Main Memory

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu).

Review

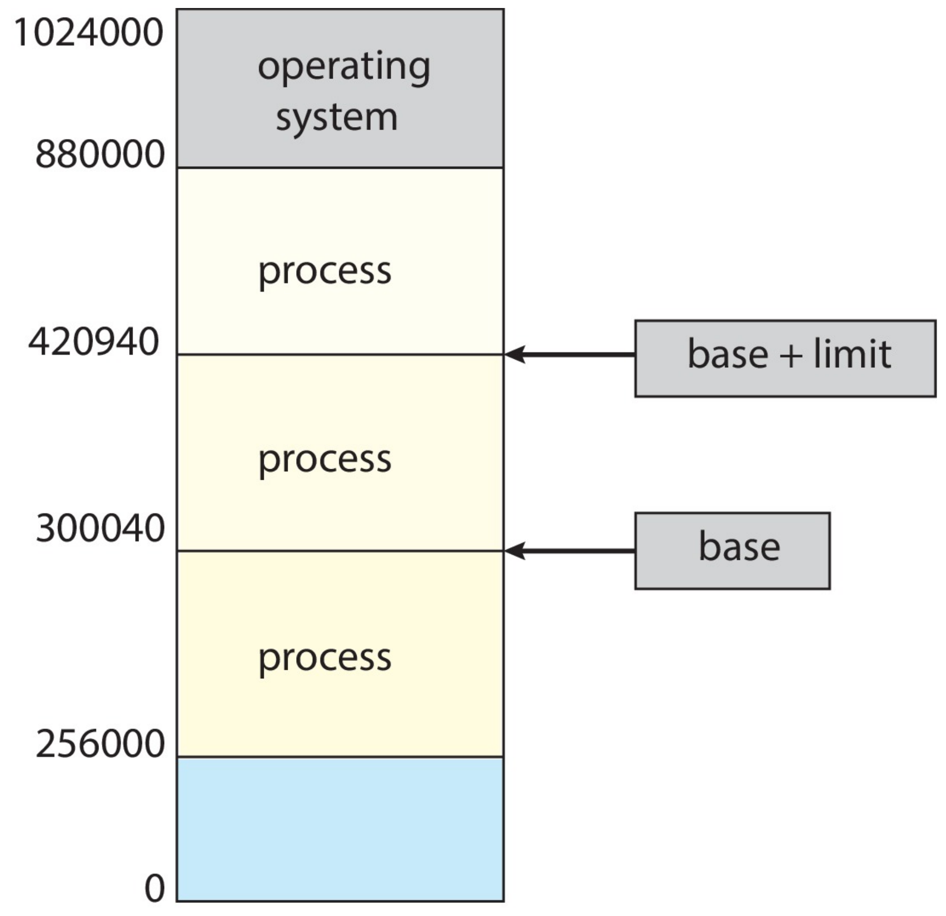
- Deadlock problem: Four conditions for deadlock
- System model
- Resource allocation graph
    - No cycle -> no deadlock, cycle -> possible deadlock
- Handling deadlocks
    - deadlock prevention
        - Break four conditions, ordering of resources
    - deadlock avoidance
        - Extra information, safe state
        - Single instance: resource allocation graph -> claim edge
        - Multiple instances: banker’s algorithm
- deadlock detection
    - Wait-for graph, similar with banker’s algorithm

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- 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
  - hardware vs software

Protection

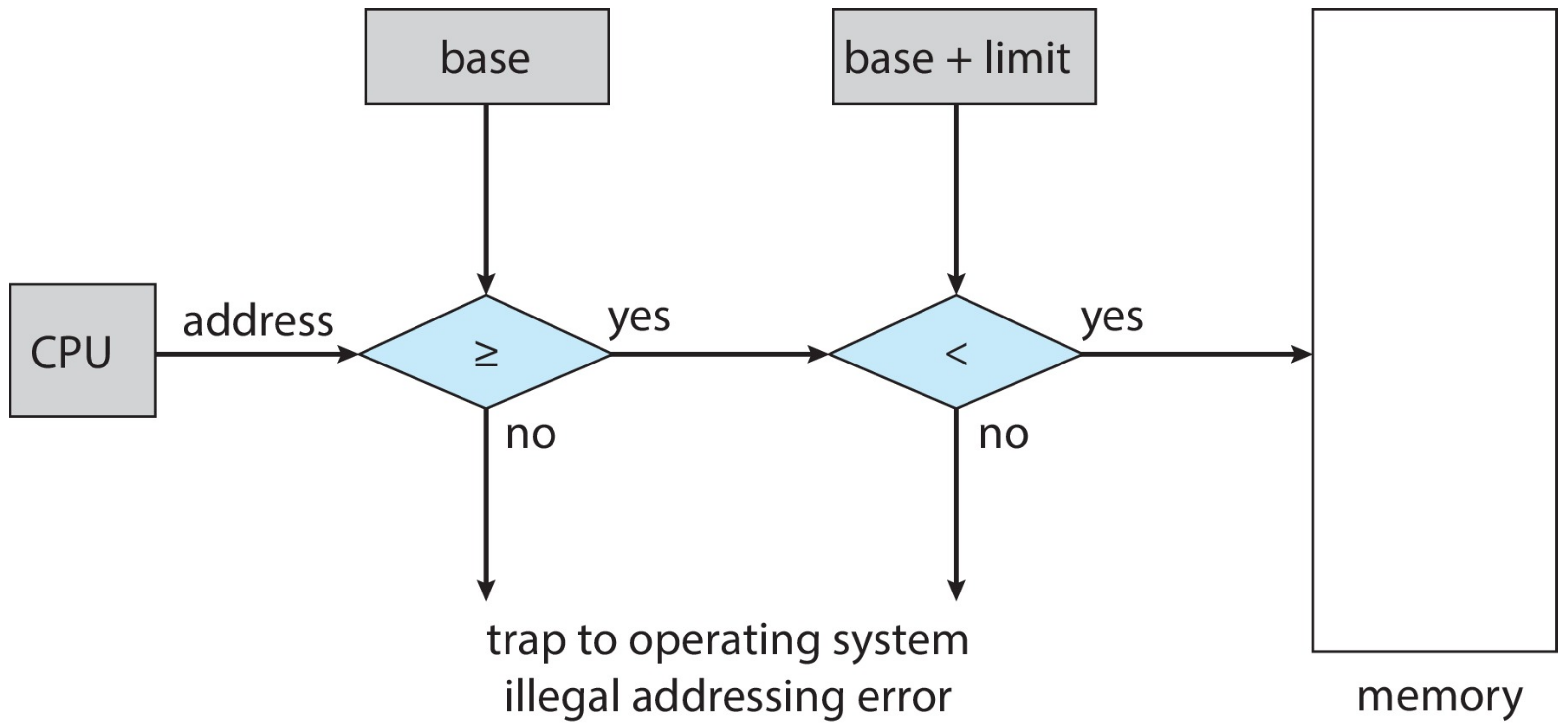
- Need to censure that a process can access only access those addresses in it address space.
- We can provide this protection by using a pair of **base** and **limit** registers define the logical address space of a process



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



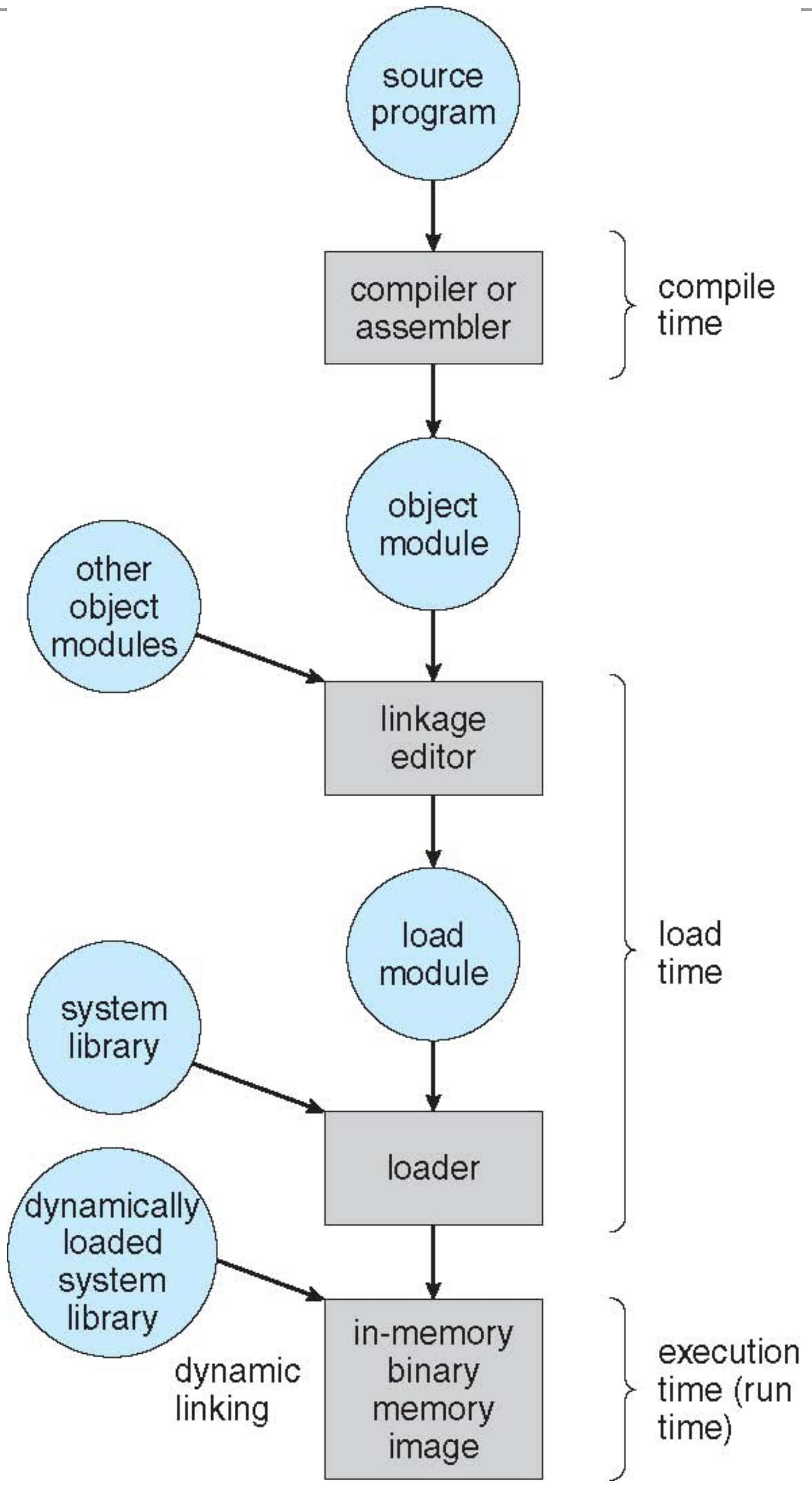
Address Binding

- Inconvenient to have first user process address always at 0 (**why**?)
    - (shared) libraries, *nil* pointer dereference detection…
- Addresses are represented in different ways at different stages of a program's life
    - **source code** addresses are usually **symbolic** (e.g., temp)
    - **compiler** binds symbols to **relocatable addresses**
        - e.g., "14 bytes from beginning of this module"
    - **linker** (or loader) binds relocatable addresses to **absolute addresses**
        - e.g., 0x0e74014
    - 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)

