Main Memory

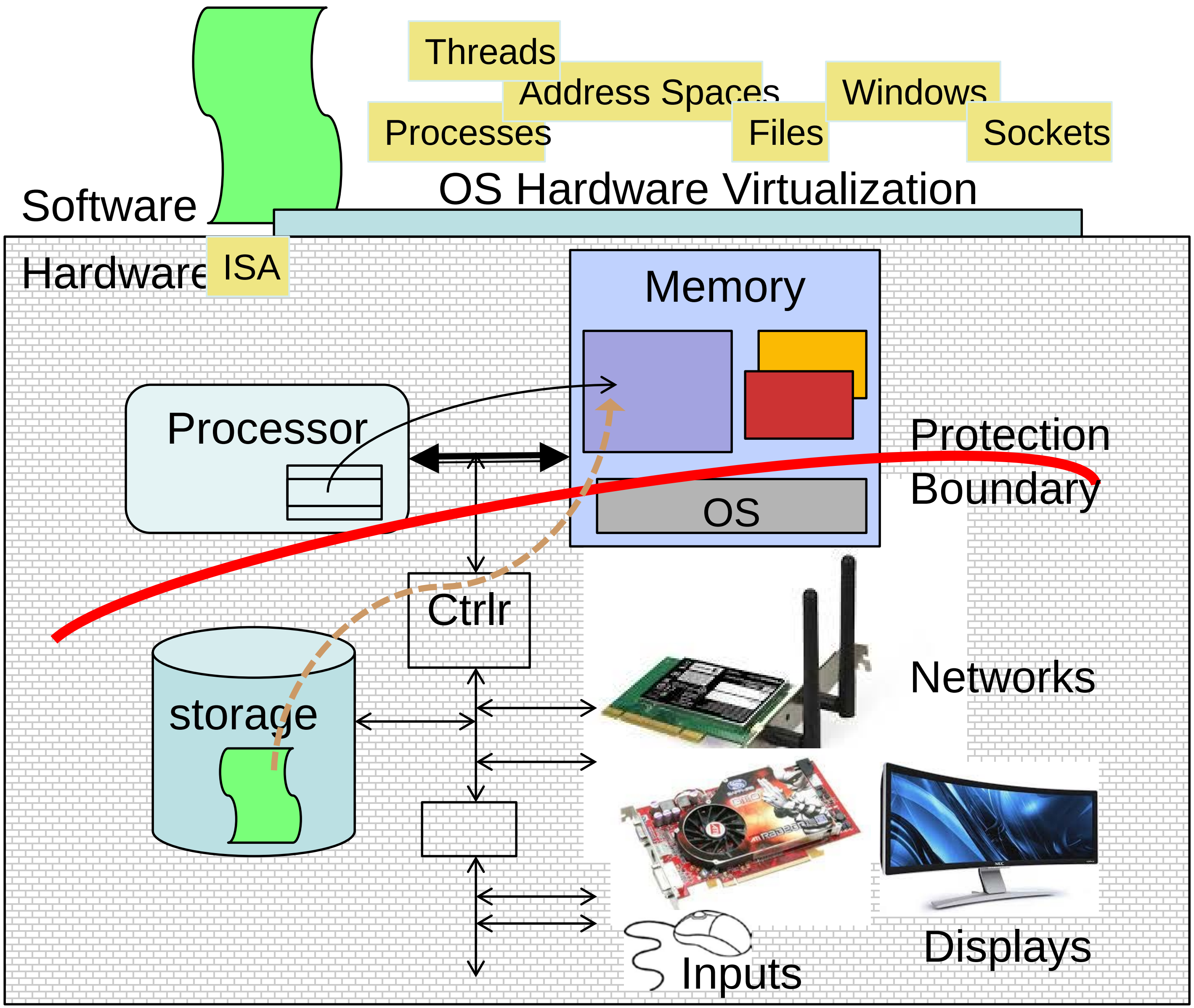
Outline

- Background
- Protection: Address Spaces
  - What is an Address Space?
  - How is it Implemented?
- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Inverted page tables
- Comparison among options

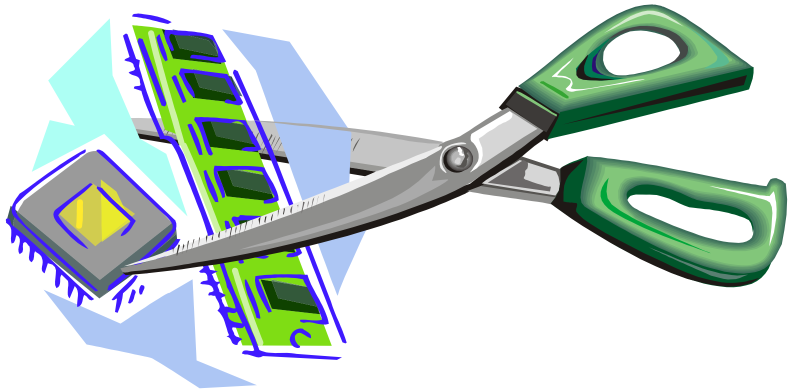
Background

- Program must be brought (from 2nd memory like HDD/SSD) into main memory and placed within a process for it to be run
- Main memory and registers are only storage units CPU can access directly in almost no time
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests

Loading a program



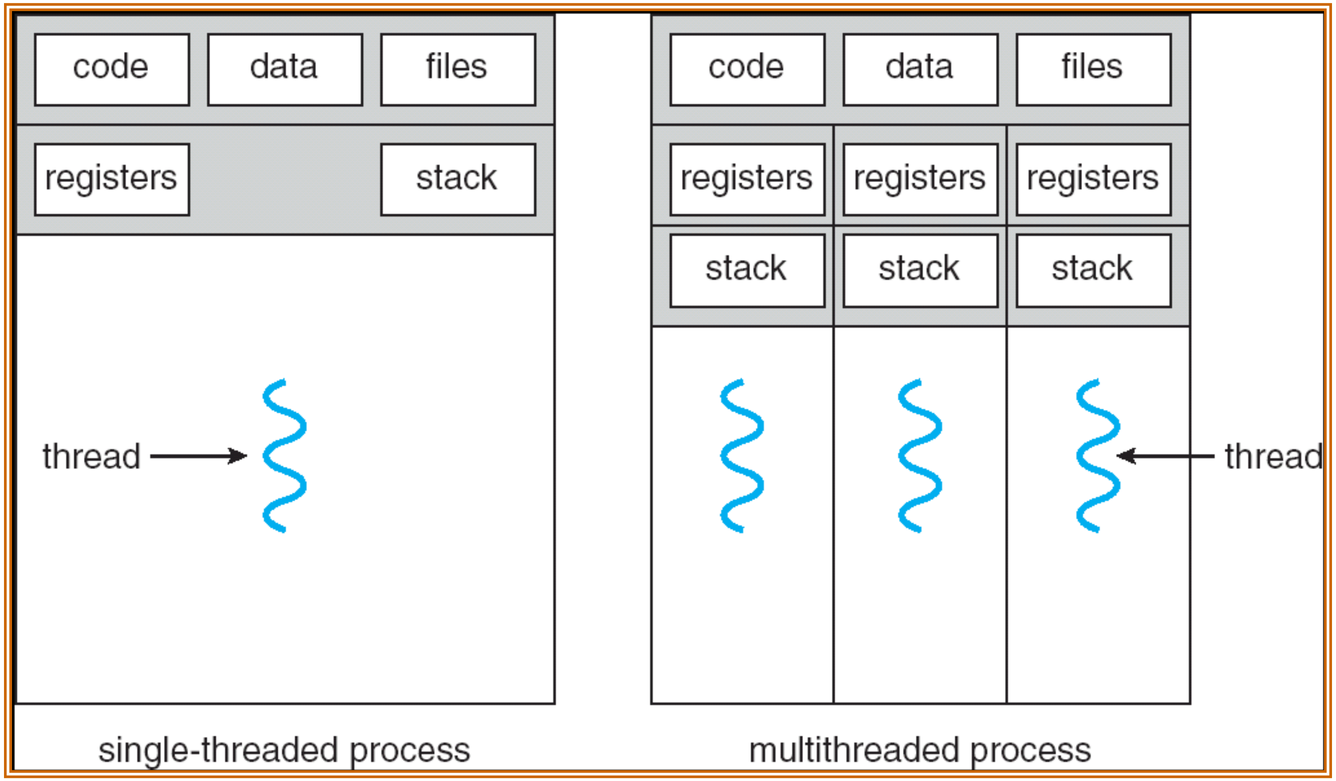
Virtualizing Resources



- Physical Reality:

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

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

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!
- 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

Binding of Instructions and Data to Main Memory

Process/User view of memory:
MIPS Assembly Code

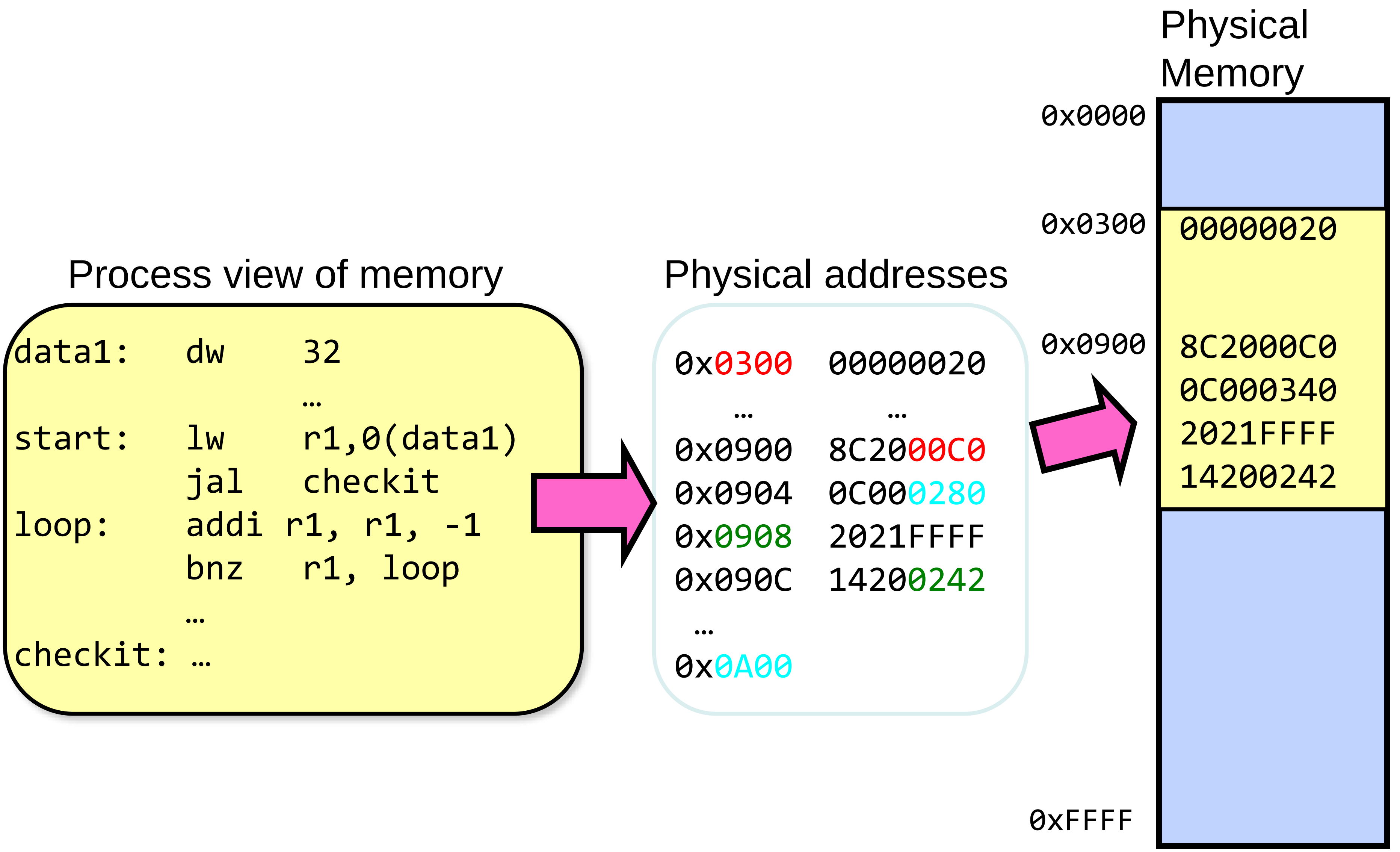
```
data1:   dw    32
               …
start:   lw    r1,0(data1)
         jal   checkit
loop:    addi r1, r1, -1
         bnz   r1, loop
         …
checkit: …
```

Physi

```
0x0300  00    20
   …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
 …
0x0A00
```

```
Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000
```

Binding of Instructions and Data to Memory



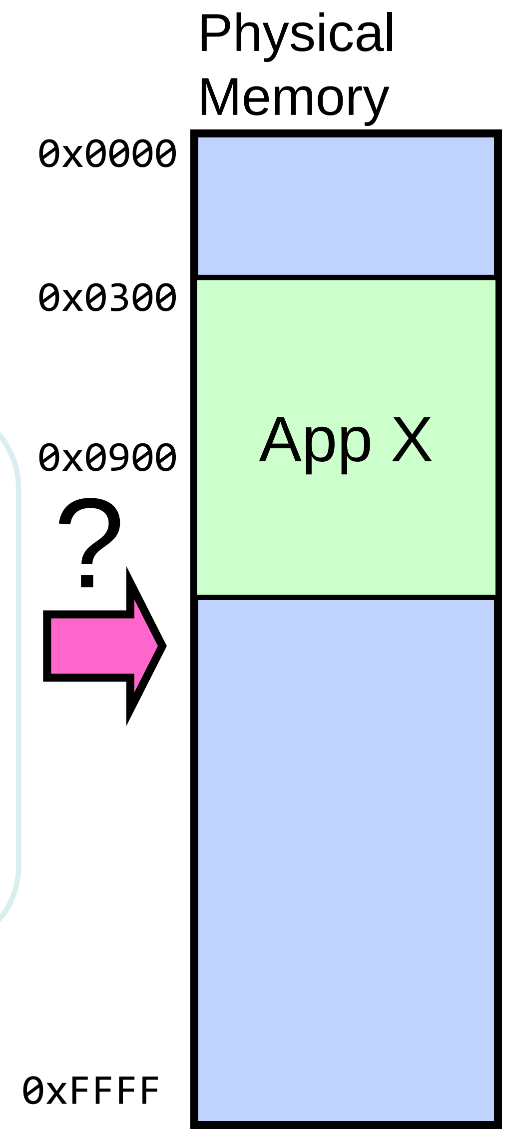
Second copy of program from previous example

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

```
0x0300  00000020
   …       …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
 …
0x0A00
```



Need address translation!

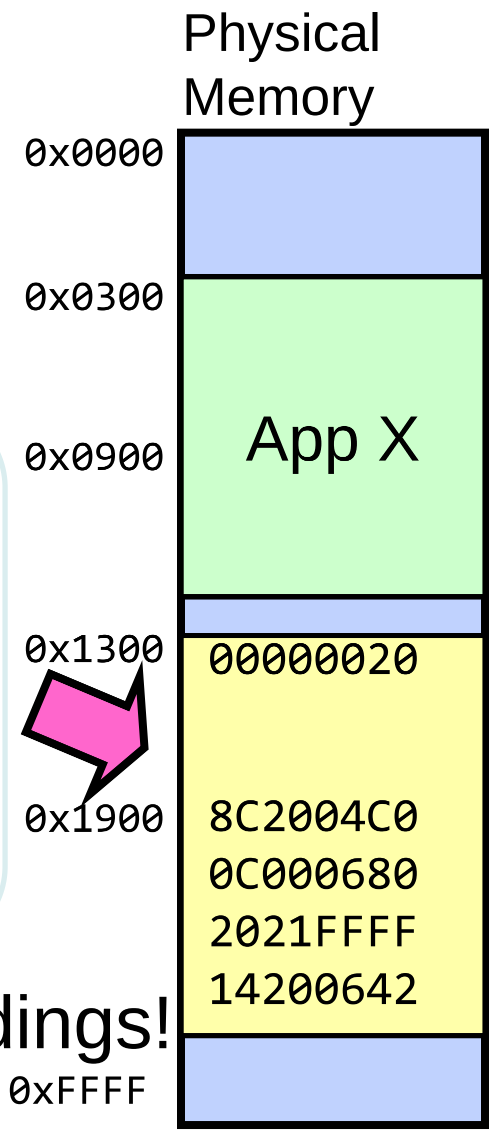
Second copy of program from previous example

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

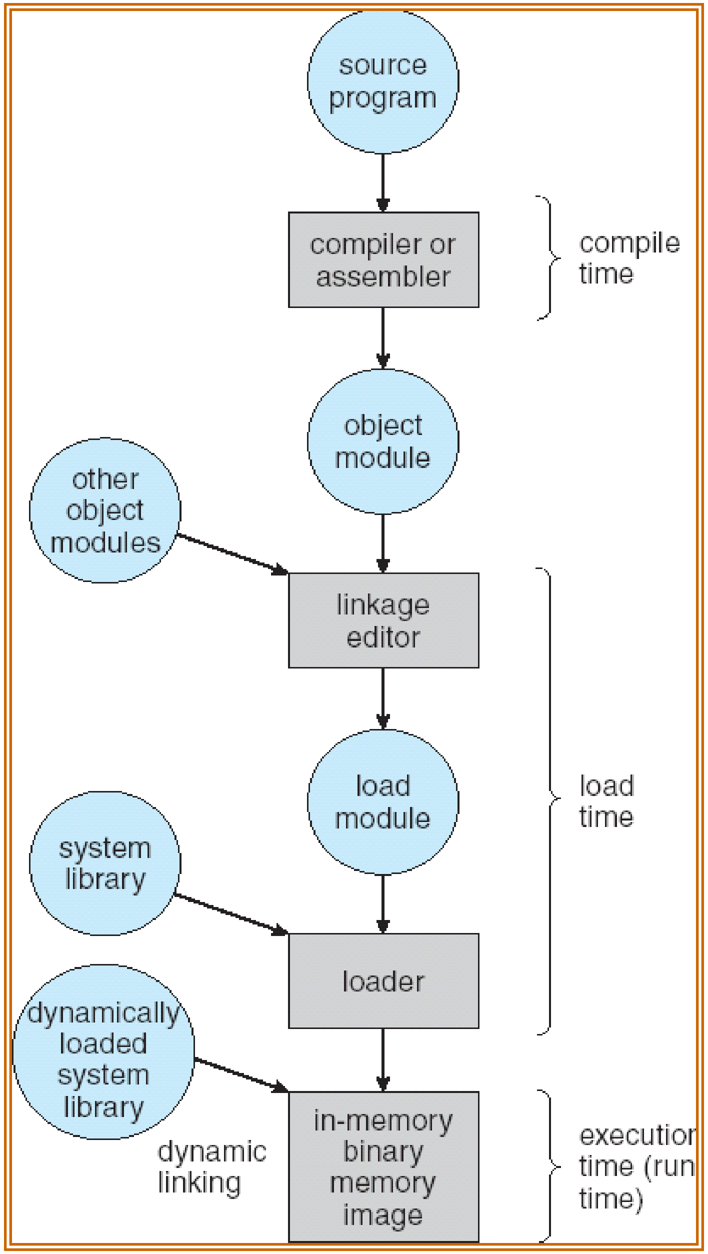
```
0x1300   00000020
   …        …
0x1900   8C2004C0
0x1904   0C000680
0x1908   2021FFFF
0x190C   14200642
 …
0x1A00
```



- One of many possible translations/bindings!
- When does binding take place?
    Compile time, 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)
- Dynamic Libraries (DLL)
  - 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
  - So, all processes share only one copy of the shared library code
  - It improves both RAM and disk utilization
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system

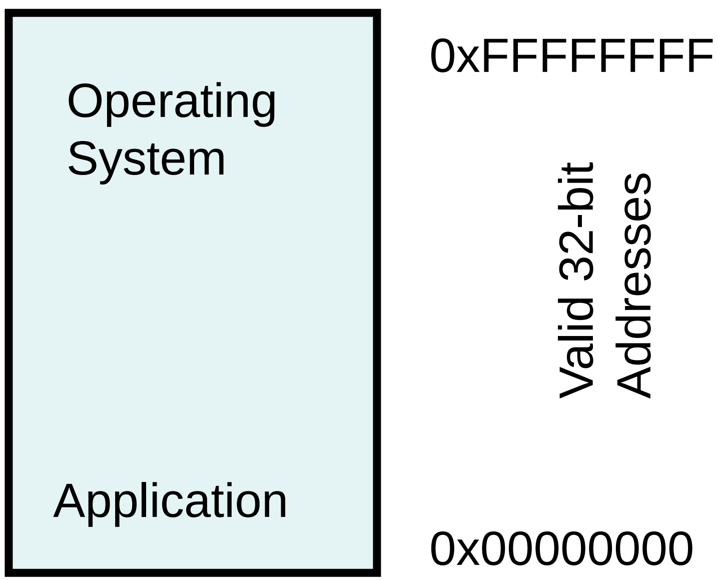


Binding of Instructions and Data to Memory

- Source program generally contains symbolic addresses
  - E.g., *pid, count, i, j*
- Binding (mapping from one address space to other) can happen at 3 different stages and hence addresses may be represented in different ways
  - **Compile time**: If main memory location known a priori, absolute addresses can be generated
    - Must recompile the code if starting location changes
  - **Load time**: Must generate *relocatable* code if main memory location is not known at compile time
    - e.g., "10 bytes from the start of process CODE block"
    - Linker/loader binds relocatable addresses to absolute addresses
      - Symbol table in the compiled file lists address values that need to be modified
    - Must reload the compiled code if starting location changes
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address mappings (e.g., MMU, *base* and *limit registers, page/segment table*)
    - Most common in general-purpose OSs where compiler generates relocatable addresses and then linker/loader generates absolute addresses

