

Operating Systems

12. Memory: Main Memory 2

Hyunchan, Park

<http://oslab.chonbuk.ac.kr>

Division of Computer Science and Engineering

Chonbuk National University

Contents

- Memory Management: Segmentation and Paging
- Structure of the Page Table
- Examples
 - The Intel 32 and 64-bit Architectures
 - ARM Architecture

Memory Management: Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments

- A segment is a logical unit such as:

main program

procedure

function

method

object

local variables, global variables

common block

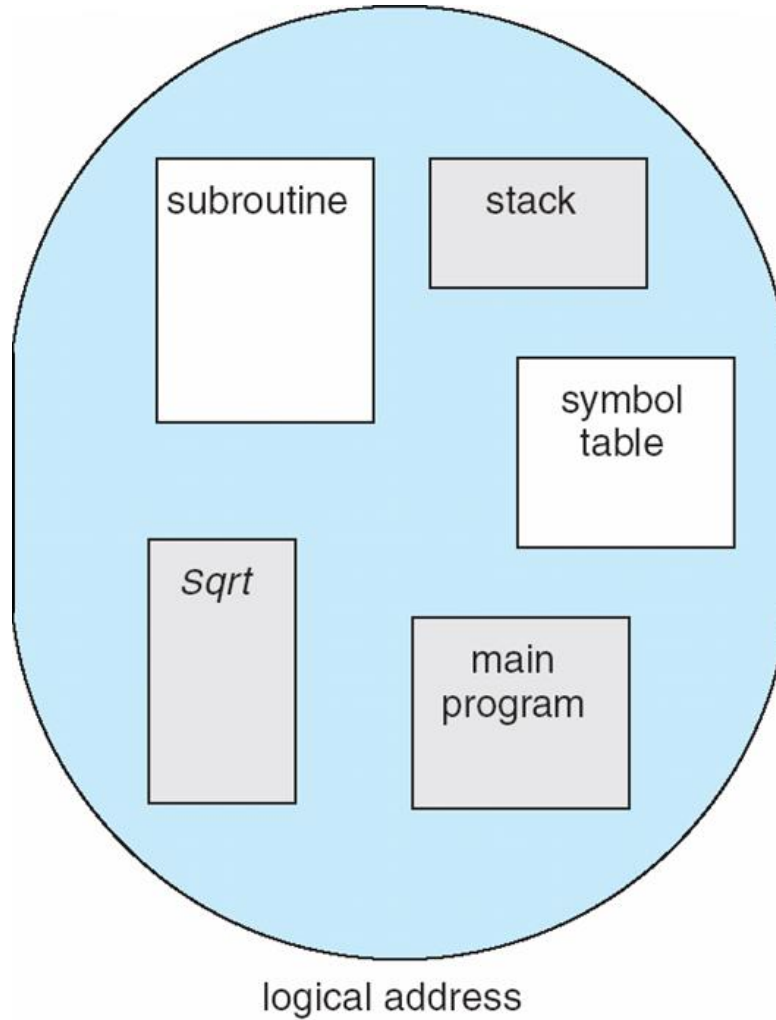
stack

symbol table

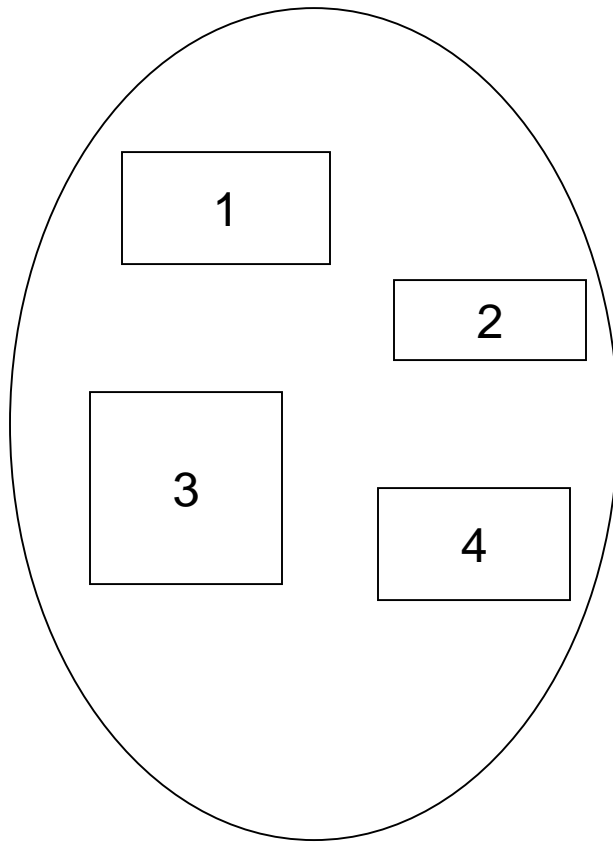
arrays



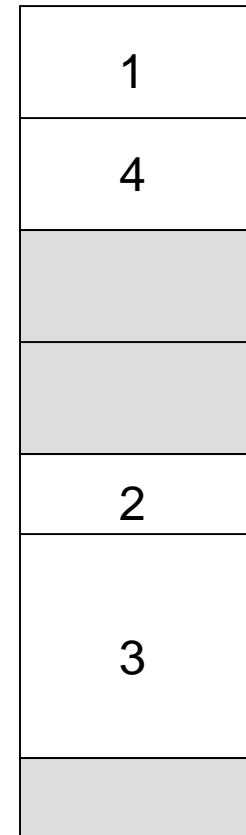
User's View of a Program



Logical View of Segmentation



user space



physical memory space



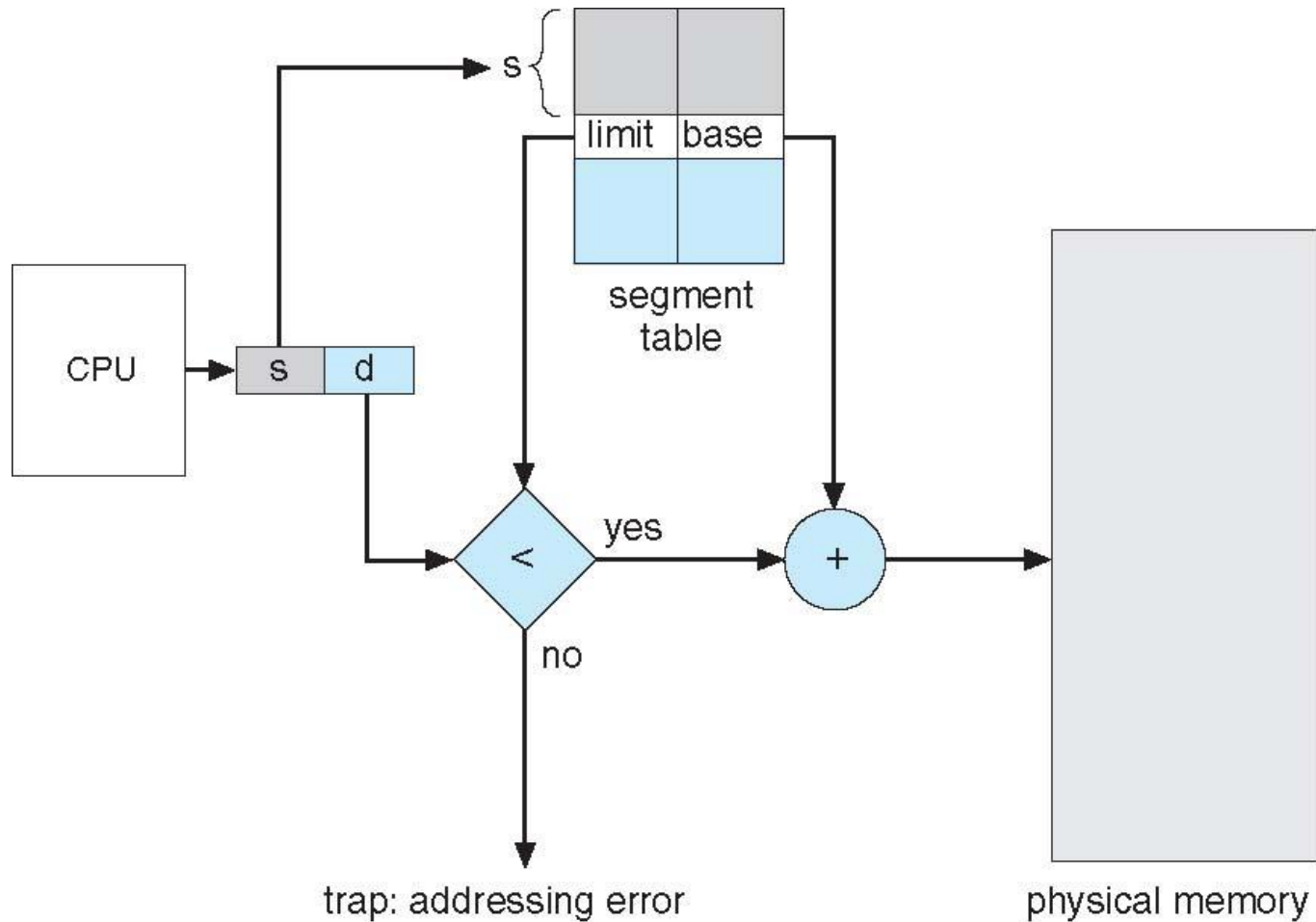
Segmentation Architecture

- Logical address consists of a two tuple:
 $\langle \text{segment-number}, \text{offset} \rangle$,
- Segment table – maps two-dimensional physical addresses; each table entry has:
 - base – contains the starting physical address where the segments reside in memory
 - limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory

Segmentation Architecture (Cont.)