Multi-step Processing of a User Program

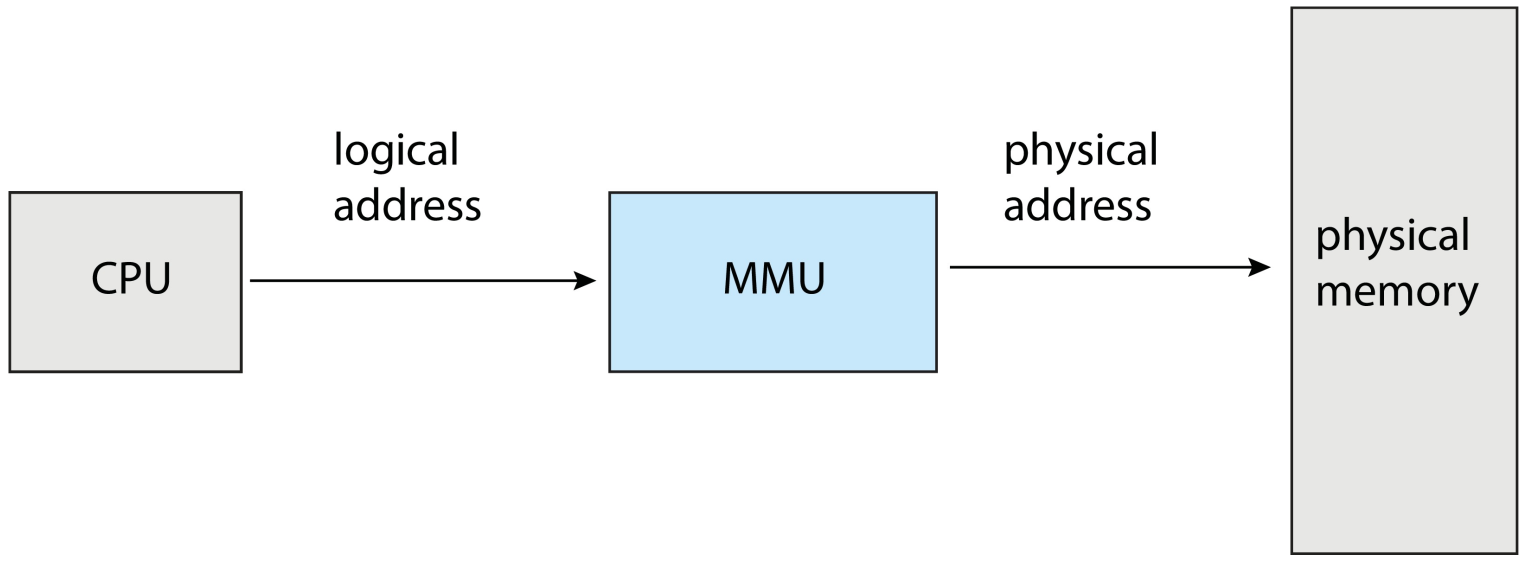


Logical vs. Physical Address Space

- 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



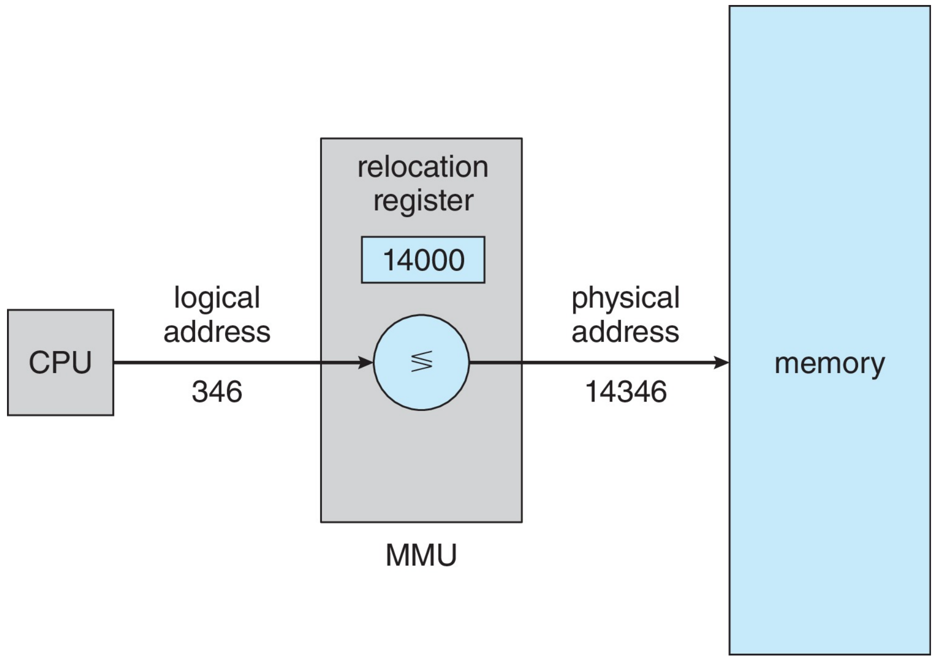
- Many methods possible, covered in the rest of this chapter

Memory-Management Unit

- 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
    - Execution-time binding occurs when reference is made to location in memory
    - Logical address bound to physical addresses

Memory-Management Unit

- 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
- Logical address 0 to max, physical address: R, R+max. What if these two sizes differ?



Dynamic Loading

- The entire program does need to be in memory to execute
- 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
- Advantage
  - Routine is loaded only when it is needed -> 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

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until **execution time**
  - Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for libraries
  - System also known as **shared libraries**
  - Consider applicability to patching system libraries
    - Versioning may be needed
    - What will happen without dynamic linking?
- Help from OS: share libraries between processes

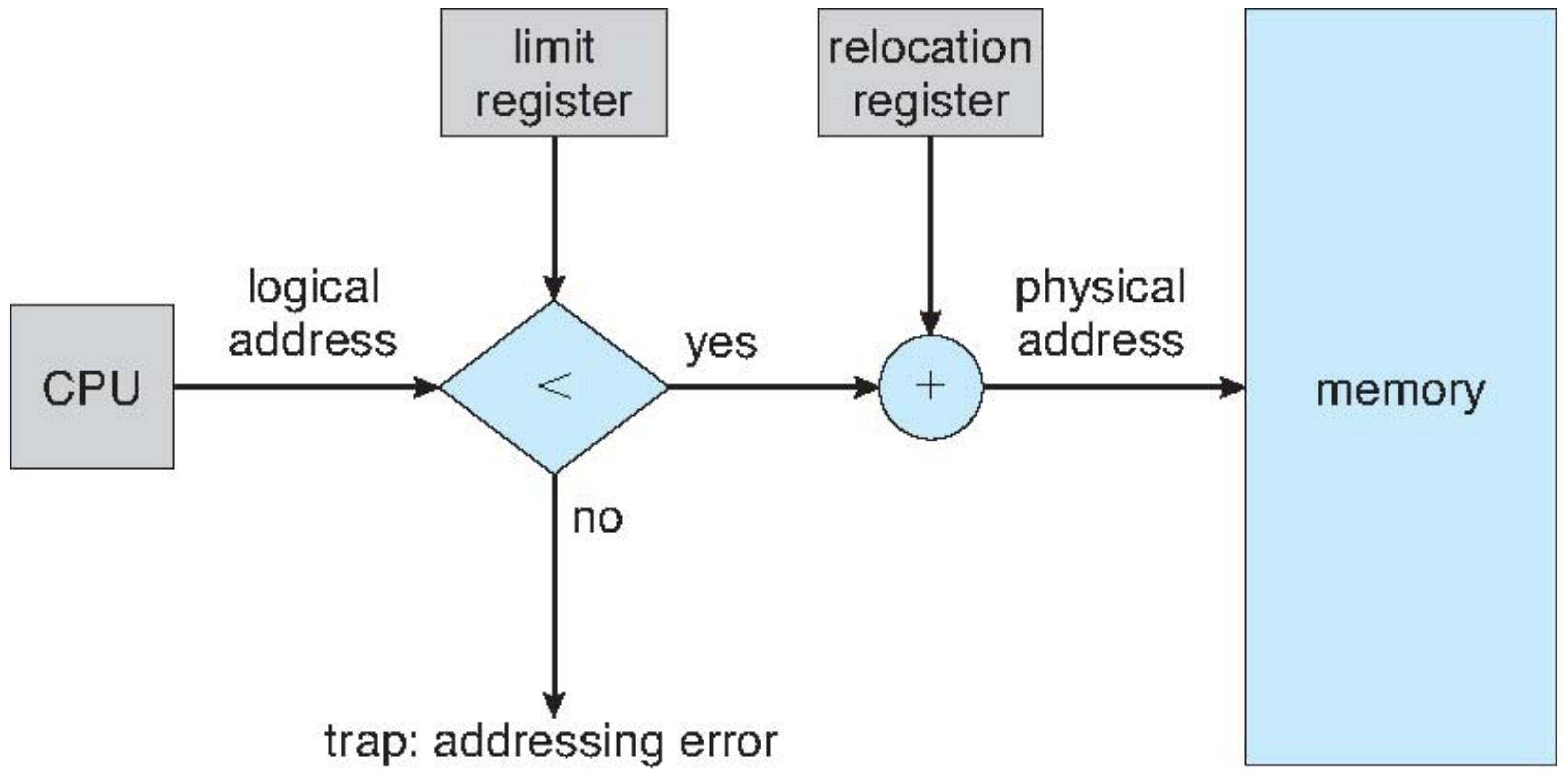
Contiguous Allocation

- 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**:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes then held in high memory
    - Each process contained in single contiguous section of memory

Contiguous Allocation: 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
  - MMU maps logical address dynamically
  - Can then allow actions such as kernel code being **transient** and kernel changing size

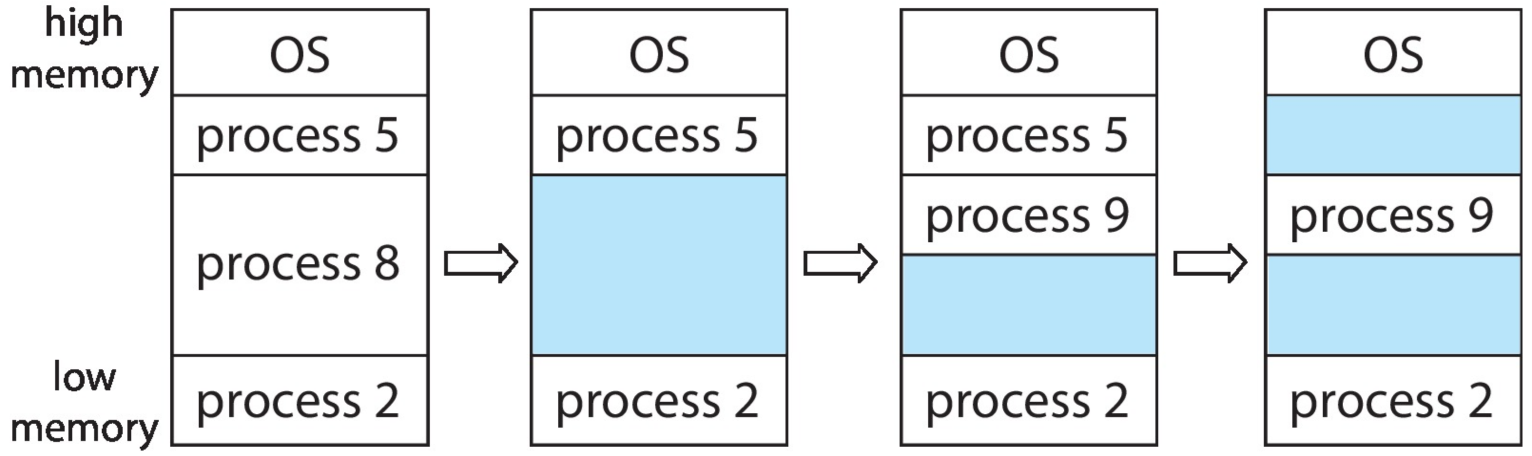
Hardware Support for Relocation and Limit Registers



Memory Allocation: Variable Partition

- Multiple-partition 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
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:

  a) allocated partitions b) free partitions (hole)



Memory Allocation

- How to satisfy a request of size *n* from a list of free memory blocks?
  - **first-fit**: allocate from the first block that is big enough
  - **best-fit**: allocate from the smallest block that is big enough
    - must search entire list, unless ordered by size
    - produces the smallest leftover hole
  - **worst-fit**: allocate from the largest hole
    - must also search entire list
    - produces the largest leftover hole
- **Fragmentation** is big problem for all three methods
  - first-fit and best-fit usually perform better than worst-fit

Fragmentation

- **External fragmentation**
  - unusable memory between allocated memory blocks
    - **total amount** of free memory space **is larger than a request**
    - the request cannot be fulfilled because the free memory is not **contiguous**
  - external fragmentation can be reduced by **compaction**
    - shuffle memory contents to place all free memory in one large block
    - program needs to be **relocatable** at runtime
    - Performance overhead, timing to do this operation
  - Another solution: **paging**
  - 50-percent rule: N allocated blocks, 0.5N will be lost due to fragmentation. 1/3 is unusable!

Fragmentation

- **Internal fragmentation**
  - memory allocated may be larger than the requested size
  - this size difference is memory *internal to a partition*, but not being used
  - Example: free space 18464 bytes, request 18462 bytes
- Sophisticated algorithms are designed to avoid fragmentation
  - none of the first-/best-/worst-fit can be considered sophisticated

Review

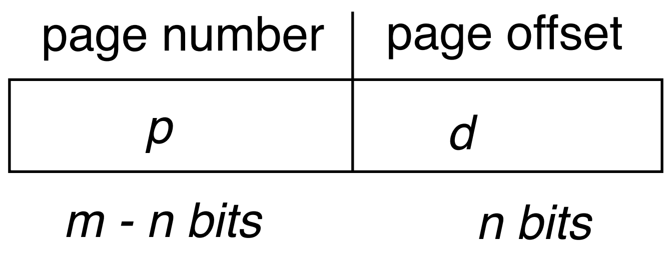
- Address binding
- Logical address vs physical address
- MMU: address translation + protection
- Memory allocation
  - Contiguous allocation: first, best, worst fit
- Fragmentation: external/internal

Paging

- Physical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
  - Avoids **external fragmentation -> avoid for compacting**
  - Avoids problem of varying sized memory chunks
- Basic methods
  - 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
  - Backing store likewise split into pages
  - Still have Internal fragmentation

