Memory Management

Politecnico di Torino

Department of Control and Computer Engineering

System Programming - Sarah Azimi

CAD & Reliability Group
DAUIN– Politecnico di Torino

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Background

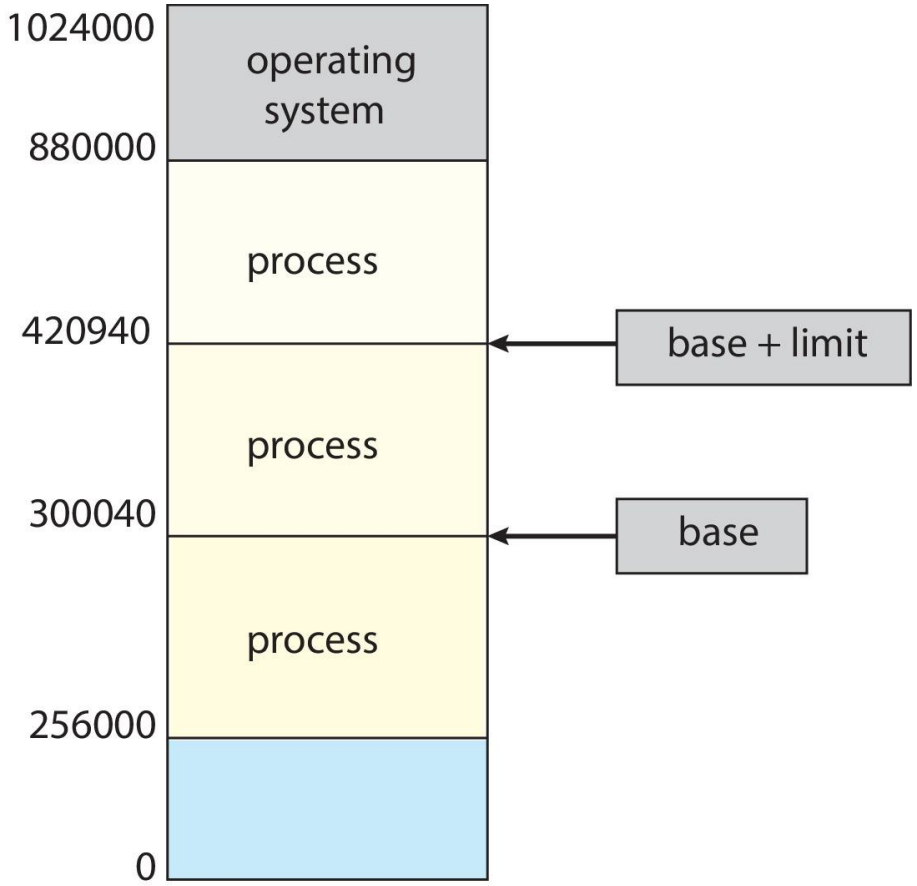
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- An instructions in execution and any data being used by them, must be in one of the direct access memory.
  - Register access is done in one CPU clock (or less)
  - Main memory can take many cycles, so the processor does not have the data to compete the instruction, 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.
- Protecting the processes from each other which is fundamental for having multiple processes for concurrent executions.
- We can provide this protection by using a pair of base and limit registers define the logical address space of a process

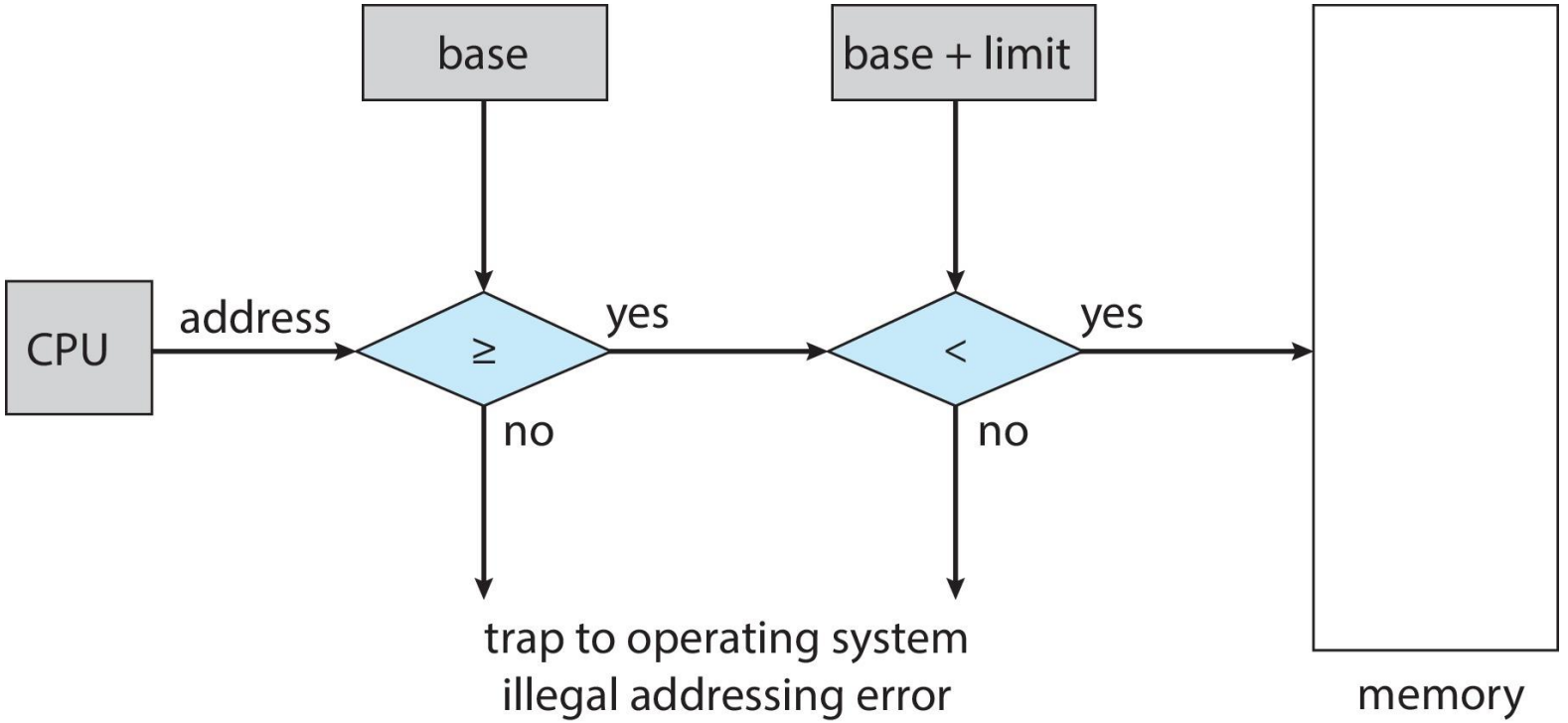
Protection

- To separate memory spaces, we need to determine the range of legal addresses that the process may access and to ensue that the process can accesses only these legal addresses.
- We can provide this protection by using a pair of base and limit registers define the address space of a process.
  - The base register specify the smallest legal physical memory addresses.
  - The limit register specifies the size of the range.



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



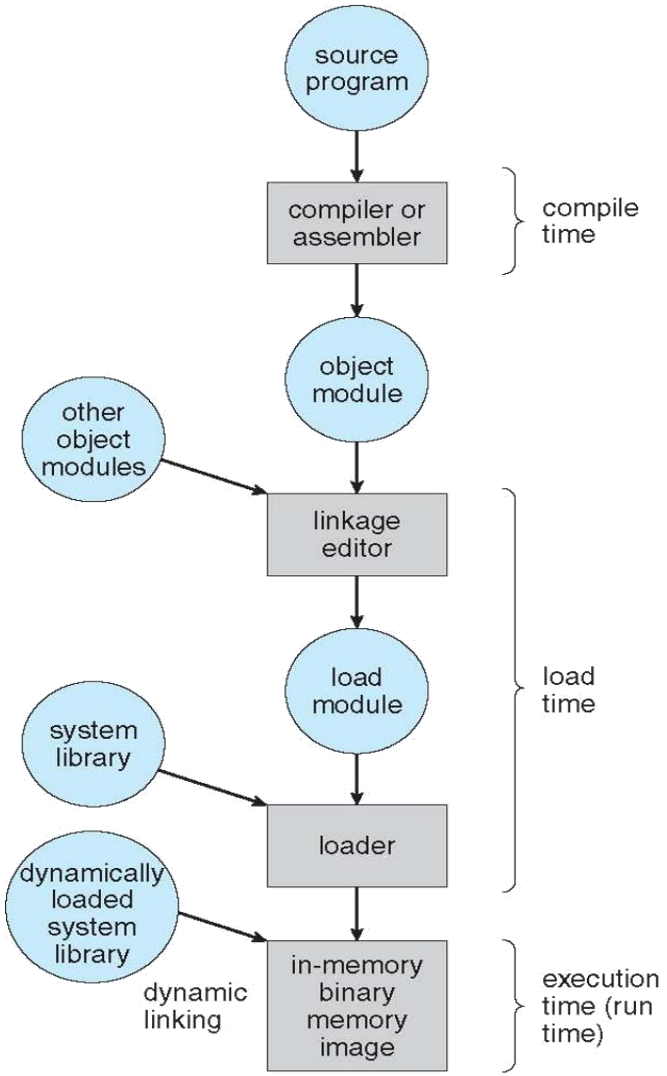
- The base and limit registers can be loaded only by the operating systems using a special privileged instructions.

Address Binding

- Usually, a program is on disk as a binary executable file.
- To run, it should be brought into memory, becoming eligible for execution on the CPU.
- Most systems allow a user process to stay in any part of the physical memory, for example at 0000.
- Addresses in the source program are generally symbolic.
- A compiler typically binds these symbolic addresses to relocatable addresses.
- Address binding **in an operating system refers to the process of assigning a memory address to a program or a process at the time of execution**.

Multistep Processing of a User Program

- Compile time, load time, and run-time are important stages in the life cycle of a program:
  - **Compile time**: This is the phase in which the source code of a program is converted into machine code. The machine code is an executable file that can be run on the target system.
  - **Load time:** After the executable file is created, it is loaded into memory when it is executed.
  - **Run time:** This is the phase in which the program is executed. During the run time, the program interacts with the operating system and other processes to perform its tasks.



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**: if memory location is not known at compile time, then the compiler must generate relocatable code time. In this case, the final binding is delayed until load time.
  - **Execution time**: if the process can be moved during its execution from one memory segment to another, then binding delayed until run time.
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

- An address generated by CPU is commonly referred to as a **logical address**, also referred to as **virtual address.**
- An address seen by the memory unit is referred to as **physical address**.
- Binding addresses at the compile or load time generated identical logical and physical addresses.
- The execution time binding results in differing logical and physical addresses.
  - **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

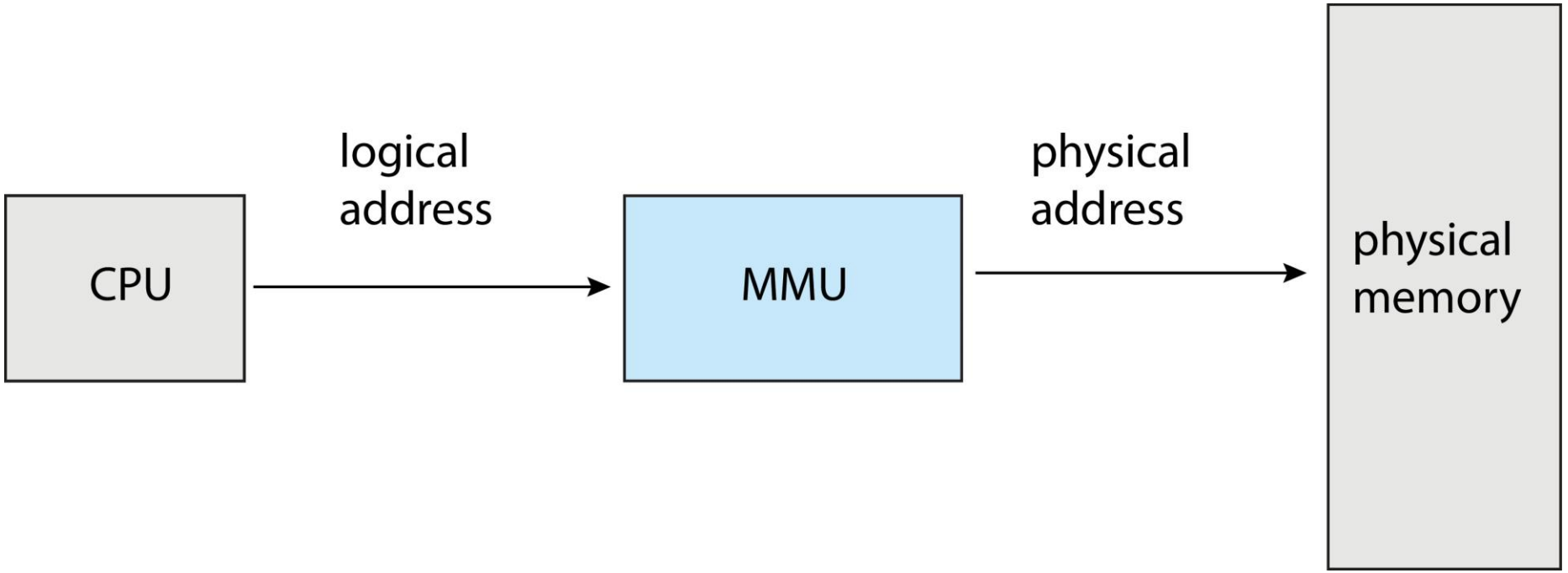
Logical vs. Physical Address Space

- An address generated by CPU is commonly referred to as a **logical address**, also referred to as **virtual address.**
- An address seen by the memory unit is referred to as **physical address**.
- Binding addresses at the compile or load time generated identical logical and physical addresses.
- The execution time binding results in differing logical and physical addresses.
  - **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

