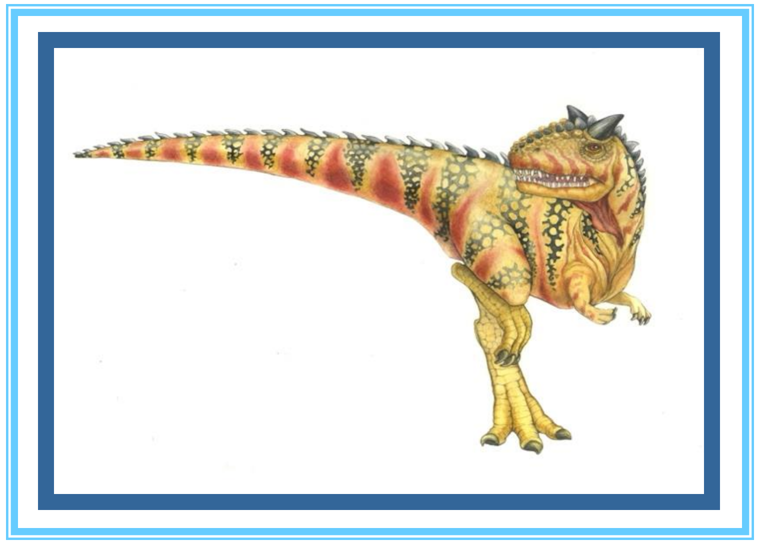
New direction

Memory management

Chapter 9: Main Memory



Chapter 9: Memory Management Strategies

- Background
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Examples: the Intel 32 and 64-bit Architectures and ARM Architecture

(process → part of memory)

↓

holes of memory

↓

seperate to different part of main memory

↓

segmentation

→ solve fragmentation:

Objectives

linker and loader → virtual address

→ relocate address

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in address translation.

(memory space) → allocate (not difference size) but how?

- Dynamic storage-allocation problem - apply **first-, best-,** and **worst-fit** strategies for allocating memory contiguously. → 有holes
- Explain the distinction between **internal** and **external fragmentation**.
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB). (segmentation & paging)
- Describe hierarchical paging, and hashed paging

how this do in address space

- Describe address translation for IA-32, x86-64, and ARMv8 architectures

Background

1 byte = 8 bits

- Main memory is central to the operation of computer systems
- Memory consists of a large array of **bytes**, each with its own address – **byte-addressable**
- Program must be brought (from disk or other secondary storage) into memory and placed within a process for it to run, i.e., PCB points to the address space of the process
- A typical instruction-execution cycle include
  - CPU first fetches an instruction from memory (or cache)
  - The instruction is then decoded and may cause operands to be fetched from memory.
  - After the instruction executed on the operands, results may be stored back in memory.
- The memory management unit or MMU only sees a stream of addresses
  - It does not know how they are generated, could be by the instruction counter, indexing, indirection, literal addresses, and so on or what they are for (instructions or data).
  - Accordingly, MMU is interested only in the sequence of memory addresses generated by a running program.

✗ care how you inside/ generate the address

virtual address (generated by CPU)

Background (Cont.)

implemented in CPU!

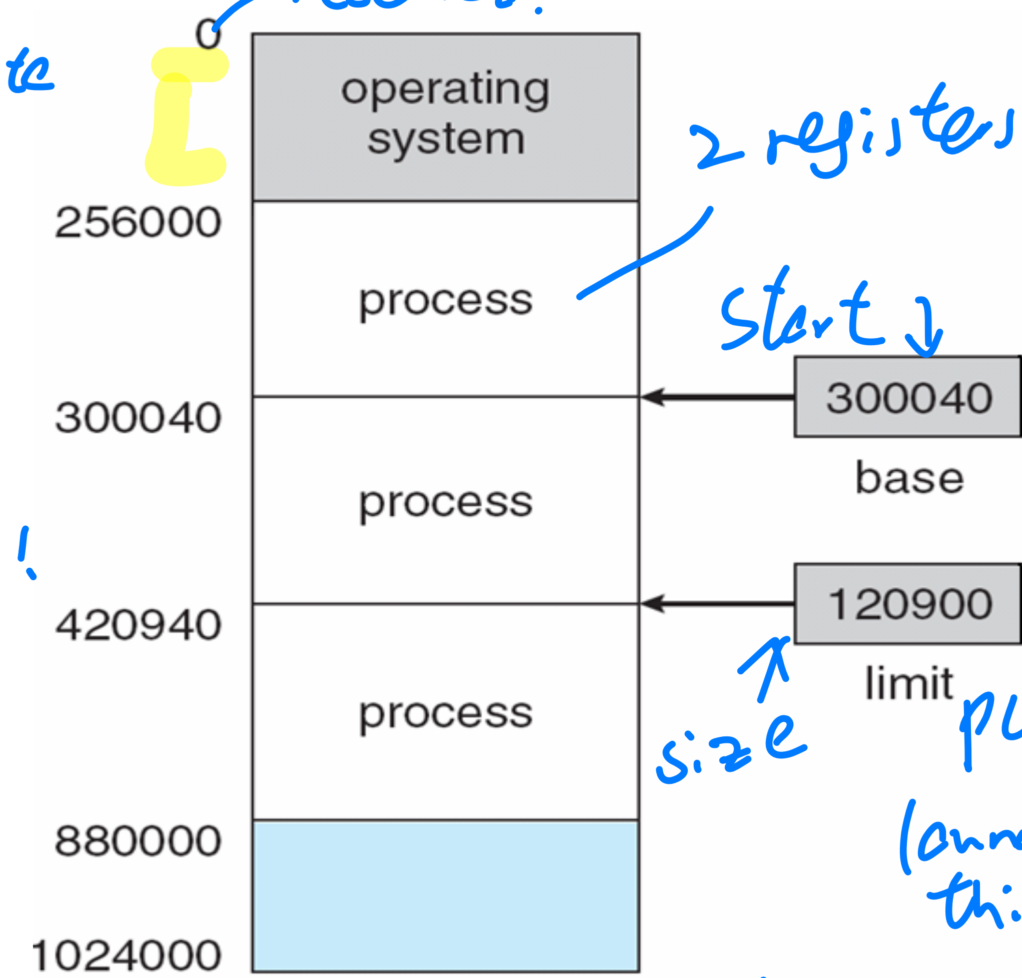
- Main memory (including cache) and registers are the only storage CPU can access directly; in another word, CPU cannot access secondary storage such as disk directly, but indirectly through I/O controllers
- Memory management unit or MMU only sees a stream of addresses + read requests, or address + data and write requests
  - Registers can be accessed in one CPU clock (cycle)
  - Accessing main memory (through memory bus) may take many CPU clocks, causing a **memory stall**, when it does not yet have the data required to complete the instruction it is executing
  - **Cache** sits between main memory and CPU registers, residing on CPU chips for fast access
- Memory protection is required to ensure correct operation → separate !
  - We must protect the operating system memory space from being accessed by user processes, as well as protect user processes from one another
  - This protection must be provided by the hardware for performance – performance or speed

Per-process memory space separated from each other

- Separate per-process memory space protecting the processes from each other is fundamental to have multiple processes loaded in memory for concurrent execution – multiprogramming systems
- A pair of **base** and **limit registers** define the legal range of a process address space
  - For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive)

kernel program can update the data

User cannot:
↓
increase protection!



reserved!

2 registers → store in PCB

physical memory (real)

start ↓

size

PCB launches this

CPU loads process
↓
launch new process/context switch

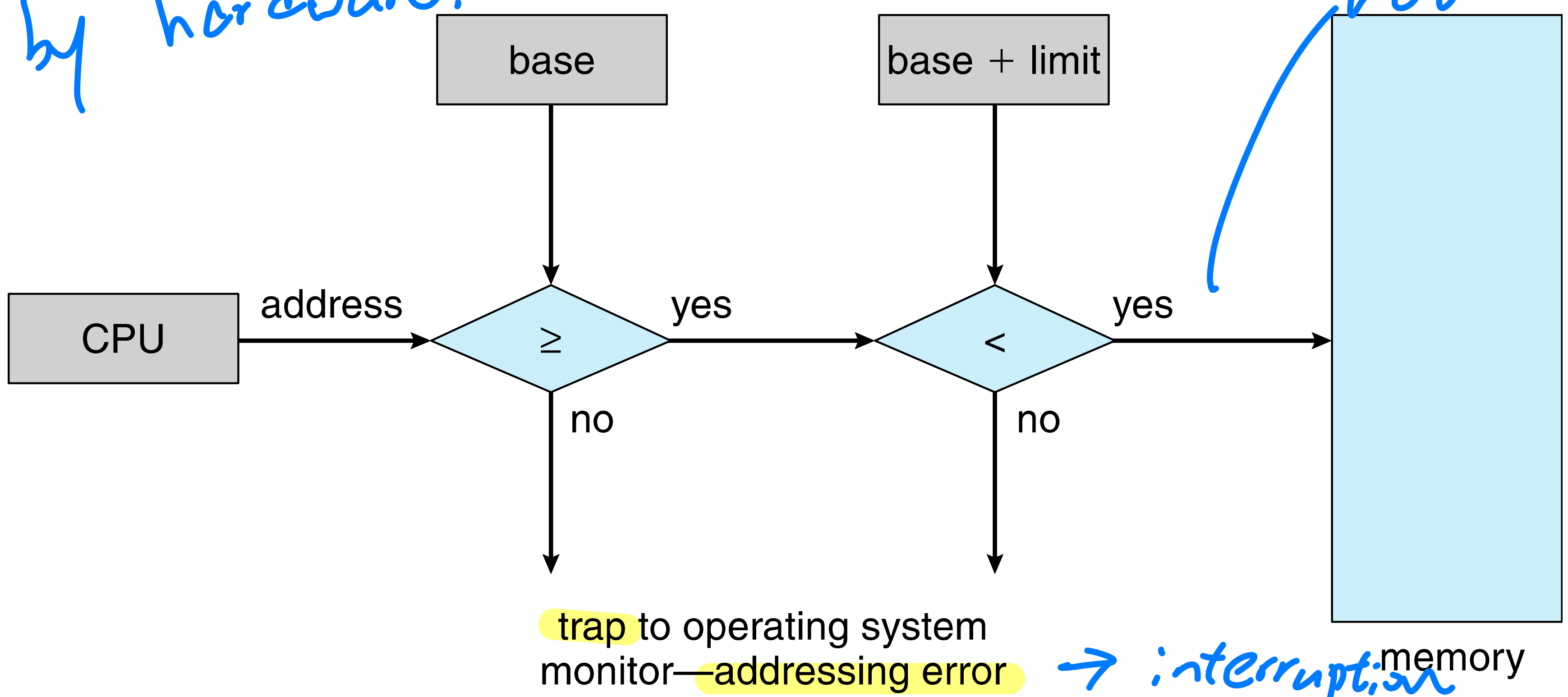
base + limit = new address (end)

Hardware Address Protection

- Protection of memory space is accomplished by having CPU compare every address generated in user mode with these two registers
- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction (executed only in the kernel mode)
- The operating system, executing in kernel mode, however, is given unrestricted access to both operating-system memory and users' memory

Done by hardware!

address relocation!



→ interruption

exceed size!

Address Binding

segmentation/paging/relocation

APP file! OS9A.exe

- Usually, a program resides on a disk as a binary executable file, which must be brought into memory and placed within a process (part of its address space) for it to run
- Most systems allow a user process to reside in any part of the physical memory. Although the computer may start at 00000, the first address of a user process need not be 00000
- A user program goes through several steps - some of which may be optional before being executed. Addresses may be represented in different ways during these steps:
  - Source code addresses usually symbolic – the variable count
  - A compiler typically binds these symbolic addresses to relocatable addresses, such as "14 bytes from beginning of this module"
  - Linker or loader in turn binds the relocatable addresses to absolute addresses, i.e., 74014
  - Each binding is essentially a mapping from one address space to another

Multistep Processing of a User Program

GDB

Address binding of instructions and data to memory addresses can happen in three different stages

final address Cold time!)

- **Compile time**: If memory location is known a priori, absolute code can be generated; must recompile code if starting location changes (e.g., "gcc"). MS-DOS uses this

virtual address X, 因有 multi-programming concepts!

- **Link** or **Load time**: Compiler must generate relocatable code if memory location is not known at compile time (e.g, Unix "ld" does link). The binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value

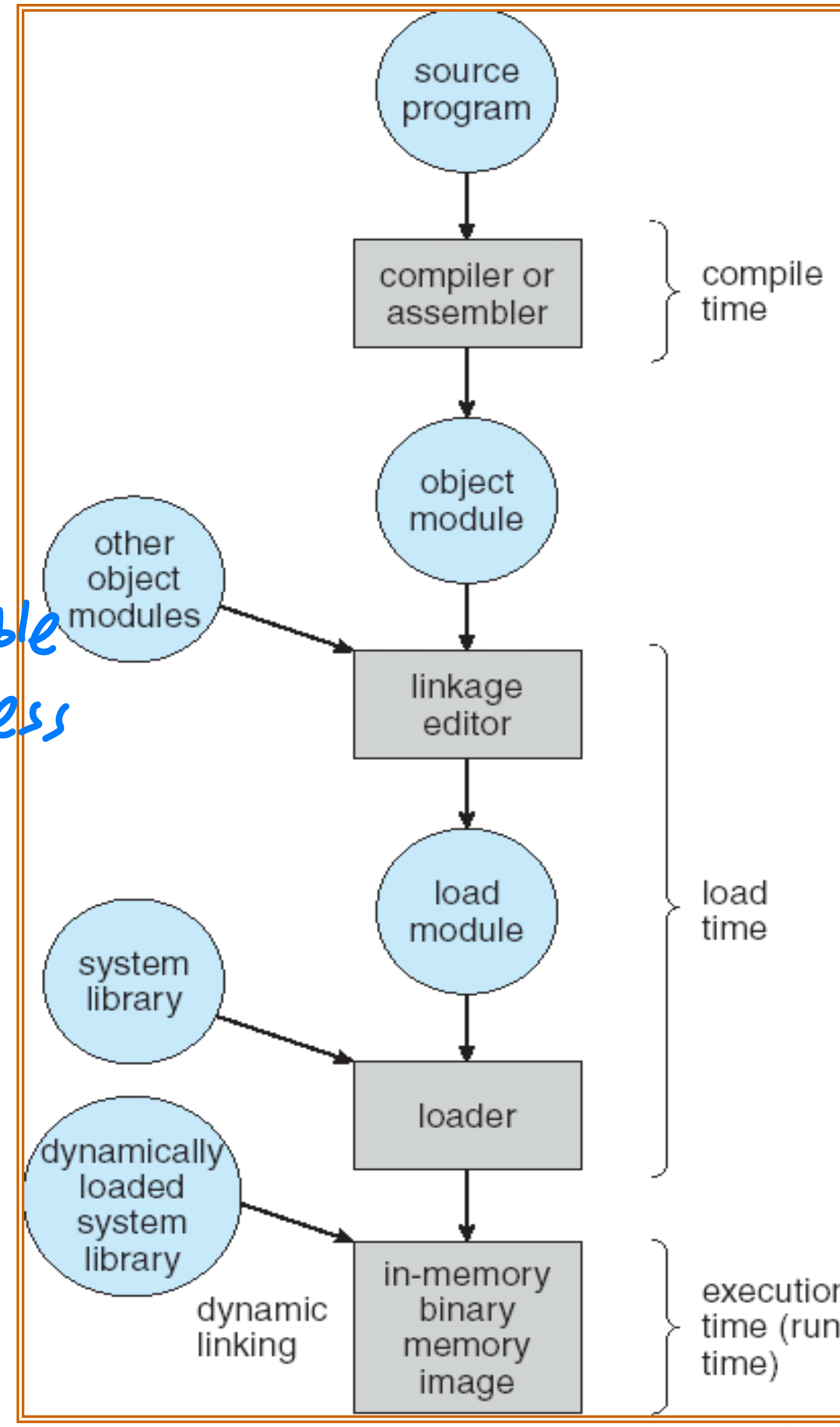
changeable address

More complicated address translation

- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. This needs hardware and operating system support for address maps (e.g., base and limit registers), e.g., dynamic libs. **Most general-purpose operating systems use this method.**

need very very large address space!

separate into different parts!