- Protection
 - With each entry in segment table associate:
 - validation bit = 0 \Rightarrow illegal segment
 - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Segmentation

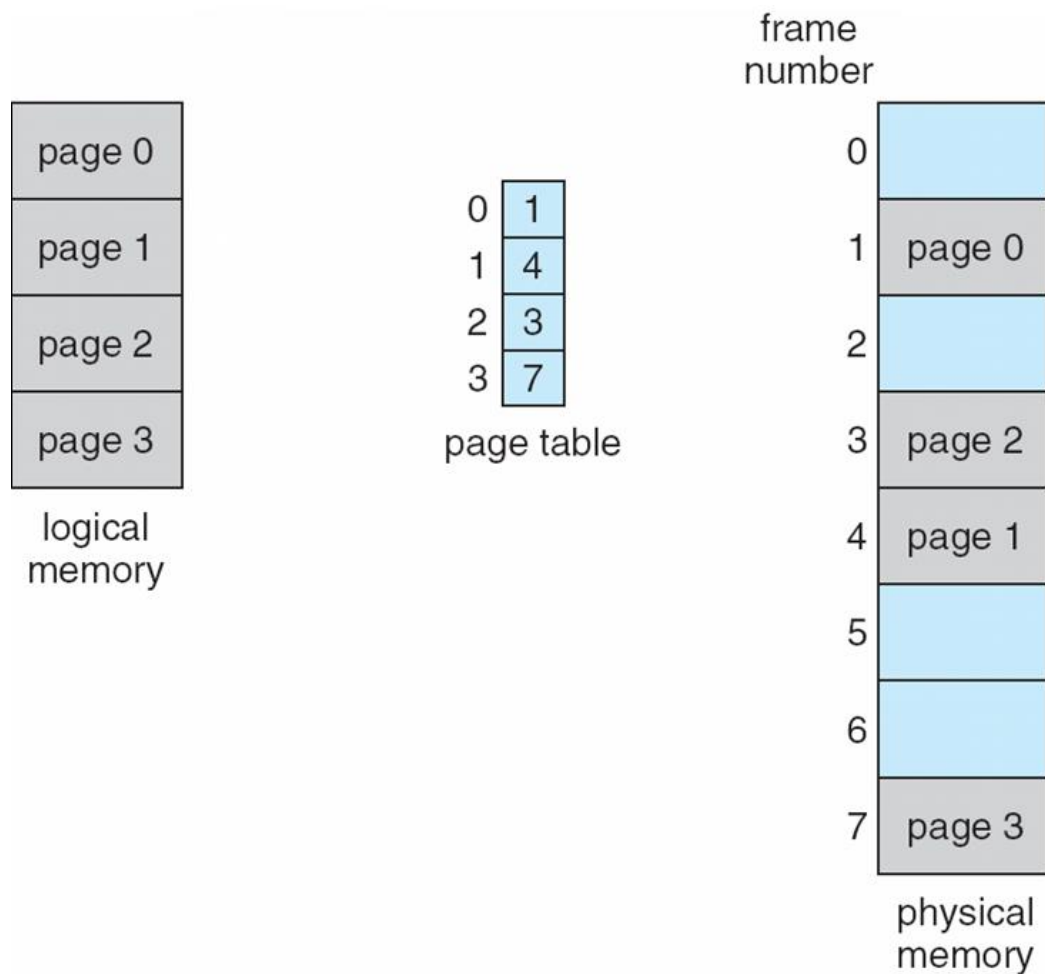
- Swapping?
- Fragmentation?
 - Internal?
 - External?



Memory Management: Paging

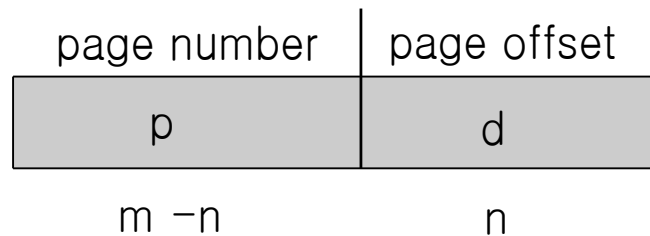
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames
 - Size is power of 2, between 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
- Backing store likewise split into pages
- Still have Internal fragmentation

Paging Model of Logical and Physical Memory



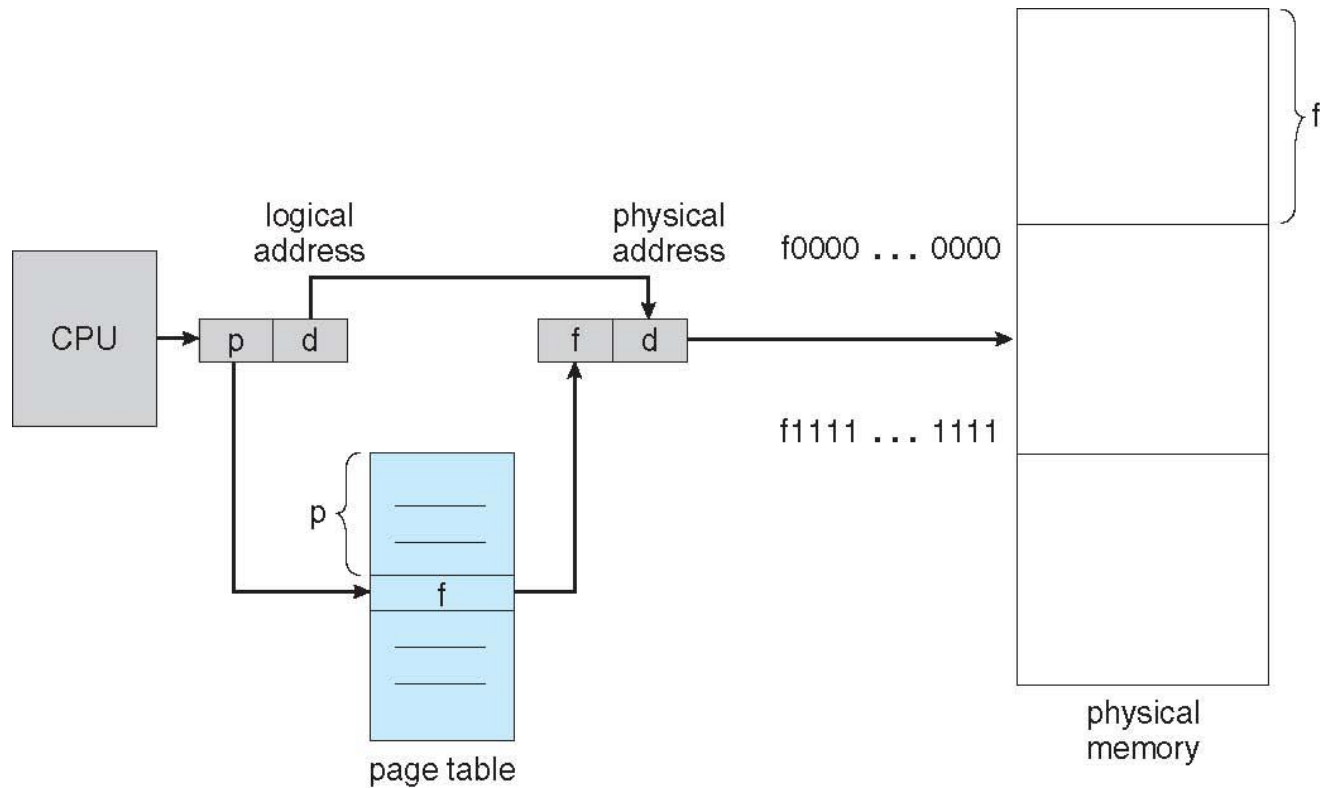
Address Translation Scheme

- 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



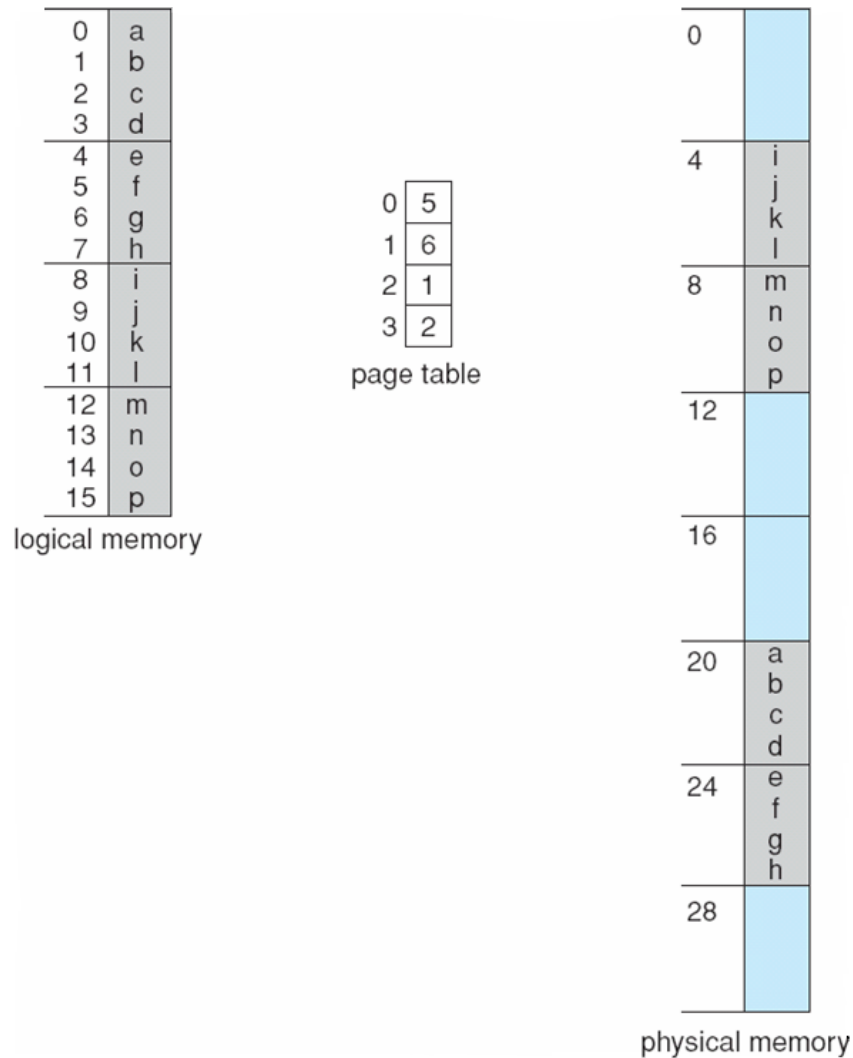
- For given logical address space 2^m and page size 2^n