- The run-time mapping from the virtual to physical addresses is done by a hardware device called **Memory Management Unit (MMU)**

Memory-Management Unit (MMU)

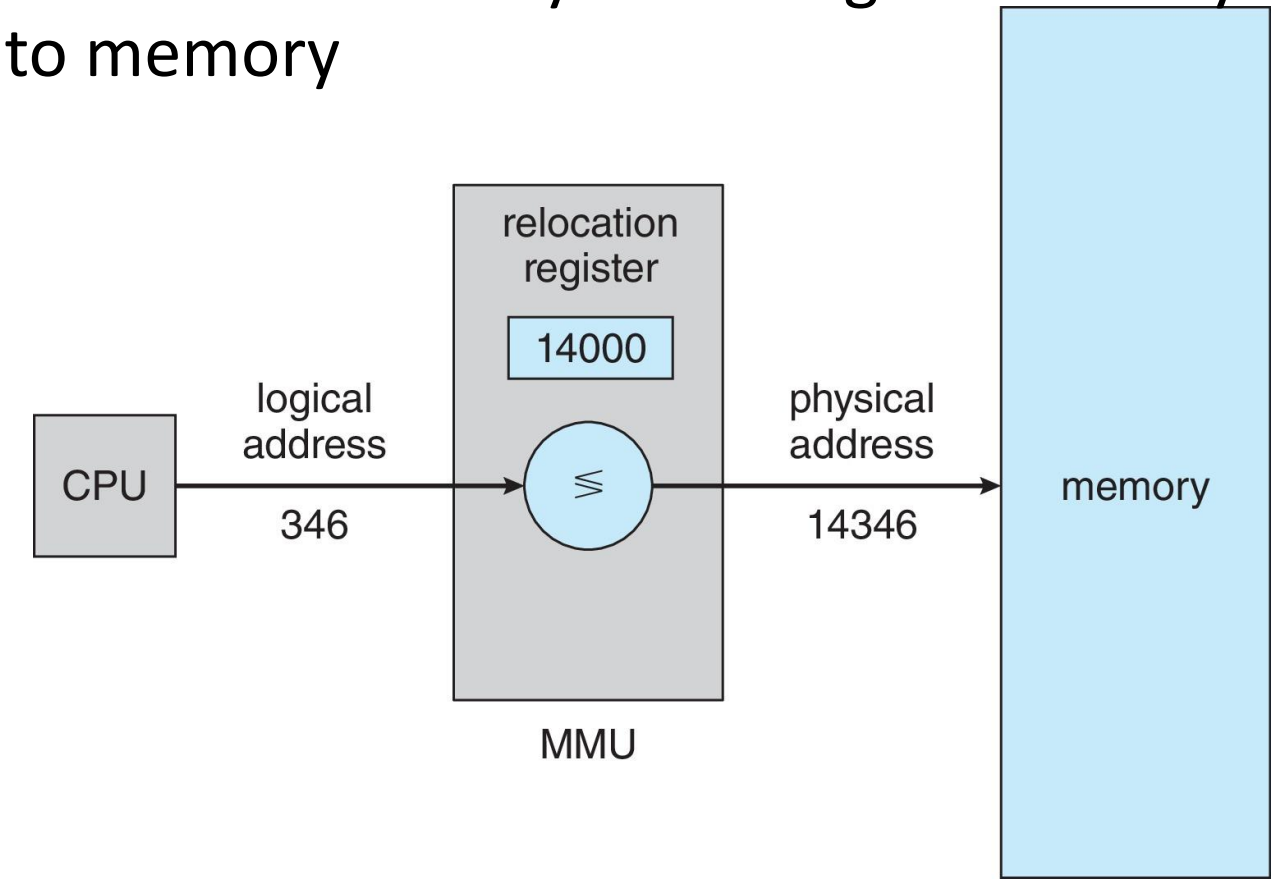
- Hardware device that at run time maps virtual to physical address



- 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



Dynamic Loading

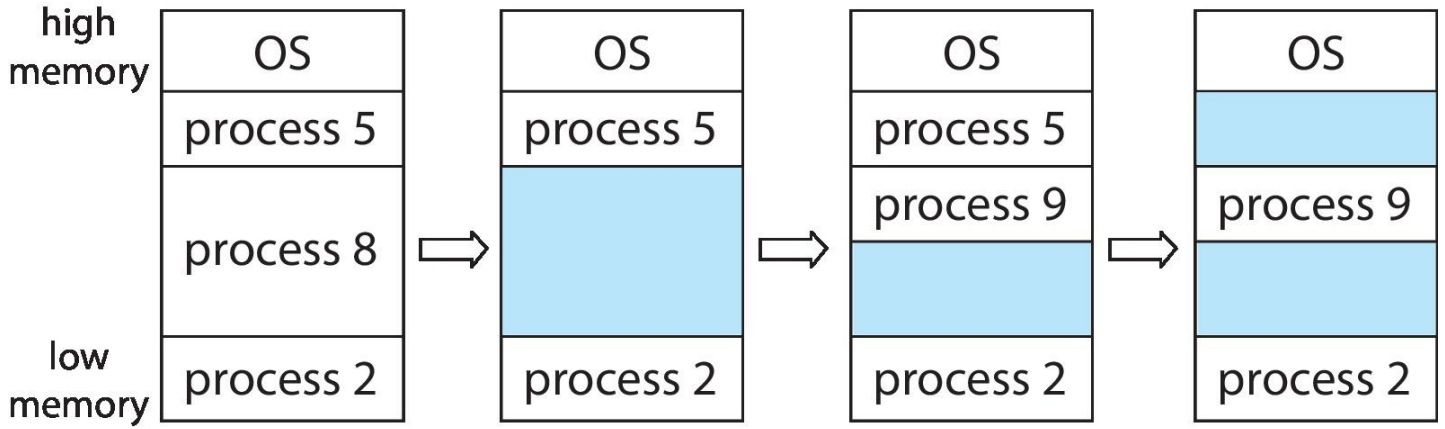
- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring 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
- 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

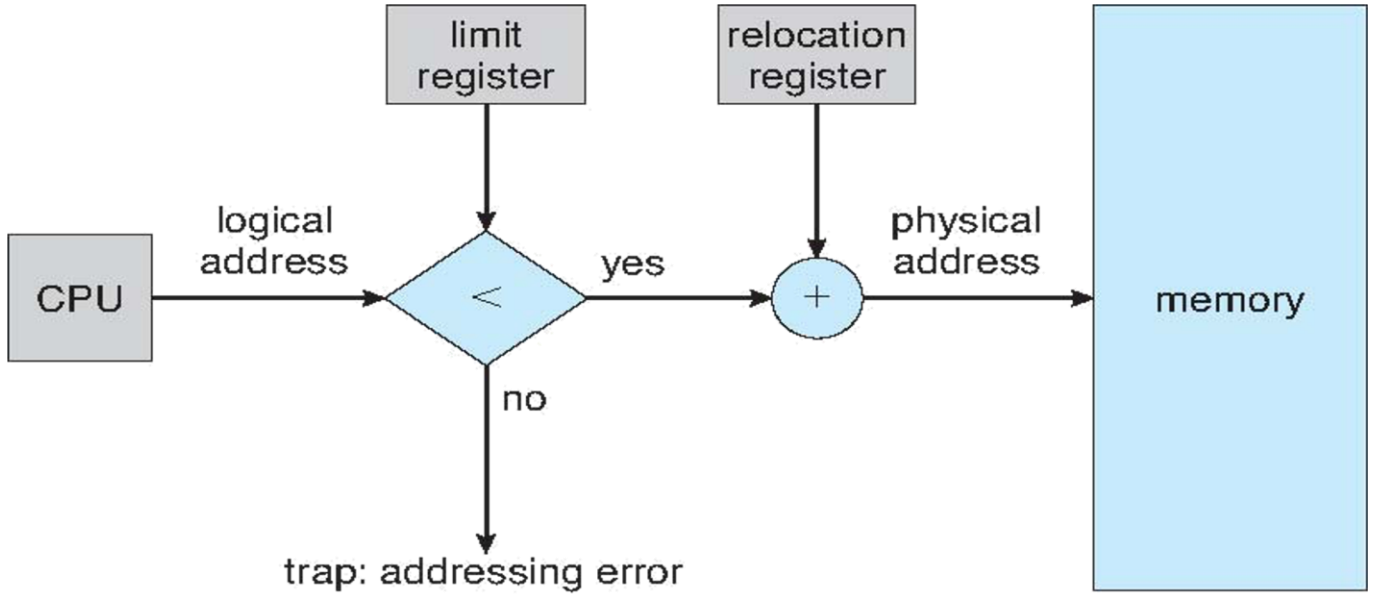
Contiguous Allocation

- Main memory must accommodate both OS and various user processes.
- Main memory usually into two partitions:
  - Placing the OS in either low memory addresses or high memory addresses.
  - Residing the user processes in memory at the same time.
- How to allocate the available memory to the processes?
- In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.



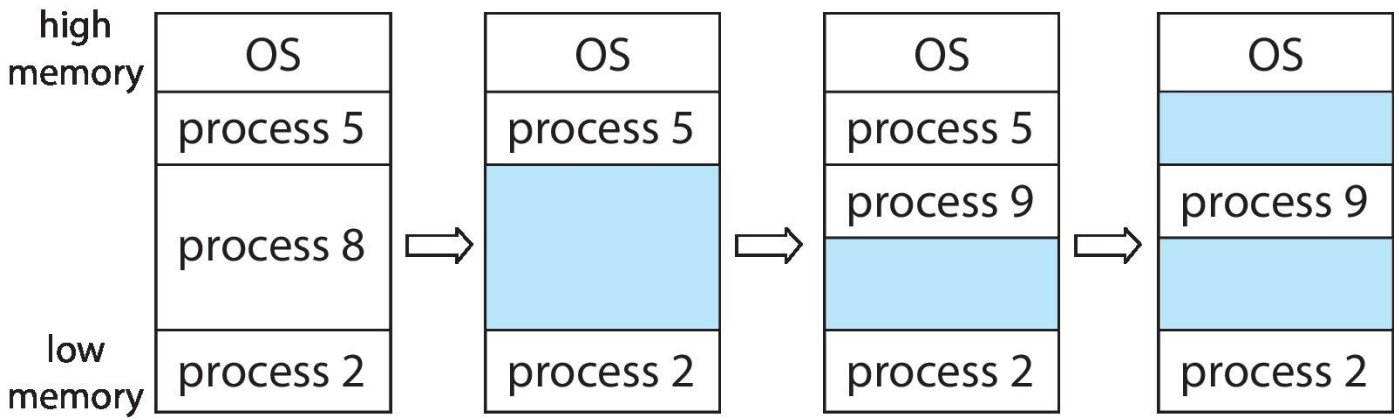
Contiguous Allocation (Cont.)

- 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*



Variable Partition

- One of the simplest method for allocating memory is to assign processes to variably sized partitions where each partitions may contain exactly one process **Variable partition** scheme
  - Initially, the memory is available for user processes, considered one large block.
  - Eventually, when the process terminates, it releases its memory, creating a **hole**.
  - When a new process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about: allocated partitions and free partitions (hole)



Dynamic Storage-Allocation Problem

- When a process arrived and needs memory, the system searches for a hole that is large enough for this process.
- How to satisfy a request of size n from a list of free holes – **dynamic storage allocation problem**
  - **First-fit**: Allocate the ***first*** hole that is big enough
  - **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless the list is ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
    - Produces the largest leftover hole which might be more useful than the smaller leftover hole from best-fit
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Exercise:

- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358KB, 200KB, and 375KB (in order)?
- Answer:
  - First Fit: 115KB >> 300 KB partition, 500 KB >> 600 KB partition, 358 KB >> 750 KB partition, 200 KB >> 350 KB partition, 375 KB >> 392 KB partition.

Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- 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.
- 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 (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
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Paging

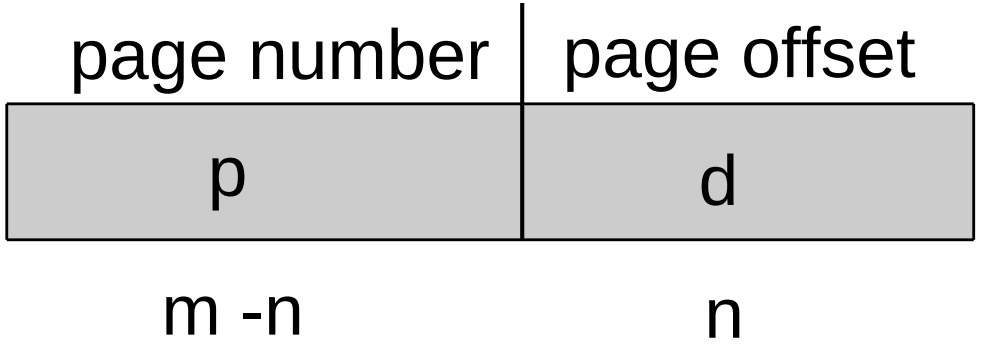
- Another possible solution to the external fragmentation problem is to permit the logical address space of processes to be **non-contiguous**.
- This is possible by adapting a strategy called **paging**.
  - Avoids external fragmentation
  - Needs for compaction
- Since paging offers numerous advantages, it is used in most operating systems, from those for large servers through those for mobile devices.
- Paging is implemented through cooperation between the operating system and the computer hardware.

Paging

- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, between 512 bytes and 16 Mbytes, depending on the computer architecture
- Divide logical memory into blocks of same size called pages
- When a process is to be executed, its pages are loaded into any available memory frames.
- Keep track of all free frames
  - It has great functionality, for example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

Address Translation Scheme

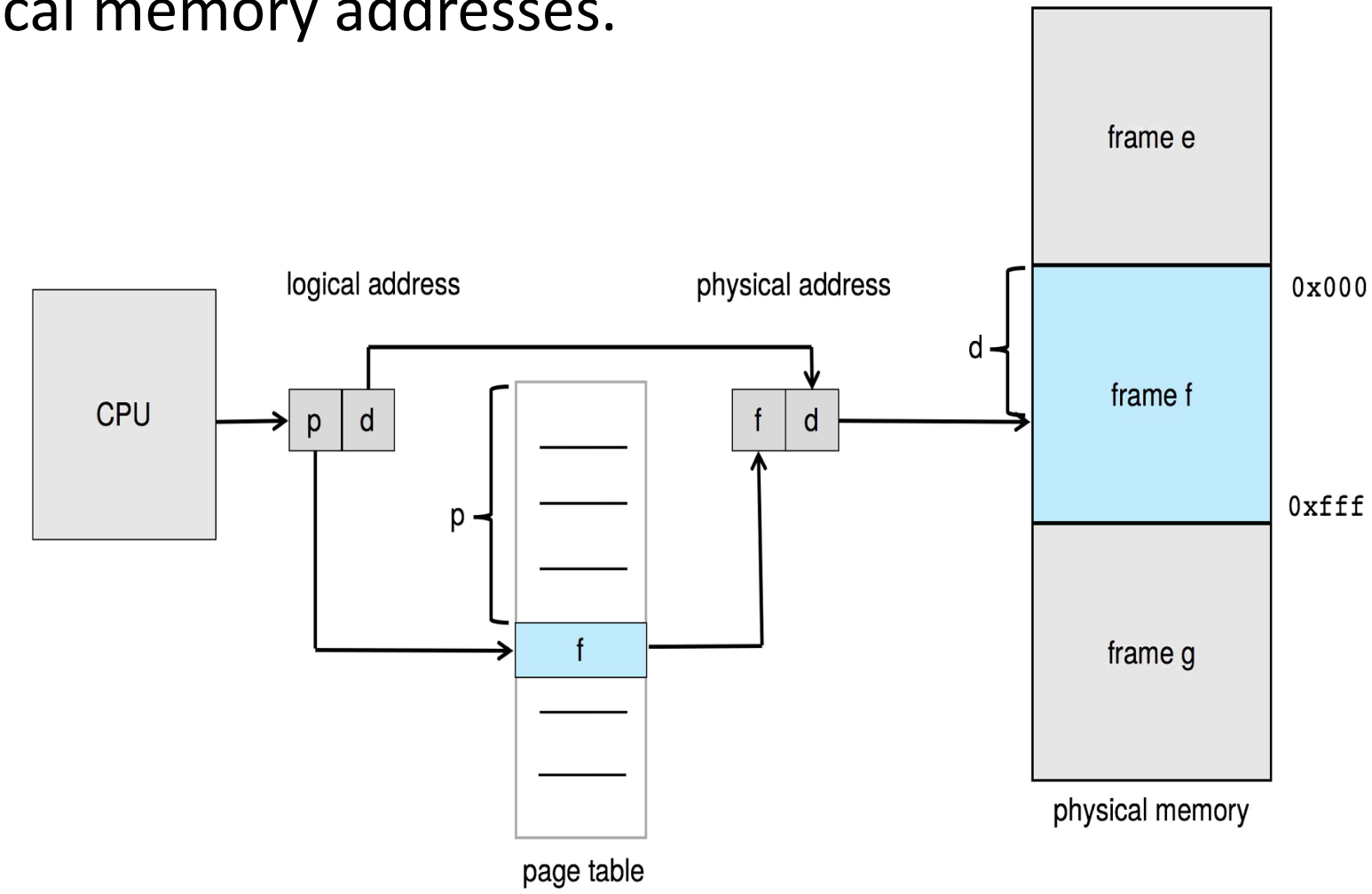
- Address generated by CPU is divided into:
  - Page number (*p*) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (*d*) – combined with base address to define the physical memory address that is sent to the memory unit



- For given logical address space 2^m and page size 2^n
- This information is used by Page Table

Paging Hardware

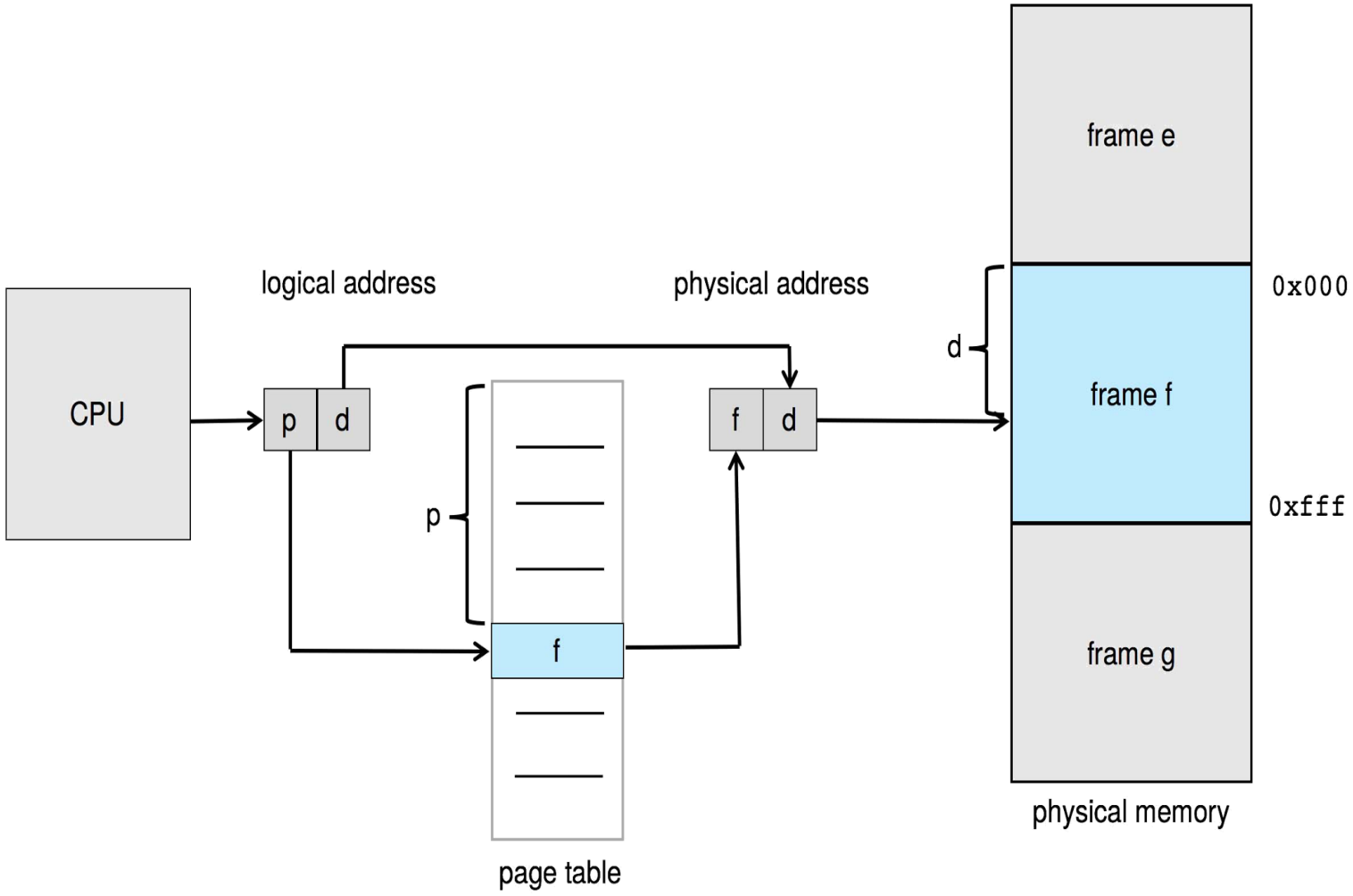
- The **page table** is a data structure used by the MMU to translate virtual memory addresses into physical memory addresses.
- The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being references.
- The based address of a frame is combined with the page offset to define the physical memory address.



Paging Hardware

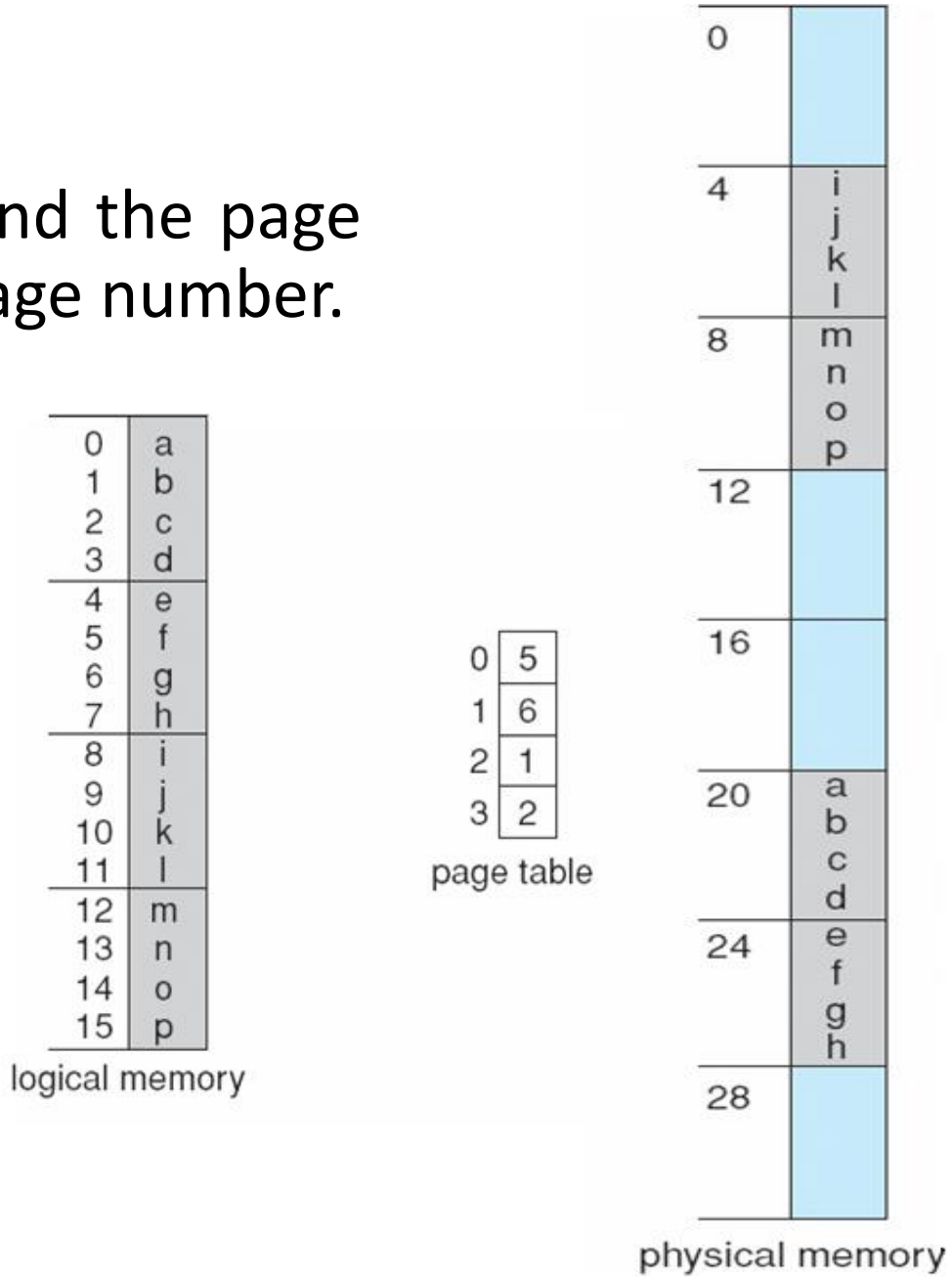
- The steps taken by the MMU to translate a logical address generated by CPU to a physical address:

1. Extract the page number P and use it as an index into the page table.
2. Extract the corresponding frame number F from the page table.
3. Replace the page number P in the logical address with the frame number.