Recall: Uniprogramming

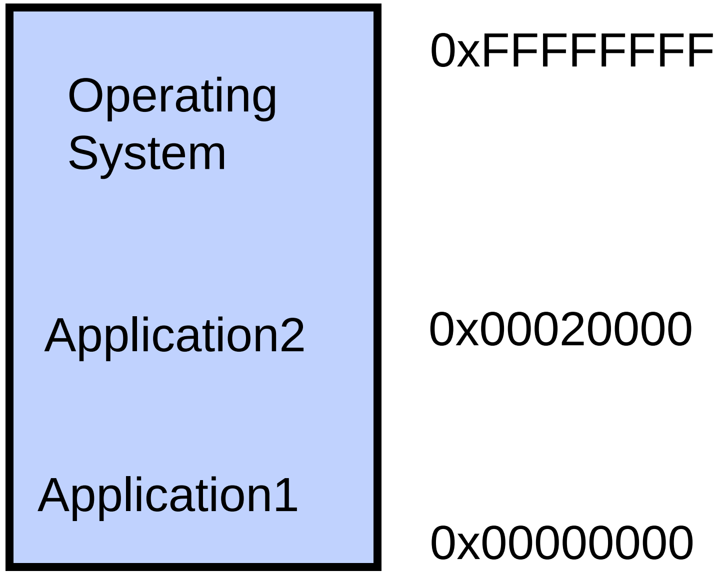
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address

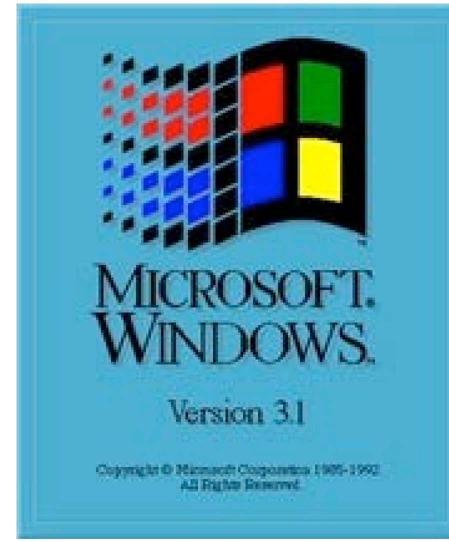


  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Multiprogramming (primitive stage)

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

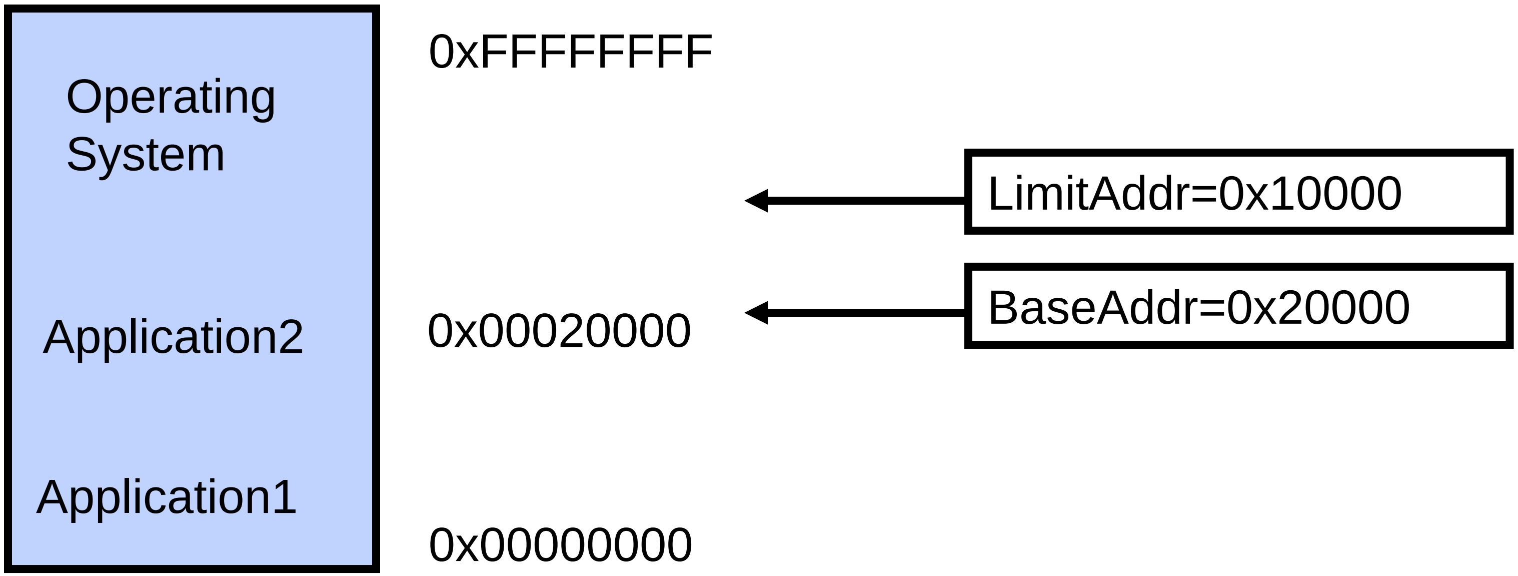




  - Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - Everything adjusted to memory location of program
    - Translation done by a linker-loader (relocation)
    - Common in early days (… till Windows 3.x, 95?)
- With this solution, no protection: bugs in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

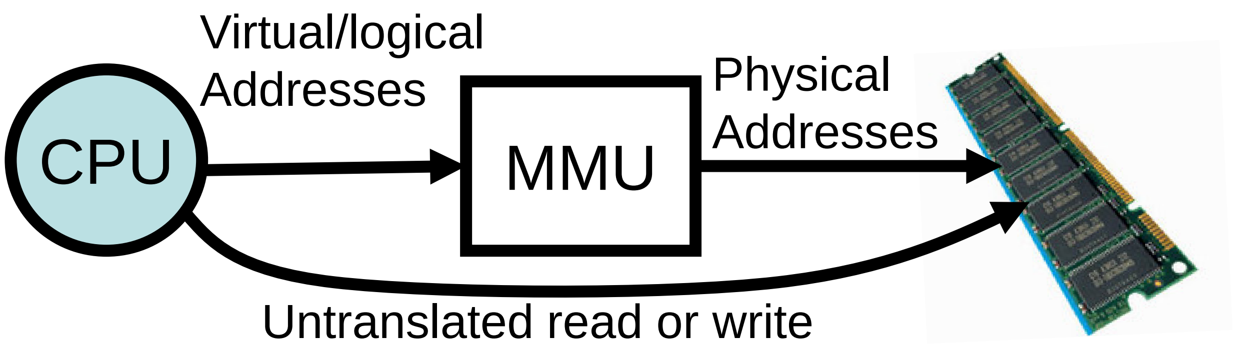
Multiprogramming (with Protection)

- Can we protect programs from each other without address translation?



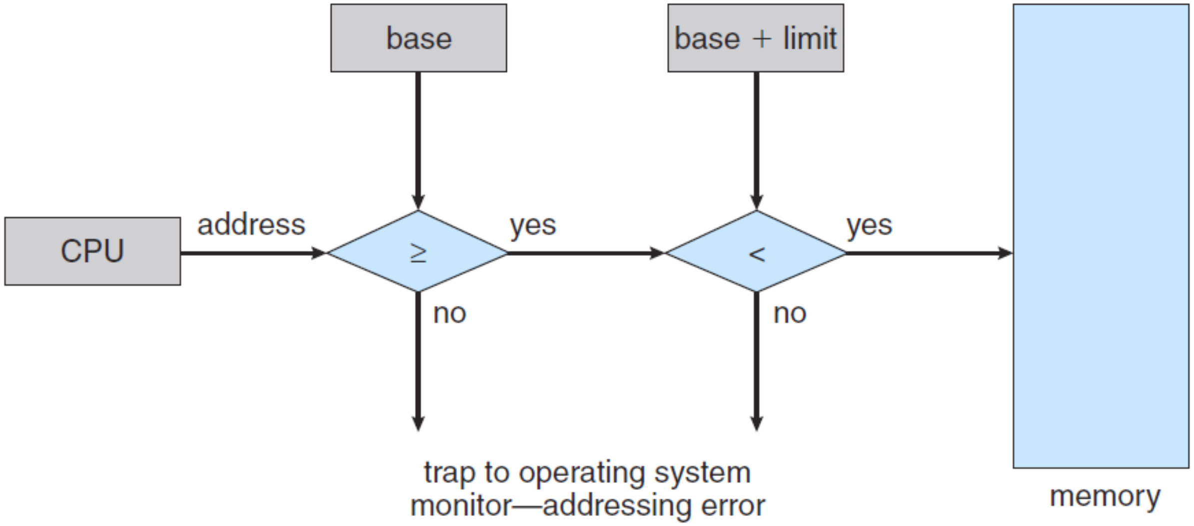
  – Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
    - Cause error if user tries to access an illegal address
  – During switch, kernel loads new base/limit from PCB (Process Control Block)
    - User not allowed to change base/limit registers

Recall: General Address translation



- Recall: Address Space:
  – All the addresses and state a process can touch
  – Each process and kernel have different address spaces
- Consequently, two views of memory:
  – View from the CPU (what program sees, logical/virtual memory addresses)
  – View from memory unit (physical memory addresses)
  – Translation box (MMU) converts between the two views

Simple Contiguous Memory: Base and Limit



https://en.wikipedia.org/wiki/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 → much easier to implement protection!
- 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 changed when program placed in different region of DRAM
- During context switch, kernel loads new base/limit from PCB
  - User not allowed to change base/limit registers

Base and Limit contiguous memory discussion

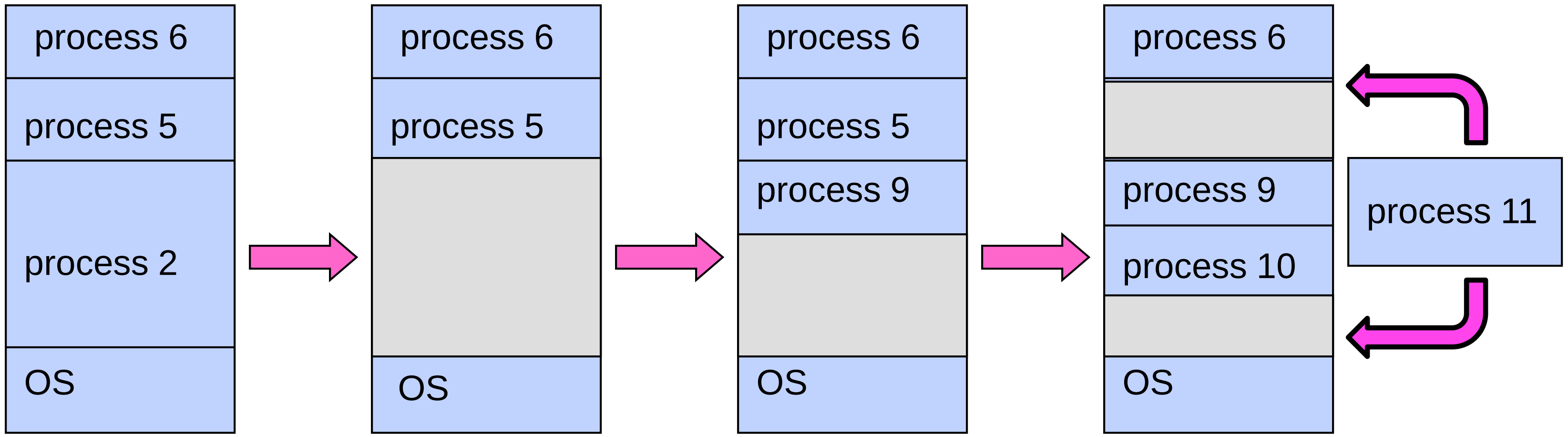
- Base and Limit Pros: Simple, relatively fast
- Provides level of indirection
  - OS can move bits around behind program's back
  - Can be used to correct if program needs to grow beyond its bounds or coalesce fragments of memory
- Only OS gets to change the base and limit!
  - Would defeat protection
- What gets saved/restored on a context switch?
  - Everything from before + base/limit values
  - Or: How about complete contents of memory (out to disk)?
    - Called "Swapping"
- Hardware cost
  - 2 registers/Adder/Comparator
  - Slows down system because it need to do add/compare on every memory access for fetching instructions and Data

Dynamic Storage-Allocation Problem

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization. First-fit is faster than best-fit.

Issues with Simple Contiguous Address Method

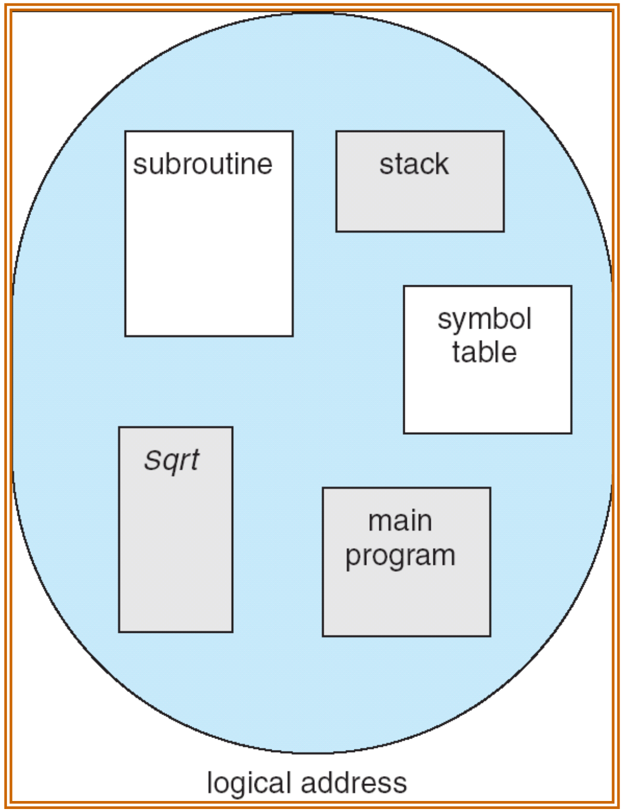


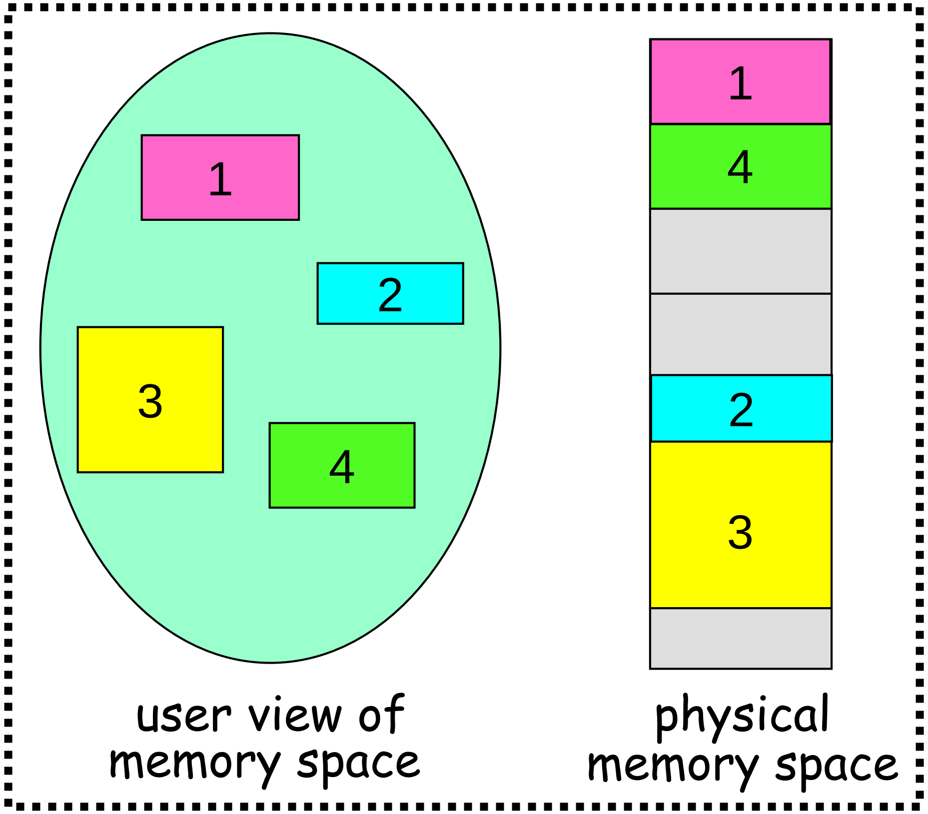
- Fragmentation problem over time
  – Not every process is same size $\Rightarrow$ memory becomes fragmented over time with small holes
  – Really bad if want space to grow dynamically (e.g. heap and stack)
- Missing support for sparse address space
  – Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc) → non-contiguous memory allocation
- Hard to do inter-process sharing
  – Want to share code segments when possible
  – Want to share memory between processes

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - First-fit and best-fit suffer most from this
  - One Solution: Compaction (not always possible) only for execution time binding
  - 2nd Solution: let process to get its DRAM allocated in non-contiguous fashion →Segmentation, Paging
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Hole of 18,464 B, process of 18,462 B
  - Overhead to keep track of hole is substantially larger than the hole itself!

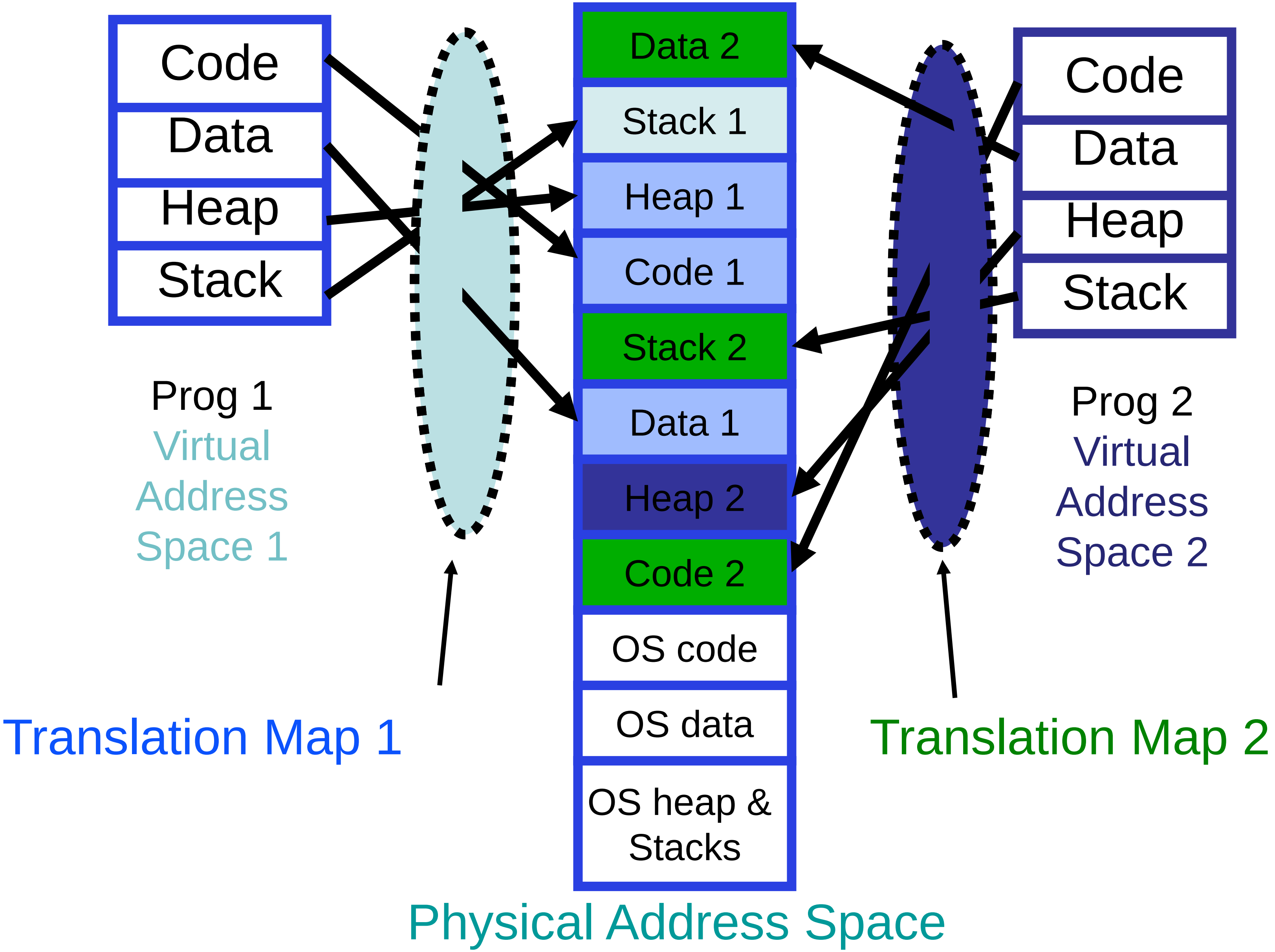
Segmentation



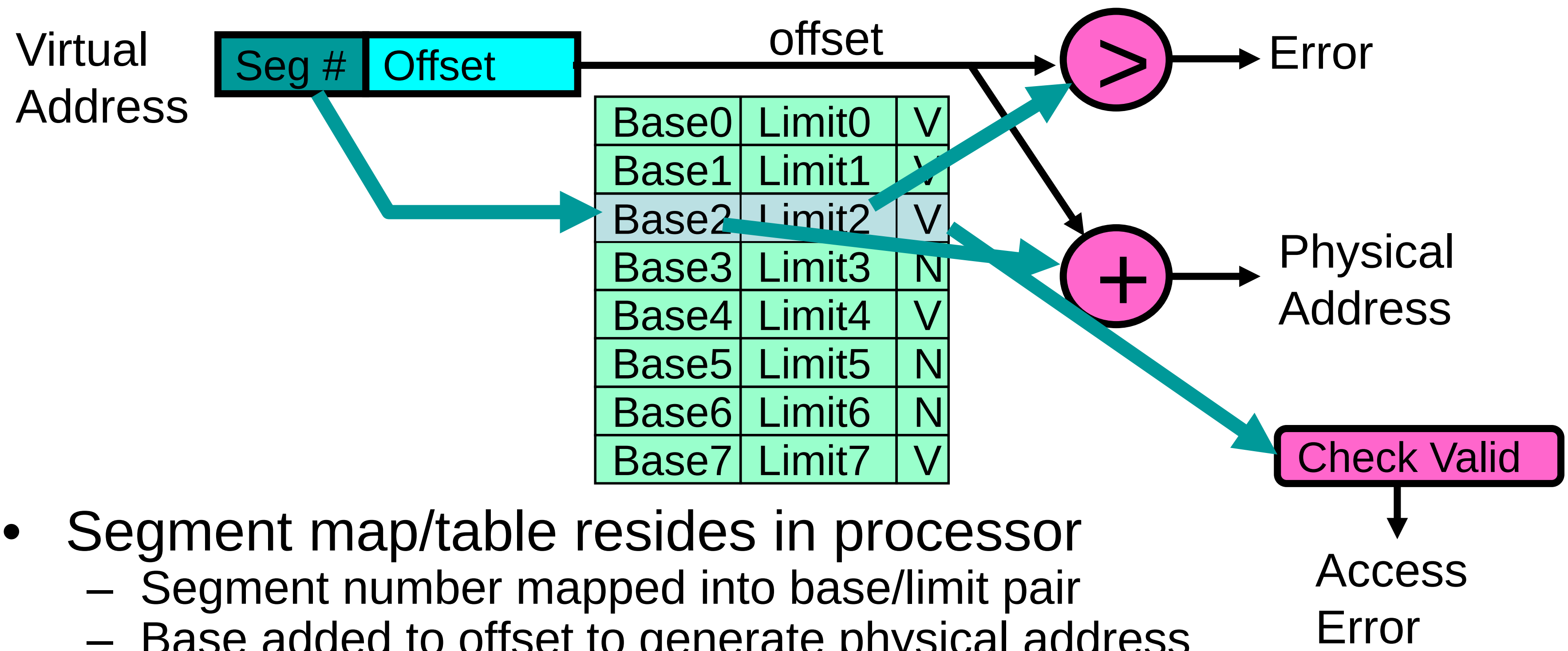


- Logical View: multiple separate segments
  - Typical: Code, Data, Stack per each thread, Heap
  - Others: memory sharing, std C library
- Each segment is given a region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