Paging Hardware

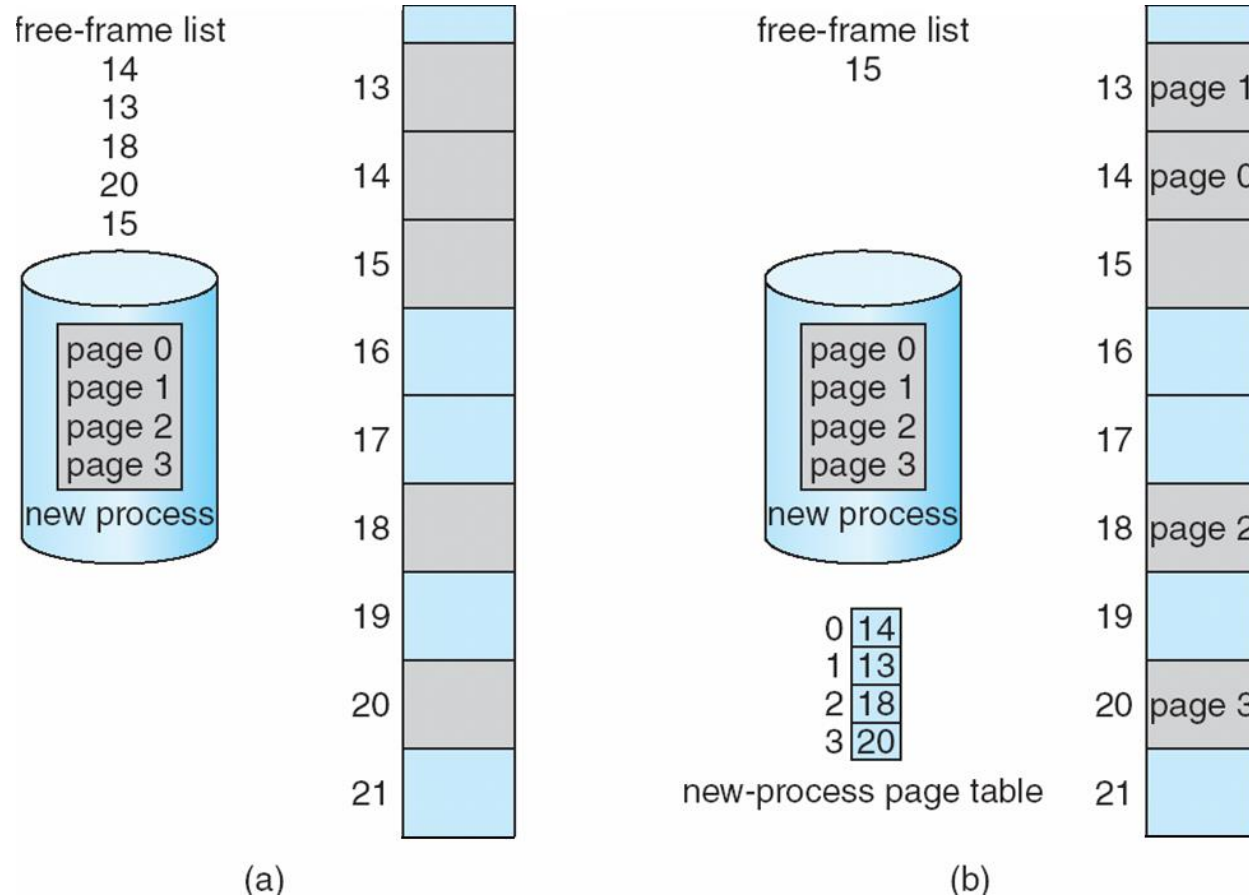


Paging Example

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



Free Frames



Before allocation

After allocation

Paging (Cont.)

- 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
 - So small frame sizes desirable?
 - But each page table entry takes memory to track
 - Page sizes growing over time
 - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Implementation of Page Table

- 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 associative memory or translation look-aside buffers (TLBs)

Associative Memory

- Associative memory – parallel search

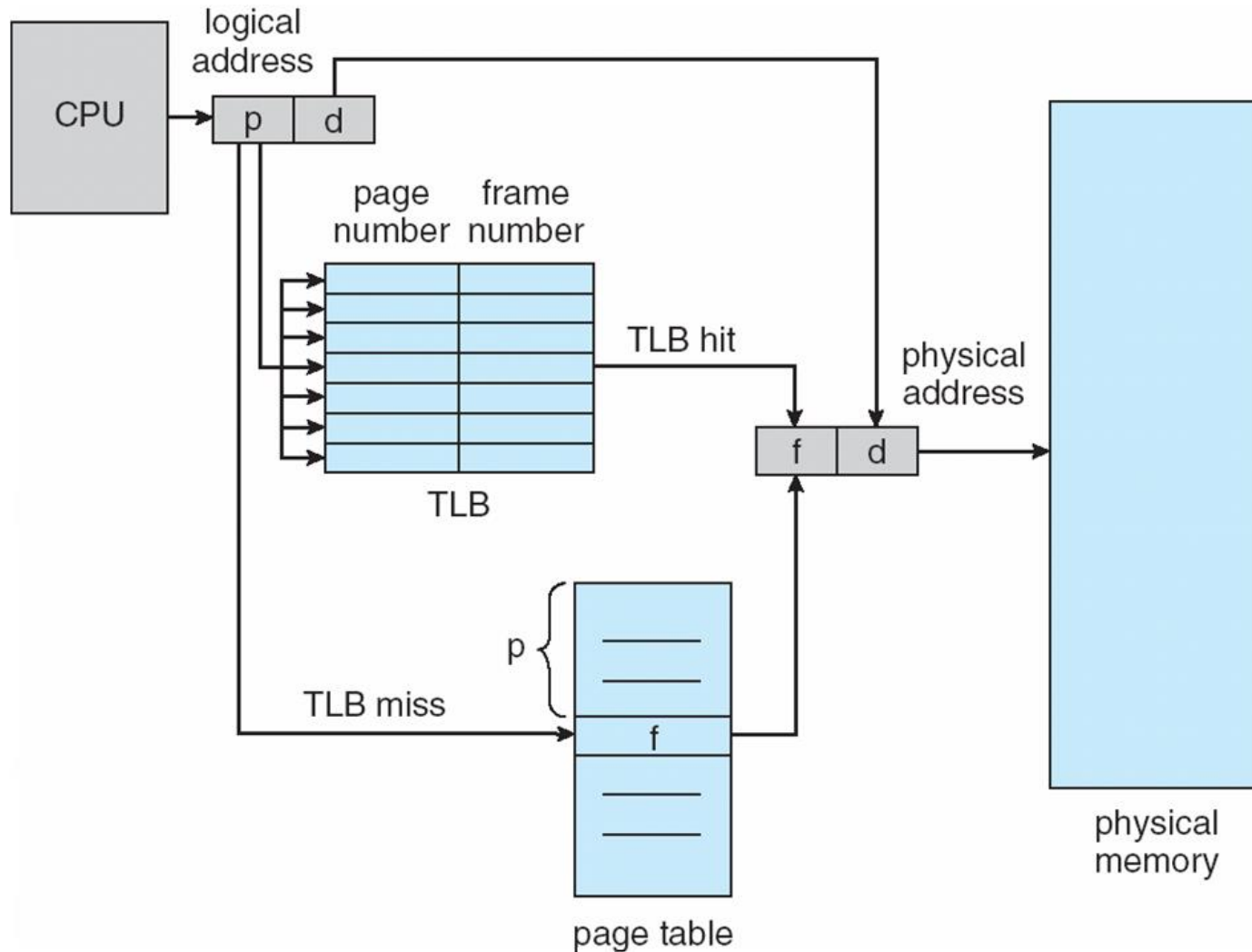
Page #	Frame #

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

Implementation of Page Table (Cont.)

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry (tagged TLB)
 - 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

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
 - Can be $< 10\%$ of memory access time
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Effective Access Time (EAT)
 - α = hit ratio, t = Access time for TLB, m = Access time for memory

- $$\begin{aligned} \text{EAT} &= \text{TLB access time} + \text{Hit case} + \text{Miss case} \\ &= t + \alpha * m + 2 * m * (1 - \alpha) \end{aligned}$$

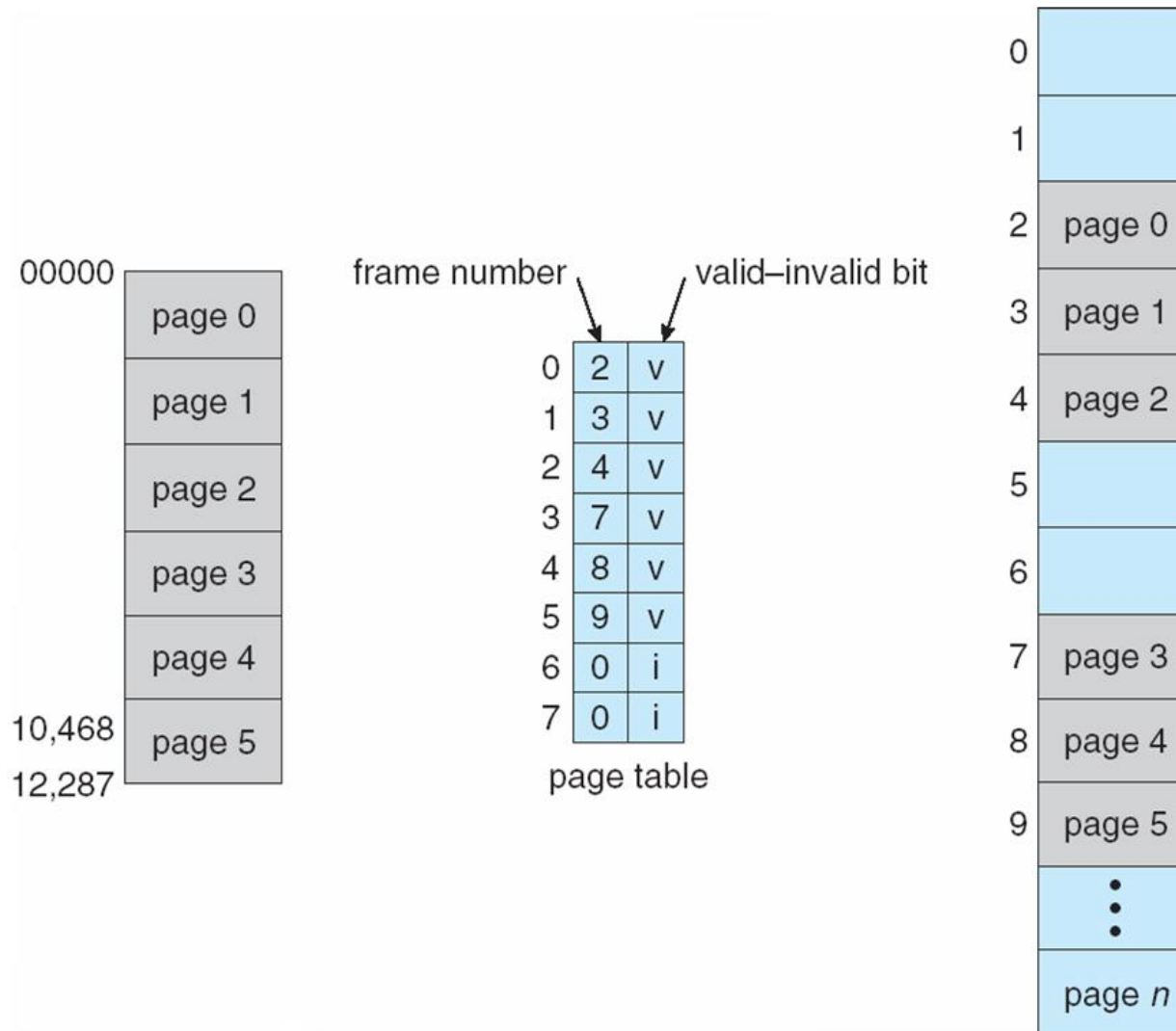
Effective Access Time

- Consider $\alpha = 80\%$, $t = 1\text{ns}$ for TLB search, $m = 100\text{ns}$ for memory access
 - $\text{EAT} = 1 + 0.80 \times 100 + 0.20 \times 200 = 121\text{ns}$
- Consider more realistic hit ratio $\rightarrow \alpha = 99\%$, $t = 1\text{ns}$ for TLB search, $m = 100\text{ns}$ for memory access
 - $\text{EAT} = 1 + 0.99 \times 100 + 0.01 \times 200 = 102\text{ns}$

