

Chapter 9: Main Memory





Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping





Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,





Background

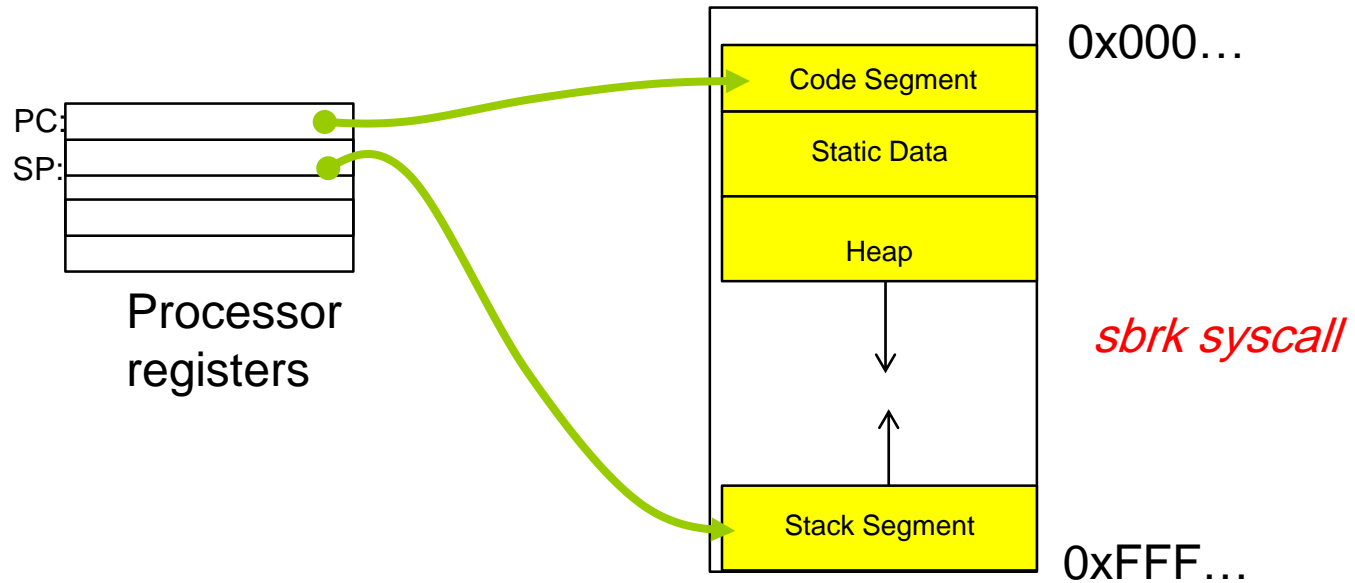
- Memory is central to the operation of a modern computer system.
- Memory consists of a large array of bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of PC.
- Instruction-execution cycle, for example
 - first fetches an instruction from memory
 - The instruction is then decoded
 - Operands fetched from memory
 - After the instruction has been executed on the operands, results may be stored back in memory





Recall: Address Space

Set of memory addresses accessible to program
(for read or write)

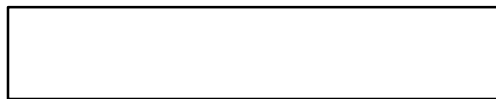




An Address

A memory address refers to the location of a byte in memory.

Most machines are byte-addressable



K bits



2^K
things





Bits & Addresses

If an address space has 32 bits,
how many unique addresses do I have?

$$2^{32} = (4294967296)$$

How many bits necessary to exclusively enumerate 4 elements?

$$2 \text{ bits} \Rightarrow 2^2 = 4. \Rightarrow \log_2(4)$$





Background

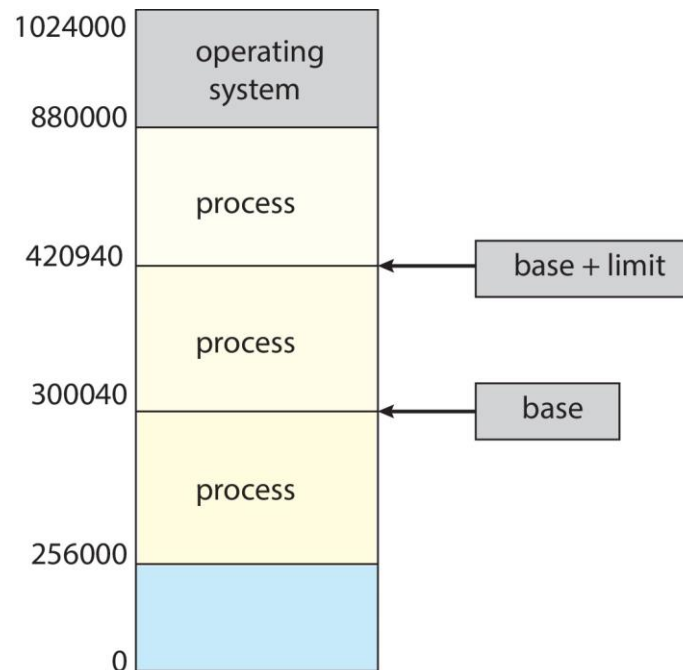
- Main memory and registers are only storage CPU can access directly
- There are machine instructions that take memory addresses as arguments, but none that take disk addresses
- Any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Memory unit only sees a stream of:
 - addresses + read requests, or
 - address + data and write requests
- 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
- Protection of memory required to ensure correct operation





Protection

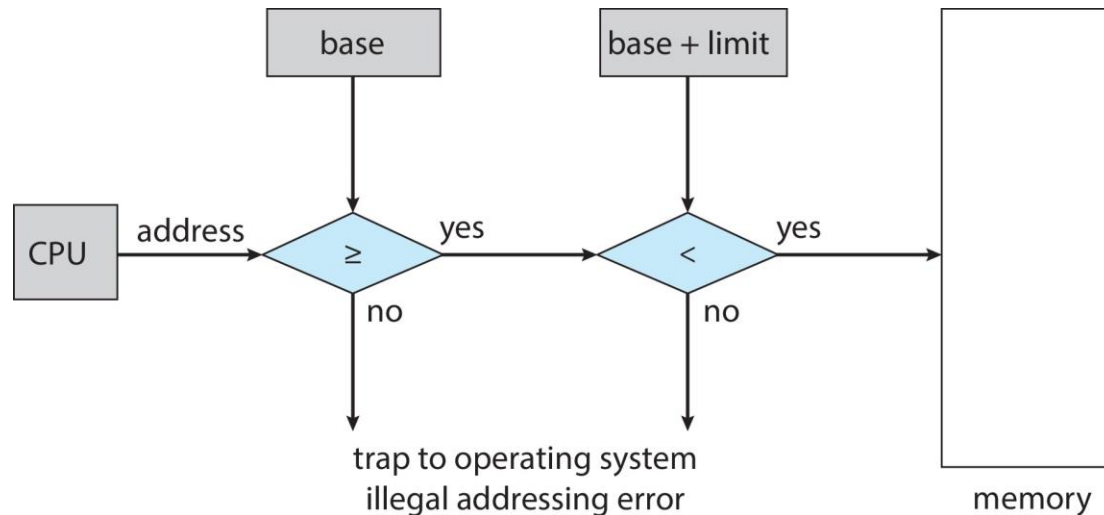
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process





Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- the instructions to loading the base and limit registers are privileged





Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- Addresses represented in different ways at different stages of a program's life
 - Source code addresses usually symbolic
 - Compiled code addresses **bind** to relocatable addresses
 - ▶ i.e., “14 bytes from beginning of this module”
 - Linker or loader will bind relocatable addresses to absolute addresses
 - ▶ i.e., 74014
 - Each binding maps one address space to another





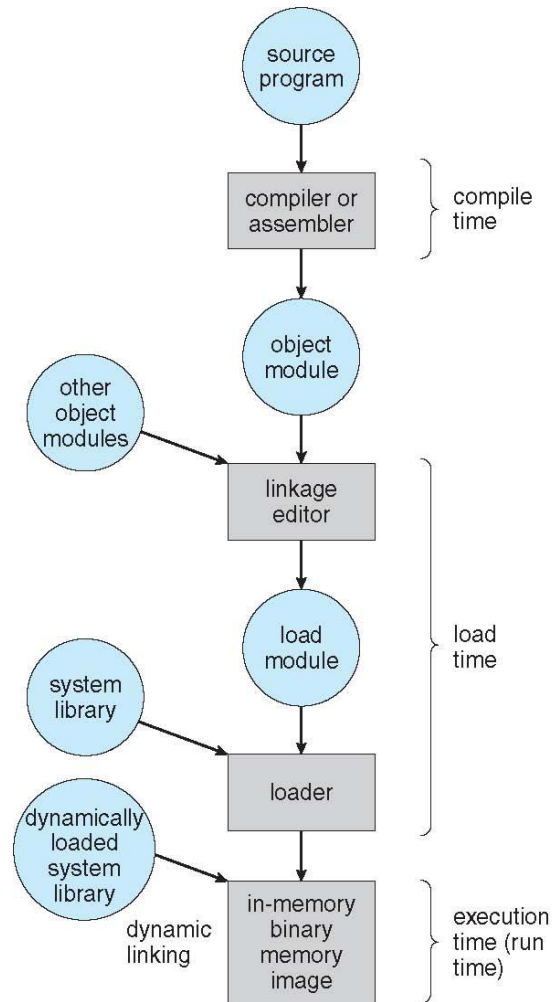
Binding of Instructions and Data to Memory

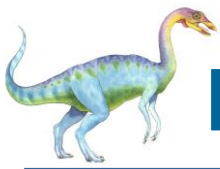
- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory 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 (e.g., base and limit registers)





Multistep Processing of a User Program





Logical vs. Physical Address Space

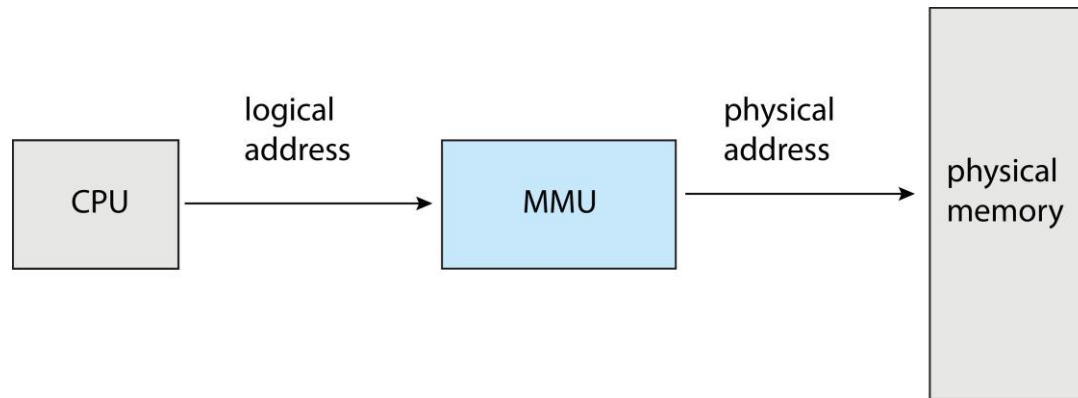
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program





Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this chapter





Memory-Management Unit (Cont.)