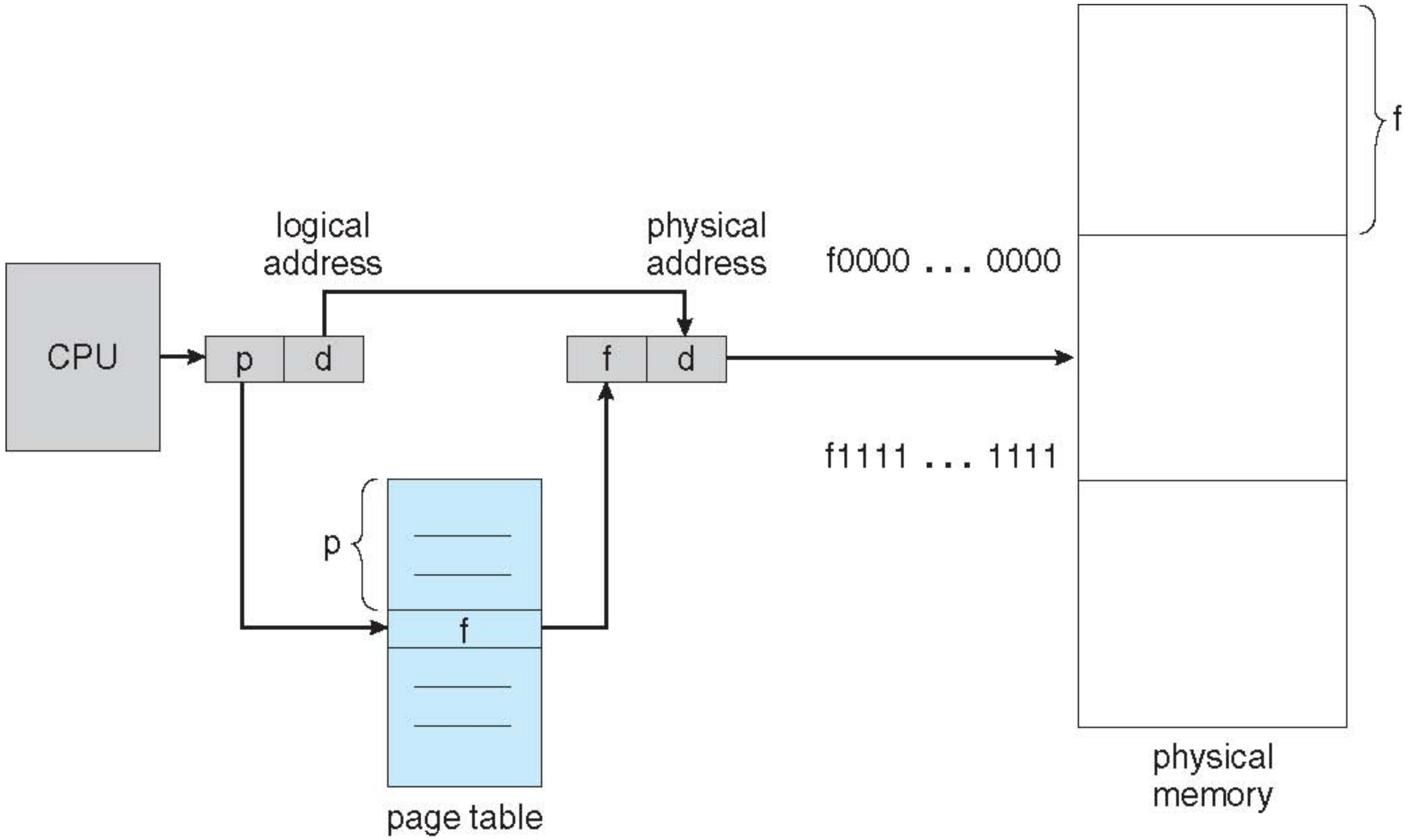
Paging: Address Translation

- A logical address is divided into:
  - **page number** (p)
    - used as an index into a page table
    - page table entry contains the corresponding **physical** frame number
  - **page offset** (d)
    - offset within the page/frame
    - combined with frame number to get the physical address

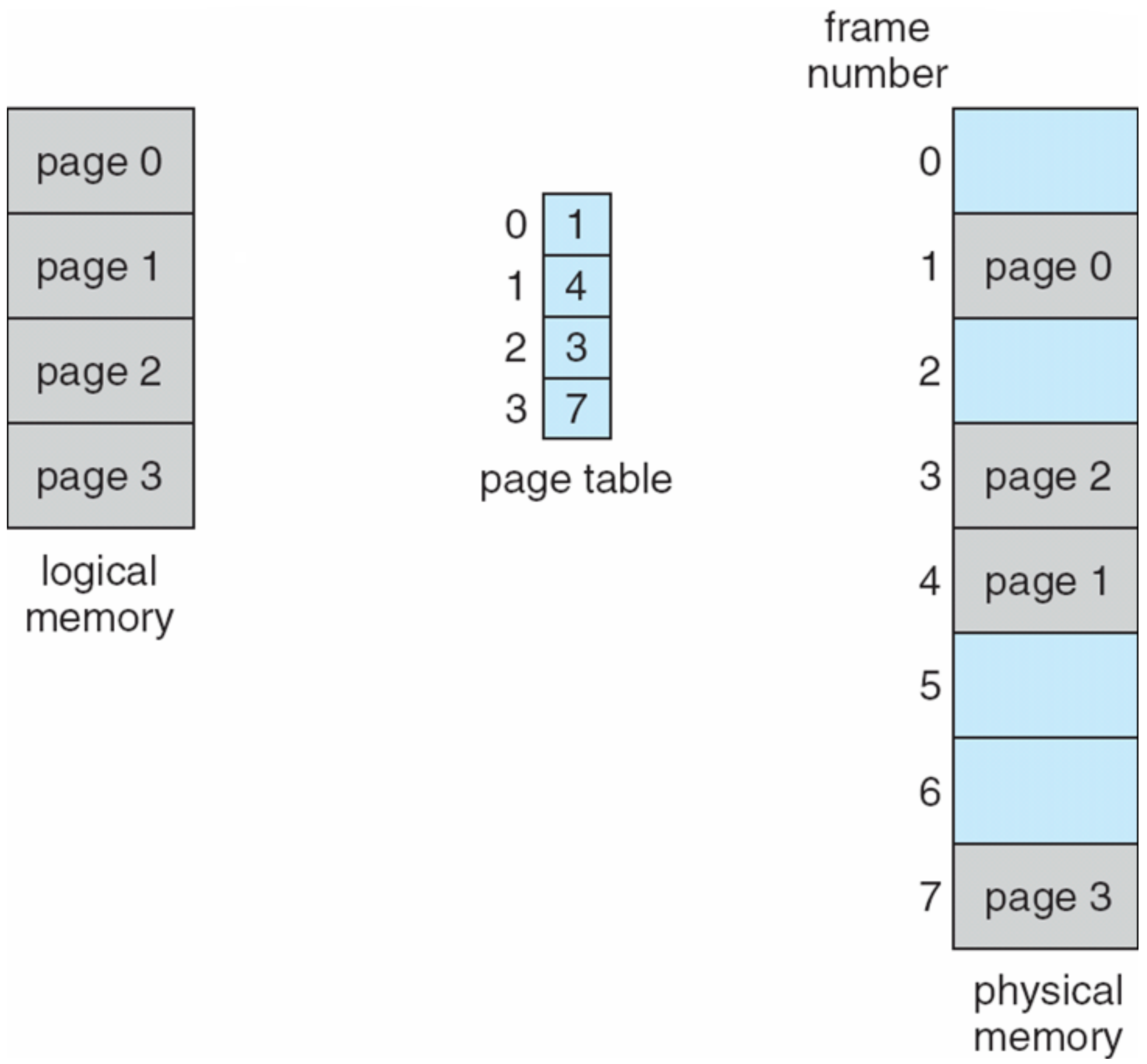


m bit logical address space, *n* bit page size

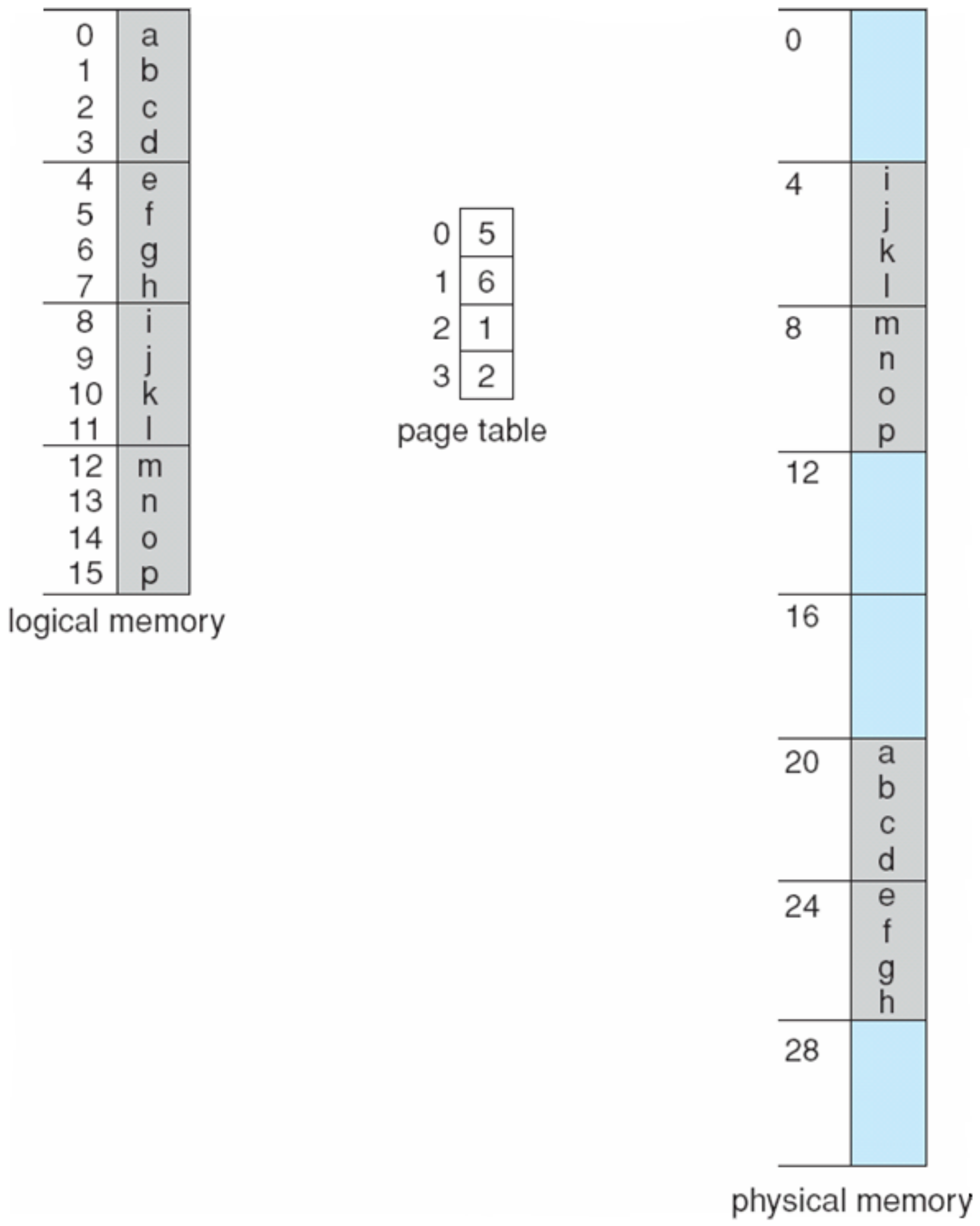
Paging Hardware



Paging Example



Paging Example II



m = 4 and *n* = 2   32-byte memory and 4-byte pages

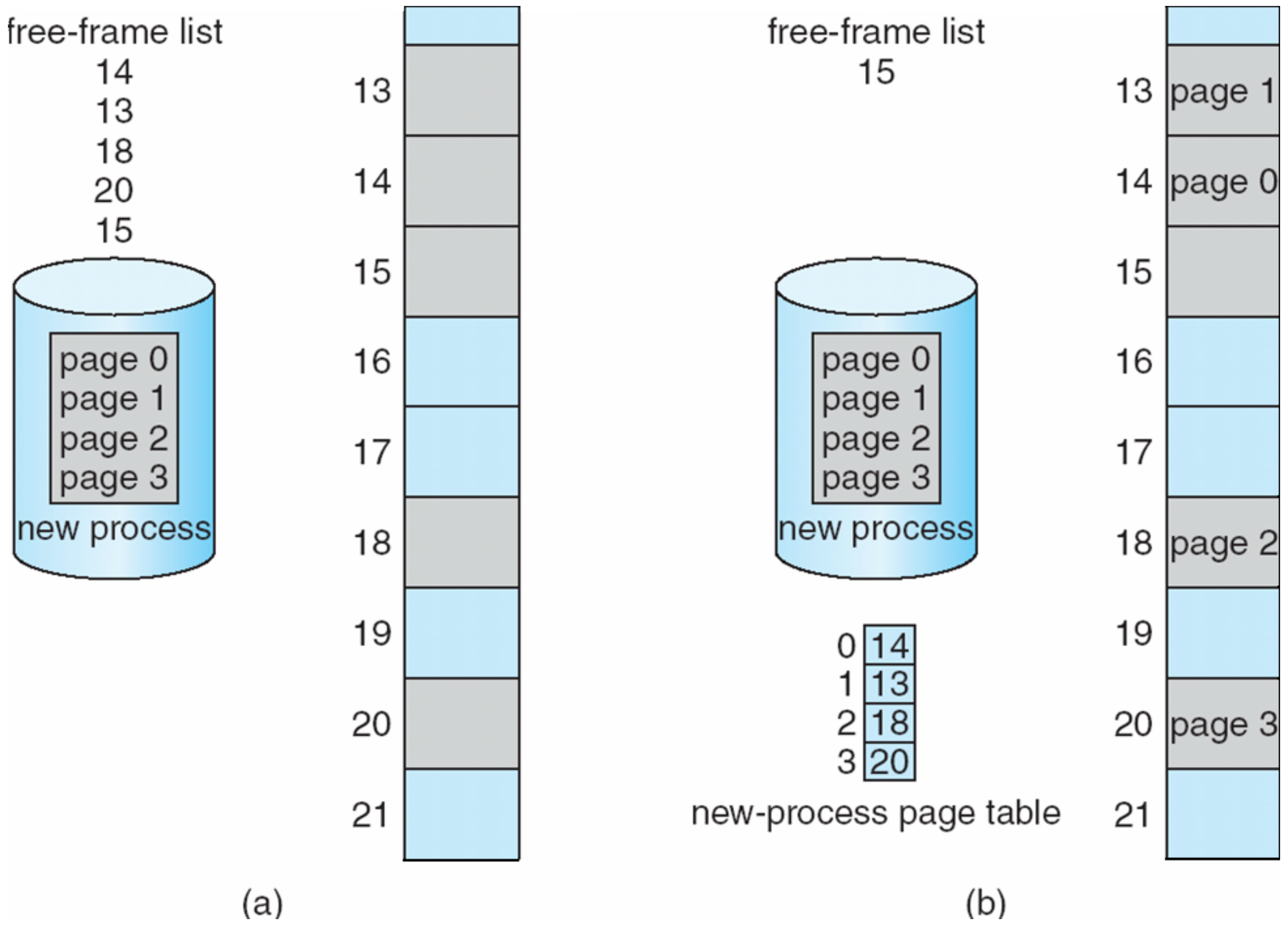
Paging: Internal Fragmentation

- Paging has **no external fragmentation**, but **internal fragmentation**
  - e.g., page size: 2,048, program size: 72,766 (35 pages + 1,086 bytes)
    - internal fragmentation: 2,048 - 1,086 = 962
  - **worst** case internal fragmentation: **1 frame – 1 byte**
  - **average** internal fragmentation: 1 / 2 frame size
- Small frame sizes more desirable than large frame size?
  - memory becomes larger, and page table takes memory
  - page sizes actually grow over time
    - 4KB ➔ 2MB ➔ 4MB ➔ 1GB ➔ 2GB