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)
- Any violations result in a trap to the kernel

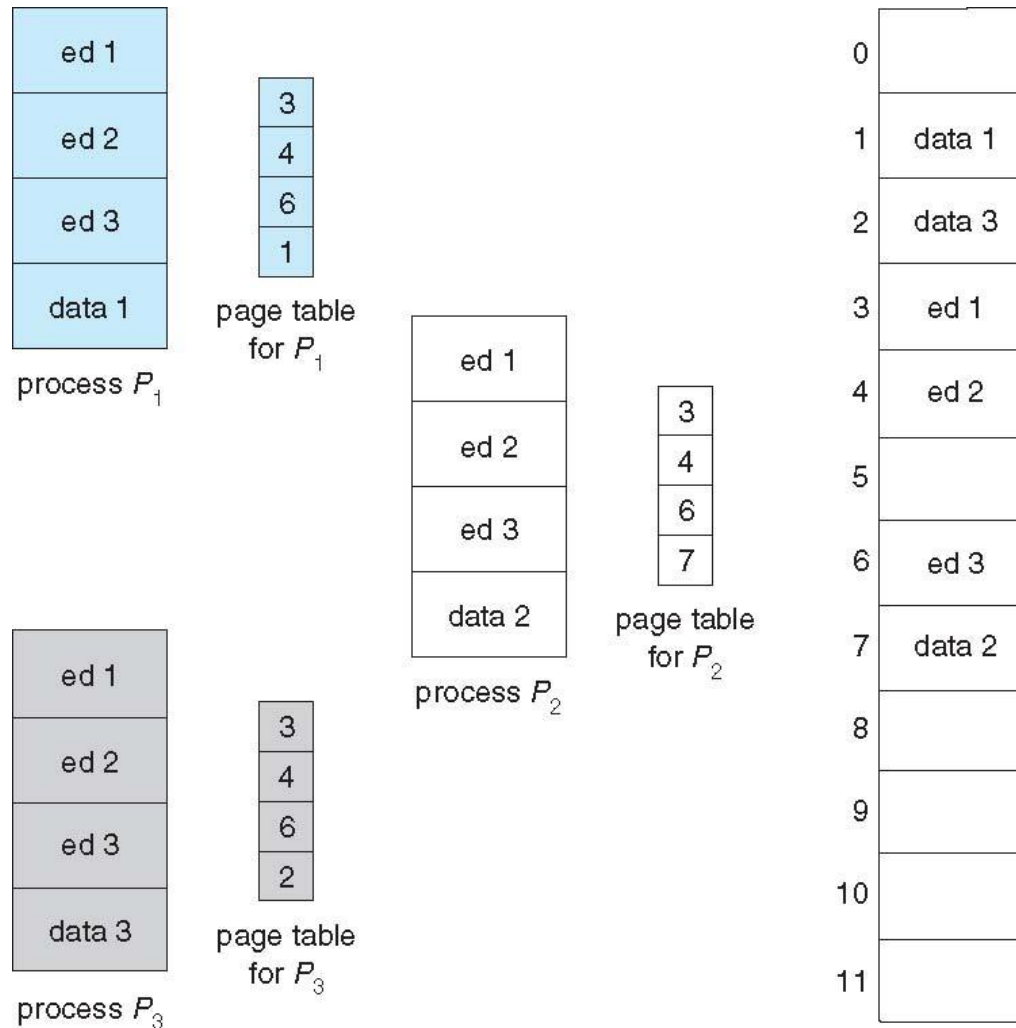
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- 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
- 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

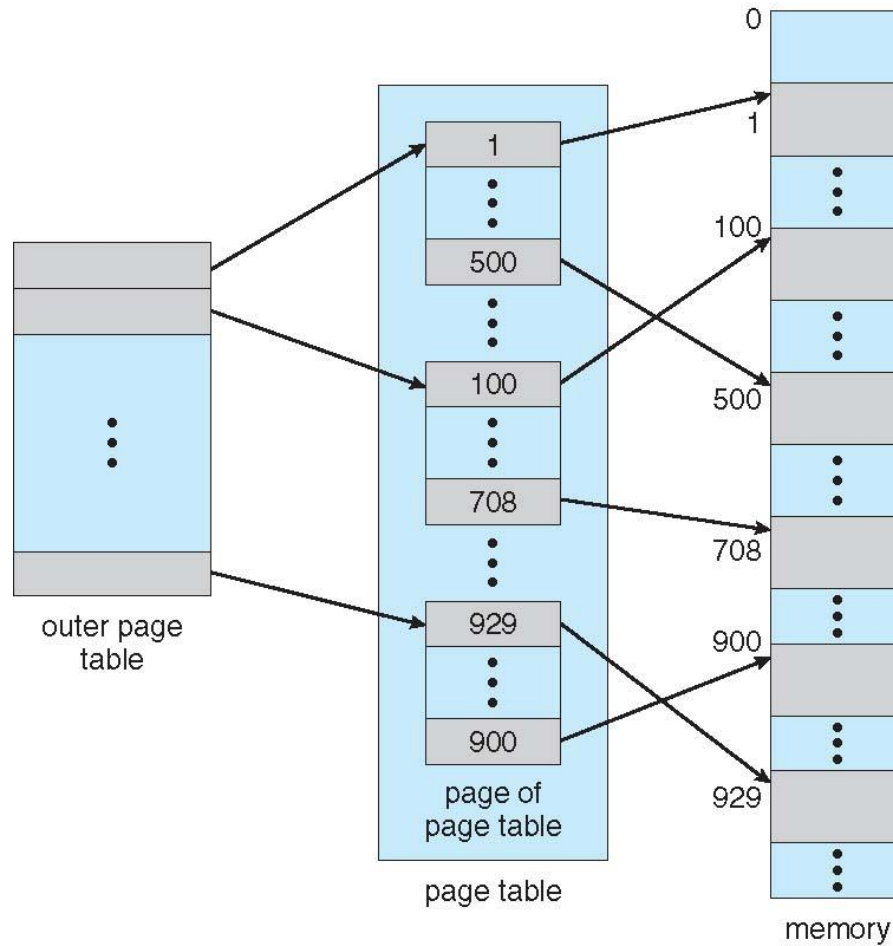
- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have $1,048,576 = 1\text{M}$ entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes \rightarrow 4 MB of physical address space / memory for page table alone
 - That amount of memory used to cost a lot
 - Don't want to allocate that contiguously in main memory
- 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 Page-Table Scheme



Two-Level Paging Example

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

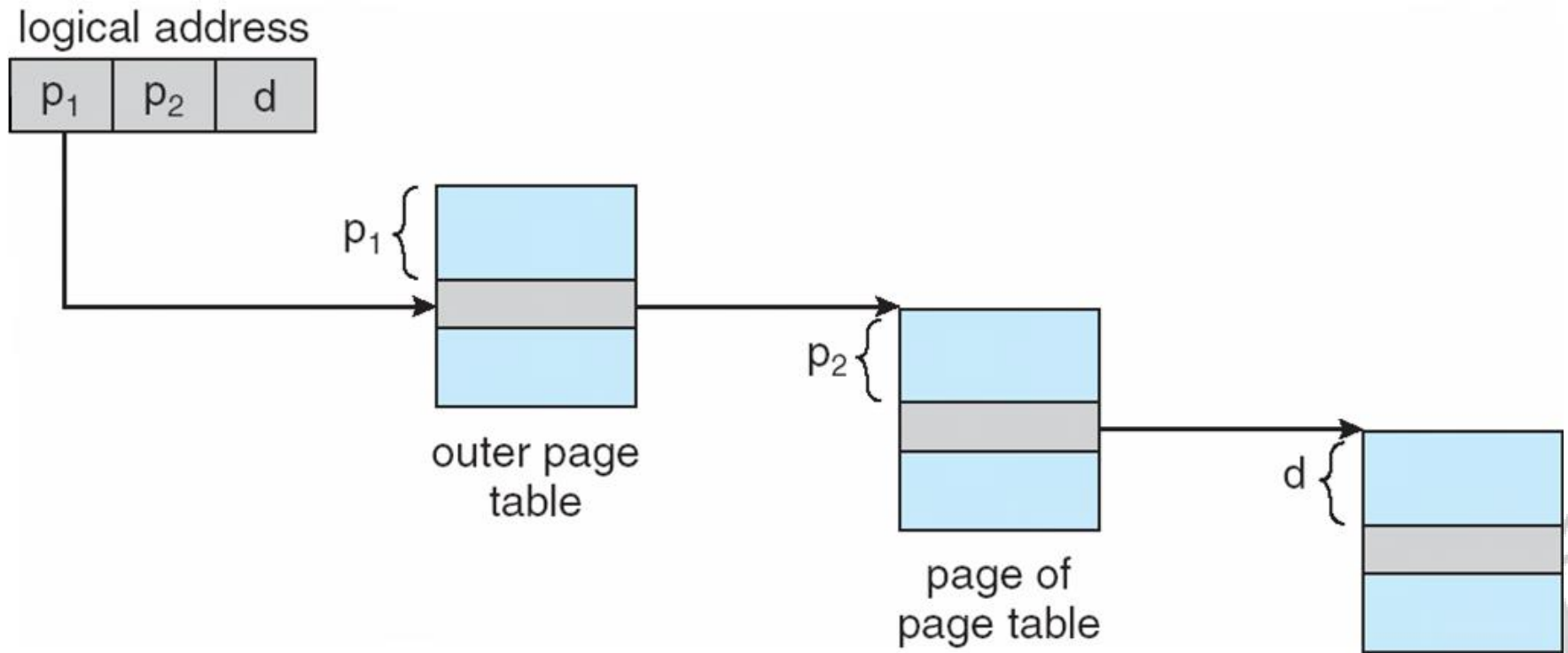
page number		page offset
p_1	p_2	d
12	10	10