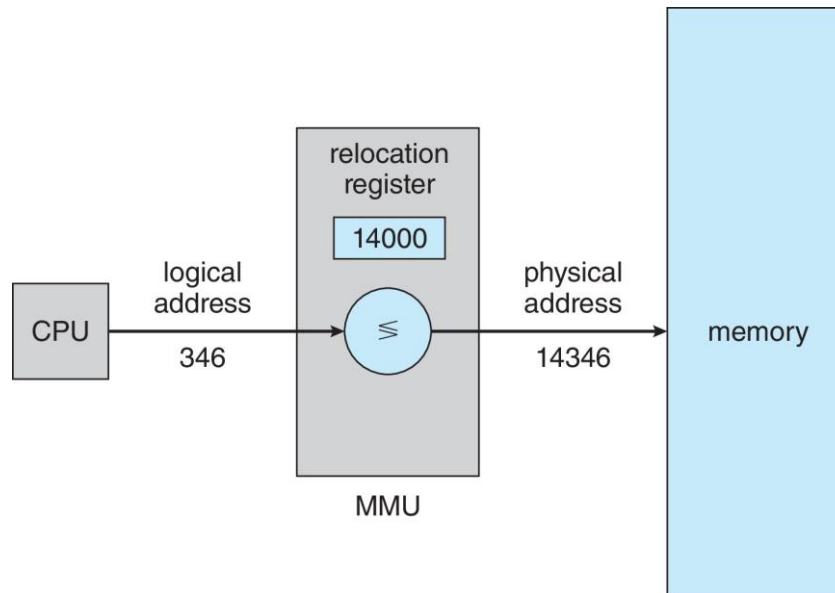
- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses





Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory





Address Translation

A logical address consists of two parts: a segment identifier (top bits) and an offset that specifies the relative address within the segment (bottom bits)





Address Translation

Assume we have 16 bit addresses

Question: if I have 4 segments (code, data, stack, heap), how many segment bits do I need?

$$\text{Log}(4) = 2$$

Segment 0: 00

Segment 1: 01

Segment 2: 10

Segment 3: 11





Address Translation

Assume we have 16 bit addresses

Question: if I have 4 segments (code, data, stack, heap), how many segment bits do I need?

$$\text{Log}(4) = 2$$

Question: what is the maximum size of each segment?

$$16 - 2 = 14 \text{ bits left. } \Rightarrow 2^{14} \text{ bytes}$$

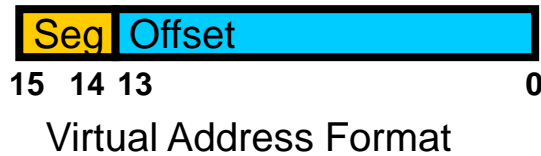
Question: if I have 7 segments and an address size of 32 bits, what is the maximum size of a segment?

$$\text{Log}_2(7) = 2.8 \Rightarrow 3 \text{ bits. } 2^{(32-3)} = 2^{29}$$

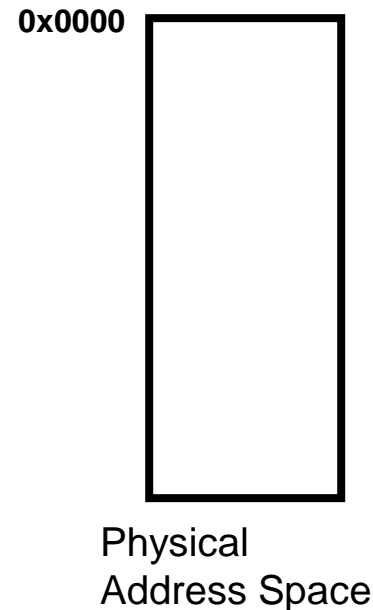
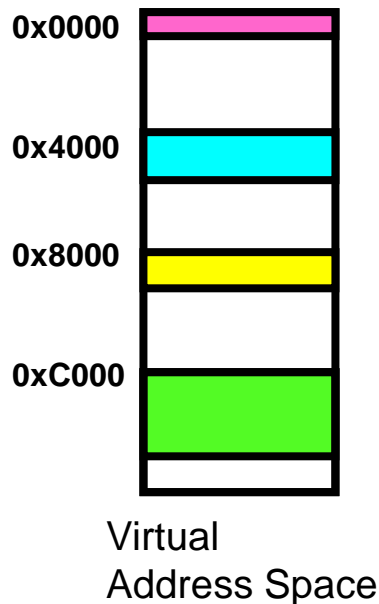




Example: Four Segments (16 bit addresses)



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



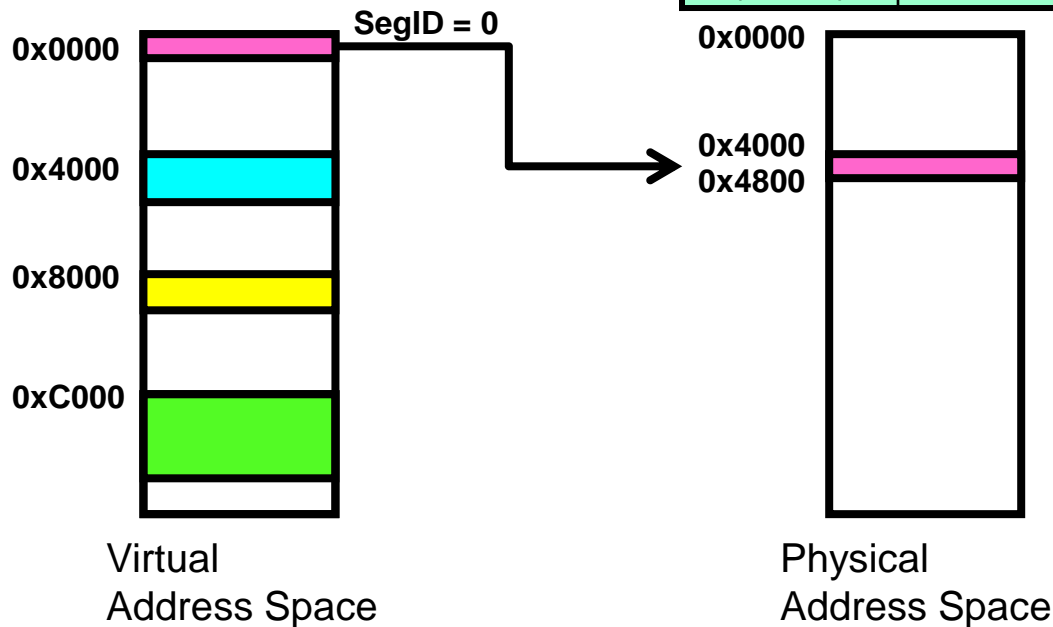


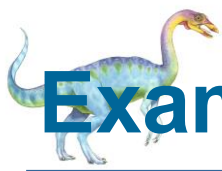
Example: Four Segments (16 bit addresses)



Virtual Address Format

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



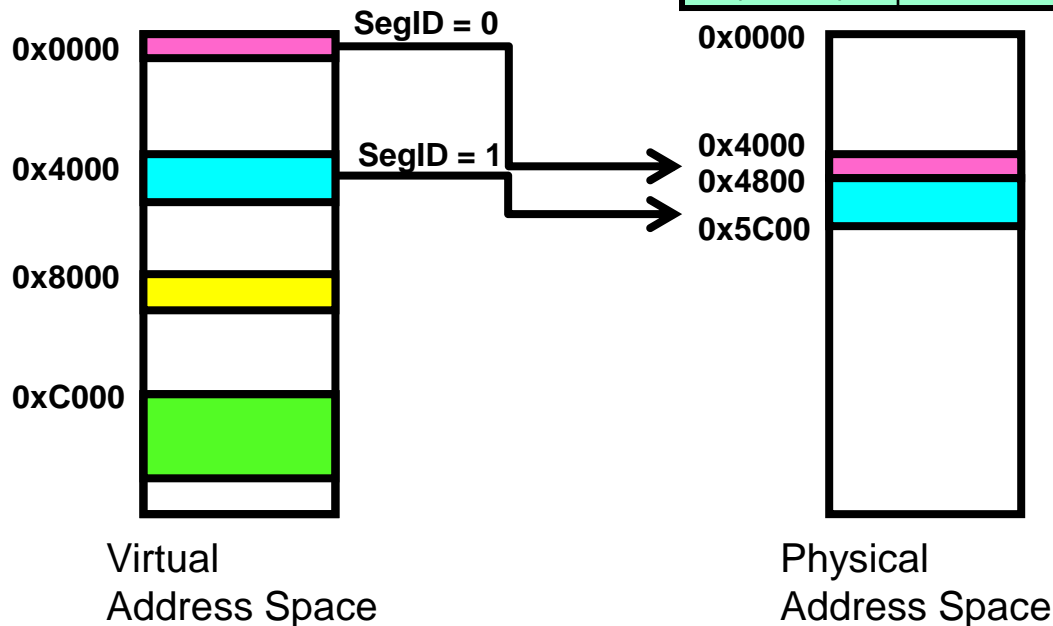


Example: Four Segments (16 bit addresses)



Virtual Address Format

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



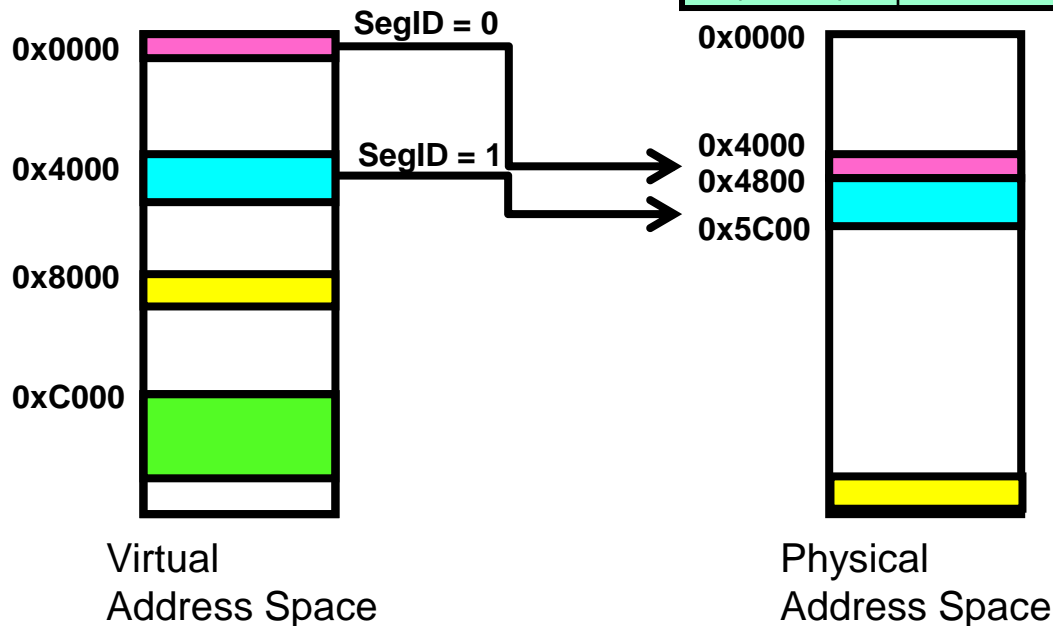


Example: Four Segments (16 bit addresses)



Virtual Address Format

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





Dynamic Loading

- The entire program does need to be in memory to execute
- The size of a process has thus been limited to the size of physical memory
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading





Dynamic Linking

- **Dynamically linked libraries (DLLs)** are system libraries that are linked to user programs when the programs are run
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
 - Not only increases the size of an executable image but also may waste main memory
 - DLLs are libraries can be shared among multiple processes, so that only one instance of the DLL in main memory.