Frame Table

- OS is managing physical memory, it should be aware of the allocation details of physicals
    - Which frame is free, and how many frames have been allocated …
    - One entry for each physical frame
    - the allocated frame belongs to which process

Free Frames



Before allocation

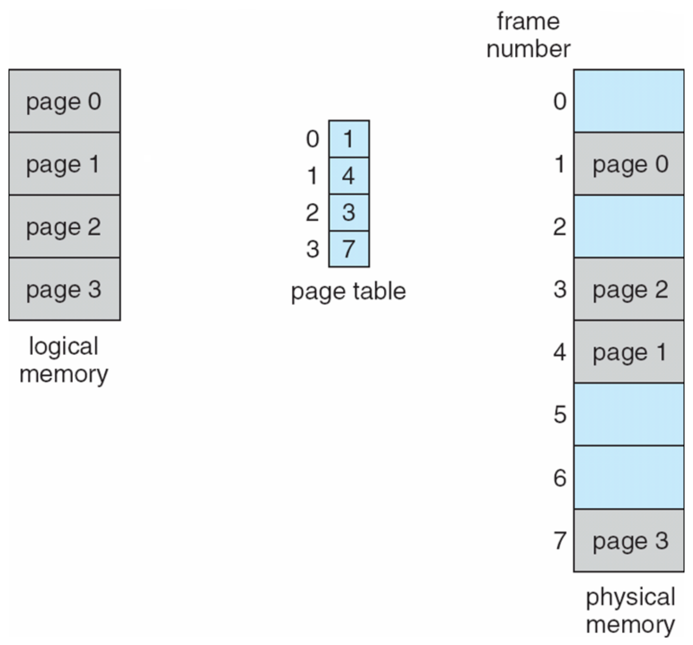
After allocation

Hardware Support: Simplest Case

- Page table is in a set of dedicated registers
    - Advantages: very efficient - access to register is fast
    - Disadvantages: the table size is very small, and the context switch need to save and restore these registers

Hardware Support: Alternative Way

- One big page table maps logical address to physical address
  - the page table should be **kept in main memory**
  - **page-table base register** (**PTBR**) points to the page table
    - **does PTBR contain physical or logical address?**
  - **page-table length register** (**PTLR**) indicates the size of the page table
- Every data/instruction access requires **two memory accesses**
  - one for the page table and one for the data / instruction
  - CPU can cache the translation to avoid one memory access (**TLB**)



TLB

- TLB (**translation look-aside buffer**) caches the address translation
    - if page number is in the TLB, no need to access the page table
    - if page number is not in the TLB, need to replace one TLB entry
    - TLB usually use a fast-lookup hardware cache called **associative memory**
    - TLB is usually small, 64 to 1024 entries
- Use with page table
    - TLB contains a few page table entries
    - Check whether page number is in TLB
        - If -> frame number is available and used
        - If not -> **TLB miss**. access page table and then fetch into TLB
            - TLB flush: TLB entries are full
            - TLB wire down: TLB entries should not be flushed

TLB

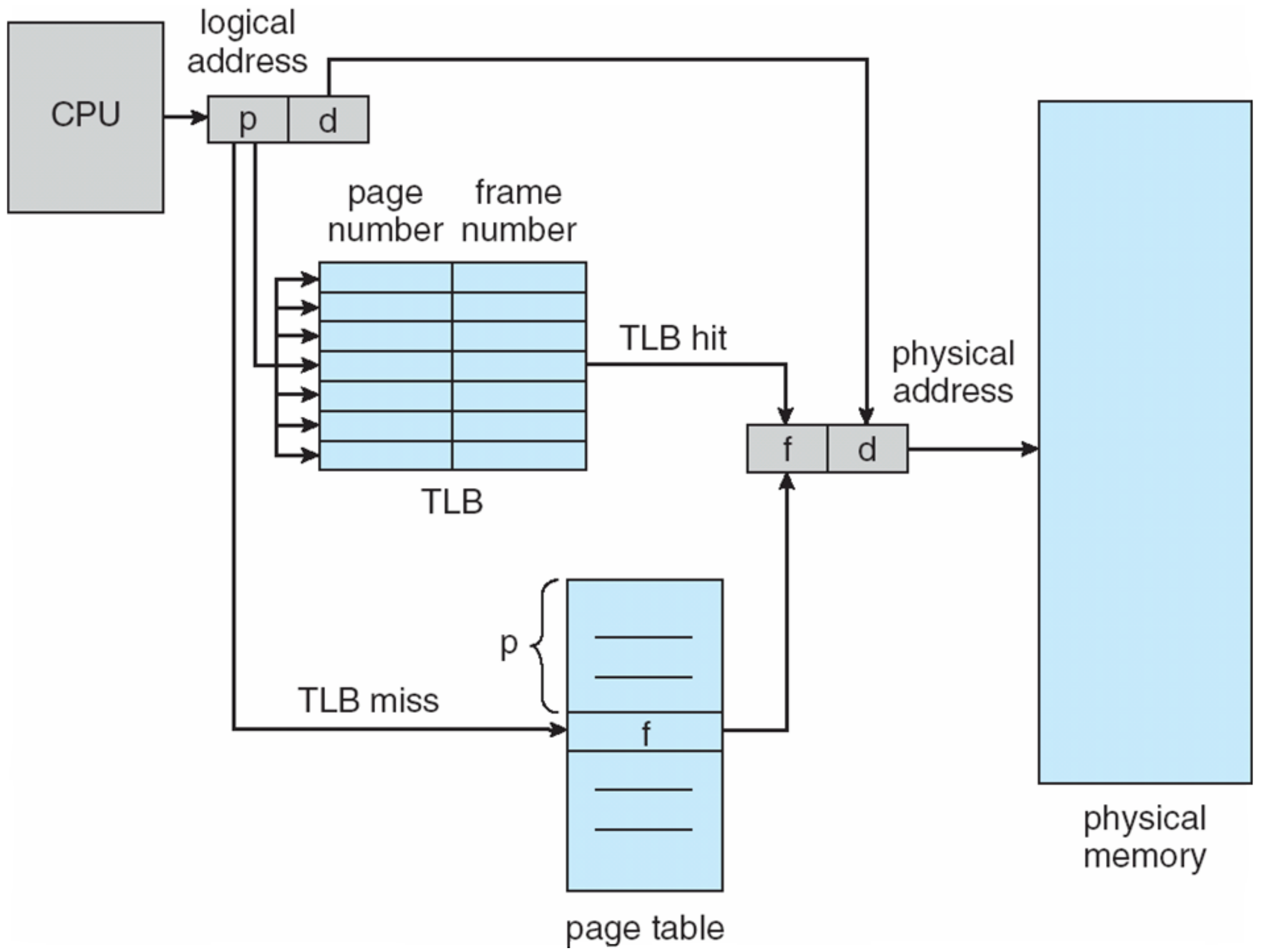
- TLB and context switch
  - Each process has its own page table
    - switching process needs to switch page table
  - **TLB must be consistent with page table**
    - Option I: Flush TLB at every context switch, **or**,
    - Option II: Tag TLB entries with **address-space identifier** (ASID) that uniquely identifies a process
  - some TLB entries can be **shared** by processes, and fixed in the TLB
    - e.g., TLB entries for the kernel
- TLB and operating system
  - MIPS: OS should deal with TLB miss exception
  - X86: TLB miss is handled by hardware

Associative Memory

- Associative memory: memory that supports **parallel search**
- Associative memory is not addressed by “addresses”, but **contents**
  - if p is in associative memory’s key, return frame# (value) directly
  - think of hash tables

Page #	Frame #
1	7
2	12
3	15
4	31

Paging Hardware With TLB



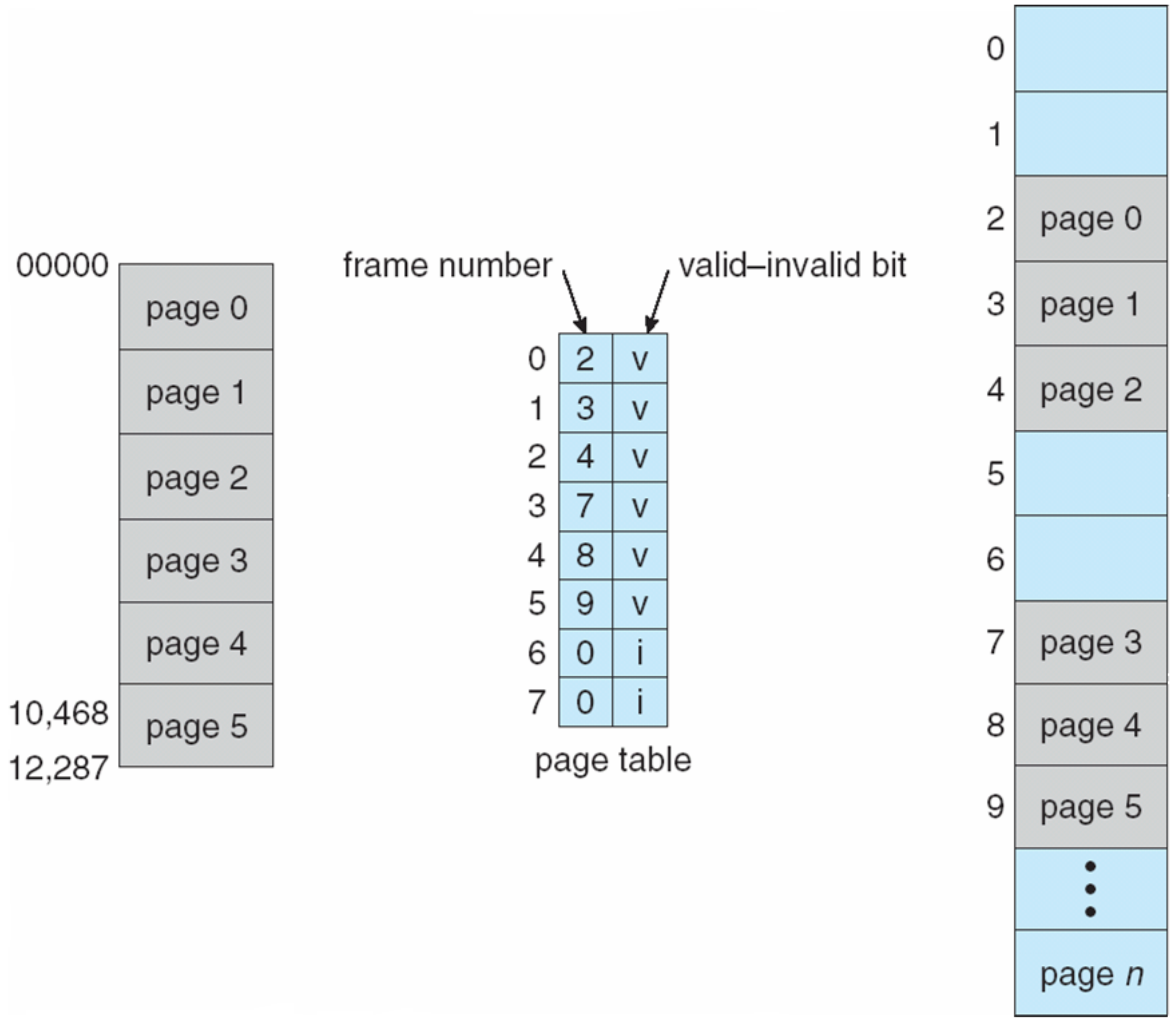
Effective Access Time

- **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: page table + memory access
- Effective Access Time (EAT)
- EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds
- implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,
- EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns
- implying only 1% slowdown in access time.