Paging Example

- If the size of the logical address space is 2^m and the page size is 2^n bytes, then the 2^{m-n} is designate the page number.
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

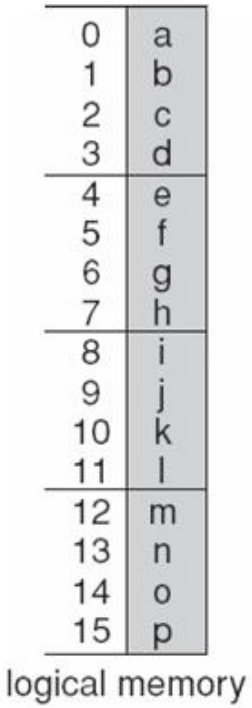


Paging Example

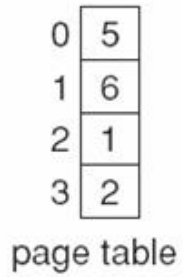
- If the size of the logical address space is 2^m and the page size is 2^n bytes, then the 2^{m-n} is designate the page number.
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).

Logical address 0 is page 0, offset 0. from page table, page 0 is in frame 5. Logical address 0 maps to physical address:

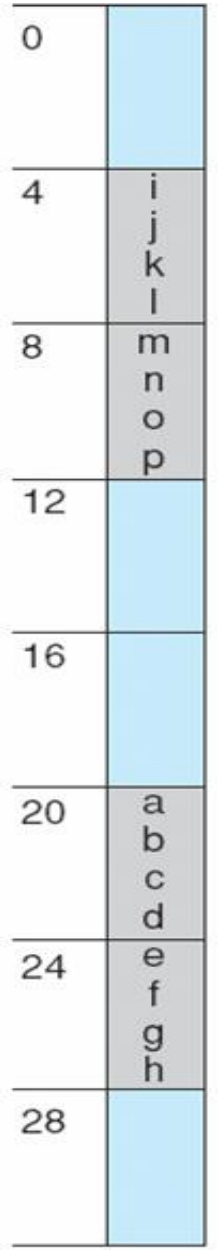
20 = (5 (*frame number*)*4 (*frame size*) + 0 (*offset*)



logical memory



page table



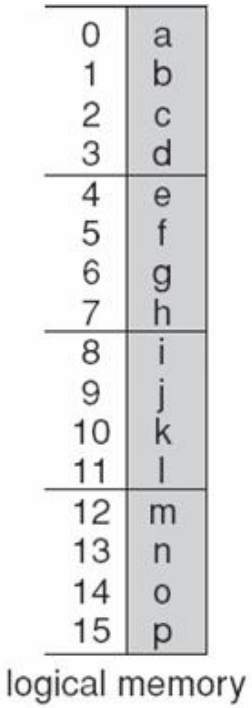
physical memory

Paging Example

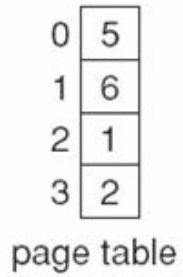
- If the size of the logical address space is 2^m and the page size is 2^n bytes, then the 2^{m-n} is designate the page number.
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).

Logical address 5 is in page 1, offset 1. from page table, page 1 is in frame 6. Logical address 5 maps to physical address:

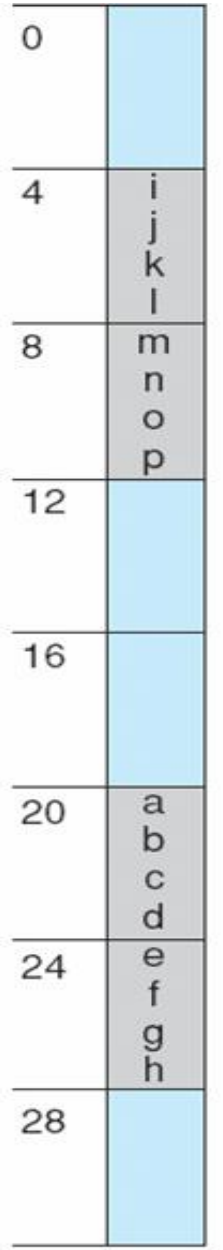
25 = (6 (*frame number*)*4 (*frame size*) + 1 (*offset*)



logical memory



page table



physical memory

Paging -- Calculating internal fragmentation

- With paging, we have no external fragmentation, but we may have internal fragmentation. For example, considering a case in which:
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - The process will need 35 pages + 1,086 bytes
  - It will be allocated 36 frames with an internal fragmentation of 962 (2,048-1,086)
  - Worst case fragmentation = 1 frame – 1 byte, almost an entire frame
- If the process size is independent of the page size, we expect internal fragmentation to average one-half page per process.
- So, small page size is suggested, but it will result in overhead in page table entry. This overhead is increasing as the page size is decreasing.

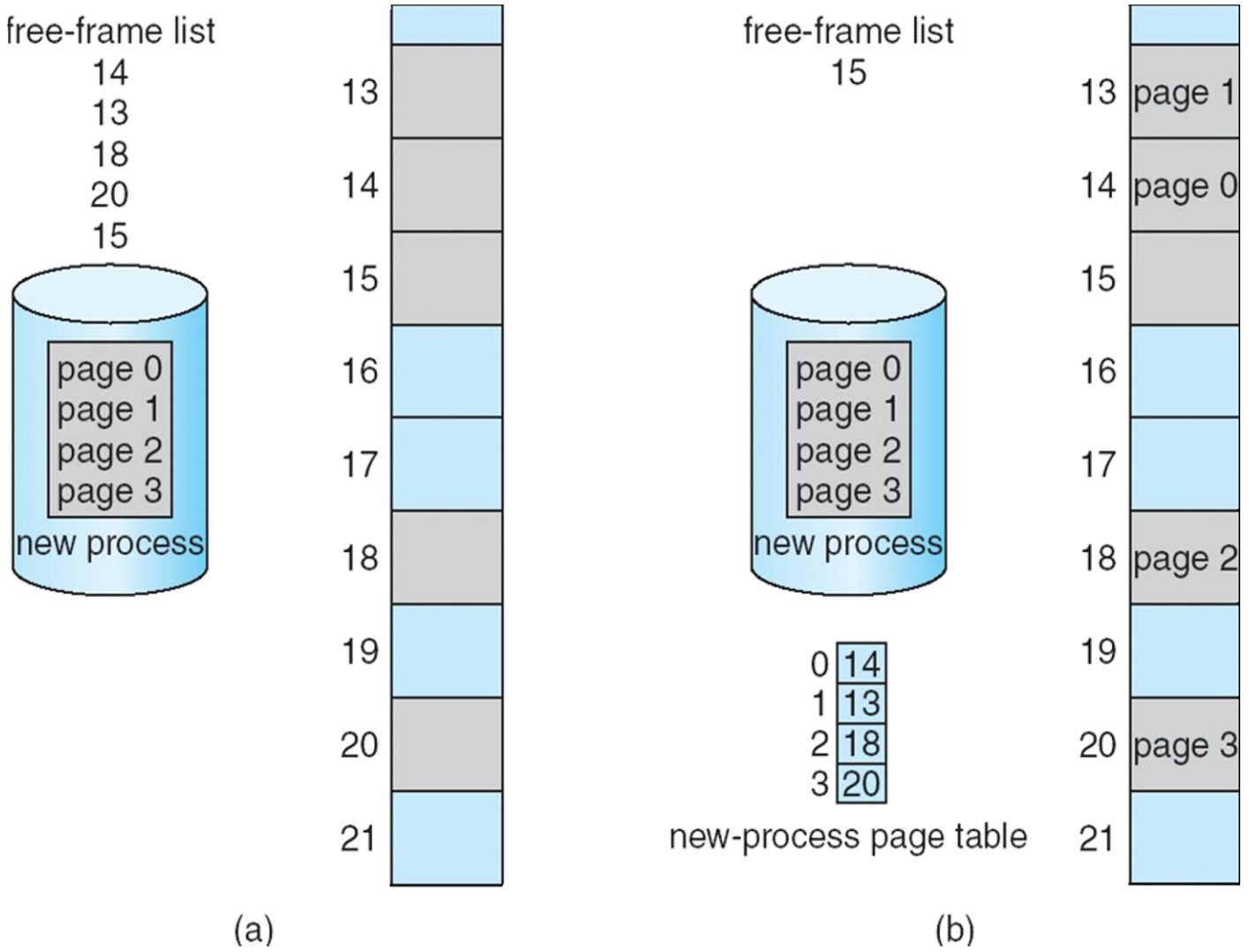
Exercise:

- Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
  - How many bits are there in the logical address?
  - How many bits are there in the physical address?
- Answer:
  - Logical Address: 16 bits
  - Physical Address: 15 bits

Exercise:

- Assuming a 1.KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
  - 3085
  - 42095
  - 215201
  - 650000
  - 2000001
- Answer:
  - Page = 3, offset = 13,
  - Page = 41, offset = 111,
  - Page = 210, offset = 784
  - Page = 1953, offset = 129

Free Frames



Before allocation

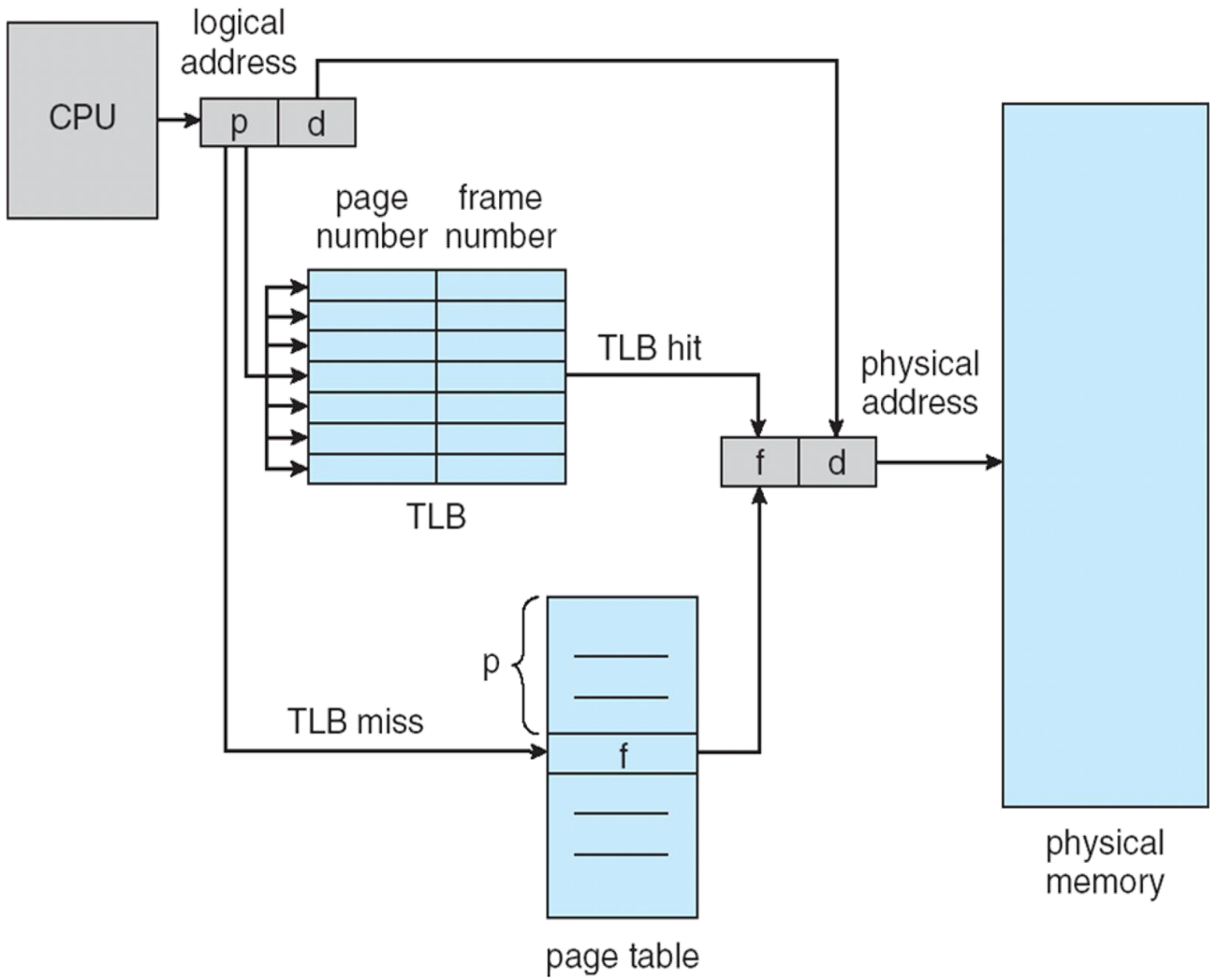
After allocation

Implementation of Page Table

- Page table is kept in main memory
  - Page-table base register (PTBR) points to the page table (Memory address of page table)
  - Page-table length register (PTLR) indicates the 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).

Paging Hardware With TLB

- The TLB contains only a few of the page-table entry.
- When a logical address is generated by the CPU, the MMU first checks if its page number is present in TLB.
- If the page number is found, its frame number is immediately available and used to access memory.
- These steps adds negligible performance penalty.
- If the page number is not in the TLB, **TLB miss**, address translation is done as before.