Compile → Address relocation

↓

mapping between virtual address and physical address

↓

by virtual memory system

Address Translation and Protection

- In the old days of uni-programming (e.g., MS-DOS), only one program can run at the time, thus it occupies "nearly" the entire physical memory
  - No address translation is needed, nor is there any protection

fixed!

old time!

- In the earlier stage of multi-programming such as Windows 3.1 (1990-1992)
  - No address translation, binding occurs at link or loader time – address adjusted when programs are loaded into memory. Bugs in programs can crash the system
  - Protection was added later using base and limit registers, preventing illegal access by another user program

? virtual address!

no protection initially!

Why need virtual address?

Because may want spilt program into different address space!

Logical vs. Physical Address Space

virtual address

continous address space!

- Recall: Address Space
  - All the addresses space a process can "touch"
  - Each process has its own unique address space, different from kernel address space
- Consequently, there are two views of memory

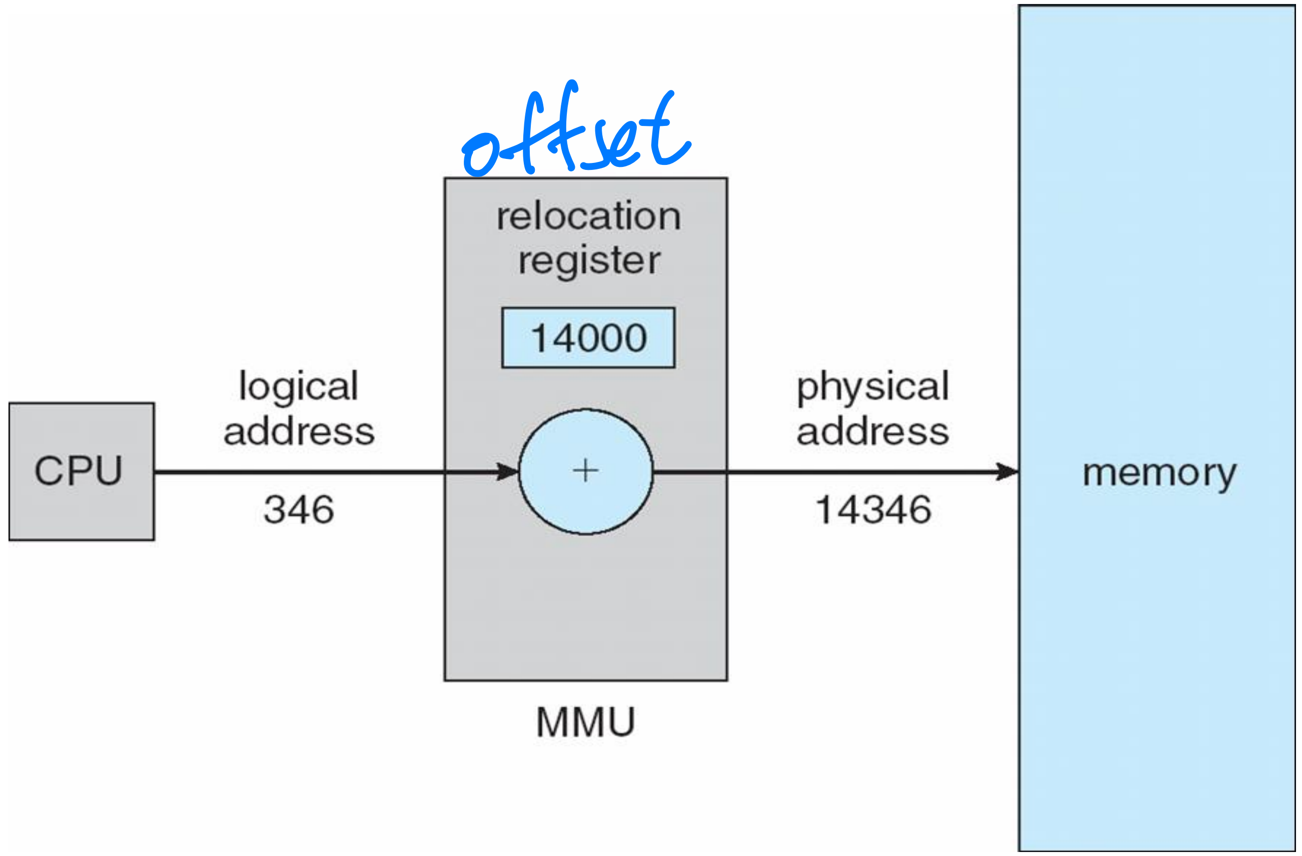
  Only meaningful by process itself!
  - Logical address – generated by the CPU; also referred to as virtual address
  - Physical address – address seen by the memory Real address!
  - The translation is done by memory management unit or MMU – hardware device
- Translation makes it much easier to implement protection
  - To ensure a process can not access another process's address space.

  俩一样!
- Logical and physical addresses are the same if compile-time and load-time address-binding schemes are used (in old days); logical (virtual) and physical addresses differ only 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!

- MMU is a hardware mechanism that at runtime maps virtual address to physical address
- There are many different mapping methods, covered in the rest of this chapter
- To start, consider a simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - User programs never accesses the real physical addresses
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to the actual location in memory



however, very naive idea!

Contiguous Allocation

first strategy!

Mapping (many strategy!)

Multi-programming

relocatable

- Contiguous allocation is one of the early memory allocation methods
- Main memory is usually divided into two partitions – one for operating system and one for user processes

  lower value — higher value
- The operating system can be placed in either low memory addresses or high memory addresses, depending on many factors, such as the location of the interrupt vector
- Many operating systems (including Linux and Windows) place the operating system in high memory
- In a multi-programmed operating system, several user processes 住 reside in memory at the same time. In **contiguous memory allocation**, each process contained in a single section of memory that is contiguous to the section containing the next process.

address space (virtual address) $\xrightarrow[\text{loading!}]{\text{MMU}}$ physical address

↓

point some data

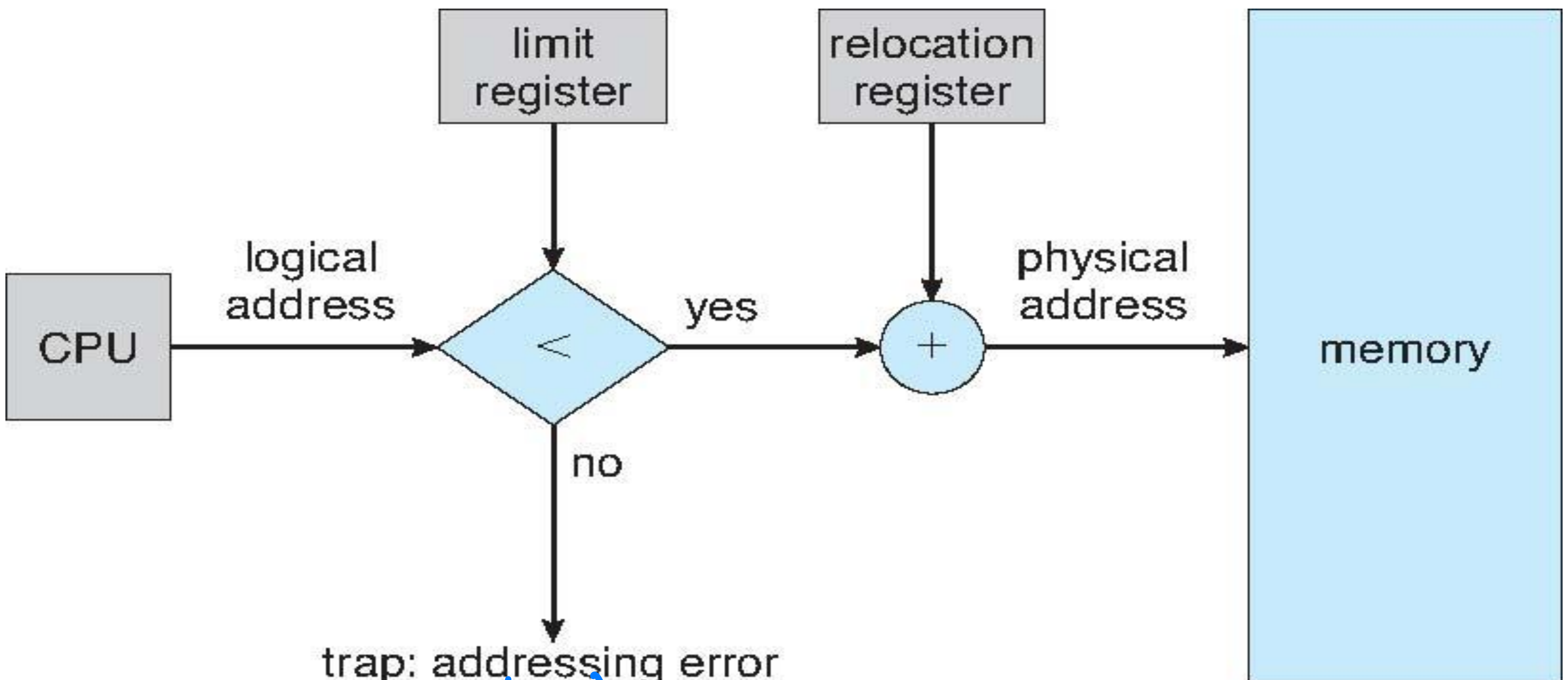
A 1 process, 1 part of memory. 不可 seperate!

Hardware Support for Relocation and Limit Registers

- Relocation register and limit register are used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of the 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

only have the limit:

Protection not needed:
因为有不同 mapping value



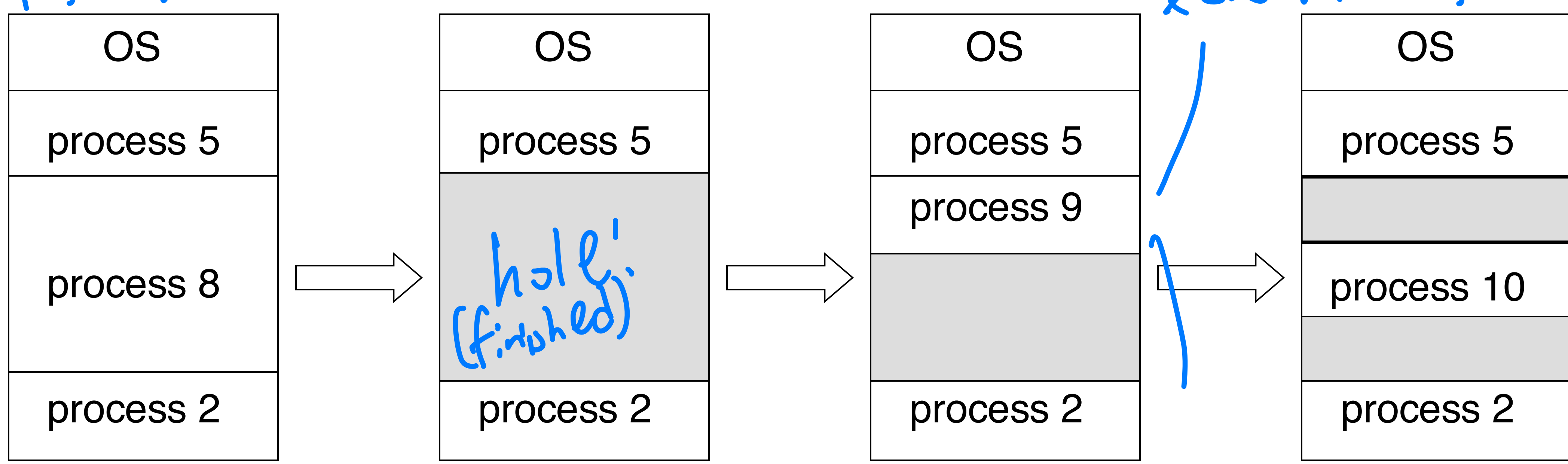
(interruption)

Contiguous Allocation (Cont.)

- Multiple-partition allocations
  - Degree of multiprogramming is bounded by number of partitions
  - Variable-partition sizes (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)

2-D requirement of size!

less memory!



⇒ utilization of CPU

dynamic storage

Dynamic Storage-Allocation Problem

How to satisfy a request of (variable) size n from a list of free holes? 多种方法!