Memory Protection

- Accomplished by protection bits with each frame
- Each page table entry has a **present** (aka. valid) bit
  - present: the page has a valid physical frame, thus can be accessed
- Each page table entry contains some protection bits
  - **kernel/user**, **read/write**, **execution**?, **kernel-execution**?
  - why do we need them?
- Any violations of memory protection result in a trap to the kernel

Memory Protection



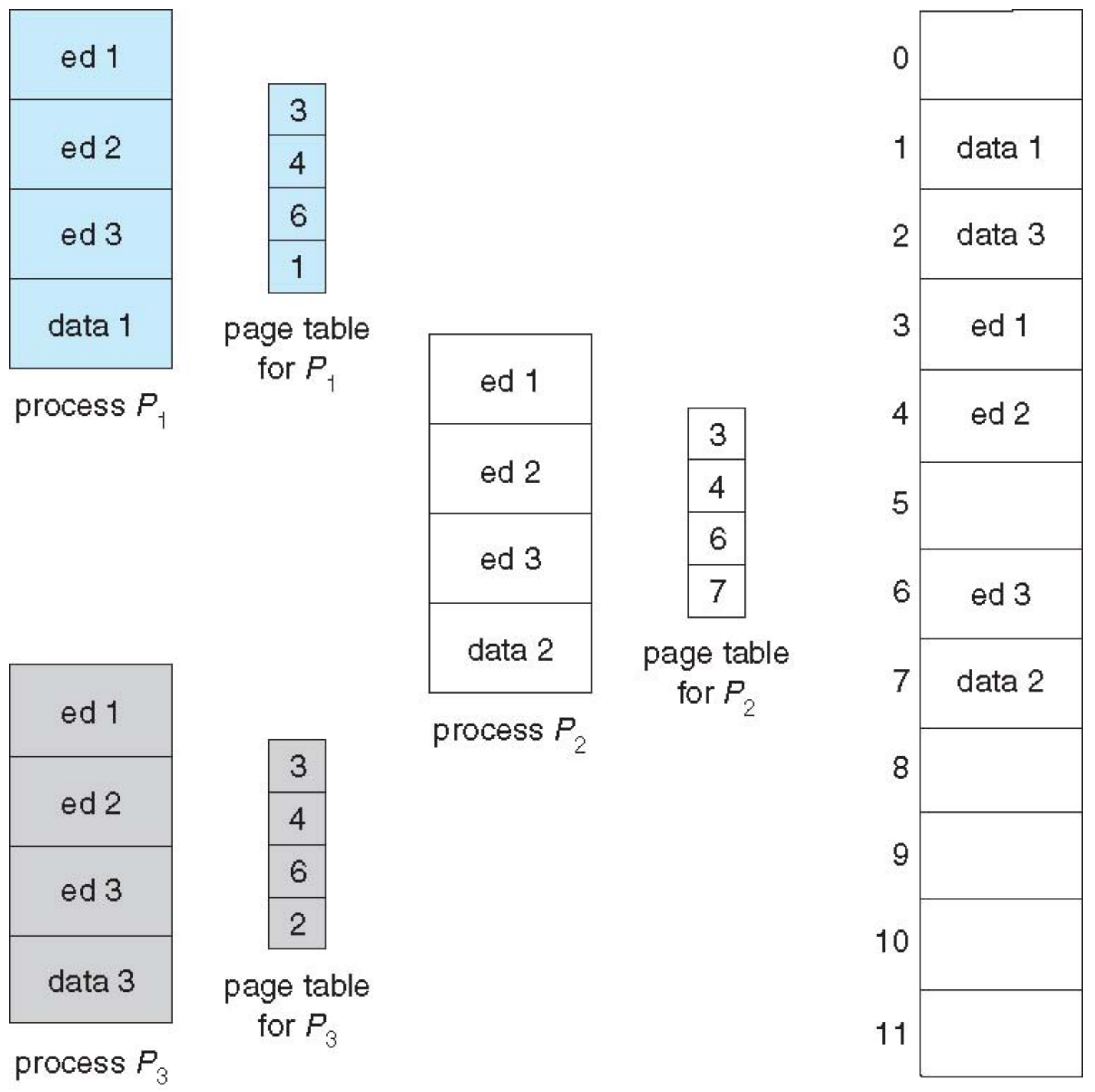
Memory Protection (more)

- **NX bit**
  - segregate areas of memory for use by either storage of processor instructions (code) or for storage of data
  - Intel: XD(execute disable), AMD: EVP (enhanced virus protection), ARM: XN (execute never)
- PXN: Privileged Execute Never （intel: SMEP)
  - A Permission fault is generated if the processor is executing at **PL1(kernel)** and attempts to execute an instruction fetched from the corresponding memory region when this PXN bit is 1 (usually user space memory)

Page Sharing

- Paging allows to share memory between processes
  - e.g., one copy of **code** shared by **all processes of the same program**
    - text editors, compilers, browser..
  - shared memory can be used for **inter-process communication**
  - shared libraries
- Reentrant code: non-self-modifying code: never changes between execution
- Each process can, of course, have its private code and data

Page Sharing



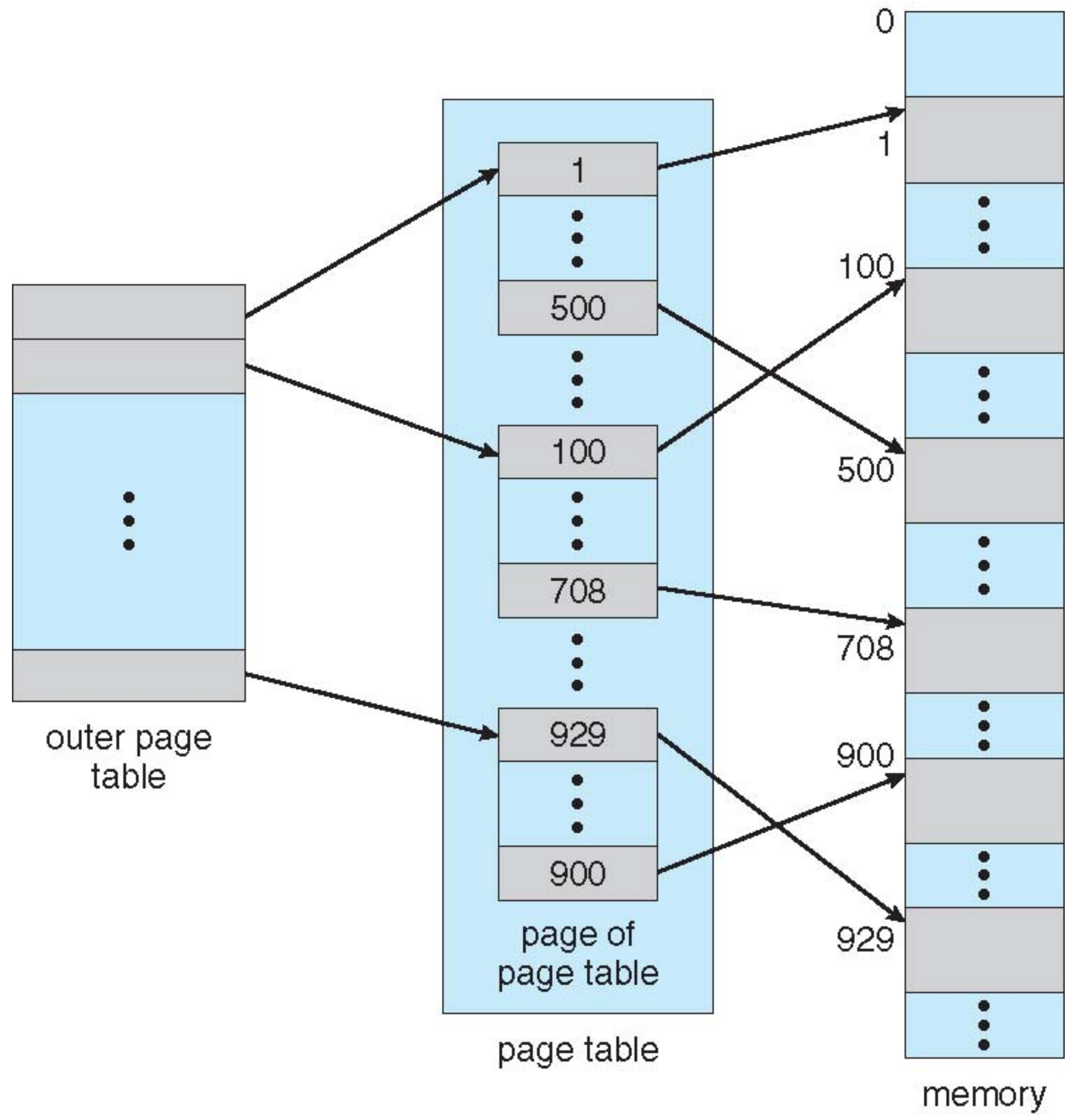
Structure of Page Table

- One-level page table can consume lots of memory for page table
  - e.g., 32-bit logical address space and 4KB page size
    - page table would have 1 million entries (2^{32} / 2^{12})
    - if each entry is 4 bytes ➔ 4 MB of memory for page table alone
  - each process requires its own page table
  - page table must be **physically contiguous**
- To reduce memory consumption of page tables:
  - **hierarchical page table**
  - **hashed page table**
  - **inverted page table**

Hierarchical Page Tables

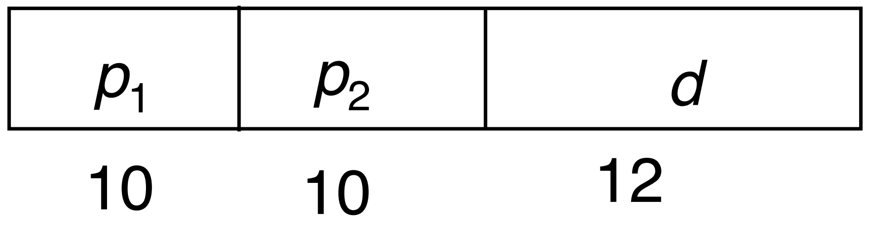
- Break up the logical address space into **multiple-level** of page tables
  - e.g., two-level page table
  - first-level page table contains the frame# for second-level page tables
    - “page” the page table
- Why hierarchical page table can save memory for page table?

Two-Level Page Table

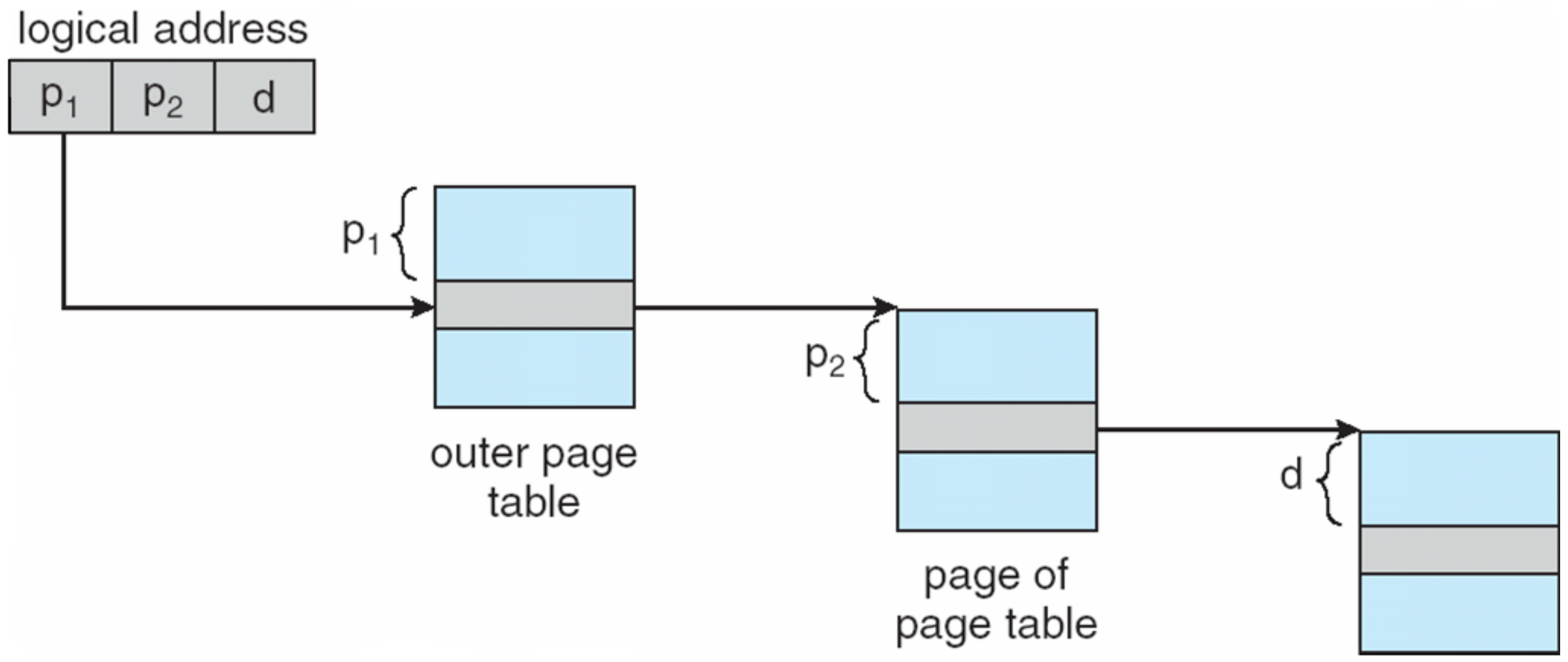


Two-Level Paging

- A logical address is divided into:
  - a **page directory number** (first level page table)
  - a **page table number** (2nd level page table)
  - a **page offset**
- Example: 2-level paging in 32-bit Intel CPUs
  - 32-bit address space, 4KB page size
  - 10-bit page directory number, 10-bit page table number
  - each page table entry is 4 bytes, one frame contains 1024 entries (2^{10})