Translation Look-Aside Buffer

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be wired down for permanent fast access

Hardware

- Associative memory – parallel search

Page #	Frame #

Translation Lookaside Buffers

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

Figure 3-12. A TLB to speed up paging.

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

Effective Access Time

- The percentage of times that the page number of interest is found in the TLB is called the hit ratio.
  - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that it takes 10 ns to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - If we fail to find the page number in the TLB, then we must first access memory for the page table and frame number (10 ns) and then access to desired byte in memory (10 ns), for a total of 20 ns.
- Effective Access Time (EAT)

  $$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

  - Suffering from a 20% slowdown in average memory access

Effective Access Time

- The percentage of times that the page number of interest is found in the TLB is called the hit ratio.
  - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that it takes 10 ns to access memory.
- Effective Access Time (EAT)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds
  - Suffering from a 20% slowdown in average memory access
- 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.

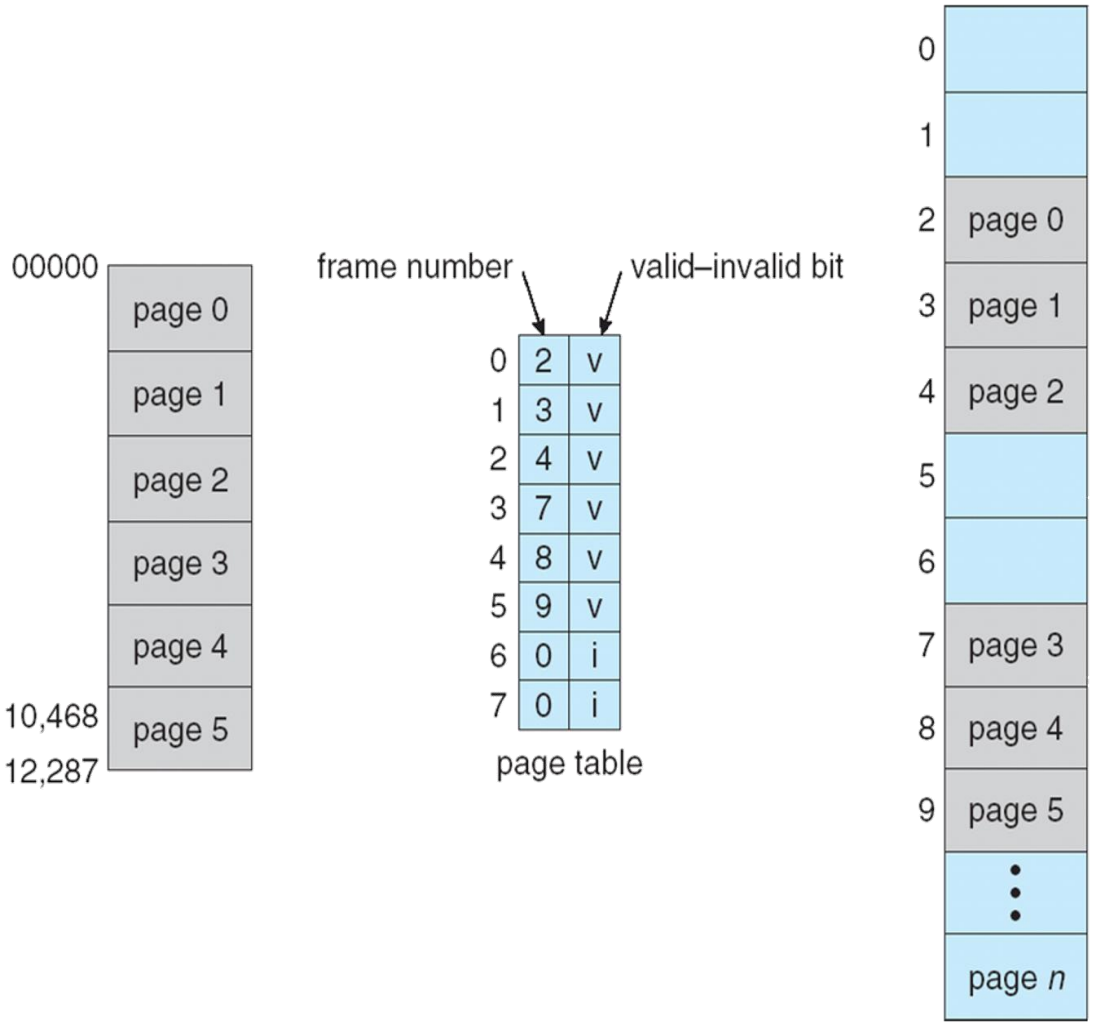
Memory Protection

- Memory protection in a paged environment is accomplished by protection bits associated with each frame.
- These bits are kept in the page table which can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.

Memory Protection

- One additional bit, Valid-invalid bit, is attached to each entry in the page table

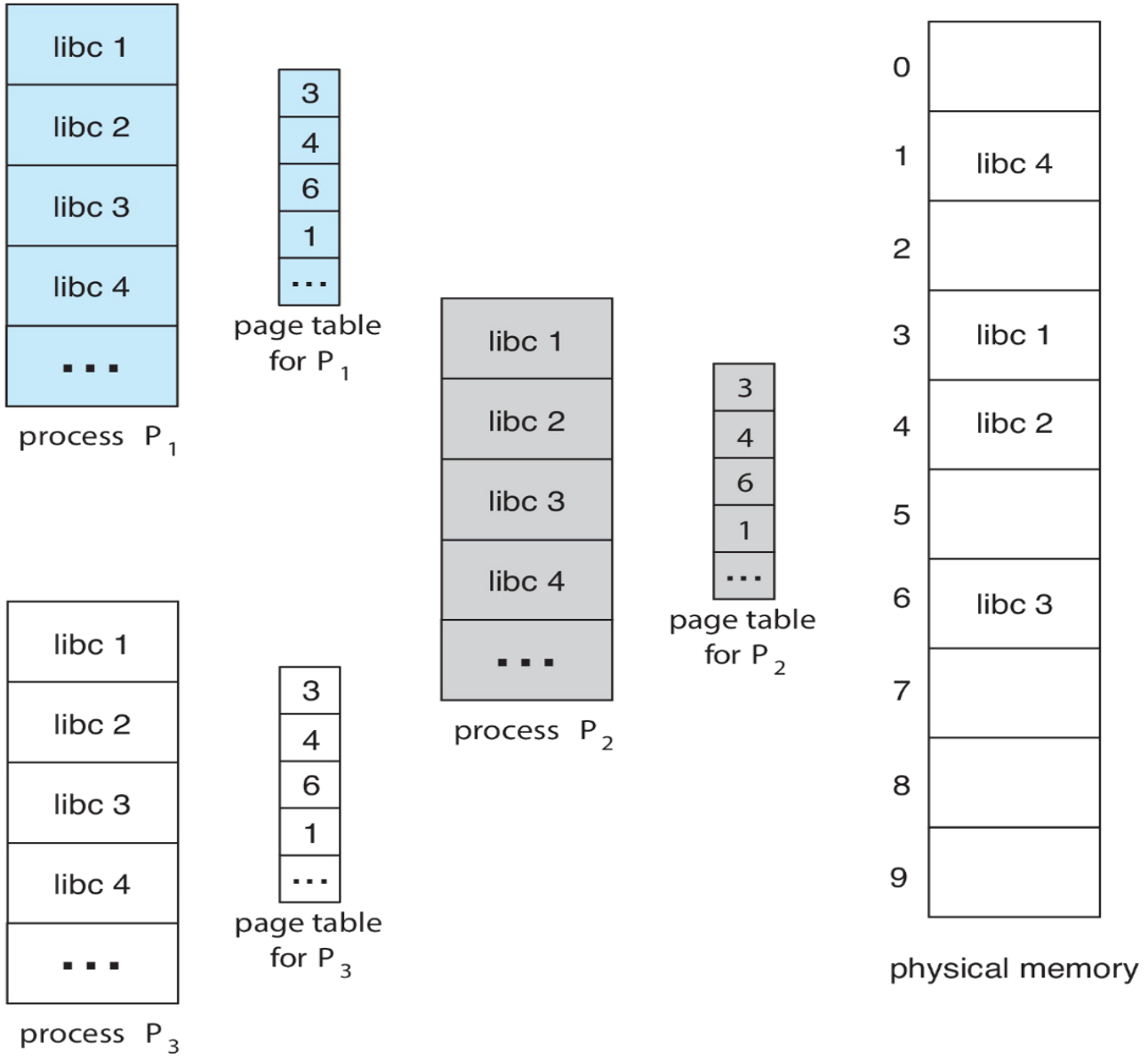
- When the bit is set to **valid**, the associated page is in the process's logical address and is legal
- When the bit is set to **invalid**, the page is not in the process's logical address space, illegal addresses are trapped.
- The operating system sets this bit for each page to allow or disallow access to the page.



Shared Pages

- An advantage of paging is the possibility of sharing common code which is really important in multiple processes environment.
  - If the code is **reentrant code**, it can be shared among processes.
  - Reentrant code is non-self modifying code, it never changes during execution., so more processes can share the same code at the same time.
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

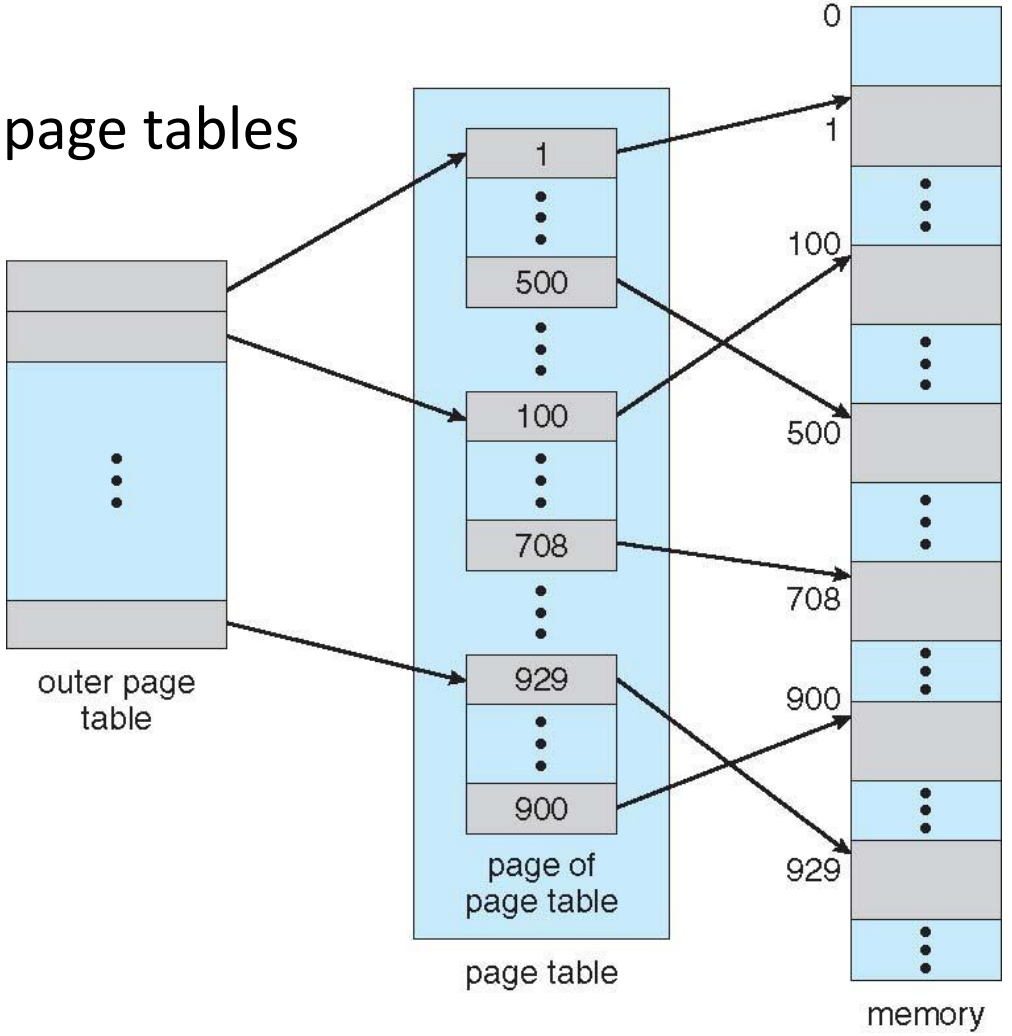


Structure of the Page Table

- Most modern computer systems support a large logical address space. Therefore, the page table itself becomes excessively large.
- Therefore, we would not want to allocate the page table continuously in main memory.
  - 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**

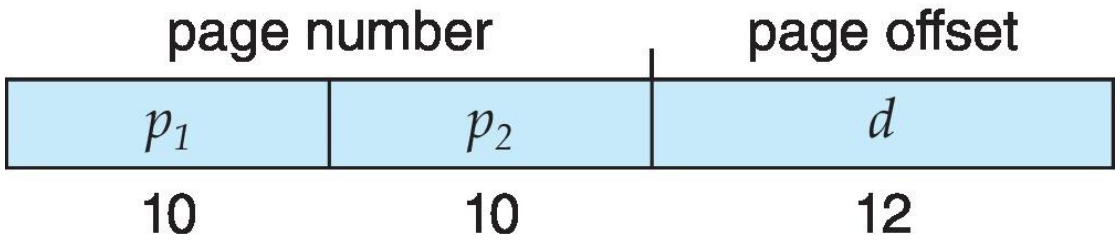
Hierarchical Page Tables

- One solution is to use a two-level paging algorithm, in which the page table itself is also paged.
  - 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