Dynamic Linking

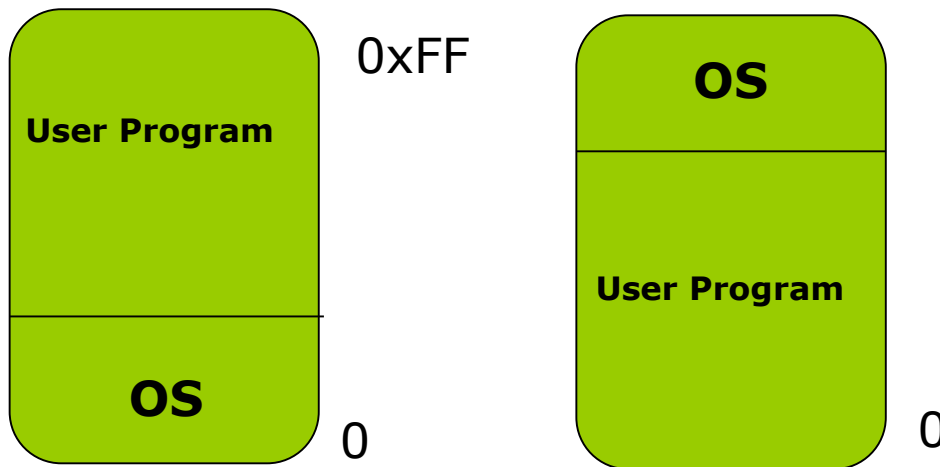
- Operating system checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
 - Versioning may be needed





Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
 - Operating system
 - User Process
- Linux and Windows place the operating system in high memory





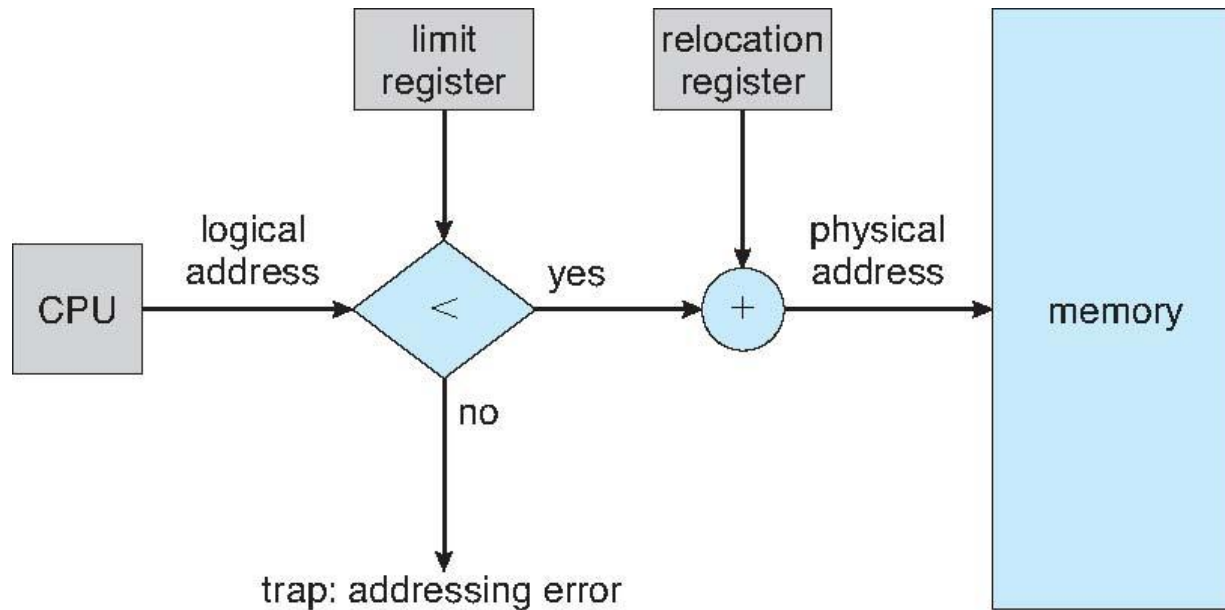
Contiguous Allocation (Cont.)

- We usually want several user processes to reside in memory at the same time.
- In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process
- 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
 - MMU maps logical address *dynamically*





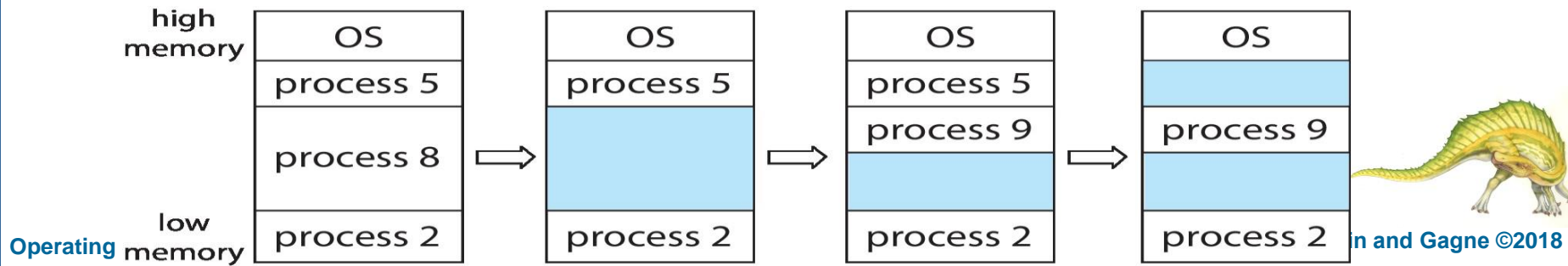
Hardware Support for Relocation and Limit Registers

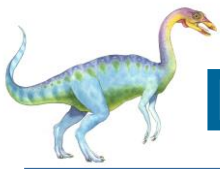




Variable Partition

- One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory
- each partition may contain exactly one process
- Operating system keeps a table indicating which parts of memory are available and which are occupied
- **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
- Process exiting frees its partition, adjacent free partitions combined
- 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

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used (break the physical memory into fixed-sized blocks and allocate memory in units based on block size)





Fragmentation (Cont.)

- Reduce external fragmentation by **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





Paging

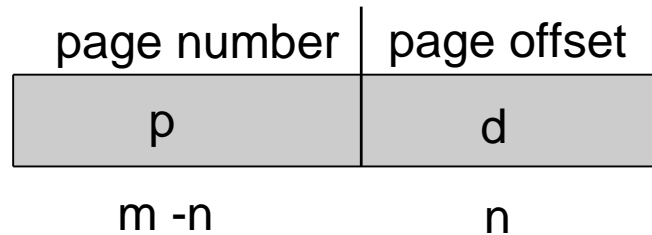
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size **N** pages, need to find **N** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Still have Internal fragmentation





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

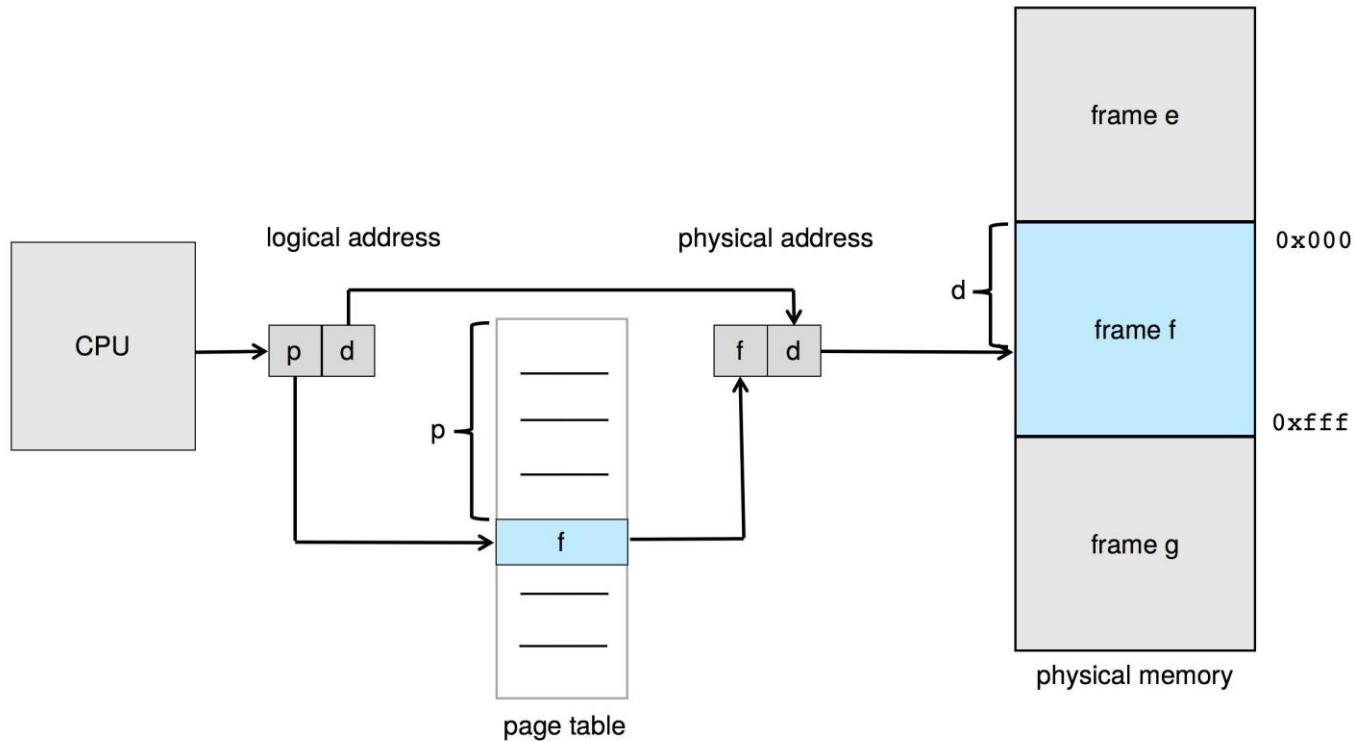


- For given logical address space 2^m and page size 2^n



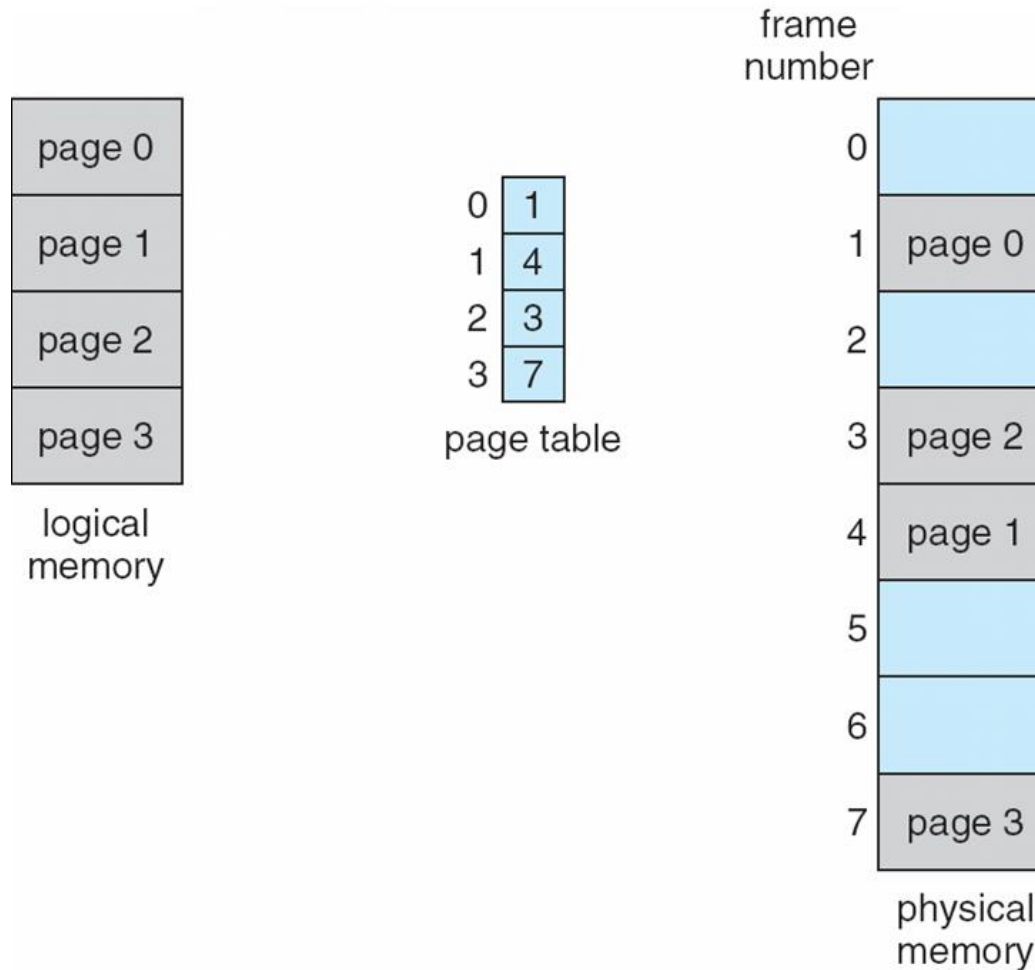


Paging Hardware





Paging Model of Logical and Physical Memory





Example: A Mini Page Table

Assume we have a 64 bytes (2^6) of physical memory

Assume we want pages of 4 bytes (2^2)

How long should our addresses be?

6 bits

How many offset bits should we assign?

2 bits

How many virtual pages can we have?

