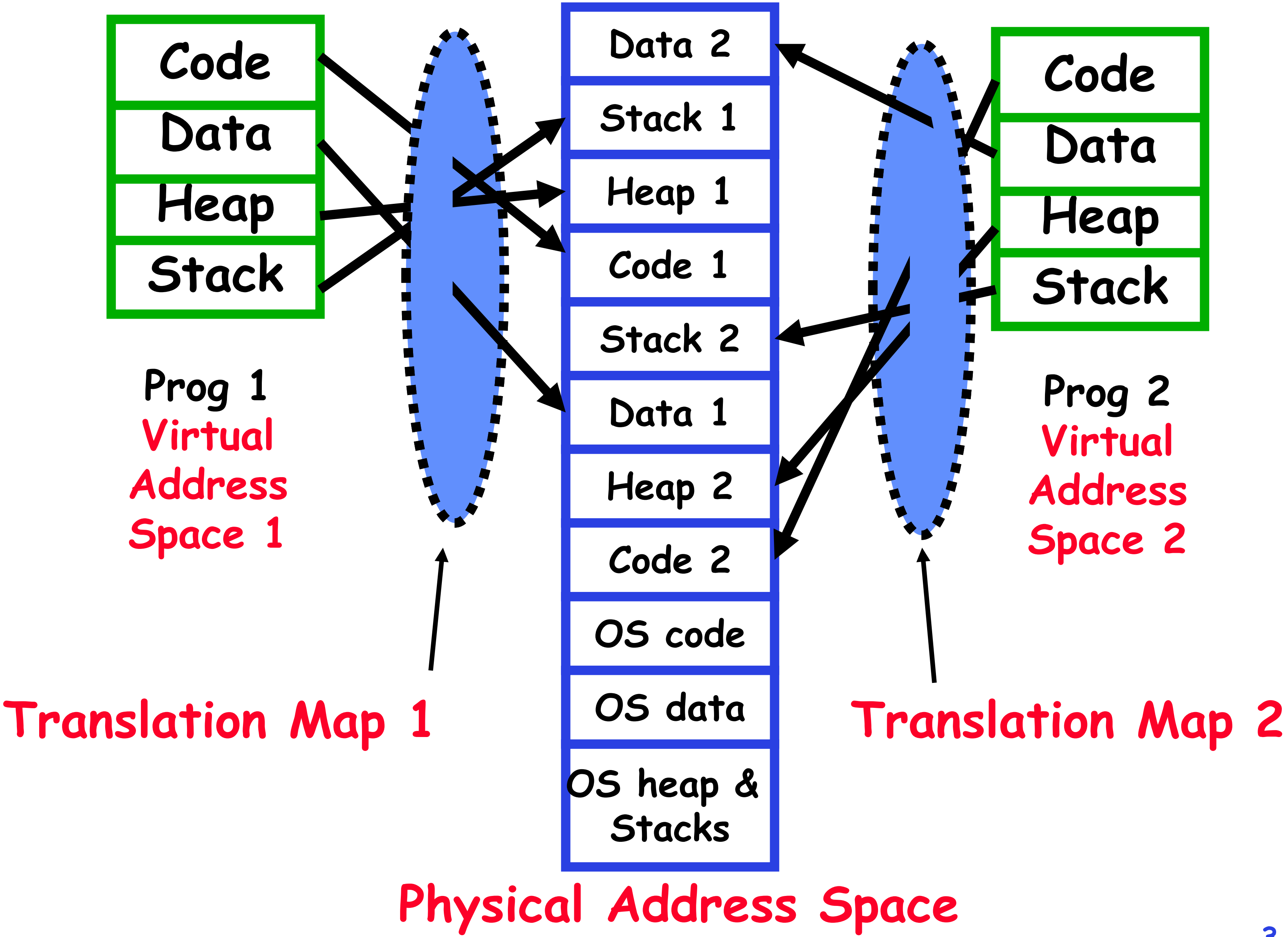
Address Translation

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

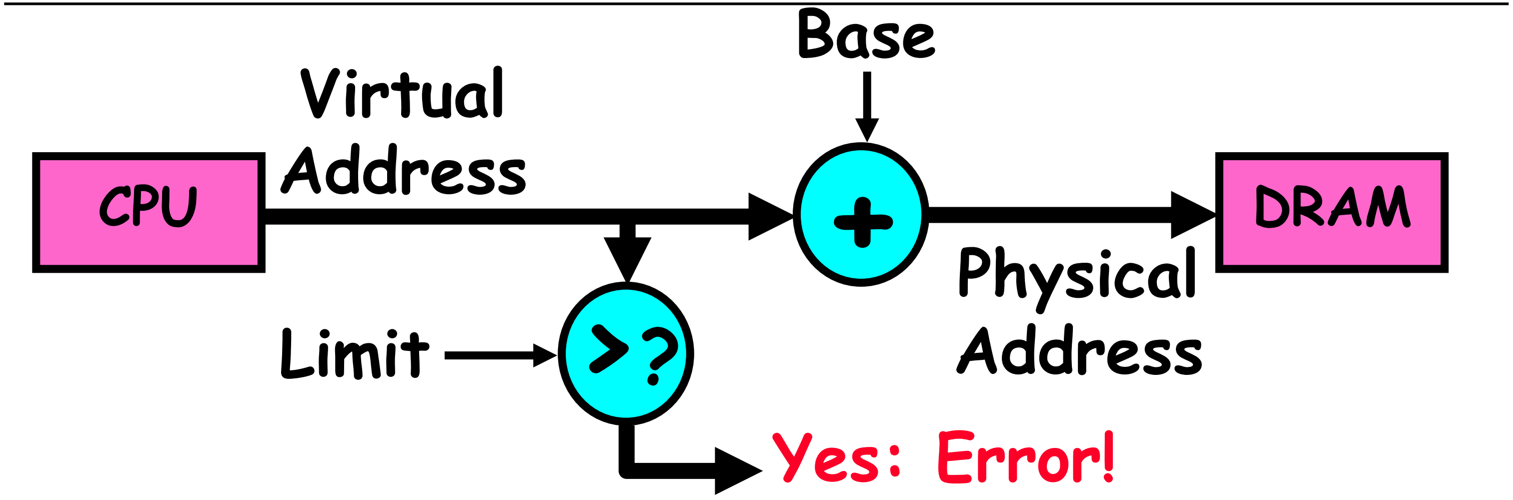
Review: Important Aspects of Memory Multiplexing