- **First-fit**: Allocate the *first* hole that is big enough

  very quick!, cause more holes!

(reasonable)

- **Best-fit**: Allocate the *smallest* hole that is big enough

  time consuming!

  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole – intended not use it

  As small hole as possible!

  converse idea!

- **Worst-fit**: Allocate the *largest* hole

  => hope can use resource!

  - Must also search entire list
  - Produces the largest leftover hole – intended to reuse the remaining hole

Experiments have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Fragmentation

holes'.

- **External Fragmentation** – total memory space available to satisfy a request, but it is not contiguous – scattered holes
  - The storage is fragmented into a large number of small holes.

- **Internal Fragmentation** – memory allocated to a process may be larger than requested memory; this size difference is memory internal to a partition, but not being used

consume more memory!

Fragmentation (Cont.)

reduce problem!

- Reduce external fragmentation by a technique called **compaction**

  overhead, time consuming!

  - 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. In another word, if relocation is static and is done at assembly or load time, compaction cannot be done
  - The compaction is expensive (time-consuming)

    Too small!

- The backing store (the secondary storage) has similar fragmentation problems, which will be discussed later
- Another more feasible solution is to permit logical address space of the processes to be **non-contiguous**, by dividing into multiple pieces (variable-sized segments or fixed-size pages), thus allowing a process to be allocated physical memory wherever such a piece of memory is available. These techniques include **segmentation** and **paging**

(not pro

seperate to different part < different size / same size

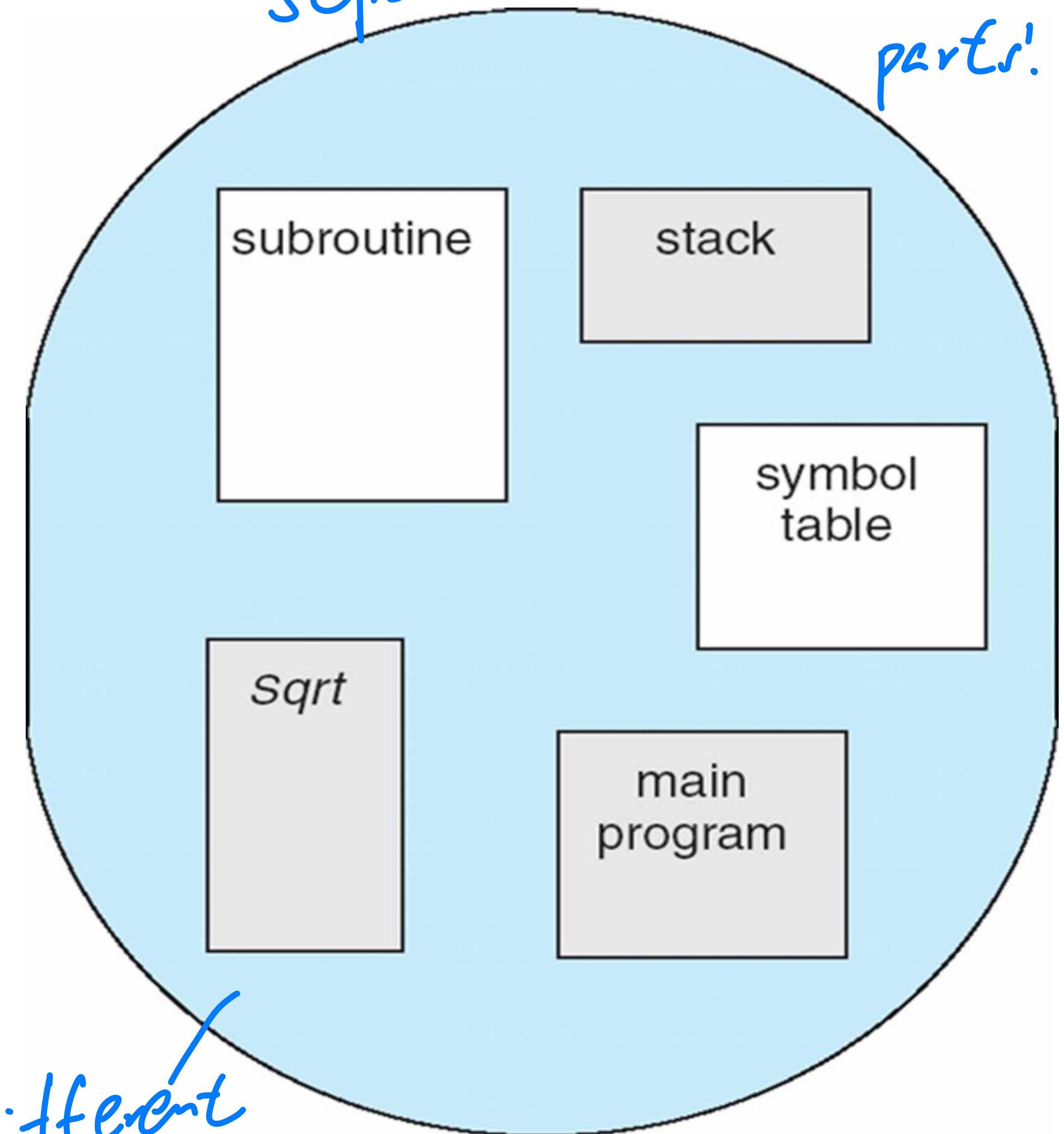
Segmentation- User View

separate them into different parts!

- Memory-management scheme that supports user view of memory

- A program is a collection of segments. A segment is a logical unit such as:

  main program
  procedure
  function
  method
  object
  local variables, global variables
  common block
  stack
  symbol table
  arrays



logical address

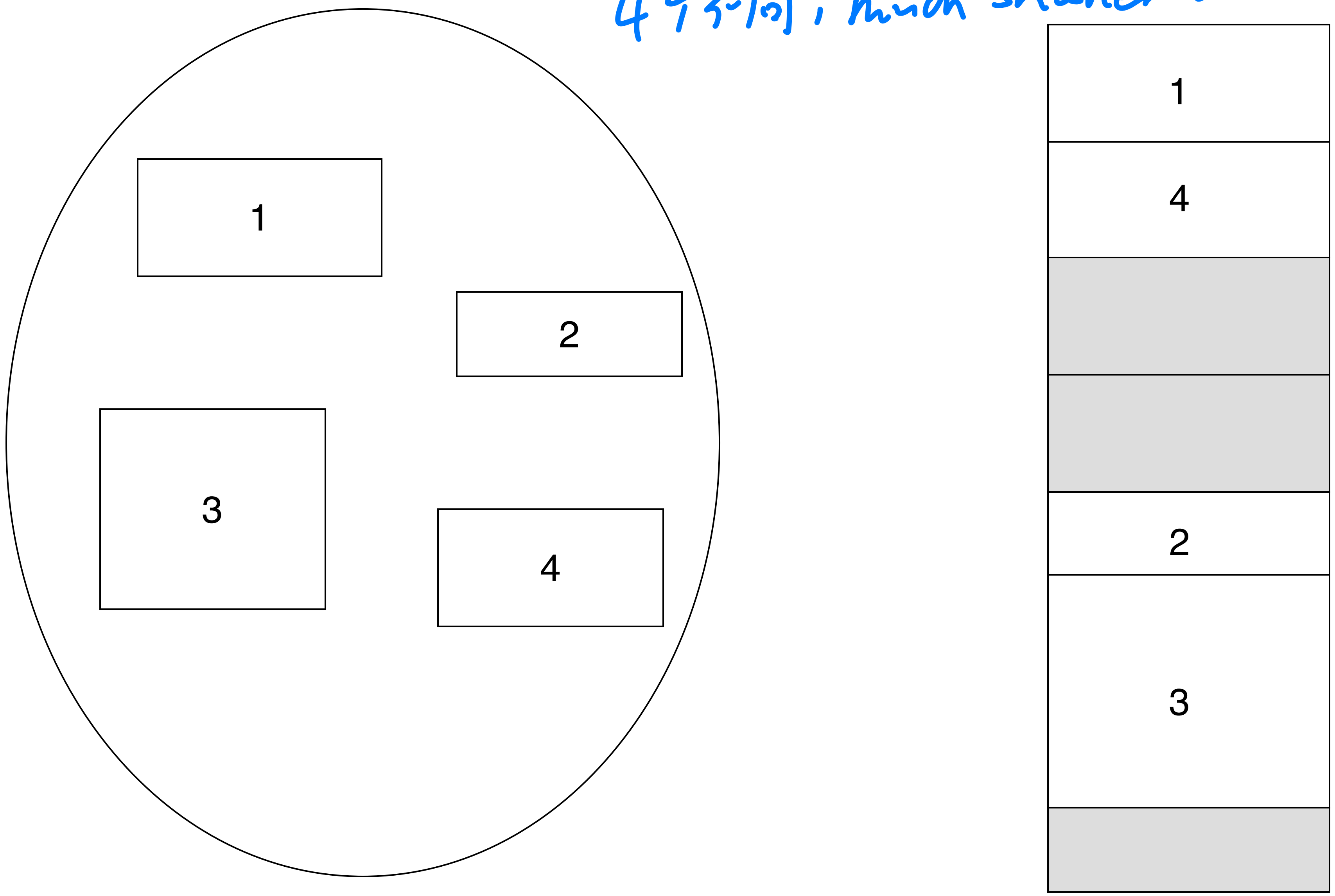
different part of programs

Segmentation: Logical vs. Physical

4个不同, much smaller than whole process

⇓

easy to find a 'holes'.



user space

physical memory space

Segmentation Architecture

access which part!

- The Logical address now consists ~~of~~ a **two-tuple**:

  <segment-number, offset>, where is the data?

  example: find x in main function: (main function(), offset of x)

- **Segment table** – maps two-dimensional programmer-defined addresses (virtual address) into one-dimensional physical addresses; each entry in the segmentation table has:
  - 1 base – contains the starting physical address where the segments reside in memory
  - 2 limit – specifies the length of the segment how large!
- Both STBR and STLR stored in the PCB of a process – Recall a process PCB contains all the information about the process
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates the number of segments used by a program;

  length of the table

  segment number s is legal if s < STLR

  smaller than length of the table!

  ID Base limit!

Segmentation Architecture (Cont.)

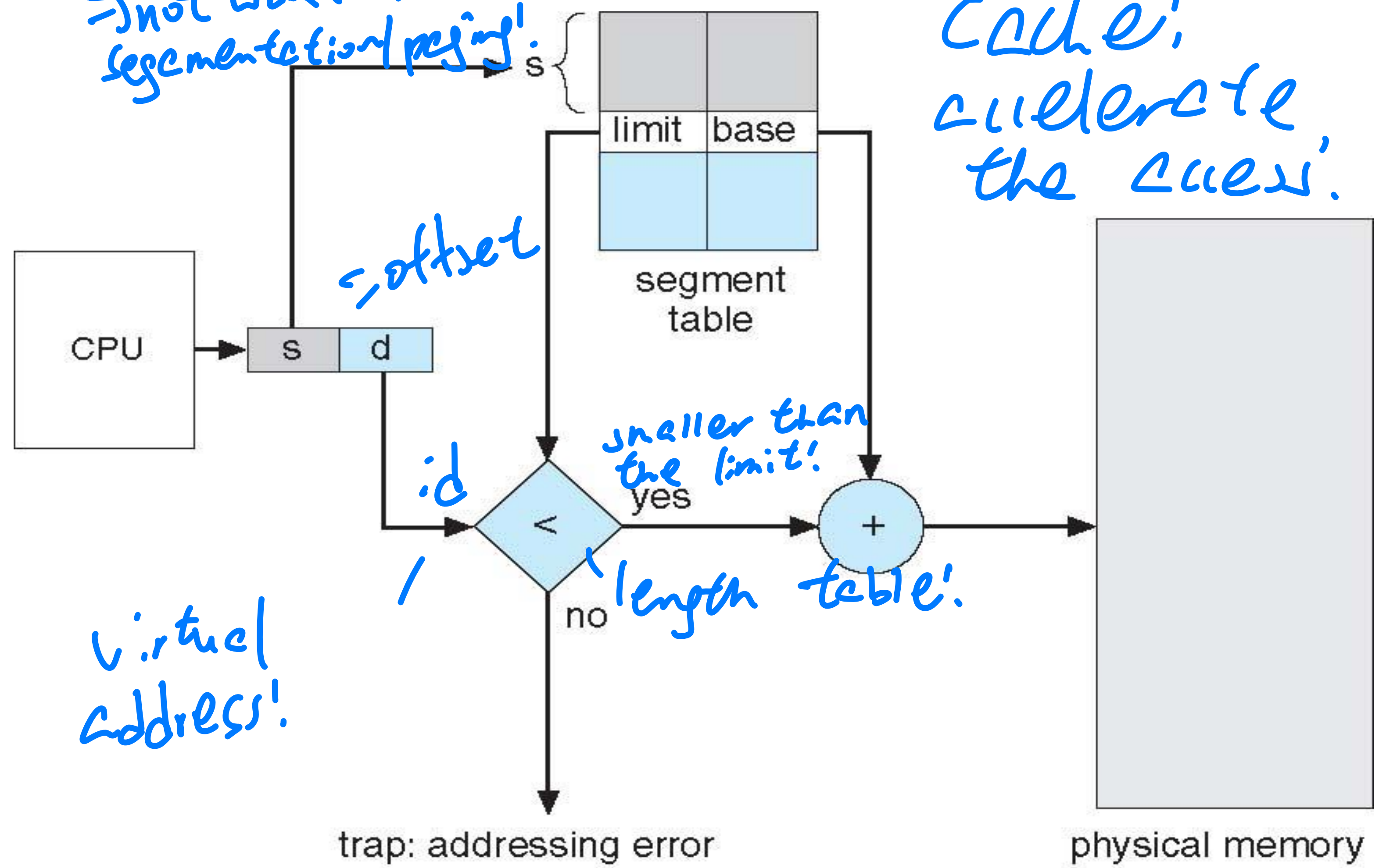
48 bit? contiguous! still suffer from dynamic data!

- **Protection**. each entry in a segment table associates with:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- A protection bit is associated with each segment; code sharing occurs at segment level
- Since segments vary in length, memory allocation for a segment is also a dynamic storage-allocation problem
- External fragmentation still exists, but is much less severe than that in a contiguous allocation – since each segment is considerably smaller than the total memory space of a process

Can define read only / ...