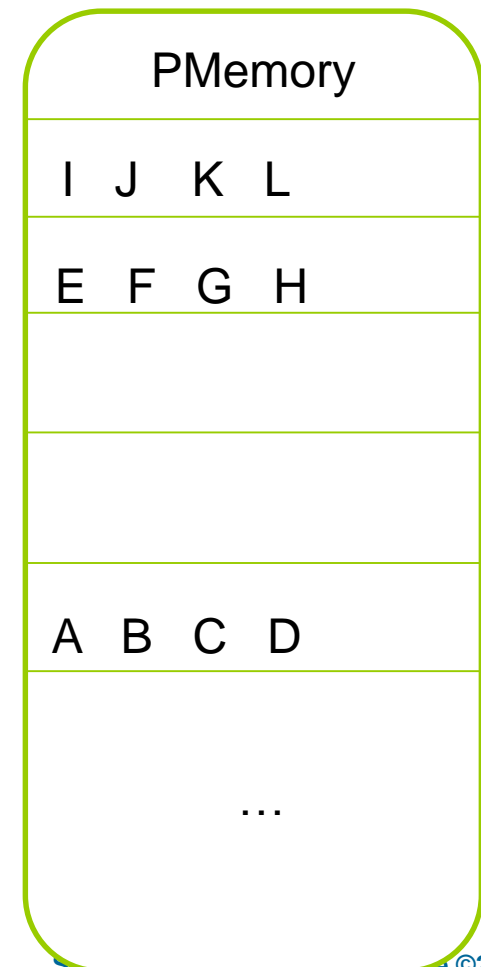
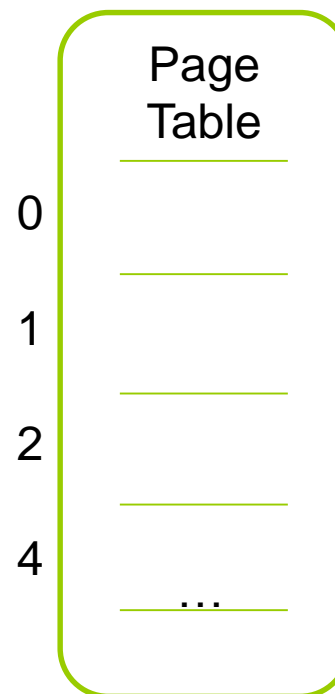
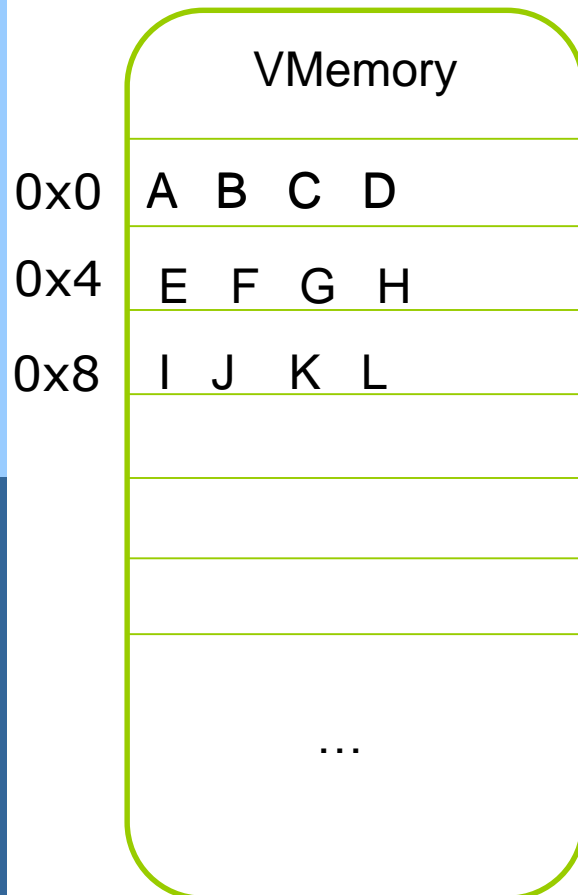
6 bit addresses, 2 bit for offsets, 4 bits for VPN.