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

Review: General Address Translation



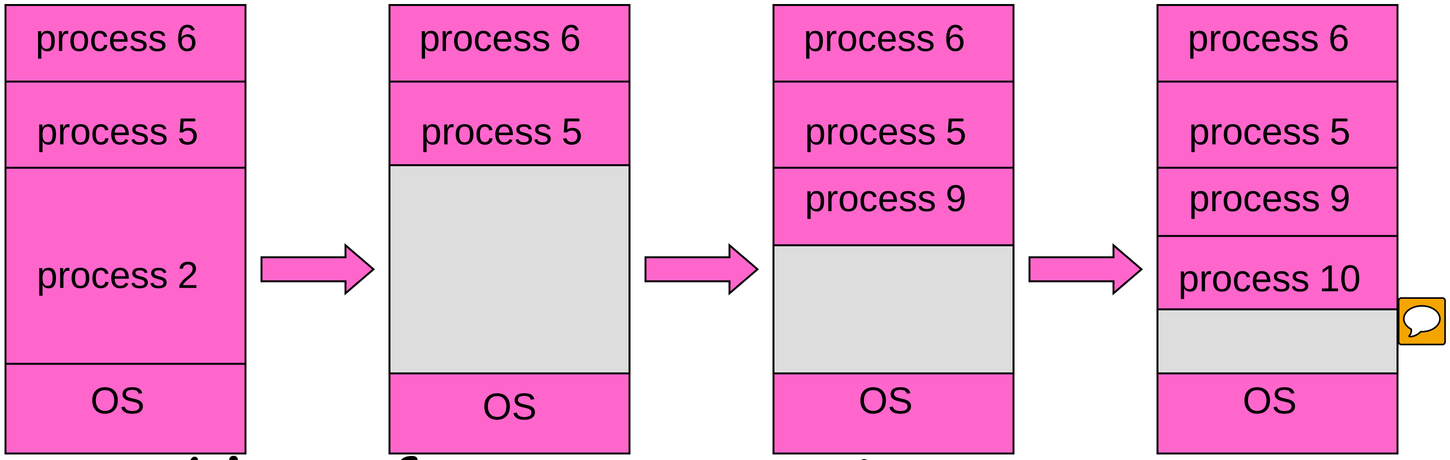
Review: Simple Segmentation: Base and Bounds (CRAY-1)



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

Review: Cons for Simple Segmentation Method

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



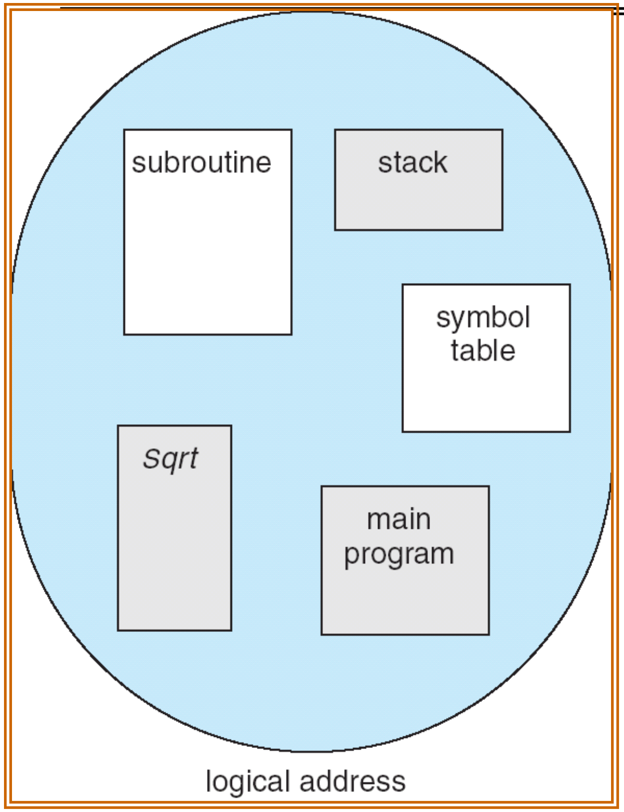
- Other problems for process maintenance
  - Doesn't allow heap and stack to grow independently
  - Want to put these as far apart in virtual memory space as possible so that they can grow as needed
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory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