Recall: General Address Translation



Implementation of Multi-Segment Model

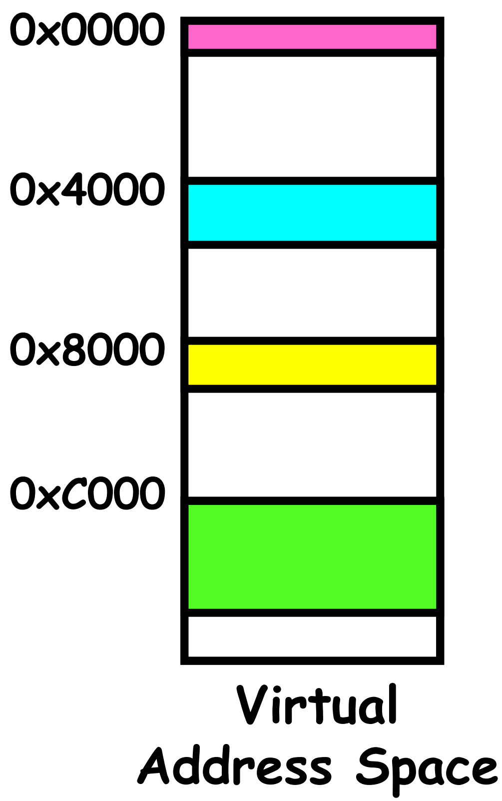


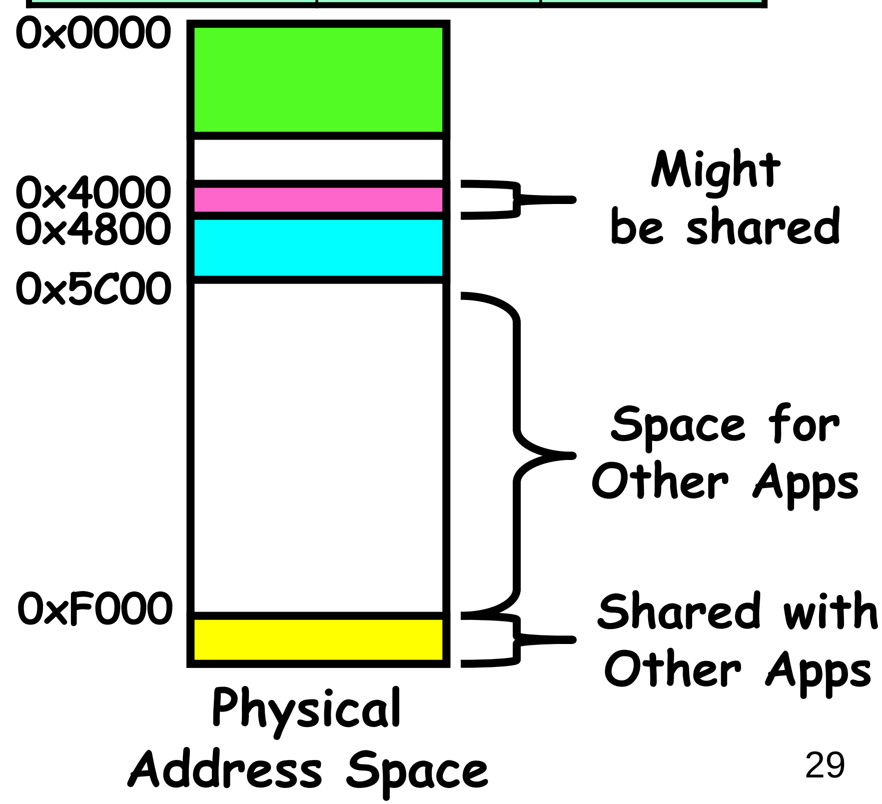
- Segment map/table resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out-of-range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: mov ax, [ds:bx]
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well

Example: Four Segments (16 bit addr)

Segment Table

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

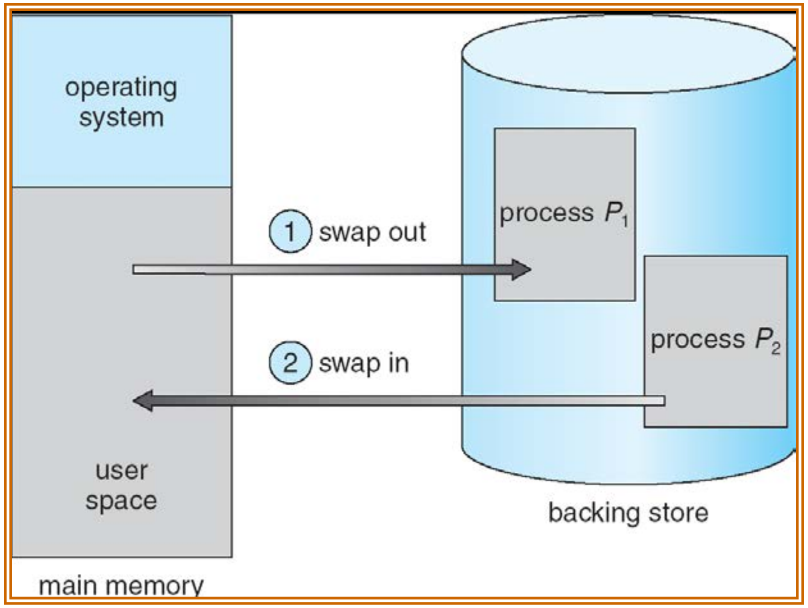




Observations about Segmentation

- Sharing of segments among processes to achieve
  - sharing of CODE/libraries and to realize IPC
- 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 → segmentation fault
- 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)
  - Or segment table base register (STBR)
  - 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
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed next

- Desirable alternative to Segmentation?
  - 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 ("**page frames**")
    - Size is power of 2, between 512 B and 1 GB
  - 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
      $1 \Rightarrow$allocated, $0 \Rightarrow$free
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames in physical memory
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - Typically have small pages (4K-16K)
  - Consequently: need multiple pages/segment
- To run a program of size ***N*** pages, need to find ***N*** free frames to load it

Address Translation Scheme

❖ Address generated by CPU is divided into:

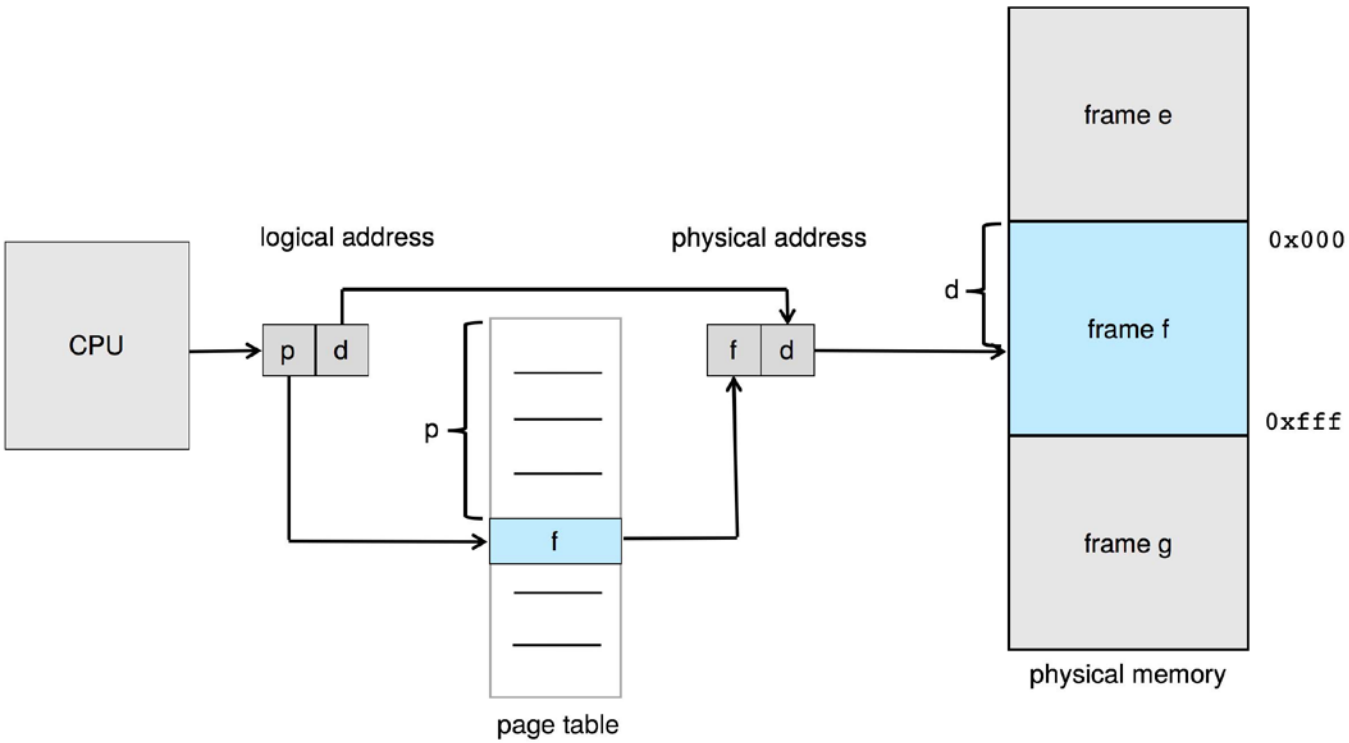
– **Page number** (***p***) – used as an index into a **page table** which contains base address of each page (aka page frame ID) in physical memory

– **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	d
m - n	n

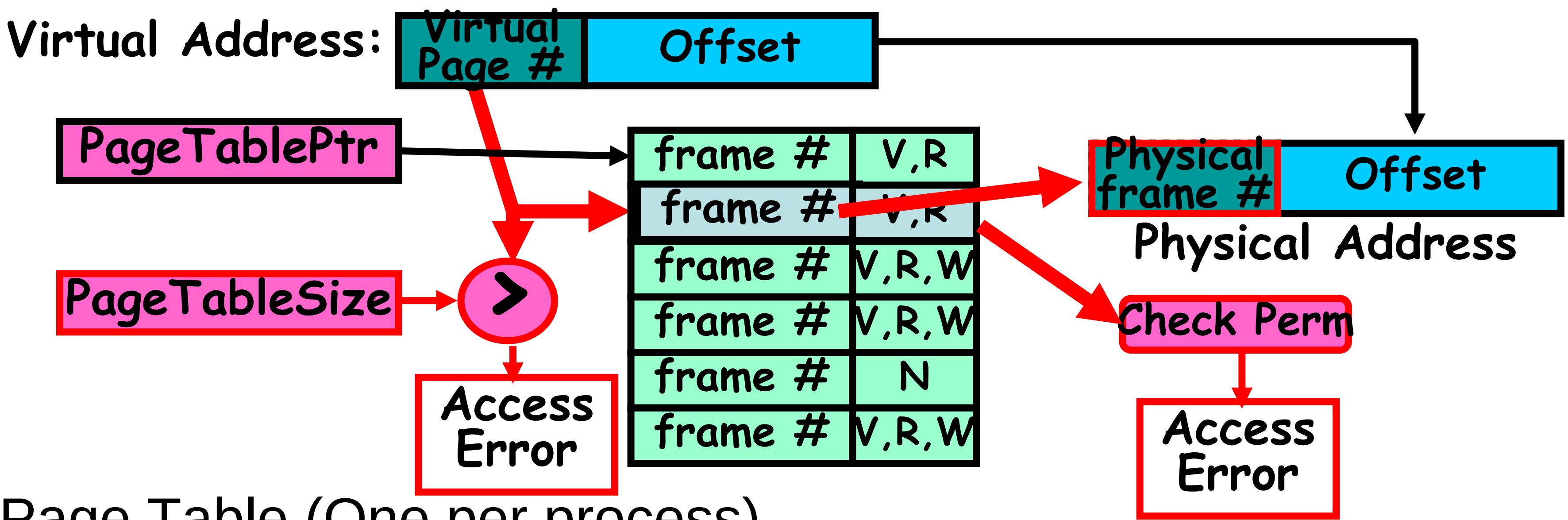
– A for given logical address space 2^m and page size 2^n

Paging Hardware



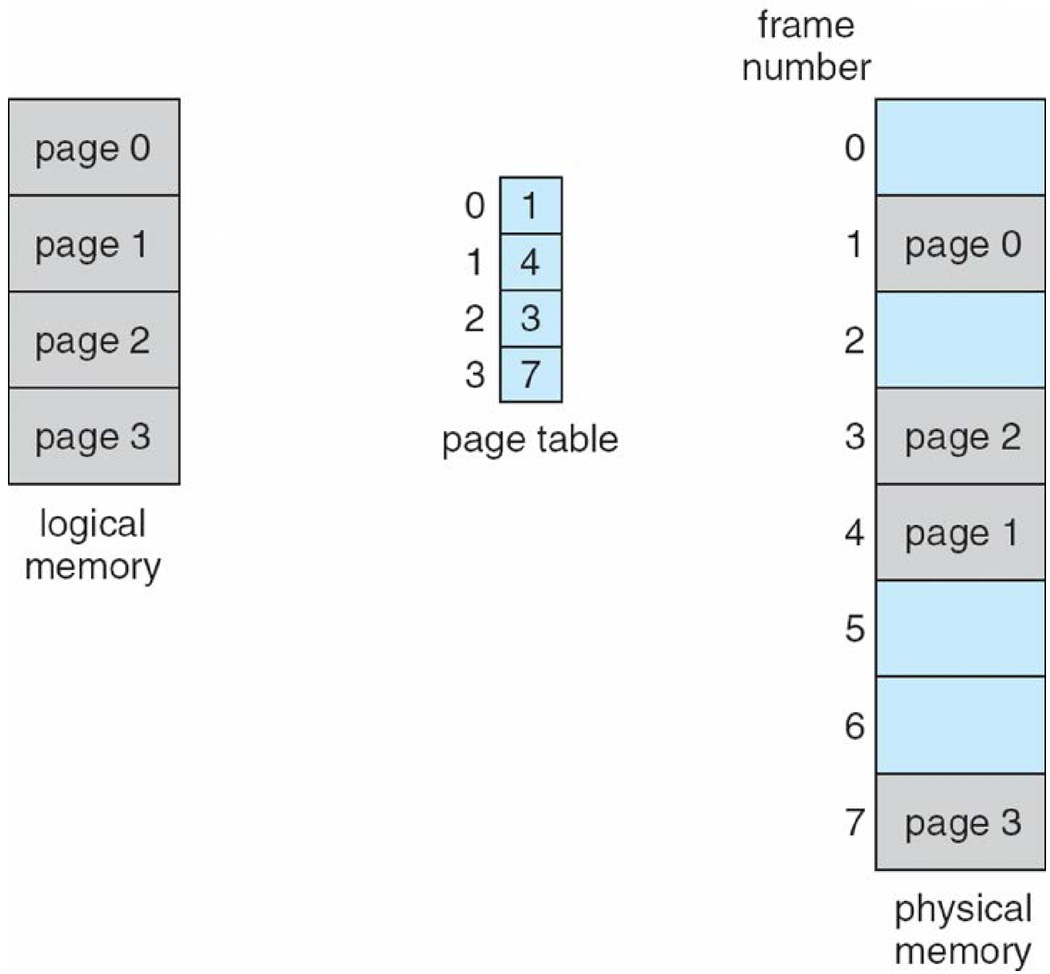
- **Page table** to translate logical to physical addresses
- Backing store (HDD) likewise split into pages of same size

How to Implement Paging?

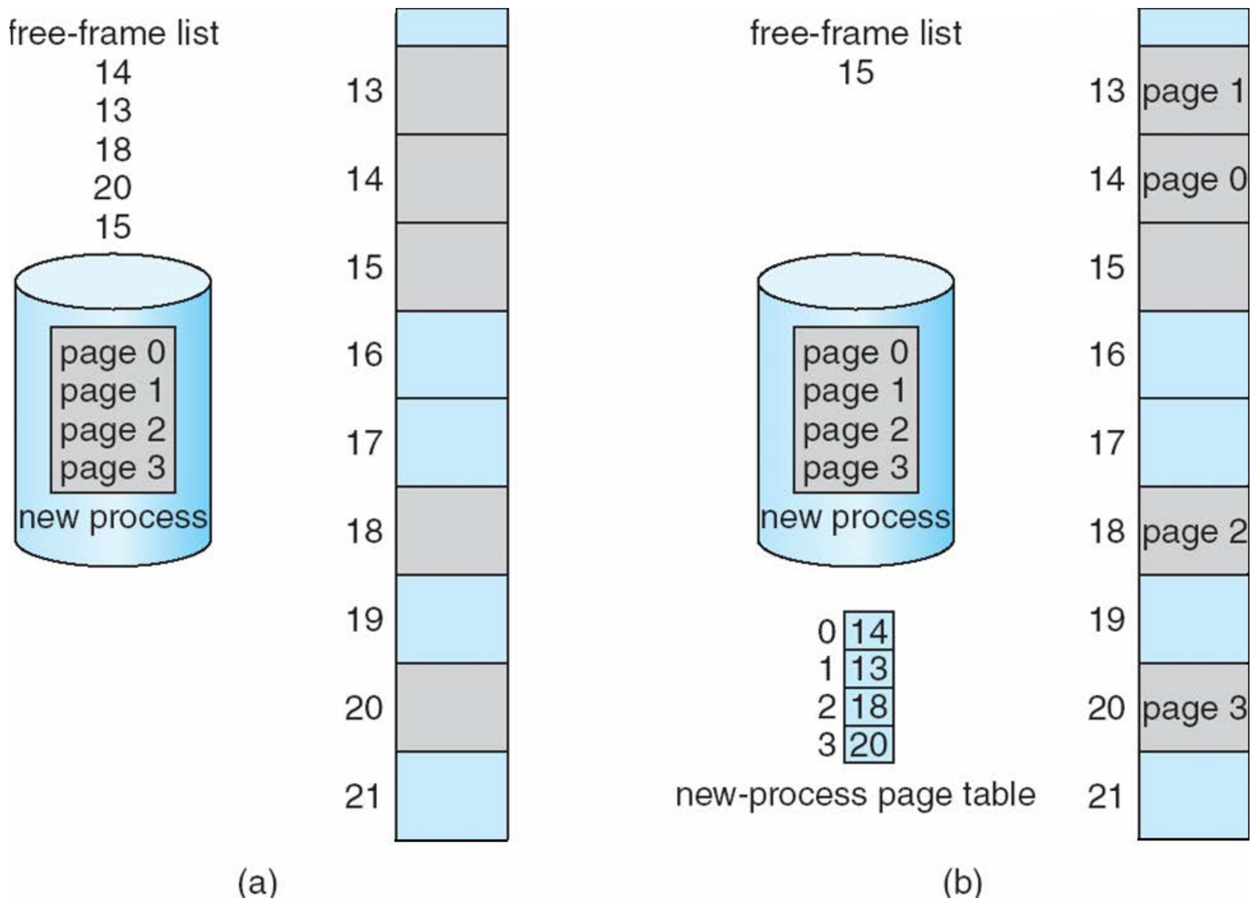


- Page Table (One per process)
  - Resides in physical memory!
  - Contains physical page (page frame) ID (#) and permissions for each virtual page
    - Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # (or simply page no) is all remaining bits
    - Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - Physical page (frame) # ID copied from table to get physical address
  - Check Page Table bounds and permissions