Why virtual address:
⇒ not want have difficult segmentation/paging!

cache: accelerate the access.



, offset

id

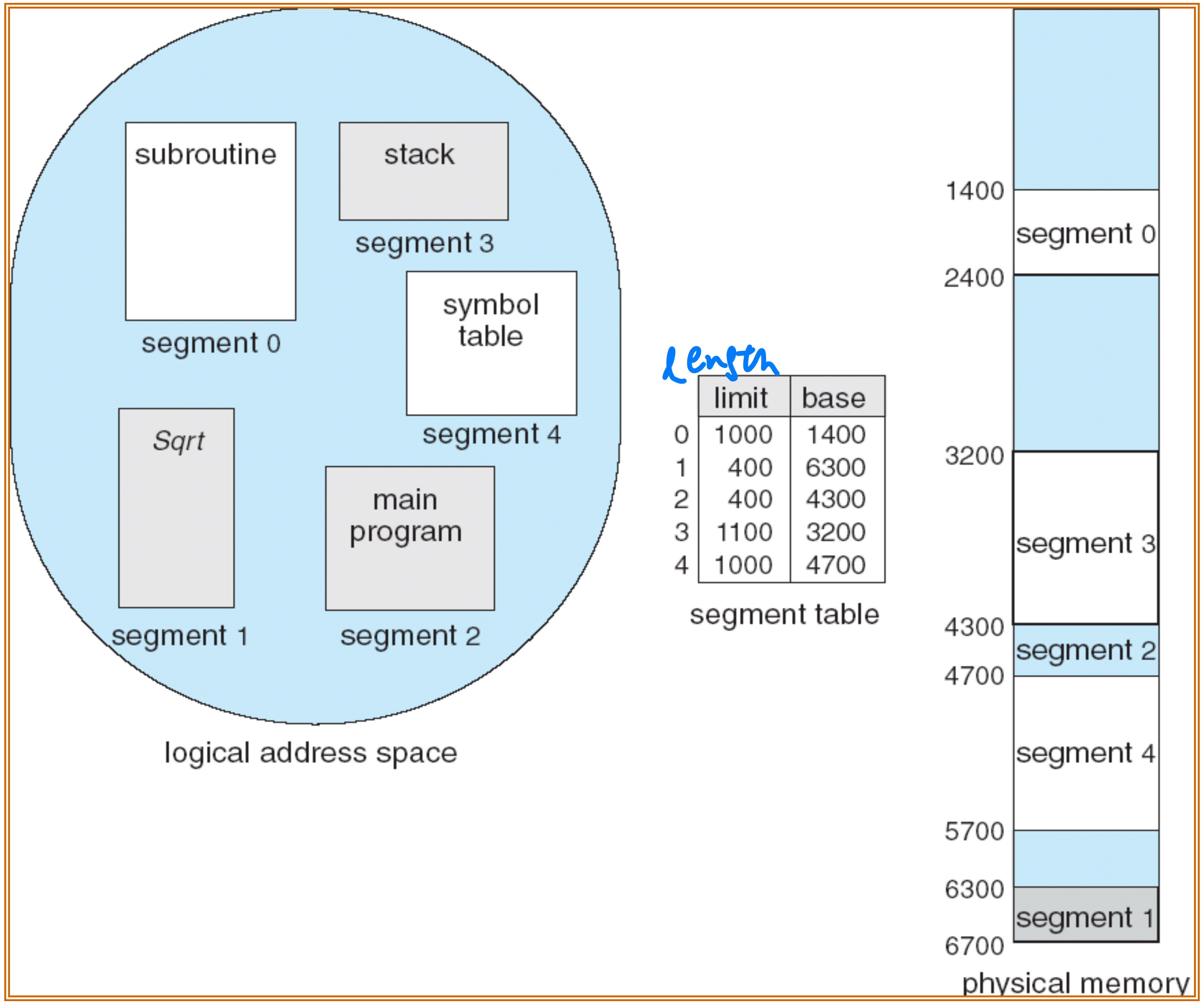
smaller than the limit!

length table!

virtual address!

pointer → find table in cpu
↓
add offset
↓
find real address!

Example of Segmentation



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

offset < length-of-s0

- <0, 800> → 2200

outside

- <2, 420> illegal address X
- <4, 930> → 5630

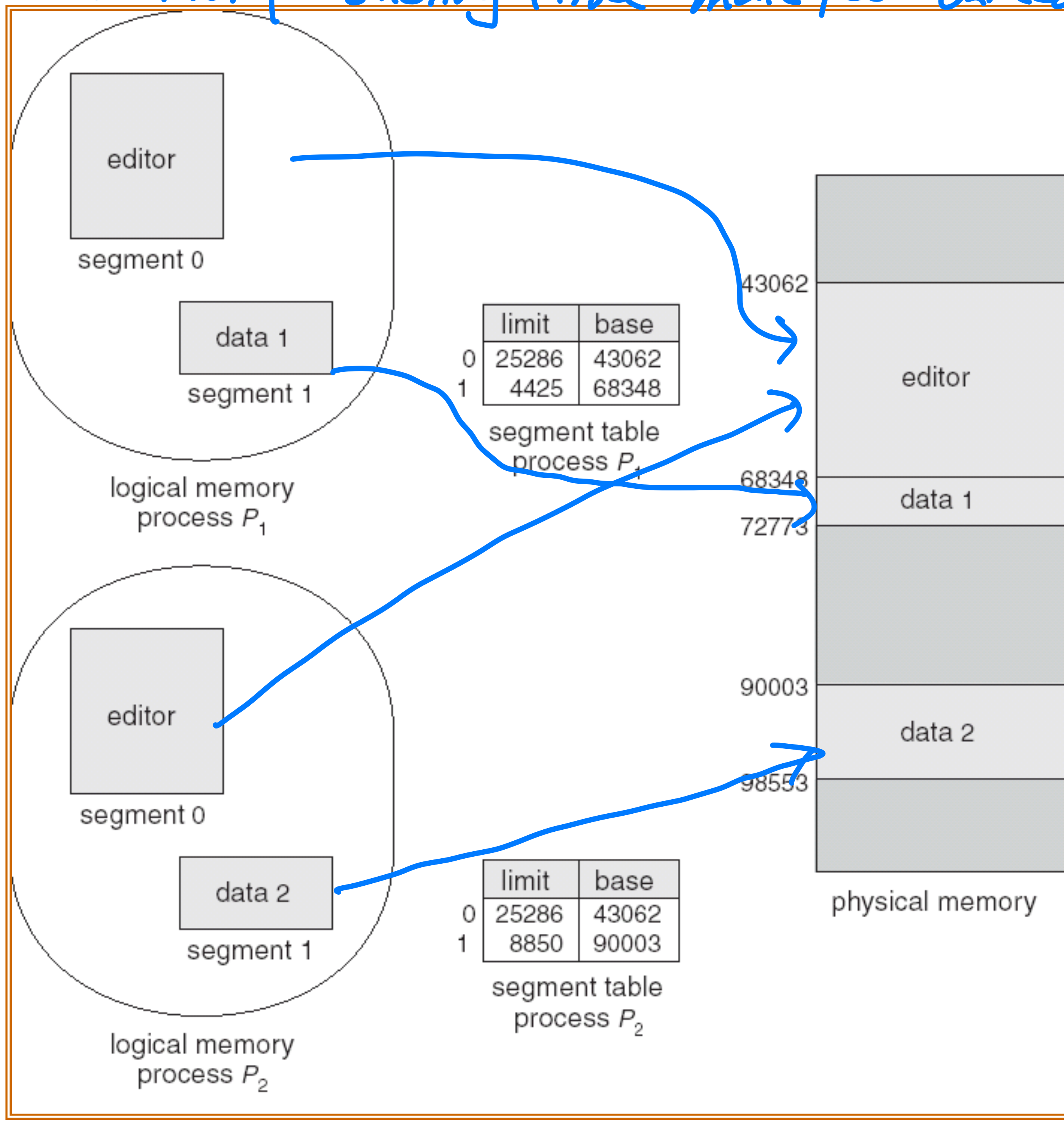
<5, 1000> :illegal addressX

↓

different format of address!

Sharing of Segments

memory sharing (like multiple threads)



for example, read only!

only one data!

define by myself!

Summary - Segmentation

- Protection is easy in a segmentation scheme – in segment table
  - Code segment would be read-only, data and stack would be read-write (stores allowed), shared segment could be read-only or read-write
- Can address be outside the valid range? can change the segmentation table!
  - Yes, this is how stack and heap are allowed to grow. For instance, stack takes fault, system automatically increases the size of stack HW 可以當 fixed value!
- The main problem with segmentation
  - It must fit variable-sized chunks into physical memory – classical **Dynamic Storage Allocation Problem** suffer a lot of holes!
  - It may move processes multiple times to fit everything
  - External fragmentation → compaction!

↓ moving blocks from one-side to another!

chunk with same size!
有 hole → 進入!
no holes!

Paging

X various size of blocks

- Physical address space of a process can be non-contiguous; process is allocated physical memory whenever the physical memory space is available
  - Avoids external fragmentation
  - Avoids the problem of varying sized memory chunks

X suffer from dynamic allocation problem!

- Divide physical memory into fixed-sized blocks called frames
  - Usually, the size is power of 2, between 4KB (12 bit offset in physical memory address) and 1 GB (30 bit offset) – defined by the hardware

determined by hardware (fixed number)

logical!

- Divide logical memory into blocks of same size called pages - the same size of a frame
- The OS needs to keep track of all free (physical) frames available in main memory
- To run a program of size **N** pages, need to find **N** free frames (often non-contiguous) in memory and load the program into that

keep tracks on free frames!

- A **page table** is used to translate logical to physical addresses – keep track of allocated frames
- This, however, suffers from internal fragmentation in the last page/frame

-memory waste

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

similar to segmentation table!

page number	page offset
p	*d*
m - n	*n*

$\underbrace{\hspace{6em}}_{m}$

- Total m bits logical address, the logical address space 2^m bytes, and the page size or frame size is 2^n, the number of pages this logical address contains is 2^{m-n}

choose appropriate value!

total bits = address space!
decided by hardware!

Example of Paging Scheme

- Suppose logical address of a system is

第n个page?

4 KB

page number	page offset
p	*d*
20-bit	12-bit

最多32 bit virtual address!

friend number!

每个page的容量

- The page size - 12 bits are used to offset into a page, the page size is $2^{12} = 4$ KB
- 32 bits virtual address indicates virtual address space is $2^{32} = 4$ GB
- 20 bits page number, the number of page is 2^{20}

Paging Hardware

X caching!

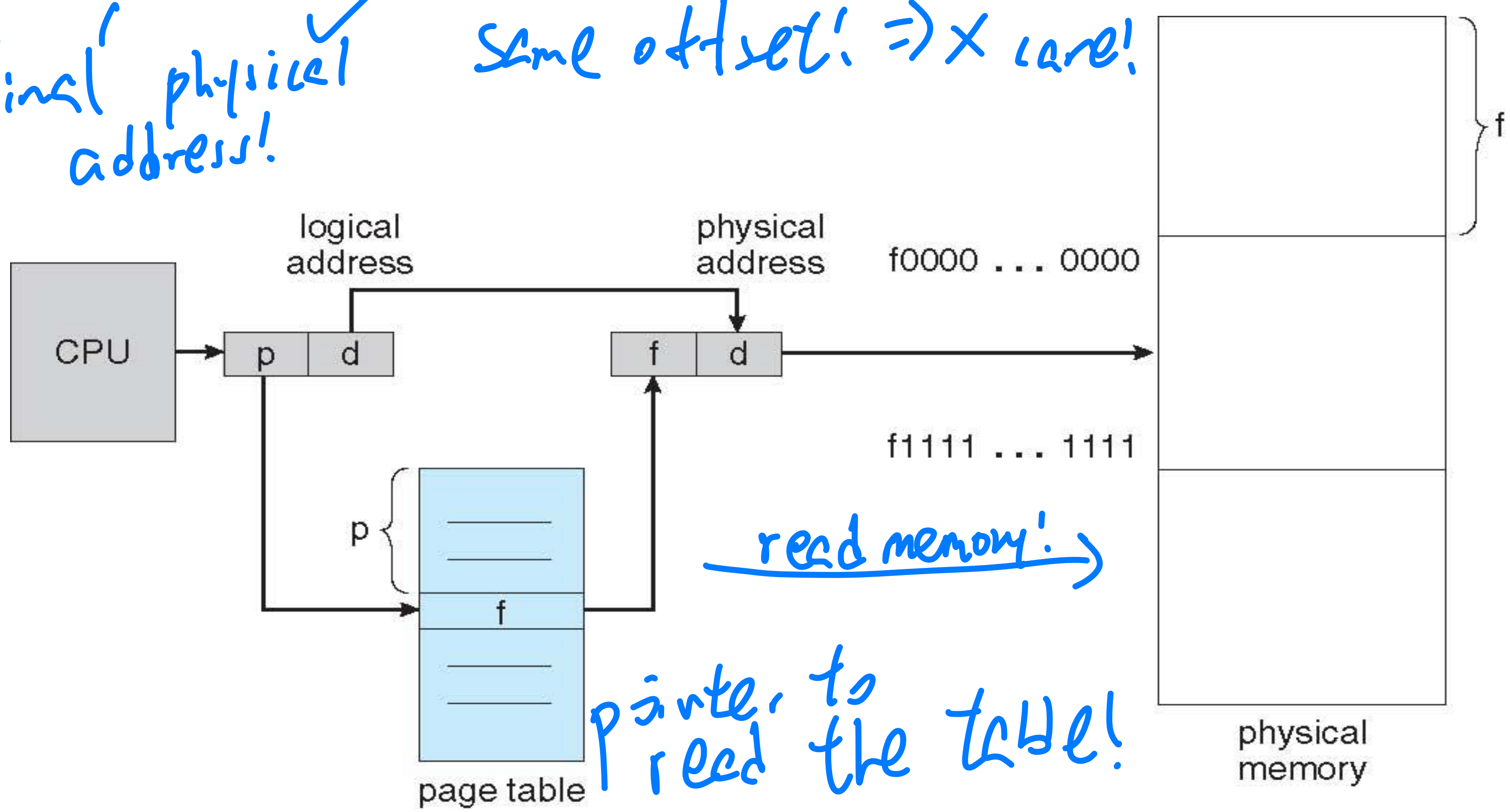
- The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:
  1. Extract the page number **p** and use it as an index into the page table.

     to physical address!
  2. Extract the corresponding frame number **f** from the page table.