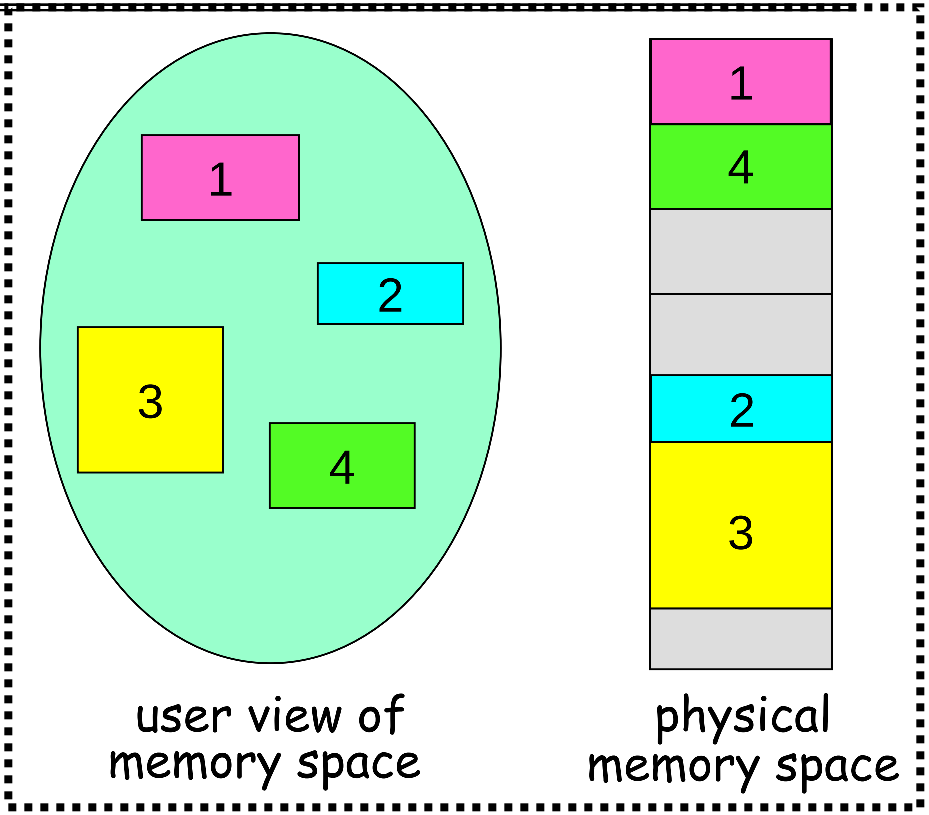
Goals for Today

- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Inverted page tables

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

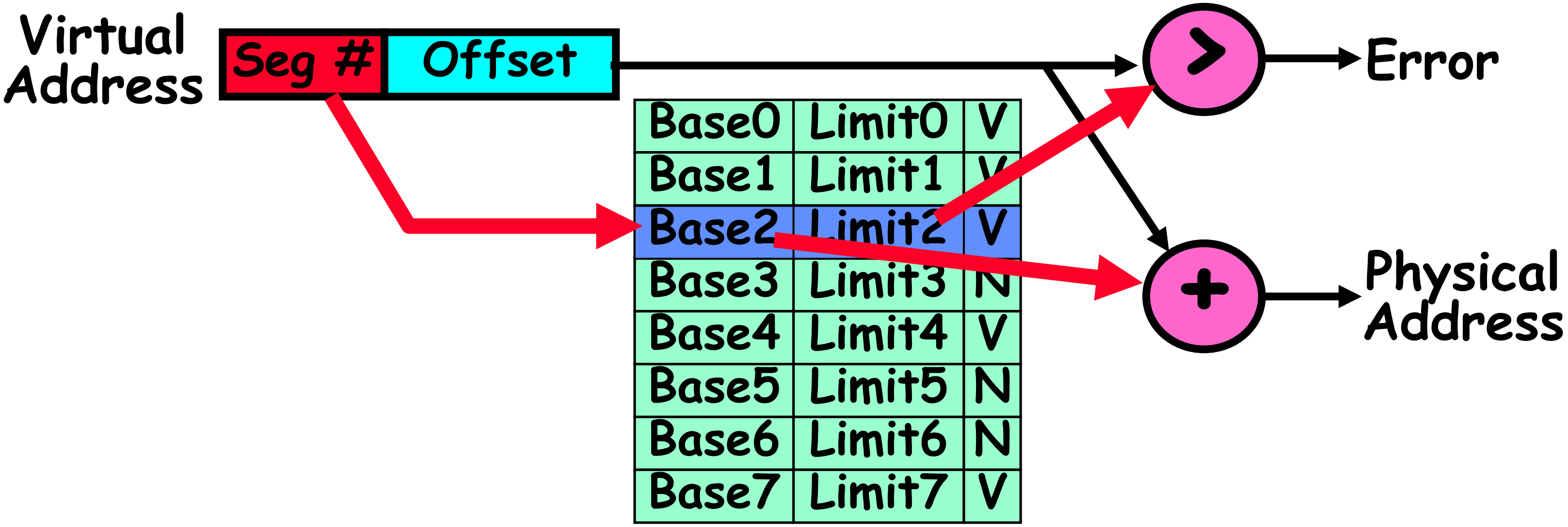
More Flexible Segmentation





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

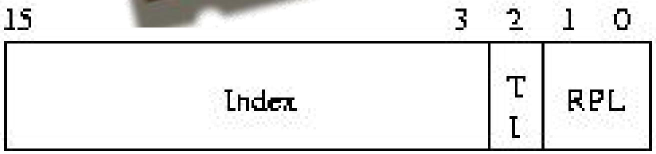
Implementation of Multi-Segment Model



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

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

Intel x86 Special Registers

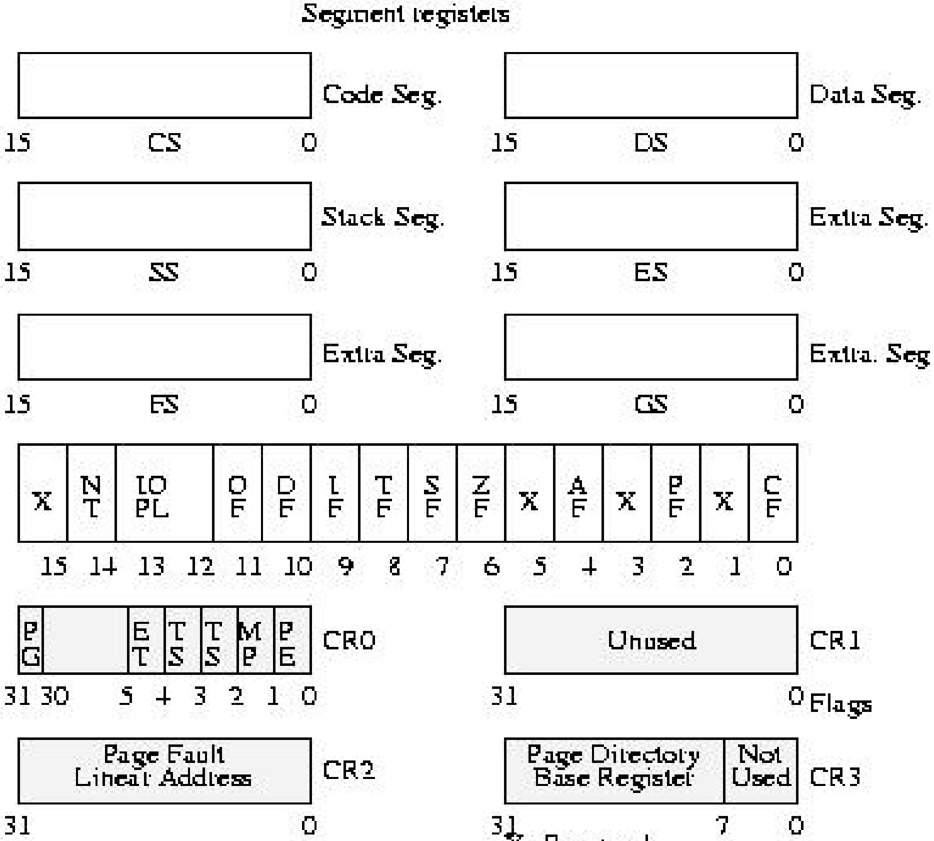


RPL = Requestor Privilege Level
TI = Table Indicator
(0 = GDT, 1 = LDT)
Index = Index into table