Paging Example



Free Frames

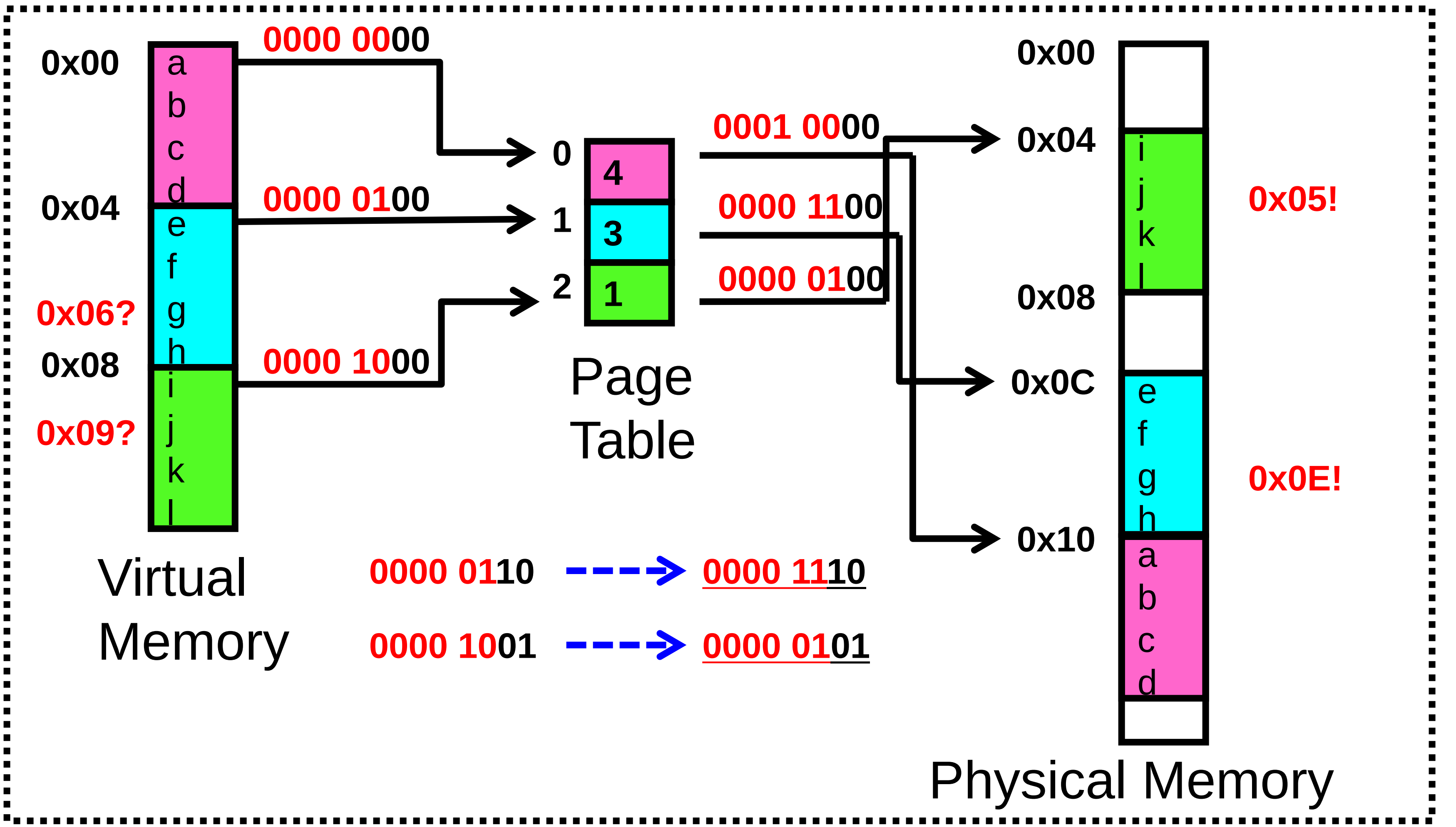


Before allocation

After allocation

Simple Page Table Example

Example (4 byte pages)

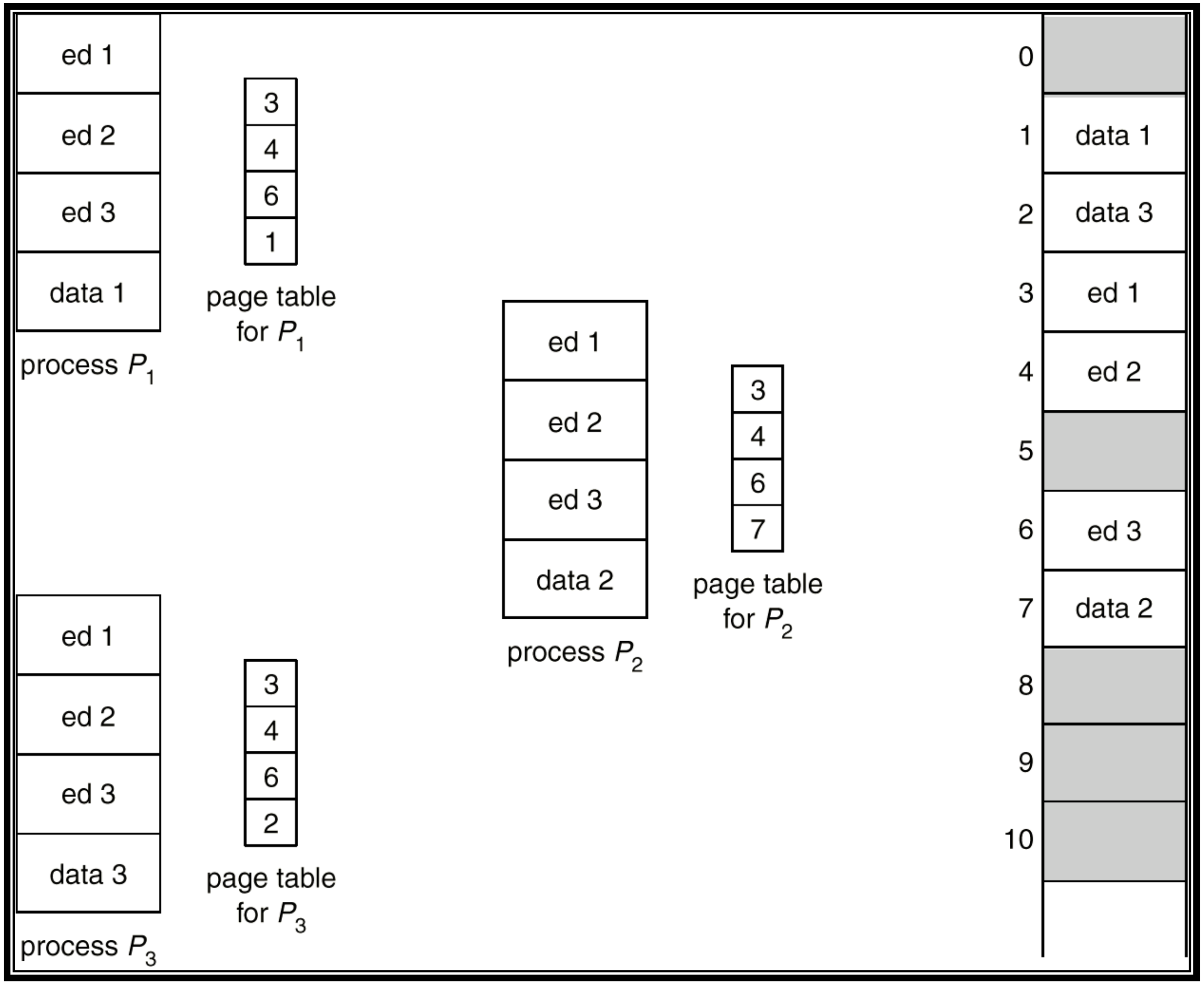


Shared Pages

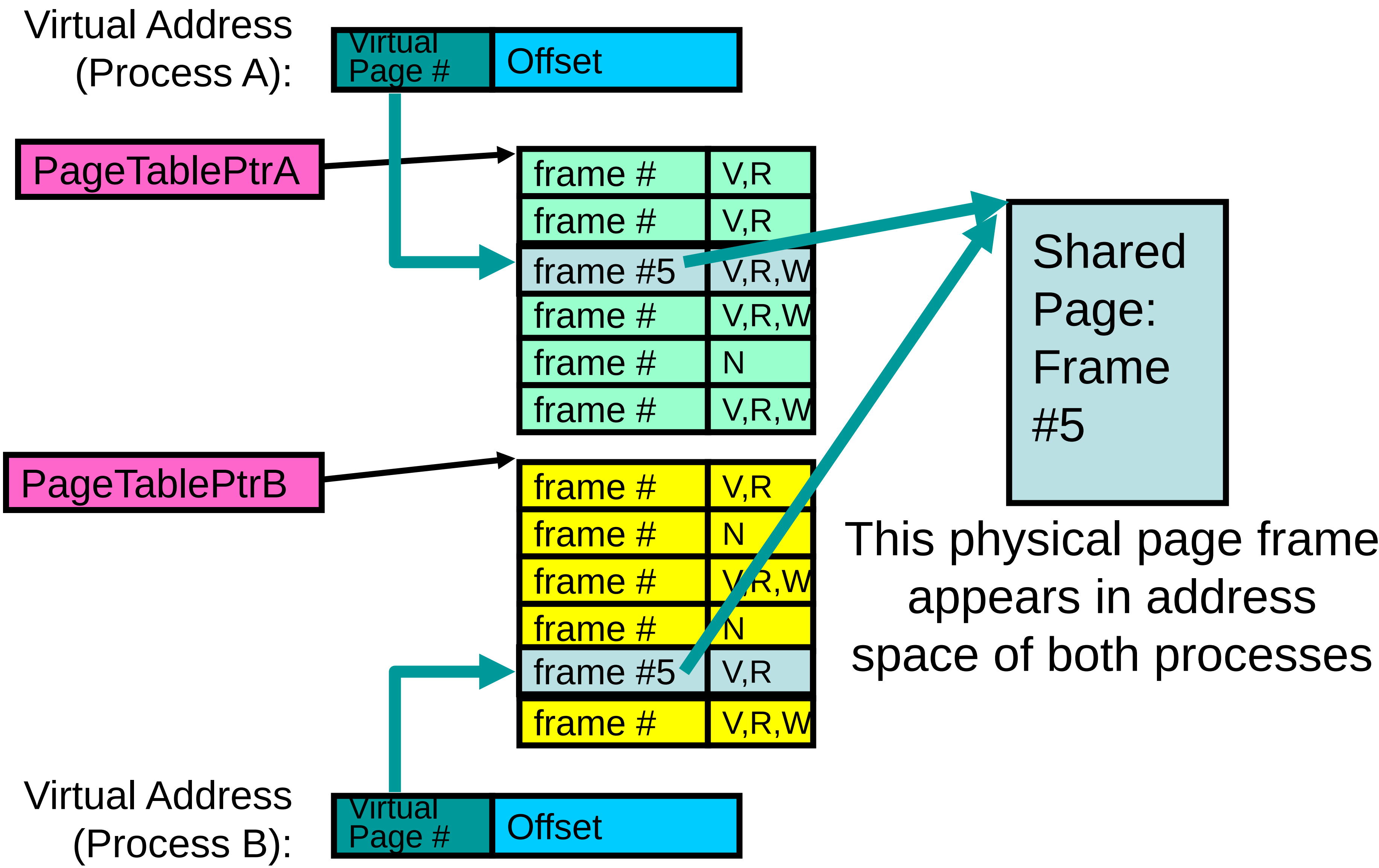
- Shared code
  - One copy of read-only (reentrant) code shared among processes
  - E.g., text editors, compilers, window systems

- Private code and data
  - Each process keeps a separate copy of the code and data

Shared Pages Example



What about Sharing?



Paging -- Internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame with 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

What is in a 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, execute-only
- Example: Intel x86 architecture PTE:
  – Addressing format (10, 10, 12-bit offset)
  – Top-level page tables called “Directories”

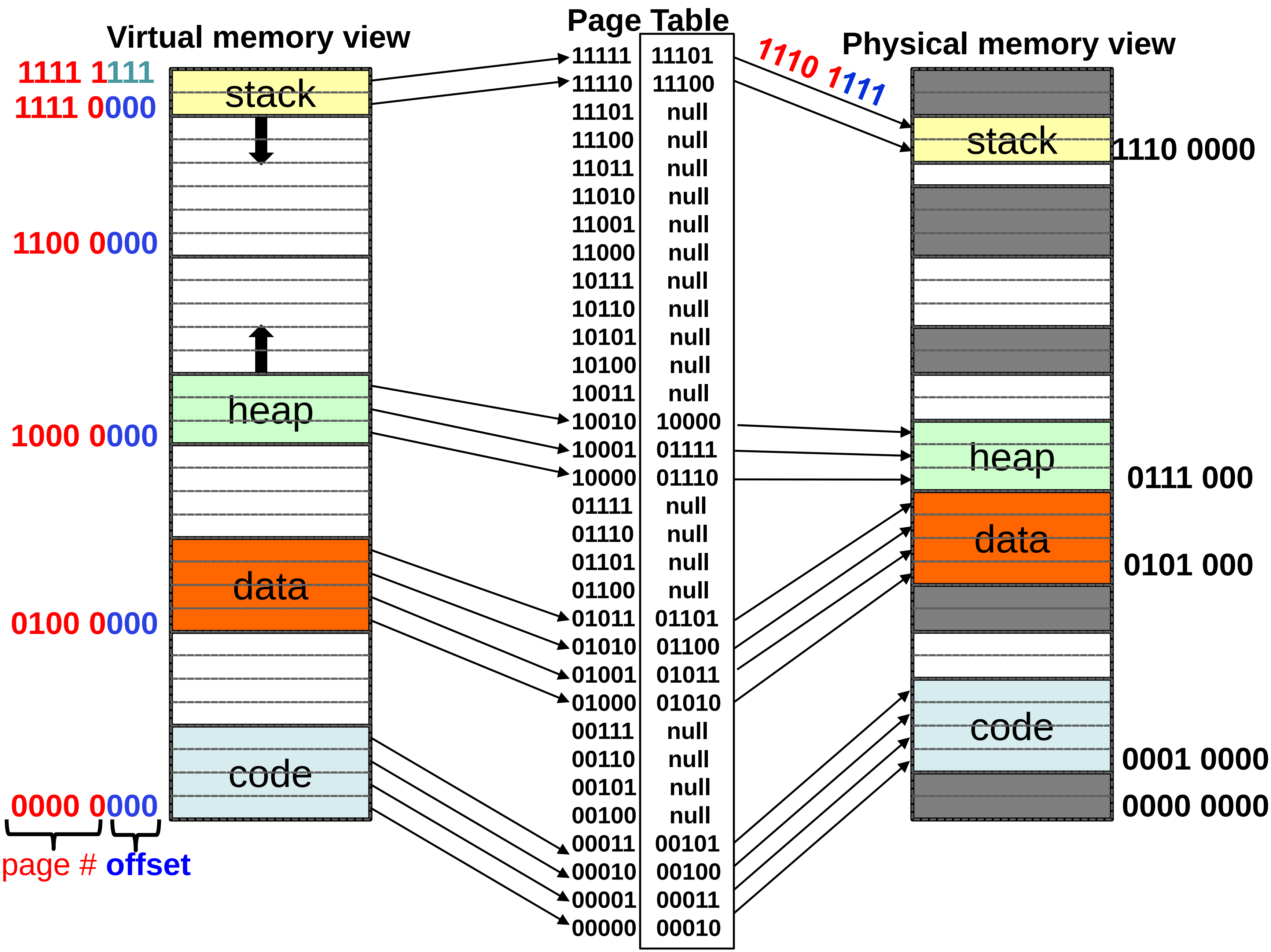
Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

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$\Rightarrow$4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

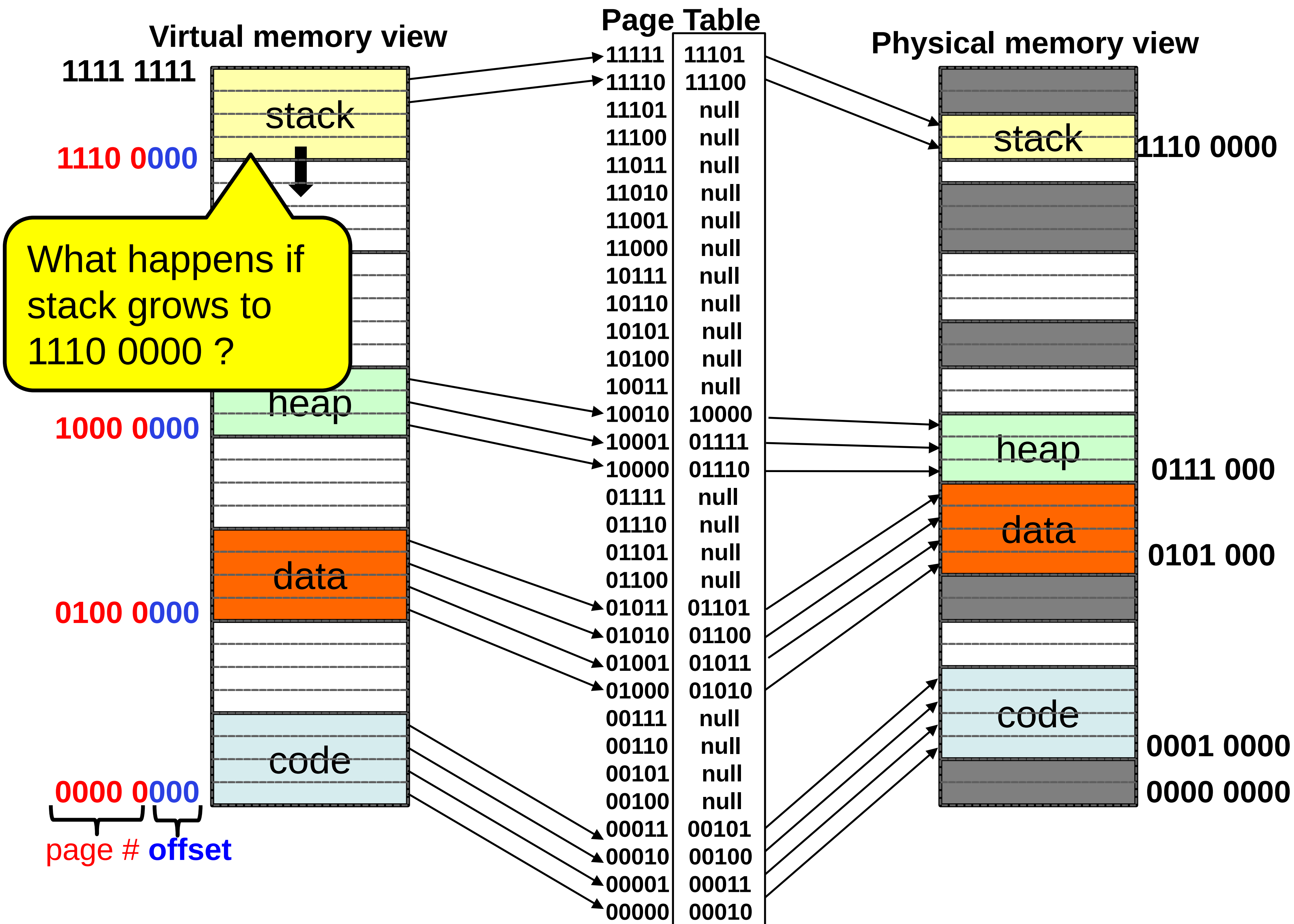
Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - Region of address space is actually invalid in virtual address space or
    - Page/directory is just somewhere else than in main 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

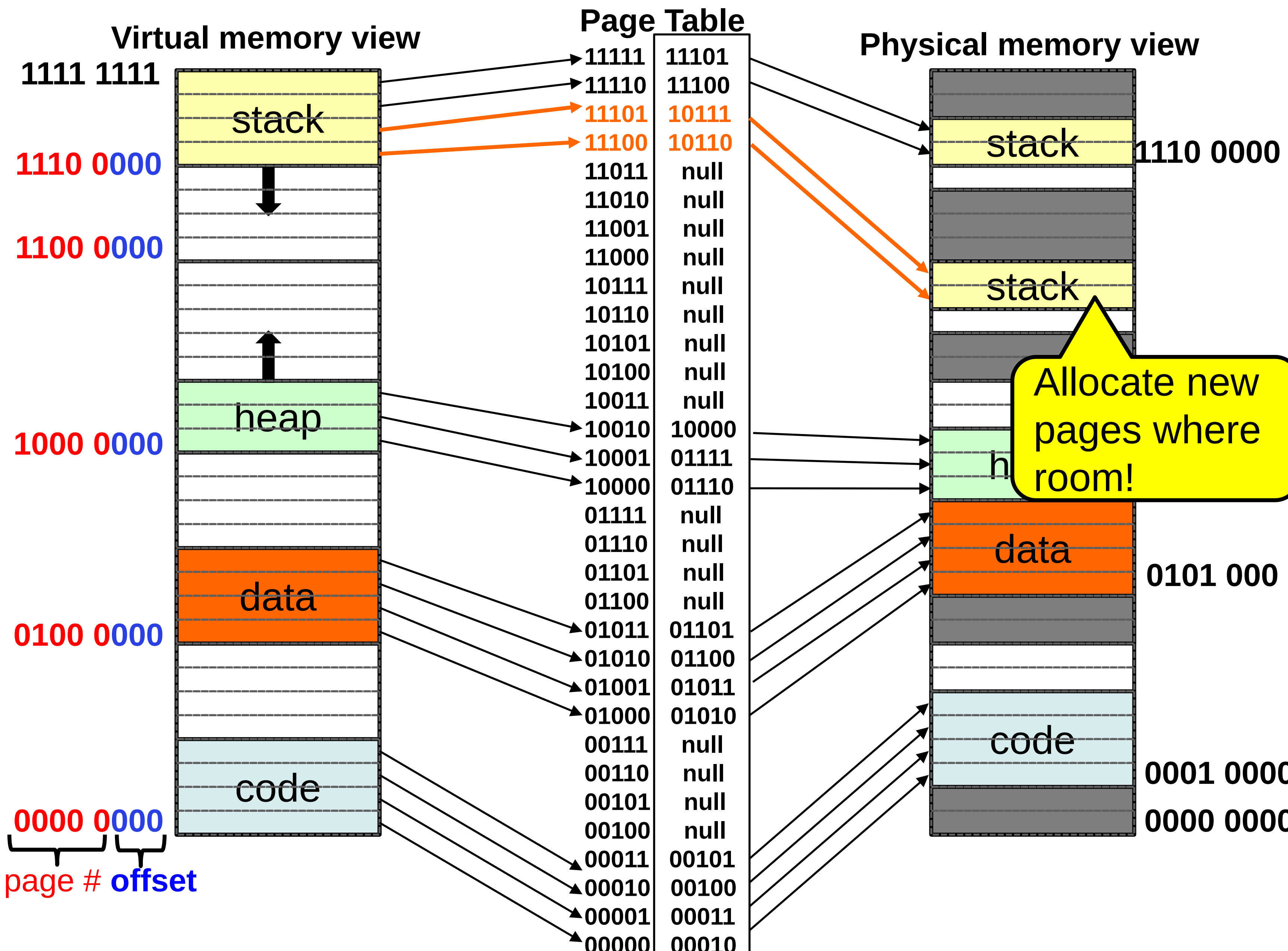
Summary: Paging



Summary: Paging



Summary: Paging



Paging Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit/length pointer
- Analysis
  - Pros
    - Page Table: An array of structures
    - Simplifies free-space allocation using bitmap
    - Easy to share by setting flags of those frames in PTs
    - Copy-on-Write, Zero Fill on Demand
  - Con: What if address space is sparse?
    - Stack and heap grow dynamically, so cause sparsity
    - E.g., on UNIX, code starts at 0, stack starts at (2^{31}-1)
    - With 1K pages, need 4 million page table entries!
      - With PTBR, PT needs to be stored contiguously in DRAM!
    - Multi-threading: more stacks, each needs to grow!
  - Con: What if table is really big?
    - 64-bit virtual address space → PT array is most empty
    - Not all pages used all the time ⇒ would be nice to have only the working set of page table in memory
- Better data structure than arrays for lookups in sparse address space?
  - Trees and hash tables
    - Multi-level translation: Multi-level paging or combining paging and segmentation

Page Table Structures

- Hierarchical Paging
- Paged Segmentation
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables.

