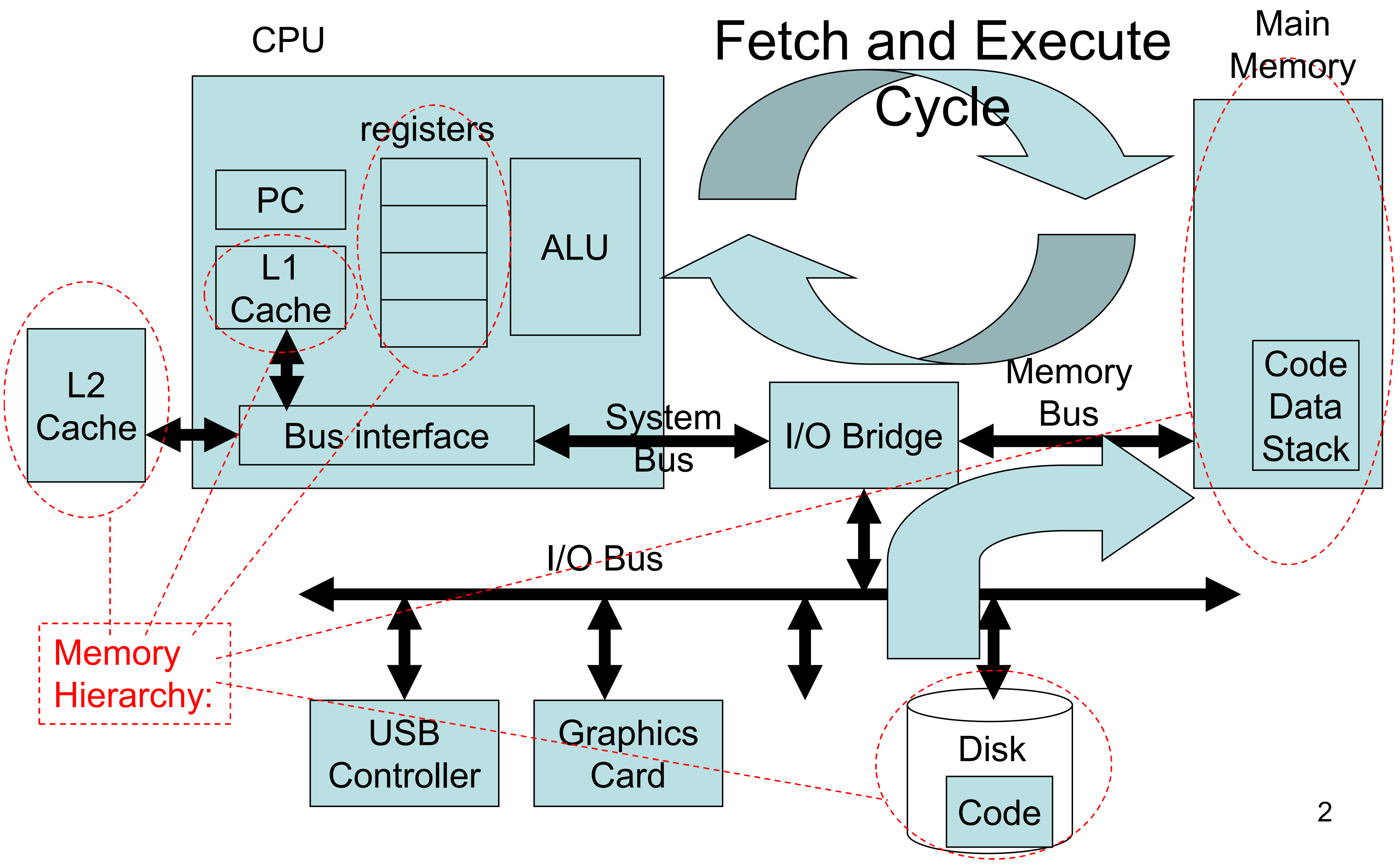
Operating System

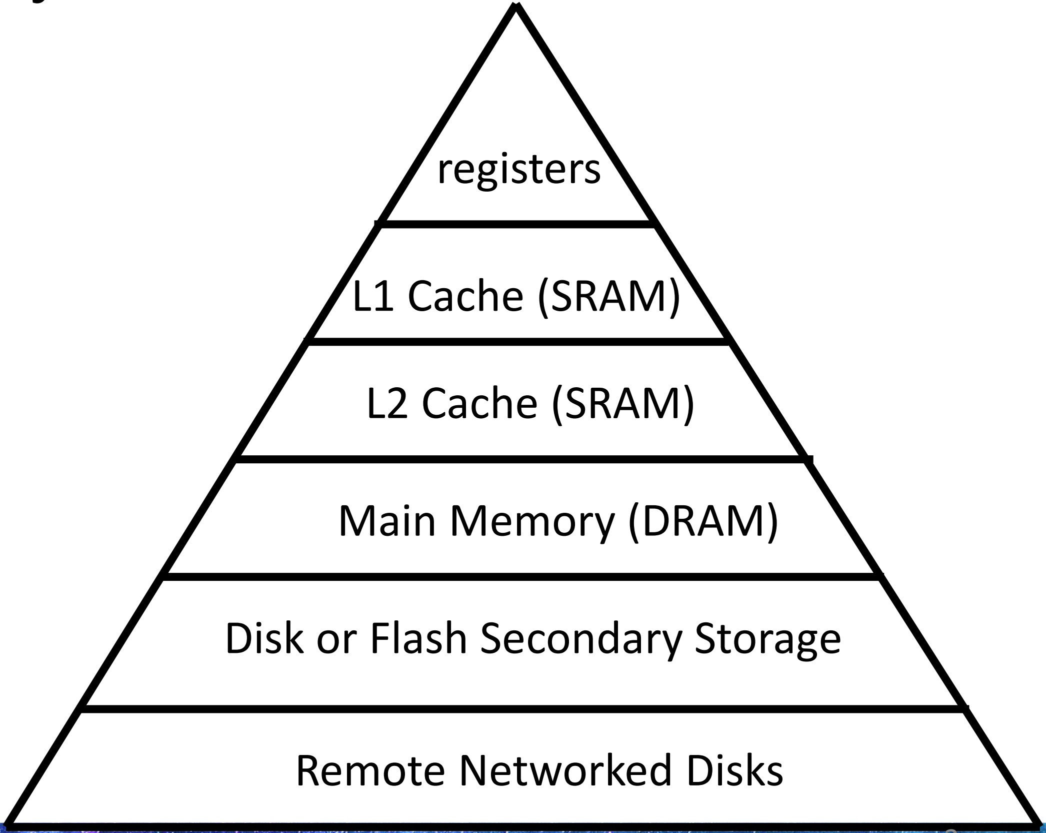
Main Memory Management

Memory Management

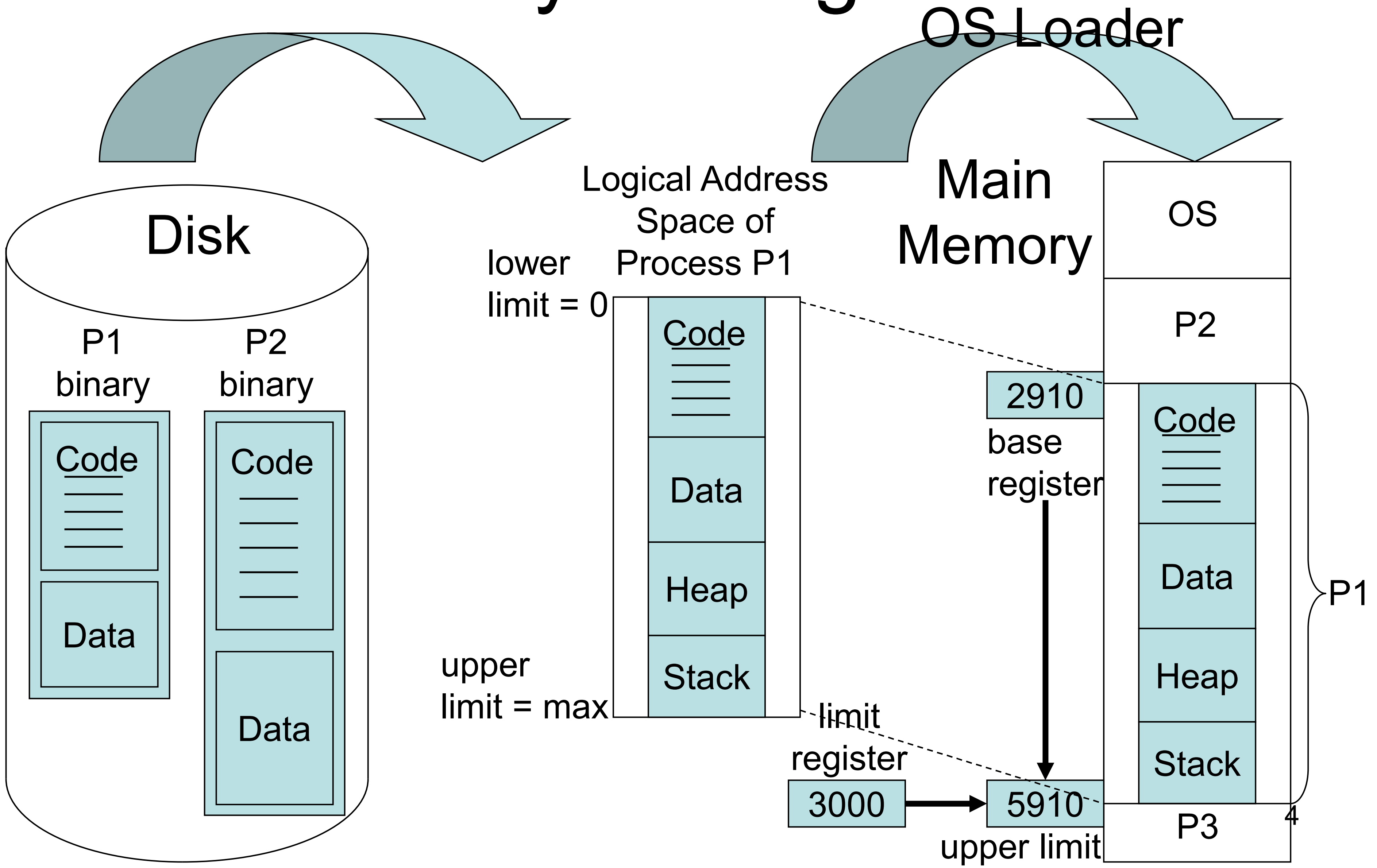


Memory Management

- Memory Hierarchy
  - cache frequently accessed instructions and/or data in local memory that is faster but also more expensive



Memory Management



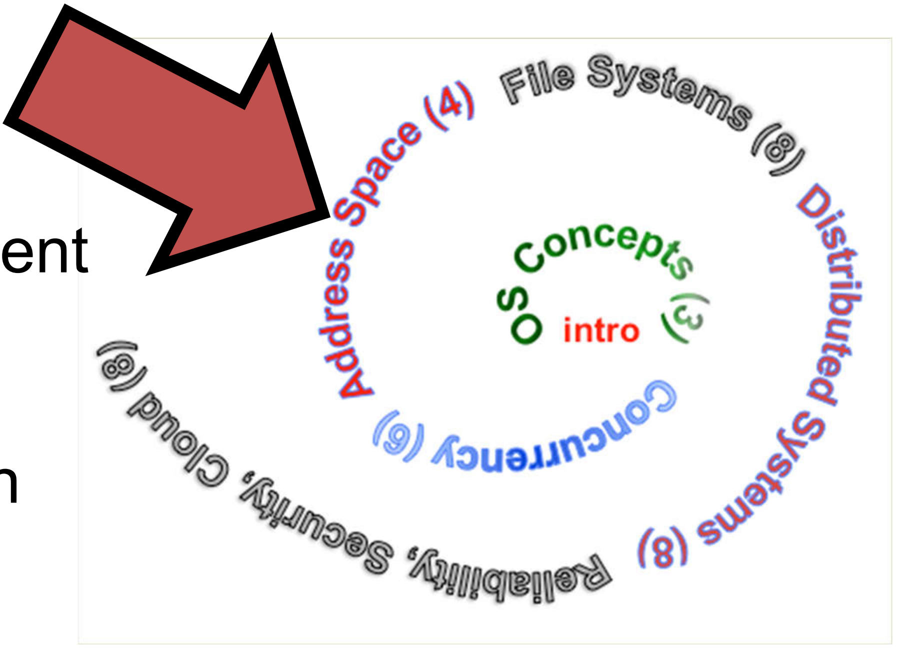
Why memory management?

- CPU utilization and increase of performance
- → need of scheduler to schedule multiple processes
- → they should be resident in RAM (physical memory)

- Why need of memory management?
  - Having multiple processes
  - Protection (processes from one another, processes from OS)
- Protection should be done by Hardware than OS (why?)
  - Performance penalty and latency

Next Objective

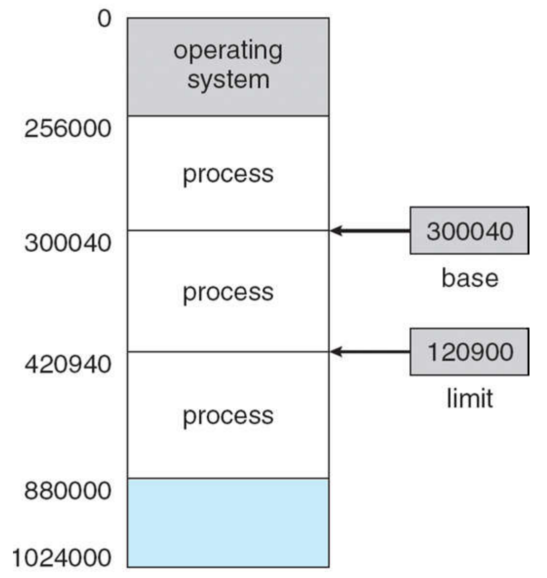
- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
 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



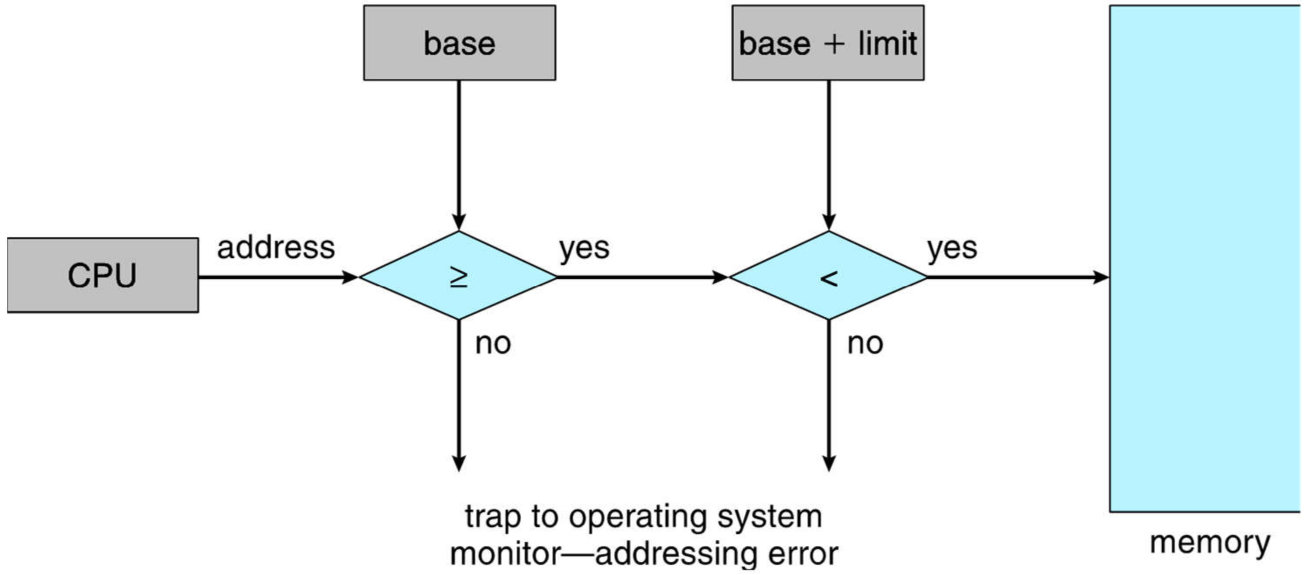
How to protect process memory space?

- Each process has a separate memory space
- To protect processes' spaces
  - Determining legal address
    - Base register: smallest legal physical memory address
    - Limit register: size of the range
  - Example:
    - Base register = 300040
    - Limit register = 120900
    - Legal address space:

      Base <= (any address) < Base+Limit
    - Can easily be checked by hardware



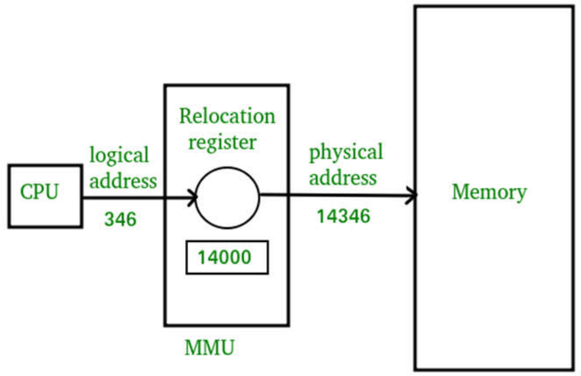
HW address protection (base & limit regs.)



Any illegal address generates a trap exception which is known as fatal error

Who loads Base and Limit registers?