- Thus, a logical address is as follows:

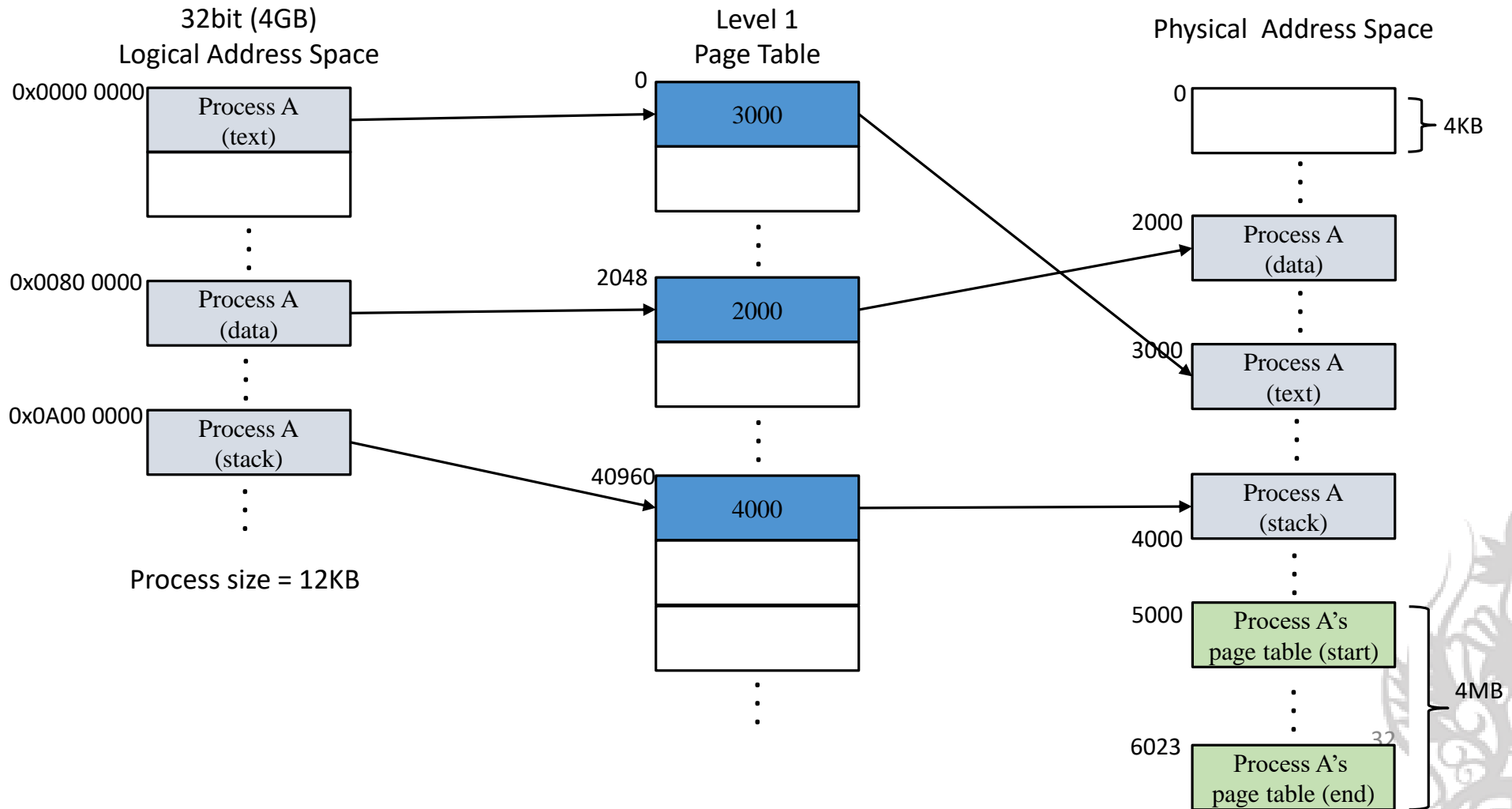
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

- Known as forward-mapped page table

Address-Translation Scheme



E.g. 12 KB-size process with 1-level page table

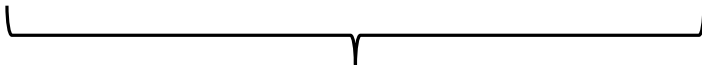


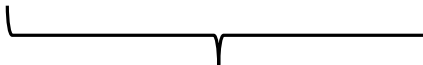
$$\text{Size of PT} = 4B * 2^{20} = 4B * 1024 * 1024 = 4MB$$



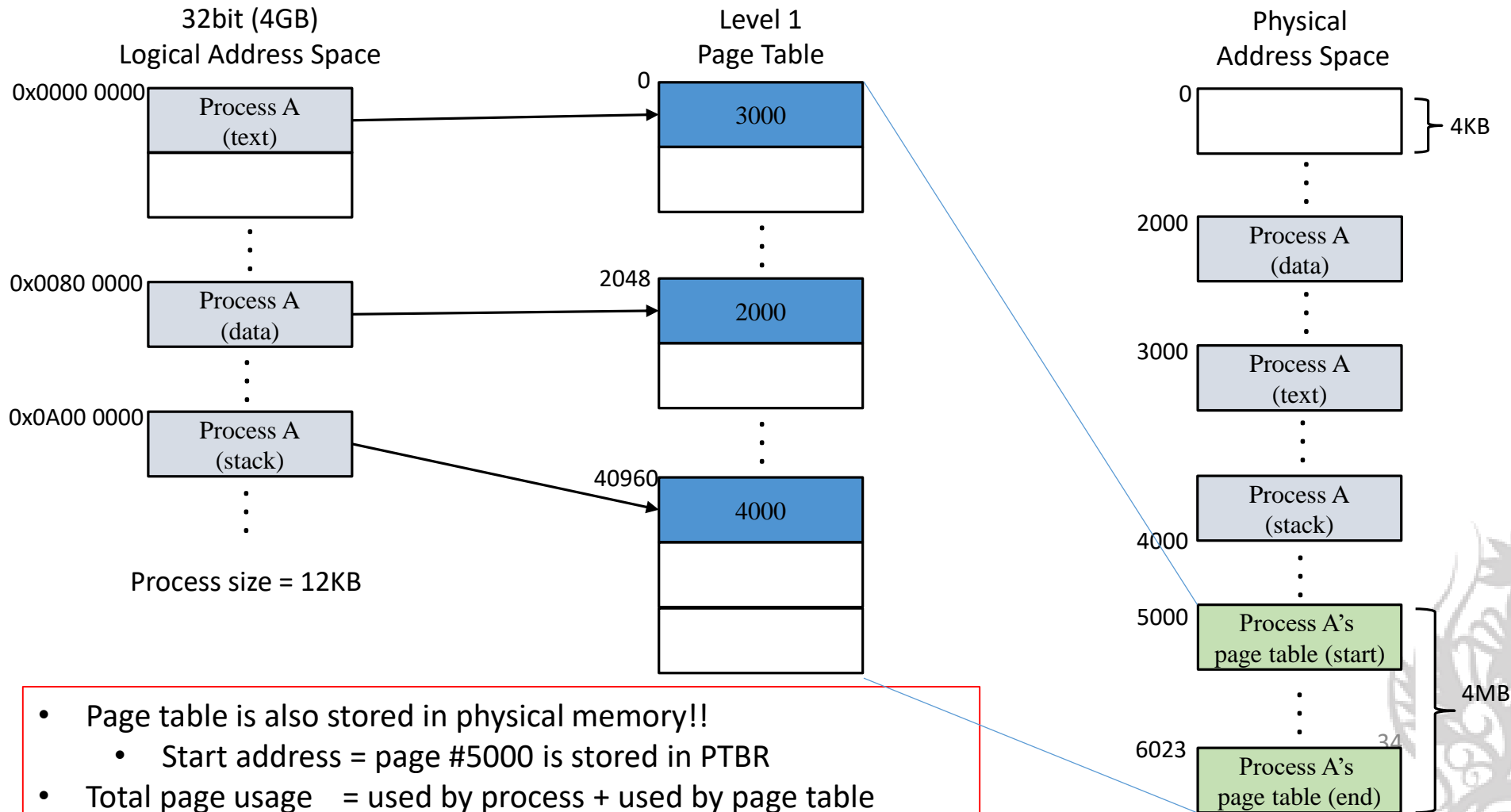
E.g. 12 KB-size process with 1-level page table

- 0x0000 0000 = 0000 0000 0000 0000 0000 0000 0000 0000 0000
- 0x0080 0000 = 0000 0000 1000 0000 0000 0000 0000 0000 0000
- 0x0A00 0000 = 0000 1010 0000 0000 0000 0000 0000 0000 0000


page number


offset
- So, 0x0000, 0x0800 and 0xA000 are page numbers
 - 0, 2048, 40960 : indexes of a page table