Protected Mode segment selector

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

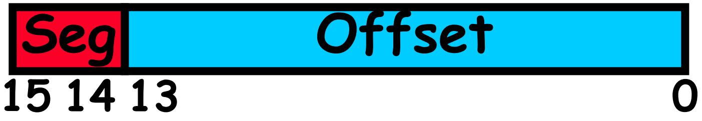
80386 Special Registers



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

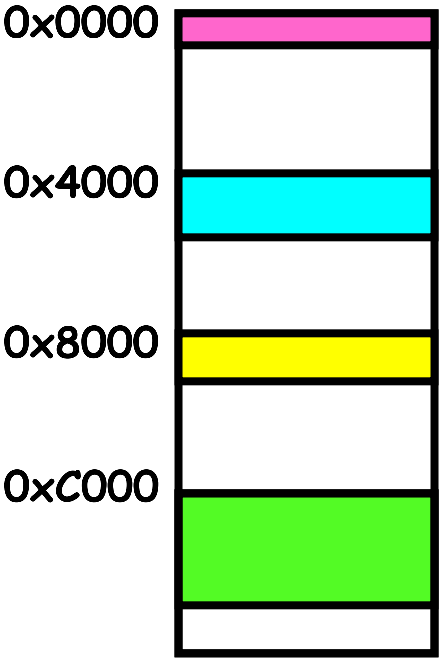
X=Reserved
NT=Nested Task
IOPL=I/O Privilege Level
OF=Overflow Flag
DF=Direction Flag
IF=Interrupt Flag
TF=Trap Flag
SF=Sign Flag
ZF=Zero Flag
AF=Auxiliary Flag
PF=Parity Flag
CF=Carry Flag

Example: Four Segments (16 bit addresses)

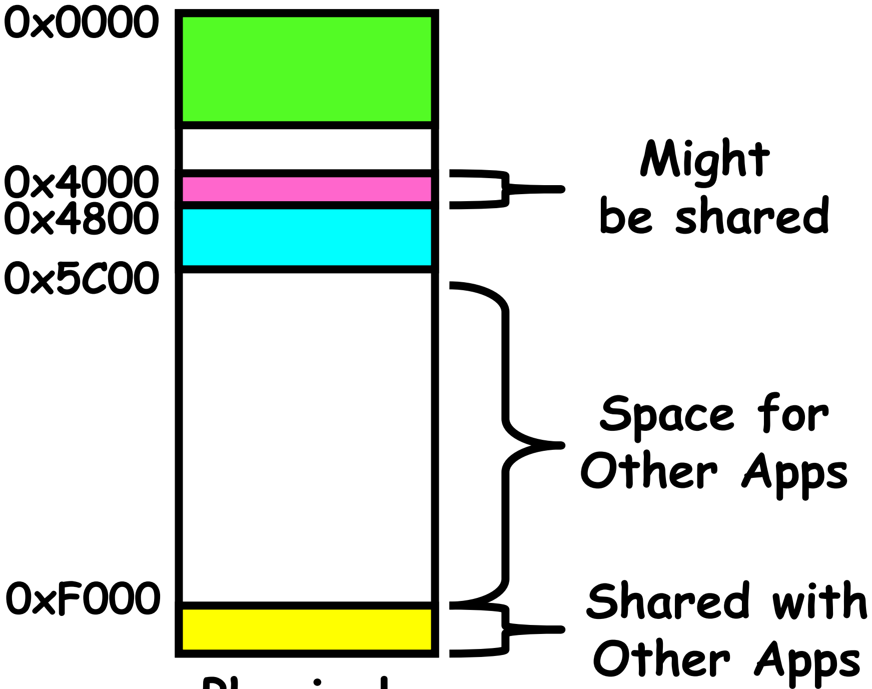


Virtual Address Format

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



Virtual Address Space



Physical Address Space

Example of segment translation

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

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

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

- Fetch 0x240. Virtual segment #? 0; Offset? 0x240
  Physical address? Base=0x4000, so physical addr=0x4240
  Fetch instruction at 0x4240. Get "la $a0, varx"
  Move 0x4050 → $a0, Move PC+4→PC

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

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")

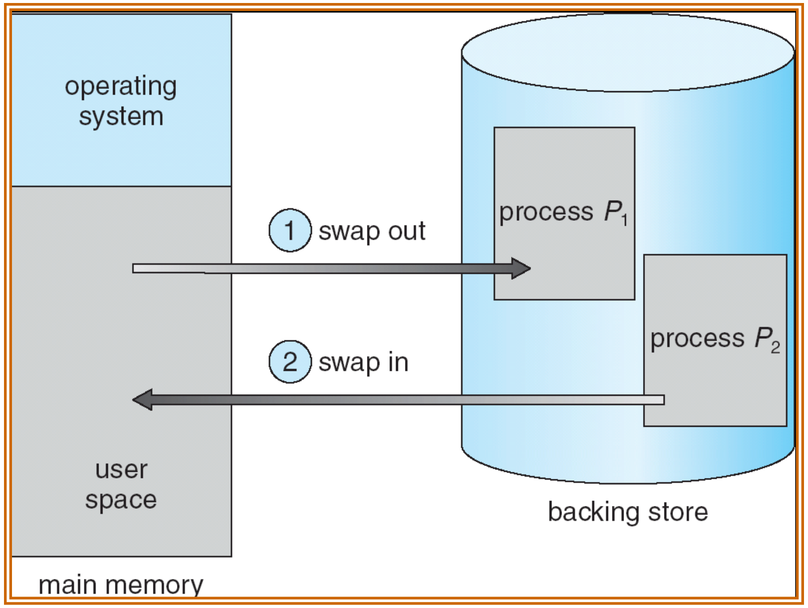
Segmentation pros & cons

- Pros
  - Efficient for sparse address space
  - Easy to share whole segments

- Cons
  - Complex memory alloation

    Still need first fit, best fit, etc., and re-shuffling to coalesce free fragments, if no single free space is big enough for a new segment