$2^4 = 16$ pages



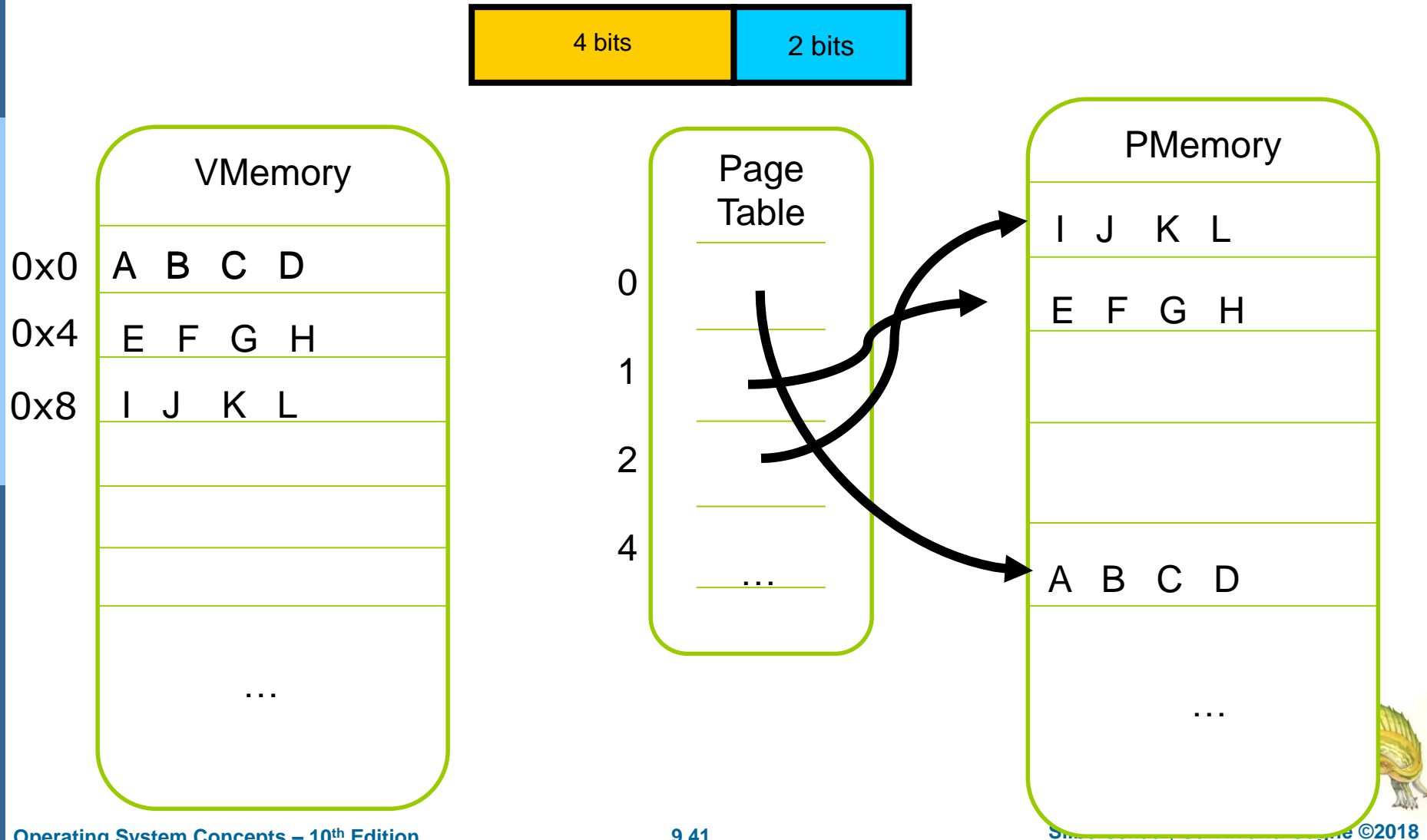


Example: A Mini Page Table





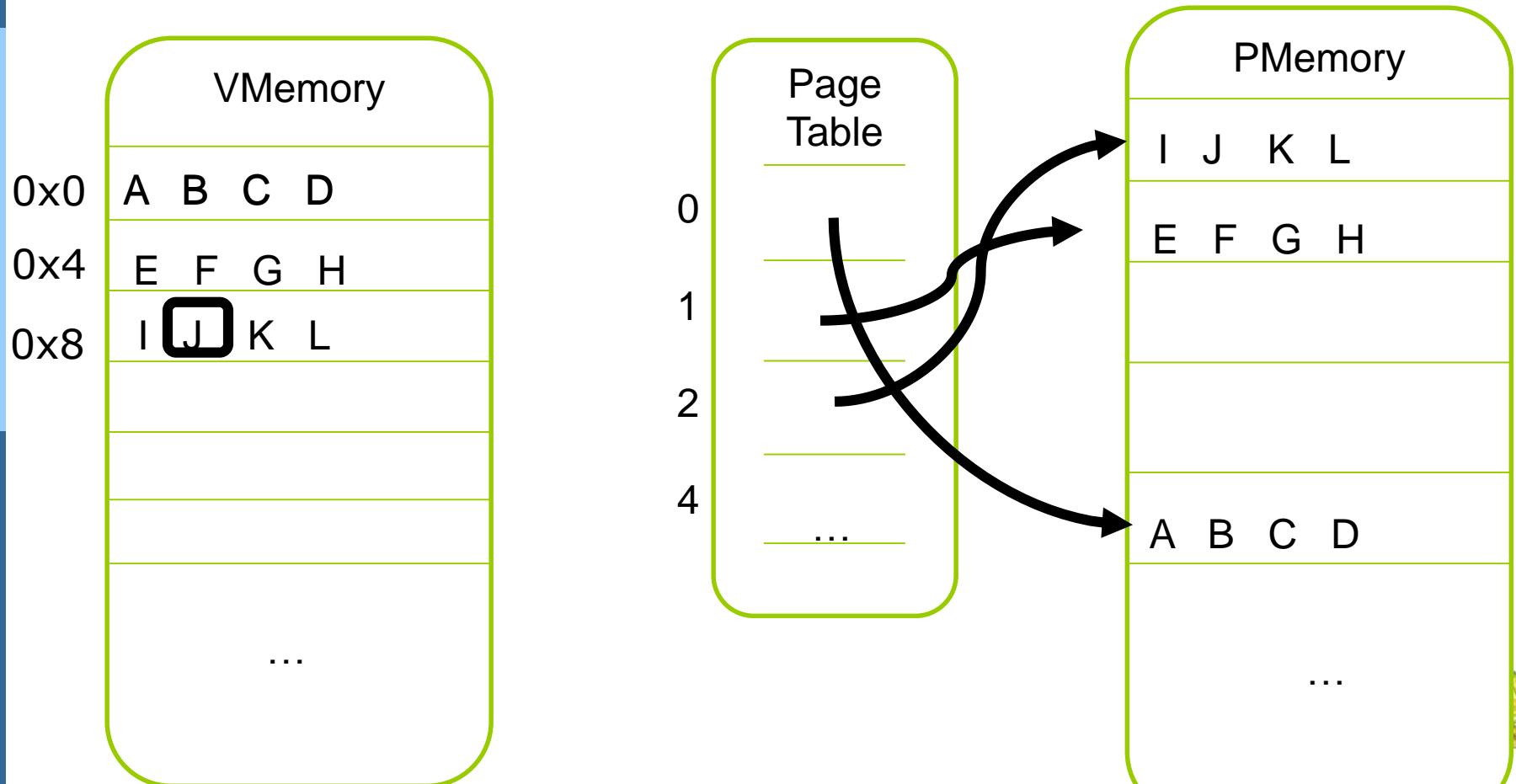
Example: A Mini Page Table





Example: A Mini Page Table

0x9 =



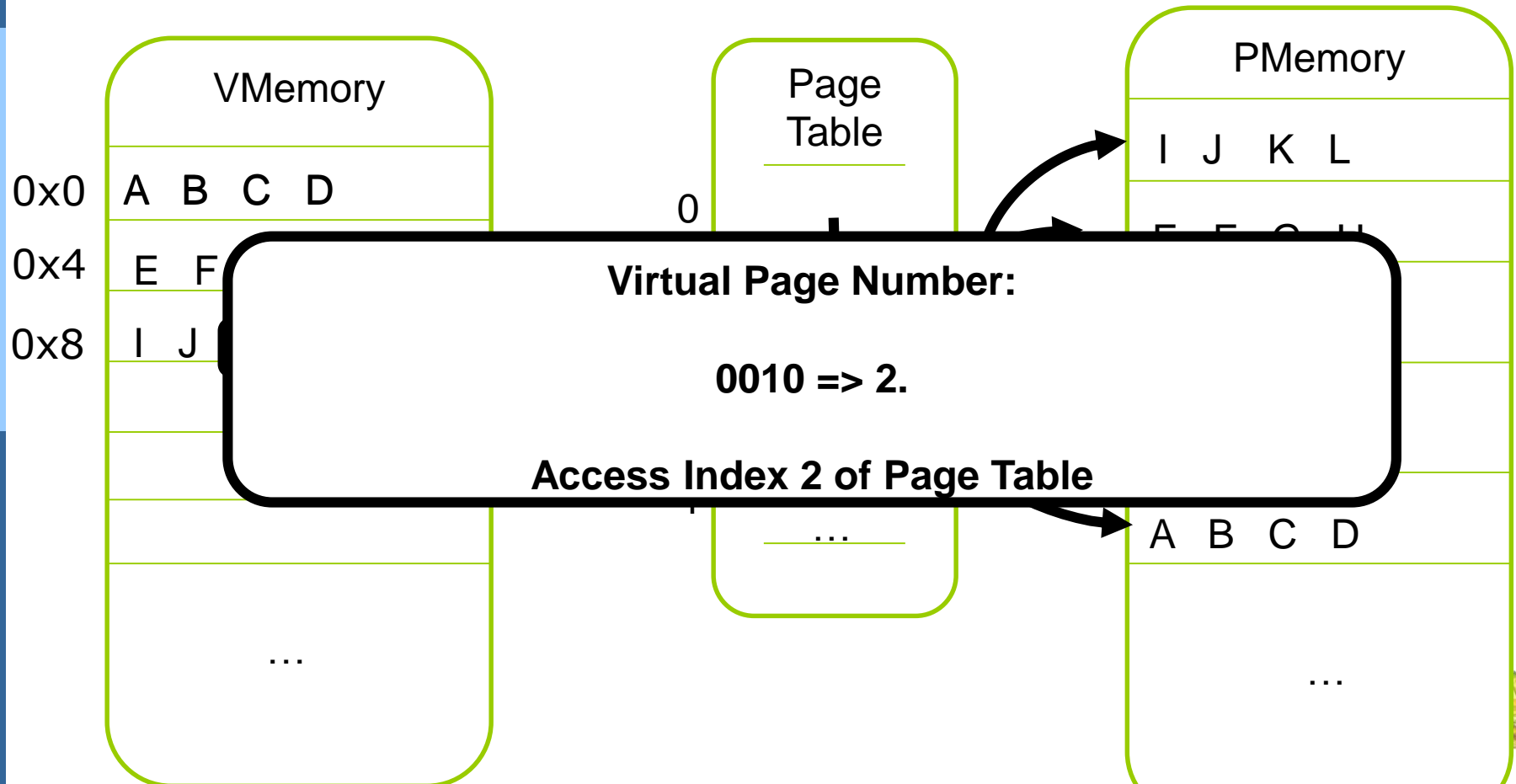


Step 1: Extract Virtual Page Number

0x9 =

0010

01



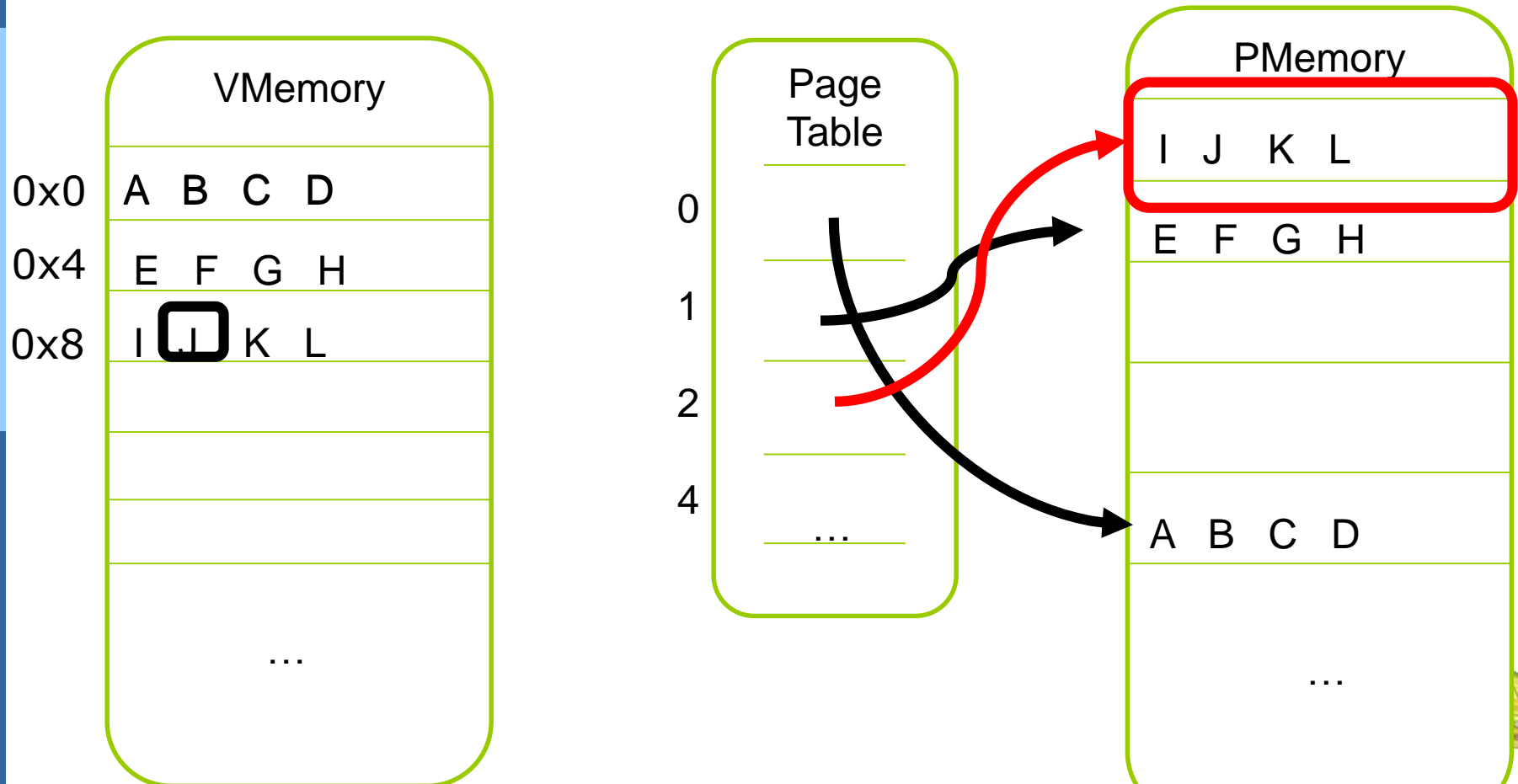


Step 2: Identify Physical Page Number

0x9 =

0010

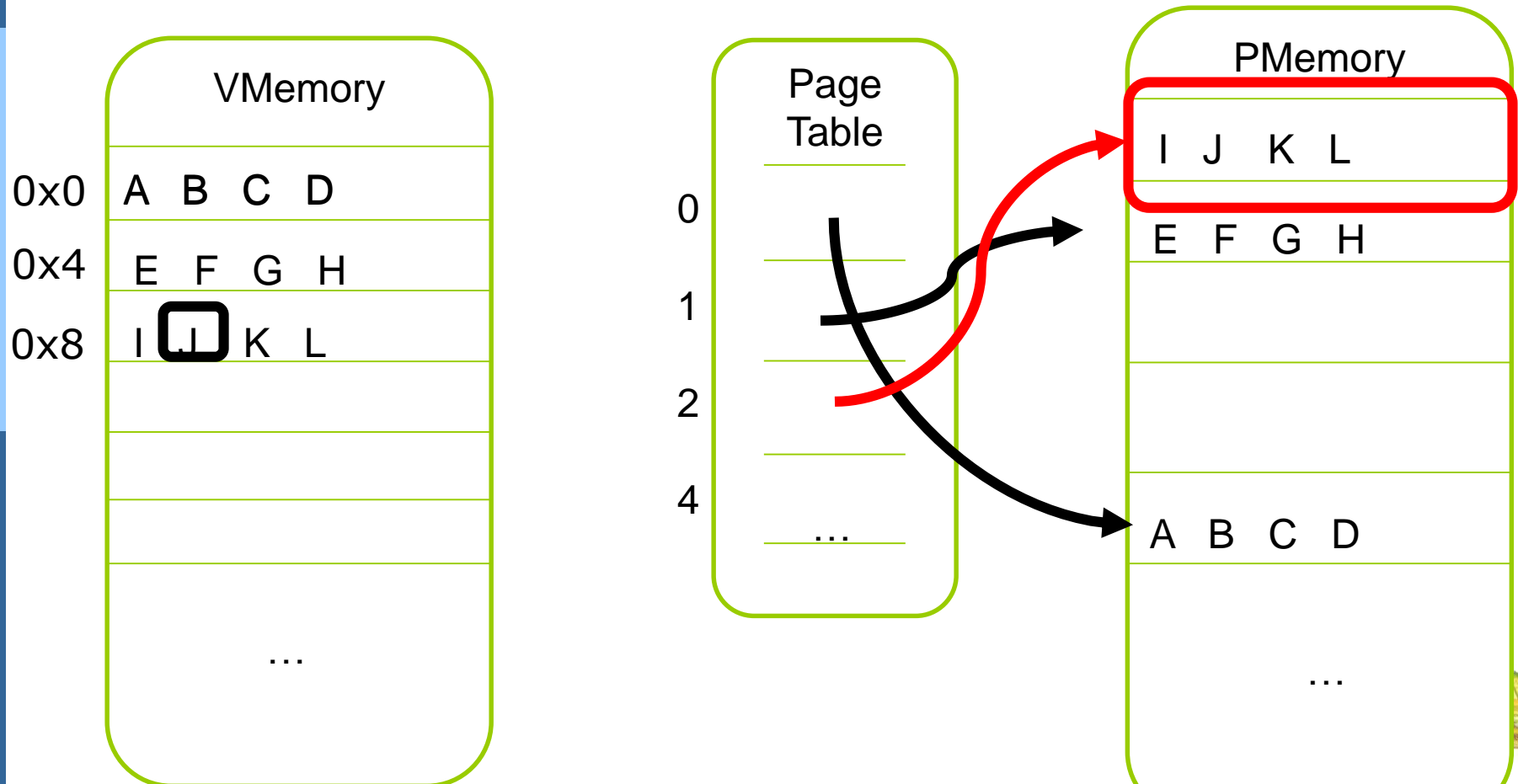
01





Step 3: Extract Frame Offset

0x9 =





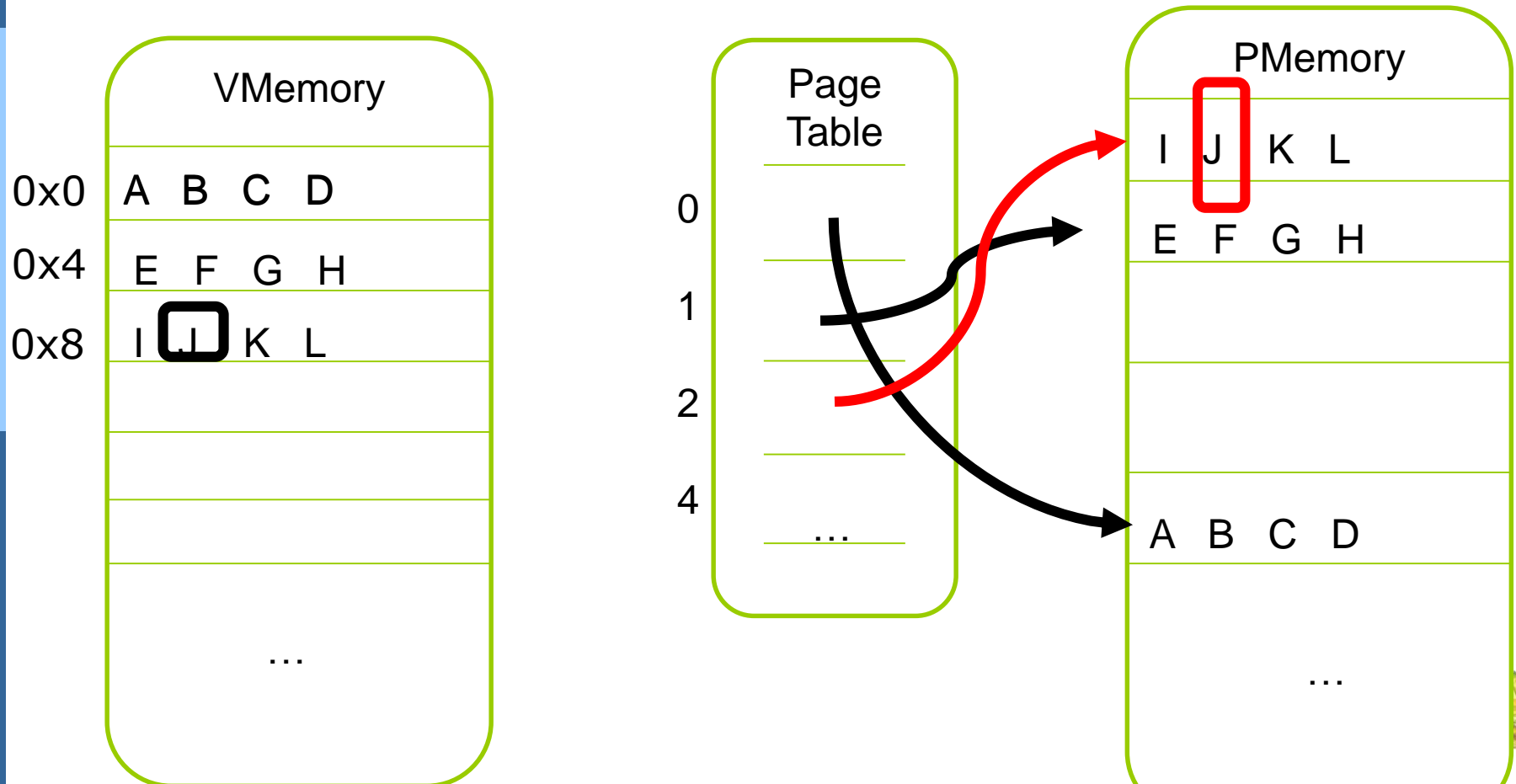
01





Step 3: Extract Frame Offset

0x9 =



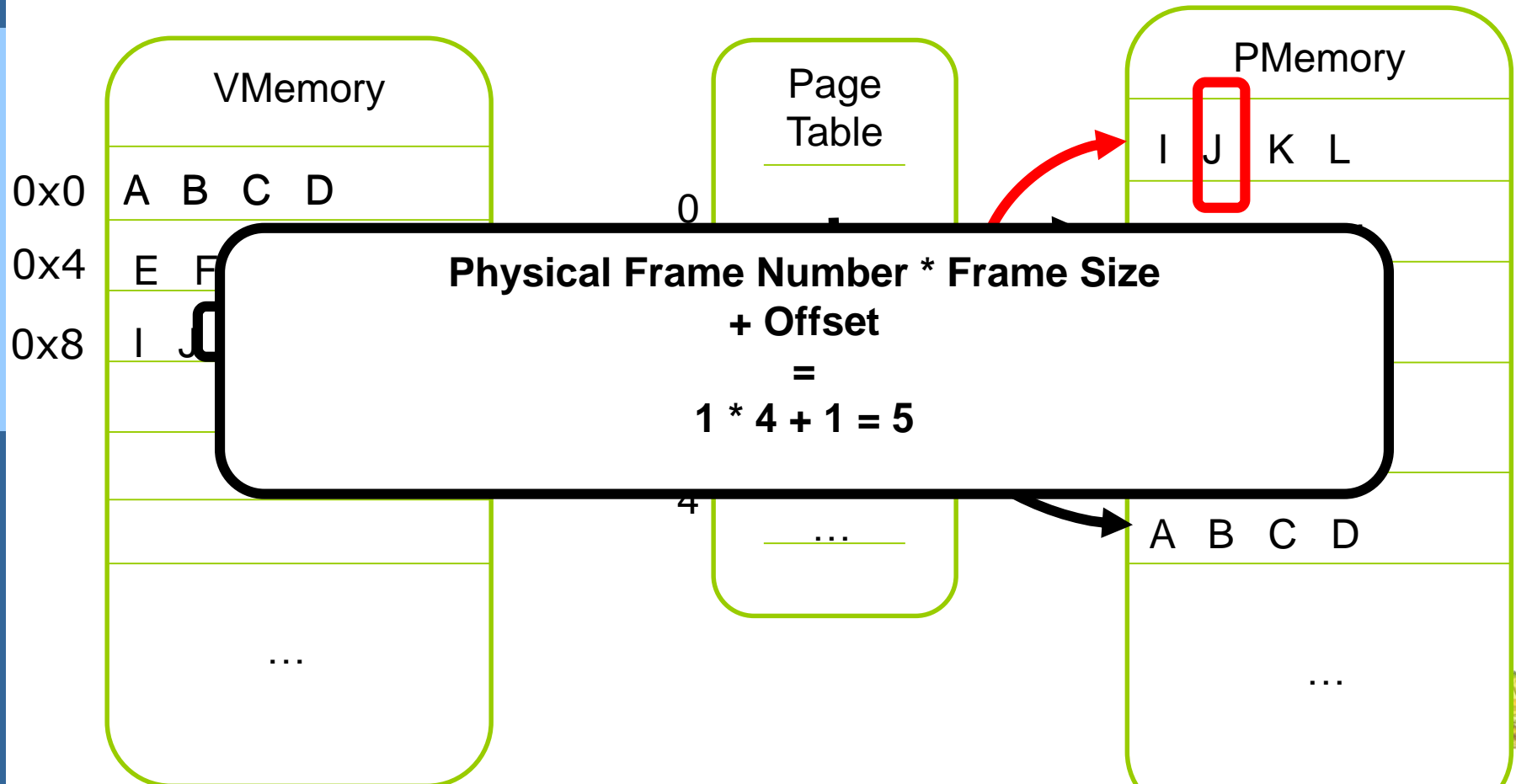


Step 4: Convert to Physical Address

0x9 =

0010

01





Paging Example

- Logical address: $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)
- Logical address 0 is page 0, offset 0
page 0 is in frame 5. logical address 0 maps to physical address
 - $20 = [(5 \times 4) + 0]$
- Logical address 4 is page 1, offset 0
 - $24 = [(6 \times 4) + 0]$

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory



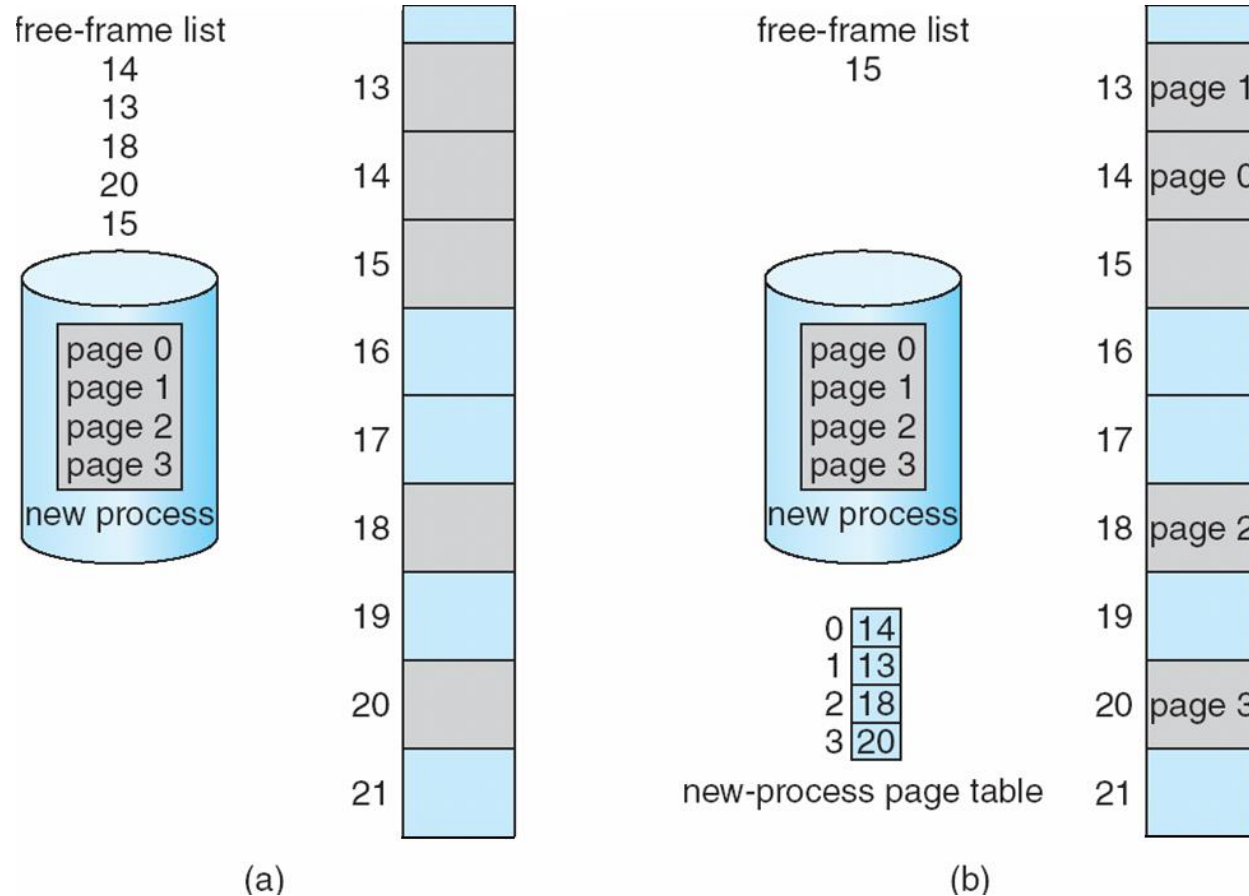
Paging -- Calculating internal fragmentation

- 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 – 1 byte
- On average fragmentation = $1 / 2$ frame size





Free Frames



Before allocation

After allocation





Implementation of Page Table

- page table is implemented as a set of dedicated high-speed hardware registers
- Use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries)
- contemporary CPUs, however, support much larger page tables (for example, 2^{20} entries)
- Page table is kept in main 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 and one for the data / instruction
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs)** (also called **associative memory**).





Translation Look-Aside Buffer

- Each entry in the TLB consists of two parts: a key(or tag) and a value
- The associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned
- TLBs typically small (64 to 1,024 entries)
- The TLB is used with page tables
- A logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
 - If the page number is found, its frame number is immediately available and is used to access memory.
 - On a TLB miss, value is loaded into the TLB for faster access next time
 - ▶ Replacement policies must be considered (If the TLB is already full of entries)
 - ▶ Some entries can be **wired down** for permanent fast access (they cannot be removed from the TLB. (kernel code))