Address binding

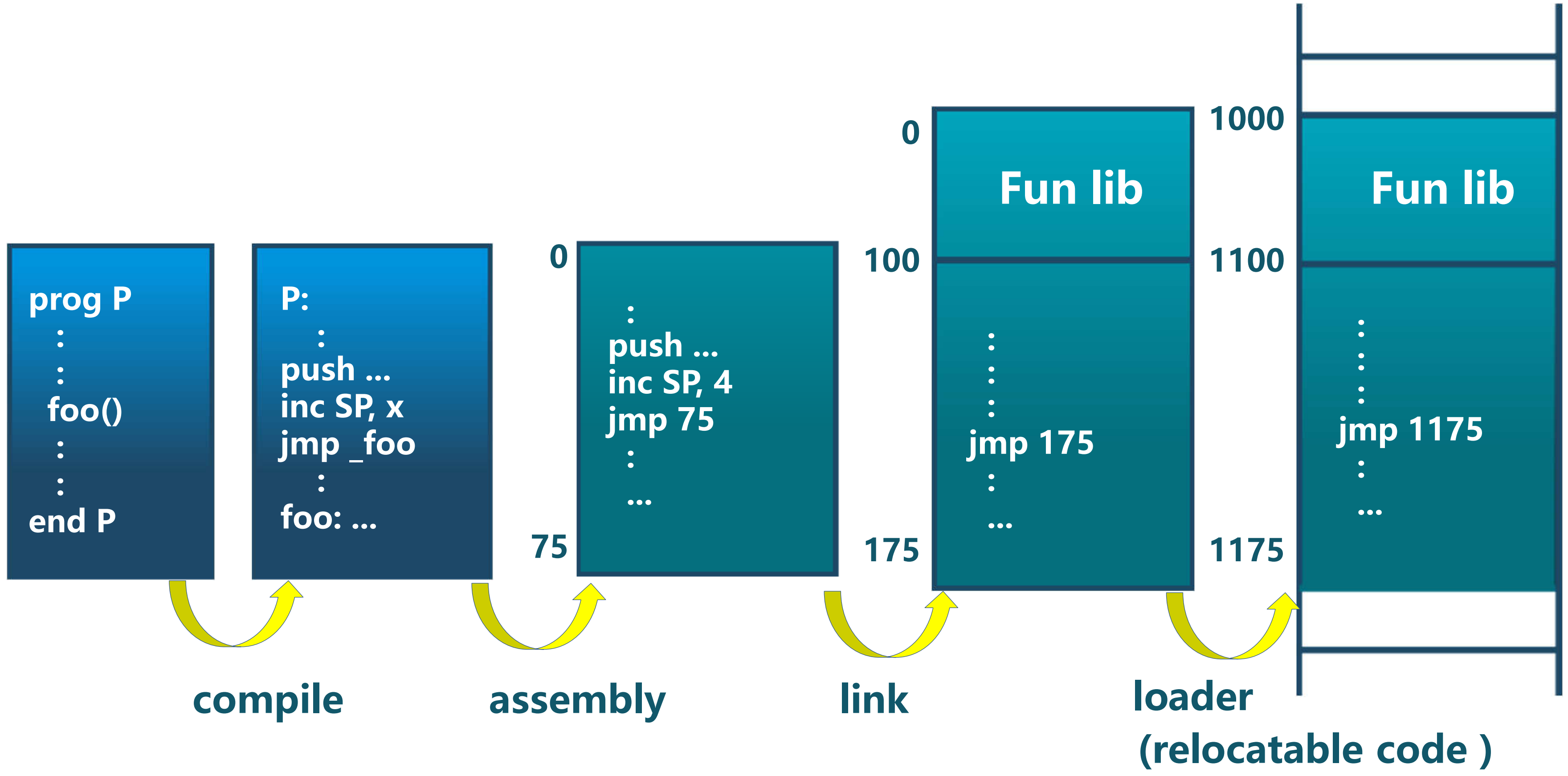


Address binding

- Input Queue
  - Processes on disk that are waiting to be brought into memory during execution (Part of ready queue which is on disk)

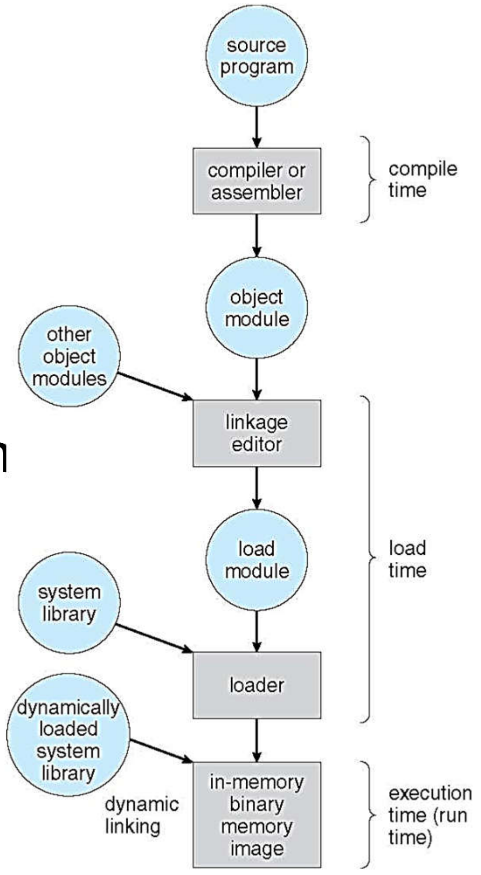
- How to put a process in a physical address?
  - Addresses in source program are symbolic
    - Example: *count* variable
  - A compiler binds them to relocatable addresses
    - Example: 14 bytes from beginning of this module
  - Linker and loader bind them next to absolute addresses
    - Example: 74014

Generate logic address



How to bind inst./data to mem. address?

- Compile time
  - If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
    - Example: COM files in MS-DOS
- Load time
  - Must generate **relocatable code** if memory location is not known at compile time
- 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 maps
      - Example: Base-Limit registers



Compile Time	Load Time	Execution Time
Code is translated into machine readable format	Program code is loaded into memory	Program code is executed
Completed with a compiler	Completed with a loader	Completed with run-time libraries

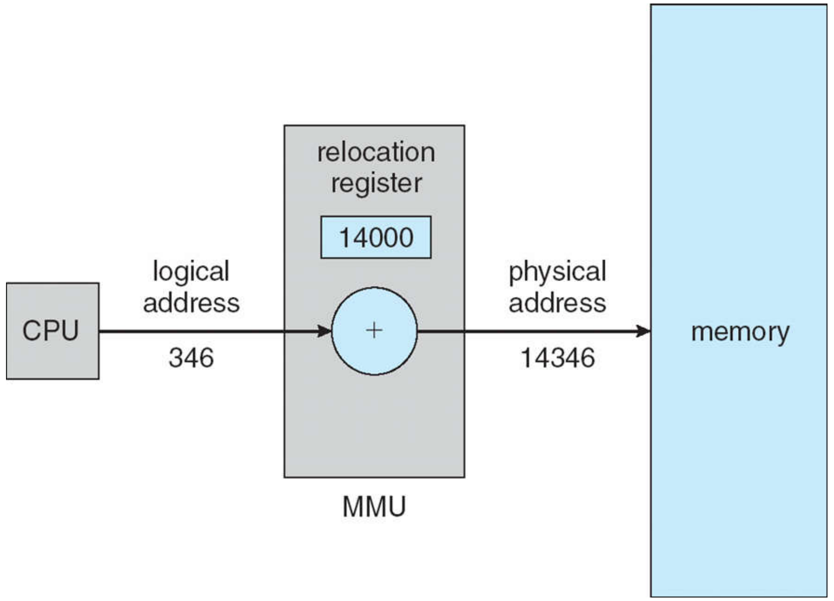
Logical vs. physical address space

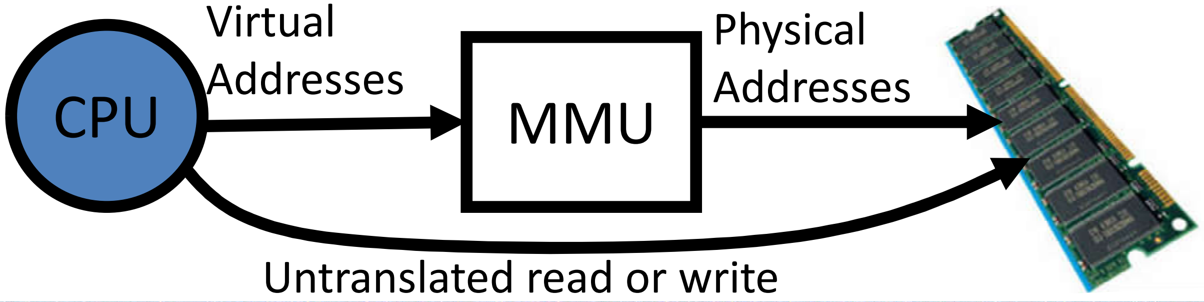
- Logical address (CPU address)
  - CPU logically sees addresses
  - Logical address space: set of all logical addresses generated by a program

- Physical address (Memory address)
  - Address of memory line
  - Physical address space: a physical address (also real address, or binary address)
  - The set of all physical addresses corresponding to these logical addresses is a physical address space.
- May be equal or not
  - Equal: compile-time and load-time address bindings
  - Not equal: execution-time address binding
    - In this case, logical address is said virtual address

What is MMU (Memory Management Unit)?

- MMU is responsible of mapping virtual (logical) address to physical address
- Simple version: Base-Limit registers
- Here Base register is called relocation register





Dynamic linking and loading

Dynamic loading

- Dynamic loading
  - A routine is not loaded until it is called
  - All routines except *main()* are kept on disk
  - In the case of call, if it is not loaded, linking loader first loads it to the memory and update program's address table to reflect this change

Pros:

- Better space utilization
- Some routines are infrequently needed: *error functions*

Dynamic linking

- Dynamic linking (Dynamically linked libraries)
  - System libraries that are linked to user programs when they are run
  - No waste of memory and disk space
  - Example: Language subroutines
  - Stub: small piece of code used to locate dynamic linked libraries
    - Replace itself with the address of the routine & executes the routine
  - Also known as Shared Libraries
- Static linking
  - System libraries are linked to user programs during compile time

Swapping

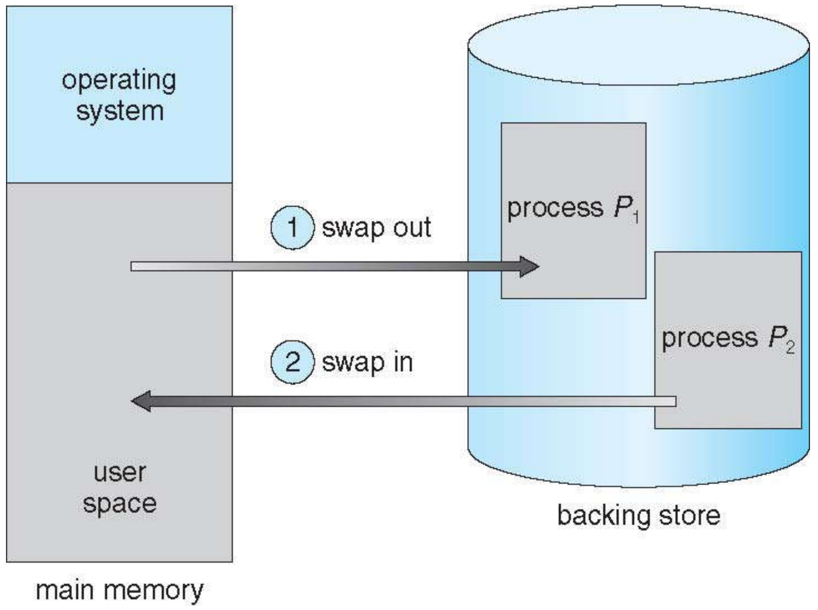
Swapping

- Swapping is performed temporarily between memory and backing store

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

- Possible of total process memory spaces exceeds real physical memory

  → Use of Backing store



Swapping issues

- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
  - Plus consider pending I/O to/from process memory space

- Pending IO: cannot swap out as IO would occur to wrong swapped in process!
  - Or always transfer IO to kernel space, then to IO device; this is known as double buffering; adds overheads

Swapping and modern OSs

- Standard swapping is not used in Linux & Windows!
  - Modified version is used:
    - Swap only when free memory is extremely low (less than threshold)
    - Disable swapping when free memory is more than threshold

Swapping cost

- Major part of swap time is transfer time
- Total transfer time is proportional to the amount of memory swapped.
- Example:
  - 100 MB process swapping to hard disk with transfer rate of 50 MB/sec
  - Swap out time of 2(sec)+swap in of same size process
  - Total context switch swapping component time of 4 sec.

Memory allocation

Memory allocation

- Different types
  1. Contiguous
     - Each process is in a single section of memory that is contiguous to sections of others
  2. Segmentation
     - Each process is divided into different segments; each one is located in different part
  3. Paging
     - Each process is divided into same–small–size pages; some of them are swapped in/out

Criteria and problems

- CPU Utilization
  - Utilization = Percent of time a CPU is busy

$$= \frac{CPU\ time\ used}{Total\ time\ system\ is\ busy}$$

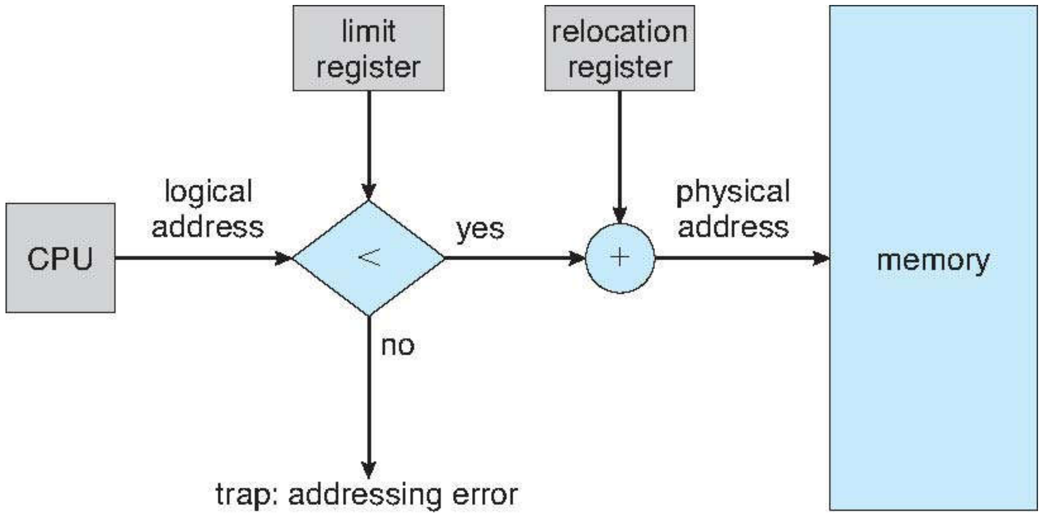
Some times is $\frac{CPU\ time\ used}{CPU\ time\ used + 2 * swapping\ time}$

Some times is $\frac{CPU\ time\ used}{\max(CPU\ time\ used, 2 * swapping\ time)}$

- Fragmentation
  - External
  - Internal

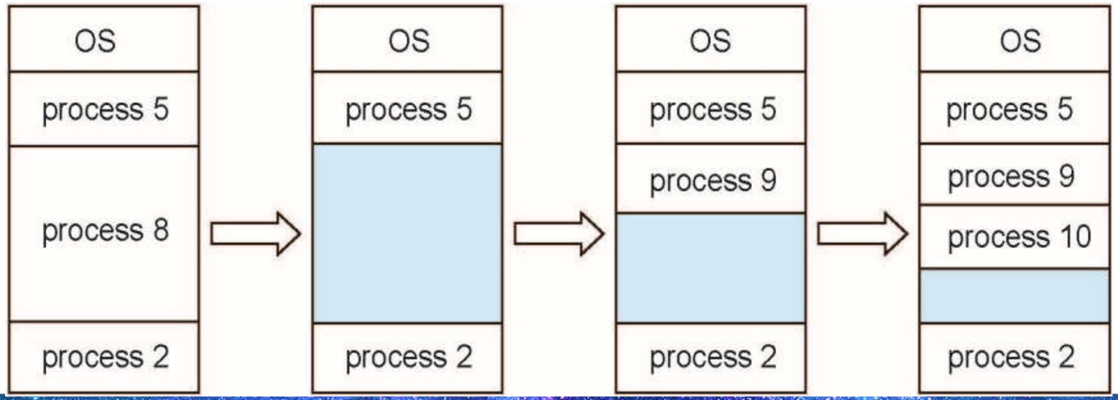
Memory Protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses
    - Each logical address must be less than the limit register



1) Contiguous allocation

- Multiple-partition memory allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (**hole**)



Dynamic storage-allocation problem

- How to satisfy a request of size ***n*** from a list of free holes?
  - **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

Fragmentation problem

- **External Fragmentation**
  - Total memory space exists to satisfy a request, but it is not contiguous

- **Internal Fragmentation**
  - Not whole of allocated memory is filled by the process memory

- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation solution

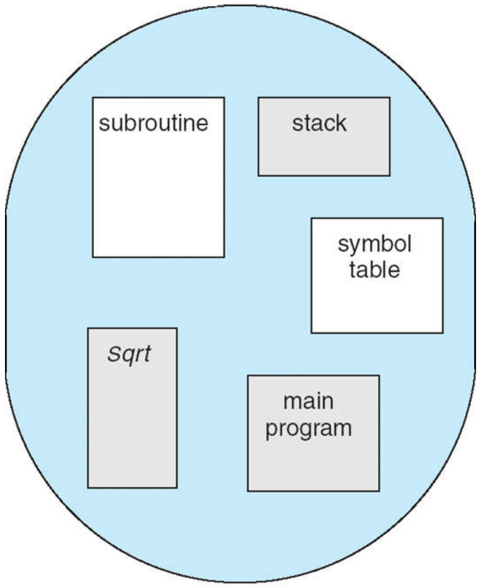
- Compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
  - Now consider that backing store has same fragmentation problems

solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous

2) Segmentation

- Memory-management scheme that supports user view of memory

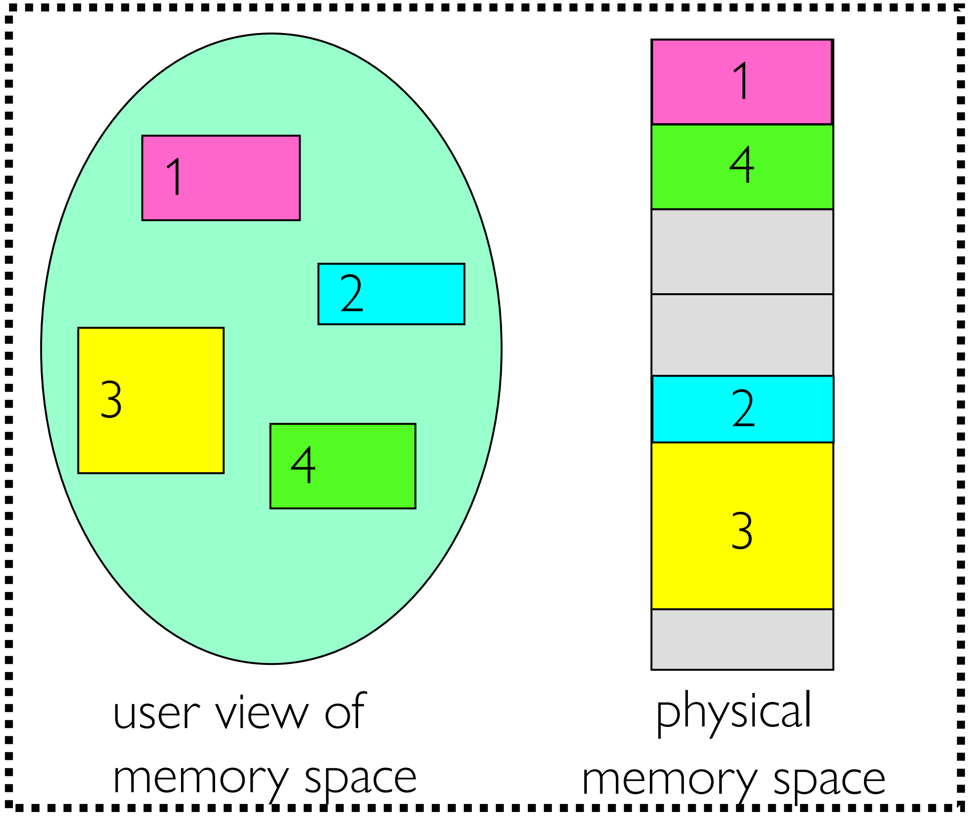
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays



logical address

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

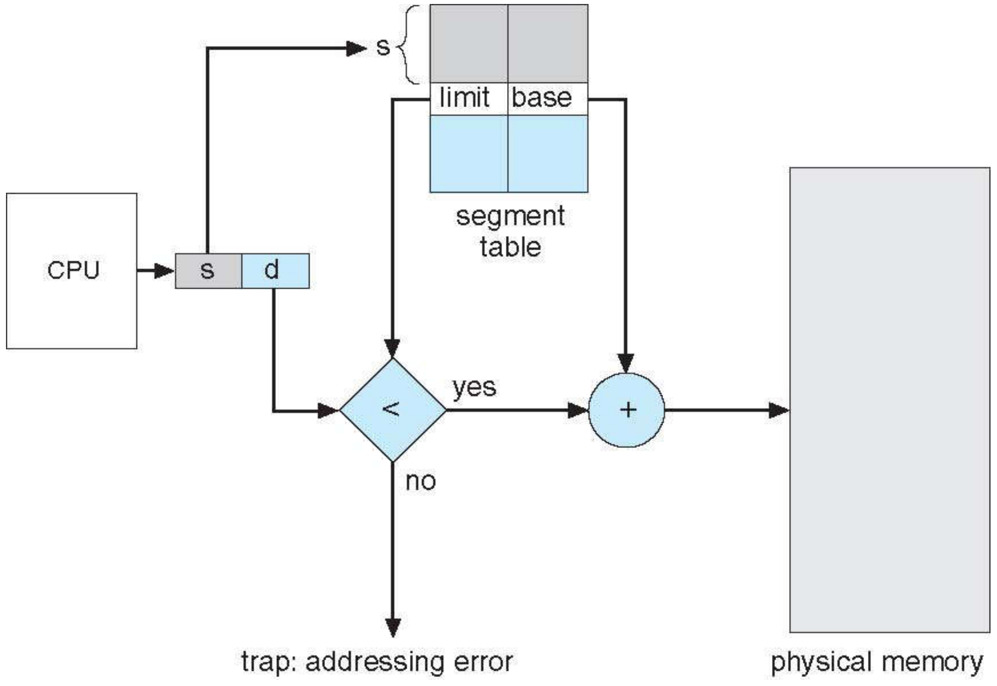


Segmentation implementation

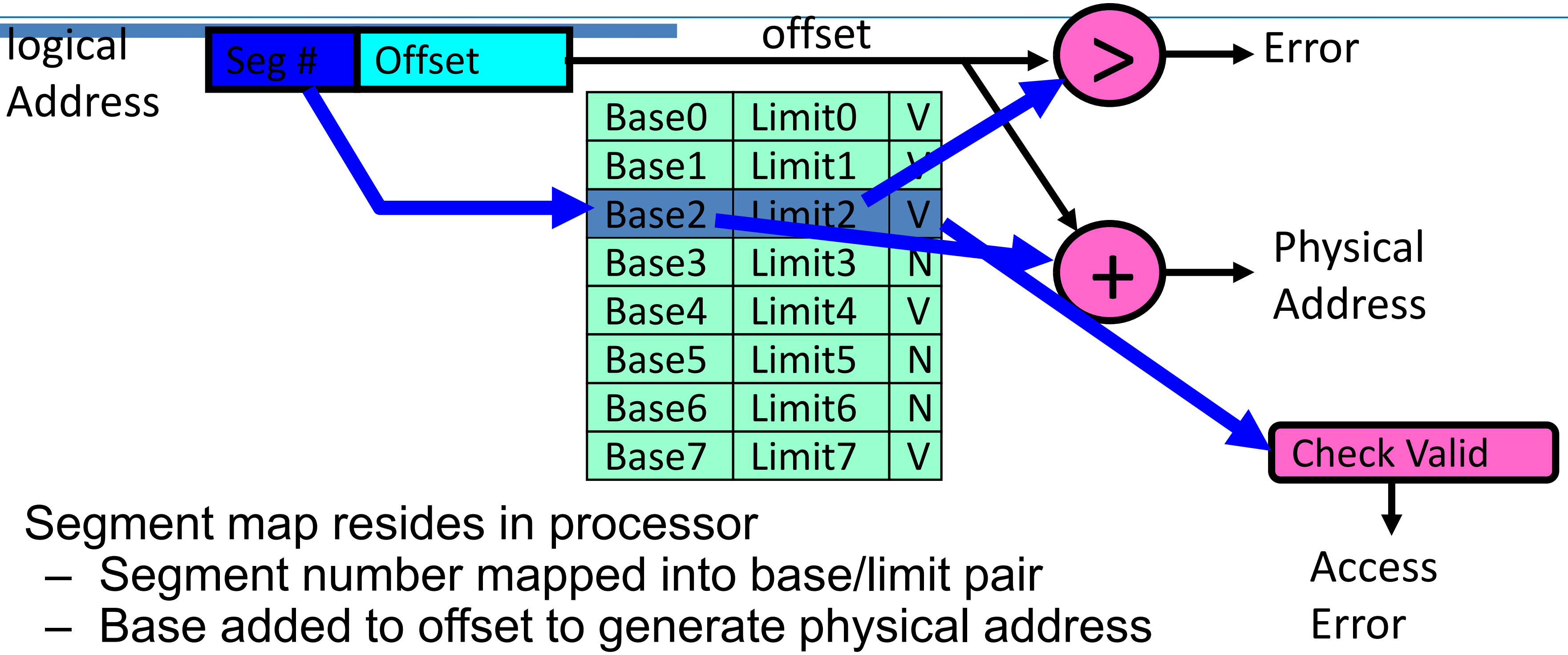
- Logical address consists of a two tuple:

  <segment-number, offset>

- **Segment table**
  - Maps two-dimensional physical addresses;
  - each table entry has:
    - **base** – contains the starting physical address where the segments reside in memory
    - **limit** – specifies the length of the segment



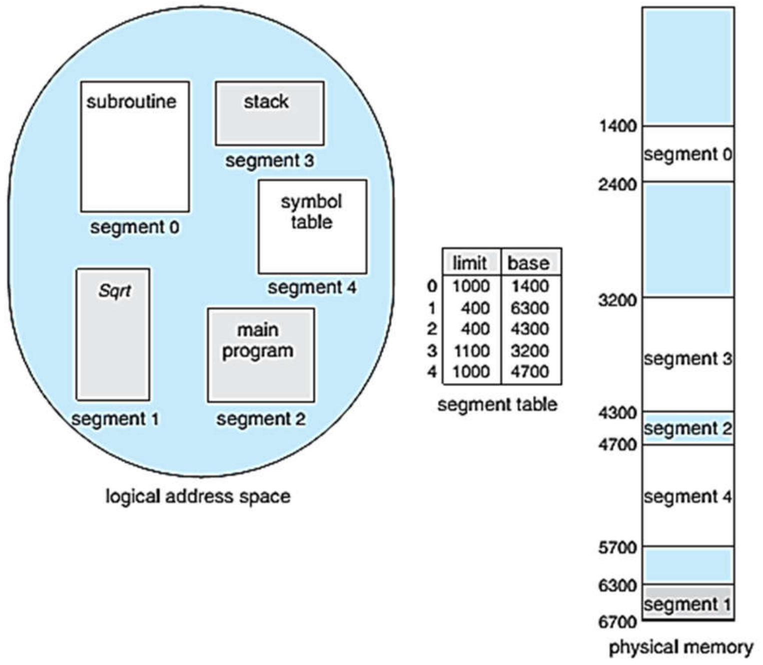
Implementation of Multi-Segment Model



- 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 logical 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

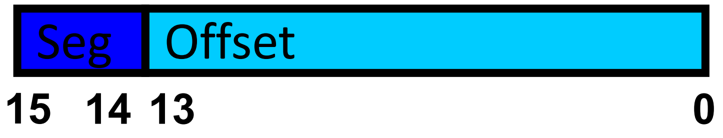
Example of segmentation

- A reference to byte 53 of segment 2:
  4300+53=4353
- A reference to byte 852 of segment 3:
  3200+852=4052
- A reference to byte 1222 of segment 0:
  Trap to OS!



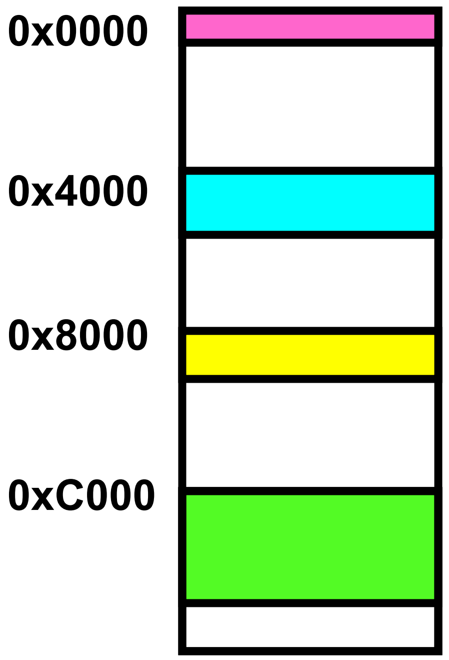
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Example: Four Segments (16 bit addresses)

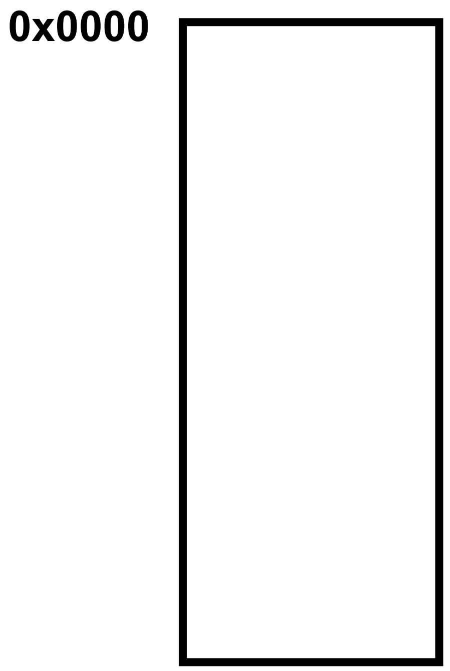


Virtual Address Format

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

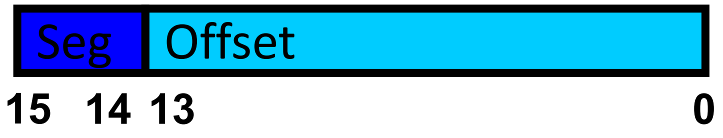


Virtual
Address Space



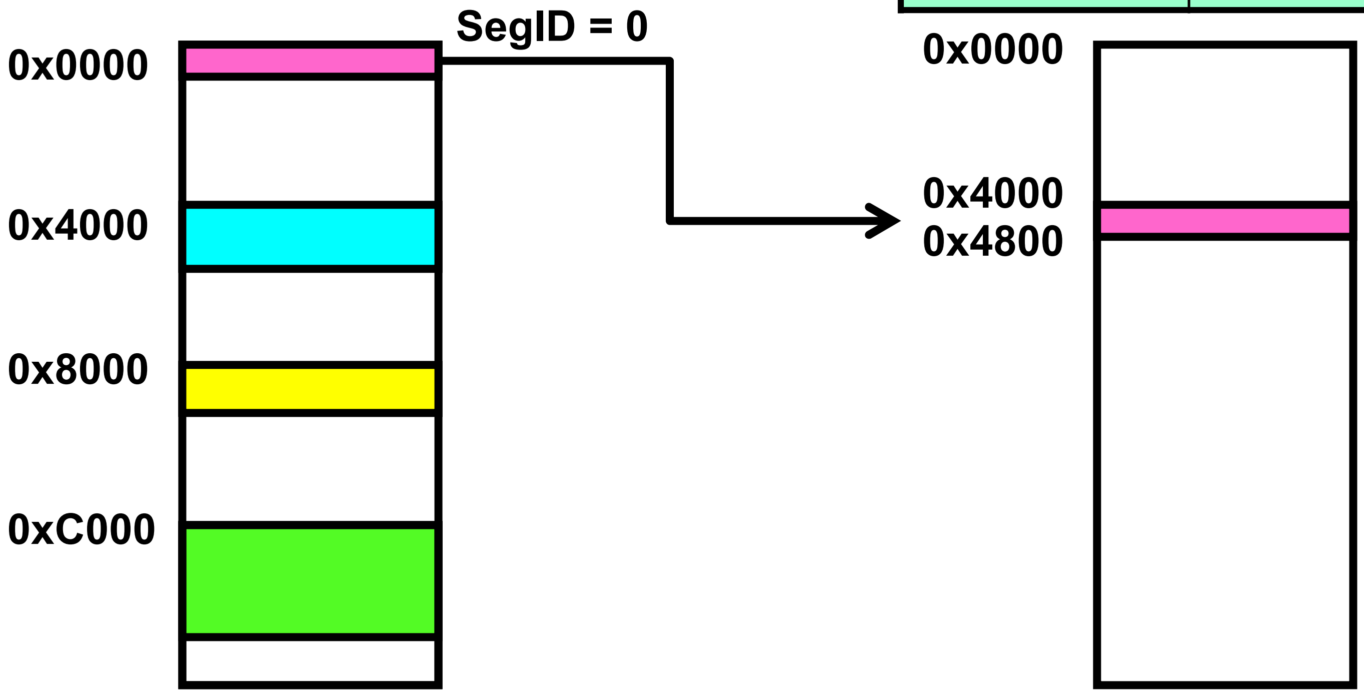
Physical
Address Space

Example: Four Segments (16 bit addresses)



Virtual Address Format

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

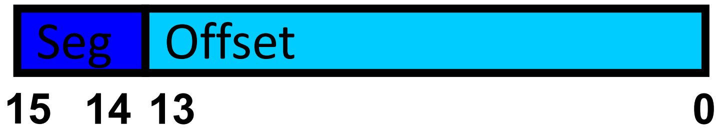


Virtual
Address Space

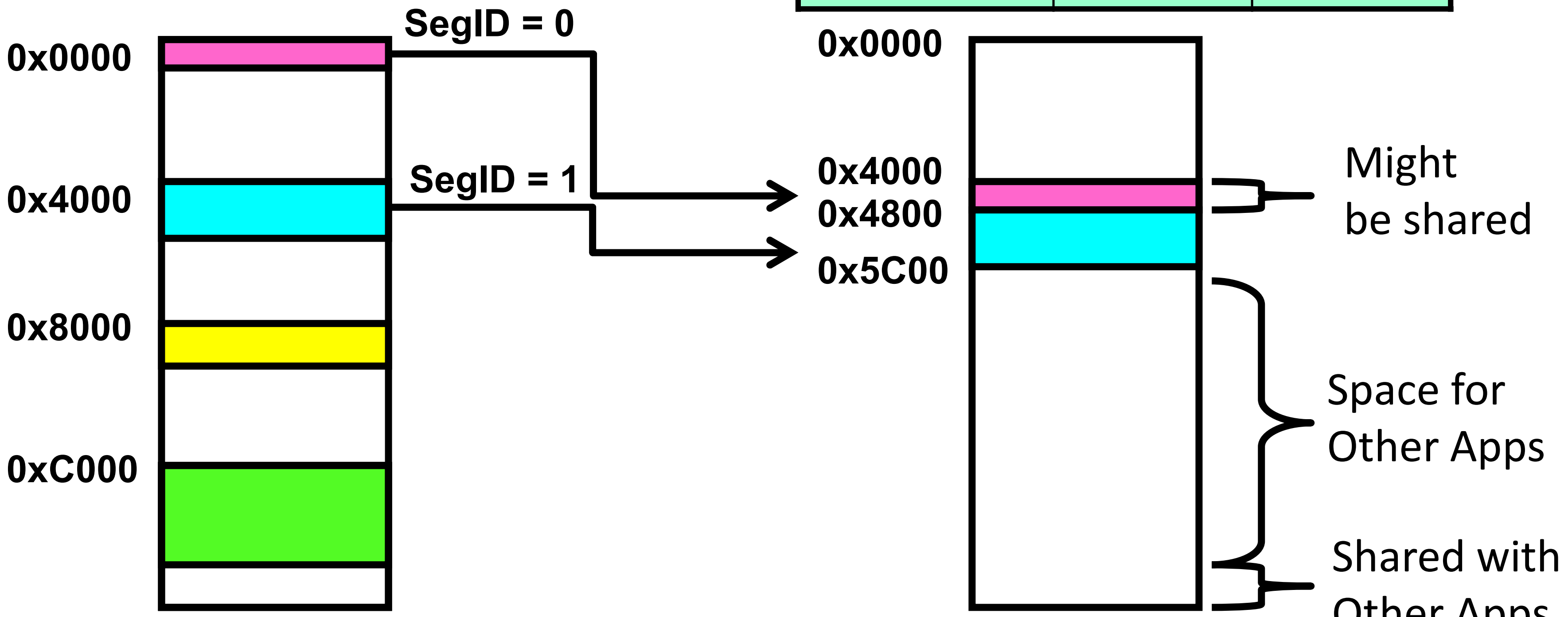
Physical
Address Space

Example: Four Segments (16 bit addresses)