Schematic View of Swapping

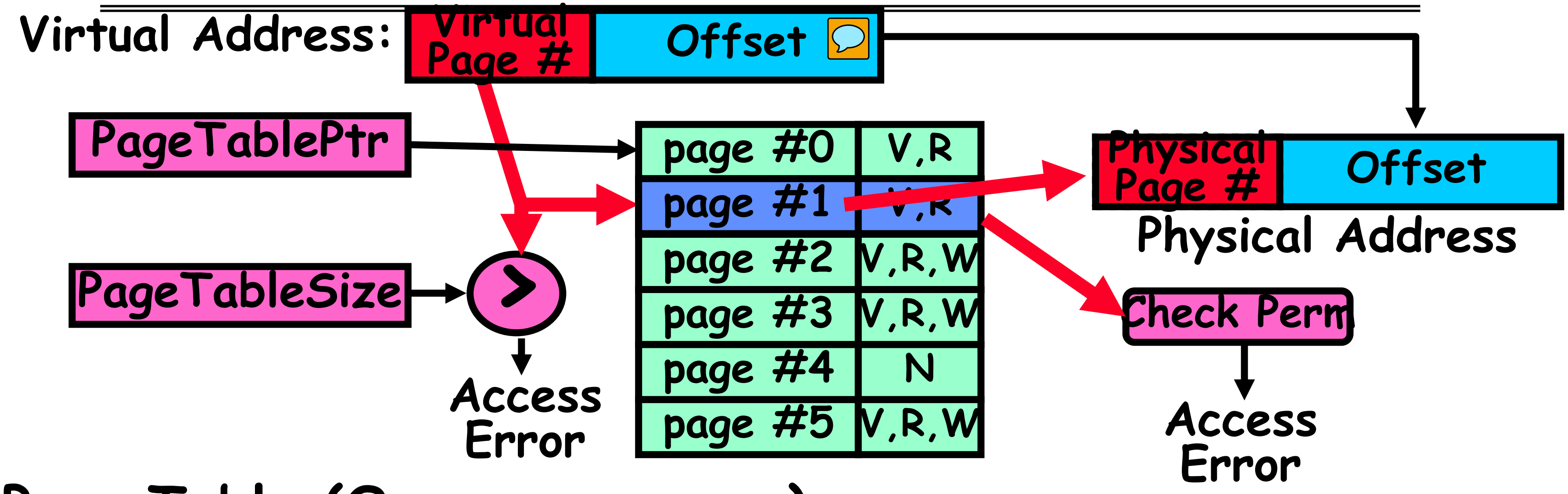


- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- Desirable alternative?
  - Some way to keep only active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

Paging: Physical Memory in Fixed Size Chunks

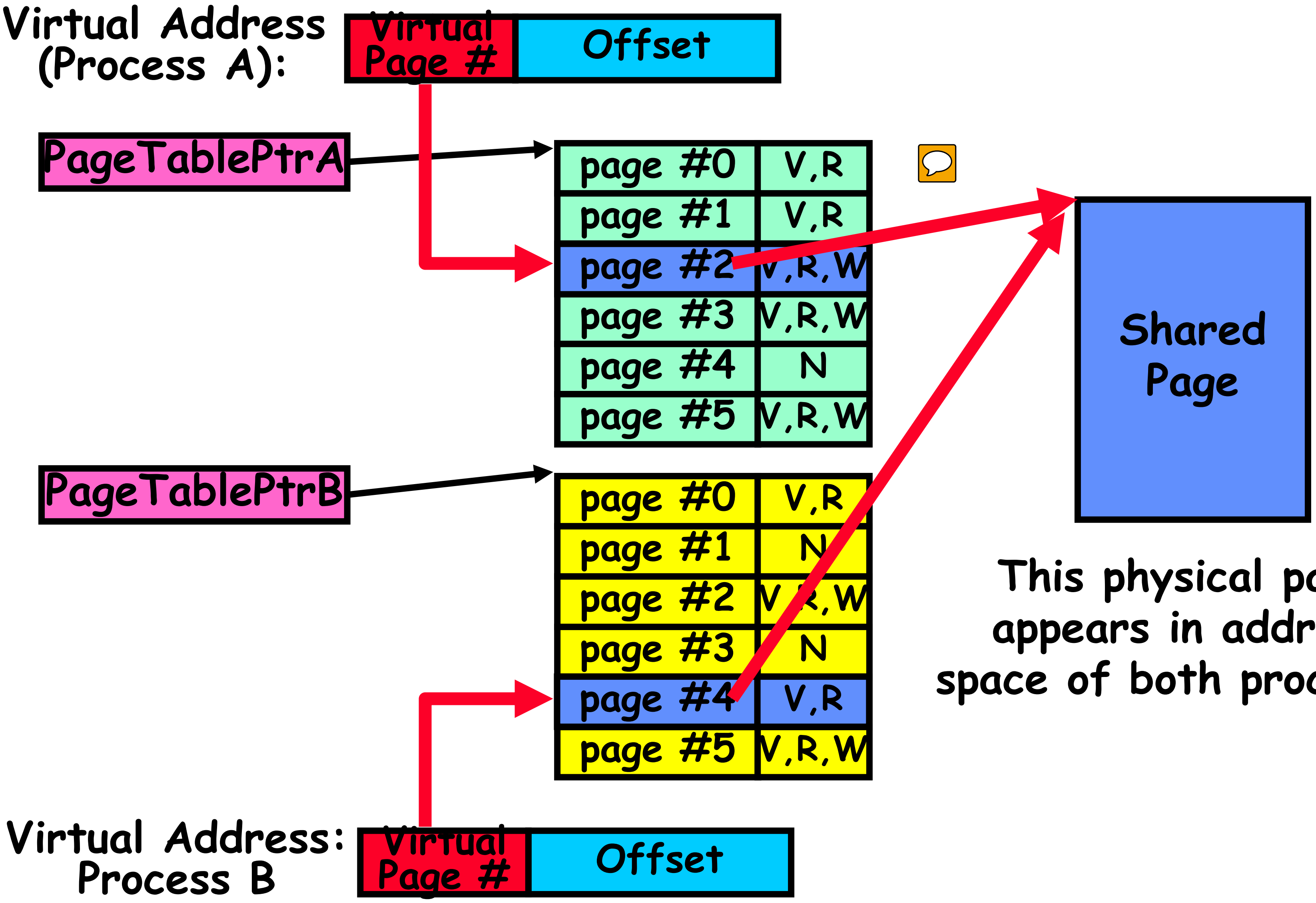
- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      0011000111000110 1 … 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

What about Sharing?



Virtual Address (Process A):