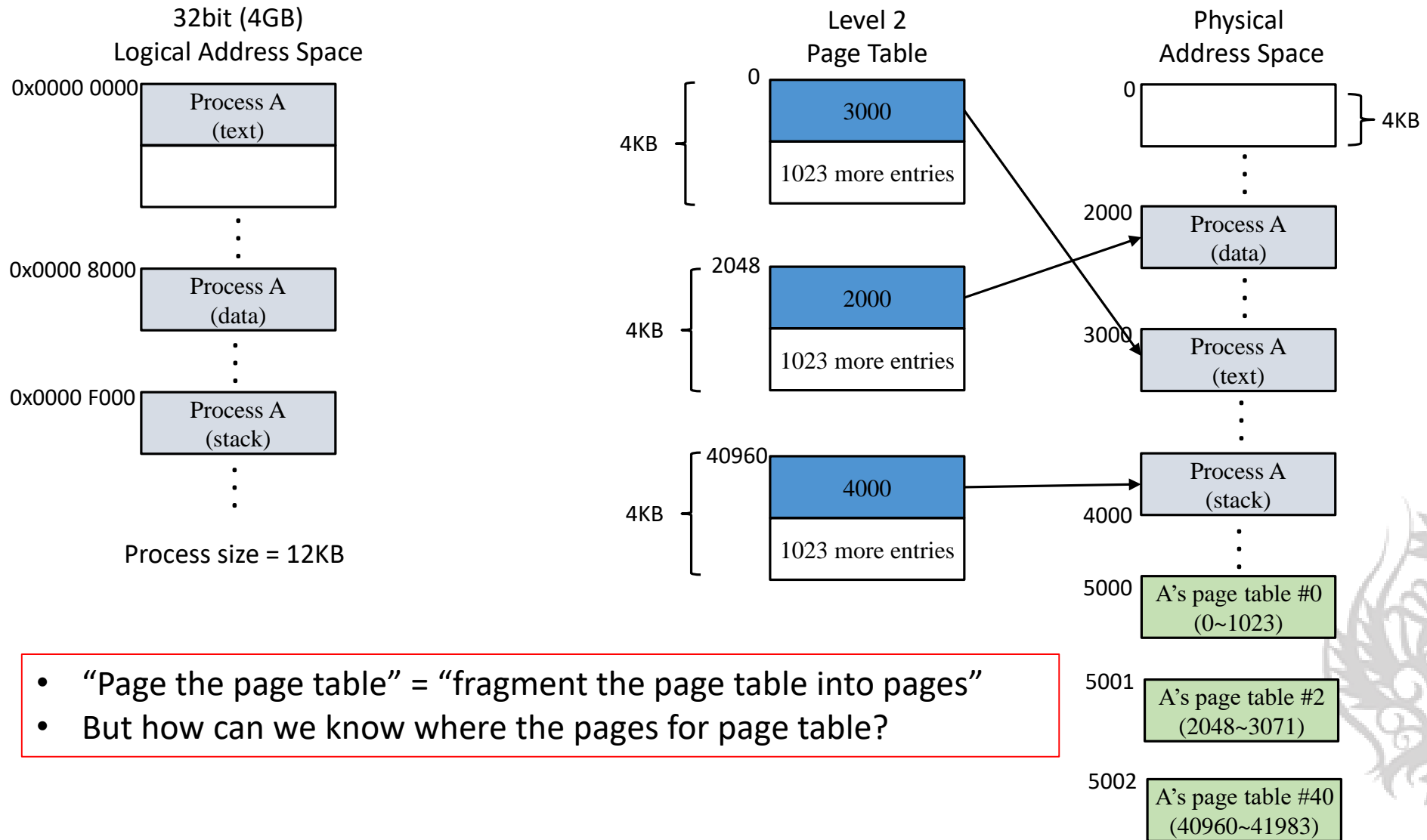
E.g. 12 KB-size process with 1-level page table



- Page table is also stored in physical memory!!
 - Start address = page #5000 is stored in PTBR
- Total page usage = used by process + used by page table
= 3 pages + 1024 pages = 1027 pages



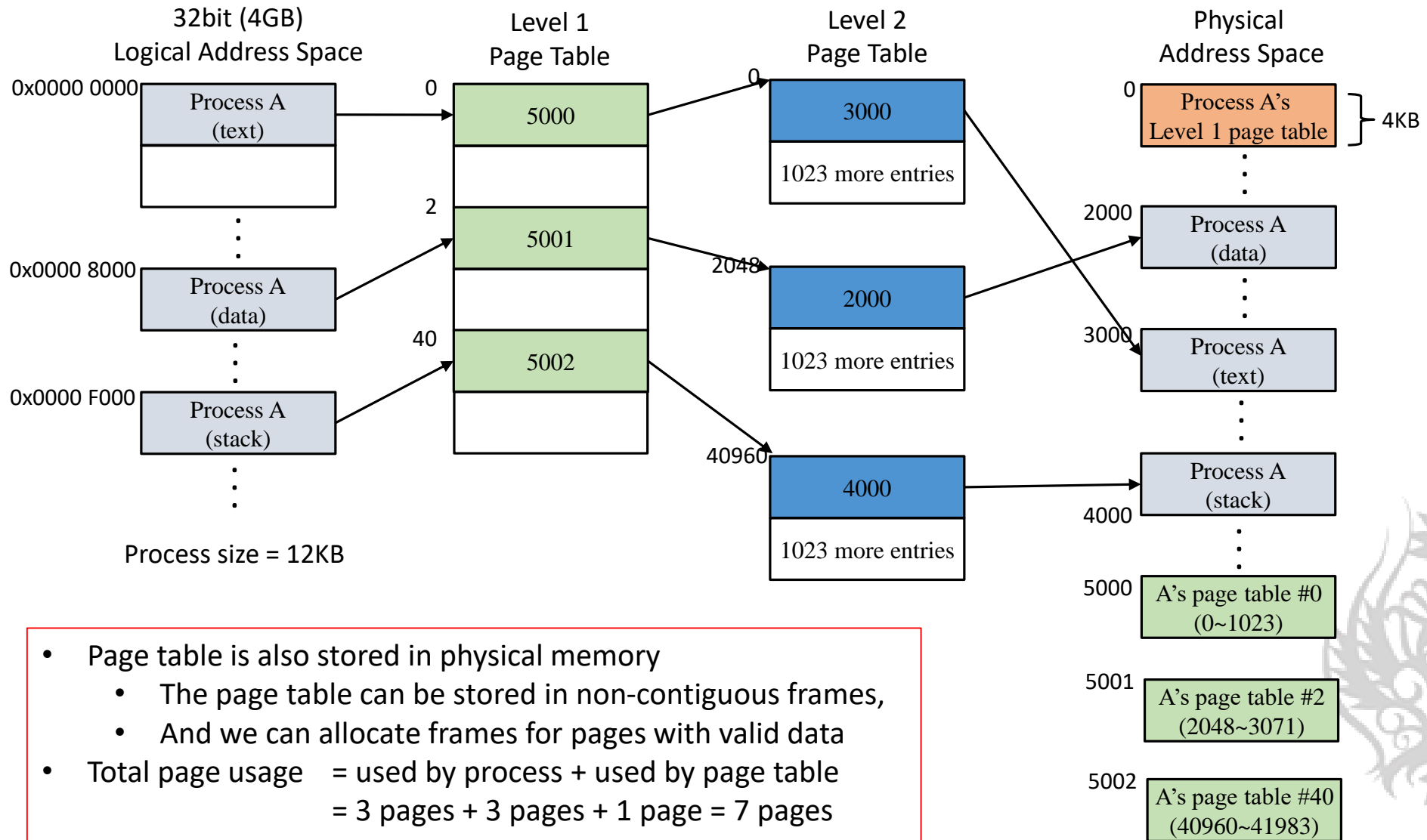
E.g. 12 KB-size process with 2-level page table



- “Page the page table” = “fragment the page table into pages”
- But how can we know where the pages for page table?



E.g. 12 KB-size process with 2-level page table



- Page table is also stored in physical memory
 - The page table can be stored in non-contiguous frames,
 - And we can allocate frames for pages with valid data
- Total page usage = used by process + used by page table
= 3 pages + 3 pages + 1 page = 7 pages

E.g. 12 KB-size process with 1-level page table

- 0x0000 0000 = 0000 0000 0000 0000 0000 0000 0000 0000
- 0x0080 0000 = 0000 0000 1000 0000 0000 0000 0000 0000
- 0x0A00 0000 = 0000 1010 0000 0000 0000 0000 0000 0000

└──────────┘
1-level index

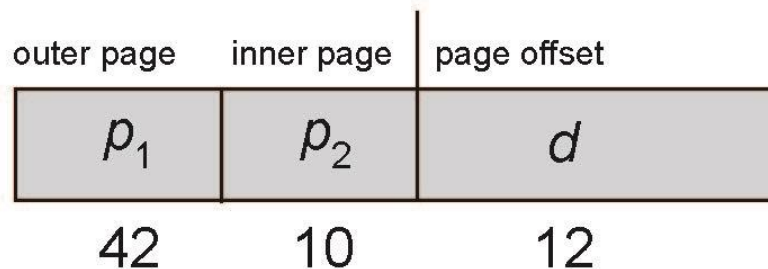
└──────────┘
2-level index

└──────────┘
offset

- So, 0x0000, 0x0800 and 0xA000 are page numbers
- 1-level indexes
 - 0, 2, 40
- 2-level indexes
 - 0, 0, 0

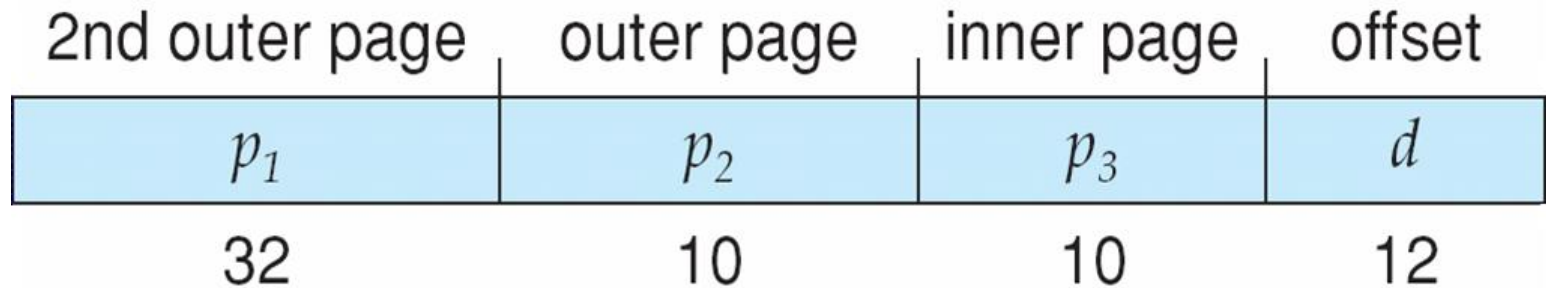
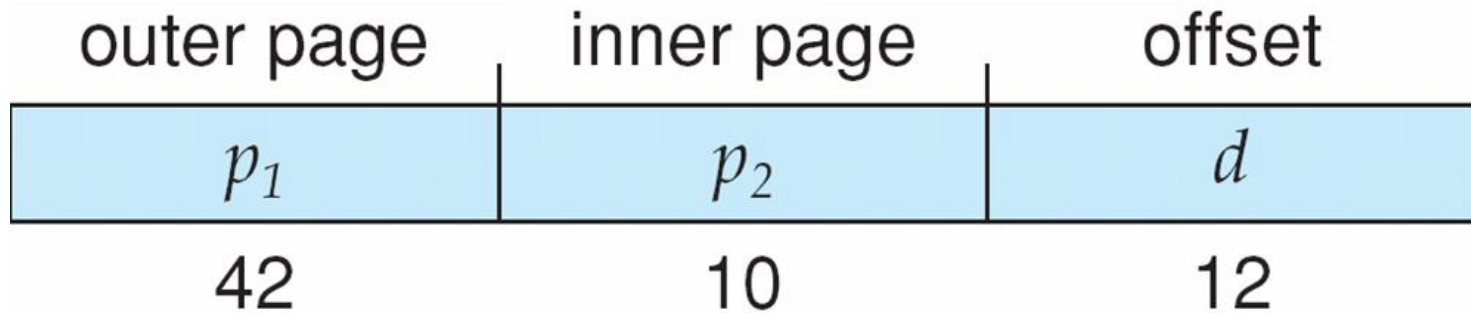
64-bit Logical Address Space