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



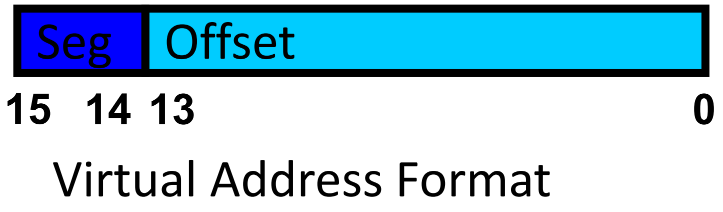
Virtual Address Format



Virtual Address Space

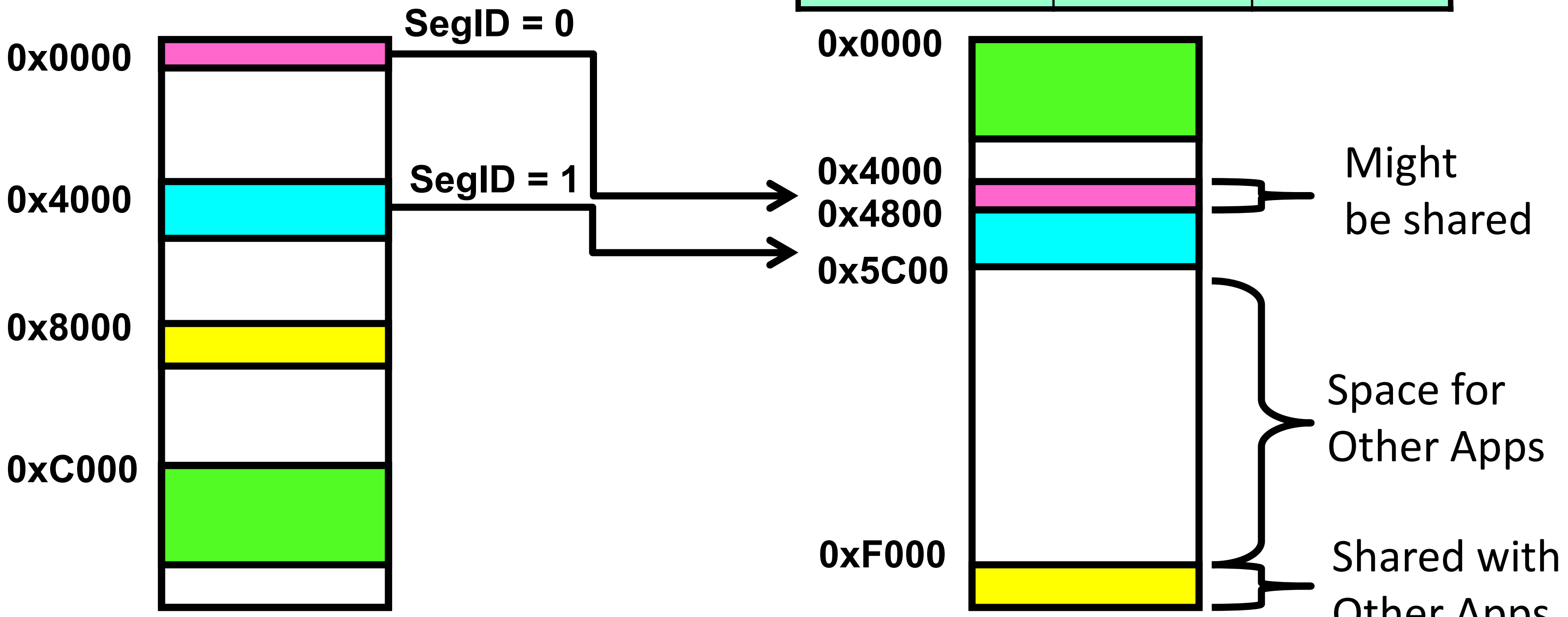
Physical Address Space

Example: Four Segments (16 bit addresses)



Virtual Address Format

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



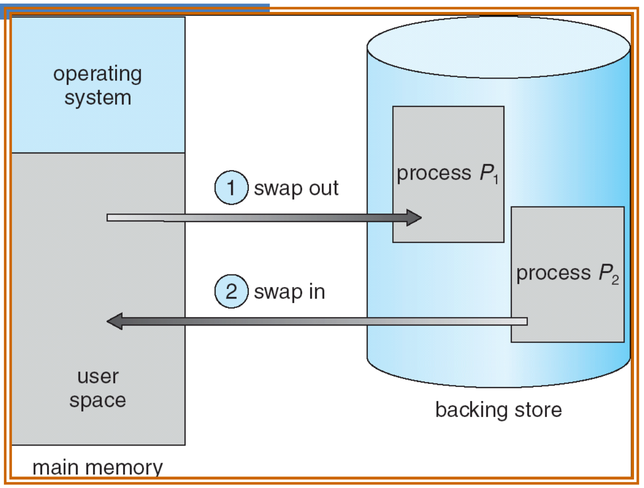
Virtual Address Space

Physical Address Space

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)

What if not all segments fit into memory?



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

Problems with Segmentation

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

3) Paging

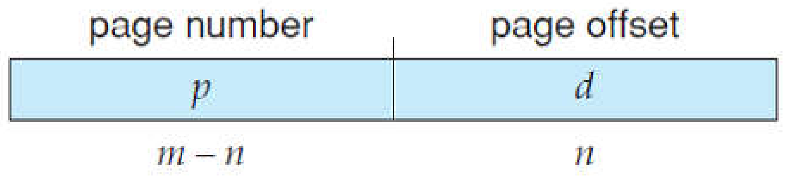
- Noncontiguous memory allocations:
  - Segmentation
  - Paging
- Paging avoids external fragmentation, and need of compaction, whereas segmentation does not.
- Process is allocated physical memory whenever physical memory is available
  - Avoid external fragmentation
  - Avoid problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames.
  - Size is power of 2, between 512 bytes to 16 Mbytes
- Divide logical memory into blocks of same size called pages.

Paging

- Keep tracks of all free frames
- To run a program with N pages, need to find N different free frames and load program.
- Setup a page table to translate logical to physical address
- Still have internal fragmentation

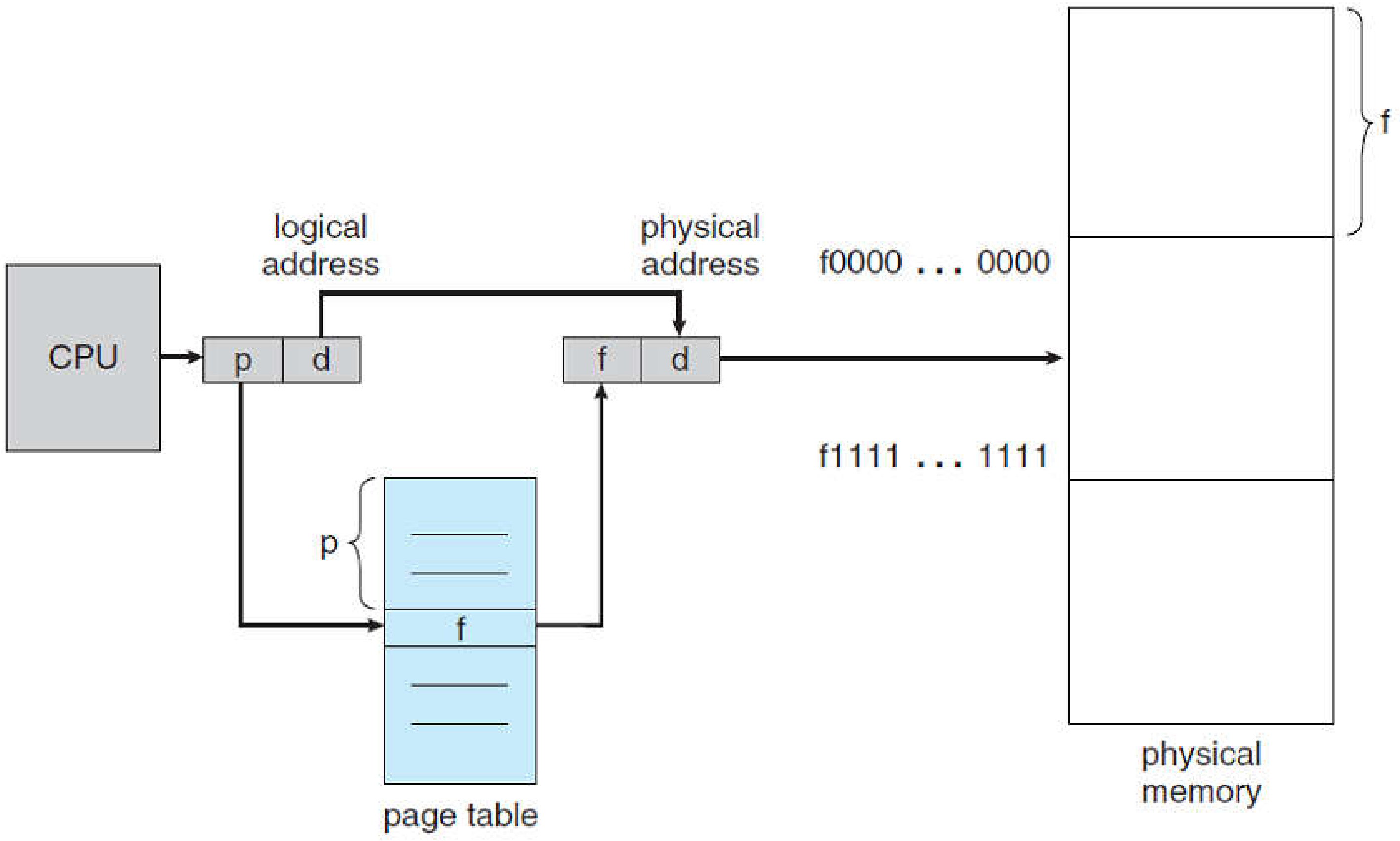
Address translation scheme

- CPU address (Logical address) is divided into two parts:
  - Page number (*p*): used as an index into a page table that contains base register of each page in physical memory
  - Page offset (*d*): combined with base address to define the physical memory address that is sent to the memory unit.

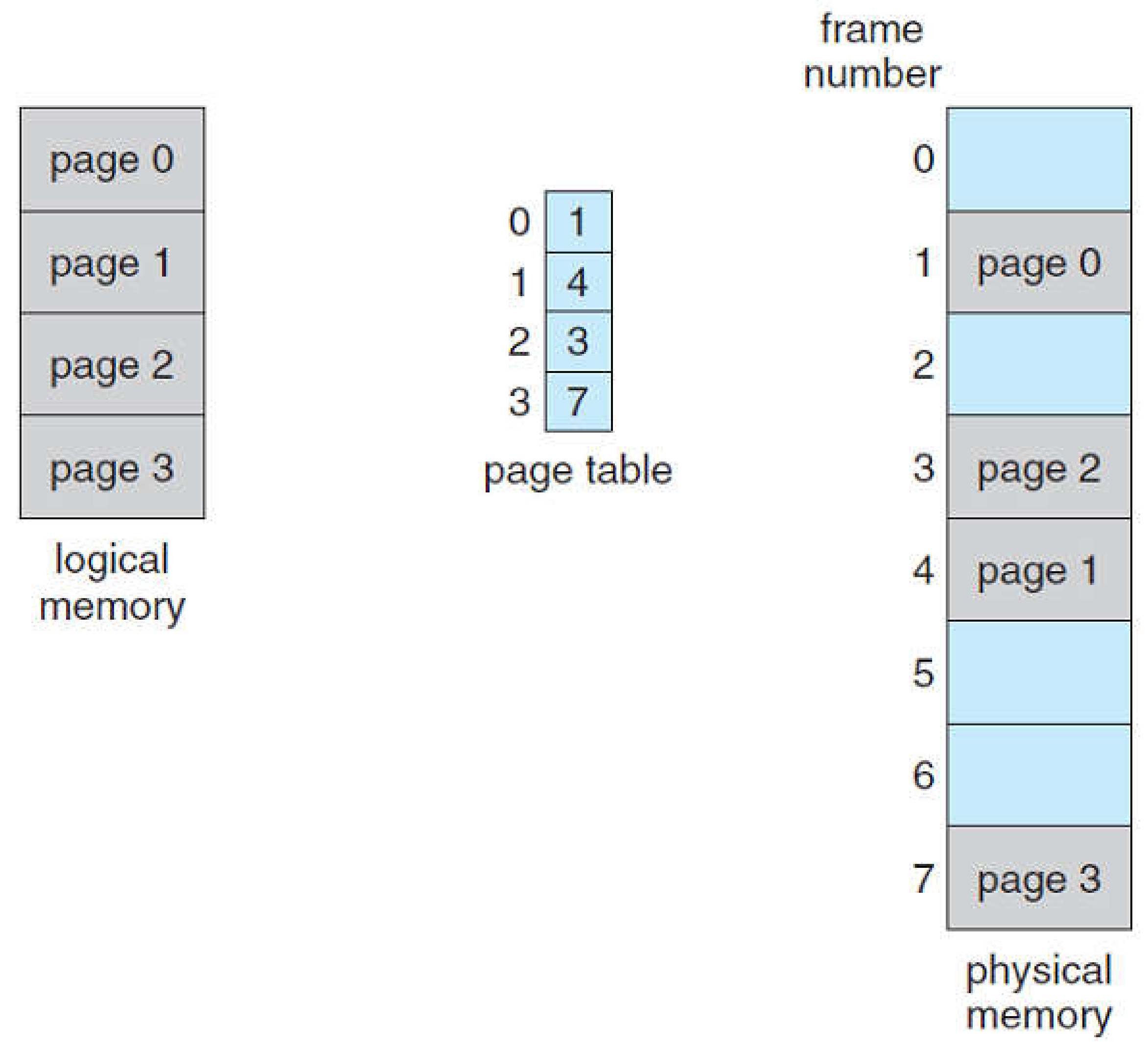


- For given logical address space 2^m and page size 2^n

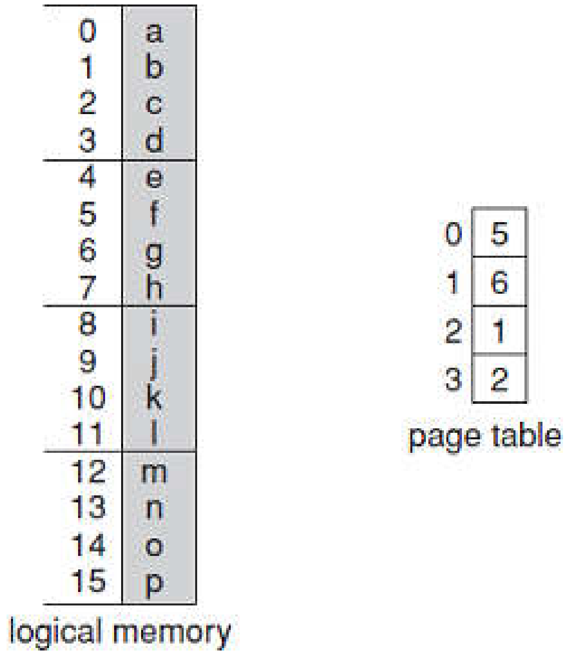
Paging hardware

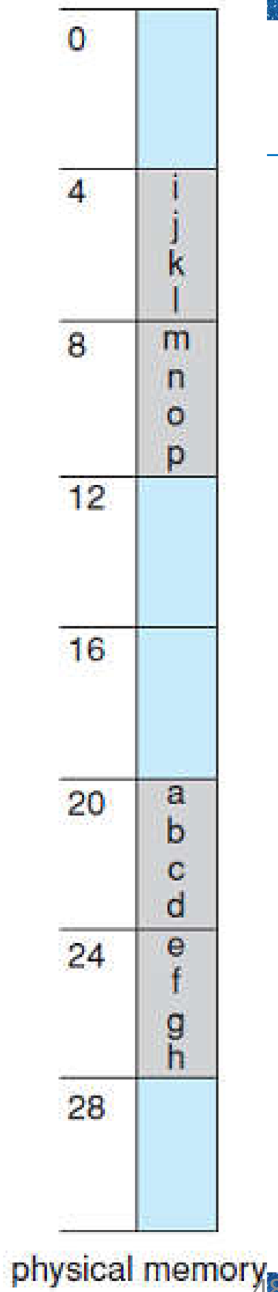


Paging model of logical and physical memory



Paging example

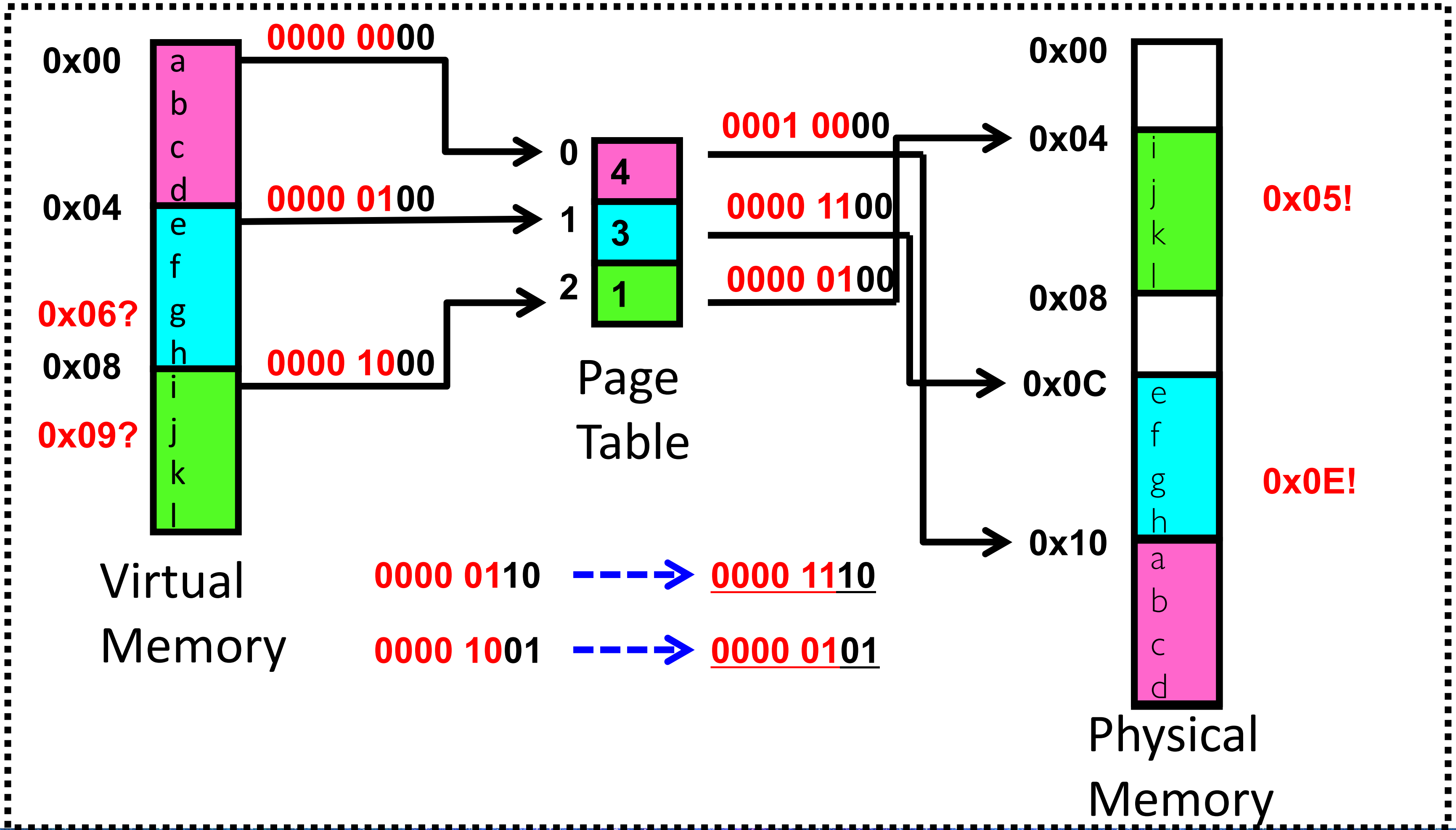




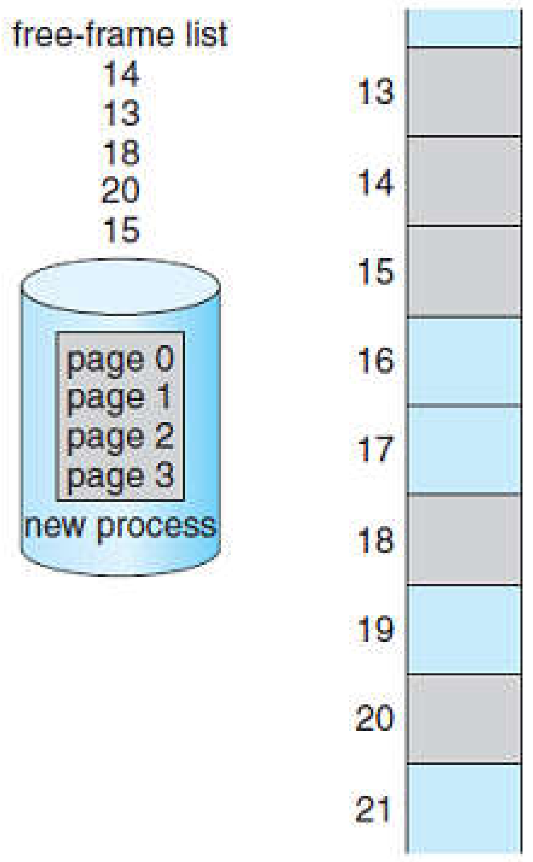
- Logical address: n=2 and m=4
- Physical memory: 32 byte memory and 4-byte pages.