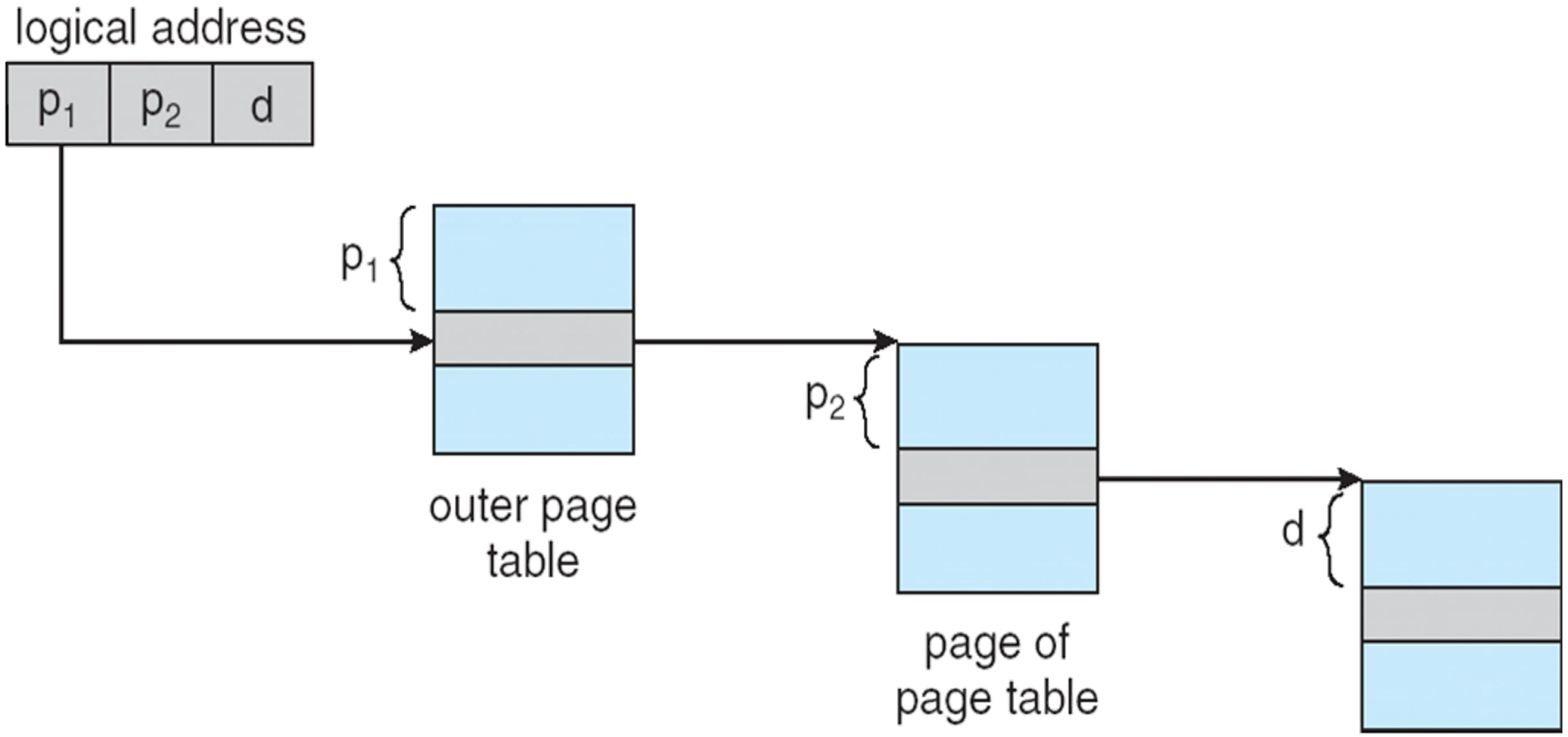
Two-Level Paging Example

- 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:



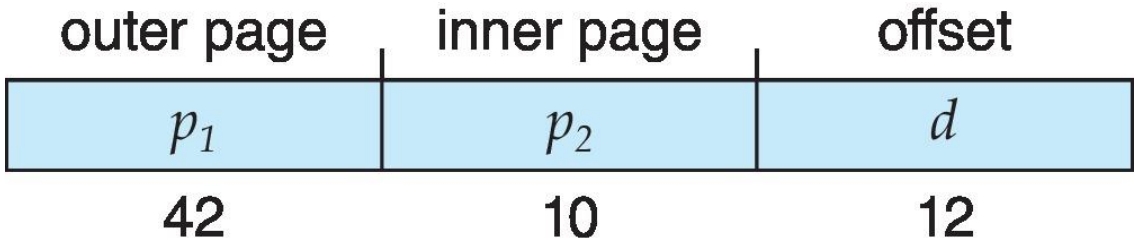
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as forward-mapped page table

Address-Translation Scheme



64-bit Logical Address Space

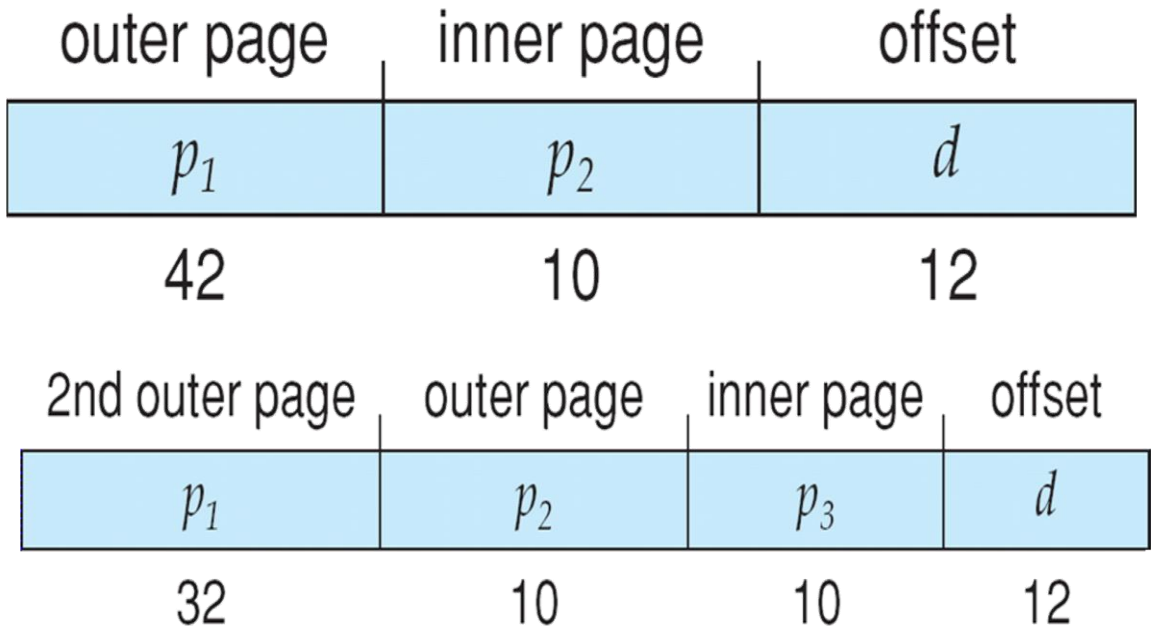
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} 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

- The way to avoid such a large table is to divide the outer page table into smaller pieces
- For example, we can page the outer page table, giving us a three-level paging scheme.

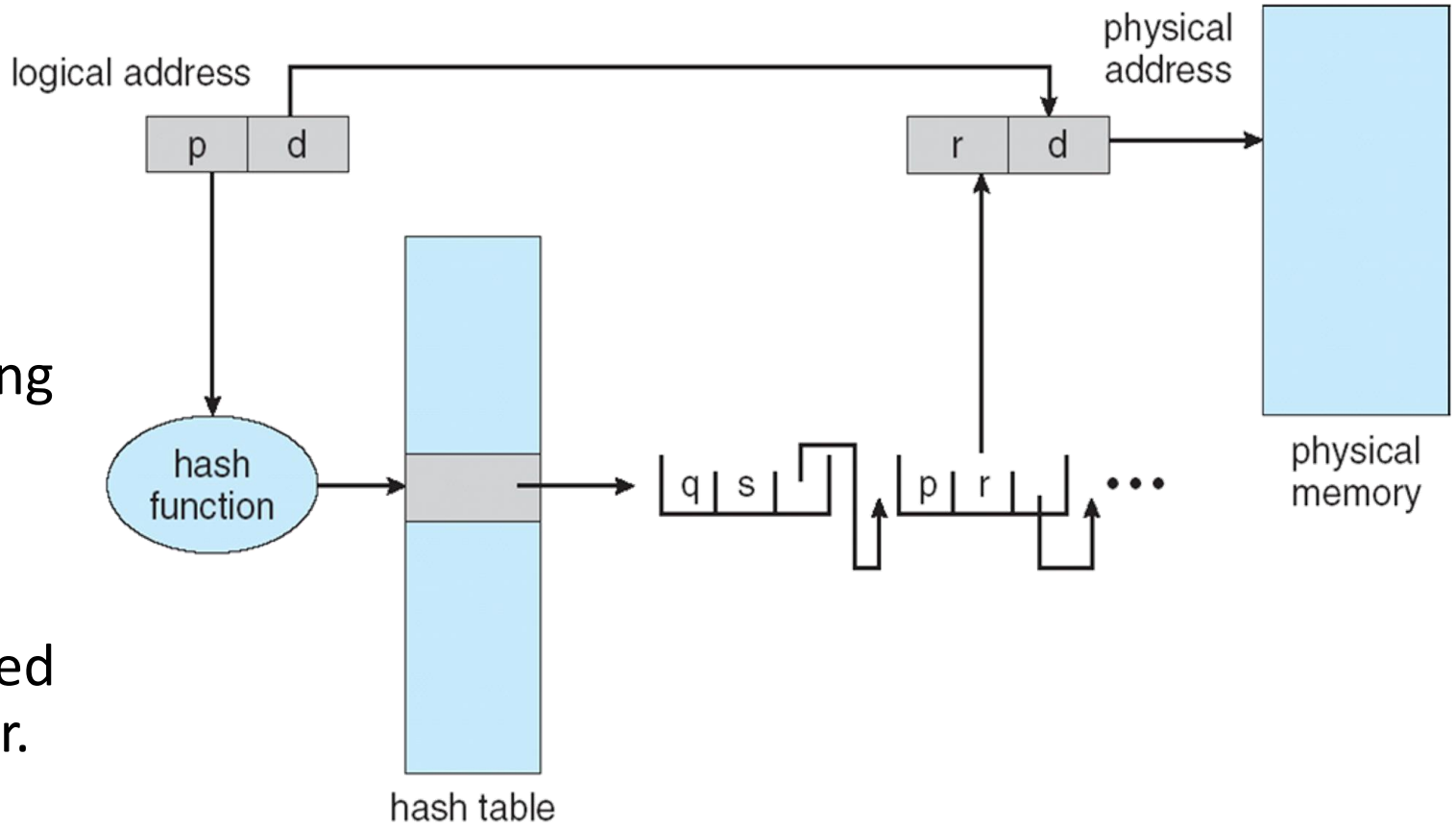


Hashed Page Tables

- One approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location.
- Each elements consists of three fields:
  1) The virtual page number
  2) the value of the mapped page frame
  3) a pointer to the next element in the linked list