     不用做 calculation!
  3. Replace the page number **p** in the logical address with the frame number **f**
- As the offset **d** does not change, it is not replaced, and the frame number and offset now comprise the physical address.

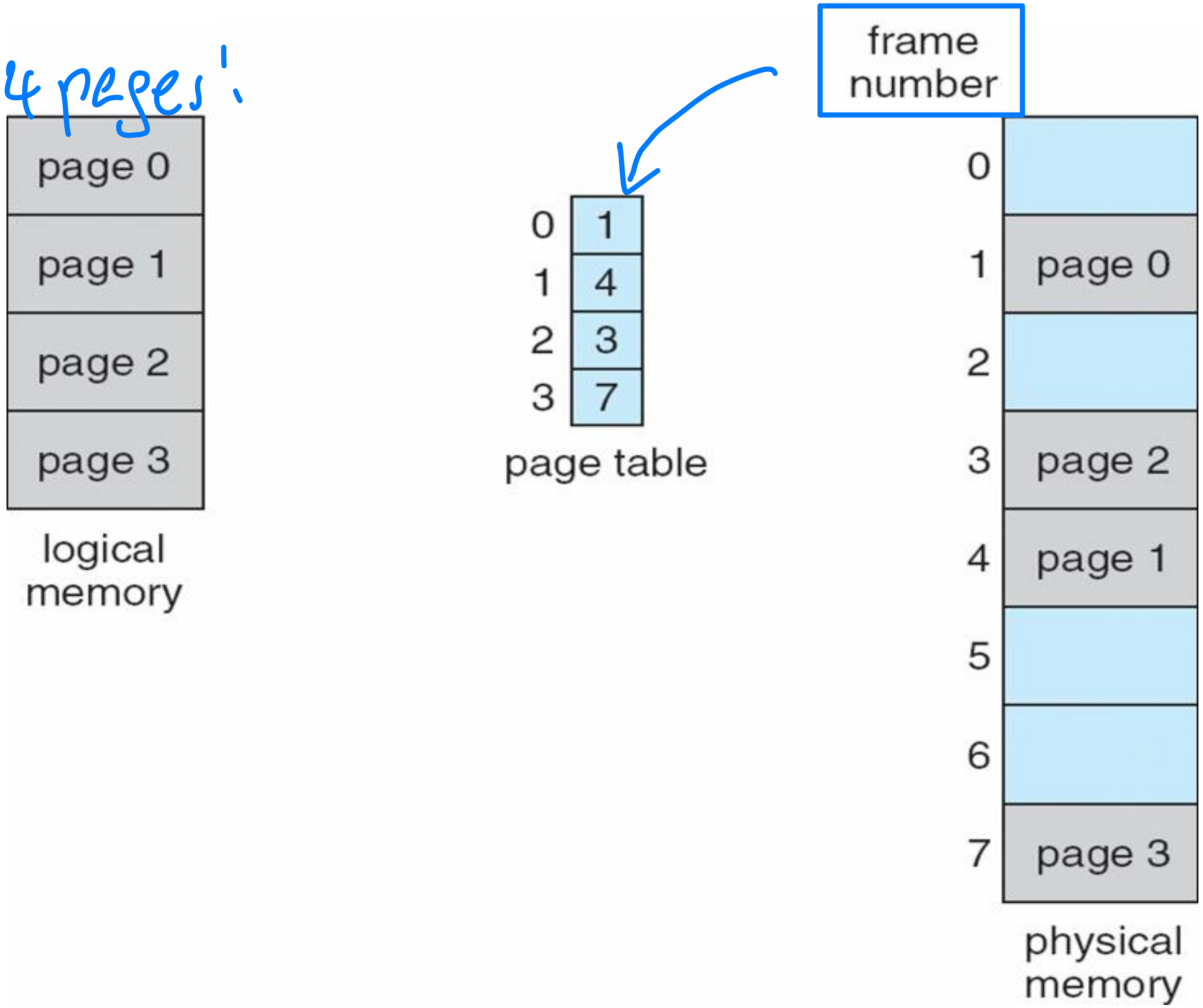
final physical address!

same offset! => X care!



Paging Model of Logical and Physical Memory

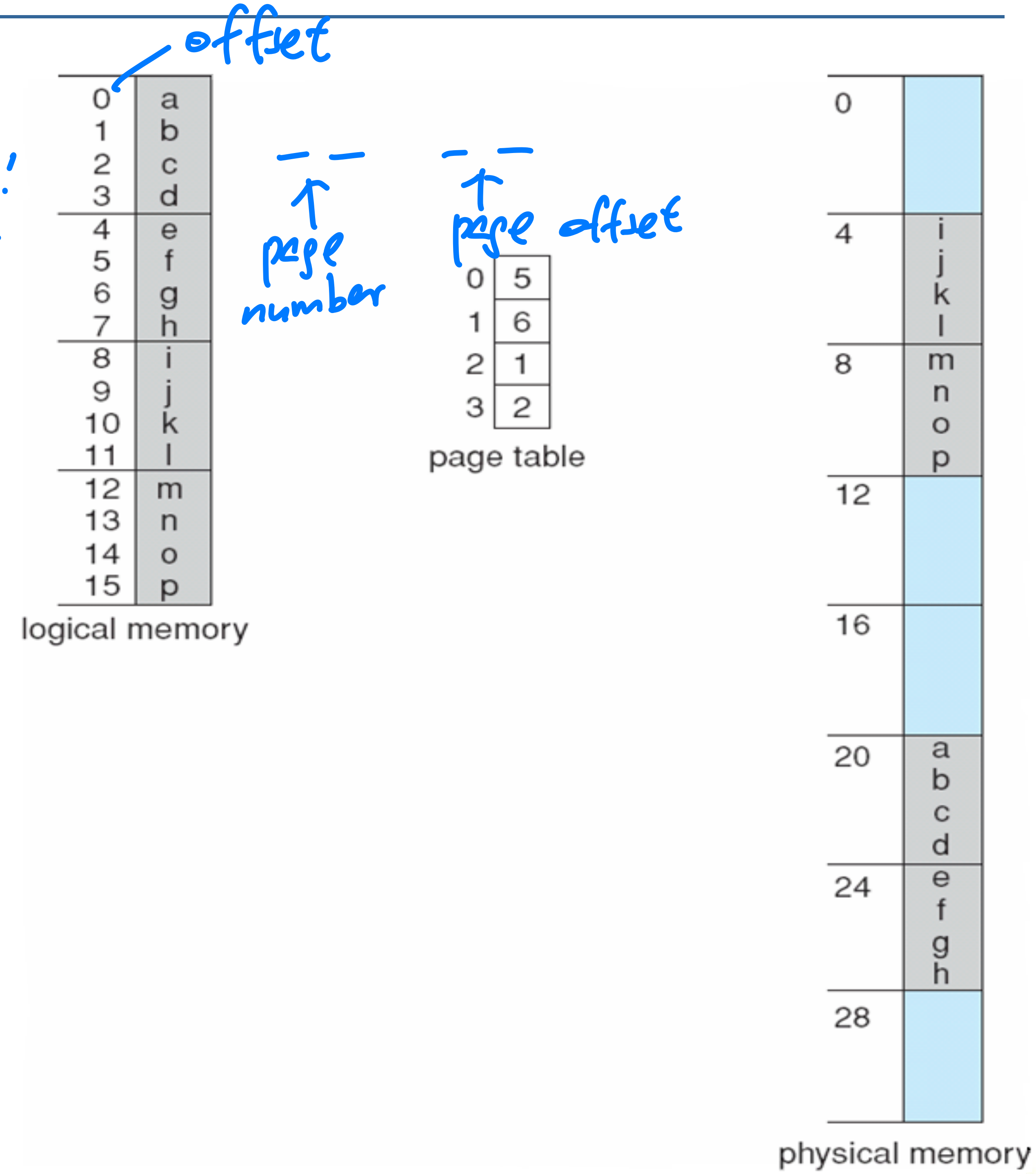
4 pages!



Paging Example

page number:

- Frame number can automatically derive the starting address of each frame 4 offsets!
- m=4 represents logical address of 16 bytes 2^4
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0]
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].
- Logical address 13 maps to physical address 9.



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

Paging (Cont.)

frame size = 4KB in mordern computer!

- Calculating internal fragmentation
  - Page size = 2,048 bytes (2KB)
  - 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

last page;

independent of process size!

- So small frame sizes desirable? - not really

不是越小越好

  - The overhead involved in each page table entry reduces when page size increases.
  - Each page table entry takes memory to track, smaller page size results increased number of page table entries (one per page), thus it leads to larger page table

smaller item in page table!

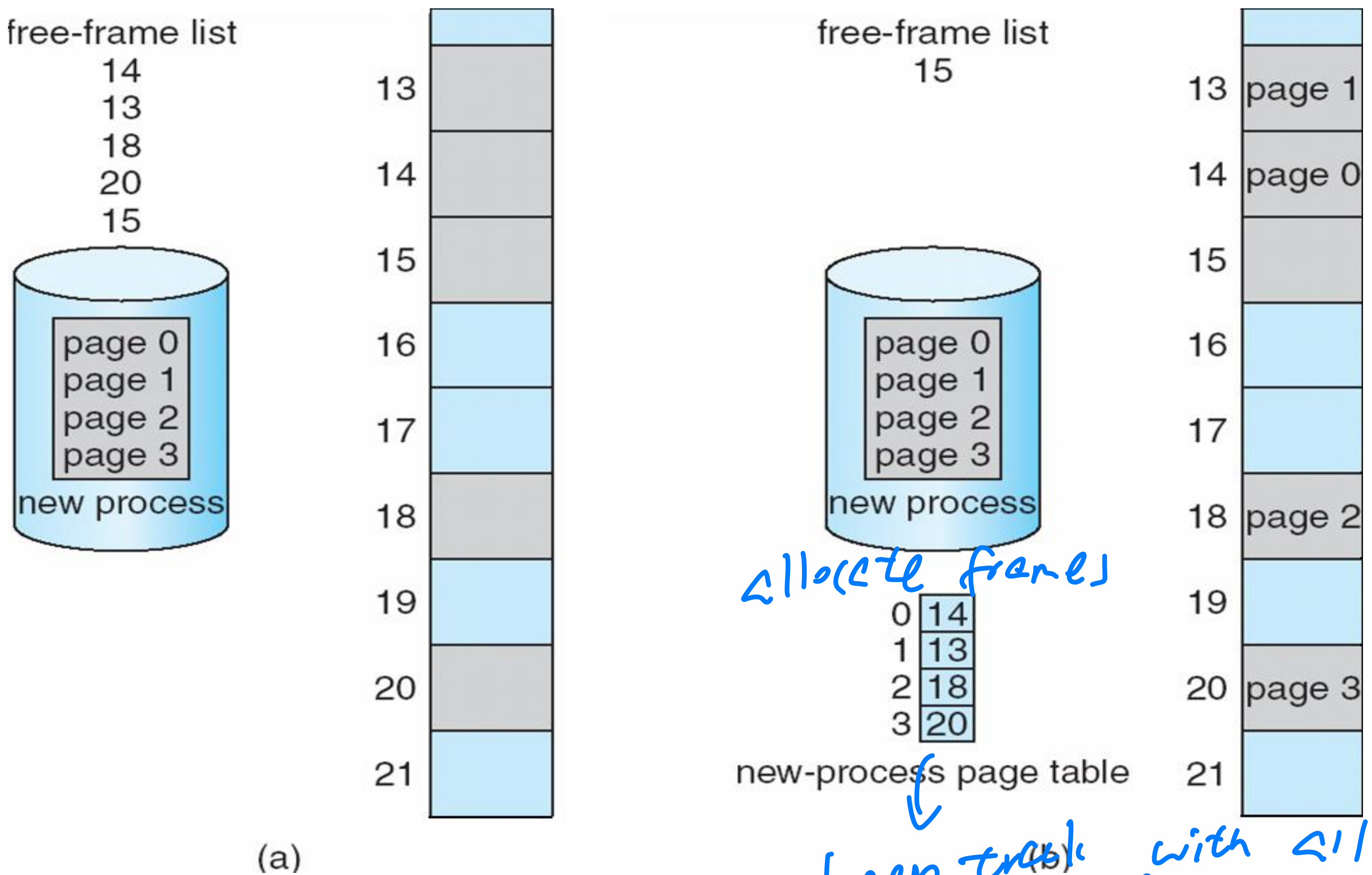
- Page size has grown over the time
  - SunMicro Solaris OS supports two-page sizes – 8 KB and 4 MB
- Process view and physical memory now very different

size ↑ ⇒ consumes memory!

不统一

  - The programmer views memory as one single space, containing only this one program. But in fact, user program is scattered throughout physical memory, which also holds other programs.

Free Frames



allocate frames

keep track with all frames!

Before allocation

After allocation

how to implement?

Implementation of Page Table

pointer to this table, length of the table!

- Page table is kept in main memory – PCB has to keep track of memory allocation for the process
- Page-table base register (PTBR) points to the page table (starting address)
- Page-table length register (PTLR) indicates size of the page table

① Access page table ② find final data

- In this scheme every data/instruction access requires **two memory accesses** × 2
  - One for the page table (translation) and one for fetching the data / instruction – same with segmentation scheme

like segmentation table!

access memory is too slow!

- 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)** (small) (64-1024 entries)

2种 independent techniques! cache!

- TLBs typically small (64 to 1,024 entries). Some CPUs implement separate instruction and data address TLBs. This is shared and used by all processes (kernel and user processes) in a system

cache policy!

- On a TLB miss (if the page number is not in the TLB), value is loaded into the TLB for faster access next time. This can be done by hardware (MMU) or software (running kernel codes)
  - Replacement policies must be considered – LRU, round-robin or even random are possible
  - Some entries can be wired down for permanent fast access, for example TLB entries for key kernel code for fast access

不 detailed in lecture!