- 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 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

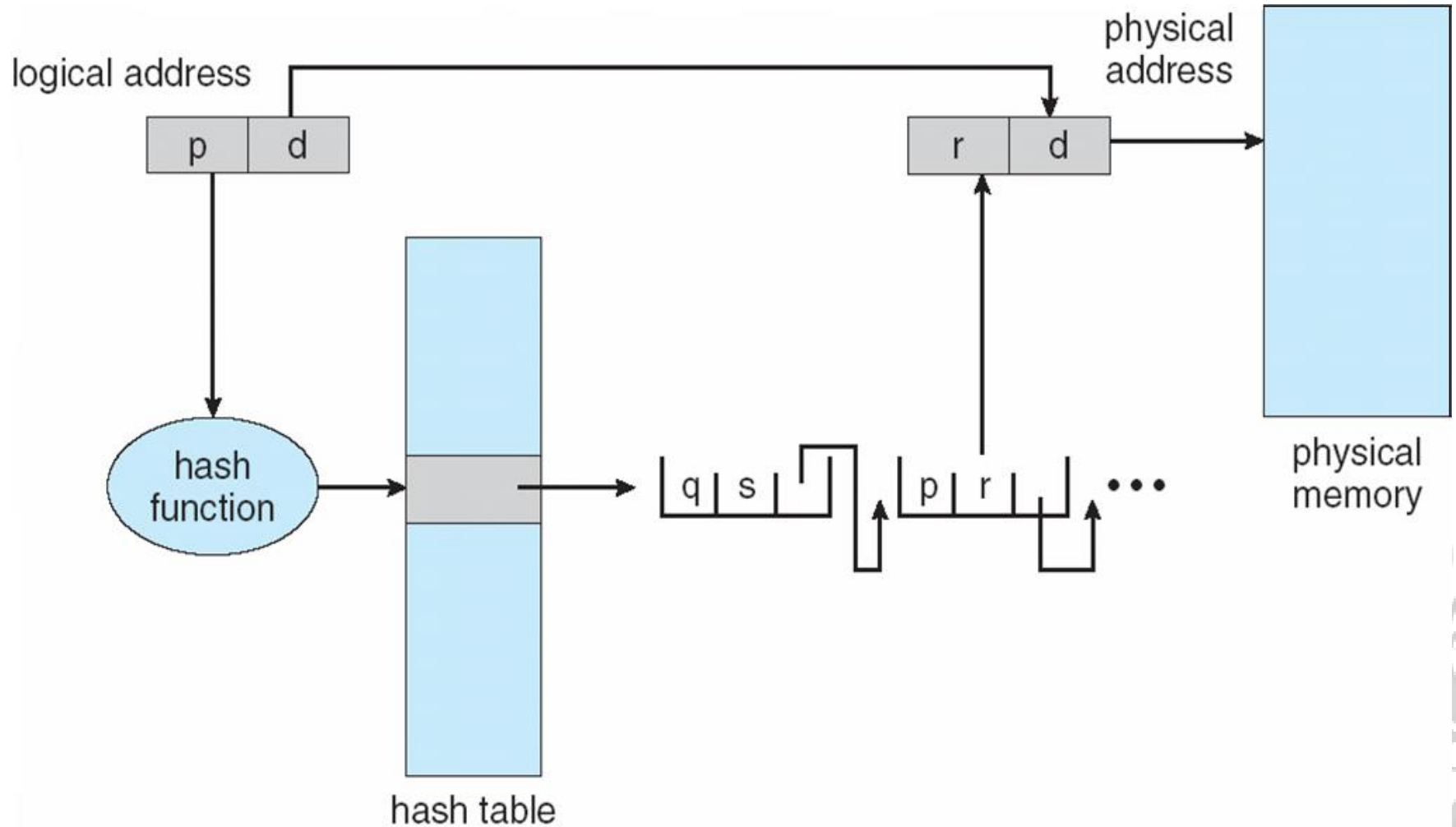
Three-level Paging Scheme



Hashed Page Tables

- 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

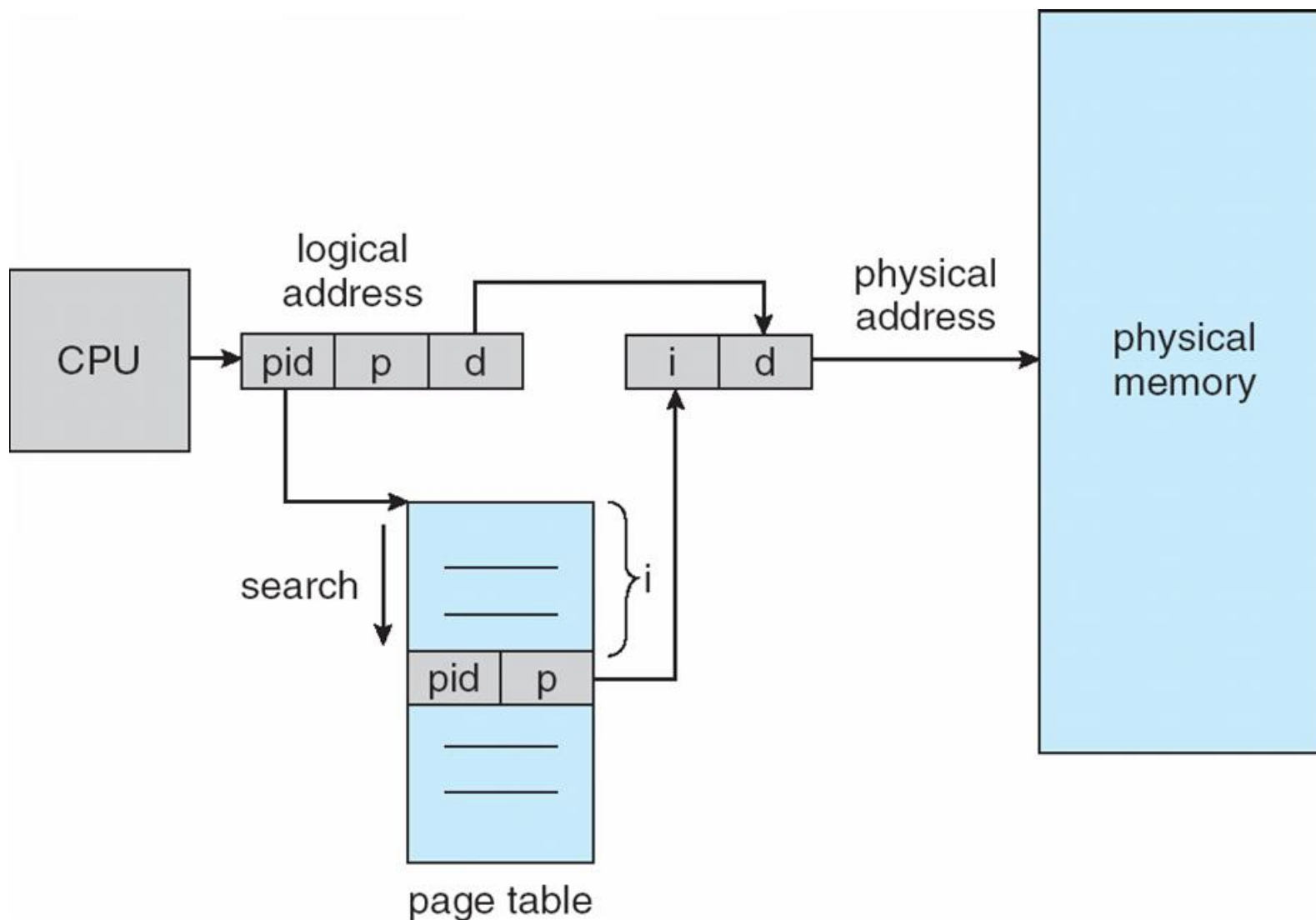
Hashed Page Table



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
 - 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



Example: 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)

Example: 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.

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))

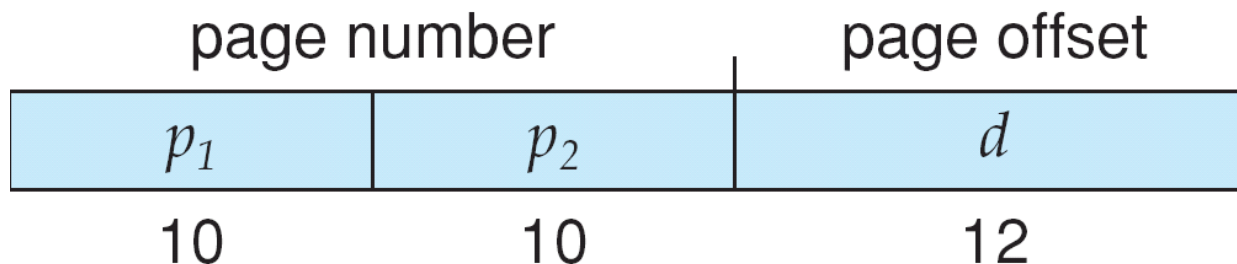
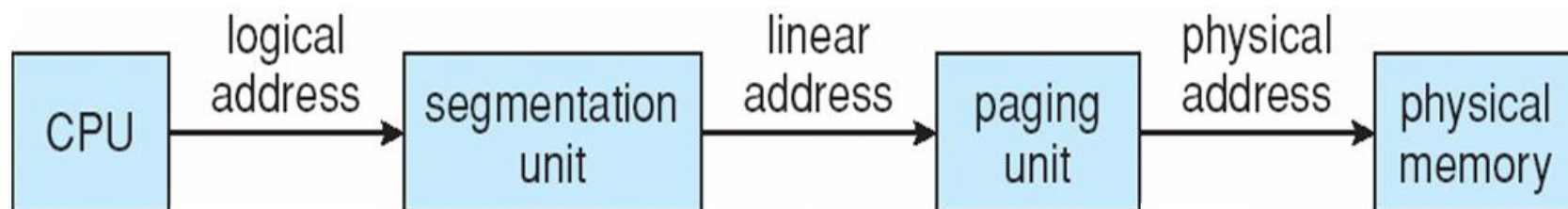
Example: The Intel IA-32 Architecture (Cont.)