Virtual Page #	Offset

PageTablePtrA

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

Shared Page

PageTablePtrB

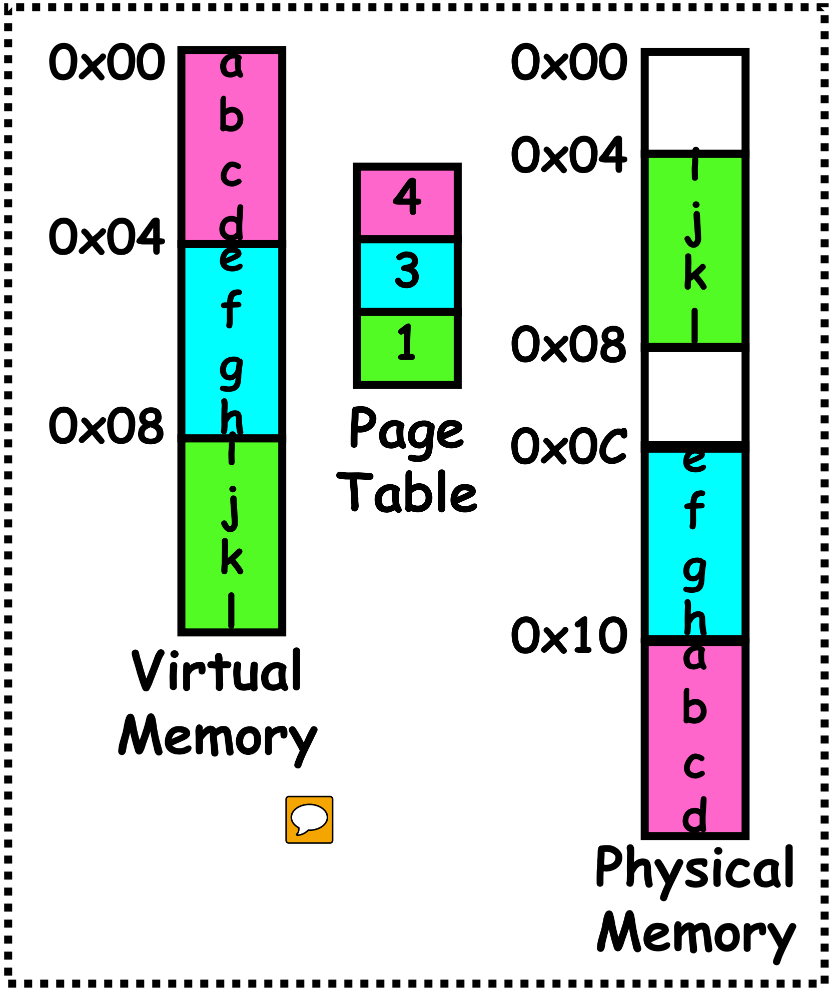
page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

Virtual Address: Process B

Virtual Page #	Offset

This physical page appears in address space of both processes

Simple Page Table Discussion

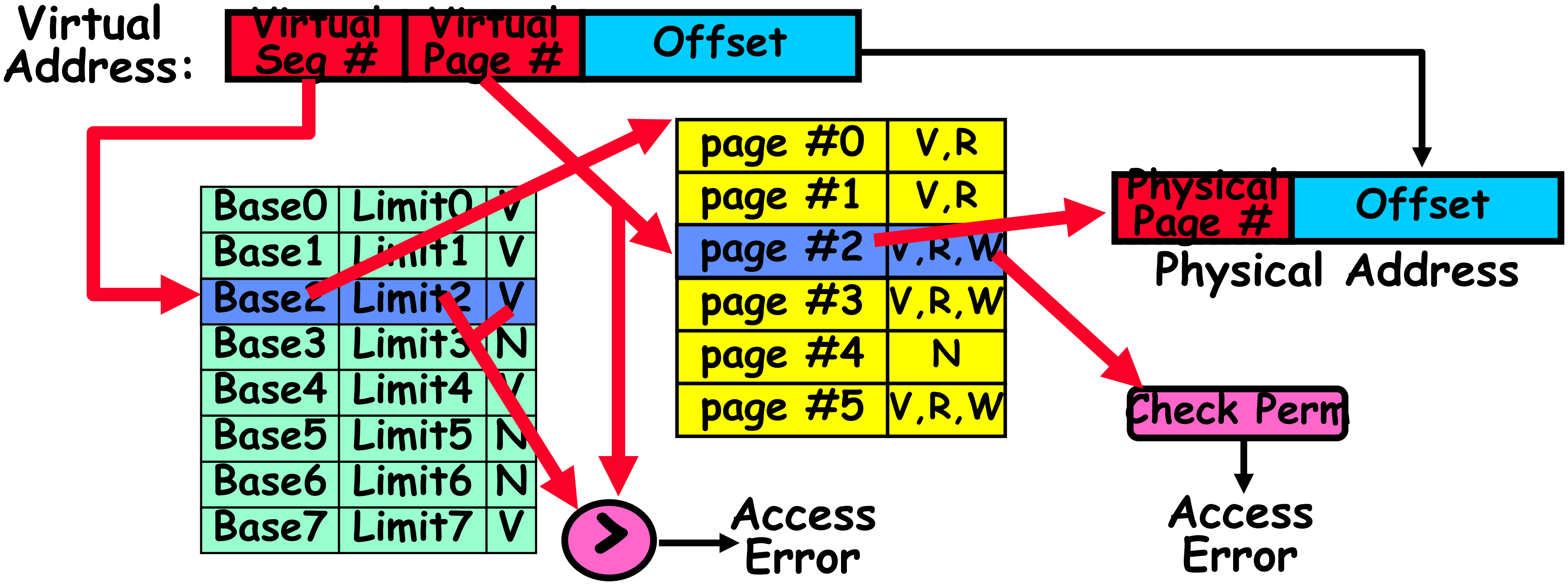


Example (4 byte pages)

- 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 (2^{31}-1).
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

Multi-level Translation

- 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):



Virtual Address:

Virtual Seg #

Virtual Page #

Offset

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

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

Access Error

Physical Page #

Offset

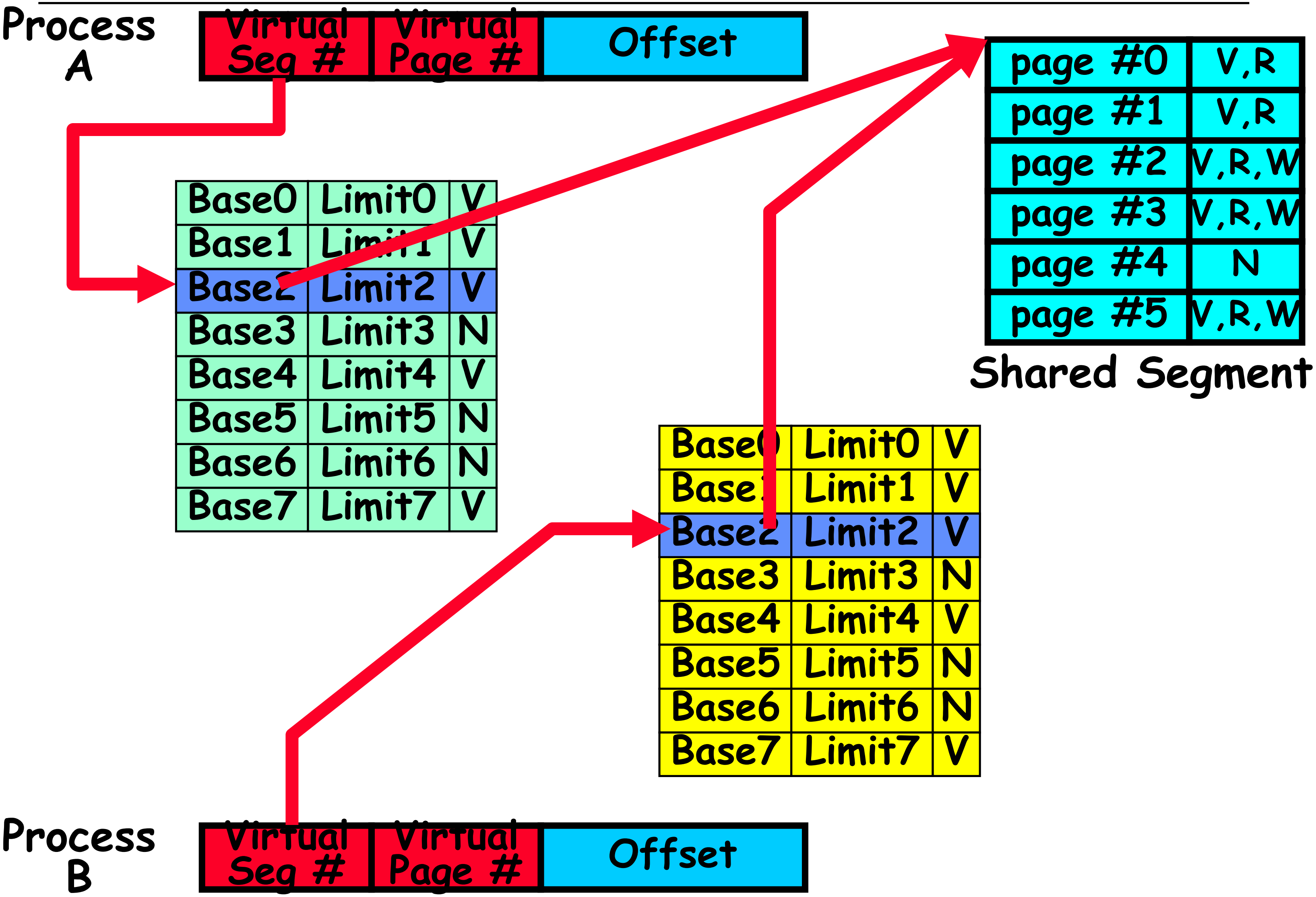
Physical Address

Check Perm

Access Error

- 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)?



Process A

Virtual Seg #	Virtual Page #	Offset

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

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

Shared Segment

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

Process B

Virtual Seg #	Virtual Page #	Offset

Example of Multi-level Translation

	4 bits	8 bits	12 bits
Virtual Address:	Virtual Seg #	Virtual Page #	Offset

Segment Table

Page Table Ptr	Page Table Size
0x2000	0x14
0x0000	0x0
0x1000	0xd
0x0000	0x0

Physical Memory

…	…
0x1000	0x6
	0xb
	0x4
…	…
0x2000	0x13
	0x2a
	0x3
…	…

- What do following addresses translate to?
  - 2070
  - 202016
  - 104c684
  - 210014

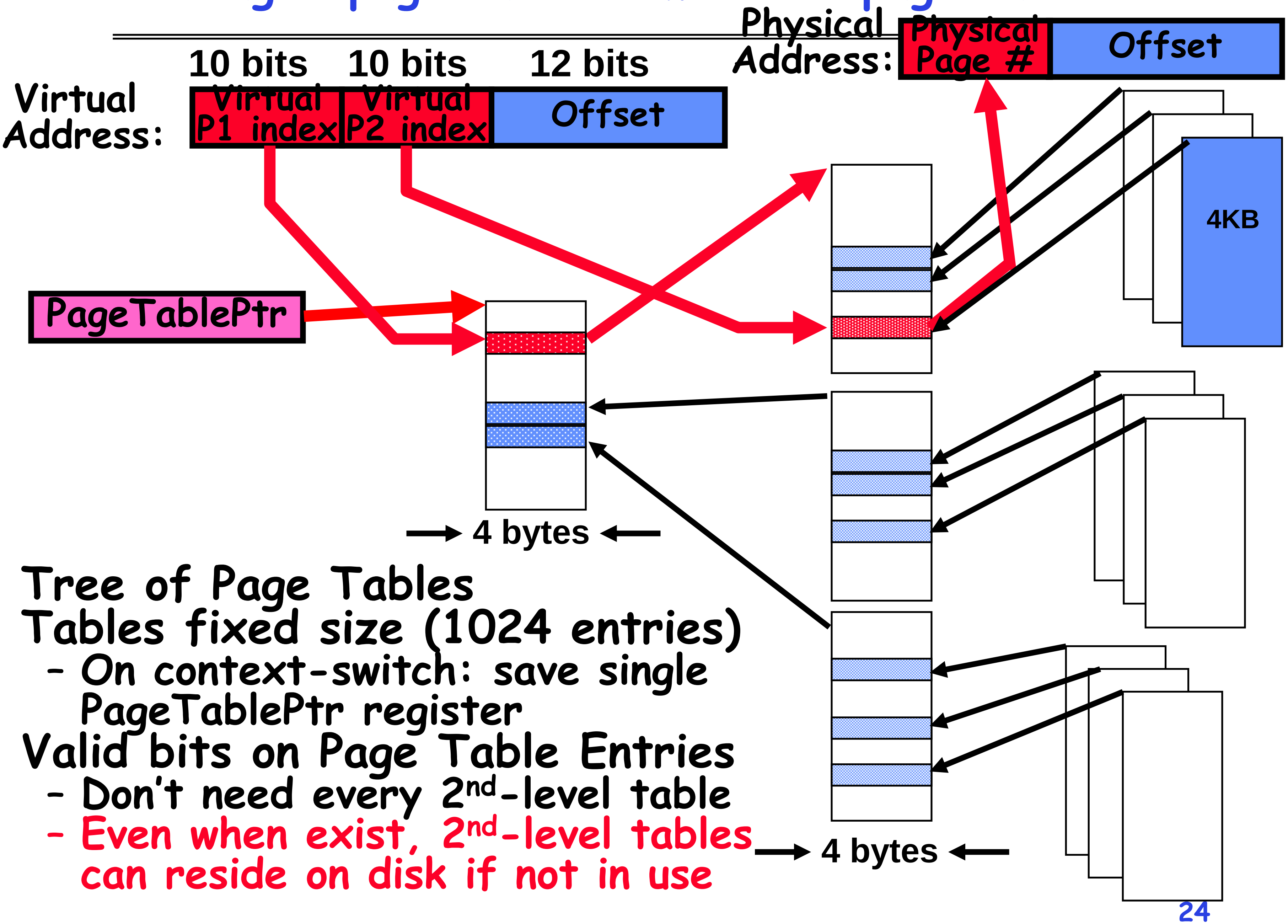
Multi-level Translation Analysis

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

Paged page tables: two-level page table

- A different solution to sparse address spaces is to allow the page tables to be paged
  - only need to allocate physical memory for page table entries you really use.
  - Top level page table is in physical memory, all lower levels of hierarchy are in virtual memory (and therefore can be allocated in fixed size page frames in physical memory).