Associative Memory

technique!

traditional:
for loop →
scan up table: O(n)

- Associative memory (TLB) – parallel search

check by hardware (circuit)

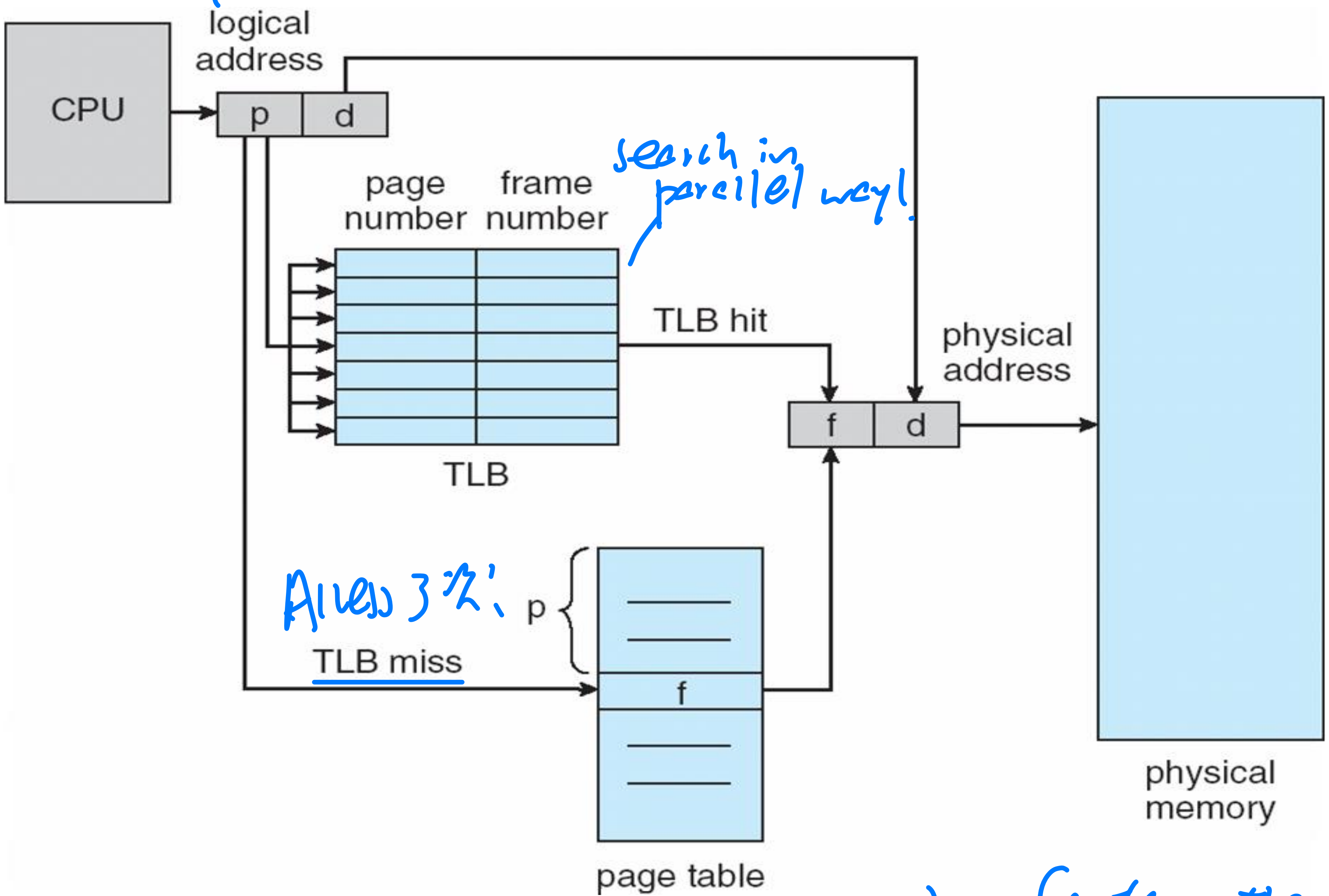
Page #	Frame #

key → broadcast
to all entries
do circuit →
checking → same time
O(1) time

- Address translation (p, d)
  - **Check all entries in parallel** (hardware) – TLB shared and used by all processes (process ID and page number)
  - If p is in associative register (TLB), get frame # out – TLB hit
  - Otherwise get frame # from page table in memory, and also bring this entry to the TLB
- Locality on TLB
  - Instruction usually stays on the same page (sequential access nature)
  - Stack exhibits locality (pop in and out)
  - Data less locality, still quite a bit

Paging Hardware With TLB

↑ access memory! (too slow!)



search in parallel way!

Access 3%;

memory

=> faster than two memory access!

Effective Access Time

- Associative Lookup = ε time unit
  - Usually < 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

hit

- **Effective Access Time** (EAT) – memory access time normalized to 1

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

memory! $= 2 + \varepsilon - \alpha$ access memory 2次!

TLB

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns

- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

as higher as possible! very high!

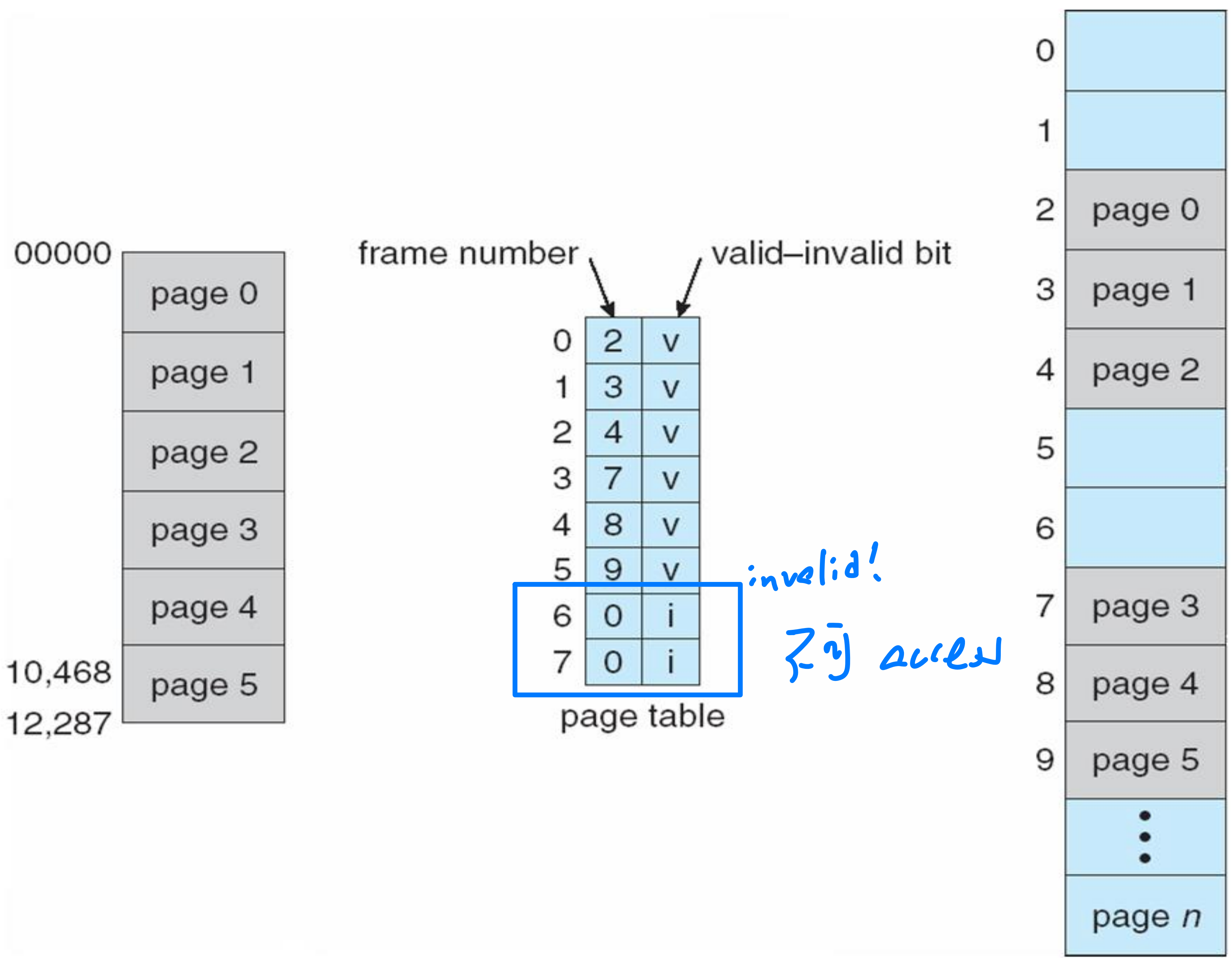
Memory Protection

- Memory protection in paging is accomplished by protection bits associated with each frame. Normally, these bits are kept in a page table.
  - One bit can define a page to be read–write or read-only, or more bits to indicate execute-only
- Valid-invalid bit attached to each entry in the page table: Check the permission!
  - "valid" indicates that the associated page is in the process' logical address space
  - "invalid" indicates that the page is not in the process' logical address space
  - The operating system sets this bit for each page to allow or disallow access to the page
- Any violations result in a trap to the kernel
  - Example: a 14-bit address space (0 to 16383). If a program only uses address 0 to 10468, with a page size 2KB, pages 0-5 are valid, pages 6-7 are invalid.
- Rarely does a process use all it address range, usually only a small fraction of the address space available. It would be wasteful to create a page table with entries for every page in the address range, and most of the table entries are unused but would take up memory space
  - use page-table length register (PTLR) to indicate the size of the page

different pages table for different sizes

=) keep this table as small as possible

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



Shared Pages

similar to segmentation!

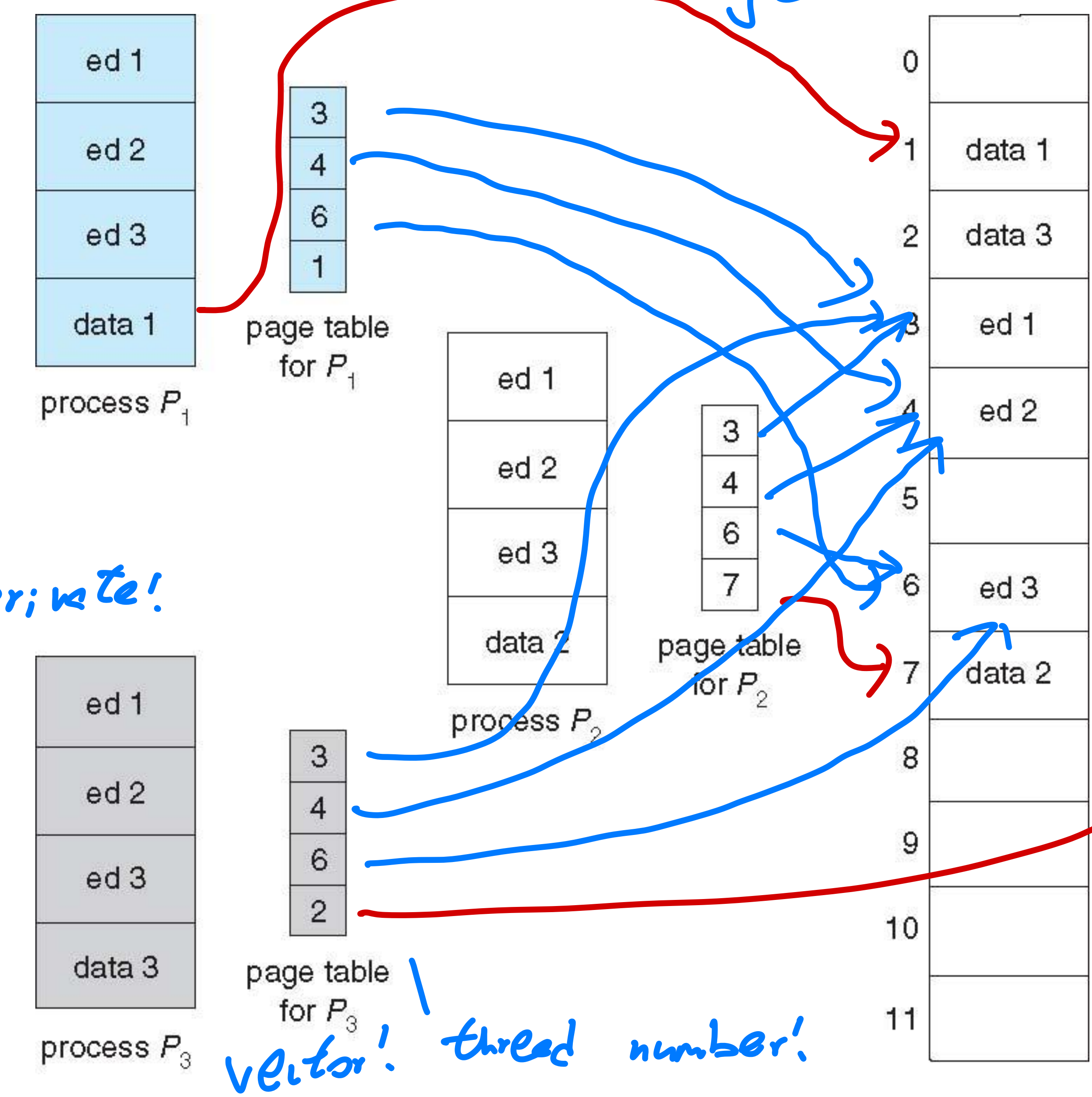
Shared code

- 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 IPC if sharing of read-write pages is allowed

Private code and data

↗ share, keep private!

- 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



An Example: Intelx86 Page Table Entry

- What is in a **Page Table Entry** or **PTE**?
  - For page translation, each page table consists of a number of PTEs
  - Permission bits: valid, read-only, read-write, write-only

page number = index!

- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

only care this in HW/Exam!

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1 ⇒ 4MB page (directory only).
Bottom 22 bits of virtual address serve as offset within a page

mapping!!!!

Structure of the Page Table

program

- Memory structures for paging can grow enormously using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers

    4 Gb
  - Page size of 4 KB (2^{12}) , offset bits = 12
  - Page table would have 1 million entries (2^{32} / 2^{12}) 都好多!