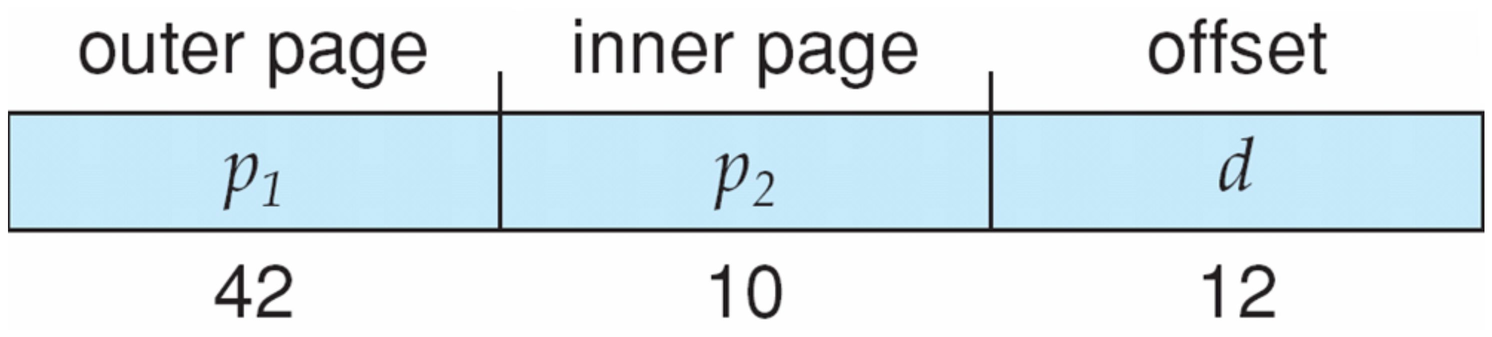
Address-Translation Scheme

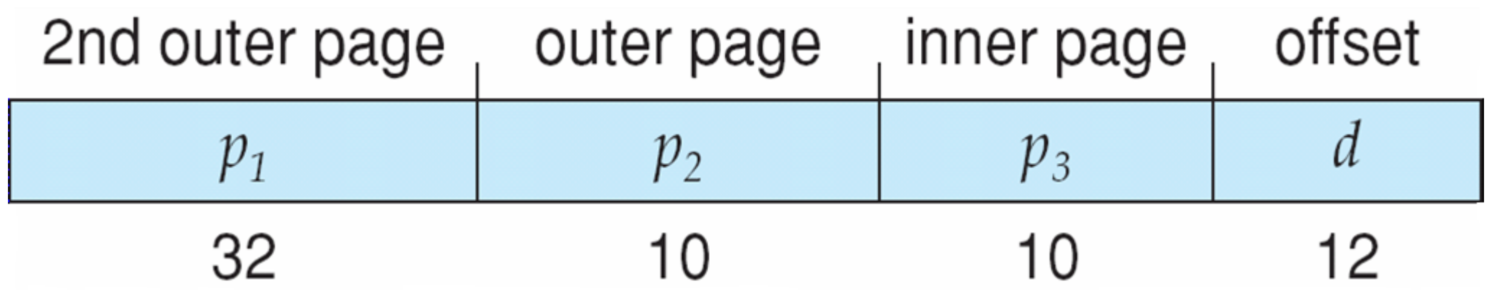


64-bit Logical Address Space

- 64-bit logical address space requires more levels of paging
  - two-level paging is not sufficient for 64-bit logical address space
    - if page size is 4 KB (2^{12}), outer page table has 2^{42} entries, inner page tables have 2^{10} 4-byte entries
  - one solution is to add more levels of page tables
    - e.g., three levels of paging: 1st level page table is 2^{34} bytes in size
    - and possibly 4 memory accesses to get to one physical memory location
  - usually **not support full 64-bit virtual address space**
    - AMD-64 supports 48-bit
    - canonical form: 48 through 63 of valid virtual address must be copies of bit 47

64-bit Logical Address Space

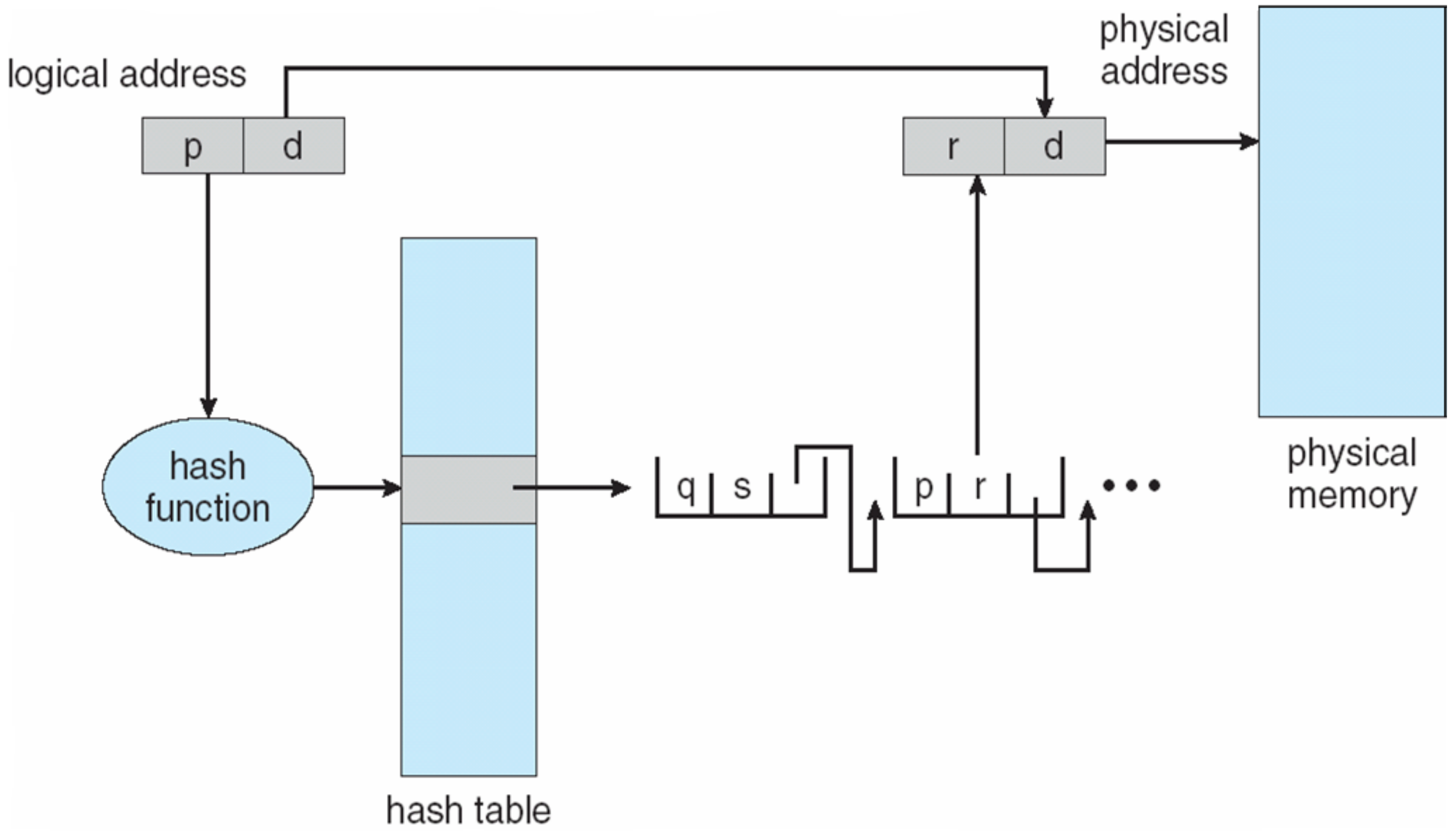




Hashed Page Tables

- In hashed page table, virtual page# is hashed into a frame#
  - the page table contains a chain of elements hashing to the same location
  - each element contains: **page#**, **frame#**, and 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 frame# is returned
- Hashed page table is common in address spaces > 32 bits
- **Clustered page tables**
  - Each entry refers to several pages

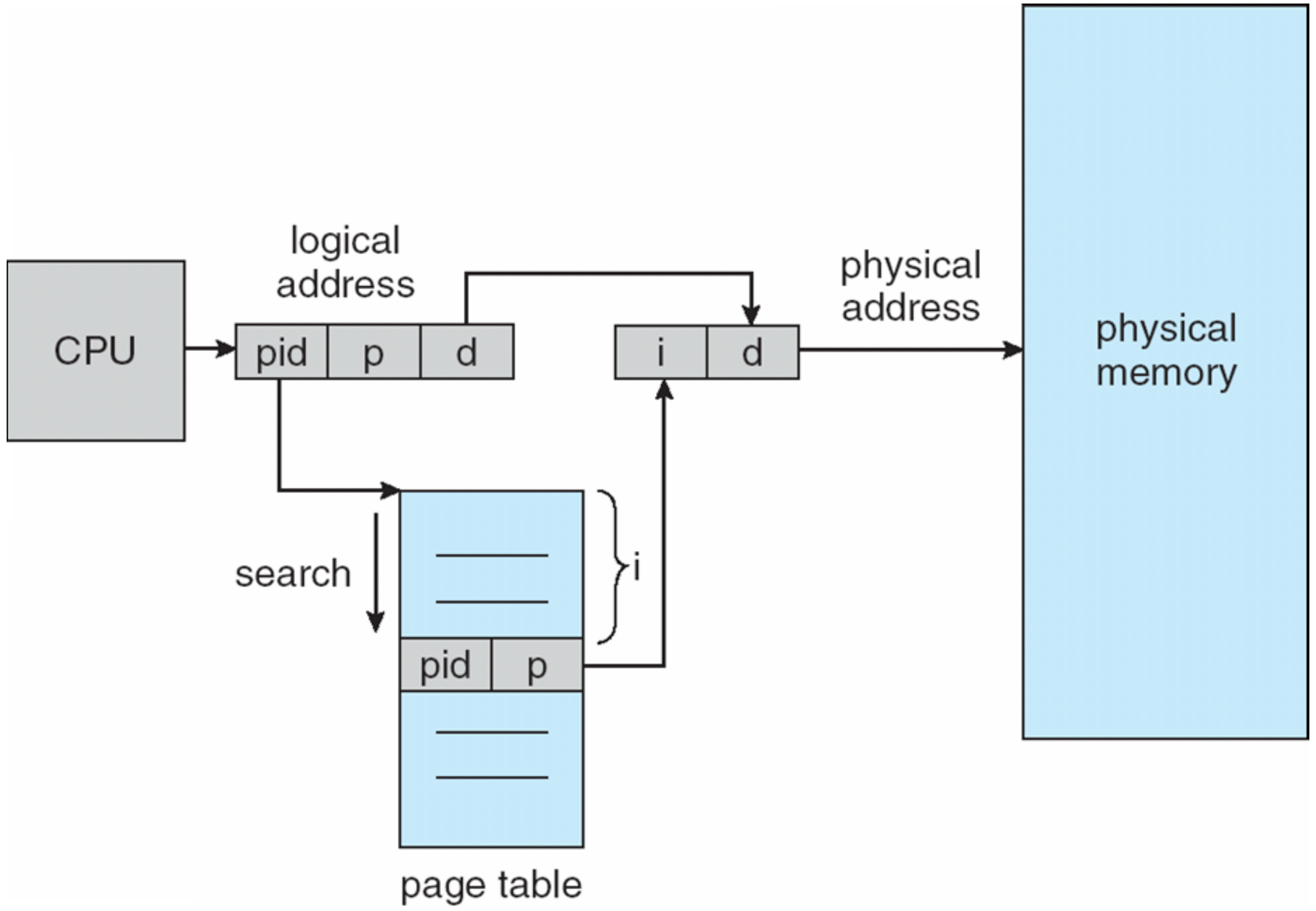
Hashed Page Table



Inverted Page Table

- Inverted page table tracks allocation of physical frame to a process
  - one entry for each physical frame ➔ fixed amount of memory for page table
  - each entry has the **process id** and the **page#** (virtual address)
- Sounds like a brilliant idea?
  - to translate a virtual address, it is necessary to search the (**whole**) page table
    - can use TLB to accelerate access, TLB miss could be very expensive
  - how to implement shared memory?
    - a physical frame can only be mapped into one process!
    - Because one physical memory page cannot have multiple virtual page entry!

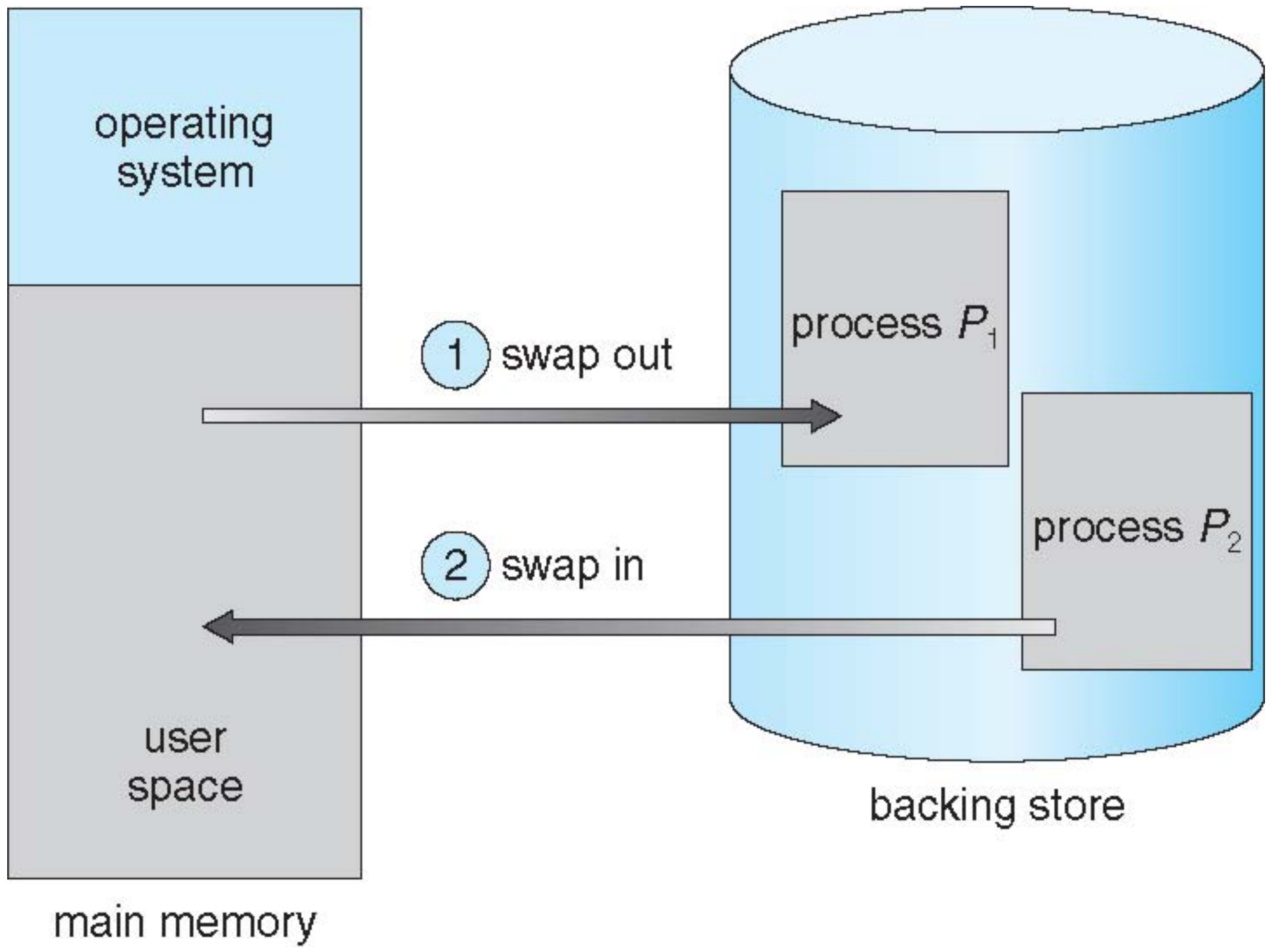
Inverted Page Table



Swapping

- **Swapping** extends physical memory with backing disks
  - a process can be swapped temporarily out of memory to a backing store
    - backing store is usually a (fast) disk
  - the process will be brought back into memory for continued execution
    - does the process need to be swapped back in to same physical address?
- Swapping is usually only initiated under memory pressure
- Context switch time can become very high due to swapping
  - if the next process to be run is not in memory, need to swap it in
  - disk I/O has high latency