Simple Page Table Example

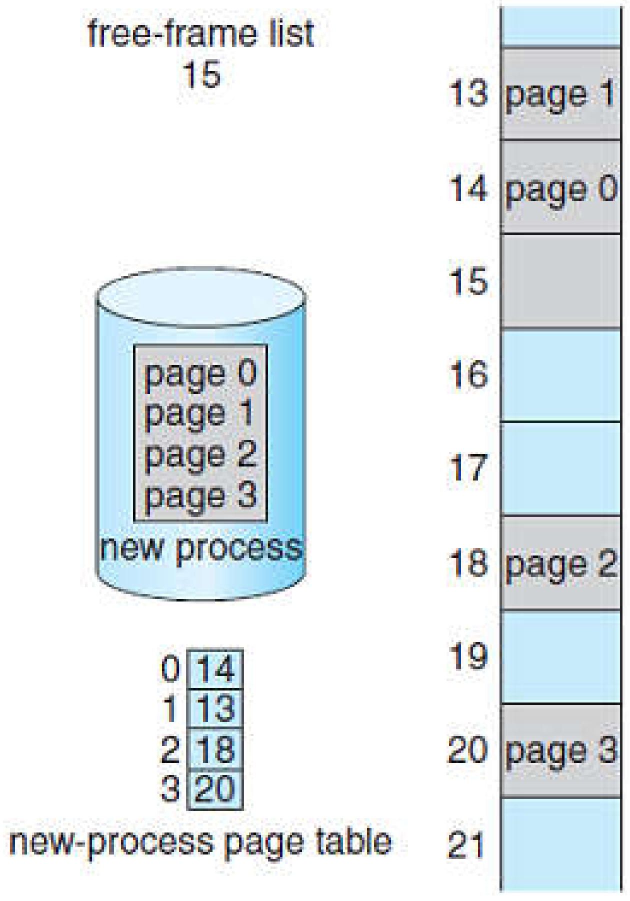
Example (4 byte pages)



Free frames



before allocation



after allocation

Paging example – Internal fragmentation

- Page size = 2048 bytes (2kB)
- Process size = 72766 bytes
- 72766/2048 = 35 pages +1086 bytes
- Internal fragmentation: 2048 – 1086 = 962 bytes
- Worst case fragmentation: 1 frame – 1 byte
- On average fragmentation: ½ frame size

Small page size vs big page size

- On average fragmentation: ½ page size, small page size are good.
- Small page size, more overhead is in the page-table, this overhead is reduced when page size increases.
- Disk I/O is more efficient when the amount of data being transferred is larger (e.g. big pages)
- Page typically are between 4 kB and 8 kB in size.

Page table implementation

Page table

- In the simplest case, the page table is implemented as a set of dedicated registers
- Page table is kept in memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses:
  - One for the page table, another for data/instruction

Translation look-aside buffer (TLB)

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffer (TLB).

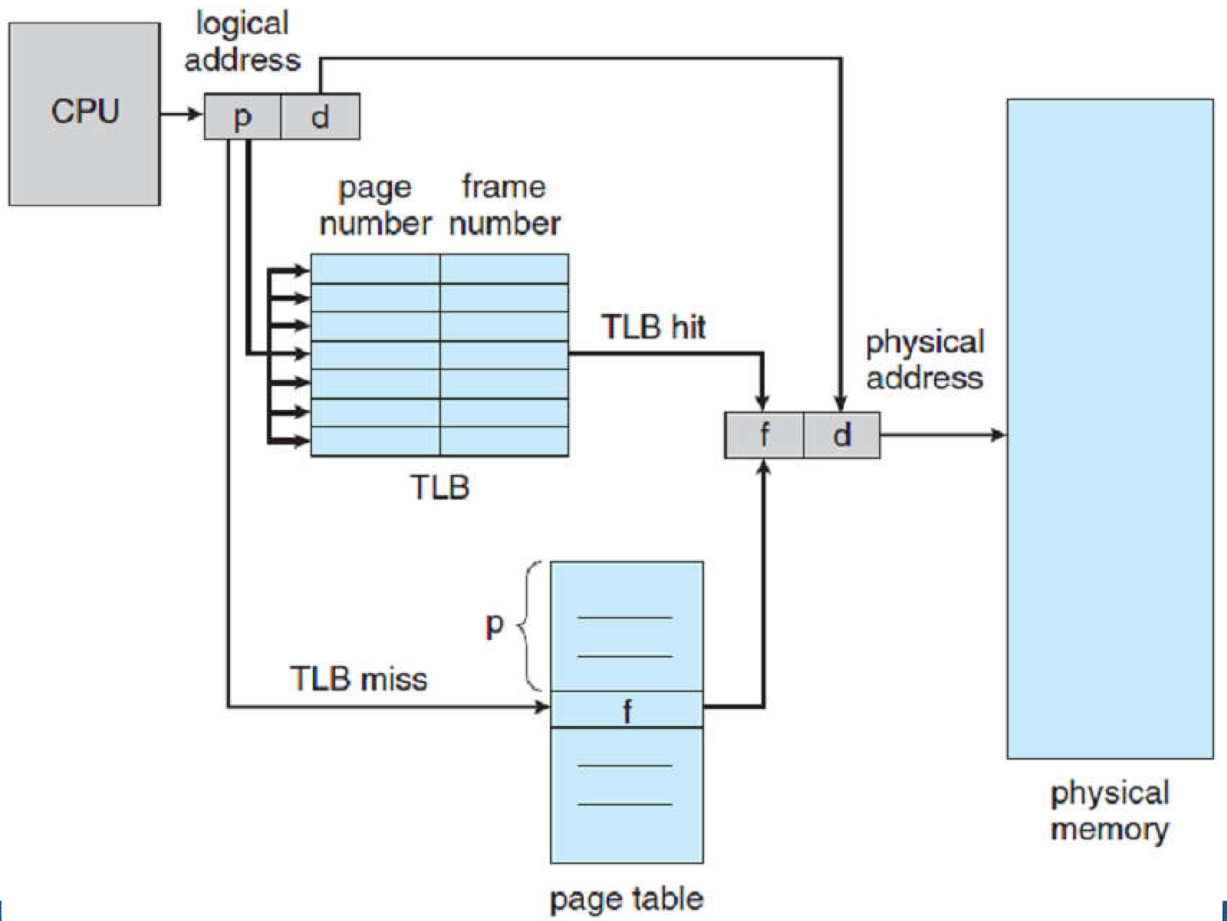
- Associative memory: parallel search

Page #	Frame #

- Address translation (*p, d*):
  - If *p* is in associative memory, get *frame#* out
  - Otherwise, get *frame#* from page table in memory

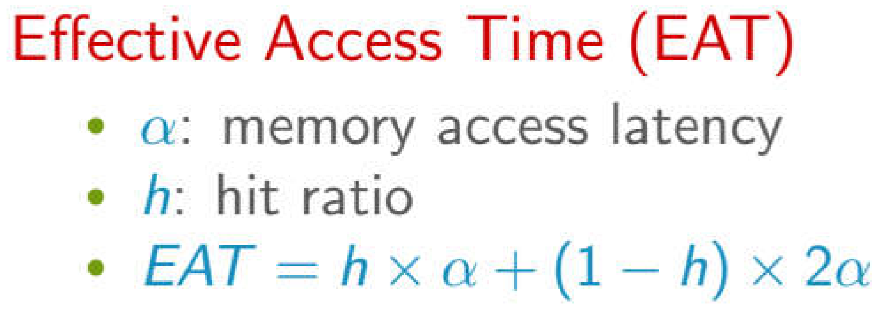
Paging hardware with TLB

- The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer** (**TLB**).



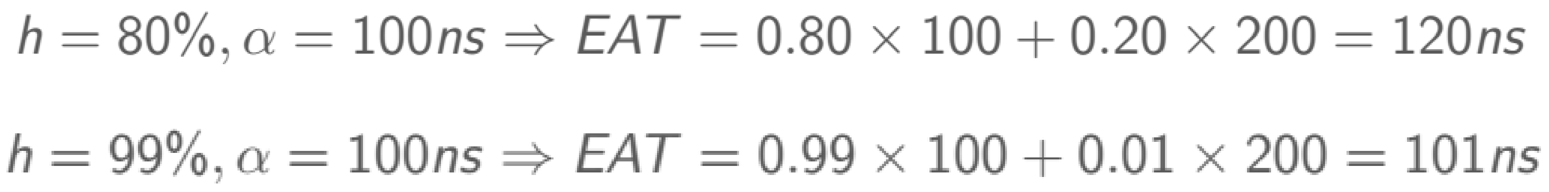
Effective access time

- Hit ratio: percentage of times that a page number is found in the TLB.



Effective Access Time (EAT)

- α: memory access latency
- h: hit ratio
- $EAT = h \times \alpha + (1 - h) \times 2\alpha$



$$h = 80\%, \alpha = 100ns \Rightarrow EAT = 0.80 \times 100 + 0.20 \times 200 = 120ns$$

$$h = 99\%, \alpha = 100ns \Rightarrow EAT = 0.99 \times 100 + 0.01 \times 200 = 101ns$$

More about TLB

- Some TLBs store address-space identifier (ASID) in each TLB entry
  - Uniquely identifies each process to provide address-space protection for that process
  - Otherwise, need to flush at every context switch
- TLB is typically small (64 to 1024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time.
  - Replacement policies must be considered.

Memory protection

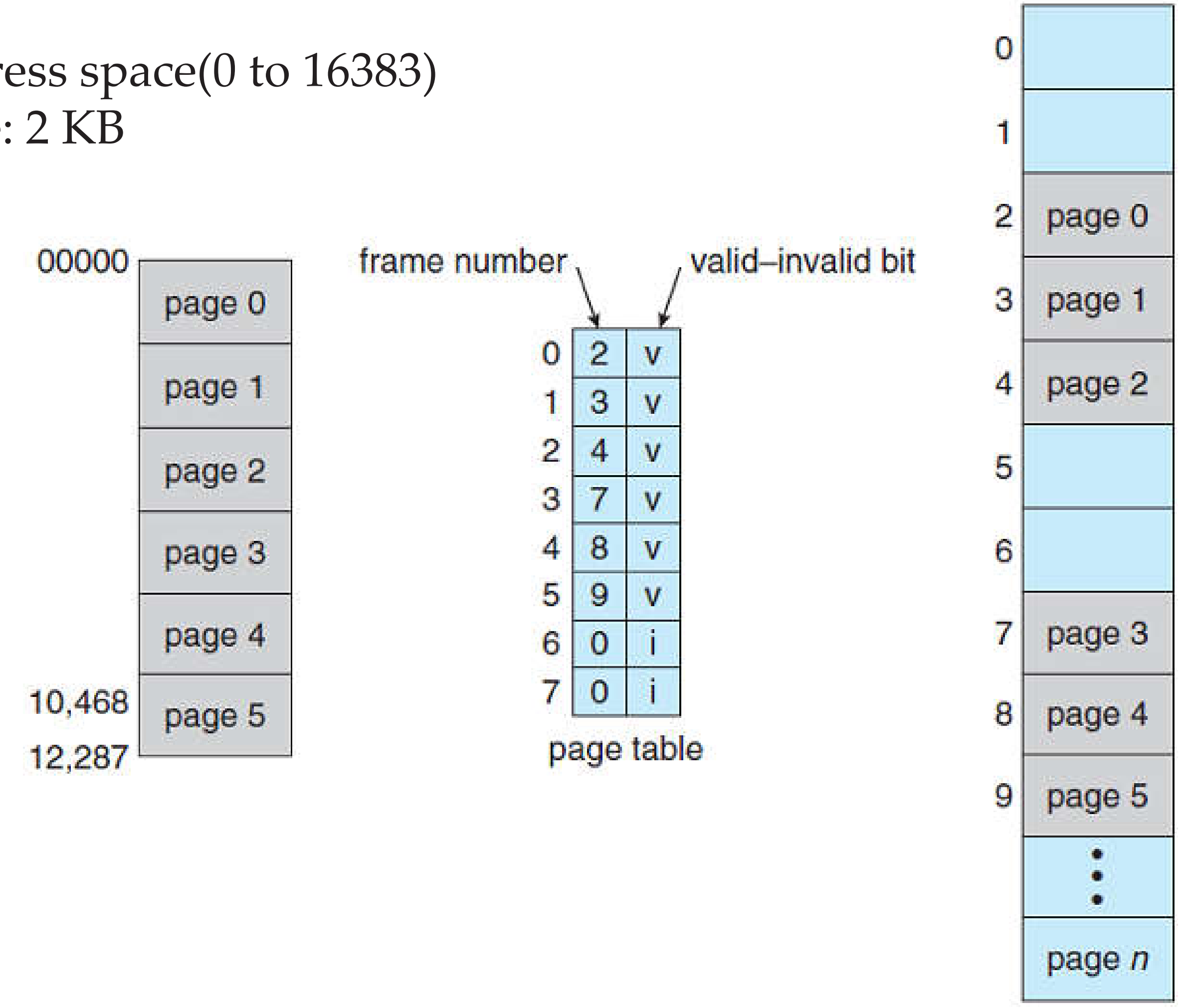
- Memory protection is implemented by protection bit with each frame to indicate if read-only or read-write access is allowed.