Paged page tables: two-level page table

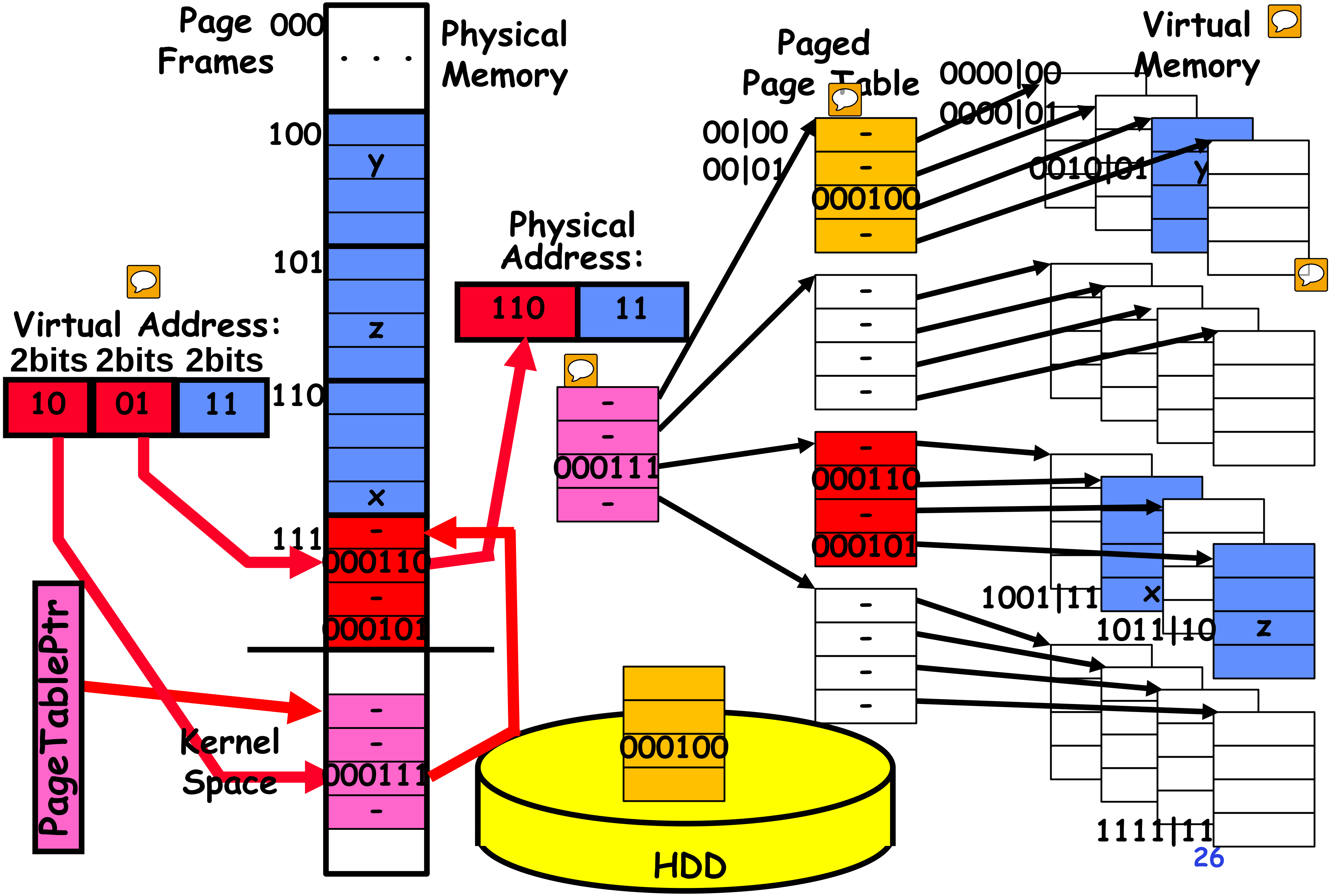


- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Paged page tables-Discussion

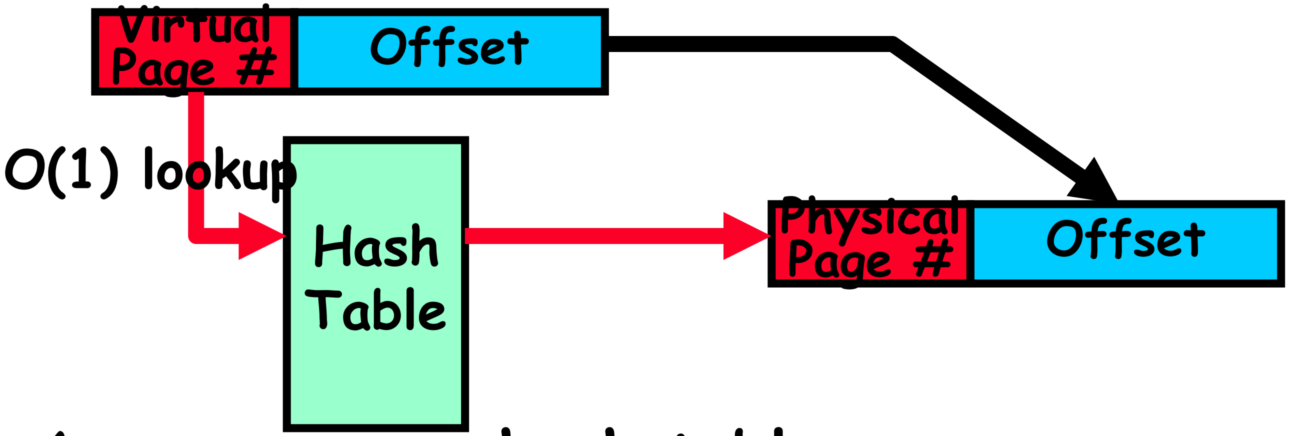
- Potentially, each memory reference involves three memory references
  - one for the system page table, one for the user page table, and one for the real data
- How do we reduce the overhead of translation?
  - Caching in translation lookaside buffers (TLB's).
- Relative to multilevel translation, paged page tables are more efficient if using a TLB.
  - If virtual address of page table entry is in TLB, can skip one or more levels of translation.
- Page tables need to be contiguous
  - However, previous example keeps tables to exactly one page in size

Paged page tables: Example



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
    - » Proportional to how many pages are actually being used
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

Summary

- 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