- A simple technique is a two-level page table.

Two-Level Paging Example

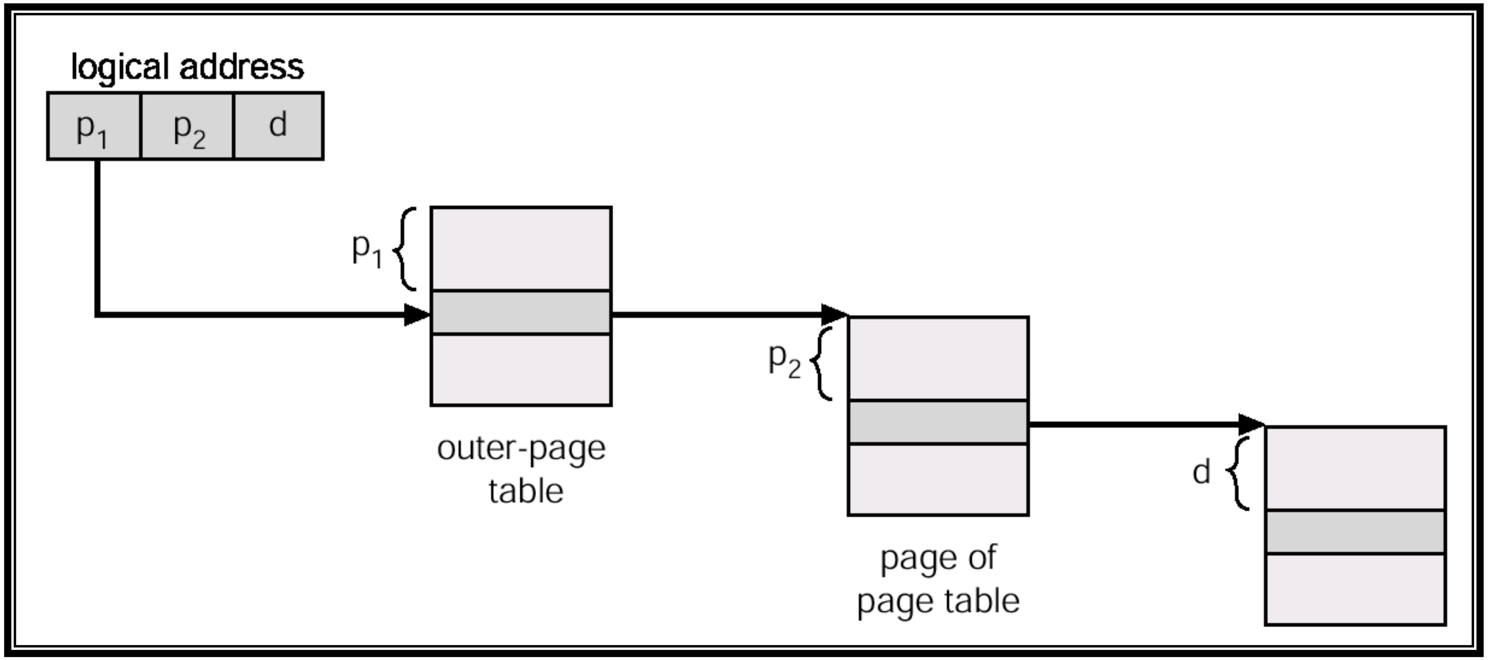
- A logical address (on 32-bit machine with 4KB page size and 4B page table entry) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Page table is too big (2^{22} B) to fit in one page frame (4 KB)
  - No of frames needed to store page table is 2^{10}
  - Page table of Page table contains 2^{10} entries and fits in 1 page frame
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

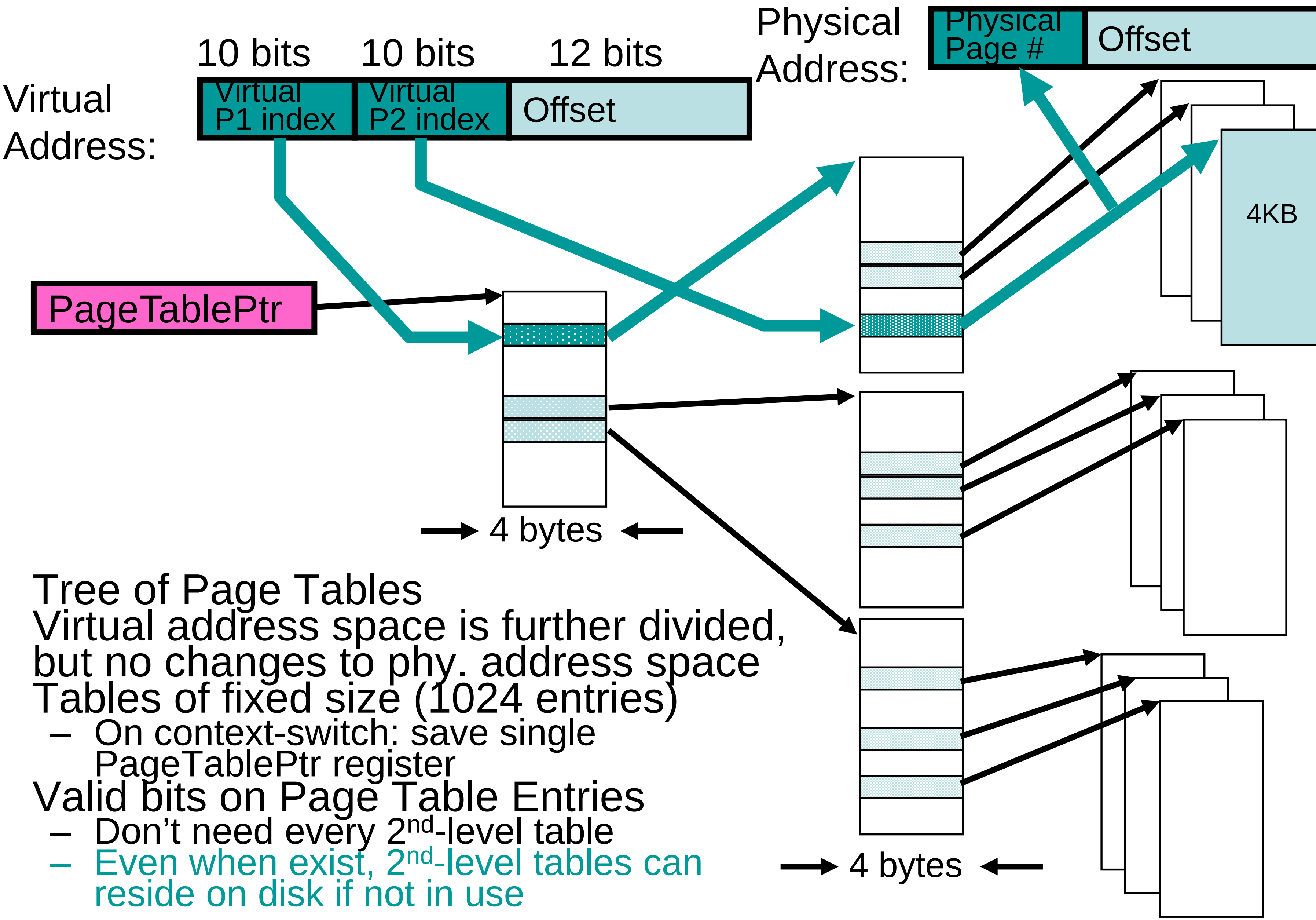
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

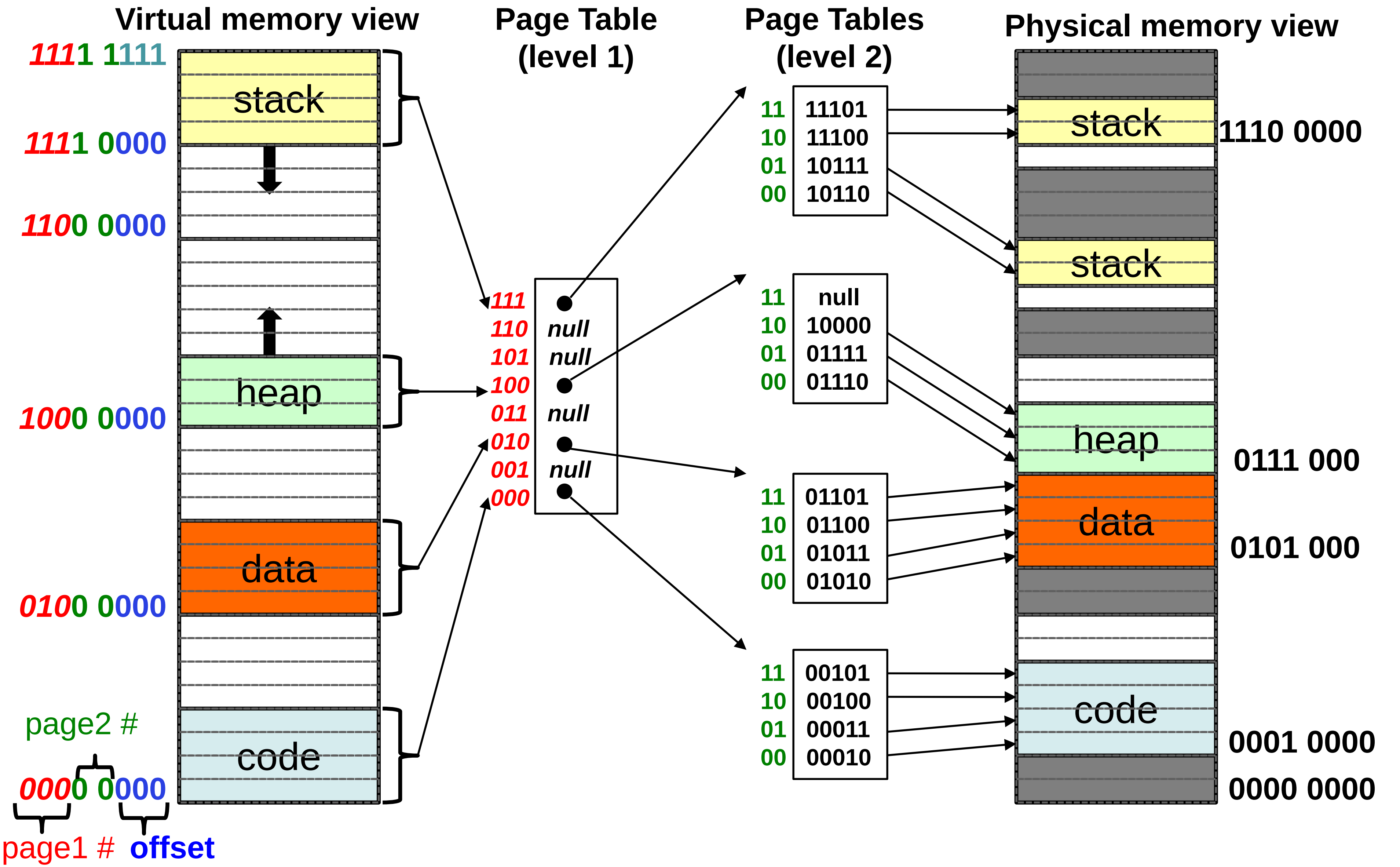


Fix for sparse address space: two-level page table

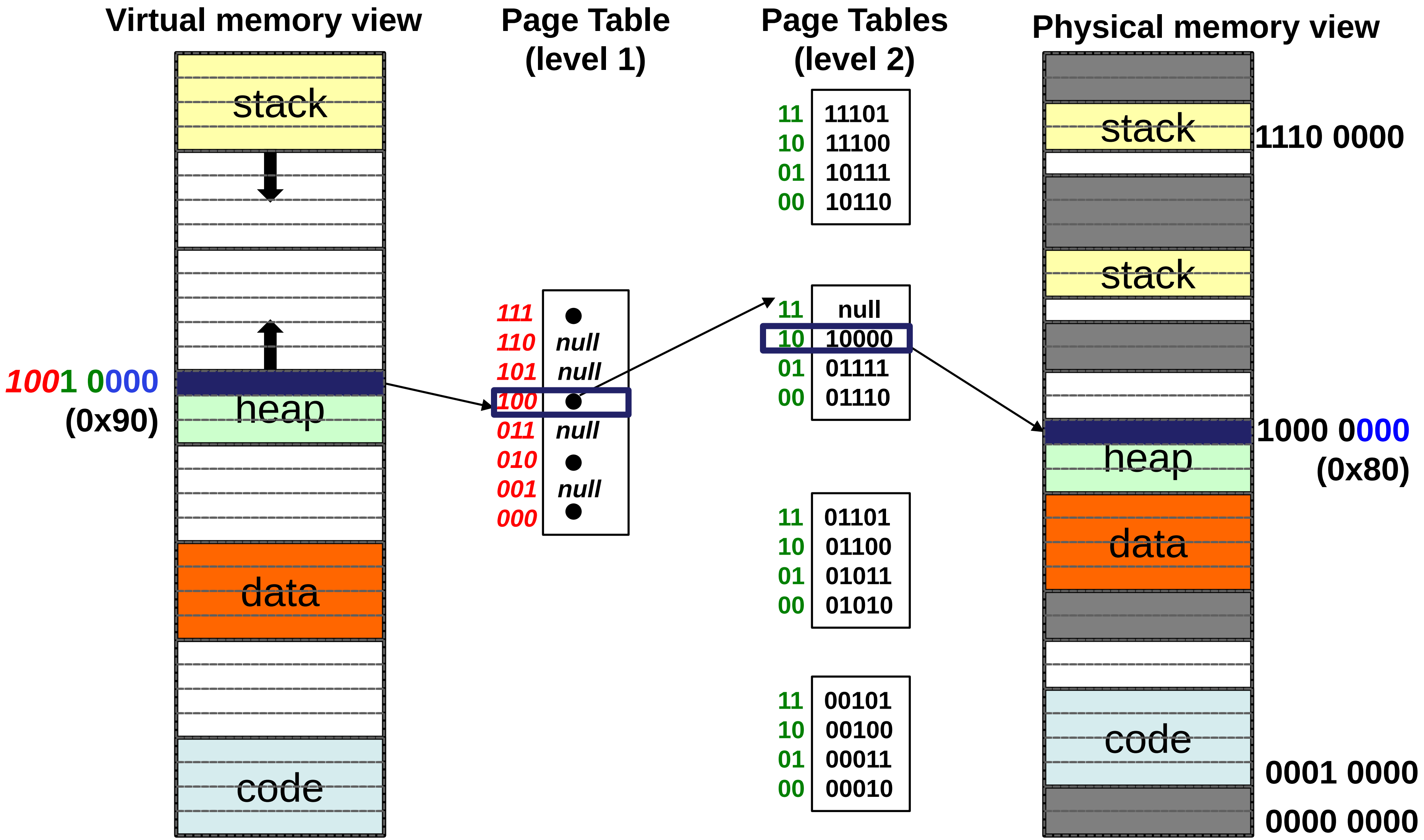


- Tree of Page Tables
- Virtual address space is further divided, but no changes to phy. address space
- Tables of fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2^{nd}-level table
  - Even when exist, 2^{nd}-level tables can reside on disk if not in use

Summary: Two-Level Paging



Summary: Two-Level Paging



IA64: 64bit addresses: Six-level page table?!?

64bit Virtual Address:	7 bits	9 bits	9 bits	9 bits	9 bits	9 bits	12 bits
	Virtual P1 index	Virtual P2 index	Virtual P3 index	Virtual P4 index	Virtual P5 index	Virtual P6 index	Offset

No!

Too slow

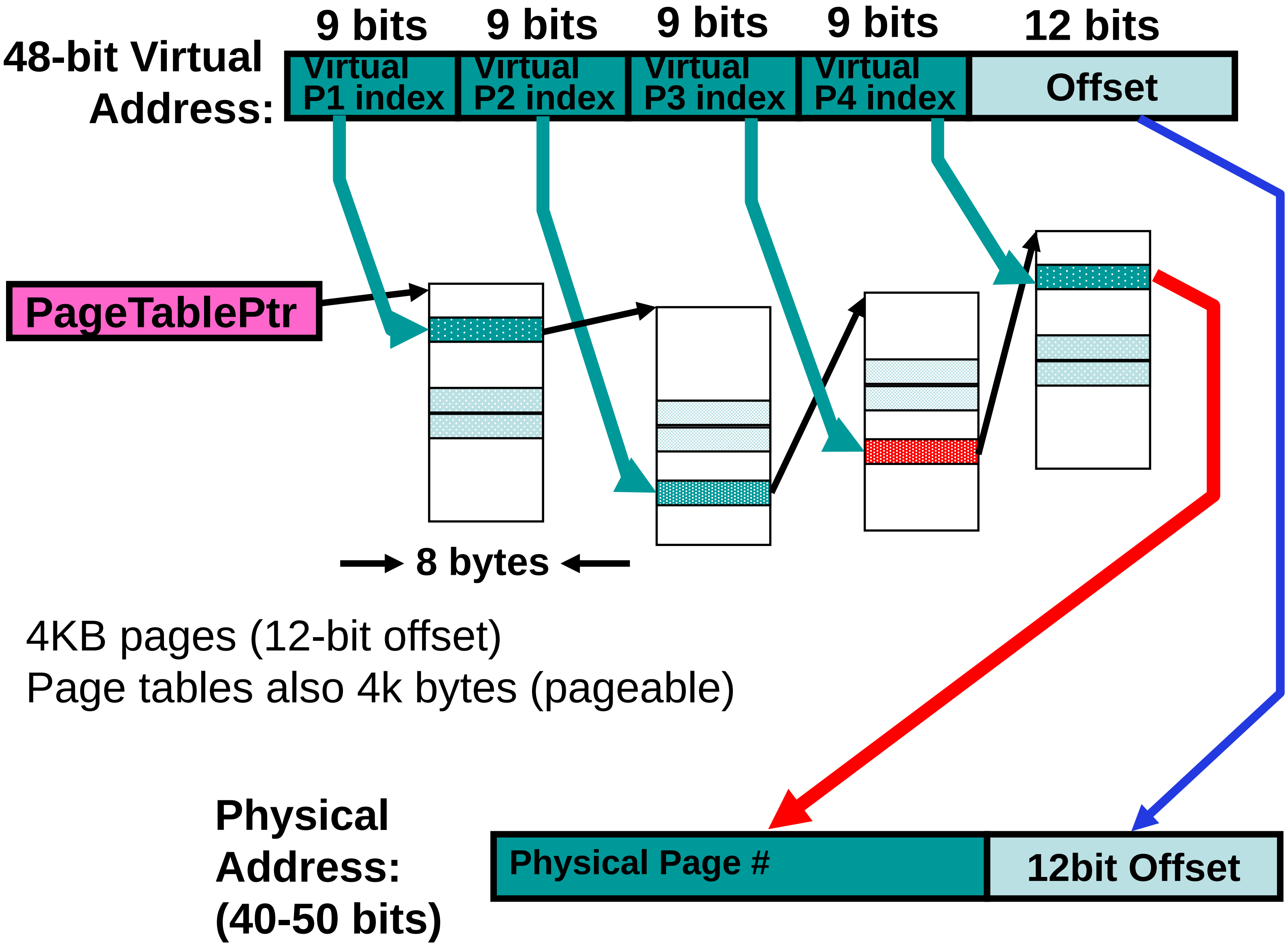
Too many almost-empty tables

Intel/AMD: x86-64

- 64 bits is ginormous (16 exabytes)
- In practice only implement 48-bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy

unused	page map level 4	page directory pointer table	page directory	page table	offset
63 48	47 39	38 30	29 21	20 12	11 0

X86_64: Four-level page table!