    32 bits!
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot

      very very big!
    - Page table cannot be allocated contiguously in main memory either, which will be allocated into multiple pages (frames) – **paging the page table**, meaning another page table is needed to track the page table inside memory

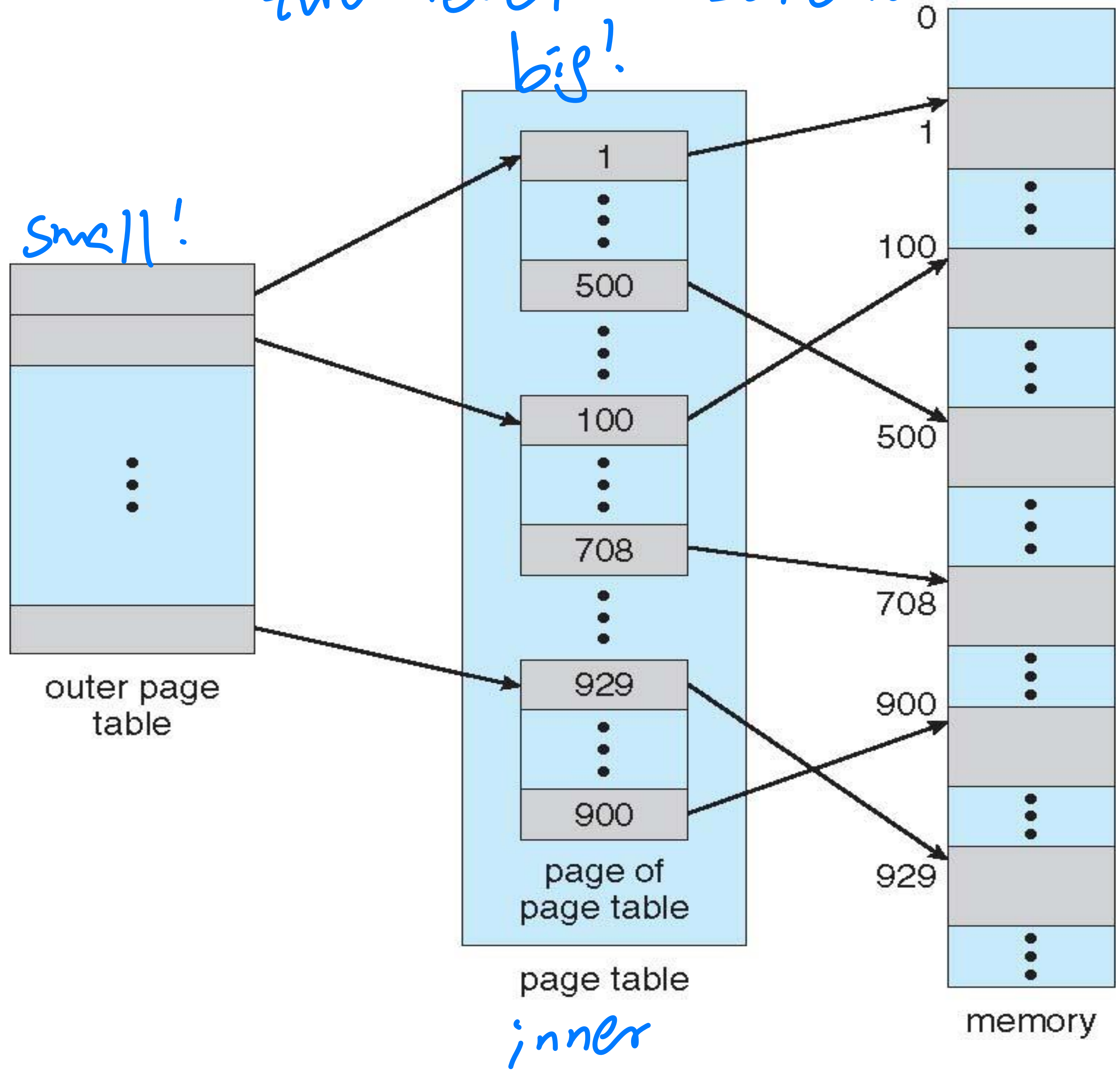
      each page in page table
  - Page size 4 MB (2^{22}) results in a page table with 1,000 entries, less of a problem
  - What about 64-bit logical address?
- Hierarchical Paging
- Hashed Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table or hierarchical page tables
- To page the page table

Two-Level Page-Table Scheme

two-level scheme!



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 - 4kB per page!
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number P_2, this must be 10 bits if each PTE occupies 4 bytes, **ensuring each inner page table can be stored within one frame (page)**,
  - a 10-bit page offset P_1. If PTE also occupies 4 bytes, the number of bits in P_1 can not be more than 10, or it needs to be further divided – more than two levels ≤ 4kB
- Thus, a logical address is as follows:

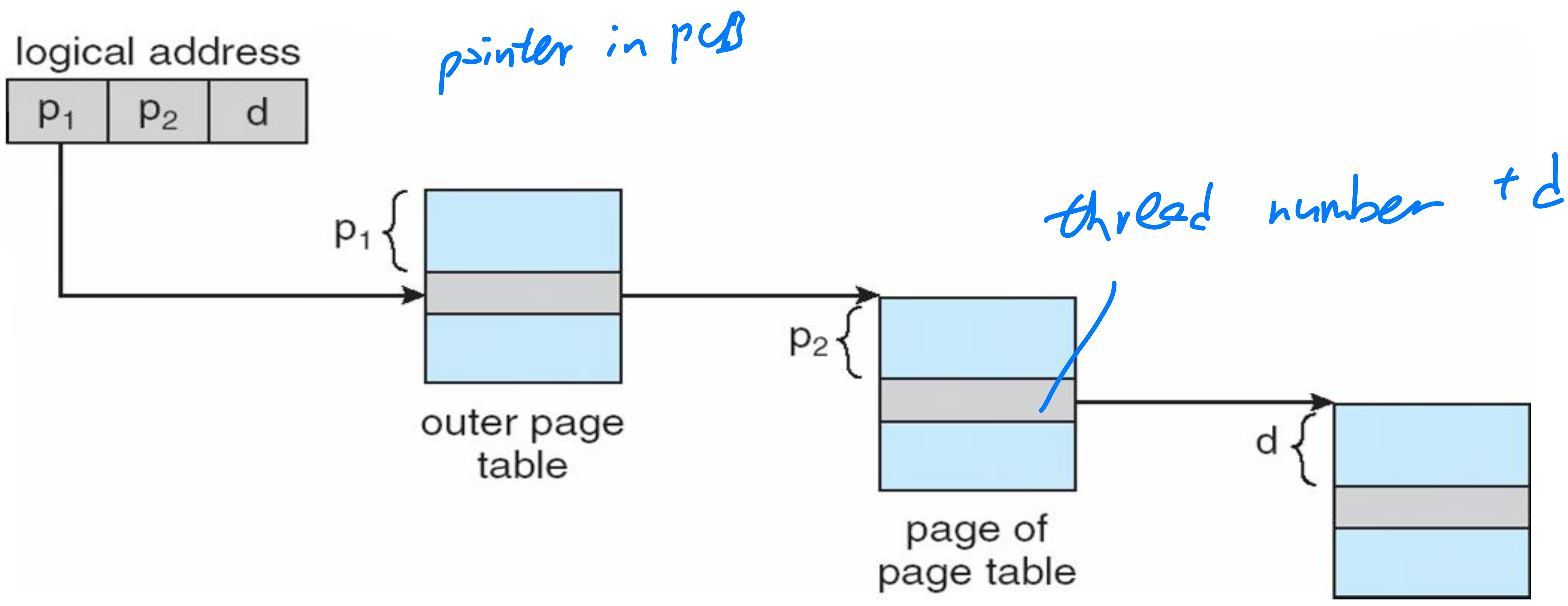
outer side!

page number		page offset - 4kB
p_1	p_2	d
10	10	12

- where p_1 is an index into the outer page table (in Intel architecture, this is called “directories”, and p_2 is the displacement within the page of the inner page table
- because the address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table

Address-Translation Scheme

frame = 4kB



64-bit Logical Address Space

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

outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes (16 GB) in size
    - And possibly 4 memory access to get to one physical memory location
    - The 64-bit UltraSPARC would require seven levels of paging – a prohibitive number of memory accesses – to translate each logical address

$4KB / 2^{12}$

12 bits

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

4KB page!

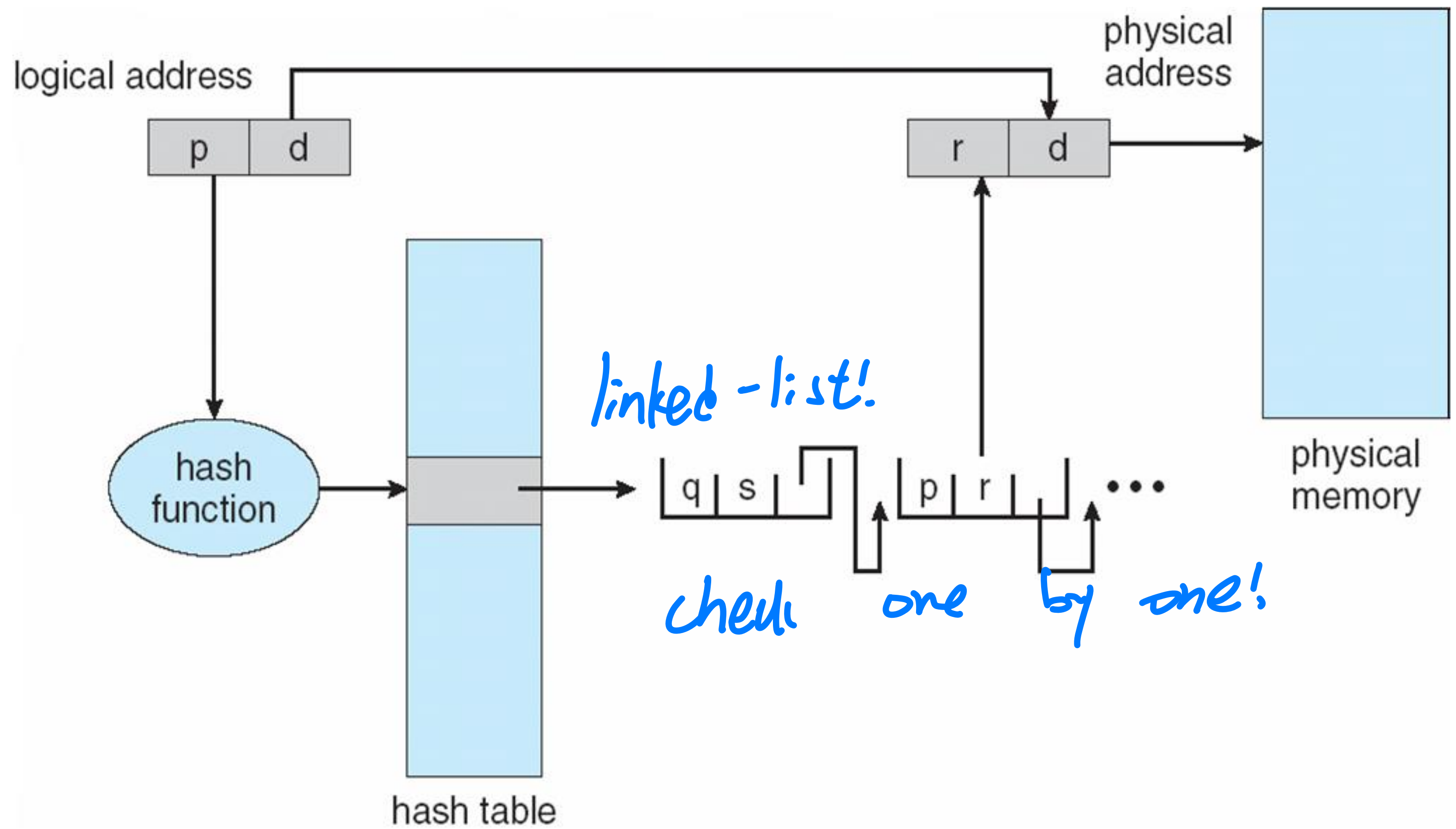
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

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

involves lots!
hash function!

Collision!



Example: The Intel 32 and 64-bit Architectures

Example in modern CPU!

- Dominant industry chips
  - 16-bit Intel 8086 (late 1970s) and 8088 was used in original IBM PC

enable virtual memory.

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

not have this, have Arm CPU!

- Current Intel CPUs are 64-bit and called IA-64 architecture
  - Currently most popular PC operating systems run on Intel chips, including Windows, MacOS, and Linux (Linux runs on several other architectures as well)
  - Intel's dominance has not spread to mobile systems, where they mainly use ARM architecture

- Many variations in the chips, only the main ideas are covered here

Example: The Intel IA-32 Architecture

不重要；

- Supports both segmentation and segmentation with paging
  - Each segment can be as large as 4GB (32 bits) further divided into pages
  - Each process can have up to 16K segments per process (14 bits), divided into two partitions
    - First partition of up to 8 K segments 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))

share!

- CPU generates logical address
  - Selector given to segmentation unit - which produces linear addresses – s for segment number, g indicates whether the segment is in LDT or GDT, p deals with protection

s=1 = private / 0 = share

– protection!

s	g	p
13	1	2

13+1 = 14

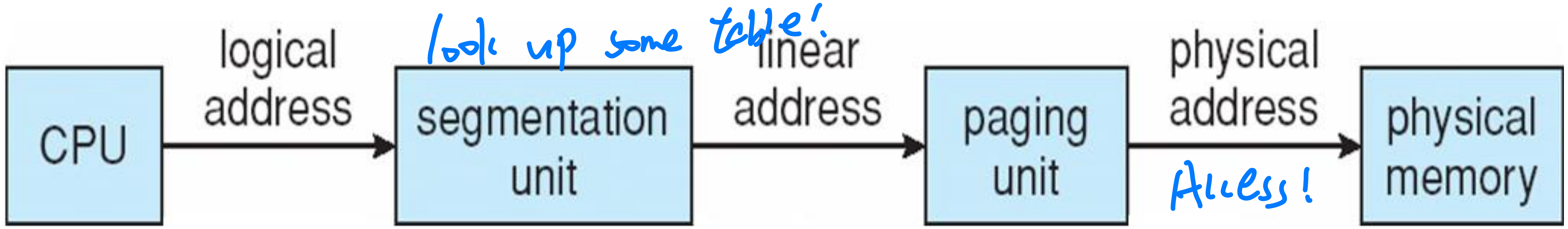
  - Linear address is then given to the paging unit
    - Which generates physical address in main memory

final

    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Intel IA-32 Segmentation

- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the **base location** and **limit**
- The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment is used to generate a linear address.