Hashed Page Tables

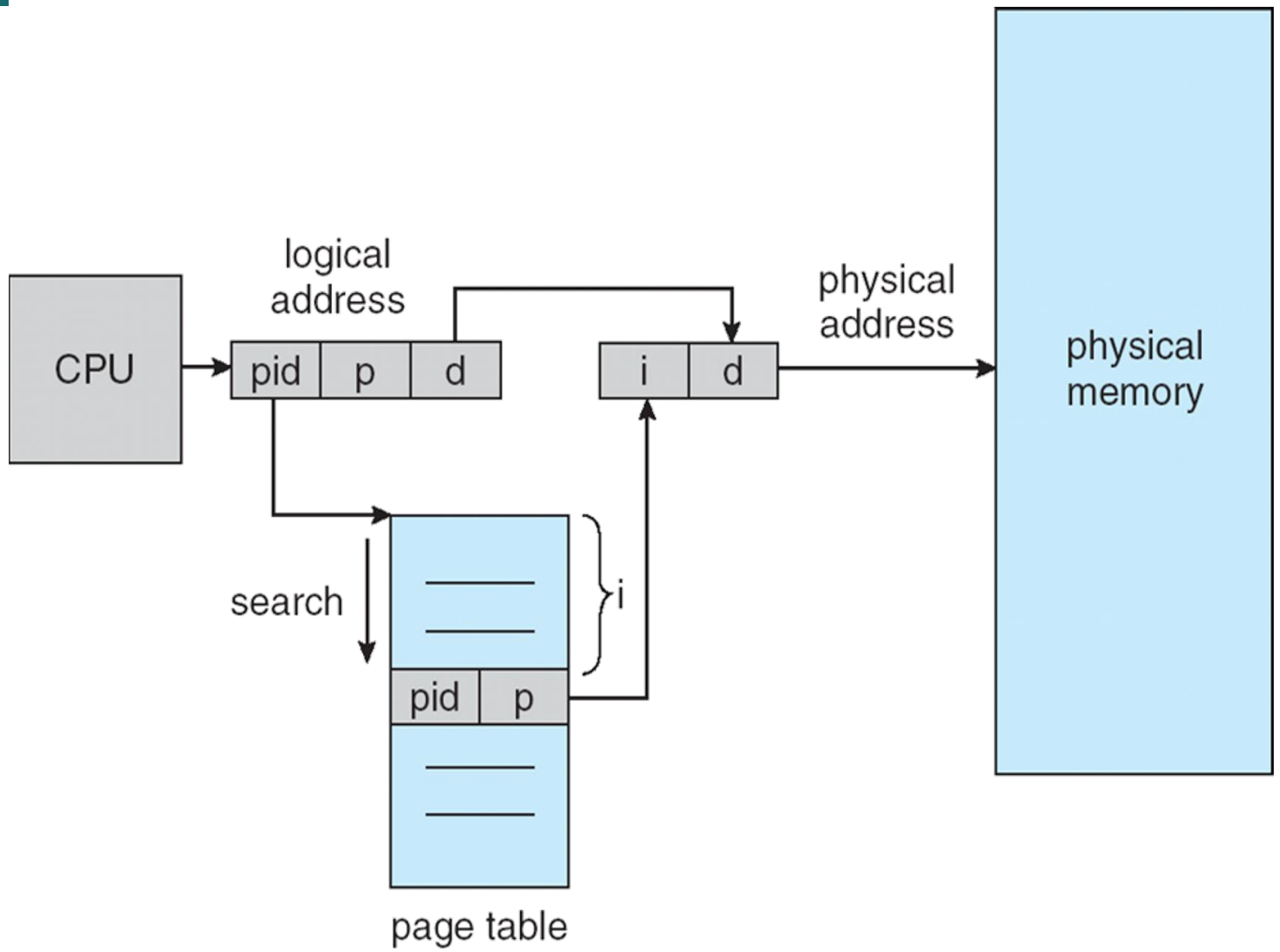
- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, he corresponding page frame is used to form the physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



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
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

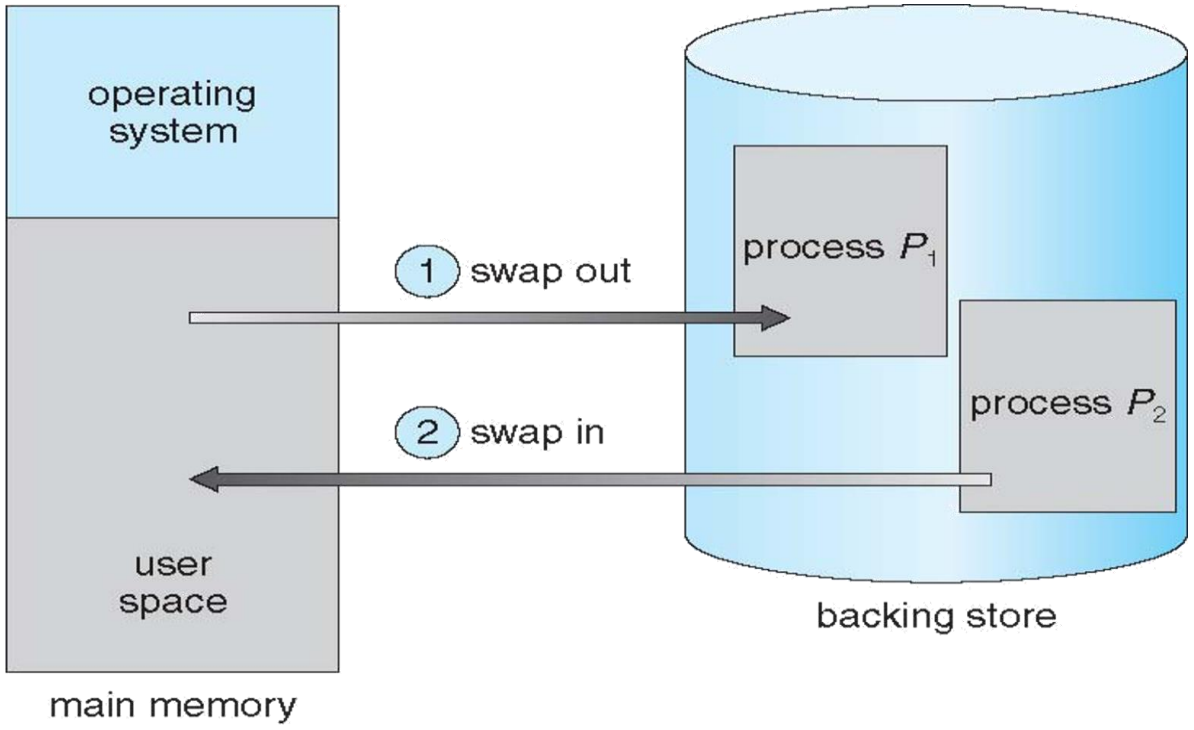
Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

Swapping

- Process instructions and data they operate on must be in memory to be executed.
- However, a process or a portion of a process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system.

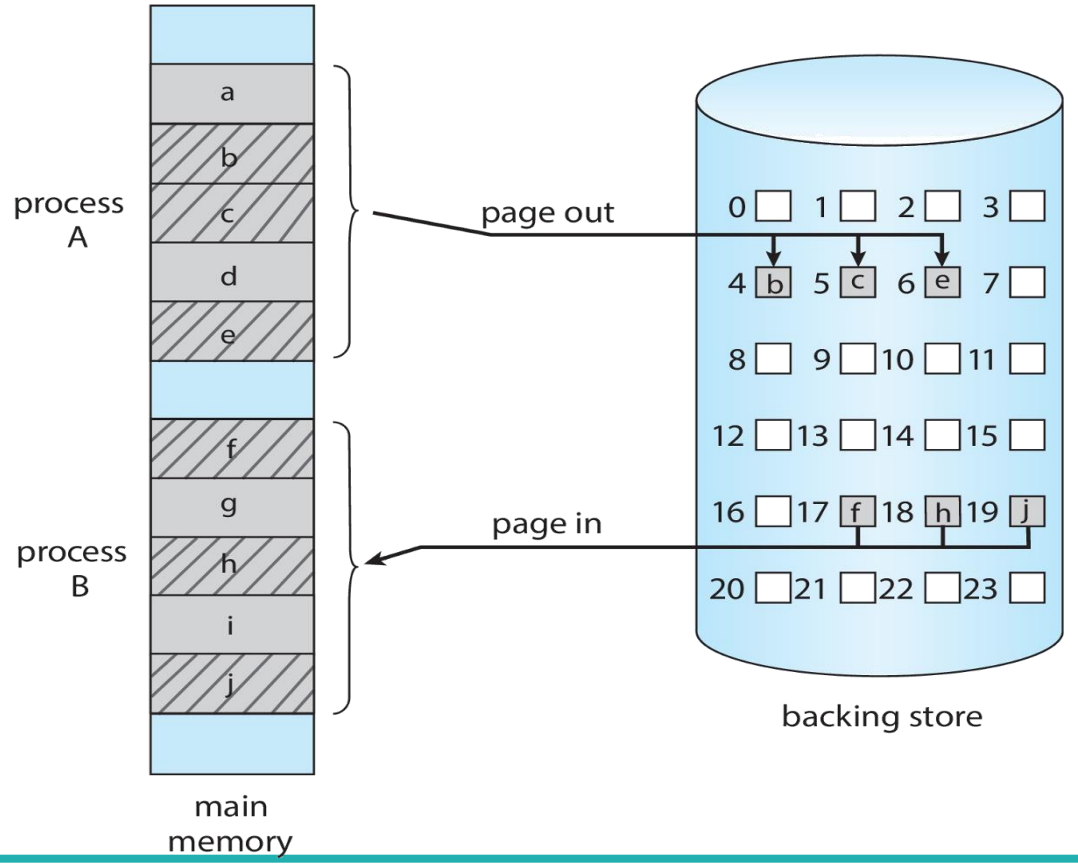


Standard Swapping

- Standard swapping involves moving entire processes between main memory and a backing store.
- The backing store must be large enough to accommodate whatever parts of the process needs to be stored and retrieved.
- When a process is swapped to the backing store, the data structures associated with the process must be written to the backing store.
- The advantages is that it allows physical memory to be oversubscribed, so the system can accommodate more processes than there is actual physical memory.

Swapping with Paging

- A problem with standard swapping is that the amount of time required to move entire process between memory and the backing store is expensive.
- Therefore, instead of the entire process, pages of a process can be swapped.
  - A **page out** operation moves a page from a memory to the backing stored.
  - The reverse process is known as **page in**



Swapping on mobile systems

- Typically, mobile systems do not support swapping.
  - Mobile devices generally use flash memory rather than more spacious hard disks with limited number of writes that it can tolerate before it becomes reliable.

- Instead of swapping, when free memory falls below threshold:
  - Apple iOS asks application to voluntarily relinquish allocated memory.
  - The application that fails to free up sufficient memory may be terminated by the operating system.
  - Android may terminate a process if insufficient free memory is available. However, before terminating, android writes it application state to flash memory so it can be quickly restared.

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
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- 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
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

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()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as double buffering, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Example: The Intel 32 and 64-bit Architectures

- Currently, the most popular PC operating systems are run on Intel chip such as Linux, windows, macOS.
- Intel architecture has not dominance to mobile systems where the ARM architecture currently is successful.
- 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
  - The IA-32 architecture allows a segment to be as large as 4GB
  - The maximum number of segments per process is 16K
  - The logical address space of a process is divided into two partitions:
    - First partition consists of up to 8 K segments are private to process, information about the first partition is kept in the local descriptor table (LDT).
    - Second partition of up to 8K segments shared among all processes, information about the second partition is kept in global descriptor table (GDT).