Hardware

- Associative memory – parallel search

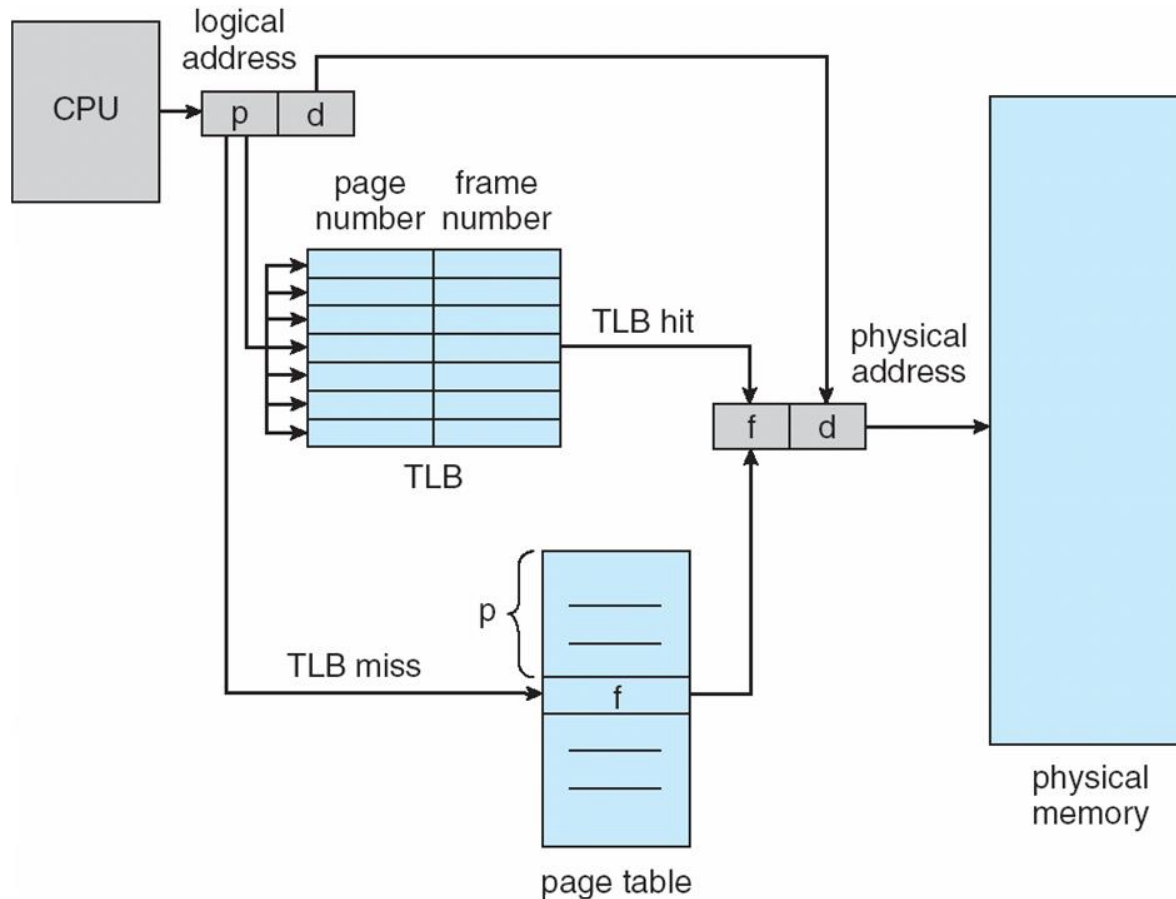
Page #	Frame #

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





Paging Hardware With TLB





Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If we find the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time (EAT)**
$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,
$$\text{EAT} = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$

implying only 1% slowdown in access time.





Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
 - Or use **page-table length register (PTLR)** (indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process)
- Any violations result in a trap to the kernel





Valid (v) or Invalid (i) Bit In A Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	

frame number		valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page n





Shared Pages

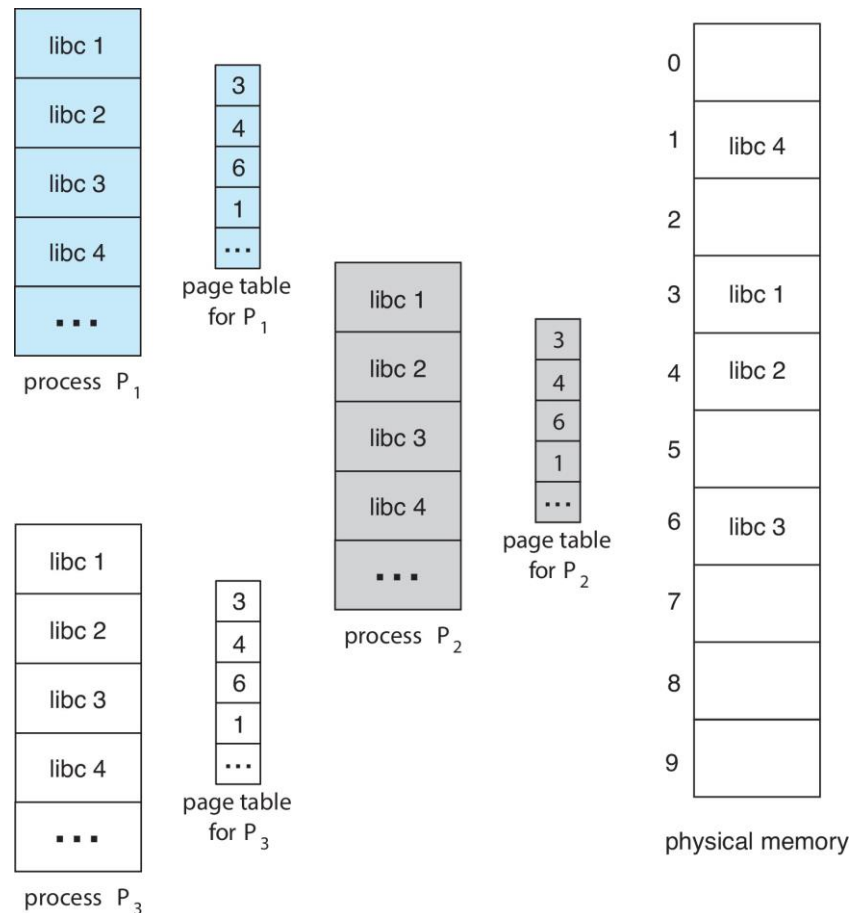
- Most user processes require the standard C library libc
- Each process load its own copy of libc into its address space
- If a system has 40 user processes, and the libc library is 2 MB, this would require 80 MB of memory.
- **Shared code**
 - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
 - Similar to multiple threads sharing the same process space
 - Also useful for interprocess communication if sharing of read-write pages is allowed





Shared Pages Example

- we see three processes sharing the pages for the standard C library libc





Structure of the Page Table

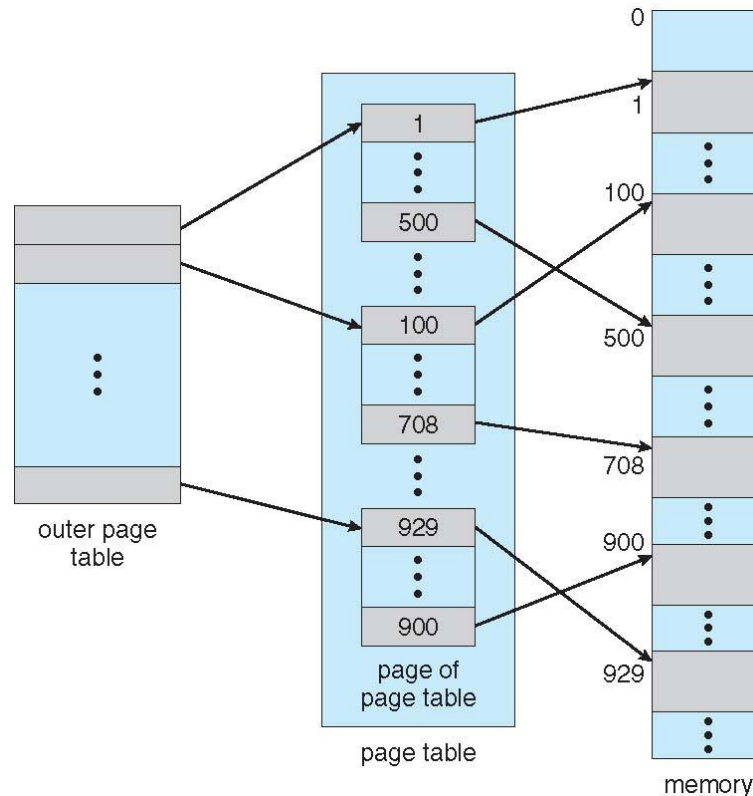
- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
 - ▶ Don't want to allocate that contiguously in main memory
 - One simple solution is to divide the page table into smaller units
 - ▶ Hierarchical Paging
 - ▶ Hashed Page Tables
 - ▶ Inverted Page Tables





Hierarchical Page Tables

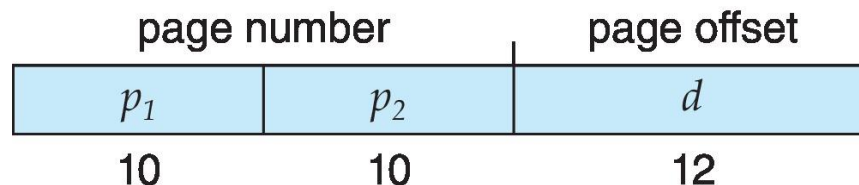
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- 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:

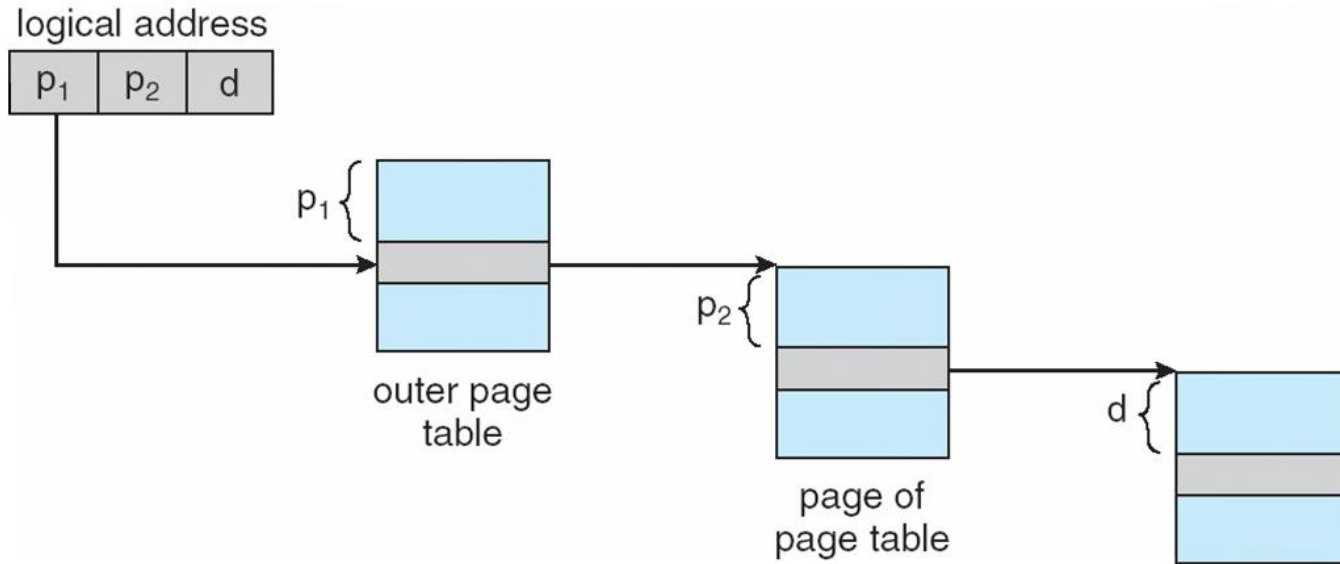


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





Address-Translation Scheme





64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like

outer page	inner page	offset
p_1	p_2	d
42	10	12

- Outer page table has 2^{42} entries or 2^{44} bytes
- 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 possibly 4 memory access to get to one physical memory location





Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12





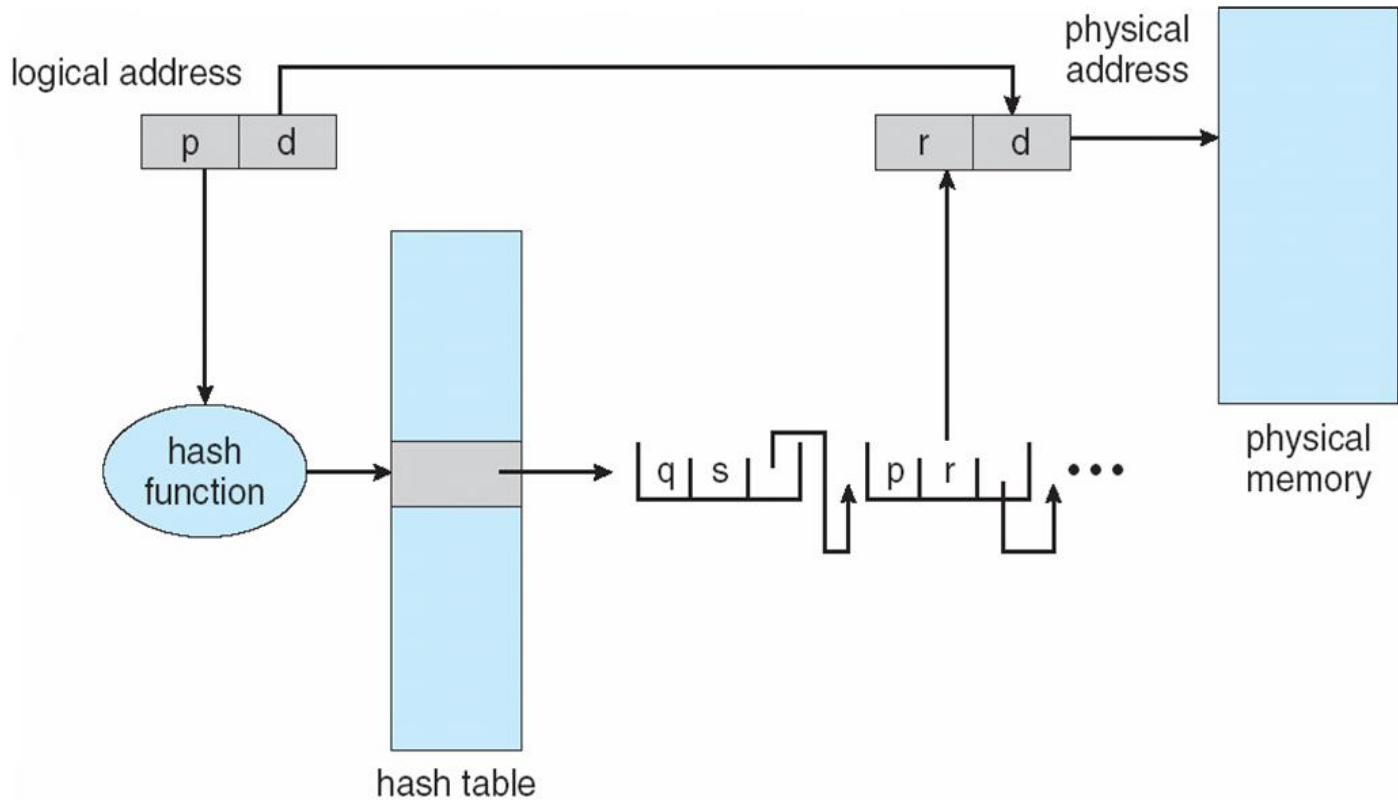
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted





Hashed Page Table





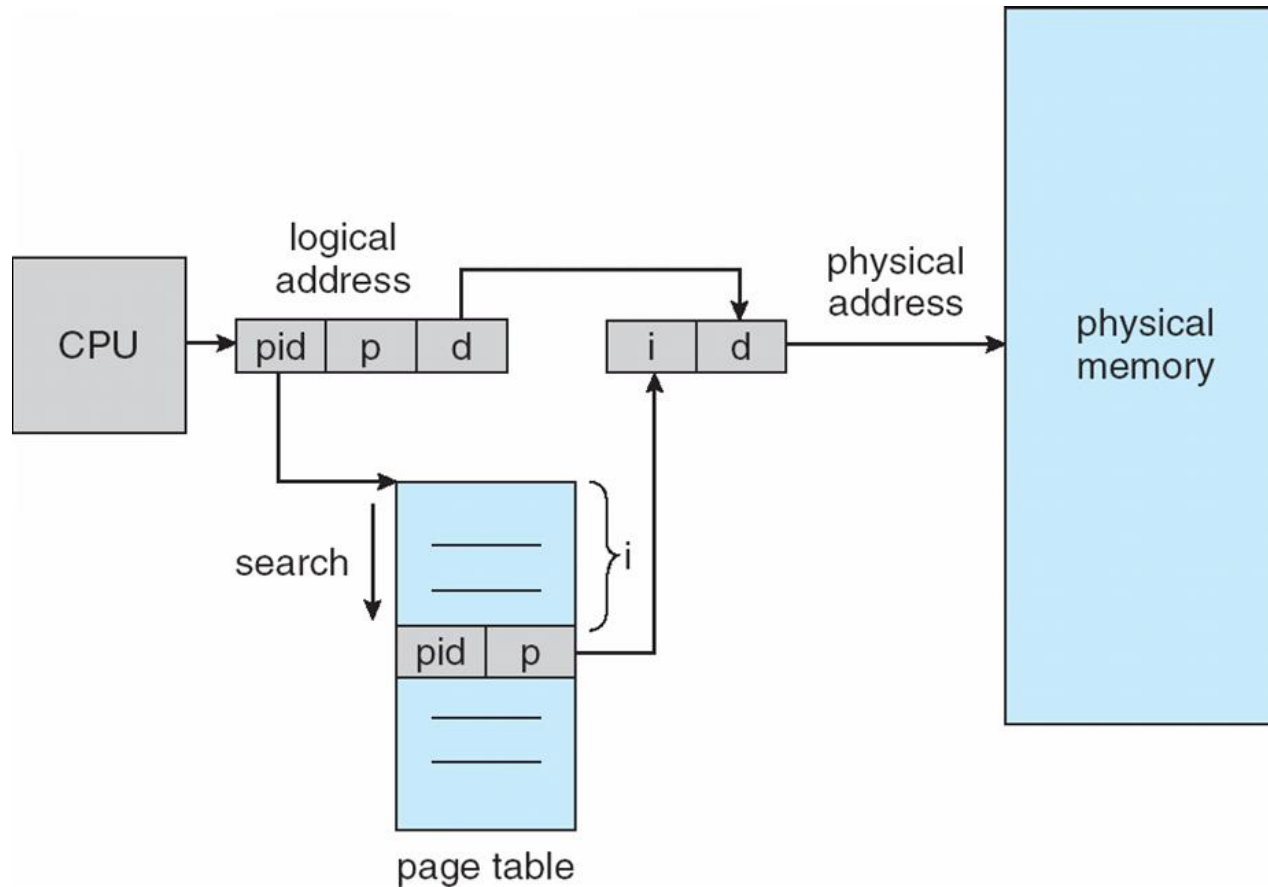
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry 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 store each page table, but increases time needed to search the table when a page reference occurs





Inverted Page Table Architecture





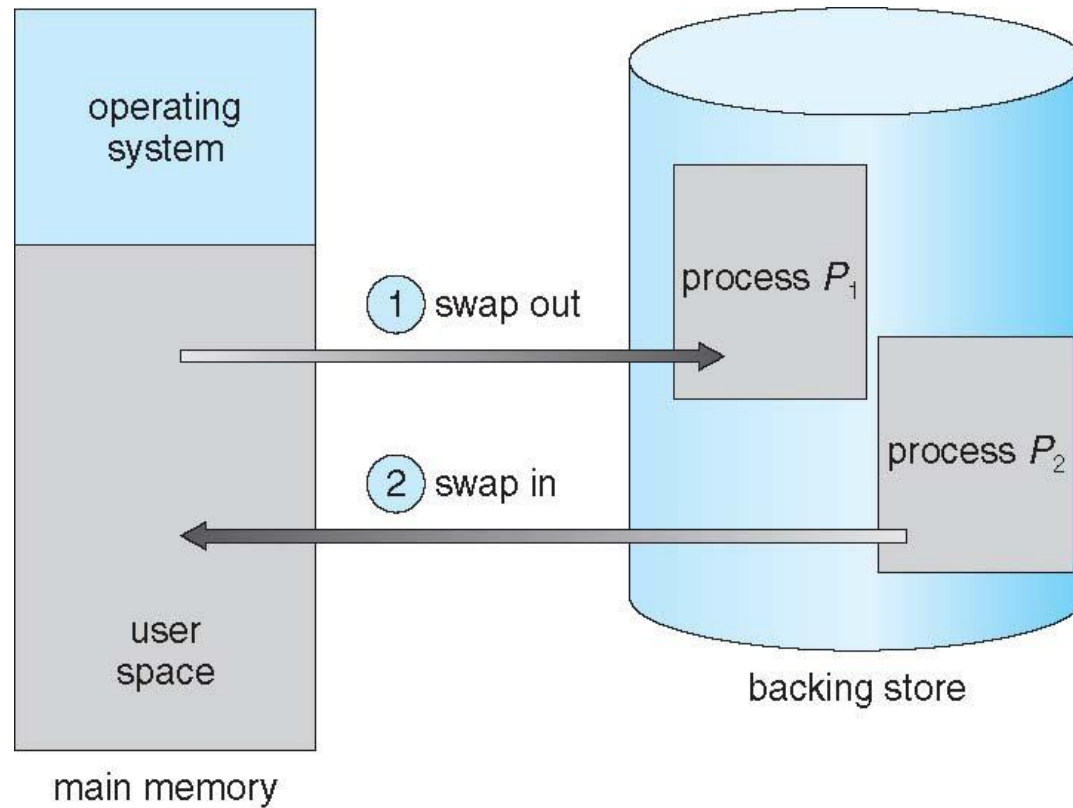
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
 - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped





Schematic View of Swapping





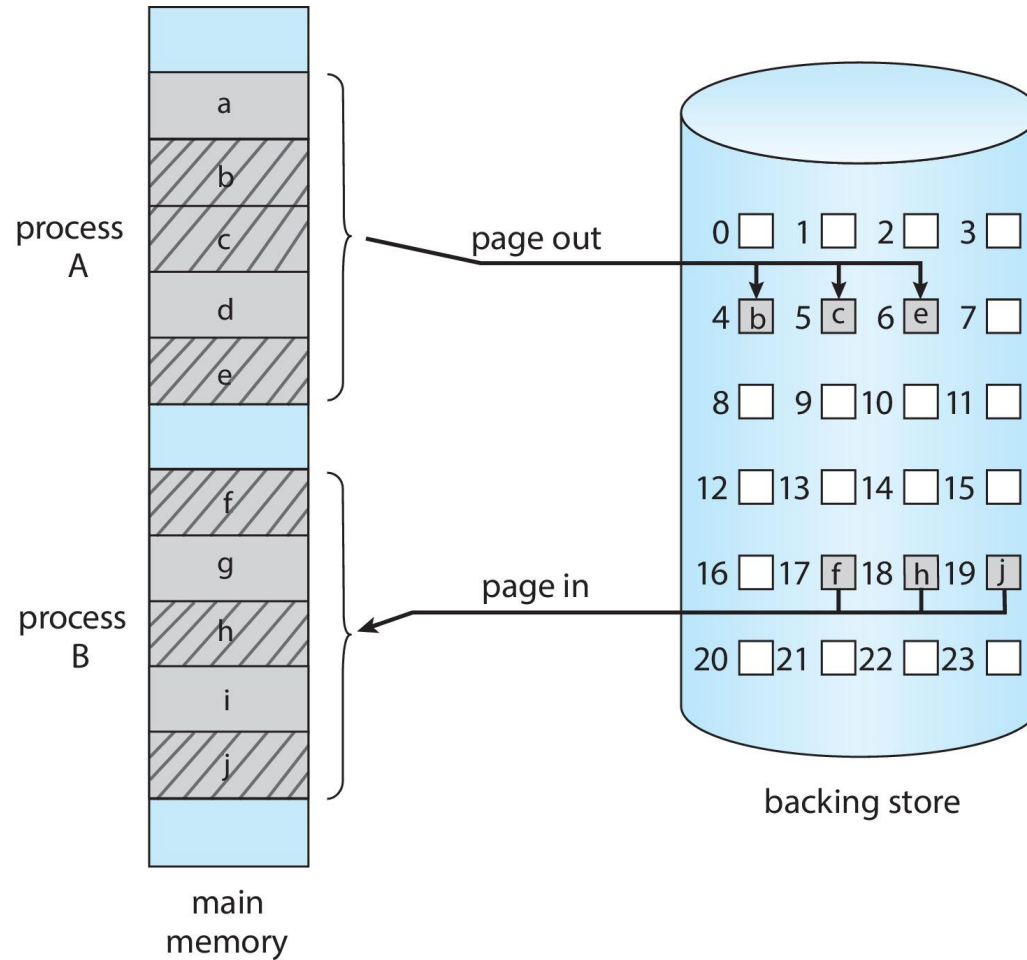
Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - Swap out time of 2000 ms
 - Plus swap in of same sized process
 - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
 - System calls to inform OS of memory use via `request_memory()` and `release_memory()`





Swapping with Paging



End of Chapter 9