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

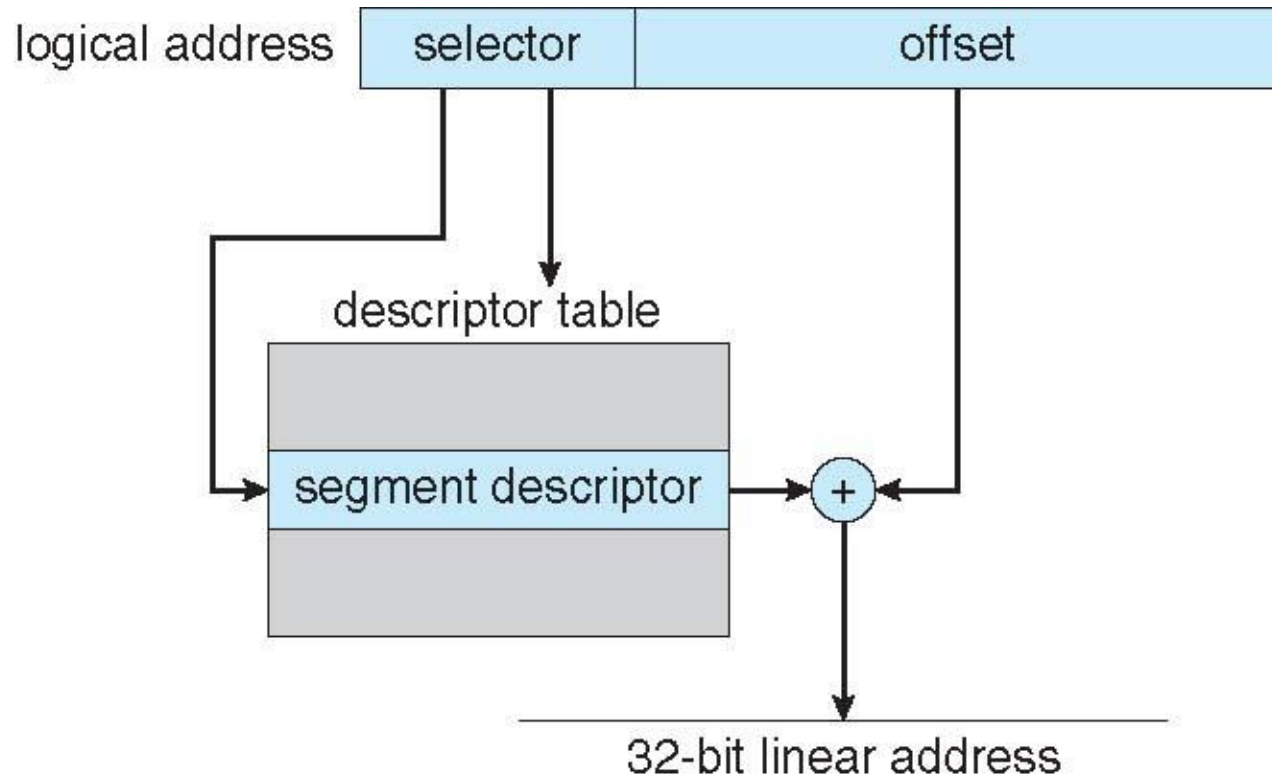


- Linear address given to paging unit
 - Which generates physical address in main memory
 - Paging units form equivalent of MMU
 - Pages sizes can be 4 KB or 4 MB

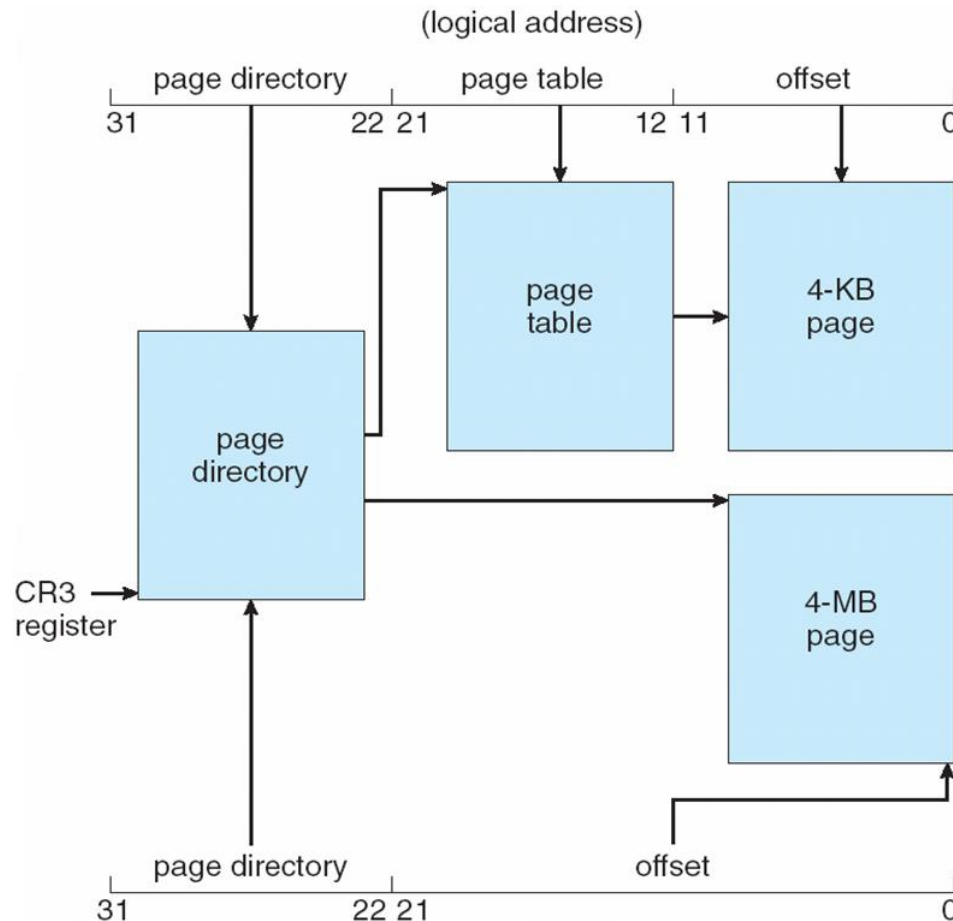
Logical to Physical Address Translation in IA-32



Intel IA-32 Segmentation



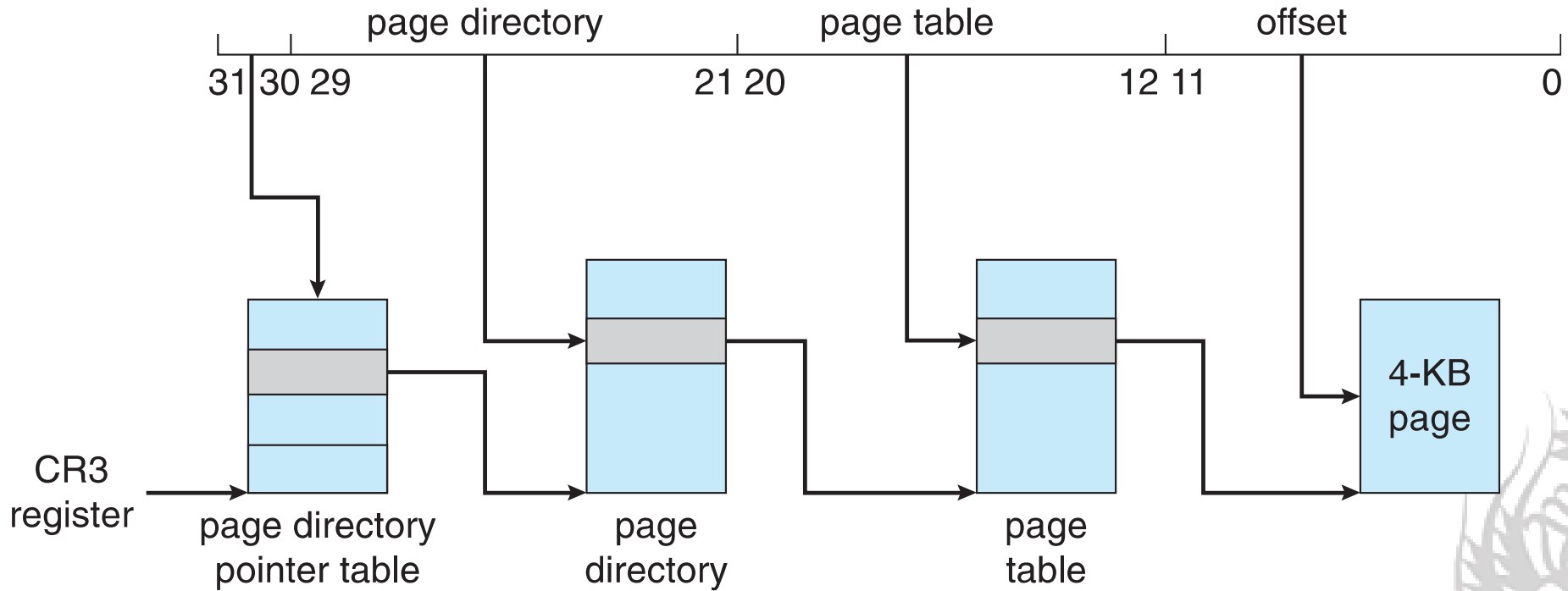
Intel IA-32 Paging Architecture



Intel IA-32 Page Address Extensions

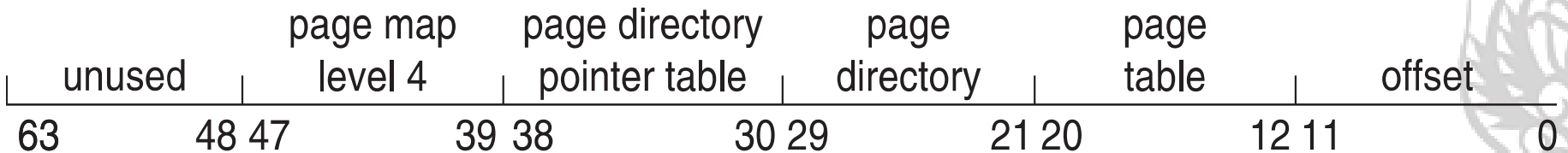
- 32-bit address limits led Intel to create page address extension (PAE), allowing 32-bit apps access to more than 4GB of memory space
 - Paging went to a 3-level scheme
 - Top two bits refer to a page directory pointer table
 - Page-directory and page-table entries moved to 64-bits in size
 - Net effect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

Intel IA-32 Page Address Extensions



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: 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)
- 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 others are checked, and on miss page table walk performed by CPU

Example: ARM Architecture