4KB pages (12-bit offset)

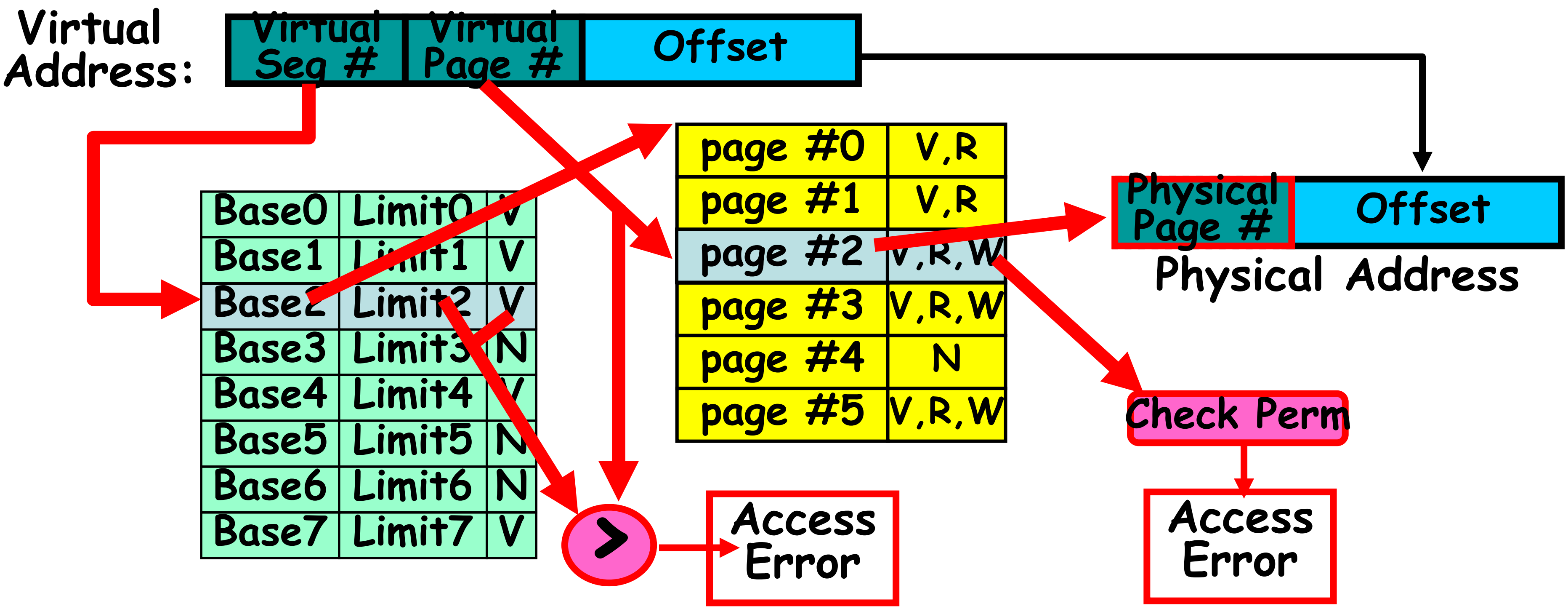
Page tables also 4k bytes (pageable)

Observations on Multi-level Paging

- Two-level paging
  - TLB cache hit → 1 memory access (discussed later)
  - But, TLB cache miss → 3 memory accesses
- Beyond 32-bit addressing, two-level paging is not sufficient
  - E.g., 64-bit addressing need 6-level paging!!
- N-Level paging → N+1 memory accesses on TLB miss!
- So, multi-level paging is very inefficient
- Alternatives are
  - Paged Segmentation
  - Hashed page tables
  - Inverted page tables

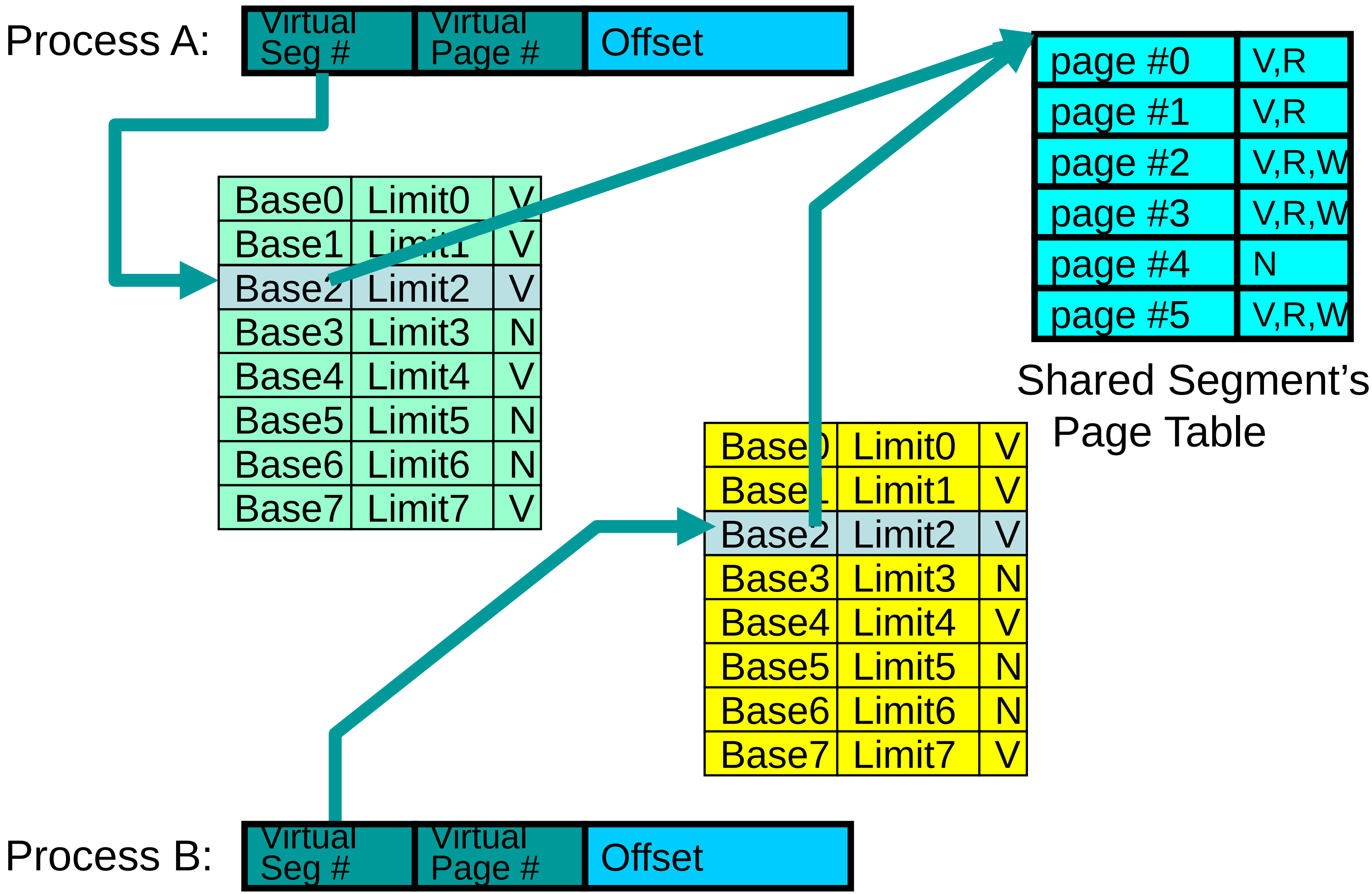
Multi-level Translation: Segments + Pages

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

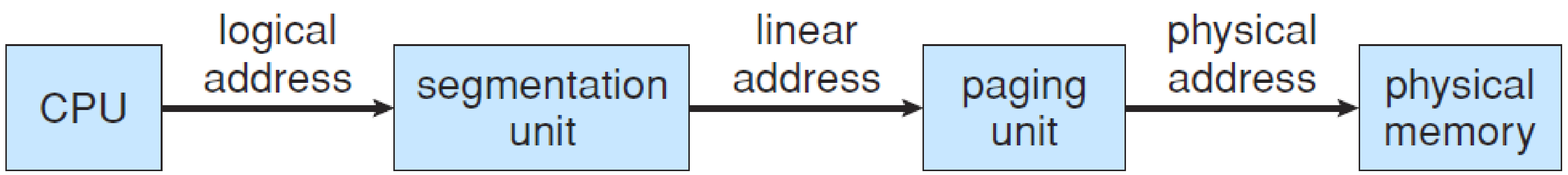


- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (in multi-level paging)

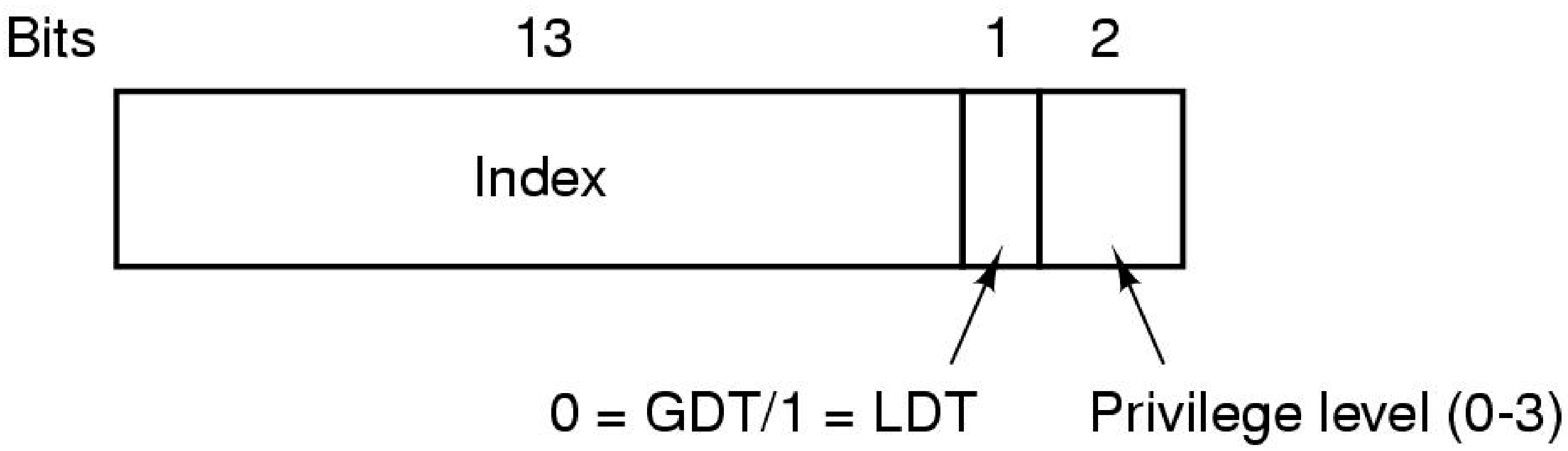
What about Sharing (Complete Segment)?



Segmentation with Paging: x86 32-bit

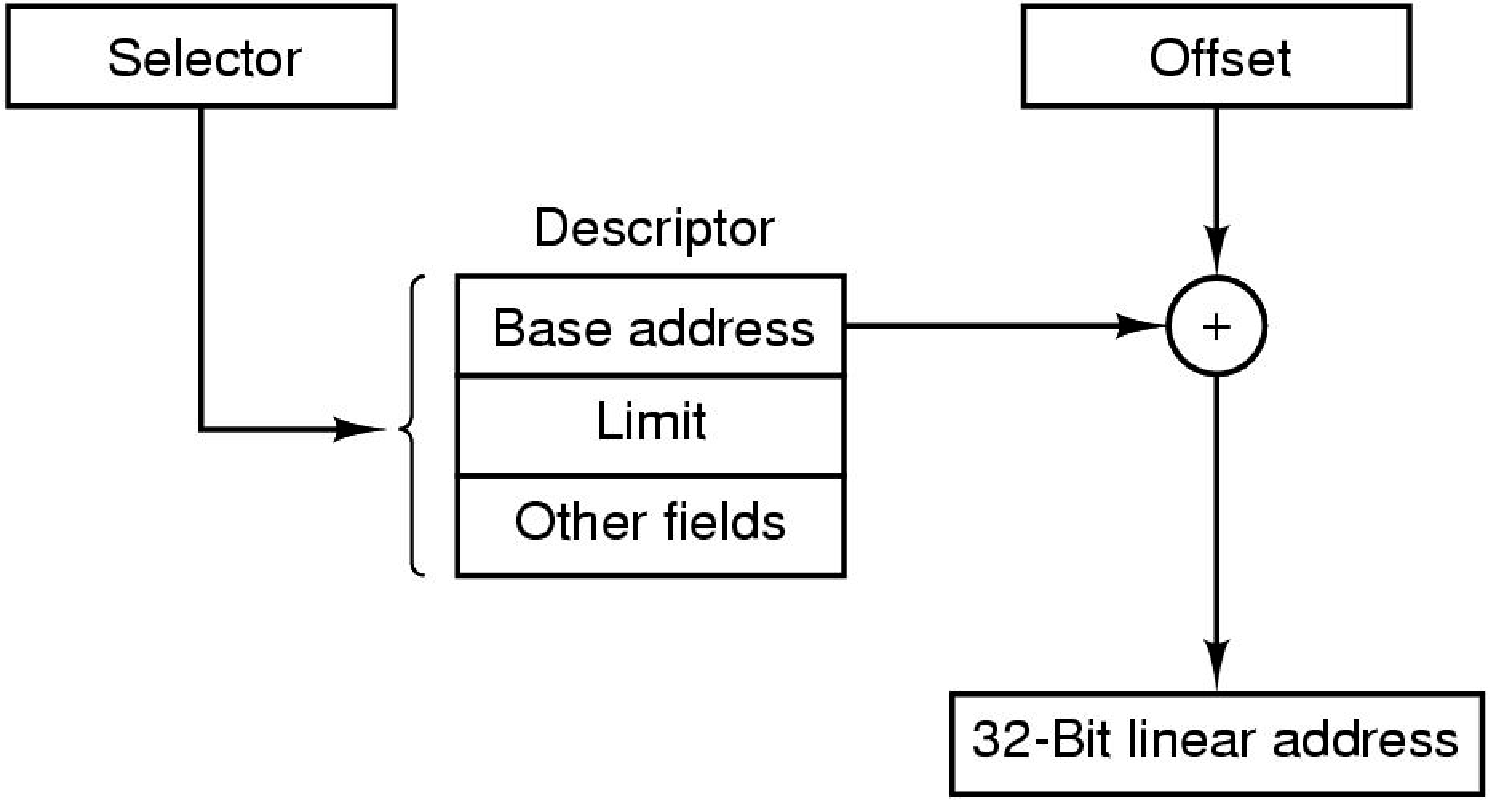


- Segment size: 4GB (32-bit)
- 16 K segments: 8K segments private (13-bit) and rest shared (13-bit)



A Pentium selector

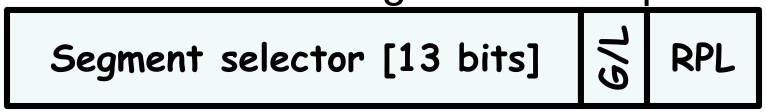
Segmentation with Paging: Pentium



Conversion of a (selector, offset) pair to a linear address

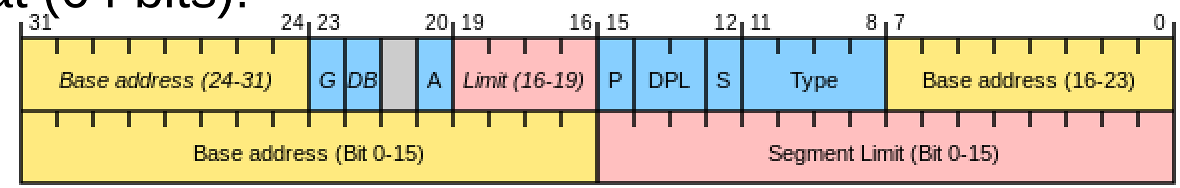
X86 Segment Descriptors (32-bit Protected Mode)

- 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:



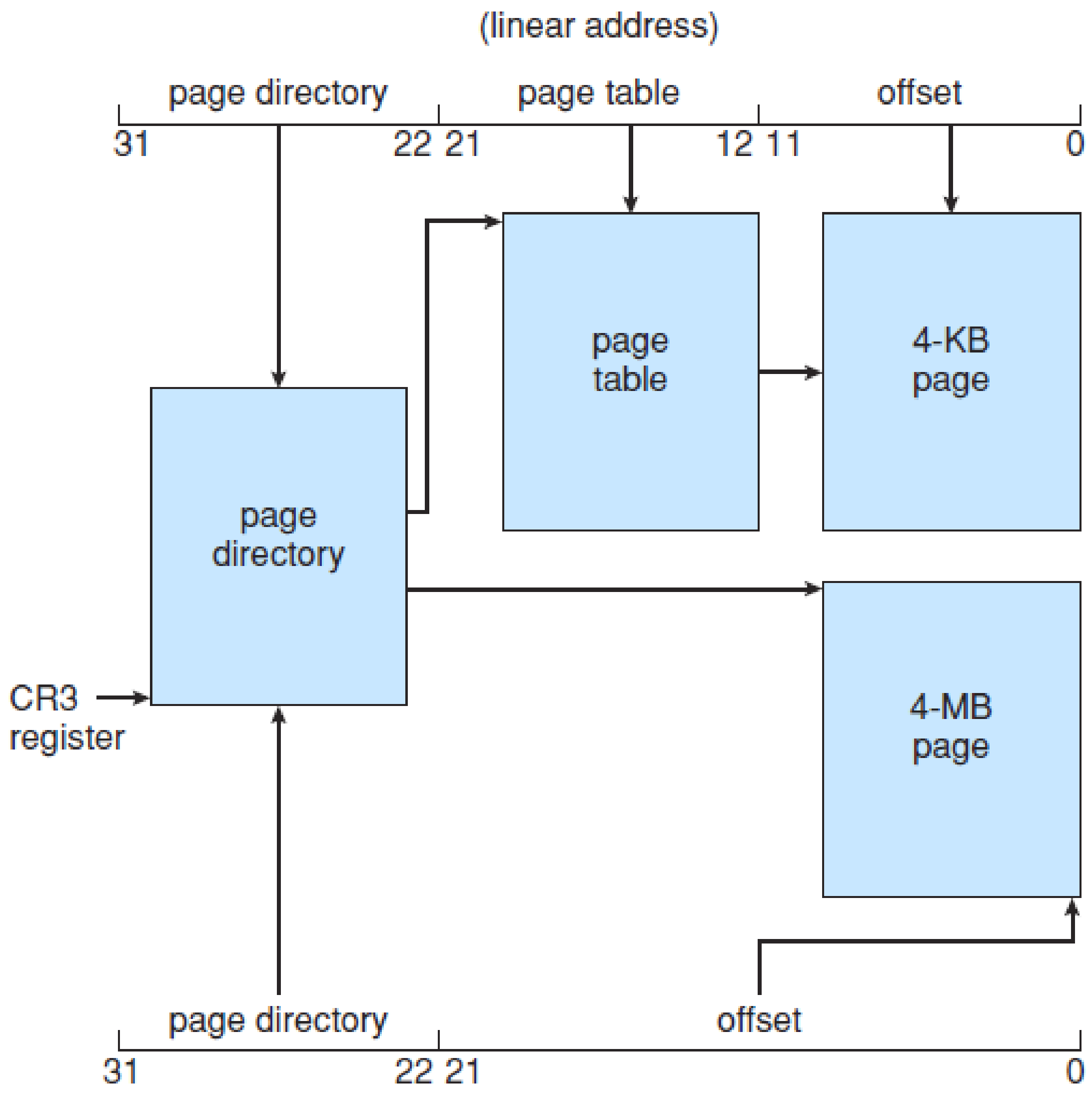
G/L selects between GDT and LDT tables (global vs local descriptor tables)

- 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):