- Valid-invalid bit attached to each entry in page table:
  - Valid indicates that the page is in the process logical address space (legal page)
  - Invalid indicates that the page is not in the process logical address space (illegal page)
  - Or use page-table length register (PTLR)

- Any violation result in trap to the kernel

Valid/invalid bit in a page table

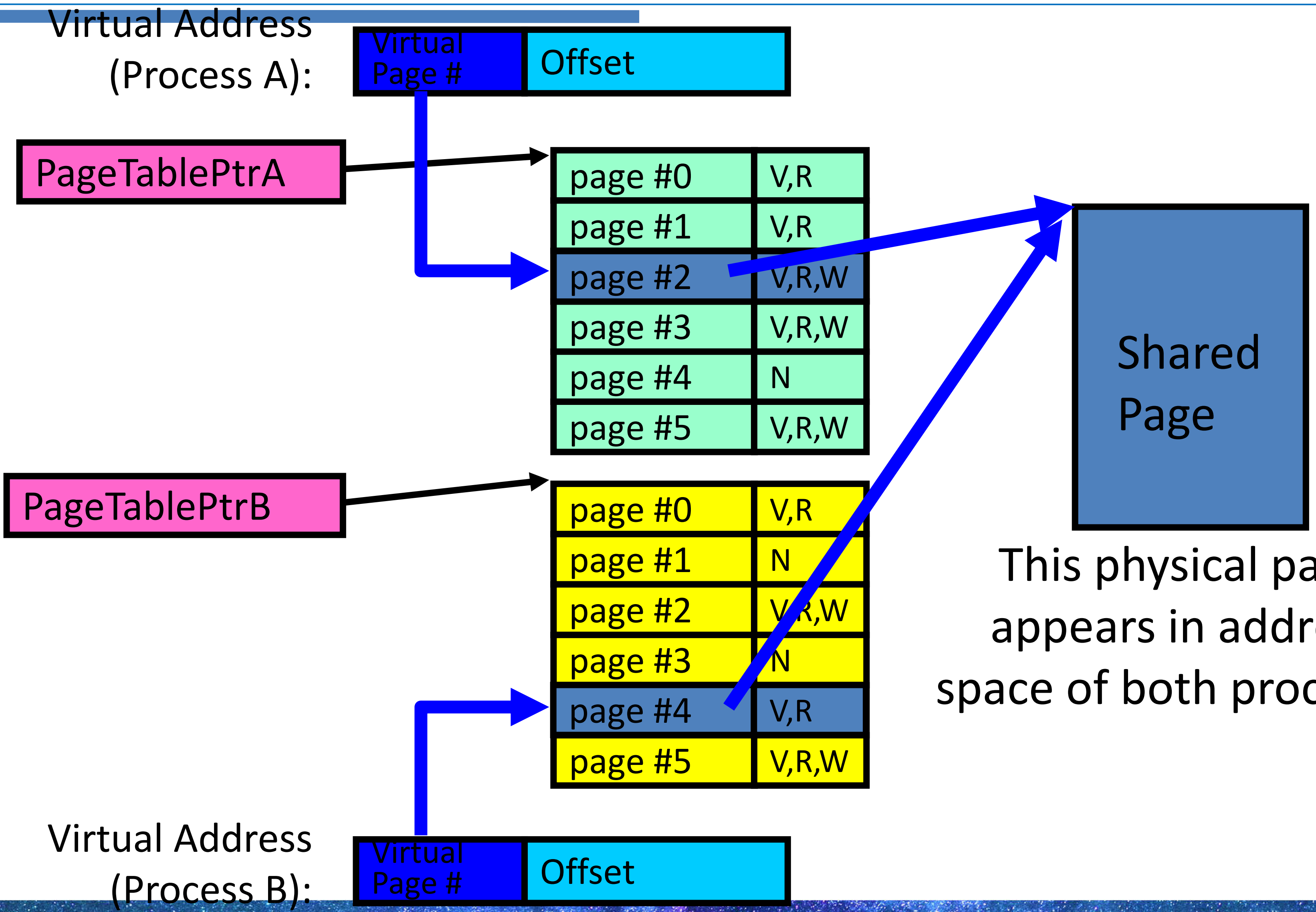
14-bit address space(0 to 16383)
a page size: 2 KB



Shared pages

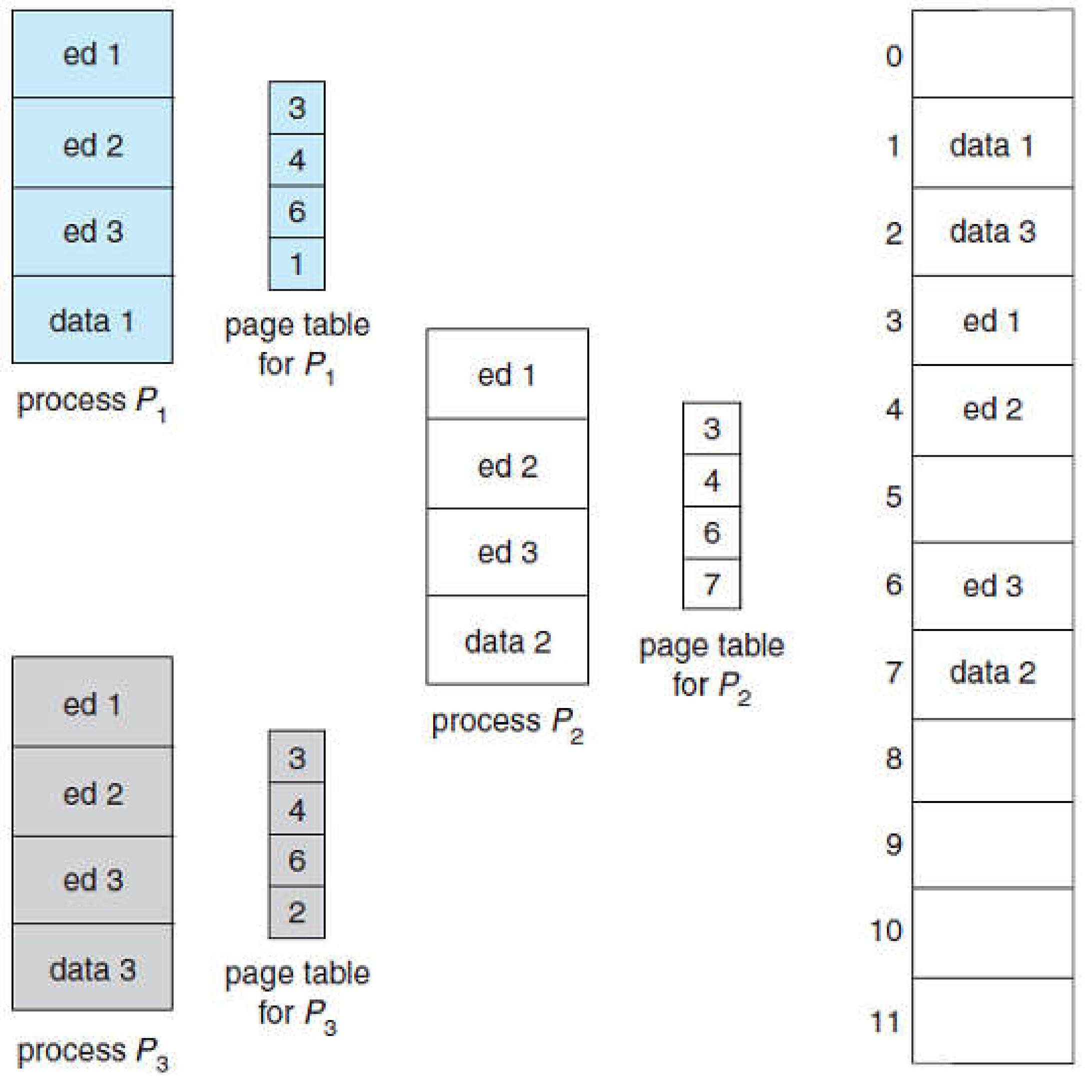
- Shared code
  - One copy of read-only (reentrant) code shared among processes (e.g., text editors)
  - Similar to multiple threads sharing the same process space.
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The page for the private code and data can appear anywhere in the logical address space

What about Sharing?

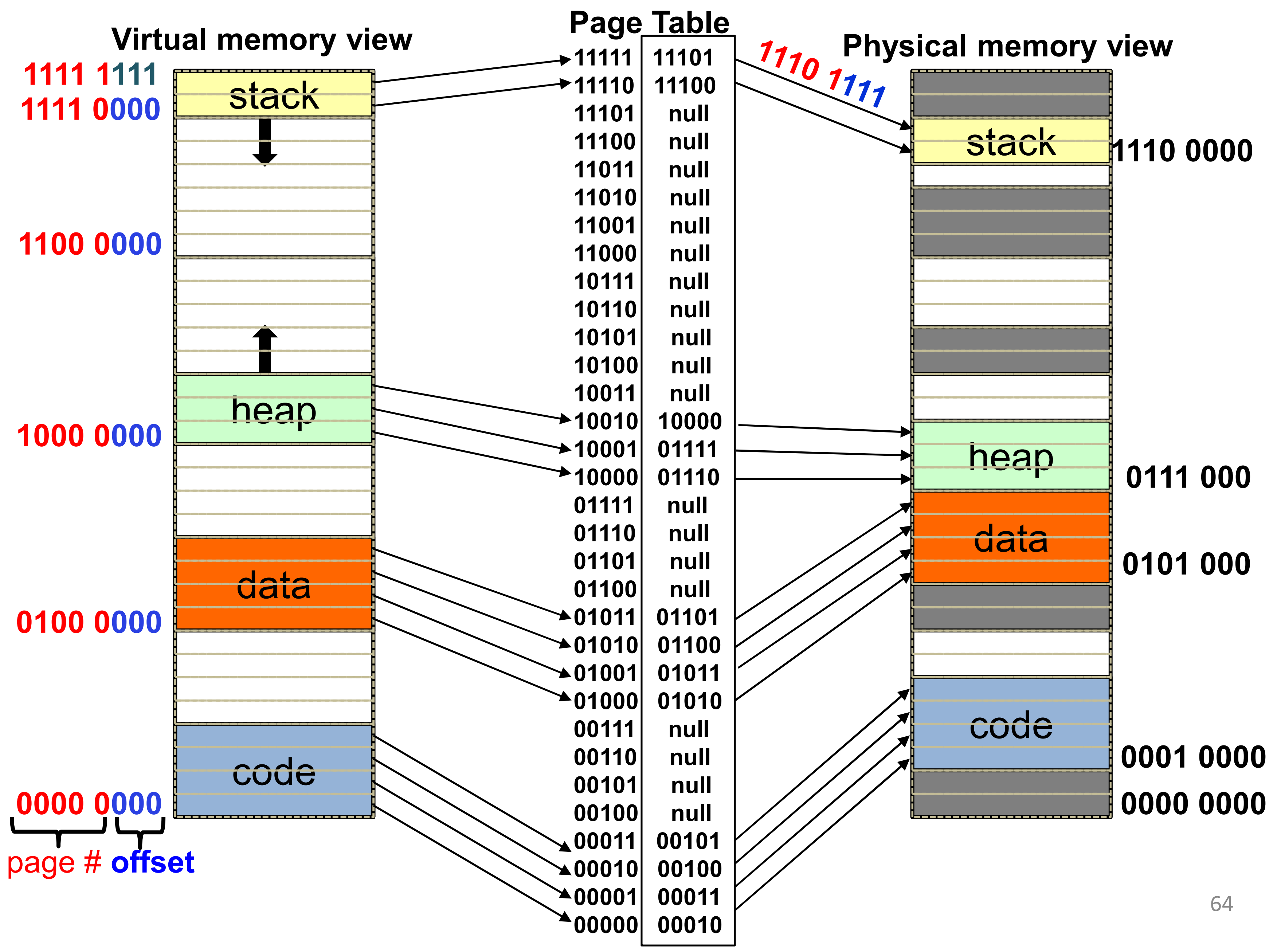


This physical page appears in address space of both processes

Shared pages - Example

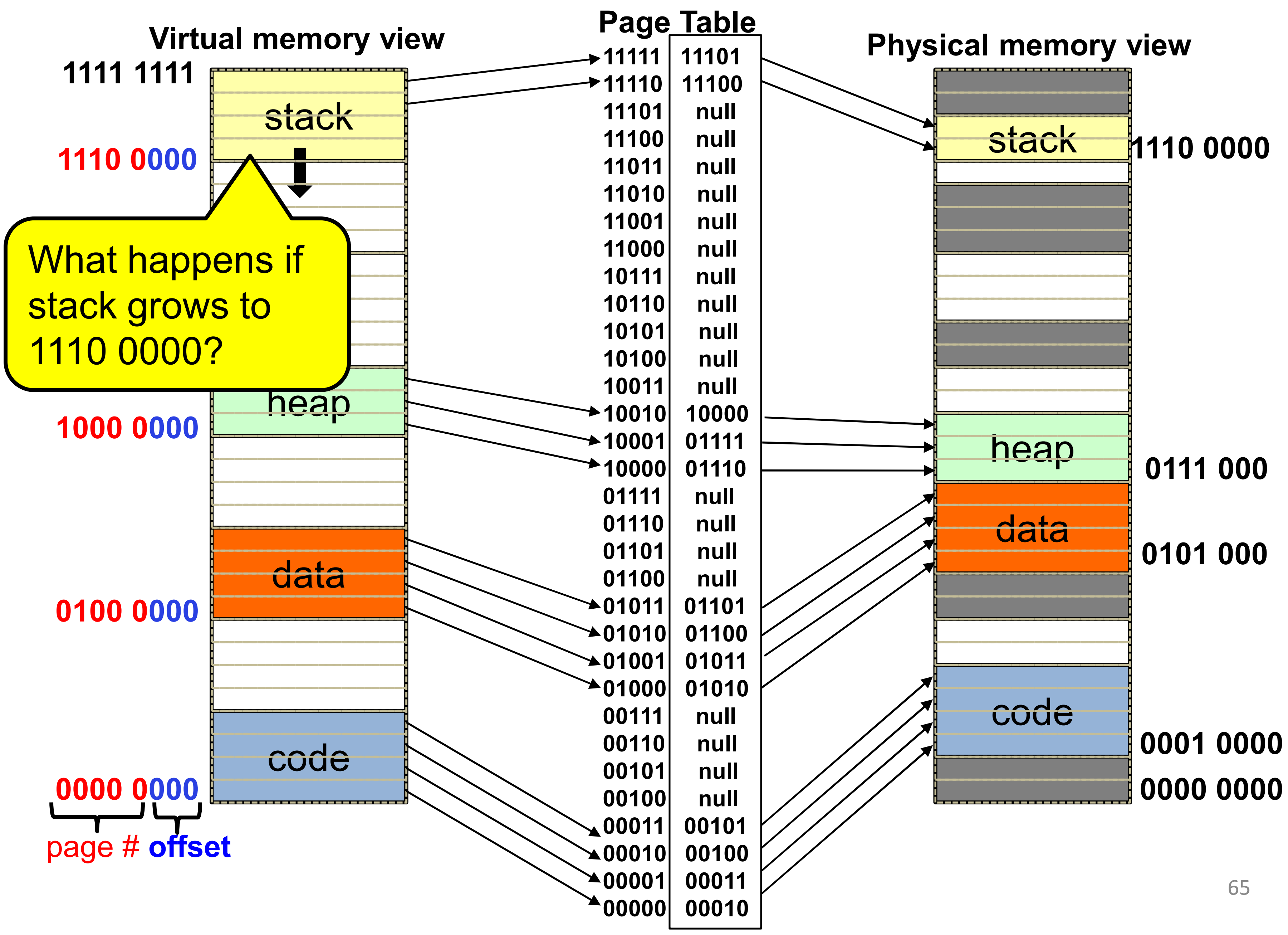


Summary: Paging



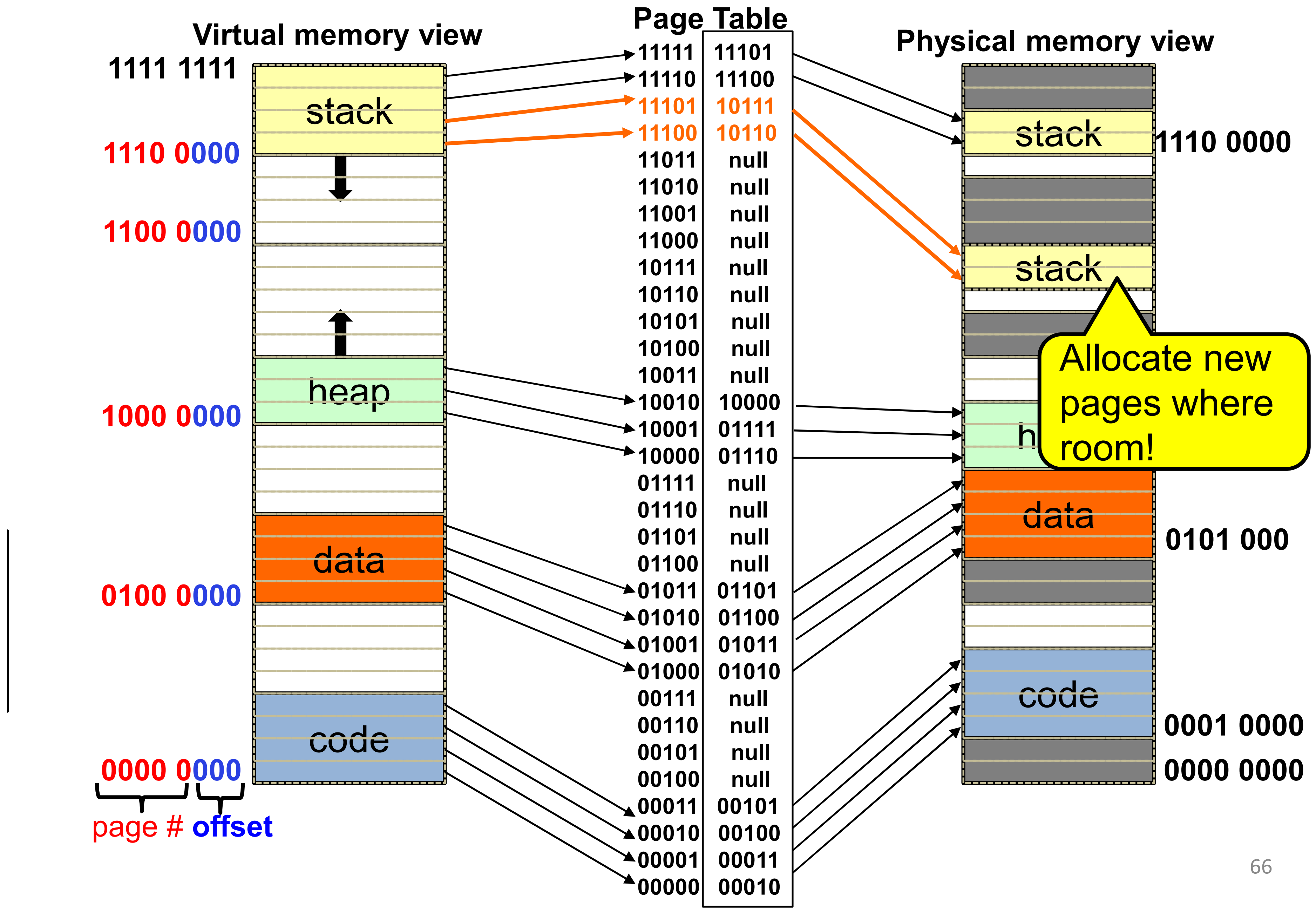
11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Summary: Paging



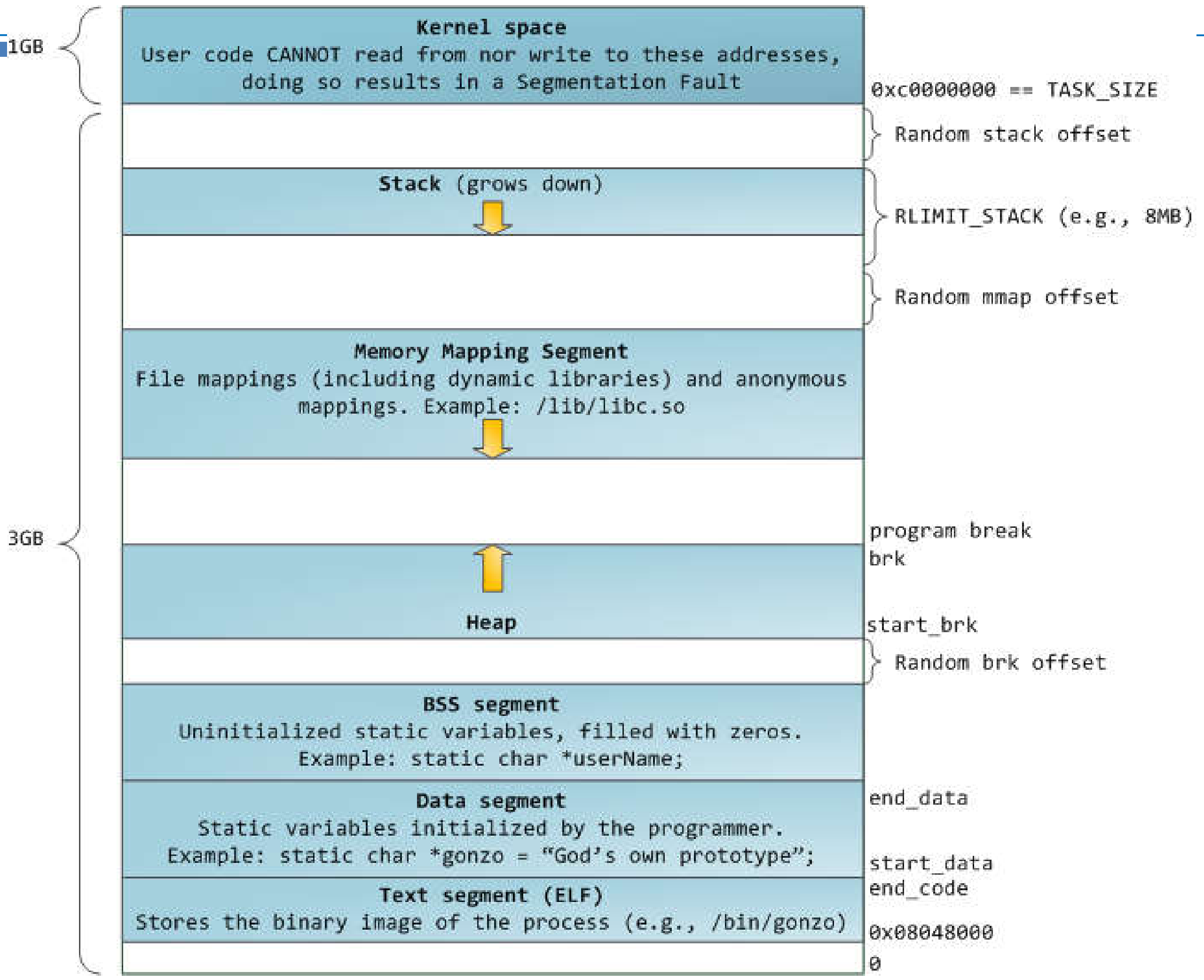
11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Summary: Paging



11111	11101
11110	11100
11101	10111
11100	10110
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Example: Memory Layout for Linux 32-bit


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

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

Problem of big page tables

- Memory structure for paging can get huge using straight-forward methods.