Swapping

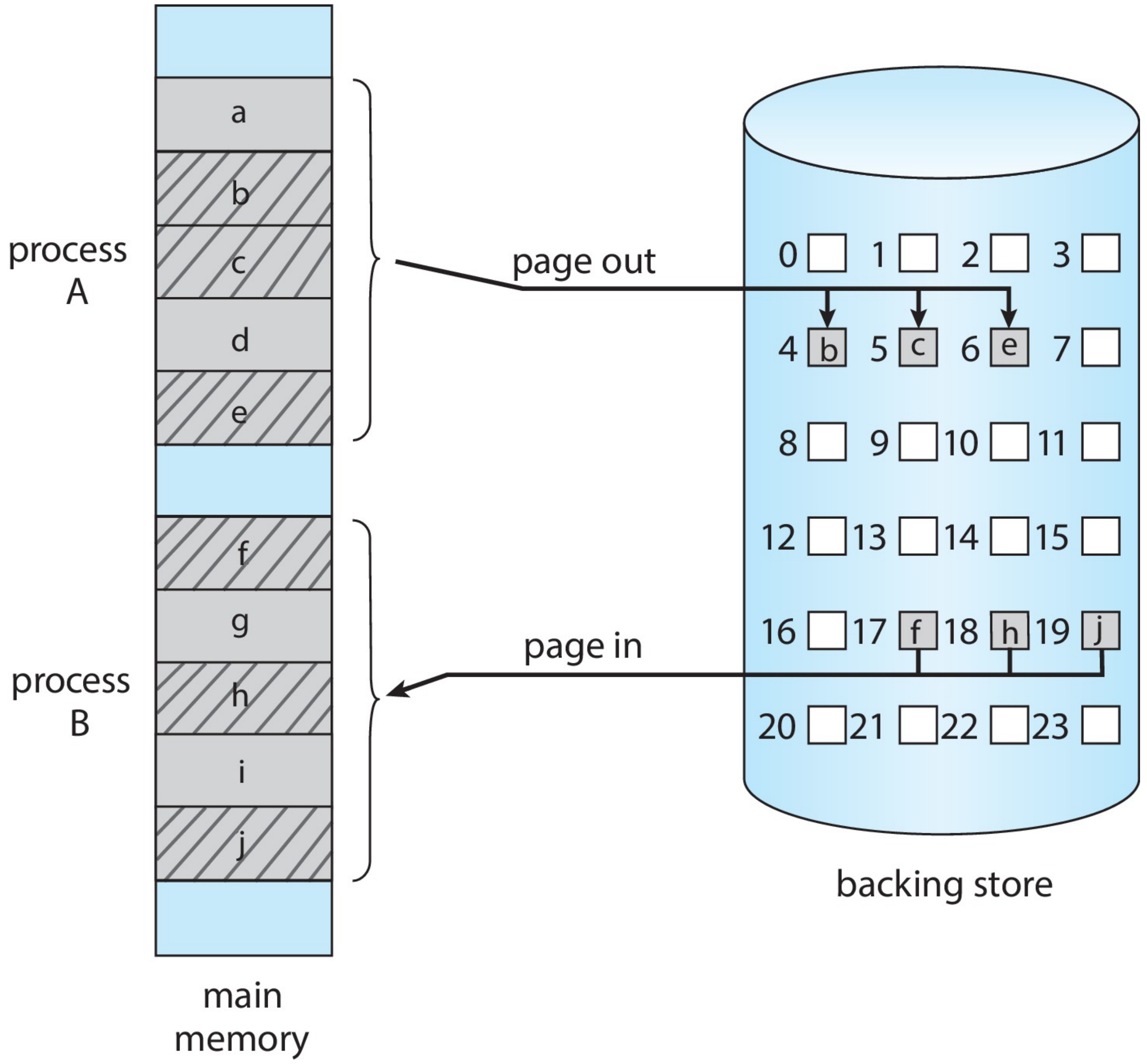

operating system
1 swap out
process P_1
process P_2
2 swap in
user space
backing store
main memory

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 2,000 **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

Swapping with Paging

- Swap pages instead of entire process



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 below

Example: The Intel 32 and 64-bit Architectures

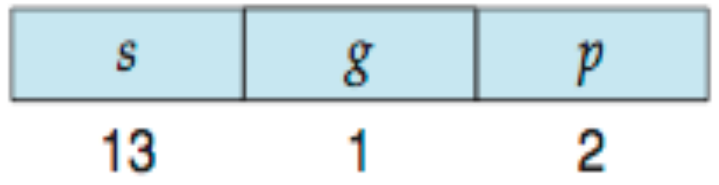
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
    - Each segment can be 4 GB
    - Up to 16 K segments per process
    - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)

Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
  - **Selector** given to segmentation unit
    - Which produces linear addresses

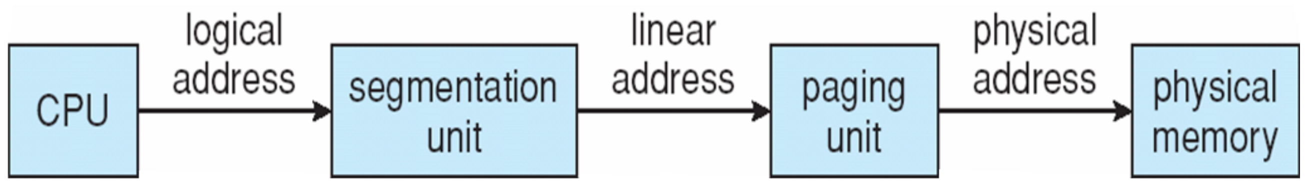


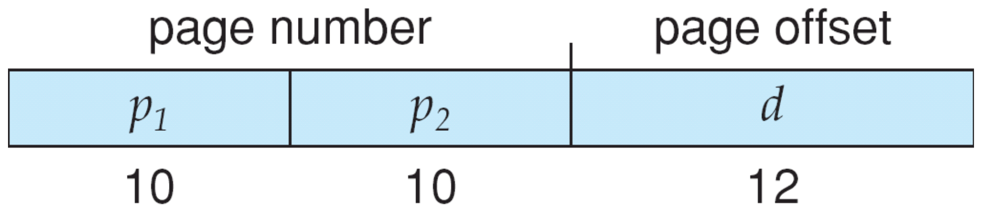
  - s-> segment number, g-> local/global, p->protection
  - Segment selector is stored in the segment registers: CS, DS etc.
  - GDTR, LDTR -> base address of the descriptor table
  - descriptor: base, limit and other bits

Example: The Intel IA-32 Architecture

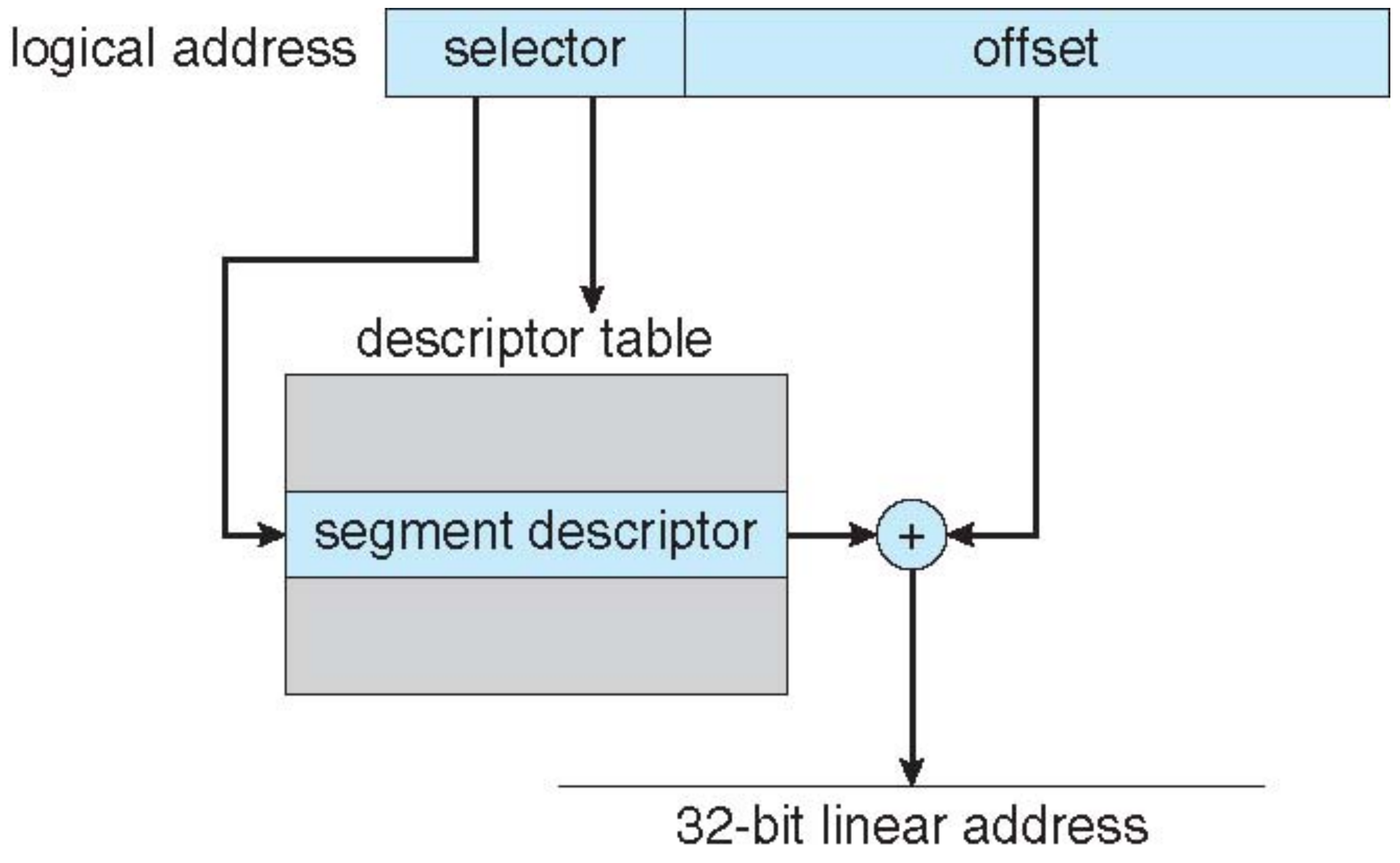
- Linear address given to paging unit
  - Which generates physical address in main memory
  - Paging units form equivalent of MMU
  - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