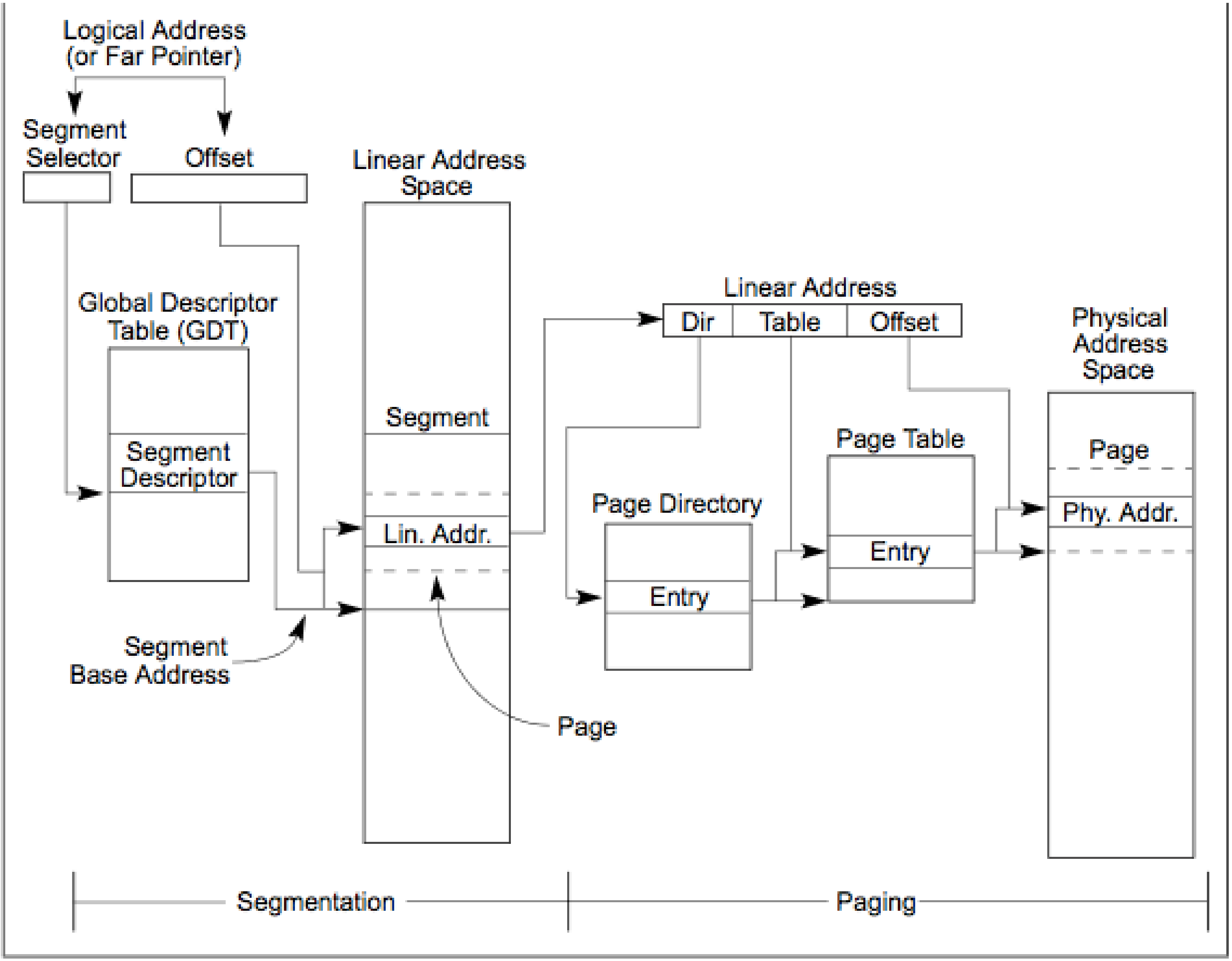


G: Granularity of segment [Limit Size] (0: 16bit, 1: 4KB unit)
DB: Default operand size (0: 16bit, 1: 32bit)
A: Freely available for use by software
P: Segment present
DPL: Descriptor Privilege Level
S: System Segment (0: System, 1: code or data)
Type: Code, Data, Segment

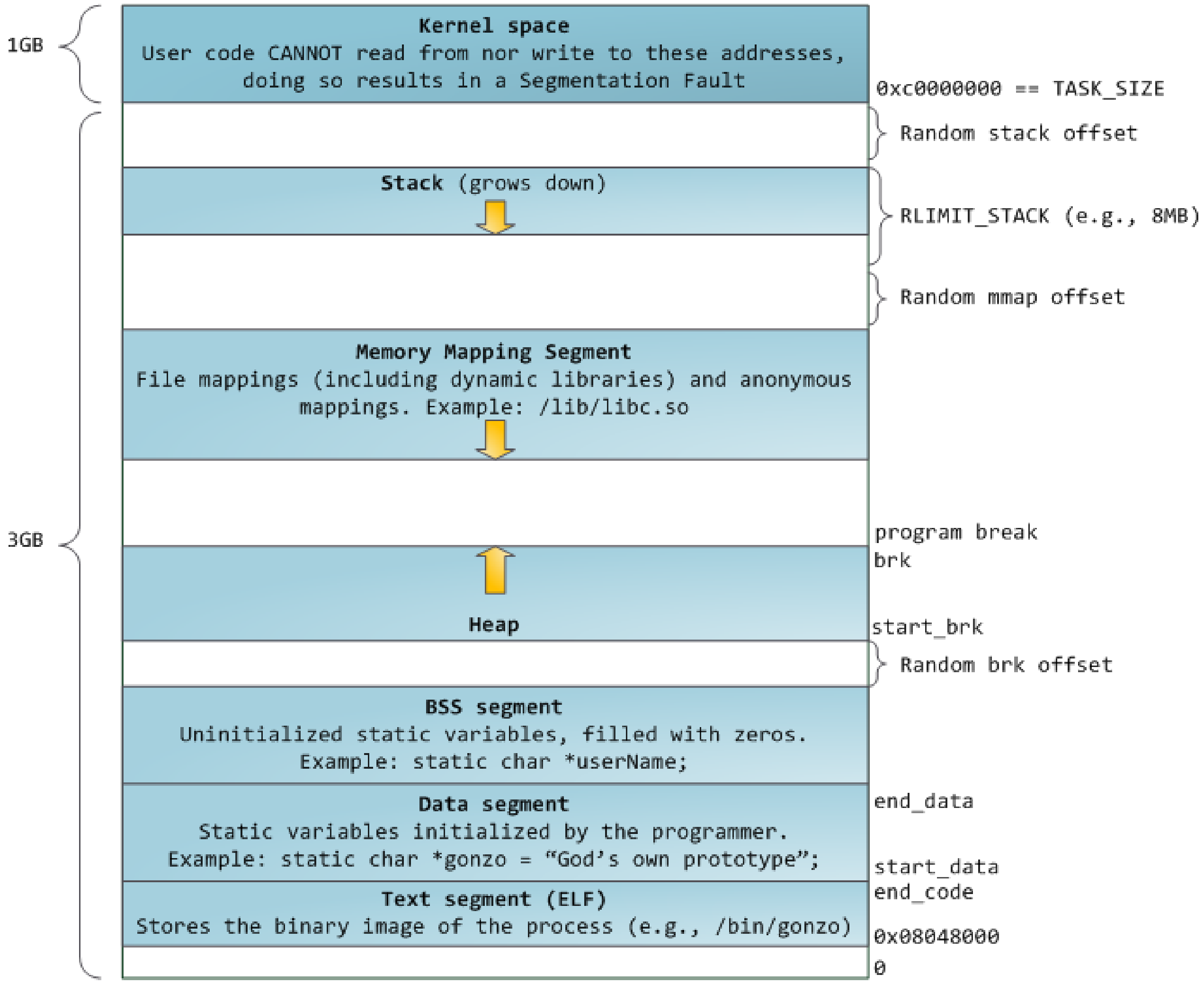
Paging in x86 32-bit Arch



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



Example: Memory Layout for Linux (32-bit)

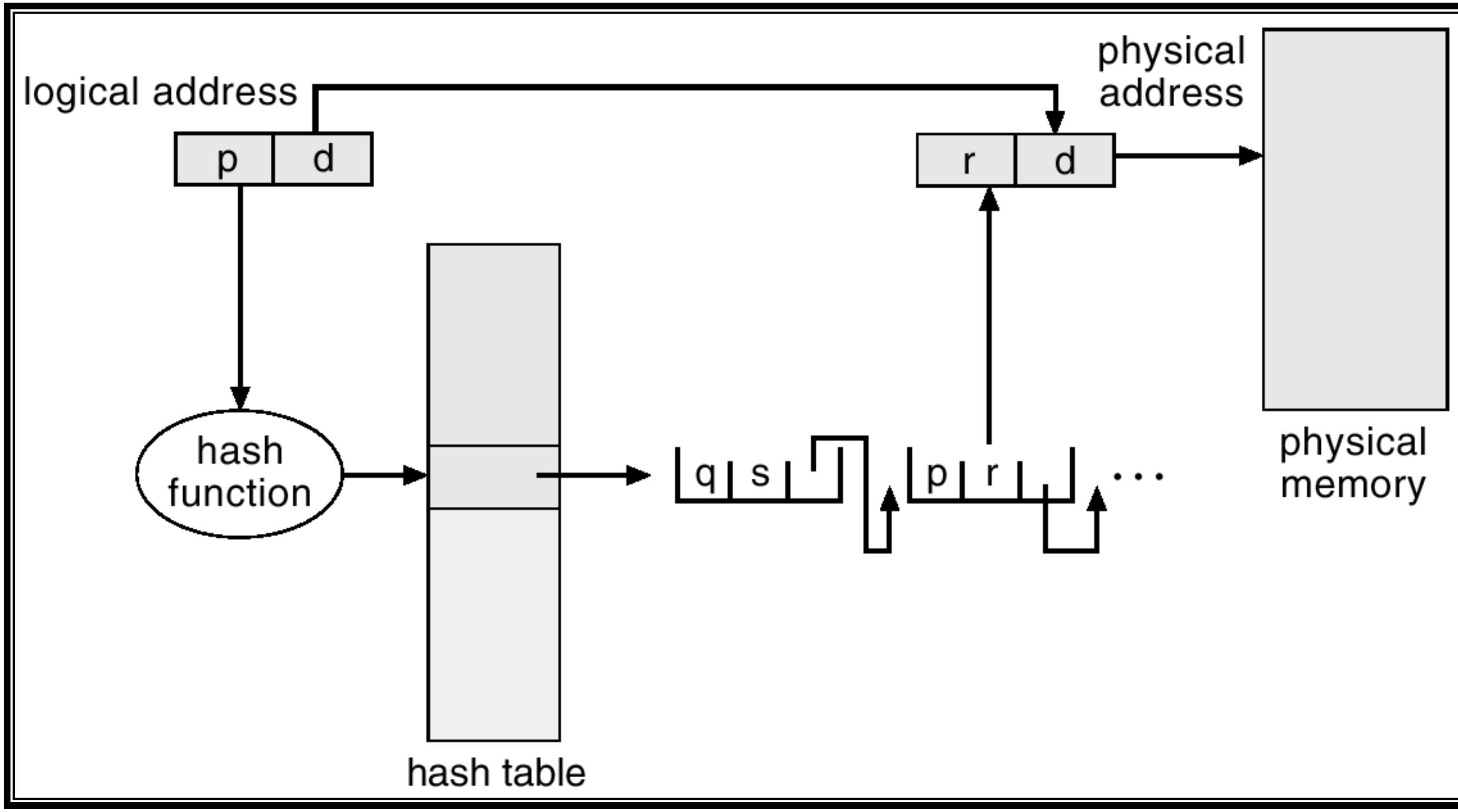


http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Hashed Page Tables

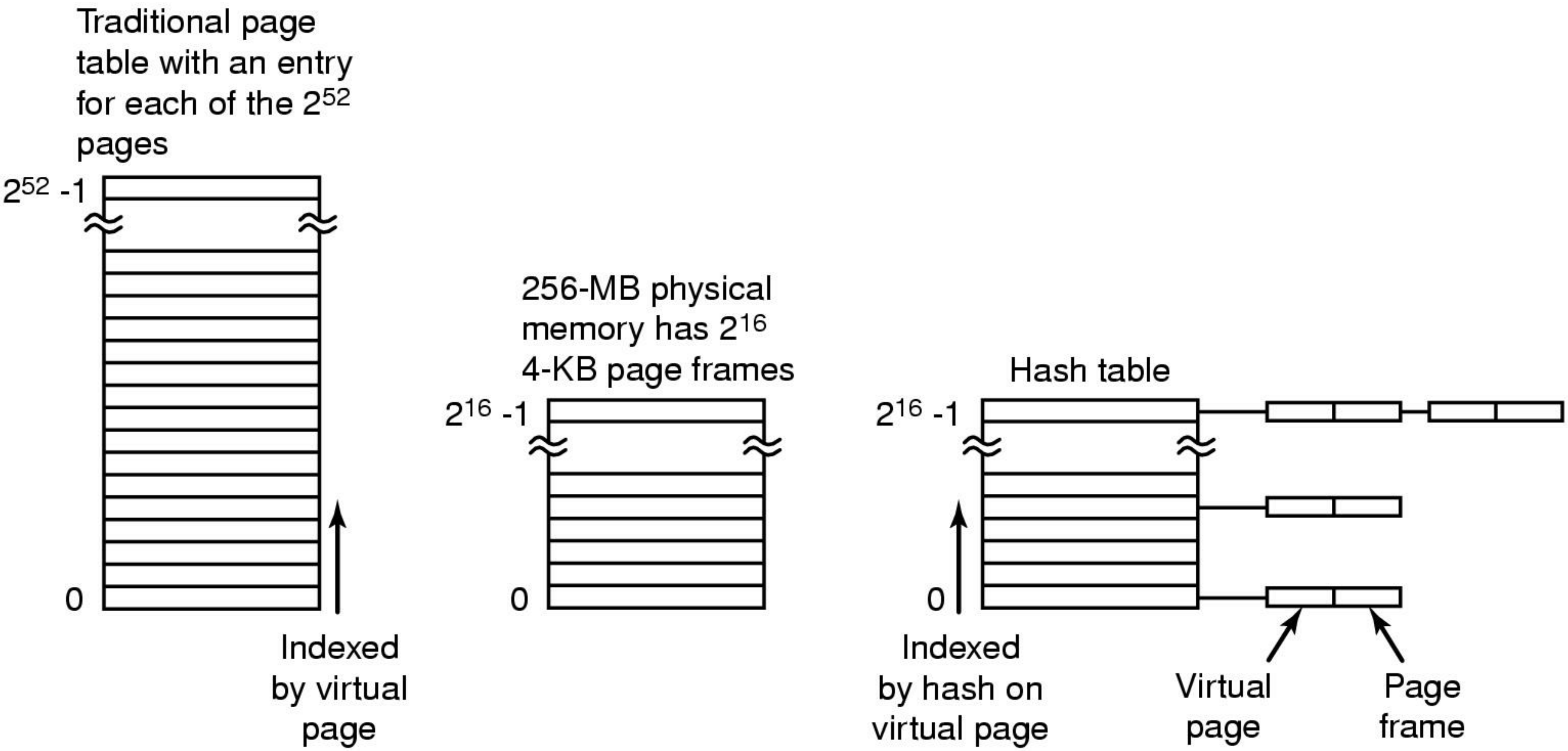
- Common in address spaces > 32 bits
- Efficient for sparse logical address spaces
- The virtual page number is hashed into a page table.
- Page table is a function of physical page frames and at maximum occupies one page frame in the main memory
- This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match.
  - If a match is found, the corresponding page frame is extracted.
- Reduces no. of memory accesses, but increases time needed to search inside the table
  - TLB and sparse address spaces help

Hashed Page Table



How many memory access?

Hash Tables

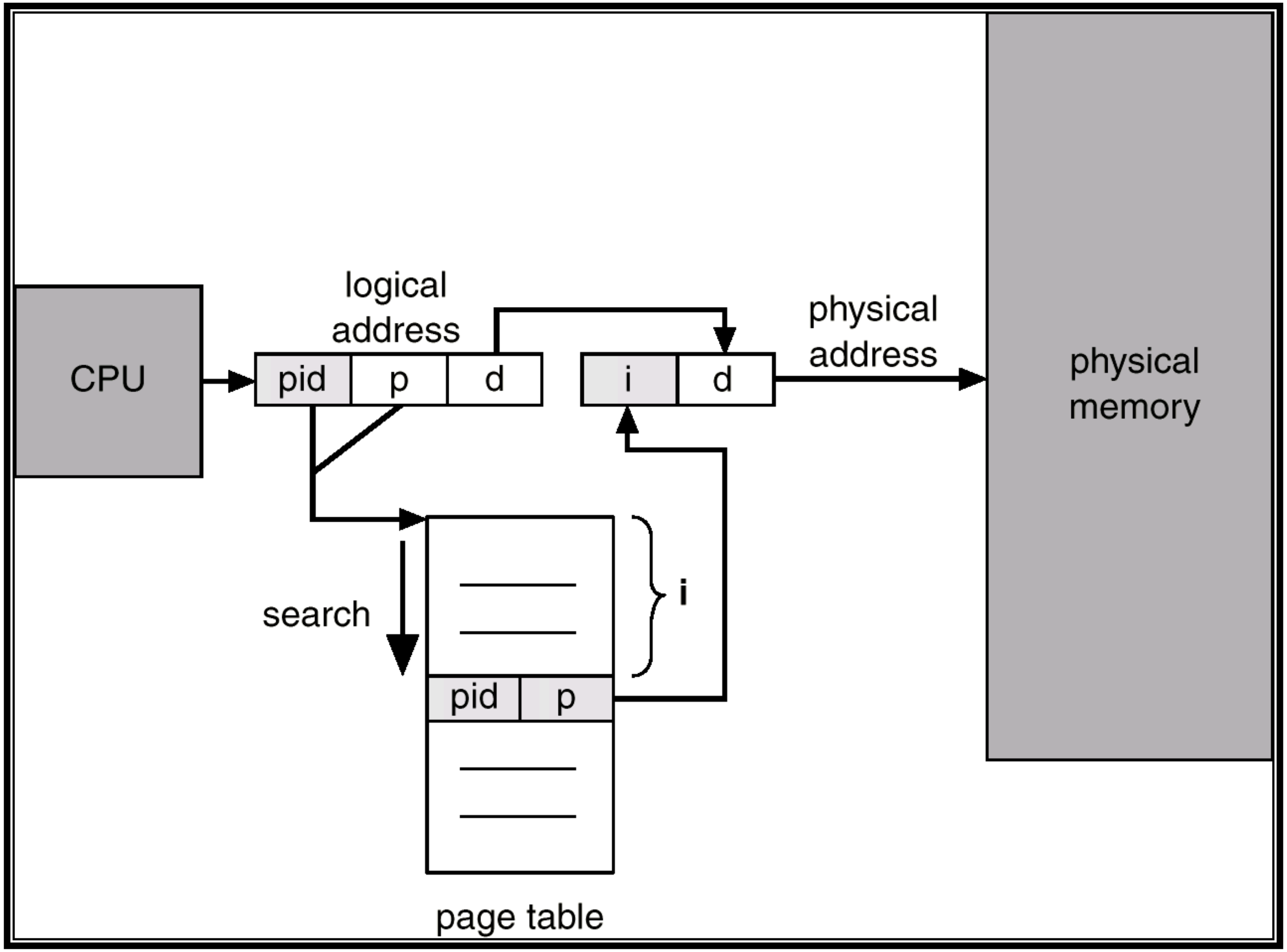


Comparison of a traditional page table with Hash table

Inverted Page Table

- One entry for each real page (frame) of main memory.
- Page table entry (PTE) consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to keep page table per process: now one inverted page table for whole system
- But increases time needed to search the table when a page reference occurs → no more indexing possible like in Page tables, Segment tables
  - Use hash table to limit the search to one — or at most a few — page table entries.

Inverted Page Table Architecture



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 to manage
  - Easy memory allocation using bitmap
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables (or segment 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!

Comparison of Memory Mgmt Schemes

Diff schemes: contiguous allocation, paging, segmentation and multi-level translation

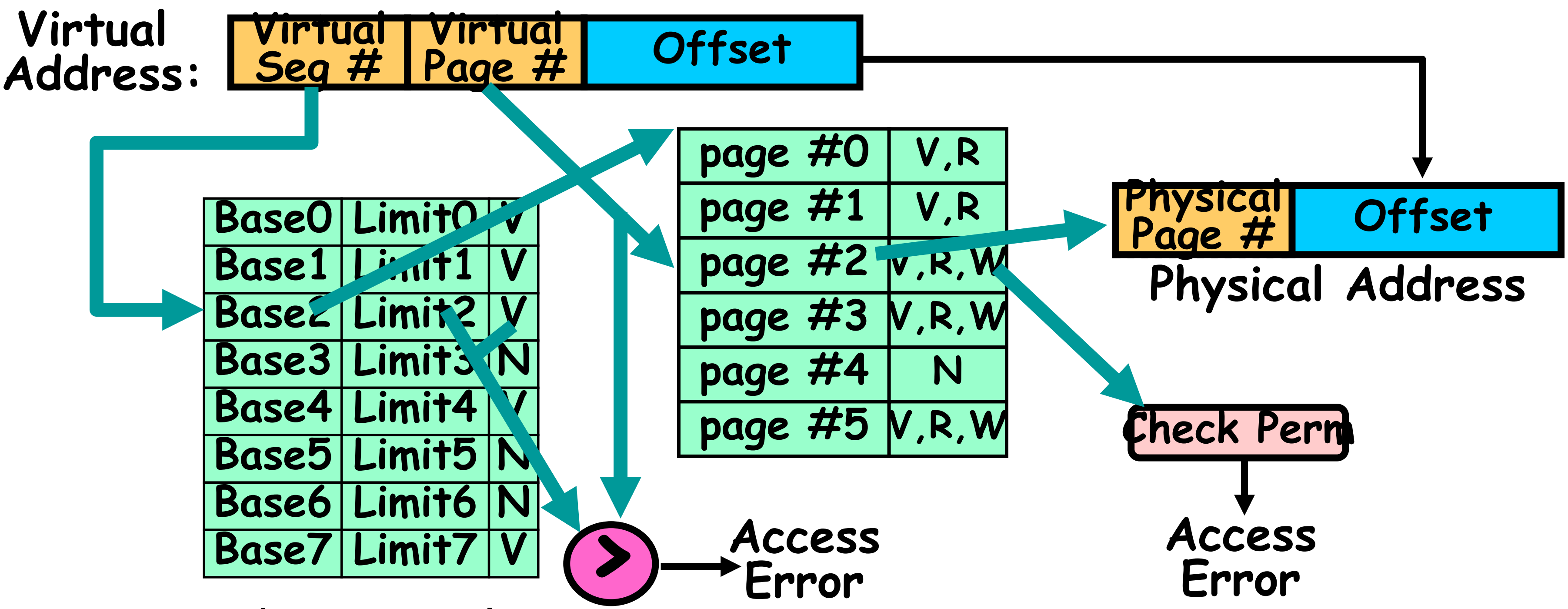
- Hardware support: Registers, MMU, tables
- Performance: Mapping delay can be reduced by TLB
- Fragmentation: Avoid external fragmentation to improve degree of multiprogramming
- Relocation: To avoid external fragmentation, use relocatable (virtual) addresses
- Swapping: To let more programs to run on limited RAM, but not on Flash-based systems like tablets/Smart phones
- Sharing: To reduce memory footprint of processes
- Protection: Helps sharing and avoids accidental errors by keeping protection bits in Page/Segment tables

Address Translation Comparison

	Advantages	**Disadvantages**
Simple Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation	Table size ~ # of pages in virtual memory, fast easy allocation	Multiple memory references per page access
Two-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

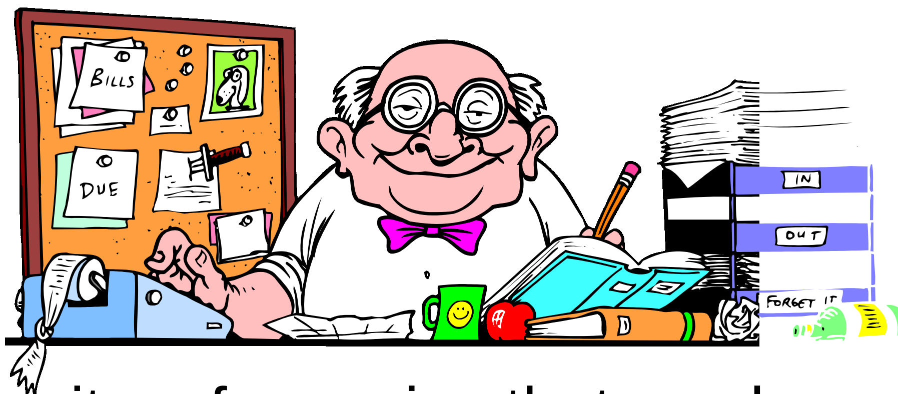
Major Reason to Deal with Caching

- Page table is kept in main memory. *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.