- Consider a 32-bit logical address space on a modern computers:
  - Page size of 4 kB = 2^{12}
  - Page table would have 1 million entries ($2^{32}/2^{12}$)
  - If each entry is 4 B: 4 MB of physical address space memory for page table alone.
  - The amount of memory used, cost a lot
  - Don’t want to allocate that contiguously in the memory

Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- Analysis
  - Pros
    - Simple memory allocation
    - Easy to share
  - Con: What if address space is sparse?
    - E.g., on UNIX, code starts at 0, stack starts at (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

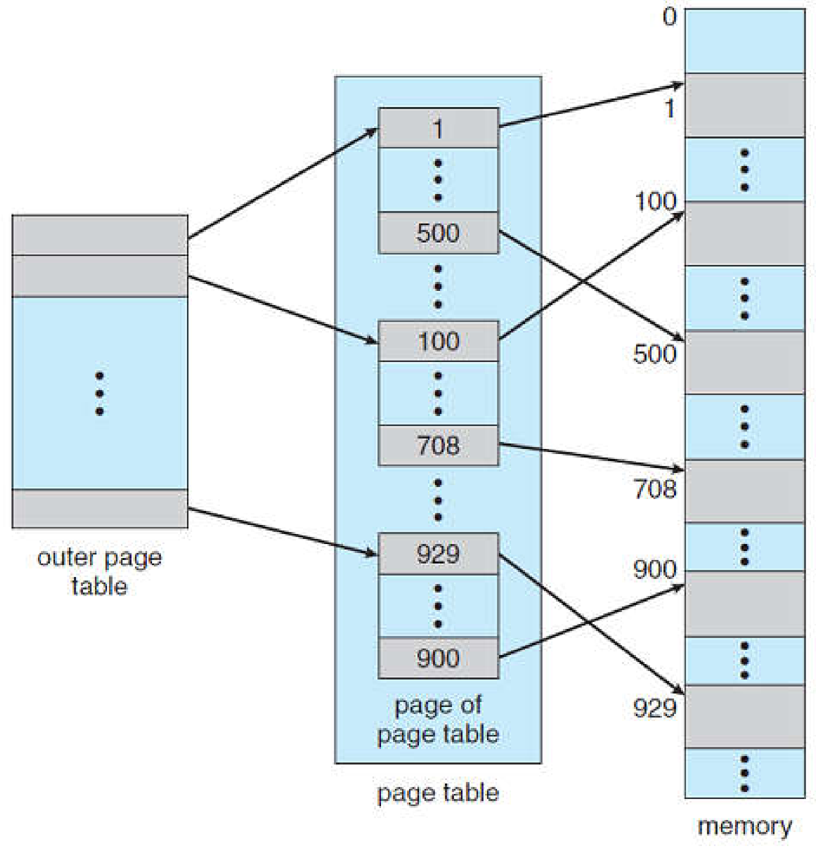
- How about multi-level paging or combining paging and segmentation?

Solutions to maintain huge page tables

- A) Hierarchical paging
- B) Hashed page tables
- C) Inverted page tables

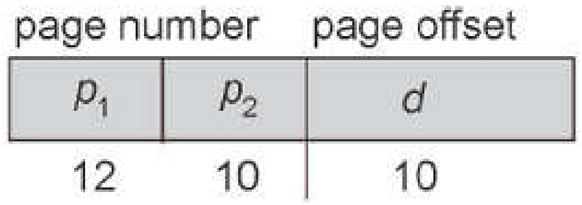
A) Hierarchical paging

- Use of two-level page table!
- We then page the page table!



Two-level paging - example

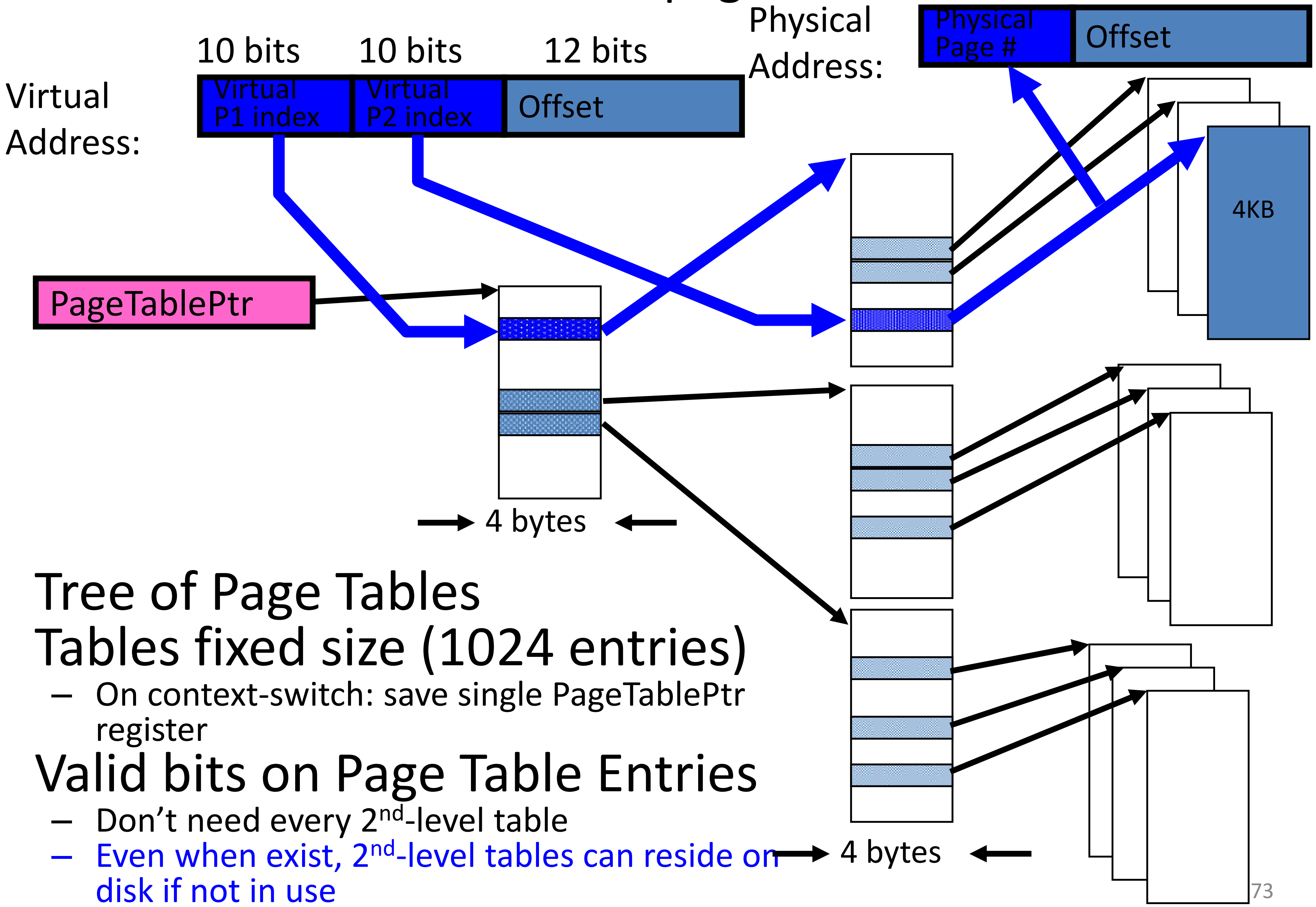
- A logical address, on 32-bit machine with 1 kB page size, is divided:
  - A page offset consisting 10 bits
  - A page number consisting of 22 bits
- Since the page table is paged, the page number is divided into:
  - A 12-bit page number
  - A 10 bit page offset
- Thus a logical address is:



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

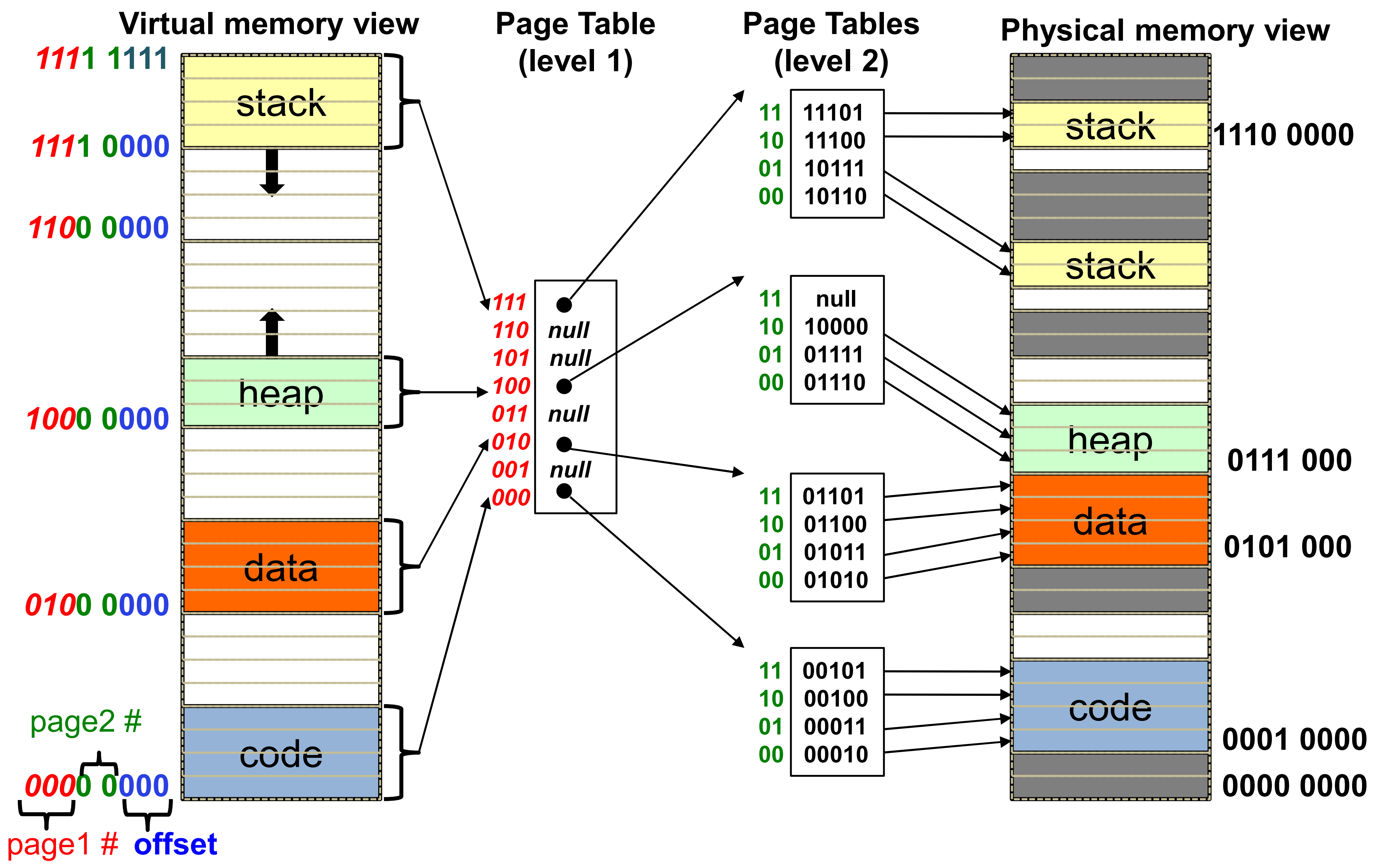
- Known as forward-mapped page table

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

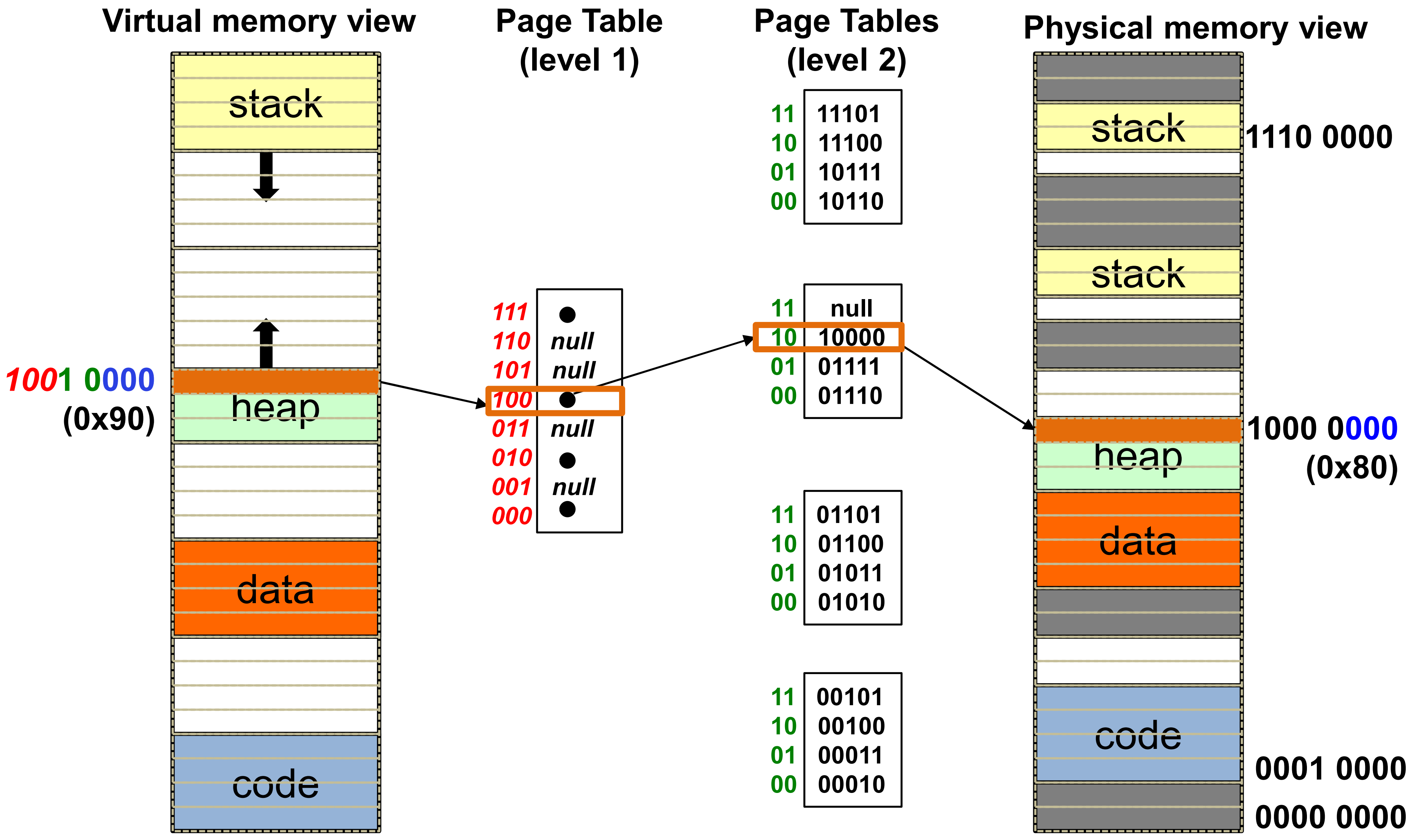


- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don’t need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Summary: Two-Level Paging

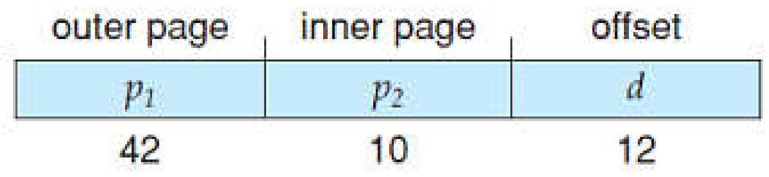


Summary: Two-Level Paging



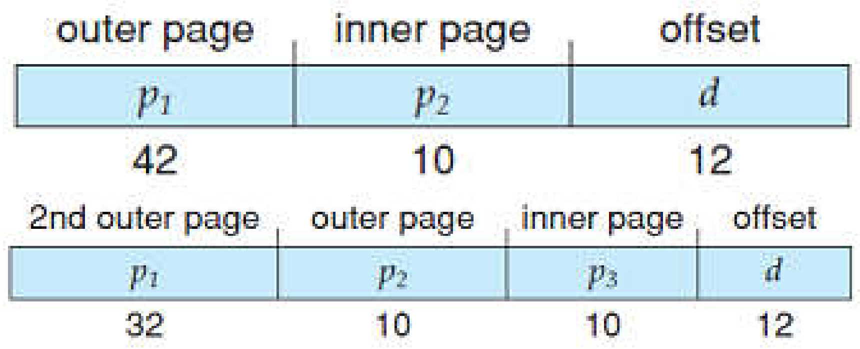
64-bit logical address space

- Even two-level paging scheme is not sufficient
- If page size of 4 kB (2^{12})
  - Page table entries = 2^{52}
  - Assuming each page table entry size = 4 B
  - If two-level scheme, inner page table could be 2^{10}, 4 kB entries
  - Outer page table has 2^{42} entries or 2^{44} B
  - Address would look like:



Three-level paging scheme

- One solution is to add a 2nd outer page table
- But in the following example, the 2nd outer page table is still 2^{34} bytes in size!
- And possible 4 memory access to get to one physical memory location:



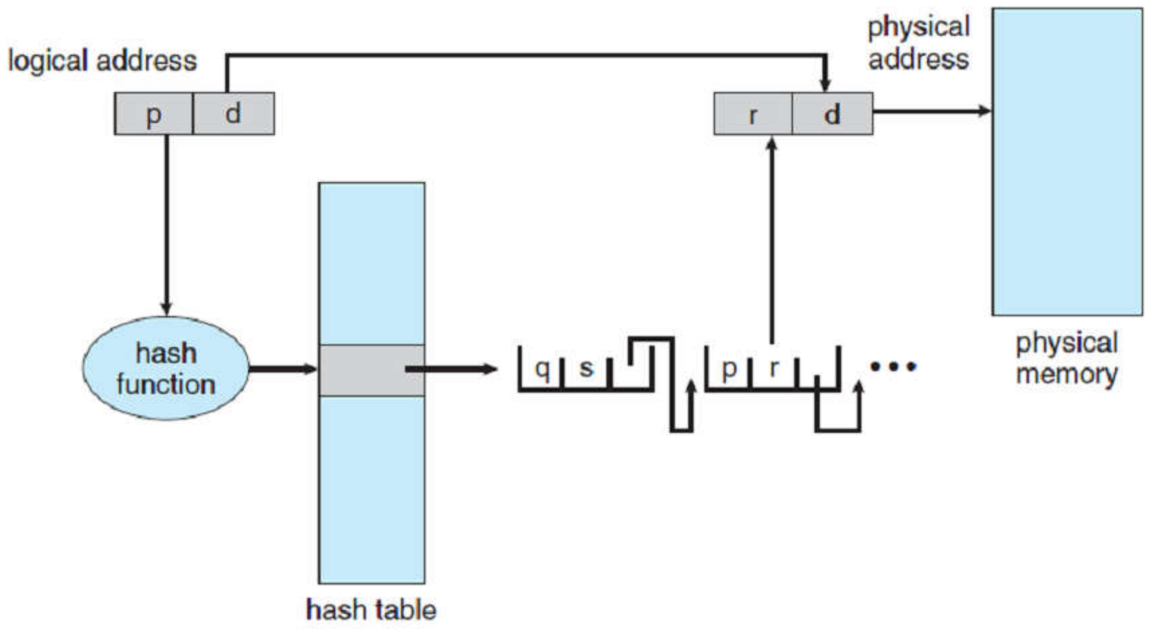
B) Hashed page tables

- Common in address space > 32 bits
- The logical page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location

Hash page table scheme

- Each element contains
  - Logical page number
  - Physical frame number
  - Pointer to the next node

- Search is done serially in the linke list



C) Inverted page table

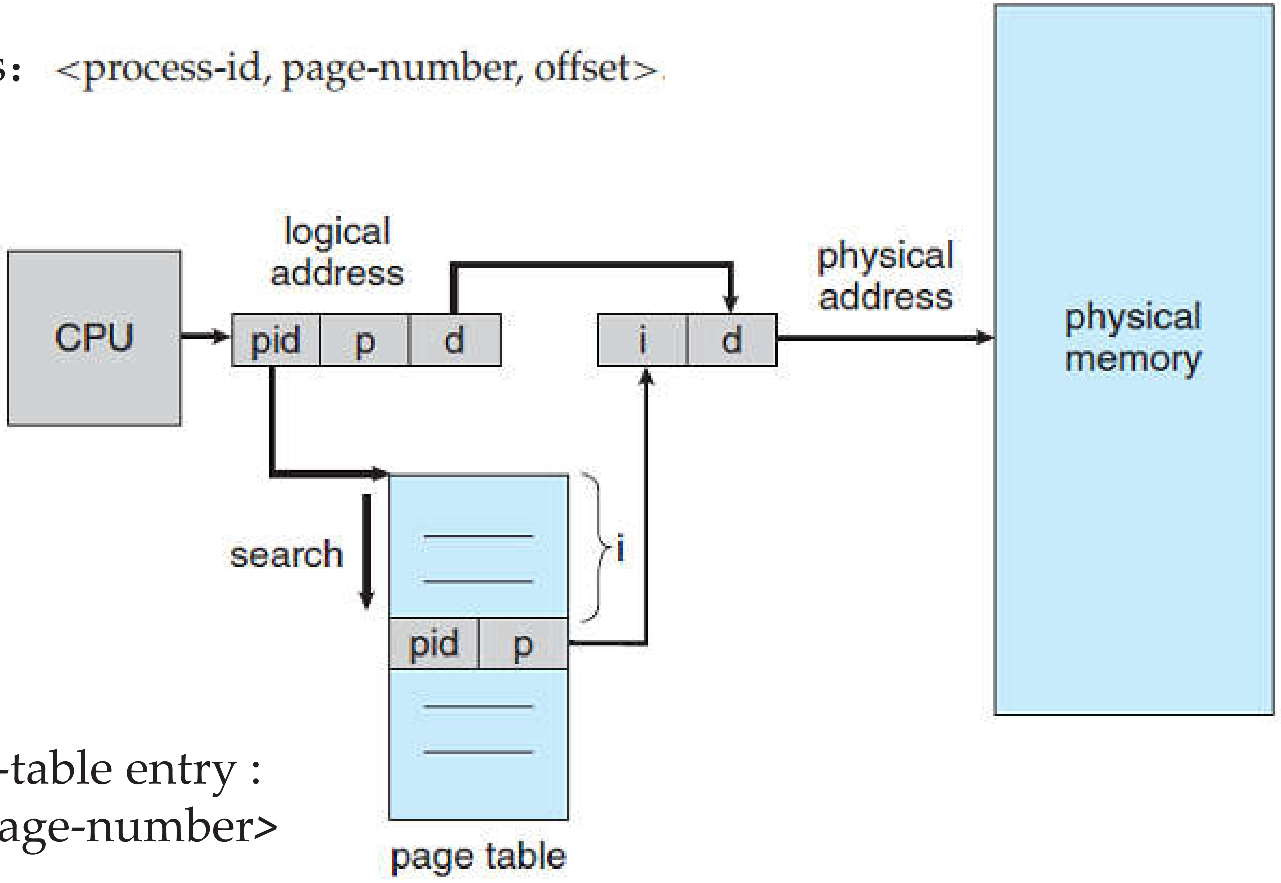
- Rather than keeping all possible logical page numbers, track all physical pages (frame) numbers

- One entry for each physical page of memory

- Entry consists of
  - Virtual address of the page stored in that frame
  - + Process info

Inverted page table scheme

virtual address: <process-id, page-number, offset>.



inverted page-table entry :
<process-id, page-number>

Problem and solutions

- Good:
  - Decrease memory needed
- Bad:
  - Increase time needed to search the table

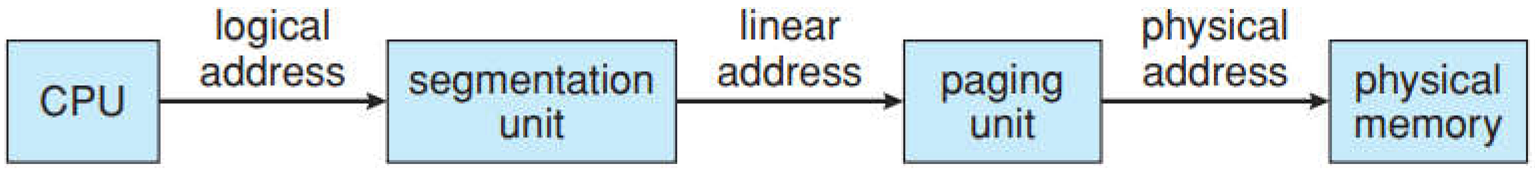
- Use hash table to limit the search to one, or at most a few, page-table entries.

- How to implement shared memory?
  - One mapping of a virtual address to the shared physical address

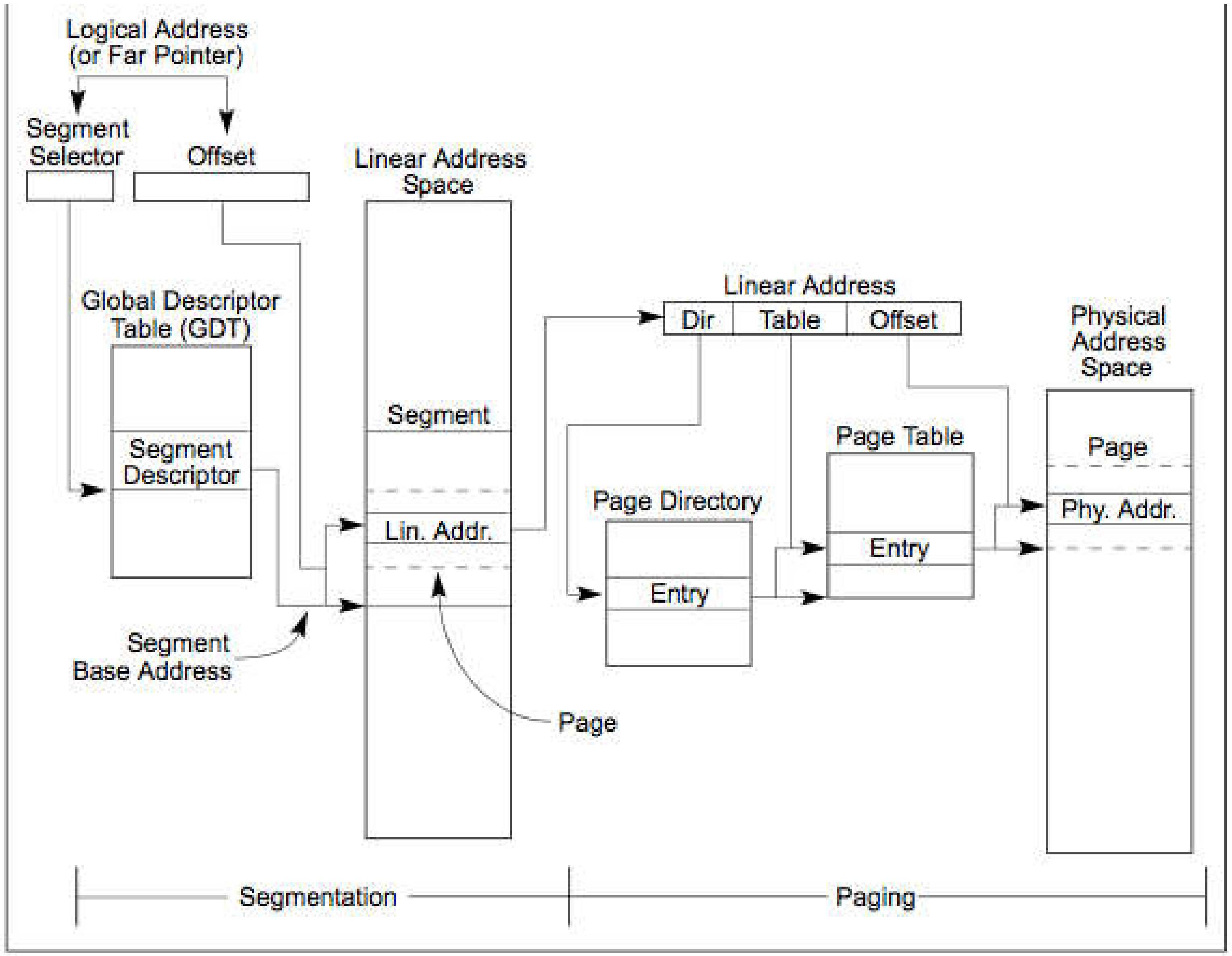
X86 architecture

Example: Intel 32 Architectures

- The IA-32 architecture supported both paging and segmentation
  - The CPU generates logical addresses, which are given to the segmentation unit.
  - The segmentation unit produces a linear address for each logical address.
  - The linear address is then given to the paging unit, which in turn generates the physical address in main memory.

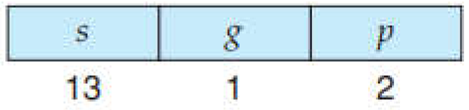


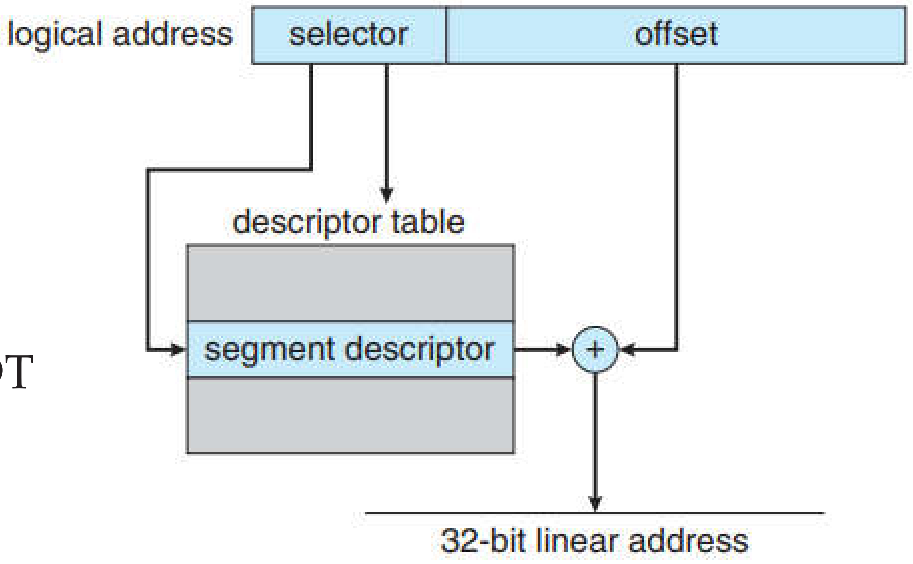
Making it real:
X86 Memory model with segmentation (16/32-bit)



IA-32 Segmentation

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

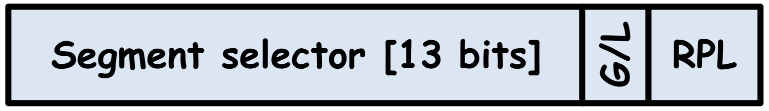




The segment register points to the appropriate entry in the LDT or GDT

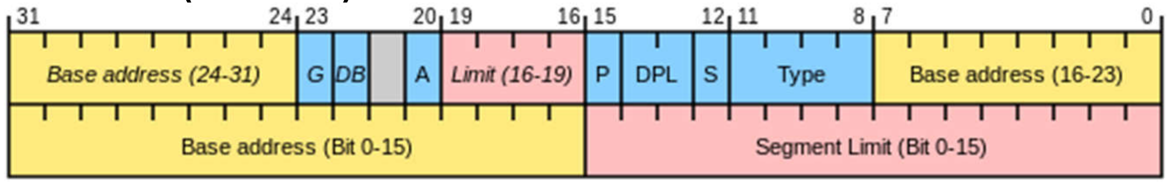
X86 Segment Descriptors (32-bit Protected Mode)

- 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)

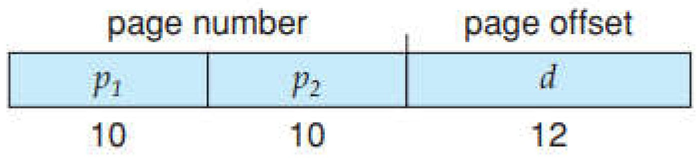
- Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
  - Includes length of table (for < 2^{13}) entries
- Descriptor format (64 bits):



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

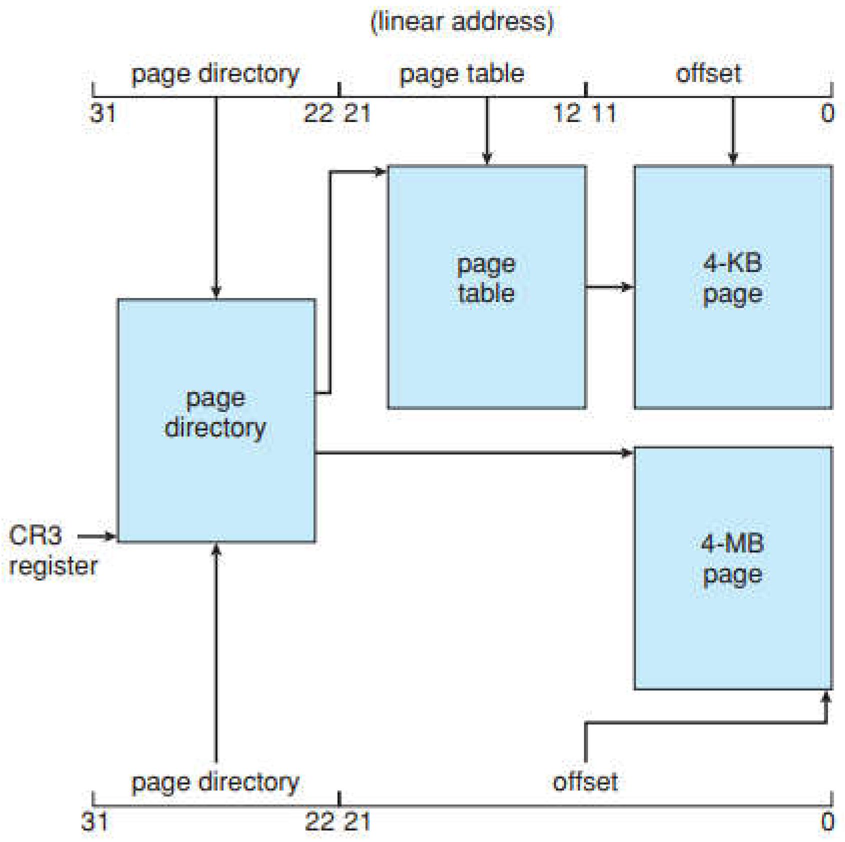
IA-32 Paging

- The IA-32 architecture allows a page size of either 4 KB or 4 MB



uses a two-level paging scheme

- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits
- the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table



Paging in the IA-32 architecture

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

Questions?