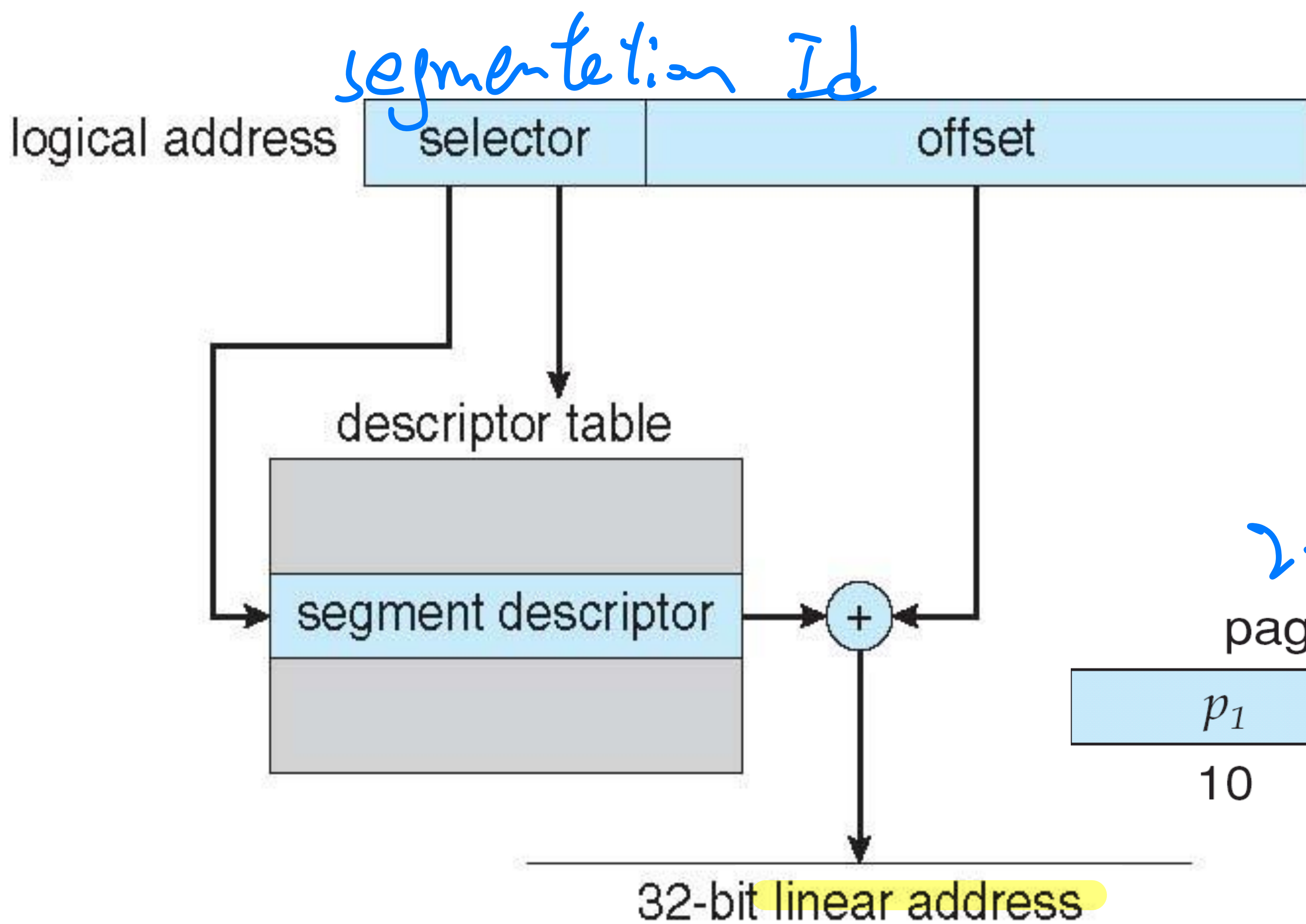


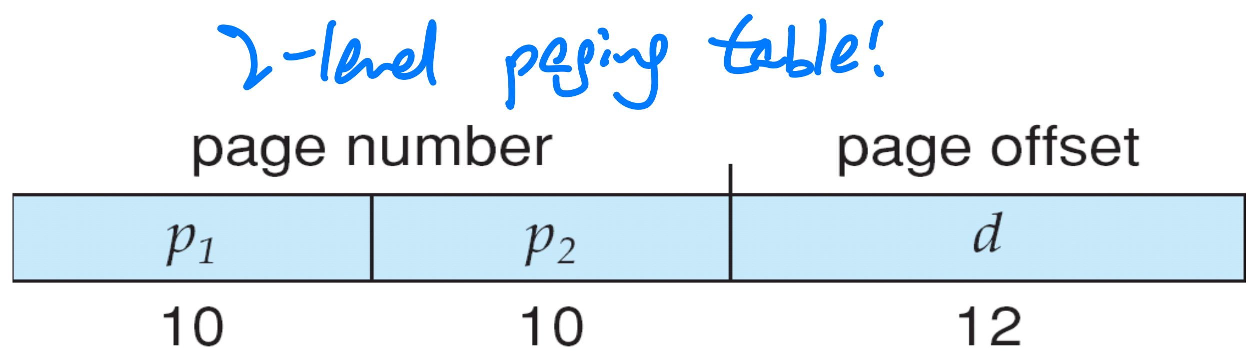
Combination of segmentation and paging!

Intel IA-32 Segmentation

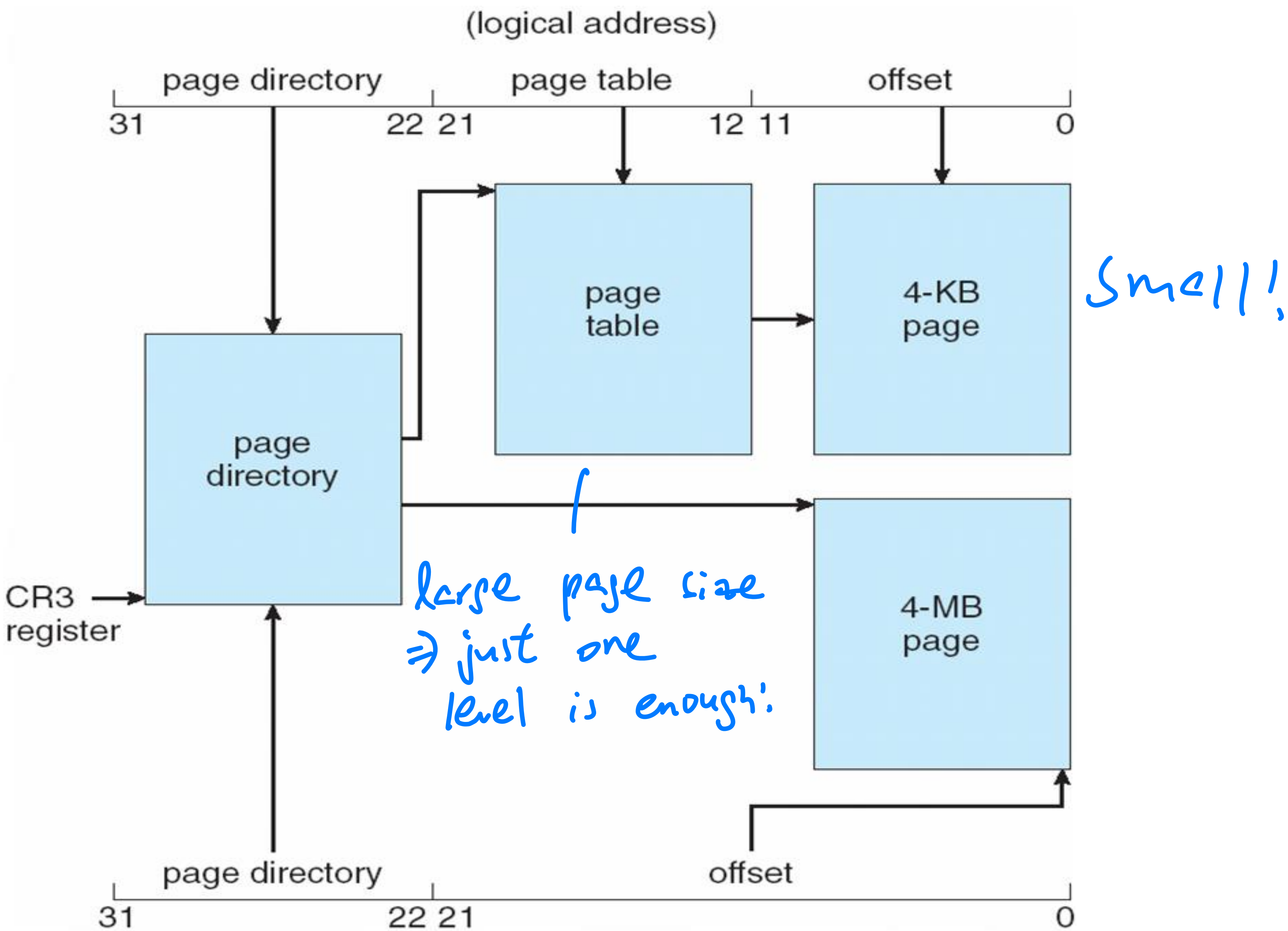
The linear address on the IA-32 is 32 bits long and is formed as follows

- The **limit** is used to check for address validity. If the address is not valid, a memory fault is generated, trap to the operating system. If it is valid, the value of the offset is added to the value of the **base**, resulting in a 32-bit linear address (i.e., each segment can be 4GB)





Intel IA-32 Paging Architecture



Intel x86-64

- Current generation Intel x86-64 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 (page address extension) so virtual addresses are 48 bits and physical addresses are 52 bits (4096 terabytes)

current structure!

reduce level of indexing

larger!

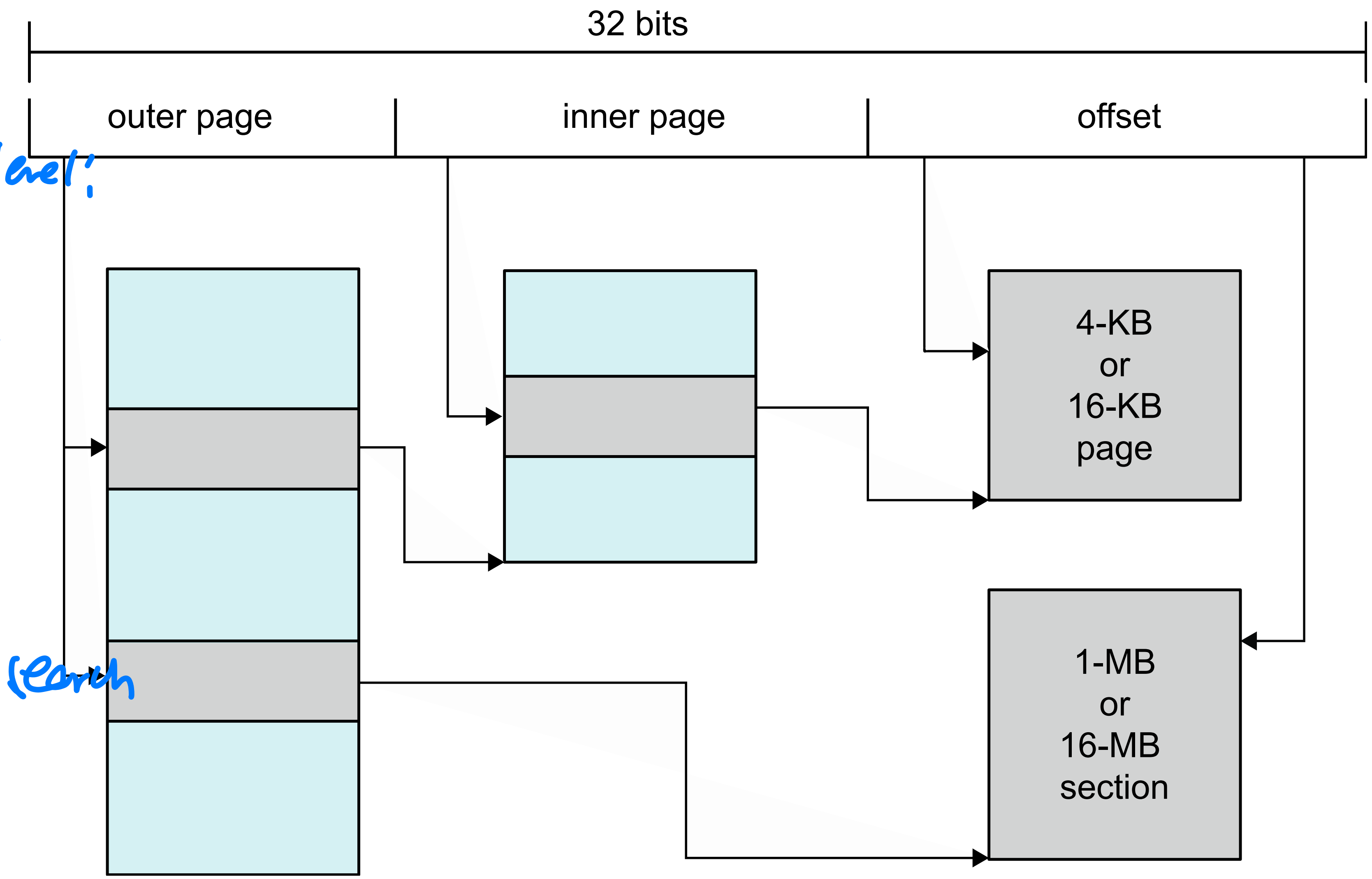
不用!

unused	page map level 4	page directory pointer table	page directory	page table	offset
63 – 48	47 – 39	38 – 30	29 – 21	20 – 12	11 – 0

Example: ARM Architecture

widely adopted in modern os

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages — smaller → more level!
- 1 MB and 16 MB pages (termed sections) — larger, one level!
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs — Cache ↓ 加速 search
  - 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



End of Chapter 9