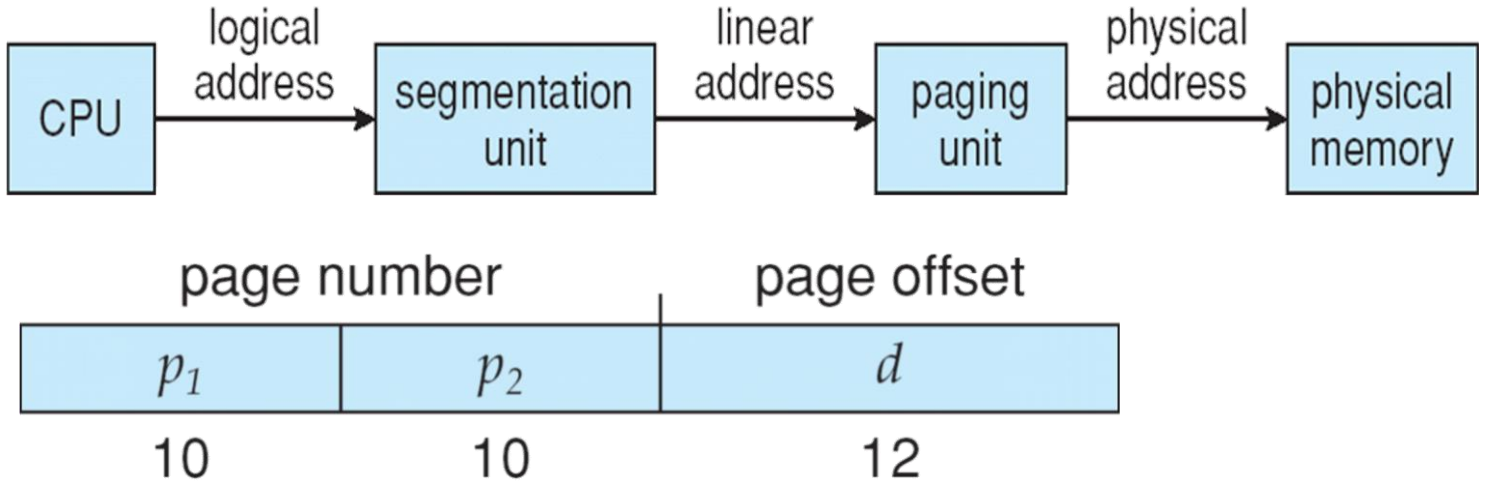
Example: The Intel IA-32 Architecture

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

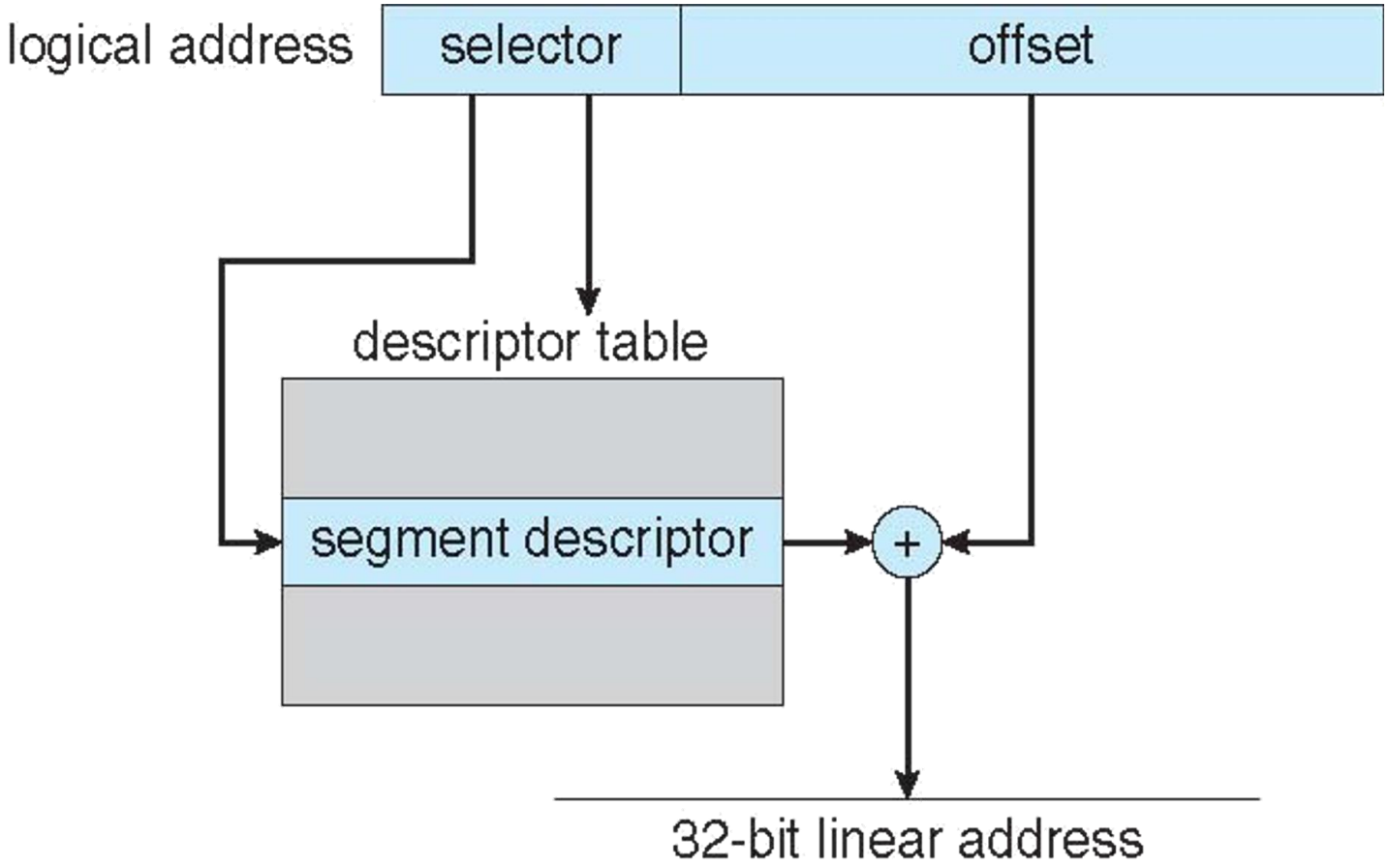
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - **Pages sizes can be 4 KB or 4 MB**

Logical to Physical Address Translation in IA-32

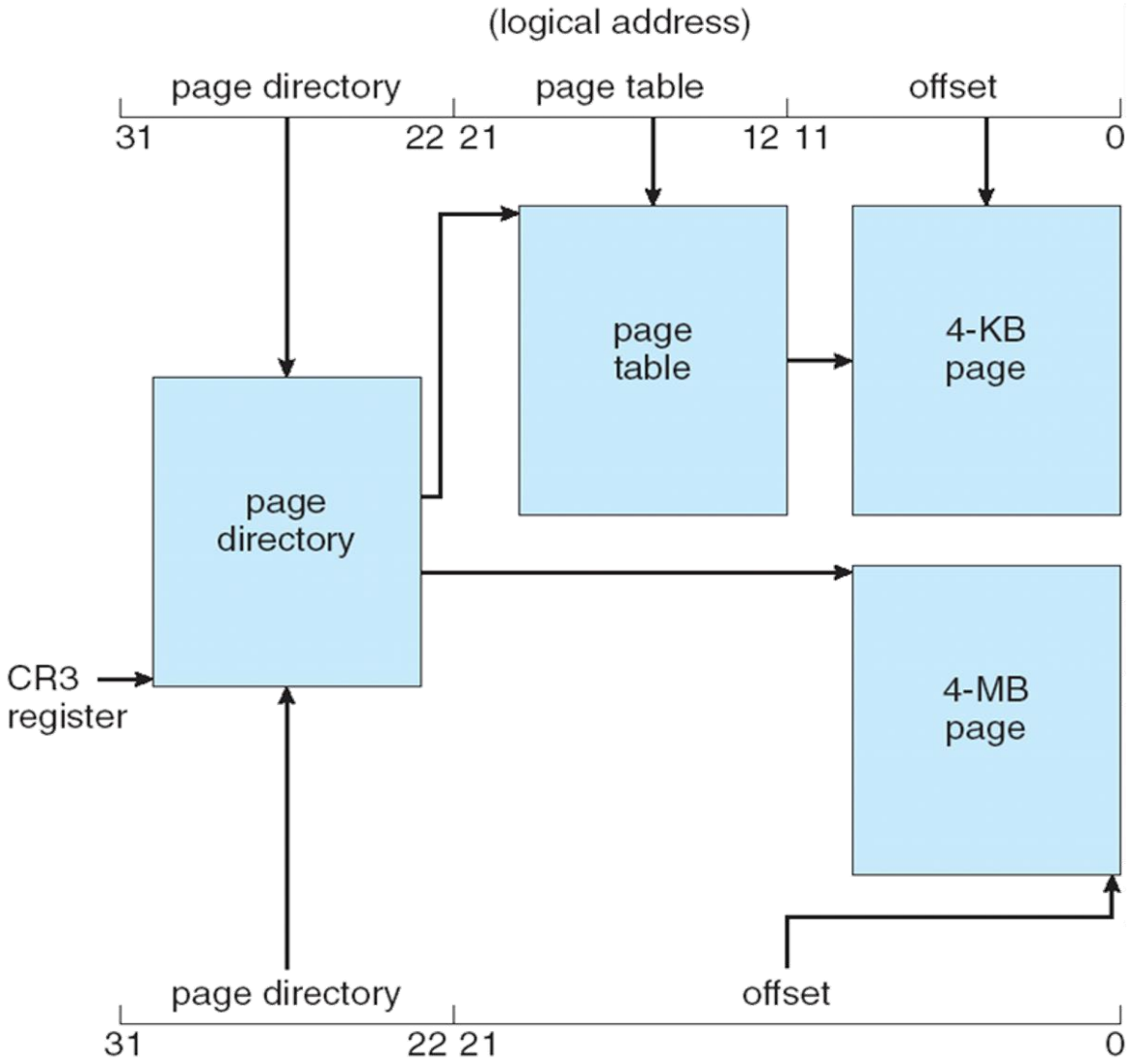
- The logical address pass through the segmentation unit, creating a linear address which will pass through the paging unit for being mapped on physical memory.



Intel IA-32 Segmentation

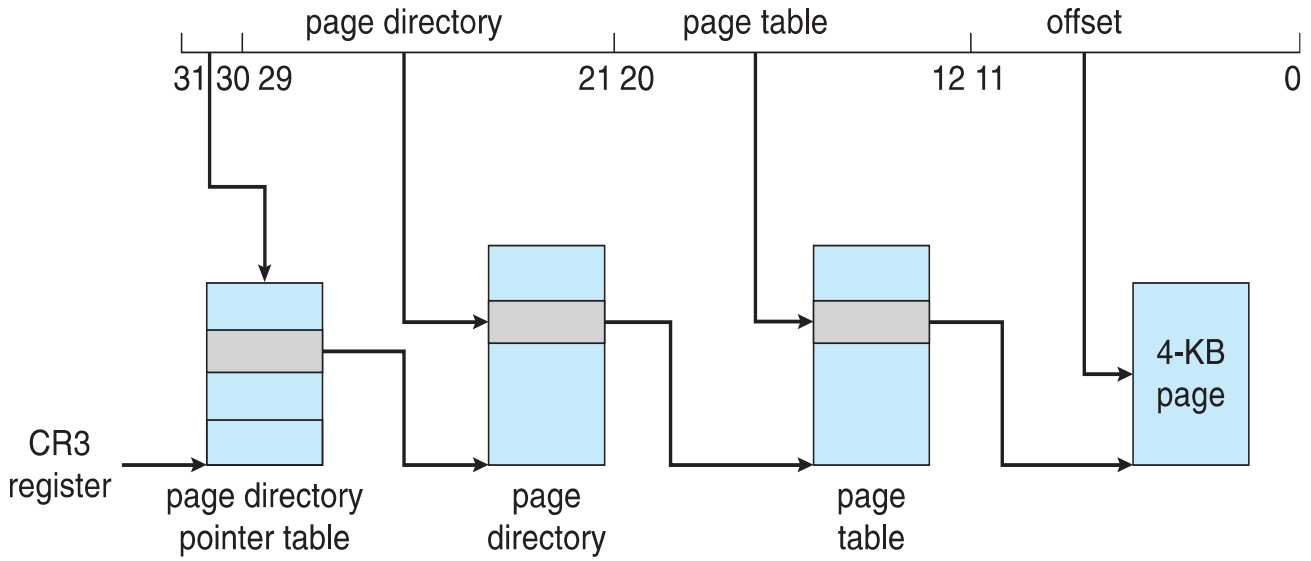


Intel IA-32 Paging Architecture



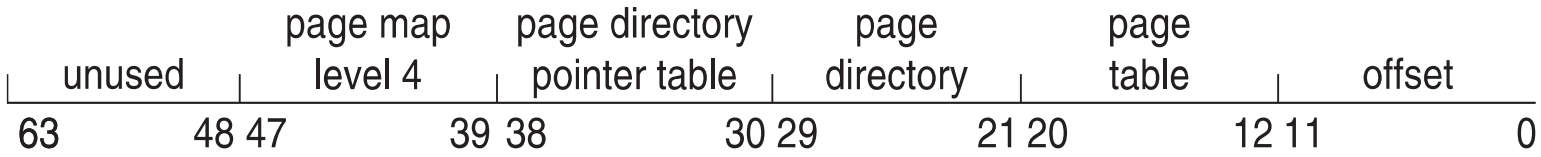
Intel IA-32 Page Address Extensions

- 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



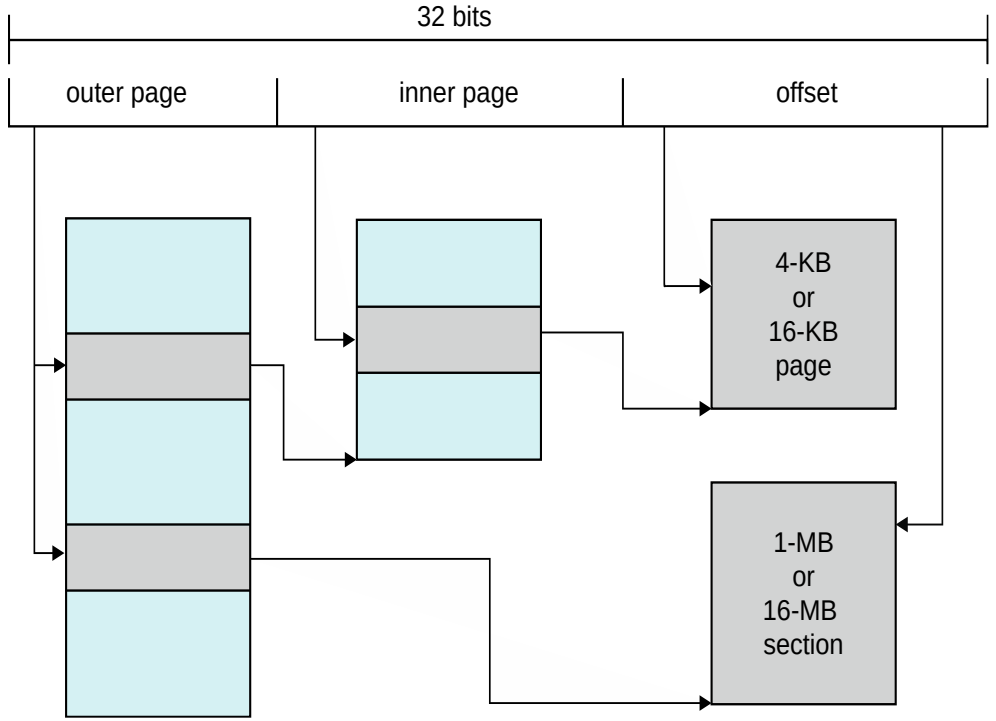
Intel x86-64

- Current generation Intel x86 architecture
- Support for a 64-bit address space yields an astonishing 264 bytes of addressable memory which is huge.
- 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: ARM 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)
- Up to four level of paging may be used
- 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



Thanks

Politecnico di Torino

Department of Control and Computer Engineering

Questions?

sarah.azimi@polito.it