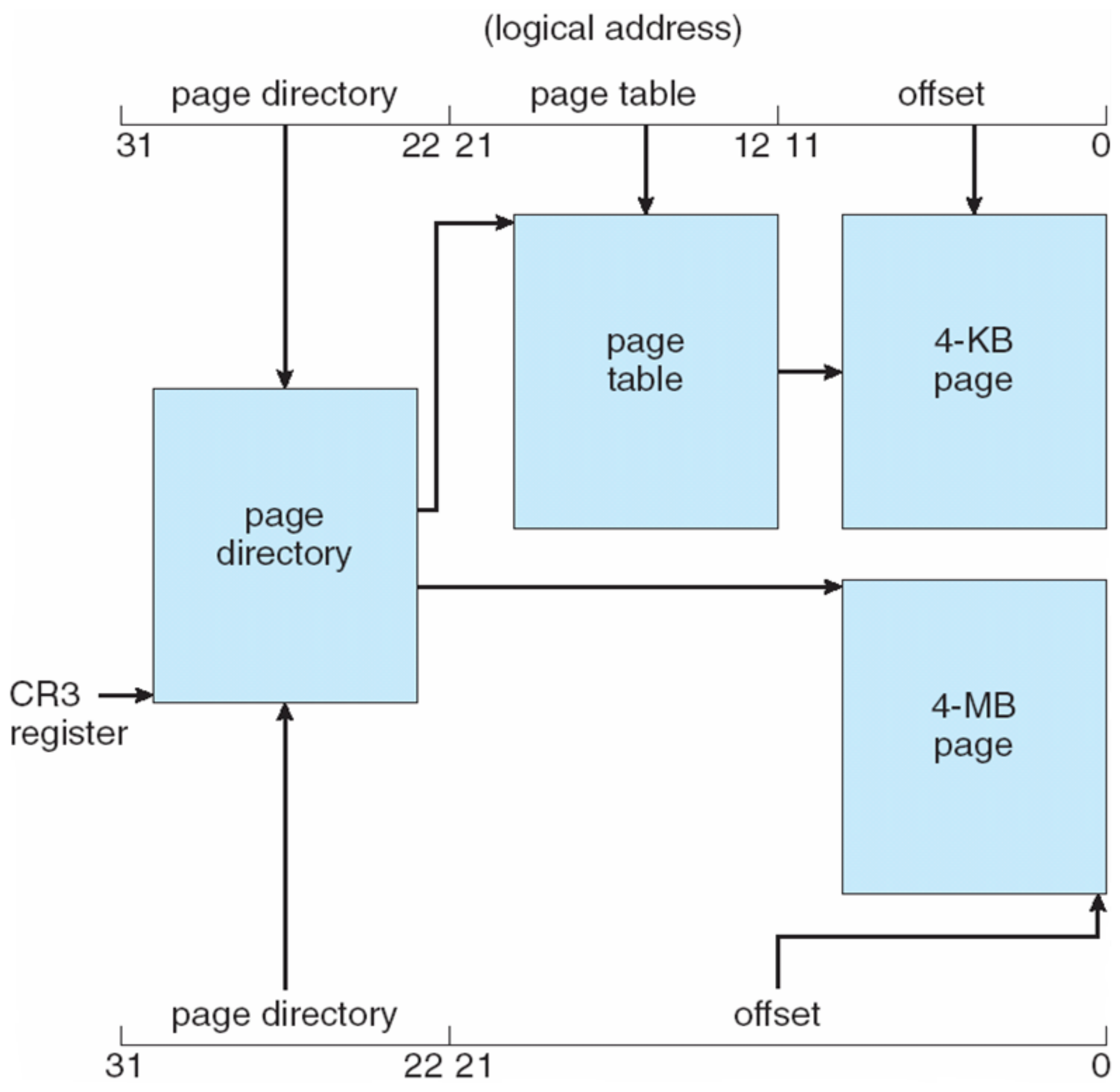




Intel IA-32 Segmentation

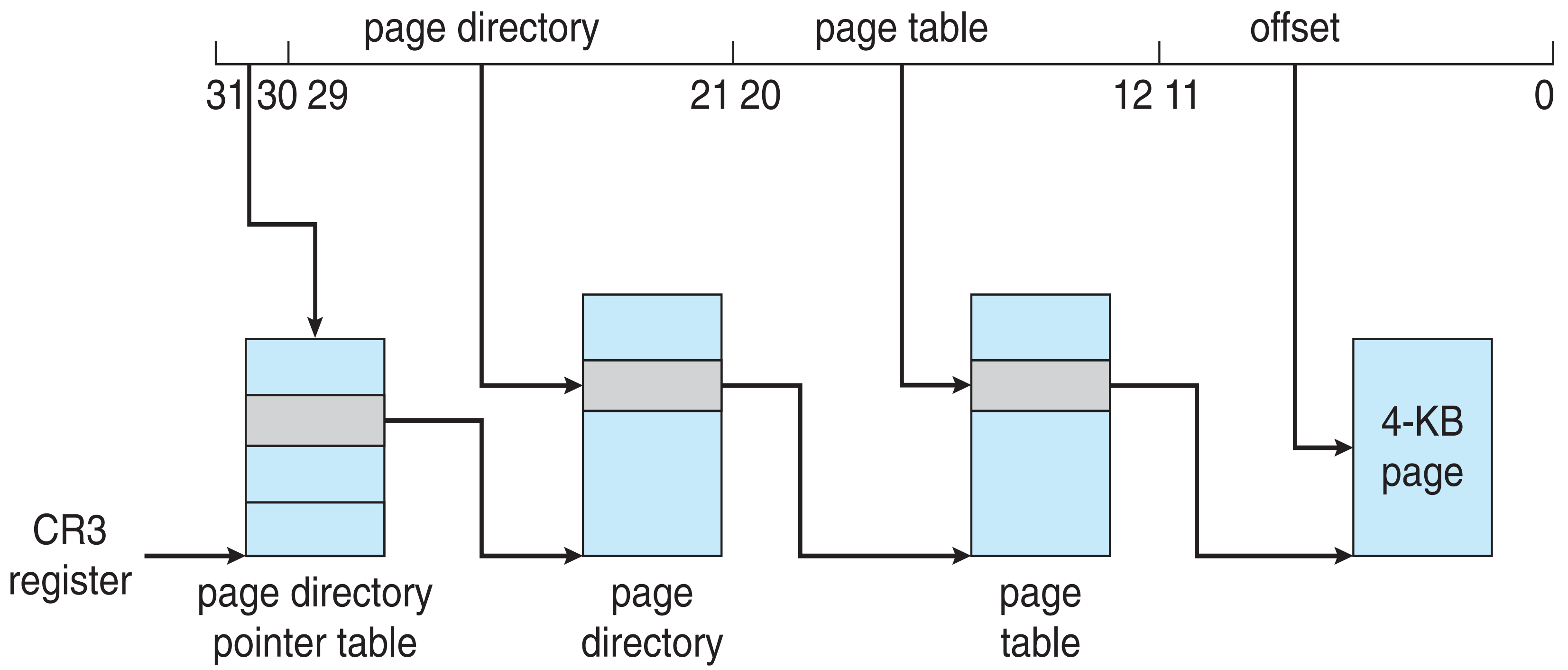


Intel IA-32 Paging Architecture



Intel IA-32 Page Address Extensions

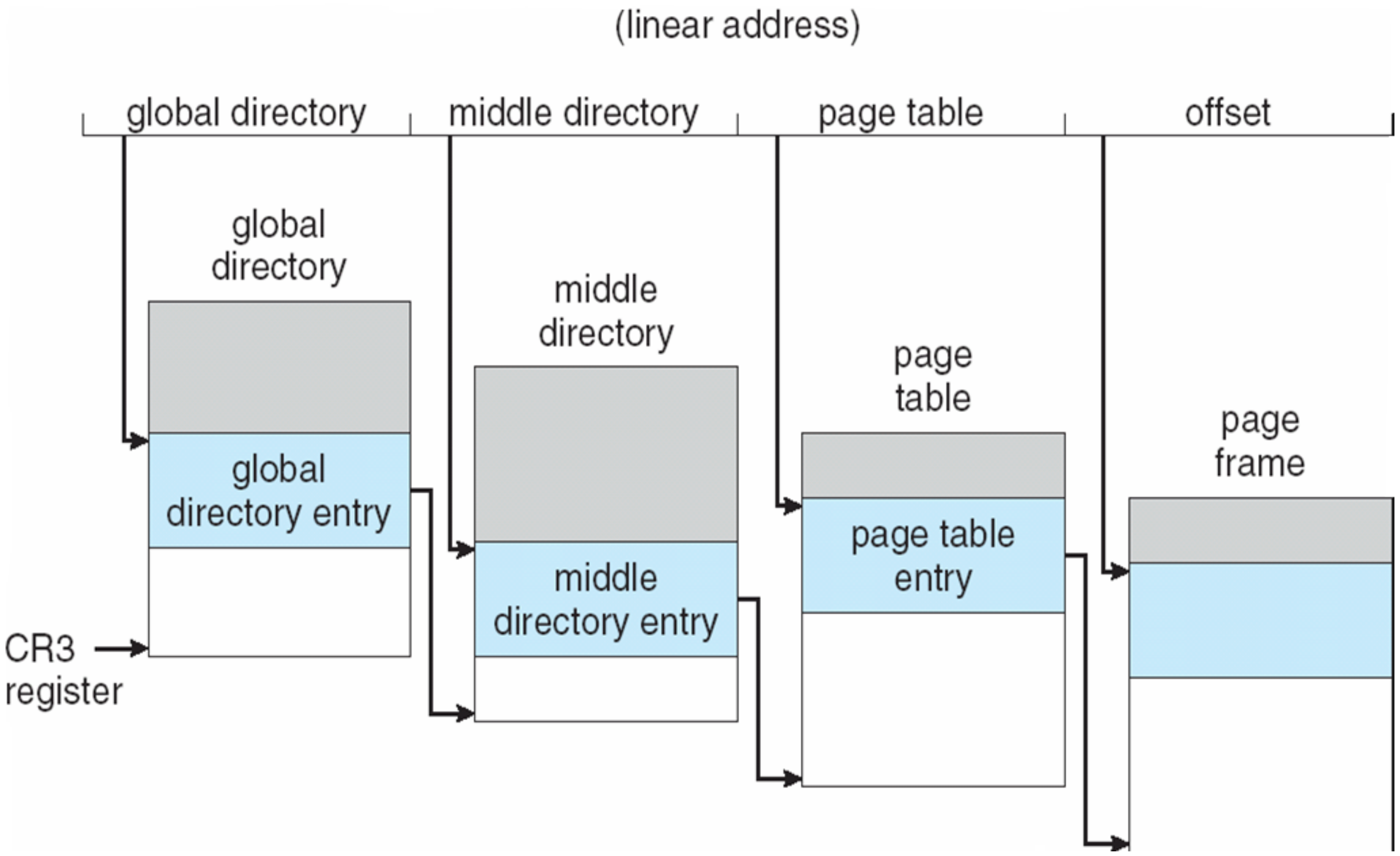
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



Linux Support for Intel Pentium

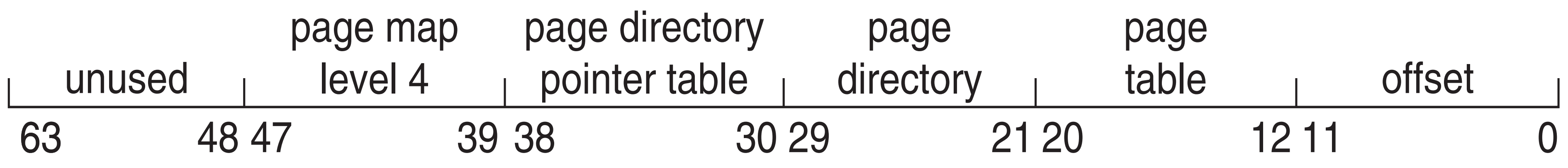
- Linux uses only 6 segments
    - kernel code, kernel data, user code, user data
    - task-state segment (TSS), default LDT segment
- Linux only uses two of four possible modes
    - kernel: ring 0, user space: ring 3
- Uses a generic four-level paging for 32-bit and 64-bit systems
    - for two-level paging, middle and upper directories are omitted
    - older kernels have three-level generic paging

Three-level Paging in Linux



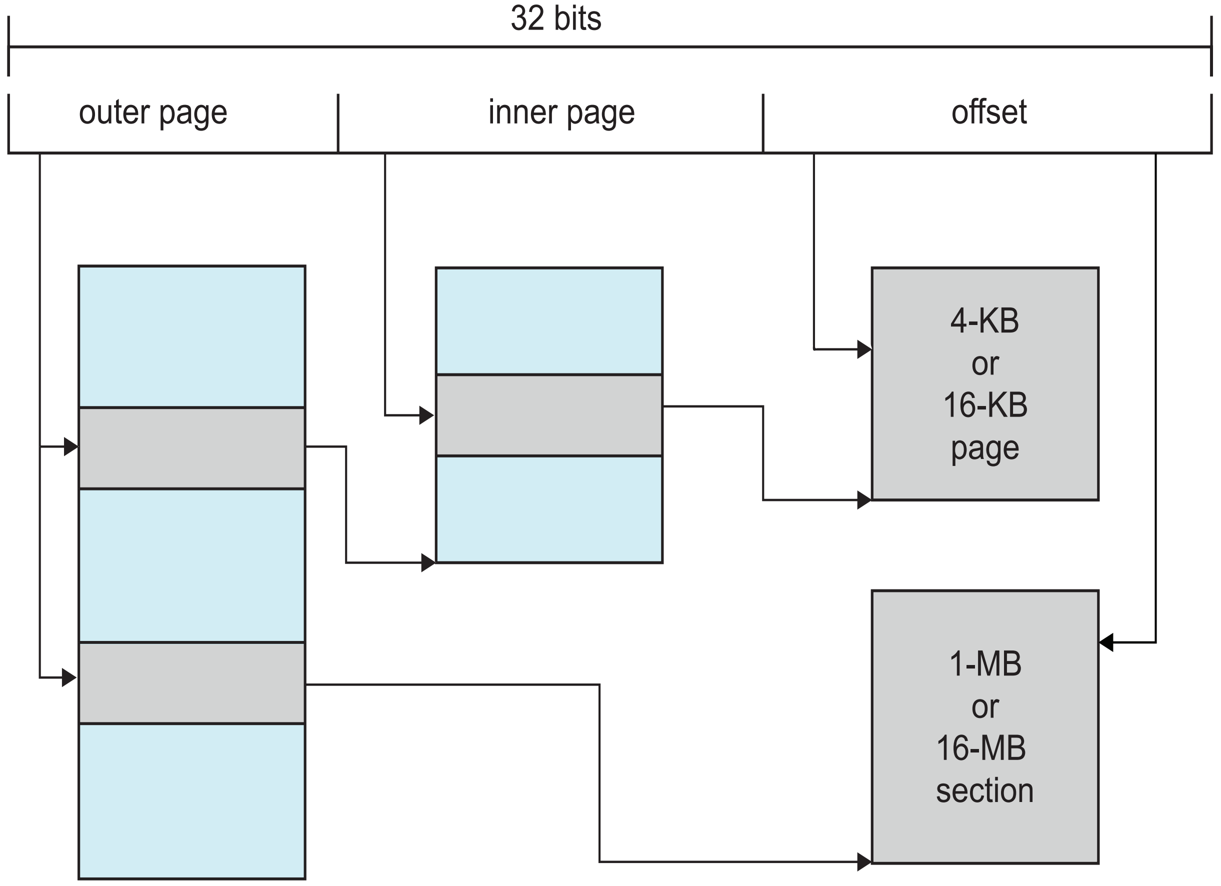
Intel x86-64

- Current generation Intel x86 architecture
- 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
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



Example: ARM32 Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- 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 inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



HW9 is out