- Too expensive to translate on every access
  - At least two DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

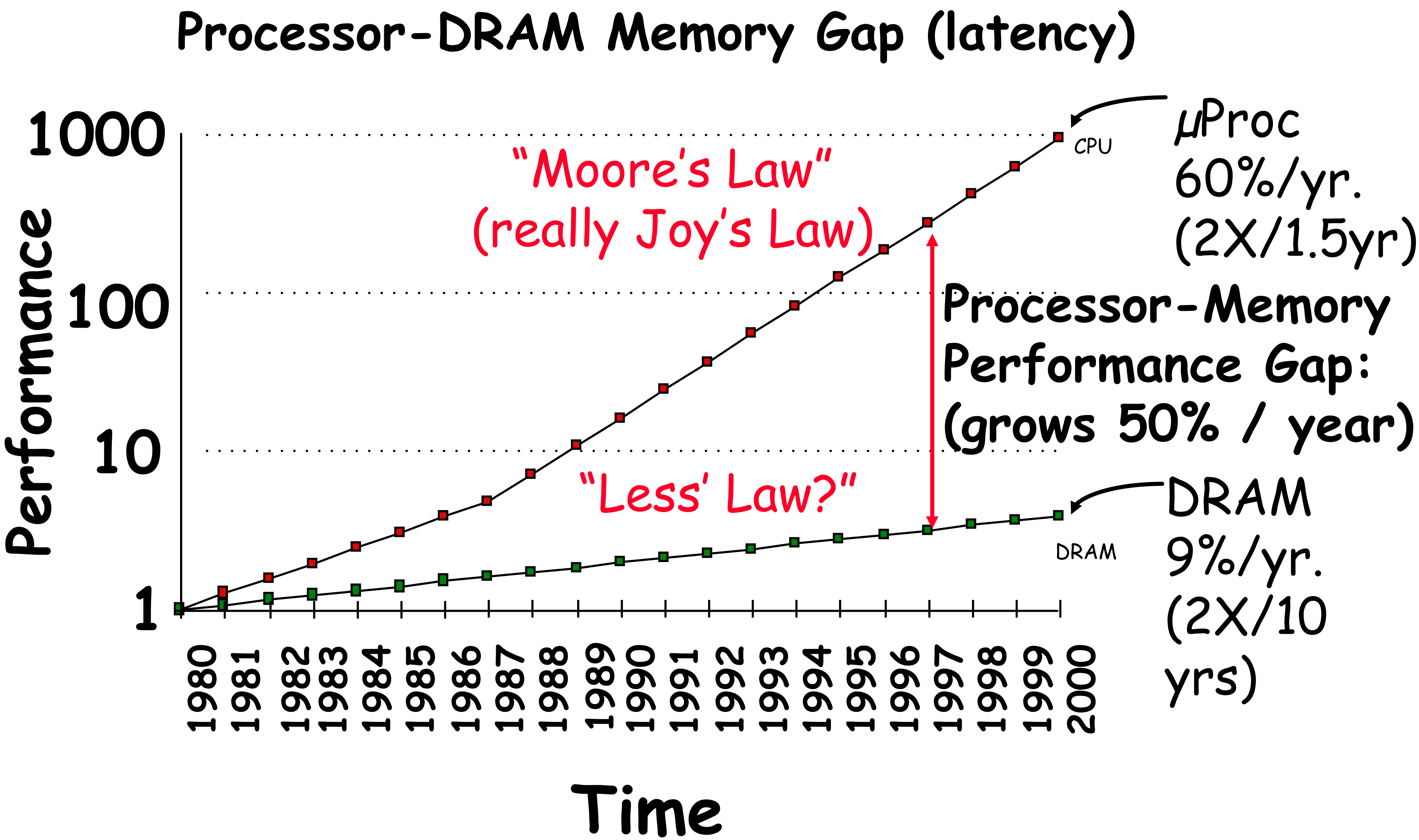
Caching Concept



- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many of the techniques that are used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time =

  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

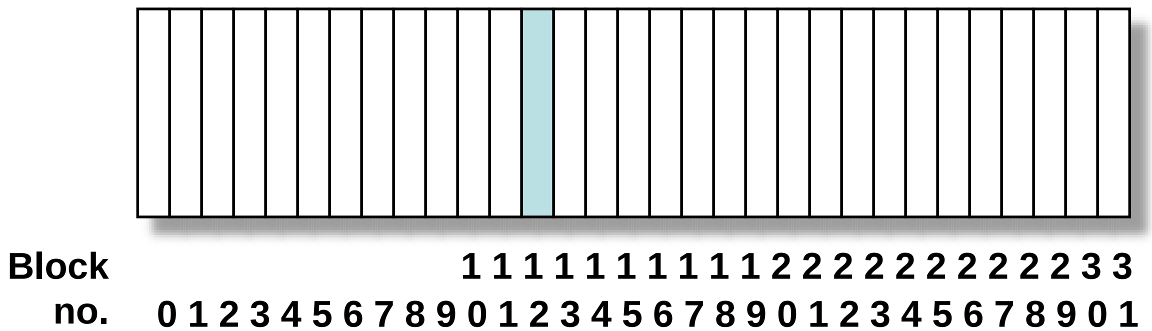
Why Bother with Caching?



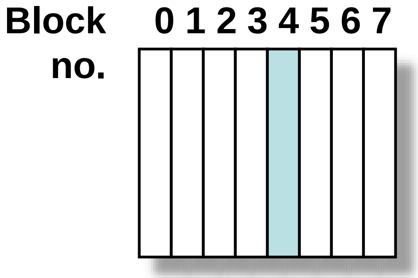
Review: Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

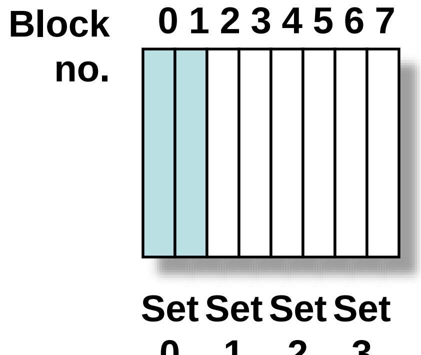
32-Block Address Space:



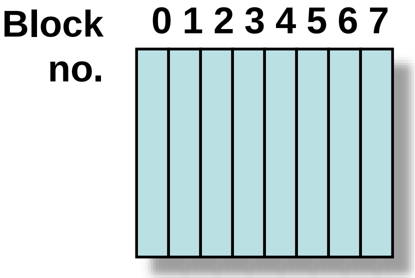
Direct mapped:
block 12 can go only into block 4 (12 mod 8)



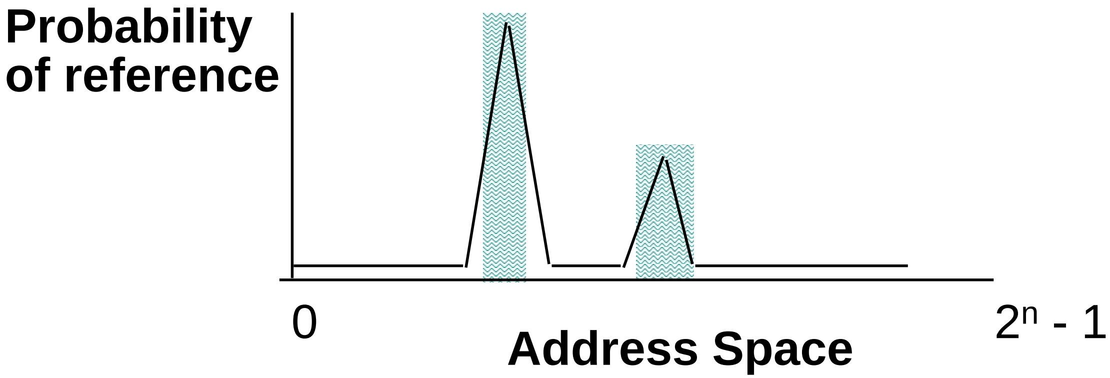
Set associative:
block 12 can go anywhere in set 0 (12 mod 4)



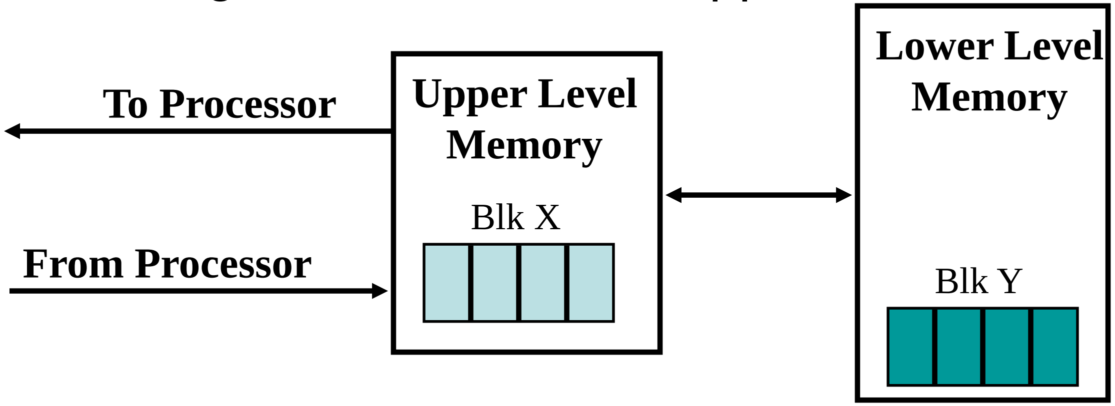
Fully associative:
block 12 can go anywhere



Why Does Caching Help? Locality!

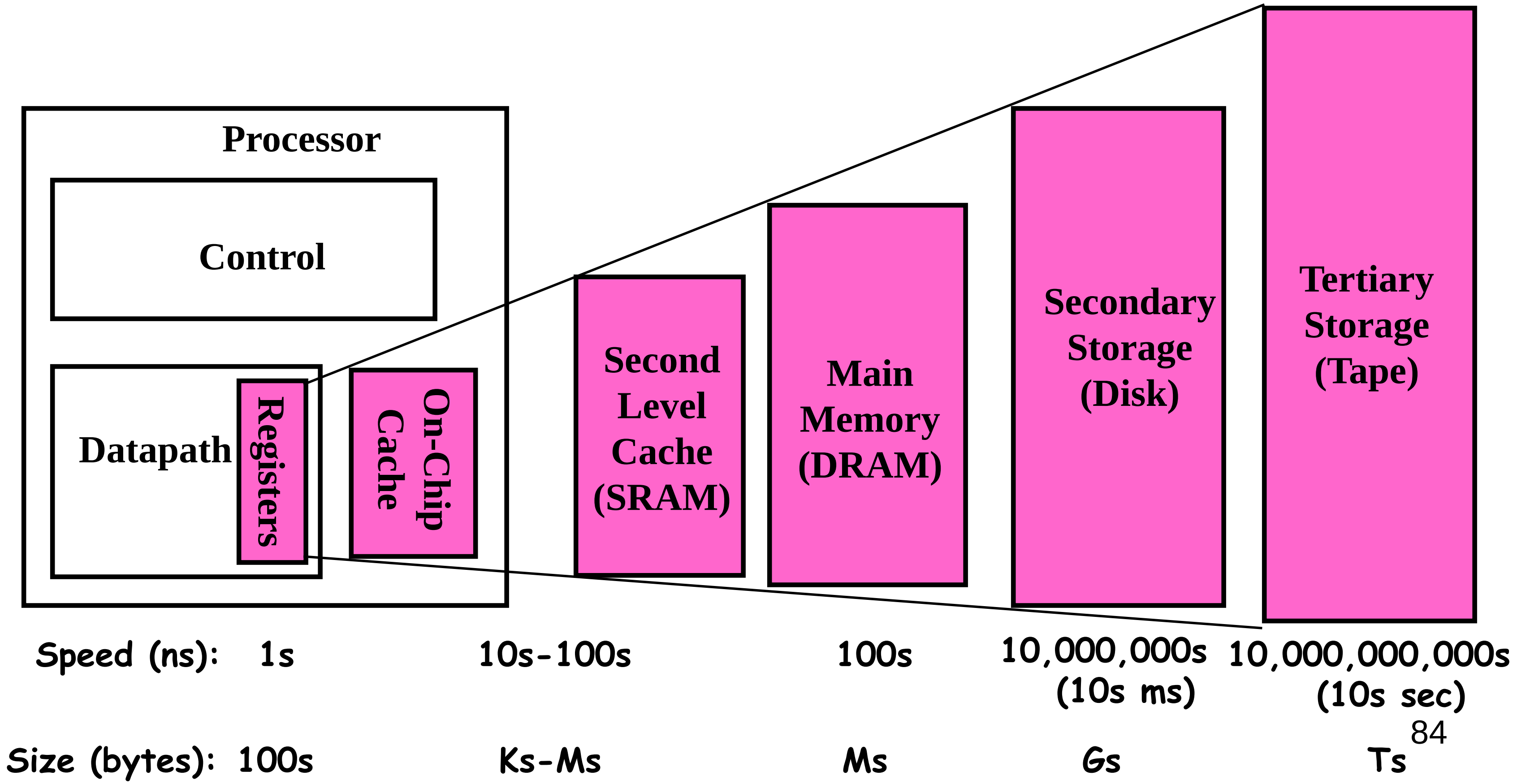


- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



Review: Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



A Summary on Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

Other Caching Questions

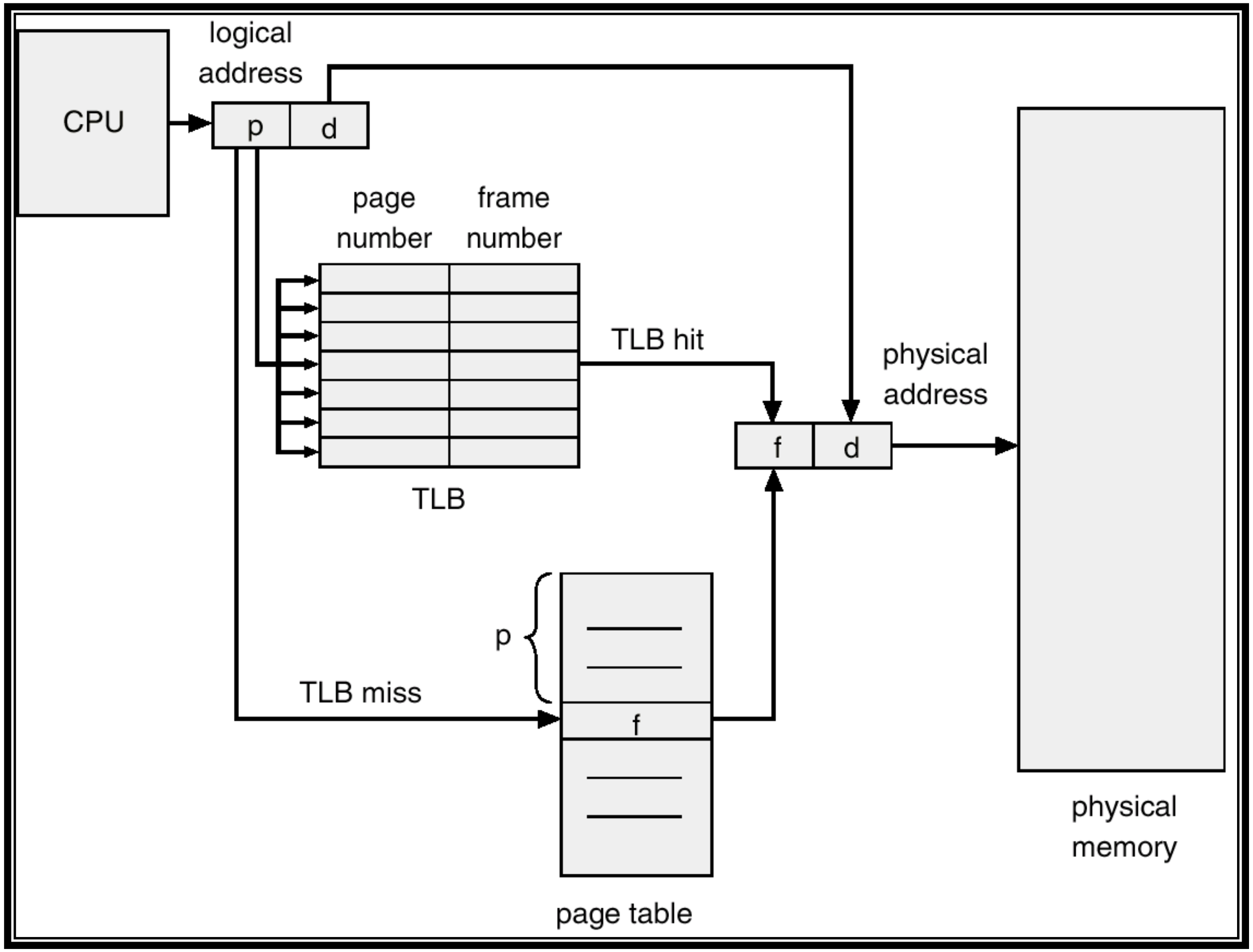
- What entry gets replaced on cache miss?
  - Easy for Direct Mapped: Only one possibility
  - Set Associative or Fully Associative:
    - Random
    - LRU (Least Recently Used)
- What happens on a write?
  - Write through: The information is written to both the cache and to the block in the lower-level memory
  - Write back: The information is written only to the block in the cache
    - Modified cache block is written to main memory only when it is replaced
    - Question is block clean or dirty?

TLB: Associative Memory

- Relatively small number of entries (< 512)
- Associative memory – parallel search (since misses are expensive)
- TLB entries contain virtual page ID, PTE and optional process ID
- TLB is logically in front of cache
  - needs to be overlapped with cache access to be really fast

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Paging Hardware With TLB



Cache, to be precise, multiple levels of caches are not shown in above diagram!

Intel i7 has L1 Data TLB, L1 Text (Code) TLB and L2 TLB!

What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - If PTE valid, hardware fills TLB and processor never knows
    - If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (like MIPS)
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - If PTE valid, fills TLB and returns from fault
    - If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - shared segments
    - user-level portions of an operating system

Effective Access Time

- TLB Lookup = 20 ms
- Assume memory access time is 100 ms
- Hit ratio – percentage of times that a page number is found in the associative registers IN TLB; ration related to number of associative registers.
- Hit ratio = α
- Effective Access Time (EAT)

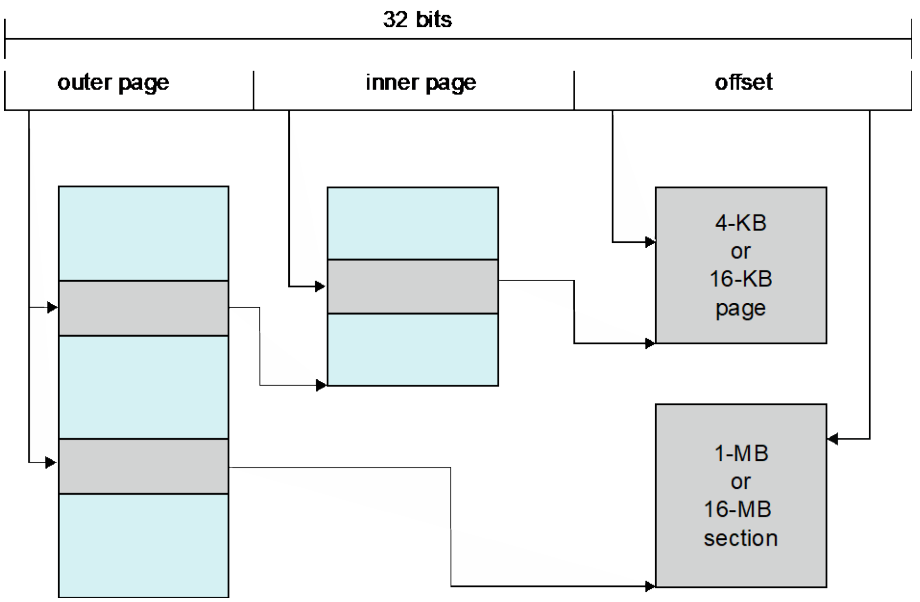
$$\text{EAT} = (100 + 20)\,\alpha + (200 + 20)(1 - \alpha)$$

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include unique process ID in TLB (ASID field in virtual address)
    - » This is an architectural solution: needs hardware
- What if translation (page) tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!

ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First micro TLB is checked, on miss inner, main TLB is checked, and on miss page table walk performed by CPU



Summary (1/2)

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

Summary (2/2)

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